Kapitel 3

Invarianten, Operationsspezifikationen und Komponentenspezifikationen

Prof. Dr. Rolf Hennicker

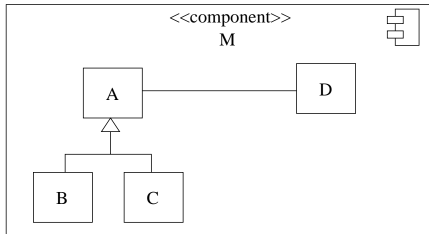
27.05.2010

Ziele

- Komponentenbegriff und Klassensignaturen kennen
- Invarianten für Klassen und Komponenten formulieren können
- Operationsspezifikationen (in der Form von Vor- und Nachbedingungen) schreiben können
- Das Vertragskonzept hinter Operationsspezifikationen verstehen
- Komponentenspezifikationen erstellen können
- Implizite Invarianten aus Klassendiagrammen herleiten können
- Die formale Repräsentation einer Komponentenspezifikation kennen

3.1 Komponenten

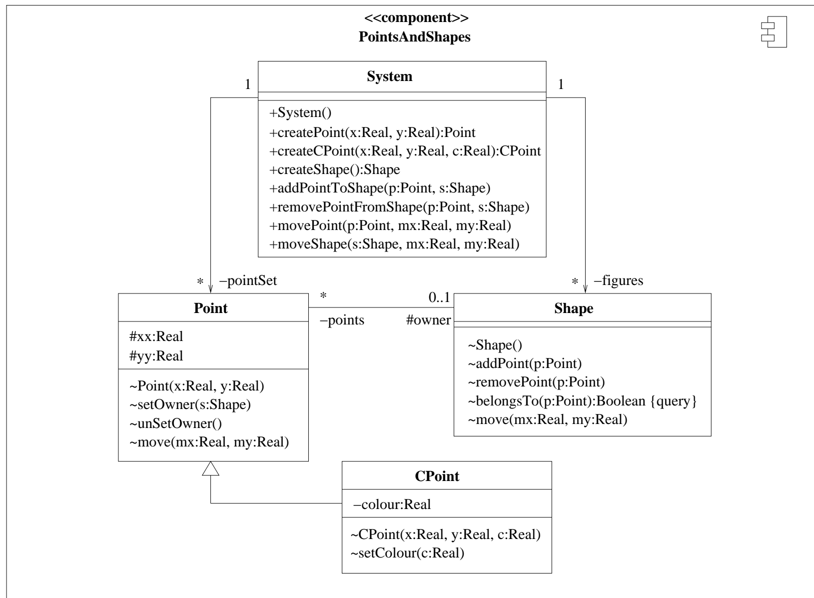
Eine Komponente ist ein modularer Teil eines Systems, mit einer internen Struktur und Schnittstellen zu ihrer Umgebung. Wir betrachten hier nur die interne Sicht von Komponenten, die durch ein Klassendiagramm beschrieben ist.



Annahmen

- Eine Klasse enthält keine zwei Operationen mit demselben Namen und derselben Anzahl von Parametern.
- Beim Überschreiben (Redefinieren) von Operationen in Subklassen wird die Sichtbarkeit erhalten.

Komponente für Punkte und Formen



3.2 Klassensignaturen

Definition (Klassensignatur)

Sei Δ ein Klassendiagramm mit OCL-Signatur $\Sigma_\Delta^{OCL} = (S_\Delta^{OCL}, \leq, F_\Delta^{OCL})$. Die Klassensignatur von Δ ist definiert durch $\Sigma_\Delta = (S_\Delta, \leq, OP_\Delta, \mathit{Visibility})$, wobei

1. $S_\Delta = S_\Delta^{OCL} \cup \{\mathit{Void}\}$
2. $\leq$ ist die partielle Ordnung auf S_Δ^{OCL}
3. $OP_\Delta = F_\Delta^{OCL} \cup M_\Delta \cup Q_\Delta \cup Con_\Delta$, wobei

M_Δ enthält

- für jede Methode $m(x_1 : T_1, \ldots, x_n : T_n)$ einer Klasse C
  $m : C \times T_1 \times \ldots \times T_n \rightarrow \mathit{Void}$
- für jede Methode $m(x_1 : T_1, \ldots, x_n : T_n) : T$ einer Klasse C
  $m : C \times T_1 \times \ldots \times T_n \rightarrow T$

Q_Δ enthält

- für jede Query $q(x_1 : T_1, \ldots, x_n : T_n) : T\ \{\mathit{query}\}$ einer Klasse C
  $q : C \times T_1 \times \ldots \times T_n \rightarrow T$

