Technische Universität Berlin

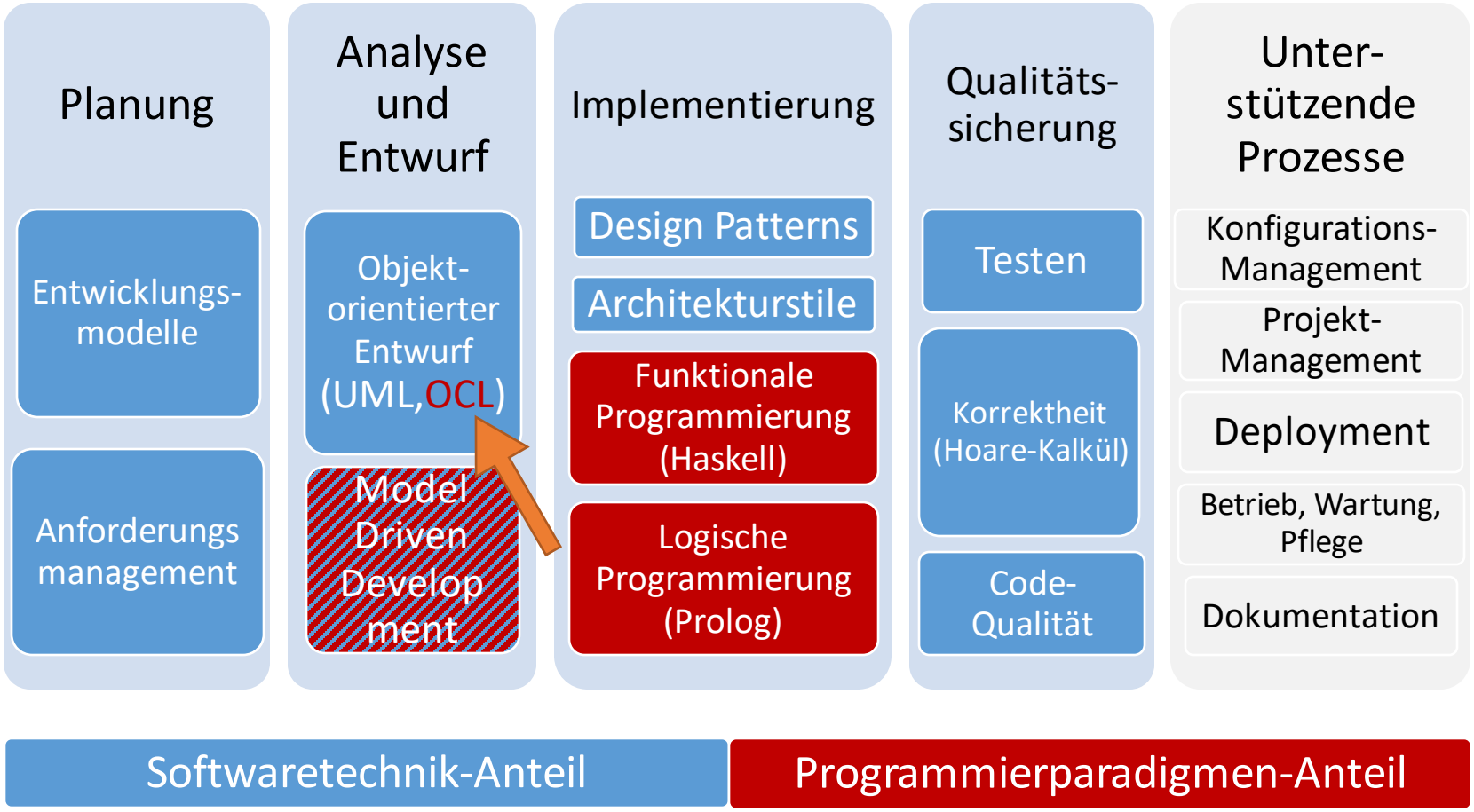
Softwaretechnik und Programmierparadigmen

09 Formale Spezifikation

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

SESE

Diese VL


Planung
Entwicklungs-modelle
Anforderungs management
Analyse und Entwurf
Objekt-orientierter Entwurf (UML,OCL)
Model Driven Develop ment
Implementierung
Design Patterns
Architekturstile
Funktionale Programmierung (Haskell)
Logische Programmierung (Prolog)
Qualitäts-sicherung
Testen
Korrektheit (Hoare-Kalkül)
Code-Qualität
Unter-stützende Prozesse
Konfigurations-Management
Projekt-Management
Deployment
Betrieb, Wartung, Pflege
Dokumentation
Softwaretechnik-Anteil
Programmierparadigmen-Anteil

Inhalt

Formale Spezifikation

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

Inhalt

Formale Spezifikation

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

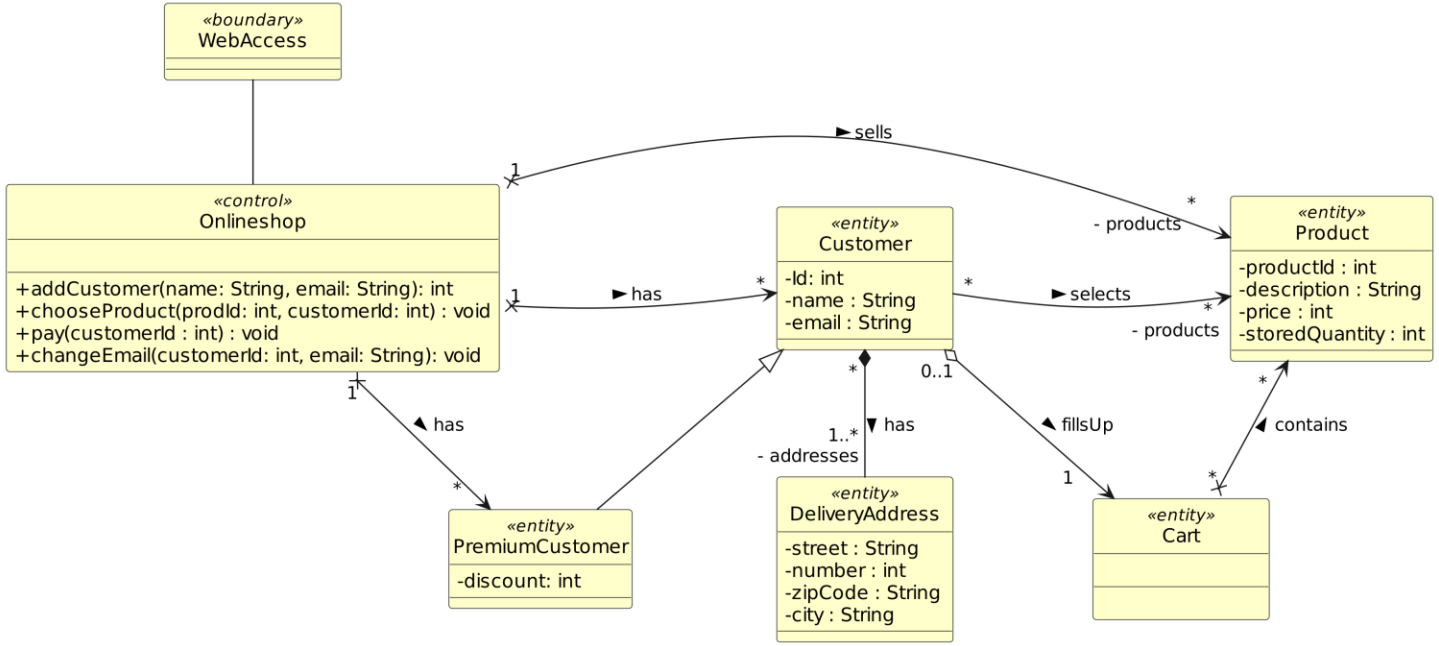
Motivation

Via Jordi Cabot

Motivation

Klassendiagramme beschreiben nur die **Struktur**

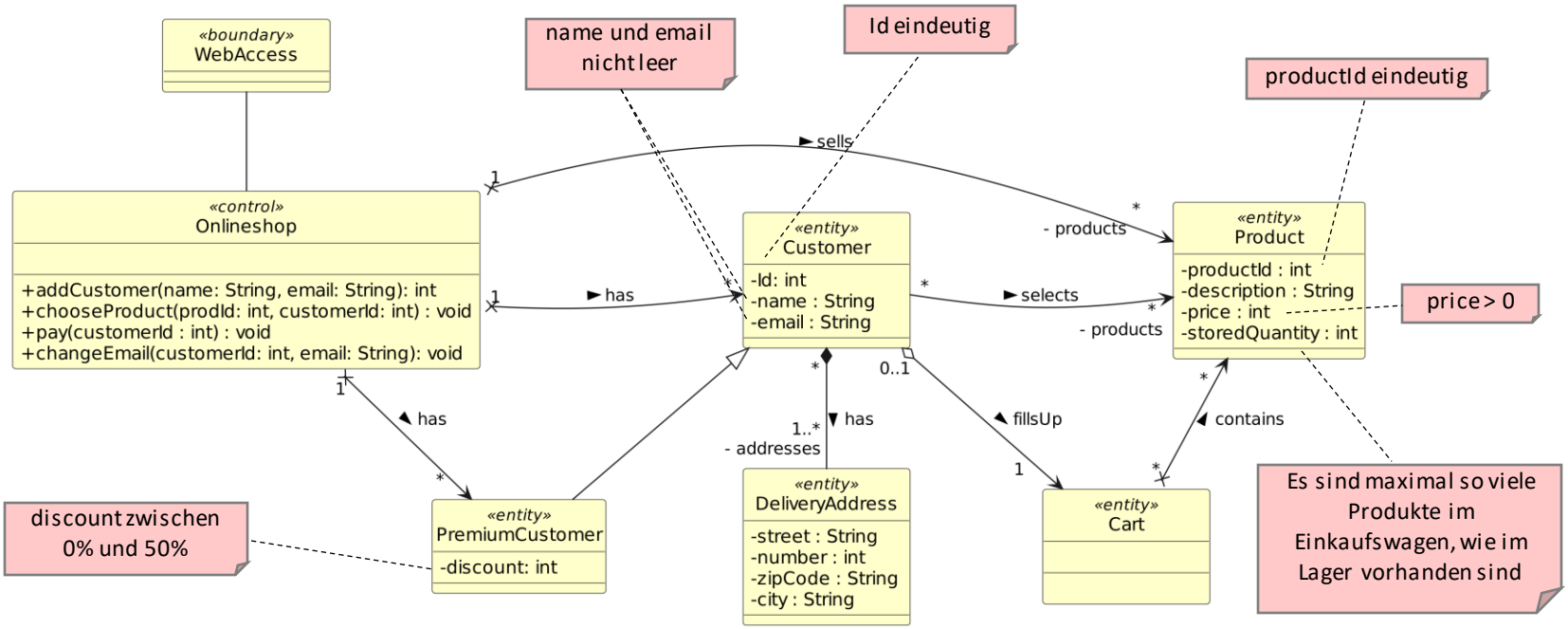
- Bereiche und Bedeutung von Variablen/Operationen wird nicht dargestellt



