Software Product Lines

Concepts, Analysis and Implementation

Programmier-Paradigmen für Software-Produktlinien (2/3)

Dr. Malte Lochau
Malte.Lochau@es.tu-darmstadt.de

Inhalt

I. Einführung
- Motivation und Grundlagen
- Feature-orientierte Produktlinien

II. Produktlinien-Engineering
- Feature-Modelle und Produktkonfiguration
- Variabilitätsmodellierung im Lösungsraum
- Programmierparadigmen für Produktlinien

III. Produktlinien-Analyse
- Feature-Interaktion
- Testen von Produktlinien
- Verifikation von Produktlinien

IV. Fallbeispiele und aktuelle Forschungsthemen

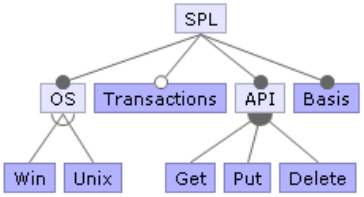
- Präprozessoren, Komponenten/Frameworks
- **FOP, AOP, DOP**
- Build-Systeme

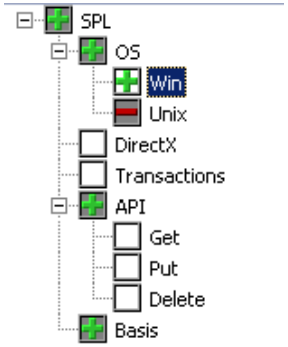
Software-Product-Line Engineering

Domain Eng.

Application Eng.

Feature-Modell

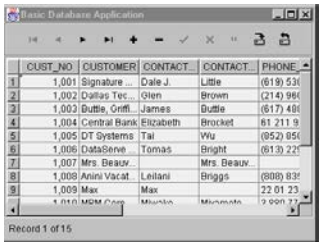




Feature-Auswahl

Wiederverwendbare Implementierungs-artefakte

Generator



Fertiges Program

Ziele von Programmierparadigmen für SPL

- Entwurf neuartiger Implementierungskonzepte und Sprachkonstrukte für Variabilität auf Programmebene
- Lösungsansätze für
  - Feature Traceability
  - Querschneidene Belange (Crosscutting Concerns)
  - Preplanning Problem
  - Unflexible Vererbungshierarchien
- Feature-orientiert: Modulare Feature-Implementierung (zumindest Kohäsion)

Feature Traceability Problem

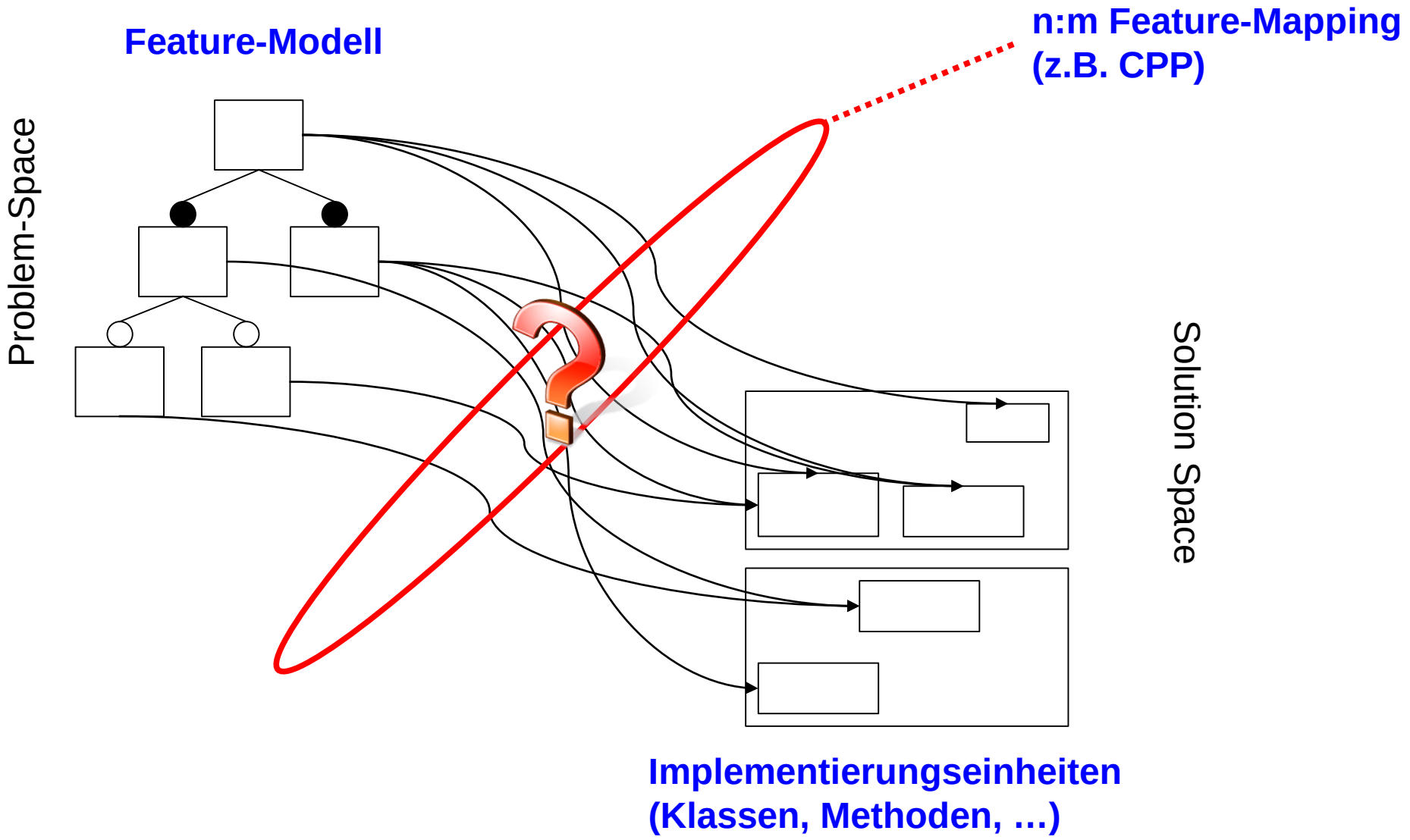
Idee: Alle Feature-spezifischen (syntaktischen) Implementierungseinheiten einer Produktlinie an einer bestimmten Stelle im Quelltext lokalisieren

- Features als explizites Element im Programmcode (Feature-Module)
- Features als Abstraktionskonzept der Programmiersprache

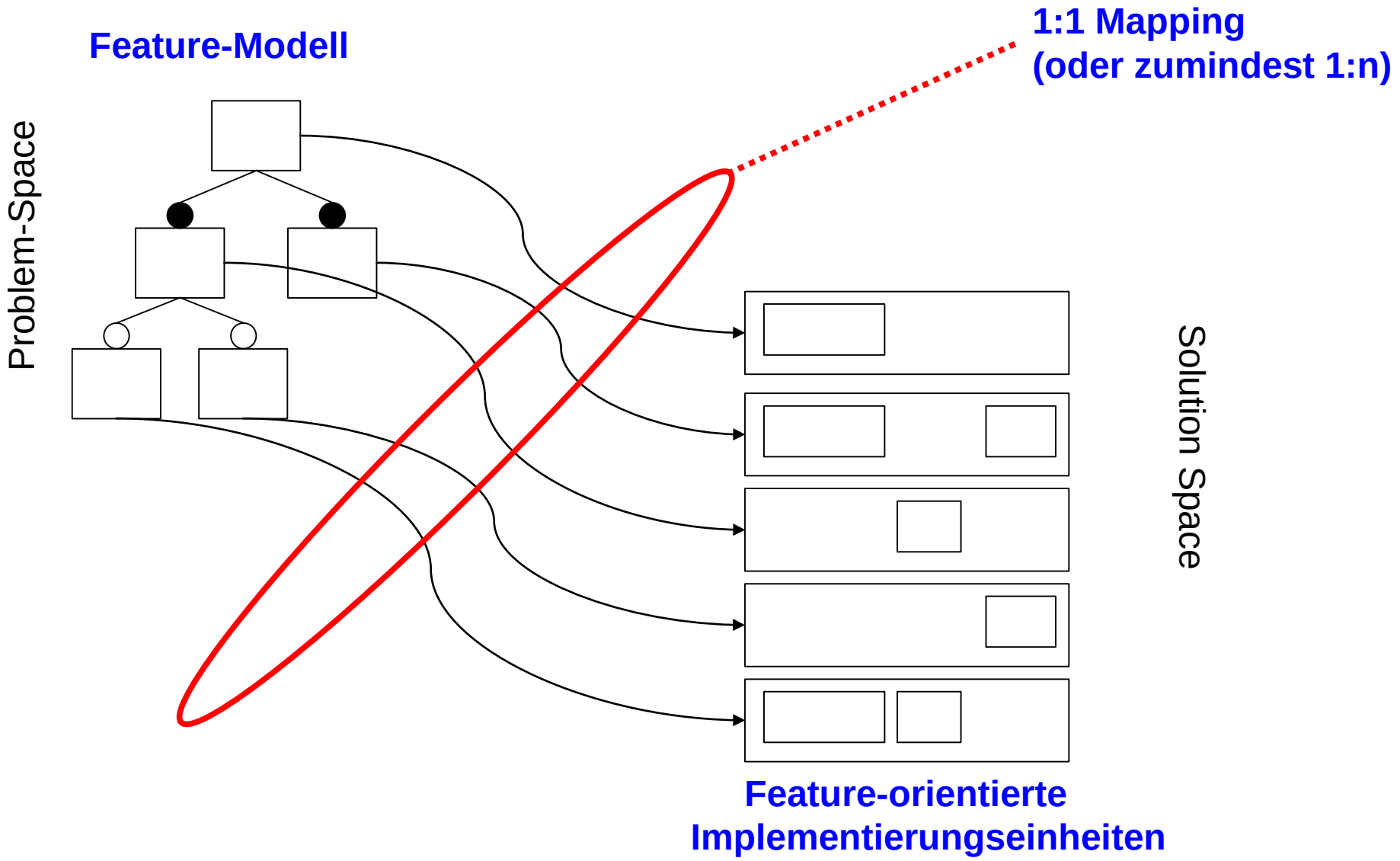
Lösung: Abhängig vom Programmierparadigma und der konkreten Programmiersprache bzw. der Programmier-umgebung

- Virtuelle vs. physische Trennung
- Statische vs. dynamische Variabilität

Syntaktisches Feature Mapping



Semantische Feature Traceability

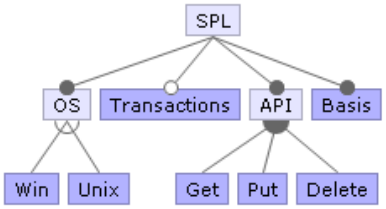


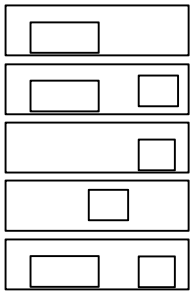
Produktlinien mit Feature-Modulen

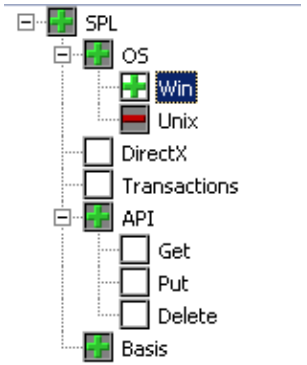
1:1 Mapping von Features
zu Feature-Modulen

Feature-Module

Feature-Modell



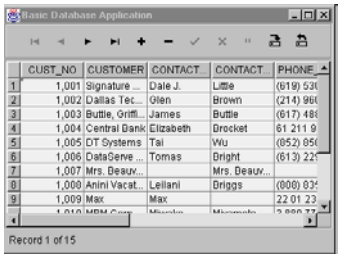




Feature-Auswahl

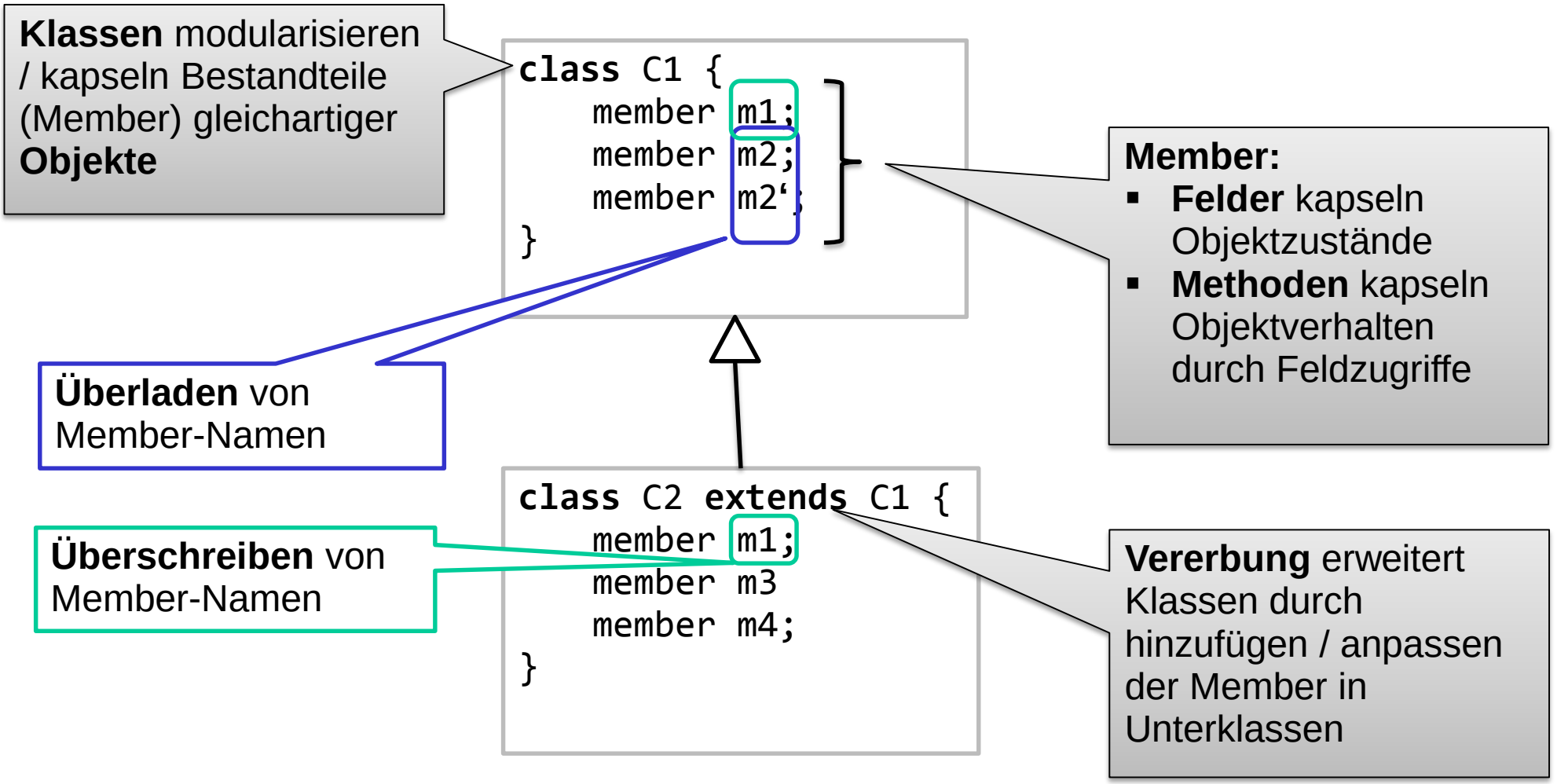
Feature-Auswahl
als Eingabe

Kompositions-Algorithmus



Fertiges Program

Exkurs: Objektorientierte Programme



Programmkomposition

Algebraische Programmkomposition

- Programme als **Wertemenge** (Strukturierte Artefaktmenge)
- Programmkompositionsoperatoren als **Verfeinerungsfunktionen** auf Programmen
- Beispiel objektorientierte Programmkomposition: Membership-Aggregation, Unterklassenvererbung

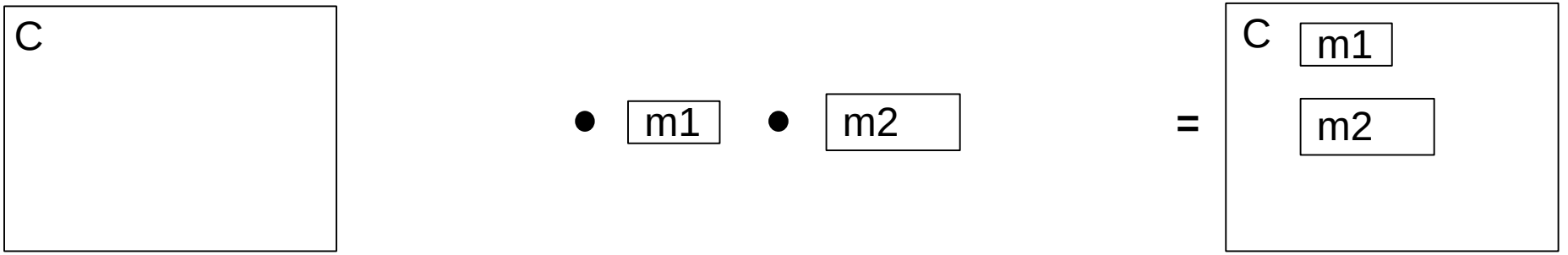
Komposition durch Aggregation

Syntax: Aggregation / Kapselung

```
class C { }
```


```
class C {
    member m1;
    member m2;
}
```

Kompositionssemantik: Hinzufügen in Klassen



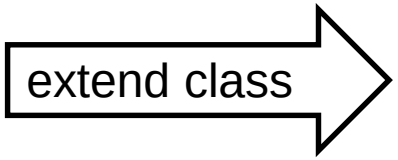
Formale Semantik: (Disjunkte) Mengenvereinigung

$$C = \{\ \} \quad \cup \{ m1 \} \quad \cup \{ m2 \} \qquad \{ m1, m2 \}$$

Komposition durch Vererbung

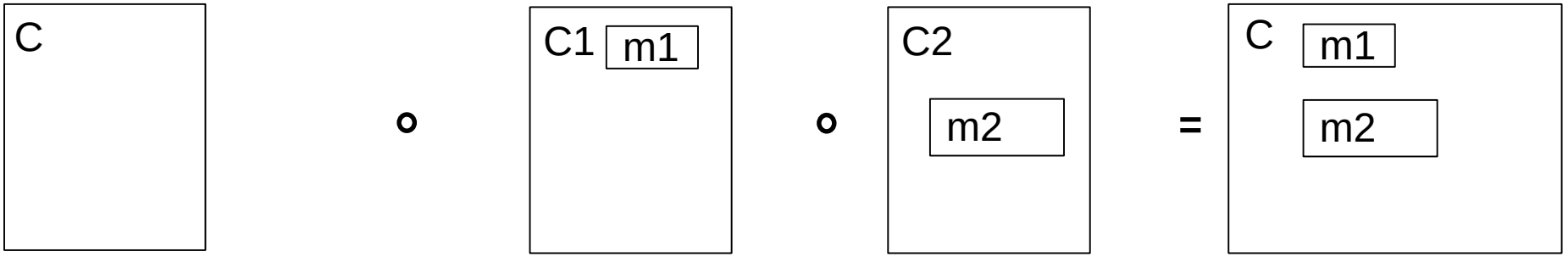
Syntax: Vererbung

```
class C { }
```




```
class C  extends C1 {}
class C1 extends C2 { member m1;}
class C2 { member m2; }
```

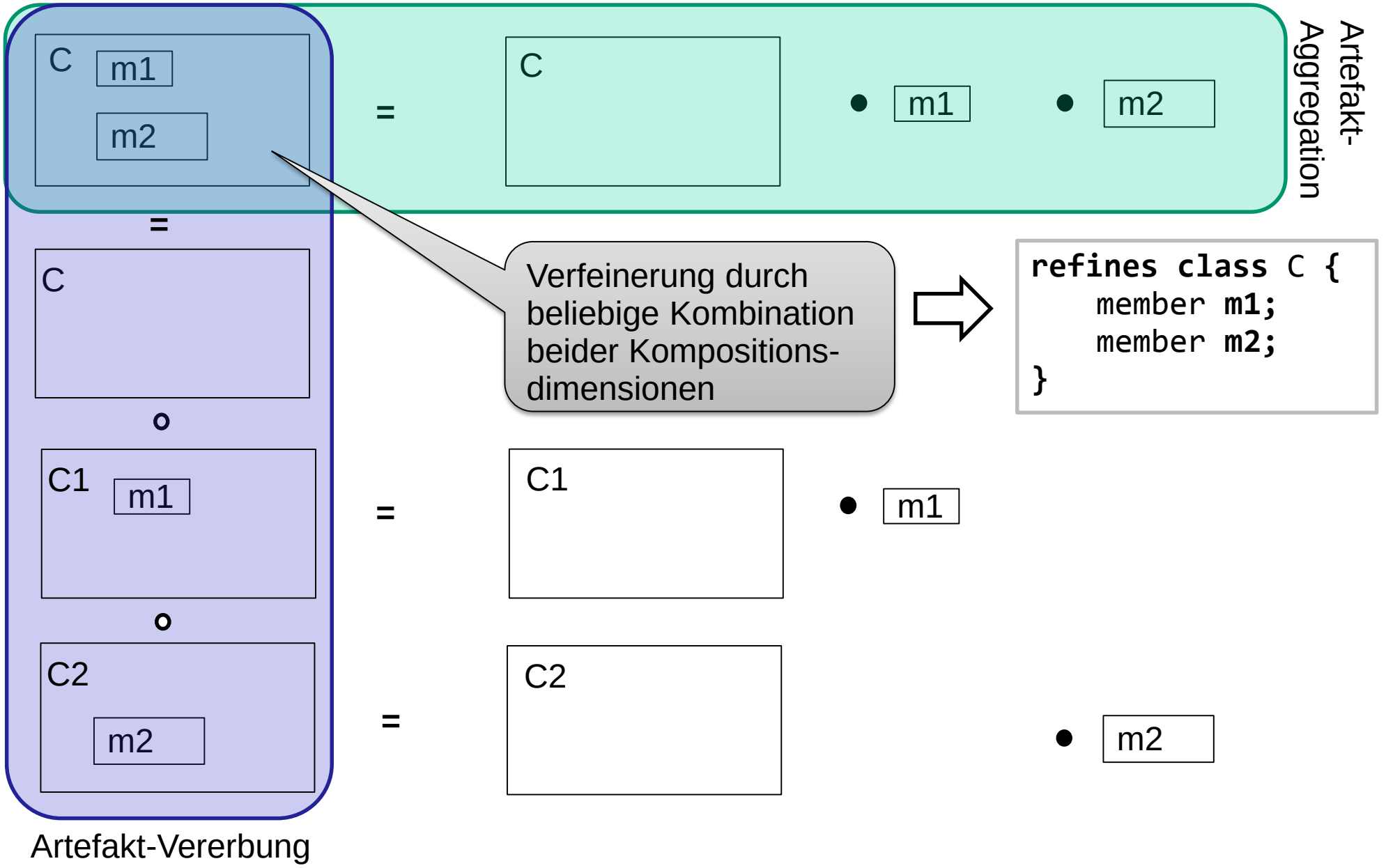
Kompositionssemantik: Erweitern von Klassen



