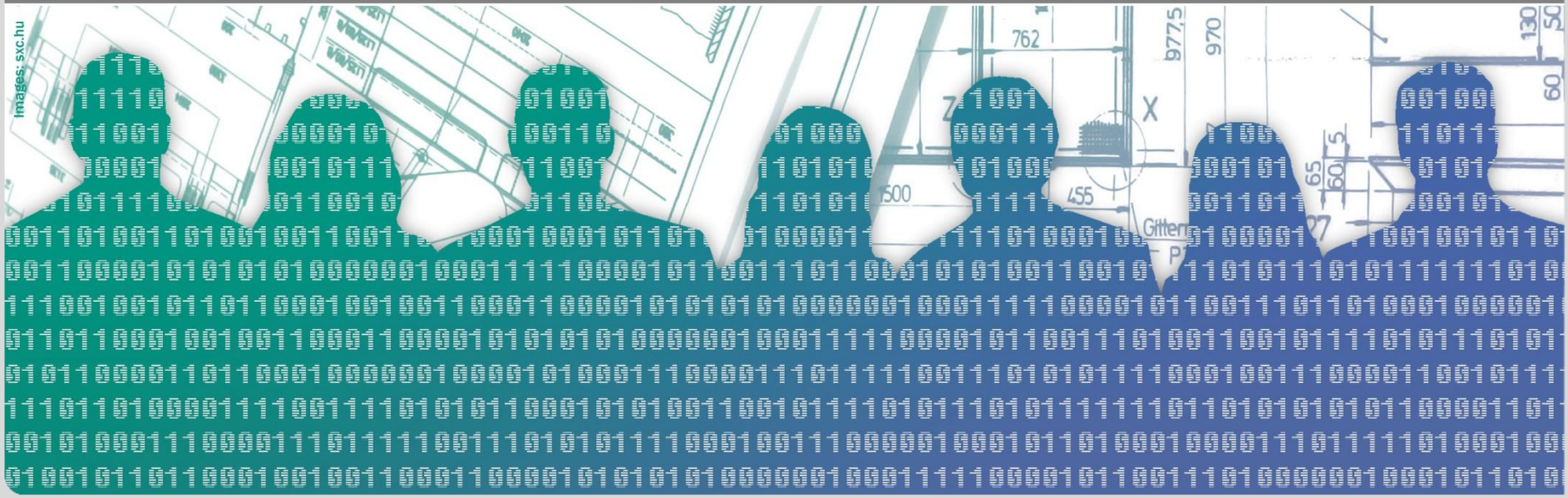
Einschub - Die Object Constraint Language in UML
Oder: Wie man Zusicherungen in UML angibt

SWT I – Sommersemester 2010
Walter F. Tichy, Andreas Höfer, Korbinian Molitorisz

IPD Tichy, Fakultät für Informatik



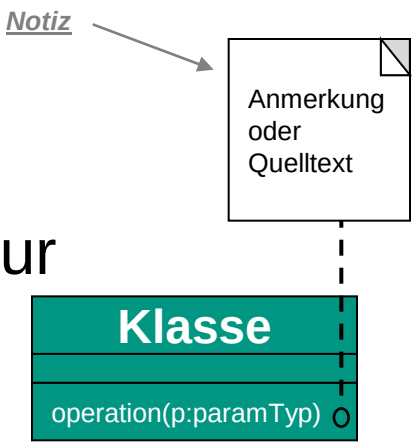
Zusicherungen

- Def. **Zusicherung** (engl. *constraint*, auch *assertion*): ein Ausdruck, der die zulässigen Ausprägungen, Inhalte oder Zustände eines Modellelementes beschreibt und sich stets zu „wahr“ auswerten lassen muss
- Formulierung der Zusicherungen auf Modellebene mit der „Object Constraint Language“ (OCL) der OMG.