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

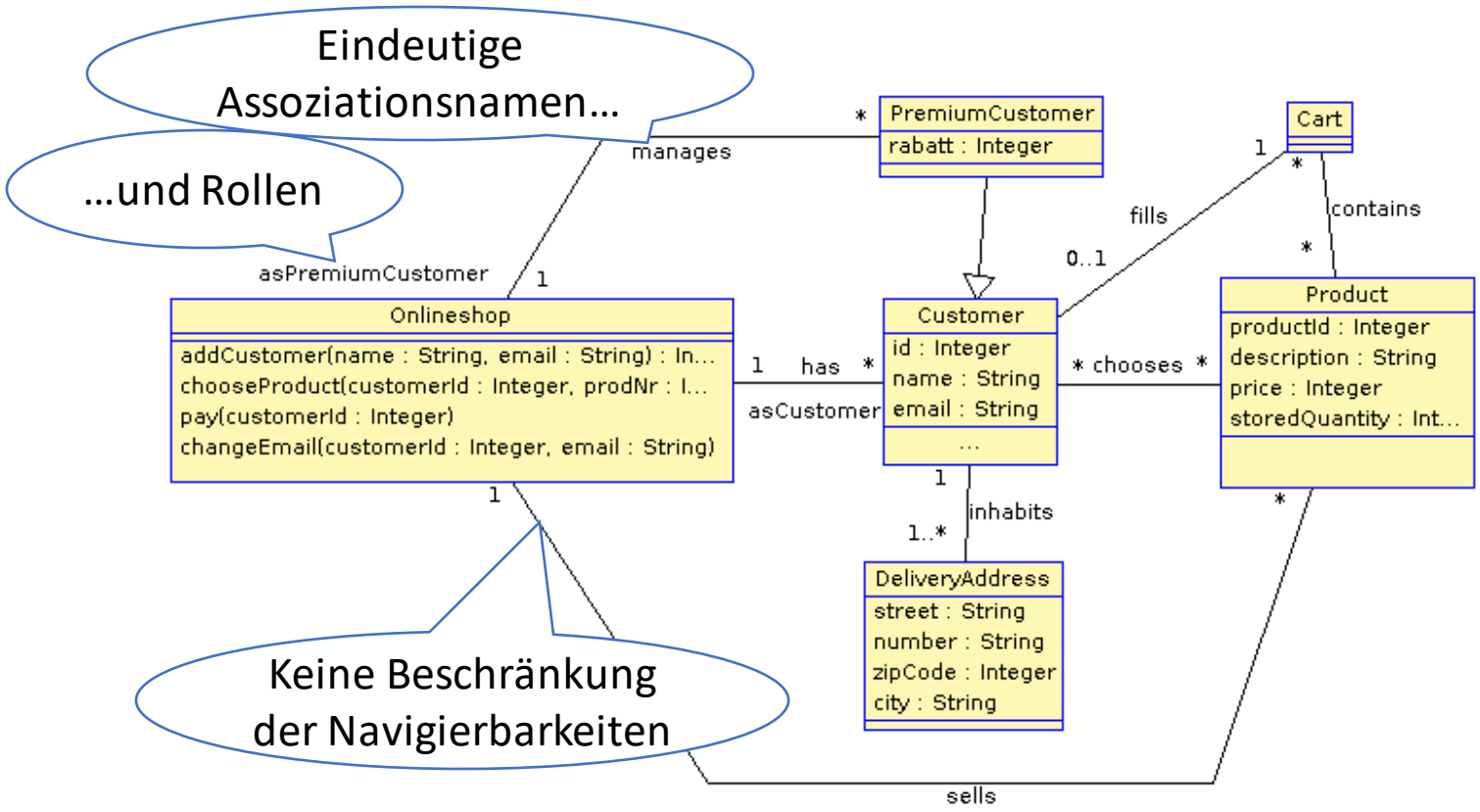
Enthält vordefinierte Mechanismen um…

…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)…

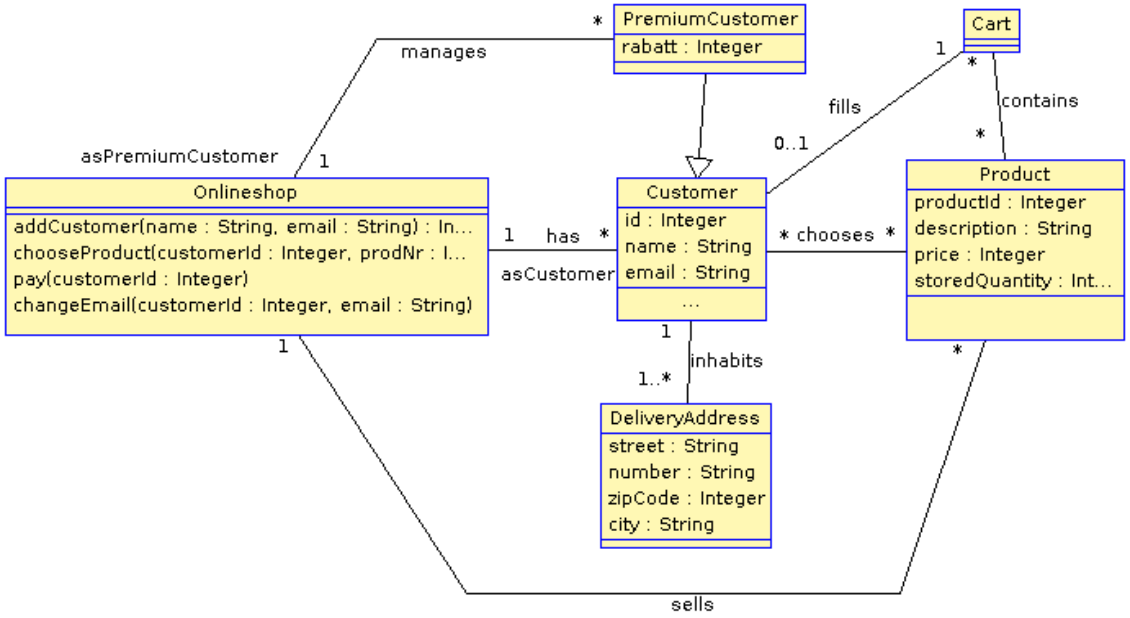


Ein erster Eindruck

a) Der Preis von jedem Produkt ist immer größer als 0

b) Die IDs der Kund:innen existieren nur einmal im System

Sprache



a) context Product inv : self.preis > 0

b) context Customer inv: Customer.allInstances().id->count(self.id) = 1

OCL

Inhalt

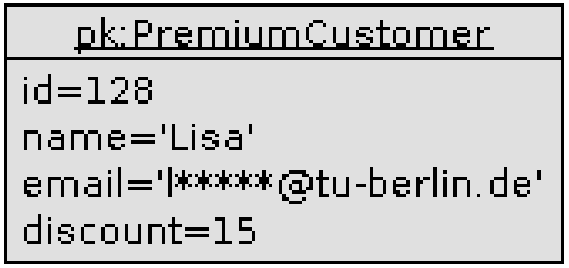
Formale Spezifikation

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

Werte

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

[Objektbezeichner].[Attributbezeichner] ≡ [Wert] : [Typ]



Beispiele

pk.id≡ 128 : Integer

pk.name≡ 'Lisa' : String

pk.email≡ 'l*****@tu-berlin.de' : String

Pk.discount≡ 15 : Integer

Primitive Typen

In OCL sind vier **primitive Typen** vordefiniert

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

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

Primitive Typen

Operationen auf **primitiven Typen** funktionieren wie gewohnt

- Dabei findet kein implizites Typcasting statt

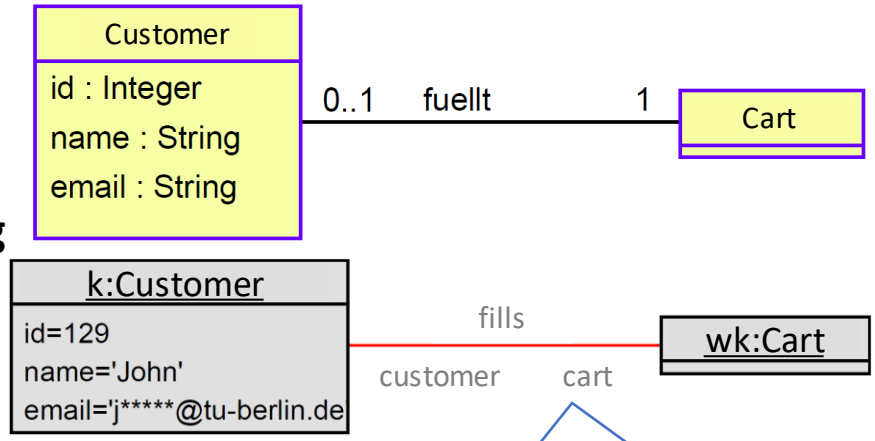
Beispiele

p1.storedQuantity- 12	≡ 6 : Integer
p1.productId > 0	≡ true : Boolean
p1.productId = p2.productId	≡ false : Boolean
if p1.price > p2.price then 1 else 2 endif	≡ 1: Integer
p1.description.size()	≡ 7 : Integer
p1.description.concat(' Brille')	≡ 'Raybaem Brille' : String
p1.description.substring(1, 3)	≡ 'Ray' : String

Navigation

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

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



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

Beispiele

wk.customer≡ k : Customer

wk.customer.id≡ 129 : Integer

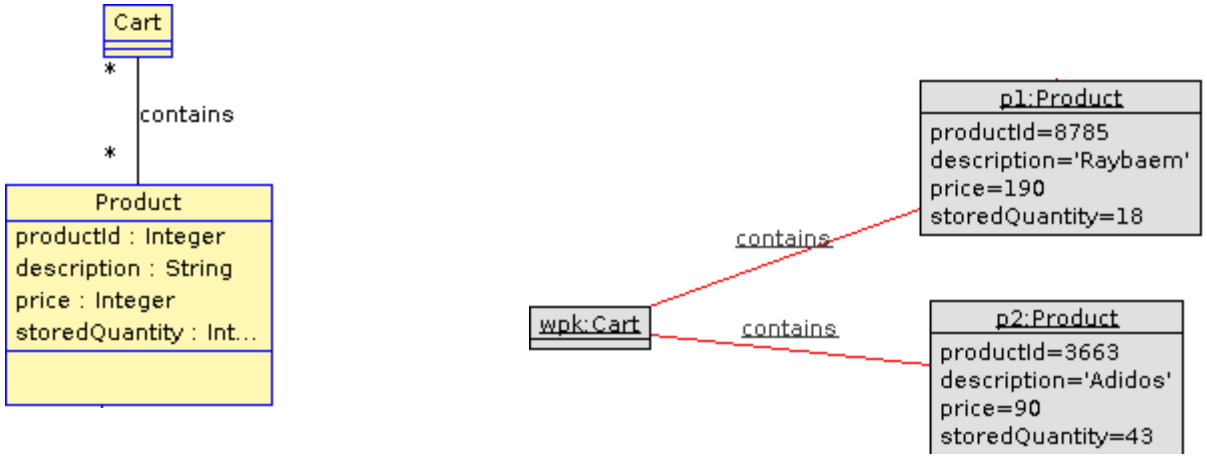
wk.customer.name≡ 'John' : String

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.product≡ Set{p1,p2} : Set(Produkt)

wpk.product->size()≡ 2 : Integer

Collections

In OCL existieren vier **unterschiedliche Collections**…

Elemente…	**Set**	**Bag**	**OrderedSet**	**Sequence**
…können mehrfach enthalten sein	*Nein*	*Ja*	*Nein*	*Ja*
…haben eine Reihenfolge	*Nein*	*Nein*	*Ja*	*Ja*

…mit gemeinsamen, vordefinierten Operationen

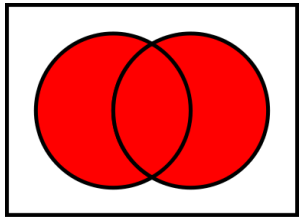
c->**size**() : Integer	Anzahl Elemente in Collection c
c->**isEmpty** : Boolean	true, wenn c leer ist
c->**includes**(obj:OclAny) : Boolean	true, wenn obj in c vorkommt
c->**excludes**(obj:OclAny) : Boolean	false, wenn obj in c vorkommt
c->**count**(obj:OclAny) : Integer	Häufigkeit von obj in c

Set/Bag Operationen

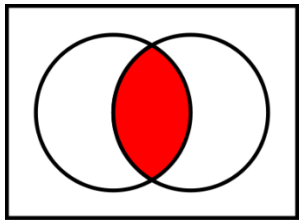
Collections ohne Beachtung der Reihenfolge erlauben **Mengenoperationen**

= (y : Set(T)) : Boolean	Gleichheit
including(y : T) : Set(T)	Hinzufügen
excluding(y : T) : Set(T)	Entfernen
union(y : Set(T)) : Set(T)	Vereinigung
intersection(y : Set(T)) : Set(T)	Schnitt
−(y : Set(T)) : Set(T)	Differenz (nur Set)

