Kapitel 5

Java Realisierungen von Komponentenspezifikationen

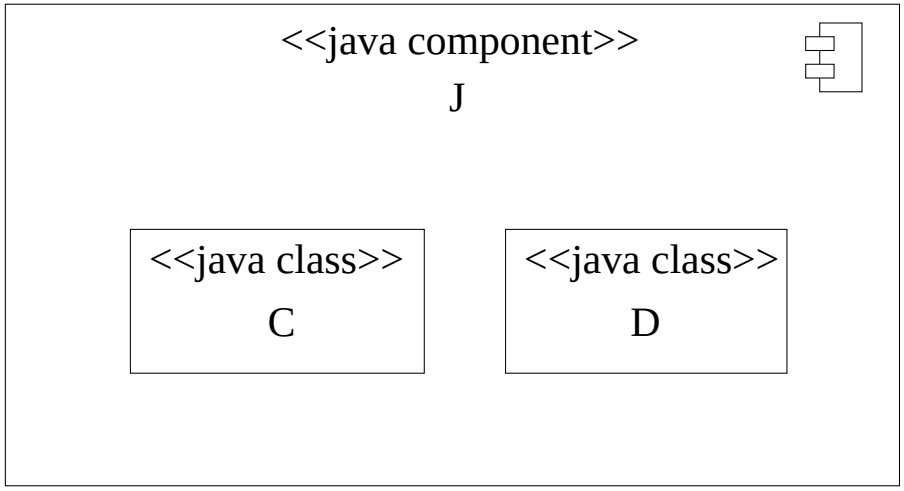
Prof. Dr. Rolf Hennicker

15.07.2010

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 $(\_.a : C \to T) \in A_\Delta$ existiert eine entsprechende Instanzvariable (mit demselben Namen) in der zugehörigen Java Klasse `C`.
3. Für jede Operation $(op : C \times T_1 \times \ldots \times T_n \to T) \in M_\Delta \cup Q_\Delta$ existiert eine entsprechende Methode in der zugehörigen Java Klasse `C` und für jeden Konstruktor $(C : T_1 \times \ldots \times T_n \to C) \in Con_\Delta$ 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",
   "$\sim$" wird abgebildet auf Java-Default-Sichtbarkeit,
   "$+$" wird abgebildet auf "public" innerhalb einer öffentlichen Java Klasse.
6. Für jede Queryoperation $(q : C \times T_1 \times \ldots \times T_n \to T) \in Q_\Delta$ besitzt die entsprechende Java Methode keine Seiteneffekte (semantische Bedingung!).

Bemerkung:

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

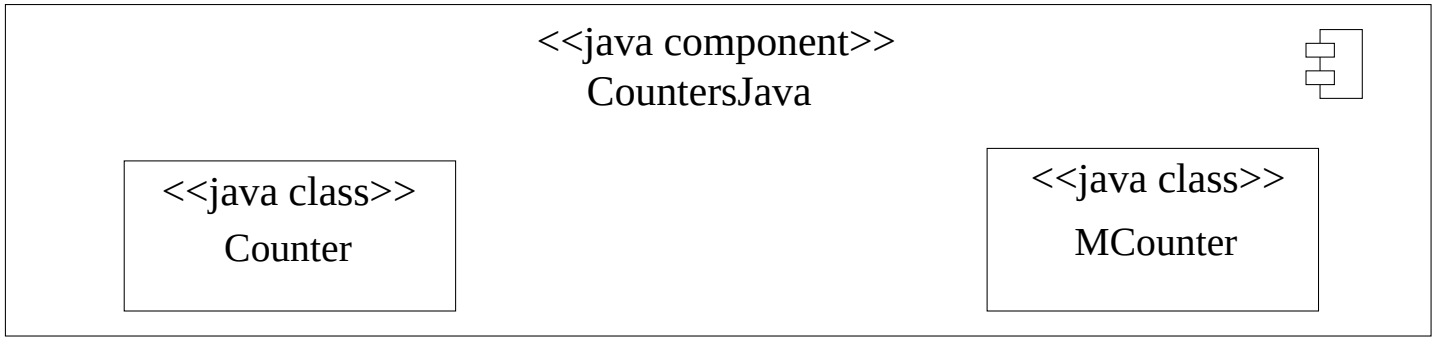
   Insbesondere gilt:
   Für jeden Rollennamen $(\_.a : C \to \mathit{Set}(T)) \in A_\Delta$ existiert ein entsprechendes Referenzattribut (z.B. `Set a;`) in der Java-Klasse `C`.
