

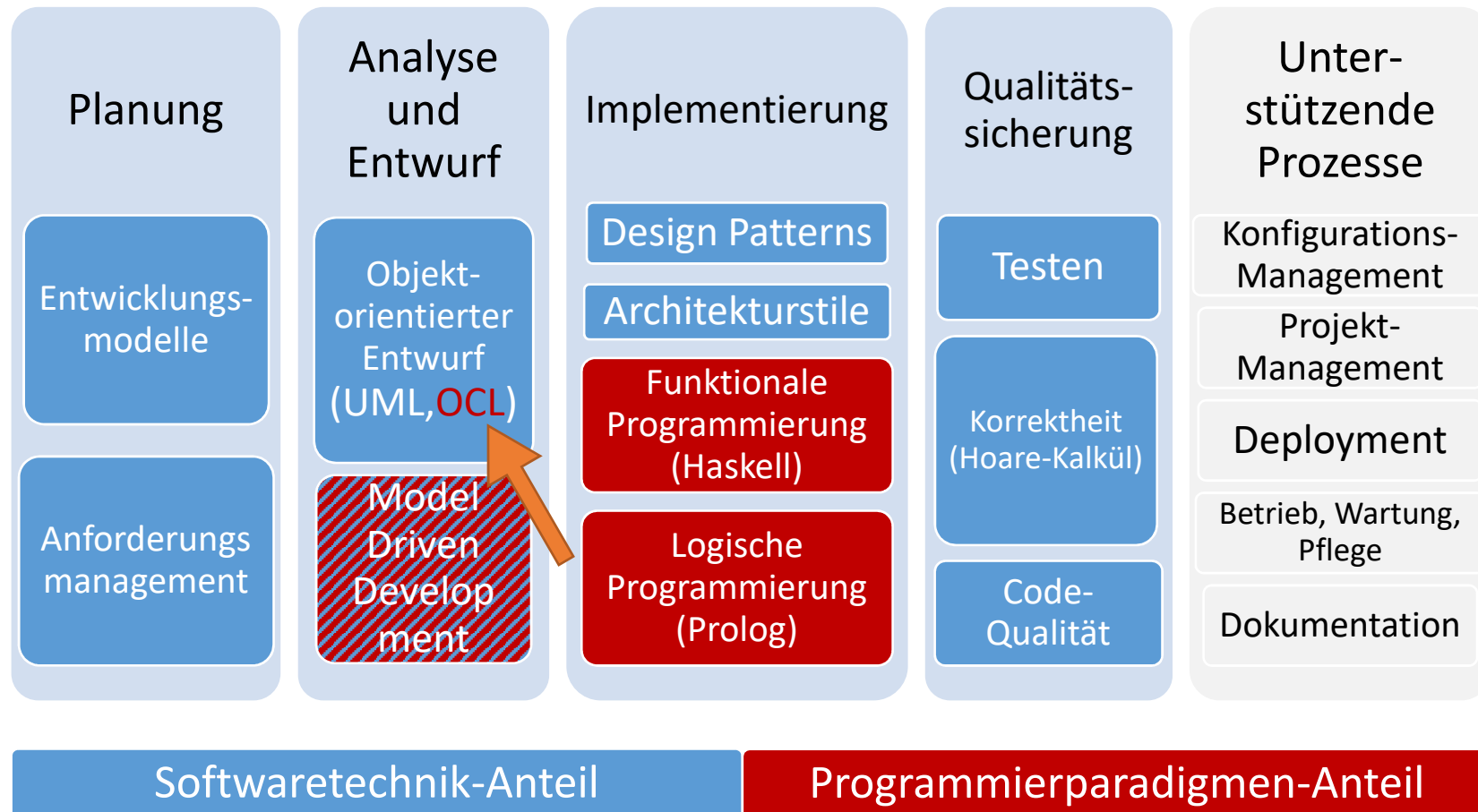
Softwaretechnik und Programmierparadigmen

09 Formale Spezifikation

Prof. Dr. Sabine Glesner
Software and Embedded Systems Engineering
Technische Universität Berlin



Diese VL



Inhalt

Formale Spezifikation

- Grundlagen
- Object Constraint Language (OCL)
- Invarianten
- Contracts
- Weiteres

Inhalt

Formale Spezifikation

- Grundlagen
- Object Constraint Language (OCL)
- Invarianten
- Contracts
- Weiteres

Motivation

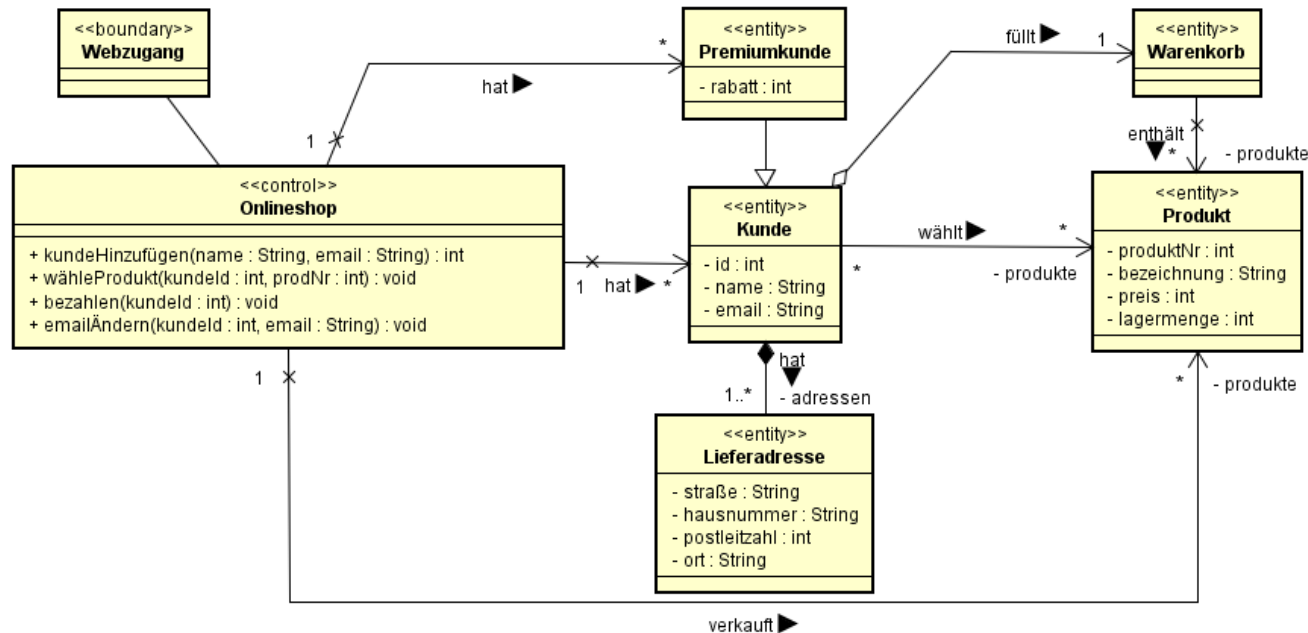


[Via Jordi Cabot](#)

Motivation

Klassendiagramme beschreiben nur die **Struktur**

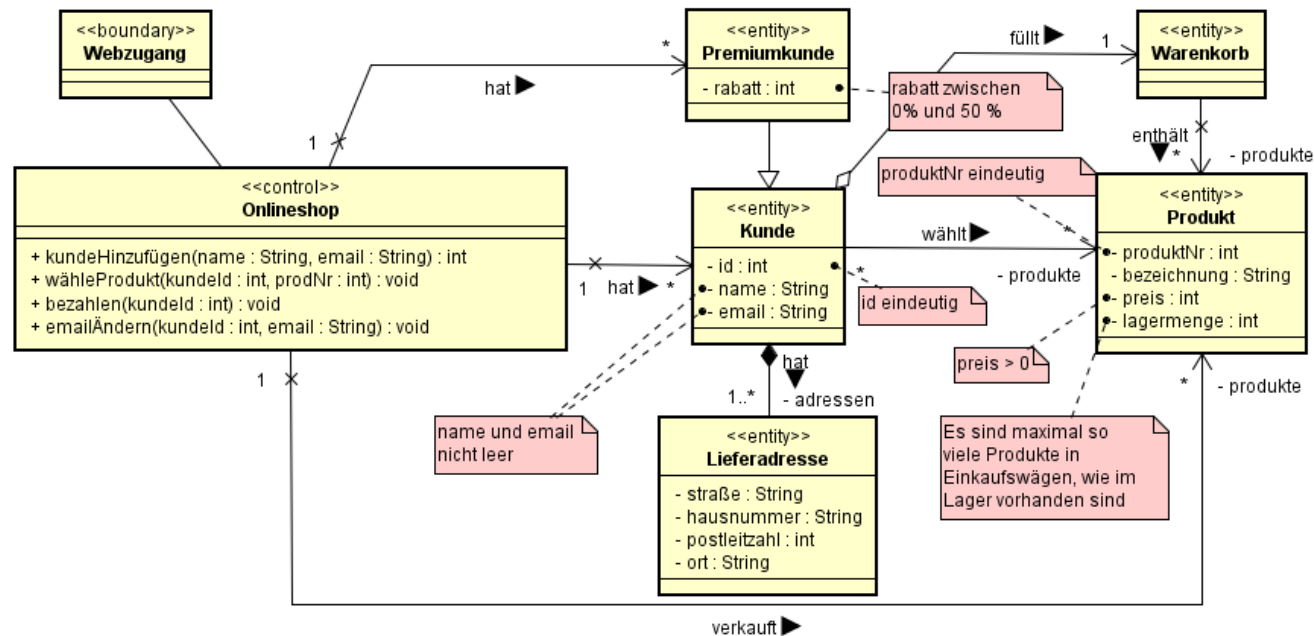
- Bereiche und Bedeutung von Variablen/Operationen wird nicht dargestellt



Welche sinnvollen Einschränkungen fehlen hier?

Einschränkung durch Kommentare

Natürlich können Einschränkungen textuell hinzugefügt werden, aber...



...geht das nicht besser?

Object Constraint Language

Sprache zur **formalen Beschreibung** von Eigenschaften in UML

- Mitte der 90er von IBM entwickelt, später standardisierter Teil von UML
- Logische Sprache und damit frei von Seiteneffekten
- Typisiert und typsicher

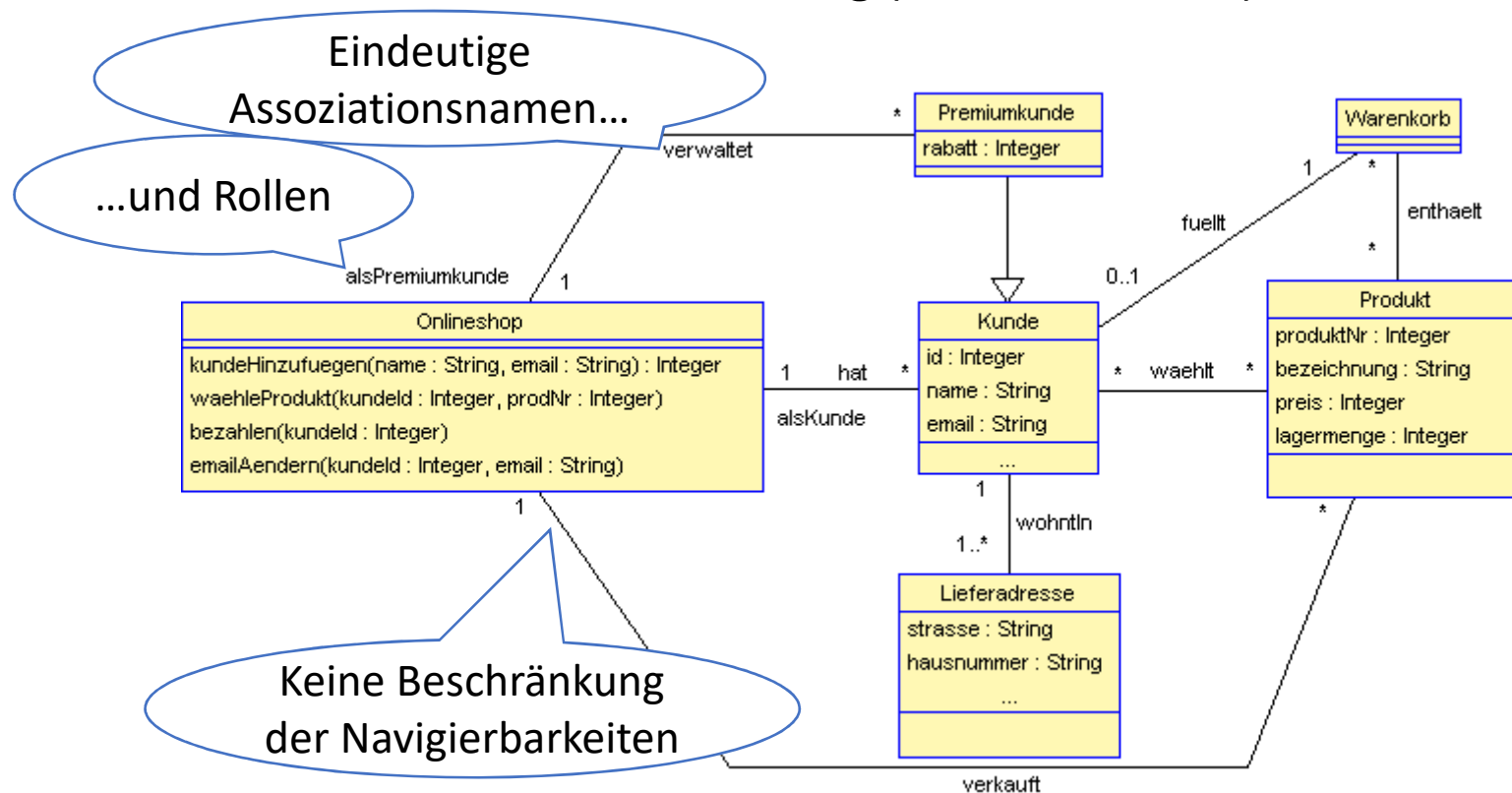
Enthält vordefinierte Mechanismen um...

