

---

# Vorlesung „ Methoden des Software Engineering“

Block C „Formale Methoden“

## Spezifikation objektbasierter Systeme mit OCL

Martin Wirsing

Einheit C.2, 7.12.2004

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

## Ziele

- 
- Objektorientierte Systeme mit Hilfe von Invarianten und Zusicherungen spezifizieren lernen
  - Verfeinerung von Spezifikationen und Korrektheit von Implementierungen nachweisen lernen

# Eigenschaften von Transitionssystemen

---

## Invarianten

Invarianten spielen die Rolle eines Vertrags zwischen der Implementierung und der Benutzung einer Menge von Klassen. Durch Invarianten wird der Zustandsraum eingeschränkt.

## Zusicherungen

Durch Vor- und Nachbedingungen einer Operation  $m$  spezifiziert man die Menge der erlaubten Zustandsübergänge für die Implementierung von  $m$ .

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# OCL

---

## OCL (Object Constraint Language)

- Ursprünglich entwickelt bei IBM (1995) aufgrund von Erfahrungen mit der (formalen) OO Entwicklungsmethode Syntropy
- Sprache zur formalen Spezifikation von Constraints für UML
- Deklarative streng getypte Sprache zur Formulierung boolescher Ausdrücke  
≅ Logik erster Stufe mit Quantoren über endlichen Mengen

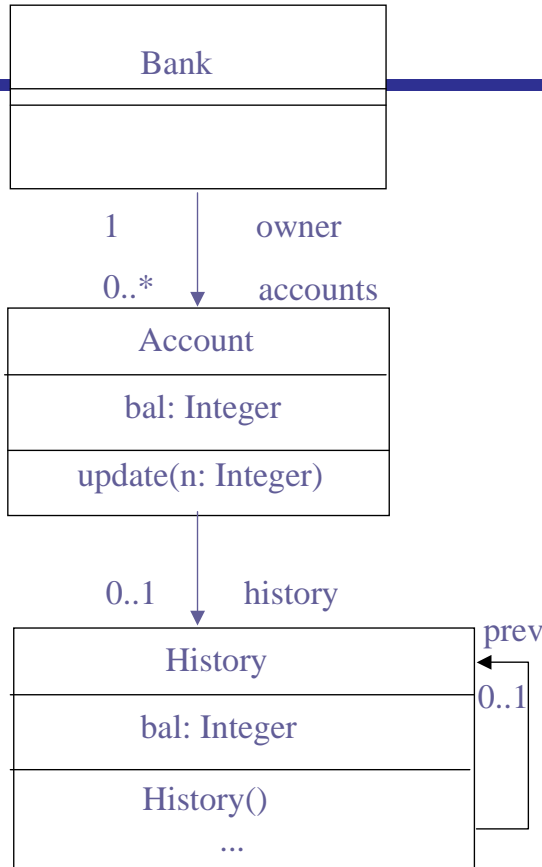
## Anwendungen von OCL

### Spezifikation von

- Invarianten und Vor/Nachbedingungen
- Bedingungen in Statecharts
- Constraints des UML Metamodells

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

**Beispiel: Bank**



context Bank  
 inv: accounts →forall(a | a.owner = self)

context Account  
 inv: bal ≥ 0

context Account: : update(n)  
 pre: bal + n ≥ 0  
 post: bal = bal@pre + n      and  
       history.oclIsNew()      and  
       history.bal = bal@pre  
       history.prev=history@pre

**Typen und Operationen von OCL**

**OCL-Syntax (vereinfacht)**

Integer, Real, Boolean, String  
 C, Set( C), Sequence( C), ...

x.bal

=

not, and, implies,...

set → forall(x | P)

set → select(x | P)

C.allInstances()

„alle existierenden Objekte der Klasse C“

Zusätzlich, in Nachbedingungen

x.bal@pre

x.oclIsNew()

Grunddatentypen

Klassen C und Kollektionstypen

Instanzvariable

Gleichheit

Junktoren

beschränkte Quantifizierung

≅ ∃ x ∈ set. P

≅ {x ∈ set | P}

„vorheriger“ Wert von x.bal

x bezeichnet neues Objekt

# Semantik von OCL: OCL-Algebra

Wir definieren eine **Algebra**  $\mathcal{O}$  zur Interpretation der Datentypen und Operationen von OCL