Con_Δ enthält

- für jeden Konstruktor $C(x_1 : T_1, \ldots, x_n : T_n)$ einer Klasse C
  $C : T_1 \times \ldots \times T_n \rightarrow C$

4. $\mathit{Visibility} : A_\Delta \cup M_\Delta \cup Q_\Delta \cup \mathit{Con}_\Delta \to \{+, \sim, \#, -\}$ ist eine Funktion, wobei

   $+$ steht für komponenten-öffentliche Sichtbarkeit
   $\sim$ steht für komponenten-private Sichtbarkeit
   $\#$ steht für geschützte Sichtbarkeit
   $-$ steht für klassen-private Sichtbarkeit

Notation

$\mathit{Opns}_\Delta =_{def} M_\Delta \cup Q_\Delta \cup \mathit{Con}_\Delta$ bezeichnet die Operationen aus Δ

Beispiel (Punkte und Formen)

$\mathit{Visibility}(op) = +$ für alle Operationen *op* der Klasse System
$\mathit{Visibility}(op) = \sim$ für alle übrigen Operationen
$\mathit{Visibility}(_.xx : \mathit{Point} \to \mathit{Real}) = \#$
$\mathit{Visibility}(_.yy : \mathit{Point} \to \mathit{Real}) = \#$
$\mathit{Visibility}(_.owner : \mathit{Point} \to \mathit{Shape}) = \#$
$\mathit{Visibility}(a) = -$ für alle übrigen Attribute und Rollennamen

$$
\begin{aligned}
M_\Delta &= \{ setColour : CPoint \times Real \rightarrow Void \\
&\quad setOwner : Point \times Shape \rightarrow Void, \\
&\quad unSetOwner : Point \rightarrow Void, \\
&\quad move : Point \times Real \times Real \rightarrow Void, \\
&\quad addPoint : Shape \times Point \rightarrow Void, \\
&\quad removePoint : Shape \times Point \rightarrow Void, \\
&\quad move : Shape \times Real \times Real \rightarrow Void, \\
&\quad createPoint : System \times Real \times Real \rightarrow Point, \\
&\quad \ldots \} \\
Q_\Delta &= \{ belongsTo : Shape \times Point \rightarrow Boolean \} \\
Con_\Delta &= \quad CPoint : Real \times Real \times Real \rightarrow CPoint, \\
&\quad Point : \ Real \times Real \rightarrow Point, \\
&\quad Shape : \rightarrow Shape, \\
&\quad System : \rightarrow System \}
\end{aligned}
$$

3.3 Klassen- und Komponenteninvarianten

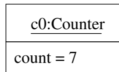
Sei Δ ein Klassendiagramm.

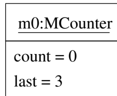
- Für jede Klasse in Δ kann eine Klasseninvariante angegeben werden.
- Eine Klasseninvariante ist ein OCL-Ausdruck vom Typ Boolean, der die möglichen Zustände, unter denen Objekte der betreffenden Klasse von anderen Objekten aus gesehen werden können, einschränkt.

Beispiel (Zähler)

```
(InvCounter)    context Counter
                  inv : count >= 0

(InvMCounter)   context MCounter
                  inv : last >= 0
```





Ein Zustand, in dem die Klasseninvarianten erfüllt sind.

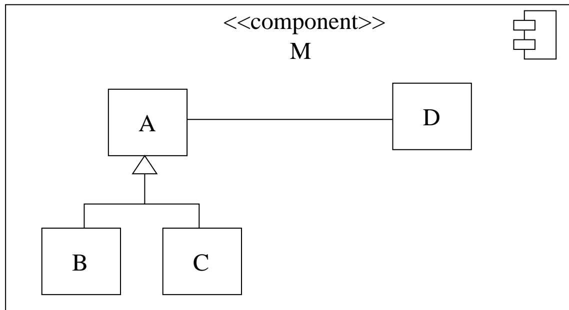
Allgemeine Form von Klasseninvarianten

Eine Invariante für eine Klasse C in Δ wird spezifiziert durch

context C	oder durch	**context** C
inv : Inv		**inv** $invname$: Inv

wobei $Inv \in EXP^{OCL}_{Boolean}$ mit $FV(Inv) \subseteq \{self : C\}$.

