ModSoft

Modellbasierte Software-Entwicklung mit UML 2
im WS 2014/15

Teil IV: Object Constraint Language

Prof. Dr. Joachim Fischer
Dr. Markus Scheidgen
Dipl.-Inf. Andreas Blunk

fischer@informatik.hu-berlin.de

ModSoft (UML): Inhalt

Teil I- Einführung

Teil II-Struktur

a)

- Klassen, Assoziationen, Abhängigkeiten
- Interface, Datentypen, Signale, Port,
- Strukturierter Classifier
- Aktive Klassen

b)

- Präzisierungen
- Klassendiagramm, vertiefende Betrachtung
- Operationen

Teil III-Verhalten

a)

- Einfacher Zustandsautomat
- Beispiel: DemonGame
- UML-Modellierungsdefizite

b)

- Aktivitäten

Teil IV- OCL

Inhalt: OCL

~ OCL 2.4
(Feb 2014)

1. Allgemeine Charakterisierung
2. Typen, Metatypen und Werte
3. Typkonformität
4. Standardtypen und Operationen (Überblick)
5. Mächtigkeit von OCL-Ausdrücken
6. OCL-Ausdrucksbildung (anhand von Beispielen)
7. Einfache OCL-Ausdrücke
8. Präzedenzen, OCL-Schlüsselworte
9. OCL-Ausdrücke über Kollektionen
10. Abschließendes Beispiel

Object Constraint Language (OCL)

- ist eine formale Sprache für die Definition von Constraints
- ergänzt die Unified Modeling Language (UML) mit Zusicherungen und Anfragen auf UML-Modellen
- standardisiert von der OMG
- deklarativ und seiteneffektfrei
- fügt graphischen (UML-)Modellen präzisierte Semantik hinzu
- verallgemeinert einsetzbar für alle MOF-basierten Metamodelle
- ist inzwischen allgemein akzeptiert und zählt als „Core Language“ von Modelltransformationssprachen (QVT), Regelsprachen (PRR)

Constraint

Definition

„A constraint is a restriction on one or more values of (part of) an object-oriented model or system."

oder

„Eine Einschränkung ist ein Prädikat, dessen Wert wahr oder falsch ist.
– Boolesche Ausdrücke sind … Einschränkungen. …

OCL erlaubt die formale Spezifikation von **Einschränkungen für**

- einzelne Modellelemente (z.B. Attribute, Operationen, Klassen)
- sowie für Gruppen von Modellelementen (z.B. Assoziationen)"

Wir benutzen im folgenden weiter den Begriff des Constraints.

OCL-Ausdrücke

… sind Boolesche Ausdrücke über …

- Standardbibliothek
  - *Basistypen*:
    - Boolean
    - Integer
    - UnlimitNatural
    - Real
    - String

- *Kollektionstypen*:
  - Collection
  - Set
  - Ordered Set
  - Bag
  - Sequence

- TupelType

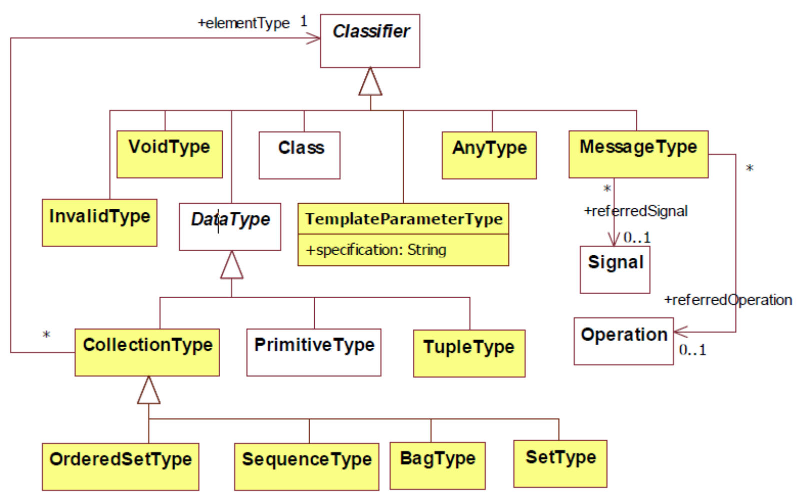
- UML-Classifier

Problem: Typhierarchie, Typkonformität

AnyType

AnyType is the metaclass of the special type *OclAny*, which is the type to which all other types conform. *OclAny* is the sole instance of *AnyType*. This metaclass allows defining the special property of being the generalization of all other Classifiers, including Classes, DataTypes, and PrimitiveTypes.

1 Aussage aus dem Standard



4 OCL-Tools bieten OclAny-Operationen auch für nicht Regel-konforme Typen an

Konflikt

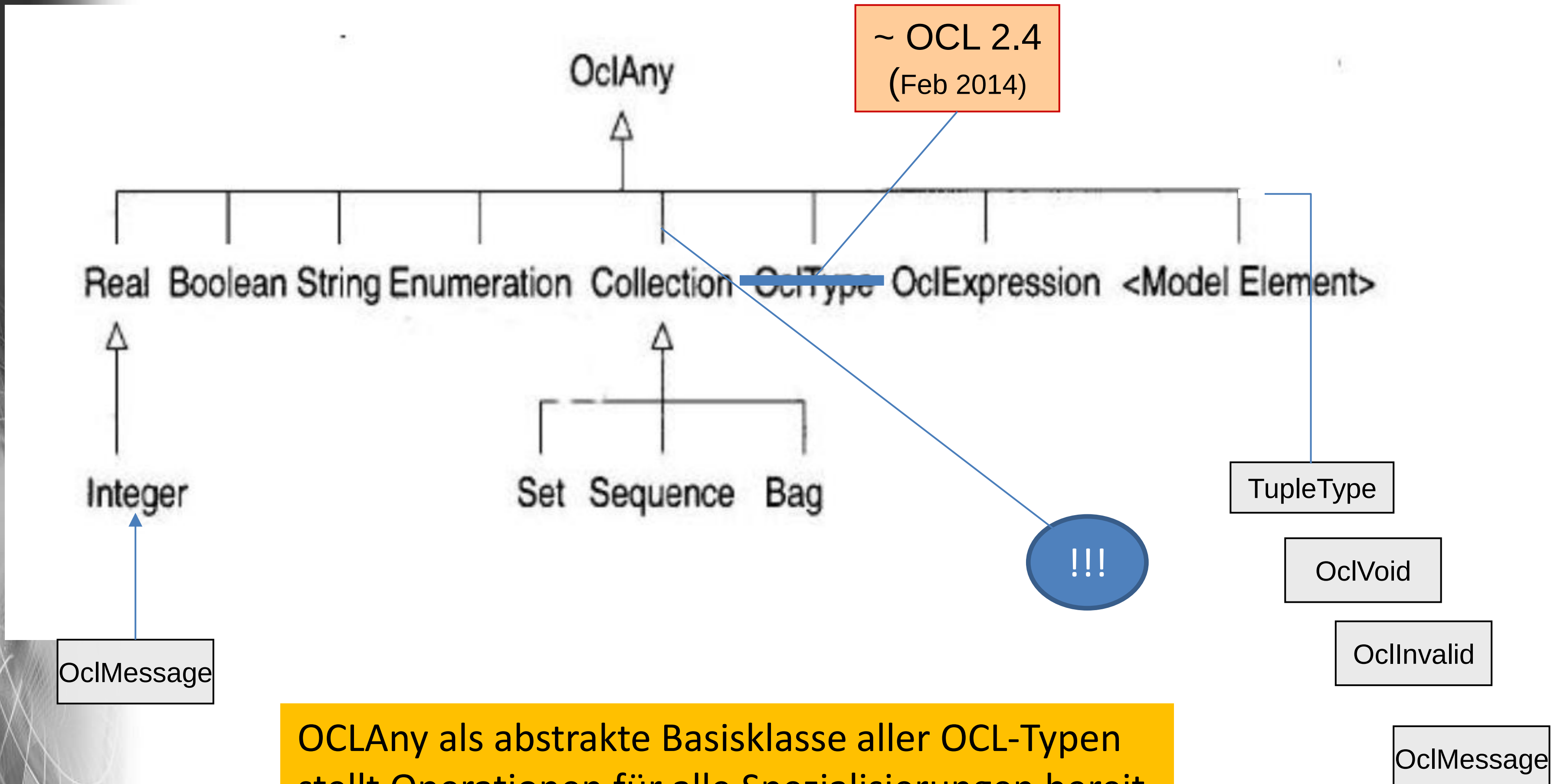
2 **Regeln (laut OCL-Standard)**

1. UnlimitedNatural ist ein Untertyp von Integer
2. Integer ist Untertyp von Real
3. alle Typen (bis auf Tupel und Collection) sind Untertypen von OclAny
4. Set(T), Sequence(T), Bag(T) sind Untertypen von Collection(T)

3 Einschränkung

- *OclAny* is the supertype of all other types except for the collection types. The exception has been introduced in UML because it considerably simplifies the type system [CKM+99]. A simple set inclusion semantics for subtype relationships as described in the next sub section would not be possible due to cyclic domain definitions if *OclAny* were the supertype of *Set*(*OclAny*).

Unsere pragmatische Lösung



OCLAny als abstrakte Basisklasse aller OCL-Typen stellt Operationen für alle Spezialisierungen bereit
OCL-Tools unterstützen das auch so

Typ-Konformität

- OCL-Ausdruck ist gültig:
  alle aktuellen Typen sind konform zu ihren formalen Typvereinbahrungen

- Beispiel:
  seien Fahrrad und PKW Untertypen von Fahrzeug

  – Set(Fahrrad) ist konform zu Set(Fahrzeug)
  – Set(Fahrrad) ist konform zu Collection(Fahrrad)
  – Set(Fahrrad) ist konform zu Collection(Fahrzeug)
  – Set(Fahrrad) ist **nicht** konform zu Bag(Fahrrad)

Beispiel: Integer konform zu Real (Wdh.)

```
context PrimitiveType
inv: (self.name = 'Integer') implies
    PrimitiveType.allInstances()->forAll (p | (p.name = 'Real') implies
                                              (self.conformsTo(p)))
```

In dieser Art werden im Standard Untertypen definiert

Navigationsausdrücke

- Assoziationsenden (Rollennamen) können verwendet werden, um von einem Object im Modell/System zu einem anderen zu navigieren (**Navigation**)

- Navigationen werden in OCL als Attribute behandelt (*dot-Notation*)

- Der Typ einer Navigation ist entweder
  - **Nutzerdefinierter Typ** (Assoziationsende mit Multiplizität maximal 1)
  - **Kollektion** von nutzerdefinierten Typen (Assoziationsende mit Multiplizität > 1)

Invariante

- **Definition**
  Eine **Invariante** ist ein Constraint,
  das für ein Objekt während seiner ganzen Lebenszeit wahr sein sollte.

- **Syntax**

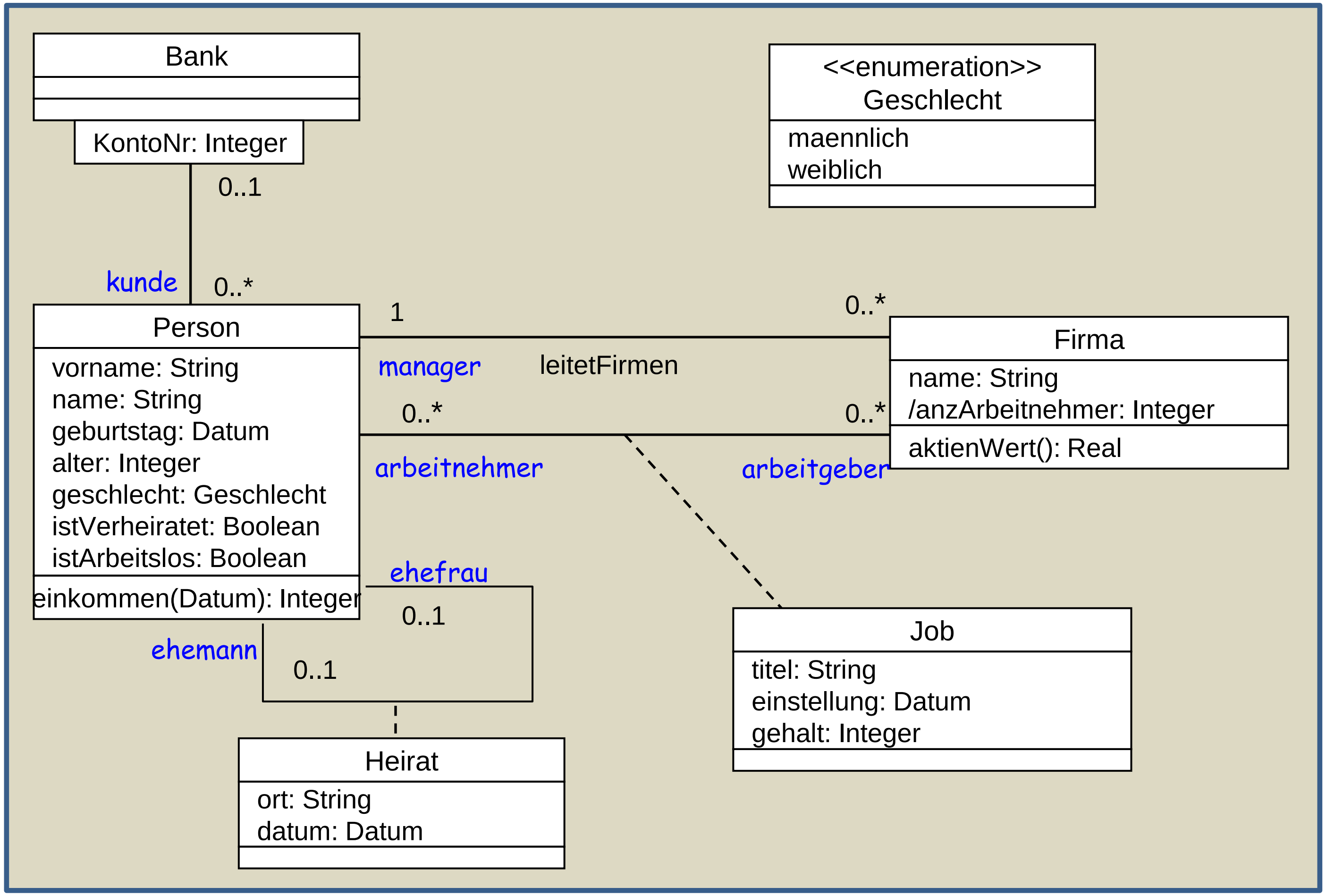
  context <class name>

  inv [<constraint name>]: <OCL expression>

Inhalt: OCL

~ OCL 2.4
(Feb 2014)

Erstes Beispiel



Einfache Invarianten

Festlegung eines **Constraints** , das für alle Instanzen des Typs gilt

- *Attributwerte anzArbeitnehmer aller Instanzen von Firma müssen kleiner-gleich 50 sein*

  context Firma **inv**: -- *self* bezieht sich auf das Objekt, für das das Constraint berechnet wird

  self. anzArbeitnehmer <= 50

- *Äquivalente Formulierung –f übernimmt Rolle von Self*

  context f: Firma

  inv KMU: f. anzArbeitnehmer <= 50 -- Vergabe eines Namens für das Constraint

- *Äqivalente Formulierung*

  context f: Firma

  inv anzArbeitnehmer <= 50 -- bei Eindeutigkeit des Namens

- *Der Aktien-Wert aller Firma-Instanzen ist größer als Null*

  context Firma **inv**:

  self. aktienWert() > 0

Sichtbarkeiten von Attributen u.ä. werden durch OCL standardmäßig ignoriert.