- ...auf Werte von Objekten **zuzugreifen**
- ...durch verbundene Objekte zu **navigieren**
- ...über Collections zu **iterieren**
- ...mit primitiven Typen und Collections zu **arbeiten**

„You may not like it but right now there is **nothing better** than OCL“
Jordi Cabot

Hinweis

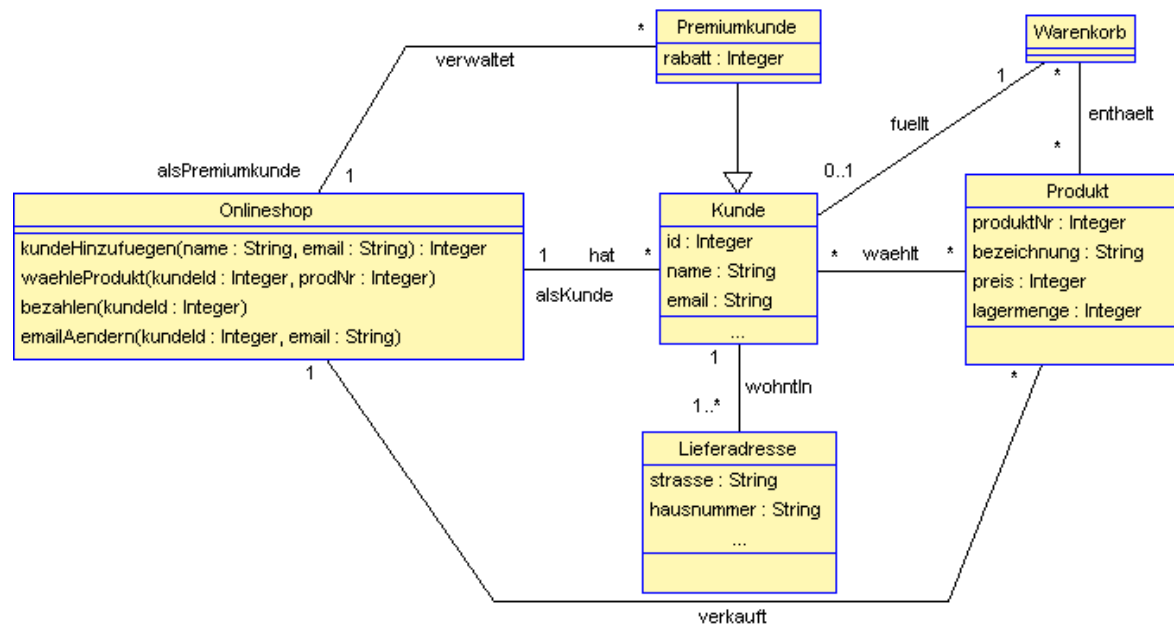
Für OCL wechseln wir das Werkzeug (USE statt Astah)...



Ein erster Eindruck

- a) Der Preis von jedem Produkt ist immer größer als 0
- b) Die IDs der Kunden existieren nur einmal im System

} Sprache



- a) context Produkt inv : self.preis > 0
- b) context Kunde inv: Kunde.allInstances().id->count(self.id) = 1

} OCL

Inhalt

Formale Spezifikation

- Grundlagen
- Object Constraint Language (OCL)
- Invarianten
- Contracts
- Weiteres

Werte

Auf Werte von Objekten kann mit der **Punktnotation** zugegriffen werden

$$[\text{Objektbezeichner}].[\text{Attributbezeichner}] \equiv [\text{Wert}] : [\text{Typ}]$$

Beispiele

pk.id	$\equiv 128 : \text{Integer}$
pk.name	$\equiv \text{'Lisa'} : \text{String}$
pk.email	$\equiv \text{' *****@tu-berlin.de'} : \text{String}$
pk.rabatt	$\equiv 15 : \text{Integer}$

pk:Premiumkunde
id=128 name='Lisa' email=' *****@tu-berlin.de' rabatt=15

Primitive Typen

In OCL sind vier **primitive Typen** vordefiniert

- Außerdem sind einige spezifische **Operationen** bereits vorhanden

Typ	Operationen	Beispiel
Integer	<code>*</code> , <code>+</code> , <code>-</code> , <code>/</code> , <code>abs()</code> , ... , <code>toString()</code>	0, 1, -1, 3, -42, ...
Real	<code>*</code> , <code>+</code> , <code>-</code> , <code>/</code> , <code>floor()</code> , ... , <code>toString()</code>	0.0, 0.3, -1.7, 187.238, ...
Boolean	<code>and</code> , <code>or</code> , <code>xor</code> , <code>not</code> , <code>implies</code> , <code>if-then-else-endif</code> , <code>toString()</code>	true, false
String	<code>concat()</code> , <code>size()</code> , <code>substring()</code> , ...	<code>"</code> , <code>'a'</code> , <code>'abcd'</code> , ...

Primitive Typen

Operationen auf **primitiven Typen** funktionieren wie gewohnt

- Dabei findet kein implizites Typcasting statt

Beispiele

p1.lagermenge - 12

p1.produktNr > 0

p1.produktNr = p2.produktNr

if p1.preis > p2.preis then 1 else 2 endif

p1.bezeichnung.size()

p1.bezeichnung.concat(' Brille')

p1.bezeichnung.substring(1, 3)

≡ 6 : Integer

≡ true : Boolean

≡ false : Boolean

≡ 1: Integer

≡ 7 : Integer

≡ 'Raybaem Brille' : String

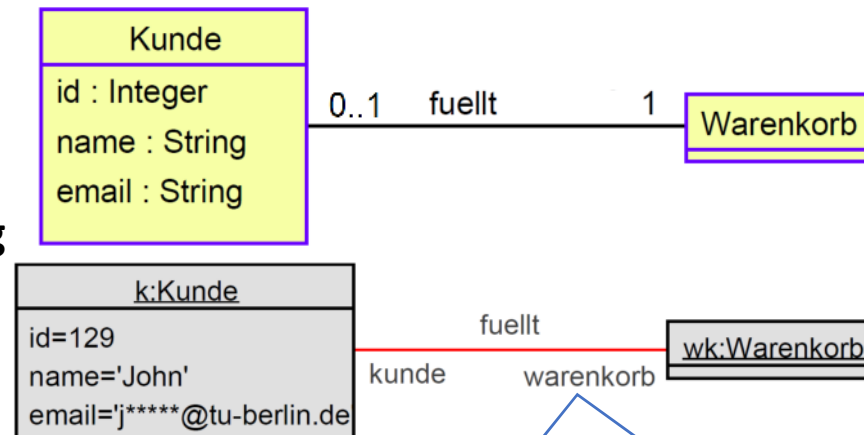
≡ 'Ray' : String

p1:Produkt	p2:Produkt
produktNr=8785 bezeichnung='Raybaem' preis=190 lagermenge=18	produktNr=3663 bezeichnung='Adidos' preis=90 lagermenge=43

Navigation

Der **Zugriff über Assoziationen** erfolgt **analog** zum Zugriff auf Attribute

- Dabei gibt die **Rollenbezeichnung** den Attributnamen an



Beispiele

<code>wk.kunde</code>	\equiv <code>k : Kunde</code>
<code>wk.kunde.id</code>	\equiv <code>129 : Integer</code>
<code>wk.kunde.name</code>	\equiv <code>'John' : String</code>

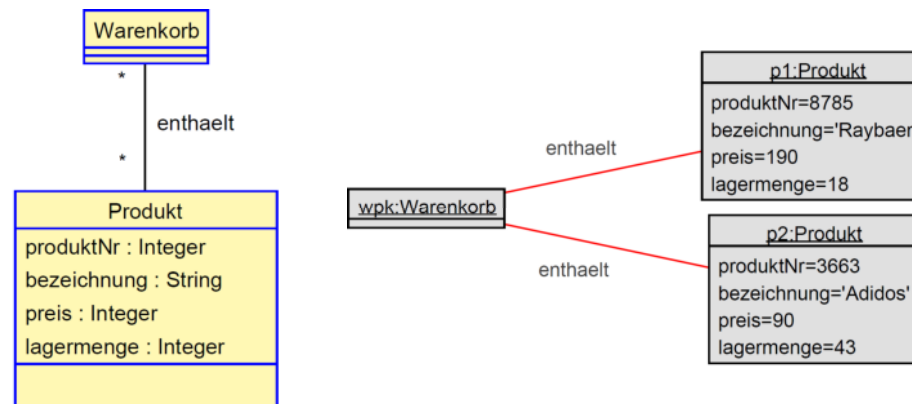
Sind keine Rollen
spezifiziert wird der
**kleingeschriebene
Klassenname**
verwendet

Wie funktioniert Navigation über Assoziationen mit *****?

Navigation auf Collections

Beim Zugriff auf **Assoziationen mit *** gibt OCL eine **Collection** zurück

- Operationen auf **Collections** verwenden als Notation den Pfeil (->)



Beispiele

`wpk.produkt` \equiv `Set{p1,p2} : Set(Produkt)`

`wpk.produkt->size()` \equiv `2 : Integer`

Collections

In OCL existieren vier **unterschiedliche Collections...**

Elemente...	Set	Bag	OrderedSet	Sequence
...können mehrfach enthalten sein	<i>Nein</i>	<i>Ja</i>	<i>Nein</i>	<i>Ja</i>
...haben eine Reihenfolge	<i>Nein</i>	<i>Nein</i>	<i>Ja</i>	<i>Ja</i>

...mit gemeinsamen, vordefinierten Operationen

c->**size()** : Integer

Anzahl Elemente in Collection c

c->**isEmpty** : Boolean

true, wenn c leer ist

c->**includes**(obj:OclAny) : Boolean

true, wenn obj in c vorkommt

c->**excludes**(obj:OclAny) : Boolean

false, wenn obj in c vorkommt

c->**count**(obj:OclAny) : Integer

Häufigkeit von obj in c

Set/Bag Operationen

Collections ohne Beachtung der Reihenfolge erlauben
Mengenoperationen

$= (y : \text{Set}(T)) : \text{Boolean}$

including($y : T$) : $\text{Set}(T)$

excluding($y : T$) : $\text{Set}(T)$

union($y : \text{Set}(T)$) : $\text{Set}(T)$

intersection($y : \text{Set}(T)$) : $\text{Set}(T)$

$-(y : \text{Set}(T)) : \text{Set}(T)$

Gleichheit

Hinzufügen

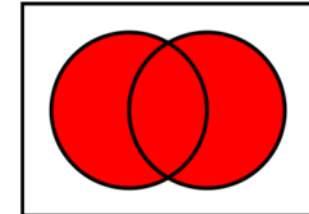
Entfernen

Vereinigung

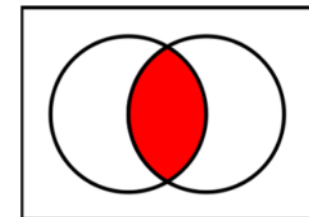
Schnitt

Differenz (nur Set)

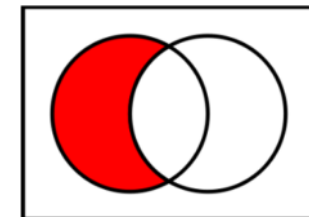
Vereinigung



Schnitt



Differenz



OrderedSet/Sequence Operationen

Collections mit Berücksichtigung der Reihenfolge erlauben **Indexzugriff**

at(y : Integer) : T

Indexzugriff

append(y : T) : Sequence(T)

Hinten anfügen

prepend(y : T) : Sequence(T)

Vorne anfügen

insertAt(i : Integer, y : T) : Sequence(T)

An i einfügen

Achtung:
USE fügt
nach i ein

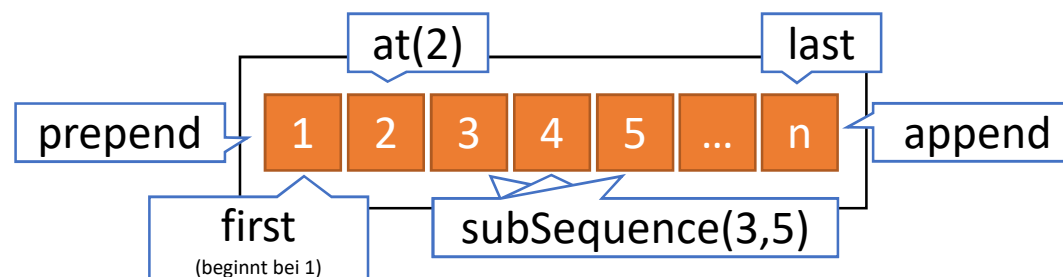
first() : T $\equiv S \rightarrow \text{at}(1)$

Erstes Element

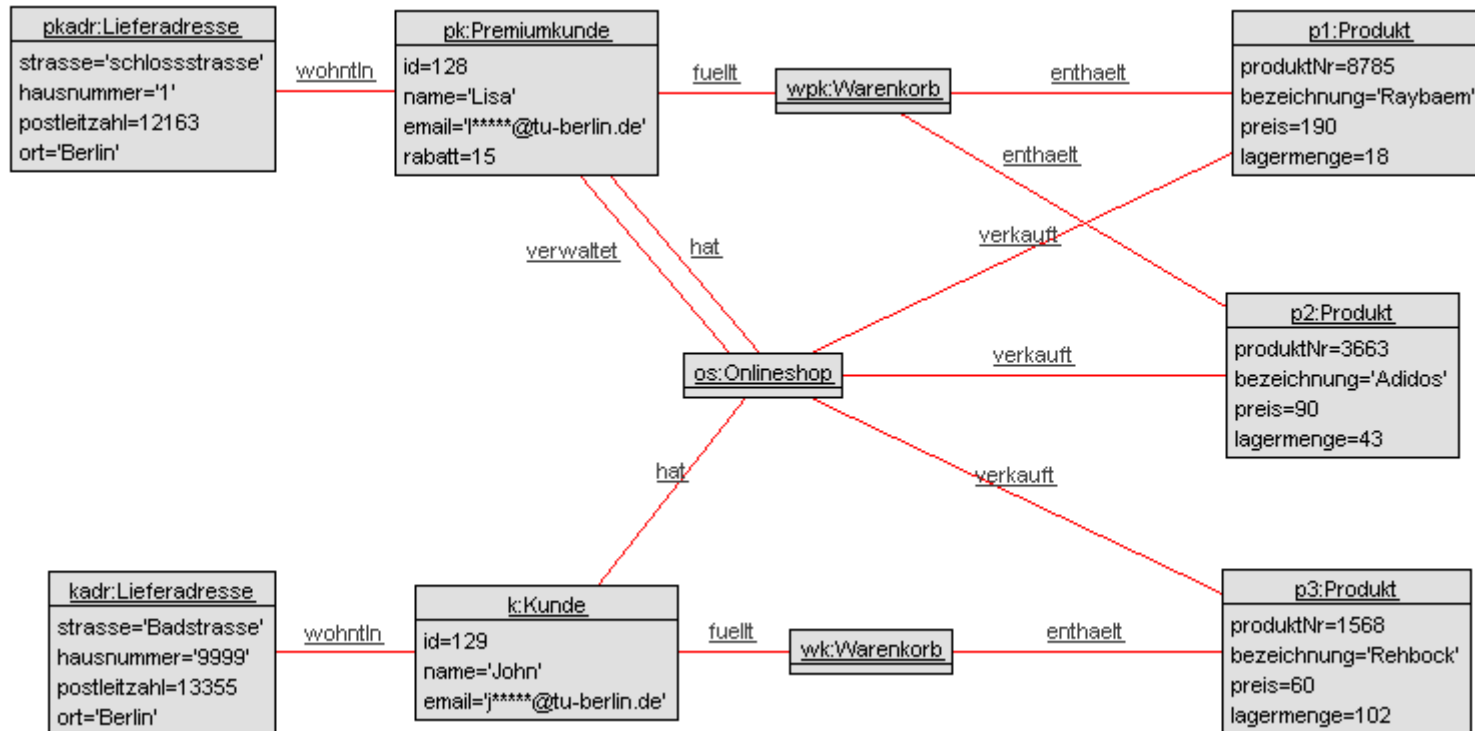
last() : T $\equiv S \rightarrow \text{at}(S \rightarrow \text{size}())$

