

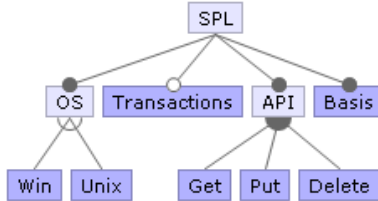
Variabilitätsrealisierungsmechanismen

Michael Nieke, M.Sc.
Sven Schuster, M.Sc.

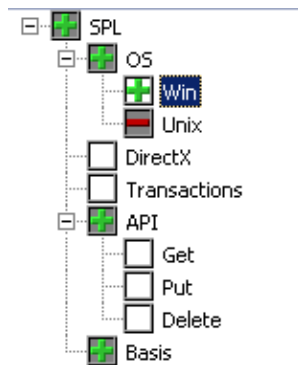
Wie Variabilität implementieren?

Domain Eng.

Feature-Modell



Application Eng.



Feature-Auswahl



Wiederverwendbare
Implementierungs-
artefakte



Generator

	CUST_NO	CUSTOMER	CONTACT	CONTACT..	PHONE
1	1,001	Signature ...	Dale J.	Little	(619) 531
2	1,002	Dallas Tec...	Glen	Brown	(214) 961
3	1,003	Buttle, Grril	James	Buttle	(617) 481
4	1,004	Central Bank	Elizabeth	Brocket	61 211 9
5	1,005	DT Systems	Tai	Wu	(852) 851
6	1,006	DataServe ...	Tomas	Bright	(613) 221
7	1,007	Mrs. Beauv...		Mrs. Beauv...	
8	1,008	Anini Vacat...	Lellani	Briggs	(808) 831
9	1,009	Max	Max		22 01 23
10	1,010	MDM Corp	Mark	Mark	2 000 77

Record 1 of 15

Fertiges Program

Variabilität zur Übersetzungszeit

- ▶ Ziel: Nur benötigter Quelltext wird kompiliert
- ▶ Aber Features flexibel auswählbar

Annotative Ansätze

Präprozessoren

- ▶ Transformieren Quelltext vor Compileraufruf
- ▶ Von einfachen `#include` Befehlen und bedingter Übersetzung zu komplexen Makrosprachen und Regeln
- ▶ In vielen Programmiersprachen üblich
 - ▶ C, C++, Fortran, Erlang mit eigenem Präprozessor
 - ▶ C#, Visual Basic, D, PL/SQL, Adobe Flex

#ifdef Beispiel aus Berkeley DB

```
static int __rep_queue_filedone(dbenv, rep, rfp)
    DB_ENV *dbenv;
    REP *rep;
    __rep_fileinfo_args *rfp; {
#ifndef HAVE_QUEUE
    COMPQUIET(rep, NULL);
    COMPQUIET(rfp, NULL);
    return (__db_no_queue_am(dbenv));
#else
    db_pgno_t first, last;
    u_int32_t flags;
    int empty, ret, t_ret;
#ifdef DIAGNOSTIC
    DB_MSGBUF mb;
#endif
    // over 100 lines of additional code
}
#endif
```

Präprozessor in Java?

- ▶ Nicht nativ vorhanden
- ▶ Bedingte Kompilierung in manchen Compilern nur auf Statement-Ebene, nicht für Klassen oder Methoden

```
class Example {  
  public static final boolean DEBUG = false;  
  
  void main() {  
    System.out.println("immer");  
    if (DEBUG)  
      System.out.println("debug info");  
  }  
}
```

- ▶ Externe Tools vorhanden, z.B. CPP, Antenna, Munge, XVCL, Gears, pure::variants

Antenna

- ▶ Simpler Präprozessor für Java Code
- ▶ Ursprünglich für JavaME zur effizienten Kompilierung (Stichwort Ressourcenbeschränkung)

```
class Example {  
    void main() {  
        System.out.println("immer");  
        //#if DEBUG  
        System.out.println("debug info");  
        //#endif  
    }  
}
```

java Munge **-DDEBUG -DFEATURE2** Datei1.java Datei2.java ... Zielverzeichnis

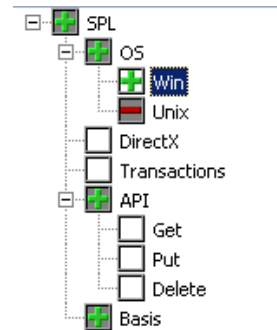
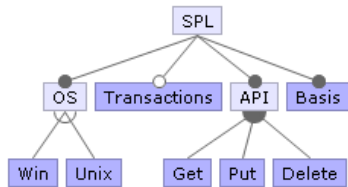


Feature-Auswahl aus Feature-Modell

Produktlinien mit Präprozessoren

Domain Eng.
Application Eng.