## Interpretation der Sorten

$$\begin{array}{ll} \mathcal{O}_{\text{Integer}} = \mathbb{Z} & \mathcal{O}_{\text{Boolean}} = \mathbb{B} = \{\text{true}, \text{false}\} \\ \mathcal{O}_{\text{Real}} = \mathbb{R} & \mathcal{O}_{\text{String}} = A^* \end{array} \quad \text{wobei } A \text{ ein Alphabet}$$

$$\mathcal{O}_C = \text{Old}_C \cup \{\text{null}\}$$

wobei  $\text{Old}_C$  die (abzählbare) Menge der Objektidentifikatoren der Klasse  $C$  ist

$$\begin{array}{ll} \mathcal{O}_{\text{Set}(T)} = F(\mathcal{O}_T) & \text{„endliche Mengen mit Elementen aus } \mathcal{O}_T\text{“} \\ \mathcal{O}_{\text{Sequence}(T)} = \mathcal{O}_T^* & \text{„endliche Folgen mit Elementen aus } \mathcal{O}_T\text{“} \end{array}$$

Die Operationen werden definiert Typen mit  $\perp$ -Element:

$$T_{\perp} = \mathcal{O}_T \cup \{\perp\} \quad \text{für jeden Typ } T$$

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# Semantik von OCL: OCL-Algebra

## Interpretation der Booleschen Operationen

haben die übliche Bedeutung (mit strikter Interpretation), z.B.

$$\text{not}^{\circ}: B_{\perp} \rightarrow B_{\perp}$$

$$\text{not}^{\circ}(b) = \begin{cases} \text{false} & \text{falls } b = \text{true} \\ \text{true} & \text{falls } b = \text{false} \\ \perp & \text{falls } b = \perp \end{cases}$$

$$\text{and}^{\circ}: B_{\perp} \times B_{\perp} \rightarrow B_{\perp}$$

$$b_1 \text{ and}^{\circ} b_2 = \begin{cases} \text{true} & \text{falls } b_1 = \text{true} \text{ und } b_2 = \text{true} \\ \text{false} & \text{falls } b_1 = \text{false} \text{ oder } b_2 = \text{false} \\ \perp & \text{sonst} \end{cases}$$

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

## Semantik von OCL: OCL-Algebra

Eine Operation  $op: T_1 \times \dots \times T_n \rightarrow T$  heißt **strikt**, wenn gilt:

$$op^o(x_1, \dots, x_n) = \perp \text{ falls } x_1 = \perp \text{ oder } \dots \text{ oder } x_n = \perp$$

### Interpretation der Operationen

Arithmetische Operationen und Mengenoperationen werden standardmäßig als strikt interpretiert und haben die übliche Bedeutung, z.B.

$$x =^o y = \begin{cases} \text{true} & \text{falls } x = y \text{ und } x \neq \perp \text{ und } y \neq \perp \\ \text{false} & \text{falls } x \neq y \text{ und } x \neq \perp \text{ und } y \neq \perp \\ \perp & \text{sonst} \end{cases}$$

$$s \rightarrow \text{includes}^o(x) = \begin{cases} \text{true} & \text{falls } x \in s \text{ und } x \neq \perp \text{ und } s \neq \perp \\ \text{false} & \text{falls } x \notin s \text{ und } x \neq \perp \text{ und } s \neq \perp \\ \perp & \text{sonst} \end{cases}$$

$$s \rightarrow \text{including}^o(x) = \begin{cases} s \cup \{x\} & \text{falls } x \neq \perp \text{ und } s \neq \perp \\ \perp & \text{sonst} \end{cases}$$

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

## Semantik von OCL: Systemzustand

### Systemzustand

Der Systemzustand  $\sigma$  eines Klassendiagramms  $\Delta$  ist gegeben durch

- ⊗ die Menge  $C_\sigma$  der existierenden Objekte (für jede Klasse  $C$  aus  $\Delta$ )
- ⊗ ein Objektdiagramm

definiert durch die Belegung  $\sigma_{\text{Val}}$  der Instanzvariablen

so dass  $\sigma_{\text{Val}}$  nur Objekte aus  $C_\sigma$  als Werte besitzt und typkorrekt ist, d.h.

für jedes  $o \in \text{OID}_C$  und Attribut  $_a: C \rightarrow T$ :

- ⊗  $\sigma_{\text{Val}}(o.a) \in T^o_\perp$
- ⊗  $\sigma_{\text{Val}}(o.a) \neq \perp$  gdw  $o \in C_\sigma$
- ⊗  $\sigma_{\text{Val}}(o.a) \in T_\sigma \cup \{\text{null}\}$ , falls  $o \in C_\sigma$  und  $T$  eine Klasse von  $D$
- ⊗  $\sigma_{\text{Val}}(o.a) \subset T_\sigma \cup \{\text{null}\}$ , falls  $o \in C_\sigma$  und  $T$  ein Kollektionstyp von  $D$

Außerdem benötigt man

eine Belegung  $\beta$  der (in OCL-Ausdrücken auftretenden) Variablen.

