

Formale objektorientierte Software-Entwicklung

Prof. Dr. Rolf Hennicker

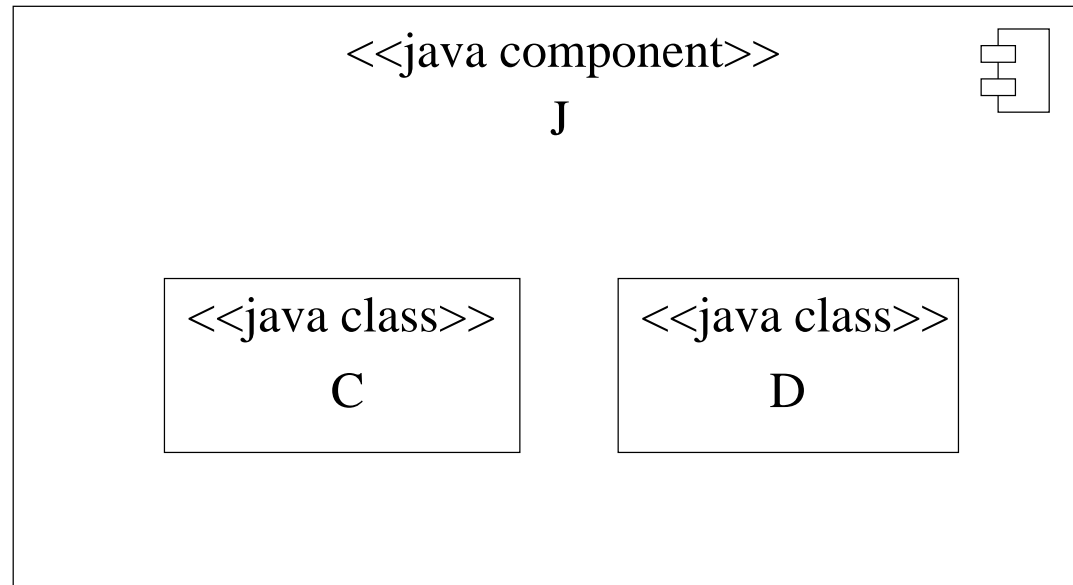
19.07.2006

Kapitel 5: Java Realisierungen von Komponentenspezifikationen

Ziele

- Wissen, was eine Java Komponente ist.
- Wissen, wann eine Java Komponente konform zu einer Klassensignatur Σ_{Δ} ist.
- Das durch eine Java Komponente induzierte Σ_{Δ} -Transitionssystem konstruieren können.
- Verstehen, wann eine Java Komponente eine korrekte Realisierung einer Komponentenspezifikation ist.
- Die (vereinfachten) Beweisverpflichtungen kennen, die (unter bestimmten Annahmen) die Korrektheit einer Java Realisierung garantieren.
- Das Theorem über die Korrektheit einer Java Realisierung kennen und verstehen.

5.1 Java Komponenten



Eine Java Komponente besteht aus einer Menge von Java Klassen.

Definition (Konformität):

Sei Σ_{Δ} eine Klassensignatur. Eine Java Komponente J ist konform zu Σ_{Δ} falls gilt:

1. Für jeden Klassennamen $C \in Class_{\Delta}$ existiert eine entsprechende Java Klasse c in J
2. Für jedes Attribut $(_ . a : C \rightarrow T) \in A_{\Delta}$ existiert eine entsprechende Instanzvariable (mit demselben Namen) in der zugehörigen Java Klasse c .
3. Für jede Operation $(op : C \times T_1 \times \dots \times T_n \rightarrow T) \in M_{\Delta} \cup Q_{\Delta}$ existiert eine entsprechende Methode in der zugehörigen Java Klasse c und für jeden Konstruktor $(C : T_1 \times \dots \times T_n \rightarrow C) \in Con_{\Delta}$ existiert ein entsprechender Konstruktor in c .
4. Für alle Klassen $C, B \in Class_{\Delta}$ ist B genau dann eine direkte Oberklasse von C (d.h. $C < B$ und $\nexists D \in Class_{\Delta} : C < D < B$) wenn die Java Klasse c vermöge "extends" direkt von der Java Klasse B erbt.
5. Die Sichtbarkeiten von Attributen, Rollen und Operationen in $A_{\Delta} \cup Opns_{\Delta}$ werden erhalten, d.h.
 - "-" wird abgebildet auf "private", "#" auf "protected"
 - "~" wird abgebildet auf Java-Default-Sichtbarkeit,
 - "+" wird abgebildet auf "public" innerhalb einer öffentlichen Java Klasse.
6. Für jede Queryoperation $(q : C \times T_1 \times \dots \times T_n \rightarrow T) \in Q_{\Delta}$ besitzt die entsprechende Java Methode keine Seiteneffekte.

Bemerkung:

1. Basistypen von OCL werden der Java Syntax entsprechend umbenannt, z.B. $Integer \mapsto int$.
2. OCL Kollektionstypen werden durch Java Kollektionstypen ersetzt.

Insbesondere gilt:

Für jeden Rollennamen $(_.a : C \rightarrow Set(T)) \in A_\Delta$ existiert ein entsprechendes Referenzattribut (z.B. `Set a;`) in `C.java`.

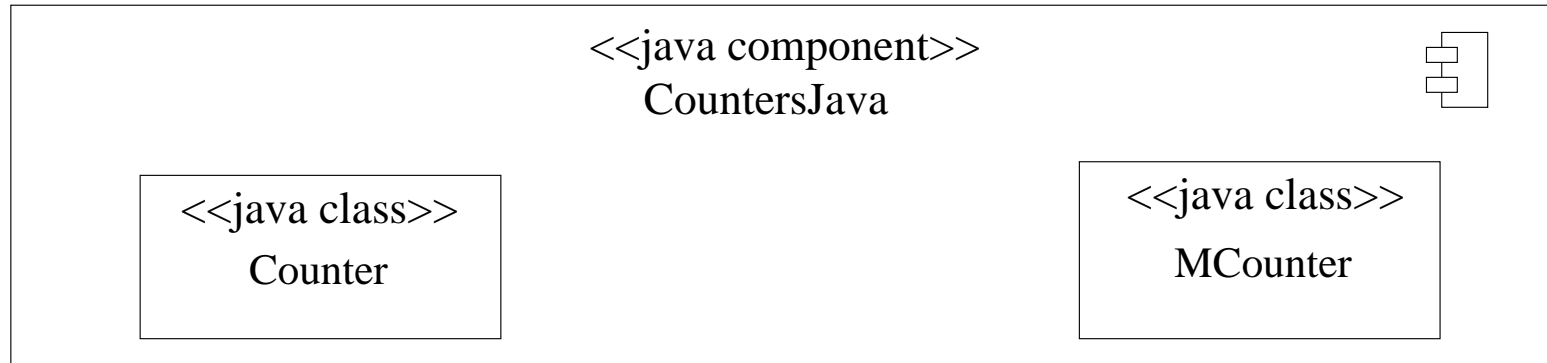
3. Die Argument- und Ergebnistypen von Operationen $op \in Opns_\Delta$ werden von den entsprechenden Java Methoden und Konstruktoren respektiert, z.B.