Feature-Modell



Feature-Auswahl

Programm mit
Präprozessor-Anweisungen



Feature-Auswahl
als Eingabe



Präprozessor

CUS...	NO	CUSTOMER	CONTACT...	CONTACT...	PHONE
1	001	Signature ...	Dale J.	Little	(619) 531
2	002	Dallas Tec...	Oren	Brown	(214) 961
3	003	Buttle, Griff...	James	Buttle	(617) 491
4	004	Central Bank	Elizabeth	Brocket	61 211 9
5	005	DT Systems	Tai	Wu	(852) 851
6	006	DataServe	Tomas	Bright	(613) 221
7	007	Mrs. Beauv...	Mrs. Beauv...		
8	008	Anni Vacat...	Lellani	Briggs	(808) 831
9	009	Max	Max		22 01 23
10	010	MDM Corp	Mundon	Mundon	5 988 77

Record 1 of 15

Fertiges Program

Kompositionale Ansätze








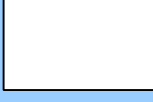
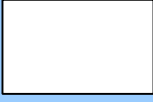
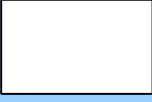
Feature-Orientierte Programmierung

- ▶ Prehofer, ECOOP'97 und Batory, ICSE'03
- ▶ Sprachbasierter Ansatz zur Überwindung des Feature Traceability Problems
- ▶ Jedes Feature wird durch ein Feature-Modul implementiert
 - ▶ Gute Feature Traceability
 - ▶ Trennung und Modularisierung von Features
 - ▶ Einfache Feature-Komposition
- ▶ Feature-basierte Programmgenerierung

Implementierung von Feature-Modulen

- ▶ Aufteilung in Klassen ist etabliert und als Grundstruktur nutzbar
- ▶ Features werden oft von mehreren Klassen implementiert
- ▶ Klassen implementieren oft mehr als ein Feature
- ▶ Idee: Klassenstruktur prinzipiell beibehalten, aber Klassen aufteilen anhand von Features
- ▶ AHEAD (Algebraic Hierarchical Equations for Application Design) oder FeatureHouse als mögliche Werkzeuge