Precondition (Vorbedingung)

Pre- und Postconditions sind Constraints,

die die Anwendbarkeit und die Auswirkung von Operationen spezifizieren, ohne dass dafür ein Algorithmus oder eine Implementation angegeben wird.

Definition

Eine **Precondition** ist ein Boolescher Ausdruck, der zum Zeitpunkt des Beginns der Ausführung der zugehörigen Operation wahr sein muss.

Syntax

```
context <class name>::<operation> (<parameters>)
pre [<constraint name>]: <OCL expression>
```

Postcondition (Nachbedingung)

Definition

Eine **Postcondition** ist ein Boolescher Ausdruck, der unmittelbar nach der Ausführung der zugehörigen Operation wahr sein muss.

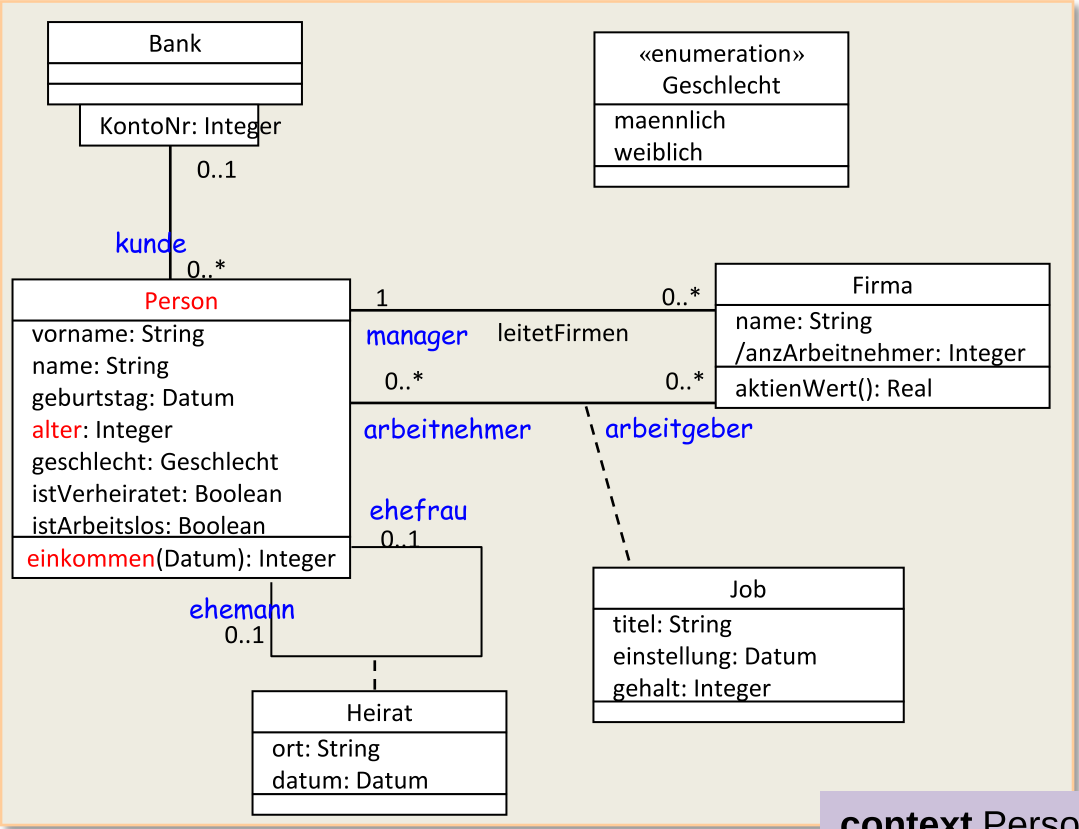
Syntax

context <class name>::<operation> (<parameters>)

post [<constraint name>]: <OCL expression>

Postcondition mit result

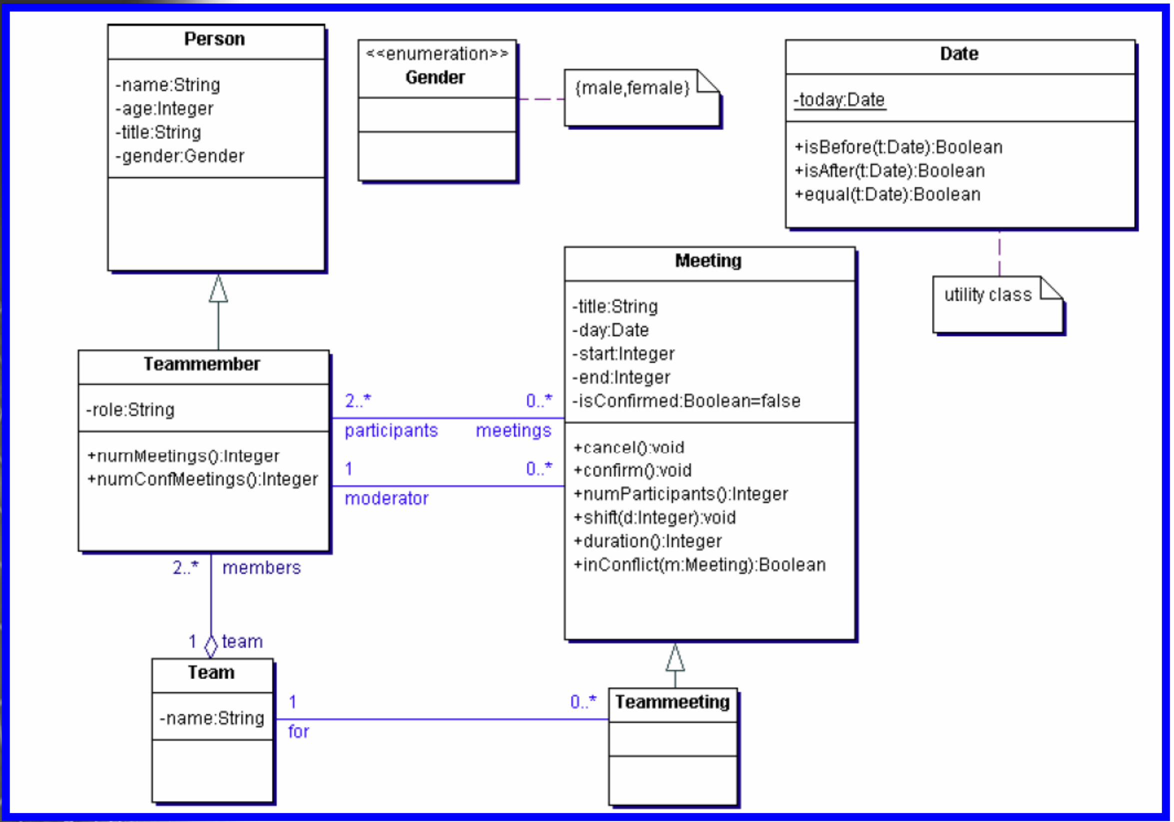
result bezieht sich auf den Rückgabewert der Operation



context Person::einkommen (d: Datum) : Integer
    pre: alter > 18
    post: result >= 2000

context Firma **inv**:
    self**.** aktienWert() > 0.0   -- Bezugnahme auf parameterlose FKt.

@pre-Operator



Zweites Beispiel

context Meeting::shift(d: Integer)

post: start = start**@pre** +d **and** end = end**@pre** + d

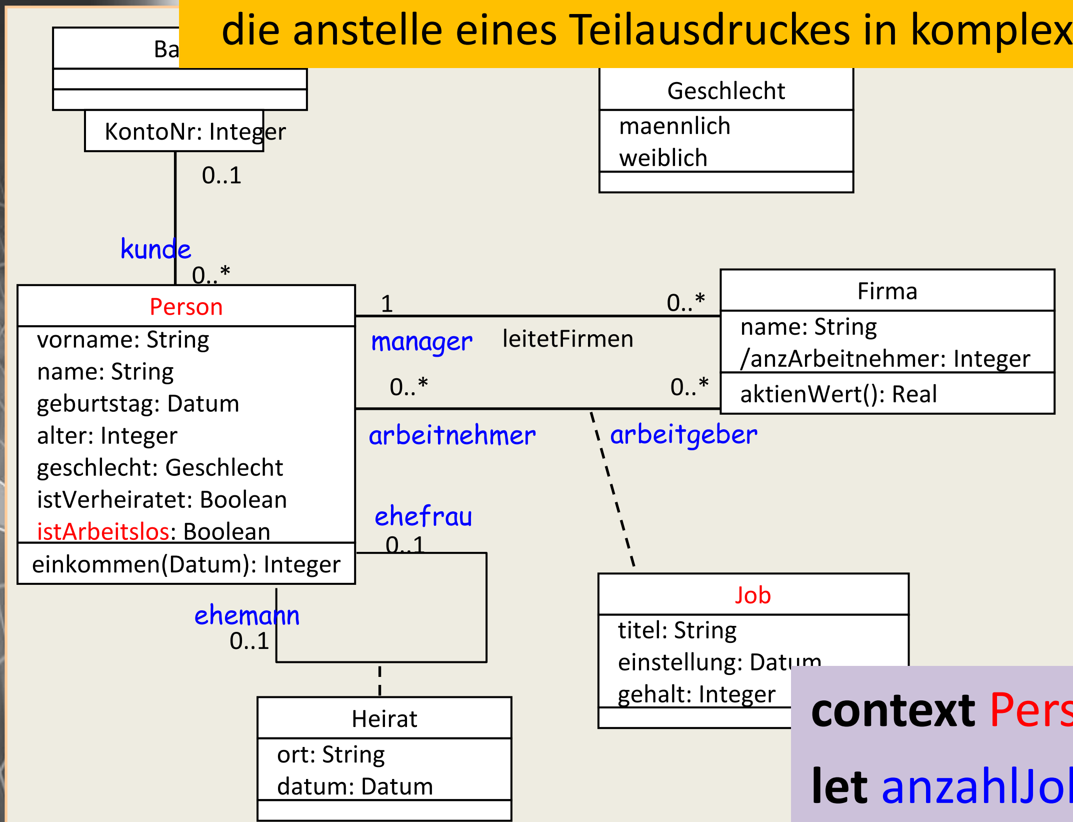
-- ***start@pre*** bezieht sich auf den Wert **vor** Ausführung der Operation

-- *start* bezieht sich auf den Wert **nach** Ausführung der Operation

-- **@pre** ist nur in Postconditions erlaubt

Let-in-Operator: Teilausdruck in OCL

Ein let-Ausdruck definiert eine Variable (z.B. anzahlJobs), die anstelle eines Teilausdruckes in komplexen Ausdrücken benutzt werden kann.



Standardoperation von Collection

context Person **inv**:

let anzahlJobs: Integer = self.job->size() **in**

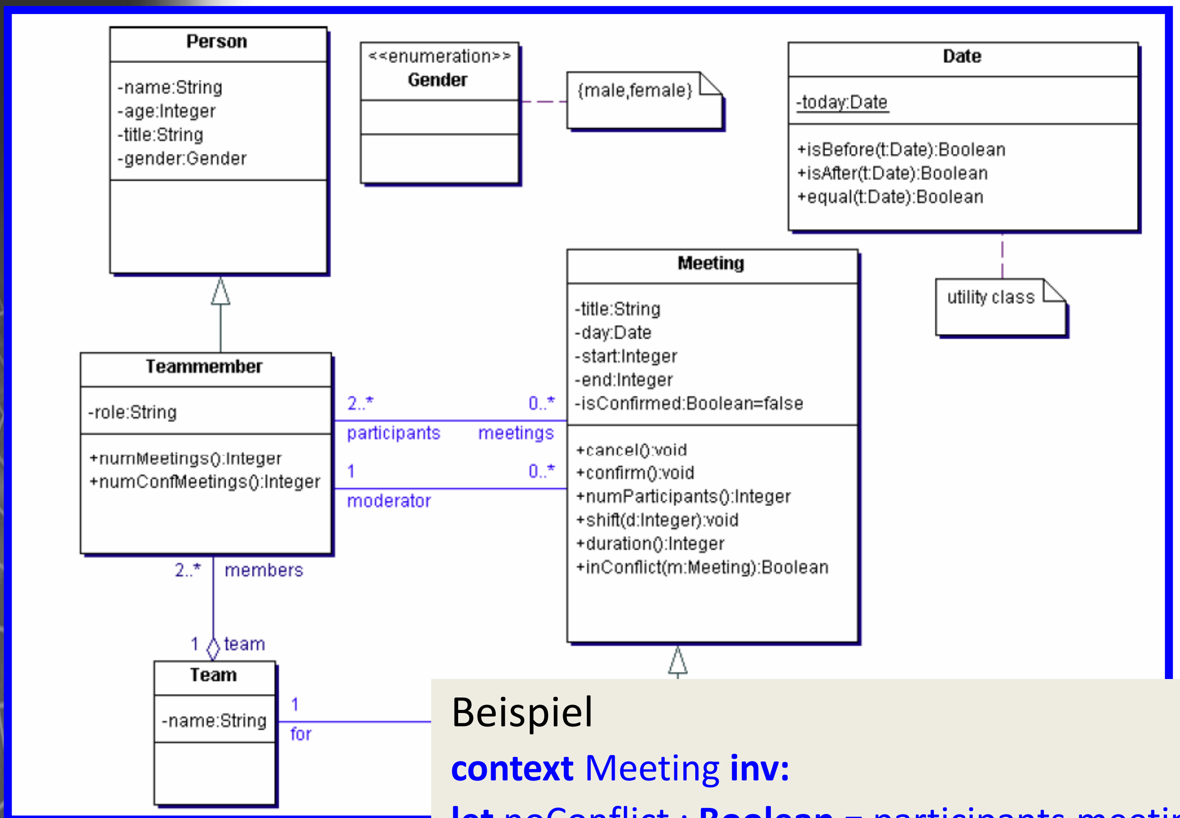
if istArbeitslos **then** anzahlJobs = 0