$$[createPoint : System \times Real \times Real \rightarrow Point] \mapsto [Point \ createPoint(double \ x, \ double \ y)]$$

$$[move : Point \times Real \times Real \rightarrow Void] \mapsto [void \ move(double \ mx, \ double \ my)]$$

$$[Point : Real \times Real \rightarrow Point] \mapsto [Point(double \ x, \ double \ y)]$$

Beispiel (Java Komponente für Counters):



```
package CountersJava;
public class Counter {
    protected int count;

    public Counter() {
        count = 0;
    }

    public void inc() {
        count++;
    }

    public void dec() {
        count--;
    }
}
```

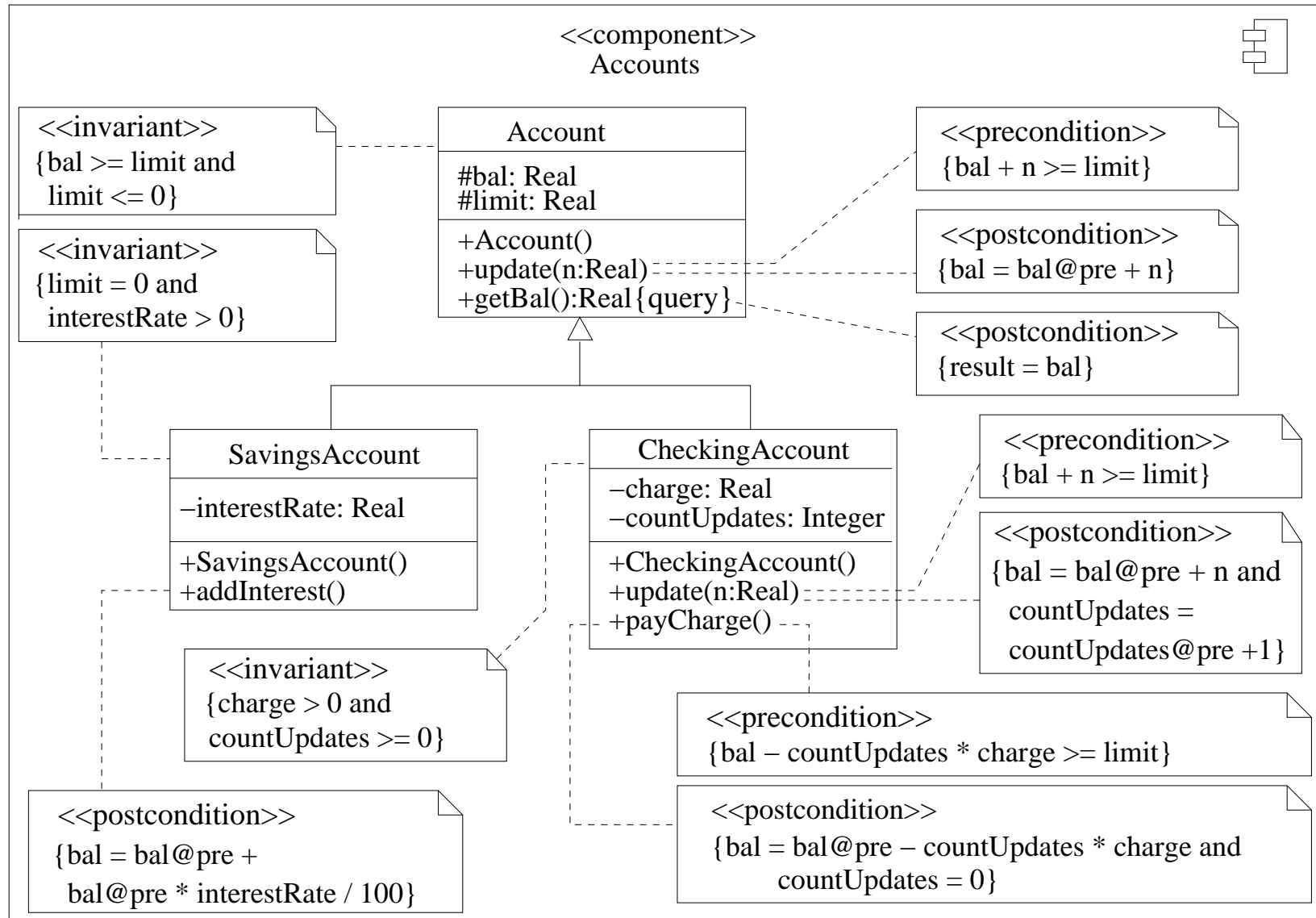
```
package CountersJava;
public class MCounter extends Counter {
    private int last;

    public MCounter() {
        count = 0;
        last = 0;
    }

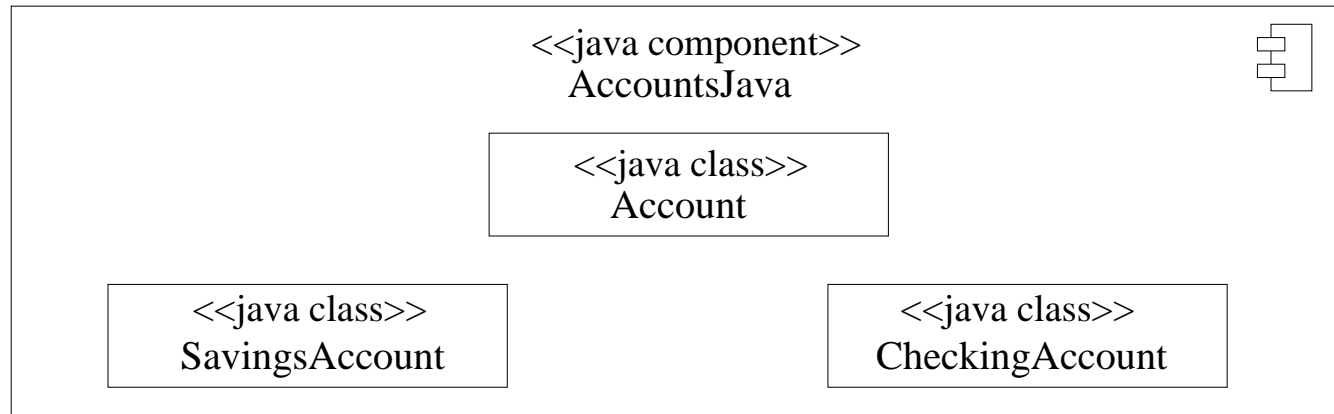
    public void inc() {
        last = count;
        count++;
    }

    public void dec() {
        last = count;
        count--;
    }
}
```

