Kapitel 3: OO Grob-Design by Contract mit **OCL**

*3.1 Beispiele zur **O**bject **C**onstraint **L**anguage*

*3.2 Zur Definition der **OCL***

*3.3 Animation mit **USE***

*3.4 Fortgeschrittene **USE**-Verwendung*

3.5 Zusammenfassung und Bewertung

> Ein Bild sagt mehr als tausend Worte.
>
> *Alte Volksweisheit*

3.1 Beispiele zur OCL (1)

*Die Object Constraint Language ist seit der Version **UML 1.1** fester Bestandteil der Unified Modeling Language der **OMG**.*

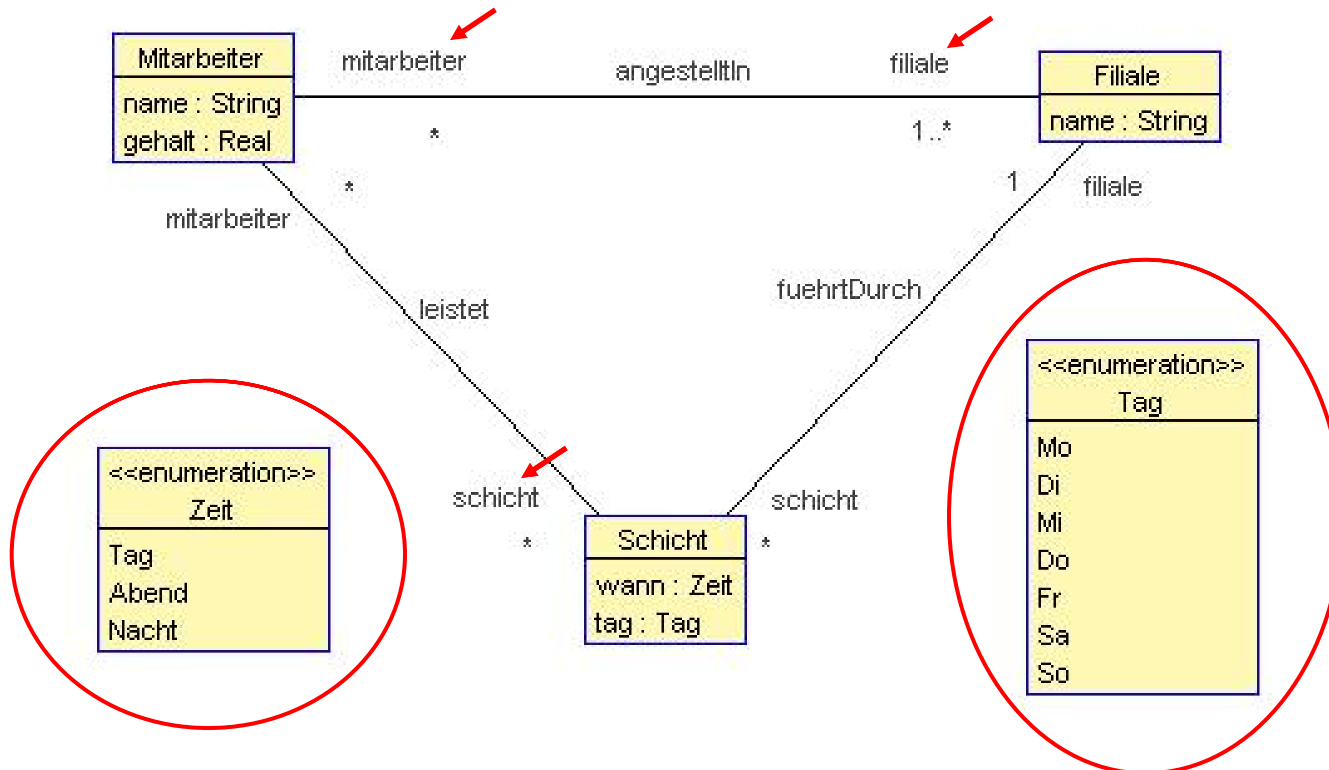
*OCL verhält sich zu **UML** etwa so wie JML zu **Java**: Die beiden Spezifikationssprachen unterstützen **Design by Contract**, d.h. die Beschreibung der Semantik von OO-Modellen durch **Invarianten** der Klassen sowie durch **Vor**- und **Nachbedingungen** der Methoden.*

*Es gibt aber auch wesentliche **Unterschiede**:*

- *während in JML einzelne Klassen beschrieben werden, sind es bei OCL UML-Klassen im **Kontext eines Klassendiagramms**;*
- *während sich die Semantik von JML auf die von Java abstützt, wird die **Semantik** von OCL zur Definition der UML-Semantik verwendet;*
- *OCL ist deutlich **abstrakter** (also weniger detailbehaftet) als JML .*

3.1 Beispiele zur OCL (2)

*Als **Beispiel** betrachten wir die **Fast-Food-Kette McBurger**, wo **Mitarbeiter** im Drei-**Schichten**-Betrieb in verschiedenen **Filialen** arbeiten; die Einplanung erfolgt wöchentlich.*



3.1 Beispiele zur OCL (3)

*Anforderungen an das **McBurger-System**, die sich nicht einem Klassendiagramm beschreiben, aber durch OCL-**Invarianten** ausdrücken lassen:*

(1) Eine Filiale muss mindestens soviele Mitarbeiter haben wie sie Schichten durchführt.

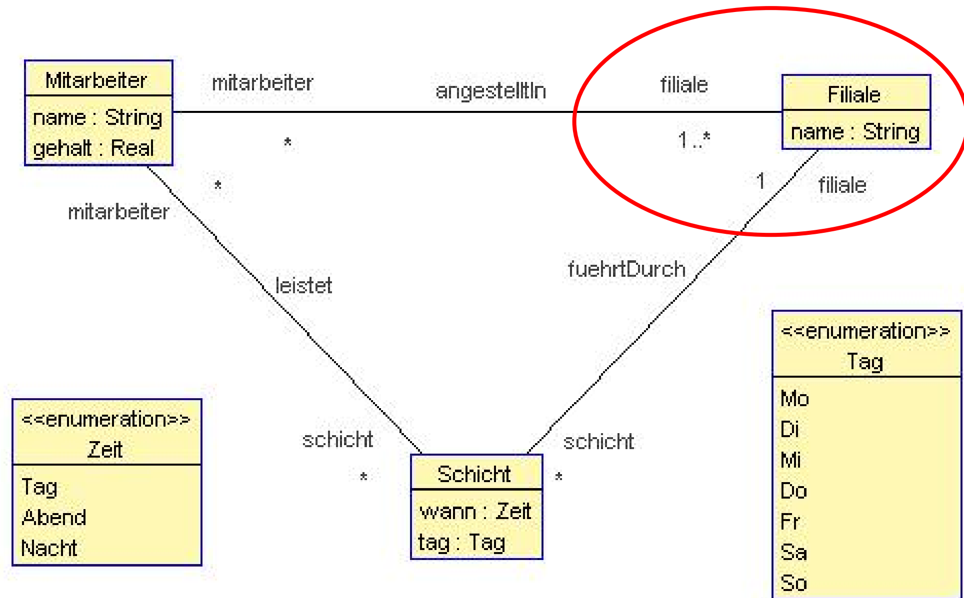
(2) Die Mitarbeiter einer Schicht müssen in der Filiale angestellt sein, welche die Schicht durchführt.

(3) Wer mehr Schichten arbeitet, soll auch mehr verdienen.

(4) Jede Filiale ist mindestens tagsüber geöffnet.

*Wir formalisieren nun diese Anforderungen in **OCL**.*

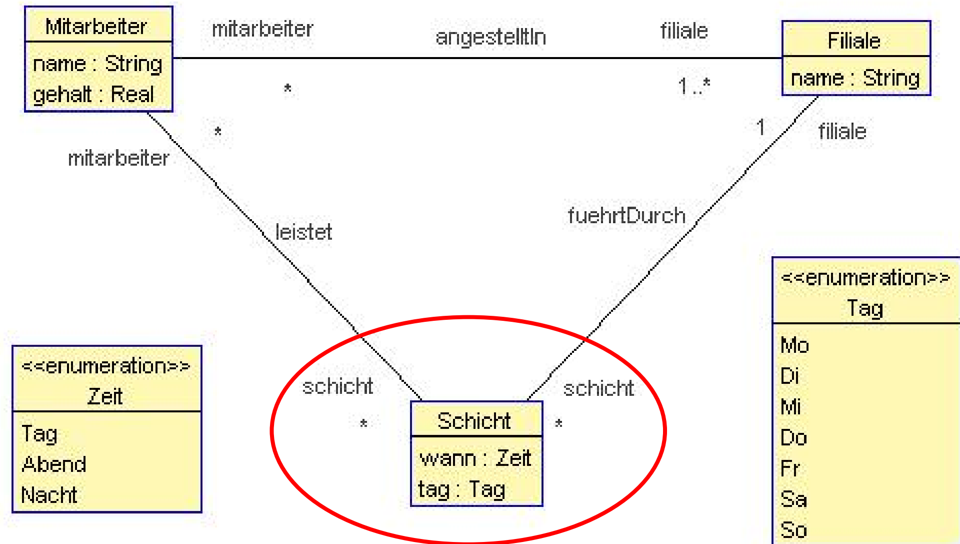
3.1 Beispiele zur OCL (4)



(1) *Eine Filiale muss mindestens soviele Mitarbeiter haben wie sie Schichten durchführt.*

```
context Filiale
  inv GenugMitarbeiter :
    self.mitarbeiter->size >= self.schicht->size
```