# Semantik von OCL: Interpretation der Ausdrücke

## Interpretation der OCL-Ausdrücke

- OCL-Ausdrücke werden über der Algebra  $O$  interpretiert und machen Aussagen über einen Systemzustand  $\sigma$  und dessen Vorgängerzustand  $\sigma@pre$
- Wir definieren die Semantik  $[[ e ]]$  eines OCL-Ausdrucks  $e$  induktiv über die syntaktische Struktur des Ausdrucks.
- Die Interpretationsfunktion  $[[ \_ ]]$  hängt ab von Zustand  $\sigma$ , Vorzustand  $\sigma@pre$  und der Belegung  $\beta$  der Variablen von  $e$ . (Der besseren Lesbarkeit halber verzichten wir im Folgenden meist auf diese Parameter ebenso wie auf den Index von  $\sigma_{val}$ ).

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# Semantik von OCL: Interpretation der Ausdrücke

$[[x]]_{\sigma, \sigma@pre, \beta}$	$= \beta(x)$	für jede Variable $x$
$[[ e.a ]]$ <sub><math>\sigma@pre, \sigma, \beta</math></sub>	$= \sigma([[e]]_{\sigma@pre, \sigma, \beta}.a)$	für jedes Attribut $a$
$[[ e.a@pre ]]$ <sub><math>\sigma@pre, \sigma, \beta</math></sub>	$= \sigma@pre([[e]]_{\sigma@pre, \sigma, \beta}.a)$	für jedes Attribut $a$
$[[ op(e_1, \dots, e_n) ]]$ <sub><math>\sigma@pre, \sigma, \beta</math></sub>	$= op^O([[e_1]]_{\sigma@pre, \sigma, \beta}, \dots, [[e_n]]_{\sigma@pre, \sigma, \beta})$	für jede OCL-Operation, z.B.
$[[ e_1 = e_2 ]]$ <sub><math>\sigma@pre, \sigma, \beta</math></sub>	$= [[e_1]]_{\sigma@pre, \sigma, \beta} =^O [[e_2]]_{\sigma@pre, \sigma, \beta}$	
$[[ C.allInstances() ]]$ <sub><math>\sigma@pre, \sigma, \beta</math></sub>	$= C_\sigma$	für alle Klassen $C$ aus $\Delta$
$[[ e.ocllsNew() ]]$ <sub><math>\sigma@pre, \sigma, \beta</math></sub>	$= \begin{cases} \text{true} & \text{falls } [[e]]_{\sigma@pre, \sigma, \beta} \in C_\sigma \text{ und } [[e]]_{\sigma@pre, \sigma, \beta} \notin C_{\sigma@pre} \\ \perp & \text{falls } [[e]]_{\sigma@pre, \sigma, \beta} = \perp \\ \text{false} & \text{sonst} \end{cases}$	

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# Vor-/Nachbedingungen

Vor-/Nachbedingungen schränken das mögliche Verhalten von Operationen ein:

- Die Vorbedingung muss erfüllt sein, wenn die Operation aufgerufen wird.
- Die Nachbedingung muss nach Beendigung der Ausführung der Operation erfüllt sein.

# Zusicherungsspezifikation

Sei  $SP_m = \text{context } C :: m(x_1: T_1, \dots, x_n: T_n)$

pre: PRE      post: POST

- $SP_m$  heißt **Zusicherungsspezifikation (Operation specification, Vor-/Nachbedingungsspez.)** und definiert eine Transitionsrelation, deren Vorzustände PRE und deren Nachzustände POST erfüllen.
- Die Semantik von  $SP_m$  ist durch die folgende Transitionsrelation gegeben:

$$\sigma@pre \xrightarrow{o.m(v_1, \dots, v_n)} \sigma \in [[SP_m]] \text{ gdw}$$

$$[[PRE]]_{\sigma@pre, \sigma@pre, \beta} = \text{true} \text{ und } [[POST]]_{\sigma@pre, \sigma, \beta} = \text{true}$$

$$\text{für alle } \beta \text{ mit } \beta(\text{self}) = o, \beta(x_1) = v_1, \dots, \beta(x_n) = v_n$$