Komponenteninvarianten



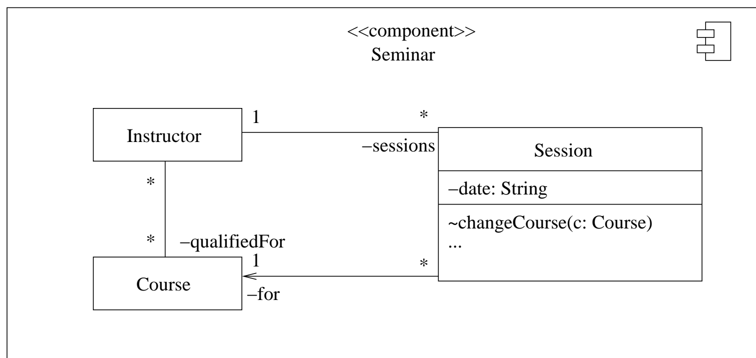
- Für jede Komponente M mit Klassendiagramm Δ kann eine Komponenteninvariante angegeben werden.
- Eine Komponenteninvariante ist ein OCL-Ausdruck vom Typ Boolean, der die möglichen Objektkonfigurationen einschränkt, die von außerhalb der Komponente gesehen werden können.

Beispiel (Punkte und Formen)

Die folgende Komponenteninvariante verlangt, dass assoziierte Punkte und Formen zum selben System gehören:

```
context PointsAndShapes
   inv  invSameSystem:
        System.allInstances() -> forAll(sys |
          sys.pointSet -> forAll(p |
            p.owner <> null implies
              sys.figures -> includes(p.owner)) and
          sys.figures -> forAll(s |
            s.points -> forAll(p |
              sys.pointSet -> includes(p))))
```

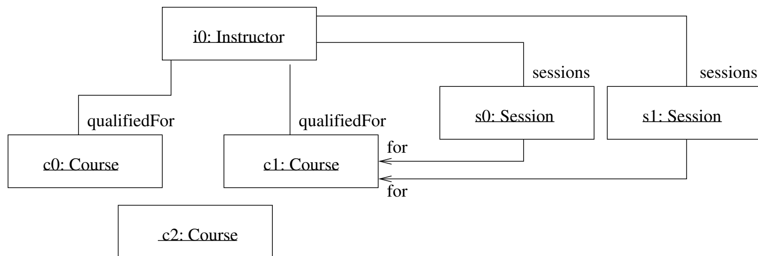
Beispiel (Seminar)




```
(InvSeminar)   context Seminar
                 inv : Instructor.allInstances()->forAll(i|
                         i.qualifiedFor->
                         includesAll(i.sessions.for->asSet()))
```

Beispiel (Seminar)

Ein Zustand, in dem die Komponenteninvariante erfüllt ist:



Allgemeine Form von Komponenteninvarianten

Eine Komponenteninvariante für eine Komponente M mit Klassendiagramm Δ wird spezifiziert durch

context M
inv : Inv

oder durch

context M
inv $invname$: Inv

wobei $Inv \in EXP^{OCL}_{Boolean}$ mit $FV(Inv) = \emptyset$.

3.4 Operationsspezifikationen

Sei Δ ein Klassendiagramm.

- Für jede Operation $op \in Opns_{\Delta}$ kann eine Operationsspezifikation angegeben werden, welche das mögliche Verhalten der Operation einschränkt.
- Eine Operationsspezifikation ist in der Form von Vor- und Nachbedingungen gegeben:
  - Die Vorbedingung muss wahr sein, wenn die Operation aufgerufen wird.
  - Die Nachbedingung muss wahr sein, wenn die Operation beendet ist (d.h. nach Ausführung der Operation) unter der Annahme, dass die Operation korrekt aufgerufen wurde.

Beispiel (Punkte und Formen)

```
context System::addPointToShape(p:Point, s:Shape)
   pre: p<>null and s<>null and
        pointSet -> includes(p) and
        figures -> includes(s) and
        p.owner = null and s.points -> excludes(p)
  post: p.owner = s and
        s.points = s.points@pre -> including(p)

context System::createPoint(x:Real, y:Real):Point
  post: result.oclIsNew() and
        result.xx = x and result.yy = y and
        result.owner = null and
        pointSet = pointSet@pre -> including(result)
```

```
context Point::move(mx:Real, my:Real)
  post: xx = xx@pre + mx and
        yy = yy@pre + my

context Shape::belongsTo(p:Point):Boolean
  post: result = points -> includes(p)

context CPoint::CPoint(x:Real, y:Real, c:Real)
  post: xx = x and yy = y and colour = c

context Point::setOwner(s:Shape)
   pre: s<>null and owner = null
  post: owner = s

context Shape::addPoint(p:Point)
   pre: p<>null and points -> excludes(p)
  post: points = points@pre -> including(p)
```

