Vorlesung „Aspektorientierte Softwareentwicklung“

Sommersemester 2008

Chapter 11.
Aspect Interaction Analysis

Günter Kniesel

Computer Science Department

University of Bonn

gk@cs.uni-bonn.de

universitätbonn ROOTS

Interactions and Interferences

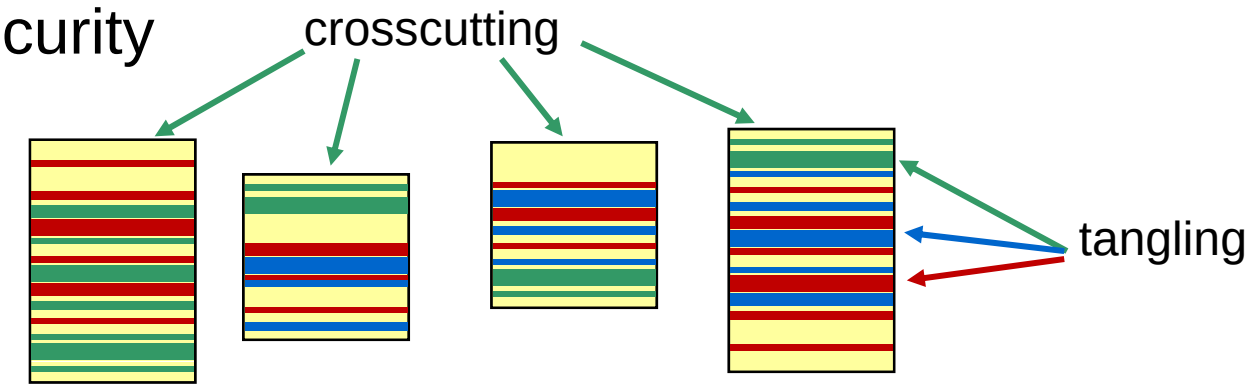
The Problem

Distinguishing Interactions and Interferences

Classification of Interferences and Examples

The Bad

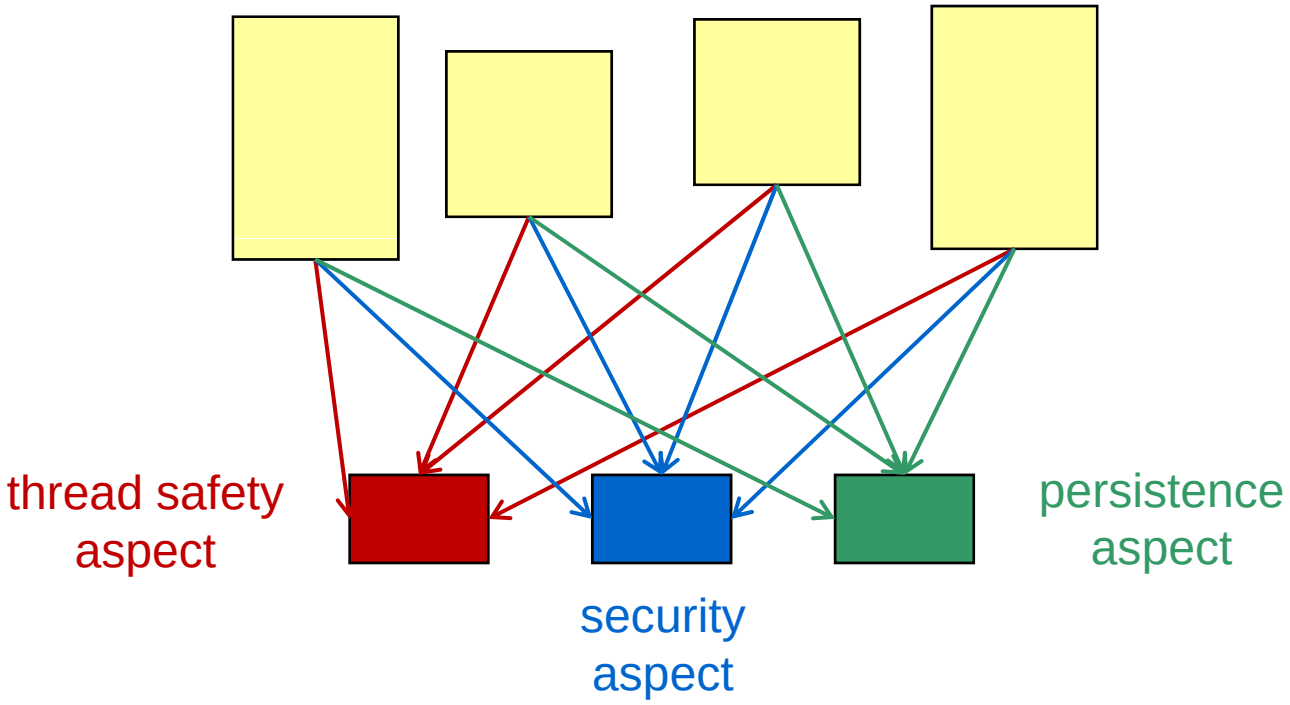
- Problem: crosscutting and tangled concerns
  - thread safety
  - persistence
  - security
  - ...



The Good

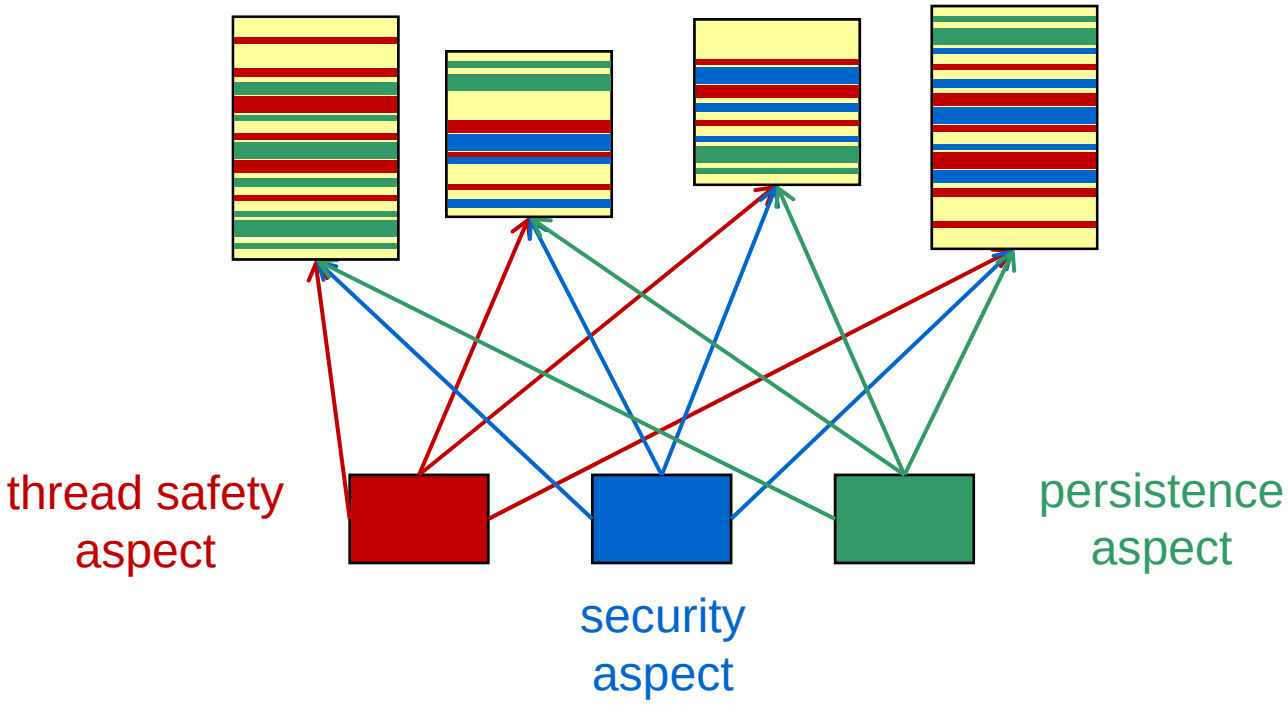
- Multi-dimensional separation of concerns

- Aspects = Modular representation of concerns

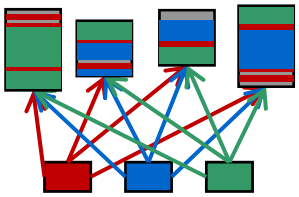


The Ugly

- To be effective, aspects must be "woven" back into their base classes
- Aspect Interaction Problem: How to compose independently developed aspects???