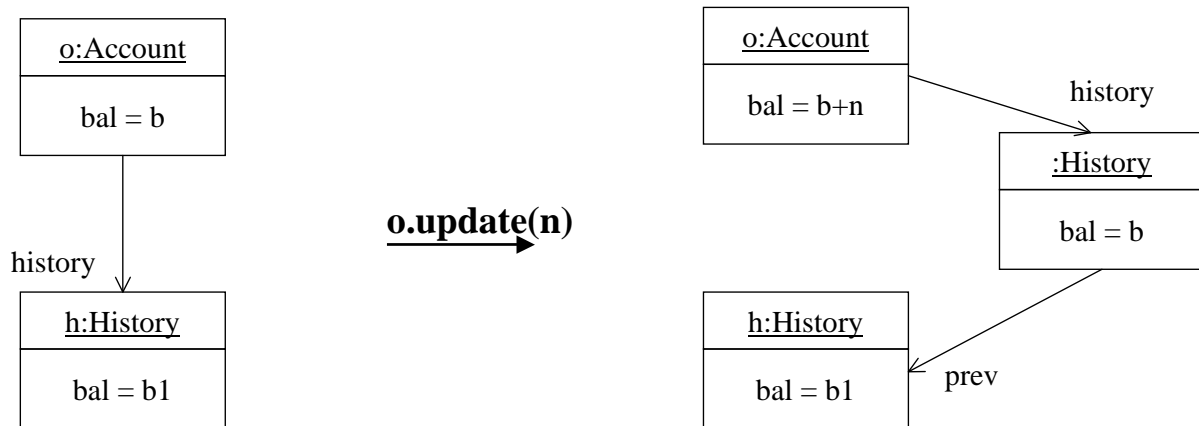
# Transitionssystem für Bank (objektorientiert)

context Account : : update(n)

pre:  $bal + n \geq 0$

post:  $bal = bal@pre + n$  and  $history.oclIsNew()$  and  
 $history.bal = bal@pre$  and  $history.prev = history@pre$

definiert folgendes Transitionssystem für alle  $b, n$  mit  $b+n \geq 0$ :



Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# Vor-/Nachbedingungen als Vertrag

Eine Vor-/Nachbedingungsspezifikation kann als Vertrag dienen zwischen

- einem Kunden, der die Operation benutzt und
- einem Programmierer, der die Operation realisiert.

## Verantwortung des Kunden

Der Kunde ruft die Operation nur auf, wenn die Vorbedingung erfüllt ist

## Verantwortung des Programmierers

Der Programmierer garantiert, dass

nach Ausführung der Operation die Nachbedingung gilt,  
 falls die Operation in einem Zustand aufgerufen wurde,  
 der die Vorbedingung erfüllt.

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# Vor-/Nachbedingungen als Vertrag

## Bemerkungen:

- Der Vertrag sagt **nichts** aus über Situationen, in denen die Vorbedingung **nicht** erfüllt ist.
- Bei einer query-Operation muss der Programmierer auch garantieren, dass der Systemzustand unverändert bleibt.
- Bei einem Konstruktor muss der Programmierer auch garantieren, dass ein neues Objekt erzeugt wird.

# Verfeinerung

- Eine Zusicherungsspezifikation SPK heißt **(Operations-) Verfeinerung** einer Zusicherungsspezifikation SPA, falls
  - SPK alle Eingaben akzeptiert, die auch SPA akzeptiert und
  - SPK determinierter ist als SPA in Bezug auf Eingaben von SPA
- **Formal**
  - $\text{dom} [[\text{SPA}]] \subset \text{dom} [[\text{SPK}]]$
  - $(\text{dom} [[\text{SPA}]] \triangleleft [[\text{SPK}]]) \subset [[\text{SPA}]]$

wobei für ein Transitionssystem  $\Gamma = (Z, A, T)$  und eine Menge  $Y$  von Zuständen

$\text{dom } \Gamma = \{(z_1, a) \mid \exists z_2 \in Z. (z_1, a, z_2) \in T\}$  **Definitionsbereich von  $\Gamma$**