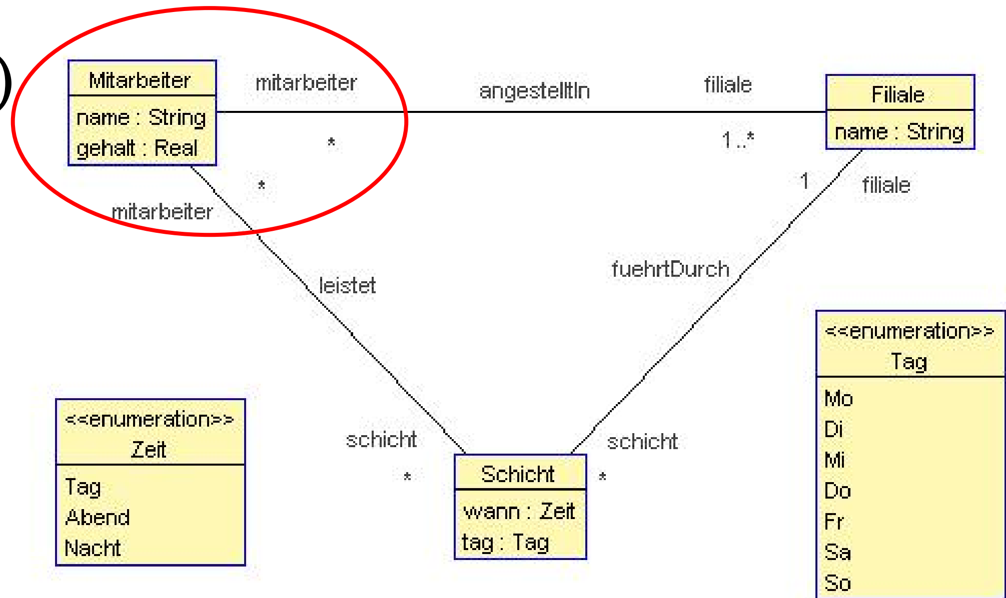
3.1 Beispiele zur OCL (5)



(2) Die Mitarbeiter einer Schicht müssen in der Filiale angestellt sein, welche die Schicht durchführt.

```
context Schicht
  inv SchichtmitarbeiterInFiliale :
    self.filiale.mitarbeiter->includesAll(self.mitarbeiter)
```

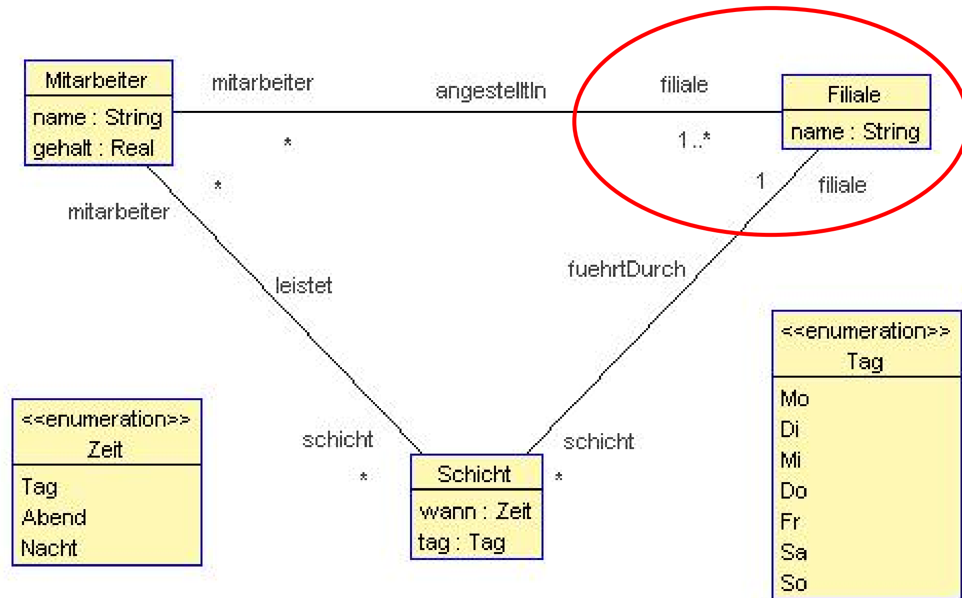
3.1 Beispiele zur OCL (6)



(3) *Wer mehr Schichten arbeitet, soll auch mehr verdienen.*

```
context Mitarbeiter
  inv GerechtesGehalt :
    Mitarbeiter.allInstances->forAll(m1, m2 |
      m1.schicht->size > m2.schicht->size
        implies m1.gehalt > m2.gehalt)
```

3.1 Beispiele zur OCL (7)



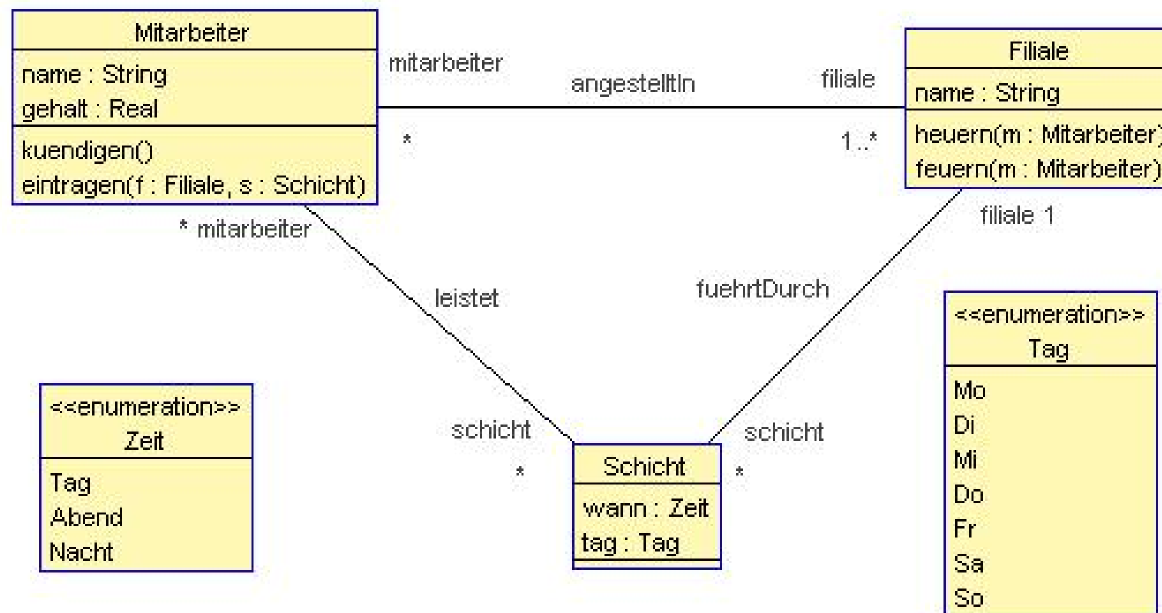
(4) *Jede Filiale ist mindestens tagsüber geöffnet.*

```
context Filiale
  inv TagsOffen :
    Filiale.allInstances->forAll( f |
      f.schicht->select( s | s.wann = #Tag )->size = 7)
```

3.1 Beispiele zur OCL (8)

*Wir ergänzen ein paar **Methoden**:*

- ***Filialen** können Mitarbeiter **heuern** und **feuern**;*
- ***Mitarbeiter** können sich für bestimmte Schichten **eintragen** und sie können **kündigen**.*



3.1 Beispiele zur OCL (9)

*Dem Klassendiagramm kann man die **Signatur** (also die **Syntax** von Methodenaufrufen) entnehmen, nicht aber die **Semantik**. Wir spezifizieren die Methoden der Klasse* `Filiale` *im Sinne des Design by Contract durch **Vor-** und **Nachbedingungen**:*

```
context Filiale::heuern(m : Mitarbeiter)
  pre  heuernVor  : mitarbeiter->excludes(m)
  post heuernNach : mitarbeiter->includes(m)

context Filiale::feuern(m : Mitarbeiter)
  pre  feuernVor  : mitarbeiter->includes(m)
  post feuernNach : mitarbeiter->excludes(m)
```

3.1 Beispiele zur OCL (10)

Analog spezifizieren wir die Methoden der Klasse **Mitarbeiter***:*

```
context Mitarbeiter::eintragen(f : Filiale, s : Schicht)
  pre  eintragenVor  : f.mitarbeiter->includes(self) and
                       s.mitarbeiter->excludes(self)
  post eintragenNach : s.mitarbeiter->includes(self)

context Mitarbeiter::kuendigen()
  pre  kuendigenVor  : self.filiale->notEmpty()
  post kuendigenNach : Filiale.allInstances().mitarbeiter
                       ->excludes(self@pre) and
                       Schicht.allInstances().mitarbeiter
                       ->excludes(self@pre)
```

3.2 Zur Definition der OCL (1)

*Die OCL ist eine **getypte** OO Spezifikationssprache.*

*Die Typen von OCL kann man einerseits unterscheiden nach (unveränderlichen) **Werttypen** und (veränderlichen) **Objekttypen**; andererseits lassen sie sich aufteilen in **vordefinierte** und in **benutzerdefinierte** Typen. Die vordefinierten Typen zerfallen weiter in Basistypen und Kollektionstypen, benutzerdefinierte in **Enumerationstypen** und **Objekttypen**.*

*Die **Basistypen** sind* `Boolean`, `Integer`, `Real` *und* `String`.

*Die **Kollektionstypen** sind* `Set`, `Bag` *und* `Sequence`.