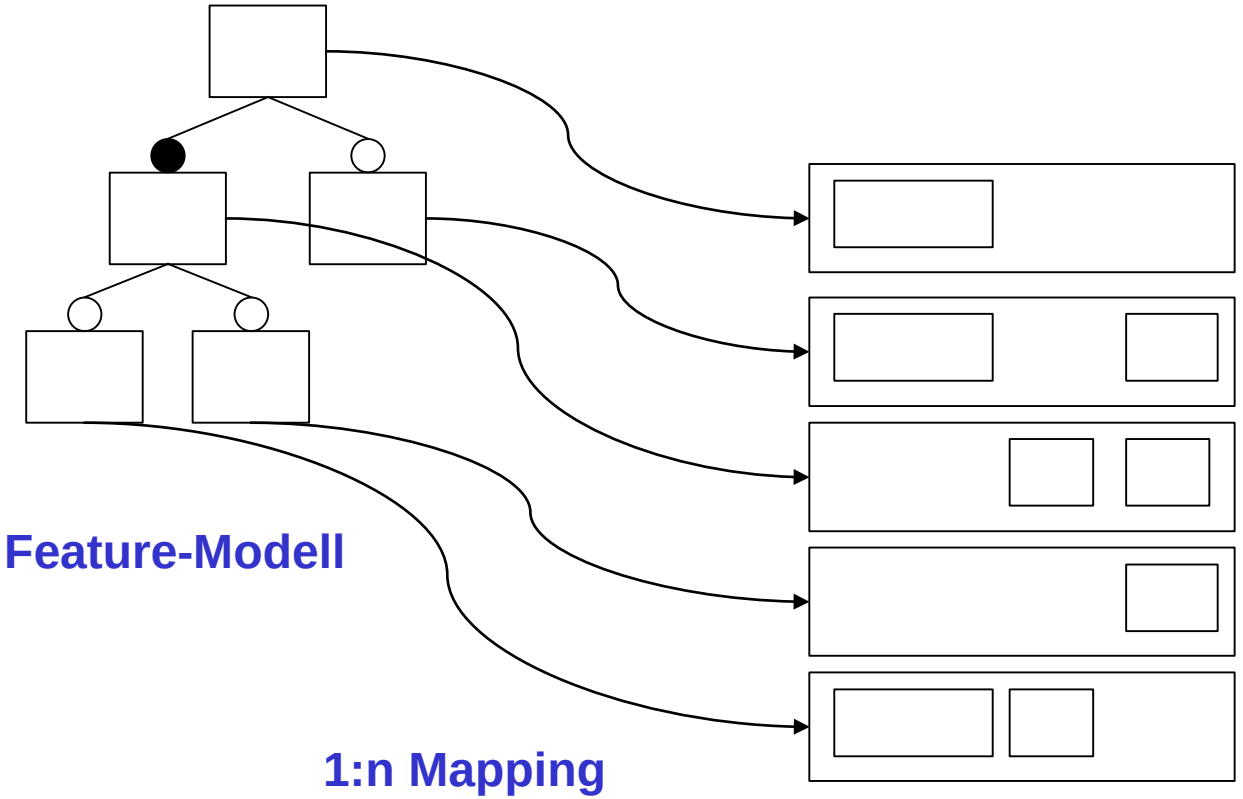
Formale Semantik:

- *Overriding: rechtsassoziative Mengenvereinigung*
- *Einschränkung durch Sichtbarkeit, abstrakte Methoden etc.: partielle Kompositionsfunktion*
- *…*

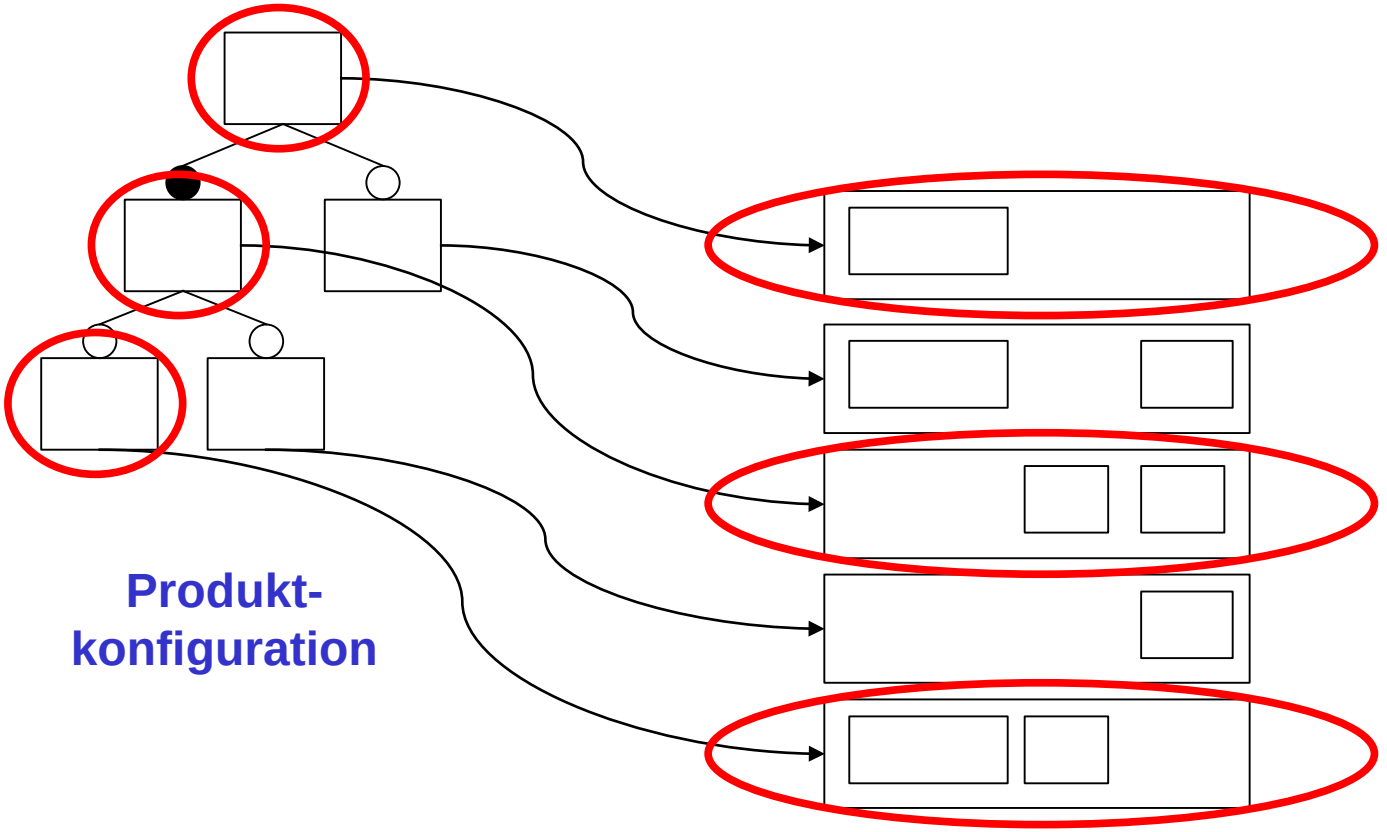
Objektorientierte Komposition



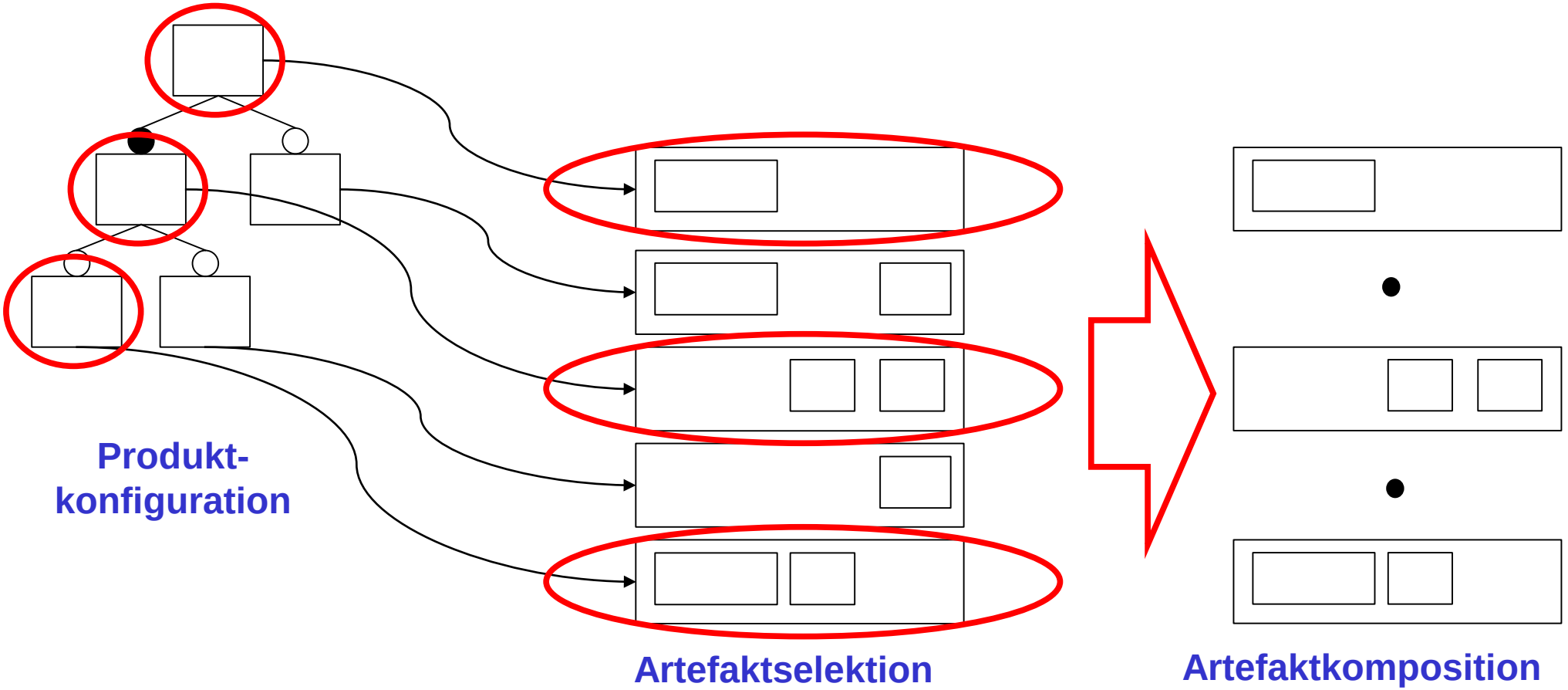
Feature Komposition



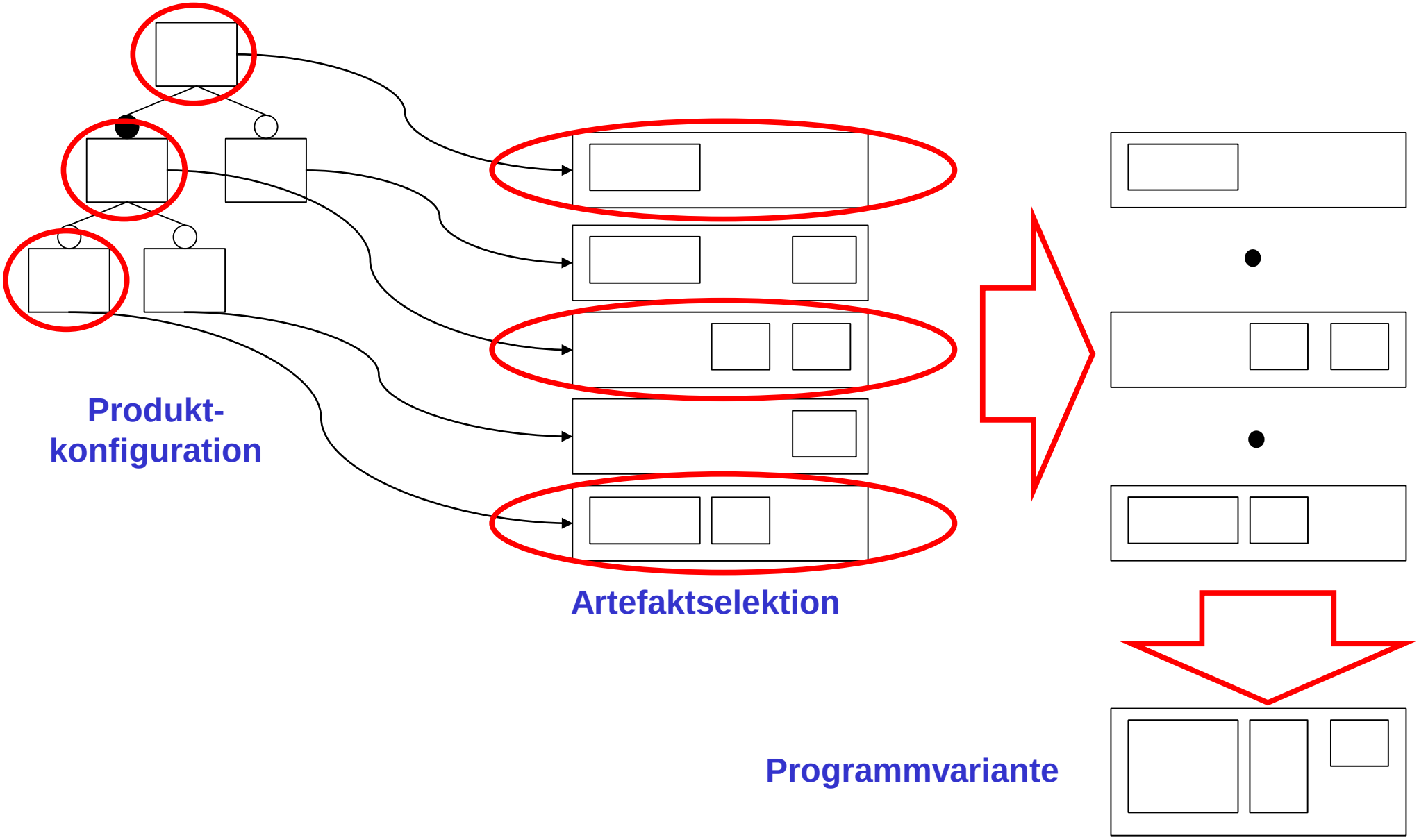
Feature Komposition



Feature Komposition



Feature Komposition



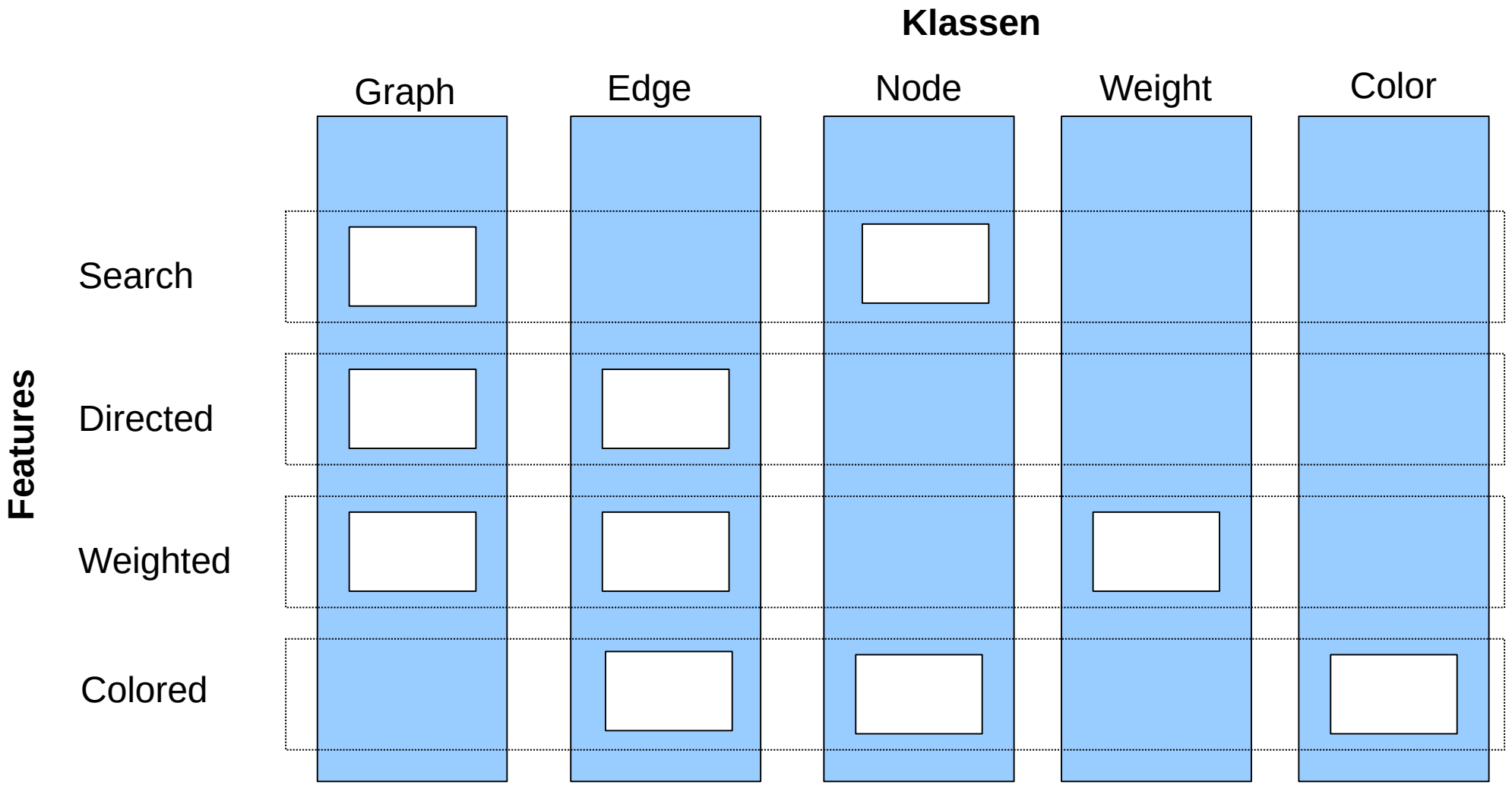
Feature Oriented Programming (FOP)

- Kompositionsbasierter Ansatz zur Lösung des Feature Traceability Problems
- Jedes Feature wird durch ein **Feature-Modul** implementiert
  - Ermöglicht 1:n Feature Traceability
  - Syntaktische Trennung und Modularisierung von Feature-Code
  - Einfache Feature-Komposition auf Basis der objektorientierten Programmkomposition
- Features als explizites Design-Konzept im Lösungsraum

Implementierung von Feature-Modulen

- **Grundlage:** Objektorientierte Dekomposition von Programmen durch Klassenstruktur
- **Ansatz:** Klassenartefakte Features zuordnen
- **Problem:** Features sind Querschnittsbelange
  - Mehrere Klassen tragen zur Implementierung eines Features bei
  - Eine Klasse trägt zur Implementierung mehrerer Features bei
- **Tools:**
  - AHEAD (Algebraic Hierarchical Equations for Application Design)
  - FeatureHouse
  - …

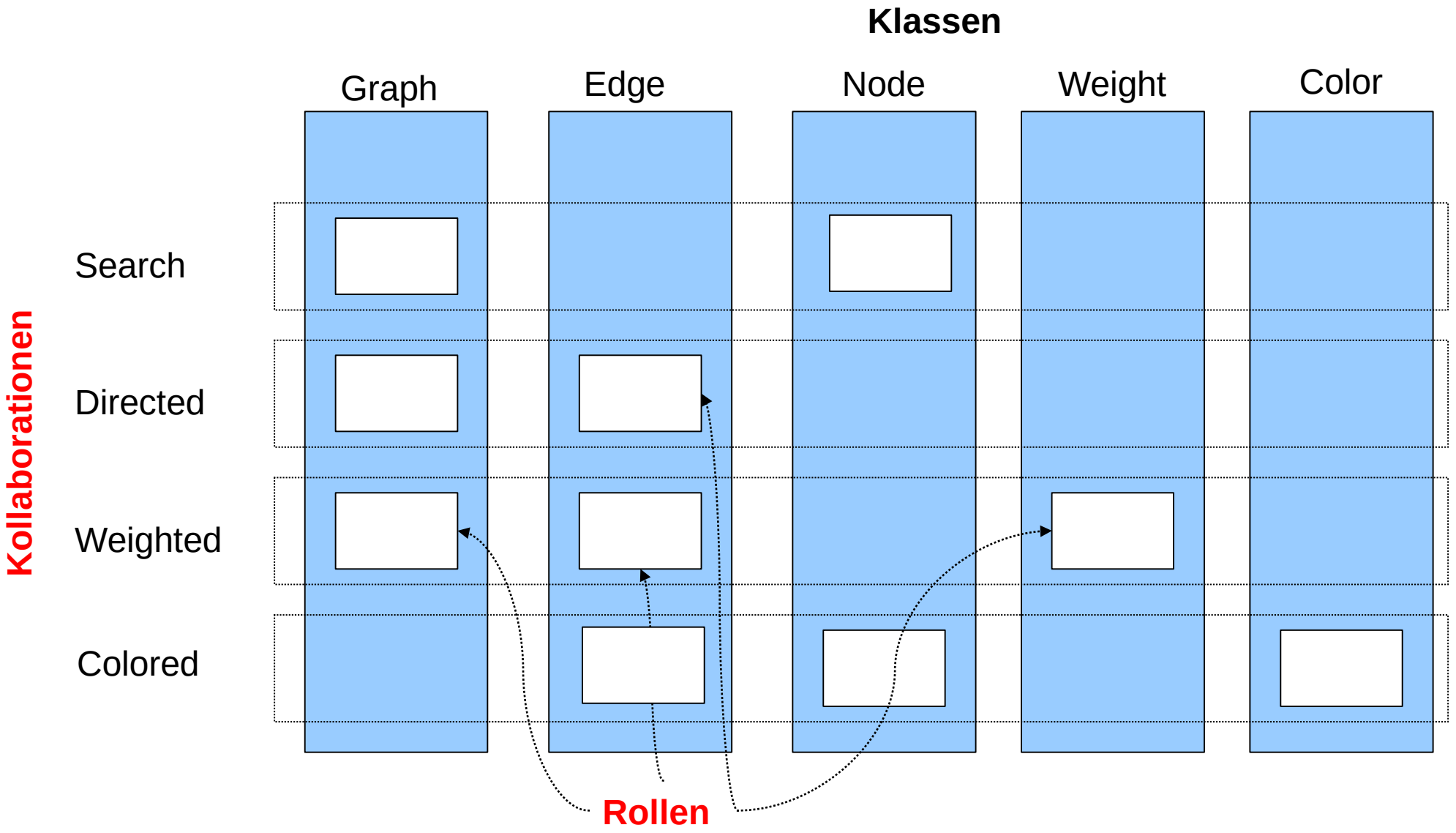
Beispiel: Aufteilen von Klassenartefakten auf Features



Kollaborationen und Rollen

- Eine Menge von Klassen, die miteinander interagieren, um ein Feature zu implementieren, bilden eine **Kollaboration**
- Jede Klasse in einer Kollaboration übernimmt eine bestimmte **Rolle** innerhalb der Kollaboration
- Eine Klasse kann Teil mehrerer Kollaborationen sein und in diesen verschiedene Rollen einnehmen (**Rollenpolymorphie**)
- Eine Rolle **kapselt** das Verhalten/die Funktionalität einer Klasse, welche(s) für eine Kollaboration (ein Feature) relevant ist