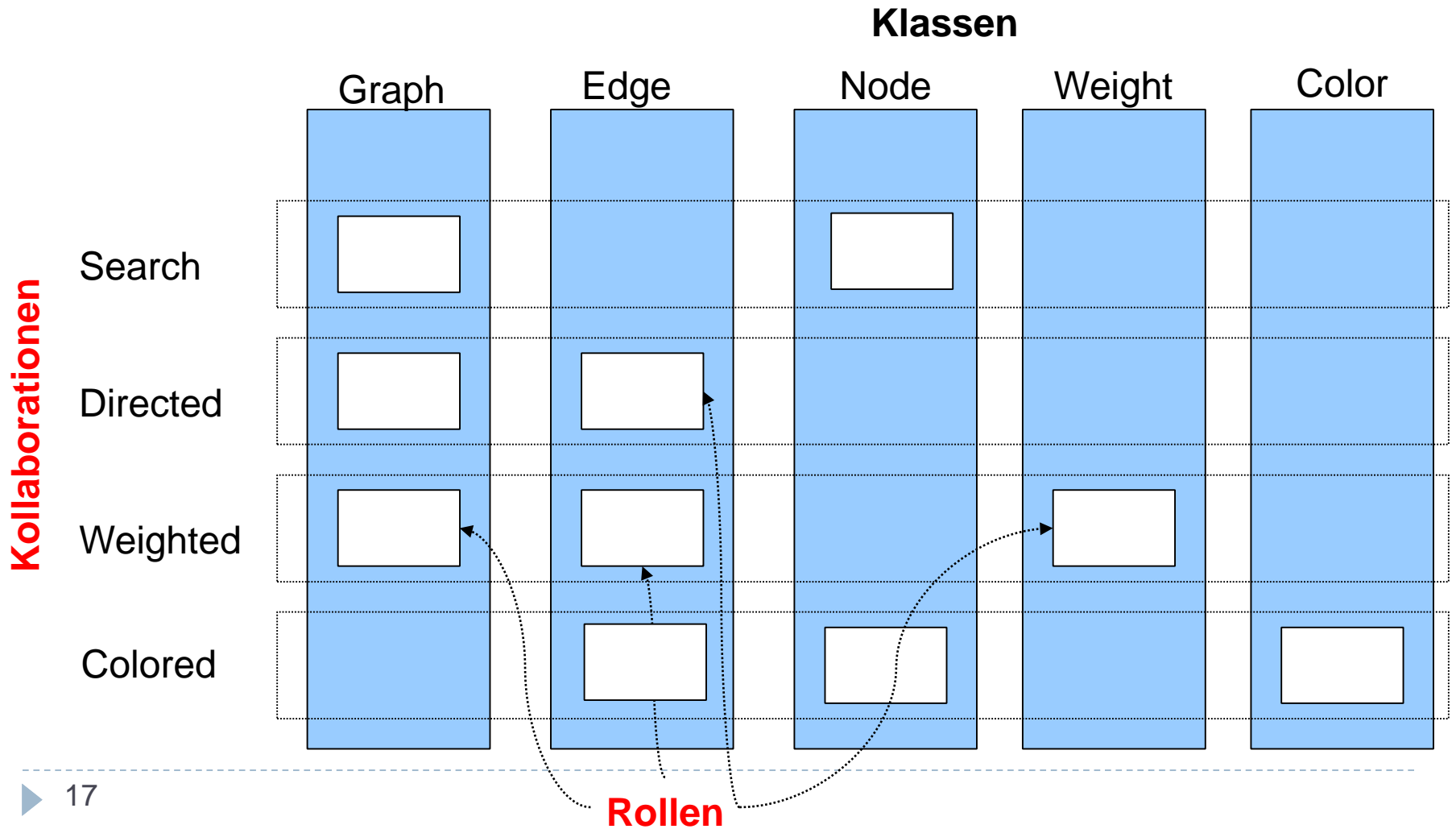
Aufspalten von Klassen

		Klassen				
		Graph	Edge	Node	Weight	Color
Features	Search					
	Directed					
	Weighted					
	Colored					

Kollaborationen & Rollen

- ▶ Kollaboration: eine Menge von Klassen, die miteinander interagieren, um ein Feature zu implementieren
- ▶ Verschiedene Klassen spielen verschiedene Rollen innerhalb einer Kollaboration
- ▶ Eine Klasse spielt verschiedene Rollen in verschiedenen Kollaborationen
- ▶ Eine Rolle kapselt das Verhalten/die Funktionalität einer Klasse, welche(s) für eine Kollaboration relevant ist

Kollaborationen & Rollen



Auszug: Kollaborationenentwurf

```
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();  
    }  
}
```

