

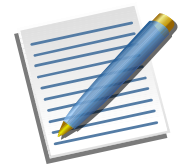


# Software- Engineering für langlebige Systeme

# Softwarewartung auf Modell-Ebene

## VL4

- Evolution
- Refactorings
- Co-Evolution
- Ziele:
  - Theoretische Grundlagen legen

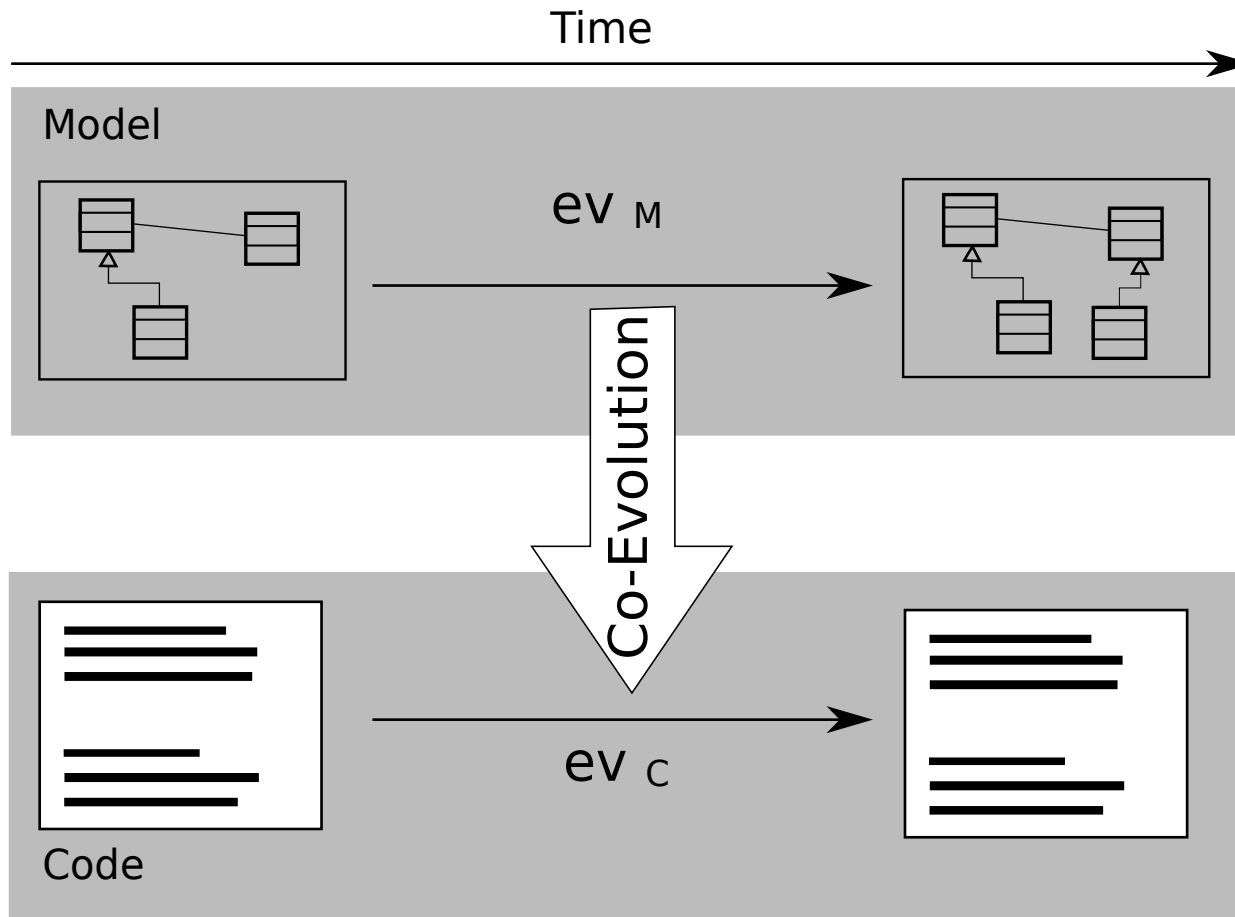


## Wiederholung – Modell

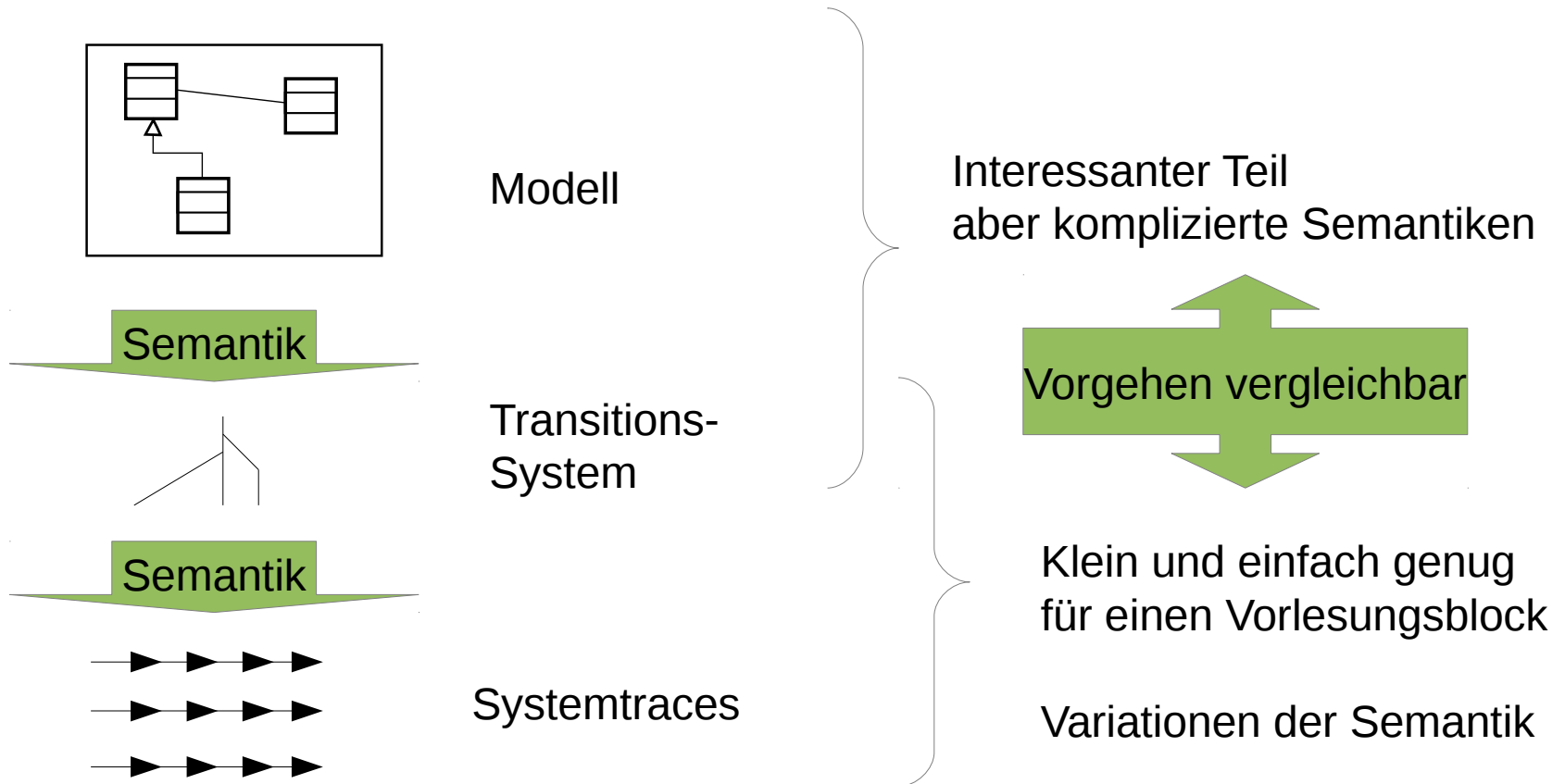
Ein Modell ist ein beschränktes Abbild der Wirklichkeit:

- **Abbildung**
  - Repräsentation eines natürlichen oder eines künstlichen Originals
- **Vereinfachung**
  - Nicht relevante Eigenschaften sind nicht dargestellt
- **Pragmatismus**
  - Modelle gelten nur
    - Für bestimmte Subjekte
    - Innerhalb eines bestimmten Zeitintervalls
    - Für einen bestimmten Zweck

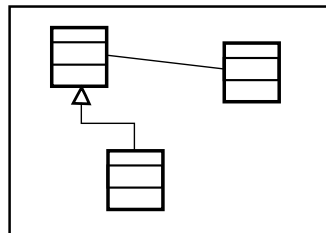
## Modelle und Code



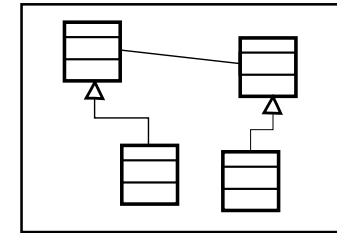
# Aufbau



## Evolution - Ziel

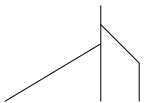


Evolution

