universitätbonn ROOTS

AOSD: Rückblick

Kap. 3 Aspekt-Generizität / LogicAJ

Kap. 6. AO-Anwendungen / Design Patterns

Kap 10. Semantik von Aspekten

Kap 11. Aspekt Interferenzen

Kap. 3 Aspektgenerizität / LogicAJ

Generic Aspects
– Enhanced Expressiveness and Reusability for AOP –

- Genericity
  - A language is said to support genericity if its language concepts can be parameterized by variables that represent language concepts
- Genericity for Aspects
  - Adaptation of type genericity from the base language
    - ⇨ AspectJ 5
    - ⇨ see Chapter on AspectJ5
  - External parameterization and aspect generation
    - ⇨ Framed Aspects
  - Pointcuts determine values for generic elements
    - ⇨ LogicAJ

Framed Aspects

- Basic Idea
  - Use Frames to adapt aspect code, not base code!
  - Preprocessor: No change of the Aspect Language
- Limitations of Framed Aspects (FA): Assumption of statically enumerable variations
  - Strongly restricts applicability of FA
  - Makes FA highly error-prone
  - Makes FA tedious
  - Makes FA hard to evolve

LogicAJ: Uniform Aspect Genericity

- Example: Aspects for contract enforcement
  - Hash-Code Contract in AspectJ?

```
declare error:
    execution(* *.equals(Object))
    && !method(* ClassOfEqualsMethod.hashCode()) :
       "Contract violation!";
```

  - Generic Hash-Code Contract

```
declare error:
    method(* ?ClassOfEqualsMethod.equals(Object)) &&
    !method(* ?ClassOfEqualsMethod.hashCode()) :
    "Contract violation!";
```

Metavariables get consistent values from matches of pointcuts (including „predicate pointcuts“)

„Predicate pointcuts“ match static structure regardless of any join point

LogicAJ: Uniform Aspect Genericity

- Meta-Variables
  - Range over
    - ⇨ Types
    - ⇨ Identifiers, Modifiers
    - ⇨ Method Bodies
    - ⇨ Parameters
    - ⇨ Literals (Strings, Integers)
    - ⇨ Lists of the previous types
  - Meta-variables may appear wherever the above syntactic elements may be used in AspectJ
    - ⇨ In pointcuts, introductions and advice code!
  - Meta-variables in LogicAJ are logic meta-variables
    - ⇨ binding instead of assignment
  - Logic MVs / Logic list MVs

LogicAJ: Uniform Aspect Genericity

- Unification
  - Equality test that makes its parameters equal, if possible.
  - Can be performed implicitly by pointcut matching or explicitly via the “equals(arg1, arg2)” predicate
  - Explicit Unification: “equals(arg1, arg2)”
  - Implicit Unification: Pointcut Evaluation

Generic Pointcut Semantics (1)

- A binding is an association of a logic variable to a constant, a program element or a logic variable
  - ?X ←1
  - ?type ← Class1
  - ?Y ← ?X
- A substitution is a set of bindings for different logic variables
  - {?X ← 1, ?Y ← ?X}
- A ground substitution is a substitution that binds all variables directly or indirectly to a constant or program element
  - {?X ← 1, ?Y ← ?X} is a ground substitution for an expression that only contains the variables ?X and ?Y

Generic Pointcut Semantics (2)

- A substitution for a pointcut is a set of bindings for each of its logic variables, including the implicit ?jp variable
  - {?jp ← o.method1(),
    ?type ← Class,
    ?m ← method1}
- The semantics of a generic pointcut GPC with respect to a program P is the set of all ground substitutions for GPC that make GPC true for P

```
pointcut GPC :
    call(* *.?m) &&
    method(* ?type.?m())
```

Program P

{ {?jp ← o.m1(), ?type ← Class1, ?m ← m1} ,
{?jp ← this.m2(), ?type ← Class1, ?m ← m1} ,
{?jp ← this.m2(), ?type ← Class2, ?m ← m2}
}

Predicate Pointcuts

Primitve Pointcuts

- bind join point (shadow) to implicit logic variable "?jp"
- bind explicit logic variables to parts of the join point shadow's context

call(* *.?m)

join point shadows and their context

Predicate Pointcuts

- do not bind join point
- bind explicit logic variables to any static program elements and constants

method(* ?type.?m())

arbitrary static program elements

{{ *?jp* = o.method1(), ?m = method1},
{ *?jp* = this.method1(), ?m = method1},
{ *?jp* = this.method2(), ?m = method2}
}

{ { ?type = Class1, ?m = method1},
{ ?type = Class2, ?m = method2} ,
{ ?type = Class3, ?m = method3}
}

Summary "Generic Pointcuts"

- Generic pointcuts replace wildcards by named logic meta-variables
- The semantics of a generic pointcut is the set of all ground substitutions that make it true for a program
- Next: Generic pointcuts are the "heart and soul" of generic advice and generic introductions