Beispiel: Kollaborationen und Rollen



Beispiel: Graph SPL

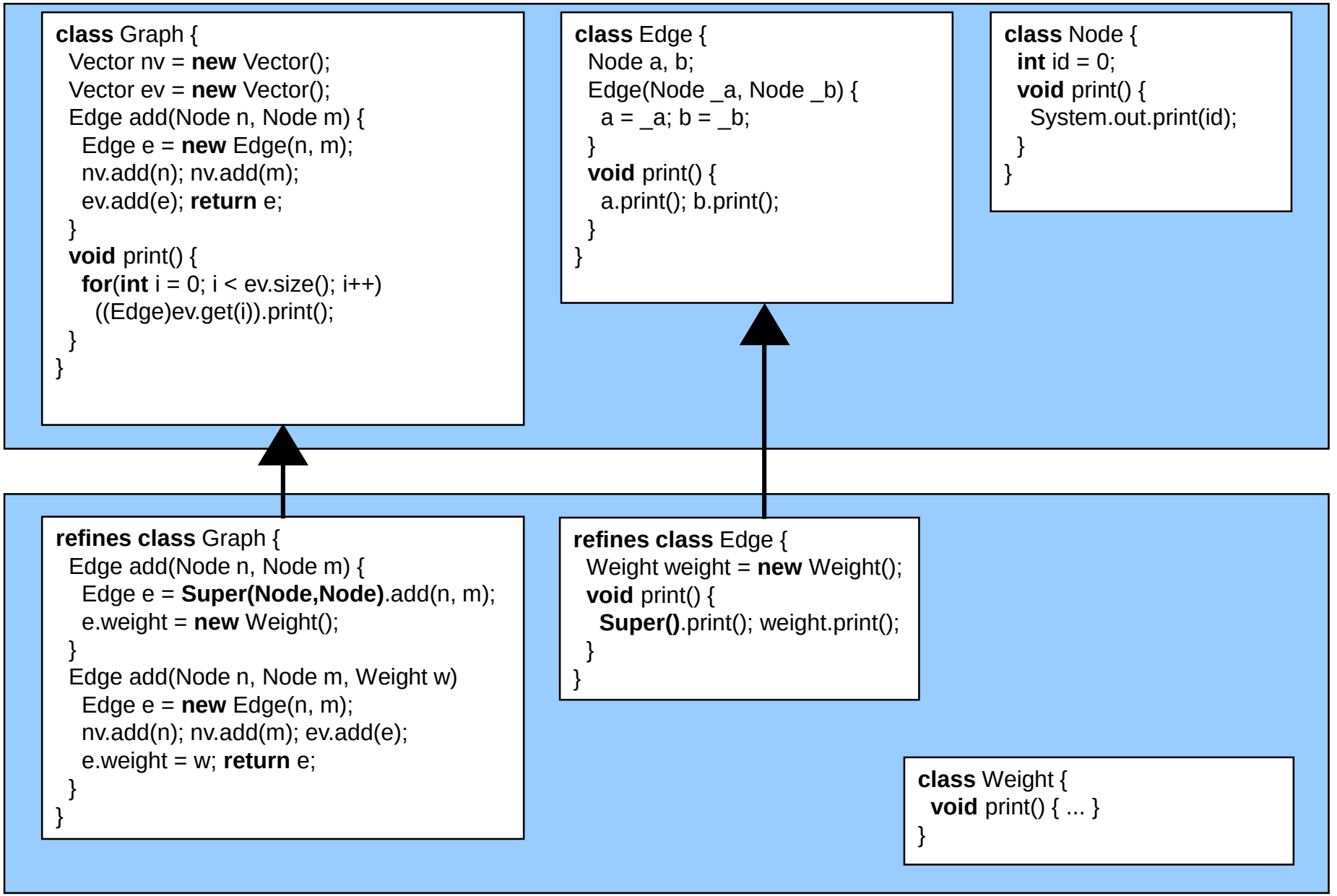
- Die Kollaboration *Weighted* umfasst alle Rollen, die nötig sind, um gewichtete Kanten zu implementieren (`Graph, Edge` und `Weight`)
- Die Klasse `Edge` hat eine Rolle, die den relevanten Teil für gewichtete Kanten implementiert, z. B. das Feld `weight`, die Methoden `getWeight` und `setWeight` und die Erweiterung der Methode `print`.
- Die Klasse `Edge` hat eine weitere Rolle für gerichtete Kanten

FOP Implementierung in JAK

- Erweiterung von Java 1.4 mit neuen Schlüsselwörtern für Verfeinerungen
  - **`refines`** für Klassenverfeinerungen
  - **`Super`** für Methodenverfeinerungen
- *Aus technischen Gründen müssen hinter Super die erwarteten Typen der Methode angegeben werden, z. B. `Super(String, int).print('abc', 3)`*

```
refines class Edge {
    private int weight;
    void print() {
        Super().print();
        System.out.print(' weighted with' + weight);
    }
}
```

Kollaborationen in FOP/JAK

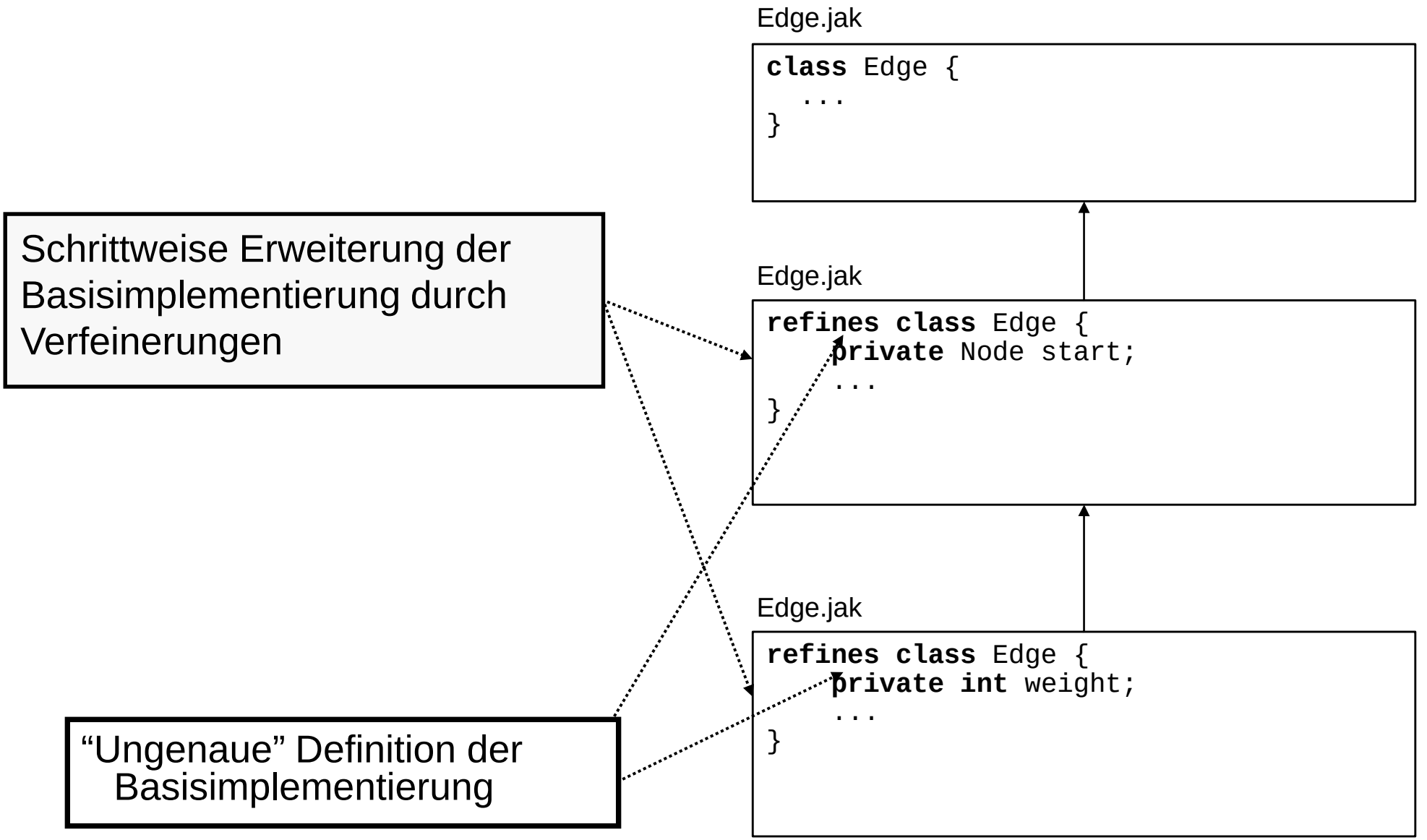


FOP Implementierung in FeatureHouse

- Behält Java 1.6 Syntax, aber neuer Kompositionsprozess
- Jede Klasse kann verfeinert werden und andere verfeinern (kein „refines“ Schlüsselwort)
- `original()` kennzeichnet Methodenverfeinerungen

```
class Edge {
    private int weight;
    void print() {
        original();
        System.out.print(" weighted with" + weight);
    }
}
```

Beispiel: Graph SPL in JAK



Methodenverfeinerung (JAK)

- Methoden können in jeder Verfeinerung eingeführt oder erweitert werden
- Überschreiben von Methoden
- Aufruf der Methode aus vorheriger Verfeinerung mit **`Super`**
- Ähnlich zu Vererbung

```
class Edge {
    void print() {
        System.out.print(
   " Edge between " + node1 +
          " and " + node2);
    }
}
```

```
refines class Edge {
    private Node start;
    void print() {
        Super().print();
        System.out.print(
            " directed from " + start);
    }
}
```

```
refines class Edge {
    private int weight;
    void print() {
        Super().print();
        System.out.print(
              " weighted with " + weight);
    }
}
```

Methodenverfeinerung (FeatureHouse)

- Methoden können in jeder Verfeinerung eingeführt oder erweitert werden
- Überschreiben von Methoden
- Aufruf der Methode aus vorheriger Verfeinerung mit **`original`**

```
class Edge {
    void print() {
        System.out.print(
   " Edge between " + node1 +
          " and " + node2);
    }
}
```

```
class Edge {
    private Node start;
    void print() {
        original();
        System.out.print(
            " directed from " + start);
    }
}
```

```
class Edge {
    private int weight;
    void print() {
        original();
        System.out.print(
              " weighted with " + weight);
    }
}
```

Alternativen zu Klassenverfeinerungen

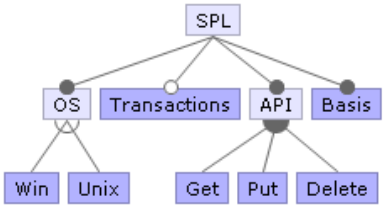
- Traits
- Virtuelle Klassen
  - Kollaboration ➔ Familienklasse
  - Rolle ➔ virtuelle (innere) Klasse
  - Komposition und Instanziierung zur Laufzeit
- Verschachtelte Klassen-, Layer- und Package-Hierarchien
  - z.B. Scala, Classbox/J, ContextL

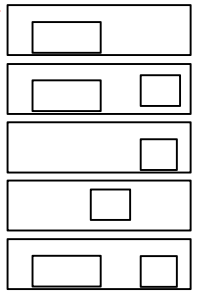
Produktlinien mit Feature-Modulen

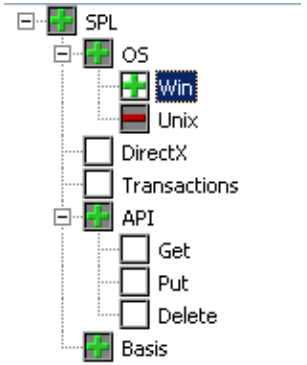
1:1 Mapping von Features
zu Feature-Modulen

Feature-Module

Feature-Modell



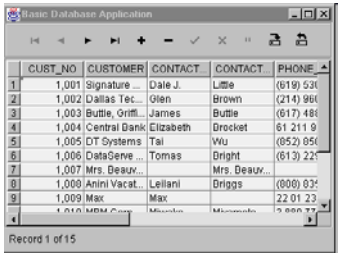




Feature-Auswahl

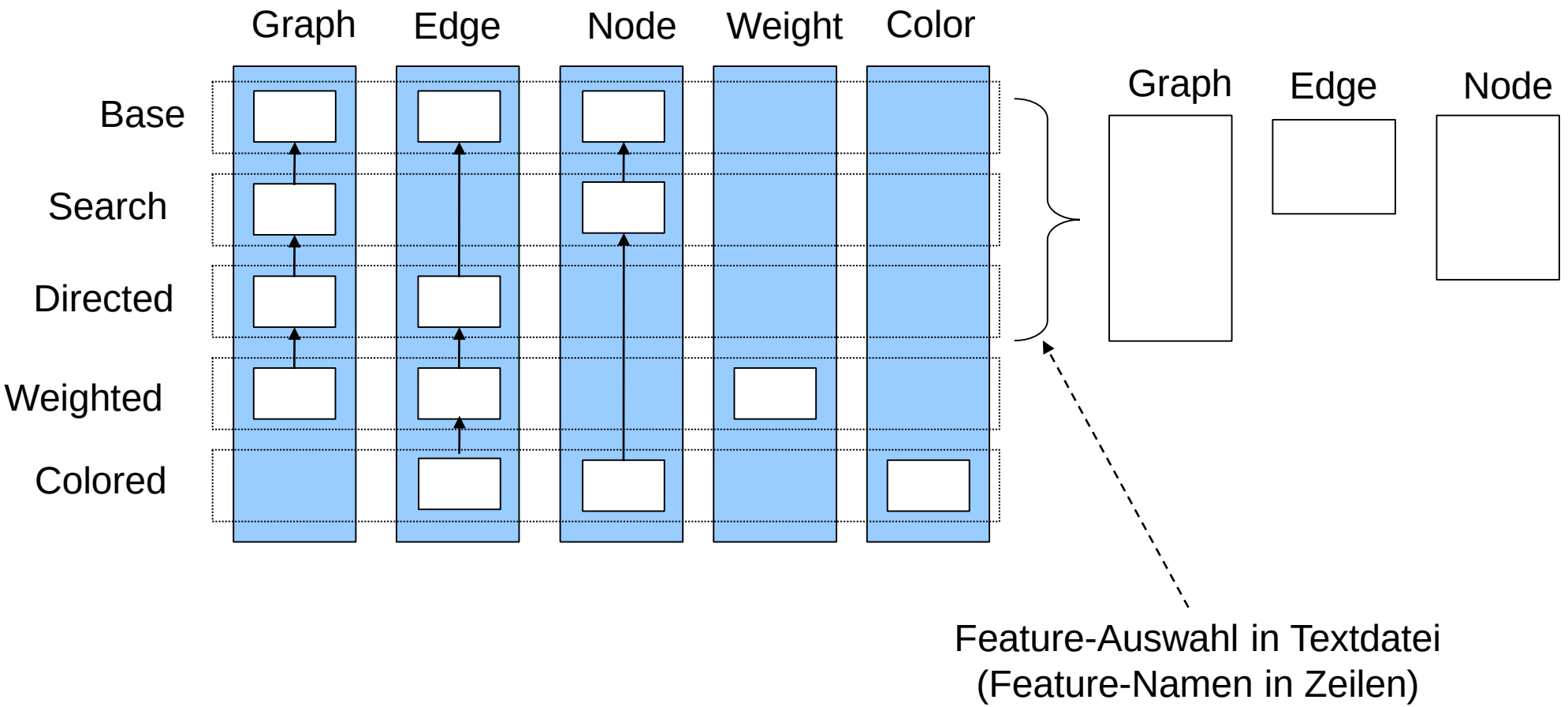
Feature-Auswahl
als Eingabe

Kompositions-Algorithmus



Fertiges Program

Beispiel: Komposition



Komposition mit Jampack

- Zusammenbauen einer Klasse durch Überlagerung (**superimposition**)
- Super/original-Aufrufe werden durch **inlining** integriert
- Ergebnis: eine Klasse

```
class Edge {
    private Node start;
    private int weight;
    void print() {
        System.out.print(
   " Edge between " + node1 +
          " and " + node2);
        System.out.print(
            " directed from " + start);
        System.out.print(
              " weighted with " + weight);

    }
}
```

Komposition mit Mixin

- Generierung einer Klasse pro Rolle mit entsprechender Hierarchie
- Umbenennung, so dass die finale Klasse den Klassennamen erhält
- Ersetzen von **`Super`** durch **`super`**

```
class Edge$$Base {
    void print() { ... }
}
class Edge$$Directed
  extends Edge$$Base {
    private Node start;
    void print() {
        super.print();
        System.out.print(
            " directed from " + start);
    }
}
class Edge extends Edge$$Directed{
    private int weight;
    void print() {
        super.print();
        System.out.print(
              " weighted with " + weight);
    }
}
```

Mixin vs. Jampack

- Jampack
  - Zuordnung des generierten Code zu Rollen nach Generierung nicht mehr vorhanden
- Mixins
  - Code-Overhead
  - Langsamer durch Indirektion bei Methodenaufrufen

Komposition mit AHEAD

- Eine Basisklasse + beliebige Verfeinerungen (Rollen)
- Klassenverfeinerungen können…
  - Felder einführen
  - Methoden einführen
  - Methoden verfeinern
- Feature-Modul (Kollaboration): Verzeichnis mit Basisklassen und/oder Verfeinerungen
- Beim Übersetzen werden Basisklasse + Verfeinerungen der ausgewählten Feature-Module zusammengebaut

Abstract Feature Composition

- Feature-Module können mit anderen Features „komponiert“ werden und bilden so komplexere Features
- Programme sind somit selber auch (zusammengesetzte) Features
- Menge F der Features; Kompositionsoperator $\bullet$

$$\bullet : F \times F \longrightarrow F$$

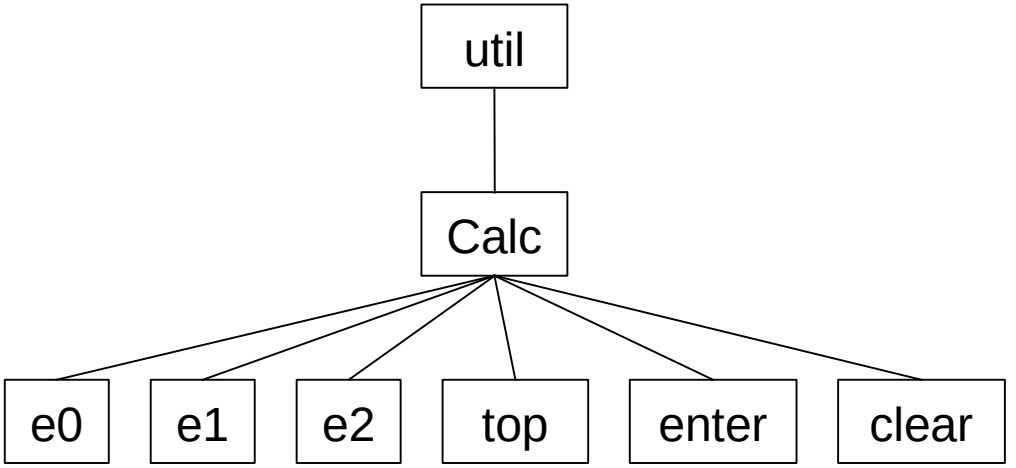
$$p = f_n \bullet f_{n-1} \bullet \ldots \bullet f_2 \bullet f_1$$

(assoziativ, aber nicht kommutativ)

Feature-Repräsentation als Baum

- Ein Feature besteht aus einem oder mehreren Code-Artefakten, jeweils mit einer internen Struktur
- Features können als Syntax-Bäume dargestellt werden (Feature Structure Tree – FST) – siehe Choice-Kalkül
- Syntaktische Struktur der Programm-Artefakte

```
package util;
class Calc {
  int e0 = 0, e1 = 0, e2 = 0;
  void enter(int val) {
    e2 = e1; e1 = e0; e0 = val;
  }
  void clear() {
    e0 = e1 = e2 = 0;
  }
  String top() {
    return String.valueOf(e0);
  }
}
```



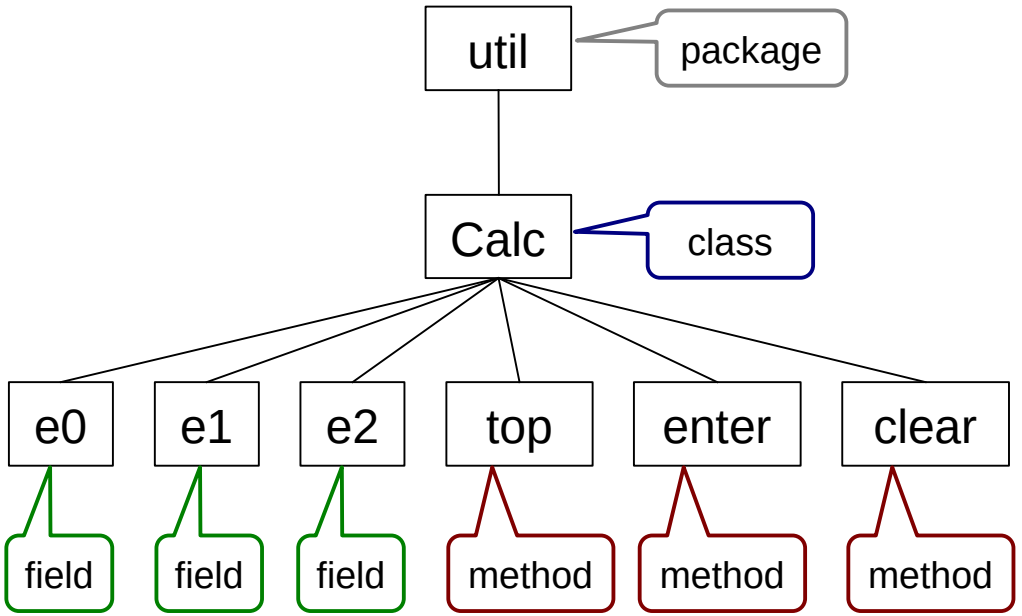
FST: Eigenschaften

- Nur die wesentlichen **Grundartefakte** der Programmiersprache sind im FST explizit als Knoten repräsentiert
- Detaillierungsgrad entsprechend der **Granularität** der Variabilität
- Datenstruktur als Grundlage für **Programmkomposition**
- Beispiel: Java
  - Packages, Klassen, Methoden, und Felder abgebildet
  - Statements/Expressions, Parameter, Initialwerte von Feldern nicht im FST abgebildet

FST: Aufbau

- Knoten im FST haben einen **Namen** und einen **Typ** entsprechend des zugehöirgen Programmartefakts
- Kindknoten sind **geordnet**

```
package util;
class Calc {
  int e0 = 0, e1 = 0, e2 = 0;
  void enter(int val) {
    e2 = e1; e1 = e0; e0 = val;
  }
  void clear() {
    e0 = e1 = e2 = 0;
  }
  String top() {
    return String.valueOf(e0);
  }
}
```