*Wie betrachten zunächst die **Standardoperationen** der vordefinierten Typen.*

3.2 Zur Definition der OCL (2)

*Die **Standardoperationen** des Typs* `Boolean` *sind*

`and or xor not = <>`

*Boolesche Ausdrücke werden **von links nach rechts** ausgewertet, was bei **undefinierten Werten (⊥)** eine Rolle spielt; daher z.B.*

`(false and ⊥) = false` *und* `(true or ⊥) = true`

Daneben gibt es den `implies`*-Operator:*

`a implies b` *ist gleichbedeutend mit* `not a or b`

`false implies ⊥` *ergibt daher* `true`

Und den `if`*-Operator:*

`if t then b1 else b2 endif`

`t` *ist ein Wahrheitswert, der Ergebnistyp ist der von* `b1` *bzw.* `b2`*.*

3.2 Zur Definition der OCL (3)

*Die **Standardoperationen** der Typen* `Integer` *und* `Real` *sind*

`= <> < <= > >= + - *`

Der Operator / ergibt stets ein Ergebnis vom Typ `Real`*:*

`1/2 = 0.5` *und* `2.0/0.5 = 4.0`

Die restlichen Operatoren werden in Methodenaufruf-Notation geschrieben:

- `a.abs`, `a.min(b)` *und* `a.max(b)` *werden auf Operanden gleichen Typs* `T` *angewandt und ergeben wieder einen Wert vom Typ* `T`*.*
- `div` *und* `mod` *auf* `Integer` *angewandt ergeben* `Integer`.
- `round` *und* `floor` ***konvertieren*** `Real` *in* `Integer`.

3.2 Zur Definition der OCL (4)

Der Typ `String` *hat neben den Vergleichsoperatoren*

= <>

typische `String`*-Operatoren, die in Methodenaufruf-Notation geschrieben werden (an Beispielen):*

```
'Lo'.concat('thar')         = 'Lothar'
'Lothar Schmitz'.size       = 14
'Lothar'.substring(4,5)     = 'ha'
'Lothar'.toUpper            = 'LOTHAR'
'Lothar'.toLower            = 'lothar'
```

Weitere `String`*-Operatoren sind nicht vordefiniert.*

3.2 Zur Definition der OCL (5)

*Die **Enumerationstypen** aus dem vorigen Abschnitt wären wie folgt zu definieren:*

```
enum Zeit {Tag, Abend, Nacht}

enum Tag {Mo, Di, Mi, Do, Fr, Sa, So}
```

*Die **Werte** eines **Enumerationstyps** werden aus syntaktischen Gründen mit einem vorgestellten* `#`*-Symbol geschrieben, z.B. in:*

```
(geschlecht = #weiblich) implies (anrede = 'Frau')
```

*Die **Kollektionstypen** sind implizit (!) generisch getypt und haben neben **Konstruktoren** der Form* `Set{1,2,1}`, `Bag{1,2,1}` *und* `Sequence{1,2,1}` *i.W. gleichlautende - jeweils zum Typ passend interpretierte - **Standardoperationen** (☞ nächste Folie).*

3.2 Zur Definition der OCL (6)

***Reine Abfragen** sind:*

```
=, size, isEmpty, notEmpty
count(x), includes(x), includesAll(coll), excludes(x)
```

***Neue Kollektions-Werte erzeugen**:*

```
union(coll), including(x), excluding(x)
```

*Viele Operatoren sind über folgenden **Iterator** definiert:*

```
iterate(<var>;<startval>|<expression>)
```

Dazu gehören (angelehnt an Smalltalk-Konstrukte):

```
sum() (bei Zahlen-Colls)      collect(<var>|<expression>)
exists(<var>|<predicate>)     select(<var>|<predicate>)
forAll(<var>|<predicate>)     reject(<var>|<predicate>)
```

3.2 Zur Definition der OCL (7)

Daneben gibt es Operatoren, die nicht für alle Kollektionstypen definiert sind:

```
intersection(coll)
```

*ist definiert **für Sets und Bags**, aber nicht für Sequences.*

*Nur **für Sequences** sind definiert:*

```
first, last, at(i), append(x), prepend(x)
```

wobei `prepend(x)` *dem **cons-Operator (:)** von Haskell entspricht.*

*Nur **für Sets** sind Mengendifferenzen definiert:*

```
Set{2,7,5} - Set{1,5,11,2} = Set{7}

S1.symmetricDifference(S2) = (S1-S2).union(S2-S1)
Set{2,7,5}.symmetricDifference(Set{1,5,11,2}) = ??
```

3.2 Zur Definition der OCL (8)

*Wie in Abschnitt 3.1 am Beispiel gezeigt, kann man in **OCL** mit **Pfadausdrücken** ausgehend von der als **Kontext** gewählten Klasse das Klassendiagramm entlang Assoziationen durchqueren.*

*OCL unterscheidet bei **Methodenaufrufen** zwischen Kollektionen und anderen Typen: Bei Kollektionen geht dem Methodennamen ein Pfeil (`->`) voraus, sonst wie in Java ein Punkt (`.`).*

*Die **Pseudovariable** `self` entspricht in OCL (wie in Smalltalk) der Pseudovariablen `this` von Java (`me`, `current`, ... anderswo).*

*In **Nachbedingungen** bezeichnet `<expr>@pre` den Wert des Ausdrucks `<expr>` vor dem Aufruf der Methode. Das ist bewusst asymmetrisch: **Vorbedingungen** können sich **nicht** auf Werte beziehen, die nach Abarbeitung des Aufrufs vorliegen.*

3.2 Zur Definition der OCL (9)

*Die OCL-**Objekttypen** bilden eine Hierarchie, an deren Spitze der Typ* `OclAny` *steht (also vergleichbar zu* `Object` *in Java).*

Die in `OclAny` *definierten Operationen stehen daher in allen OCL-Klassen zur Verfügung. Dazu gehören die **Gleichheitsoperatoren*** `=` *und* `<>`*, wobei* `=` *dem* `equals()` *von Java entspricht.*

*Daneben gibt es hier eine Reihe von Operationen, mit denen man auf der **Meta-Ebene** auf die **Struktur des Modells** zugreifen kann:*

`o.oclType` *(ergibt Klasse)* `o.oclAsType(T)` *(Cast)*

`o.oclIsTypeOf(T)` *(liegt in* `T`*)*

`o.oclIsKindOf(T)` *(liegt in* `T` *oder Unterklasse von* `T`*)*

`T.allInstances` *(ergibt Menge der Objekte der Klasse T)*

`T.name/attributes/supertypes(T)` *(Infos)* ***u.v.m.***

3.2 Zur Definition der OCL (10)

Wie anfangs erwähnt, dient OCL nicht nur zur Beschreibung von Anwendungsmodellen (wie in Abschnitt 3.1), sondern zur auf der Meta-Ebene zur ***Definition der Semantik von UML*** *selbst.*

Dafür braucht man Methoden, mit denen man ***auf die Struktur des Objektsystems zugreifen*** *kann - wie auf der letzten Folie angegeben.*

Außerdem muss die ***Bedeutung der OCL-Elemente*** *selbst genau definiert werden - sonst wären UML-**Diagramme** nur hübsche Graphen aus Kästchen und Linien, aber ohne Semantik!*

Um einen Eindruck von diesen Definitionen zu gewinnen, betrachten wir die Erklärung von `iterate` ***und einigen darauf abgestützten Operationen in den OCL-Dokumenten.***

3.2 Zur Definition der OCL (11)

*Die **Semantik** des **Iterators** wird im Referenz-Buch [J. Warmer, A. Kleppe 1999] wie folgt **operational** über ein Stück **Pseudocode** definiert:*

```
coll.iterate(<var>;<startval>|<combine-expression>)
```

entspricht demnach der Wirkung von

```
result := <startval>;
while coll.notEmpty do
    <var> := coll.nextElement();
    result := <combine-expression>(<var>, result);
endwhile
return result
```

3.2 Zur Definition der OCL (12)

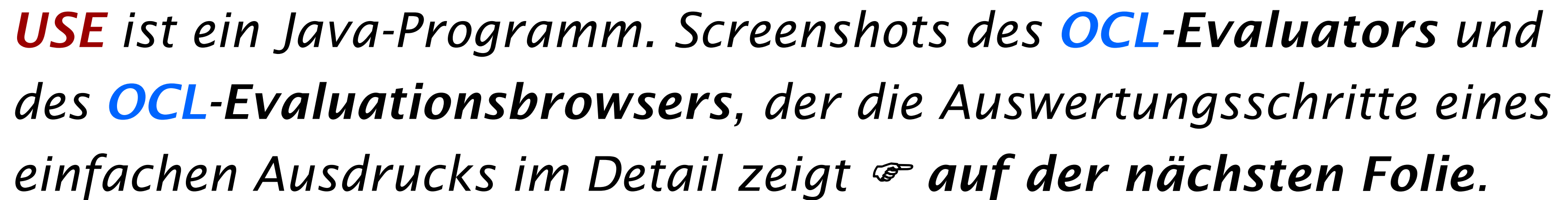
Im Anhang des definierenden OMG-Dokuments [OCL 2.0] wird die Bedeutung von Operationen wie `forAll` *und* `collect` *mit Hilfe des* `iterate`*-Konstrukts wie folgt beschrieben (*$\mathcal{I}$ *ist Interpretation):*

```
I[[ e1→forAll(v1 | e3) ]](r) =
I[[ e1→iterate(v1; v2 = true | v2 and e3) ]](r)

I[[ e1→collect(v1 | e3) ]](r) =
I[[ e1→iterate(v1; v2 = mkBag type-of-e3() |
    v2→including(e3)) ]](r)

I[[ e1→select(v1 | e3) ]](r) =
I[[ e1→iterate(v1; v2 = e1 |
    if e3 then v2 else v2→excluding(v1) endif) ]](r)

I[[ e1→reject(v1 | e3) ]](r) =
I[[ e1→iterate(v1; v2 = e1 |
    if e3 then v2→excluding(v1) else v2 endif) ]](r)
```