Generic Advice

- Problem: Context Dependent Effects
  - Scenario
    - ⇨ New subclass should be used consistently instead of its superclass
  - Challenge
    - ⇨ Replace superclass constructor calls by calls of subclass constructors: `new Super(args)` → `new Sub(args)`
    - ⇨ In Objective C this is built-in: „Class posing“ does just that.
    - ⇨ In other language it is tedious and highly crosscutting
  - Can we solve this generically with aspects?
    - ⇨ Many constructors
    - ⇨ Different signatures
    - ⇨ Constructor names depend on the replaced classes

Generic Advice Syntax and Semantics

- Generic Advice contains meta-variables in its pointcut, parameter list and body.
  - The parameter list includes the implicit "join point" meta-variable
- The body substitutions are the set of successful substitutions for the pointcut projected onto the meta-variables in the parameter list
  - The projection of { {?jp←o.m1(), ?type←Class1, ?m←m1},
    {?jp←o.m1(), ?type←Class1, ?m←m2} }
    onto ?jp, ?type is { {?jp←o.m1(), ?type←Class1} }
  - Note that projection can eliminate substitutions!
- The advice body is executed for each body substitution

Mock-Objects – Generic Solution

Select all constructor calls and their arguments

Aspect

```
aspect MockAspect {

   Object around useMock(?mock, ??args) :
      call(?class.new(..)) && args(??args)
      concat(?class,"Mock",?Mock) &&
      type(?Mock) {
   return new ?Mock(??args);
   }
}
```

Base program excerpt

```
u = new User("John");
um = new UserManager(u);
// ...
um = new UserManager();
// ...
other = new Other("demo")
// ...
```

Successful Matches

?class	??args	?mock
User	["John"]	--- yet unbound ---
UserManager	[u]	--- yet unbound ---
UserManager	[]	--- yet unbound ---
Other	[„demo“]	--- yet unbound ---

Generic Introductions

- Context-Dependent Inter-Type-Declarations?
  - Problem in AspectJ
    - ⇨ Inter-Type-Declarations have no pointcuts.
    - ⇨ Introducing meta-variables in advice bodies only (as in "Framed Aspects") only allows anticipated, static enumeration of values.
  - LogicAJ Approach
    - ⇨ Inter-Type-Declarations have pointcuts!
    - ⇨ Context-dependent behavior for all aspect language elements (advice, introductions, declarations)
    - ⇨ Uniform language design!
- Semantics of Introductions
  - Same as for advice
    - ⇨ All successful substitutions "produced" by the pointcut for the meta-variable parameters are applied to the introduction body.
    - ⇨ A generic introductions whose pointcut yields N sucessful substitutions triggers N different concrete introductions.

Summary: Uniform Genericity

- Use of pointcuts in all aspect language elements
  - introductions, advice, declarations
- Use of meta-variables in all aspect language elements
  - pointcut, introBody, adviceBody, params, ex, interface

➔ Context-dependent aspect effects

- Meta-variables are bound in pointcuts providing values for the body of introductions / advice / declarations

The Power of Generic Advice – Context-Dependent Aspect Effects

Aspect with Generic Advice

```
aspect MockAspect {

   Object around useMock(?mock, ??args) :
      call(?class.new(..)) && args(??args)
      concat(?class,"Mock",?Mock) &&
      type(?Mock) {
   return new ?Mock(??args);
   }
}
```

- ➔ Metavariables bound in the pointcut influence the advice
- ➔ The advice has a different effect depending on the join point context

Base program excerpt

```
u = new User("John");
um = new UserManager(u);
// ...
um = new UserManager();
// ...
other = new Other("demo")
// ...
```

Run-time actions

```
new UserMock("John")
new UserManagerMock(u)
new UserManagerMock()
```

Context-dependent Aspect Effects

Objects

- Static binding
  - **static variable type** determines behaviour
  - fixed, rigid behaviour
- Dynamic binding
  - **dynamic receiver type** determines behaviour
  - context-dependent message behaviour

Aspects

- Homogeneous Aspects
  - executed **advice** determines behaviour
  - fixed, rigid behaviour
- Heterogeneous Aspects
  - **join point context** and **advice** determine behaviour
  - context-dependent aspect behaviour

