

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, Störrie, Wirsing, LMU München

Ziele

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

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

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, Störrie, Wirsing, LMU München

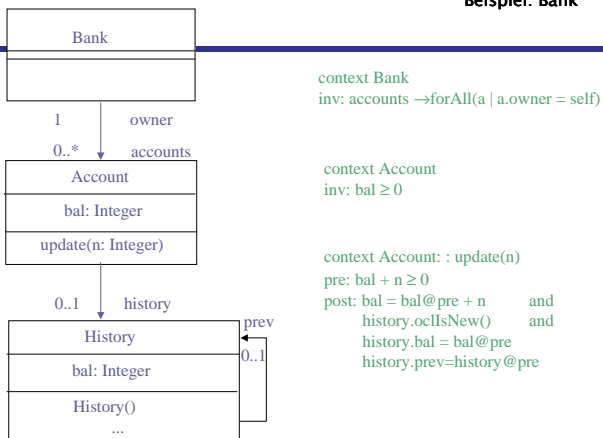
OCL**OCL (Object Constraint Language)**

- 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, Störrie, Wirsing, LMU München

Beispiel: Bank

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

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()

Zusätzlich, in Nachbedingungen

x.bal@pre

x.oclIsNew()

Grunddatentypen

Klassen C und Kollektionstypen

Instanzvariable

Gleichheit

Junktoren

beschränkte Quantifizierung

≡ $\forall x \in \text{set. } P$ ≡ $\{x \in \text{set} \mid P\}$

„alle existierenden Objekte der Klasse C“

„vorheriger“ Wert von x.bal

x bezeichnet neues Objekt

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

Semantik von OCL: OCL-Algebra

Wir definieren eine **Algebra O** zur Interpretation der Datentypen und Operationen von OCL

Interpretation der Sorten

$$\begin{array}{l} O_{\text{Integer}} = \mathbb{Z} \quad O_{\text{Boolean}} = \mathbb{B} = \{\text{true}, \text{false}\} \\ O_{\text{Real}} = \mathbb{R} \quad O_{\text{String}} = A^* \end{array} \quad \text{wobei } A \text{ ein Alphabet}$$

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

$$\begin{array}{l} O_{\text{Set}(T)} = F(O_T) \quad \text{„endliche Mengen mit Elementen aus } O_T \text{“} \\ O_{\text{Sequence}(T)} = O_T^* \quad \text{„endliche Folgen mit Elementen aus } O_T \text{“} \end{array}$$

O_T

Die Operationen werden definiert Typen mit \perp -Element:
 $T_{\perp} = O_T \cup \{\perp\}$ für jeden Typ T

Semantik von OCL: OCL-Algebra

Interpretation der Booleschen Operationen

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

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

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

Semantik von OCL: OCL-Algebra

Eine Operation $op: T_1 \times \dots \times T_n \rightarrow T$ heißt **strik**, wenn gilt:
 $op^{\circ}(x_1, \dots, x_n) = \perp$ falls $x_1 = \perp$ oder ... oder $x_n = \perp$

Interpretation der Operationen

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

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

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

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

Semantik von OCL: Systemzustand

Systemzustand

Der Systemzustand σ eines Klassendiagramms Δ ist gegeben durch

- § die Menge C_{σ} der existierenden Objekte (für jede Klasse C aus Δ)
- § ein Objektdiagramm

definiert durch die Belegung σ_{Val} der Instanzvariablen

so dass σ_{Val} nur Objekte aus C_{σ} als Werte besitzt und typkorrekt ist, d.h.

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

- § $\sigma_{\text{Val}}(o.a) \in T_{\perp}$
- § $\sigma_{\text{Val}}(o.a) \neq \perp$ 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
- § $\sigma_{\text{Val}}(o.a) \in T_{\sigma} \cup \{\text{null}\}$, falls $o \in C_{\sigma}$ und T ein Kollektionstyp von D

Außerdem benötigt man eine Belegung β 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 σ 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 σ , Vorzustand $\sigma@pre$ und der Belegung β der Variablen von e. (Der besseren Lesbarkeit halber verzichten wir im Folgenden meist auf diese Parameter ebenso wie auf den Index von σ_{Val}).