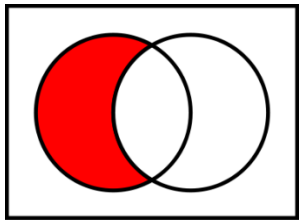
Vereinigung



Schnitt



Differenz

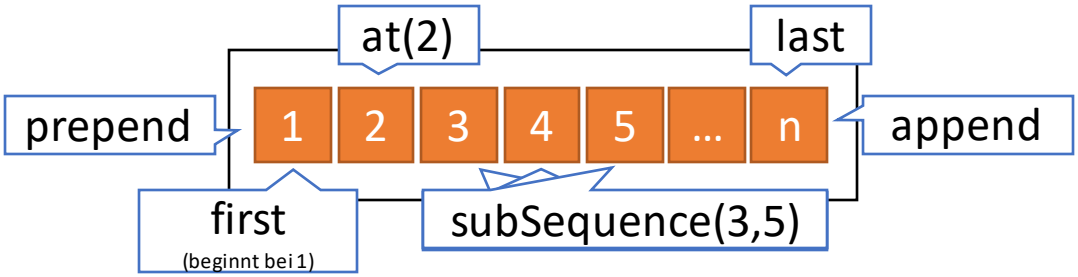


OrderedSet/Sequence Operationen

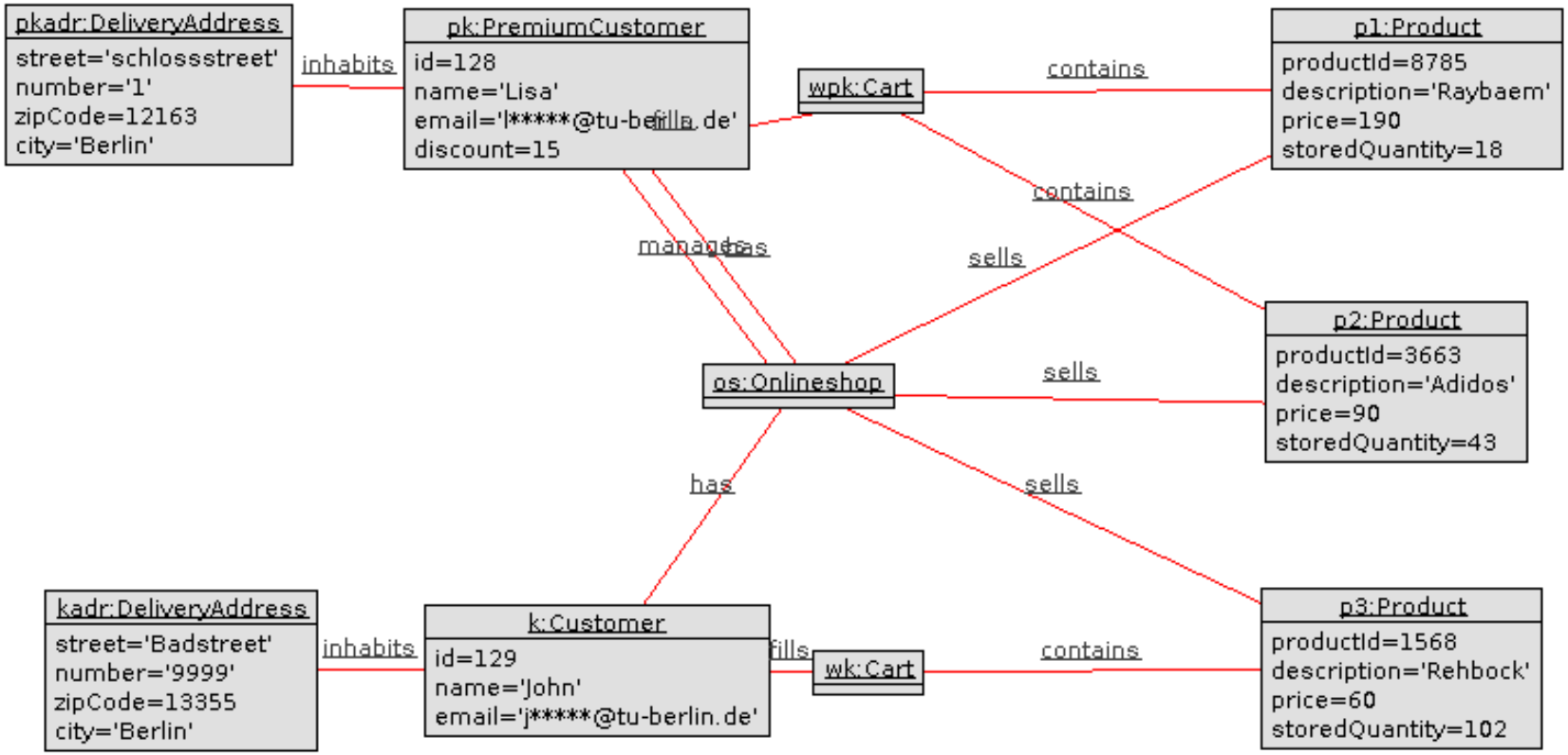
Collections mit Berücksichtigung der Reihenfolge erlauben **Indexzugriff**

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
first() : T ≡ S->at(1)	Erstes Element
last() : T ≡ S->at(S->size())	Letztes Element
subSequence(y : Integer, z : Integer) : Sequence(T)	Ausschnitt

Achtung: USE fügt nach i ein

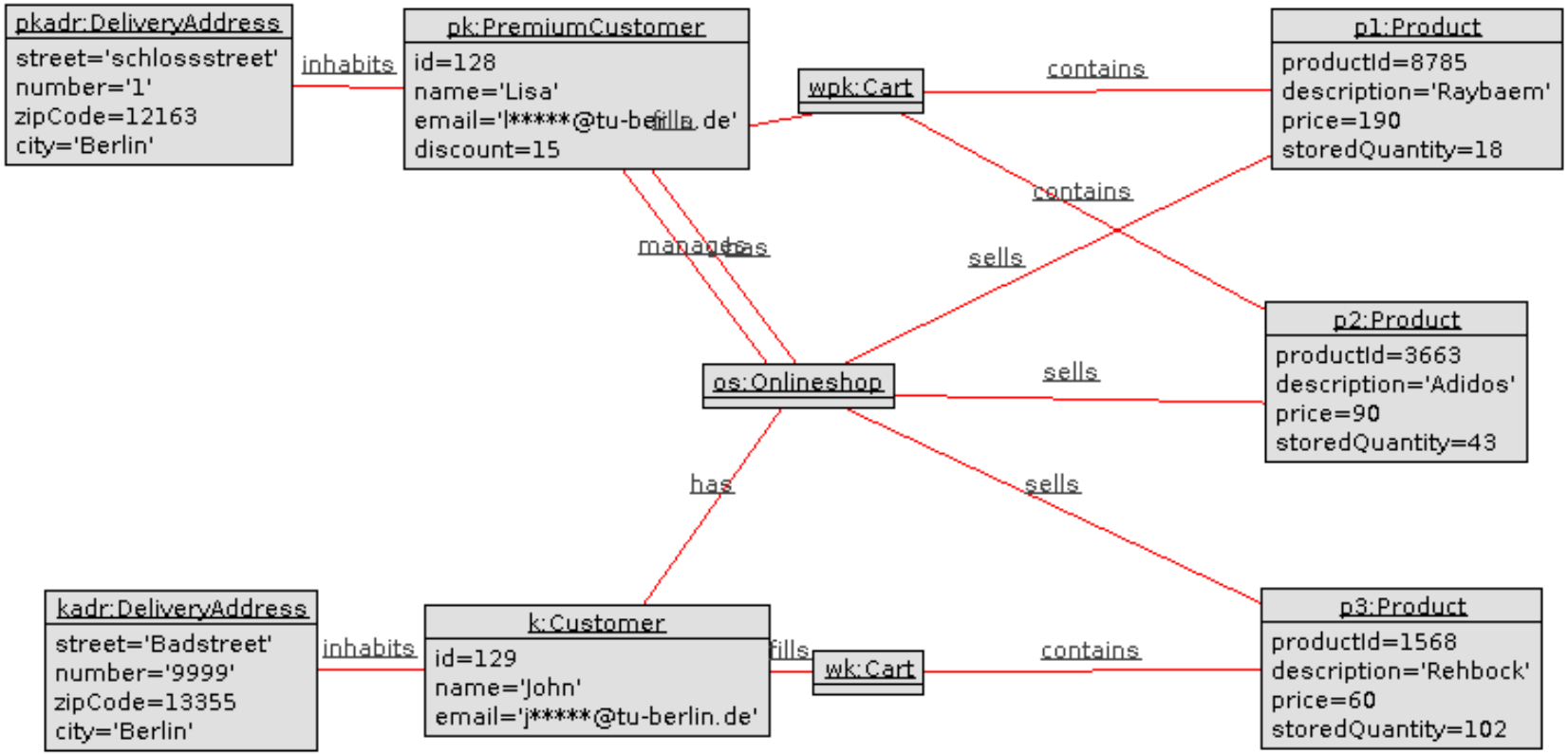


Beispiele



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

Beispiele



k.cart.product->union(pk.cart.product) = os.product

k.cart.product->union(pk.cart.product) : Set(Product) = Set{p1,p2,p3} = os.product : Set(Product)

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

Auswahl

Aus Collections können gewünschte Werte **gefiltert** werden

c->**select**(x : T | P(x)) $\{x \in C \mid P(x)\}$

c->**reject**(x : T | P(x)) $\{x \in C \mid \neg P(x)\}$

Beispiele

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

Set{1,2,3}->reject(i : Integer | i <= 1) ≡ Set{2,3} : Set(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->**collect**(x : T | E(x)) $\{e \mid \exists\, x \in C.\ e = E(x)\}$

collect gibt immer ein Bag zurück!

Beispiele

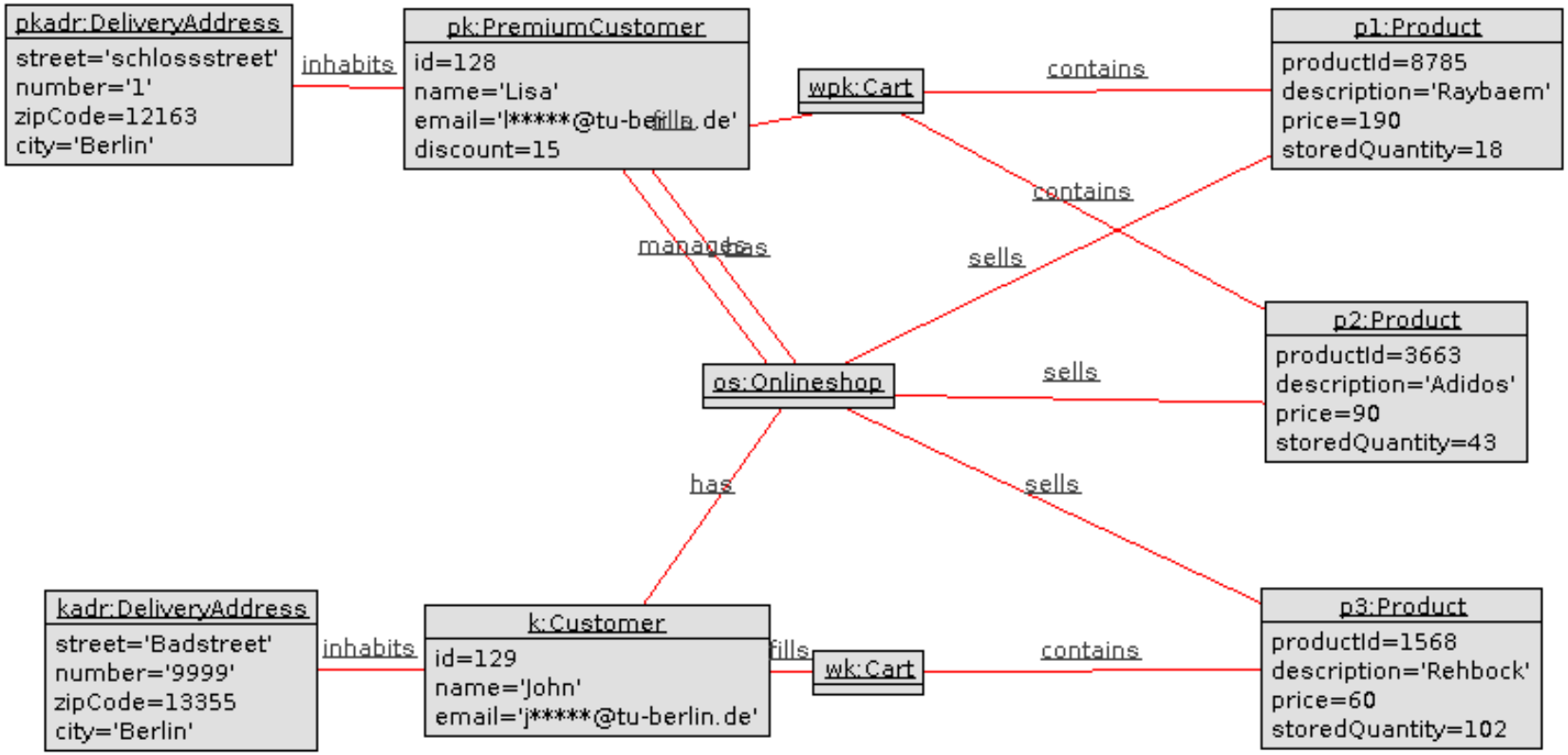
Set{1,2,3}->collect(i : Integer | i+1) ≡ Bag{2,3,4} : Bag(Integer)

Set{1,2,3}->collect(i : Integer | i.toString()) ≡ Bag{'1','2','3'} : Bag(String)

Pseudo-Java

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

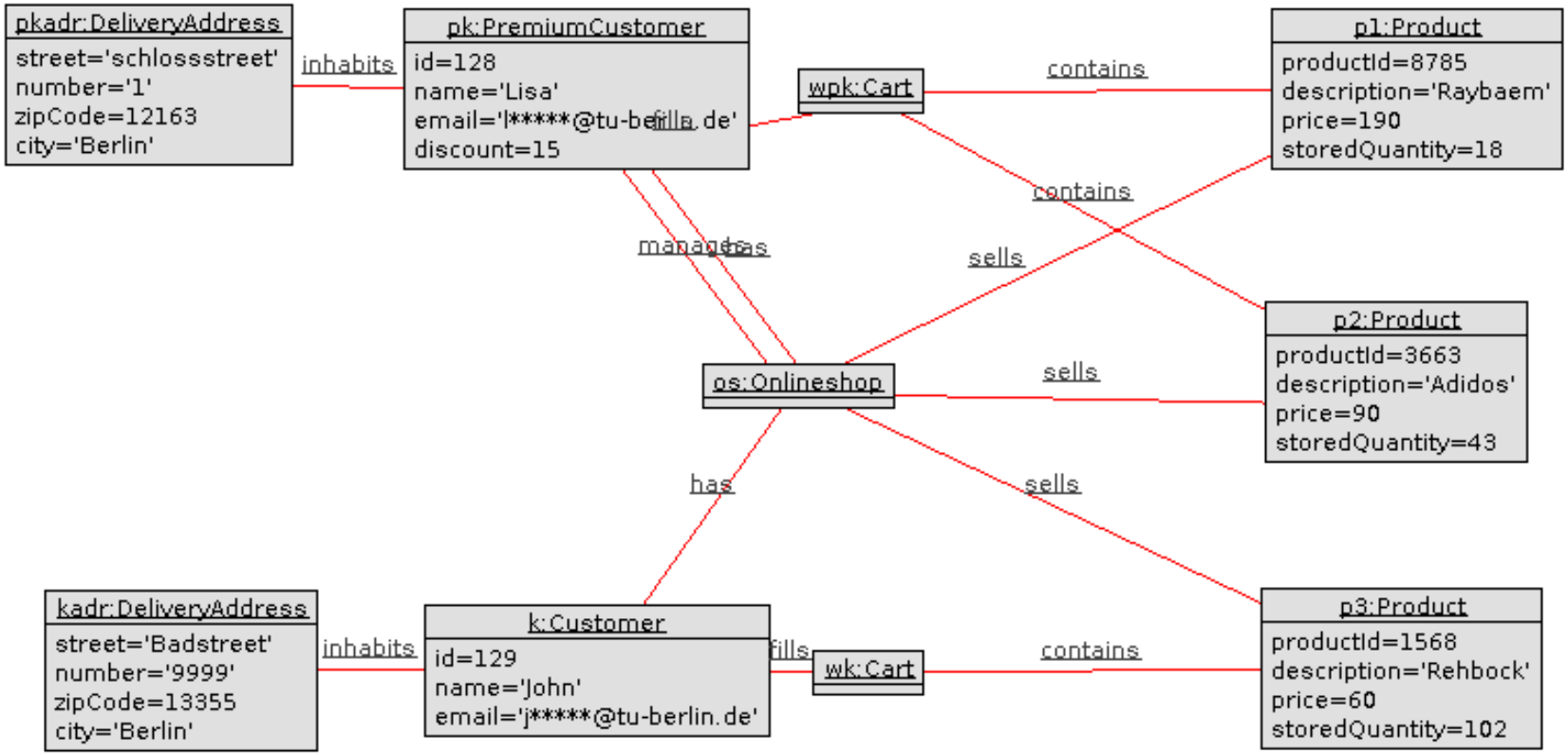
Beispiele



Von welchen Produkten sind mehr als 30 Stück auf Lager?