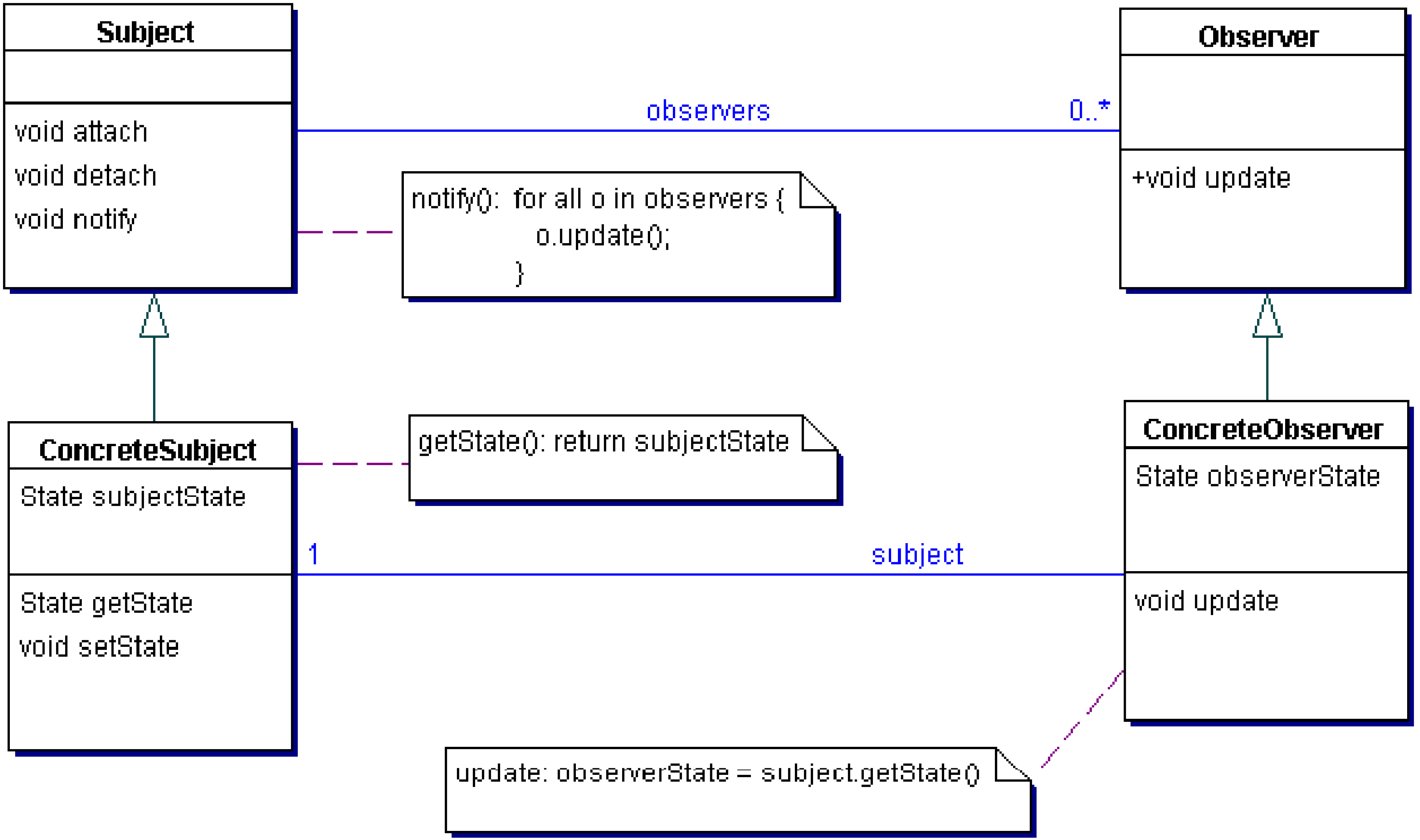
Comparison of LoogicAJ and Frames

Uniformly Generic Aspects	Framed Aspects
Language integrated genericity	External macro system
One programming paradigm	Two programming paradigms
Parameters enumerated in concrete aspects	Parameters enumerated in external representation
Parameters specified by program analysis (pointcut evaluation)	No analysis capability
Context-dependent variation	No context notion
Open ended variations	Fixed enumeration of variations
→ more expressive	→ externally configurable

Approaches could be combined…

Kap. 6 AO-Anwendungen / Design Patterns

The Observer Pattern



Abstract Observer Protocol aspect, Jan Hannemann 2002

```
public abstract aspect ObserverProtocol {

  protected interface Subject  { }
  protected interface Observer { }

  abstract protected pointcut
                subjectChange(Subject s);

  abstract protected void
     updateObserver(Subject s, Observer o);

  after(Subject s): subjectChange(s) {
    Iterator iter = getObservers(s).iterator();
    while ( iter.hasNext() ) {
      updateObserver(s,((Observer)iter.next()));}
...
}
```

Roles

Conceptual OPs

Observer update

Update logic

Observer Pattern in AspectJ

- A concrete observer aspect
  - introduces the marker interfaces to the classes that should play the corresponding roles
  - defines the `updateObserver()` method
  - defines the `subjectChange()` poincut