3.3 Animation mit **USE** (1)

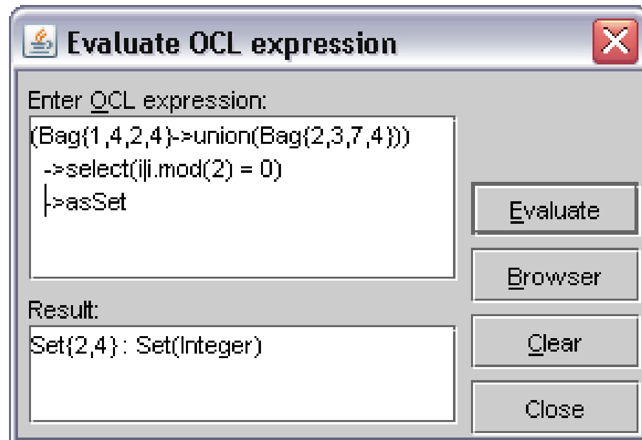
*Das **U****ML**-based **S**pecification **E**nvironment ist ein Werkzeug zur Animation von **OCL**-Spezifikationen. "Animation" bedeutet hier, dass entweder*

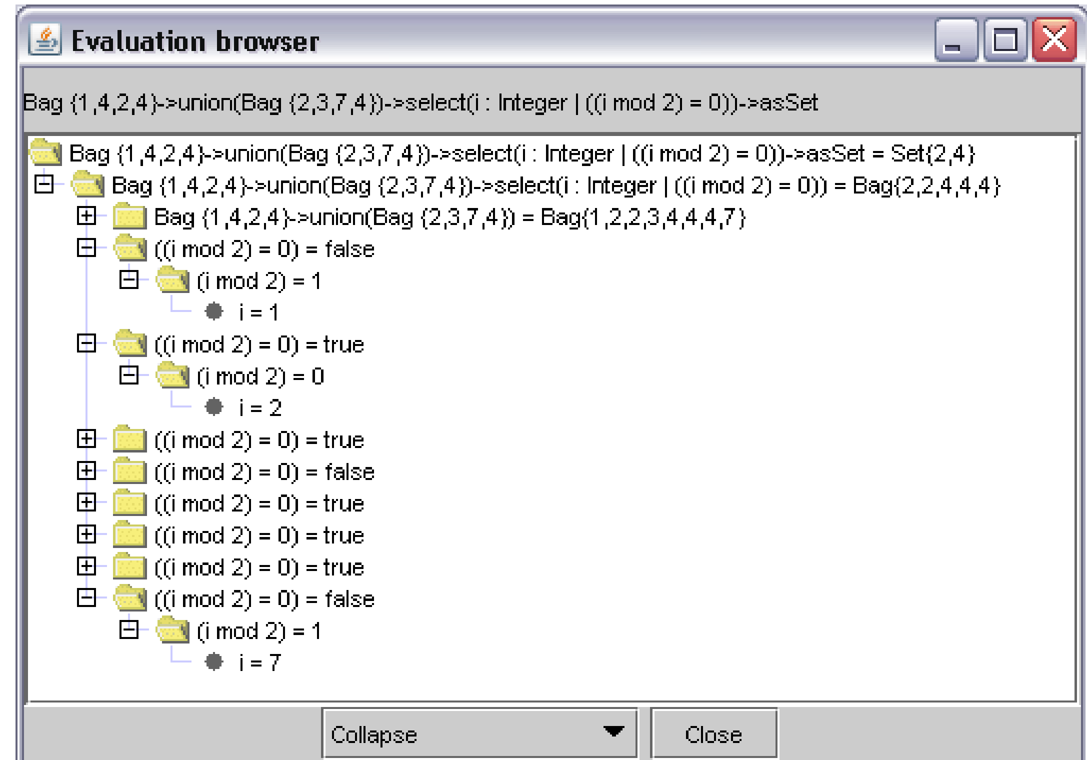
1. *einfache **OCL**-Ausdrücke direkt ausgewertet werden*
2. *oder dass zu einem Klassendiagramm mit zugehörigen **OCL**-Ausdrücken ein **UML-Objektdiagramm** schrittweise aufgebaut wird, an dem man die Einhaltung von Invarianten und Zusicherungen laufend überprüfen kann.*



***USE** ist ein Java-Programm. Screenshots des **OCL-Evaluators** und des **OCL-Evaluationsbrowsers**, der die Auswertungsschritte eines einfachen Ausdrucks im Detail zeigt ☞ **auf der nächsten Folie**.*

3.3 Animation mit **USE** (2)





3.3 Animation mit **USE** (3)

*In einer OCL-Spezifikation, die sich auf ein **UML**-Klassendiagramm bezieht, müssen das **Klassendiagramm** nebst Invarianten und Zusicherungen **textuell** in **USE** eingebracht werden.*

Wir betrachten dazu die Datei `McBurger.use`*, die zum laufenden Beispiel aus Abschnitt 3.1 gehört.*

```
model McBurger

-- globale Typen

enum Zeit {Tag, Abend, Nacht}
enum Tag {Mo, Di, Mi, Do, Fr, Sa, So}

-- weiter auf naechster Folie
```

3.3 Animation mit USE (4)

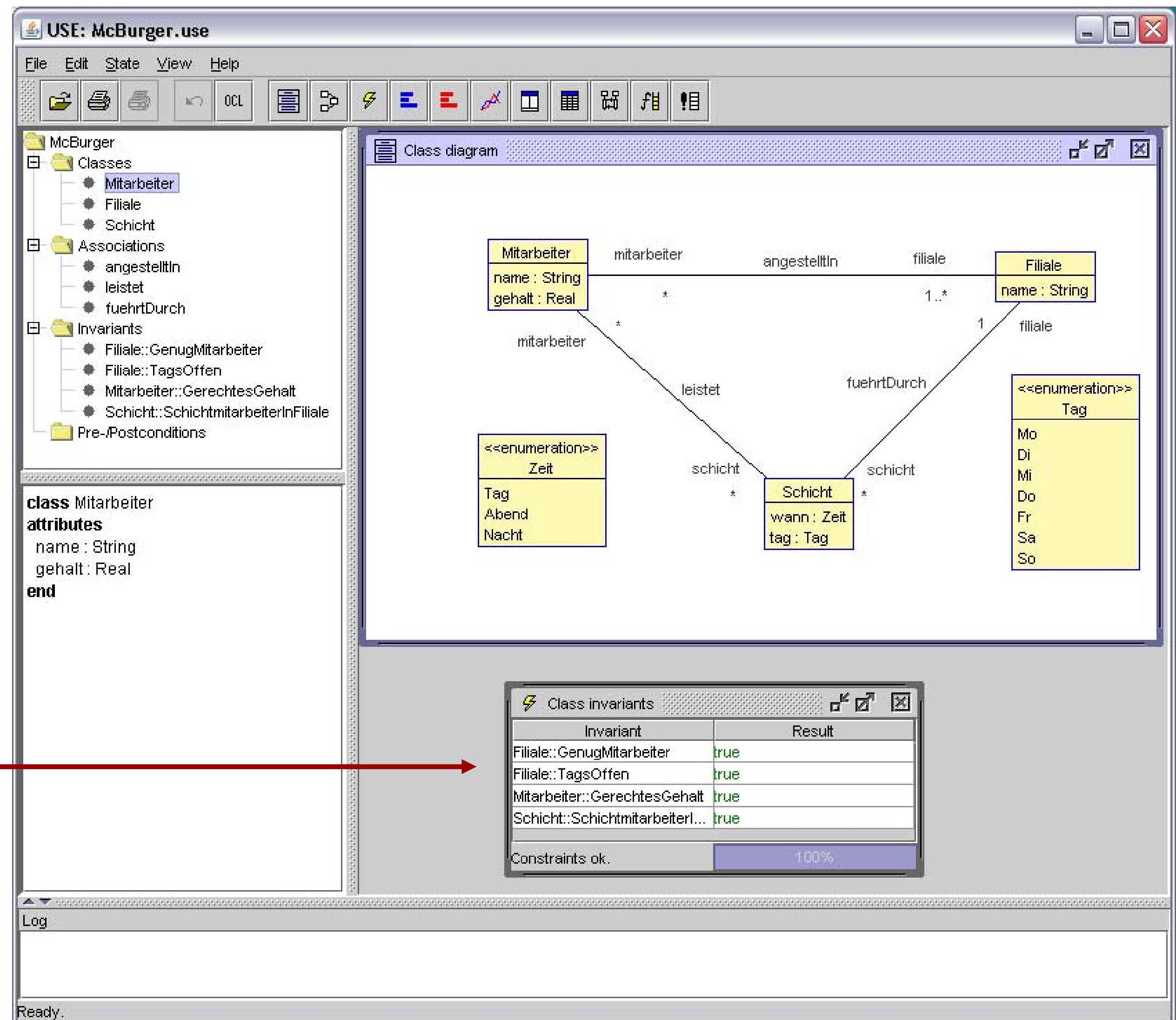
```
-- Klassen

class Mitarbeiter
attributes
  name : String
  gehalt : Real
end

class Filiale
attributes
  name : String
end

class Schicht
attributes
  wann : Zeit
  tag  : Tag
end
```

```
-- Assoziationen

association angestelltIn between
  Mitarbeiter[*]
  Filiale[1..*]
end

association leistet between
  Mitarbeiter[*]
  Schicht[*]
end

association fuehrtDurch between
  Filiale[1]
  Schicht[*]
end

-- danach Invarianten und Zusicherungen
-- genau wie in Abschnitt 3.1 gezeigt
```