Komposition durch Baum-Superposition

```
package util;
class Calc {
  void add() {
    e0 = e1 + e0;
    e1 = e2;
  }
}
```

feature: Add

●

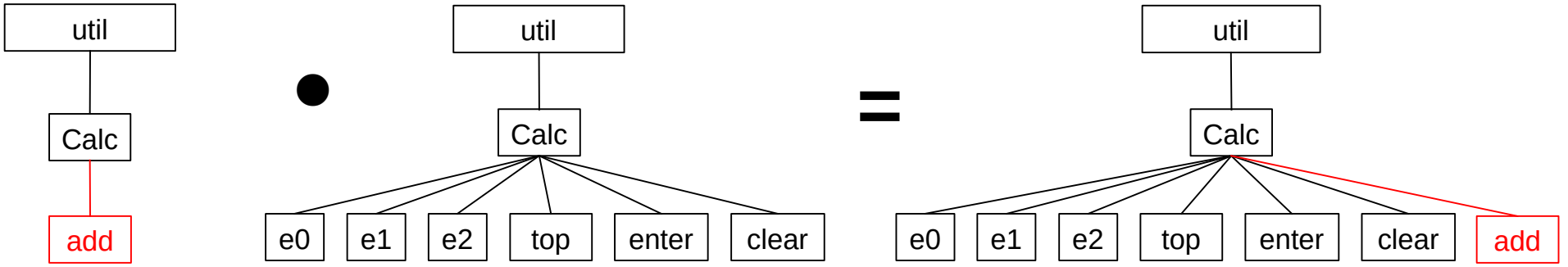
```
package util;
class Calc {
  int e0 = 0, e1 = 0,
          e2 = 0;
  void enter(int val) {
    e2 = e1; e1 = e0;
    e0 = val;
  }
  void clear() {
    e0 = e1 = e2 = 0;
  }
  String top() {
    return
     String.valueOf(e0);
  }
}
```

feature: CalcBase

feature: CalcAdd

=

```
package util;
class Calc {
  int e0 = 0, e1 = 0,
          e2 = 0;
  void enter(int val) {
    e2 = e1; e1 = e0;
    e0 = val;
  }
  void clear() {
    e0 = e1 = e2 = 0;
  }
  String top() {
    //...
  }
  void add() {
    e0 = e1 + e0;
    e1 = e2;
  }
}
```



Superposition von Bäumen

- Rekursive Überlagerung (**superimposition**) der Knoten, beginnend bei der Wurzel
- Zwei Knoten werden überlagert, wenn ...
  - ... ihre Vaterknoten überlagert wurden
  - ... und beide den gleichen Namen und Typ haben
- Nach der Überlagerung von zwei Knoten werden ihre Kinder rekursiv überlagert
- Wenn zwei Knoten nicht überlagert wurden, werden beide dem Ergebnisbaum an entsprechender Stelle hinzugefügt

Knotenarten im FST

- Nichtterminalknoten
  - Transparente Knoten
  - Können Kinder haben
  - Name und Typ, aber keinen weiteren Inhalt
  - Können problemlos überlagert werden
- Terminalknoten
  - Haben keine Kinder
  - Name und Typ
  - Können weiteren Inhalt haben, Überlagerung daher nicht trivial

Feature Komposition

- Rekursive Komposition der FST-Elemente:
  - package ● package → package (auch für Unterpakete)
  - class ● class → class (auch für innere Klassen)
  - method ● method → method, falls eine Methode die andere erweitert, z. B. indem sie Super oder original aufruft
  - field ● field → field, falls min. ein Attribut keinen Initialwert hat

Komposition von Terminalknoten

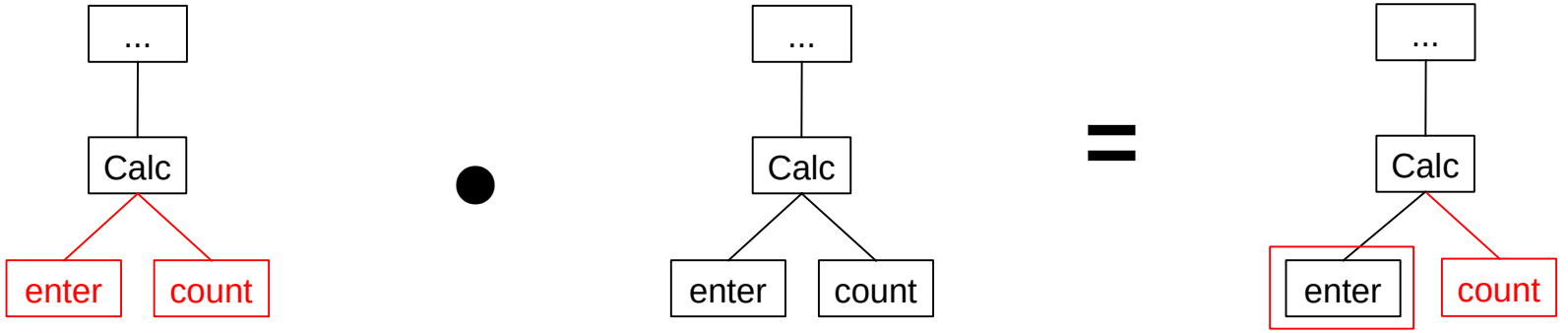
```
class Calc {
  int count = 0;
  void enter(int val) {
    original(val);
    count++;
  }
}
```

●

```
class Calc {
  int count;
  void enter(int val){
    e2 = e1;
    e1 = e0;
    e0 = val;
  }
}
```

=

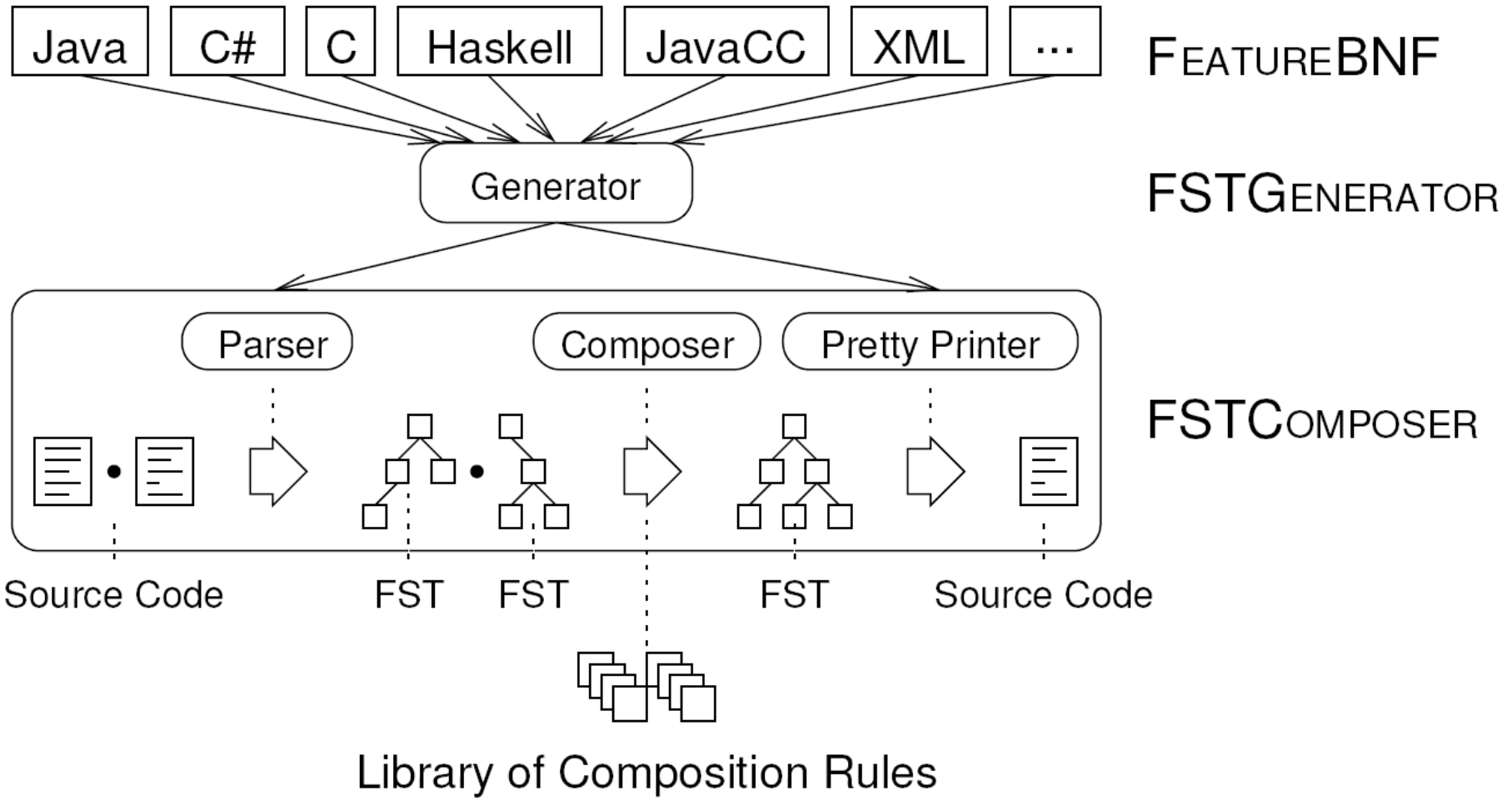
```
class Calc {
  int count = 0;
  void enter(int val) {
    e2 = e1;
    e1 = e0;
    e0 = val;
    count++;
  }
}
```



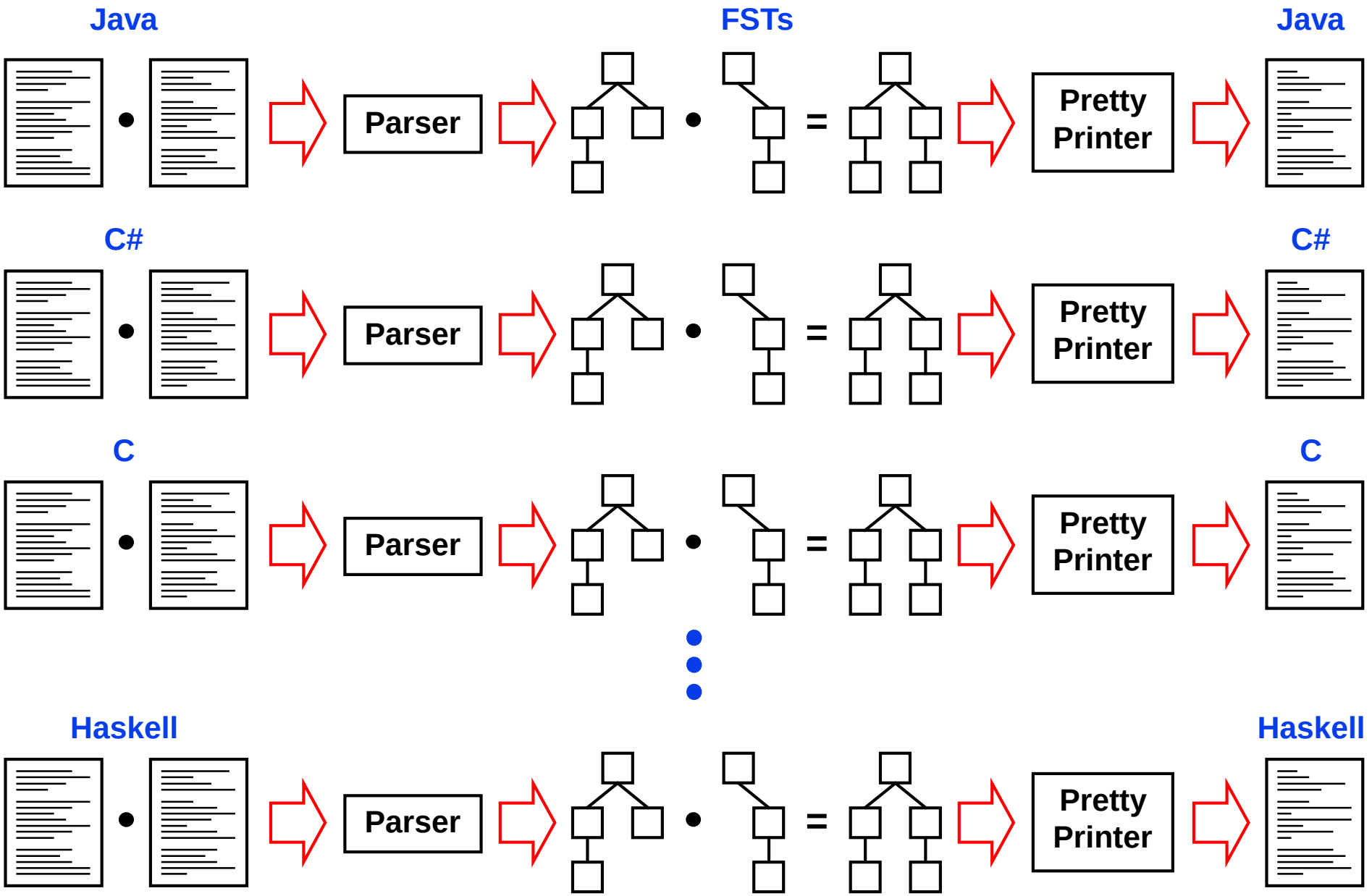
Komponierbarkeit: Bedindungen

- Die Struktur eines Features muss hierarchisch sein (Baumstruktur)
- Jedes Strukturelement muss einen Namen und einen Typ haben
- Ein Element darf nicht zwei Kinder mit dem gleichen Namen und dem gleichen Typ haben
- Elemente, die keine hierarchische Unterstruktur haben (Terminale) müssen eine spezielle Kompositionsregel angeben oder können nicht komponiert werden
- Objektorientierte Sprachen erfüllen meistens diese Bedingungen
- Einige Sprachen können mit zusätzlicher Struktur angereichert werden, z. B. XML

FST in FeatureHouse



FST in FeatureHouse



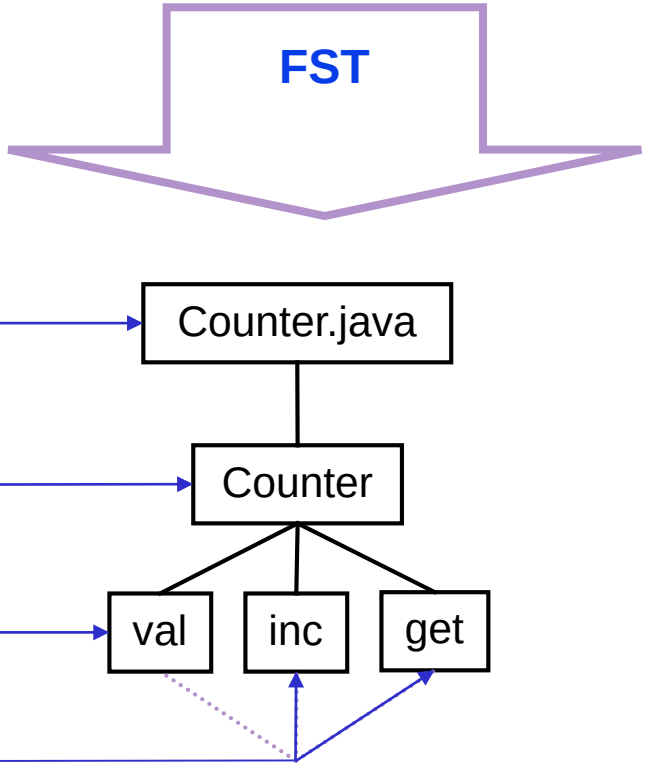
Annotierte Java-Grammatik

Grammatik

```
@FSTNonTerminal()
JavaFile : (ClassDecl)*;

@FSTNonTerminal(name="{Type}")
ClassDecl : "class" Type "extends" ExtType "{"
    (VarDecl)* (ClassConstr)* (MethodDecl)*
"}";

@FSTTerminal(name="{<IDENTIFIER>}({Params})",
    compose="MethodOverriding")
MethodDeclaration :
    Type <IDENTIFIER> "(" (Params)? ")" "{"
        "return" Expression ";"
    "}";

@FSTTerminal(name="{<IDENTIFIER>}",
    compose="FieldSpecialization")
VarDecl : Type <IDENTIFIER> ";";
```

Beispiel

```
class Counter {
  int val = 0;
  void inc() { val++; }
  int get() { return val; }
}
```



FOP: Diskussion

- Implementierung abhängiger Kollaborationen?
- Was würde es bedeuten, wenn ein Feature in einer Komposition mehrfach vorkommen kann (z. B. X ● Y ● X)?
- Wie können wir Komposition von Strukturen erreichen, bei dem die Reihenfolge der Kinder wichtig ist (z. B. XML)?
- Unter welchen Voraussetzungen ist Feature-Komposition kommutativ?
- Wie können wir eine Sprache „feature-ready“ gestalten (insb. Definition von Sprachmechanismen für Terminalüberlagerung)?
- Was passiert wenn wir das Löschen von Methoden erlauben wollen?

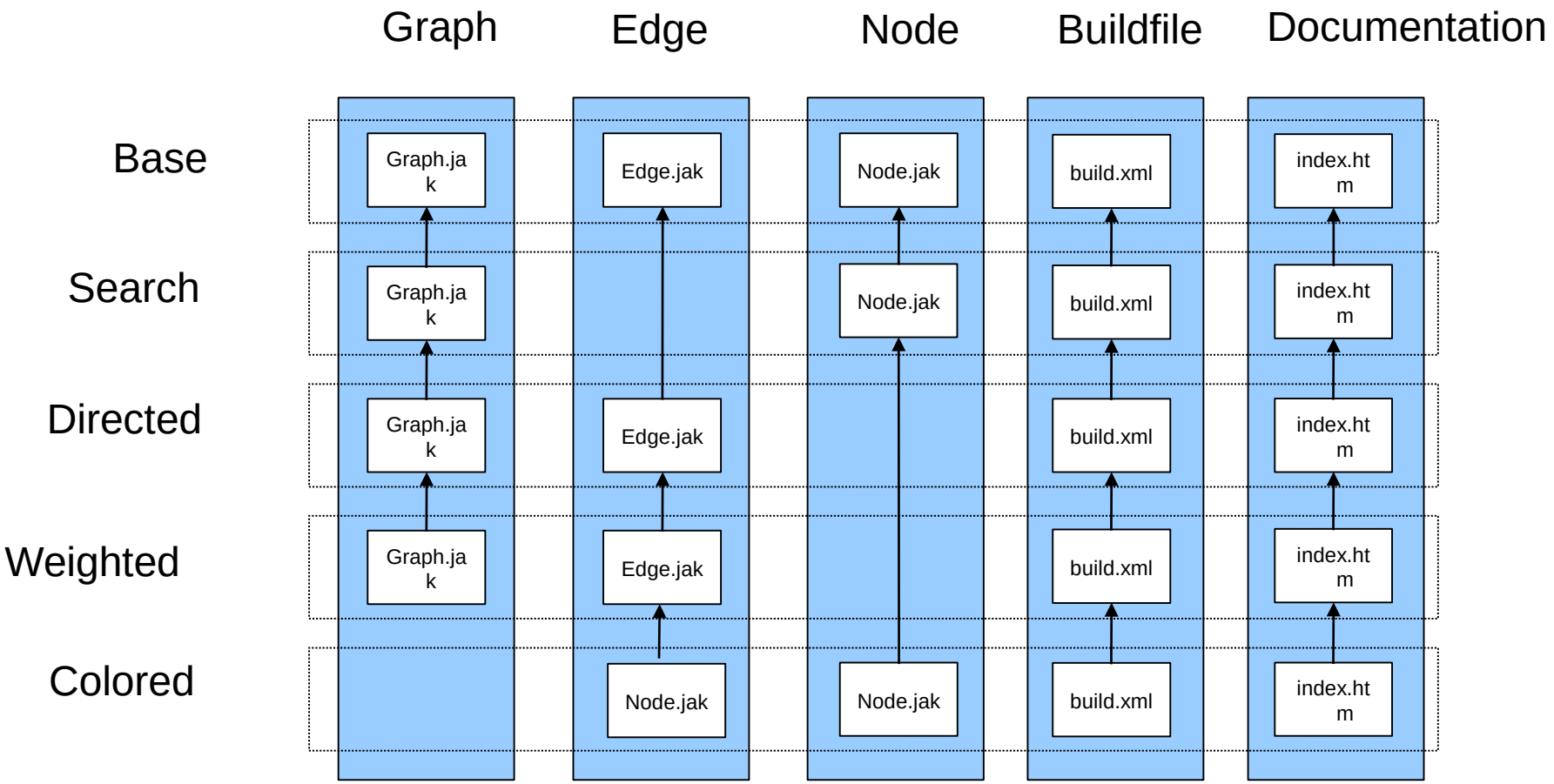
FOP: Zusammenfassung

- Feature-orientierte Programmierung löst u.a. das Feature-Traceability-Problem durch Kollaborationen und Rollen (1:1 Mapping)
- Implementierung mittels Klassenverfeinerungen
- Prinzip der Uniformität
- Modell auf Basis von Feature Structure Trees
- Implementierung von querschneidenen Belangen kann in bestimmten Fällen sehr aufwendig sein
  ➔ Aspektorientierung

Uniformity

- Feature-Artefakte bestehen nicht nur aus Java-Quellcode
  - Andere Programmiersprachen (z. B. C++, Javascript)
  - Build-Skripte (Make, XML)
  - Dokumentation (XML, HTML, PDF, Text, Word)
  - Grammatiken (BNF, ANTLR, JavaCC, Bali)
  - Modelle (UML, XMI, ...)
  - …
- Alle Arten von SPL Basis-Artefakten müssen verfeinert werden können
- Integration verschiedener Artefakte in Kollaborationen

Beispiel: Uniformity



Weitere Dateien: Grammatiken, Unit-Tests, Modelle, Spezifikationen, u.v.m.