Semantik von OCL: Interpretation der Ausdrücke

$$[[x]]_{\sigma, \sigma@pre, \beta} = \beta(x) \quad \text{für jede Variable } x$$

$$\begin{array}{l} [[e.a]]_{\sigma@pre, \sigma, \beta} = \sigma([[e]]_{\sigma@pre, \sigma, \beta}.a) \quad \text{für jedes Attribut } a \\ [[e.a@pre]]_{\sigma@pre, \sigma, \beta} = \sigma@pre([[e]]_{\sigma@pre, \sigma, \beta}.a) \quad \text{für jedes Attribut } a \end{array}$$

$$[[op(e_1, \dots, e_n)]_{\sigma@pre, \sigma, \beta} = op^{\circ}([[e_1]]_{\sigma@pre, \sigma, \beta}, \dots, [[e_n]]_{\sigma@pre, \sigma, \beta}) \\ \text{für jede OCL-Operation, z.B.} \\ [[e_1 = e_2]]_{\sigma@pre, \sigma, \beta} = [[e_1]]_{\sigma@pre, \sigma, \beta} =^{\circ} [[e_2]]_{\sigma@pre, \sigma, \beta}$$

$$[[C.\text{allInstances}()]]_{\sigma@pre, \sigma, \beta} = C_{\sigma} \quad \text{für alle Klassen } C \text{ aus } \Delta$$

$$[[e.\text{oclIsNew}()]]_{\sigma@pre, \sigma, \beta} = \begin{cases} \text{true} & \text{falls } [[e]]_{\sigma@pre, \sigma, \beta} \in C_{\sigma} \text{ und } [[e]]_{\sigma@pre, \sigma, \beta} \notin C_{\sigma@pre, \sigma, \beta} \\ \perp & \text{falls } [[e]]_{\sigma@pre, \sigma, \beta} = \perp \\ \text{false} & \text{sonst} \end{cases}$$

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:

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

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

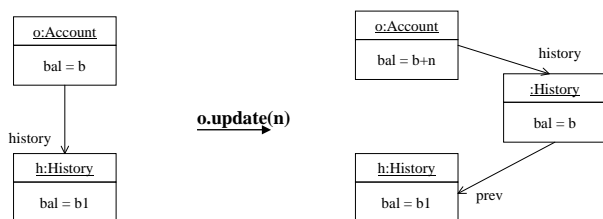
für alle β mit $\beta(\text{self}) = o, \beta(x_1) = v_1, \dots, \beta(x_n) = v_n$

Transitionssystem für Bank (objektorientiert)

context Account : : update(n)

pre: bal + n ≥ 0 post: bal = bal@pre + n and history.ocIsNew() and history.bal = bal@pre and history.prev=history@pre

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



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.

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**
 - $\text{dom} [[SPA]] \subset \text{dom} [[SPK]]$
 - $(\text{dom} [[SPA]] \triangleleft [[SPK]]) \subset [[SPA]]$

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

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

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

Einschränkung des Def.bereichs von Γ 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:

- $PRE_A \Rightarrow PRE_K$ und
- $PRE_A @ \text{pre} \wedge POST_K \Rightarrow POST_A$

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 $PRE_A @ \text{pre} \wedge POST_K \Rightarrow POST_A$ 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)