else anzahlJobs > 0

endif

size
liefert für **leere** Menge: 0
Null-Test ist nicht erforderlich

Weiteres Beispiel: let-in



Beispiel

context Meeting **inv:**

let noConflict : **Boolean** = participants.meetings->**forAll** (m|m<>**self**

and

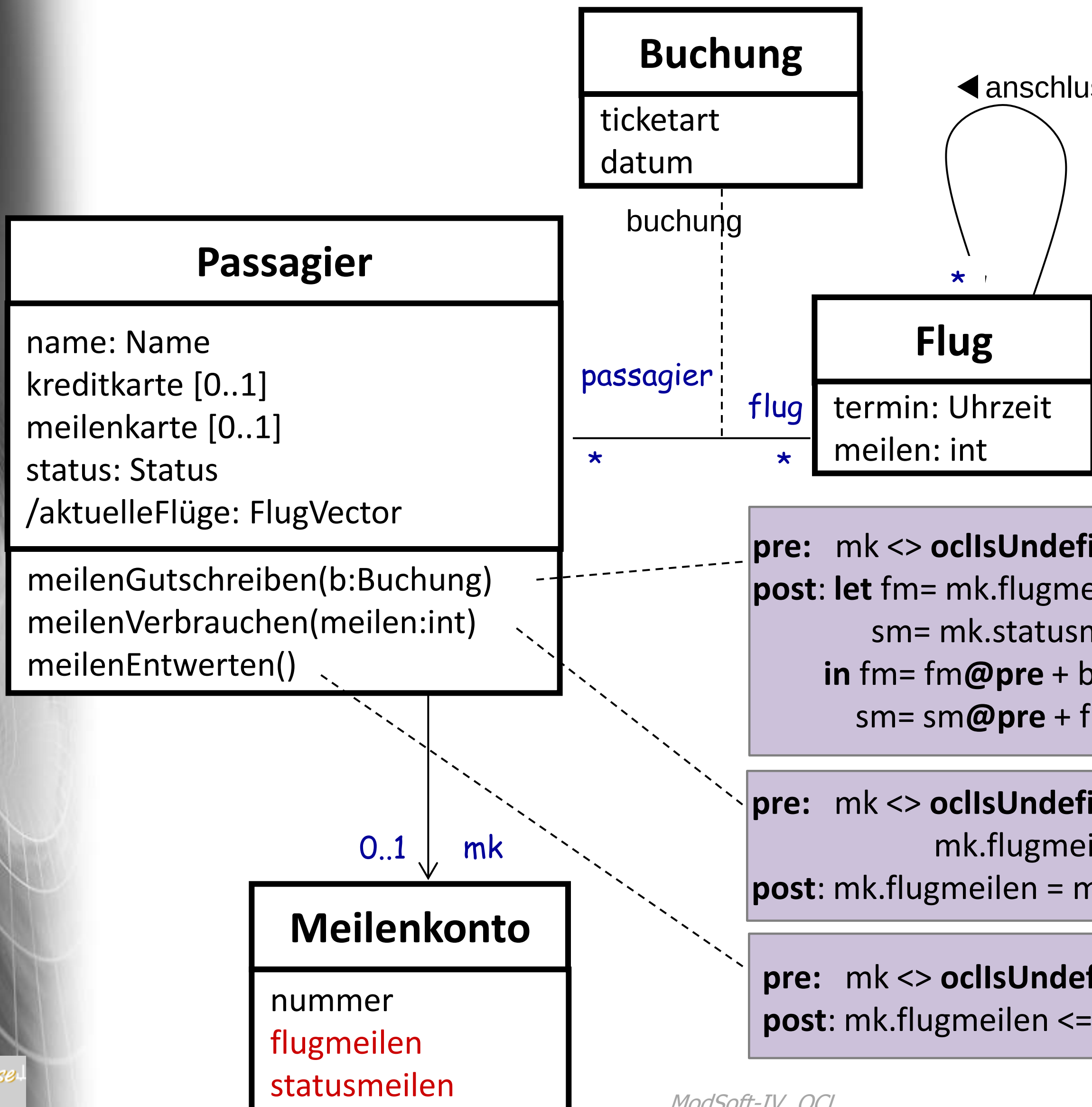
m.isConfirmed **implies not** self. inConflict(m)

)

in

isConfirmed **implies** noConflict

Beispiel: @pre, postcondition und let-in



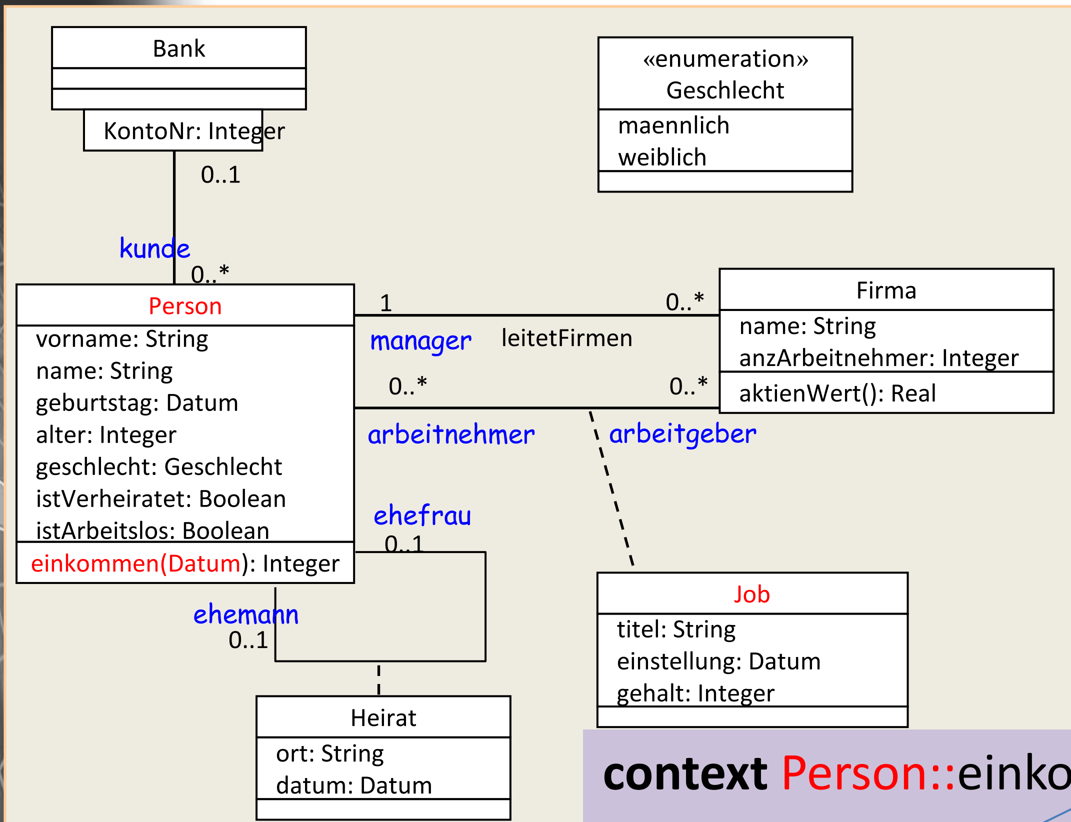
pre: mk <> **oclIsUndefined**
post: **let** fm= mk.flugmeilen,
sm= mk.statusmeilen
in fm= fm**@pre** + b.flug.meilen **and**
sm= sm**@pre** + fm**@pre** * b.ticketart

pre: mk <> **oclIsUndefined and**
mk.flugmeilen>= meilen
post: mk.flugmeilen = mk.flugmeilen**@pre** - meilen

pre: mk <> **oclIsUndefined**
post: mk.flugmeilen <= mk.flugmeilen**@pre**

body-Ausdruck

… wird benutzt um Resultat von query-Operationen anzuzeigen



Standardoperation von Collection

context Person::einkommen (Datum: d): Integer

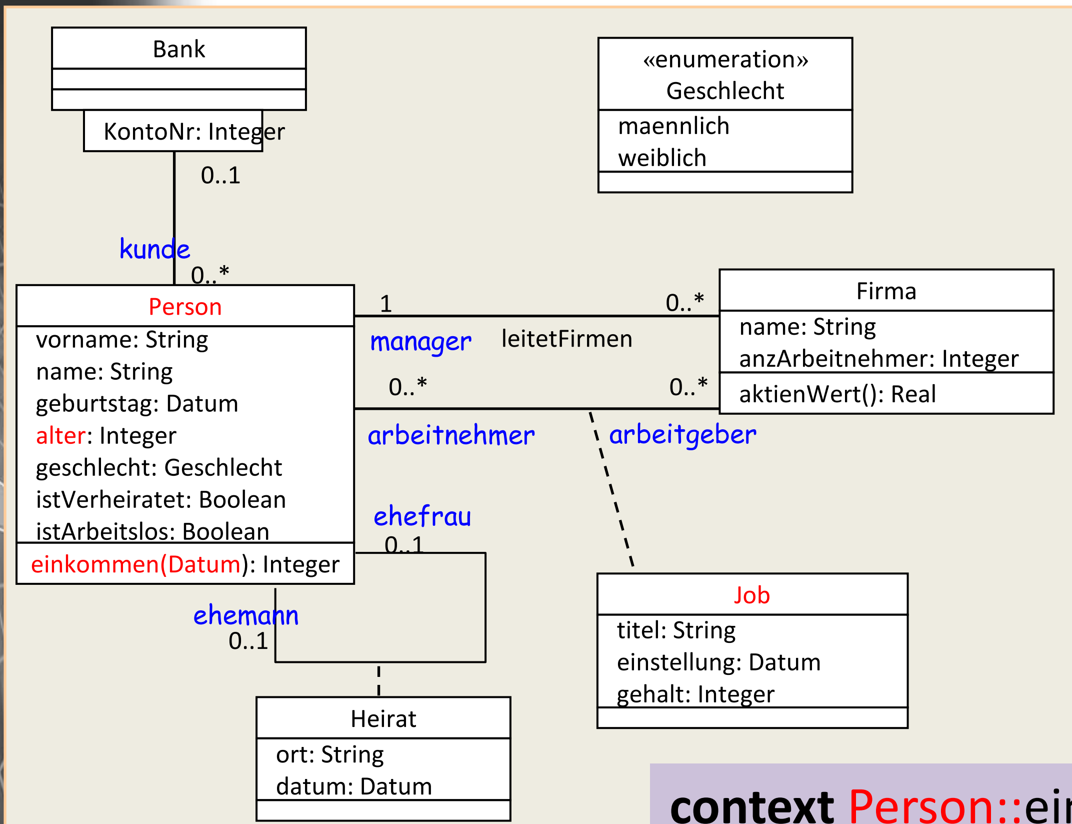
body: self.job->collect(gehalt)->sum()

Kurzform

context Person::einkommen (Datum: d): Integer

body: self.job.gehalt->sum()

body- und postcondition



Mischform

context Person::einkommen (Datum: d): Integer

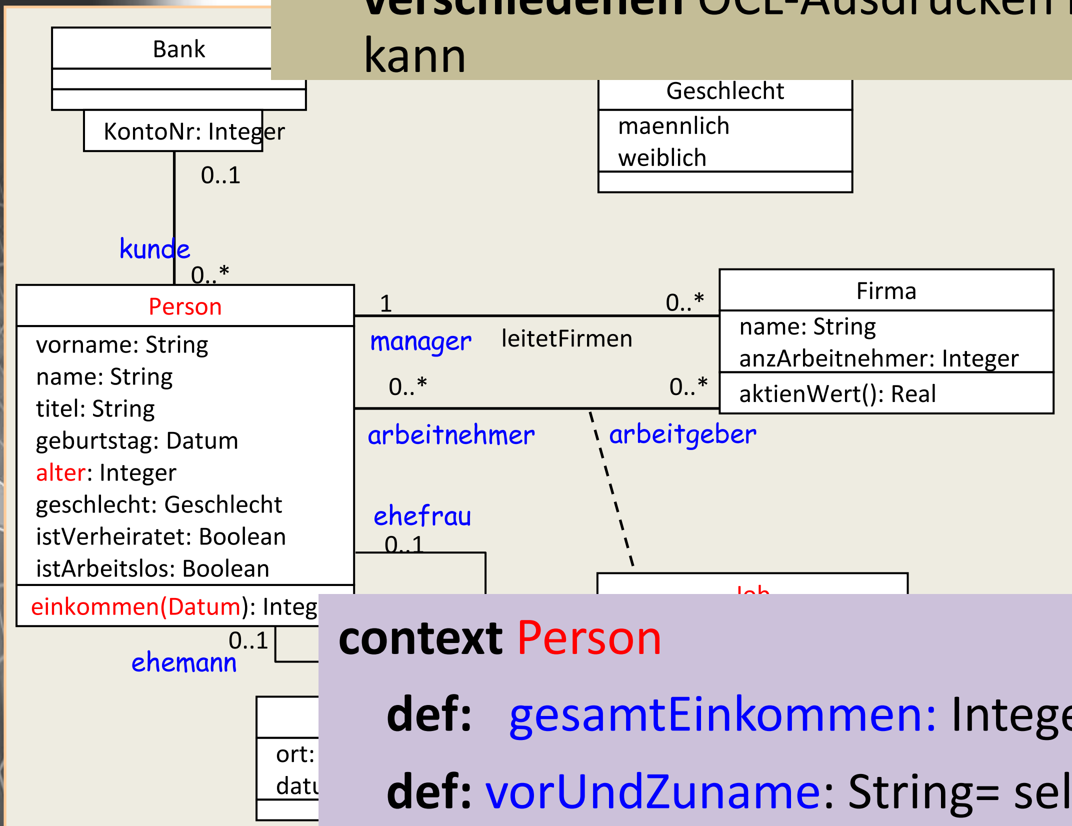
pre: self.alter > 18

body: self.job.gehalt->sum()

post: result < 5000

Definitions-Ausdruck

... erlaubt eine globale Definition von Teilausdrücken, die in **verschiedenen** OCL-Ausdrücken mehrmals verwendet werden kann



Standardoperation von String → String

context Person

def: gesamtEinkommen: Integer = self.job.gehalt->sum()

def: vorUndZuname: String= self.vorname.concat(' ').concat(name)

def: hatTitel(t: String): Boolean **=** self.job->exists(titel =t)

Bezeichner dürfen nicht in Konflikt zu Attributnamen oder Assoziationsenden stehen

Standardoperation von Collection → Boolean

Weiteres komplexeres Beispiel

für Definitionsausdruck statistik im Kontext von Person,

der mehrmals bei einer **Postcondition** einer Operation

- von Firma angewendet wird, die ihrerseits ein Objekt von Person als Parameter führt, für den statistik angewendet wird

```
istTopArbeitnehmer (p:Person) : Boolean
```

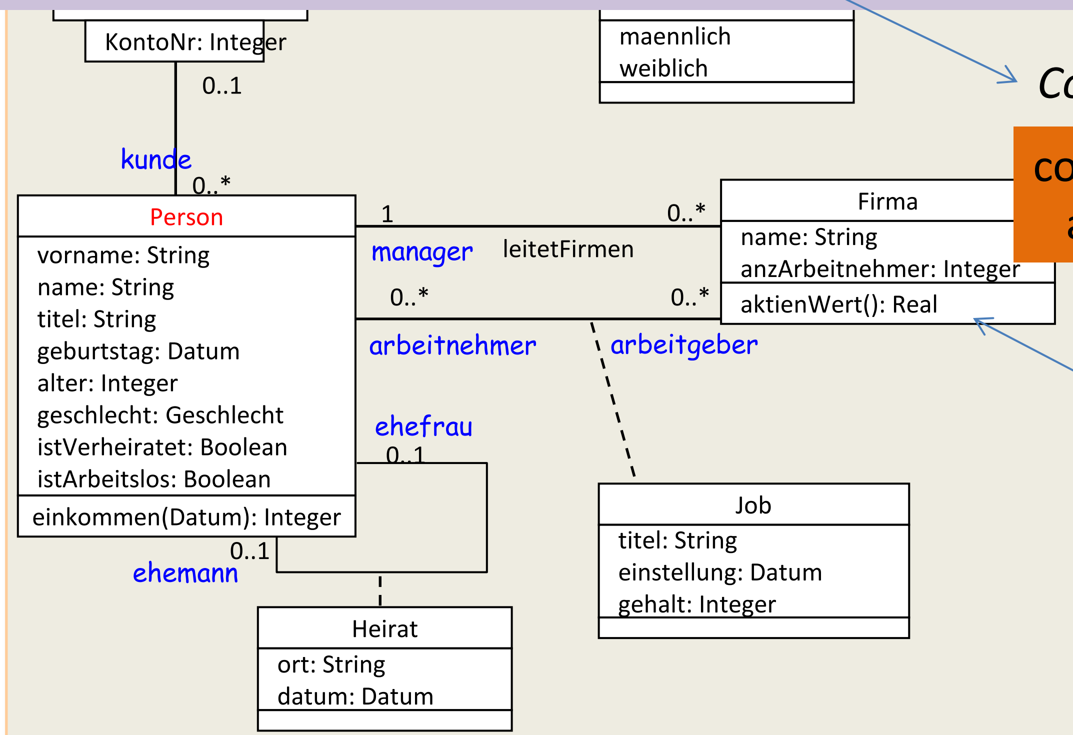
```
context Person def:      -- Multimenge von Tupeln (ein Tupel je von einer Person geleitete Firma)
        statistik: Set (TupleType {
                fi: Firma,
                anzAN: Integer,
                gutVerdiener: Set (Person),
                gesamtGehalt: Integer
        })        =
        firma-> collect( f: Firma |Tuple {
                fi: Firma= f,
                anzAN: Integer= f.arbeitnehmer->size(),
                gutVerdiener: Set(Person)= f.job->select(gehalt>10000).arbeitnehmer->asSet(),
                gesamtGehalt: Integer= f.job.gehalt→sum()
        })
```