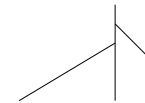


Semantik

Semantik



ähnlich



## Ähnlich Semantik

- Gleichheit ist (meist) zu stark, da sich bei der Evolution etwas ändern soll.
- Es soll nicht etwas vollkommen anders sein, da die Evolution eine Anpassung ist.

### **Sprachvergleich**

- Sprachäquivalenz
- Bisimulation
- Refinement
  - Trace
  - Failure
- Aus Refinement abgeleitete Gleichheit



## Sprach-Äquivalenz

Zwei Transitionssysteme  $T_1$  und  $T_2$  sind sprachäquivalent, wenn Sie die gleiche Menge von Systemtraces  $Tr$  besitzen:

$$Tr(T_1) = Tr(T_2)$$

## Bisimulation [Park, Milner]

Seien  $T_i = (Q_i, \rightarrow_i, q_{0,i})$ ,  $i = 1, 2$  Transitionssysteme

Eine Relation  $R \subseteq Q_1 \times Q_2$  heißt

starke Bisimulation zwischen  $T_1$  und  $T_2$ ,

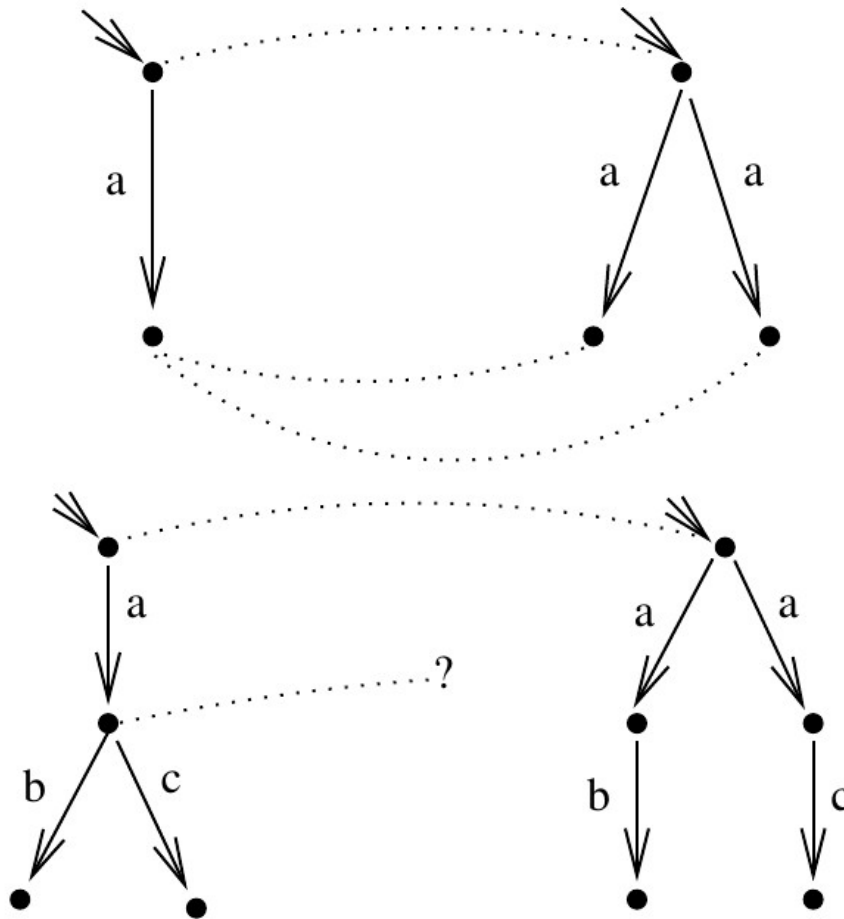
falls  $(q_{0,1}, q_{0,2}) \in R$  ist und für alle  $(q_1, q_2) \in R$ ,  $\alpha \in \text{Act}$  gilt:

1.  $q_1 \xrightarrow{\alpha}_1 p_1 \Rightarrow \exists p_2 : q_2 \xrightarrow{\alpha}_2 p_2$  und  $(p_1, p_2) \in R$ ,
2.  $q_2 \xrightarrow{\alpha}_2 p_2 \Rightarrow \exists p_1 : q_1 \xrightarrow{\alpha}_1 p_1$  und  $(p_1, p_2) \in R$ .

$T_1$  und  $T_2$  heißen stark bisimulationsäquivalent ( $T_1 \sim T_2$ ),

falls es eine starke Bisimulation zwischen  $T_1$  und  $T_2$  gibt.

## Beispiel



(Bisimulation  $\neq$  Sprachäquivalenz)

## Trace - Refinement

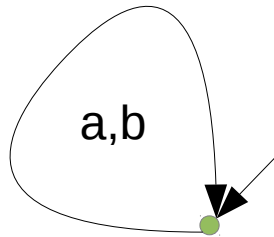
- Idee Refinement:
  - Ein konkretes System ist ein Refinement eines abstrakten Systems, wenn das konkrete System nur Verhalten zeigt welches das abstrakte auch zeigen kann:

- Trace-Refinement:

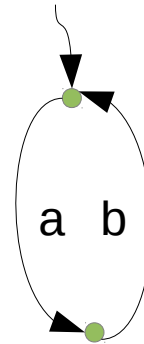
Ein Transitionssystem  $T_c$  ist ein Refinement eines Transitionssystems  $T_a$ , gdw

$$\text{Tr}(T_c) \subseteq \text{Tr}(T_a)$$

## Beispiel



$(a|b)^*$



$(ab)^*$

## Failure

Ein System wird durch seine Failures dargestellt.

Als Failure wird ein Paar  $(s, X)$  bezeichnet :

- $s$  : ein Trace eines Prozesses ist,
- $X$  : Menge der Ereignisse, die nach  $s$  ablehnen kann.

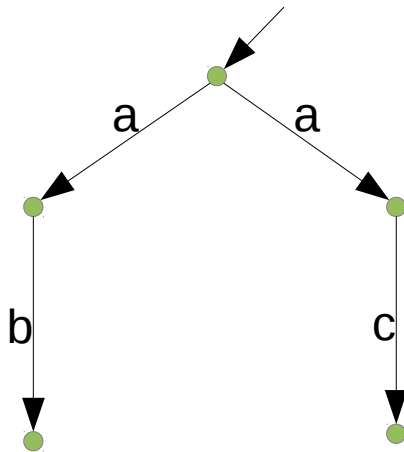
Die Menge  $F$  beschreibt die Menge aller Failures eines Transitionssystems.

Failure-Refinement:

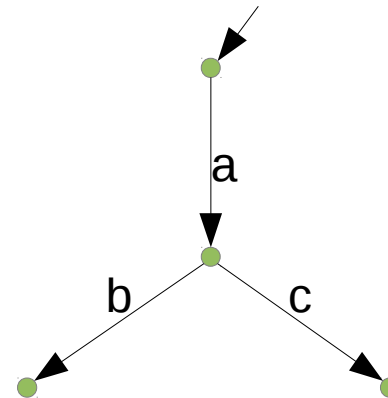
Ein Transitionssystem  $T_c$  ist ein Failure-Refinement eines Transitionssystems  $T_a$ , gdw