3. Die Argument- und Ergebnistypen von Operationen $op \in \mathit{Opns}_\Delta$ werden von den entsprechenden Java Methoden und Konstruktoren respektiert, z.B.

$$[\mathit{createPoint} : \mathit{System} \times \mathit{Real} \times \mathit{Real} \to \mathit{Point}] \mapsto [\mathit{Point}\ \mathit{createPoint}(\mathit{double}\ x,\ \mathit{double}\ y)]$$

$$[\mathit{move} : \mathit{Point} \times \mathit{Real} \times \mathit{Real} \to \mathit{Void}] \mapsto [\mathit{void}\ \mathit{move}(\mathit{double}\ mx,\ \mathit{double}\ my)]$$

$$[\mathit{Point} : \mathit{Real} \times \mathit{Real} \to \mathit{Point}] \mapsto [\mathit{Point}(\mathit{double}\ x,\ \mathit{double}\ y)]$$

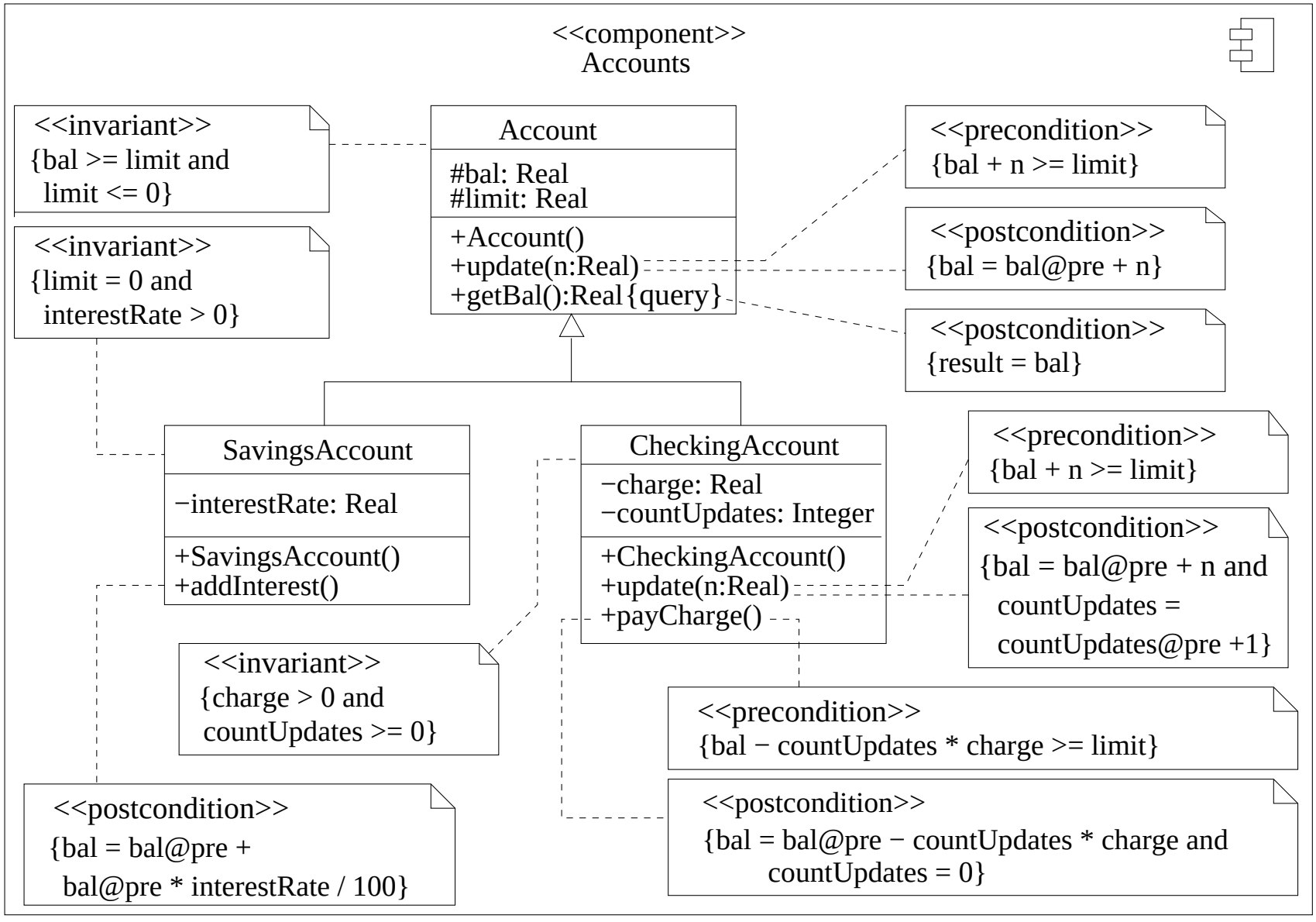
Beispiel (Java Komponente für Counters)



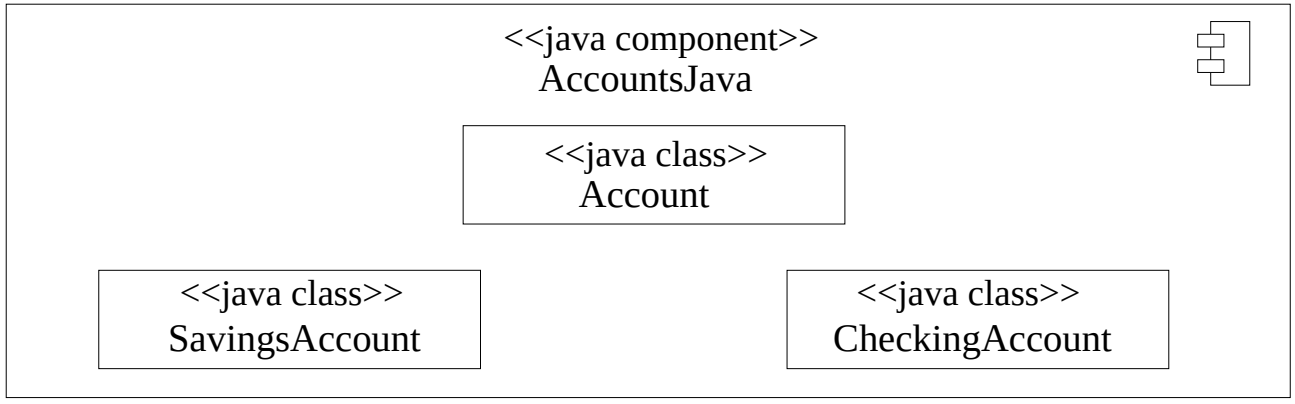

```
package CountersJava;
public class Counter {
  protected int count;

  public Counter() {
    count = 0;
  }

  public void inc() {
    count++;
  }

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

```
package CountersJava;
public class MCounter extends
  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)



Beipiel (Java Komponente AccountsJava)




```
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;}
}
```

```
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);
  }
}
```

```
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

$$\mathcal{T}_\Delta^J \;=\; (State_\Delta, \sigma_{init}, Label_\Delta, \Omega_\Delta, R_\Delta^J)$$

wobei R_Δ^J die kleinste Relation ist, die folgende Bedingungen erfüllt:

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

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

   $(\sigma^-, o, v_1, \ldots, v_n) \in State_\Delta$ X $([\![C]\!] \times [\![T_1]\!] \times \ldots \times [\![T_n]\!])$, $o \neq null$ 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,