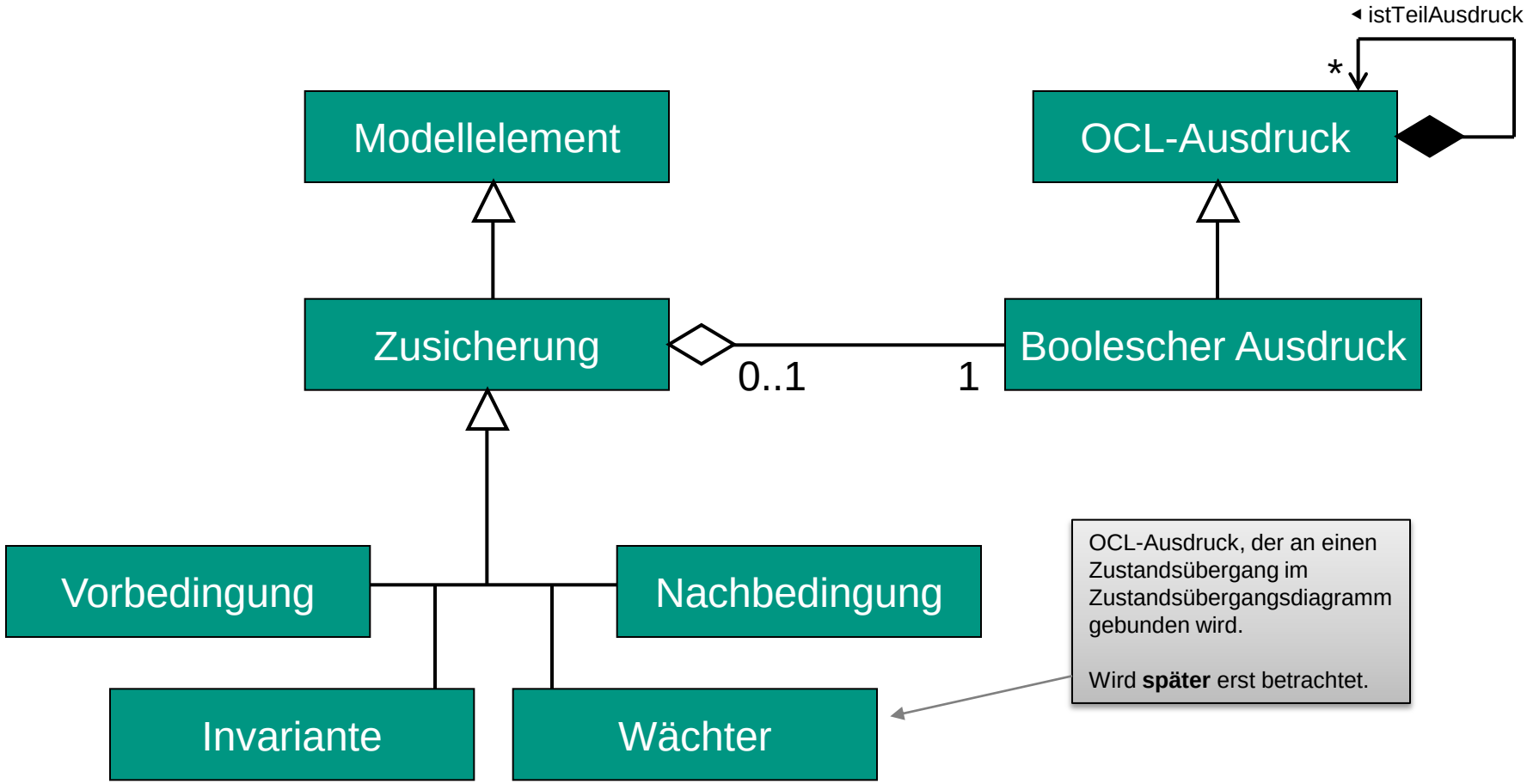
Zusicherungen

- Eine Zusicherung beschreibt eine Bedingung, die immer gelten muss
  - Sie kann die zulässige Wertemenge eines Attributes einschränken (Restriktion der Domäne)
  - Sie kann Vor- und Nachbedingung für Nachrichten bzw. Operationen angeben
  - Einen speziellen Kontext für Nachrichten oder Beziehungen festlegen
  - Strukturelle Eigenschaften zusichern
  - Ordnungen definieren
  - Zeitliche Zusammenhänge definieren
  - Etc.