Letztes Element

subSequence(y : Integer, z : Integer) : Sequence(T) Ausschnitt

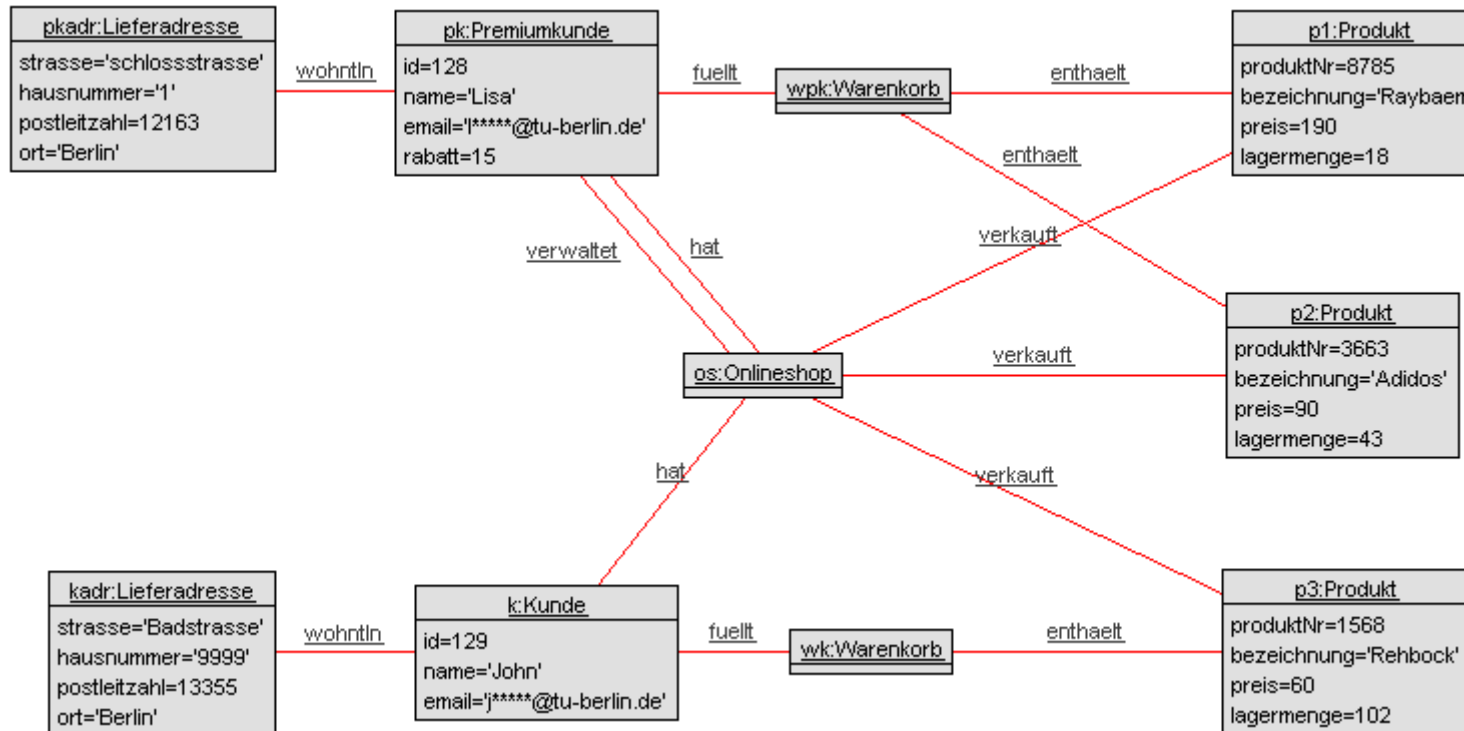


Beispiele



Ist von jedem Produkt min. eins in den Warenkörben von Lisa oder John?

Beispiele



$k.\text{warenkorb}.\text{produkt} \rightarrow \text{union}(pk.\text{warenkorb}.\text{produkt}) = os.\text{produkt}$

$k.\text{warenkorb}.\text{produkt} \rightarrow \text{union}(pk.\text{warenkorb}.\text{produkt}) : \text{Set}(\text{Produkt}) = \text{Set}\{p1, p2, p3\} = os.\text{produkt} : \text{Set}(\text{Produkt})$

Umwandlung von Collections

Zwischen den verschiedenen Collections kann **konvertiert** werden

- Dabei gehen eventuell doppelte Elemente **verloren** oder werden **umsortiert***

Sequence{6,5,5,6}-> asSet()	≡ Set{5,6}	Reihenfolge undefiniert & reduziert
Sequence{6,5,5,6}-> asBag()	≡ Bag{5,5,6,6}	Reihenfolge undef.
Sequence{6,5,5,6}-> asOrderedSet()	≡ OrderedSet{6,5}	reduziert

Mengen von Mengen können in einfache Mengen umgeformt werden

flatten() : Set(T)

Set{Set{1,2},Set{3,4},Set{5,6}}->**flatten()** ≡ Set{1,2,3,4,5,6}

* Die Reihenfolge ist in Sets nicht spezifiziert, aber abhängig vom Tool meist deterministisch. Wie hier zu sehen, sortiert USE die Werte eines Set(Integer) vor der Ausgabe

p1

collectNested?

pfeffer; 11.11.2016

Auswahl

Aus Collections können gewünschte Werte **gefiltert** werden

$c \rightarrow \text{select}(x : T \mid P(x))$	$\{x \in C \mid P(x)\}$
$c \rightarrow \text{reject}(x : T \mid P(x))$	$\{x \in C \mid \neg P(x)\}$

Beispiele

$\text{Set}\{1,2,3\} \rightarrow \text{select}(i : \text{Integer} \mid i > 1) \equiv \text{Set}\{2,3\} : \text{Set}(\text{Integer})$

$\text{Set}\{1,2,3\} \rightarrow \text{reject}(i : \text{Integer} \mid i \leq 1) \equiv \text{Set}\{2,3\} : \text{Set}(\text{Integer})$

Pseudo-Java

```
T[] c;  
T[] select() {  
    T[] result;  
    for (T x : c) {  
        if (P(x)) result.add(x);  
    }  
    return result;  
}
```


Sammlung

Auf Werte in Collections können auch Ausdrücke **angewandt** werden

- Dabei kann sich der **Typ** der resultierenden Collection ändern!

$c \rightarrow \text{collect}(x : T \mid E(x))$

$\{e \mid \exists x \in C. e = E(x)\}$

collect gibt
immer ein
Bag zurück!

Beispiele

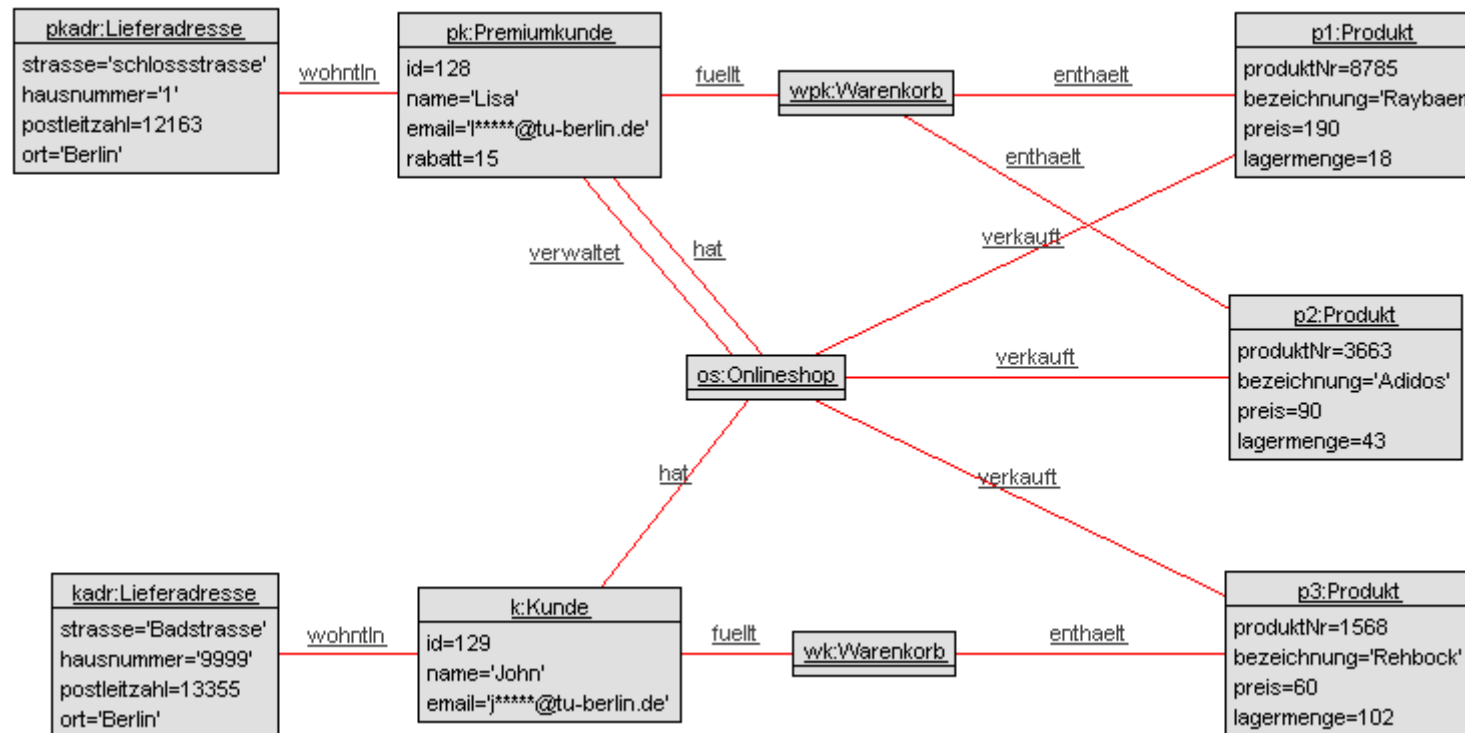
$\text{Set}\{1,2,3\} \rightarrow \text{collect}(i : \text{Integer} \mid i+1) \equiv \text{Bag}\{2,3,4\} : \text{Bag}(\text{Integer})$

$\text{Set}\{1,2,3\} \rightarrow \text{collect}(i : \text{Integer} \mid i.toString()) \equiv \text{Bag}\{'1','2','3'\} : \text{Bag}(\text{String})$

Pseudo-Java

```
T[] C;  
T2[] collect() {  
    T2[] result;  
    for (T x : C) {  
        T2 e = E(x); result.add(e);  
    }  
    return result;  
}
```

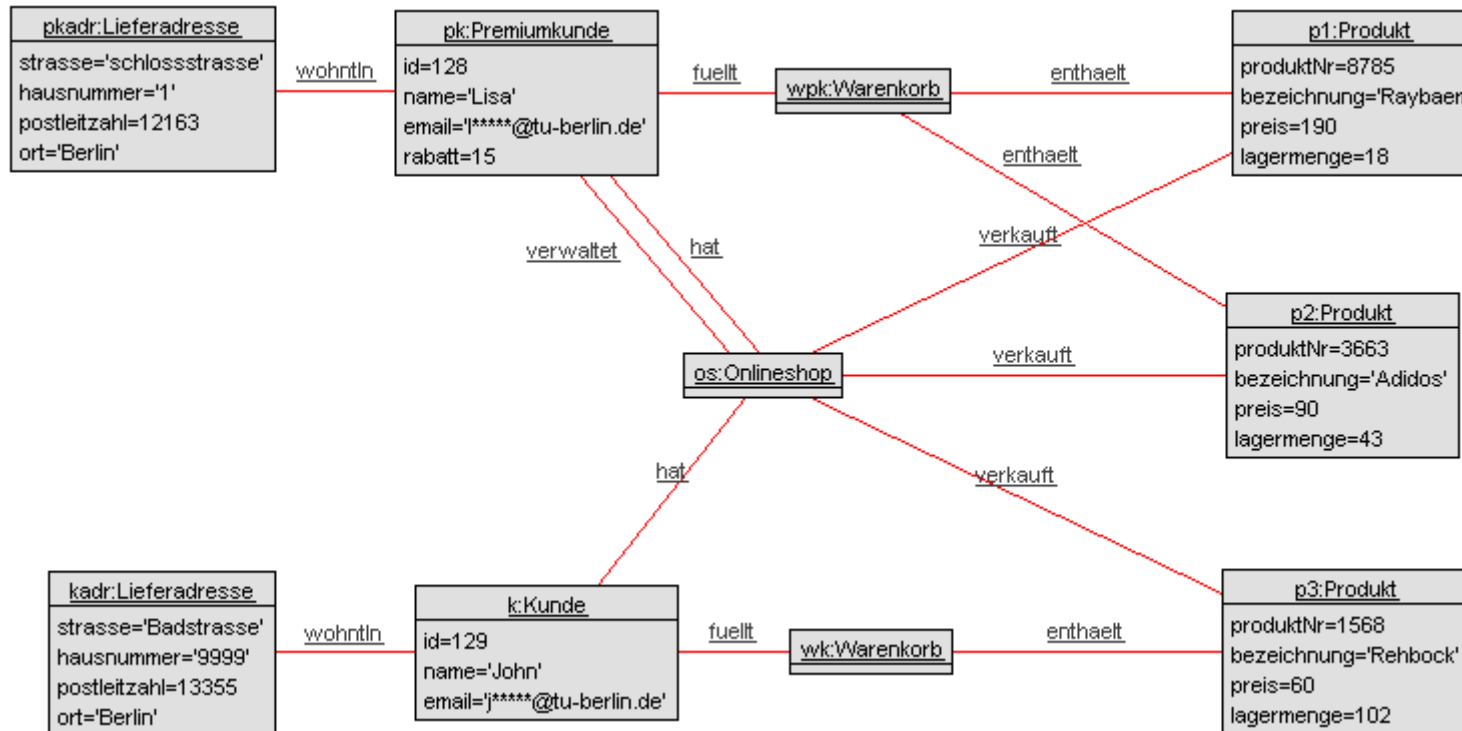
Beispiele



Von welchen Produkten sind mehr als 30 Stück auf Lager?

Welche Produktnummern sind im System vergeben?