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

   $(\sigma^-, o, v_1, \ldots, v_n) \in State_\Delta$ X $([\![C]\!] \times [\![T_1]\!] \times \ldots \times [\![T_n]\!])$, $o \neq null$ 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 $(op : C \times T_1 \times \ldots \times T_n \longrightarrow T) \in M_\Delta \cup Q_\Delta$, $T \neq \mathit{Void}$ mit zugehöriger Java Methode $T\ op(T_1\ x_1, \ldots, T_n\ x_n)\ \mathit{body}$ der Klasse C ist

   $$\sigma^- \xrightarrow{o.op_C(v_1,\ldots,v_n):r} \sigma \in R^J_\Delta \qquad \text{falls}$$

   $(\sigma^-, o, v_1, \ldots, v_n) \in \mathit{State}_\Delta$ X $([\![C]\!] \times [\![T_1]\!] \times \ldots \times [\![T_n]\!])$, $o \neq \mathit{null}$ 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,

   $$\sigma^- \xrightarrow{o.op_C(v_1,\ldots,v_n)} \perp \in R^J_\Delta \qquad \text{falls}$$

   $(\sigma^-, o, v_1, \ldots, v_n) \in \mathit{State}_\Delta$ X $([\![C]\!] \times [\![T_1]\!] \times \ldots \times [\![T_n]\!])$, $o \neq \mathit{null}$ 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 $(C : T_1 \times \ldots \times T_n \longrightarrow C) \in Con_\Delta$ mit zugehörigem Java Konstruktor $C(T_1\ x_1, \ldots, T_n\ x_n)\ body$ ist

   $$\sigma^- \xrightarrow{C(v_1,\ldots,v_n):o} \sigma \in R^J_\Delta \qquad \text{falls}$$

   $(\sigma^-, v_1, \ldots, v_n) \in State_\Delta$ X $([\![T_1]\!] \times \ldots \times [\![T_n]\!])$ und *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,

   $$\sigma^- \xrightarrow{C(v_1,\ldots,v_n)} \perp \in R^J_\Delta \qquad \text{falls}$$

   $(\sigma^-, v_1, \ldots, v_n) \in State_\Delta$ X $([\![T_1]\!] \times \ldots \times [\![T_n]\!])$ und *body* angewandt auf σ^- und die lokale Umgebung mit den Werten v_i für x_i kann zu einem Laufzeitfehler führen oder nicht terminieren.