$Y \triangleleft \Gamma = \{(z_1, a, z_2) \in T \mid z_1 \in Y\}$

**Einschränkung des Def.bereichs von  $\Gamma$  auf Elemente von  $Y$ .**

# Verfeinerung

- **Folgerung**

Sei  $SPA = \text{context } C: m(x:T) \text{ pre: } PRE_A \text{ post: } POST_A$

und  $SPK = \text{context } C: m(x:T) \text{ pre: } PRE_K \text{ post: } POST_K$

Dann ist  $SPK$  (Operations-) Verfeinerung von  $SPA$ , falls gilt:

1.  $PRE_A \Rightarrow PRE_K$  und
2.  $PRE_A@pre \wedge POST_K \Rightarrow POST_A$

# Verfeinerung: Beispiel Account

- **Spezifikation ohne Anforderung an die Kontoentwicklung (Signatur mit History)**

```
SPA_update = context Account::update(n)
             pre: bal + n ≥ 0
             post: bal = bal@pre + n
```

- **Spezifikation mit Anforderungen an die Kontoentwicklung (gleiche Signatur)**

```
SPK_update = context Account::update(n)
             pre: bal + n ≥ 0
             post: bal = bal@pre + n      and
                  history.oclIsNew()    and
                  history.bal = bal@pre
                  history.prev=history@pre
```

- $SPK_{update}$  ist Operationsverfeinerung von  $SPA_{update}$   
 denn  $PRE_{SPA} \Rightarrow PRE_{SPK}$  wg.  $PRE_{SPA} = PRE_{SPK}$   
 und  $PRE_A@pre \wedge POST_K \Rightarrow POST_A$  wg.  $POST_K \Rightarrow POST_A$

# Verfeinerung: Bemerkungen

---

- Zur Verfeinerung eines Klassendiagramms müssen natürlich **alle Methoden und Konstruktoren verfeinert** werden.
- Bei der **Operationsverfeinerung** gibt es **keine Änderung der Signatur**; insbesondere ändert sich nichts an den Elementen des Systemzustands.
- Bei einem **Wechsel der Datenstruktur** haben der abstrakte und der konkrete Datentyp unterschiedliche Signaturen. Es wird eine Verfeinerungsrelation zwischen den Werten des abstrakten und des konkreten Datentyps definiert; die Bedingungen der Operationsverfeinerung werden modulo Verfeinerungsrelation und Signaturwechsel formuliert.

(Für Genaueres siehe Grundlagen der Systementwicklung)

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# Invarianten

---

- Eine **Invariante** ist eine Bedingung, die die Menge der möglichen Systemzustände einschränkt.
- Eine Invariante sollte deshalb **vor und nach jedem Aufruf einer (öffentlichen) Methode** gelten.

**Beispiele:**

**context** Account

**inv:**  $bal+n \geq 0$

**context** Bank

**inv:**  $accounts \rightarrow \text{forAll}(a \mid a.owner = self)$

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# Klasseninvarianten und Komponenteninvarianten

- Eine **Klasseninvariante** einer Klasse C ist ein Boolescher OCL–Ausdruck, der die möglichen Zustände einschränkt, in denen Objekte von C von einer **anderen** Klasse gesehen werden können.

Beispiel: **context** Account

**inv:** bal+n >= 0

- Eine **Komponenteninvariante** betrifft im Allgemeinen mehrere Klassen (einer Komponente) und ist ein Boolescher OCL–Ausdruck, der die möglichen Objektkonfigurationen einschränkt, die von außerhalb der Komponente gesehen werden können.

Beispiel: **context** Component

**inv:** Bank.allInstances →forall( b |  
b.accounts →forall( a | a.owner =b ) )

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# Implizite Invarianten durch Assoziationen

Die Assoziationen eines Klassendiagramms induzieren implizite Invarianten:

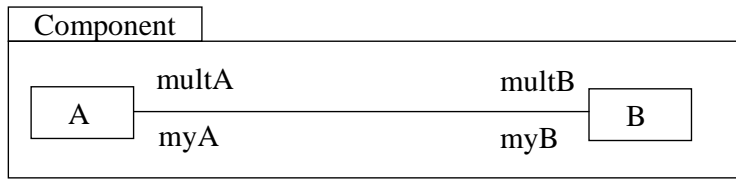


- **multB = 0..1** ist vollständig formalisiert durch **\_.myB: A → B**
- **multB = 1** induziert die Klasseninvariante  
**context** A **inv:** myB <> null
- **multB = \*** ist vollständig formalisiert durch **\_.myB: A → Set(B)**
- **multB = 1..\*** induziert die Klasseninvariante  
**context** A **inv:** myB →exists( b | b <> null)