Allgemeine Form von Operationsspezifikationen

- Für Methoden $(m : C \times T_1 \times \ldots \times T_n \rightarrow \mathit{Void}) \in M_\Delta$ ohne Rückgebewert:

  $$\begin{array}{l} \textbf{context}\ C :: m(x_1 : T_1, \ldots, x_n : T_n) \\ \quad \textbf{pre}\ \ : P \\ \quad \textbf{post} : Q \end{array}$$

  so dass

  - $P \in \mathit{EXP}^{\mathit{OCL}}_{\mathit{Boolean}}, Q \in \mathit{EEXP}^{\mathit{OCL}}_{\mathit{Boolean}}$
  - $\mathit{FV}(P) \subseteq \{\mathit{self} : C, x_1 : T_1, \ldots, x_n : T_n\}$
  - $\mathit{FV}(Q) \subseteq \{\mathit{self} : C, x_1 : T_1, \ldots, x_n : T_n\}$

- Für Methoden $(m : C \times T_1 \times \ldots \times T_n \to T) \in M_\Delta$ mit Rückgabewert:

  context $C :: m(x_1 : T_1, \ldots, x_n : T_n) : T$
  pre : P
  post : Q

  so dass

  - $P \in EXP^{OCL}_{Boolean}, Q \in EEXP^{OCL}_{Boolean}$
  - $FV(P) \subseteq \{self : C, x_1 : T_1, \ldots, x_n : T_n\}$
  - $FV(Q) \subseteq \{self : C, x_1 : T_1, \ldots, x_n : T_n, result : T\}$

- Für Queries $(q : C \times T_1 \times \ldots \times T_n \to T) \in Q_\Delta$ haben Operationsspezifikationen dieselbe Form wie für Methoden mit Rückgabewert.

- Für Konstruktoren $(C : T_1 \times \ldots \times T_n \rightarrow C) \in Con_\Delta$:

  context $C :: C(x_1 : T_1, \ldots, x_n : T_n)$
  pre : P
  post : Q

  so dass

  - $P \in EXP^{OCL}_{Boolean}$, $Q \in EEXP^{OCL}_{Boolean}$
  - $FV(P) \subseteq \{x_1 : T_1, \ldots, x_n : T_n\}$
  - $FV(Q) \subseteq \{self : C, x_1 : T_1, \ldots, x_n : T_n\}$

Anmerkung:
Vor- und Nachbedingungen kann ein benutzerdefinierter Name gegeben werden:

context $C :: op(x_1 : T_1, \ldots, x_n : T_n) : T$
pre *prename* : P
post *postname* : Q

Frame Assumption

Unter einer “Frame Assumption” versteht man eine implizite Verstärkung der Nachbedingung, die besagt, dass außer den explizit geforderten Zustandsänderungen keine weiteren Änderungen des Vorzustands erfolgen dürfen.

Operationsspezifikationen als Verträge

Eine Operationsspezifikation kann als ein Vertrag zwischen

- einem *Kunden*, der die Operation benutzt, und
- einem *Programmierer*, der die Operation implementiert,

betrachtet werden.

Beide einigen sich, dass sie die folgenden Verpflichtungen erfüllen:

Verpflichtung des Kunden:
Der Kunde ruft die Operation nur auf, wenn die Vorbedingung erfüllt ist.

Verpflichtung des Programmierers:
Unter der Voraussetzung, dass die Operation in einem Zustand aufgerufen wird, in dem die Vorbedingung erfüllt ist, sichert der Programmierer zu:

- Die Operation terminiert und verursacht keinen Laufzeitfehler.
- Nach Ausführung der Operation gilt die Nachbedingung.

Beachte:

1. Der Vertrag sagt nichts über die Wirkung der Operation aus, wenn die Vorbedingung nicht erfüllt ist.
2. Falls die Operation eine Query ist, dann sichert der Programmierer zusätzlich zu, dass sich der Zustand des Systems während der Ausführung der Operation nicht ändert.
3. Falls die Operation ein Konstruktor ist, dann sichert der Programmierer zusätzlich zu, dass nach Ausführung des Konstruktors ein neues Objekt erzeugt worden ist.

Bemerkung:

- Je stärker die Vorbedingung ist, desto einfacher wird die "Arbeit" für den Implementierer und desto schwieriger wird die "Arbeit" für den Benutzer.
- Je stärker die Nachbedingung ist, desto schwieriger wird die "Arbeit" für den Implementierer und desto einfacher wird die "Arbeit" für den Benutzer.