3.3 Animation mit **USE** (5)

*Zustand des **USE**-Fensters nach dem Laden der **USE**-Datei.*

*Warum sind im leeren Modell alle **Invarianten** erfüllt?*



3.3 Animation mit **USE** (6)

Start per Doppelklick auf Invariante

*Das wird anhand der **Invarianten-Auswertungen** klar:*

Evaluation browser

context Schicht **inv** SchichtmitarbeiterInFiliale:
self.filiale.mitarbeiter->includesAll(self.mitarbeiter)

Schicht.allInstances->forAll(self : Schicht | self.filiale.mitarbeiter->includesAll(self.mitarbeiter)) = true
Schicht.allInstances = Set{}

Display options | Close

Evaluation browser

context Filiale **inv** GenugMitarbeiter:
(self.mitarbeiter->size >= self.schicht->size)

Filiale.allInstances->forAll(self : Filiale | (self.mitarbeiter->size >= self.schich
Filiale.allInstances = Set{}

Evaluation browser

context Mitarbeiter **inv** GerechtesGehalt:
Mitarbeiter.allInstances->forAll(m1 , m2 : Mitarbeiter | ((m1.schicht->size > m2.schicht->size) implies (m1.gehalt > m2.gehalt)))

Mitarbeiter.allInstances->forAll(self : Mitarbeiter | Mitarbeiter.allInstances->forAll(m1 , m2 : Mitarbeiter | ((m1.schicht->size > m2.schicht->size) implies (m1.gehalt > m2.gehalt)))) = true
Mitarbeiter.allInstances = Set{}

Evaluation browser

context Filiale **inv** TagsOffen:
Filiale.allInstances->forAll(f : Filiale | (f.schicht->select(s : Schicht | (s.wann = #Tag))->size = 7))

Filiale.allInstances->forAll(self : Filiale | Filiale.allInstances->forAll(f : Filiale | (f.schicht->select(s : Schicht | (s.wann = #Tag))->size = 7))) = true
Filiale.allInstances = Set{}

Display options | Close

3.3 Animation mit **USE** (7)

*Wir bauen nun schrittweise ein **Objektdiagramm** auf, ersetzen dazu das Klassendiagramm und fügen ein **Mitarbeiter-Objekt** ein:*

3.3 Animation mit **USE** (8)

*In der Log-Scheibe (unten im Fenster) beklagt **USE** die Verletzung einer Multiplizitätsangabe des unterliegenden Klassendiagramms:*

```
checking structure...
Multiplicity constraint violation in association
`angestelltIn':
  Object `Mitarbeiter1' of class `Mitarbeiter' is
  connected to 0 objects of class `Filiale'
  but the multiplicity is specified as `1..*'.
```

*Um diese **Verletzung aufzuheben**, benötigen wir also ein* **`Filiale`**-*Objekt, mit dem der Mitarbeiter über eine Assoziation der Art* **`angestelltIn`** *verbunden ist.*

3.3 Animation mit **USE** (9)

Als Folge dieser Einfügungen wird nun die Invariante **TagsOffen** *verletzt - klar: eine bestehende Filiale muss 7mal die Woche offen sein! Also brauchen wir 7 Schichten (und 7 Mitarbeiter) nebst Assoziationen ...*

USE: McBurger.use

File Edit State View Help

McBurger
- Classes
  - Mitarbeiter
  - Filiale
  - Schicht
- Associations
  - angestelltIn
  - leistet
  - fuehrtDurch
- Invariants
  - Filiale::GenugMitarbeiter
  - Filiale::TagsOffen
  - Mitarbeiter::GerechtesGehalt
  - Schicht::SchichtmitarbeiterInFiliale
- Pre-/Postconditions

```
association angestelltIn between
  Mitarbeiter[*] role mitarbeiter
  Filiale[1..*] role filiale
end
```

Object diagram

Mitarbeiter1:Mitarbeiter
name='Adam'
gehalt=1111.11

angestelltIn
mitarbeiter
filiale

Filiale1:Filiale
name='Am Hauptbahnhof'

Class invariants

Invariant	Result
Filiale::GenugMitarbeiter	true
Filiale::TagsOffen	false
Mitarbeiter::GerechtesGehalt	true
Schicht::SchichtmitarbeiterI...	true

1 constraint failed. 100%

Log

Ready.

3.3 Animation mit USE (10)

Man sieht: *Die Bedienung über die Oberfläche wird sehr mühsam, wenn viele Objekte erzeugt werden müssen.*

*In **USE** gibt es daher neben der **GUI** ein **Skripting-Interface** - und zwar in der Kommando-Shell, von der aus das Java-Programm **USE** gestartet wurde.*

*Vom **Skripting-Interface** aus können wir*

- *das Modell (UML-Klassendiagramm und Constraints) einsehen;*
- *den aktuellen Systemzustand einsehen;*
- *den aktuellen Systemzustand durch Kommandos verändern;*
- *ganze Dateien mit solchen Kommandos auf den aktuellen Systemzustand anwenden.*

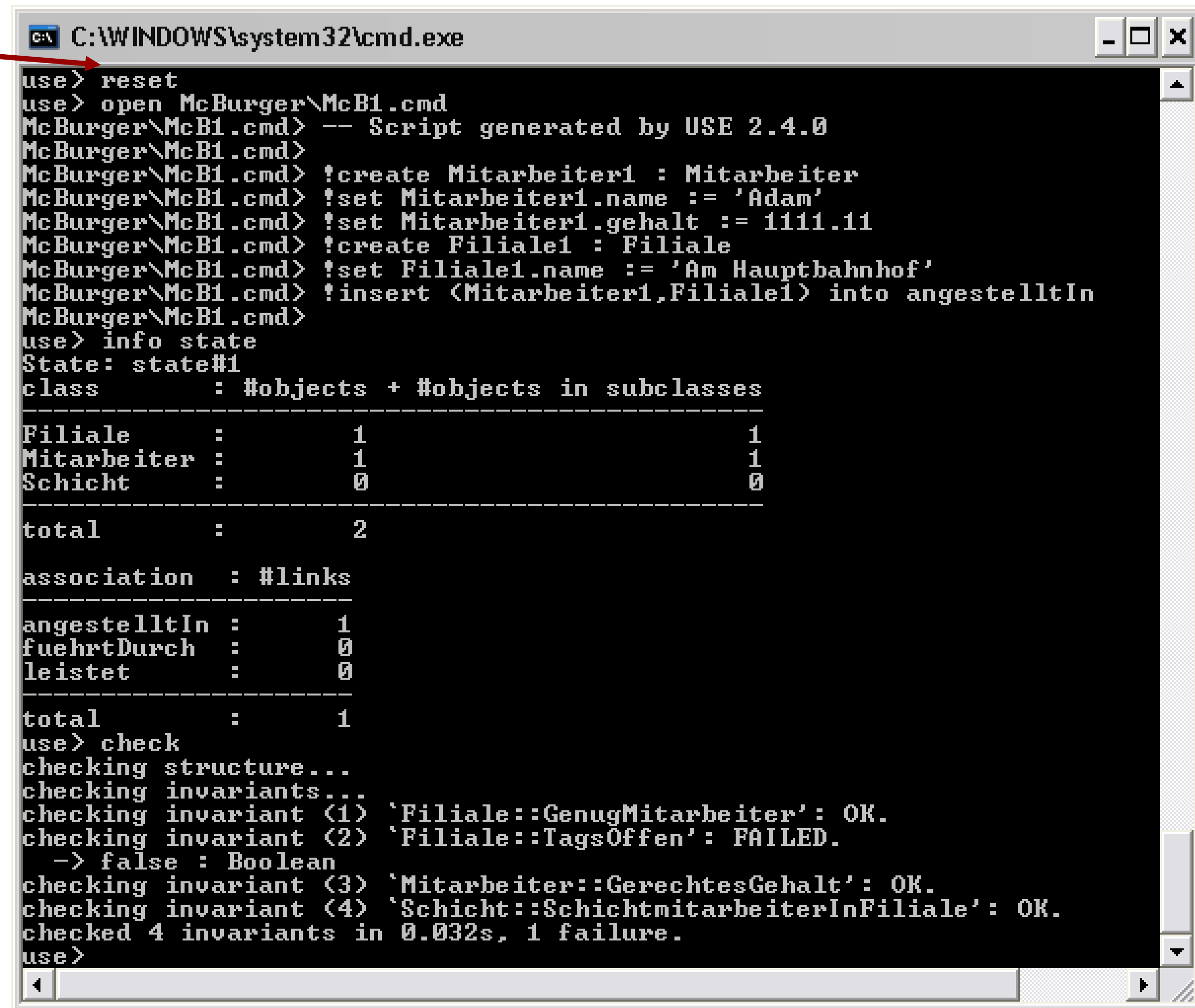
3.3 Animation mit **USE** (11)

Umgekehrt kann man von der ***USE-GUI*** *aus die zum aktuellen Systemzustand führende Folge von Kommandos in eine cmd-Datei ausgeben (um später genau diesen Zustand herstellen zu können):*

```
-- Script generated by USE 2.4.0

!create Mitarbeiter1 : Mitarbeiter
!set Mitarbeiter1.name := 'Adam'
!set Mitarbeiter1.gehalt := 1111.11

!create Filiale1 : Filiale
!set Filiale1.name := 'Am Hauptbahnhof'

!insert (Mitarbeiter1,Filiale1) into angestelltIn
```

3.3 Animation mit USE (12)

Nach einem **`reset`** *zum Normieren des Systemzustands lesen wir die unter* **`McB1.cmd`** *gespeicherte Kommandofolge ein und erhalten wieder den gleichen Systemzustand. Der Screenshot zeigt auch die Kommandos* **`info`** *und* **`check`**.