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

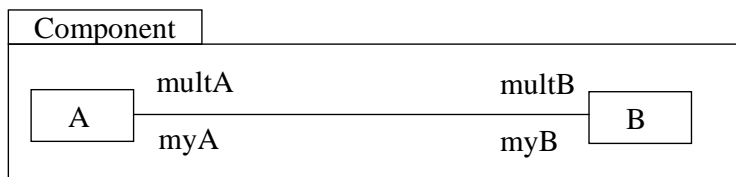
# Implizite Invarianten durch Assoziationen

## Bidirektionale Assoziationen:



- multA = multB = 0..1** induziert die Komponenteninvariante  
**context** Component  
**inv:** A.allInstances() →forall( a | a.myB <>null implies a.myB.myA =a) and  
 B.allInstances() →forall( b | b.myA <>null implies b.myA.myB =b)
- multA = multB = 1** induziert die Komponenteninvariante  
**context** Component  
**inv:** A.allInstances() →forall( a | a.myB <>null and a.myB.myA =a) and  
 B.allInstances() →forall( b | b.myA <>null and b.myA.myB =b)

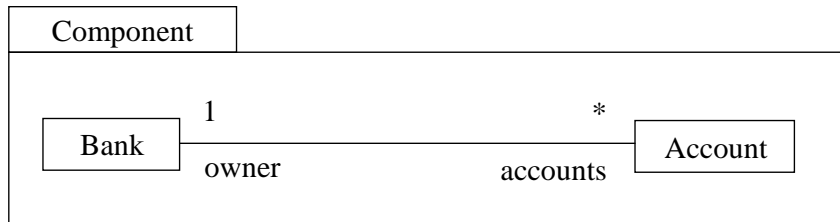
# Implizite Invarianten durch Assoziationen



- multA = multB = \*** induziert die Komponenteninvariante  
**context** Component  
**inv:** A.allInstances() →forall( a |  
 a.myB →forall( b | b<>null implies b.myA →includes(a)) and  
 B.allInstances() →forall( b |  
 b.myA →forall( a | a<>null implies a.myB →includes(b))
- Alle anderen Fälle werden analog behandelt

# Implizite Invarianten durch Assoziationen

## Beispiel:



- **Komponenteninvariante**

context Component

inv: Bank.allInstances()  $\rightarrow$ forAll( b |

b.accounts  $\rightarrow$ forAll( a | a.owner =b) and

Account.allInstances()  $\rightarrow$ forAll( a | a.owner.accounts  $\rightarrow$ includes(a))

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# Zusicherungsspezifikation und Invarianten

- Sei INV die Konjunktion aller Invarianten eines Klassendiagramms  $\Delta$  und  $SP_m =$  context C :: m(x1: T1, ..., xn: Tn)  
pre: PRE post: POST  
eine Zusicherungsspezifikation von  $\Delta$ .
- Die **von INV induzierte Zusicherungsspezifikation**  $SP_m^{INV}$  lautet:  
 $SP_m^{INV} =$  context C :: m(x1: T1, ..., xn: Tn)  
pre: PRE and INV post: POST and INV
- Die Spezifikation ist **wohlgeformt**, wenn der Definitionsbereich von Nachbedingung und Invariante in der Vorbedingung (+Invariante) enthalten ist, d.h.  
 $dom([[POST \text{ and } INV]]) \subset [[PRE \text{ and } INV]]$  bzw.  
 $PRE@pre \text{ and } INV @pre \Rightarrow dom(POST \text{ and } INV)$

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# Zusicherungsspezifikation und Invarianten: Beispiel

- Die BankAccount-Spezifikation

**context** Account

**inv:**  $bal \geq 0$

**context** Account::update(n)

**pre:**  $bal + n \geq 0$

**post:**  $bal = bal@pre + n$

ist wohlgeformt:

-  $\text{dom}(\text{POST and INV}) = \text{dom}(bal=bal@pre+n \text{ and } bal \geq 0)$

$\Leftrightarrow bal@pre+n \geq 0$

-  $\text{PRE}@pre \text{ and } \text{INV}@pre \Leftrightarrow (bal@pre + n \geq 0 \text{ and } bal@pre \geq 0)$