Aspect Interaction



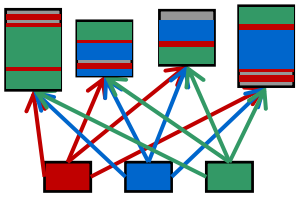
- Aspect-base interaction
  - Each aspect interacts with the base class to which it is applied
  - Changes its semantics in desired and undesired ways

- Aspect-aspect interaction
  - Each aspect interacts with the other aspects applied to the same base class
  - may be intended
  - may be unintended but not undesired
  - may be undesired
  - may be broken

aspect interference

How to distinguish these?

Aspect Interaction



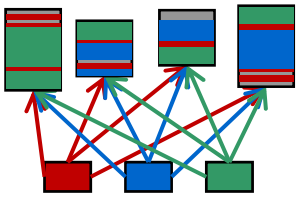
- Aspect-base interaction
  - Each aspect interacts with the base class to which it is applied
  - Changes its semantics

- Aspect-aspect interaction
  - Each aspect interacts with the other aspects applied to the same base class
  - may be intended
  - may be unintended but not undesired
  - may be undesired but not broken
  - may be broken

aspect interference

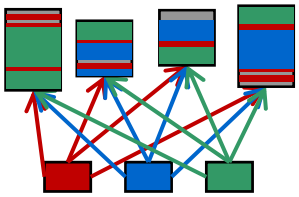
How to distinguish these?

Unintended Interaction



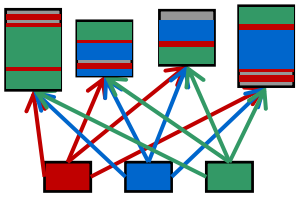
- An aspect interacts with another one although the programmer didn't anticipate it
- The interaction is necessary for fulfilling the task of the aspect
  - E.g. logging applies to a component of which I didn't even know that it exists
  - That‘s OK, I want to log everything, anyway!
- Analysis makes interactions visible
  - Programmer gets confidence into his code → 'Hey, it works better than I thought!'

Undesired Interaction



- An aspect interacts with onother one although it shouldn't
- Possibly a consequence of a too generic pointcut
  - E.g. Security aspect contains a wildcard template that matches code in other aspects, preventing them from working as intended
  - Not OK, I intended the security policy only for the base code. My aspects must have more priviledges.
- Analysis makes intractions visible
  - Programmer becomes aware of undesired interactions and can fix them

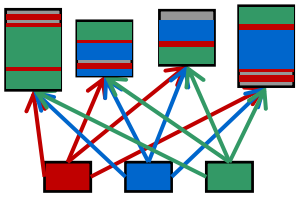
Undesired vs. Broken Interaction



- Undesired interaction ($\subset$ Interference)
  - An aspect interacts with onother one although it shouldn't
    - ⇨ Write Interference: Change state that affects behaviour of other
    - ⇨ Read Interference: Read state that should not be accessible
    - ⇨ Output Interference: Side effects in the wrong order

- Broken interaction ($\subset$ Interference)
  - Two aspects interact in a way that will lead to problems at weave-time
    - ⇨ Incomplete weaving
    - ⇨ Incorrect weaving
    - ⇨ Infinite weaving
  - Analysis makes intraction patterns visible and diagnoses the fault
    - ⇨ Programmer must find a solution, otherwise the program is unweavable

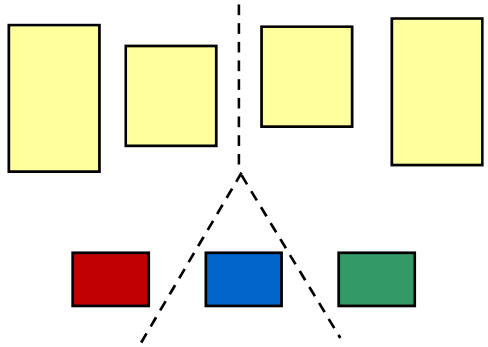
Topics of this Talk



- Aspect-aspect interaction and interference
- Interaction Detection
  - How to capture interaction precisely?
- Interaction Resolution
  - How to control interaction such that it does not break anything?
- Interference Detection
  - How to capture interference precisely?

Constraints

- Independent Development
  - Base classes
  - Aspects
- Aspects as Components
  - Implementation details unknown
- Unanticipated Composition
  - Nobody can tell you how aspects related to each other
- Stressed Programmers
  - Lightweight approach
  - No extra specifications of aspect (or base) behaviour



Ordering of Aspects In AspectJ

- Possibly Interfering Aspects

```
aspect DisallowNulls {
    pointcut allTypeMethods(Type obj): call(* *(..)) && args(obj, ..);

    before(Type obj): allTypeMethods(obj) {
        if (obj == null) throw new RuntimeException();
    }
}
```

```
aspect CountEntry {
    pointcut allTypeMethods(Type obj): call(* *(..)) && args(obj, ..);

    static int count = 0;
    before(): allTypeMethods(Type) {
        count++;
    }
}
```

- Explicit Aspect Precedence Declaration

```
aspect Ordering {
    declare precedence: CountEntry, DisallowNulls;
}
```

Ordering of Aspects isn‘t sufficient!

- Example
  - Base program: contains method „work()“
  - Aspect „Move“: goInsideRoom before starting work and goOutsideRoom afterwards
  - Aspect „Door“: openTheDoor before starting work and closeTheDoor afterwards
  - Only sensible order
    - ⇨ openTheDoor goInsideRoom work goOutsideRoom closeTheDoor
  - Not achievable with aspect-level ordering!
    - ⇨ Too coarse-grained
  - Ostermann and Kniesel: „Independent Extensibility -- an open challenge for AspectJ and Hyper/J“, March 30, 2000

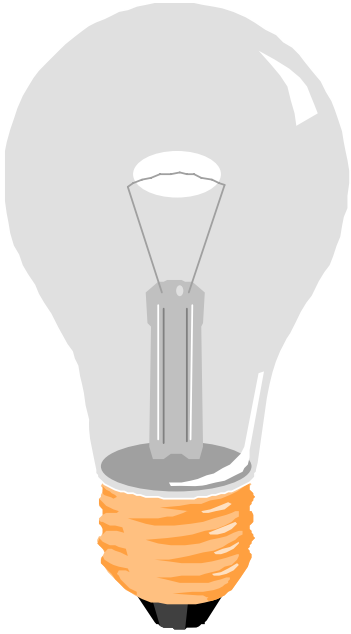
Need for Automated Analysis

- On the previous slide, one might conclude that finer-grained ordering would be sufficient.
  - Explicitly specify relative order of individual advices!
- However, this is …
  - … tedious and error-prone if the number of advices and introductions is not very small
  - …impossible under the assumptions mentioned before
    - ⇨ independent development
    - ⇨ black-box aspects
    - ⇨ unanticipated composition
- Therefore we need automated analysis!

Weaving Interferences

The Problem isn't Aspect-Specific!

- Collaborative software development
  - How to synchronize independent, simultaneous refactorings?
- Configuration management
  - How to merge different branches?
- Pattern sequences
  - How do multiple patterns interact?
  - How to apply them together?

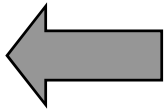


Common baseline

- Program transformations
  - How do transformations interact?

Outline

- From Aspect Interaction to Transformation Interaction
- Conditional Program Transformations (CTs)
- Example with CTs
- Interaction Analysis for CTs
- Related work
- Conclusions



Conditonal Transformations

- Textual Representation -

- Concrete Aspect Interaction Example -

- Interaction Resolution by Topological Sorting -

- Dealing with Cyclic Interaction Graphs -

Example

Concerns

- Thread safety → “Access” transformer
  - Adds getField() and setField() methods for every field
  - Replaces field accesses by accessor method invocations
- Performance tuning → “Counter” transformer
  - Adds field "Field_counter" for every field named "Field"
  - Adds code to increment "Field_counter" before any access to "Field"

Program, Conditions, Transformations, CTs

Alphabet Σ of program elements (choose your own)

e.g. Σ = { class(name), field(name,class), method(name,class),
getfield(field,encl), call(method, encl), increment(field, encl) }

Condition Language

$C \equiv EC \mid EC \wedge C \mid EC \vee C \mid \text{not}(C)$
$EC \equiv \text{true} \mid \text{false} \mid \text{exists(elem)} : \text{elem} \in \Sigma$

Transformation Language

$T \equiv ET \mid ET\ ,\ T$
$ET \equiv \text{add(elem)} \mid \text{delete(elem)} \mid \text{replace}(\text{elem}_1,\text{elem}_2) : \text{elem} \in \Sigma$

Conditional Transformation Language

$CT \equiv C \rightarrow T$ // single CT
$CTS \equiv CT \mid CT \,\&\&\, CTS \mid CT \,||\, CTS$ // CT sequences

Conditional Transformations

Syntax

$CT \equiv C \rightarrow T$

Precondition — Transformation

Example

For brevity we leave out the 'exists' predicate on the next slides

$CT \equiv$ exists(field(F,C)) $\wedge$ exists(class(C)) $\wedge$ not(exists(method(<get>F,C))