⇒ Zusicherungen stellen letztlich einen Ansatz zur Formalisierung der Notiz dar.



Beispiel: Metamodell der Zusicherungen in UML (vereinfacht)



Notation von Zusicherungen in OCL

- Invarianten können in geschweiften Klammern direkt hinter einem Modellelement definiert werden:
  - Muster:

```
{ Zusicherung }
{ Name: Zusicherung }
```

  - Beispiel:
    - Für ein Attribut `radius` geben wir an:

```
radius : float    {radius>=0.0}
```

Notation von Zusicherungen in OCL

- Separat mit dem Schlüsselwort `context`:
  - Muster:
    - `context` *VerantwortlichesElement*

```
        pre  Name0 : Zusicherung0
        inv  Name1 : Zusicherung1
        post Name2 : Zusicherung2
```

Vorbedingung (optional)
Invariante (optional)
Nachbedingung (optional)
Name der Zusicherung (optional)

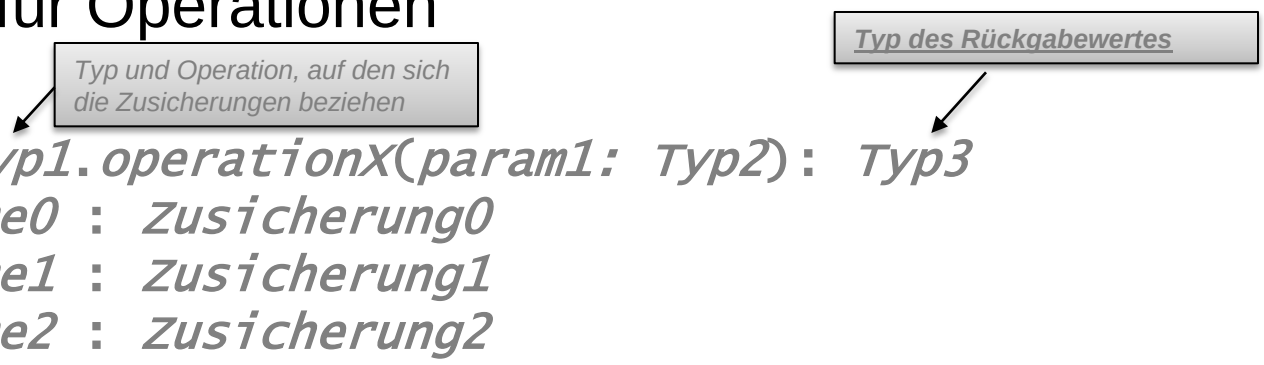
  - Beispiel:
    - `context Person`

```
        inv Volljährigkeit: self.alter >= 18
```

  - Die vordefinierte `self`-Referenz bezieht sich stellvertretend auf jedes Exemplar der genannten Klasse.

Notation von Zusicherungen in OCL

- Zusicherungen für Operationen
  - Muster:



  - Hier ist `param1` explizit als Parameter der Operation `operationX` definiert worden.
  - In den Zusicherungsausdrücken können alle Parameter und alle Attribute des Typs verwendet werden.

Zusicherungen von Vor- und Nachbedingungen

- Im Ausdruck der Nachbedingung darf die vordefinierte Variable `result` verwendet werden. Sie enthält das Ergebnis, also den Rückgabewert der Operation.
  - ```
    context Rect.getArea(): Float
      post: result = self.width * self.height
    ```
- An den Wert eines Attributes oder Parameters vor der Berechnung kommt man mit dem Suffix „`@pre`“
  - ```
    context Person.birthDayHappens()
      post: age = age@pre + 1
    ```

Vordefinierte OCL-Basistypen

- Typen
  - Boolean — true, false
  - Integer — -1, 2, 543523553
  - Real — 3.14
  - String — 'Raumschiff'
  - Set — {55, 23, 47}, { 'R', 'G', 'B'} *ohne Duplikate*
  - Bag — {12, 8, 8} *mit Duplikaten*
  - Sequence — {1..10}, {8, 17, 25, 26} *mit Ordnung*

- Es gilt: *Set*, *Bag* und *Sequence* sind Untertypen von *Collection*.