$$F(T_c) \subseteq F(T_a)$$

## Beispiel

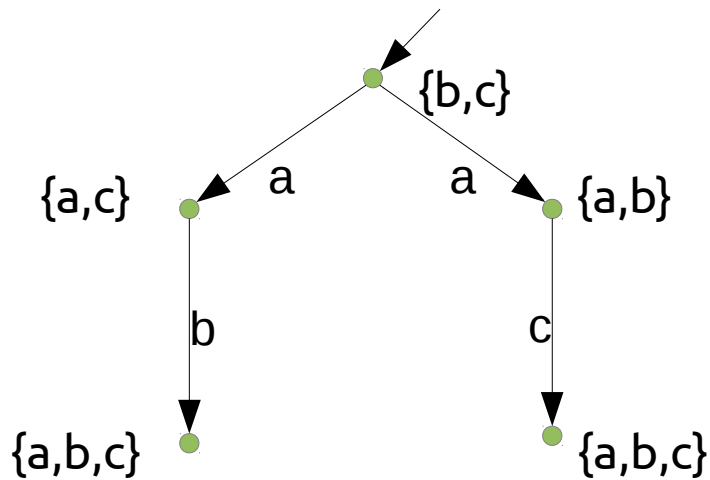


<a,b>  
<a,c>  
<a>  
<>

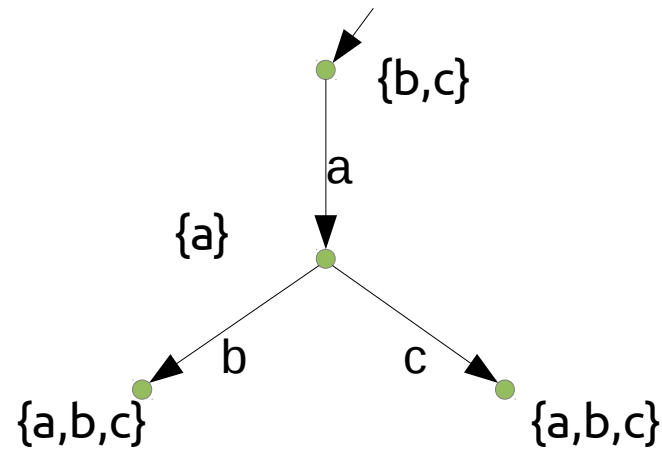


<a,b>  
<a,c>  
<a>  
<>

## Beispiel



$(\langle a,b \rangle, \{a,b,c\}), (\langle a,b \rangle, \{b,c\}), \dots$   
 $(\langle a,c \rangle, \{a,b,c\}), (\langle a,c \rangle, \{b,c\}), \dots$   
 $(\langle a \rangle, \{a,c\}), (\langle a \rangle, \{a,b\}), \dots$   
 $(\langle \rangle, \{b,c\}), (\langle \rangle, \{b\}), (\langle \rangle, \{c\}), (\langle \rangle, \{\})$



$(\langle a,b \rangle, \{a,b,c\}), (\langle a,b \rangle, \{b,c\}), \dots$   
 $(\langle a,c \rangle, \{a,b,c\}), (\langle a,c \rangle, \{b,c\}), \dots$   
 $(\langle a \rangle, \{a\}), (\langle a \rangle, \{\}),$   
 $(\langle \rangle, \{b,c\}), (\langle \rangle, \{b\}), (\langle \rangle, \{c\}), (\langle \rangle, \{\})$



## Refinement- Äquivalenzen

- Refinements bilden eine Halbordnung
  - Transitiv
  - Reflexiv
  - Antisymmetrisch
- Die aus der Antisymmetrie folgende Äquivalenz wird auch als Refinement-Äquivalenz bezeichnet

## Evolutionenfilter

- Bei der Evolution soll das System an definierten Stellen geändert werden, an allen anderen Stellen soll das System gleich bleiben.
- Die sich ändernden Eigenschaften kann man aus der Semantik „herausfiltern“ und dann die Systeme vergleichen:

$T_1$  und  $T_2$  seien zwei Transitionssysteme

$f_i$  sei eine Funktion, die Transitionssysteme auf Transitionssysteme abbildet (Filterfunktion)

Wenn  $f_i(T_1) =_s f_i(T_2)$

sind die Transitionssysteme ähnlich in Bezug auf den Filter  $f_i$  unter der Semantik  $S$ .

## Evolutions-Transformation Formal

Eine Evolutions-Transformation ist eine Funktion  $t$ : Model  $\times$  Parameter  $\rightarrow$  Model unter Berücksichtigung einer Semantik  $s$  und eines Evolutionsfilters  $fi$ , gdw. Wenn für alle  $M_1$ , und  $M_2$  mit

$$M_2 = t(M_1, \dots)$$

gilt

$$fi( [[M_1]] ) =_s fi( [[M_2]] )$$

## Evolution

- Häufig ist es schwer eine Evolution  $ev$  wie in der Definition zu definieren.
  - Gründe:
    - Eine Evolution muss wieder in der Sprache liegen (s. Def)
    - Eine Evolution muss ein wohlgeformtes Ergebnis haben (sonst ist die Semantik in der Def nicht definiert)
    - Es sollen keine exotischen Sonderfälle betrachtet werden müssen.
- Eine Evolution  $ev$  ist ein Tupel  $(p,t)_s$  wobei  $p$  ein Prädikat über das Modell und die Parameter ist.  $t$  ist eine Evolutions-Transformation. Beide Bestandteile hängen evtl. von einer Semantik  $s$  ab.

## Refactorings

- Refactorings sind Evolutionen bei denen der Evolutionsfilter  $f_i$  die Identität ist

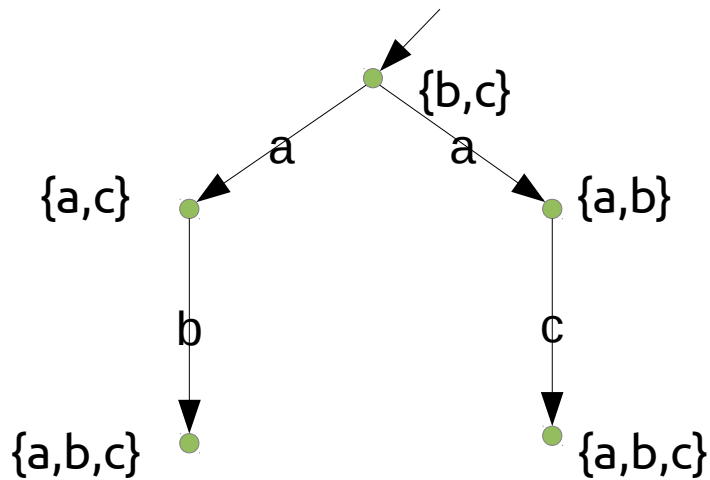
$$f_i = id$$

- Refactorings verändern das Programm/Modell im Sinne der Semantik nicht.

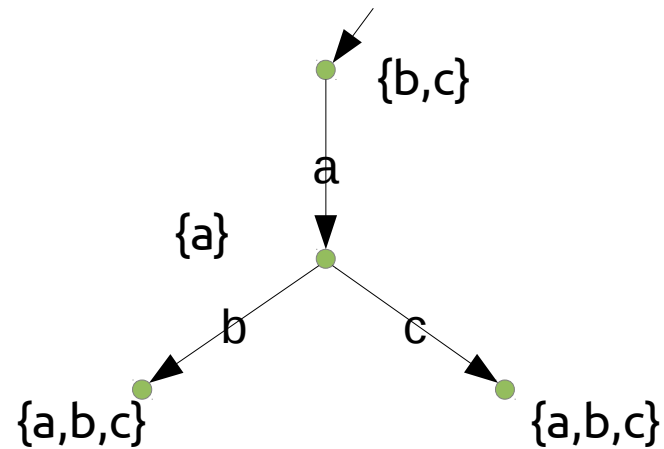
## Exkurs: Code-Refactorings

- Auf Code kann die formale Definition angewandt werden
- Statt einer formalen Semantik findet man die Ausführung des Programms vor.
- Statt formaler Beweise der Verhaltenserhaltung
  - Werden Tools eingesetzt, bei denen man vertraut, dass sie richtig funktionieren
  - Test-Suites zur Prüfung des Verhaltens verwendet.
- Refactorings werden häufig wie Designpattern in Sammlung gepflegt
- Mehr, wenn wir wieder beim Code angekommen sind.....