Beispiel:

```
public class System {
  ...
  public void addPointToShape(Point p, Shape s) {
    s.addPoint(p);//Benutzung von "addPoint" der Klasse "Shape"
    p.setOwner(s);//Benutzung von "setOwner" der Klasse "Point"
  }
}
```

- Wegen der Vorbedingung “p<>null” und “s.points → excludes(p)” von “addPointToShape” ist beim Aufruf s.addPoint(p) die Vorbedingung von “addPoint” erfüllt.
- Analog ist beim Aufruf p.setOwner(s) die Vorbedingung von “setOwner” erfüllt, wenn man die Frame Assumption für “addPoint” voraussetzt.
- Aus den Nachbedingungen von “addPoint” und “setOwner” folgt dann, dass nach Ausführung von “addPointToShape” die Nachbedingung von “addPointToShape” erfüllt ist, wenn man die Frame Assumption für “setOwner” voraussetzt.

Vertragszusätze im Kontext von Invarianten

Für komponenten-private Operationen:

Für den Kunden:
Der Kunde verpflichtet sich zusätzlich die Operation nur dann aufzurufen, wenn alle Klasseninvarianten erfüllt sind.

Für den Programmierer:
Der Programmierer kann auch vorraussetzen, dass beim Operationsaufruf alle Klasseninvarianten erfüllt sind, verpflichtet sich aber zusätzlich dafür zu sorgen, dass nach Ausführung der Operation wieder alle Klasseninvarianten erfüllt sind.

Für komponenten-öffentliche Operationen:

Für den Kunden:
Der Kunde verpflichtet sich zusätzlich die Operation nur dann aufzurufen, wenn alle Klassen- und Komponenteninvarianten erfüllt sind.

Für den Programmierer:
Der Programmierer kann auch vorraussetzen, dass beim Operationsaufruf alle Klassen- und Komponenteninvarianten erfüllt sind, verpflichtet sich aber zusätzlich dafür zu sorgen, dass nach Ausführung der Operation wieder alle Klassen- und Komponenteninvarianten erfüllt sind.

Bemerkung:

Unter bestimmten Voraussetzungen, die später untersucht werden (keine öffentlichen Attribute, Lokalitätsprinzip von Klasseninvarianten), können die Verpflichtungen beider Beteiligter zur Berücksichtigung der Invarianten deutlich vereinfacht werden.

Methodisches Vorgehen bei der Erstellung von Operationsspezifikationen

1. Die offensichtlichen Vorbedingungen formulieren.
2. Die gewünschten Nachbedingungen angeben.
3. Die Vorbedingungen ggf. verstärken um die Verträglichkeit der Nachbedingung mit den Invarianten zu gewährleisten.

3.6 Komponentenspezifikationen

Eine Komponentenspezifikation

$$CompSpec = (\langle M, \Delta \rangle, Invs^e, OpSpecs)$$

besteht aus

- einer Komponente M mit Klassendiagramm Δ,
- einer Menge $Invs^e$ von (expliziten) Klasseninvarianten und (expliziten) Komponenteninvarianten und
- einer Menge $OpSpecs$ von Operationsspezifikationen für die Operationen aus $Opns_\Delta$.

Annahme:

$OpSpecs$ enthält für jede Operation $op \in Opns_\Delta$ genau eine Operationsspezifikation.

(Default: `context C::op(`$x_1 : T_1$`,...,`$x_n : T_n$`) pre: true post: true`)

Komposition von Operationsspezifikationen

Gegeben seien zwei Operationsspezifikationen für dieselbe Operation:

(*Spec1*) **context** $C :: op(x_1 : T_1, \ldots, x_n : T_n) : T$
pre : P_1
post : Q_1

(*Spec2*) **context** $C :: op(x_1 : T_1, \ldots, x_n : T_n) : T$
pre : P_2
post : Q_2

Wir betrachten zwei Möglichkeiten, wie die beiden Operationsspezifikationen in eine Operationsspezifikation überführt werden können.

1. Join-Operator

Join(Spec1, Spec2) ist definiert durch

$$\begin{aligned} \textbf{context} \quad & C :: op(x_1 : T_1, \ldots, x_n : T_n) : T \\ \textbf{pre} : \quad & P_1 \text{ and } P_2 \\ \textbf{post} : \quad & Q_1 \text{ and } Q_2 \end{aligned}$$

Beispiel (Join)

Counter1
count: Integer last: Integer
dec() ...

```
(Spec1)             context Counter1::dec()
                       pre: count > 0
                      post: count = count@pre - 1

(Spec2)             context Counter1::dec()
                       pre: true
                      post: last = count@pre

 Join(Spec1,Spec2)  context Counter1::dec()
                       pre: count > 0
                      post: count = count@pre - 1 and
                            last = count@pre
```

2. Combine-Operator

Combine(Spec1, Spec2) ist definiert durch

$$\begin{array}{rl} \textbf{context} & C :: op(x_1 : T_1, \ldots, x_n : T_n) : T \\ \textbf{pre} : & P_1 \text{ or } P_2 \\ \textbf{post} : & (P_1@pre \text{ implies } Q_1) \text{ and} \\ & (P_2@pre \text{ implies } Q_2) \end{array}$$

wobei $P_i@pre$ (für $i = 1, 2$) den OCL-Ausdruck bezeichnet, der aus P_i entsteht, wenn man

- alle Vorkommen von Namen a mit $(_.a : C \rightarrow T) \in A_\Delta$ ersetzt durch $a@pre$ und
- alle Vorkommen von Ausdrücken $D.allInstances()$ ersetzt durch $D.allInstances@pre()$.