Bemerkung:

Sei Σ_Δ eine Klassensignatur und J eine zu Σ_Δ konforme Java Komponente. Das durch J induzierte Σ_Δ-Transitionssystem $\mathcal{T}^J_\Delta$ ist input-enabled und erfüllt auch alle anderen Bedingungen aus der Defintion von Σ_Δ-Transitionssystemen.

Beispiel

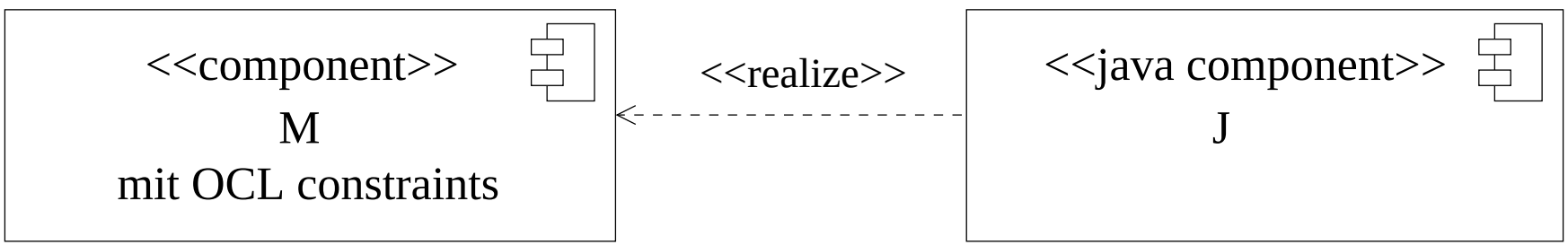
Die Java Komponente CountersJava induziert das Σ_Δ-Transitionssystem $\mathcal{T}^{\text{CountersJava}}_{\text{Counters}} = \mathcal{T}_{\text{Counters}}$ von Abschnitt 4.3.

5.2 Korrekte Java Realisierungen

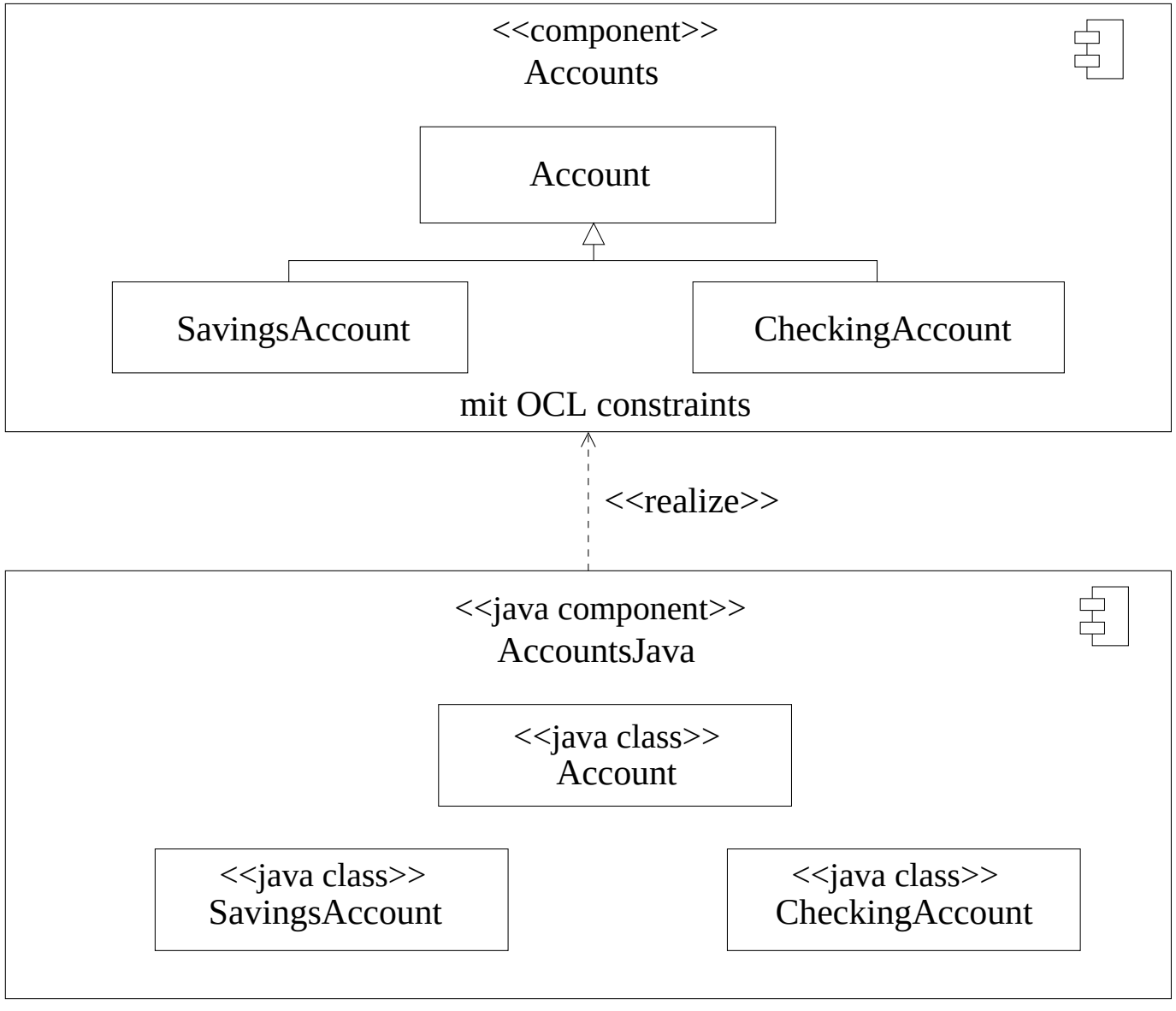
Definition (Korrekte Java Realisierung)
Sei $CompSpec = (\langle M, \Delta \rangle, Invs^e, OpSpecs)$ eine Komponentenspezifikation und sei J eine Java Komponente. J ist eine *korrekte Java Realisierung* von $CompSpec$, wenn die beiden folgenden Bedingungen erfüllt sind:

1. J ist konform zu Σ_Δ.
2. Das durch J induzierte Σ_Δ-Transitionssystem $\mathcal{T}_\Delta^J$ ist ein Modell von $CompSpec$, d.h. $\mathcal{T}_\Delta^J \in [\![CompSpec]\!]$.

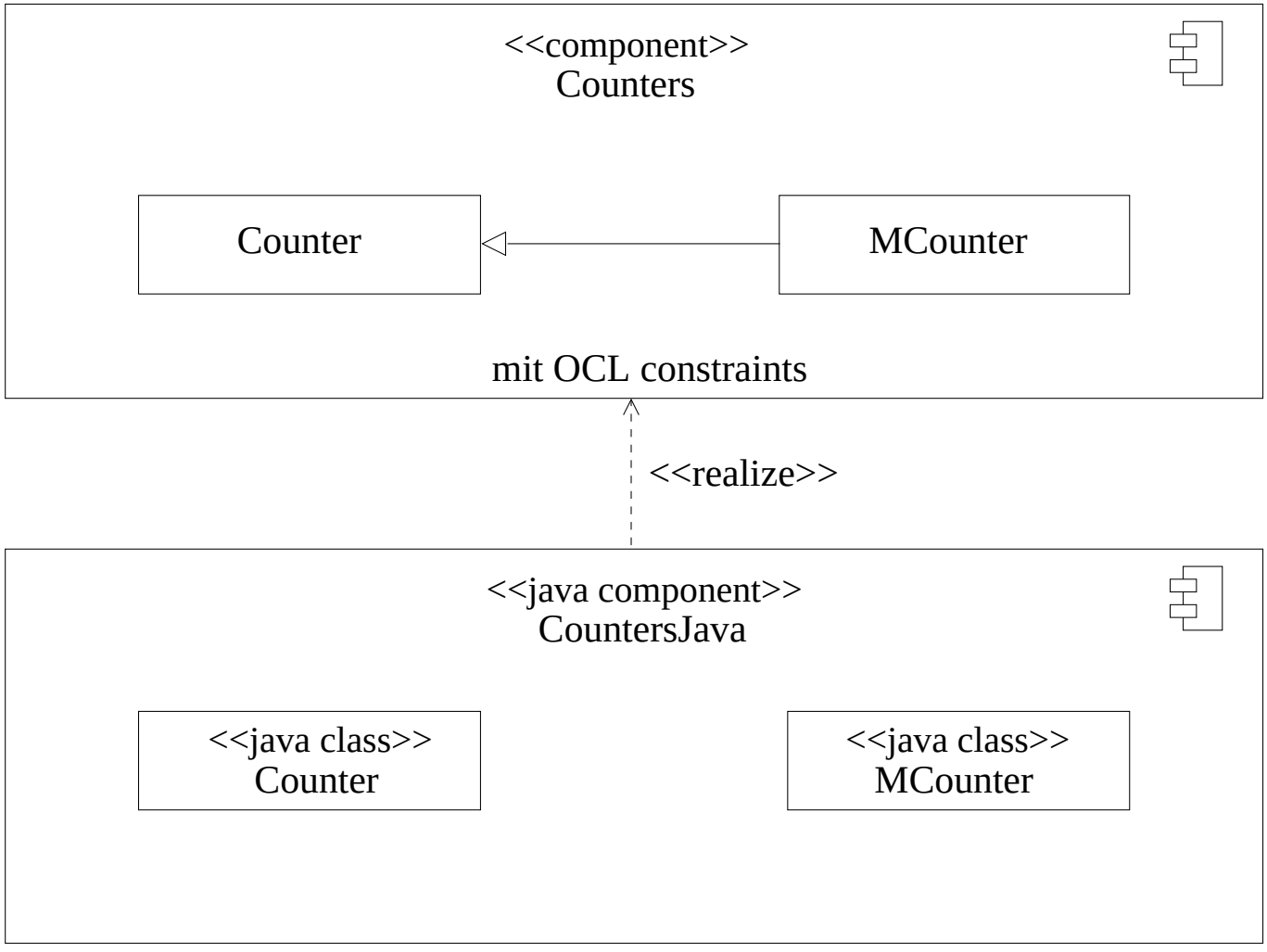
Notation:



Example (AccountsJava realisiert Accounts)



Example (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

- Die Komponentenspezifikation erfüllt das Geheimnisprinzip.
- Alle Klasseninvarianten erfüllen das Lokalitätsprinzip.
- Es gilt die Verhaltensverträglichkeit bzgl. Subtypen.
- Alle Komponenteninvarianten gelten im Anfangszustand σ_{init}, d.h. $[\![COMPINV_M]\!]_{\beta,\sigma_{init},\sigma_{init}} = true$.

Generelle Voraussetzungen für Java Programme

- 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.

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

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