Observer Pattern Summery

- Improved properties of implementation in AspectJ
  - Locality
    - ⇨ All the code that implements the pattern is in the abstract and concrete aspects
    - ⇨ No coupling between the participant classes exists
  - Reusability
    - ⇨ Core pattern code (ObserverProtocol) is abstracted and reusable
    - ⇨ For each pattern instance, only define one concrete aspect
  - Composition transparency
    - ⇨ Even if a class takes part in multiple observing relationships, codes of the class and pattern are not confused
  - (Un)pluggability
    - ⇨ Participants need not be aware of their role
    - ⇨ It is easy to switch between using a pattern or not using it

Result Overview, Jan Hannemann 2002

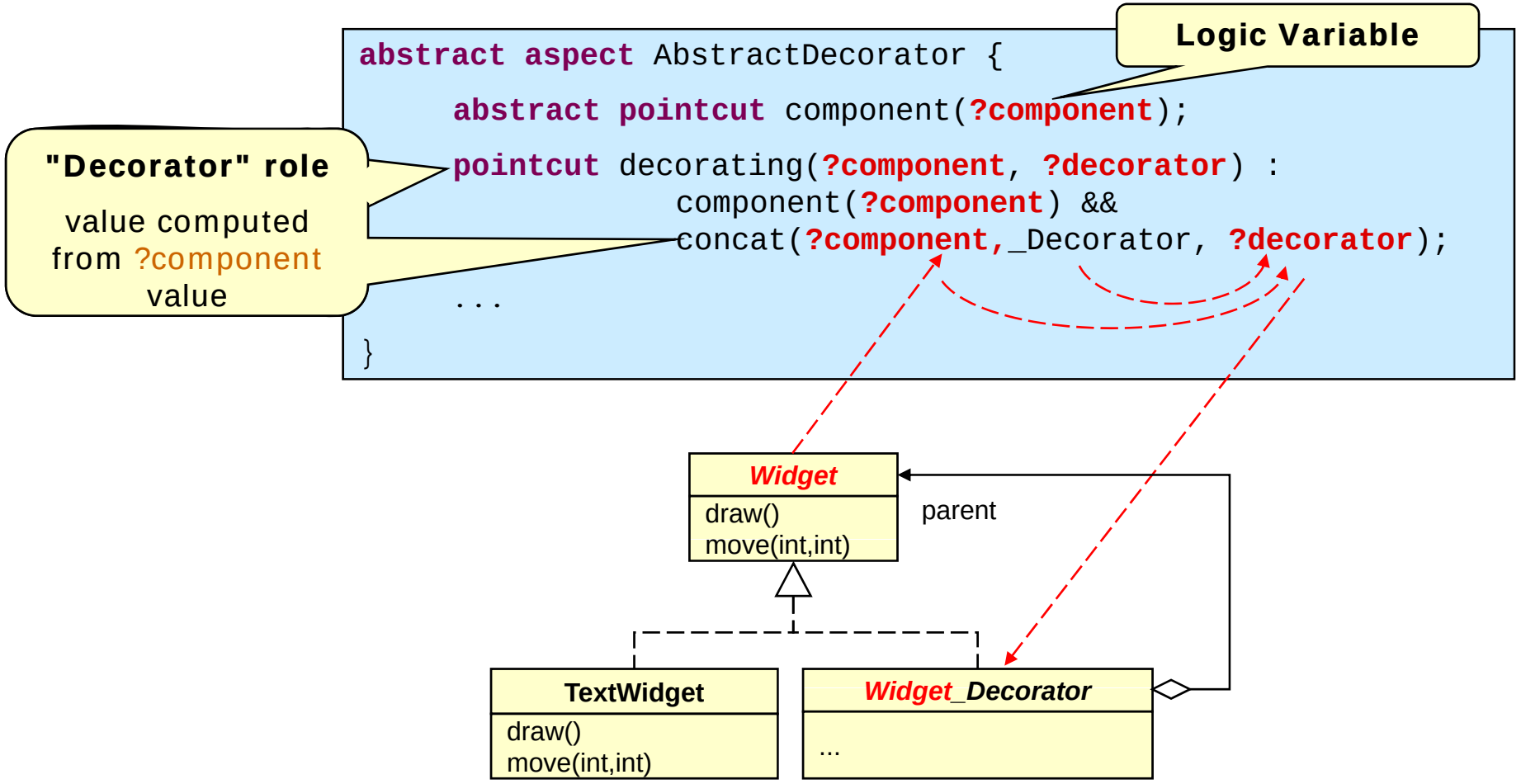
Defining: The pattern is the only reason why the class is there.
Superimposed: An existing class takes a role in a pattern.