Beispiel (Combine)

Counter2
count: Integer
dec() ...

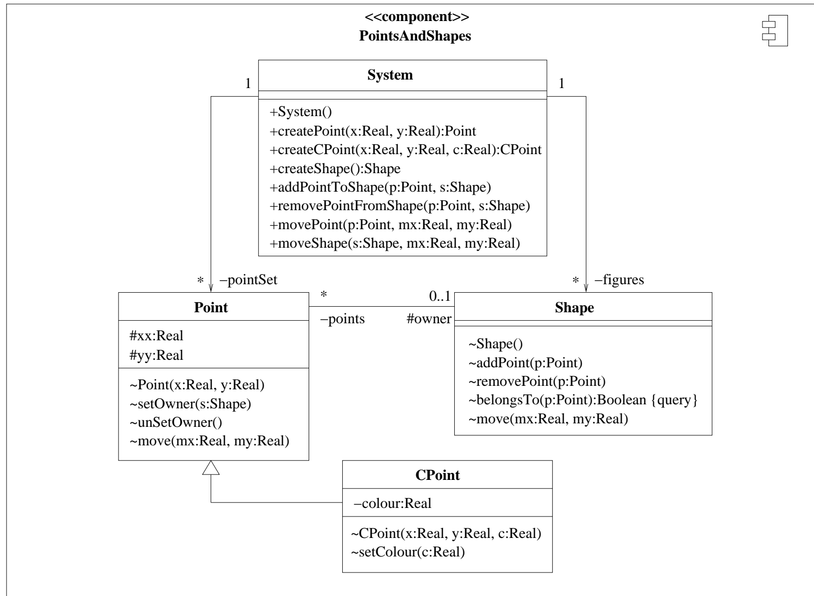
```
(Spec1)           context Counter2::dec()
                     pre: count > 0
                    post: count = count@pre - 1

(Spec2)           context Counter2::dec()
                     pre: count = 0
                    post: count = 0

Combine(Spec1,Spec2)

                  context Counter2::dec()
                     pre: count >= 0
                    post: (count@pre > 0 implies count = count@pre - 1) and
                          (count@pre = 0 implies count = 0)
```

Komponentenspezifikation für Punkte und Formen



- Es wird keine explizite (nicht-triviale) Klasseninvariante gefordert.
- Es wird die explizite Komponenteninvariante *invSameSystem* von Abschnitt 3.3 verwendet, die verlangt, dass assoziierte Punkte und Formen zum selben System gehören:

```
context PointsAndShapes
   inv  invSameSystem:
        System.allInstances() -> forAll(sys |
          sys.pointSet -> forAll(p |
            p.owner <> null implies
              sys.figures -> includes(p.owner)) and
          sys.figures -> forAll(s |
            s.points -> forAll(p |
              sys.pointSet -> includes(p))))
```

- Als Operationsspezifikationen werden die in Abschnitt 3.4 angegebenen Vor- und Nachbedingungen verwendet, ergänzt um geeignete Spezifikationen für die restlichen Operationen.

3.7 Formale Repräsentation von Komponentenspezifikationen

Motivation

- Ein Klassendiagramm Δ drückt Bedingungen für Assoziationen durch Multiplizitäten und Bidirektionalitäts-Anforderungen aus.
- Solche Bedingungen führen zu "impliziten" Klassen- und Komponenteninvarianten, die berücksichtigt werden müssen, wenn man eine präzise Semantik von Komponentenspezifikationen angeben sowie über die Korrektheit von Realisierungen (d.h. Programmen) sprechen will.

Definition

Sei $CompSpec = (\langle M, \Delta \rangle, Invs^e, OpSpecs)$ eine Komponentenspezifikation.
Die formale Repräsentation von *CompSpec* ist gegeben durch

$$FRep(CompSpec) = (\langle M, \Sigma_\Delta \rangle, Invs, OpSpecs)$$

wobei

- Σ_Δ die Klassensignatur von Δ ist und
- $Invs = Invs^e \cup Invs^i$ die Menge aller expliziten und impliziten Invarianten ist.

Im Folgenden wird beschrieben, wie implizite Klassen- und Komponenteninvarianten hergeleitet werden.