$\rightarrow$ add(method(<get>F,C)) , add(getfield(F,<get>F))

```
public class C {

  B b = new B();



  ...

}
```

```
public class C {

  B b = new B();

  B getB() {
     return b;
  }
  ...

}
```

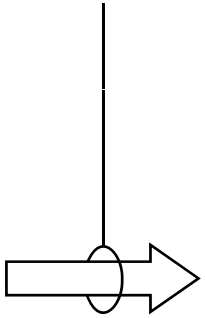
AddGetter: Create Accessor Method

- AddGetter
  - for all fields that have no getter method ...
  - ... add method that returns the field's value

AddGetter ≡ field(F,C) ∧ class(C) ∧ not(method(<get>F,C)

→ add(method(<get>F,C)) , add(getfield(F,<get>F))

```
public class C {

  B b = new B();




  ...

}
```



```
public class C {

  B b = new B();

  B getB() {
     return b;
  }

   ...

}
```

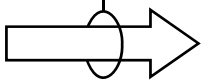
UseGetter: Replace Field Accesses

- UseGetter
  - for all fields that have a getter method and
    for all read accesses to the field outside of its getter method...
  - ... replace the read access by the getter invocation

UseGetter ≡ class(C) ∧ field(F,C) ∧ method(<get>F,C) ∧ getfield(F,M) ∧ M≠<get>F

→ replace(getfield(F,M) , call(<get>F,M))

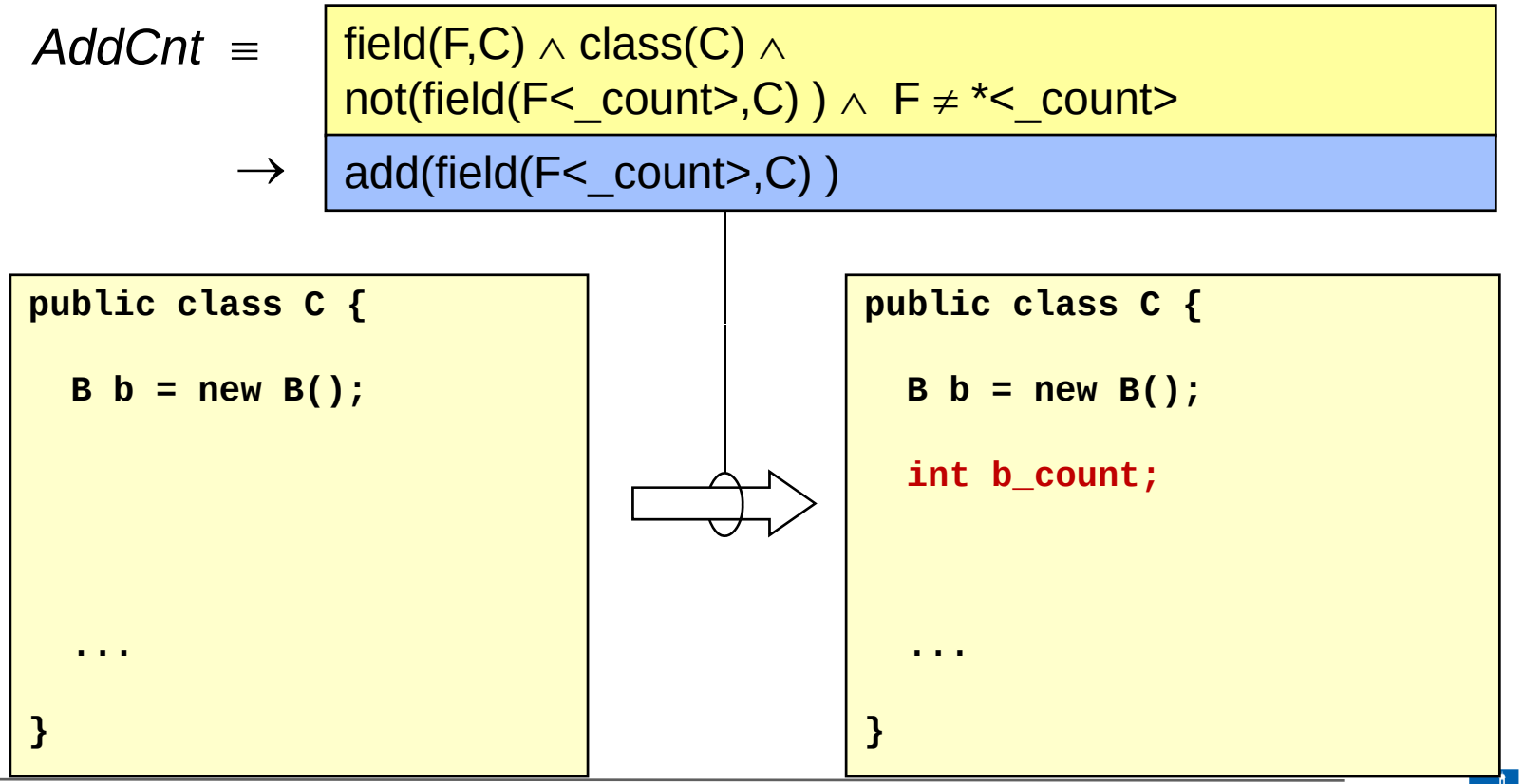
```
public class C {

  B b = new B();

  B getB() {...}

  foo() {
     ...
     b.m();
  }
}
```



```
public class C {

  B b = new B();

  B getB() {...}

  foo() {
     ...
     getB().m();
  }
}
```

AddCnt: Add Counter Field

- AddCnt
  - for all fields F that do not have a counter field and are not themselves counter fields ...
  - ... add a field named F_counter



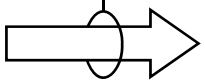
IncCnt: Increment Counter

- IncCnt
  - for all read accesses to a field that has a counter that has no increment operation ...
  - ... add an increment of the counter

IncCnt ≡ getfield(F,M) ∧ field(F,C) ∧ field(F<_count>,C) ∧ not(increment(F<_count>,M))

→ add(increment(F<_count>,M))

```
public class C {

  B b = new B();

  int b_count;

  foo() {

     b.m();
  }
}
```



```
public class C {

  B b = new B();

  int b_count;

  foo() {
     b_count++;
     b.m();
  }
}
```

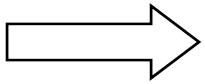
Joint Application

- Independent development of
  - AddCnt, IncCnt and
  - AddGetter, UseGetter
- Joint application to same program

```
read
b_count++;
as shorthand for
setB_count(getB_count()+1);
```

initial programm

```
public class C {
  B b = new B();

  void manipulB() {
    b.m();
  }




}
```



expected result

```
public class C {
  B b = new B();
  int b_count=0;
  void manipulB(){
    getB().m();
  }
  B getB() {
    b_count++;
    return b;
  }
  int getB_count() {
    return b_count;
  }
}
```

Initial Program

AddCnt, IncCnt, AddGetter, UseGetter ← Chosen order of application

```
public class C {
  B b = new B();

  void manipulB(){

    b.m();
  }
}
```

Step 1

AddCnt, IncCnt, AddGetter, UseGetter

```
public class C {
  B b = new B();
  int b_count=0;
  void manipulB(){

    b.m();
  }
}
```

Step 2

AddCnt, IncCnt, AddGetter, UseGetter

```
public class C {
  B b = new B();
  int b_count=0;
  void manipulB(){
    b_count++;
    b.m();
  }
}
```

Step 3

AddCnt, IncCnt, AddGetter, UseGetter

```
public class C {
  B b = new B();
  int b_count=0;
  void manipulB(){
    b_count++;
    b.m();
  }
  B getB(){

    return b;
  }
  int getB_count(){
    return b_count;
  }
}
```