Uniformity: Tool Support

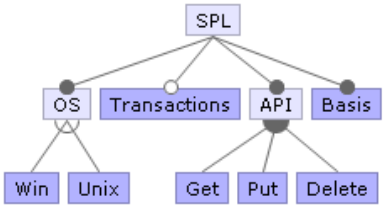
- AHEAD – Konzept sprachübergreifend, separate Tools für:
  - Jak (Java 1.4)
  - Xak (XML)
  - Bali-Grammatiken
- FeatureHouse – sprachübergreifendes Tool, leicht erweiterbar, z. Z. implementiert für:
  - Java 1.5
  - C#
  - C
  - Haskell
  - JavaCC- und Bali-Grammatiken
  - XML
- Virtuelle Klassen, Traits, Scala … sind sprachabhängig

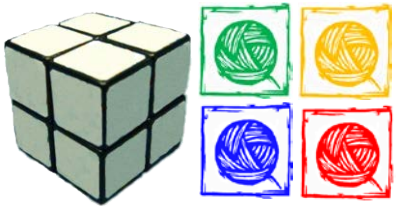
Produktlinien mit Aspekten

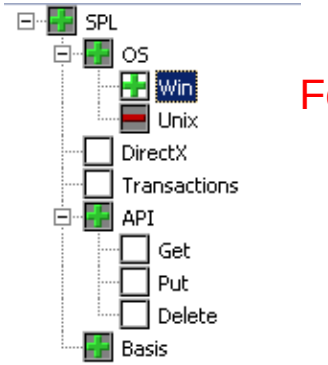
1:1 Mapping von Features zu Aspekten

Java-Programm + Aspekte

Feature-Modell



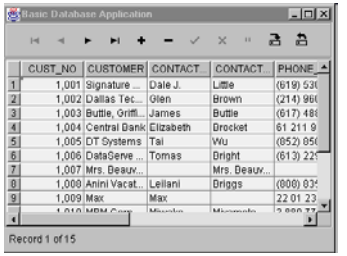




Feature-Auswahl

Feature-Auswahl als Eingabe

AspectJ-Compiler



Fertiges Program

Kompiliert mit ausgewählten Aspekten

AOP: Grundidee

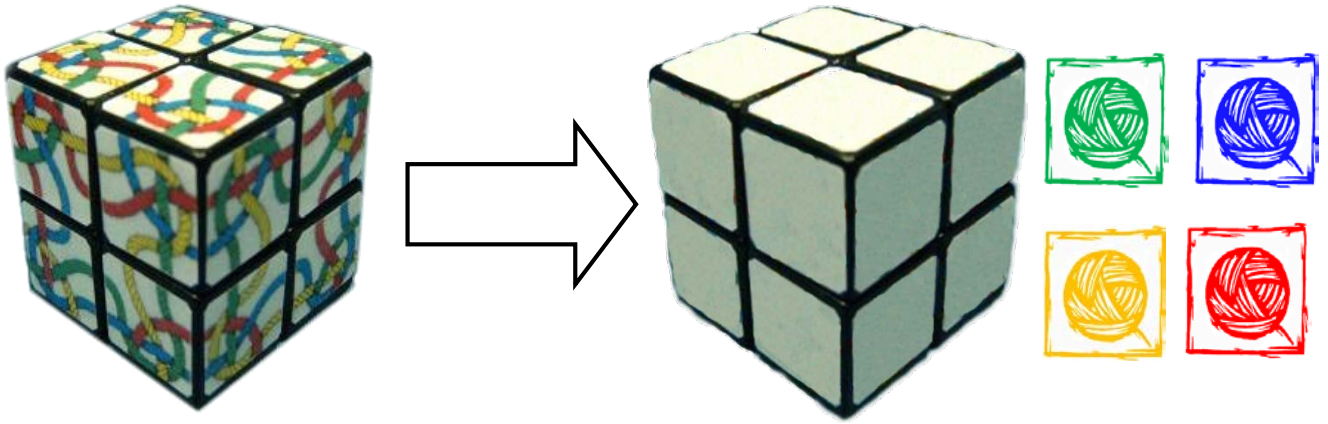
- Ein Programm hat zur Laufzeit eine „Sicht auf sich selbst“
- Objekte werden durch Klassen beschrieben, Klassen durch Metaklassen und Metaklassen durch Metametaklassen…
- Quelltext eines Programmes wird durch Metaklassen-Instanz repräsentiert
- Das Programm kann sich selbst anpassen, indem seine eigenen Metaklassen-Instanzen anpasst

Meta Object Protocols (MOP)

- Programme können auf Meta-Ebene manipuliert werden, um ihre Struktur und ihr Verhalten zu ändern
- Verhaltensänderungen erfolgt Ereignis-basiert: *„wenn das Ereignis X eintritt (z. B. Methode Y wird aufgerufen), führe Code Z aus“*
- Bekannt aus LISP; in Standard-Java nur sehr eingeschränkt möglich (Reflection-API)

Aspect-Oriented Programming (AOP)

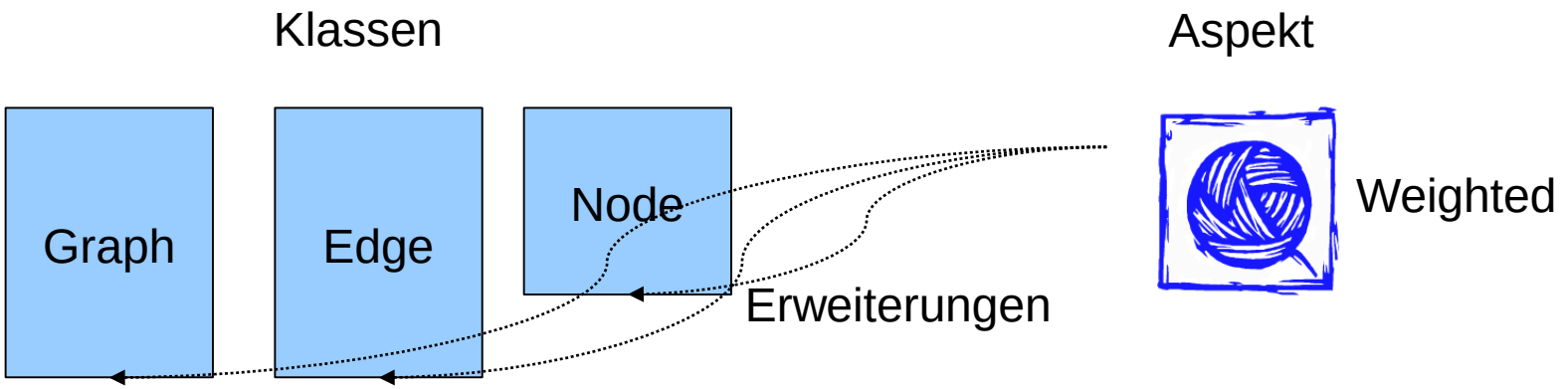
- Entwicklung aus Metaobjektprotokollen
- Ziel: Modularisierung von querschneidenden Belangen
- Erlaubt **statische und dynamische** Anpassungen
- Deklarativ mittels Quantifizierung



Hinweis: Es gibt viele verschiedene AOP-Ansätze; auch Mixins usw. werden gelegentlich als AOP klassifiziert. Wir folgen in dieser Vorlesung dem Pointcut/Advice-Ansatz populärer AOP-Sprachen wie AspectJ

AOP: Grundidee

- Modularisierung von querschneidenen Belangen in einem **Aspekt**
- Ein Aspekt beschreibt die **Änderungen** dieses Belangs in der restlichen Software
- Interpretation als **Programmtransformation**, als Meta-Objekt-Protokoll oder als Feature-Modul möglich



AspectJ

- AspectJ ist eine AOP-Erweiterung für Java
- Aspekte werden ähnlich wie Klassen implementiert, aber es gibt ein Reihe neuer Sprachkonstrukte
- Der Basiscode wird weiterhin in Java implementiert
- Aspekte werden von einem speziellen Aspekt-Compiler in den Basiscode „gewebt“ (**weaving**)

Was kann ein Aspekt?

Ein Aspekt in Programmiersprachen wie AspectJ kann

- Klassenhierarchien manipulieren
- Methoden und Felder zu einer Klasse hinzufügen
- Methoden mit zusätzlichem Code erweitern
- Ereignisse wie Methodenaufrufe oder Feldzugriffe abfangen und zusätzlichen oder alternativen Code ausführen

Beispiel: Statischer Aspekt

- Statische Erweiterungen mit „Inter-Typ-Deklarationen“
  - z. B. füge Methode X in Klasse Y ein

```
aspect Weighted {
        private int Edge.weight = 0;
        public void Edge.setWeight(int w) {
            weight = w;
        }
}
```

Beispiel: Dynamischer Aspekt

- Dynamische Erweiterungen auf Basis von Join-Points
  - Ein Ereignis in der Programmausführung, wie ein Methodenaufruf, ein Feldzugriff o.a., wird als (dynamischer) **Join-Point** bezeichnet
  - Ein **Pointcut** ist ein Prädikat um Join-Points (JPs) auszuwählen
  - Advice ist Code, der ausgeführt wird, wenn ein JP von einem Pointcut ausgewählt wurde

```
aspect Weighted {
    ...
    pointcut printExecution(Edge edge) :
        execution(void Edge.print()) && this(edge);

    after(Edge edge) : printExecution(edge) {
        System.out.print(' weight ' + edge.weight);
    }
}
```

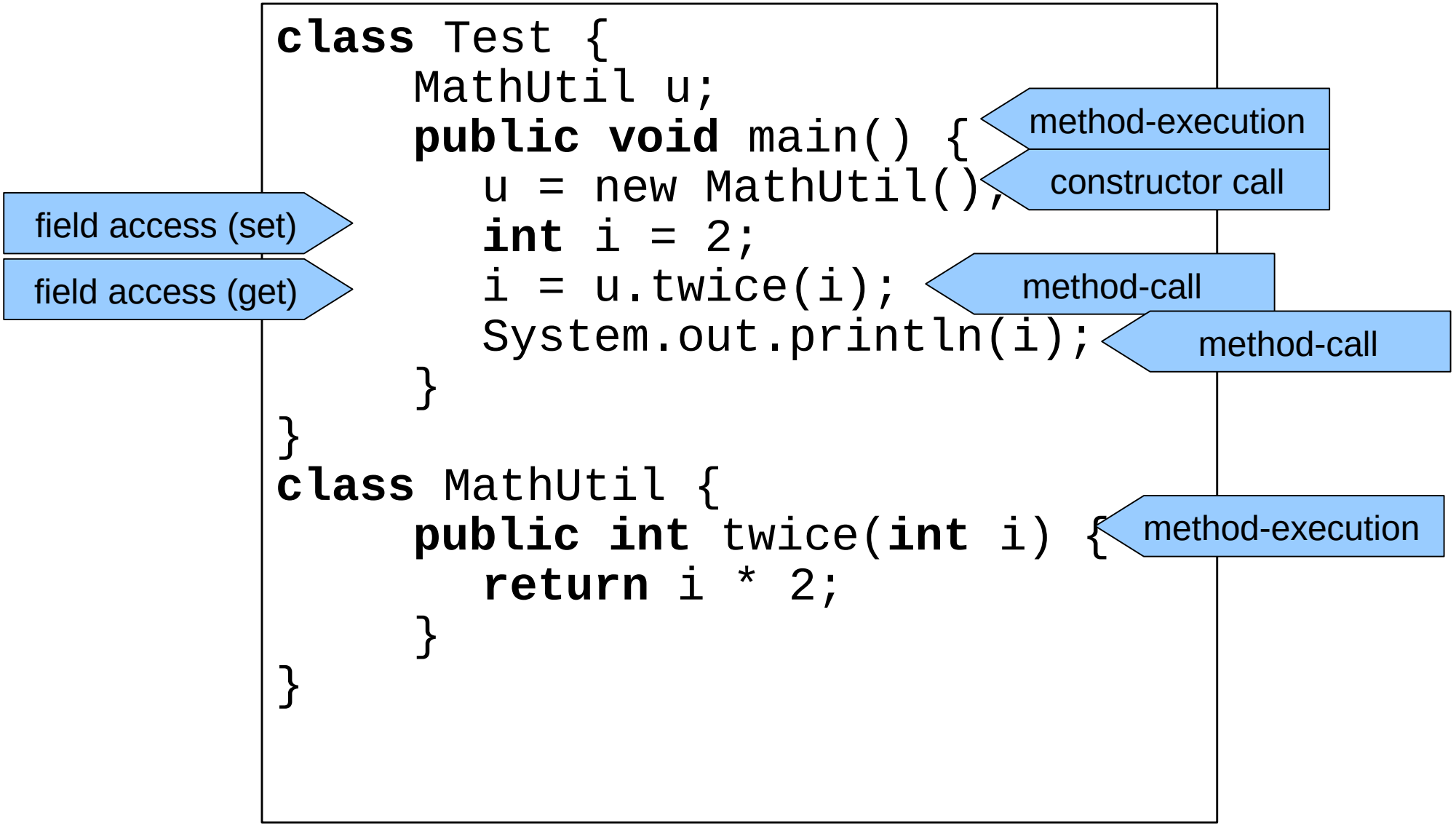
Quantifikation

- Eine wichtige Eigenschaft von Pointcuts ist, dass sie **deklarativ** Join-Points quantifizieren und somit auf mehrere Join-Points matchen können
- Beispiele:
  - Führe Advice X immer aus, wenn die Methode „setWeight“ in Klasse „Edge“ aufgerufen wird
  - Führe Advice Y immer aus, wenn auf irgendein Feld in der Klasse „Edge“ zugegriffen wird
  - Führe Advice Z immer aus, wenn irgendwo im System eine öffentliche Methode aufgerufen wird, und vorher die Methode „initialize“ aufgerufen wurde

AspectJ – Join Point Model

- Join Points treten auf bei(m)
  - Aufruf einer Methode
  - Ausführung einer Methode
  - Aufruf eines Konstruktors
  - Ausführung eines Konstruktors
  - Zugriff auf ein Feld (lesend oder schreibend)
  - Fangen einer Exception
  - Initialisierung einer Klasse oder eines Objektes
  - Ausführen von Advice

Beispiel: Join Points



Pointcut Execution

- Erfasst die Ausführung einer Methode

```
aspect A1 {
    after() : execution(int MathUtil.twice(int)) {
        System.out.println("MathUtil.twice executed");
    }
}
```

```
class Test {
        public static void main(String[] args)
{
            MathUtil u = new MathUtil();
            int i = 2;
            i = u.twice(i);
            System.out.println(i);
        }
}
class MathUtil {
        public int twice(int i) {
            return i * 2;
        }
}
```

Ausführung

Syntax:
execution(ReturnType ClassName.Methodname(ParameterTypes))

Explicit / Anonymous Pointcuts

```
aspect A1 {
    after() : execution(int MathUtil.twice(int)) {
        System.out.println("MathUtil.twice executed");
    }
}
```

```
aspect A2 {
    pointcut executeTwice() : execution(int MathUtil.twice(int));
    after() : executeTwice() {
        System.out.println("MathUtil.twice executed");
    }
}
```

Advice

- Zusätzlicher Code vor (`before`), nach (`after`) oder anstelle (`around`) des Join-Points
- Beim „`around`-Advice“ ist es möglich, den originalen Join-Point mittels des Schlüsselworts „`proceed`“ fortzusetzen

Beispiel: Advice

```
public class Test2 {
    void foo() {
        System.out.println("foo() executed");
    }
}
aspect AdviceTest {
    before(): execution(void Test2.foo()) {
        System.out.println("before foo()");
    }
    after(): execution(void Test2.foo()) {
        System.out.println("after foo()");
    }
    void around(): execution(void Test2.foo()) {
        System.out.println("around begin");
        proceed();
        System.out.println("around end");
    }
    after() returning (): execution(void Test2.foo()) {
        System.out.println("after returning from foo()");
    }
    after() throwing (RuntimeException e): execution(void Test2.foo()) {
        System.out.println("after foo() throwing "+e);
    }
}
```

thisJoinPoint

- In Advice kann „`thisJoinPoint`“ verwendet werden, um Informationen über den aktuellen Join Point zu ermitteln

```
aspect A1 {
    after() : call(int MathUtil.twice(int)) {
        System.out.println(thisJoinPoint);
        System.out.println(thisJoinPoint.getSignature());
        System.out.println(thisJoinPoint.getKind());
        System.out.println(thisJoinPoint.getSourceLocation());
    }
}
```

```
Ausgabe:
call(int MathUtil.twice(int))
int MathUtil.twice(int)
method-call
Test.java:5
```

Pattern

- Erlaubt „unvollständige“ Angaben bei der Quantifizierung

```
aspect Execution {
    pointcut P1() : execution(int MathUtil.twice(int));

    pointcut P2() : execution(* MathUtil.twice(int));

    pointcut P3() : execution(int MathUtil.twice(*));

    pointcut P4() : execution(int MathUtil.twice(..));

    pointcut P5() : execution(int MathUtil.*(int, ..));

    pointcut P6() : execution(int *Util.tw*(int));

    pointcut P7() : execution(int *.twice(int));

    pointcut P8() : execution(int MathUtil+.twice(int));

    pointcut P9() : execution(public int package.MathUtil.twice(int)
    throws ValueNotSupportedException);

    pointcut Ptypisch() : execution(* MathUtil.twice(..));
}
```

***** als Platzhalter für einen Wert

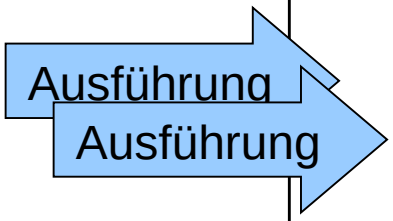
.. als Platzhalter für mehre Werte

\+ für Subklassen

Pointcut Call

- Erfasst den Aufruf einer Methode
- Ähnlich zu execution, aber auf Aufruferseite

```
aspect A1 {
    after() : call(int MathUtil.twice(int)) {
        System.out.println("MathUtil.twice called");
    }
}
```

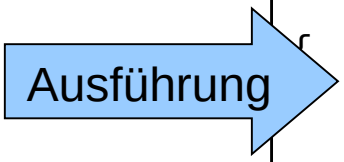



```
class Test {
    public static void main(String[] args) {
        MathUtil u = new MathUtil();
        int i = 2;
        i = u.twice(i);
        i = u.twice(i);
        System.out.println(i);
    }
}
class MathUtil {
    public int twice(int i) {
        return i * 2;
    }
}
```

Constructor Calls

- „new“ als spezielle Methode

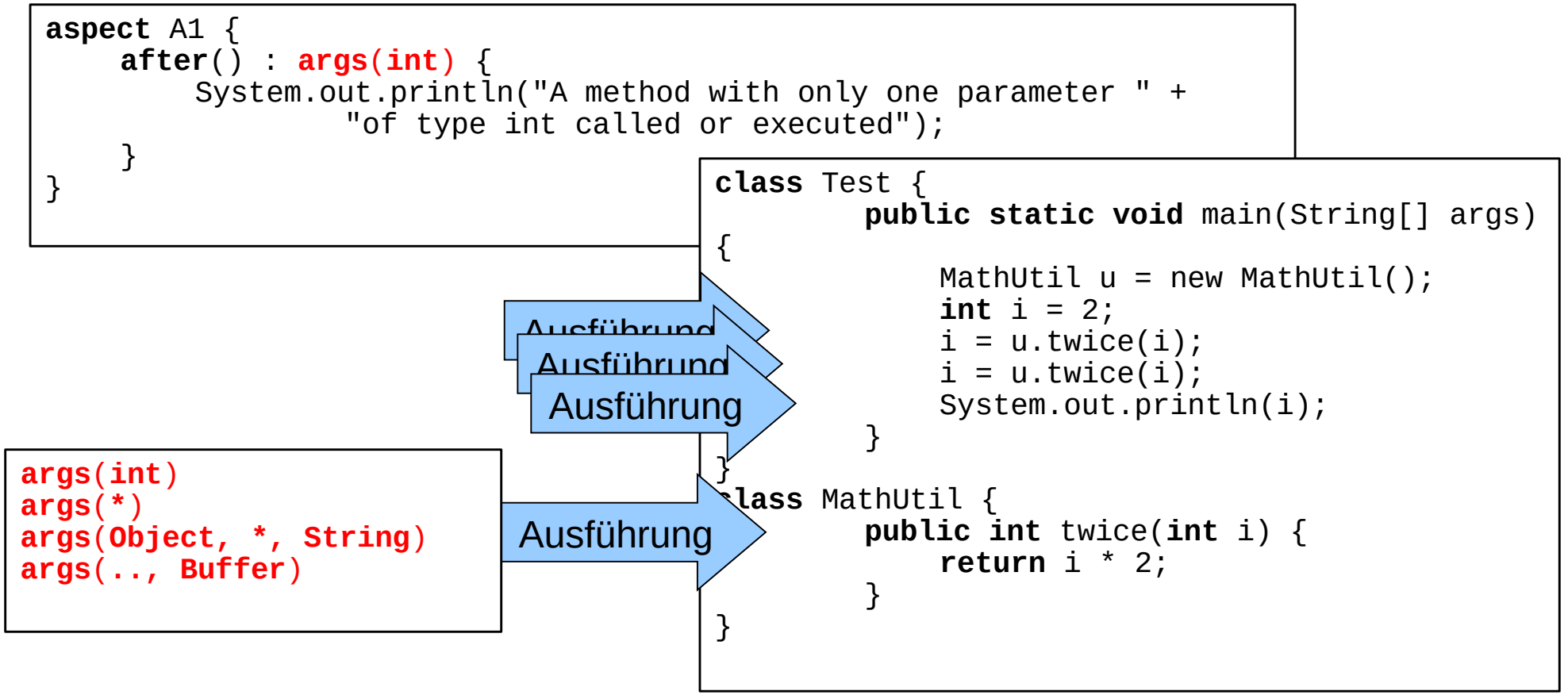
```
aspect A1 {
    after() : call(MathUtil.new()) {
        System.out.println("MathUtil created");
    }
}
```




```
class Test {
        public static void main(String[] args)
{
            MathUtil u = new MathUtil();
            int i = 2;
            i = u.twice(i);
            i = u.twice(i);
            System.out.println(i);
        }
}
class MathUtil {
        public int twice(int i) {
            return i * 2;
        }
}
```

Pointcut Argumente

- Matched nur die Parameter einer Methode
- Ähnlich zu execution(* *.*(X, Y)) oder call(* *.*(X, Y))



Kombination von Pointcuts

- Pointcuts können mit &&, || und ! verbunden werden

```
aspect A1 {
    pointcut P1(): execution(* Test.main(..)) || call(* MathUtil.twice(*));
    pointcut P2(): call(* MathUtil.*(..)) && !call(* MathUtil.twice(*));
    pointcut P3(): execution(* MathUtil.twice(..)) && args(int);
}
```

Parametrisierte Pointcuts

- Pointcuts können Parameter haben, die im Advice genutzt werden können
- Damit erhält der Advice Informationen zum Kontext
- Der Pointcut `args` wird dazu mit einer Variablen, statt mit einem Typ, verwendet

```
aspect A1 {
    pointcut execTwice(int value) :
                execution(int MathUtil.twice(int)) && args(value);
    after(int value) : execTwice(value) {
        System.out.println("MathUtil.twice executed with parameter " + value);
    }
}
```

```
aspect A1 {
    after(int value) : execution(int MathUtil.twice(int)) && args(value) {
        System.out.println("MathUtil.twice executed with parameter " + value);
    }
}
```

Advice mit Parametern

- Beispiel für Advice der Parameter nutzt und ändert:

```
aspect DoubleWeight {
    pointcut setWeight(int weight) :
        execution(void Edge.setWeight(int)) && args(weight);

    void around(int weight) : setWeight(weight) {
        System.out.print('doubling weight from ' + weight);
        try {
          proceed(2 * weight);
        } finally {
          System.out.print('doubled weight from ' + weight);
        }
    }
}
```

Pointcuts this / target

- `this` und `target` erfassen die involvierten Klassen
- Können mit Typen (inkl. Muster) und mit Parametern genutzt werden

```
aspect A1 {
    pointcut P1(): execution(int *.twice(int)) && this(MathUtil);
    pointcut P2(MathUtil m) : execution(int MathUtil.twice(int)) && this(m);
    pointcut P3(Main source, MathUtil target): call(* MathUtil.twice(*)) &&
                              this(source) && target(target);
}
```

- Bei execution: `this` und `target` erfassen das Objekt auf dem die Methode aufgerufen wird
- Bei call, set und get: this erfasst das Objekt, das die Methode aufruft oder auf das Feld zugreift; target erfasst das Objekt, auf dem die Methode aufgerufen oder auf das Feld zugegriffen wird

Pointcuts within / withincode

- Schränken Join-Points nach Ort des Vorkommens ein
- Beispiel: nur Aufrufe zur twice Methode, die aus Test bzw. Test.main kommen