## Beispiel



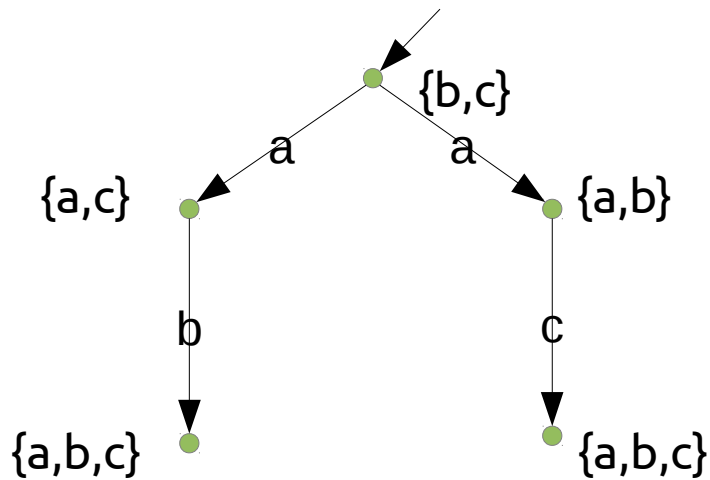
$(\langle a,b \rangle, \{a,b,c\}), (\langle a,b \rangle, \{b,c\}), \dots$   
 $(\langle a,c \rangle, \{a,b,c\}), (\langle a,c \rangle, \{b,c\}), \dots$   
 $(\langle a \rangle, \{a,c\}), (\langle a \rangle, \{a,b\}), \dots$   
 $(\langle \rangle, \{b,c\}), (\langle \rangle, \{b\}), (\langle \rangle, \{c\}), (\langle \rangle, \{\})$



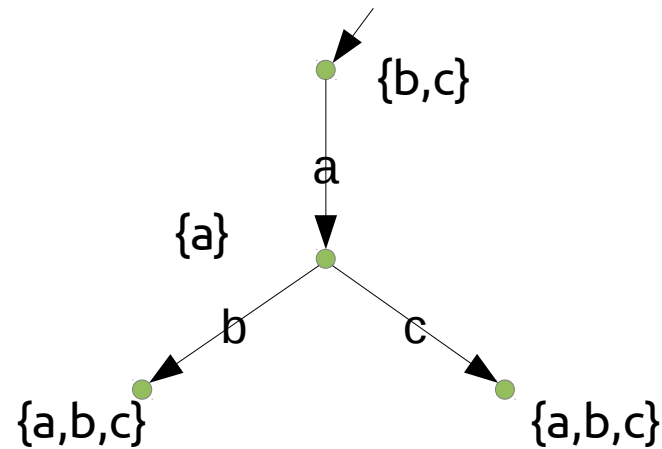
$(\langle a,b \rangle, \{a,b,c\}), (\langle a,b \rangle, \{b,c\}), \dots$   
 $(\langle a,c \rangle, \{a,b,c\}), (\langle a,c \rangle, \{b,c\}), \dots$   
 $(\langle a \rangle, \{a\}), (\langle a \rangle, \{\})$   
 $(\langle \rangle, \{b,c\}), (\langle \rangle, \{b\}), (\langle \rangle, \{c\}), (\langle \rangle, \{\})$

## Beispiel

$$f_i(x) = \{s \mid \exists f \text{ mit } (s,f) \in x\}$$



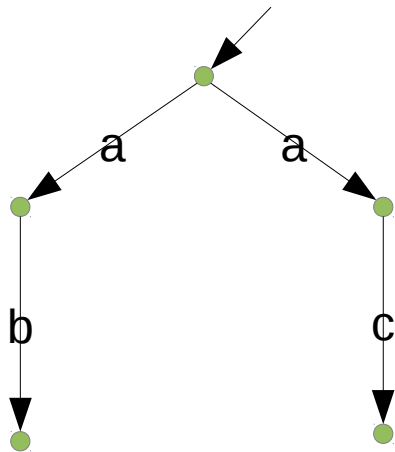
(<a,b>, {a,b,c}), (<a,b>, {b,c}),...  
 (<a,c>, {a,b,c}), (<a,c>, {b,c}),...  
 (<a>, {a,c}), (<a>, {a,b}),...  
 (<>, {b,c}), (<>, {b}), (<>, {c}), (<>, {}))



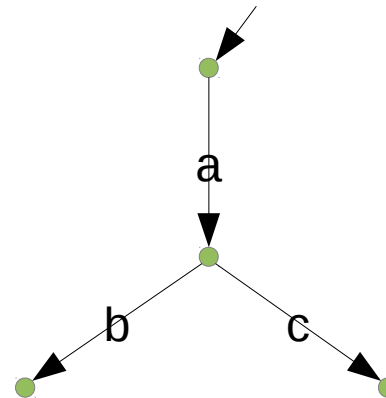
(<a,b>, {a,b,c}), (<a,b>, {b,c}),...  
 (<a,c>, {a,b,c}), (<a,c>, {b,c}),...  
 (<a>, {a}), (<a>, {}),  
 (<>, {b,c}), (<>, {b}), (<>, {c}), (<>, {}))