		Modularity Properties			
Pattern Name	**Kinds of Roles**	**Locality**	**Reusability**	**Composition Transparency**	**(Un)pluggability**
Abstract Factory, Bridge, Builder, Factory Method, Façade, Interpreter	Defining	✗	✗	✗	✗
Adapter, Command, Composite, Decorator, Flyweight, Iterator, Template Method, State, Proxy, Visitor, Memento, Strategy	Both Defining and Superimposed	✓	7 of 12	10 of 12	✓
Observer, Sinlgeton, Prototype, Chain of Responsibiltiy, Mediator	Superimposed	✓	✓	✓	✓

Conclusion

- Problem
  - tangled OO implementation of core logic and DPs
- Observation
  - Patterns describe collaborations → aspects
  - Patterns are inherently generic → logic variables
- Solution
  - Pattern implementations as uniformly generic aspects in LogicAJ
    - ⇨ Representation of roles as metavariables
    - ⇨ Binding of roles via pointcuts
    - ⇨ Use of context-dependent effects for expressing the dependency of pattern code on the context of the application

1. Representing Roles by Logic Variables



2. Creating a Class for the Decorator Role

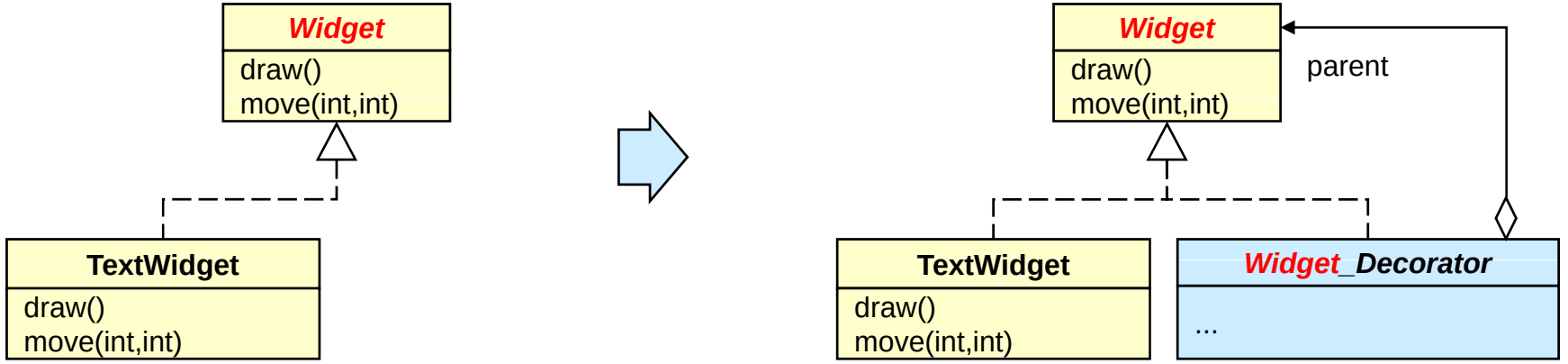
values of logic variables from pointcut

... are used in introduction

```
abstract aspect AbstractDecorator {

    ...

    introduce(?component,?decorator):
        decorating(?component, ?decorator) {
            abstract class ?decorator implements ?component {
                protected ?component parent;
            }
    }
}
```

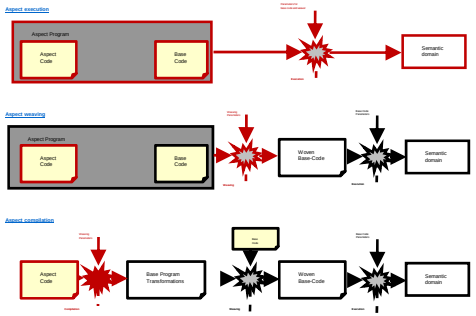


Kap 10. Semantik von Aspekten

Die folgenden Folien sind größtenteils als Prüfungsfragen formuliert, um Ihnen nicht nur als Erinnerung an Besprochenes sondern auch als Selbsttest zu dienen. Die Unterpunkte geben Hinweise auf die gewünschten Antworten. Die Details finde Sie alle im Skript.

Fragenkatalog Aspekt-Semantik (1)

- Welche semantischen Modelle für Aspektsprachen kennen Sie?
  - Interpreter, Weaver, Compiler



- Erläutern Sie das Interpreter-Modell
  - Was muss modelliert werden?
  - Wie tut macht man es? → Natural Semantics
    - ⇨ Nur Prinzip von Natural Semantics bis einschliesslich Unterscheidung von Big-Step und Small-Step Semantics (keine Details des Methoc Interception Ansatzes von Ralf Lämmel)
- Erläutern Sie das Weaver-Modell
  - Grundidee / Abgrenzung zu Interpretation
  - Vorteil gegenüber Interpreteransatz
  - Dimensionen des Webens: Zeitpunkt, Art des Pointcuts
  - Weben statischer und dynamischer Pointcuts
    - ⇨ insbesondere: cflow

Fragenkatalog Aspekt-Semantik (2)