PROBLEM
collect kann
nur Attribute von
Firma-Kollektion
einsammeln und
eine neue Menge bilden

Hier soll aber eine Menge nach obigem Muster gebaut werden, wobei die Elemente (als Tupel) Attribute der Iterationsvariable f (Firma-Instanzen übernehmen)

Collect-Operation-Definition

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



istTopArbeitnehmer (p:Person) : Boolean

Definition von statistik

context Person **def**: -- Multimenge von Tupeln (ein Tupel je von einer Person geleitete Firma)

statistik: Set (TupleType {fi: Firma, anzAN: Integer,
gutVerdiener: Set (Person), gesamtGehalt: Integer })

=

firma-> iterate (f: Firma;

acc: Set(Tuple (

fi~~: Firma~~,

anzAN~~: Integer~~,

gutVerdiener~~: Set(Person)~~,

gesamtGehalt~~: Integer~~))

= Set{}

| acc->including(Tuple

{ fi = f,

anzAN = f.arbeitnehmer->size(),

gutVerdiener = f.job->select(gehalt > 1000) .arbeitnehmer->asSet(),

gesamtGehalt = f.job.gehalt->sum()

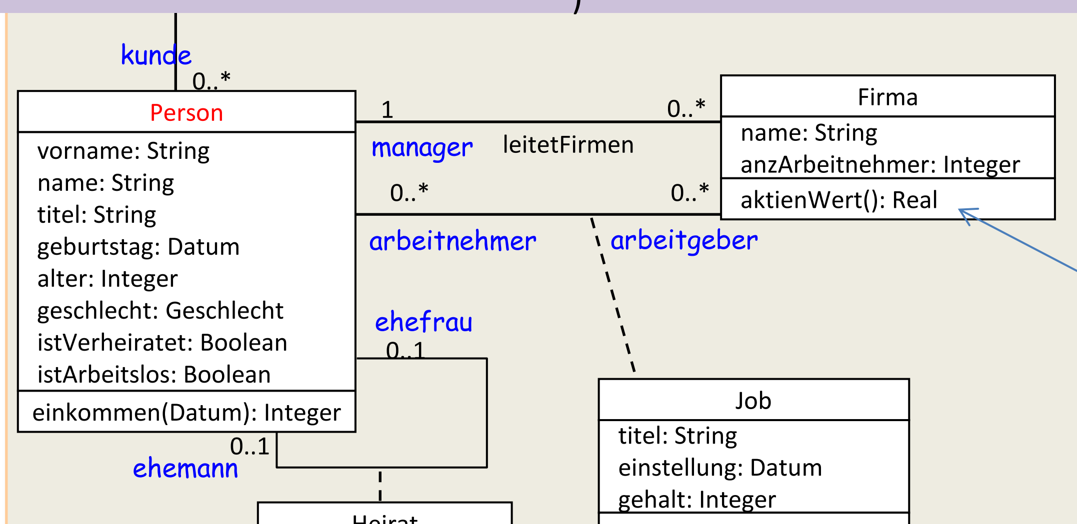
})

)

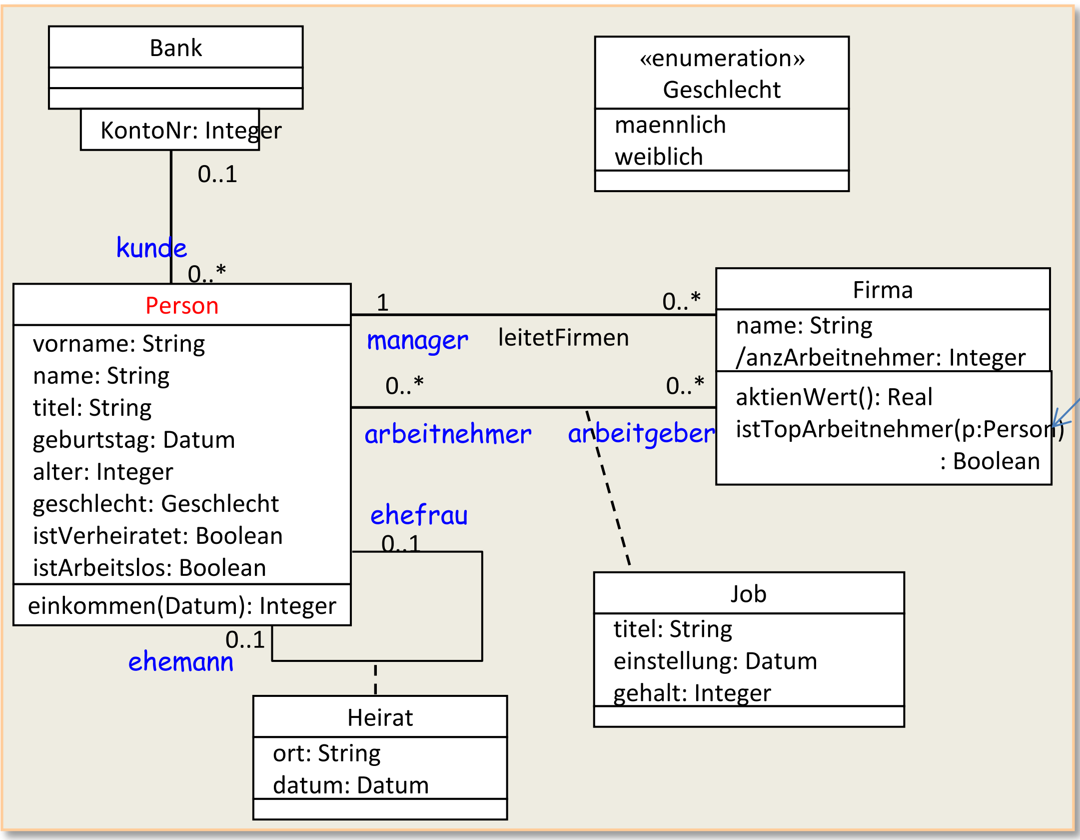
Typangabe kann entfallen

Starten mit leerer Menge und fügen in jedem Iterationsschritt ein Tupel je betrachteter Firma hinzu

Korrigierte Definition von statistik



istTopArbeitnehmer (p:Person) : Boolean



istTopArbeitnehmer (p:Person) : Boolean

Anwendung von statistik

context Firma:: istTopArbeitnehmer (p:Person) : Boolean

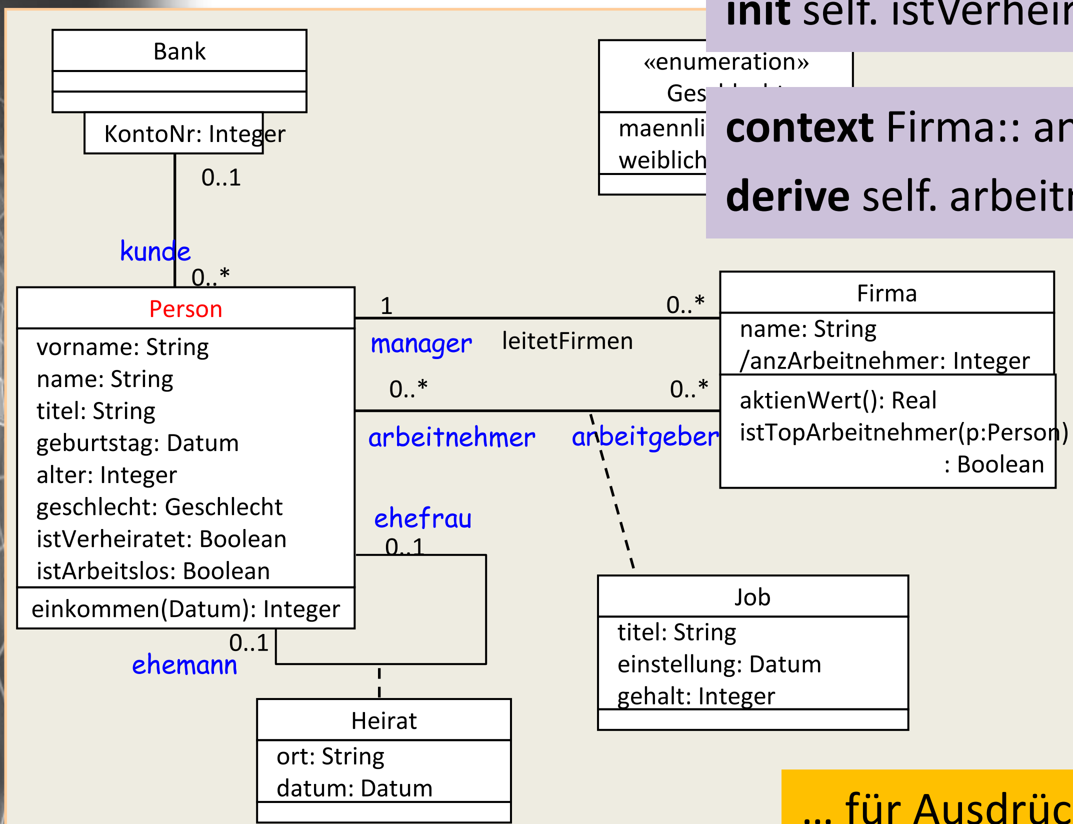
body: p. statistik->sortedBy (gesamtGehalt)->last().firma = self)

Initiale und abgeleitete Werte

… für Attribute oder Assoziationsenden

context Person:: istVerheiratet: Boolean
init self. istVerheiratet= false

context Firma:: anzArbeitnehmer: Integer
derive self. arbeitnehmer->size()



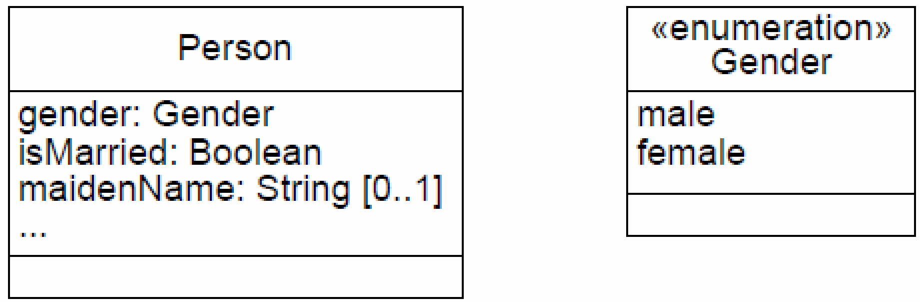
Typkonformität
ist sicherzustellen

… für Ausdrücke über Assoziationsenden gilt:
Typkonformität ~ Multiplizität

Classifier

Set(Classifier), OrderedSet(Classifier)

Aufzählungstypen



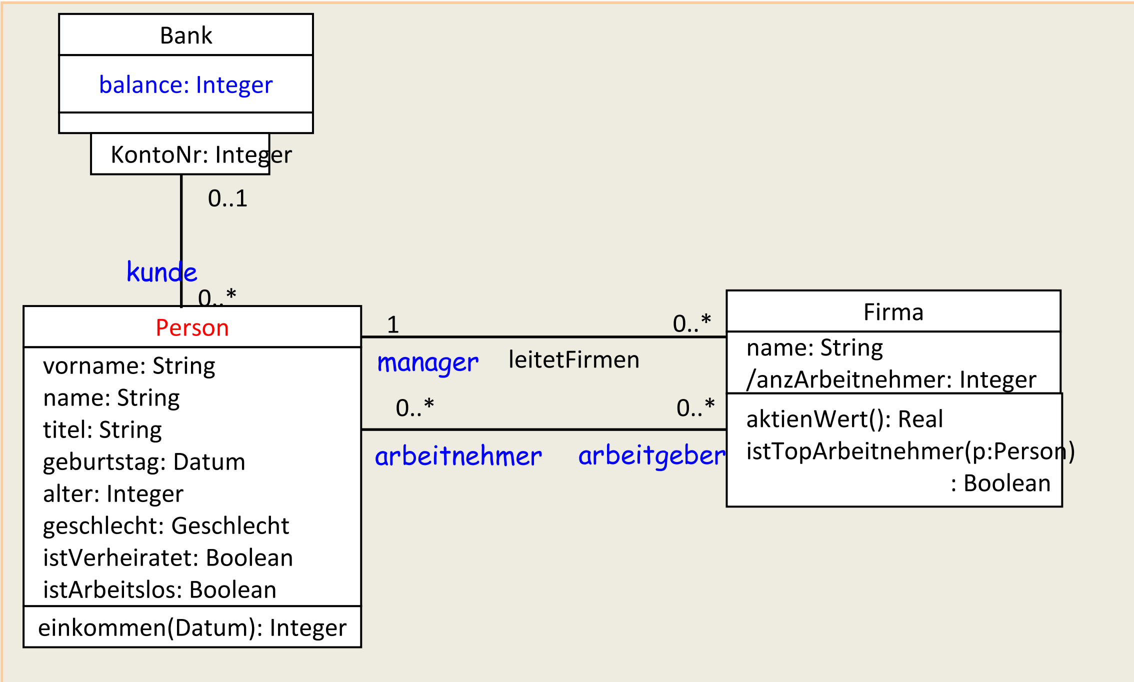
context Person **inv**:

self. maidenName <> ' ' **implies**

self.gender = Gender::female **and** self.isMarried = true

~ nur verheiratete Frauen besitzen einen Mädchennamen

Umgang mit Assoziationen



context Firma

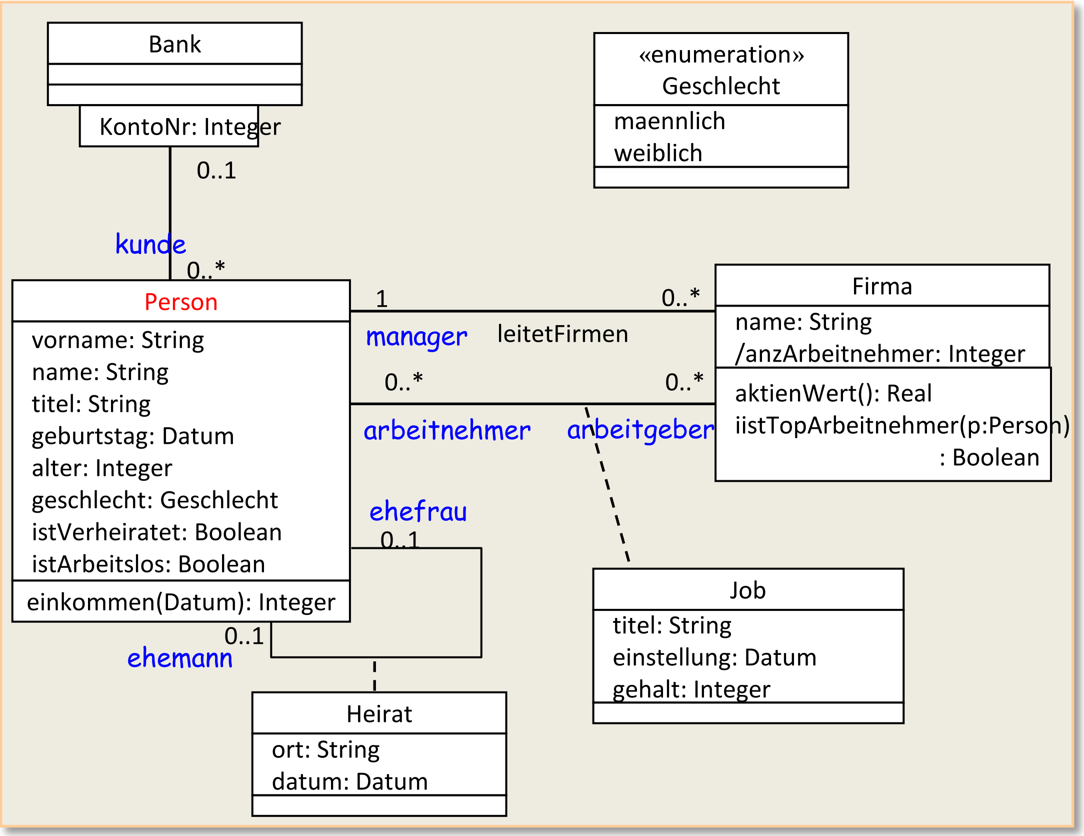
inv: self.manager.istArbeitslos = false -- Attributwert ist ein Objekt