## Beispiel

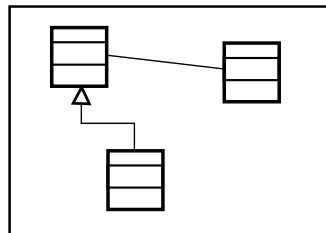


<a,b>  
<a,c>.  
<a>.  
<>

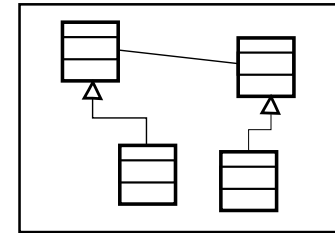


<a,b>  
<a,c>  
<a>  
<>

## Evolution - Ziel

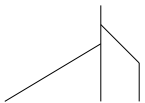


Evolution

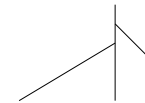


Semantik

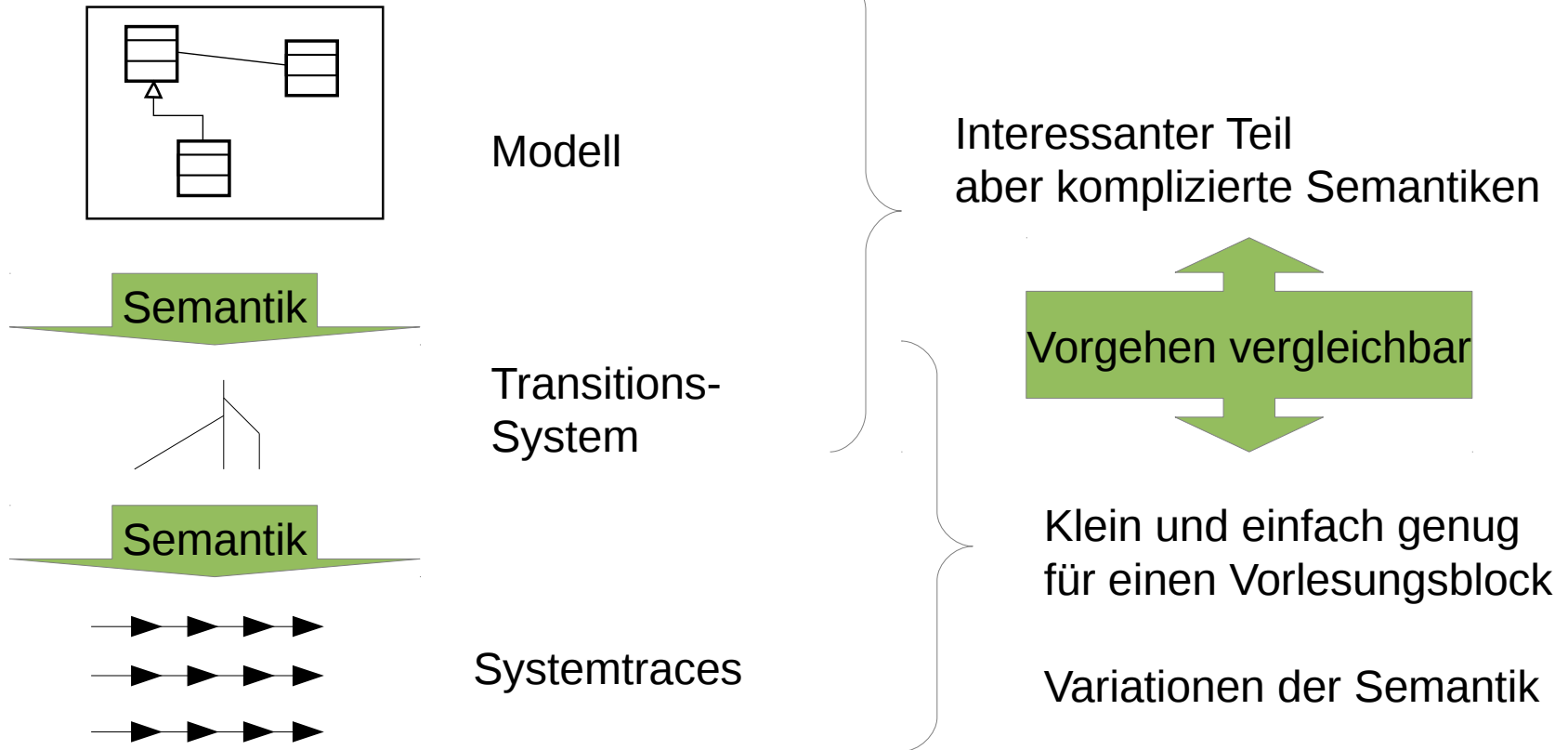
Semantik



ähnlich



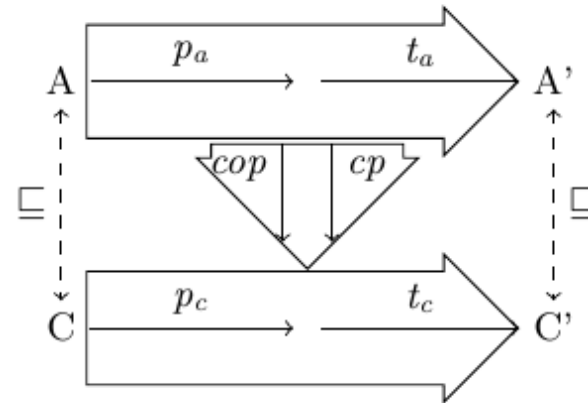
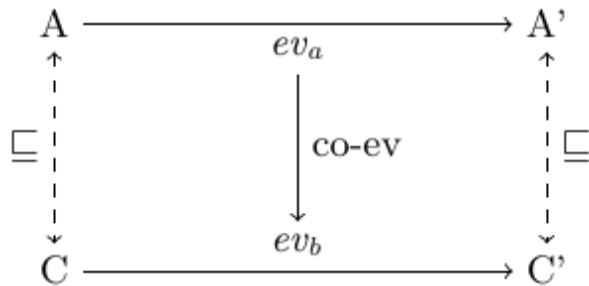
## Bisheriger Aufbau



## Einige übliche Basis-Evolutionen auf objektorientierten Sprachen

- Hinzufügen/Löschen einer leeren Klasse (Refactoring)
- Hinzufügen von Operationen, Feldern, Typen
- Löschen von nicht genutzten Operationen, Feldern, Typen (Refactoring)
- Ändern der Funktion einer Operation
- Hinzufügen/Entfernen von Assoziationen
- ...