Vordefinierte OCL-Basisoperationen

- Auf Integer & Real
  - `i1 = i2` Ergebnis: Boolean
  - `i1 + i2` Ergebnis: Integer
  - `i1 + r1` Ergebnis: Real
  - `r1 round` Ergebnis: Integer

- Auf Boolean
  - `a and b`
  - `a or b`
  - `a xor b`
  - `not a`
  - `a implies b`
  - `a = b`
  - `if a then` *`Expr1`* `else` *`Expr2`*
    Wenn `a`, dann wird das Ergebnis des Gesamtausdrucks (inkl. Typ) durch *`Expr1`* bestimmt, sonst durch *`Expr2`* (müssen natürlich beides OCL-Ausdrücke sein).

Vordefinierte OCL-Basisoperationen

- Auf allen Kollektionen (Collection)

`c1 = c2`	Boolean	Sind die Elemente gleich?
`C->size()`	Integer	Anzahl der Elemente
`c->sum()`	Integer oder Real	Summe aller Elemente, falls num.
`C->includes(e)`	Boolean	`e` ∈ `c`?
`C->isEmpty()`	Boolean	`c` = ∅ ?
`C->notEmpty()`	Boolean	`c` ≠ ∅ ?
`C->exists(`*`Expr`*`)`	Boolean	∃ Element, so dass *`Expr`* wahr?
`C->forAll(`*`Expr`*`)`	Boolean	Ist *`Expr`* wahr für alle Elemente?

Vordefinierte OCL-Basisoperationen

- Zusätzlich auf Set

`S1->union(s2)`	Set	Vereinigungsmenge
`S1->intersection(s2)`	Set	Schnittmenge
`S1-S2`	Set	$S1 \setminus S2$
`S->include(e)`	Set	$S \cup \{e\}$
`S->exclude(e)`	Set	$S \setminus \{e\}$
`S1->symmetricDifference(S2)`	Set	$(S1 \cup S2) \setminus (S1 \cap S2)$
`S->select(`*Expr*`)`	Set	$\{e \in S \mid$ *Expr*$(e)\}$
`S->reject(`*Expr*`)`	Set	$\{e \in S \mid$ `not` *Expr*$(e)\}$
`S->collect(`*Expr*`)`	**Bag**	$\{$*Expr*$(e) \mid e \in S\}$
`S->asSequence()`	**Seq.**	(beliebige Reihenfolge)
`S->asBag()`	**Bag**	(einfache Typkonversion)

Vordefinierte OCL-Basisoperationen

- Zusätzlich auf Bag

`B1->union(B2)`	Bag	Vereinigung
`B1->intersection(B2)`	Bag	Schnitt
`B1-B2`	Bag	*(analog zu Set)*
`B->include(e)`	Bag	*(analog zu Set)*
`B->exclude(e)`	Bag	*(analog zu Set)*
`B->select(`*Expr*`)`	Bag	*(analog zu Set)*
`B->reject(`*Expr*`)`	Bag	*(analog zu Set)*
`B->collect(`*Expr*`)`	Bag	*(analog zu Set)*
`B->asSequence()`	**Seq.**	(beliebige Reihenfolge)
`B->asSet()`	**Set**	(Duplikate entfernt)

Vordefinierte OCL-Basisoperationen

- Zusätzlich auf Sequence

`S1->append(S2)`	Seq.	Konkatenation `S1` ∘ `S2`
`S1->prepend(S2)`	Seq.	Konkatenation `S2` ∘ `S1`
`S->first()`	Elem.	Erstes Element
`S->last()`	Elem.	Letztes Element
`s->at(i)`	Elem.	`i`. Element
`S->include(e)`	Seq.	`e` wird hinten angefügt
`S->exclude(e)`	Seq.	`S` ohne das erste (!) `e`
`S->select(Expr)`	Seq.	*(analog zu Set)*
`S->reject(Expr)`	Seq.	*(analog zu Set)*
`S->collect(Expr)`	**Seq.**	*(analog zu Set)*
`S->asSet()`	**Set**	(Duplikate entfernt)
`S->asBag()`	**Bag**	(einfache Typkonversion)