Step 4

AddCnt, IncCnt, AddGetter, UseGetter

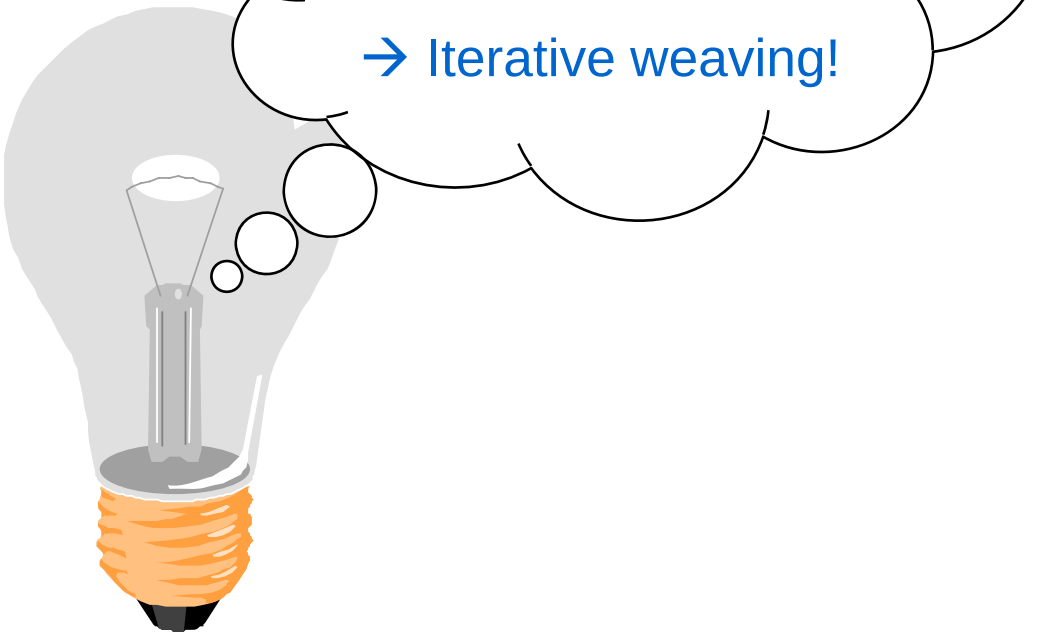
```
public class C {
  B b = new B();
  int b_count=0;
  void manipulB(){
    b_count++;
    getB().m();
  }
  B getB(){
    return b;
  }
  int getB_count(){
    return b_count;
  }
}
```

Ready!? No! IncCnt was not applied to all fields!

Step 4

AddCnt, IncCnt, AddGetter, UseGetter

```
public class C {
  B b = new B();
  int b_count=0;
  void manipulB(){
    b_count++;
    getB().m();
  }
  B getB(){
    return b;
  }
  int getB_count(){
    return b_count;
  }
}
```



Ready!? No! IncCnt was not applied to all fields!

Step 5 (2nd Iteration)

AddCnt, IncCnt, AddGetter, UseGetter

```
public class C {
  B b = new B();
  int b_count=0;
  void manipulB(){
    b_count++;
    getB().m();
  }
  B getB(){
    b_count++;
    return b;
  }
  int getB_count(){
    return b_count;
  }
}
```

ready!

Fixpoint reached!

Further application of any CT does not change the program anymore!

But: Is this the intended result?

Observations: Problem

AddCnt, IncCnt, AddGetter, UseGetter

```
public class C {
  B b = new B();
  int b_count=0;
  void manipulB(){
    b_count++;
    getB().m();
  }
  B getB(){
    b_count++;
    return b;
  }
```

- Order dependent result
  - iteration alone does not help
- Possible orders: 4! = 24

Expected Result

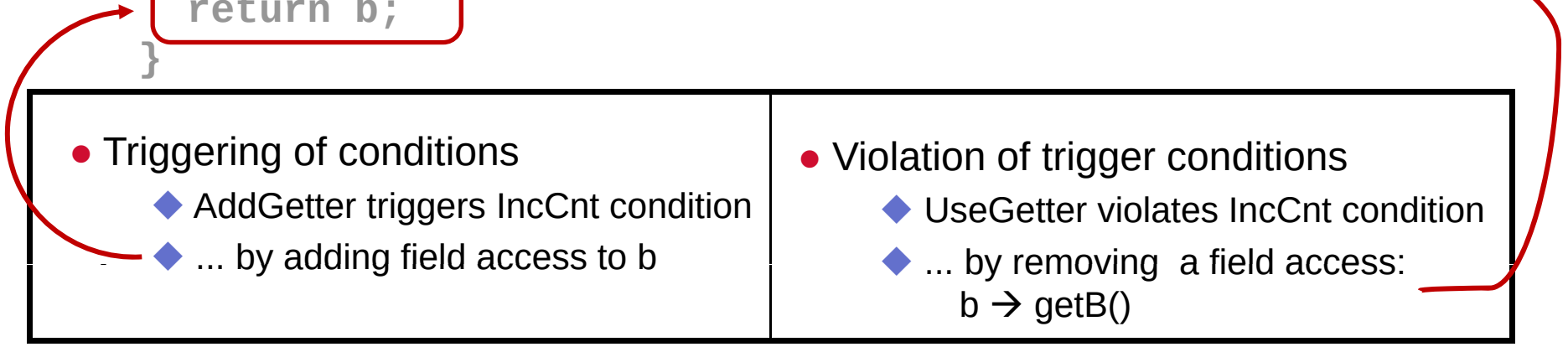
```
public class C {
  B b = new B();
  int b_count=0;
  void manipulB(){

    getB().m();
  }
  B getB() {
    b_count++;
    return b;
  }
```

- Proper order?
  - one or many?
  - how to characterize it?

Observations: Cause of Problem

AddCnt, IncCnt, AddGetter, UseGetter

```
public class C {
  B b = new B();
  int b_count=0;
  void manipulB(){
    b_count++;
    getB().m();
  }
  B getB(){
    b_count++;
    return b;
  }
```



- Triggering of conditions
  - AddGetter triggers IncCnt condition
  - ... by adding field access to b
- Violation of trigger conditions
  - UseGetter violates IncCnt condition
  - ... by removing a field access: b → getB()

Approach

Describe effect of atomic transformation as postcondition

derive effect of transformation sequence as postcondition

derive triggering as postconditions that match preconditons

derive inhibition as postconditions that match the negation of preconditons

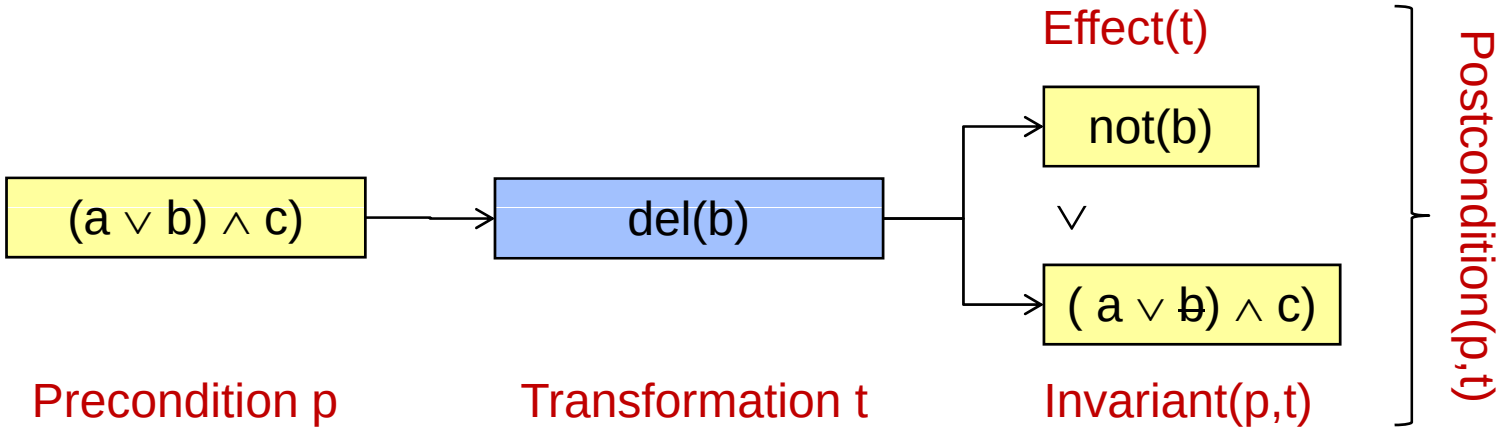
infer proper order from the interaction graph

Approach

- Observation: Transformations influence truth of conditions
  - transformations make conditions true → "triggering"
  - transformations violate conditions → "inhibition“
- Idea
  - Analyse set of CTs (independent of a base program)
  - Find out about their potential triggering and inhibition
  - Use this knowledge to determine a "good" CT execution order
- Three-Step Approach
  - Describe effect of transformation on conditions
    → postconditions
  - Compare postconditions and preconditions to identify triggering and inhibition dependencies
    → dependency graph
  - Analyse the structure of the dependency graph to identify problems or order the CTs
    → global order or error diagnostics

Deriving the Postcondition of a CT

- Preconditon
  - Expresses what must be true before a transformation is performed.
- Transformation
  - Ensures a certain effect
  - Possibly invalidates a part of the precondition
- Effect
  - Expresses what is true after a transformation, regardless of what was true before.
- Invariant
  - The part of the precondition that is still true after the transformation



Deriving the Postcondition of a CT

- Preconditon
  - Expresses what must be true before a transformation is performed.
- Transformation
  - Ensures a certain effect
  - Possibly invalidates a part of the precondition

- Effect
  - Expresses what is true after a transformation, regardless of what was true before.
- Invariant
  - The part of the precondition that is still true after the transformation

- Postcondition
  - Expresses what is true after a transformation, considering the invariant.