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

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

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

1. Sei $(op : C \times T_1 \times \ldots \times T_n \to Void) \in M_\Delta$ eine Methode ohne Rückgabewert.

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

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

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

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

$$\begin{aligned} PObs_{q}^{CompSpec} \quad = \quad & \textbf{context} \quad && C :: q(x_1 : T_1, \ldots, x_n : T_n) : T \\ & \textbf{pre} : && P^{Visibility(q)} \\ & \textbf{post} : && Q \end{aligned}$$

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

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

$$
\begin{array}{rcll}
PObs_C^{CompSpec} & = & \textbf{context} & C :: C(x_1 : T_1, \ldots, x_n : T_n) \\
 & & \textbf{pre} : & P^{Visibility(C)} \\
 & & \textbf{post} : & Q \text{ and } COMPINV_M \text{ and } FULLINV_C \\
 & & & (\text{ohne } COMPINV_M \text{ falls } Visibility(C) = \sim)
\end{array}
$$

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

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

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

Beispiel (Beweisverpflichtungen für Counters)

Zunächst bestimmen wir die vollen Klasseninvarianten:

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

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

Desweiteren wissen wir

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

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

Die Beweisverpflichtungen für die Operationen lauten

context Counter::Counter()

pre : $CLASSINV_{\Delta}$