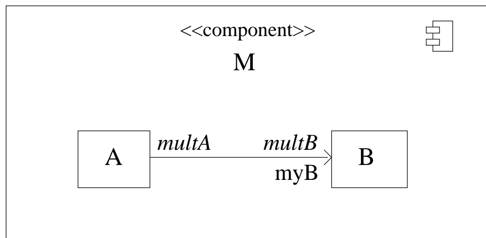
Anmerkung:

$\langle M, \Delta \rangle$ enthält dieselbe Information wie $\langle M, \Sigma_\Delta \rangle$ zusammen mit $Invs^i$.

Implizite Klassen- und Komponenteninvarianten

Multiplizitäten und bidirektionale Assoziationen induzieren Bedingungen für Objektzustände und Objektkonfigurationen, die durch implizite Klassen- und Komponenteninvarianten formalisiert werden.

Multiplizitäten an den beiden Enden unidirektionaler Assoziationen



1. Klasseninvarianten, die durch Multiplizitäten an navigierbaren Assoziationsenden induziert werden

Fall 1.1: $multB = 0..1$
ist bereits vollständig formalisiert durch das Symbol $_.myB: \; A \rightarrow B$

Fall 1.2: $multB = 1$
induziert die Klasseninvariante

context A
inv: $myB <> null$

Fall 1.3: $multB = *$
ist bereits vollständig formalisiert durch das Symbol $_.myB: \; A \rightarrow Set(B)$

Fall 1.4: $multB = 1..*$
induziert die Klasseninvariante

context A
inv: $myB \rightarrow exists(b \mid b <> null)$

2. Komponenteninvarianten, die durch durch Multiplizitäten an nicht navigierbaren Assoziationsenden induziert werden

Fall 2.1: $multA = 0..1,\ multB \leq 1$ induziert die Komponenteninvariante

$$\textbf{context}\ \ M$$
$$\textbf{inv: } B.allInstances() \rightarrow forAll(b \mid A.allInstances() \rightarrow select(a \mid a.myB = b) \rightarrow size() <= 1)$$

Fall 2.2: $multA = 1,\ multB \leq 1$ induziert

$$\textbf{context}\ \ M$$
$$\textbf{inv: } B.allInstances() \rightarrow forAll(b \mid A.allInstances() \rightarrow one(a \mid a.myB = b))$$

Fall 2.3: $multA = *$ induziert keine Einschränkung

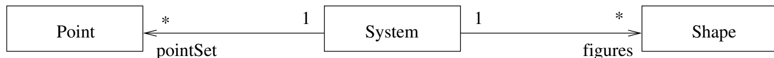
Fall 2.4: $multA = 1..*,\ multB \leq 1$ induziert

$$\textbf{context}\ \ M$$
$$\textbf{inv:}\ B.allInstances() \rightarrow forAll(b \mid A.allInstances() \rightarrow exists(a \mid a.myB = b))$$

Fall 2.5: Falls $multB > 1$ dann ersetze in den obigen Fällen

$a.myB = b$ durch $a.myB \rightarrow includes(b)$

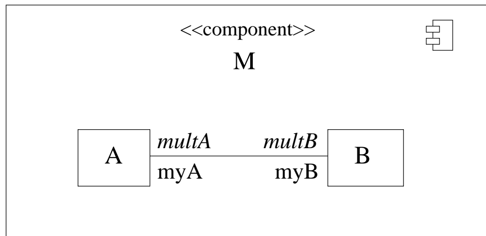
Beispiel (Punkte und Formen)



induziert die folgende Komponenteninvariante:

```
context PointsAndShapes
   inv  invOneSystem:
        Point.allInstances() -> forAll(p |
          System.allInstances() -> one(sys |
            sys.pointSet -> includes(p))) and
        Shape.allInstances() -> forAll(s |
          System.allInstances() -> one(sys |
            sys.figures -> includes(s)))
```

Bidirektionale Assoziationen



induzieren die folgenden impliziten Invarianten:

1. Klasseninvariante für *A* induziert durch die Multiplizität am navigierbaren Assoziationsende mit Rollennamen *myB* (wie im unidirektionalen Fall).
2. Klasseninvariante für *B* induziert durch durch die Multiplizität am navigierbaren Assoziationsende mit Rollennamen *myA* (wie im unidirektionalen Fall).
3. **Komponenteninvariante** für *M* induziert durch die Anforderung der Bidirektionalität.