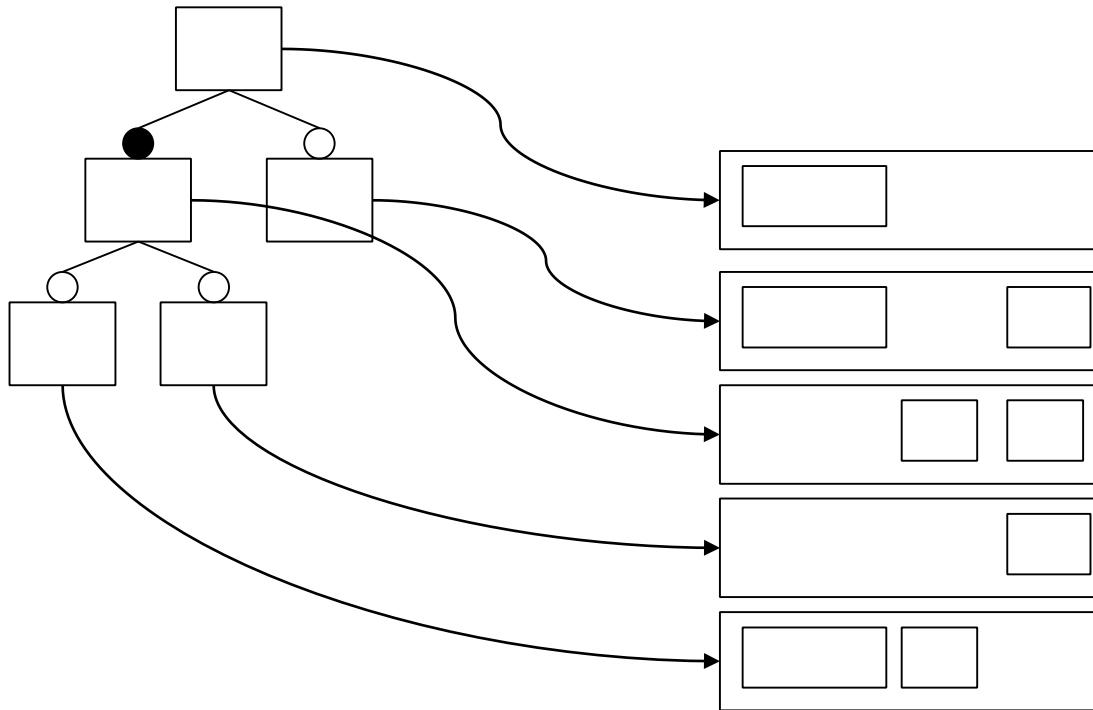
```
class Node {  
    int id = 0;  
    void print() {  
        System.out.print(id);  
    }  
}
```

```
refines class Graph {  
    Edge add(Node n, Node m) {  
        Edge e = Super.add(n, m);  
        e.weight = new Weight();  
    }  
    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;  
}
```

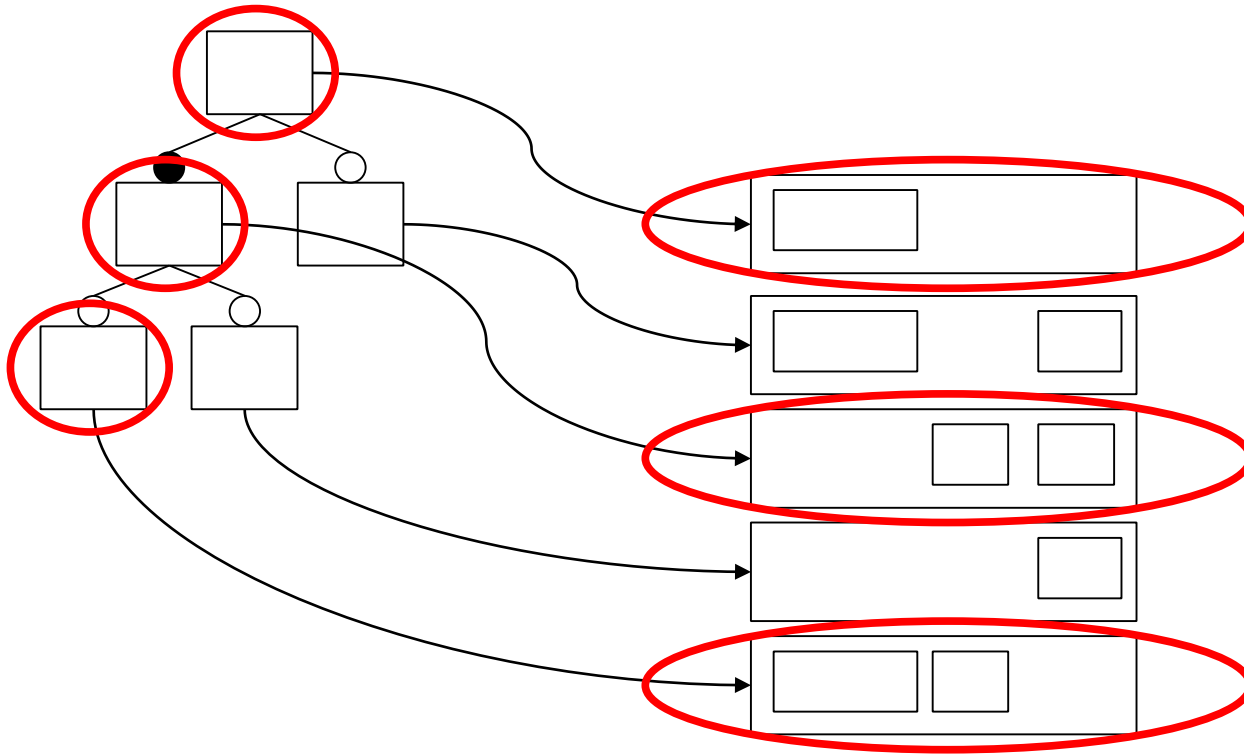
```
refines class Edge {  
    Weight weight = new Weight();  
    void print() {  
        Super.print(); weight.print();  
    }  
}
```