Iteratoren

- Vergleiche innerhalb von Kollektionen (z.B. diejenigen, die größer als ein anderer sind) lassen sich mit den gezeigten Funktionen nicht realisieren

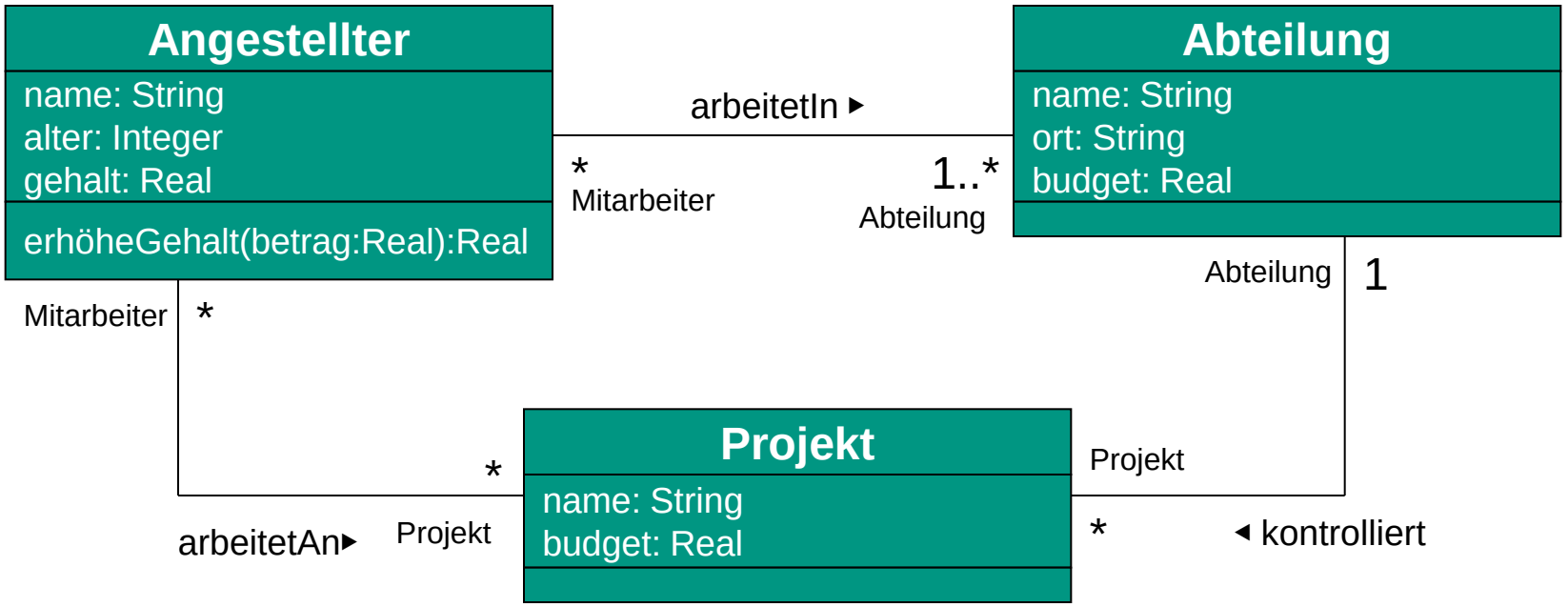
- Iteratorvariablen binden sukzessive alle Exemplare einer Klasse an sich und ermöglichen so Vergleiche innerhalb einer Kollektion

```
context Firma inv:
    self.angestellter->forAll(a1, a2 |
          a1 <> a2 implies a1.PersNr <> a2.PersNr)
```