Fall 3.1: $multA = multB = 0..1$ induziert die Komponenteninvariante

$$\begin{array}{l} \textbf{context} \;\; M \\ \quad \textbf{inv:} \; A.allInstances() \rightarrow forAll(a \mid \\ \qquad\qquad a.myB <> null \;\; implies \;\; a.myB.myA = a) \;\; and \\ \qquad\quad B.allInstances() \rightarrow forAll(b \mid \\ \qquad\qquad b.myA <> null \;\; implies \;\; b.myA.myB = b) \end{array}$$

Fall 3.2: $multA = multB = 1$ induziert die Komponenteninvariante

$$\begin{array}{l} \textbf{context} \; M \\ \quad \textbf{inv:} \; A.allInstances() \rightarrow forAll(a \mid a.myB.myA = a) \;\; and \\ \qquad\quad B.allInstances() \rightarrow forAll(b \mid b.myA.myB = b) \end{array}$$

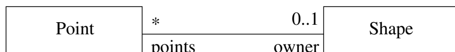
Fall 3.3: $multA = 0..1, \; multB = 1$ oder $multA = 1, \; multB = 0..1$
ist eine Mischung aus den Fällen 3.1 und 3.2

Fall 3.4: $multA = * \;\; und \;\; multB = *$ induziert die Komponenteninvariante

$$\begin{array}{l} \textbf{context} \;\; M \\ \quad \textbf{inv:} \; A.allInstances() \rightarrow forAll(a \mid \\ \qquad\qquad a.myB \rightarrow forAll(b \mid b <> null \;\; implies \;\; b.myA \rightarrow includes(a)) \;\; and \\ \qquad\quad B.allInstances() \rightarrow forAll(b \mid \\ \qquad\qquad\quad b.myA \rightarrow forAll(a \mid a <> null \;\; implies \;\; a.myB \rightarrow includes(b)) \end{array}$$

Alle übrigen Fälle sind geeignete Mischformen der oben betrachteten Fälle.

Beispiel (Punkte und Formen)



induziert die folgende Komponenteninvariante:

```
context PointsAndShapes
   inv  invBidirect:
        Point.allInstances() -> forAll(p |
          p.owner <> null implies
            p.owner.points -> includes(p)) and
        Shape.allInstances() -> forAll(s |
          s.points -> forAll(p |
            p <> null implies p.owner = s))
```

Formale Repräsentation der Komponente für Punkte und Formen

$$(\langle \mathit{PointsAndShapes}, \Sigma_\Delta \rangle, \mathit{Invs}, \mathit{OpSpecs})$$

wobei

- Σ_Δ die Klassensignatur aus Abschnitt 3.2 ist,
- $\mathit{Invs} = \{\mathit{invSameSystem}, \mathit{invOneSystem}, \mathit{invBidirect}\}$,
- *OpSpecs* die in Abschnitt 3.4 angegebenen Operationsspezifikationen sind.

3.8 Zusammenfassung

- Jedem Klassendiagramm Δ kann eine Klassensignatur $\Sigma_\Delta = (S_\Delta, \leq, OP_\Delta, \mathit{Visibility})$ zugeordnet werden, die eine Erweiterung der OCL-Signatur Σ_Δ^{OCL} ist.
- Eine Klasseninvariante spezifiziert eine Eigenschaft, die von allen Objekten der Klasse "von außen" betrachtet, erfüllt sein muss.
- Eine Komponenteninvariante spezifiziert eine Eigenschaft, die von allen Objektkonfigurationen erfüllt sein muss, wenn man die Komponente "von außen" betrachtet.
- Für alle Operationen $op \in \mathit{Opns}_\Delta = M_\Delta \cup Q_\Delta \cup \mathit{Con}_\Delta$ können Operationsspezifikationen in der Form von Vor- und Nachbedingungen angegeben werden.
- Eine Operationsspezifikation kann als Vertrag zwischen dem Benutzer der Operation (Kunde) und dem Programmierer der Operation angesehen werden.
- Operationsspezifikationen können systematisch (unter Befolgung einer aus drei Schritten bestehenden Methodik) erstellt werden.

- Eine Komponentenspezifikation $\mathit{CompSpec} = (\langle M, \Delta \rangle, \mathit{Invs}^e, \mathit{OpSpecs})$ besteht aus einer Komponente mit Klassendiagramm, einer Menge von expliziten Klassen- und Komponenteninvarianten und einer Menge von Operationsspezifikationen.
- Jede Komponentenspezifikation hat eine formale Repräsentation $\mathit{FRep}(\mathit{CompSpec}) = (\langle M, \Sigma_\Delta \rangle, \mathit{Invs}, \mathit{OpSpecs})$ wobei *Invs* die expliziten Invarianten um implizite Invarianten ergänzt, die sich aus den Multiplizitäten und bidirektionalen Assoziationen von Δ ergeben.