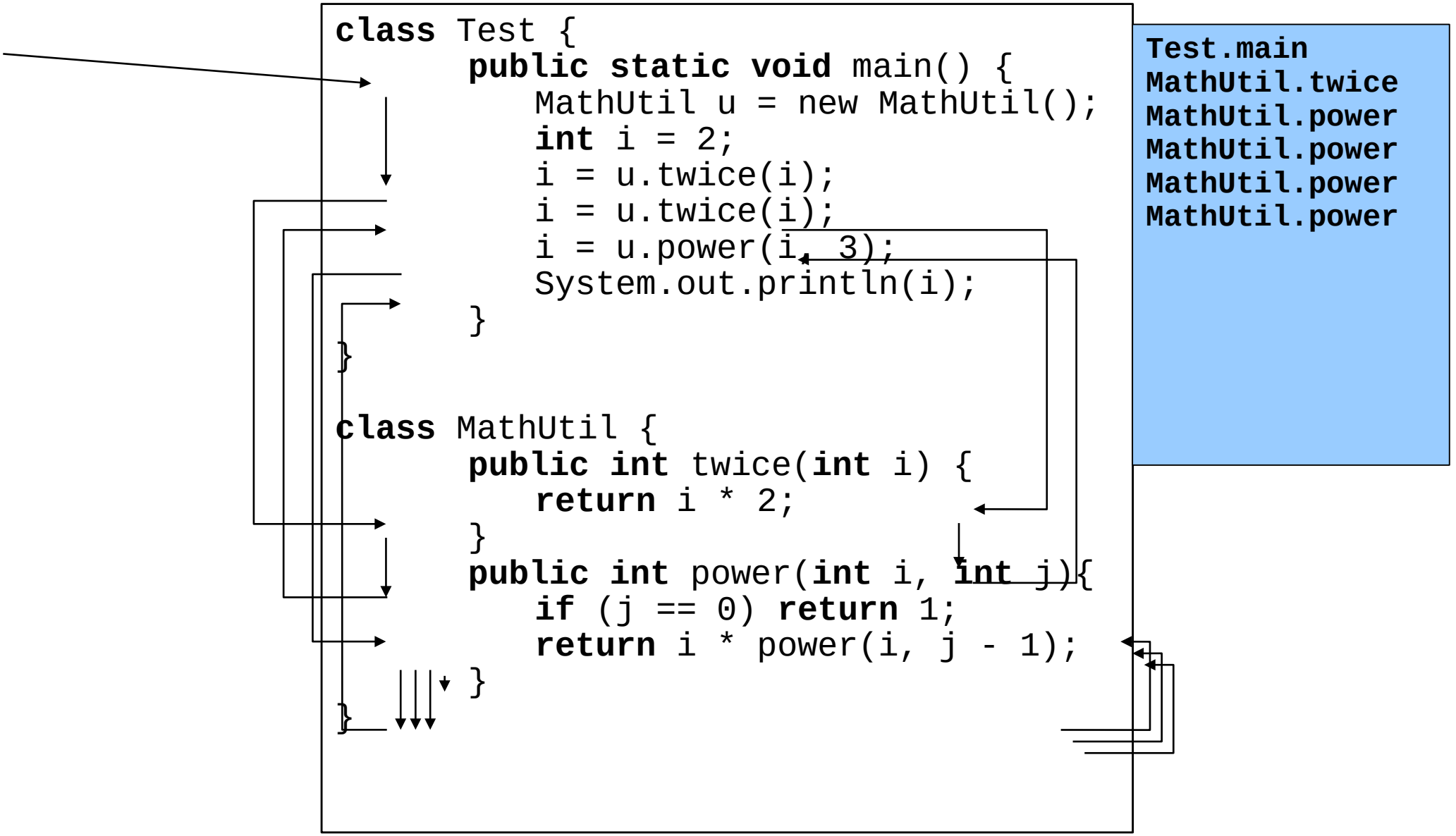
```
aspect A1 {
    pointcut P1(): call(int MathUtil.twice(int)) && within(Test);
    pointcut P2(): call(int MathUtil.twice(int)) && withincode(* Test.main(..));
}
```

Pointcuts cflow / cflowbelow

- Erfasst alle Join-Points. die im Kontrollfluss eines anderen Join-Points stattfinden

```
aspect A1 {
    pointcut P1(): cflow(execution(int MathUtil.twice(int)));
    pointcut P2(): cflowbelow(execution(int MathUtil.twice(int)));
}
```

Contol Flow


Stack:
Test.main
MathUtil.twice
MathUtil.power
MathUtil.power
MathUtil.power
MathUtil.power
class Test {
public static void main() {
MathUtil u = new MathUtil();
int i = 2;
i = u.twice(i);
i = u.twice(i);
i = u.power(i, 3);
System.out.println(i);
}
}
class MathUtil {
public int twice(int i) {
return i * 2;
}
public int power(int i, int j){
if (j == 0) return 1;
return i * power(i, j - 1);
}
}

Beispiel: Control Flow

```
before() :
execution(* *.*(..))
```

```
execution(void Test.main(String[]))
execution(int MathUtil.twice(int))
execution(int MathUtil.twice(int))
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
```

```
execution(* *.*(..)) &&
 cflow(execution(* *.power(..)))
```

```
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
```

```
execution(* *.*(..)) &&
 cflowbelow(execution(* *.power(..)))
```

```
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
execution(int MathUtil.power(int, int))
```

```
execution(* *.power(..)) &&
 !cflowbelow(execution(* *.power(..)))
```

```
execution(int MathUtil.power(int, int))
```

Weaving

- Wie werden Aspekte ausgeführt?
  - Meta-Objekt-Protokolle und interpretierte Sprachen: zur Laufzeit ausgewertet
  - AspectJ/AspectC++/…: “einweben” des Aspektes durch den Compiler
- Einweben:
  - Inter-Typ-Deklarationen werden in die entsprechenden Klassen eingefügt
  - Advice wird in Methoden umgewandelt
  - Pointcuts: Methodenaufruf von den Join-Points zum Advice hinzugefügen
  - Dynamische Erweiterungen: an allen potentiellen Join-Points Quelltext einfügen, der dynamische Bedingung prüft und ggf. Methodenaufruf zum Advice ausführt

Beispiel: Graph

Basic Graph

```
class Graph {
  Vector nv = new Vector();
  Vector ev = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m);
    ev.add(e); return e;
  }
  void print() {
    for(int i = 0; i < ev.size(); i++)
      ((Edge)ev.get(i)).print();
  }
}
```

```
class Edge {
  Node a, b;
  Edge(Node _a, Node _b) {
    a = _a; b = _b;
  }
  void print() {
    a.print(); b.print();
  }
}
```

```
class Node {
  int id = 0;
  void print() {
    System.out.print(id);
  }
}
```

Color

```
aspect ColorAspect {
  Color Node.color = new Color();
  Color Edge.color = new Color();
  before(Node c) : execution(void print()) && this(c) {
    Color.setDisplayColor(c.color);
  }
  before(Edge c) : execution(void print()) && this(c) {
    Color.setDisplayColor(c.color);
  }
  static class Color { ... }
}
```

Beispiel: Graph

Basic Graph

```
class Graph {
  Vector nv = new Vector();
  Vector ev = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m);
    ev.add(e); return e;
  }
  void print() {
    for(int i = 0; i < ev.size(); i++)
      ((Edge)ev.get(i)).print();
  }
}
```

```
class Edge {
  Node a, b;
  Edge(Node _a, Node _b) {
    a = _a; b = _b;
  }
  void print() {
    a.print(); b.print();
  }
}
```

```
class Node {
  int id = 0;
  void print() {
    System.out.print(id);
  }
}
```

Color

```
aspect ColorAspect {
  static class Colored { Color color; }
  declare parents: (Node || Edge) extends Colored;
  before(Colored c) : execution(void print()) && this(c) {
    Color.setDisplayColor(c.color);
  }
  static class Color { ... }
}
```

Aspekte: Verwendung

- Logging, Tracing, Profiling
  - Fügt identischen Code zu sehr vielen Methoden hinzu

```
aspect Profiler {
    /** record time to execute my public methods */
    Object around() : execution(public * com.company..*.* (..)) {
        long start = System.currentTimeMillis();
        try {
            return proceed();
        } finally {
            long end = System.currentTimeMillis();
            printDuration(start, end,
                thisJoinPoint.getSignature());
        }
    }
    // implement recordTime...
}
```

Aspekte: Verwendung

- Caching, Pooling
  - Cache oder Ressourcenpool zentral implementieren, der an mehreren Stellen wirken kann

```
aspect ConnectionPooling {
    ...
    Connection around() : call(Connection.new()) {
        if (enablePooling)
            if (!connectionPool.isEmpty())
                return connectionPool.remove(0);
        return proceed();
    }
    void around(Connection conn) :
         call(void Connection.close()) && target(conn) {
        if (enablePooling) {
            connectionPool.put(conn);
        } else {
            proceed();
        }
    }
}
```

Aspekte: Verwendung

- Observer
  - Verschiedene Ereignisse sammeln
  - Auf verschachtelte Ereignisse nur einmal reagieren (cflowbelow)

```
abstract class Shape {
    abstract void moveBy(int x, int y);
}
class Point extends Shape { ... }
class Line extends Shape {
    Point start, end;
    void moveBy(int x, int y) { start.moveBy(x,y); end.moveBy(x,y); }
}

aspect DisplayUpdate {
    pointcut shapeChanged() : execution(void Shape+.moveBy(..));

    after() : shapeChanged() && !cflowbelow(shapeChanged()) {
        Display.update();
    }
}
```

Aspekte: Verwendung

- Policy Enforcement
  - Policy wird extern implementiert
  - Beispiel: Autosave alle 5 Aktionen

```
aspect Autosave {
    int count = 0;
    after(): call(* Command+.execute(..)) {
        count++;
    }
    after(): call(* Application.save())
         || call(* Application.autosave()) {
        count = 0;
    }
    before(): call (* Command+.execute(..)) {
        if (count > 4) Application.autosave();
    }
}
```

Aspect Methods

- Aspekte können wie normale Klassen auch Methoden und Felder enthalten; diese können mit Inter-Typ-Deklarationen, Pointcuts und Advice gemischt werden
- Advice wird im Kontext des Aspekts ausgeführt, nicht im Kontext der erweiterten Klasse („third person perspective“)

```
aspect Logging {
    PrintStream loggingTarget = System.out;
    private void log(String logStr) {
        loggingTarget.println(logStr);
    }
    pointcut anySetMethodCall(Object o) :
                call(public *.set*(..)) && target(o);
    after(Object o) : anySetMethodCall(o) {
        log('A public method was called on '+ o.getClass().getName());
    }
}
```

Aspekthierarchie

- Engl. aspect precendence
- Wenn nicht explizit definiert, ist die Reihenfolge, in der Aspekte gewebt werden, undefiniert
- Wenn mehrere Aspekte den gleichen Join-Point erweitern kann die Reihenfolge aber relevant sein
  - Beispiel: Erster Aspekt implementiert Synchronisierung mit around-Advice, zweiter Aspekt implementiert Logging mit after-Advice auf dem gleichen Join-Point; Je nach Reihenfolge des Webens wird der Logging-Code synchronisiert oder nicht

Aspekthierarchie

- Möglichkeit explizit eine Rangfolge festzulegen mittels declare precedence

```
aspect DoubleWeight {
    declare precedence : *, Weight, DoubleWeight;
    ...
}
```

- Aspekt mit höchster Priorität wird zuerst gewebt, d. h. bei before wird der Advice aus diesem Aspekt zuerst ausgeführt, bei after zuletzt, bei around als äußerstes

Advice-Hierarchie

- Falls mehrere Advice-Statements in dem gleichen Aspekt einen Join-Point erweitern, ist die Reihenfolge durch die Anordnung deren Definition in dem Aspekt festgelegt

Abstract Aspects

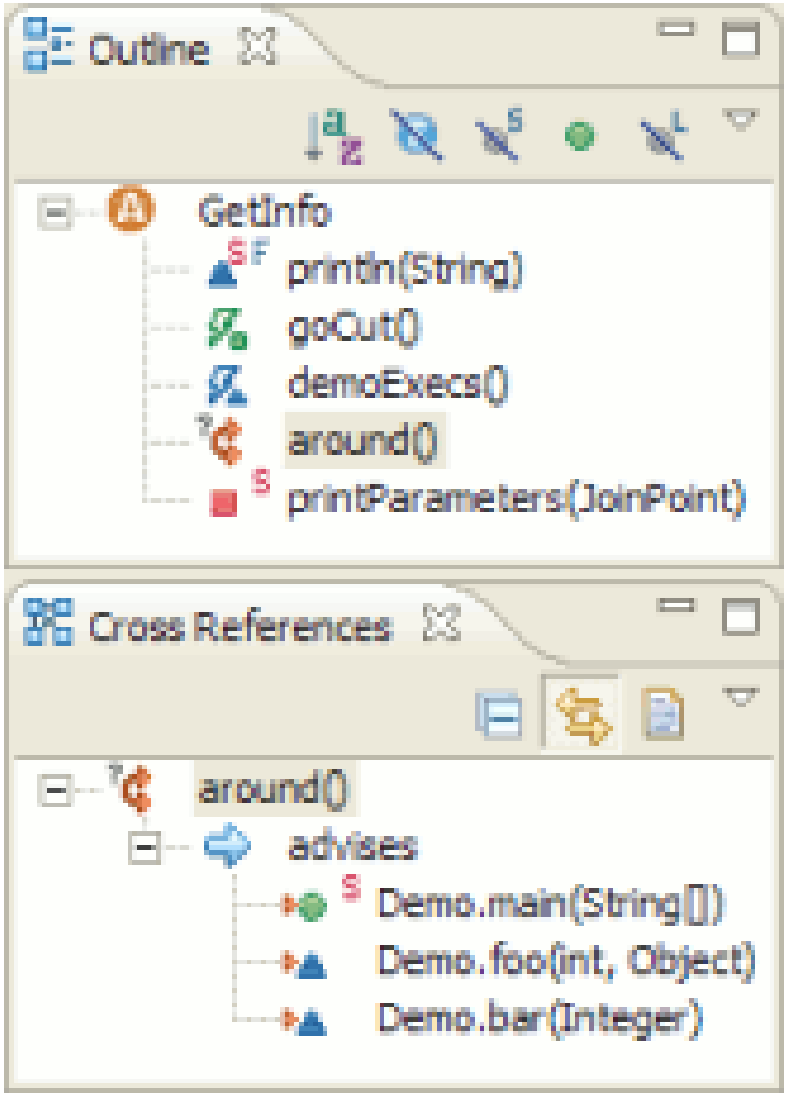
- Aspekte unterstützen eine einfache Form der Vererbung, um sie wiederverwendbar zu machen
- Wie Klassen können Aspekte als `abstract` deklariert werden.
- Abstrakte Aspekte können abstrakte Pointcuts enthalten, die erst in einem Sub-Aspekt definiert werden.
- Aspekte können nur von abstrakten Aspekten erben, nicht von konkreten Aspekten
- Überschreiben von Pointcuts ist möglich, überschreiben von Advice aber nicht, weil Advice anonym ist

Beispiel: Aspect Reuse

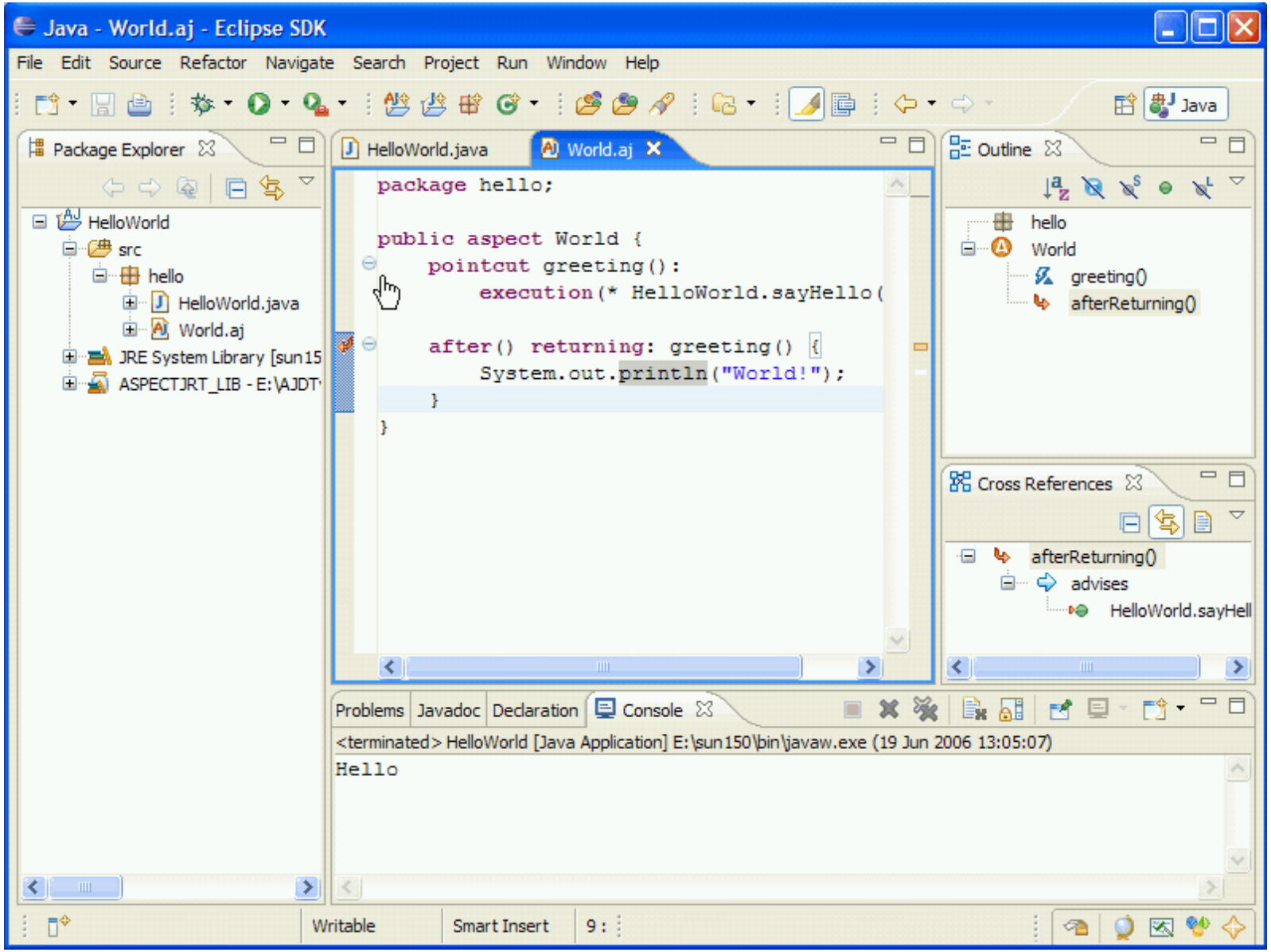
```
abstract aspect AbstractLogger {
    abstract pointcut loggingPointcut();

    PrintStream loggingTarget = System.out;
    protected void log(String logStr) {
        loggingTarget.println(logStr);
    }
    after() : loggingPointcut() {
        log('Join point reached ' + thisJoinPoint);
    }
}

aspect PrintLogger extends AbstractLogger {
    pointcut loggingPointcut() : execution(* print*(..));
}

aspect SetLogger extends AbstractLogger {
    pointcut loggingPointcut() : execution(* set*(..));

    protected void log(String logStr) {
        super.log('Set Method: ' + logStr);
    }
}
```

AJDT

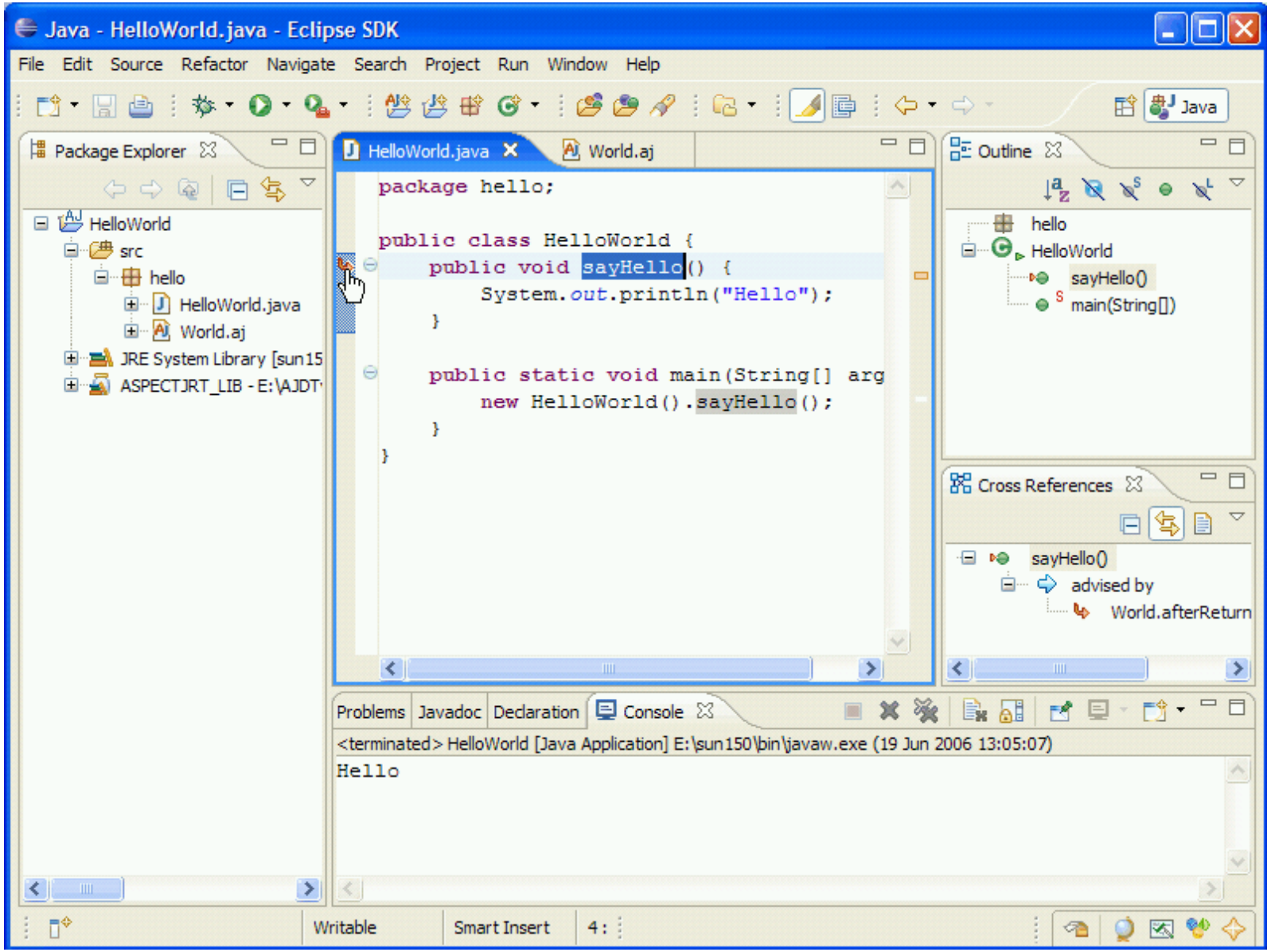
- Eclipse-Plugin für AspectJ Entwicklung
  - Integriert Aspekte in Eclipse, wie Java in JDT
  - Compiler und Debugger Integration
  - Syntax Highlighting, Outline
  - Links zwischen Aspekten und erweiterten Stellen (zeigt an wo der Quelltext von Aspekten erweitert wird)



AJDT


Java - World.aj - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Package Explorer
HelloWorld
src
hello
HelloWorld.java
World.aj
JRE System Library [sun15
ASPECTJRT_LIB - E:\AJDT
HelloWorld.java
World.aj
package hello;
public aspect World {
pointcut greeting():
execution(* HelloWorld.sayHello(
after() returning: greeting() {
System.out.println("World!");
}
}
Outline
hello
World
greeting()
afterReturning()
Cross References
afterReturning()
advises
HelloWorld.sayHell
Problems Javadoc Declaration Console
<terminated> HelloWorld [Java Application] E:\sun150\bin\javaw.exe (19 Jun 2006 13:05:07)
Hello
Writable
Smart Insert
9 :

AJDT


Java - HelloWorld.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Package Explorer
HelloWorld
src
hello
HelloWorld.java
World.aj
JRE System Library [sun15
ASPECTJRT_LIB - E:\AJDT
HelloWorld.java
World.aj
package hello;
public class HelloWorld {
public void sayHello() {
System.out.println("Hello");
}
public static void main(String[] arg
new HelloWorld().sayHello();
}
}
Outline
hello
HelloWorld
sayHello()
main(String[])
Cross References
sayHello()
advised by
World.afterReturn
Problems Javadoc Declaration Console
<terminated> HelloWorld [Java Application] E:\sun150\bin\javaw.exe (19 Jun 2006 13:05:07)
Hello
Writable
Smart Insert
4 :

Obliviousness

- Unbewusstheitsprinzip (obliviousness) besagt, dass das Basisprogramm nichts von den Aspekten wissen muss: „Programmiere einfach Java wie immer, wir fügen die Aspekte später hinzu“
- Bedeutet, dass…
  - klassisches OO-Design reicht, der Quelltext muss nicht für Aspekte vorbereitet werden („schreibe einfach die Datenbank ganz normal, wir fügen Transaktionen später hinzu“)
  - Basisentwickler brauchen keine Kenntnisse über Aspekte, wenige Spezialisten reichen
  - die AOP-Sprache sehr mächtig sein muss

Obliviousness

- Unbewusstheitsprinzip wurde neben Quantifizierung als zweites Merkmal von AOP vorgeschlagen
- Sehr kontrovers, weil...
  - ... Programmierer den Basisquelltext ändern können, ohne Aspekte wahrzunehmen und anzupassen („Fragile Pointcut Problem“; explizit keine Schnittstellen)
  - ... teils sehr schlechtes Design die Konsequenz ist, wenn die Aspekte nicht beim Entwurf berücksichtigt werden, sondern später „reingehackt“ werden
  - ... dann typischerweise komplexe Sprachmittel wie cflow oder call && withincode verwendet werden, um Erweiterungen in unvorbereitetem Quelltext dennoch auszudrücken

Warum Aspekte?