Beispiel (Komponentenspezifikation für Accounts):



Beispiel (Java Komponente AccountsJava):



<pre> package AccountsJava; public class Account { protected double bal; protected double limit; public Account() { bal = 0; limit = 0; } public void update(double n) { bal = bal + n; } public double getBal() { return bal; } } </pre>	<pre> package AccountsJava; public class SavingsAccount extends Account { private double interestRate; public SavingsAccount() { bal = 0; limit = 0; interestRate=2.5; } public void addInterest() { update(bal*interestRate/100); } } </pre>	<pre> package AccountsJava; public class SavingsAccount extends Account { private double interestRate; public SavingsAccount() { bal = 0; limit = 0; interestRate=2.5; } public void addInterest() { update(bal*interestRate/100); } } </pre>
---	---	---

```
package AccountsJava;
public class CheckingAccount extends Account
{
    private double charge;
    private int countUpdates;

    public CheckingAccount() {
        bal = 0; limit = -100;
        charge = 8.5;
        countUpdates = 0;
    }
    public void update(double n) {
        super.update(n);
        countUpdates = countUpdates + 1;
    }

    public void payCharge() {
        super.update(-countUpdates*charge);
        countUpdates = 0;
    }
}
```

Definition (Durch eine Java Komponente induziertes Σ_{Δ} -Transitionssystem):

Sei Σ_{Δ} eine Klassensignatur und J eine Java Komponente, die konform zu Σ_{Δ} ist.
 J induziert das Σ_{Δ} -Transitionssystem

$$\mathcal{T}_{\Delta}^J = (\text{State}_{\Delta}, \sigma_{init}, \text{Label}_{\Delta}, \Omega_{\Delta}, R_{\Delta}^J)$$

wobei R_{Δ}^J die kleinste Relation ist, die folgende Bedingungen erfüllt:

1. Für alle $(op : C \times T_1 \times \dots \times T_n \longrightarrow Void) \in M_\Delta$ mit zugehöriger Java Methode $void\ op(T_1\ x_1, \dots, T_n\ x_n)\ body$ der Klasse C ist

$$\sigma^- \xrightarrow{o.op_C(v_1, \dots, v_n)} \sigma \in R_\Delta^J \quad \text{falls}$$

$$(\sigma^-, o, v_1, \dots, v_n) \in State_\Delta \mathbf{X} ([C] \times [T_1] \times \dots \times [T_n]), o \neq null$$

und $body$ angewandt auf σ^- und die lokale Umgebung mit dem Wert o für $this$ und den Werten v_i für x_i kann ohne Laufzeitfehler im Zustand σ terminieren,

$$\sigma^- \xrightarrow{o.op_C(v_1, \dots, v_n)} \perp \in R_\Delta^J \quad \text{falls}$$

$$(\sigma^-, o, v_1, \dots, v_n) \in State_\Delta \mathbf{X} ([C] \times [T_1] \times \dots \times [T_n]), o \neq null$$

und $body$ angewandt auf σ^- und die lokale Umgebung mit dem Wert o für $this$ und den Werten v_i für x_i kann zu einem Laufzeitfehler führen oder nicht terminieren.

2. Für alle $(op : C \times T_1 \times \dots \times T_n \longrightarrow T) \in M_\Delta \cup Q_\Delta$, $T \neq Void$ mit zugehöriger Java Methode $T \text{ } op(T_1 \ x_1, \dots, T_n \ x_n) \text{ } body$ der Klasse C ist

$$\sigma^- \xrightarrow{o.op_C(v_1, \dots, v_n):r} \sigma \in R_\Delta^J \quad \text{falls}$$

$(\sigma^-, o, v_1, \dots, v_n) \in State_\Delta \mathbf{X} ([C] \times [T_1] \times \dots \times [T_n])$, $o \neq null$
 und $body$ angewandt auf σ^- und die lokale Umgebung mit dem Wert o für $this$
 und den Werten v_i für x_i kann ohne Laufzeitfehler im Zustand σ terminieren
 mit Ergebniswert r ,

$$\sigma^- \xrightarrow{o.op_C(v_1, \dots, v_n)} \perp \in R_\Delta^J \quad \text{falls}$$

$(\sigma^-, o, v_1, \dots, v_n) \in State_\Delta \mathbf{X} ([C] \times [T_1] \times \dots \times [T_n])$, $o \neq null$
 und $body$ angewandt auf σ^- und die lokale Umgebung mit dem Wert o für $this$
 und den Werten v_i für x_i kann zu einem Laufzeitfehler führen oder nicht
 terminieren.

3. Für alle $(C : T_1 \times \dots \times T_n \longrightarrow C) \in \text{Con}_\Delta$ mit zugehörigem Java Konstruktor $C(T_1\ x_1, \dots, T_n\ x_n)\ \textit{body}$ ist

$$\sigma^- \xrightarrow{C(v_1, \dots, v_n): o} \sigma \in R_\Delta^J \quad \text{falls}$$

$(\sigma^-, v_1, \dots, v_n) \in \text{State}_\Delta \times ([T_1] \times \dots \times [T_n])$ und \textit{body} angewandt auf σ^- und die lokale Umgebung mit den Werten v_i für x_i kann ohne Laufzeitfehler im Zustand σ terminieren mit dem neu erzeugten Objekt o ,

$$\sigma^- \xrightarrow{C(v_1, \dots, v_n)} \perp \in R_\Delta^J \quad \text{falls}$$

$(\sigma^-, v_1, \dots, v_n) \in \text{State}_\Delta \times ([T_1] \times \dots \times [T_n])$ und \textit{body} angewandt auf σ^- und die lokale Umgebung mit den Werten v_i für x_i kann zu einem Laufzeitfehler führen oder nicht terminieren.