Beispiele



`os.produkt->select(p : Produkt | p.lagermenge > 30) ≡ Set{p2,p3} : Set(Produkt)`
`os.produkt->collect(p : Produkt | p.produktNr) ≡ Bag{1568,3663,8785} : Bag(Integer)`

Iteration

Werte in Collections können beliebig **zusammengefasst** werden

- Viele vordefinierte Collection-Operationen sind über **iterate** definiert

$c \rightarrow \text{iterate}(x : T; \text{acc} : T2 = \text{startwert} \mid E(\text{acc}, x))$

Beispiele

$\text{Set}\{1,2,3\} \rightarrow \text{iterate}(x : \text{Integer}; \text{acc} : \text{Integer} = 0 \mid \text{acc} + x) \equiv 6 : \text{Integer}$

$\text{Set}\{1,2,3\} \rightarrow \text{iterate}(x : \text{Integer}; \text{acc} : \text{String} = "" \mid \text{acc} + x.toString()) \equiv '321' : \text{String}$

Pseudo-Java

```
T[] c;  
T2 iterate(T2 startwert) {  
    T2 acc = startwert;  
    for (T x : c) {  
        E(acc, x);  
    }  
    return acc;  
}
```

Set hat keine
Reihenfolge

Quantoren

Durch Quantoren werden **Bedingungen auf Collections** übertragen

$c \rightarrow \text{forAll}(c : T \mid P(c)) \quad \forall c \in C. P(c)$

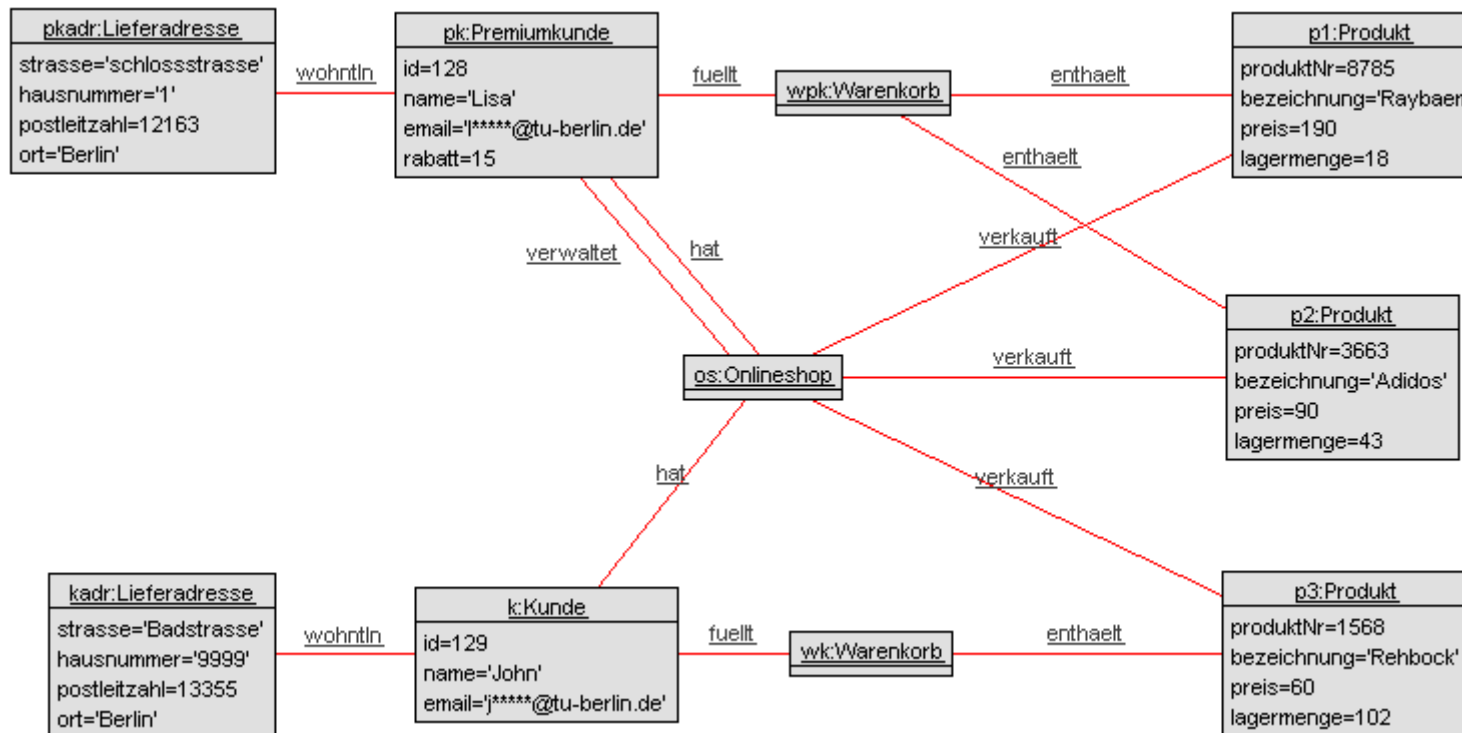
$c \rightarrow \text{exists}(c : T \mid P(c)) \quad \exists c \in C. P(c)$

Damit lässt sich z.B. der **Teilmenge**operator spezifizieren

$c \rightarrow \text{includesAll}(cy : \text{Col}(T)) : \text{Boolean} \quad \equiv \quad cy \rightarrow \text{forAll}(e \mid c \rightarrow \text{includes}(e))$

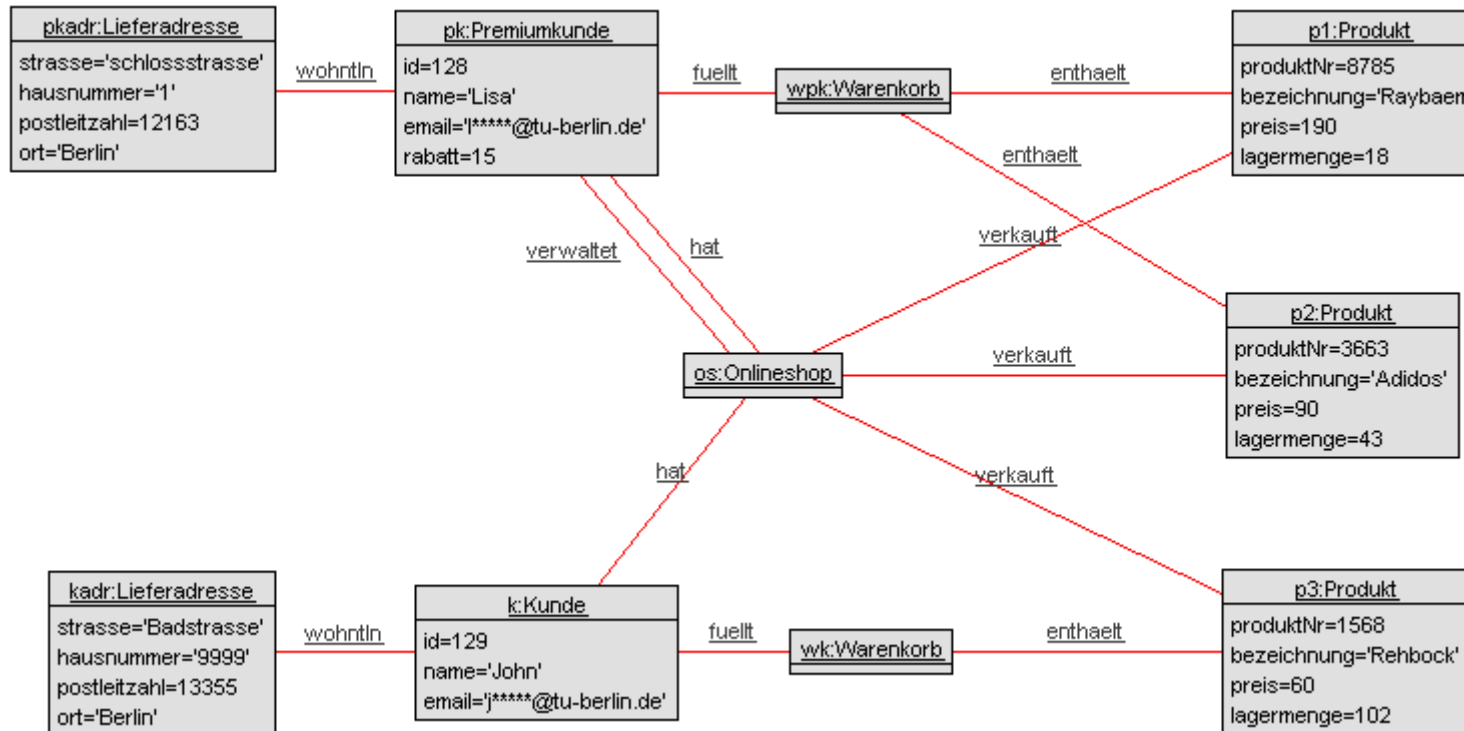
$c \rightarrow \text{excludesAll}(cy : \text{Col}(T)) : \text{Boolean} \quad \equiv \quad cy \rightarrow \text{forAll}(e \mid c \rightarrow \text{excludes}(e))$

Beispiele



Wieviel kostet Lisas Warenkorb zur Zeit?
Sind alle Produkte darin günstiger als 180€?

Beispiele

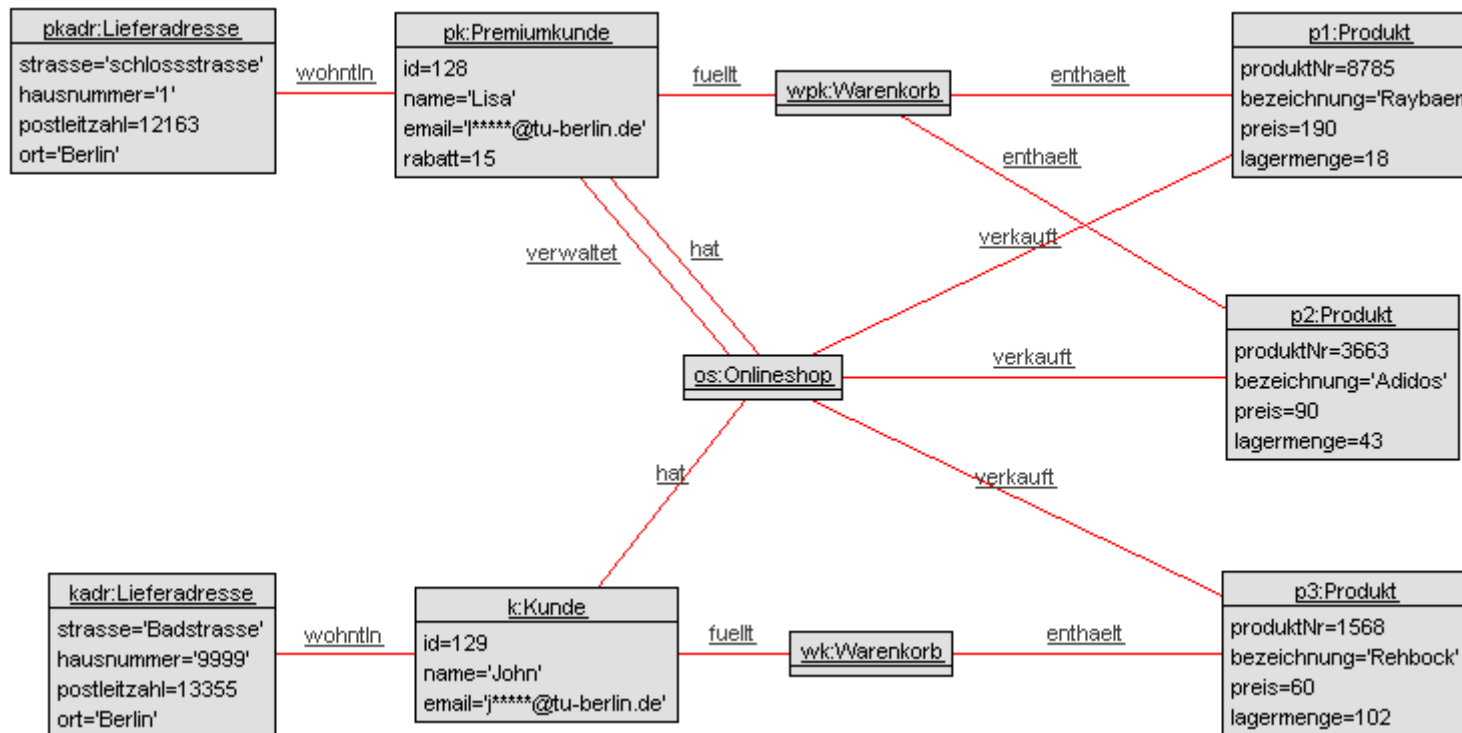


wpk.produkt->iterate(p : Produkt ; acc : Integer = 0 | acc + p.preis) \equiv 280 : Integer

wpk.produkt->forAll(p : Produkt | p.preis < 180) \equiv false : Boolean

Stimmt das?

Beispiele



`wpk.produkt->iterate(p : Produkt; acc : Integer = 0 | acc + p.preis) * (100-pk.rabatt)/100 ≡ 238.0 : Real`

`wpk.produkt->forAll(p : Produkt | p.preis * (100-pk.rabatt)/100 < 180) ≡ true : Boolean`

Tupel

Durch Tupel können Werte **strukturiert** werden

- Tupel sind **keine Collections**, eher temporäre Klassen/Objekte

Tuple{[Bezeichner [: Typ]] = [Wert] [, ...]}

Auf Werte in Tupeln wird **analog zu Werten in Objekten** zugegriffen

Beispiele

Tuple{sensor='Temp',wert=12} : **Tuple**(sensor:String,wert:Integer)

Tuple{**sensor**='Temp',wert=12}.**sensor** ≡ 'Temp' : String

Tuple{sensor='Temp',**wert**=12}.**wert** ≡ 12 : Integer

Closure

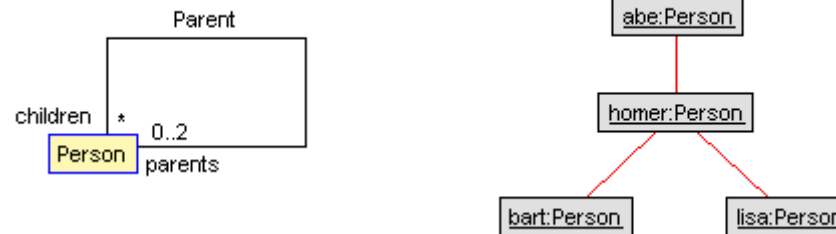
Die Closure-Operation bildet die transitive Hülle einer Beziehung, z.B. einer Assoziation oder Generalisierung

$C \rightarrow \text{closure}(c : T \mid P(c))$

- Akkumulation der Ergebnisse der rekursiven Anwendung von $P(c)$ auf alle Elemente der Menge
- Terminiert wenn sich die Menge nicht mehr ändert
- Hinzugefügt 2011 in Version 2.3

Beispiele

Nützlich zur Akkumulation von rekursiven Daten



$\text{bart.parents} \rightarrow \text{closure}(\text{parents}) \equiv \text{Set}\{\text{homer}, \text{abe}\} : \text{Set}(\text{Person})$
 $\text{Set}\{\text{abe}\} \rightarrow \text{closure}(\text{children}) \equiv \text{Set}\{\text{abe}, \text{bart}, \text{homer}, \text{lisa}\} : \text{Set}(\text{Person})$

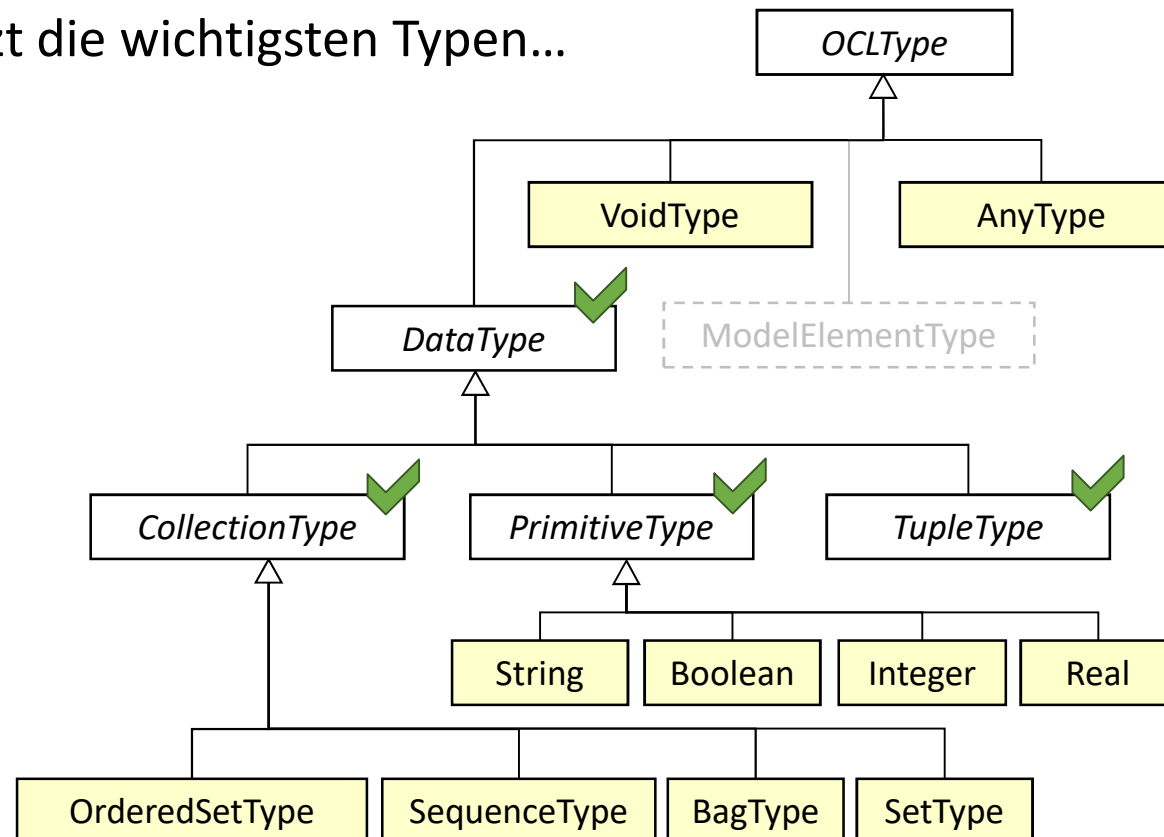
Die Closure kann auch für allgemeine Schleifen verwendet werden:

$\text{Set}\{1\} \rightarrow \text{closure}(x \mid \text{if } x < 5 \text{ then } x+1 \text{ else } x \text{ endif}) \equiv \text{Set}\{1, 2, 3, 4, 5\} : \text{Set}(\text{Integer})$

Zum Knobeln: Wie berechne ich die Fakultät der ersten n Zahlen mit closure und Tupeln?