post : count = 0 and count >= 0

context Counter::inc()

pre : $CLASSINV_{\Delta}$

post : count = count@pre + 1 and
count >= 0 and
(self.oclIsTypeOf(MCounter) implies (count >= 0 and last >= 0))

context Counter::dec()

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

post : count = count@pre - 1 and
count >= 0 and
(self.oclIsTypeOf(MCounter) implies (count >= 0 and last >= 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 $>=$ 0 and last $>=$ 0

context MCounter::dec()

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

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

Beispiel (Beweisverpflichtungen für Accounts)

Wir bestimmen zunächst die vollen Klasseninvarianten:

$$FULLINV_{Account} = \text{bal} >= \text{limit and limit} <= 0$$

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

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

Die Beweisverpflichtungen für die Operationen lauten:

context Account::Account()
pre : $CLASSINV_{\Delta}$
post : bal >= limit and limit <= 0

context Account::update(n: Real)
pre : bal + n >= limit and $CLASSINV_{\Delta}$
post : bal = bal@pre + n and $FULLINV_{Account}$ and
self.oclIsTypeOf(SavingsAccount) implies $FULLINV_{SavingsAccount}$ and
self.oclIsTypeOf(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 : bal = bal@pre + bal@pre * interestRate/100 and $FULLINV_{SavingsAccount}$

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

context CheckingAccount::update(n: Real)
pre : bal + n >= limit and $CLASSINV_{\Delta}$
post : bal = bal@pre + n and countUpdates = countUpdates@pre + 1 and $FULLINV_{CheckingAccount}$

context CheckingAccount::payCharge()
pre : bal - countUpdates * charge >= 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.