inv: self.arbeitnehmer->notEmpty() -- Attributwert ist eine Menge von Objekten

context Person

inv: self. bank. balance >= 0 -- Assoziationsende ~ kleingeschriebener Typ

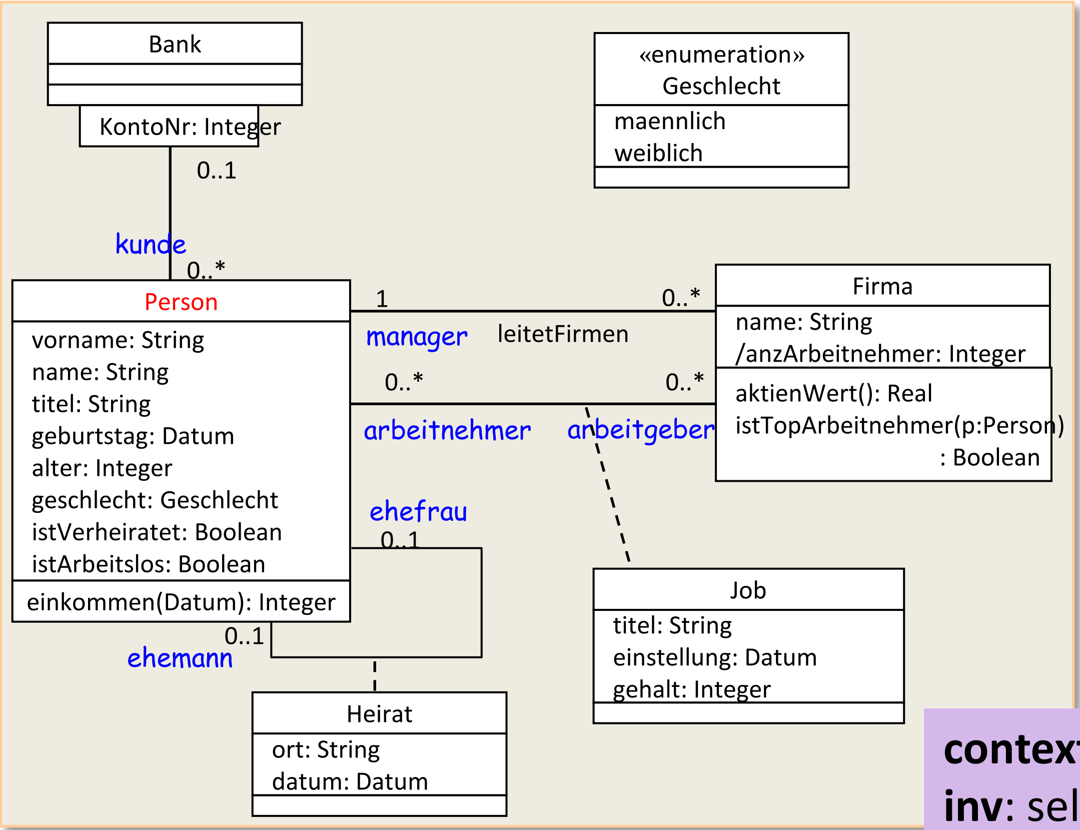
Prüfung einer Objektexistenz vor Navigation



gleichgeschlechtliche Ehen wären mit diesem Constraint damit ausgeschlossen

context Person **inv**:
self.ehefrau->notEmpty() **implies** self.geschlecht = Geschlecht::maennlich **and**
self.ehemann->notEmpty() **implies** self.geschlecht = Geschlecht::weiblich

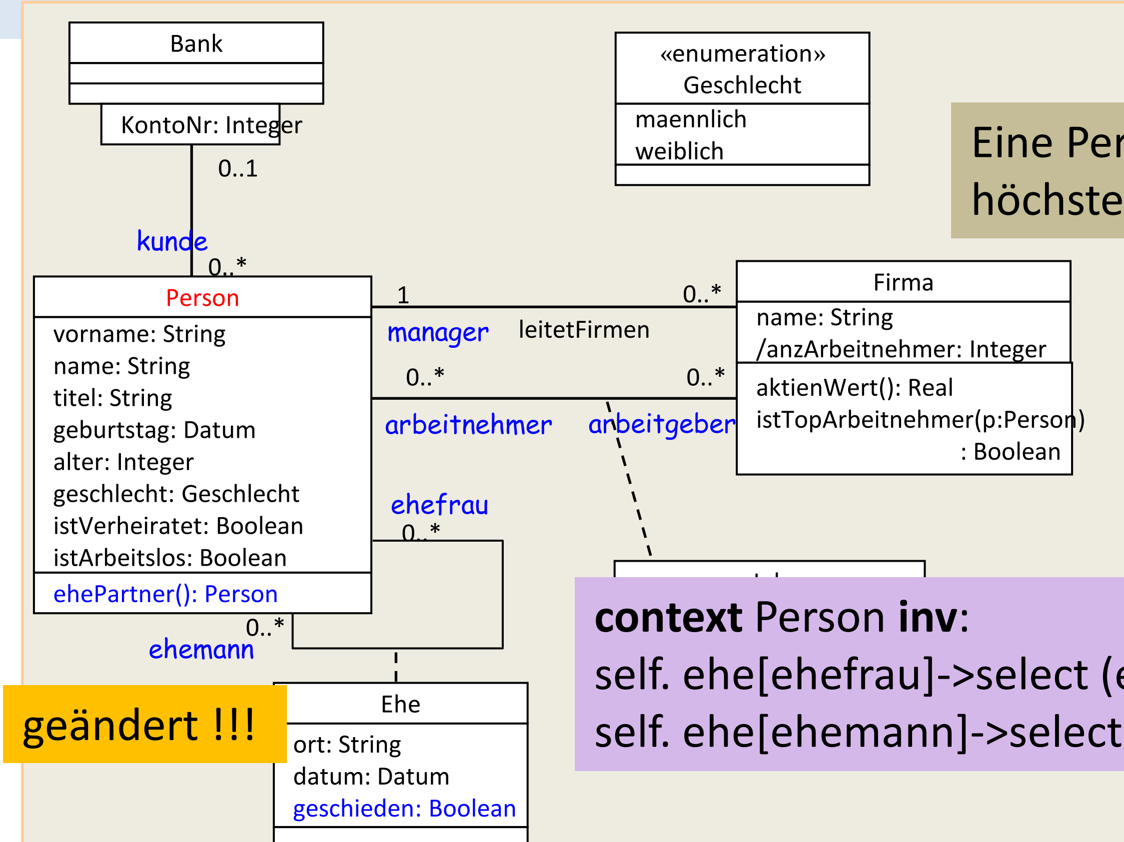
Assoziationsklasse



context Person
inv: self. istArbeitslos = false
implies self. job->size() >= 1

context Job
inv: self. arbeitgeber. anzArbeitnehmer >= 1
inv: self. arbeitnehmer. alter >= 18

Rekursive Assoziationsklasse



Eine Person ist zu einem Zeitpunkt mit höchstens einer anderen Person verheiratet

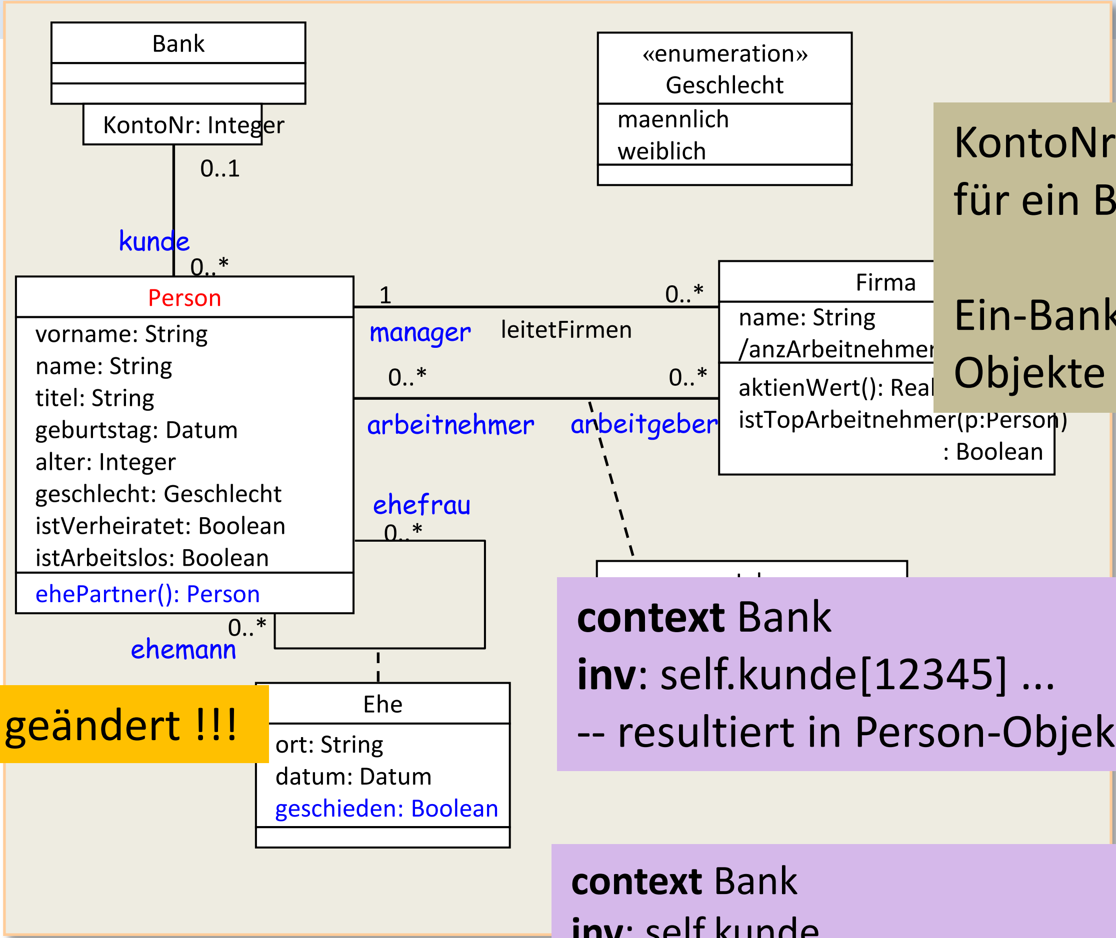
geändert !!!

context Person **inv**:
self. ehe[ehefrau]->select (e | e.geschieden = false)->size()=1 **and**
self. ehe[ehemann]->select (e | e.geschieden = false)->size()=1

In Ziel-Rolle

context Person::ehePartner(): Person
pre: self.istVerheiratet = true
body:
if geschlecht= Geschlecht::maennlich
then
self.ehe[ehefrau] ->select(e | e.geschieden = false).ehefrau
else self.ehe[ehemann]->select(e | e.geschieden = false).ehemann
endif

Qualifizierte Assoziation



geändert !!!

KontoNr ist ein Qualifier-Attribut von Person für ein Bank-Objekt

Ein-Bank-Objekt benutzt KontoNr um Kunden-Objekte identifizieren zu können

context Bank
inv: self.kunde[12345] ...
-- resultiert in Person-Objekt mit der Kontonummer 12345

context Bank
inv: self.kunde ...
-- resultiert in einem Set(Person)

Allgemeine anwendbare Operationen (Wdh.)

- allInstances(): Set (T) /* T ist Typ von Self */

Anwendbar auf Classifier nicht auf Objekte!!!

- = (OclAny): Boolean
- <> (OclAny): Boolean

- oclAsSet() : Set(T)
- oclIsNew (): Boolean /* Objekterzeugung während einer Op-Ausführung */
  /* als Differenz von Pre- und PostCondition */
- oclIsUndefined (): Boolean /* Test auf null */
- oclIsInvalid (): Boolean
- oclAsType (t : Classifier): instance of Classifier /* cast */
- oclIsTypeOf (t : Classifier): Boolean
- oclIsKindOf (t : Classifier): Boolean /* transitive Hülle*/
- oclType() : Classifier
- oclIsInState (s: OclState): Boolean

Tools:
anwendbar auf Operanden deren Typ nicht konform zu OclAny ist.

Inhalt: OCL

~ OCL 2.4
(Feb 2014)

OCL Constraints und Vererbung

Constraints allgemein

- Constraints einer Superklasse werden von den Subklassen geerbt.

Invarianten

- Eine Subklasse kann die Invariante verstärken, sie aber nicht abschwächen.

Preconditions

- Eine Vorbedingung kann bei einem Überschreiben einer Operation einer Subklasse aufgeweicht, aber nicht verstärkt werden.

Postconditions

- Eine Nachbedingung kann bei einem Überschreiben einer Operation einer Subklasse verstärkt, aber nicht aufgeweicht werden.

Präzedenzregeln für OCL-Operatoren

- **@pre** -- Wert eines Attributs zum Zeitpunkt der Precondition
  -- (benutzt in Postcondition-Ausdrücken)
- .
  →
- **not**
  -
  (unär)
- *
  /
- +
  -
  (binär)
- **if**, **then else**, **endif**
- <, <=
  >, >=
  =, <>
- **and**, **or**, **xor**
- **implies**

abnehmende Priorität

Klammern verändern
wie üblich die Rangfolge

OCL- Schlüsselworte

if
then
else
endif
not
let
or
and
xor

implies
endpackage
package
context
def
inv
pre
post
in

reserviert Bezeichner
self, result

Inhalt: OCL

~ OCL 2.4
(Feb 2014)

Operationen auf Kollektionen (1)

- 22 Operationen mit unterschiedlicher Semantik ~ Kollektionstyp, z.B.
  - Vergleichsoperationen **=**, **<>**
  - Konvertierungsoperationen
    asBag(), asSet(), asOrderedSet(), asSequence()
  - verschiedene including- und excluding-Operationen
  - Operation flatten() erzeugt aus einer Kollektion von Kollektionen eine Kollektion mit einzelnen Objekten, z.B.
    Set{Bag{1,2,2}, Bag{2}} → Set{1,2}
  - Mengenoperationen
    union, intersection, minus, symmetricDifference
  - Operationen auf sortierten Kollektionen
    z.B. first(), last(), indexOf()

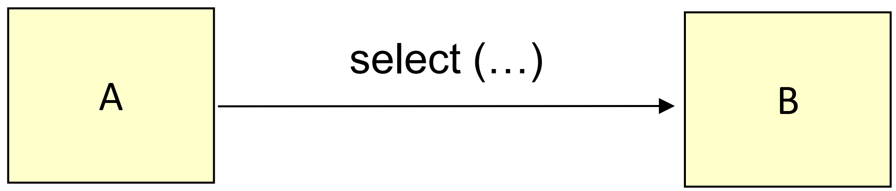
Operationen auf Kollektionen (2)

Iterierende Operationen auf allen Kollektionstypen, z.B.