OCL Typhierarchie

Wir kennen jetzt die wichtigsten Typen...



[OCL, A definitive guide, Jordi Cabot](#)

AnyType als Supertyp

Alle Objekte in OCL erben von **OclAny**

- Ähnlich dem generellen java.lang.Object in Java

Dadurch können Collections **verschiedene Typen** enthalten

Set{'a',1.0,true} : Set(**OclAny**)

Werte können auch explizit **gecastet** werden

obj.**oclAsType**(t : OclType) : T

Gleichheit auf OclAny überprüft die **Referenz** (wie in Java)

=(obj2 : OclAny) : Boolean

Generalisierung/Spezialisierung

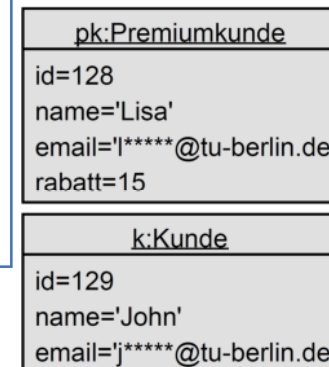
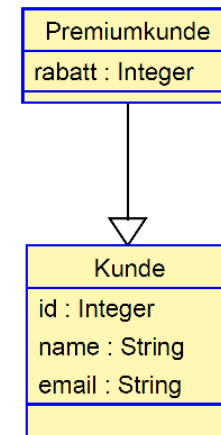
Typen lassen sich per `ocllsTypeOf` oder `ocllsKindOf` **bestimmen...**

`pk.ocllsTypeOf(Kunde)` \equiv **false**
`pk.ocllsTypeOf(Premiumkunde)` \equiv **true**
`k.ocllsTypeOf(Kunde)` \equiv **true**
`k.ocllsTypeOf(Premiumkunde)` \equiv **false**

Type
Überprüft den
tatsächlichen Typ
des Objekts

`pk.ocllsKindOf(Kunde)` \equiv **true**
`pk.ocllsKindOf(Premiumkunde)` \equiv **true**
`k.ocllsKindOf(Kunde)` \equiv **true**
`k.ocllsKindOf(Premiumkunde)` \equiv **false**

Kind
Überprüft ob das
Objekt dem Typ
entspricht
(Typ oder Subtyp)



...was ist der Unterschied?

VoidType als Nullwert

Es gibt einen speziellen Nullwert vom Typ **OclVoid**

- Kann als **null** oder **Undefined** verwendet werden
- Kann mit allen anderen Objekten **verglichen** werden

null : OclVoid

1/0 \equiv **Undefined** : OclVoid

1/0 = **Undefined** \equiv true : Boolean

1/0 = **null** \equiv true : Boolean

(1/0).oclIsUndefined \equiv true : Boolean

Sequence{1,2,**null**,4,5} : Sequence(Integer)

Sequence{1,2,**null**,4,5}->size() \equiv 5 : Integer

Dreiwertige Logik in OCL

Durch den Nullwert ergibt sich in OCL eine **dreiwertige Logik**

b	NOT b	AND	false	null	true	OR	false	null	true	XOR	false	null	true
false	true	false	false	false	false	false	false	null	true	false	false	null	true
null	null	null	false	null	null	null	null	null	true	null	null	null	null
true	false	true	false	null	true	true	true	true	true	true	true	null	false

IMPLIES	false	null	true
false	true	true	true
null	null	null	true
true	false	null	true

=	false	null	true
false	true	false	false
null	false	true	false
true	false	false	true

<>	false	null	true
false	false	true	true
null	true	false	true
true	true	true	false

Abkürzungen

OCCL kann in vielen Fällen den **Typ ableiten**

Set{-1,2,-3}->select(i : Integer | i.abs() > 0)

Set{-1,2,-3}->select(i | i.abs() > 0)

Set{-1,2,-3}->select(abs() > 0)

OCCL „weiß“, dass
i ein Integer sein
muss

Und dass die Werte als
Argument dienen

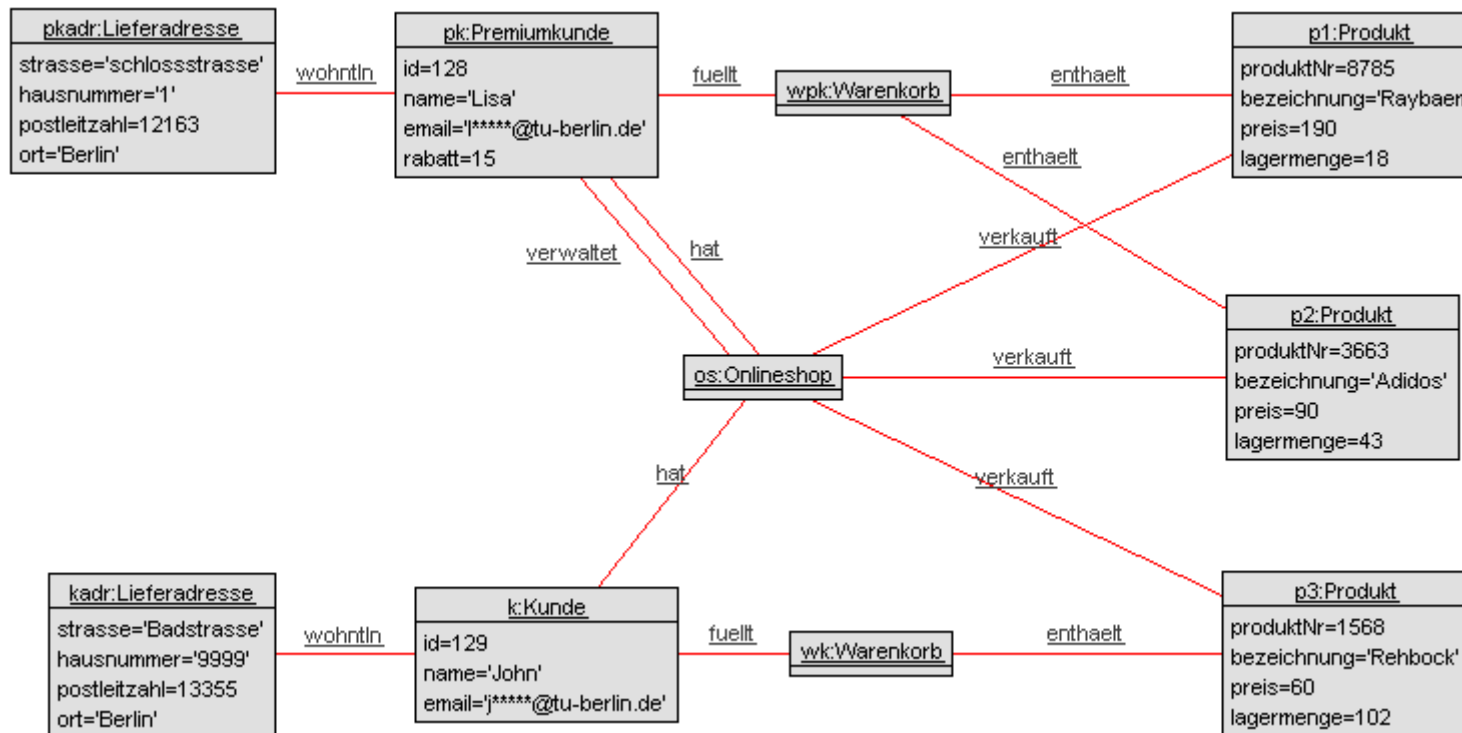
Aus praktischen Gründen gibt es auch eine Kurzschreibweise für **collect**

- Wird immer bei Anwendung der **Punknotation auf Collections** angenommen

Set{-1,2,-3}->collect(i | i.abs()) \equiv Bag{1,2,3} : Bag(Integer)

Set{-1,2,-3}.abs() \equiv Bag{1,2,3} : Bag(Integer)

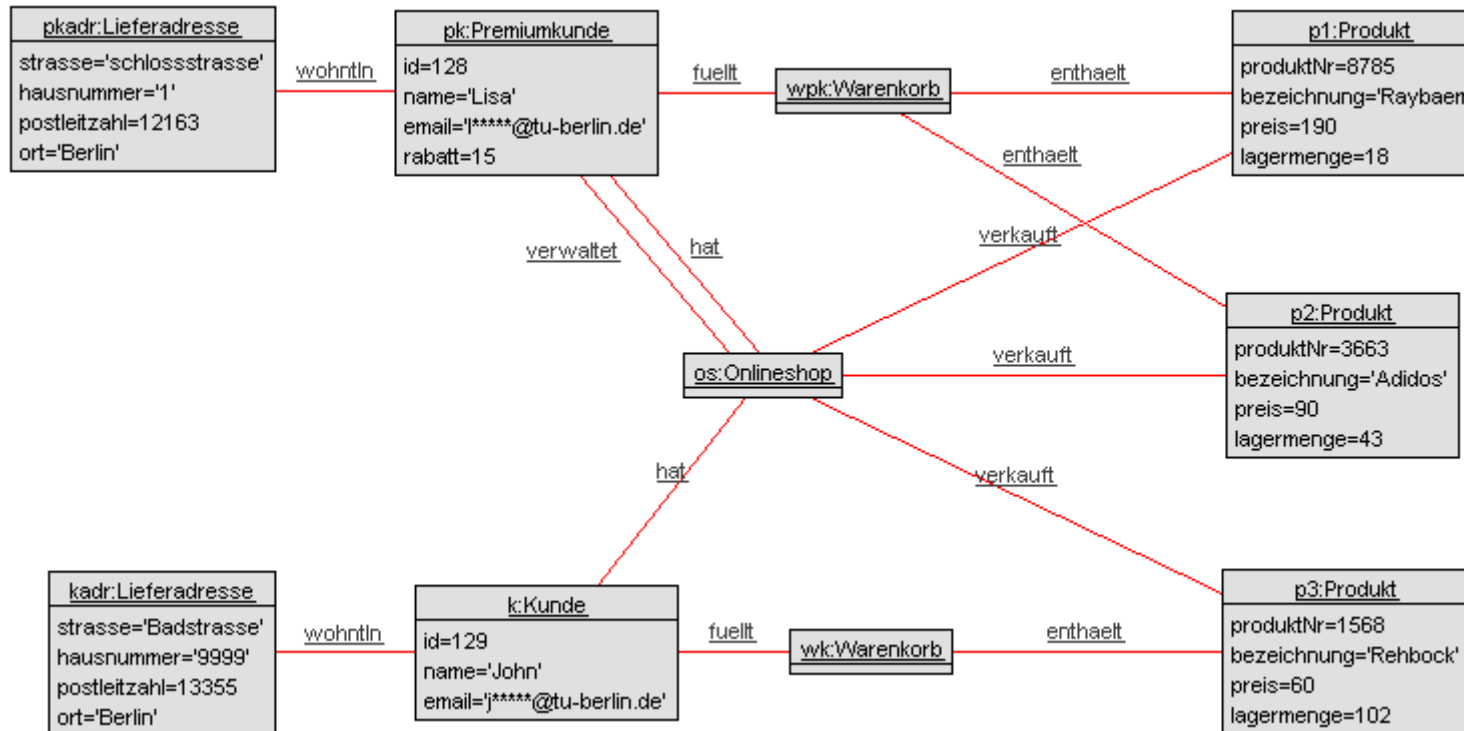
Beispiele



Wie viele Produkte sind zur Zeit im Lager?

Wer kauft Produkte über 80€?

Beispiele



os.produkt.lagermenge->iterate(m; s : Integer = 0 | s + m)

≡ 163 : Integer

os.produkt->select(preis > 80).warenkorb.kunde.name->asSet()

≡ Set{'Lisa'}

collect

collect

collect

Collect und CollectNested

os.kunde.warenkorb->collect(produkt)

Eigentlich müsste das eine Menge von Mengen sein...

os.kunde.warenkorb

$\equiv \text{Bag}\{\text{wk}, \text{wpk}\} : \text{Bag}(\text{Warenkorb})$

$\text{Bag}\{\text{wk}, \text{wpk}\} \rightarrow \text{collect}(\text{produkt})$

$\equiv \text{Bag}\{p1, p2, p3\} : \text{Bag}(\text{Produkt})$

...ist es aber nicht.

collect beinhaltet immer auch **flatten** - collectNested nicht:

$\text{Bag}\{\text{wk}, \text{wpk}\} \rightarrow \text{collectNested}(\text{produkt}) \equiv \text{Bag}\{\text{Set}\{p3\}, \text{Set}\{p1, p2\}\} : \text{Bag}(\text{Set}(\text{Produkt}))$

$\text{Bag}\{\text{wk}, \text{wpk}\} \rightarrow \text{collect}(\text{produkt}) = \text{Bag}\{\text{wk}, \text{wpk}\} \rightarrow \text{collectNested}(\text{produkt}) \rightarrow \text{flatten}()$

Inhalt

Formale Spezifikation

- Grundlagen
- Object Constraint Language (OCL)
- **Invarianten**
- Contracts
- Weiteres

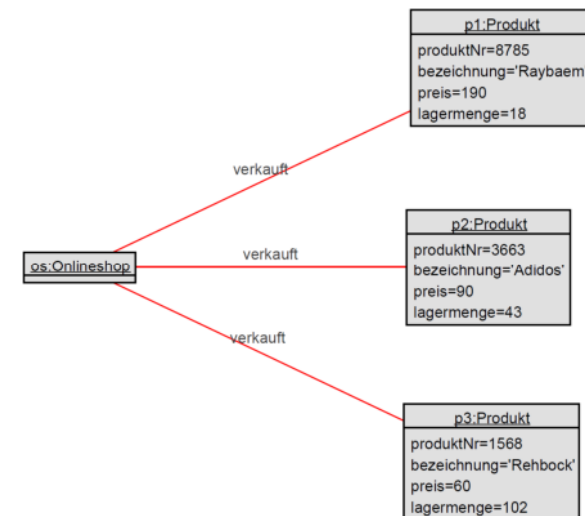
Motivation

Bisher haben wir mit OCL **einen spezifischen Zustand** überprüft

Haben zur Zeit alle Produkte einen Preis > 0?