Welche Produktnummern sind im System vergeben?

Beispiele



os.product->select(p : Product | p.storedQuantity > 30) ≡ Set{p2,p3} : Set(Product)

os.product->collect(p : Product | p.productId) ≡ Bag{1568,3663,8785} : Bag(Integer)

Iteration

Werte in Collections können beliebig **zusammengefasst** werden

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

c->**iterate**(x : T; acc : T2 = startwert | E(acc, x))

Beispiele

Set{1,2,3}->iterate(x : Integer; acc : Integer = 0 | acc + x) ≡ 6 : Integer

Set{1,2,3}->iterate(x : Integer; acc : String = '' | acc + x.toString()) ≡ '321' : 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->**forAll**(c : T | P(c)) $\forall\, c \in C.\, P(c)$

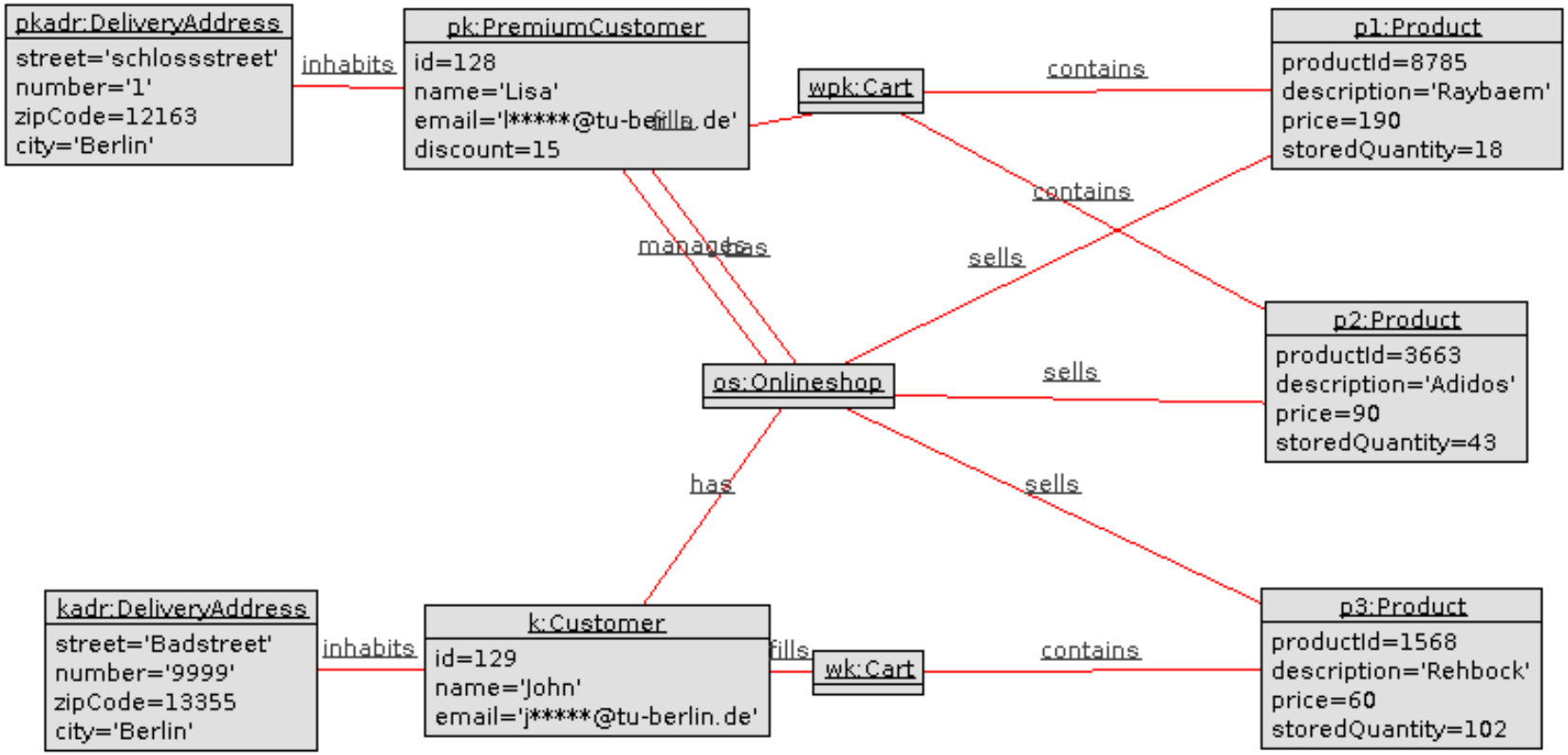
c->**exists**(c : T | P(c)) $\exists\, c \in C.\, P(c)$

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

c->**includesAll**(cy : Col(T)) : Boolean ≡ cy->forAll(e | c->**includes**(e))

c->**exludesAll**(cy : Col(T)) : Boolean ≡ cy->forAll(e | c->**excludes**(e))

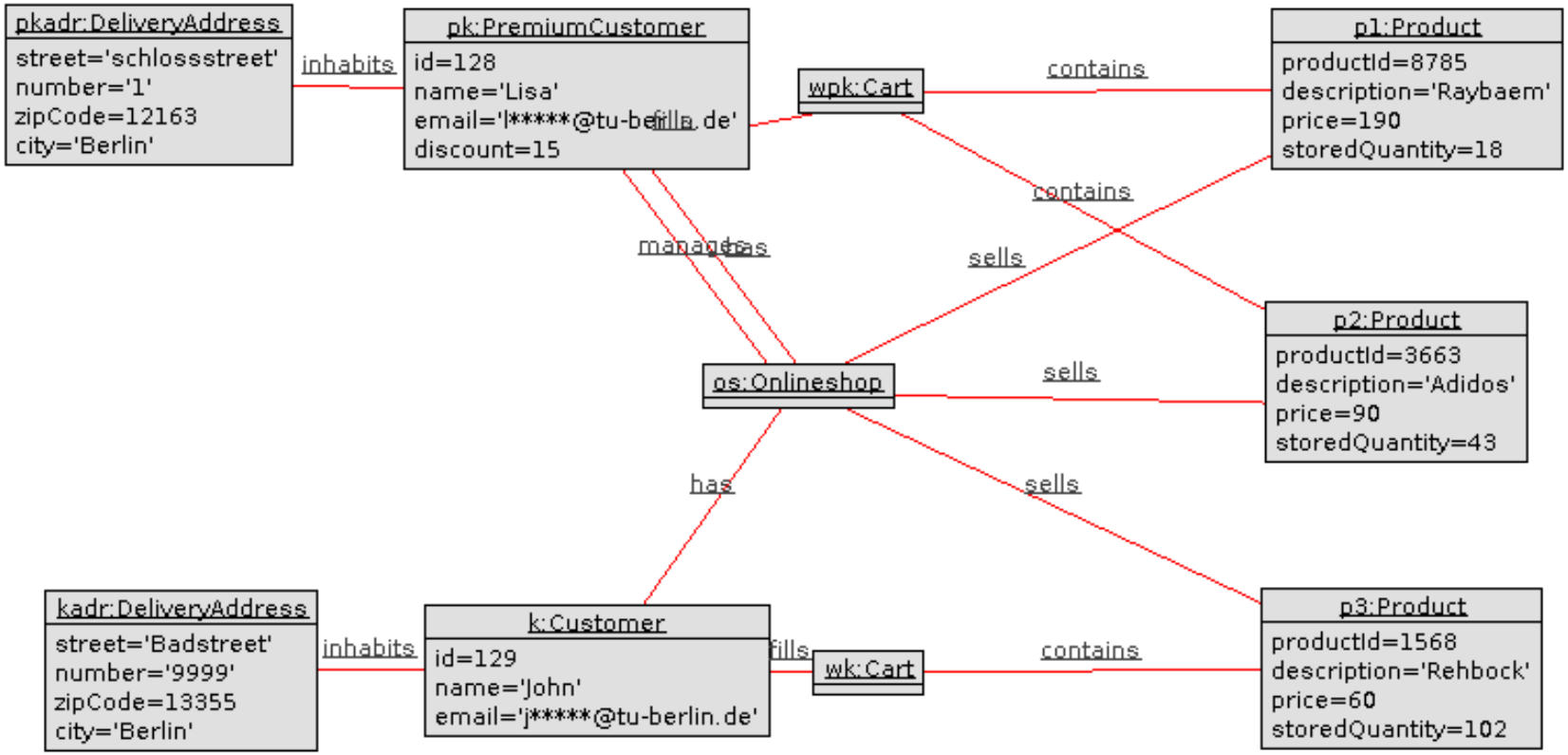
Beispiele



Wieviel kostet Lisas Warenkorb zur Zeit?

Sind alle Produkte darin günstiger als 180€?

Beispiele

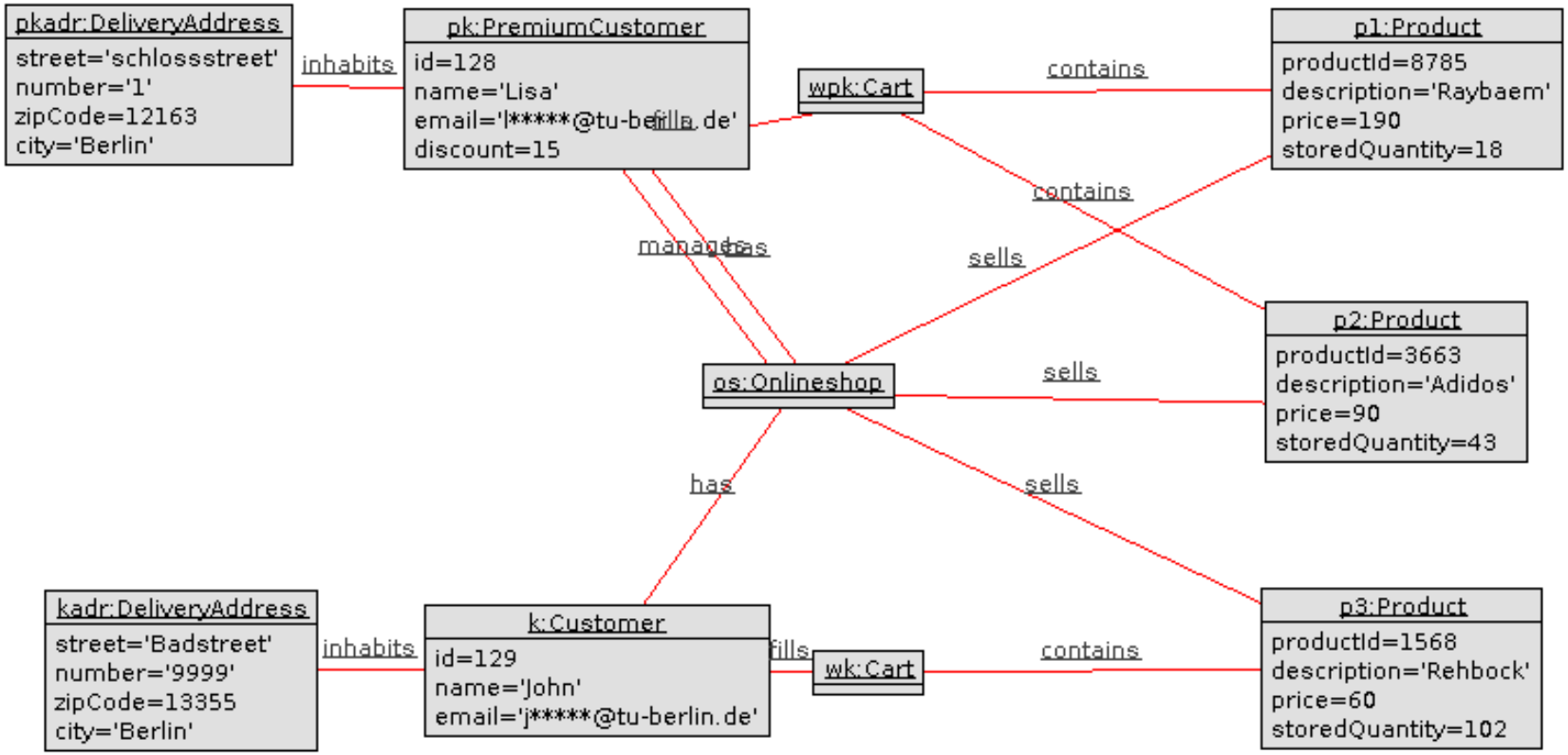


wpk.product->iterate(p : Product ; acc : Integer = 0 | acc + p.price) ≡ 280 : Integer

wpk.product->forAll(p : Product | p.price < 180) ≡ false : Boolean

Stimmt das?