- Erlauben **kohäsive** Implementierung von querschneidenden Belangen
- Erlauben deklarativ über viele Join-Points zu **quantifizieren** (homogene Erweiterungen an vielen Punkten im Programm)
- Erlauben Analysen **dynamischer** Eigenschaften wie des Kontrollfluss (cflow), die in OOP erhebliche Workarounds benötigen

Features als Aspekte

- Aspekte sind sehr ähnlich zu Kollaborationen
  - Können Code statisch einfügen
  - Können Methoden erweitern
  - Können darüber hinaus sogar homogene Erweiterungen und Erweiterungen auf Basis des dynamischen Kontrollfluss

- Aspekte können ein und ausgeschaltet werden, indem man sie mit dem Programm kompiliert, oder nicht
  - Manuell in Eclipse: Rechtsklick auf Aspekt und „Exclude from Buildpath“
  - Automatisiert mit Buildsystem
  - FeatureIDE in Kombination mit AJDT

Features als Aspekte

- Pro Feature ein Aspekt?
  - Aspekte können sehr groß und unübersichtlich werden
  - Aspekte können neue Klassen nur als statische innere Klassen einfügen

- Daher: Bündelung von mehreren Aspekten und Klassen in Feature-Modulen

  ➔ Mischung von Kollaborationen und Aspekten: „Aspectual Feature Modules“

Join-Point auswählen

- Pointcuts wählen Join-Points aufgrund von Namensvergleichen aus, obwohl Methodennamen eigentlich frei gewählt werden können
- Musterausdrücke nutzen Namenskonventionen aus, z. B. „get*“, „draw*“ usw.

```
class Chess {
    void drawKing() {...}
    void drawQueen() {...}
    void drawKnight() {...}
}

aspect UpdateDisplay {
    pointcut drawn : execution(* draw*(..));
    ...
}
```

Fragile Pointcut Problem / Evolution Paradox

- Wenn der Basiscode geändert wird, kann es passieren, dass neue Join-Points von einem existierenden Pointcut erfasst werden, oder bisherige Join-Points nicht mehr erfasst werden
- Schachbeispiel: Ein Entwickler, der den Aspekt nicht kennt, fügt eine Methode für Unentschieden hinzu „void draw()“
- Solche Änderungen im Programmverhalten können unbemerkt stattfinden, es ist nicht möglich herauszukriegen, ob die „korrekten“ Pointcuts erfasst wurden

draw = zeichnen

draw = Unentschieden

Komplexe Syntax

- AspectJ ist sehr mächtig und bietet viele Ausdrucksmöglichkeiten mit vielen Sprachkonstrukten
- Die Sprache wird dadurch komplex, insbesondere einfache Erweiterungen sind aufwendig
  z. B. einfache Methodenerweiterungen:

OOP / FOP

```
public void delete(Transaction txn, DbEntry key) {
    super.delete(txn, key);
    Tracer.trace(Level.FINE, "Db.delete", this, txn, key);
}
```

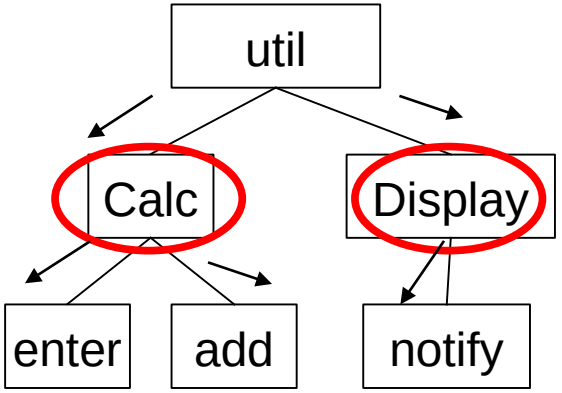
AOP

```
pointcut traceDel(Database db, Transaction txn, DbEntry key) :
    execution(void Database.delete(Transaction, DbEntry))
    && args(txn, key) && within(Database) && this(db);
after(Database db, Transaction txn, DbEntry key): traceDel(db, txn, key) {
    Tracer.trace(Level.FINE, "Db.delete", db, txn, key);
}
```

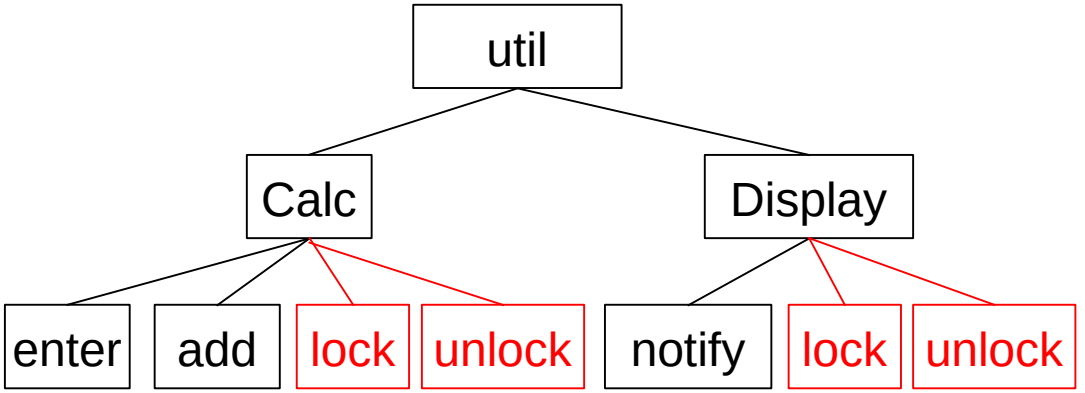
Quantifikation und Modifikation

- Eine **Modifikation** durchläuft einen FST und wählt Knoten aus, an denen später Änderungen vorgenommen werden

Schritt 1: Auswahl



Schritt 2: Änderung der ausgewählten Stellen



Bsp.: Beide Klassen sollen um die Methoden **`lock`** und **`unlock`** erweitert werden

AOP vs. FOP

- AOP und FOP implizieren keine konkreten Implementierungstechniken
- Unterscheiden sich in ihrer Philosophie
  - AOP fokussiert auf querschneidende Belange
  - FOP fokussiert auf Domänenabstraktionen
- Dennoch werden oft konkrete Implementierungs-techniken mit beiden Ansätzen in Verbindung gebracht
  - AOP ➔ Pointcuts & Advice, Inter-Typ-Deklarationen
  - FOP ➔ Klassen, Refinements, Mixin/Jampack-Komposition

Feature Module in Jak

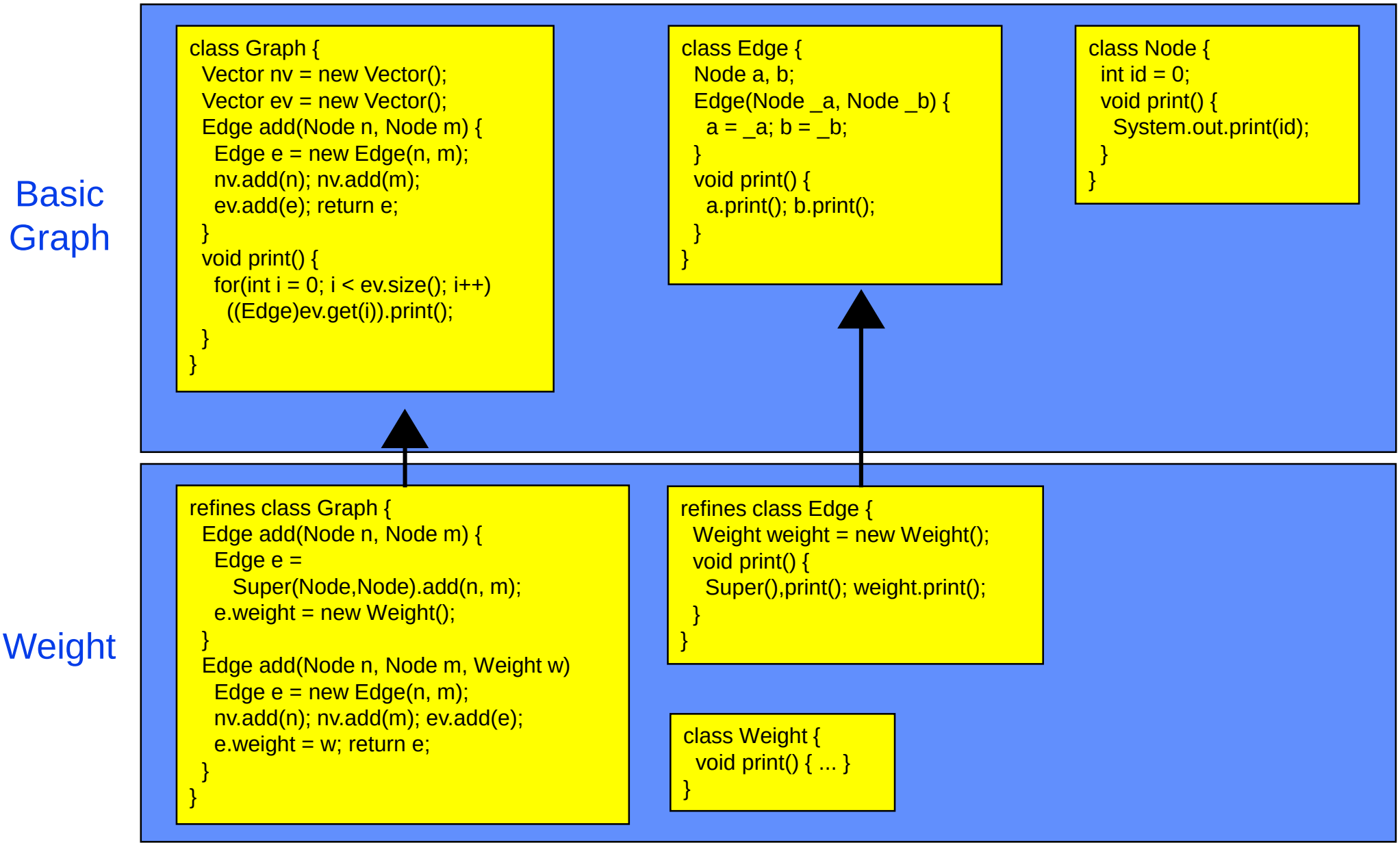
Basic Graph

```
class Graph {
  Vector nv = new Vector();
  Vector ev = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m);
    ev.add(e); return e;
  }
  void print() {
    for(int i = 0; i < ev.size(); i++)
      ((Edge)ev.get(i)).print();
  }
}
```

```
class Edge {
  Node a, b;
  Edge(Node _a, Node _b) {
    a = _a; b = _b;
  }
  void print() {
    a.print(); b.print();
  }
}
```

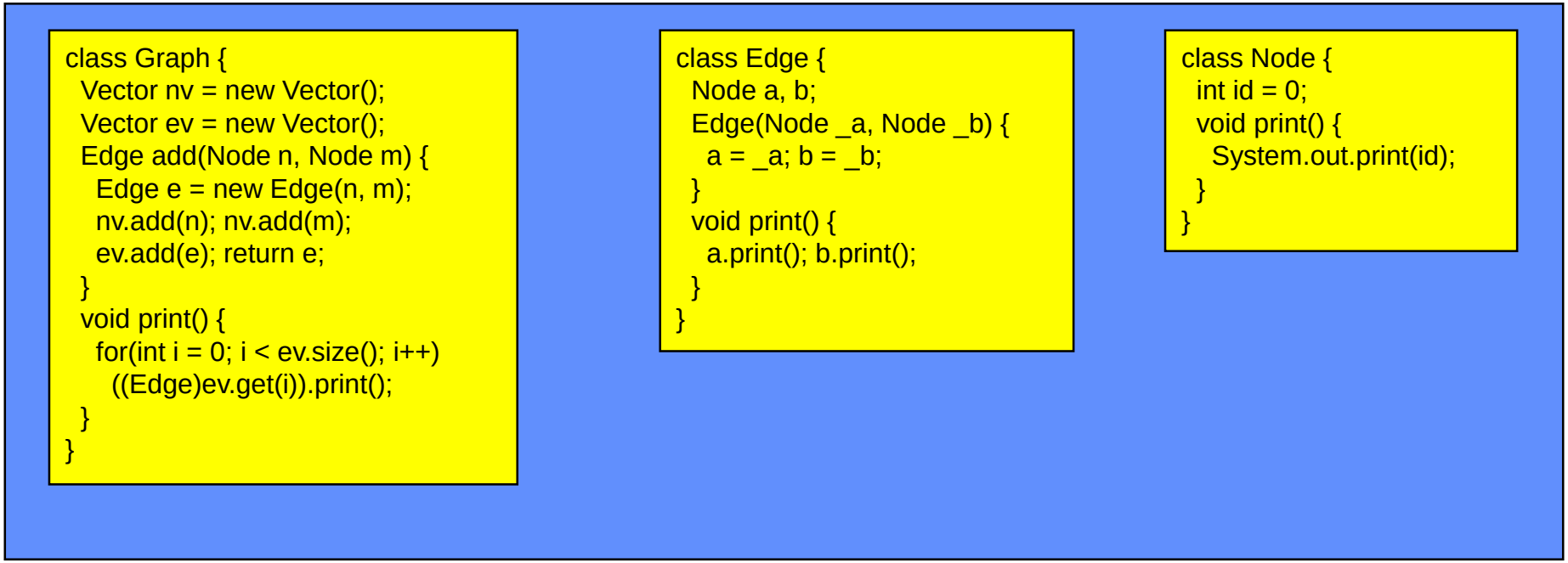
```
class Node {
  int id = 0;
  void print() {
    System.out.print(id);
  }
}
```

Feature Module in Jak



Aspects in AspectJ

Basic Graph



```
class Graph {
  Vector nv = new Vector();
  Vector ev = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m);
    ev.add(e); return e;
  }
  void print() {
    for(int i = 0; i < ev.size(); i++)
      ((Edge)ev.get(i)).print();
  }
}
```

```
class Edge {
  Node a, b;
  Edge(Node _a, Node _b) {
    a = _a; b = _b;
  }
  void print() {
    a.print(); b.print();
  }
}
```

```
class Node {
  int id = 0;
  void print() {
    System.out.print(id);
  }
}
```

Aspects in AspectJ

Basic Graph

```
class Graph {
  Vector nv = new Vector();
  Vector ev = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m);
    ev.add(e); return e;
  }
  void print() {
    for(int i = 0; i < ev.size(); i++)
      ((Edge)ev.get(i)).print();
  }
}
```

```
class Edge {
  Node a, b;
  Edge(Node _a, Node _b) {
    a = _a; b = _b;
  }
  void print() {
    a.print(); b.print();
  }
}
```

```
class Node {
  int id = 0;
  void print() {
    System.out.print(id);
  }
}
```

Color

```
aspect ColorAspect {
  static class Colored { Color color; }
  declare parents: (Node || Edge) extends Colored;
  before(Colored c) : execution(void print()) && this(c) {
    Color.setDisplayColor(c.color);
  }
  static class Color { ... }
}
```

Terminologie

	Analyse, Design	Implementierung
AOP	☐Aspekt-Orientierte Modellierung ☐Frühe Aspekte (*early aspects*) ☐Aspekte als Funktionen	☐Aspekt = Klasse, Advice, Inter-Typ-Deklarationen ☐AspectJ, AspectC++, Eos
FOP	☐Feature-Modellierung ☐Feature-Ausdrücke ☐Quarks und Co. ☐Domänen-spezifische Optimierung	☐Feature-Modul = Klassen, Refinements, Mixin/Jampack-Komposition ☐Jak, FeatureC++, FeatureHouse, Classbox/J, Jiazzi, ObjectTeams/Java

Hetrogene vs. Homogene Extensions

- Heterogen: unterschiedlicher Code an unterschiedlichen Stellen

```
class Graph {  …
   Edge add(Node n, Node m) {
   Edge e = new Edge(n, m);
   nv.add(n); nv.add(m); ev.add(e);
   e.weight = new Weight(); return e;
  }
  Edge add(Node n, Node m, Weight w)
   Edge e = new Edge(n, m);
   nv.add(n); nv.add(m); ev.add(e);
   e.weight = w; return e;
  }  …
}
```

```
class Edge {  …
  Weight weight = new Weight();
  Edge(Node _a, Node _b) { a = _a; b = _b; }
   void print() {
    a.print(); b.print(); weight.print();
  }
}
```

- Homogen: gleicher Code an unterschiedlichen Stellen

```
class Node {
  int id = 0;
  Color color = new Color();
  void print() {
    Color.setDisplayColor(color);
    System.out.print(id);
  }
}
```

```
class Edge {
  Node a, b;
  Color color = new Color();
  Edge(Node _a, Node _b) { a = _a; b = _b; }
  void print() {
    Color.setDisplayColor(color);
    a.print(); b.print();
  }
}
```

Static vs. Dynamic Extensions

- Statisch:
  verändern die statische Struktur (syntaktische Struktur)

```
class Node {
  int id = 0;
  Color color = new Color();
  void print() {
    System.out.print(id);
  }
}
```

- Dynamisch:
  ändern das Verhalten (Ereignis und Aktion)

```
class Node {
  int id = 0;
  void print() {
    Color.setDisplayColor(color);
    System.out.print(id);
  }
}
```

Simple und komplexe Dynamic Extensions

- **Einfache** dynamische Erweiterungen
  - Erweitern von Methodenausführungen
  - Ohne Bedingungen zur Laufzeit
  - Kein Zugriff auf den Kontext des Ereignisses
    - Außer Argumente, Rückgabewert und Objekt
- **Komplexe** dynamische Erweiterungen
  - Alle Arten von Ereignissen
  - Bedingungen zur Laufzeit (Kontrollfluss)
  - Zugriff auf den dynamischen Kontext

Hinweis: Einfache dynamische Erweiterungen sind Methodenerweiterungen mittels Overriding!

Beispiel: Dynamic Extension

```
class Edge {
  int weight = 0;
  void setWeight(int w) { weight = w; }
  int getWeight() { return weight; }
}
```

Jak

```
refines class Edge {
  void setWeight(int w) {
    Super(int).setWeight(2*w);
  }

  int getWeight() {
    return Super().getWeight()/2;
  }
}
```

AspectJ

```
aspect DoubleWeight {
  void around(int w) : args(w) &&
    execution(void Edge.setWeight(int)) {
      proceed(w*2);
  }
  int around() :
    execution(void Edge.getWeight()) {
      return proceed()/2;
  }
}
```

Beispiel: Complex Dynamic Extension

```
class Node {
  void print() {...}
  ...
}
```

Jak

```
refines class Node {
  static int count = 0;
  void print() {
    if(count == 0)
      printHeader();
    count++;
    Super().print();
    count--;
  }
  void printHeader() { /* ... */ }
}
```

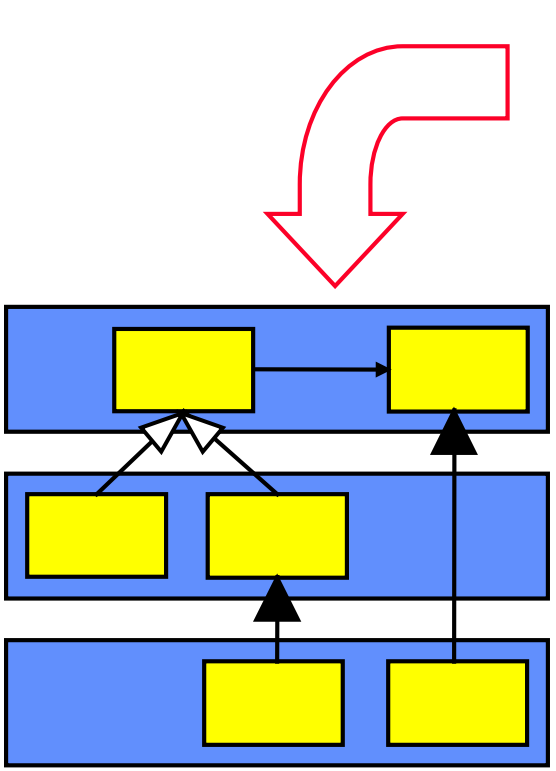
AspectJ

```
aspect PrintHeader {
  before() :
    execution(void print()) &&
    !cflowbelow(execution(void print())) {
      printHeader();
  }
  void printHeader() { /* … */ }
}
```

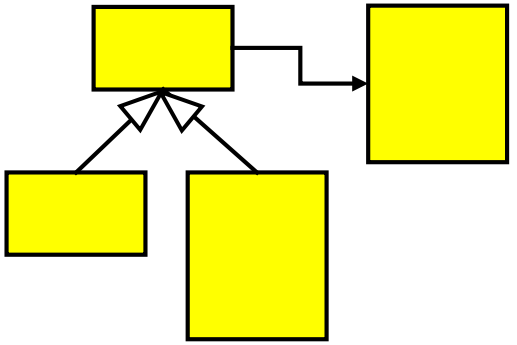
Vergleich FOP vs. AOP

	FOP	AOP
statisch	*gute Unterstützung* – Attribute, Methoden, Klassen	*eingeschränkte Unterstützung* – Attribute, Methoden
dynamisch	*schlechte Unterstützung* – einfach dynamisch (Erweiterung von Methoden)	*gute Unterstützung* – erweitert dynamisch
heterogen	*gute Unterstützung* – Verfeinerungen und Kollaborationen	*eingeschränkte Unterstützung* – keine expliziten Kollaborationen
homogen	*Keine Unterstützung* – Eine Verfeinerung pro Join-Point (Code-Replikation)	*gute Unterstützung* – Wildcards und logische Verknüpfung von Pointcuts