## Idee



## Beispiel

- Das Beispiel baut auf Object-Z auf
  - Objektorientierte Erweiterung von Z
  - Idee:
    - Aus ZF-Mengenlehre abgeleitet
    - Invarianten, Vor- und Nachbedingungen
  - Hinweise:
    - Details sind nicht prüfungsrelevant
    - Wichtig: Idee der Basis (prüfungsrelevant)
    - Ein reales und dennoch kleines Beispiel

**Table 1**

Base evolutions for Object-Z classes:  $A$  source Object-Z class,  $A'$  class obtained by evolution.

Name	Parameter	Definition	Application condition
stateExt	$u : T$ (var. decl.)	$A'.State \hat{=} A.State \wedge [u : T]$	$u \notin \mathbf{vars}(A.State)$
stateRem	$u : T$ (var. decl.)	$A'.State \hat{=} A.State \setminus [u : T]$	$[u : T] \in \mathbf{decls}(A.State)$ $\wedge u \notin \mathbf{vars}(A.Init) \wedge \forall j \in A.J : u \notin \mathbf{vars}(A.Op_j)$
initExt	$pred$ (predicate)	$A'.Init \hat{=} A.Init \wedge pred$	–
initRem	$pred$ (predicate)	$A'.Init \hat{=} A.Init \setminus \{pred\}$	$pred \in \mathbf{preds}(A.Init)$
newOp	$NOp$ (op. schema)	$A'.I \hat{=} A.I \cup \{n\}, n = \#A.I + 1$ $A'.J \hat{=} A.J \cup \{n\}$ $A'.Op_n \hat{=} NOp$	–
remOp	$l$ (op. index)	$A'.I \hat{=} A.I \setminus \{l\}$ $A'.J \hat{=} A.J \setminus \{l\}$	$l \in A.I$
hideOp	$l$ (op. index)	$A'.I \hat{=} A.I \setminus \{l\}$	$l \in A.I$
makeOpVis	$l$ (op. index)	$A'.I \hat{=} A.I \cup \{l\}$	$l \in A.J$

**Table 2**

Complex evolutions for Object-Z classes:  $A$  source Object-Z class,  $A'$  class obtained by evolution.

Name	Parameter	Definition	Application condition
stateExt2	$u_1 : T_1, \dots, u_n : T_n$ (var. decls.)	$A'.State \hat{=} A.State \wedge [u_1 : T_1, \dots, u_n : T_n]$	$u_1, \dots, u_n \notin \mathbf{vars}(A.State)$
classConj	$B$ (class)	$A'.State \hat{=} A.State \wedge B.State$ $A'.Init \hat{=} A.Init \wedge B.Init$ $A'.Op_j \hat{=} A.Op_j \wedge B.Op_j, j \in A.J \cap B.J$ $A'.Op_j \hat{=} A.Op_j, j \in A.J \setminus B.J$ $A'.Op_j \hat{=} B.Op_j, j \in B.J \setminus A.J$ $A'.I \hat{=} A.I \cup B.I$	—
opExt	$l, pred$ (op. index, pred.)	$A'.Op_l \hat{=} A.Op_l \wedge pred$	—
opConj	$l, k$ (op. indices)	$A'.I = A.I \cup \{n\}, n = \#A.I + 1$ $A'.Op_n \hat{=} A.Op_l \wedge A.Op_k$	$l, k \in A.I$
opChc	$l, k$ (op. indices)	$A'.I = A.I \cup \{n\}, n = \#A.I + 1$ $A'.Op_n \hat{=} A.Op_l \sqcup A.Op_k$	$l, k \in A.I$
ref	$\vec{p}$ (some parameters)	$A' \hat{=} ref(A, \vec{p})$	depends on refactoring



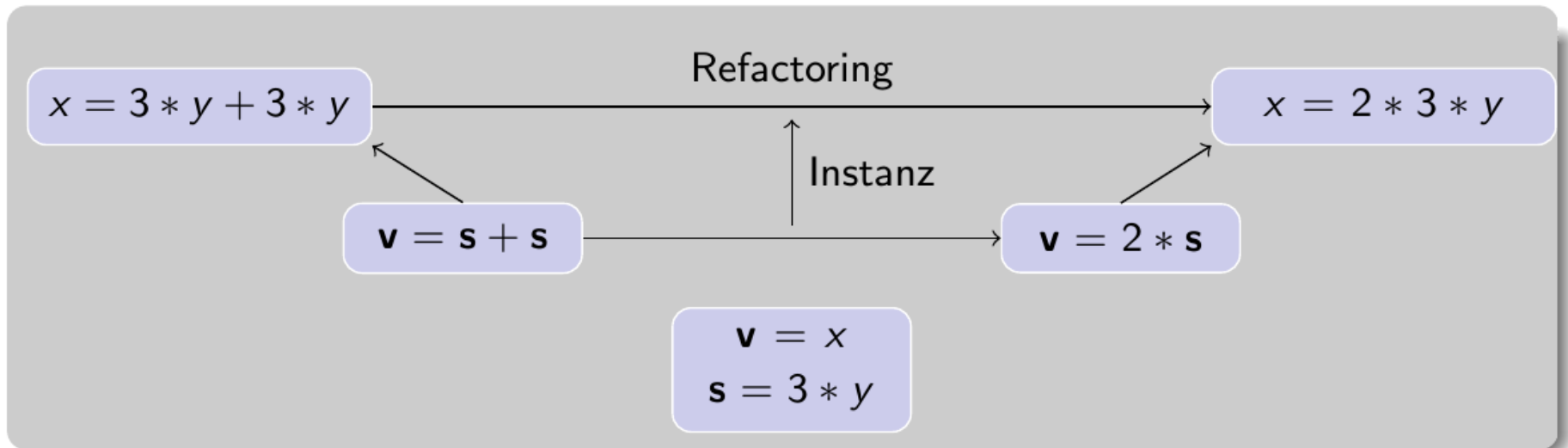
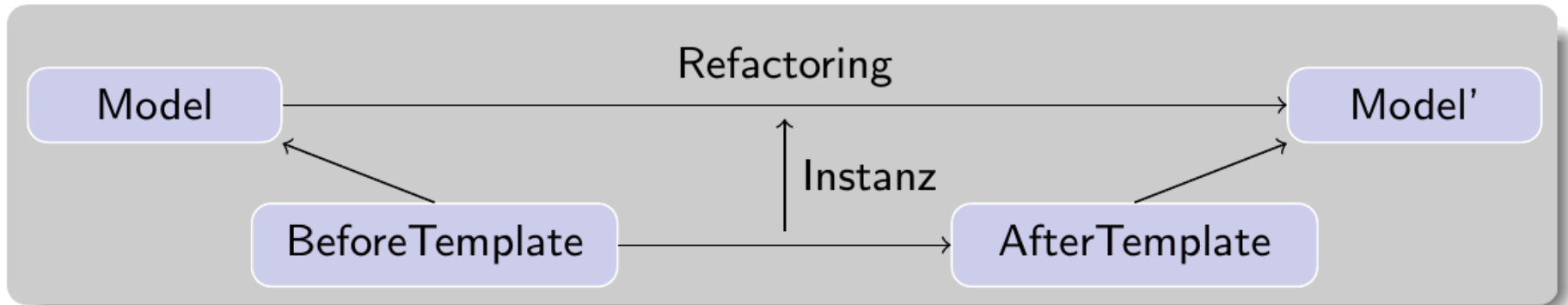
**Table 3**  
Co-evolutions on classes  $A, C$ .