Beispiele



wpk.product->iterate(p : Product; acc : Integer = 0 | acc + p.price) * (100-pk.discount)/100 ≡ 238.0 : Real

wpk.product->forAll(p : Product | p.price * (100-pk.discount)/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

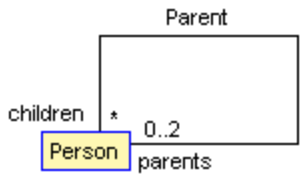
Die Closure-Operation bildet die transitive Hülle einer Beziehung, z.B. einer Assoziation oder Generalisierung

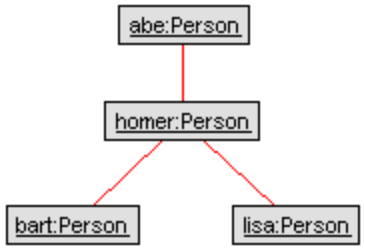
C->**closure**(c : T | P(c))

- 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





bart.parents->closure(parents) ≡ Set{homer,abe} : Set(Person)
Set{abe}->closure(children) ≡ Set{abe,bart,homer,lisa} : Set(Person)

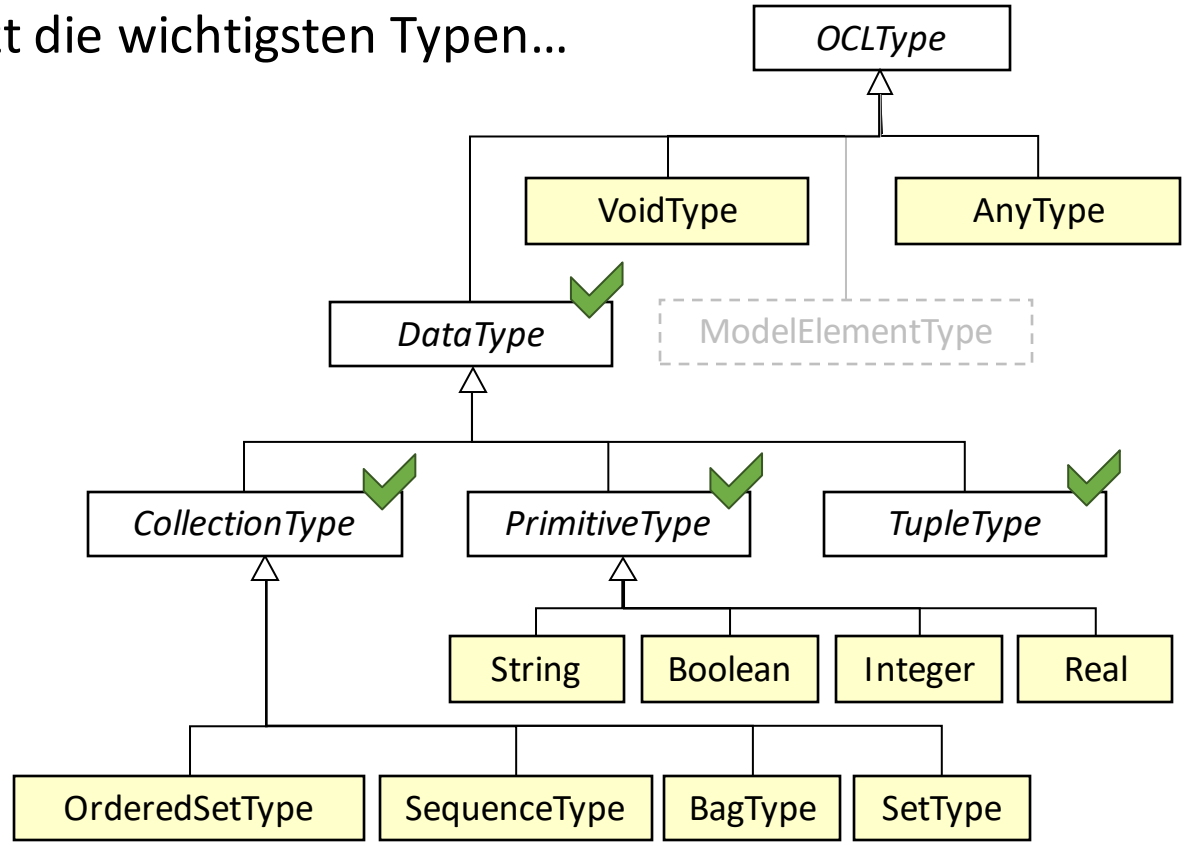
Die Closure kann auch für allgemeine Schleifen verwendet werden:

Set{1}->closure(x| if x < 5 then x+1 else x endif) ≡ Set{1,2,3,4,5} : Set(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 oclIsTypeOf oder oclIsKindOf **bestimmen**…

pk.oclIsTypeOf(Customer)≡ **false**
pk.oclIsTypeOf(PremiumCustomer)≡ true

k.oclIsTypeOf(Customer)≡ true
k.oclIsTypeOf(PremiumCustomer)≡ false

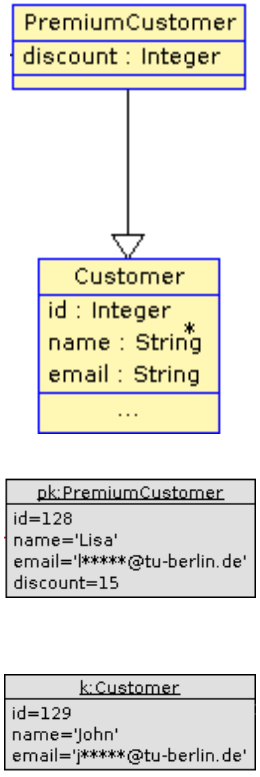
Type
Überprüft den tatsächlichen Typ des Objekts

pk.oclIsKindOf(Customer)≡ **true**
pk.oclIsKindOf(PremiumCustomer)≡ true

k.oclIsKindOf(Customer)≡ true
k.oclIsKindOf(PremiumCustomer)≡ 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	≡ **Undefined** : OclVoid
1/0 = **Undefined**	≡ true : Boolean
1/0 = **null**	≡ true : Boolean
(1/0).**oclIsUndefined**	≡ true : Boolean
Sequence{1,2,**null**,4,5}	: Sequence(Integer)
Sequence{1,2,**null**,4,5}->size()	≡ 5 : Integer

Dreiwertige Logik in OCL

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

b	NOT b
false	true
null	null
true	false

AND	false	null	true
false	false	false	false
null	false	null	null
true	false	null	true

OR	false	null	true
false	false	null	true
null	null	null	true
true	true	true	true

XOR	false	null	true
false	false	null	true
null	null	null	null
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

OCL 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)
```

OCL „weiß“, dass i ein Integer sein muss

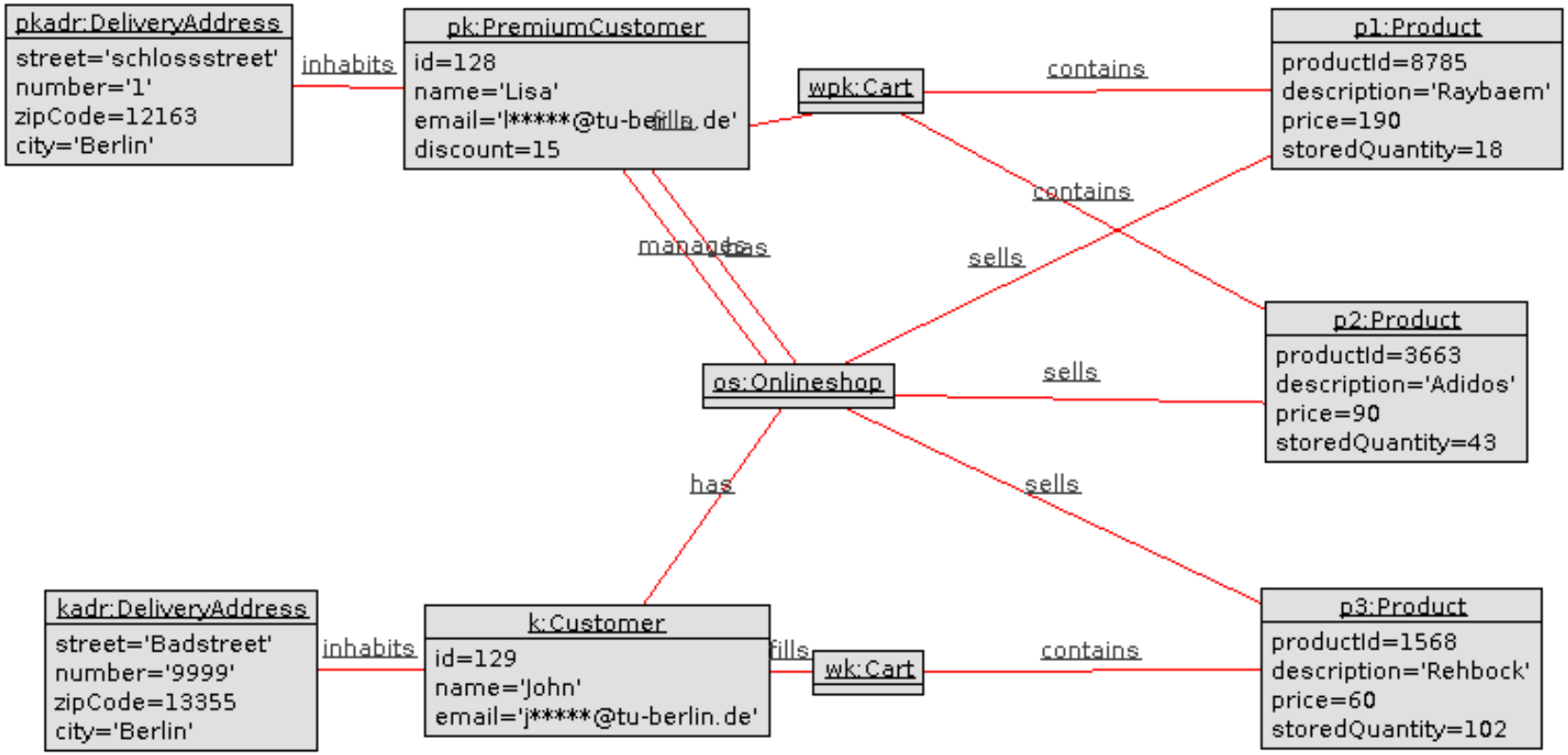
Und dass die Werte als Argument dienen

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()) ≡ Bag{1,2,3} : Bag(Integer)
Set{-1,2,-3}.abs()                ≡ Bag{1,2,3} : Bag(Integer)
```