3.3 Animation mit USE (13)

*Für weitere Experimente starten wir das **USE**-System.*

*Für das Modell **McBurger.use** sind vorbereitet*

- `McB.cmd` *(ähnlich wie oben gezeigt);*
- `McB-auf7.cmd` *führt 6 weitere Schichten und Mitarbeiter mit den zugehörigen Assoziationen ein, so dass nachher alle Invarianten erfüllt sind.*

*Für das Modell **McBurgerOps.use** sind vorbereitet*

- `McBOps.cmd` *stellt eine Ausgangssituation her;*
- `McBOpsKorr.cmd` *bringt die Invarianten in Ordnung;*
- `McBOpsAufrufe.cmd` *enthält "Handsimulationen" zum Aufruf von* `feuern` *(in korrekter Weise) und* `kuendigen` *(nicht korrekt).*

3.4 Fortgeschrittene USE-Verwendung (1)

*Mit **USE** kann man Objektdiagramme wie folgt manipulieren:*

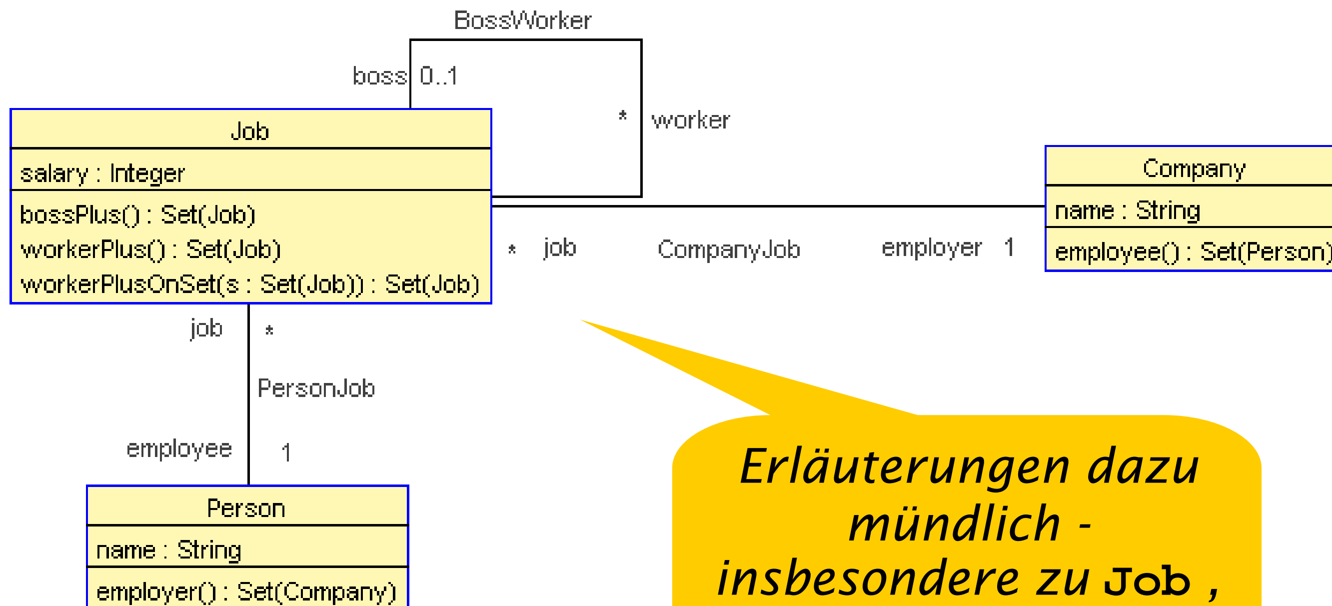
- ***Objekte** anlegen (`!create`) und löschen (`!destroy`)*
- ***Attribute** setzen (`!set`)*
- ***Assoziationslinks** einfügen (`!insert`) und entfernen (`!delete`)*
- ***Methoden** aufrufen (`!openter`) und beenden (`!opexit`)*

*Da dies immer noch wenig komfortabel ist, wurde mit **ASSL** (**A** **S**napshot **S**equence **L**anguage) ein mächtiger **Generator** ergänzt, mit dem man:*

- *Objektdiagramme durch **ASSL-Prozeduren** aufbauen,*
- *dabei **Backtracking** nutzen und*
- ***Invarianten-erhaltend erzeugen** kann.*

3.4 Fortgeschrittene USE-Verwendung (2)

*Das **Standardbeispiel** dieses Abschnitts bezieht sich auf das folgende Klassendiagramm:*



3.4 Fortgeschrittene USE-Verwendung (3)

*Die zugehörige **USE**-Datei definiert u.a. (beachte **Methoden**!!):*

```
class Person
attributes
  name:String
operations
  employer():Set(Company)
   = self.job.employer ->
     asSet()
end
class Company
attributes
  name:String
operations
  employee():Set(Person)
   = self.job.employee ->
     asSet()
end
```

```
class Job
attributes
  salary:Integer
operations
  bossPlus():Set(Job) =
   if boss.isDefined
   then boss.bossPlus()-> including(boss)
   else oclEmpty(Set(Job))
   endif
  workerPlus():Set(Job) = workerPlusOnSet(worker)
  workerPlusOnSet(s:Set(Job)):Set(Job) =
   let oneStep:Set(Job)=s.worker->asSet in
   if oneStep->exists(j|s->excludes(j))
   then workerPlusOnSet(s->union(oneStep))
   else s
   endif
end
```

3.4 Fortgeschrittene USE-Verwendung (4)

Im Modell werden folgende ***Invarianten*** *gefordert:*

```
constraints

context p1:Person inv personNamesAreUnique:
  Person.allInstances->forAll(p2|p1.name=p2.name implies p1=p2)

context c1:Company inv companyNamesAreUnique:
  Company.allInstances->forAll(c2|c1.name=c2.name implies c1=c2)

context j1:Job inv employeeEmployerAreUnique:
  Job.allInstances->forAll(j2|
    j1.employee=j2.employee and j1.employer=j2.employer implies j1=j2)

context top:Job inv bossWorkerSameEmployer:
  top.worker->forAll(low|low.employer=top.employer)

context j:Job inv bossWorkerIsHierarchy:
  j.workerPlus()->excludes(j)

context top:Job inv bossBetterPaidThanWorker:
  top.worker->forAll(low|low.salary<top.salary)
```

Bitte interpretieren!

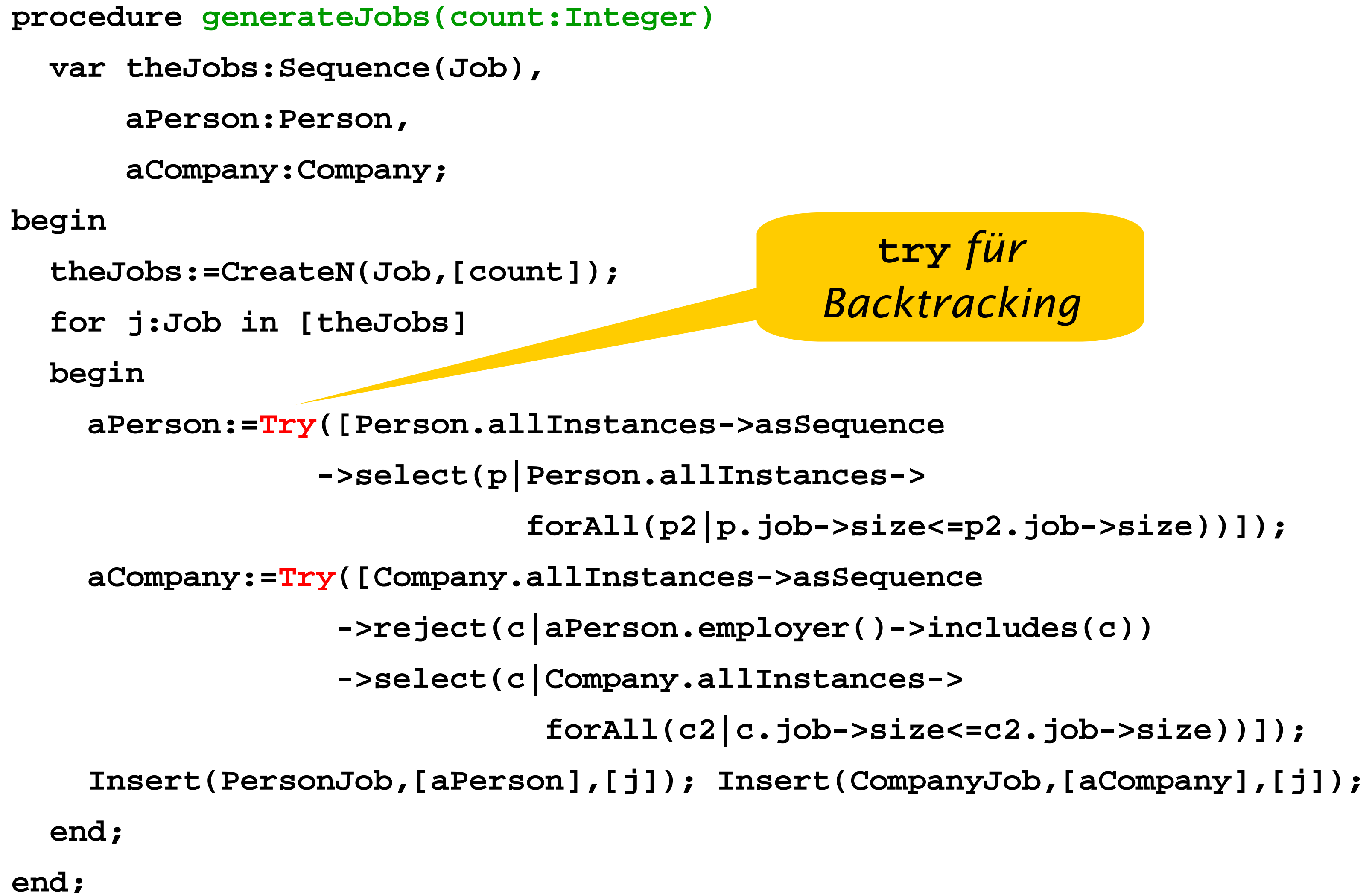
3.4 Fortgeschrittene USE-Verwendung (5)