Evolution, parameter	Co-evolution, parameter	Remark
$stateExt, u : T$	$stateExt, u : T$	
$stateRem, u : T$	$id, -$	$stateRem$ is refactoring
$initExt, pred$	$initExt, pred'$	calculation of $pred'$ if $R$ functional
$initRem, pred$	$id, -$	
$newOp, NOP$	$newOp, NOP'$	calculation of $NOP'$ if $R$ functional and total
$remOp, l$	$remOp, l$	
$hideOp, l$	$hideOp, l$	
$makeOpVis, l$	?	option: $newOp$ calculation with schema of $A.Op_l$
$stateExt2, u_1 : T_1, \dots, u_n : T : n$	$stateExt2, u_1 : T_1, \dots, u_n : T : n$	
$class\ Conj, B$	$classConj, B$	if $\mathbf{vars}(B.State) \cap \mathbf{vars}(A.State) = \emptyset$ $\wedge \mathbf{vars}(B.State) \cap \mathbf{vars}(C.State) = \emptyset$ $\wedge B.Init$ satisfiable
$opExt, l, pred$	?	option: re-build $C.Op_l$
$opConj, l, k$	$opConj, l, k$	if separability holds
$opChc, l, k$	$opChc, l, k$	

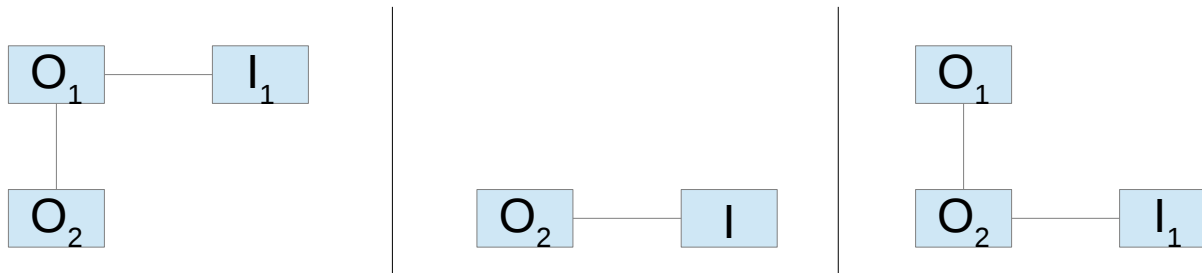
## Techniken - Transformationen

- Graphtransformationssysteme
- Rewriting Systeme
- Templatebasierte Transformationen
- Tripple-Graph-Grammatiken
- ...

## Idee templatebasierter Ansätzen



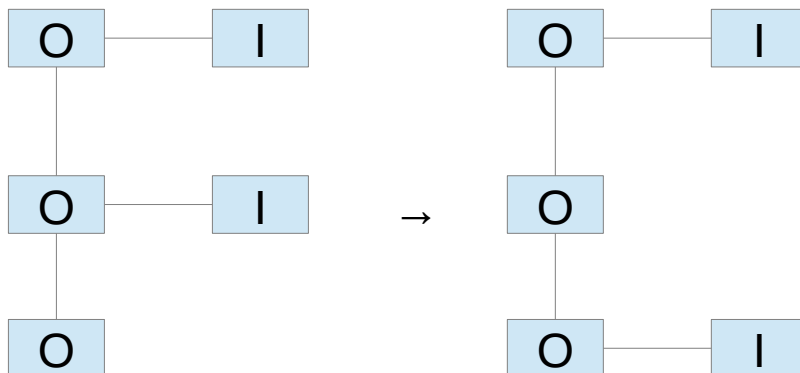
## Idee von Graphtransformationssystemen



Left

NAC

Right



NAC = Non Application Condition

## Forschungsprobleme (möglich Arbeitsthemen in unserer Gruppe)

- Wie Evolutionsfolge ableiten?
  - Logging der Operationen (evtl. Ableitung nötig)
  - Berechnen aus Differenzen der Modelle/des Codes
    - Differenzen auf Modelle
- Auswahl von Co-Evolutionen
  - Für eine Evolution gibt es in der Regel mehrere mögliche Co-Evolutionen
  - Problem: Welche Ausführen

## Freude: Ende des formalen Teil

- Co-Evolutionen werden beim „Top-Down gehen“ aufgegriffen.
- Nächstes Mal: Projekt-Management unter Gesichtspunkten der Langlebigkeit