- any(expr)
- collect(expr)
- exists(expr)
- forAll(expr)
- isUnique(expr)
- one(expr)
- select(expr)
- reject(expr)
- sortedBy(expr)

$B \subseteq A$

Collection-Operation: select



Teilmenge der Ausgangs-kollektion

$B \subseteq A$

A-> select (boolean-expr)
A-> select (v | boolean-expr-with-v)
A-> select (v: Type: | boolean-expr-with-v)

erfüllte Bedingung bestimmt die Objekt-Auslese

context Firma

inv: self. angestellte -> select(alter > 50)-> notEmpty()

context Firma

inv: self. angestellte -> select(p | p.alter > 50) -> notEmpty()

context Firma

inv: self. angestellte -> select(p: Person | p.alter > 50)-> notEmpty()

Collection-Operation: reject

$$A \xrightarrow{\text{reject (...)}} B \qquad B \subseteq A$$

A-> reject(boolean-expr)
A-> reject (v | boolean-expr-with-v)
A-> reject(v: Type: | boolean-expr-with-v)

nicht erfüllte Bedingung bestimmt die Objekt-Auslese

allg. Zusammenhang (äquivalente Ausdrücke)

A-> reject (v: Type | boolean-expr-with-v)

A -> select (v: Type | **not** boolean-expr-with-v)

context Firma **inv**:
self.arbeitnehmer->reject(alter>=18)->isEmpty()

$B \subseteq A$

Collection-Operation: collect

$$A \xrightarrow{\text{collect (...)}} B$$

A und B enthalten **unterschiedliche** Objekte (keine Teilmengen-Beziehung !)

Person

vorname: String
name: String
geburtstag: Datum
alter: Integer
geschlecht: Geschlecht
istVerheiratet: Boolean
istArbeitslos: Boolean

einkommen(Datum): Integer

context Firma

inv: self. angestellte -> collect(geburtstag) -> notEmpty()

context Firma

inv: self. angestellte -> collect(p | p.geburtstag) -> notEmpty()

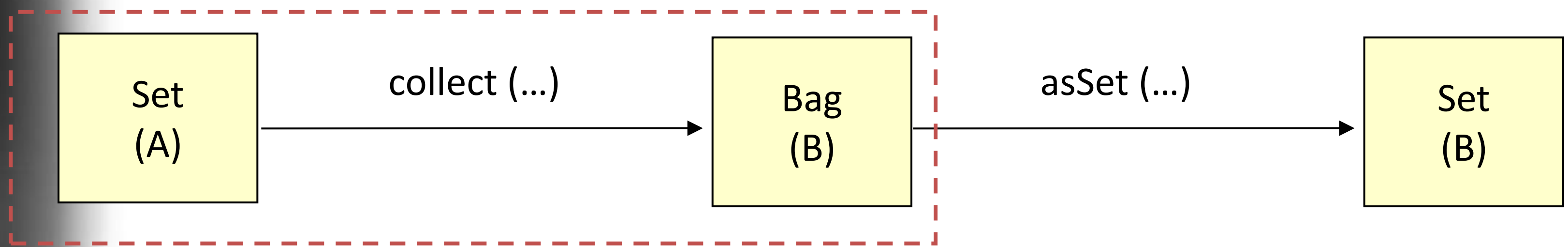
context Firma

inv: self. angestellte -> collect(p: Person | p.geburtstag) -> notEmpty()

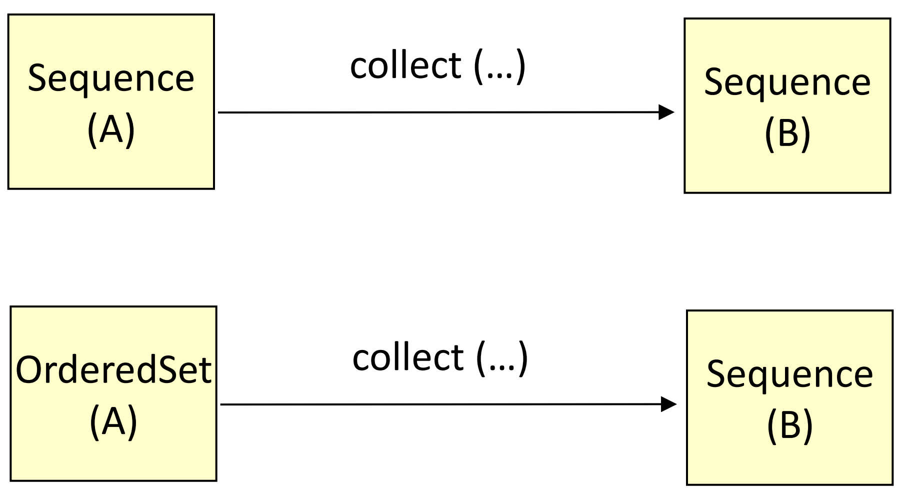
$$\text{Set (Person)} \xrightarrow{\text{collect (...)}} \text{Bag (Datum)}$$

gleiche Geburtstage können vorkommen

Select: originale u. resultierende Kollektion



Kardinalitäten von orginaler und
resultierender Kollektion sind gleich



Shorthand-Notation: collect (Wdh.)

Vereinfachung von
Navigationsangaben:

self.angestellte -> collect(geburtstag)

self.angestellte.geburtstag

Set(Datum)

automatische Interpretation als Kollektion über spezifische Eigenschaft

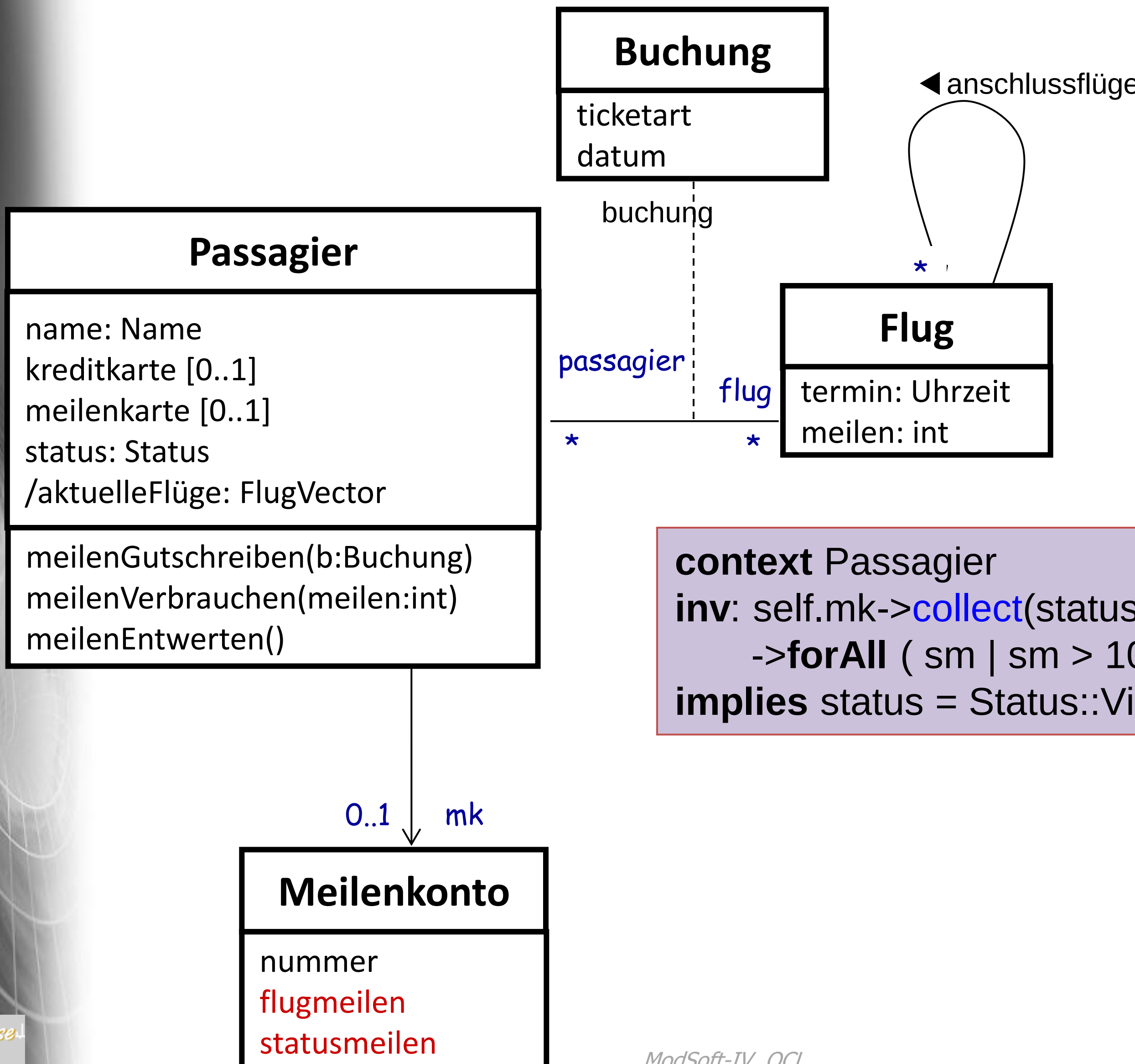
sogar, wenn Eigenschaft parametrisiert sein sollte

Allgemeine Stolperstelle

context Meeting
inv: self->collect(participants)->size()>=2

context Meeting
inv: self.participants->size()>=2

Flugabfertigung: collect



context Passagier
inv: self.mk->collect(statusmeilen)
->**forAll** (sm | sm > 10000)
implies status = Status::Vielflieger

Operation iterate()

Syntax:

Collection->iterate { element : Type1;
result : Type2 = <expression>
| <expression with element and result> }

Alle anderen iterierenden Operationen sind ein Spezialfall von iterate() und können damit ausgedrückt werden, z.B.

Set {1,2,3}->sum()
durch
Set{1,2,3}-> iterate {i: Integer, sum: Integer=0 | sum + i }

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

Collect-Operation

forAll- Operation

```
A-> forAll ( boolean-expr )
A-> forAll ( v | boolean-expr-with-v )
A-> forAll ( v: Type: | boolean-expr-with-v )
```

- Das Alter jedes Angestellten ist kleiner gleich 65

```
context Firma
inv: self.angestellte->forAll (alter <= 65)
inv: self.angestellte->forAll (p | p.alter <= 65)
inv: self.angestellte->forAll (p: Person | p.alter <= 65)
```

- Alle Personen haben verschiedene Namen

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

exists- Operation

A-> exists(boolean-expr)
A-> exists(v | boolean-expr-with-v)
A-> exists(v: Type: | boolean-expr-with-v)

- Der Vorname Otto kommt mindestens einmal bei den Angestellten vor

context Firma
inv: self. angestellte->exists (firstName = 'Otto')
inv: self. angestellte->exists (p | p.firstName = 'Otto‘)
inv: self. angestellte->exists (p: Person | p.firstName = 'Otto')

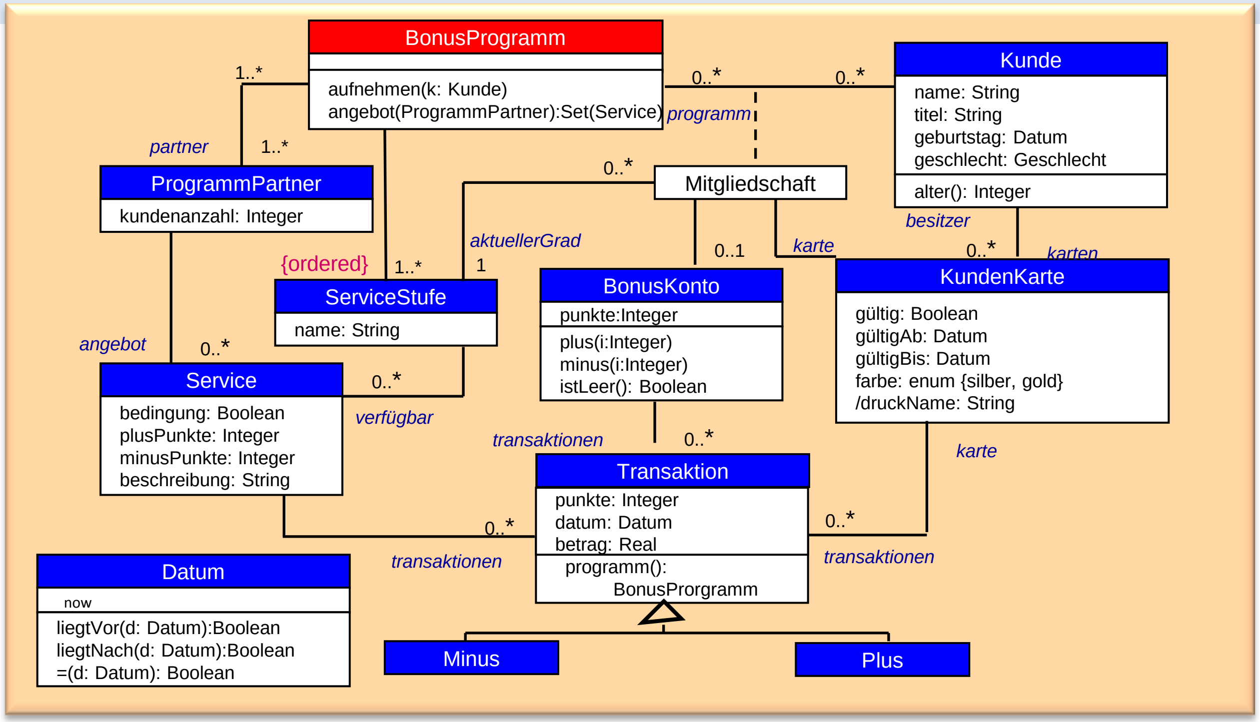
- Alle Personen haben verschiedene Namen

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

Inhalt: OCL

~ OCL 2.4
(Feb 2014)

BonusProgramm



Die Service-Stufe einer Mitgliedschaft-Instanz muss einem jedem zugeordneten Bonusprogramm bekannt sein, für das die Invariante gilt