- Erläutern Sie das Compiler-Modell
  - Grundidee / Abgrenzung zu Weben
  - Vorteil gegenüber Weberansatz
- Bedingte Transformationen
  - Syntax
    - ⇨ Parametrisierte Bedingung + Transformation
    - ⇨ CT-Sprachelemente
  - Semantik
    - ⇨ Für alle erfolgreichen Substitutionen der Bedingung führe die Transformation aus
- Logikbasierte Programmdarstellung
  - Abstract Syntax Tree (AST)
  - Darstellung von AST-Knoten und -Kanten als Fakten
  - Analogie zu DB-Relationen

Fragenkatalog Aspekt-Semantik (3)

- Beispiel für CT-Bedingungen
- Beispiel für komplette CT
- Übersetzungsschema Aspekte → CTs
  - Pointcut → CT-Vorbedingung
  - Introduction → CT
  - Advice → CT
  - Join Point
    - ⇨ Metavariable unifiziert mit Referenzelement für eine Aktion
    - ⇨ Änderung der Reihenfolge von Elementen bezogen auf Referenzelement
- Vorteile des logikbasierten Übersetzungsansatzes
  - …

universitätbonn ROOTS

Kap 11. Aspekt Interferenzen

Fragenkatalog Aspekt-Interferenzen (1)

- Was ist das Problem?
  - Komposition von Aspekten und Basiskode
  - Komposition unabhängig entwickelter Aspekte
  - Interaktionen und Interferenzen
  - Beispiel
- Unterschied Interaktion / Interferenz
  - … dabei auch die 4 verschiedenen Arten von Interaktionen erläutern (intended, unintended, undesired, broken)
- Unterschied Interferenzen allgemein und Webe-Interferenzen
  - Allgemein: „(woven) program behaves wrongly“
  - Webe-Interferenzen: „woven program has wrong structure“
  - Denksport für Einserkandidaten: Warum ist das nicht das gleiche? Beispiel!

Fragenkatalog Aspekt-Interferenzen (2)

- Was sind die Anforderungen an eine Lösung?
  - Independent Development, Aspects as Components, Unanticipated Composition, Stressed Programmers
- Was bietet AspectJ?
  - declare precedence
- Warum reicht das nicht?
  - Beispiel openTheDoor goInsideRoom work goOutsideRoom closeTheDoor

Fragenkatalog Aspekt-Interferenzen (3)

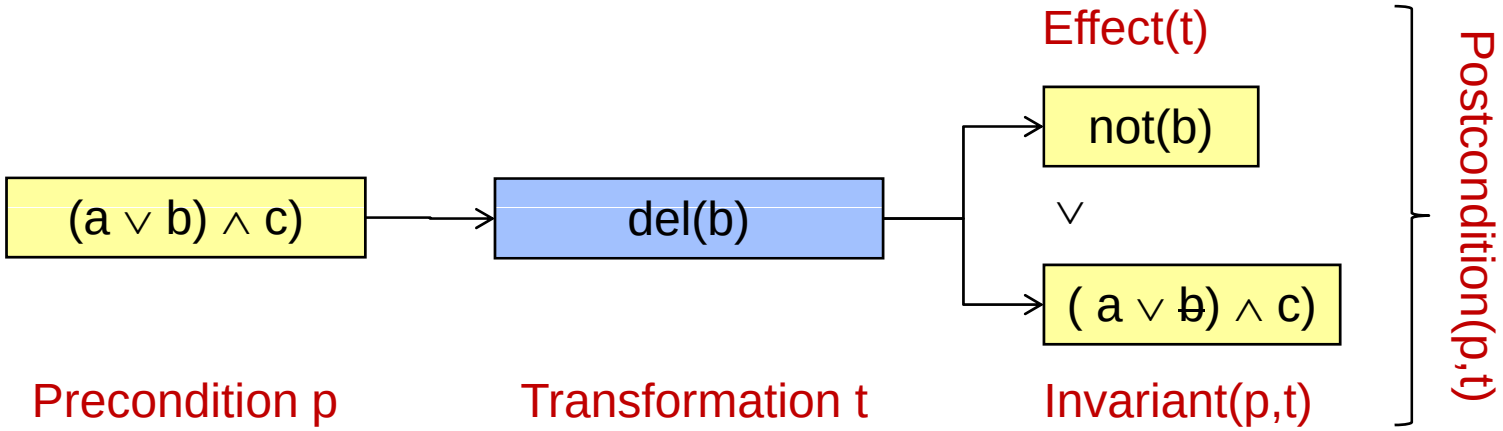
- Arten von Webe-Interferenzen?
  - Unvollständiges Weben
  - Unkorrektes Weben
- Ursachen von Webe-Interferenzen?
  - Verpasstes Triggering → Unvollständigkeit
  - Verpasste Inhibition → Unkorrektheit
- Obiges jeweils allgemein erläutern (Definition) und/oder am Counter-Getter-Beispiel erklären