Beispiel

Die Java Komponente `CountersJava` induziert das Σ_{Δ} -Transitionssystem

$\mathcal{T}_{\text{Counters}}^{\text{CountersJava}} = \mathcal{T}_{\text{Counters}}$ von Abschnitt 4.4.

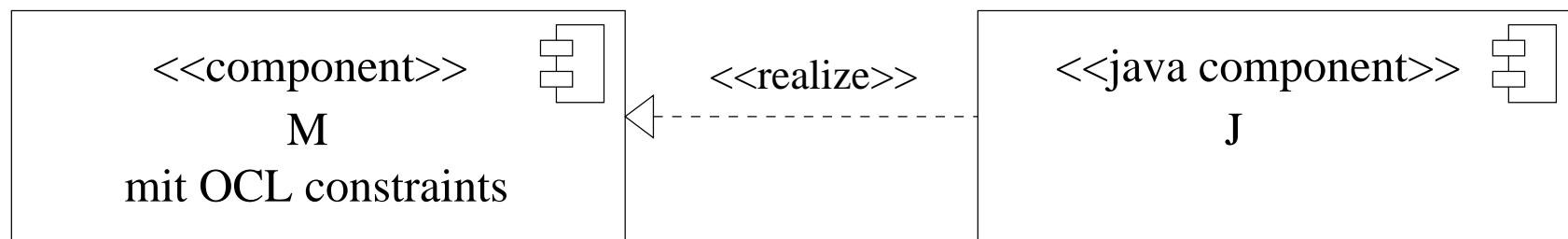
5.2 Korrekte Java Realisierungen

Definition (Korrekte Java Realisierung):

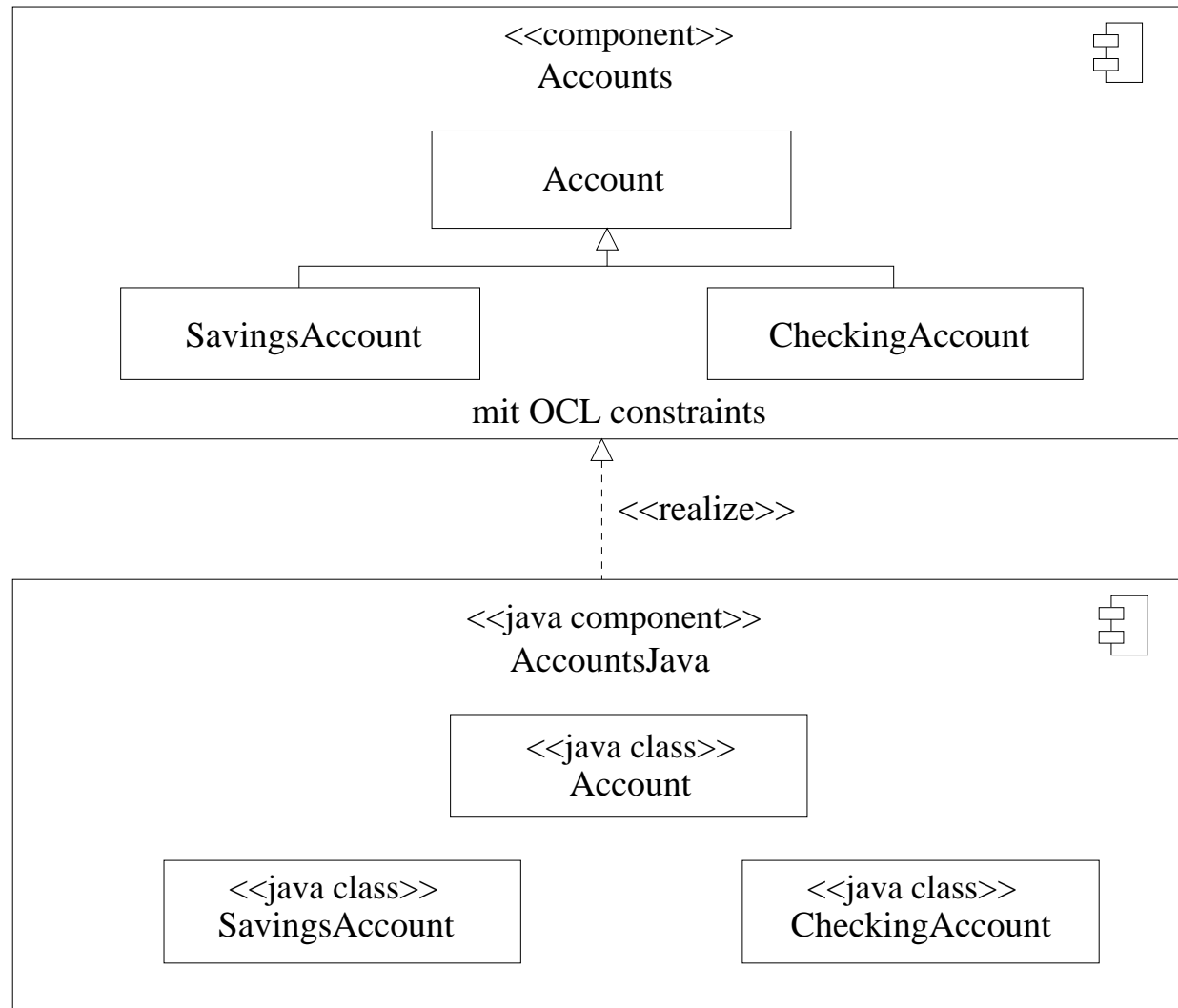
Sei $CompSpec = (\langle M, \Delta \rangle, Invs^e, OpSpecs)$ eine Komponentenspezifikation und sei J eine Java Komponente. J ist eine *korrekte Java Realisierung* von $CompSpec$, wenn die beiden folgenden Bedingungen erfüllt sind:

1. J ist konform zu Σ_Δ .
2. Das durch J induzierte Σ_Δ -Transitionssystem \mathcal{T}_Δ^J ist ein Modell von $CompSpec$.