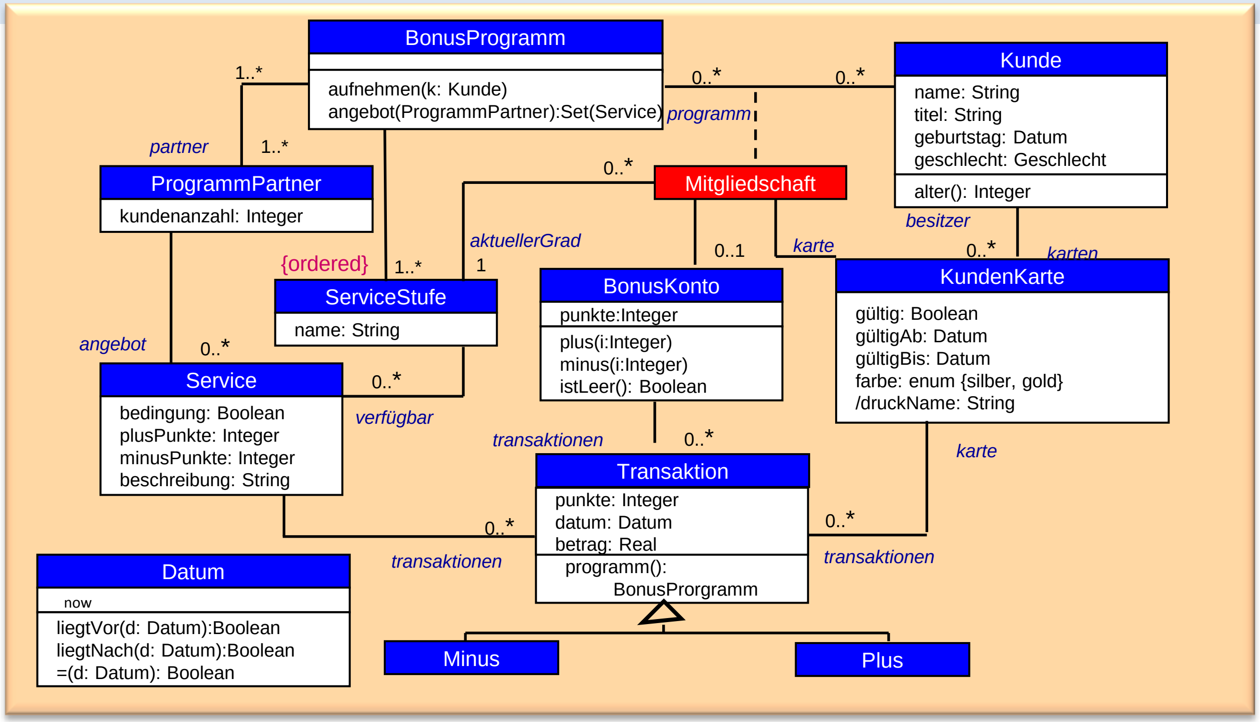
context BonusProgramm

inv bekannteServiceStufe: serviceStufe->includesAll(mitgliedschaft.aktuellerGrad)

orderedSet (ServiceStufe)

ServiceStufe einer Mitgliedschaft

Mitgliedschaft



Kundenkarte-Objekt einer jeden Mitgliedschaft ist enthalten in der Menge der Karten des Kunden, der mit der Mitgliedschaft gemeint ist

Ist-Teilmenge-von

context Mitgliedschaft

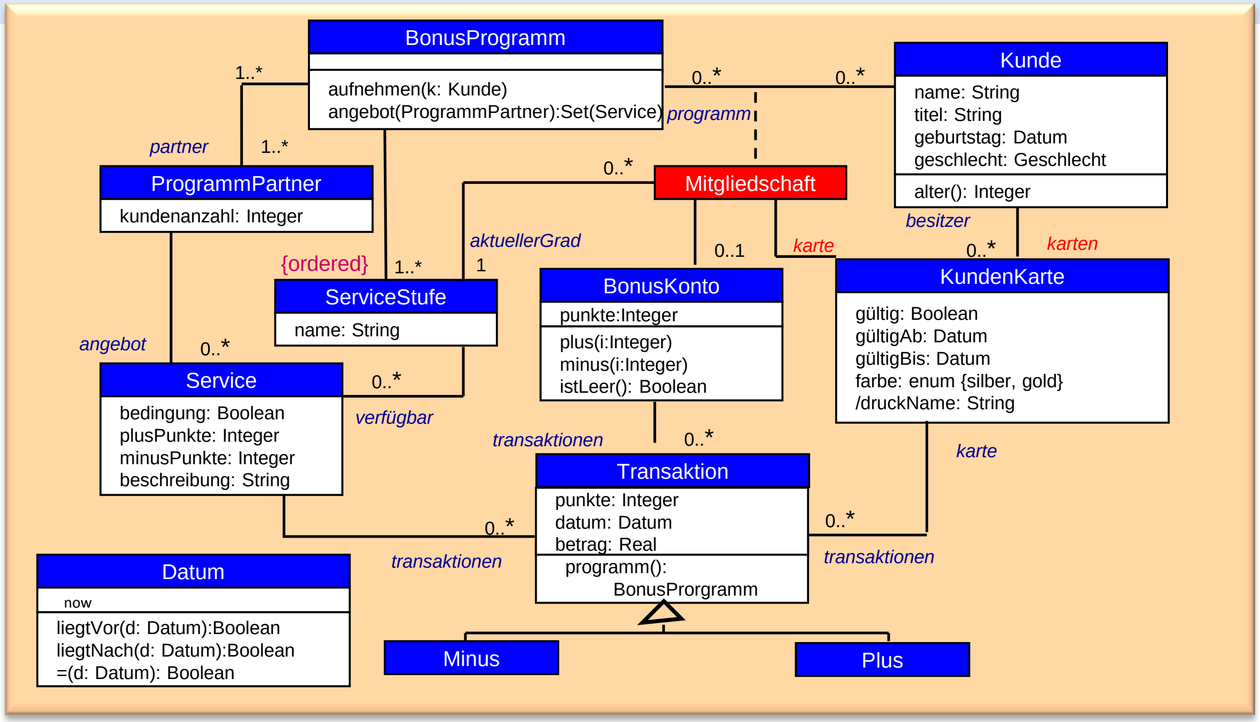
inv korrekteKarte: self.kunde.karten->includesAll (self.karte)

Collection(Kundenkarte) des Kunden

Kundenkarte-Objekt einer Mitgliedschaft

Problem kann leer sein

Mitgliedschaft (2)



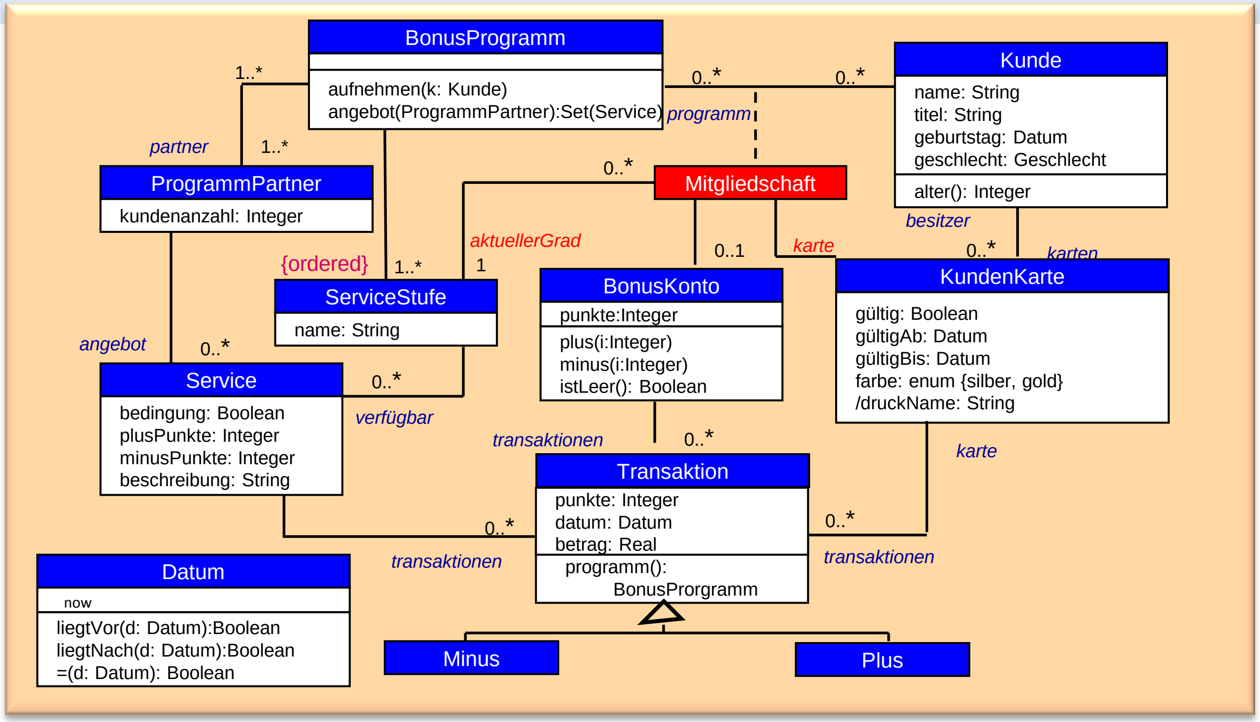
context Mitgliedschaft
inv korrekteKarte: self.kunde.karten->includesAll(self.karte)

Problematisch ?
Nein!

Test auf Empty ist nicht notwendig

context Mitgliedschaft
inv korrekteKarte: self.kunde.karten->notEmpty() **implies**
self.kunde.karten->includesAll(self.karte)

Mitgliedschaft

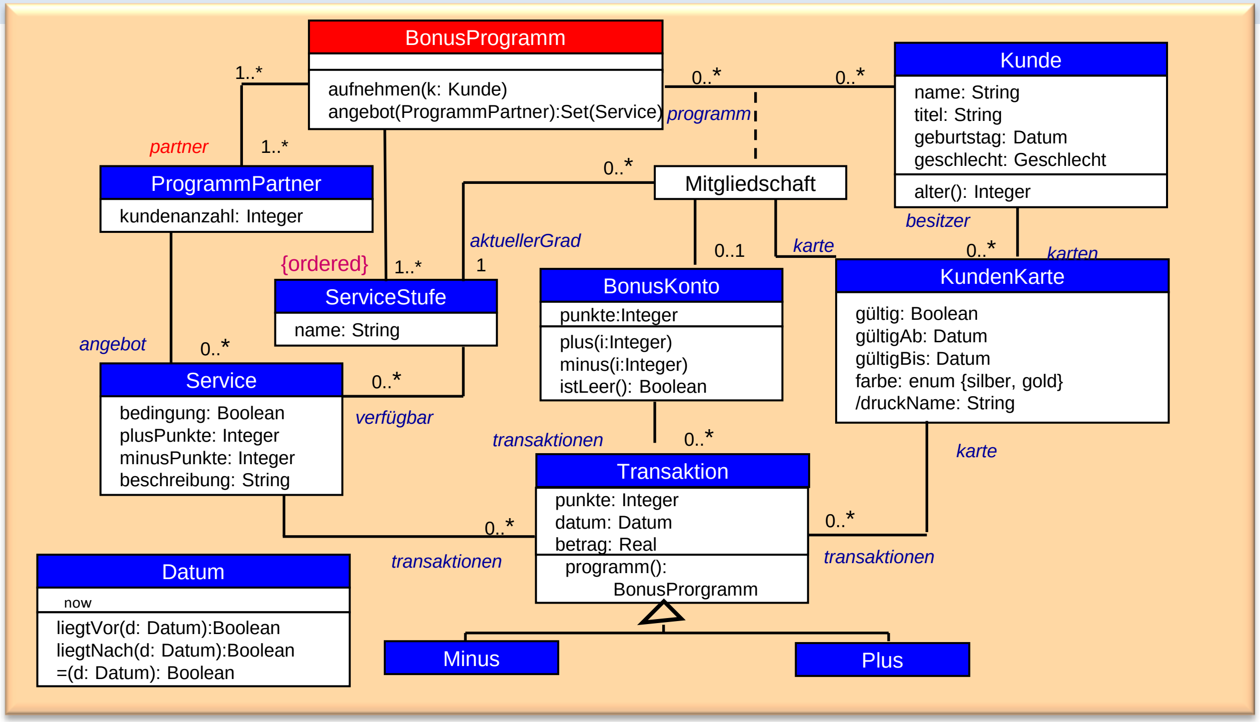


Sevicestufe der Mitgliedschaft korrespondiert zur Farbe der Mitgliedskarte

context Mitgliedschaft

inv: stufeUndFarbe: aktuellerGrad.name= ‘silber‘ **implies** karte.farbe= Colour::silber
and
aktuellerGrad.name= ‘gold‘ **implies** karte.farbe= Colour::gold

Bonusprogramm: size() und collect()



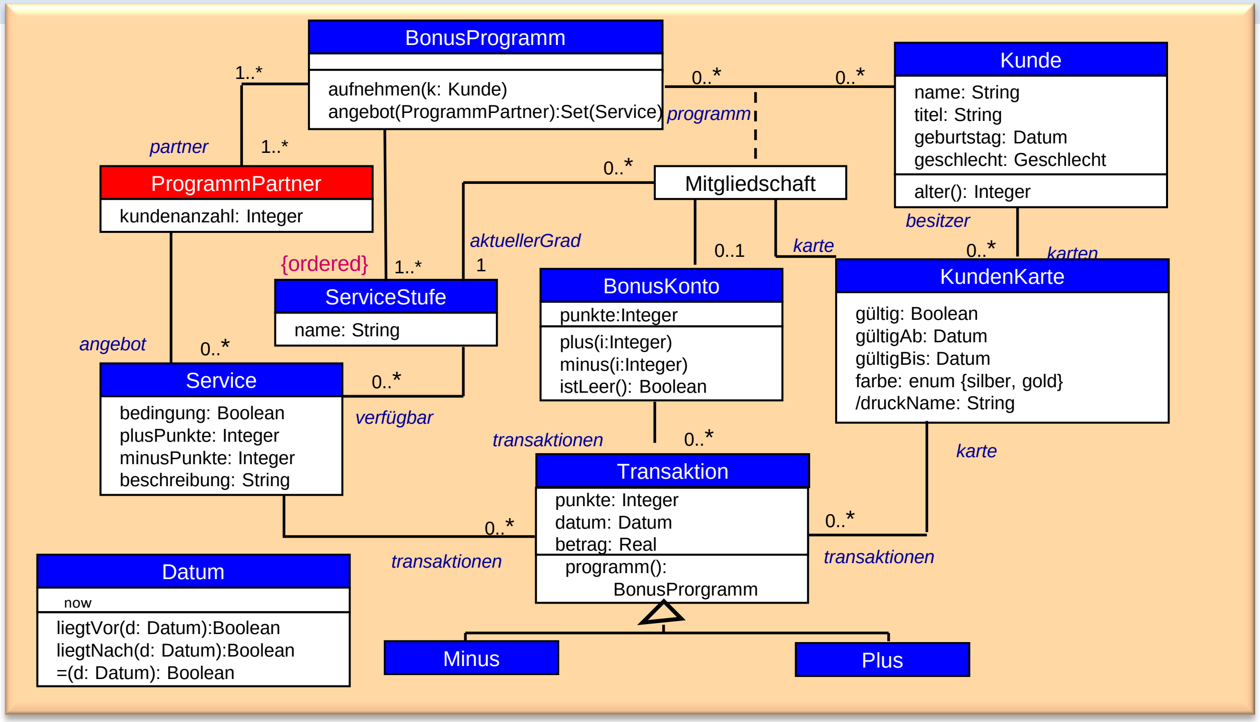
context BonusProgramm

inv self.partner.angebot->size() >=1

$\underbrace{\text{self.partner.angebot}}_{\text{Set(Service)}}$

self.partner->collect (angebot)

Programmpartner



Anzahl von Kunden eines Programmpartners

context ProgrammPartner

inv teilnehmerAnzahl: kundenanzahl= bonusProgramm.kunde->size()

Bag(Kunde)