Fragenkatalog Aspekt-Interferenzen (4)

- Ansatz zur Analyse von Webe-Interferenzen?
  - 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
- Herleitung von Effekt, Invariante, Postcondition?
  - Für einzelne Transformation
  - Für Transformationssequenz
- Was ist besonders zu beachten, bei CTs mit Variablen (Logik erster Ordnung)?
  - Negation des allgemeinsten Unifikators als Teil der Invariante

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



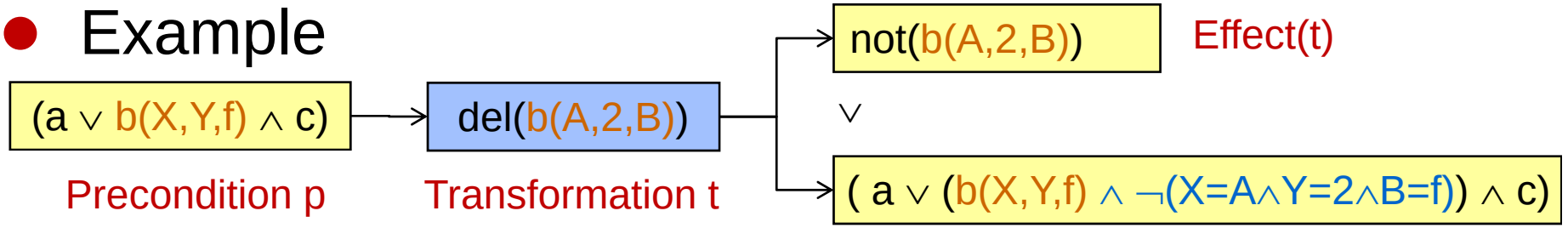
Invariant (First-order Logic)

- 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



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)))~~

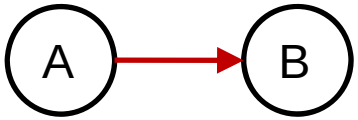
exists(method(<get>F,C)) ∨ exists(getfield(F,<get>F))

Fragenkatalog Aspekt-Interferenzen (5)

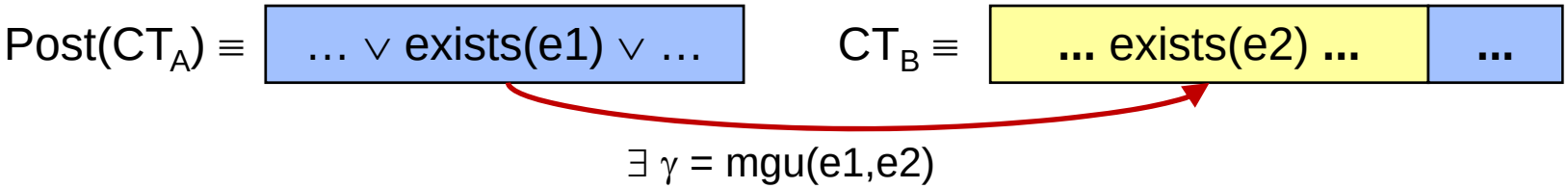
- Wie lässt sich triggering und inhibition auf Basis der Nachbedingung herleiten?
  - A triggert B: Unifizierbarkeit von Nachbedingungsliteral von A mit Vorbedingungsliteral von B
  - A behindert B: Unifizierbarkeit von … mit Negation eines …
- Aufbau des Abhängigkeitsgraphen?
  - Analyse aller Paare von Vor und Nachbedingungen
- Nutzung des Abhängigkeitsgraphen?
  - Self inhibition: Erforderlich!
  - Sonstige Zyklen: Verschiedene Problemfälle
  - Topologische Sortierung möglich: “korrekte” Ordnung(en)

Dependencies

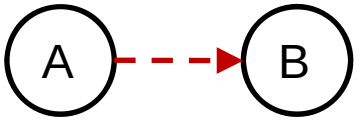
- A possibly triggers B



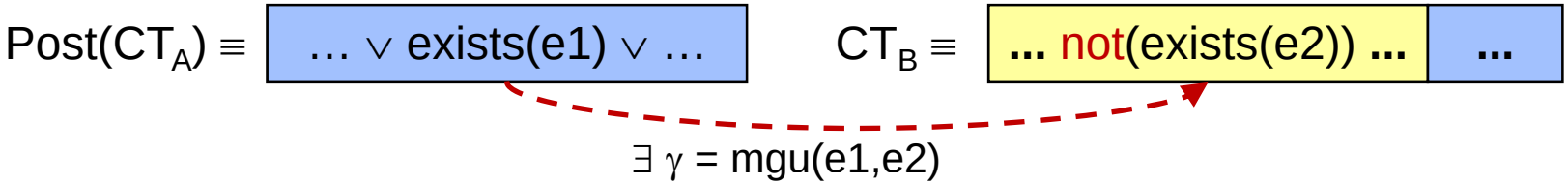
  - A‘s postcondition makes a literal in B's preconditon true