Plan for the next slides

- Single transformation *t*
  - **effect**(*t*)
- CT *c*→*t* with single transformation *t*
  - **invariant**(*c*,*t*)
  - **postcondition**(*c*,*t*)
- CT *c*→*t* with transformation sequence *t*
  - **postcondition**(*c*,*t*)

Effect of Single Transformation

- Single Transformation
  - add
  - delete

  } replace(A,B) = add(B) ° delete(A)
- Transformation effect
  - Describes effect of transformation as condition
  - Weakest condition that is guaranteed to hold after a transformation
- Effect of atomic transformation
  - **effect**(add(elem)) ≡ exists(elem)
  - **effect**(delete(elem)) ≡ not(exists(elem))

    ⇨ Example: effect(add(classDefT(Id,Pkg,N,M)) = exists(classDefT(Id,Pkg,N,M))
- Property
  - programm-independent

    Valid for all variable substitions for which the precondition is true on any *current* program.

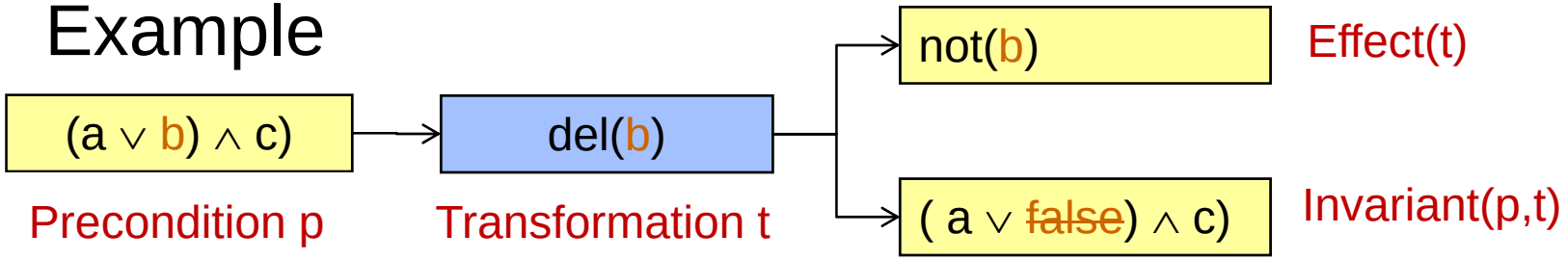
Invariant (Propositional Logic)

- Informal Definition
  - The part of the precondition that is not invalidated by a transformation
- Formal Definition
  - Let c, c_1, c_2 denote conditions and let e be the effect of the single transformation t.

$$\begin{aligned}
&\mathbf{invar}(c_1 \wedge c_2, t) \equiv \mathbf{invar}(c_1, t) \wedge \mathbf{invar}(c_2, t)\\
&\mathbf{invar}(c_1 \vee c_2, t) \equiv \mathbf{invar}(c_1, t) \vee \mathbf{invar}(c_2, t)\\
&\mathbf{invar}(c, t) \equiv c \quad : \neg c \text{ and } e \text{ different}\\
&\mathbf{invar}(c, t) \equiv \text{false} \quad : \neg c \text{ and } e \text{ equal}
\end{aligned}$$

The effect of t invalidates a previously true subcondition

- Example



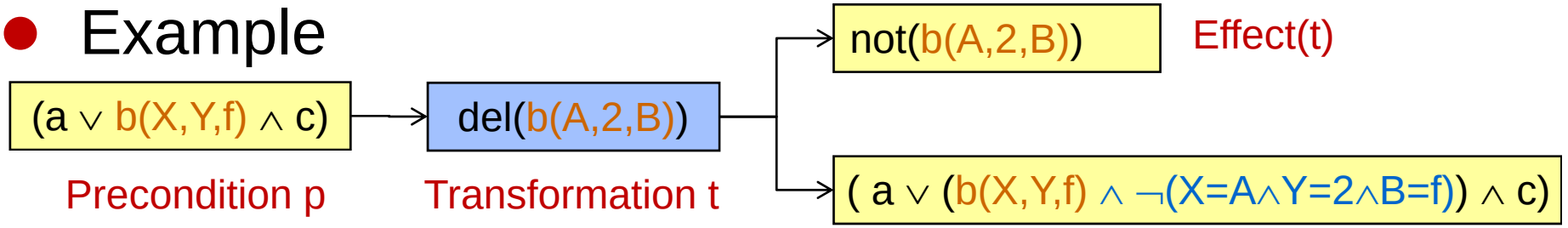
Invariant (First-order Logic)

- Informal Definition
  - The part of the precondition that is not invalidated by a transformation
- Formal Definition
  - Let c, c_1, c_2 denote conditions and let e be the effect of the single transformation t. Let θ be the most general unifier of $\neg c$ and e.

$\mathbf{invar}(c_1 \wedge c_2, t)$	$\equiv \mathbf{invar}(c_1, t) \wedge \mathbf{invar}(c_2, t)$	
$\mathbf{invar}(c_1 \vee c_2, t)$	$\equiv \mathbf{invar}(c_1, t) \vee \mathbf{invar}(c_2, t)$	
$\mathbf{invar}(c, t)$	$\equiv c$	: $\neg c$ and e not unifiable
$\mathbf{invar}(c, t)$	$\equiv c \wedge \neg\mathbf{eq}(\theta)$	: unifiable with unifier θ

Only the substitutions subsumed by θ are invalidated, not the entire literal

- Example



Unifiability Constraints (for First-Order Invariant)

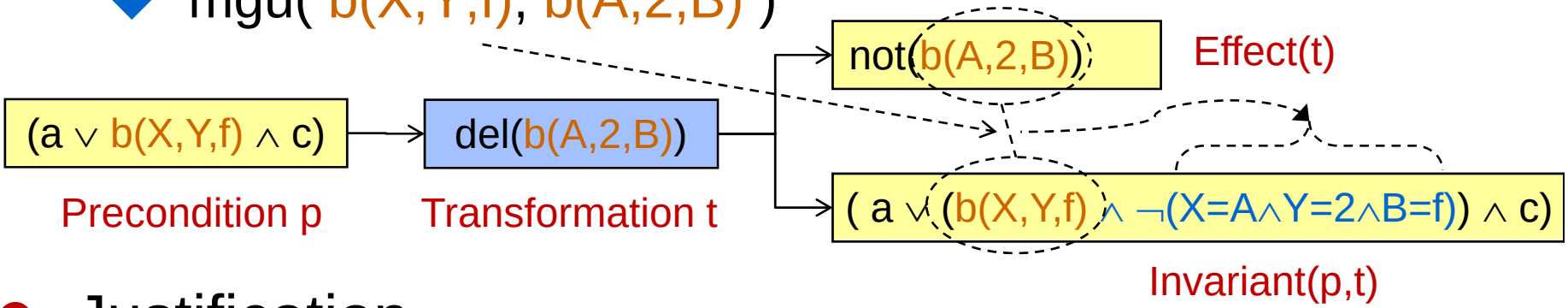
- Definition of **eq**(θ)
  - Let θ be any substitution, that is, any (possibly empty) set of bindings: θ = {$var_i \leftarrow term_i$ | i=1..n}
  - An empty substitution set corresponds to the condition „true“.
  - A non-empty substitution corresponds to a conjunction of unifiability constraints $var_i = term_i$

eq(θ) ≡ true	: if θ = { }
eq(θ) ≡ $v_1 = t_n\theta \wedge \ldots \wedge v_n = t_n\theta$	: if θ = {$v_i \leftarrow t_i$ | i=1..n}

- Example 1
  - θ = {X ← A, Y ← 2, B ← f} = mgu(b(X,Y,f), b(A,2,B))
  - eq(θ) ≡ X=A ∧ Y=2 ∧ B=f
- Example 2
  - θ = { } = mgu(meth(F,C), meth(F,C))
  - eq(θ) ≡ true

Why take the mgu?