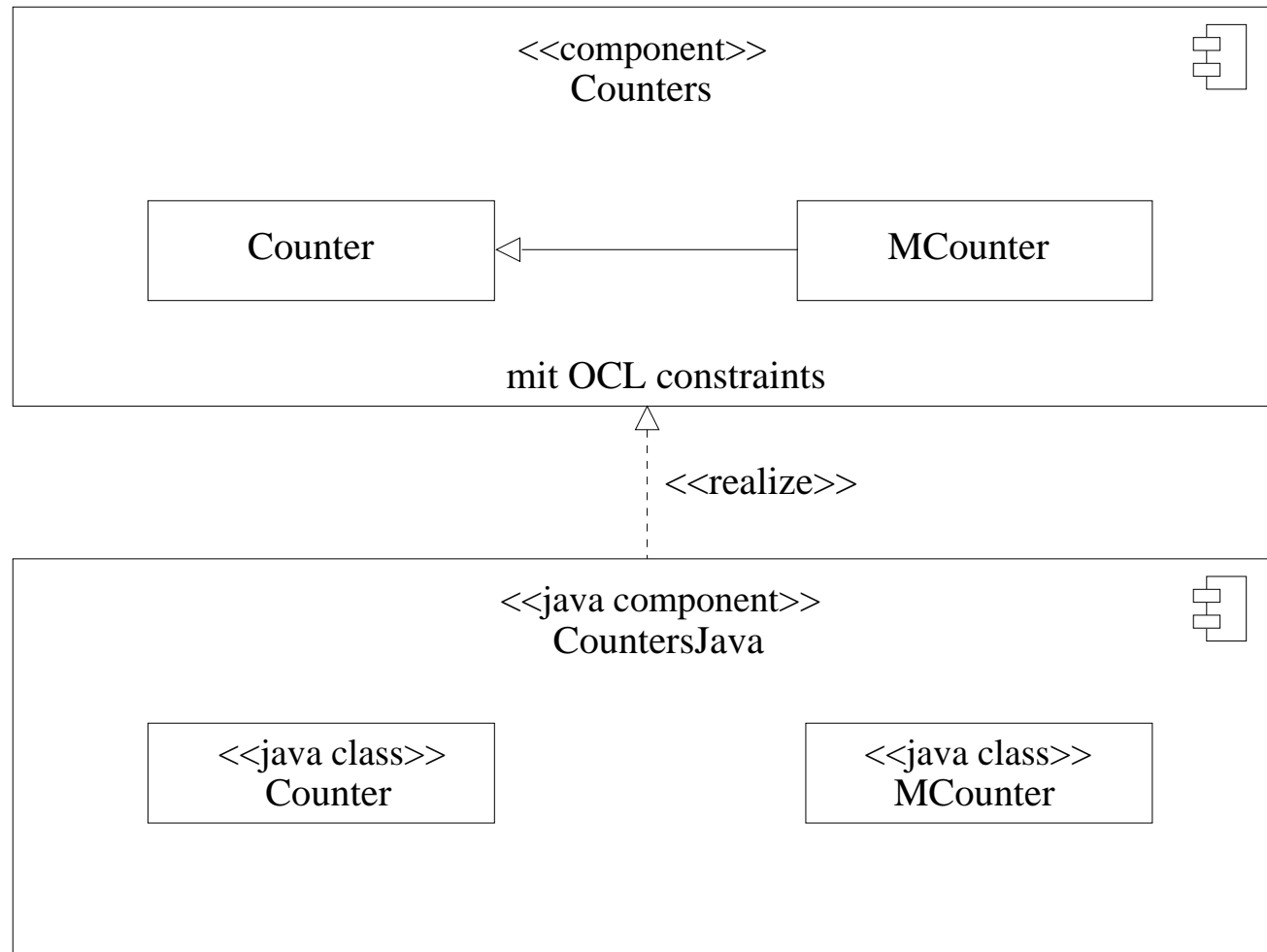
Notation:



Beispiel (AccountsJava realisiert Accounts):



Beispiel (CountersJava realisiert Counters):



Im Folgenden sind wir an vereinfachten Beweisverpflichtungen für korrekte Java Realisierungen interessiert.

Idee:

In der Nachbedingung $Q^{Visibility(op)}$ wollen wir statt $CLASSINV_{\Delta}$ nur die Klasseninvarianten für $self$ überprüfen.

Generelle Voraussetzungen für Komponentenspezifikationen

- Alle Klasseninvarianten erfüllen das Lokalisierungsprinzip.
- Es gilt die Verhaltensverträglichkeit bzgl. Subtypen.
- Alle Komponenteninvarianten gelten im Anfangszustand σ_{init} ,
d.h. $\llbracket COMPINV_M \rrbracket_{\beta, \sigma_{init}, \sigma_{init}} = true$.

Generelle Voraussetzungen für Java Programme

- Es gibt keine privaten Konstruktoren in Java Klassen.
- Alle Attribute (d.h. Instanzvariablen) von Java Klassen sind "private" oder "protected".
- Auf geschützte (protected) Attribute wird höchstens in der deklarierenden Klasse oder in deren Subklassen zugegriffen.
- Innerhalb einer Klasse können Zugriffe auf private oder geschützte Attribute nur für *this* erfolgen.

Nicht erlaubt ist z.B.

```
class C {  
    private int a;  
  
    void op(C x) {  
        x.a = 0;  
    }  
}
```

Definition (Beweisverpflichtung $PObs^{CompSpec}$):

Sei $CompSpec$ eine Komponentenspezifikation mit formaler Repräsentation $FRep(CompSpec) = (\langle M, \Sigma_{\Delta} \rangle, Invs, OpSpecs)$.
 Sei $C \in Class_{\Delta}$ und Q ein OCL-Ausdruck vom Typ Boolean.

$$FULLINV_C = \bigwedge_{A \geq C} INV_A \quad (\text{"volle" Klasseninvariante von C})$$

$$Simpl_C(Q^+) = Q \text{ and } COMPINV_M \text{ and } FULLINV_C \text{ and} \\ \bigwedge_{D < C} (self.oclIsTypeOf(D) \text{ implies } FULLINV_D)$$

$$Simpl_C(Q^{\sim}) = Q \text{ and } FULLINV_C \text{ and} \\ \bigwedge_{D < C} (self.oclIsTypeOf(D) \text{ implies } FULLINV_D)$$

$$Simpl_C(Q^-) = Q$$

Für jedes $op \in Opns_{\Delta}$ ist die zugehörige Beweisverpflichtung $PObs_{op}^{CompSpec}$ definiert wie folgt:

1. Sei $(op : C \times T_1 \times \dots \times T_n \rightarrow Void) \in M_{\Delta}$ eine Methode ohne Rückgabewert.

$$\begin{aligned}
 PObs_{op}^{CompSpec} &= \text{context } C :: op(x_1 : T_1, \dots, x_n : T_n) \\
 &\quad \text{pre : } P^{Visibility(op)} \\
 &\quad \text{post : } Simpl_C(Q^{Visibility(op)})
 \end{aligned}$$

wobei $(\text{context } C :: op(x_1 : T_1, \dots, x_n : T_n) \text{ pre : } P \text{ post : } Q) \in OpSpecs$.