Also gilt:  $\text{PRE}@pre \text{ and } \text{INV}@pre \Rightarrow \text{dom}(\text{POST and INV})$

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# Korrektheit der Implementierung

- Eine **Implementierung**  $m$  heißt **korrekt** bzgl. einer wohlgeformten Spezifikation  $SP^{INV}_m$ , wenn die Implementierung eine

Operationsverfeinerung von  $SP^{INV}_m$  ist,

d.h. wenn  $([[\text{PRE and INV}]] \subset \text{dom}([[m]]))$  und

$([[\text{PRE and INV}]] \triangleleft [[m]] \subset [[SP^{INV}_m]])$

- In anderen Worten:

$(\text{PRE and INV}) \Rightarrow \text{dom}(m)$                       **und**

$\text{PRE}@pre \text{ and } \text{INV}@pre \wedge \underline{m} \Rightarrow \text{POST and INV}$

wobei  $\underline{m}$  die Spezifikation des Transitionssystems von  $m$  bezeichnet.

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# Korrektheit der Implementierung: Beispiel

---

- Die folgende **BankAccount-Implementierung**

```
class Account
{   int bal;   History history;   Bank owner;
    ...
    void update(int n)
    {   History h1 = new History();
        h1.bal = bal;   h1.prev = history;
        history = h1;
        bal = bal+n;
    }
}
```

besitzt die Spezifikation **update** =

```
pre: true
post: bal = bal@pre + n and history.oclIsNew() and
      history.bal = bal@pre and history.prev=history@pre
```

Es gilt:  $(PRE \text{ and } INV) \Rightarrow \text{dom}(\underline{m}) (= \text{true})$                       **und**  
 $PRE@pre \text{ and } INV @pre \wedge \underline{m} \Rightarrow POST \text{ and } INV$

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# Korrektheit der Implementierung: Beispiel

---

## Die Implementierung ist korrekt bzgl. der Spezifikation

<b>context</b> Account	<b>context</b> Account::update(n)
<b>inv:</b> bal ≥ 0	<b>pre:</b> bal + n ≥ 0
	<b>post:</b> bal = bal@pre + n

Denn es gilt:

$(PRE \text{ and } INV) \Rightarrow \text{dom}(\underline{\text{update}}) (\equiv \text{true})$                       **und**  
 $PRE@pre \text{ and } INV @pre \text{ and } \underline{\text{update}}$   
 $\equiv \text{bal@pre} + n \geq 0 \text{ and } \text{bal@pre} \geq 0 \text{ and } \underline{\text{update}}$   
 $\Rightarrow \text{bal} = \text{bal@pre} + n \text{ and } \text{bal@pre} + n \geq 0$   
 $\Rightarrow POST \text{ and } INV$

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# Zusicherungsspezifikation und Invarianten

---

## Bemerkung

- Die Forderung, dass jede Methode alle Invarianten erhält ist im Allgemeinen zu stark.
- Man muss dies nur für Methoden fordern, die außerhalb einer Komponente sichtbar sind (public visibility).
- Methoden, auf die nur innerhalb einer Komponente von anderen Klassen zugegriffen werden kann, müssen nur die Klasseninvarianten erhalten (default visibility in Java).
- Private Methoden einer Klasse müssen überhaupt keine Invarianten erhalten.

(Für eine genaue Untersuchung siehe Hennicker: FOOSE).

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# Zusammenfassung

---

**OCL** ist eine Spezifikationssprache zur Formulierung von Invarianten und Vor- und Nachbedingungen

## Invarianten

- Invarianten spielen die Rolle eines Vertrags zwischen der Implementierung und der Benutzung einer Menge von Klasse. Durch Invarianten wird der Zustandsraum eingeschränkt.

## Zusicherungen

Durch Vor- und Nachbedingungen einer Operation  $m$  spezifiziert man die Menge der erlaubten Zustandsübergänge für die Implementierung von  $m$ .

## Verfeinerung

Durch (Operations-)Verfeinerung zeigt man die Korrektheit von Implementierungen.

Methoden des Software Engineering (c) 2004, Koch, Störrle, Wirsing, LMU München

# Literatur

---

- **Wirsing: Grundlagen der Systementwicklung, Kap. 8**
- **Hennicker: Formale objekt-orientierte Software-Entwicklung,  
Kap. 2-5**