`os.produkt->forAll(p | p.preis > 0) \equiv true`

Aber eigentlich wollen wir ja generelle Anforderungen spezifizieren...



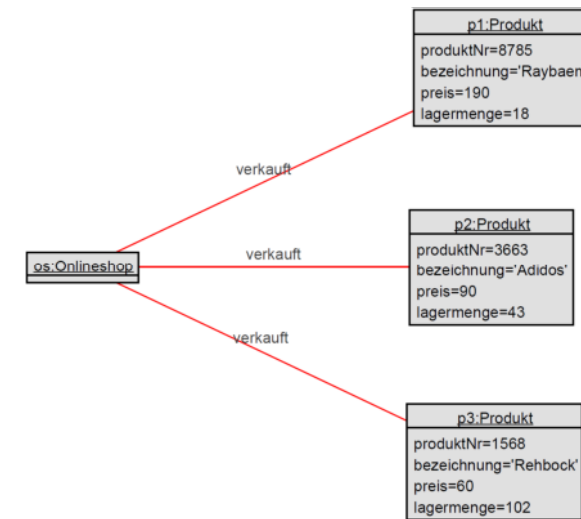
Wie können wir Aussagen über alle Instanzen machen,
ohne über einen Controller zu navigieren?

(im Beispiel also ohne os zu verwenden)

allInstances

Durch allInstances kann direkt auf **alle Instanzen** einer Klasse zugegriffen werden

[Klasse].allInstances() : Set(Klasse)



Wie können wir die Aussage ohne Controller-Instanz formulieren?

mit	<code>os.produkt</code>	<code>->forAll(p p.preis > 0)</code>	<code>≡ true</code>
ohne	<code>Produkt.allInstances()</code>	<code>->forAll(p p.preis > 0)</code>	<code>≡ true</code>

Invarianten

Mit Invarianten können **Modellen** Bedingungen hinzugefügt werden

- Die Bedingung gilt dann für die gesamte Lebenszeit **aller passenden Objekte**

Da für die Invarianten **keine konkreten Instanzen existieren**, beginnt die Navigation direkt bei einer Klasse

context [[Bezeichner :] Klasse] **inv** [Name]: [Boolescher OCL Ausdruck]

Wie können wir die Aussage als Invariante formulieren?

Übersicht

Für alle Produkte gilt: sie haben einen positiven Preis.

Als **Aussage** über einen **konkreten** Zustand der Onlineshopinstanz

os.produkt $\rightarrow \text{forAll}(p \mid p.\text{preis} > 0) \equiv \text{true}$

Als **Aussage** über einen **konkreten** Zustand

Produkt.allInstances() $\rightarrow \text{forAll}(p \mid p.\text{preis} > 0) \equiv \text{true}$

Als **Invariante** für **alle** Zustände

context p : Produkt inv: (p.preis > 0)

Alternative

Für alle Produkte gilt: sie haben einen positiven Preis.

Als **Aussage** über einen **konkreten** Zustand der Onlineshopinstanz

os.produkt $\rightarrow \text{forAll}(p \mid p.\text{preis} > 0) \equiv \text{true}$

Als **Aussage** über einen **konkreten** Zustand

Produkt.allInstances() $\rightarrow \text{forAll}(p \mid p.\text{preis} > 0) \equiv \text{true}$

Als **Invariante** für **alle** Zustände

context Produkt inv: $(\text{self}.\text{preis} > 0)$ Ohne Bezeichner gilt self

Alternative

Für alle Produkte gilt: sie haben einen positiven Preis.

Als **Aussage** über einen **konkreten** Zustand der Onlineshopinstanz

os.produkt $\rightarrow \text{forAll}(p \mid p.\text{preis} > 0) \equiv \text{true}$

Als **Aussage** über einen **konkreten** Zustand

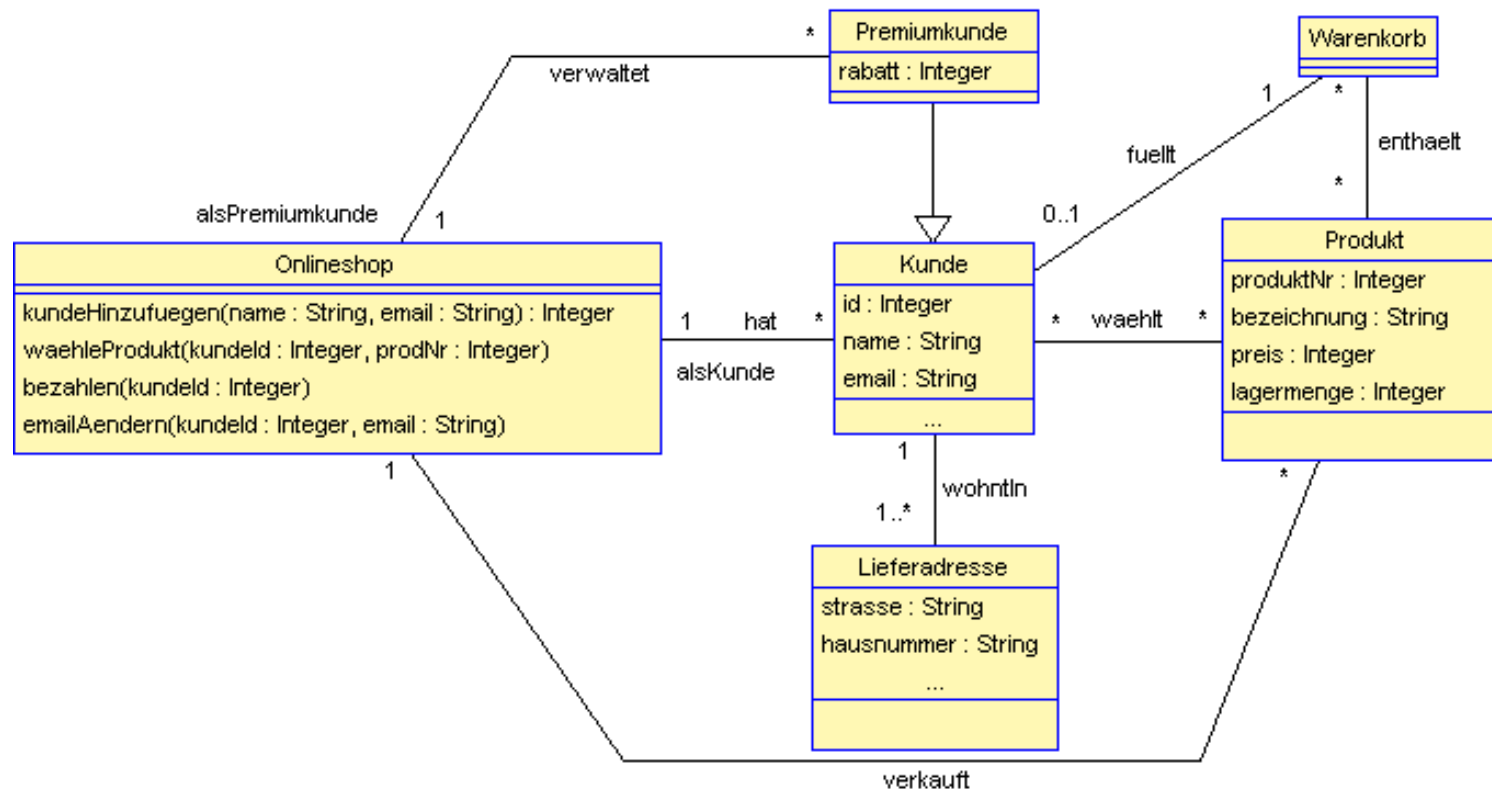
Produkt.allInstances() $\rightarrow \text{forAll}(p \mid p.\text{preis} > 0) \equiv \text{true}$

Als **Invariante** für **alle** Zustände

context Produkt inv: (preis > 0)