2. Analog für Methoden mit Rückgabewert, die keine Queries sind.

3. Sei $(q : C \times T_1 \times \dots \times T_n \rightarrow T) \in Q_\Delta$ eine Query.

$$PObs_q^{CompSpec} = \text{context } C :: q(x_1 : T_1, \dots, x_n : T_n) : T$$

$$\text{pre : } P^{Visibility(q)}$$

$$\text{post : } Q$$

wobei $(\text{context } C :: q(x_1 : T_1, \dots, x_n : T_n) : T \text{ pre : } P \text{ post : } Q) \in OpSpecs$.

4. Sei $(C : T_1 \times \dots \times T_n \rightarrow C) \in Con_\Delta$ ein Konstruktor.

$$PObs_C^{CompSpec} = \text{context } C :: C(x_1 : T_1, \dots, x_n : T_n)$$

$$\text{pre : } P^{Visibility(C)}$$

$$\text{post : } Q \text{ and } COMPINV_M \text{ and } FULLINV_C$$

(ohne $COMPINV_M$ falls $Visibility(C) = \sim$)

wobei $(\text{context } C :: C(x_1 : T_1, \dots, x_n : T_n) \text{ pre : } P \text{ post : } Q) \in OpSpecs$.

Die Menge der durch $CompSpec$ induzierten Beweisverpflichtungen für Java Programme ist definiert durch

$$PObs^{CompSpec} = \{PObs_{op}^{CompSpec} \mid op \in Opns_{\Delta}\}$$

Bemerkung:

Häufig kann im konkreten Fall $P^{Visibility(op)}$ durch eine schwächere aber kürzere Vorbedingung ersetzt werden (die weniger Invarianten voraussetzt).

Beispiel (Beweisverpflichtungen für Counters)

Zunächst bestimmen wir die vollen Klasseninvarianten:

$$FULLINV_{Counter} = \text{count} \geq 0$$

$$FULLINV_{MCounter} = \text{count} \geq 0 \text{ and } \text{last} \geq 0$$

Desweiteren wissen wir

$$COMPINV_{Counters} = \text{true}$$

$$CLASSINV_{\Delta} = \text{Counter.allInstances()} \rightarrow \text{forAll}(\text{self} \mid \text{self.count} \geq 0) \text{ and } \\ \text{MCounter.allInstances()} \rightarrow \text{forAll}(\text{self} \mid \text{self.last} \geq 0)$$

Die Beweisverpflichtungen für die Operationen lauten

context Counter::Counter()

pre : $CLASSINV_{\Delta}$

post : $count = 0$

context Counter::inc()

pre : $CLASSINV_{\Delta}$

post : $count = count@pre + 1$ and

$count \geq 0$ and

$(self.ocllsTypeOf(MCounter) \text{ implies } (count \geq 0 \text{ and } last \geq 0))$

context Counter::dec()

pre : $count > 0$ and $CLASSINV_{\Delta}$

post : $count = count@pre - 1$ and

$count \geq 0$ and

$(self.ocllsTypeOf(MCounter) \text{ implies } (count \geq 0 \text{ and } last \geq 0))$

context MCounter::MCounter()

pre : $CLASSINV_{\Delta}$

post : $count = 0$ and $last = 0$

context MCounter::inc()

pre : $CLASSINV_{\Delta}$

post : $count = count@pre + 1$ and $last = count@pre$ and
 $count \geq 0$ and $last \geq 0$

context MCounter::dec()

pre : $count > 0$ and $CLASSINV_{\Delta}$

post : $count = count@pre - 1$ and $last = count@pre$ and
 $count \geq 0$ and $last \geq 0$

Beispiel (Beweisverpflichtungen für Accounts)

Wir bestimmen die vollen Klasseninvarianten:

$$FULLINV_{Account} = \text{bal} \geq \text{limit} \text{ and } \text{limit} \leq 0$$

$$FULLINV_{SavingsAccount} = \text{bal} \geq \text{limit} \text{ and } \text{limit} = 0 \text{ and } \text{interestRate} > 0$$

$$FULLINV_{CheckingAccount} = \text{bal} \geq \text{limit} \text{ and } \text{limit} \leq 0 \text{ and } \\ \text{charge} > 0 \text{ and } \text{countUpdates} \geq 0$$

Die Beweisverpflichtungen für die Operationen lauten:

```
context Account::Account()  
  pre : CLASSINV $\Delta$   
  post : bal  $\geq$  limit and limit  $\leq$  0
```

context Account::update(n: Real)
pre : $\text{bal} + n \geq \text{limit}$ and $CLASSINV_{\Delta}$
post : $\text{bal} = \text{bal@pre} + n$ and $FULLINV_{Account}$ and
self.ocllsTypeOf(SavingsAccount) implies $FULLINV_{SavingsAccount}$ and
self.ocllsTypeOf(CheckingAccount) implies $FULLINV_{CheckingAccount}$

context Account::getBal():Real
pre : $CLASSINV_{\Delta}$
post : result = bal

context SavingsAccount::SavingsAccount()
pre : $CLASSINV_{\Delta}$
post : $FULLINV_{SavingsAccount}$

context SavingsAccount::addInterest()
pre : $CLASSINV_{\Delta}$
post : $\text{bal} = \text{bal@pre} + \text{bal@pre} * \text{interestRate}/100$ and $FULLINV_{SavingsAccount}$

context CheckingAccount::CheckingAccount()

pre : $CLASSINV_{\Delta}$

post : $FULLINV_{CheckingAccount}$

context CheckingAccount::update(n: Real)

pre : $bal + n \geq limit$ and $CLASSINV_{\Delta}$

post : $bal = bal@pre + n$ and $countUpdates = countUpdates@pre + 1$ and
 $FULLINV_{CheckingAccount}$

context CheckingAccount::payCharge()

pre : $bal - countUpdates * charge \geq limit$ and $CLASSINV_{\Delta}$

post : $bal = bal@pre - countUpdates * charge$ and $countUpdates = 0$ and
 $FULLINV_{CheckingAccount}$

Theorem:

Sei $CompSpec = (\langle M, \Delta \rangle, Invs^e, OpSpecs)$ eine Komponentenspezifikation mit formaler Repräsentation $FRep(CompSpec) = (\langle M, \Delta \rangle, Invs, OpSpecs)$ und sei J eine Java Komponente.