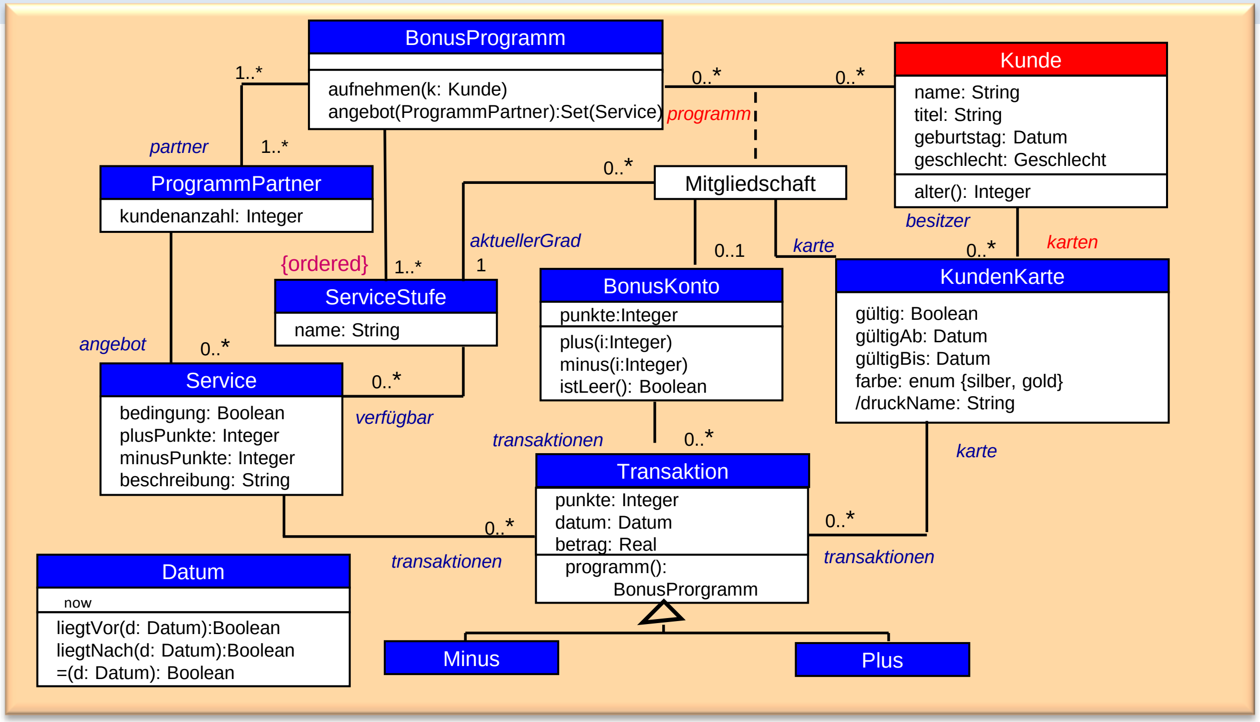
Korrektur

context ProgrammPartner

inv teilnehmerAnzahl: kundenanzahl= bonusProgramm.kunde->asSet()->size()

Set(Kunde)

Kunde: select()



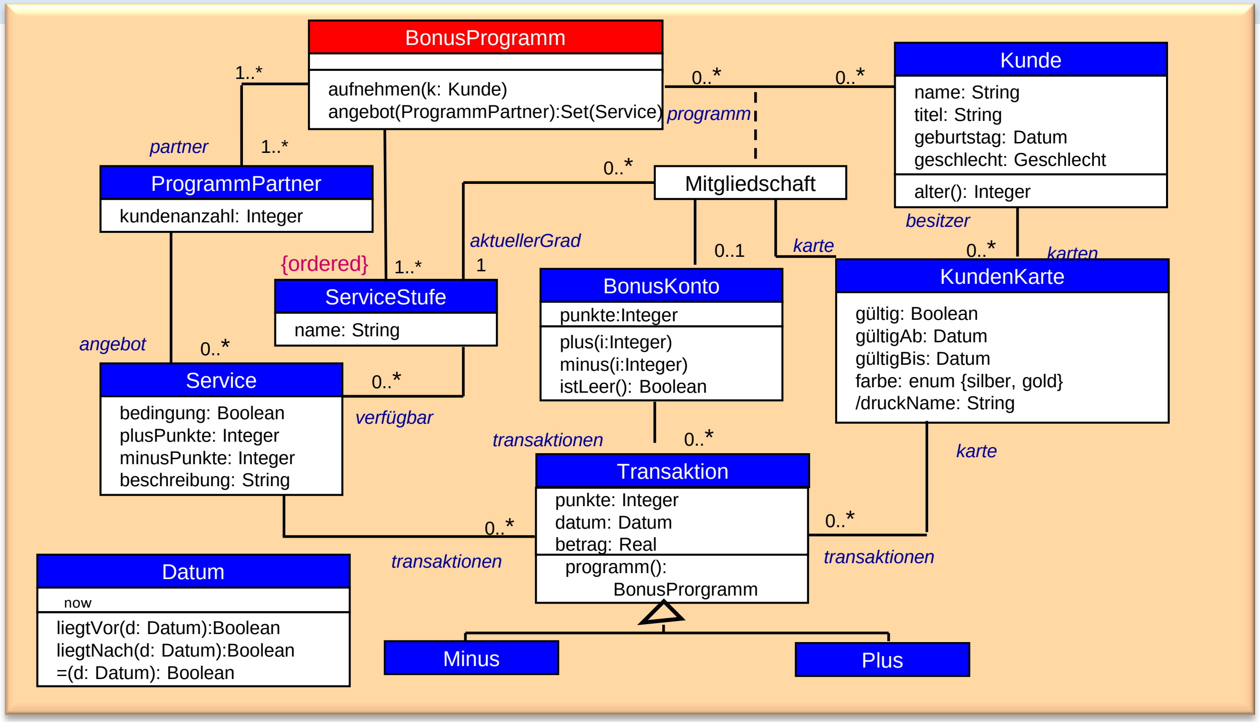
Anzahl gültiger Karten eines Kunden ist gleich der Zahl von BonusProgrammen, an denen der Kunde teilnimmt

context Kunde

inv Anzahl Vereinbahrung:

programm->size() = karten->select (gültig=true)->size()

Bonusprogramm: forAll(), isEmpty()



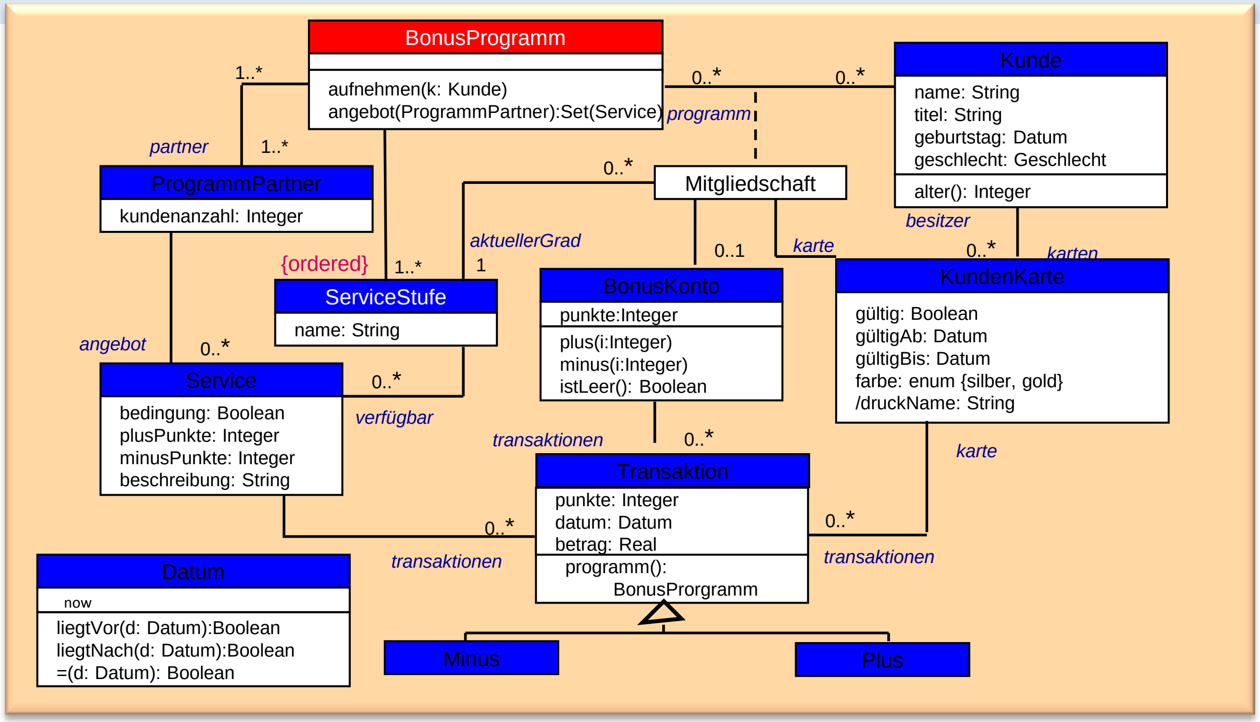
Sollte ein BonusProgramm nicht die Möglichkeit bieten, über einen der angebotenen Dienste weder Punkte zu sammeln noch diese einzusetzen, verfügen die Teilnehmer über kein Konto

context BonusProgramm
inv Kontenanzahl:
    partner.angebot->forAll (plusPunkte=0 **and** MinusPunkte=0)
    implies Mitgliedschaft.bonusKonto->isEmpty()

OP-Parameter

bonusKonto wird implizit als Menge, Set(BonusKonto), betrachtet

Kollektionsoperationen: OrderedSet



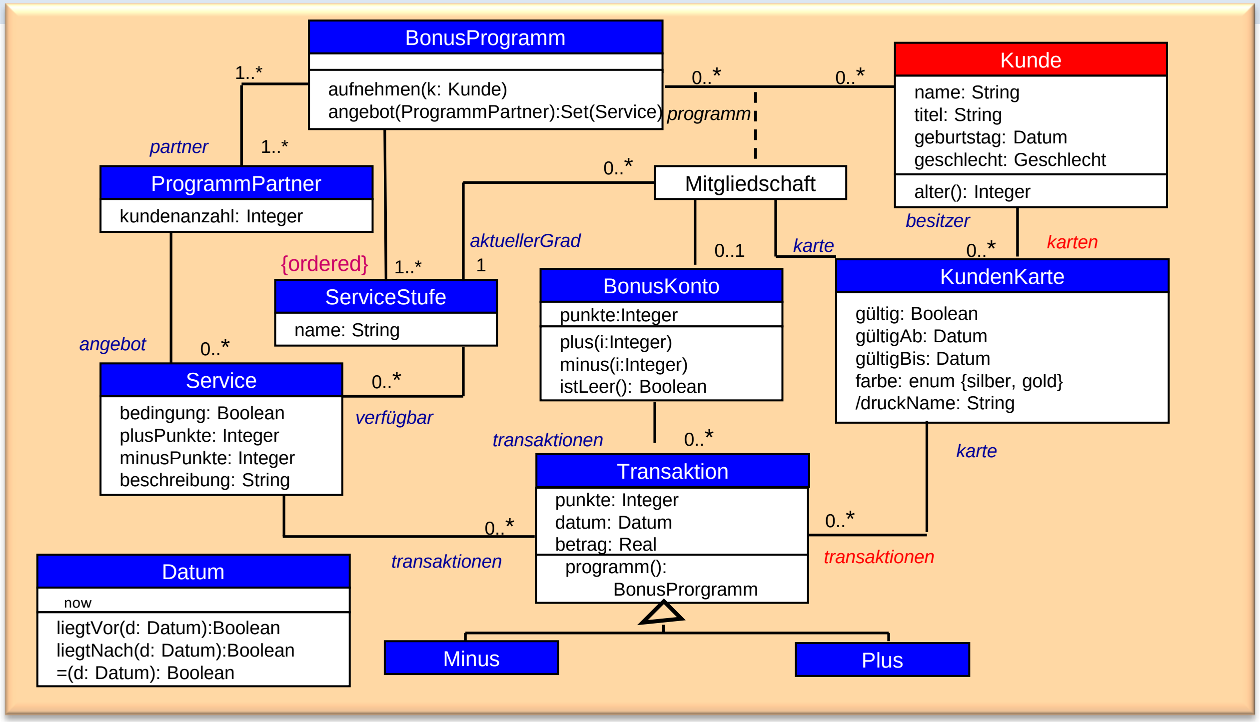
context BonusProgramm

inv ersteStufe:

serviceStufe->first ().name = ‘silber‘

OrderedSet(ServiceStufe)

Kunde: select, collect, sum



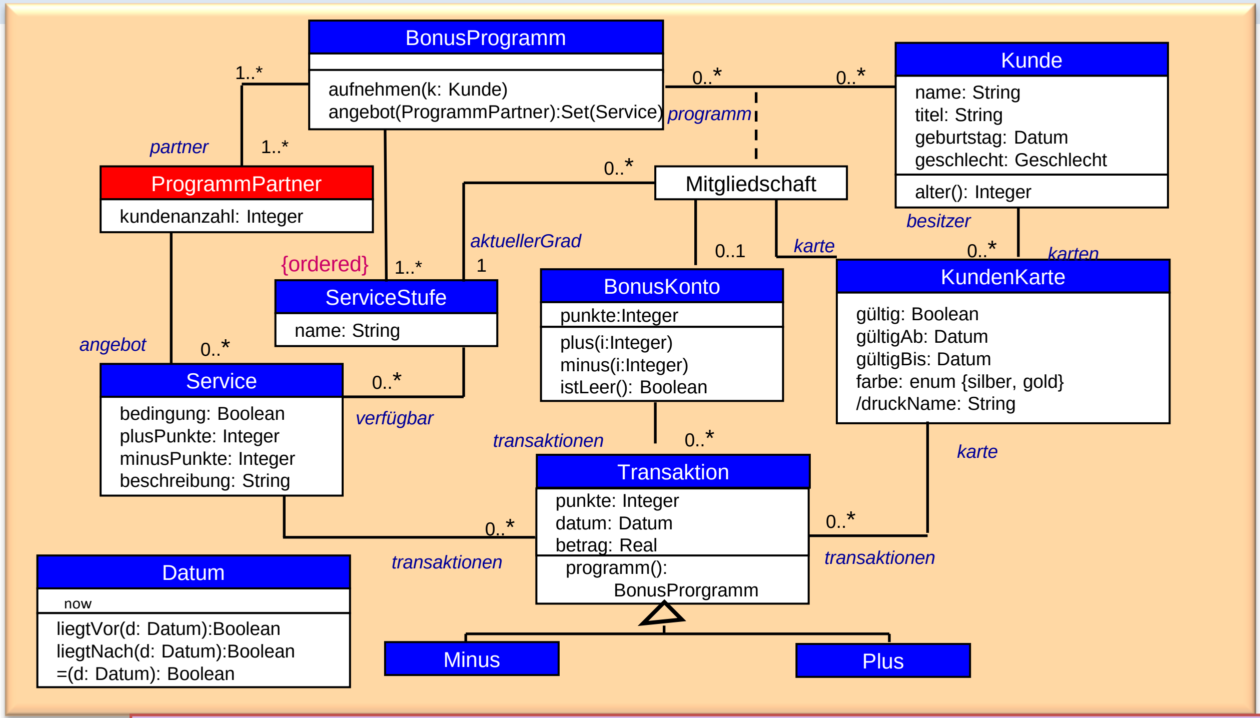
gutVerwendeteKarten
wird Attribut
von Bonuskonto

Menge von Kundenkarten, mit denen Transaktionen durchgeführt worden sind, die in der Summe mehr als 10.000 Punkte brachten

```
context Kunde
def:  gutVerwendeteKarten: Set(KundenKarte)
      = karten->select (transaktionen.punkte->sum() > 10.000)
```

Teilmenge von karten,
für jede der Elemente von karten gilt:
- bilden Menge von Integer-Werte per collect von transaktionen
- bestimmen die Summe
- formulieren das Constraint

ProgrammPartner



context ProgrammPartner **inv**:

self.angebote->collect(transaktionen)->select (oclIsTypeOf(Minus))->collect(punkte)->sum()

<=

self.angebote->collect(transaktionen)->select (oclIsTypeOf(Plus))->collect(punkte)->sum()

context ProgrammPartner **inv**:

self.angebote.transaktionen->select (oclIsTypeOf(Minus)).punkte->sum()

<=

self.angebote.transaktionen->select (oclIsTypeOf(Plus)).punkte->sum()