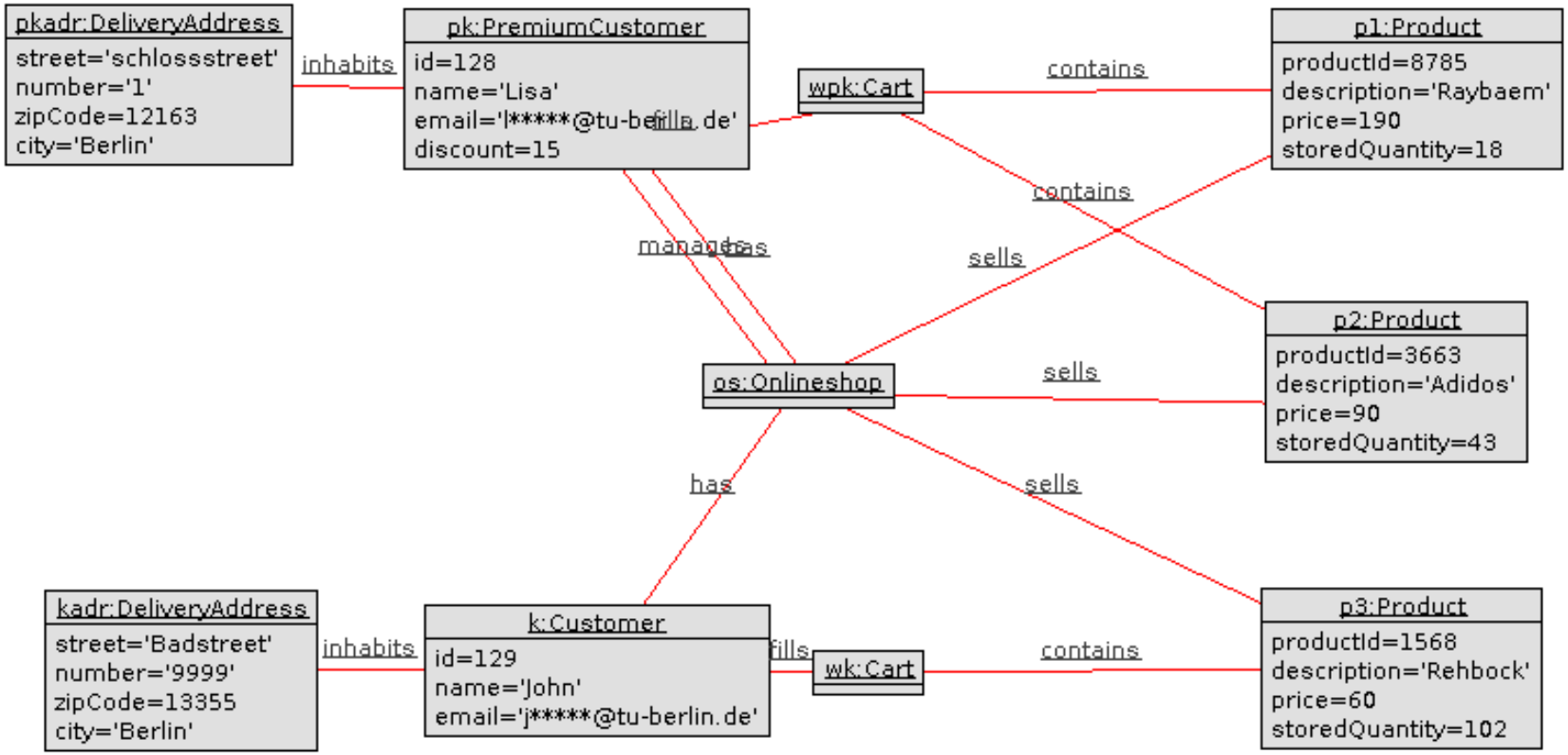
Beispiele



Wie viele Produkte sind zur Zeit im Lager?

Wer kauft Produkte über 80€?

Beispiele



os.product.storedQuantity->iterate(m; s : Integer = 0 | s + m)≡ 163 : Integer

os.product->select(price > 80).cart.customer.name->asSet()≡ Set{'Lisa'}

collect collect collect

Collect und CollectNested

os.kunde.cart->collect(product)

Eigentlich müsste das eine Menge von Mengen sein…

os.customer.cart≡ Bag{wk,wpk} : Bag(Cart)

Bag{wk,wpk} ->collect(product)≡ Bag{p1,p2,p3} : Bag(Product)

…ist es aber nicht.

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

Bag{wk,wpk}->**collectNested**(product) ≡ Bag{Set{p3},Set{p1,p2}} : Bag(Set(Product))

Bag{wk,wpk}->collect(product) **=** Bag{wk,wpk}->collectNested(product)->flatten()

Inhalt

Formale Spezifikation

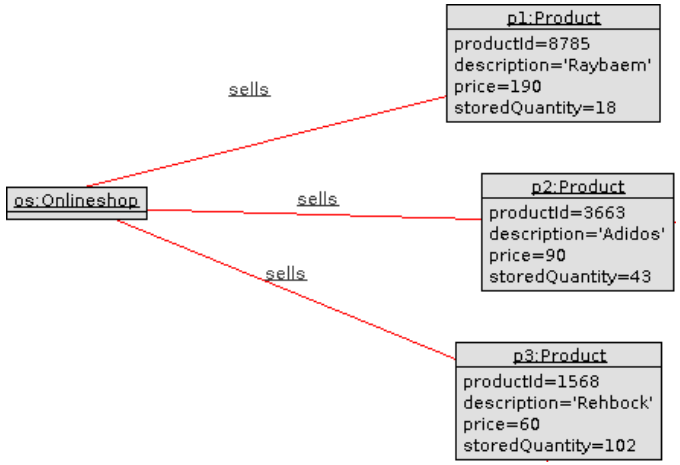
- Grundlagen
- Object Constraint Language (OCL)
- Invarianten
- Contracts
- Weiteres

Motivation

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

Haben zur Zeit alle Produkte einen Preis > 0?

os.product->forAll(p | p.price > 0) ≡ true



Aber eigentlich wollen wir ja generelle Anforderungen spezifizieren…

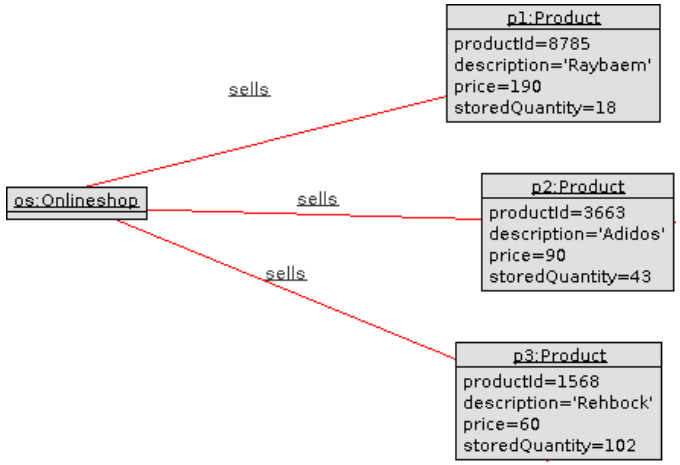
Wie können wir Aussagen über alle Instanzen machen,
ohne über einen Controller zu navigieren?

(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 **os.product**->forAll(p | p.price > 0)≡ true

ohne **Product.allInstances()**->forAll(p | p.price > 0)≡ true

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.product->forAll(p | p.price > 0)≡ true

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

Product.allInstances()->forAll(p | p.price > 0)≡ true

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

context p : Product **inv:** (p.price > 0)

Alternative

Für alle Produkte gilt: sie haben einen positiven Preis.

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

os.product->forAll(p | p.price> 0)≡ true

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

Product.allInstances()->forAll(p | p.price > 0)≡ true

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

context Product **inv:**(self.price> 0)

Ohne Bezeichner gilt **self**

Alternative

Für alle Produkte gilt: sie haben einen positiven Preis.

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

os.product->forAll(p | p.price> 0)≡ true

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

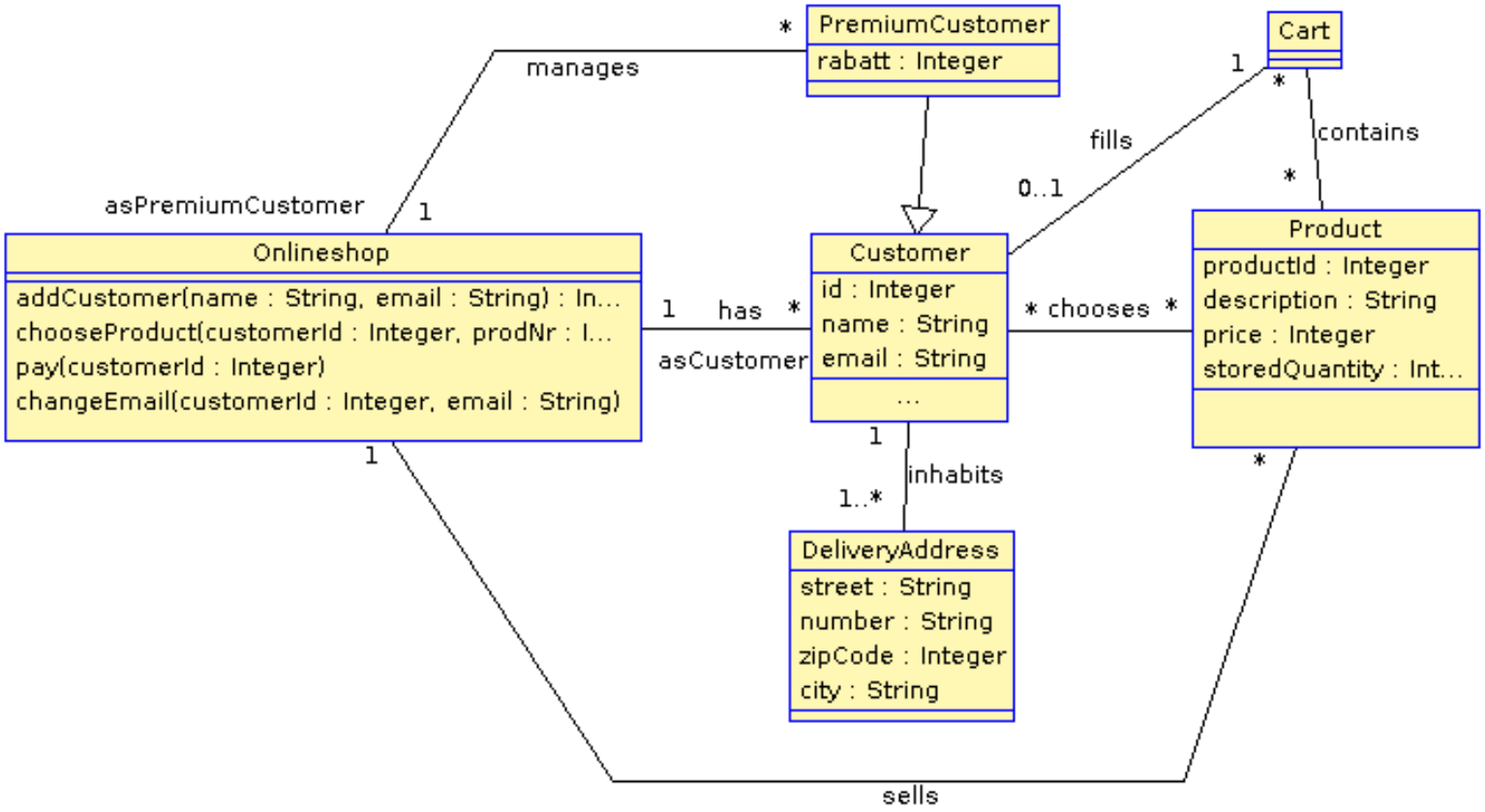
Product.allInstances()->forAll(p | p.price > 0)≡ true

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

context Product **inv:**(price > 0)