J ist eine korrekte Java Realisierung von $CompSpec$ wenn die folgenden Bedingungen erfüllt sind:

1. *Verantwortlichkeit des Implementierers:*

- (a) J ist konform zu Σ_Δ
- (b) Das durch J induzierte Σ_Δ -Transitionssystem erfüllt die Beweisverpflichtungen $PObs^{CompSpec}$, die durch $CompSpec$ induziert werden, d.h.

$$\mathcal{T}_\Delta^J \models PObs_{op}^{CompSpec} \quad \text{für alle } op \in Opns_\Delta$$

2. Verantwortlichkeit des Benutzers:

Wenn ein Methodenaufruf $o.op(v_1, \dots, v_n)$ oder ein Konstruktoraufruf $new\ C(v_1, \dots, v_n)$ erfolgt, dann gilt:

- (a) Die Methode bzw. der Konstruktor gehört zu Ops_{Δ} .
- (b) Die Vorbedingung der Operationsspezifikation der Methode op bzw. des Konstruktors C is erfüllt (bzgl. des zur Programmierzeit bekannten Klassentyps von o).
- (c) Falls op komponenten-privat ist, dann gilt die volle Klasseninvariante für das aufrufende Objekt.
- (d) Falls op komponenten-öffentlich ist, gilt (c) und zusätzlich gelten alle Komponenteninvarianten.

Beweis des Theorems:

Es ist zu zeigen $\mathcal{T}_\Delta^J \in \llbracket \text{CompSpec} \rrbracket$.

Nach der generellen Voraussetzung über Komponentenspezifikationen ist $\llbracket \text{COMPINV}_M \rrbracket_{\beta, \sigma_{init}, \sigma_{init}} = \text{true}$. Also bleibt nach der Definition eines Modells zu zeigen, dass

für alle $(op : C \times T_1 \times \dots \times T_n \longrightarrow \text{Void}) \in \text{Opns}_\Delta$ und

für alle $(\text{context } C :: op(x_1 : T_1, \dots, x_n : T_n) \text{ pre} : P \text{ post} : Q) \in \text{OpSpecs}$ gilt:

$$\begin{aligned} \mathcal{T}_\Delta^J \models & \text{context } C :: op(x_1 : T_1, \dots, x_n : T_n) \\ & \text{pre} : P^{Visibility(op)} \\ & \text{post} : Q^{Visibility(op)} \end{aligned}$$

und analog für die anderen Arten von Operationen.

Wir betrachten den oben angegebenen Fall einer Methode ohne Rückgabewert.

Außerdem sei o.B.d.A. $Visibility(op) = \sim$.

Wegen Voraussetzung (1) des Theorems gilt:

$$\begin{aligned} \mathcal{T}_\Delta^J \models & \text{context } C :: \text{op}(x_1 : T_1, \dots, x_n : T_n) \\ & \text{pre} : P \text{ and } \text{CLASSINV}_\Delta \\ & \text{post} : Q \text{ and } \text{FULLINV}_C \\ & \text{and } \bigwedge_{D < C} (\text{self.oclIsTypeOf}(D) \text{ implies } \text{FULLINV}_D) \end{aligned}$$

Es ist zu zeigen, dass daraus folgt:

$$\begin{aligned} \mathcal{T}_\Delta^J \models & \text{context } C :: \text{op}(x_1 : T_1, \dots, x_n : T_n) \\ & \text{pre} : P \text{ and } \text{CLASSINV}_\Delta \\ & \text{post} : Q \text{ and } \text{CLASSINV}_\Delta \end{aligned}$$

Sei nun $(\sigma^-, o, v_1, \dots, v_n) \in \text{State}_\Delta \times ([C] \times [T_1] \times \dots \times [T_n])$, $o \neq \text{null}$ und gelte $\llbracket P \text{ and } \text{CLASSINV}_\Delta \rrbracket_{\beta, \sigma^-, \sigma^-} = \text{true}$ (mit $\beta(\text{self}) = o, \beta(x_i) = v_i$). Folglich gilt:

- (a) Es existiert $\sigma \in \text{State}_\Delta$, so dass $\sigma^- \xrightarrow{o.\text{op}_C(v_1, \dots, v_n)} \sigma \in R_\Delta^J$
- (b) $\sigma^- \xrightarrow{o.\text{op}_C(v_1, \dots, v_n)} \perp \notin R_\Delta^J$
- (c) Für alle $\sigma^- \xrightarrow{o.\text{op}_C(v_1, \dots, v_n)} \sigma \in R_\Delta^J$ ist (mit β wie oben) $\llbracket Q \text{ and } \text{FULLINV}_C \text{ and } \bigwedge_{D < C} (\text{self.oclIsTypeOf}(D) \text{ implies } \text{FULLINV}_D) \rrbracket_{\beta, \sigma^-, \sigma} = \text{true}$

Es genügt also zu zeigen, dass dann für alle $\sigma^- \xrightarrow{o.op_C(v_1, \dots, v_n)} \sigma \in R_\Delta^J$ gilt:

$$\llbracket Q \text{ and } CLASSINV_\Delta \rrbracket_{\beta, \sigma^-, \sigma} = true \quad (\text{mit } \beta \text{ wie oben})$$

Sei nun $\sigma^- \xrightarrow{o.op_C(v_1, \dots, v_n)} \sigma \in R_\Delta^J$ mit

$$\begin{aligned} & \llbracket Q \text{ and } FULLINV_C \text{ and } \bigwedge_{D < C} (\text{self.oclIsTypeOf}(D) \text{ implies } FULLINV_D) \rrbracket_{\beta, \sigma^-, \sigma} = true \\ & \text{also } \llbracket FULLINV_C \text{ and } \bigwedge_{D < C} (\text{self.oclIsTypeOf}(D) \text{ implies } FULLINV_D) \rrbracket_{\beta, \sigma^-, \sigma} = true \end{aligned}$$

Folglich gilt für alle $D \in Class_\Delta$ mit $o \in D_\sigma$ und für alle
(context D inv : Inv) $\in Invs$: $\llbracket Inv \rrbracket_{\beta, \sigma^-, \sigma} = true$ (mit β wie oben).