- On the previous slide we used the most general unifier (mgu) of a query literal and the effect
  - mgu(b(X,Y,f), b(A,2,B))



- Justification
  - The mgu expresses the conditions under which a true substitution for the precondition literal b(X,Y,f) would be invalidated by the effect not(b(A,2,B)).
  - Adding these conditions to the invariant produces a single expression that describes exactly what is true after the transformation

Postcondition of a Simple CT

- Simple CT *cond* → *trans*
  - Contains only a single elementary transformation *trans* (see prev. slides)
- CT Postcondition
  - Is the weakest condition that is guaranteed to be true after the CT.
  - The transformation effect is part of the postcondition
    - ⇨ **effect**(*trans*) – as defined previously
  - The invariant is part of the postcondition
    - ⇨ **invar**(*cond*,*trans*) – as defined previously
  - **post**(*cond*, *trans*) = **effect**(*trans*) ∨ **invar**(*cond*, **effect**(*trans*))

We ignore anything that is true but is not expressed in the precondition *cond*

Disjunction because the effect holds even if the invariant is „false“

Postcondition of an Arbitrary CT

- Arbitrary CT *cond* → *t1…tn*
  - Contains arbitrary transformation sequence *t1…tn*
- CT Postcondition
  - Is the weakest condition that is guaranteed to be true after the CT.
  - Later transformations possibly invalidate preconditions and effects of previous ones
  - Therefore the postcondition of a transformation sequence is the postcondition of the last transformation, tn, applied to the postconditon of the transformation Sequence *t1…tn-1*
  - $\mathbf{post}(\mathit{cond}, t_1 \ldots t_n) = \mathbf{effect}(t_n) \vee \mathbf{invar}(\mathbf{post}(\mathit{cond}, t_1 \ldots t_{n-1}), \mathbf{effect}(t_n))$

Example: Derive Postcondition of CT

Summary of Formulas

effect(add(elem))	≡ exists(elem)	
effect(delete(elem))	≡ not(exists(elem))	
invar($c_1 \wedge c_2$, t)	≡ **invar**(c_1, t) ∨ **invar**(c_2, t)	
invar($c_1 \vee c_2$, t)	≡ **invar**(c_1, t) ∨ **invar**(c_2, t)	
invar(c, t)	≡ c	: $\neg c$ and e not unifiable
invar(c, t)	≡ $c \wedge \neg$**eq**(θ)	: unifiable with unifier θ
post(c, t_1)	≡ **effect**(t_1) ∨ **invar**(c, **effect**(t_1))	
post(c, $t_1 \ldots t_n$)	≡ **effect**(t_n) ∨ **invar**(**post**(c, $t_1 \ldots t_{n-1}$), **effect**(t_n))	

Input: The AddGetter CT

exists(field(F,C)) ∧ exists(class(C)) ∧ not(exists(method(<get>F,C)))

add(method(<get>F,C)) , add(getfield(F,<get>F))

Output: Its Postcondition

exists(field(F,C)) ∨ exists(class(C)) ∨ ~~not(exists(method(<get>F,C)))~~

exists(method(<get>F,C)) ∨ exists(getfield(F,<get>F))

Example: Derive Effect of CT

Summary of Formulas

effect(add(elem))	≡ exists(elem)	
effect(delete(elem))	≡ not(exists(elem))	
invar($c_1 \wedge c_2$, t)	≡ **invar**(c_1, t) ∧ **invar**(c_2, t)	
invar($c_1 \vee c_2$, t)	≡ **invar**(c_1, t) ∨ **invar**(c_2, t)	
invar(c, t)	≡ c	: $\neg c$ and e not unifiable
invar(c, t)	≡ $c \wedge \neg$**eq**(θ)	: unifiable with unifier θ
post(c, t_1)	≡ **effect**(t_1) ∨ **invar**(c, **effect**(t_1))	
post(c, $t_1 \ldots t_n$)	≡ **effect**(t_n) ∨ **invar**(**post**(c, $t_1 \ldots t_{n-1}$), **effect**(t_n))	
effect(c, $t_1 \ldots t_n$)	≡ **effect**(t_n) ∨ **invar**(**post**(*true*, $t_1 \ldots t_{n-1}$), **effect**(t_n))	

Input: The AddGetter CT

exists(field(F,C)) ∧ exists(class(C)) ∧ not(exists(method(<get>F,C)))

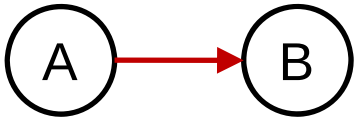
add(method(<get>F,C)) , add(getfield(F,<get>F))

Output: Its Effect

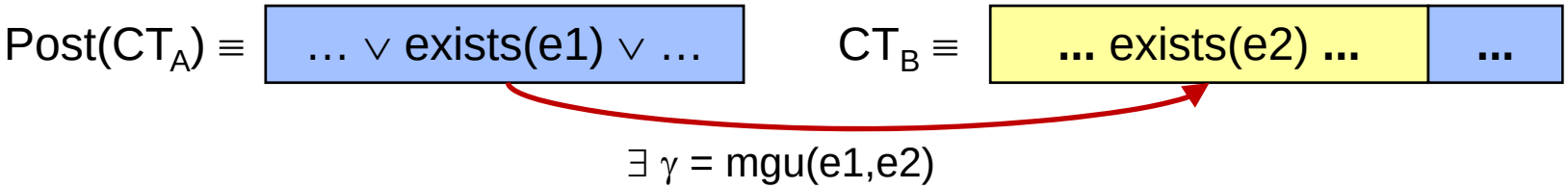
exists(method(<get>F,C)) ∨ exists(getfield(F,<get>F))

Dependencies

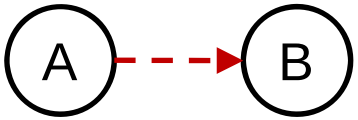
- A possibly triggers B



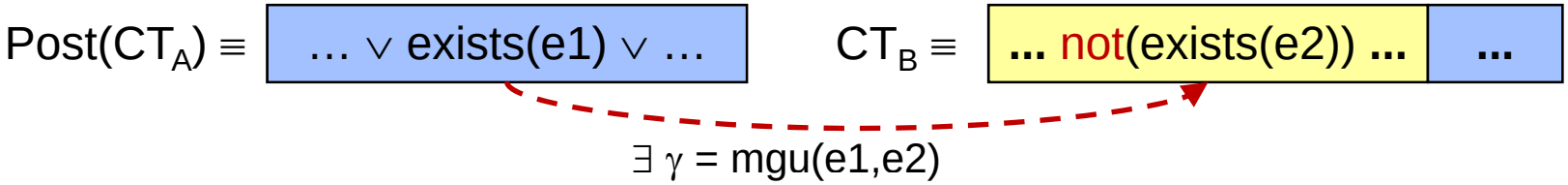
  - A‘s postcondition makes a literal in B's preconditon true



- A possibly inibits B



  - A‘s postcondition makes a literal in B's preconditon false



- Dependency computation is independent of a program
- Dependencies describe potential interactions

Dependency Analysis of Sample CTs

AddGetter

field(F,C) ∧ class(C) ∧ not(method(<get>F,C))
add(method(<get>F,C)) , add(getfield(F,<get>F))

UseGetter

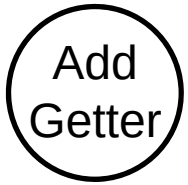
getfield(F,M) ∧ field(F,C) ∧ class(C) ∧ method(<get>F,C) ∧ M≠<get>F
replace(getfield(F,M) , call(<get>F,M))

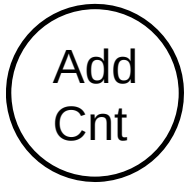
AddCnter

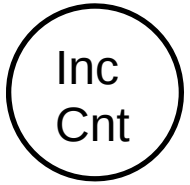
field(F,C) ∧ class(C) ∧ F ≠ *<_count> ∧ not(field(F<_count>,C))
add(field(F<_count>,C))

IncCnter

getfield(F,M) ∧ field(F,C) ∧ field(F<_count>,C) ∧ not(increment(F<_count>,M))
add(increment(F<_count>,M))







Dependency Analysis (1)

AddGetter

field(F,C) ∧ class(C) ∧ not(method(<get>F,C))
add(method(<get>F,C)) , add(getfield(F,<get>F))

UseGetter

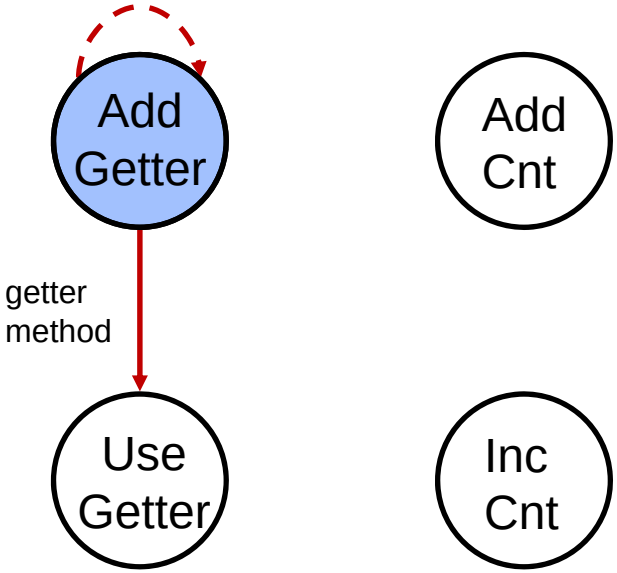
getfield(F,M) ∧ field(F,C) ∧ class(C) ∧ method(<get>F,C) ∧ M≠<get>F
replace(getfield(F,M) , call(<get>F,M))