   $$\mathcal{T}_\Delta^J \models PObs_{op}^{CompSpec} \quad \text{für alle} \quad op \in Opns_\Delta.$$

2. *Verantwortlichkeit des Benutzers:*
   Wenn ein Methodenaufruf $o.op(v_1, \ldots, v_n)$ oder ein Konstruktoraufruf $new\ C(v_1, \ldots, v_n)$ erfolgt, dann gilt:
   (a) Die Methode bzw. der Konstruktor gehört zu $Opns_\Delta$.
   (b) Die Vorbedingung der Operationsspezifikation der Methode op bzw. des Konstruktors C ist 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 (2c) und zusätzlich gelten alle Komponenteninvarianten.

Beweis des Theorems:

Es ist zu zeigen $\mathcal{T}_\Delta^J \in [\![CompSpec]\!]$.

Nach der generellen Voraussetzung über Komponentenspezifikationen ist $[\![COMPINV_M]\!]_{\beta,\sigma_{init},\sigma_{init}} = true$. Also bleibt nach der Definition eines Modells zu zeigen, dass für alle $(op : C \times T_1 \times \ldots \times T_n \longrightarrow Void) \in Opns_\Delta$ und für alle ($\texttt{context}\ C :: op(x_1 : T_1, \ldots, x_n : T_n)\ \texttt{pre} : P\ \texttt{post} : Q) \in OpSpecs$ gilt:

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

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:

$$\begin{array}{lll} \mathcal{T}_\Delta^J \models & \texttt{context} & C :: op(x_1 : T_1, \ldots, x_n : T_n) \\ & \texttt{pre} : & P \text{ and } CLASSINV_\Delta \\ & \texttt{post} : & Q \text{ and } FULLINV_C \\ & & \quad \text{and } \bigwedge_{D<C} (\texttt{self.oclIsTypeOf}(D) \texttt{ implies } FULLINV_D) \end{array}$$

Es ist zu zeigen, dass daraus folgt:

$$\begin{array}{lll} \mathcal{T}_\Delta^J \models & \texttt{context} & C :: op(x_1 : T_1, \ldots, x_n : T_n) \\ & \texttt{pre} : & P \text{ and } CLASSINV_\Delta \\ & \texttt{post} : & Q \text{ and } CLASSINV_\Delta \end{array}$$

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

(a) $\sigma^- \xrightarrow{o.op_C(v_1,\ldots,v_n)} \perp \notin R_\Delta^J$

(b) Für alle $\sigma^- \xrightarrow{o.op_C(v_1,\ldots,v_n)} \sigma \in R_\Delta^J$ ist (mit β wie oben)
$\llbracket Q \text{ and } FULLINV_C \text{ and } \bigwedge_{D<C} (\texttt{self.oclIsTypeOf}(D) \texttt{ implies } FULLINV_D) \rrbracket_{\beta,\sigma^-,\sigma} = true$

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

$$[\![Q \texttt{ and } CLASSINV_\Delta]\!]_{\beta,\sigma^-,\sigma} = true \quad \text{(mit } \beta \text{ wie oben).}$$

Sei nun $\sigma^- \xrightarrow{o.op_C(v_1,\ldots,v_n)} \sigma \in R^J_\Delta$ mit

$$[\![Q \texttt{ and } FULLINV_C \texttt{ and } \bigwedge_{D<C} (\texttt{self.oclIsTypeOf}(D) \texttt{ implies } FULLINV_D)]\!]_{\beta,\sigma^-,\sigma} = true$$

$$\text{also } [\![FULLINV_C \texttt{ and } \bigwedge_{D<C} (\texttt{self.oclIsTypeOf}(D) \texttt{ implies } FULLINV_D)]\!]_{\beta,\sigma^-,\sigma} = true$$

Folglich gilt für alle $A \in Class_\Delta$ mit dem gegebenen $\beta(self) = o \in A_\sigma$ und für alle $(\texttt{context } A \texttt{ inv} : Inv) \in Invs$: $[\![Inv]\!]_{\beta,\sigma^-,\sigma} = true$.