Es bleibt zu zeigen, dass dann auch alle Klasseninvarianten für alle Objekte $obj \neq o$ erhalten bleiben durch die aktuelle Ausführung des Rumpfes der Methode $void op(T_1 x_1, \dots, T_n x_n)$ der Klasse C (bzw. dass für neue Objekte obj die Klasseninvarianten etabliert sind in σ).

Das heißt, wir wollen zeigen, dass für alle $D \in Class_{\Delta}$,
für alle $obj \in D_{\sigma}$ mit $obj \neq o$ und für alle $(\text{context } D \text{ inv} : Inv) \in Invs$
gilt: $\llbracket Inv \rrbracket_{\beta[self \mapsto obj], \sigma^-, \sigma} = true$

Dies ist eine Konsequenz der Annahme $\llbracket P \text{ and } CLASSINV_{\Delta} \rrbracket_{\beta, \sigma^-, \sigma^-, \sigma} = true$
und des unten bewiesenen Lemmas.

Insgesamt folgt damit (vgl. Lemma in Abschnitt 4.2)
 $\llbracket CLASSINV_{\Delta} \rrbracket_{\beta, \sigma^-, \sigma} = true$ und damit, wegen (c), auch
 $\llbracket Q \text{ and } CLASSINV_{\Delta} \rrbracket_{\beta, \sigma^-, \sigma} = true$.

Lemma (Erhaltung von Invarianten)

Es seien die generellen Voraussetzungen für Komponentenspezifikationen und Java Programme sowie die Voraussetzungen (1) und (2) des Theorems erfüllt. Dann gilt für alle $C \in Class_{\Delta}$, $\sigma^- \in State_{\Delta}$, und $o \in C_{\sigma^-}$ das Folgende:

Falls

für alle $D \in Class_{\Delta}$, für (zumindest) alle $obj \in D_{\sigma^-}$ mit $obj \neq o$ und
für alle $(\text{context } D \text{ inv} : Inv) \in Invs$ gilt $\llbracket Inv \rrbracket_{\beta[self \mapsto obj], \sigma^-, \sigma^-} = true$

und falls

ein Aufruf $o.op(v_1, \dots, v_n)$ oder $new C(v_1, \dots, v_n)$ im Zustand σ^- erfolgt und ohne Laufzeitfehler mit Zustand σ terminiert

dann gilt

für alle $D \in Class_{\Delta}$, für (zumindest) alle $obj \in D_{\sigma^-}$ mit $obj \neq o$ und
für alle $(\text{context } D \text{ inv} : Inv) \in Invs$ gilt: $\llbracket Inv \rrbracket_{\beta[self \mapsto obj], \sigma^-, \sigma} = true$

Beweis des Lemmas:

Induktion über die Tiefe n der geschachtelten Methoden- und Konstruktoraufrufe:

Fall 0: $n = 0$.

Wegen der generellen Voraussetzungen für Java Programme kann sich höchstens der Zustand von o geändert haben. Folglich bleiben alle Invarianten für alle Objekte $obj \neq o$ erhalten.

Fall 1: $n > 0$.

Sei $o'.op'(v'_1, \dots, v'_n)$ oder $new C'(v'_1, \dots, v'_n)$ ein geschachtelter Aufruf, der im Rumpf von op oder $C(\dots)$ ausgeführt wird. Wir nehmen o.B.d.A. an, dass kein weiterer Aufruf im Rumpf von op bzw. $C(\dots)$ erfolgt. Nach den generellen Voraussetzungen für Java Programme haben alle Objekte $obj \neq o$ ihren Zustand nicht geändert (d.h. sie erfüllen ihre Invarianten), solange die Ausführung des Rumpfes den geschachtelten Aufruf noch nicht erreicht hat.

Im Folgenden betrachten wir nun den geschachtelten Aufruf $o'.op'(v'_1, \dots, v'_n)$ (der Konstruktoraufruf wird ähnlich behandelt).

Fall 1.1: $o' = o$.

Nach Induktionsvoraussetzung erfüllen alle Objekte $obj \neq o' = o$ ihre Invarianten nach Ausführung des geschachtelten Methodenaufrufs und folglich, nach den generellen Voraussetzungen für Java Programme und wegen der Lokalität von Klasseninvarianten, auch nach Ausführung des übergeordneten Aufrufs.

Fall 1.2: $o' \neq o$.

Nach den Bedingungen (2b) und (2c) erfüllen alle existierenden Objekte (auch o) ihre Invarianten bevor der geschachtelte Aufruf ausgeführt wird. Außerdem ist nach Bedingung (2a) und wegen der Verhaltensverträglichkeit bzgl. Subtypen die Vorbedingung der Operationsspezifikation von op' or $C'(\dots)$ erfüllt bevor der geschachtelte Aufruf ausgeführt wird. Somit ist die Vorbedingung der korrespondierenden Beweisverpflichtung erfüllt. Nach Bedingung (1b) erfüllt o' seine Invarianten nach Ausführung des geschachtelten Aufrufs. Nach Induktionsvoraussetzung erfüllen auch alle anderen Objekte ihre Invarianten nach Ausführung des geschachtelten Aufrufs. Mithin erfüllen nach den generellen Voraussetzungen für Java Programme und wegen der Lokalität von Klasseninvarianten alle existierenden Objekte ihre Invarianten nach Ausführung des übergeordneten Aufrufs.

5.3 Zusammenfassung

Sei $CompSpec = (\langle M, \Delta \rangle, Invs^e, OpSpecs)$ eine Komponentenspezifikation.

- Eine Java Komponente J ist eine korrekte Java Realisierung von $CompSpec$ falls
 - J konform zu Σ_Δ ist und
 - das (durch J induzierte) Σ_Δ -Transitionssystem \mathcal{T}_Δ^J ein Modell von $CompSpec$ ist.
- Eine Komponentenspezifikation induziert eine Menge von Beweisverpflichtungen.
- Die Erfüllung der Beweisverpflichtungen (durch das Transitionssystem \mathcal{T}_Δ^J) garantiert die Korrektheit der Java Realisierung falls
 - die generellen Voraussetzungen (z.B. keine öffentlichen Attribute, nur lokale Klasseninvarianten) und
 - die Bedingungen gemäß der Verantwortlichkeit des Implementierers und
 - die Bedingungen gemäß der Verantwortlichkeit des Benutzers erfüllt sind.