Beispiel: `select`

- Folgende Ausdrücke sind äquivalent:
  - ```
    context Abteilung inv:
    self.Mitarbeiter->select(Alter > 50)->notEmpty()
    ```
  - ```
    context Abteilung inv:
    self.Mitarbeiter->select(p | p.Alter > 50)
    ->notEmpty()
    ```
  - ```
    context Abteilung inv:
    self.Mitarbeiter->select(p : Person | p.Alter > 50)
    ->notEmpty()
    ```
- Jede Abteilung muss mindestens einen Mitarbeiter haben, der über 50 Jahre alt ist.
    ```

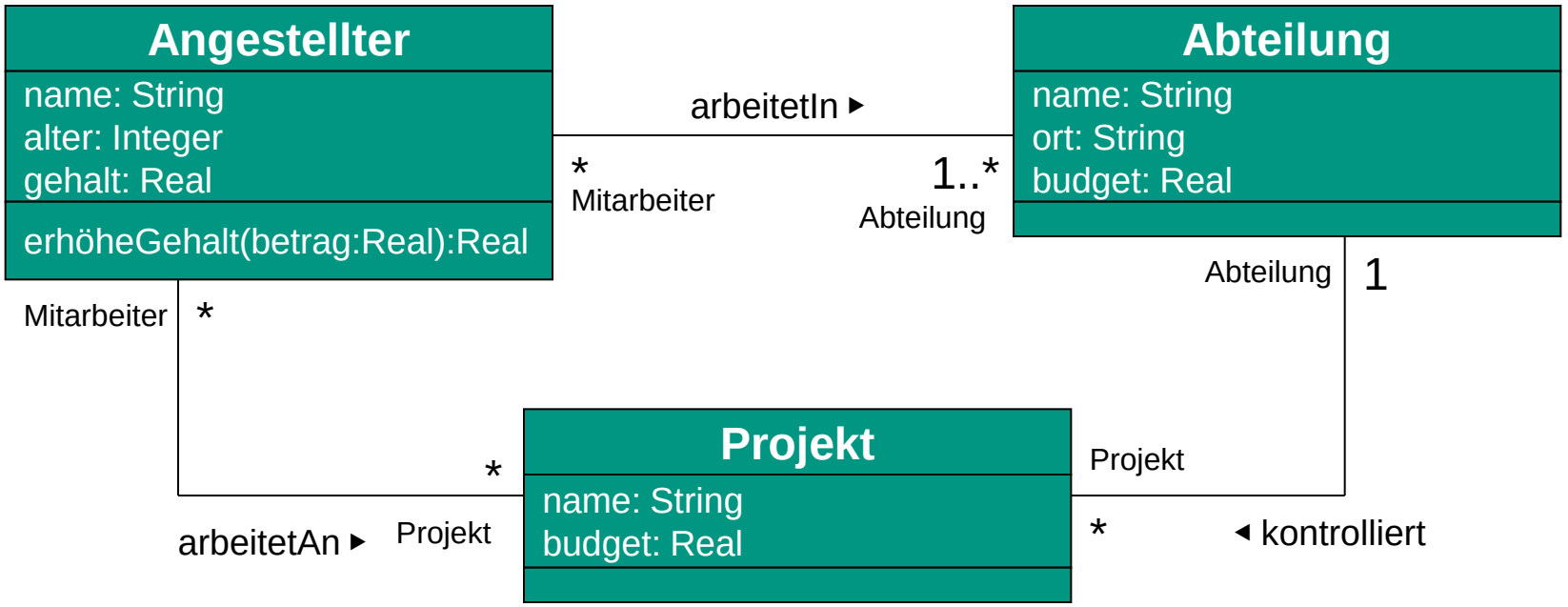
OCL-Beispiel



- Keine Abteilung darf jemals ein negatives Budget haben:

```
context Abteilung inv:
    self.budget >= 0
```

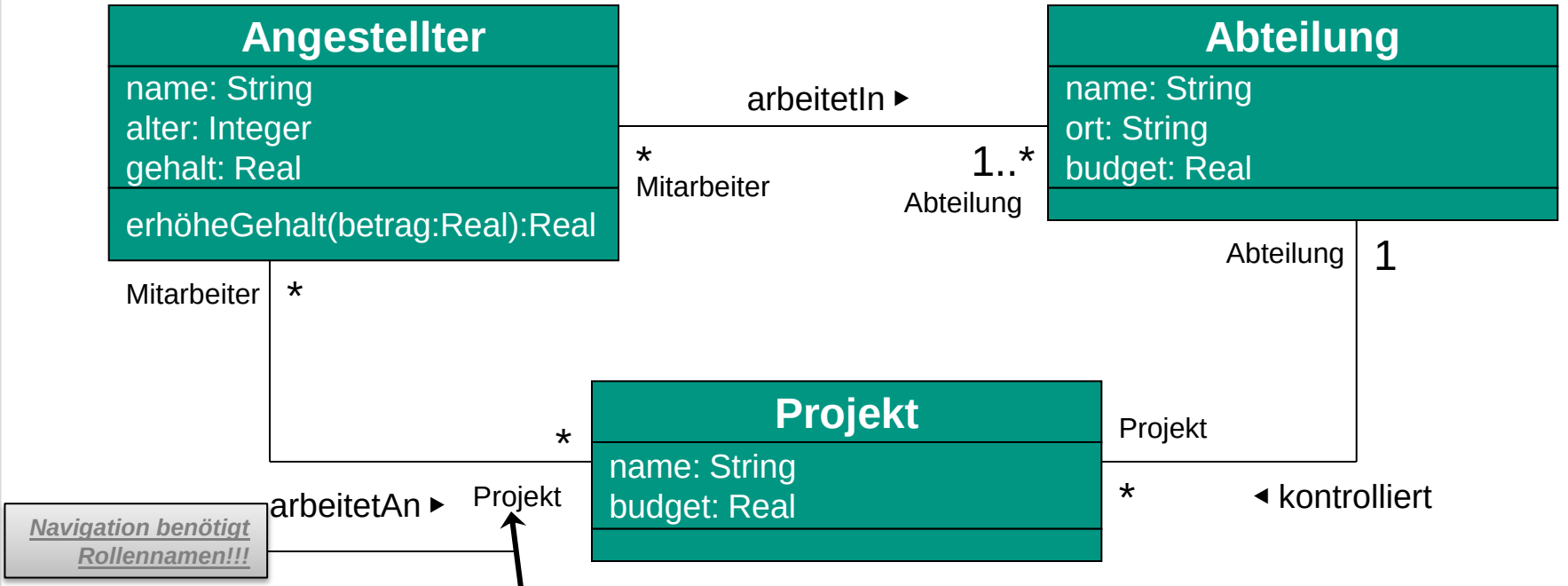
OCL-Beispiel



- Der Code für Gehaltserhöhungen muss folgenden Anforderungen genügen:

```
context Angestellter.erhöheGehalt(betrag : Real) : Real
    pre:   betrag > 0
    post:  self.gehalt = self.gehalt@pre + betrag
           and result = self.gehalt
```

OCL-Beispiel



- Angestellte, die an mehr Projekten arbeiten, als andere Mitarbeiter der selben Abteilung, bekommen ein höheres Gehalt :

```
context Abteilung inv:
    self.Mitarbeiter->forAll(a1, a2 |
          a1.Projekt->size() > a2.Projekt->size()
          implies a1.gehalt > a2.gehalt)
```

Ein weiterer Aufpunkt

- Häufig benötigt man für Vergleiche das Kreuzprodukt. Dieses lässt sich mit der Funktion `allInstances` leicht bilden:

```
context Angestellter inv:
    Angestellter.allInstances()->forAll(a1, a2 |
        a1 <> a2 implies a1.name <> a2.name)
```

Literatur

- Bernd Oestereich, „Analyse und Design mit UML 2.x – Objektorientierte Softwareentwicklung“
- UML-Spezifikation unter http://www.uml.org/#UML2.0
  - Speziell „UML 2.0 Superstructure Specification“
  - und „UML 2.0 Infrastructure Specification“, insbes. Ch. 4 Terms and Definitions
- Zu OCL
  - Richters & Gogolla, „OCL: Syntax, Semantics, and Tools“ in A. Clark and J. Warmer (Eds.): Object Modeling with the OCL, LNCS 2263, pp. 42–68, 2002.
  - „Object Constraint Language – Version 2.0“, OMG http://www.omg.org/cgi-bin/doc?formal/2006-05-01