- 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 (5)

AddGetter

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

UseGetter

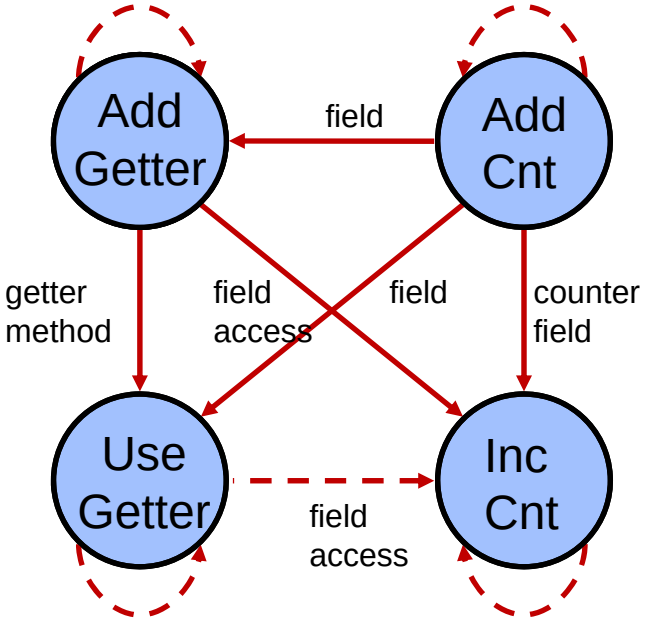
getfield(F,M) ∧ field(F,C) ∧ class(C) ∧ method(<get>F,C) ∧ M≠<get>F
replace(getfield(F,M) , call(<get>F,M))

AddCnter

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

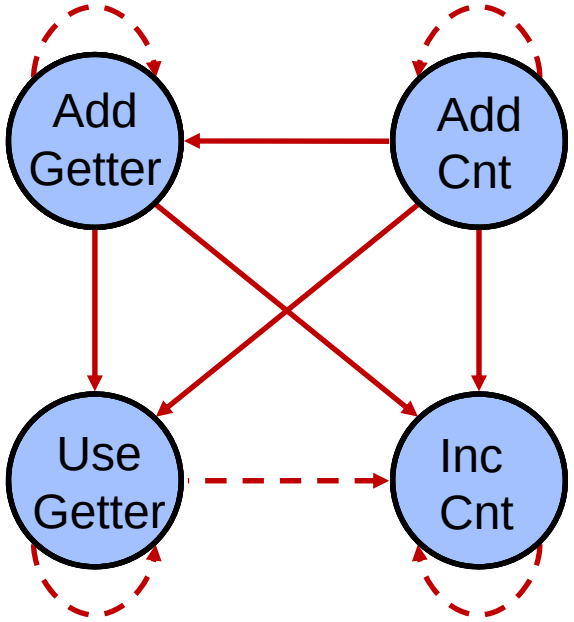
IncCnter

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



Result: Dependency Graph

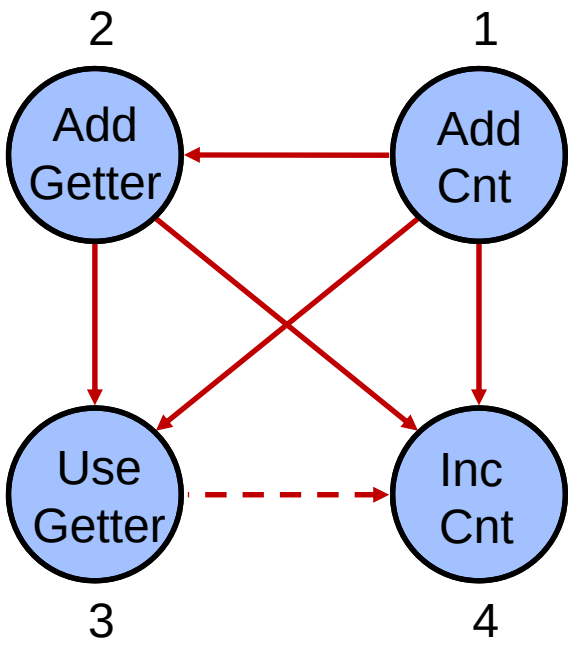
- What does it tell us?
- Self-Inhibition
  - Necessary, to prevent infinite application of the same CT to the same data
  - Issue a warning, for each CT without self-inhibition!
- Automatically determinine the "right" order
  - Use graph without self-inhibition arcs (next slide)



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!

References: Analysis of Aspect Weaving

- 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.
  - Link on homepage of Günter Kniesel → Publications