oder Kurzschreibweise

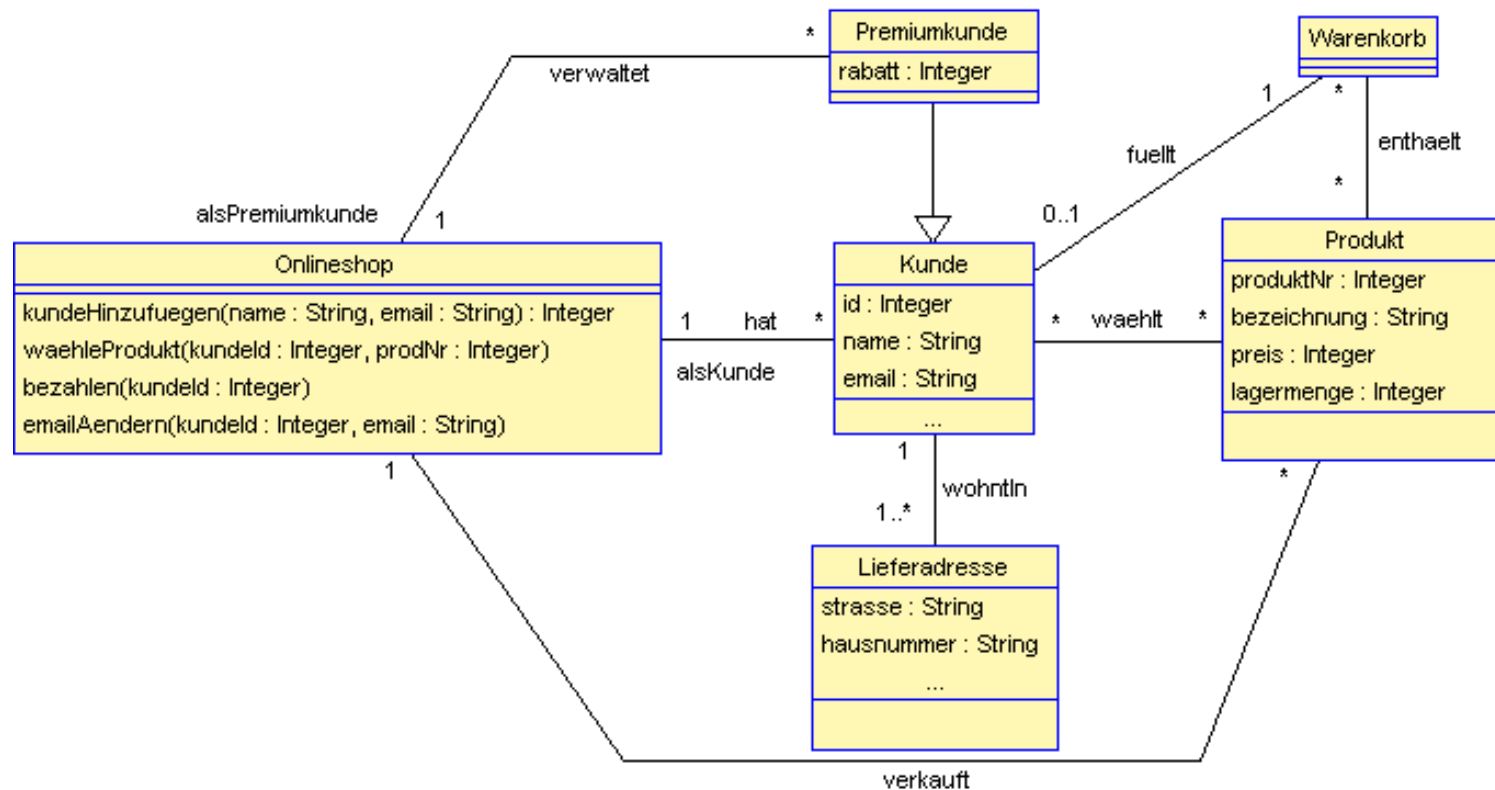
Weitere Beispiele



Name und Email von Kunden sind nicht leer

Der Rabatt von Premiumkunden liegt zwischen 0 und einschließlich 50

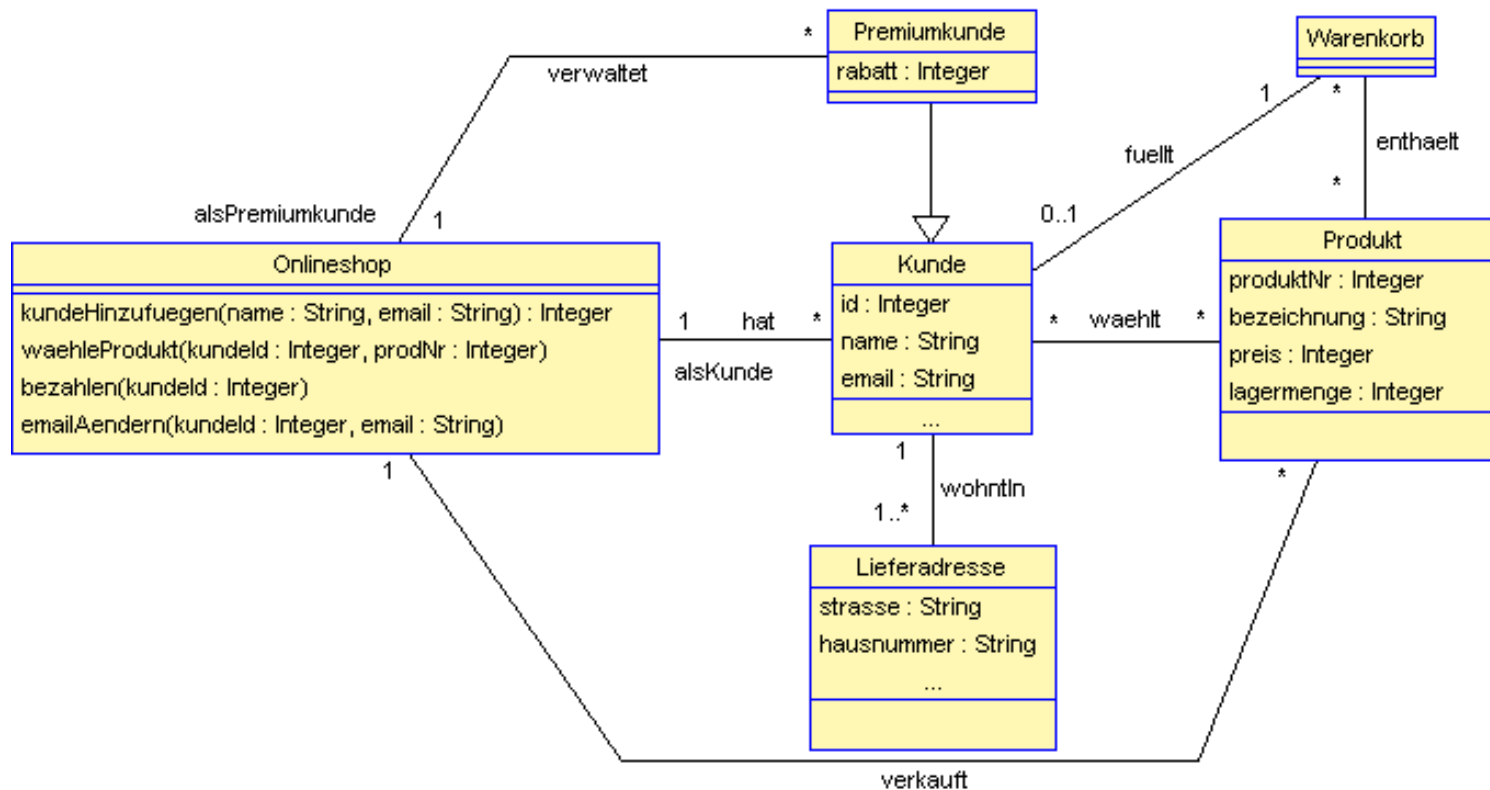
Weitere Beispiele



context Kunde **inv:** self.name <> “ and self.email <> “

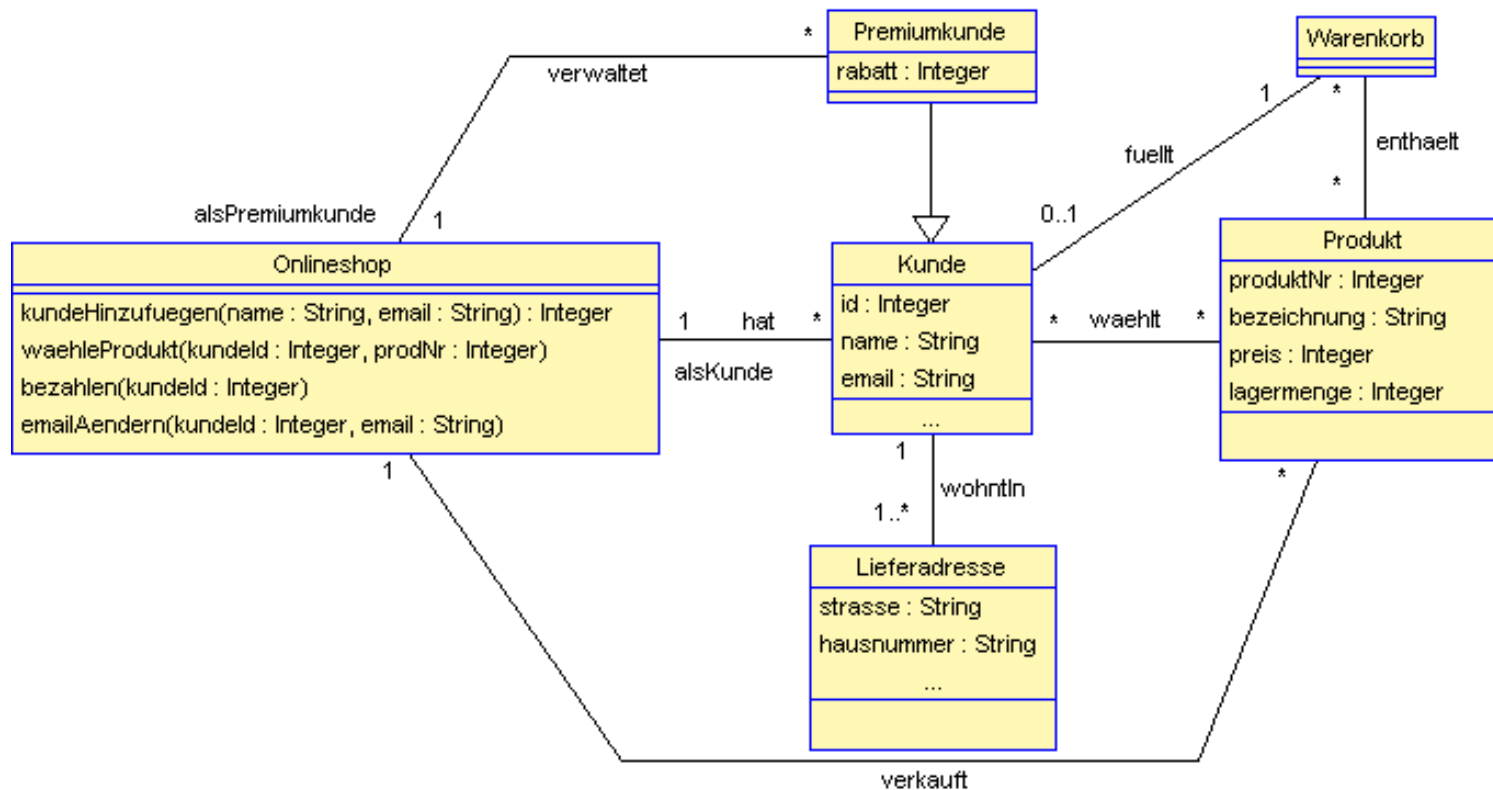
context Premiumkunde **inv:** rabatt > 0 and rabatt <= 50

Weitere Beispiele



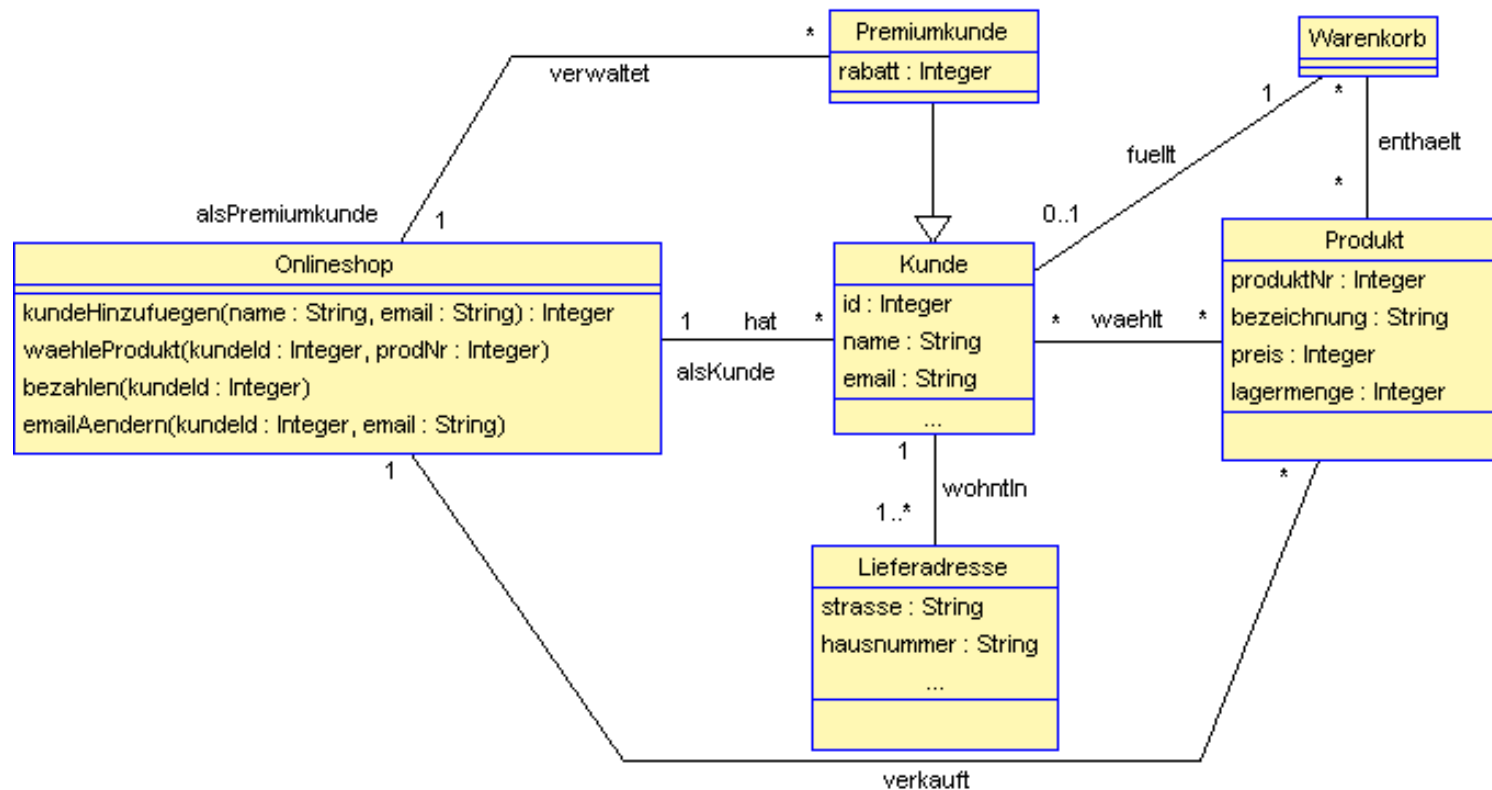
Die IDs der Kunden sind eindeutig

Weitere Beispiele



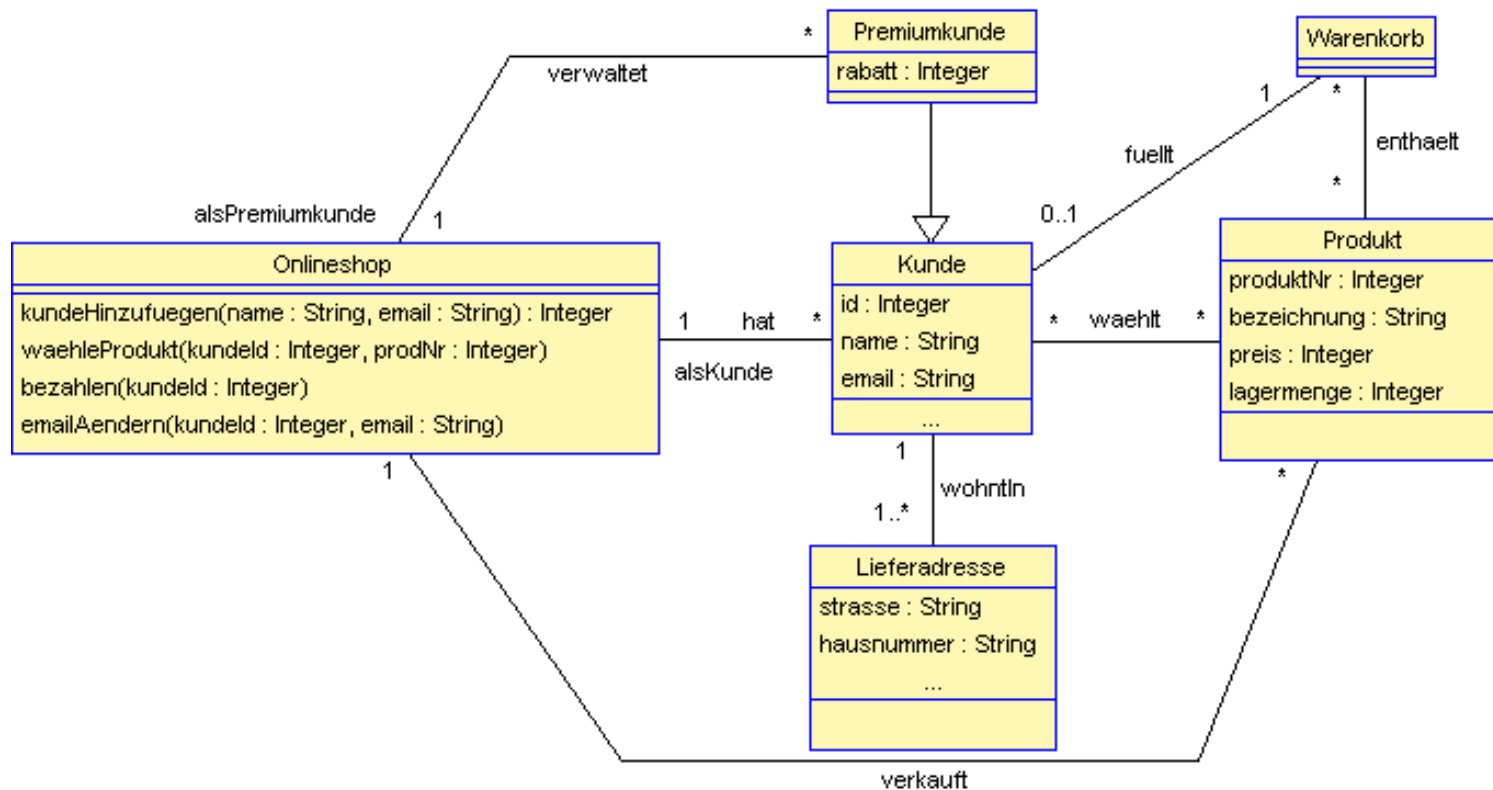
context Kunde **inv:** Kunde.allInstances().id->count(self.id) = 1

Weitere Beispiele



Es sind maximal so viele Produkte in Warenkörben, wie im Lager sind

Weitere Beispiele



context Produkt **inv:** Warenkorb.allInstances().produkt->count(self) <= lagermenge

Inhalt

Formale Spezifikation

- Grundlagen
- Object Constraint Language (OCL)
- Invarianten
- **Contracts**
- Weiteres

Motivation

Mit **Invarianten** haben wir die **Struktur** genauer spezifiziert...

Wie Nutzer die Software verwenden



V. Pfosten

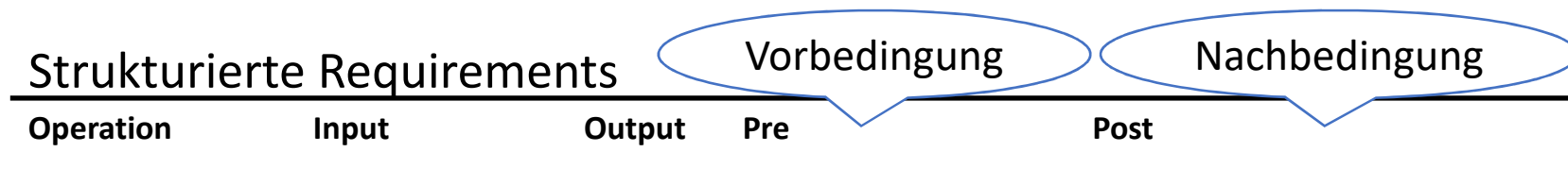
★☆☆☆☆ Schlechtes Fahrrad. Stoppt ganz plötzlich.
Rezension aus Deutschland vom 17. Februar 2020
Verifizierter Kauf

...wie geht das mit dem **Verhalten**?

Design By Contract

Contracts stellen den reibungslosen Ablauf zwischen Funktionen sicher

- Die **Vorbedingung** muss vom Aufrufenden erfüllt werden
- Die **Nachbedingung** wird von der Funktion sichergestellt
- In JAVA als **Assertions** vorhanden



Eingeführt von Bertrand Meyer für die Programmiersprache Eiffel

“Correctness is clearly the prime quality. If a system does not do what it is supposed to do, then everything else about it matters little.”

Bertrand Meyer

Contracts in OCL

Contracts können auch direkt in OCL angegeben werden

context [Klasse]::[Operation(Parameter)] : [Rückgabetyt]
pre: [Vorbedingungen]
post: [Nachbedingungen]

Beispiel

Operation	Input	Output	Pre	Post
emailÄndern	Kundeld, Email		Kundeld existiert, Email nicht leer	Email von Kunde mit Kundeld geändert

context Onlineshop::emailAendern(kundeld, email) : void

pre: self.kunde.id->includes(kundeld) and email <> “

post: self.kunde->any(id = kundeld).email = email

Wähle den Kunden mit kundeld

Let und Any

Durch **any** wird **ein** passendes Element aus einer Collection ausgewählt

Set{1,2,3}-> select (i i > 2)	≡ Set {3} : Set(Integer)
Set{1,2,3}-> any (i i > 2)	≡ 3 : Integer

Häufig ist es auch praktisch **Zwischenergebnisse** über **let** zu definieren

let zahlen = Set{1,2,3} in zahlen.any(i i > 2) ≡ 3 : Integer		
Deklaration	Definition	Ausdruck

Beispiel

```
context Onlineshop::emailAendern(kundeld, email) : void
post: let k = self.kunde->any(id = kundeld) in k.email = email
```

Beispiel

Wie lässt sich folgender Contract ausdrücken?