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 >= 0
```

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

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))

Implizite Invarianten durch Assoziationen

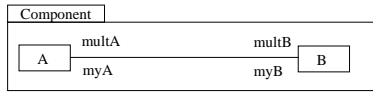
Die Assoziationen eines Klassendiagramms induzieren implizite Invarianten:



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

Implizite Invarianten durch Assoziationen

Bidirektionale Assoziationen:



- $\text{multA} = \text{multB} = 0..1$ induziert die Komponenteninvariante
context Component
inv: $A.\text{allInstances}() \rightarrow \text{forAll}(a \mid a.\text{myB} \langle \rangle \text{null implies } a.\text{myB}.\text{myA} = a)$ and
 $B.\text{allInstances}() \rightarrow \text{forAll}(b \mid b.\text{myA} \langle \rangle \text{null implies } b.\text{myA}.\text{myB} = b)$
- $\text{multA} = \text{multB} = 1$ induziert die Komponenteninvariante
context Component
inv: $A.\text{allInstances}() \rightarrow \text{forAll}(a \mid a.\text{myB} \langle \rangle \text{null and } a.\text{myB}.\text{myA} = a)$ and
 $B.\text{allInstances}() \rightarrow \text{forAll}(b \mid b.\text{myA} \langle \rangle \text{null and } b.\text{myA}.\text{myB} = b)$

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

Implizite Invarianten durch Assoziationen

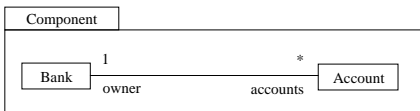


- $\text{multA} = \text{multB} = *$ induziert die Komponenteninvariante
context Component
inv: $A.\text{allInstances}() \rightarrow \text{forAll}(a \mid$
 $a.\text{myB} \rightarrow \text{forAll}(b \mid b \langle \rangle \text{null implies } b.\text{myA} \rightarrow \text{includes}(a))$ and
 $B.\text{allInstances}() \rightarrow \text{forAll}(b \mid$
 $b.\text{myA} \rightarrow \text{forAll}(a \mid a \langle \rangle \text{null implies } a.\text{myB} \rightarrow \text{includes}(b))$
- Alle anderen Fälle werden analog behandelt

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

Implizite Invarianten durch Assoziationen

Beispiel:



- **Komponenteninvariante**
context Component
inv: $Bank.\text{allInstances}() \rightarrow \text{forAll}(b \mid$
 $b.\text{accounts} \rightarrow \text{forAll}(a \mid a.\text{owner} = b)$ and
 $Account.\text{allInstances}() \rightarrow \text{forAll}(a \mid a.\text{owner}.\text{accounts} \rightarrow \text{includes}(a))$

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

Zusicherungsspezifikation und Invarianten

- Sei INV die Konjunktion aller Invarianten eines Klassendiagramms Δ und $SP_m = \text{context } C :: m(x1: T1, \dots, xn: Tn)$
pre: PRE **post:** POST
 eine Zusicherungsspezifikation von Δ .
- Die **von INV induzierte Zusicherungsspezifikation** SP_m^{INV} lautet:
 $SP_m^{INV} = \text{context } C :: m(x1: T1, \dots, xn: Tn)$
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.
 $\text{dom}([[\text{POST and INV}]]) \subset [[\text{PRE and INV}]]$ **bzw.**
 $\text{PRE}@pre \text{ and } INV @pre \Rightarrow \text{dom}(\text{POST and INV})$

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

Zusicherungsspezifikation und Invarianten: Beispiel

- **Die BankAccount-Spezifikation**
context Account **context** Account::update(n)
inv: $\text{bal} \geq 0$ **pre:** $\text{bal} + n \geq 0$
post: $\text{bal} = \text{bal@pre} + n$
- ist wohlgeformt:**
 - $\text{dom}(\text{POST and INV}) = \text{dom}(\text{bal} = \text{bal@pre} + n \text{ and } \text{bal} \geq 0)$
 $\Leftrightarrow \text{bal@pre} + n \geq 0$
 - $\text{PRE}@pre \text{ and } INV@pre \Leftrightarrow (\text{bal@pre} + n \geq 0 \text{ and } \text{bal@pre} \geq 0)$
- Also gilt: $\text{PRE}@pre \text{ and } INV@pre \Rightarrow \text{dom}(\text{POST and INV})$

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

Korrektheit der Implementierung

- Eine **Implementierung m** heißt **korrekt** bzgl. einer wohlgeformten Spezifikation SP_m^{INV} , wenn die Implementierung eine Operationsverfeinerung von SP_m^{INV} ist, d.h. wenn $[[\text{PRE and INV}]] \subset \text{dom}([[\text{m}]])$ und $[[\text{PRE and INV}]] \triangleleft [[\text{m}]] \subset [[\text{SP}_m^{INV}]]$
- In anderen Worten:
 $(\text{PRE and INV}) \Rightarrow \text{dom}(\underline{m})$ **und**
 $\text{PRE}@pre \text{ and } INV @pre \wedge \underline{m} \Rightarrow \text{POST and INV}$
 wobei \underline{m} die Spezifikation des Transitionssystems von m bezeichnet.

Methoden des Software Engineering (c) 2004, Koch, Störrie, 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 and INV) => dom(m) (= true) **und**
 PRE@pre and INV @pre ∧ m => POST and INV

Korrektheit der Implementierung: Beispiel

Die Implementierung ist korrekt bzgl. der Spezifikation

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

Denn es gilt:

(PRE and INV) => dom(update) (= true) **und**
 PRE@pre and INV @pre and update
 ≡ bal@pre + n ≥ 0 and bal@pre ≥ 0 and update
 => bal = bal@pre+n and bal@pre+n ≥ 0
 => POST and INV

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).

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.

Literatur

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