AddCnter

field(F,C) ∧ class(C) ∧ F ≠ *<_count> ∧ not(field(F<_count>,C))
add(field(F<_count>,C))

IncCnter

getfield(F,M) ∧ field(F,C) ∧ field(F<_count>,C) ∧ not(increment(F<_count>,M))
add(increment(F<_count>,M))



Dependency Analysis (2)

AddGetter

field(F,C) ∧ class(C) ∧ not(method(<get>F,C))
add(method(<get>F,C)) , add(getfield(F,<get>F))

UseGetter

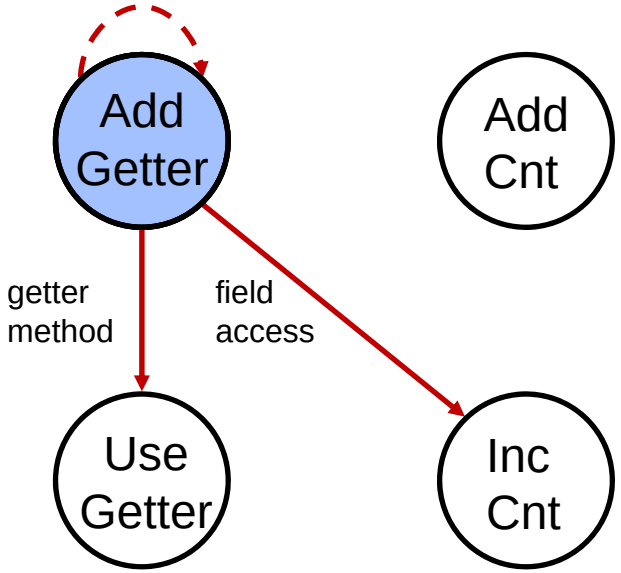
getfield(F,M) ∧ field(F,C) ∧ class(C) ∧ method(<get>F,C) ∧ M≠<get>F
replace(getfield(F,M) , call(<get>F,M))

AddCnter

field(F,C) ∧ class(C) ∧ F ≠ *<_count> ∧ not(field(F<_count>,C))
add(field(F<_count>,C))

IncCnter

getfield(F,M) ∧ field(F,C) ∧ field(F<_count>,C) ∧ not(increment(F<_count>,M))
add(increment(F<_count>,M))



Dependency Analysis (3)

AddGetter

field(F,C) ∧ class(C) ∧ not(method(<get>F,C))
add(method(<get>F,C)) , add(getfield(F,<get>F))

UseGetter

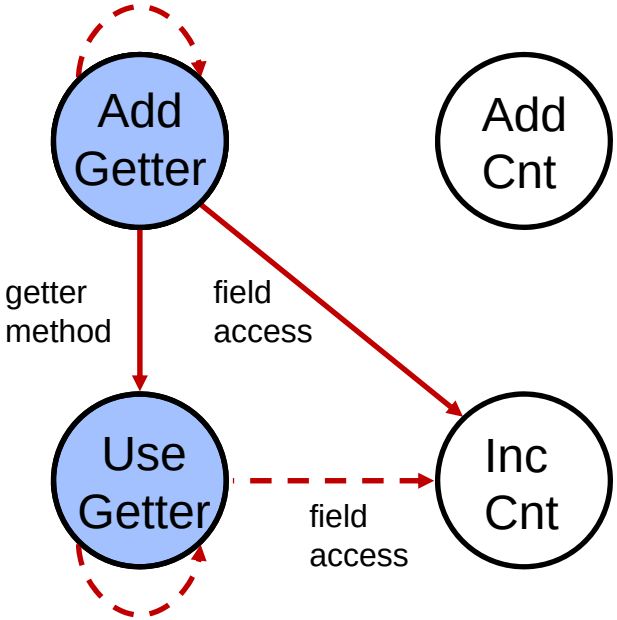
getfield(F,M) ∧ field(F,C) ∧ class(C) ∧ method(<get>F,C) ∧ M≠<get>F
replace(getfield(F,M) , call(<get>F,M))

AddCnter

field(F,C) ∧ class(C) ∧ F ≠ *<_count> ∧ not(field(F<_count>,C))
add(field(F<_count>,C))

IncCnter

getfield(F,M) ∧ field(F,C) ∧ field(F<_count>,C) ∧ not(increment(F<_count>,M))
add(increment(F<_count>,M))



Dependency Analysis (4)

AddGetter

field(F,C) ∧ class(C) ∧ not(method(<get>F,C))
add(method(<get>F,C)) , add(getfield(F,<get>F))

UseGetter

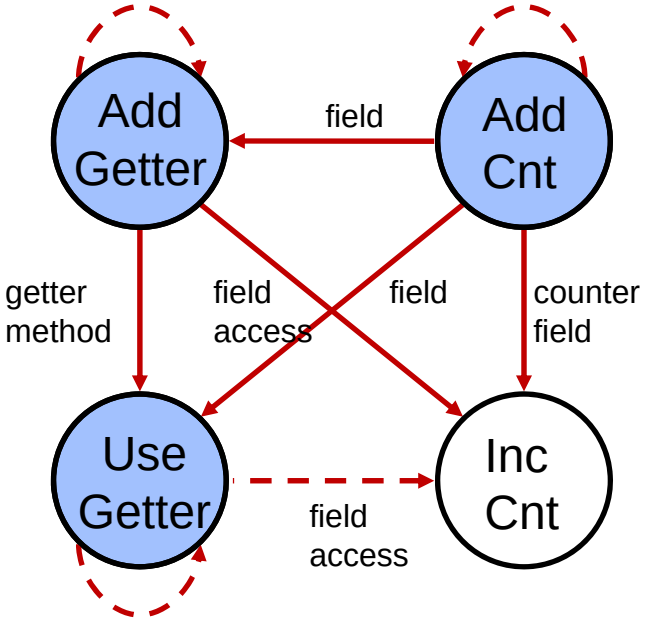
getfield(F,M) ∧ field(F,C) ∧ class(C) ∧ method(<get>F,C) ∧ M≠<get>F
replace(getfield(F,M) , call(<get>F,M))

AddCnter

field(F,C) ∧ class(C) ∧ F ≠ *<_count> ∧ not(field(F<_count>,C))
add(field(F<_count>,C))

IncCnter

getfield(F,M) ∧ field(F,C) ∧ field(F<_count>,C) ∧ not(increment(F<_count>,M))
add(increment(F<_count>,M))



Dependency Analysis (5)

AddGetter

field(F,C) ∧ class(C) ∧ not(method(<get>F,C))
add(method(<get>F,C)) , add(getfield(F,<get>F))

UseGetter

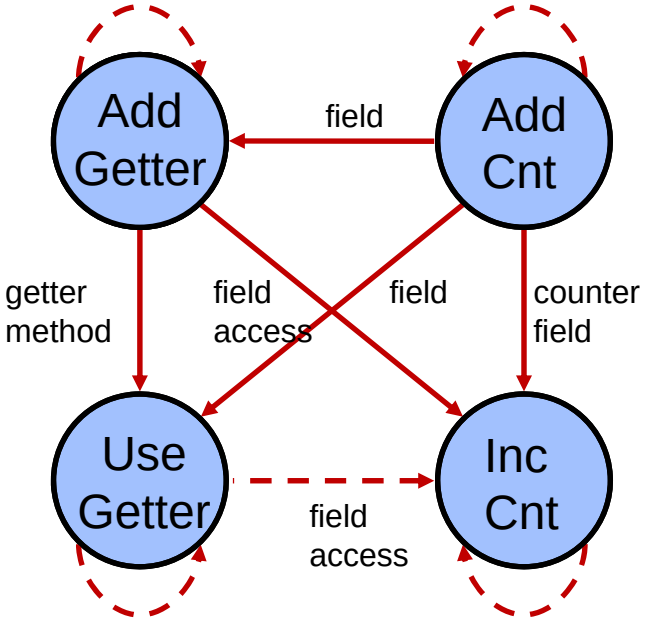
getfield(F,M) ∧ field(F,C) ∧ class(C) ∧ method(<get>F,C) ∧ M≠<get>F
replace(getfield(F,M) , call(<get>F,M))

AddCnter

field(F,C) ∧ class(C) ∧ F ≠ *<_count> ∧ not(field(F<_count>,C))
add(field(F<_count>,C))