oder Kurzschreibweise

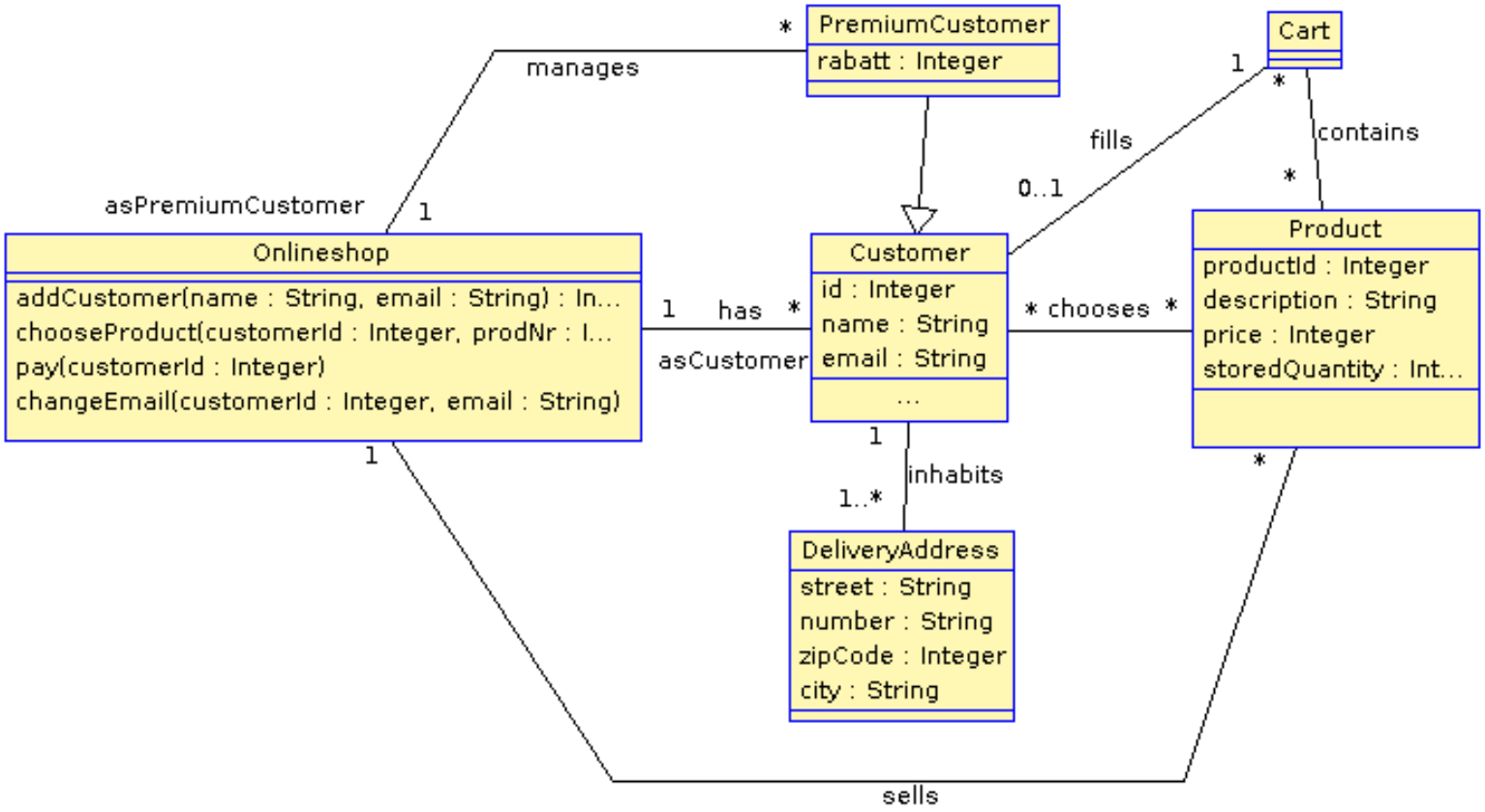
Weitere Beispiele



Name und Email von Kund:innen sind nicht leer

Der Rabatt von Premiumkund:innen liegt zwischen 0 und einschließlich 50

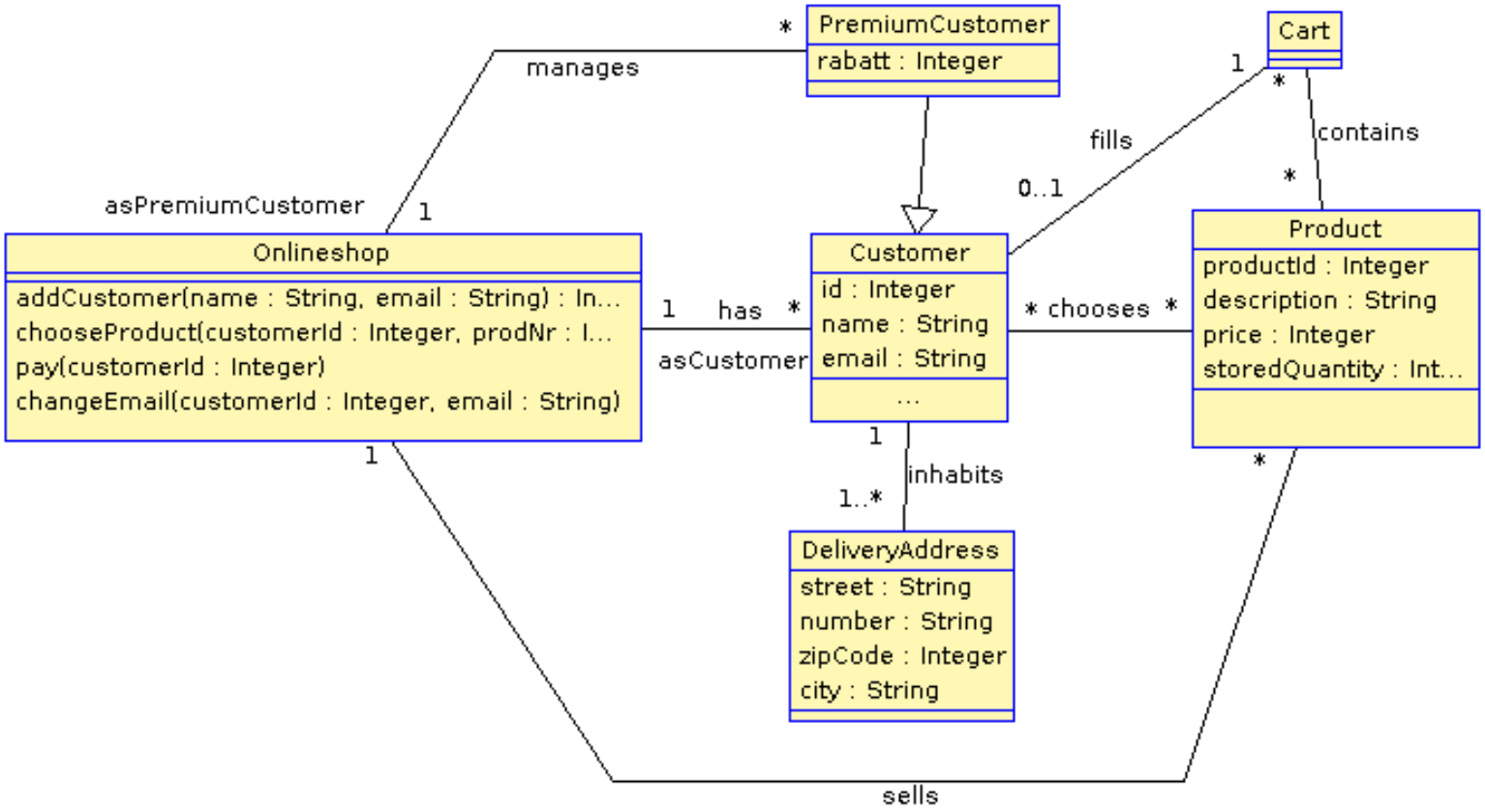
Weitere Beispiele



context Customer **inv:** self.name <> ‘‘ and self.email <> ‘‘

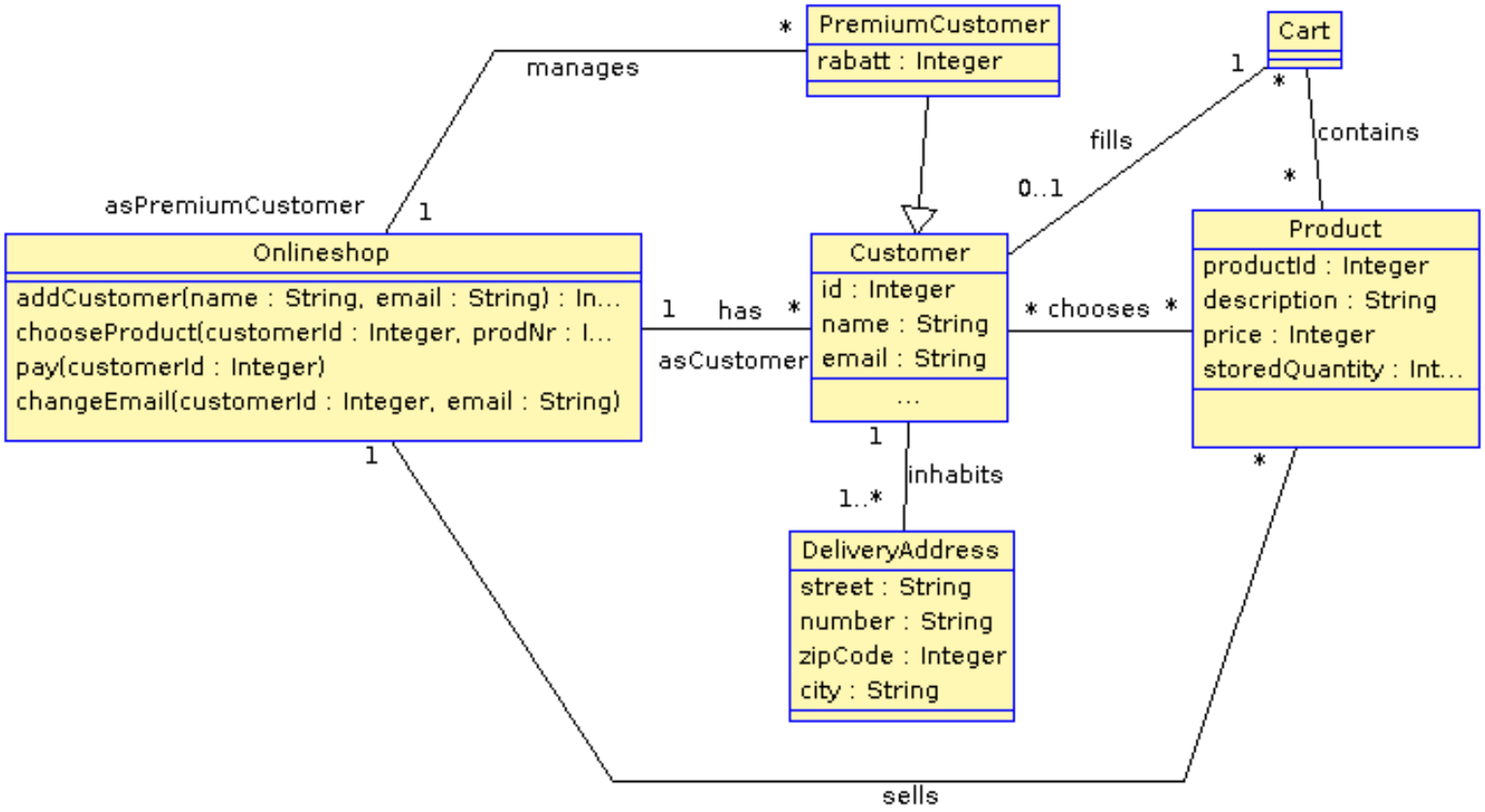
context PremiumCustomer **inv:** discount > 0 and discount <= 50