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 $void\ op(T_1\ x_1, \ldots, T_n\ x_n)$ 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 $(\texttt{context } D \texttt{ inv} : Inv) \in Invs$ gilt:
$[\![Inv]\!]_{\beta[self \mapsto obj],\sigma^-,\sigma} = true$

Dies ist eine Konsequenz der Annahme $[\![P \texttt{ and } CLASSINV_\Delta]\!]_{\beta,\sigma^-,\sigma^-} = true$ und des unten bewiesenen Lemmas.

Insgesamt folgt damit (vgl. Lemma in Abschnitt 4.2)
$[\![CLASSINV_\Delta]\!]_{\beta,\sigma^-,\sigma} = true$ und damit, wegen (b), auch
$[\![Q \texttt{ and } CLASSINV_\Delta]\!]_{\beta,\sigma^-,\sigma} = true$.

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 $C \in \mathit{Class}_\Delta$, $\sigma^- \in \mathit{State}_\Delta$, und $o \in C_{\sigma^-}$ das Folgende:

Falls

für alle $D \in \mathit{Class}_\Delta$, für (zumindest) alle $\mathit{obj} \in D_{\sigma^-}$ mit $\mathit{obj} \neq o$ und

für alle (`context` D `inv` : Inv) $\in \mathit{Invs}$ gilt $[\![\mathit{Inv}]\!]_{\beta[\mathit{self} \mapsto \mathit{obj}],\sigma^-,\sigma^-} = \mathit{true}$

und falls

ein Aufruf $o.\mathit{op}(v_1, \ldots, v_n)$ oder $\mathit{new}\ C(v_1, \ldots, v_n)$ im Zustand σ^- erfolgt

und ohne Laufzeitfehler mit Zustand σ terminiert

dann gilt

für alle $D \in \mathit{Class}_\Delta$, für (zumindest) alle $\mathit{obj} \in D_{\sigma^-}$ mit $\mathit{obj} \neq o$ und

für alle (`context` D `inv` : Inv) $\in \mathit{Invs}$: $[\![\mathit{Inv}]\!]_{\beta[\mathit{self} \mapsto \mathit{obj}],\sigma^-,\sigma} = \mathit{true}$.

Beweis des Lemmas:

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

Fall 0: $n = 0$.

Wegen des Geheimnisprinzips und der generellen Voraussetzungen für Java Programme kann sich höchstens der Zustand von o (bzgl. $State_\Delta$) geändert haben. Folglich bleiben alle Invarianten für alle Objekte $obj \neq o$ erhalten.

Fall 1: $n > 0$.

Sei $o'.op'(v'_1, \ldots, v'_{n'})$ oder $new\ C'(v'_1, \ldots, v'_{n'})$ ein geschachtelter Aufruf, der im Rumpf von op oder $C(...)$ ausgeführt wird. Wir nehmen o.B.d.A. an, dass kein weiterer Aufruf im Rumpf von op bzw. $C(...)$ erfolgt. Wegen des Geheimnisprinzips und der generellen Voraussetzungen für Java Programme haben alle Objekte $obj \neq o$ ihren Zustand (bzgl. $State_\Delta$) nicht geändert; d.h. sie erfüllen ihre Invarianten bevor der Aufruf erfolgt.

Im Folgenden betrachten wir nun den geschachtelten Aufruf $o'.op'(v'_1, \ldots, v'_{n'})$ (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, wegen des Geheimnisprinzips, der 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 (2c) und (2d) erfüllen alle existierenden Objekte (auch o) ihre Invarianten bevor der geschachtelte Aufruf ausgeführt wird. Außerdem ist nach Bedingung (2b) und wegen der Verhaltensverträglichkeit bzgl. Subtypen die Vorbedingung der Operationsspezifikation von op' bzw. $C'(...)$ erfüllt bevor der geschachtelte Aufruf ausgeführt wird. Somit ist die Vorbedingung der zugehörigen 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 wegen des Geheimnisprinzips und der generellen Voraussetzungen für Java Programme sowie 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 $\mathcal{T}_\Delta^J$ ein Modell von $CompSpec$ ist.
- Eine Komponentenspezifikation induziert eine Menge von Beweisverpflichtungen.
- Die Erfüllung der Beweisverpflichtungen (durch das Transitionssystem $\mathcal{T}_\Delta^J$) 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.