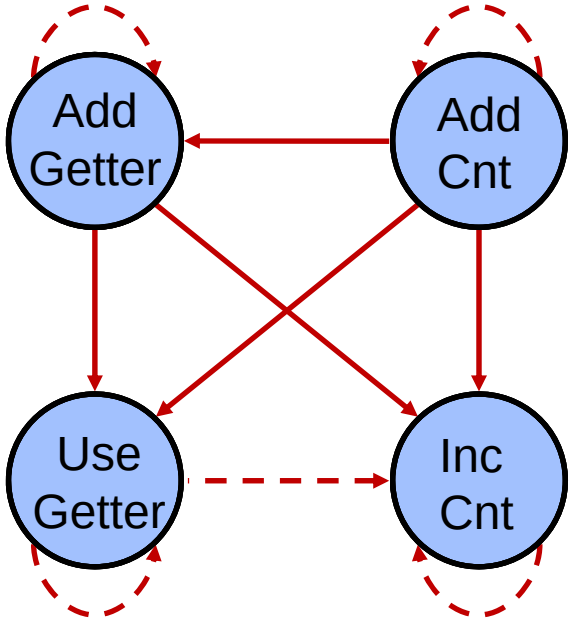
IncCnter

getfield(F,M) ∧ field(F,C) ∧ field(F<_count>,C) ∧ not(increment(F<_count>,M))
add(increment(F<_count>,M))



Result: Dependency Graph

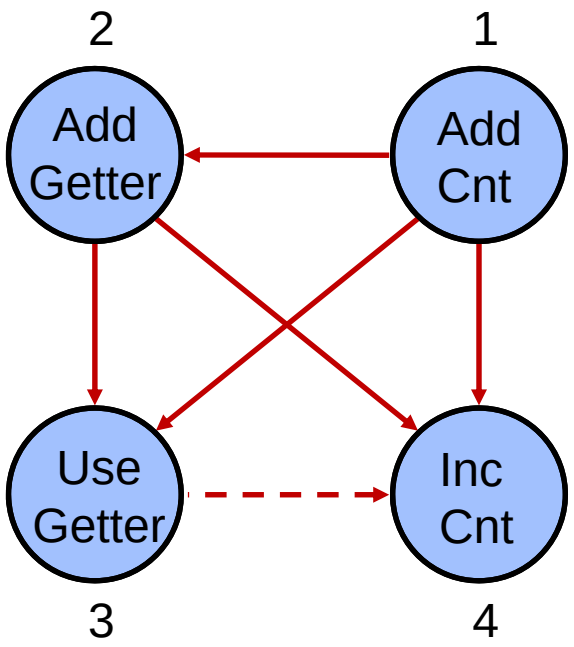
- What does it tell us?
- Self-Inhibition
  - Necessary, to prevent infinite application of the same CT to the same data
  - Issue a warning, for each CT without self-inhibition!
- Automatically determinine the "right" order
  - Use graph without self-inhibition arcs (next slide)



Automatic Ordering

- Topological sorting
  - Produces order that respects all dependencies:

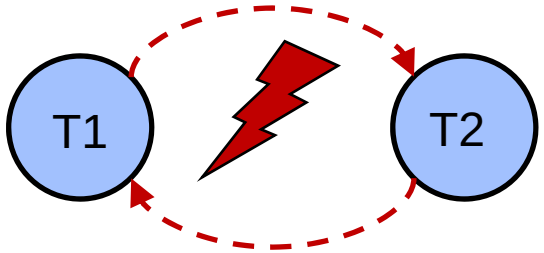
    AddCnt, AddGetter, UseGetter , IncCnt



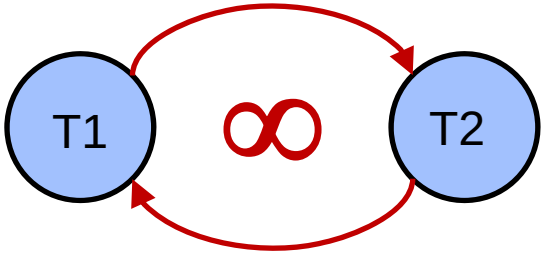
- This is the "right" order
  - Each CT is executed after all CTs that can influence it
- Consequences
  - No missed triggers: no iteration necessary
  - No missed inhibition: no undo necessary
  - Complete, correct and efficient weaving!

Cyclic Dependencies

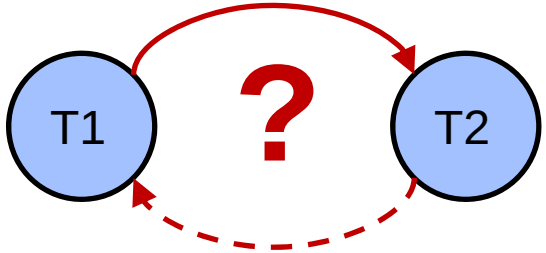
- Long Inhibition Cycle: contains inhibitions only (more than one)
  - CTs prevent each other
    - ⇒ joint application not possible
    - ⇒ potential conflict
    - ⇒ error message



- Triggering cycle: contains triggering dependencies only
  - Transformations trigger each other
    - ⇒ iteration could lead to a fixpoint
    - ⇒ iterative application required
    - ⇒ ask whether to iterate

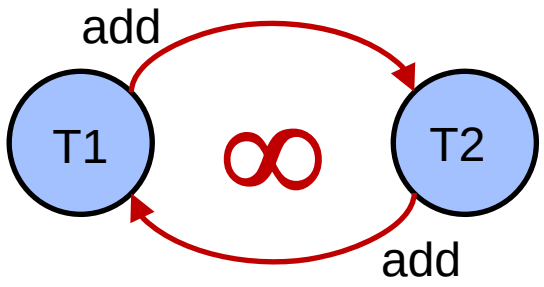


- Mixed cycle: contains triggering and inhibition dependencies
  - no automatic resolution possible
  - ask the user

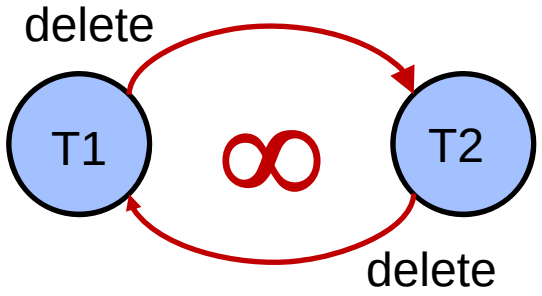


Triggering Cycles: Classification based on Trigger Action

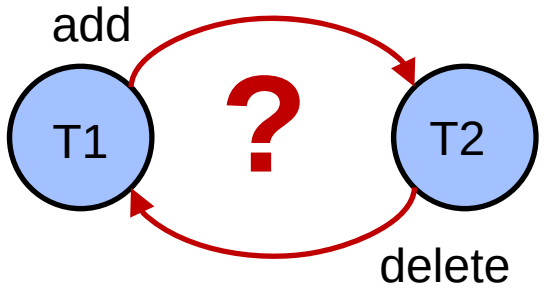
- Triggered by addition
  - implies monotonic "growth" of program
  - fixpoint guaranteed – **if** iteration terminates



- Triggered by deletion
  - implies monotonic "shrinking" of program
  - fixpoint guaranteed
  - **termination guaranteed**



- Mixed triggering
  - non-monotonic
  - no guarantee for a fixpoint



Summary

- Dependency analysis
  - Based on program-independent description of "transformation effect"
  - Generates graph of inhibition and triggering dependencies
- Interpretation of dependency graph
  - identifies incomplete preconditions / pointcuts
    - ⇨ missing self-inhibitions
  - identifies conflicts
    - ⇨ negative cycles
  - identifies required iteration
    - ⇨ monotonic positive cycles
  - identifies cases that require user input
    - ⇨ mixed cycles and mixed triggers in positive cycles
  - if there are no cycles, generates order that respects all dependencies
    - ⇨ topological sorting of dependency graph

Benefits

- Joint application of independently developed CT‘s possible
  - Completeness and correctness

➔ It works!

- No useless iteration
  - No missed triggers

➔ It is efficient

- No undo necessary
  - No missed inhibition

➔ Simple infrastructure sufficient

References: CTs and their Analysis

- A Logic Foundation for Conditional Program Transformations
  - Günter Kniesel
    Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn. ISSN 0944-8535. January 2006.
- Static Composition of Refactorings
  - Günter Kniesel, Helge Koch
    In: Ralf Lämmel (Ed.): Science of Computer Programming 52 (2004), special issue on "Program Transformation", p. 9-51, Elsevier Science, 2004.
    Digital Object Identifier: doi:10.1016/j.scico.2004.03.002
    If the DOI or Wiley website is unreachable, here is a local copy of a preprint.

Next Week

- An Analysis of the Correctness and Completeness of Aspect Weaving
  - Günter Kniesel, Uwe Bardey
    in *Proceedings of Thirteenth Working Conference on Reverse Engineering (WCRE 2006),* Oct. 23-27, Benevento, Italy, ISBN 0-7695-2719-1, p. 324-333, IEEE 2006.
- Transformation dependency analysis - A comparison of two approaches
  - Tom Mens, Günter Kniesel, Olga Runge
    Proceedings of Langages et Modèles à Objets (LMO2006), March 22-24, 2006, Nîmes, France, special issue of L'Objet, Hermes Science Publishing, London, 2006.