Weitere Beispiele



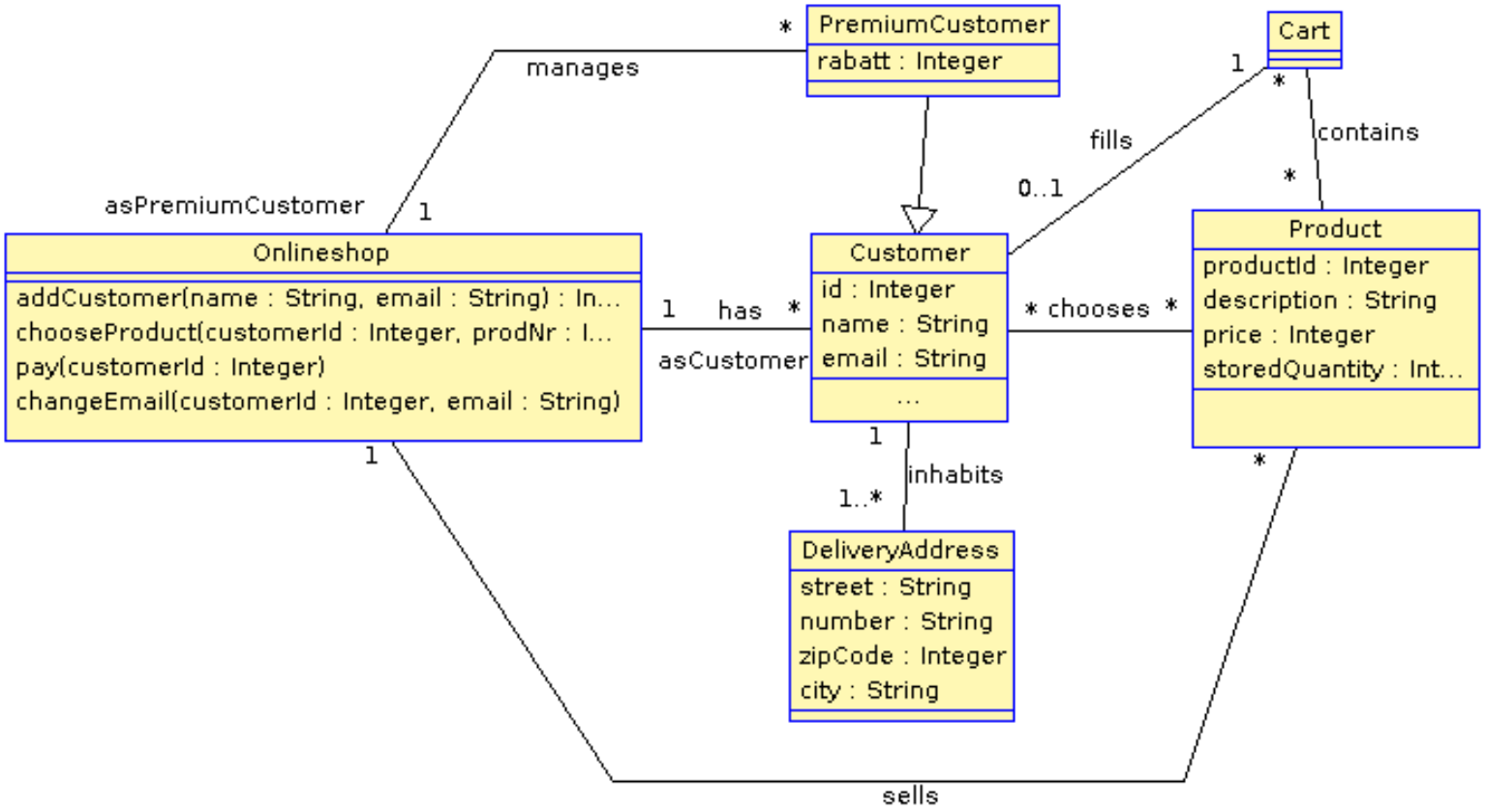
Die IDs der Kund:innen sind eindeutig

Weitere Beispiele



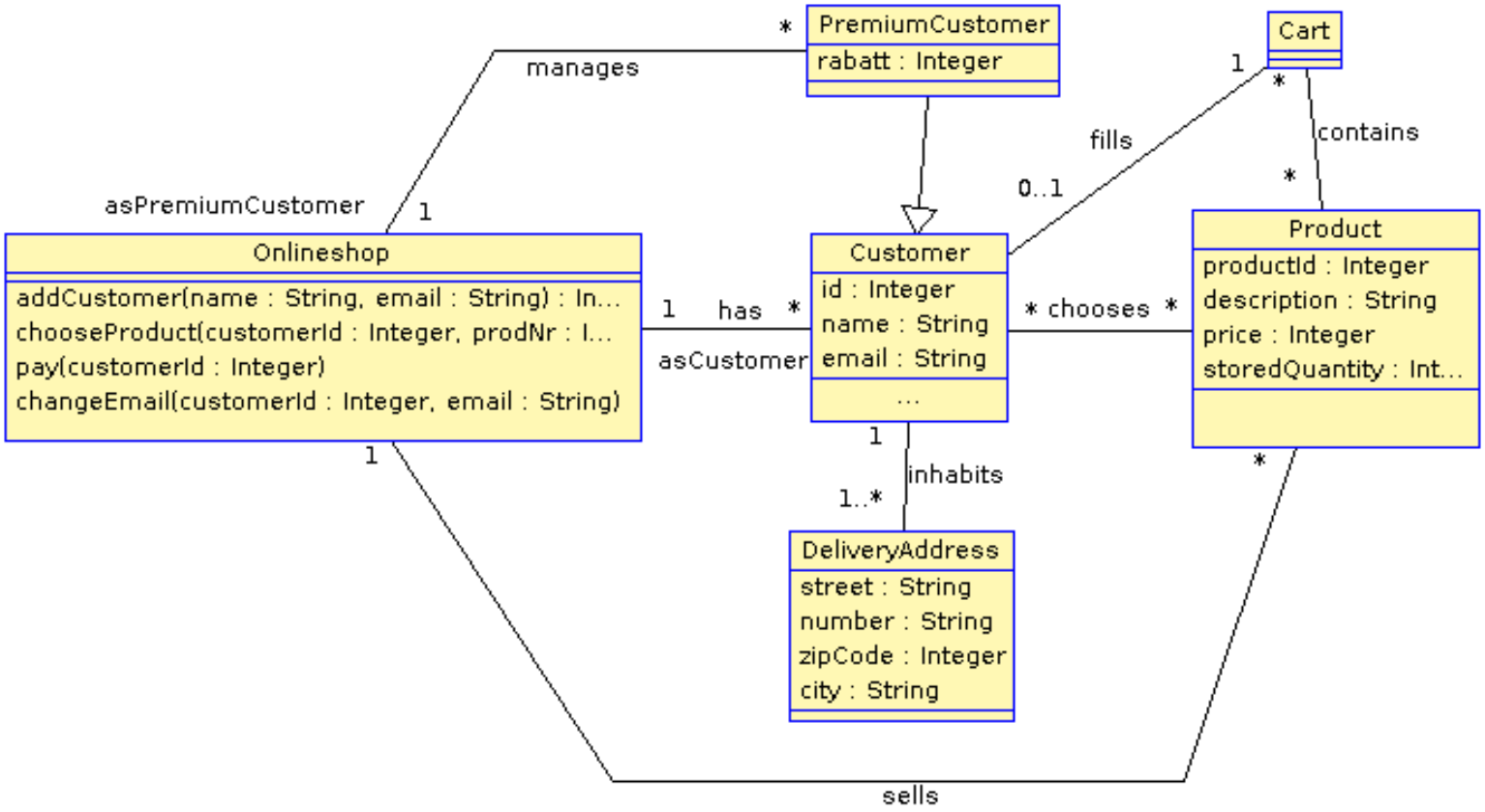
context Cusomer **inv:** Customer.allInstances().id->count(self.id) = 1

Weitere Beispiele



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

Weitere Beispiele



context Product **inv:** Cart.allInstances().product->count(self) <= storedQuantity

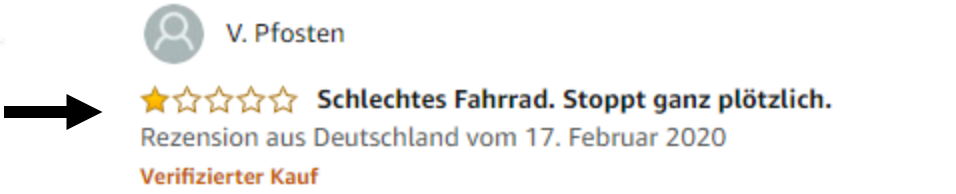
Inhalt

Formale Spezifikation

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

Motivation

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

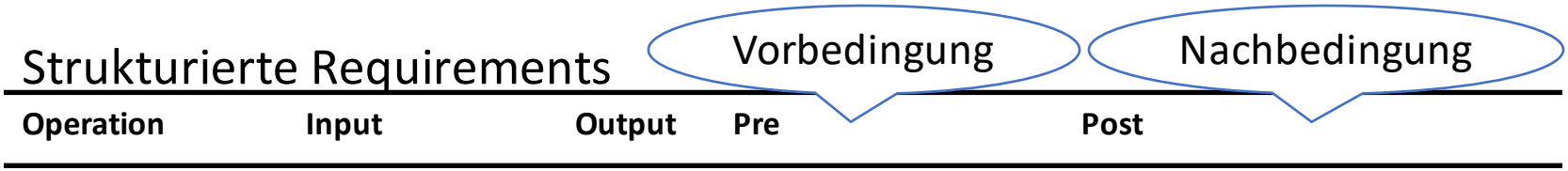


…wie geht das mit dem **Verhalten**?

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)] **:** [Rückgabetyp]

pre: [Vorbedingungen]

post: [Nachbedingungen]

Beispiel

Operation	Input	Output	Pre	Post
changeEmail	CustomerId, Email		CustomerId existiert, Email nicht leer	Email von Customer mit CustomerId geändert

context Onlineshop::changeEmail(customerId, email) : void

pre: self.customer.id->includes(customerId) and email <> ‘‘

post: self.customer->any(id = customerId).email = email

Wähle den/die Kunden/Kundin mit customerId

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::changeEmail(customerId, email) : void

post: let k = self.customer->any(id = customerId) **in** k.email = email

Beispiel

Wie lässt sich folgender Contract ausdrücken?

Operation	Input	Output	Pre	Post
wähleProdukt	customerId, ProductId		CustomerId existiert, ProductId existiert	Customer mit CustomerId hat Produkt mit ProdId im Warenkorb

context Onlineshop::chooseProduct(customerId, prodId) : void

pre: self.customer.id->includes(customerId)

pre: self.product.productId->includes(prodId)

post: **let** k = self.customer->any(id = customerId) **in**

let p = self.product->any(productId = prodId) **in**

k.cart.produkt->includes(p)

CustomerId existiert

ProdId existiert

Customer mit CustomerId…

Produkt mit ProdId…

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

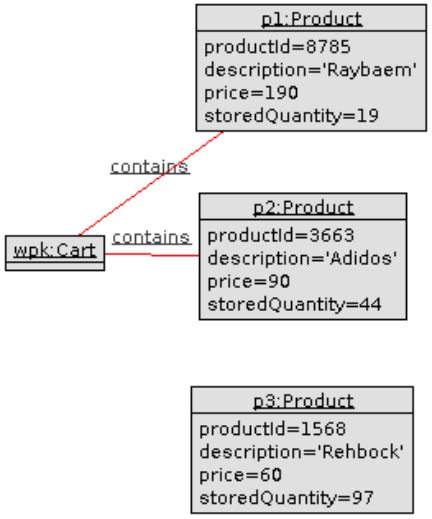
Durch Ausführung der Operation hat sich an den Produkten nichts geändert
post: Product.allInstances() = Product.allInstances()**@pre**

Ein neues Produkt p ist dazu gekommen
post: Product.allInstances() = Product.allInstances()**@pre**->including(p)

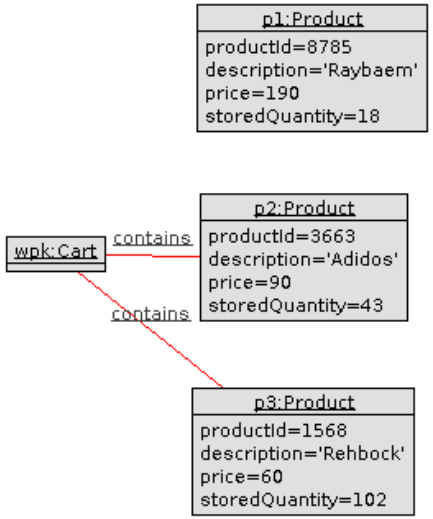
@Pre

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

pre:



post:



Set(Product)

wpk.product.storedQuantity
= Bag{43,102}
wpk.product.storedQuantity@pre
= Bag{44,97}
wpk.product@pre.storedQuantity
= Bag{18,43}
wpk.product@pre.storedQuantity@pre
= Bag{19,44}

Beispiel

Wie lässt sich die Nachbedingung des Contracts ausdrücken?

Operation	Input	Output	Pre	Post
pay	CustomerId		CustomerId existiert, Produkte auf Lager	Entsprechend weniger Produkte, Warenkorb leeren

context Onlineshop::pay(customerId) : void

pre: **let** k = self.customer->any(id = customerId) **in**
k <> null **and** k.cart.product->forAll(storedQuantity > 0)

post: **let** k : Customer = self.customer->any(id = customerId) **in**
let ps : Set(Product) = k.cart@pre.product@pre **in**
ps->forAll(storedQuantity= storedQuantity@pre - 1)
and k.cart.product->size() = 0

Customer mit customerId

Entsprechende Produkte **vor** dem bezahlen

Lagermenge verringert

Warenkorb leeren

Result und oclIsNew

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

context: Customer::getId() : Integer

post: result = self.id

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

context: Onlineshop::addProduct(…) : Product

Post: result.oclIsNew()

Beispiel

Wie lässt sich folgender Contract ausdrücken?

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

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

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

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

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

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

and self.customer@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

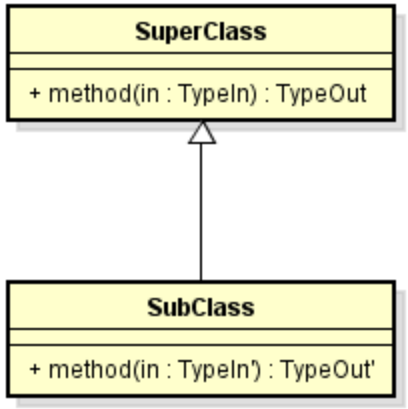
Die ID ist neu und…

…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

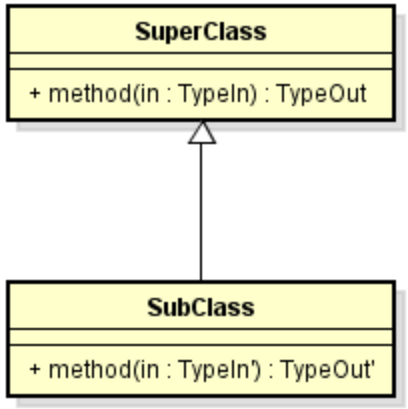
Contracts und Vererbung

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 Cart::total: Integer

derive: product.price->sum()

… oder durch einfache Operationen berechnet werden

context Cart::luxuryGood(p : Integer) : Set(Integer)

body: self.product->select(price > p).productId

Übersetzung von OCL Queries

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

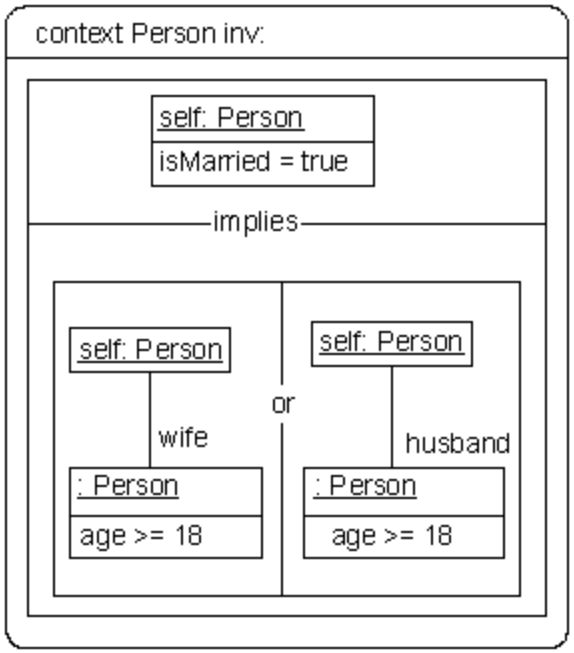
context Cart::luxuryGood(p : Integer) : Set(Integer)
body: self.product->select(price > p).productId

SELECT productId FROM Products WHERE price > 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

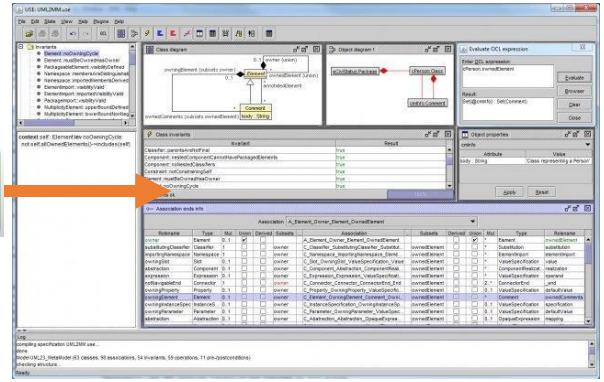


http://www.user.tu-berlin.de/o.runge/tfs/projekte/vocl/index.html

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



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?