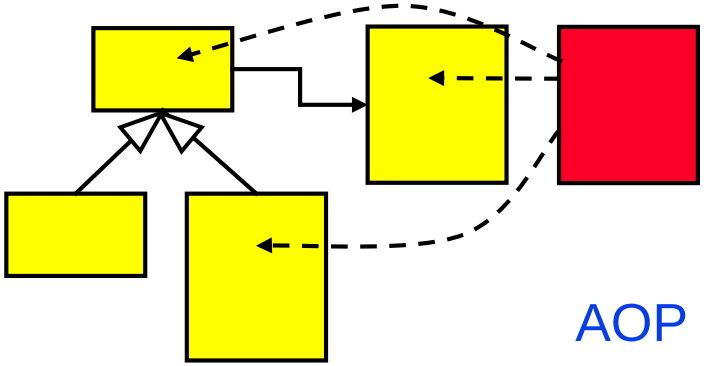
Symbiose von FOP und AOP



FOP

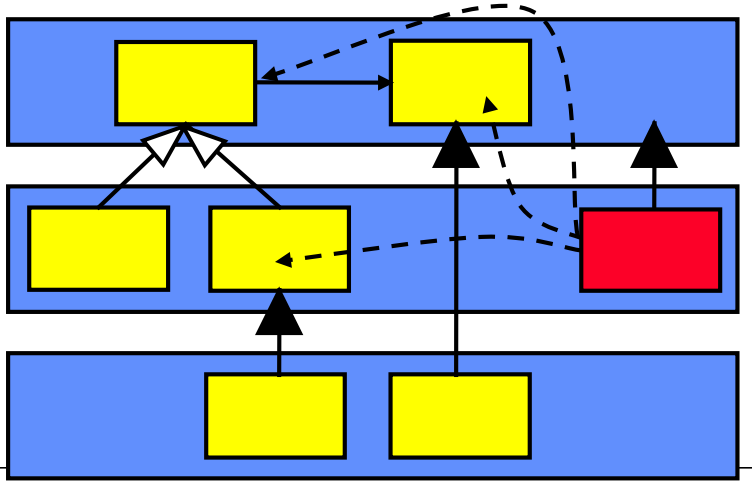


OOP



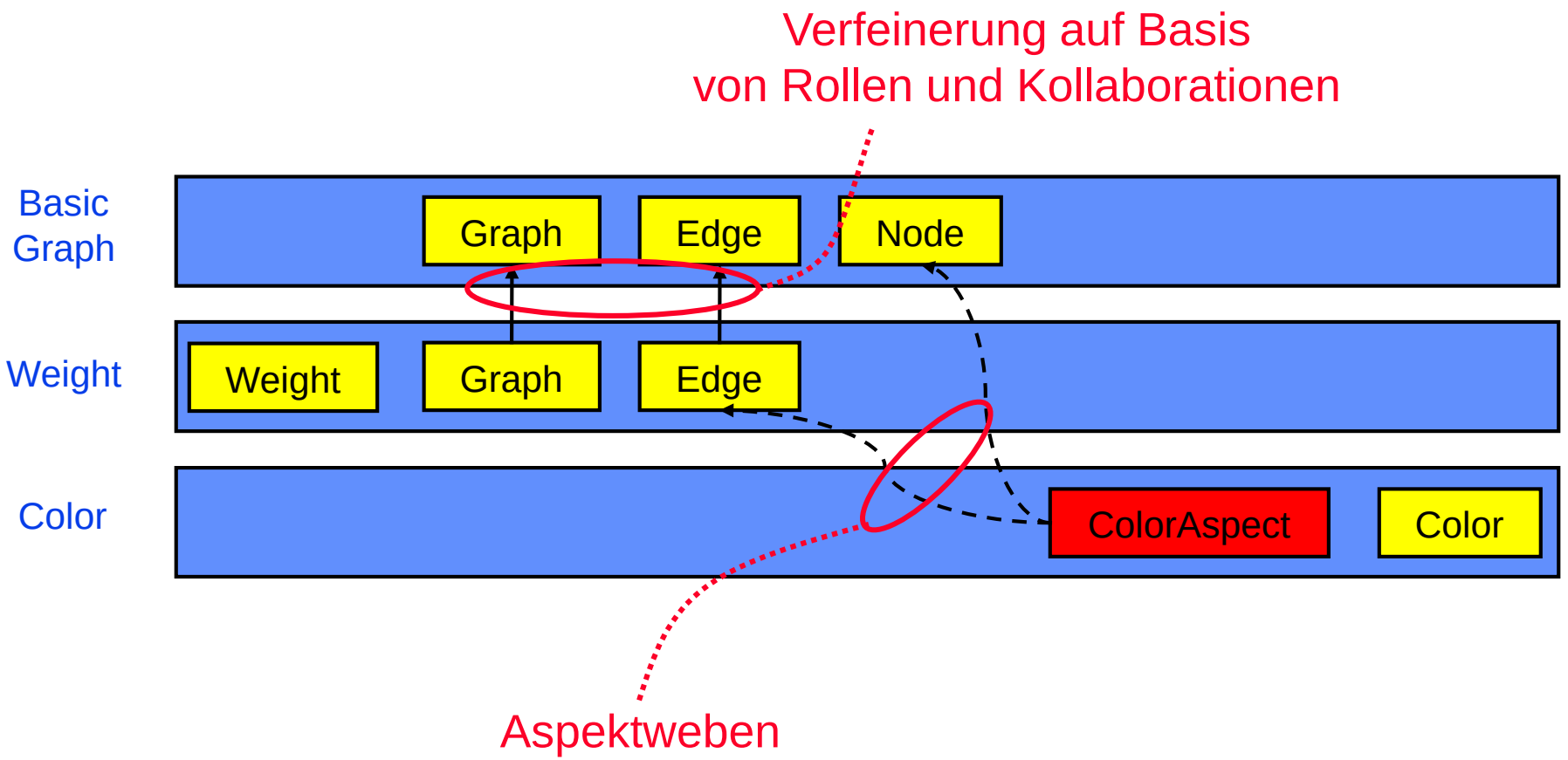
AOP

FOP + AOP



Aspectual Feature Modules

- Integration von Aspekten, Klassen und Verfeinerungen



Tool Support

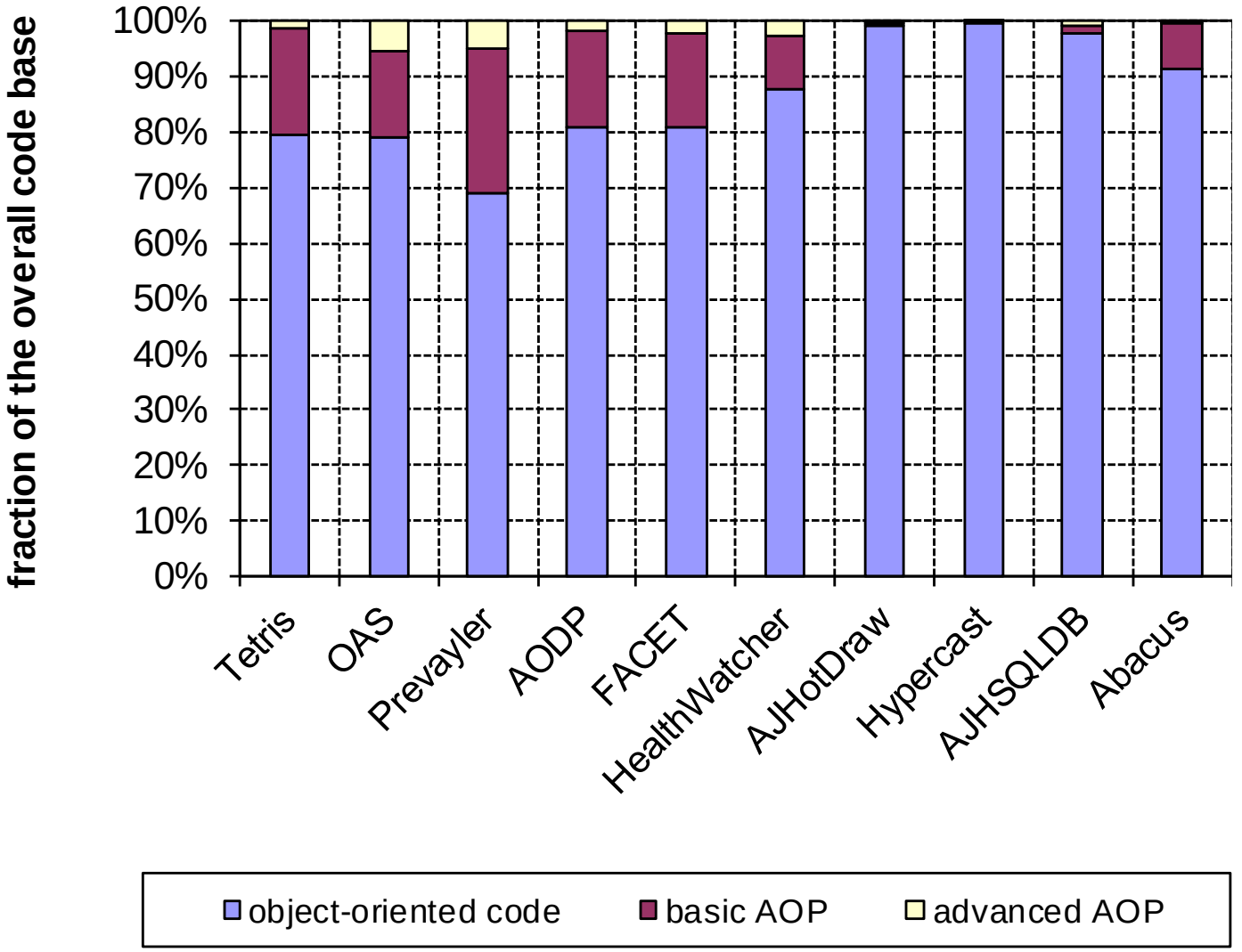
- FeatureC++ & AspectC++
  - Aspectual Feature Modules für C++
  - Direkte Sprachunterstützung
  - Kompilieren erst mit FeatureC++ dann mit AspectC++ Compiler

- AHEAD Tool Suite & AspectJ
  - Java-basierte Variante von Aspectual Feature Modules
  - Aspekt-Dateien in Feature Modulen; übersetzen mit AspectJ Compiler

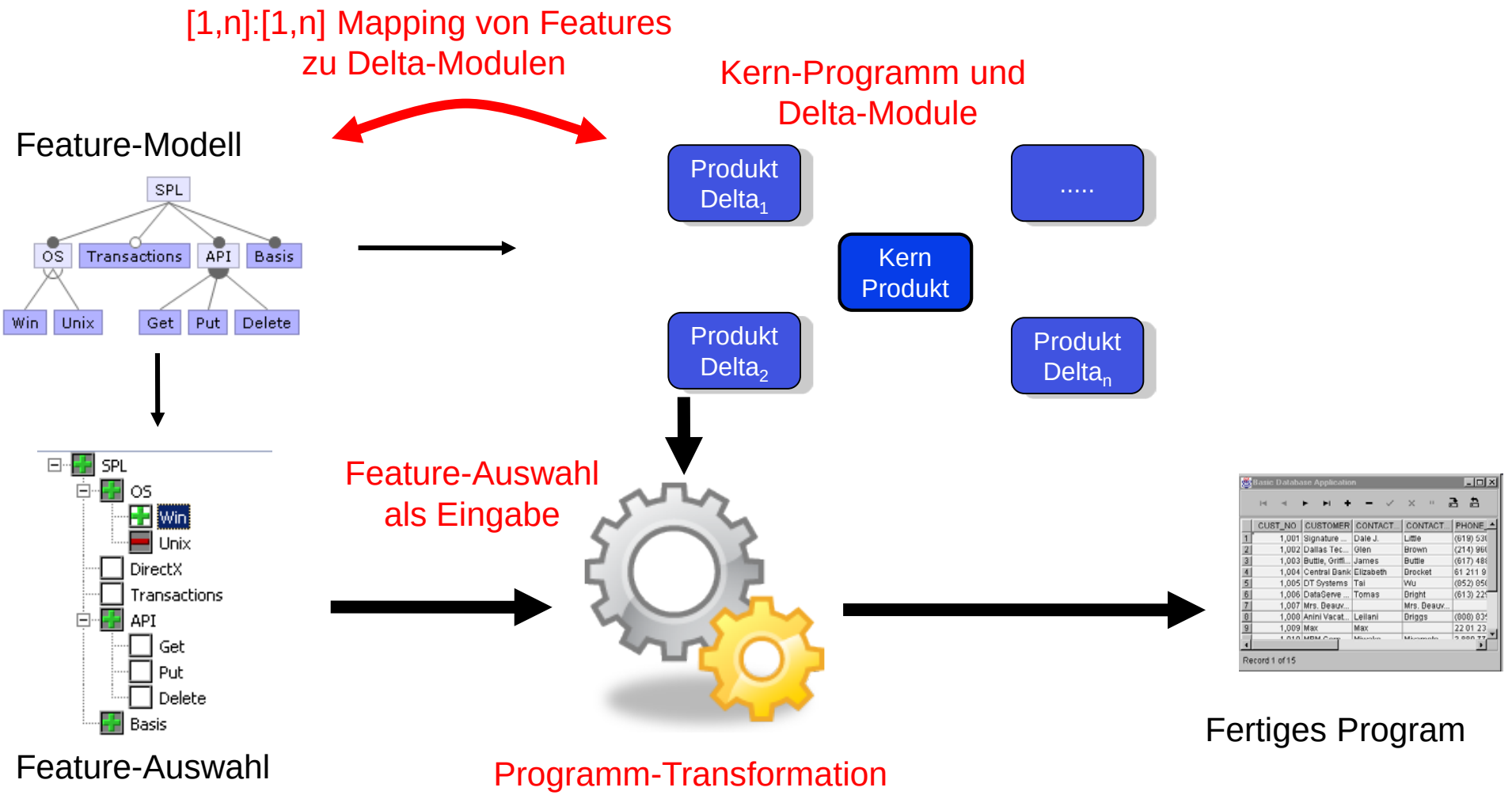
Grenzen von AspectJ

- Aspektvererbung
  - Aspekte, die verfeinert werden sind “fest”
  - Enge Kopplung zwischen Aspekt und Verfeinerung
  - Siehe “Inflexible Erweiterungsmechanismen”
- Abstrakte Aspekte
  - Nur abstrakte Aspekte können verfeinert werden
  - Verfeinerungen müssen vorausgeplant sein
- Advice-Konstrukte haben keinen Namen
  - Können nicht verfeinert werden

Verbreitung von AOP



Delta-Orientierte Produktlinien

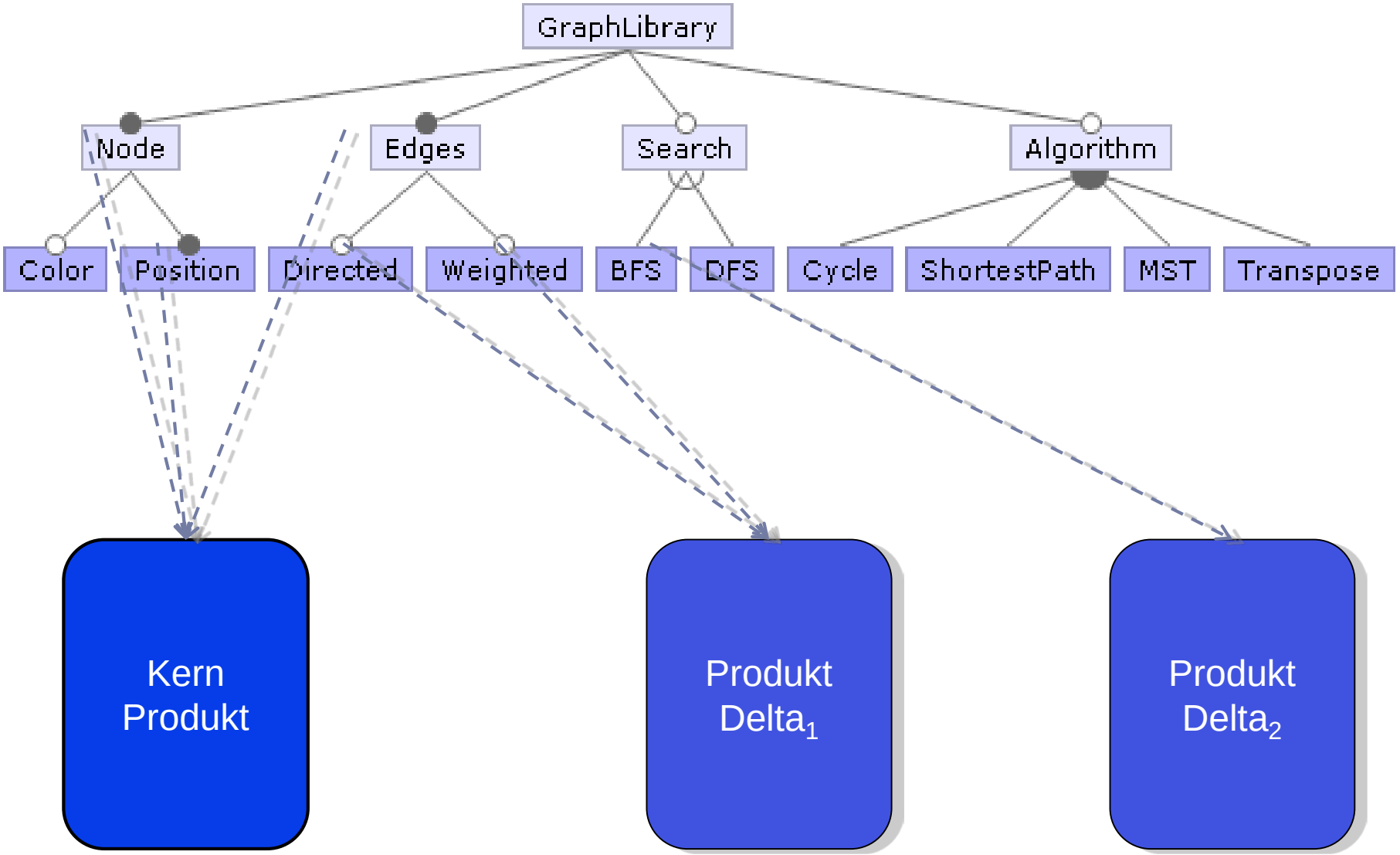


Delta-Oriented Programming (DOP)

- Kernprogramm einer beliebigen partiellen/vollständigen Konfiguration
- Entwicklung durch SPLE oder existierendes Legacy-Produkt

- Modifikationen des Kernprogrammes
- Anwendungsbedingungen über Features
- Ordnung zur Konfliktlösung

Beispiel: Graph-SPL



DeltaJava

- Erweiterung von Java um Delta-Konzept
  - Formales Typsystem basierend auf Featherweight-Java
  - Einführung von Delta-Modulen als kohäsive Implementierungs-Einheit
  - Zwei Ansätze: *CoreDOP* und *PureDOP*
  - Generalisierung von FOP (kein Kern mehr notwendig)
- Typsystem garantiert Sicherheit bei Anwendung der Deltas

Core Modul

- Das Core Modul enthält eine Menge von Java-Klassen

```
core Graph, Node, Edges {
    class Graph {
      Vector nv = new Vector();
      Vector ev = new Vector();
      Edge add(Node n, Node m) {
         Edge e = new Edge(n, m);
        nv.add(n); nv.add(m);
        ev.add(e); return e;
      }
      void print() {
        for(int i = 0; i < ev.size(); i++)
            ((Edge)ev.get(i)).print();
      }
    }
    class Node {...}
    class Edges {...}
}
```

Delta Module

Delta Module enthalten:

- Modifikationen der Klassenstruktur:
  - Hinzufügen, Entfernen und Modifikationen von Klassen
    → {**`adds, removes, modifies`**}

- Modifikation von Klassen
  - Veränderungen der Superklasse und des Konstruktors
  - Hinzufügen / Entfernen von Feldern / Methoden
  - Modifikation von Methoden (Wrapping mit **original** call)

Delta Module

- Anwendungsbedingungen in **when** clause: Aussagenlogische Formel über Features des Feature-Modells

```
delta DAddFoo when Foo
{...}
```

- Anwendungsordnung durch **after** clauses, z.B.

```
delta DAddFoo after DRemBar
{...}
```

Beispiel: Delta Module

Hinzufügen des **Weight** -Features:

```
delta DAddWeight when Weighted {

    modifies class Graph{
        modifies Edge add(Node n, Node m) {
            Edge e = original();
            e.weight = new Weight(); return e;
        }
        adds Edge add(Node n, Node m, Weight w) {
            ...
        }
    }
    modifies class Edge {
        Weight weight = new Weight();
        modifies void print() {...}
    }
}
```

CoreDOP - Einschränkungen

- Gültiges Kernprodukt notwendig
- Nicht immer explizit vorhanden
  → nachträgliche Änderungen des Kernprodukts verkompliziert Entwicklungsprozess
- Anwendungsbedingungen dezentral (pro Delta-Modul)

→ Zusammenhang zwischen Delta-Modulen und Features mitunter schwer verständlich

→ Lösung: PureDOP

PureDOP

- Kein expliziter Kern mehr erforderlich (Schlüsselwort **core** entfällt)
- Kern ergibt sich implizit durch Mapping eines Delta-Moduls auf Core-Feature
- Produktgenerierung durch Anwendung von Deltas
  → wird in **Produktdefinition** festgelegt

PureDop – SPL Implementierung

```
spl GraphSPL {
   features Graph, Node, Edges, Weighted, ...
   configurations Graph && Edges && Node && (Weighted ||...) ...
   deltas
      [DGraph when Graph && Node && Edges]
      [DExtEdges when Weighted && Directed]
      [DRemDirected when Weighted && !Directed]
      [...]
}
```

Feature-Liste und Feature-Modell

Application Condition

Produktspezifikation

Konfiguration

```
product GraphDefault from GraphSPL: {Graph, Node, Edges}
product GraphWeighted from GraphSPL: {Graph, Node, Edges,
Weighted}
```

Tools

- AHEAD Tool Suite + Dokumentation
  - Kommandozeilenwerkzeuge für Jak (Java 1.4-Erweiterung)
  - http://www.cs.utexas.edu/users/schwartz/ATS.html
- FeatureHouse
  - Kommandozeilenwerkzeug für Java, C#, C, Haskell, UML, …
  - http://www.fosd.de/fh
- FeatureC++
  - Alternative zu AHEAD für C++
  - http://www.fosd.de/fcpp
- FeatureIDE
  - Eclipse Plugin für AHEAD, FeatureHouse und FeatureC++
  - Automatisches Bauen, Syntax Highlighting, etc…
  - http://www.fosd.de/featureide

Referenzen

- *Don Batory:* **Feature-Oriented Programming and the AHEAD Tool Suite.** In *Proceedings of the 26th International Conference on Software Engineering* (ICSE '04), 2004.
- *Christian Prehofer:* **Feature-Oriented Programming: A fresh Look at Objects.** In ECOOP'97 — Object-Oriented Programming, Lecture Notes in Computer Science Volume 1241, 1997, pp 419-443.
- Apel, S.; Kastner, C.; Lengauer, C: **FEATUREHOUSE: Language-independent, Automated Software Composition.** In *IEEE 31st International Conference on Software Engineering, ICSE 2009.*, vol., no., pp.221,231.
- *Sven Apel and Christian Lengauer. Superimposition:* **A Language-Independent Approach to Software Composition.** In *Proceedings of the 7th international Conference on Software Composition* (SC'08), 2008, pp. 20-35.
- *Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, John Irwin:* **Aspect-oriented programming. In** ECOOP'97 — Object-Oriented Programming Lecture Notes in Computer Science Volume 1241, 1997, pp 220-242.
- *Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, Nico Tanzarella:* **Delta-Oriented Programming of Software Product Lines.** Software Product Lines: Going Beyond Lecture Notes in Computer Science Volume 6287, 2010, pp 77-91