*Nun **ASSL**-Prozeduren zum Generieren von Objektstrukturen:*

```
procedure generatePersons(count:Integer)
  var thePersons:Sequence(Person);
begin
  thePersons:=CreateN(Person,[count]); --< Personensequenz der Laenge count
  for p:Person in [thePersons]
    begin
     [p].name:=
      Any([Sequence{'Ada', 'Bob', 'Cher', 'Dan', 'Eva', 'Fred', 'Gil', 'Hal',
        'Ida', 'Joan', 'Kim', 'Lisa', 'Max', 'Nick', 'Ole', 'Pam', 'Qian',
        'Ron', 'Sam', 'Tom', 'Ulf', 'Vera', 'Wini', 'Xara', 'Yoko', 'Zoe'}
         -> reject(n1 | Person.allInstances.name -> exists(n2 | n1 = n2))
      ]);
  end;
end;
```

zufällige Auwahl mit Nebenbedingung

3.4 Fortgeschrittene USE-Verwendung (6)



```
procedure generateJobs(count:Integer)
  var theJobs:Sequence(Job),
      aPerson:Person,
      aCompany:Company;
begin
  theJobs:=CreateN(Job,[count]);
  for j:Job in [theJobs]
  begin
    aPerson:=Try([Person.allInstances->asSequence
                ->select(p|Person.allInstances->
                           forAll(p2|p.job->size<=p2.job->size))]);
    aCompany:=Try([Company.allInstances->asSequence
                 ->reject(c|aPerson.employer()->includes(c))
                 ->select(c|Company.allInstances->
                            forAll(c2|c.job->size<=c2.job->size))]);
    Insert(PersonJob,[aPerson],[j]); Insert(CompanyJob,[aCompany],[j]);
  end;
end;
```

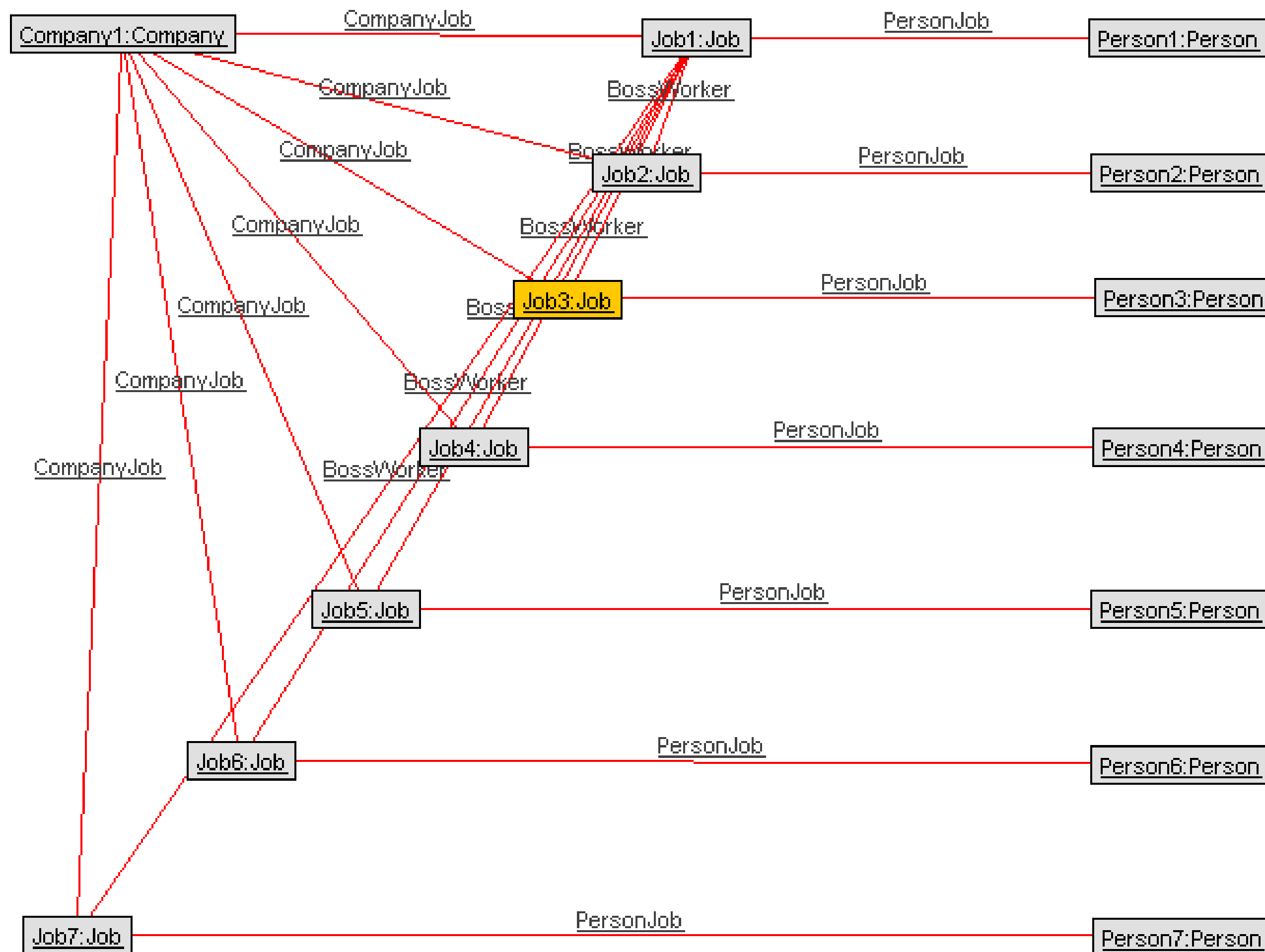
3.4 Fortgeschrittene USE-Verwendung (7)

*Aufrufe der **ASSL**-Prozeduren in einer **USE**-Kommando-Datei:*

```
-- generate a simple boss-worker hierarchy in one company

gen start PerCom/percom.assl generatePersons(7)
gen result                -- erzeugt und druckt Auswahl
gen result accept         -- legt Auswahl fest

gen start PerCom/percom.assl generateCompanies(1) -- analog zu generatePersons
gen result                -- erzeugt und druckt Auswahl
gen result accept         -- legt Auswahl fest

gen start PerCom/percom.assl generateJobs(7)
gen result                -- erzeugt und druckt Auswahl
gen result accept         -- legt Auswahl fest

gen start PerCom/percom.assl generateBossWorker(6) -- analog zu generateJobs
gen result                -- erzeugt und druckt Auswahl
gen result accept         -- legt Auswahl fest
```

3.4 Fortgeschrittene **USE**-Verwendung (8)

*Aus-
füh-
rung
der
USE-
Datei
ergibt:*



3.4 Fortgeschrittene **USE**-Verwendung (9)

*Auszug aus dem **Generator-Protokoll**:*

```
percom\percom-1.cmd> gen start PerCom/percom.assl generatePersons(7)
percom\percom-1.cmd> gen result
Random number generator was initialized with 60.
Checked 1 snapshots.
Result: Valid state found.
Commands to produce the valid state:
!create Person1,Person2,Person3,Person4,Person5,Person6,Person7 : Person
!set @Person1.name := 'Fred'
!set @Person2.name := 'Cher'
!set @Person3.name := 'Yoko'
!set @Person4.name := 'Max'
!set @Person5.name := 'Vera'
!set @Person6.name := 'Joan'
!set @Person7.name := 'Ada'
percom\percom-1.cmd> gen result accept
Generated result (system state) accepted.
```

3.4 Fortgeschrittene USE-Verwendung (10)

In diesem Zustand rufen wir eine Methode auf und prüfen die Invarianten:

```
use> !openter Job3 workerPlus()
use> check
checking structure...
checking invariants...
checking invariant (1) `Company::companyNamesAreUnique': OK.
checking invariant (2) `Job::bossBetterPaidThanWorker': OK.
checking invariant (3) `Job::bossWorkerIsHierarchy': OK.
checking invariant (4) `Job::bossWorkerSameEmployer': OK.
checking invariant (5) `Job::employeeEmployerAreUnique': OK.
checking invariant (6) `Person::personNamesAreUnique': OK.
checked 6 invariants in 0.000s, 0 failures.
use> info opstack
active operations:
1. Job::workerPlus() : Set(Job) | Job3.workerPlus()
use> !opexit Set{Job1}
use> info opstack
no active operations.
```