Operation	Input	Output	Pre	Post
wähleProdukt	Kundeld, ProdNr		Kundeld existiert, ProdNr existiert	Kunde mit Kundeld hat Produkt mit ProdNr im Warenkorb

context Onlineshop::waehleProdukt(kundeld, prodNr) : void

pre: self.kunde.id->includes(kundeld)

pre: self.produkt.produktNr->includes(prodNr)

post: **let** k = self.kunde->any(id = kundeld) **in**

let p = self.produkt->any(produktNr = prodNr) **in**

 k.warenkorb.produkt->includes(p)

Kundeld existiert

ProdNr existiert

Kunde mit Kundeld...

Produkt mit ProdNr...

K hat P im Warenkorb

@Pre

Manchmal hängt die Nachbedingung von dem **vorherigen** Zustand ab

- Dafür gibt es in OCL Contracts die **@Pre**-Notation

Beispiele

Durch Ausführung der Operation hat sich an den Produkten nichts geändert

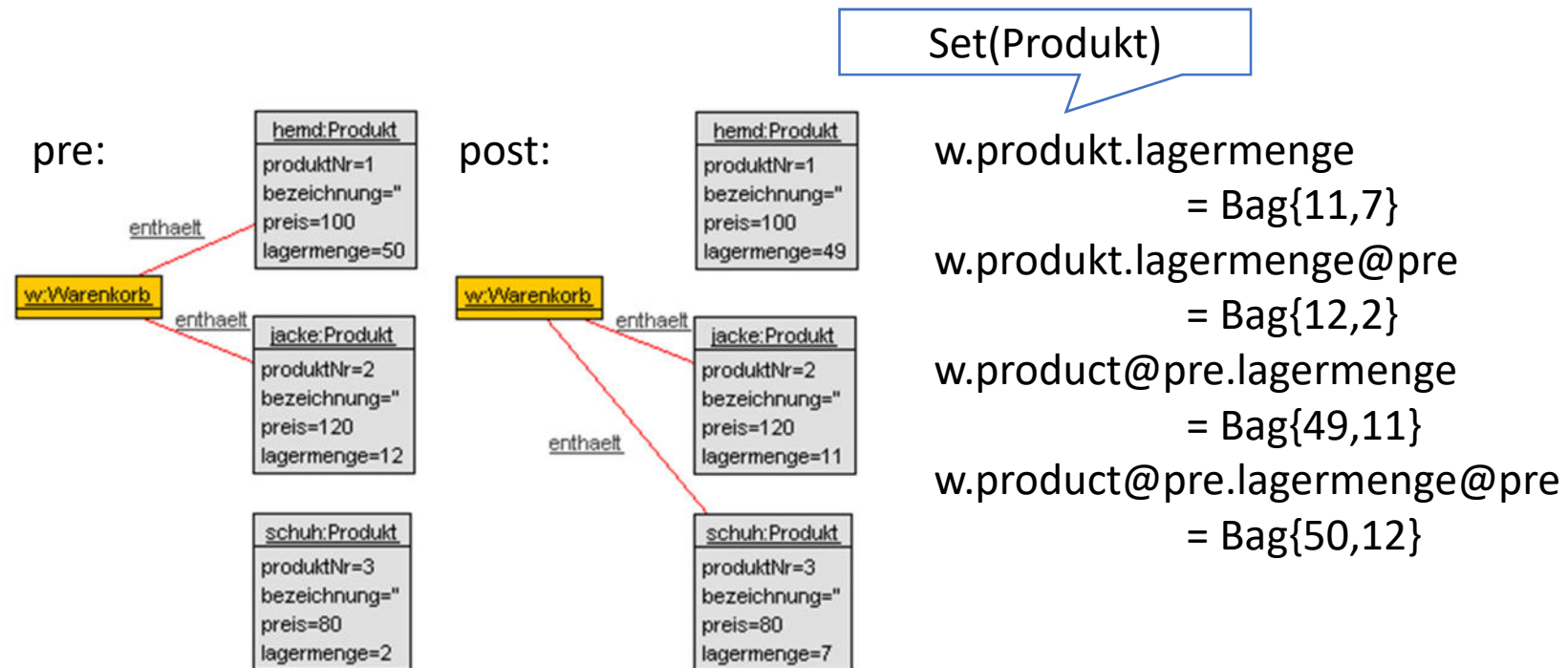
post: Produkt.allInstances() = Produkt.allInstances()**@pre**

Ein neues Produkt p ist dazu gekommen

post: Produkt.allInstances() = Produkt.allInstances()**@pre->including(p)**

@Pre

Achtung: @pre bezieht sich direkt auf das vorangehende Objekt, nicht den gesamten Ausdruck



Beispiel

Wie lässt sich die Nachbedingung des Contracts ausdrücken?

Operation	Input	Output	Pre	Post
bezahlen	Kundeld		Kundeld existiert, Produkte auf Lager	Entsprechend weniger Produkte, Warenkorb leeren

context Onlineshop::bezahlen(kundeld) : void

pre: **let** k = self.kunde->any(id = kundeld) **in**

k <> null **and** k.warenkorb.produkt->forAll(lagermenge > 0)

post: **let** k : Kunde = self.kunde->any(id = kundeld) **in**

let ps : Set(Produkt) = k.warenkorb@pre.produkt@pre **in**

ps->forAll(lagermenge = lagermenge@pre - 1)

and k.warenkorb.produkt->size() = 0

Kunde mit kundeld

Entsprechende Produkte
vor dem bezahlen

Lagermenge verringert

Warenkorb leeren

Result und oclIsNew

Die Verwendung von **result** ermöglicht Aussagen über das **Ergebnis**

context: Kunde::getId() : Integer

post: `result = self.id`

Die Erzeugung **neuer Objekte** kann per **oclIsNew** überprüft werden

context: Onlineshop::produktAnlegen(...) : Produkt

Post: `result.oclIsNew()`

Beispiel

Wie lässt sich folgender Contract ausdrücken?

Operation	Input	Output	Pre	Post
kundeHinzufügen	Name, Email	Id	Email noch nicht im System	Neuer Kunde mit Name, Email und neuer Id

context Onlineshop::kundeHinzufuegen(name, email) : int

pre: self.kunde.email->excludes(email)

post: **let** k : Kunde = self.kunde->any(k | k.email = email) **in**

k <> null **and** k.oclIsNew()

and k.name = name **and** k.id <> null

and self.kunde@pre.id@pre->excludes(k.id)

and result = k.id

Email nicht im System

Kunde mit Mail...

...existiert und ist neu

Hat Name und ID

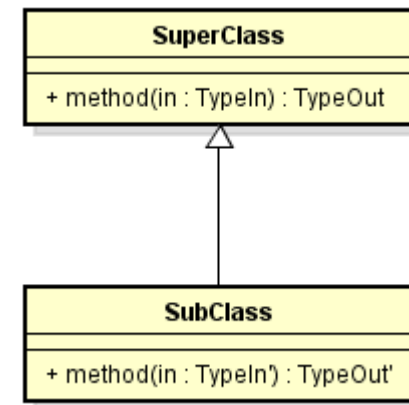
Die ID ist neu und...

...wird zurückgegeben

Contracts und Vererbung

Bei Vererbung von Methoden gilt für ihre Contracts das Substitutionsprinzip:

- Wenn vor der Ausführung einer Methode in der Unterklasse die Vorbedingungen der entsprechenden Methode der Oberklasse gelten, dann muss die Methode der Unterklasse ausführbar sein und anschließend die Nachbedingungen der Oberklasse garantieren.



- Vorbedingungen dürfen nicht verschärft werden
- Nachbedingungen dürfen nicht aufgeweicht werden

Contracts und Vererbung

TypeIn' ist ein Obertyp von TypeIn

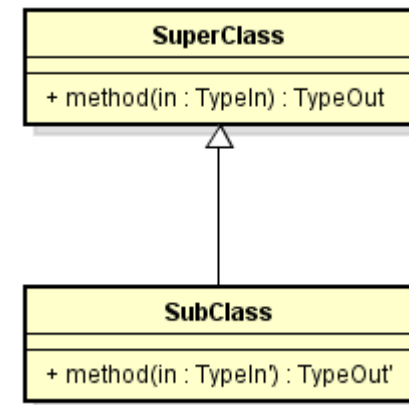
➤ Kontravarianz

TypeOut' ist ein Untertyp von TypeOut

➤ Kovarianz

Dabei gilt:

- Vorbedingungen von *method aus SubClass* werden von Vorbedingungen von *method aus SuperClass* impliziert
- Nachbedingungen von *method aus SubClass* implizieren Nachbedingungen von *method aus SuperClass*
- Diese Vererbungs-Konformität wird *Kontra- und Kovarianz* genannt
- Entsprechend lassen sich auch andere Varianten von Konformität definieren



Inhalt

Formale Spezifikation

- Grundlagen
- Object Constraint Language (OCL)
- Invarianten
- Contracts
- **Weiteres**

OCL als Query Language

Queries ermitteln **zusätzliche** Informationen aus einem Modell

Weitere Werte können abgeleitet werden (*derived value*) ...

context Warenkorb::gesamtSumme: Integer

derive: produkt.preis->sum()

... oder durch einfache Operationen berechnet werden

context Warenkorb::luxusWare(p : Integer) : Set(Integer)

body: self.produkt->select(preis > p).produktNr

Übersetzung von OCL Queries

OCL Queries können z.B. in SQL Queries überführt werden

context Warenkorb::luxusWare(p : Integer) : Set(Integer)

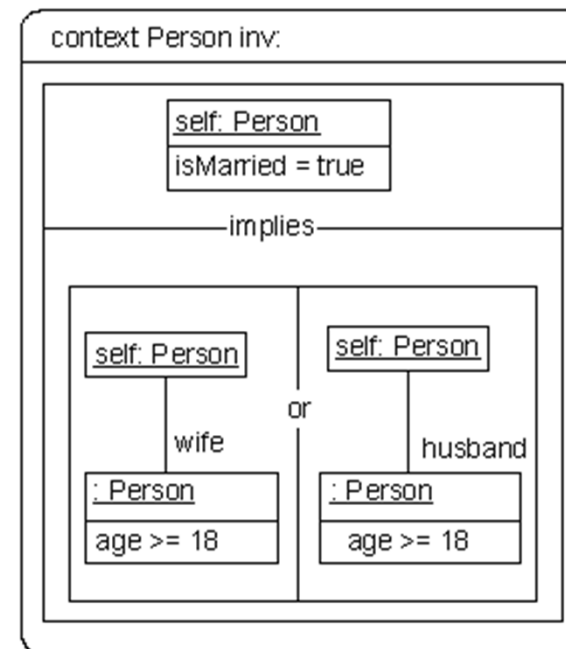
body: self.produkt->select(preis > p).produktNr

SELECT produktNr **FROM** Produkte **WHERE** preis > p;

Visual OCL

Visuelle Darstellung von OCL, um während des Entwurfs nicht die Abstraktionsebene wechseln zu müssen

- Angepasst an die UML Syntax
- Eclipse Plug-In Integration
- Entwickelt an der TU Berlin



<http://www.user.tu-berlin.de/o.runge/tfs/projekte/vocl/index.html>

Hinweise zu den Werkzeugen

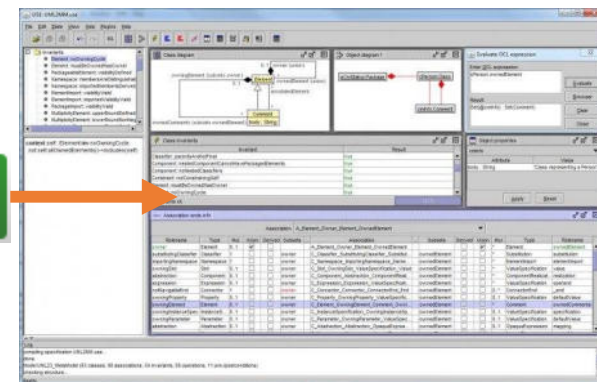
Für die Erstellung der Folien und in den Übungen verwenden wir **USE**

- Alle in diesen Folien **angegebenen Beispiele** lassen sich mit USE ausführen
- Die **verwendeten Modelle** findet Ihr auf ISIS

Software-Empfehlungen

- UML-Modellierung: [Astah Community](#)
- USE OCL der Uni Bremen: [download](#) [documentation](#)
- Eclipse IDE für Java-Beispiele im Kurs: [Eclipse Mars JZEE mit Plugins 3.9.0](#)
 - EclipseFP - Plugin für Haskell-Entwicklung (Eclipse Marketplace)
 - Eclipse Metrics - Plugin für Metriken (Update site: <http://metrics.sourceforge.net/>)
 - Eclipse Emma - Plugin für Code Coverage (Eclipse Marketplace)
- [Haskell Platform](#)
- [SWI-Prolog](#)

Download
use-4.2.0.zip



In der Übung gehen wir **weitere Beispiele** durch

- Bitte **installiert USE** bei Euch, damit Ihr direkt mitmachen könnt
- **Notwendige Modelle** werden ebenfalls über ISIS verteilt

Sprachen lernt man am schnellsten, wenn man sie benutzt

Lernziele, Teil 1

- ☐ Was ist die Object Constraint Language? Warum braucht man sie?
- ☐ Wie kann mit OCL auf Werte von Objekten zugegriffen werden?
- ☐ Welche primitiven Typen kennt OCL? Wie kann man mit ihnen arbeiten?
- ☐ Wie kann in OCL auf Werte verbundener Objekte zugegriffen werden?
- ☐ Wie funktioniert der Zugriff auf mehrere verbundene Objekte?
- ☐ Wie unterscheiden sich die verschiedenen Collections in OCL?
- ☐ Welche Operationen stehen für Set/Bag zur Verfügung? Welche für Sequence/OrderedSet?
- ☐ Was ist bei einem Wechsel der Collection zu beachten?
- ☐ Wie können spezifische Werte aus einer Collection gefiltert werden?
- ☐ Wie können Operationen auf die Elemente einer Collection angewandt werden?
- ☐ Was geschieht dabei jeweils mit dem Typ der Collection?
- ☐ Wie können die Werte in einer Collection zusammengefasst werden?
- ☐ Wie können Bedingungen für alle Elemente einer Collection angegeben werden?
- ☐ Wie lässt sich damit die Teilmengenrelation überprüfen?
- ☐ Was versteht man unter Tupeln in OCL?

Lernziele, Teil 2

- ☐ Was versteht man unter dem AnyType in OCL? Was wird dadurch möglich?
- ☐ Wie werden undefinierte Werte in OCL dargestellt?
- ☐ Welche Logik ergibt sich daraus?
- ☐ Wie kann auf alle Instanzen einer Klasse zugegriffen werden?
- ☐ Was ist eine Invariante und wie wird sie verwendet?
- ☐ Wie lässt sich das Systemverhalten genauer spezifizieren?
- ☐ Woraus besteht der Contract einer Funktion?
- ☐ Wie hängt er mit den strukturierten Requirements zusammen?
- ☐ Wie kann in der Nachbedingung Veränderung spezifiziert werden?
- ☐ Wie werden neu erzeugte Objekte überprüft? Wie Ergebnisse?
- ☐ Wie funktioniert der Contract zu kundeHinzufügen (Folie 66)?
- ☐ Wie kommt man darauf?