3.4 Fortgeschrittene **USE**-Verwendung (11)

Im nächsten Beispiel generieren wir unter Beachtung der folgenden, dynamisch hinzugeladenen ***Invarianten****:*

```
context Job inv threeLevelHierarchy:
  Job.allInstances->exists(top, mid, low |
    top.worker->includes(mid) and mid.worker->includes(low))
  -- Jobs bilden mindestens dreistufige Hierarchie (top, mid, low)

context Person inv twoRoleEmployee:
  Person.allInstances->exists(p |
    p.job->exists(low, top |
      low.boss.isDefined and
      top.worker->notEmpty and
      low.employer<>top.employer))
  -- p ist in einer Firma Vorgesetzter, in anderer Firma Untergebener
```

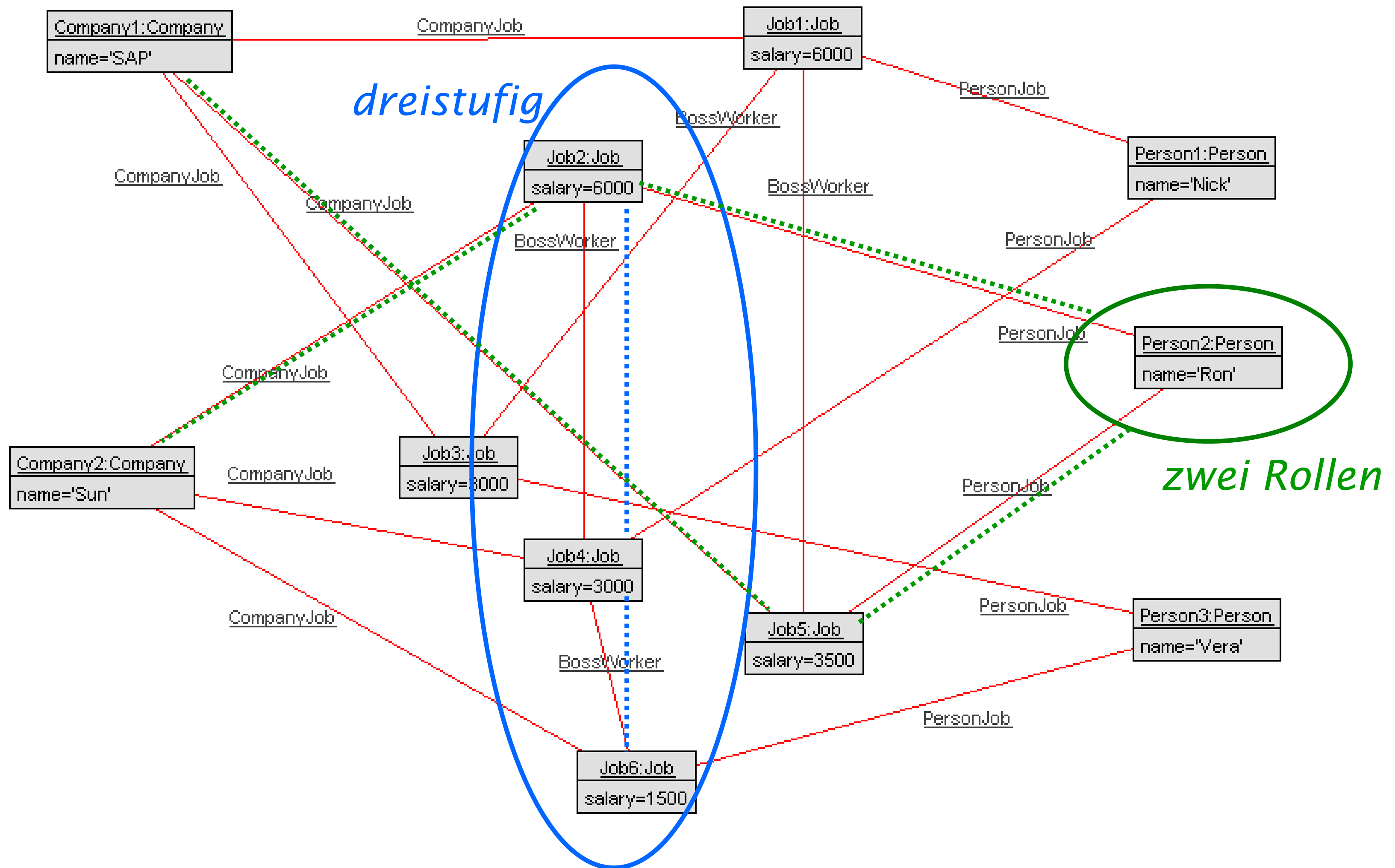
3.4 Fortgeschrittene USE-Verwendung (12)

In der USE-Datei werden die Invarianten nachgeladen:

```
-- Wir rufen zuerst generatePersons(3) generateCompanies(2) und
-- und generateJobs(6) auf (jeweils mit gen result accept danach).

...

-- Dann folgt das Nachladen der Invarianten ...

gen load PerCom/threeLevelHierarchy.invs
gen load PerCom/twoRoleEmployee.invs

-- ... welche die abschliessende Generierung
-- mit generateBossWorker(4) steuern

gen start PerCom/percom.assl generateBossWorker(4)
gen result
gen result accept
```

Das Ergebnis zeigt die nächste Folie!

3.4 Fortgeschrittene USE-Verwendung (13)



3.4 Fortgeschrittene **USE**-Verwendung (14)

*Besonders interessant ist die Verwendung von **ASSL** in der **USE**-Datei* `percom-7.cmd`*. Dort wird gezeigt, dass aus den stets geforderten **I**nvarianten (siehe Folie 40) und der **D**reistufigkeit die Eigenschaft **BB** ("niemals ist eine Person besser bezahlt als der Boss ihres Bosses") folgt, also*

I** and **D** implies **BB

Da dies gleichwertig ist mit

*not (**I** and **D** and not **BB**)*

*genügt es, **D** und das Negat von **BB** (zu **I**) hinzuzuladen und zu zeigen, dass sich nun per ASSL **kein Modell** konstruieren lässt.*

*Dazu starten wir wieder das **USE**-System.*

3.5 Zusammenfassung und Bewertung (1)

- *OCL dient zur **Grobspezifikation** von OO-Systemen.*
- *Die Syntax von OCL ist an die Programmiersprache Smalltalk angelehnt, der **Sprachumfang** und damit der **Lernaufwand** ist im Verhältnis zur Ausdrucksstärke **relativ gering**. OCL ist **deklarativ** und beschreibt Ausdrücke (Mengen, Prädikate).*
- *Die Semantik von OCL ist **"direkt definiert"**, z.B. das iterate-Konstrukt durch Angabe von definierendem Pseudocode. Weitere Konstrukte (collect, select, reject etc.) sind darauf durch Definitionen im denotationellen Stil abgestützt.*
- *OCL seinerseits dient als **Metasprache** zur Definition der Semantik von **UML-Elementen**.*

3.5 Zusammenfassung und Bewertung (2)

- *OCL ist eine **getypte** Sprache mit*
  - ***Kollektionen**: Für die vordefinierten Typen **Set**, **Bag** und **Sequence** erlauben die gegebenen Operationen einen Beschreibungsstil, der stark an das Arbeiten mit **Mengenkomprehensionen** erinnert "nahe an der Mathematik bei syntaktische Nähe zu deklarativen Sprachen".*
  - ***Skalaren Wertetypen** (**Boolean**, **Integer**, **Real**) und **Strings** sind vordefiniert.*
  - ***Benutzerdefinierten** Typen: **Enumerations**- und **Objekttypen**.*
  - ***Mehrfachvererbung**; an der **Wurzel** der Klassenhierarchie befindet sich genau eine Klasse namens* **`OclAny`**. *Diese Klasse enthält die Basis der **Reflexionsmechanismen**, die OCL als **Metasprache** benötigt.*

3.5 Zusammenfassung und Bewertung (3)

- *Im Vergleich zu der Vielzahl an **UML-Werkzeugen** gibt es nur relativ wenige Werkzeuge, die OCL nennenswert unterstützen.*
  - *Das hier beschriebene Werkzeug **USE** dient der **Animation** von OCL-Modellen, d.h. der Konstruktion von **Objektdiagrammen** und der Prüfung auf **Konsistenz** zum gegebenen OCL-Modell.*
  - *Die Erweiterung **ASSL** des **USE**-Werkzeugs unterstützt die **programmgetriebene Erzeugung** von Objektdiagrammen; dabei stehen **Backtracking** und die **Invarianten-gesteuerte** Erzeugung für die Erstellung kompexer Systemzustände bereit. Mit **ASSL** kann man systematisch untersuchen, ob eine **Systemeigenschaft** aus anderen folgt ("Prolog-artiger Ansatz").*
- *Im **Software-Entwicklungsprozess** ist OCL vorwiegend für die Anforderungsanalyse und das Grobdesign geeignet; die Werkzeuge **USE** und **ASSL** unterstützen die Validierung.*

Literatur zu diesem Kapitel

- *OMG: UML 2.0* ***OCL****-Spezifikation*
  http://www.omg.org/docs/ptc/03-10-14.pdf
- *J. Warmer, A. Kleppe: The* ***O****bject* ***C****onstraint* ***L****anguage. Addison-Wesley, 1999.*
- *Die* ***USE****-Homepage:*
  http://www.db.informatik.uni-bremen.de/projects/USE/
  - *Dort findet man das* ***USE-System*** *nebst ausführlicher Doku zum Download;*
  - *dazu viele interessante Artikel und Fallstudien zur Verwendung von* ***USE****.*