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

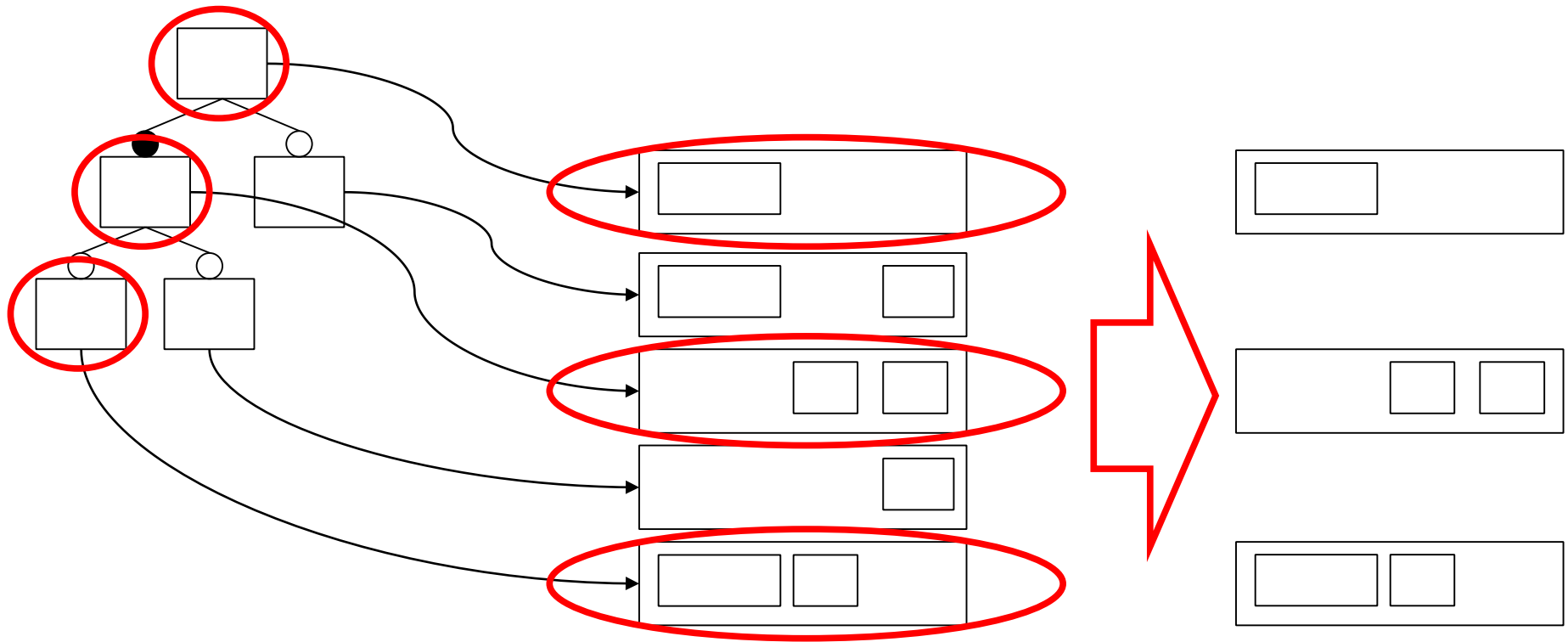

Feature-Komposition



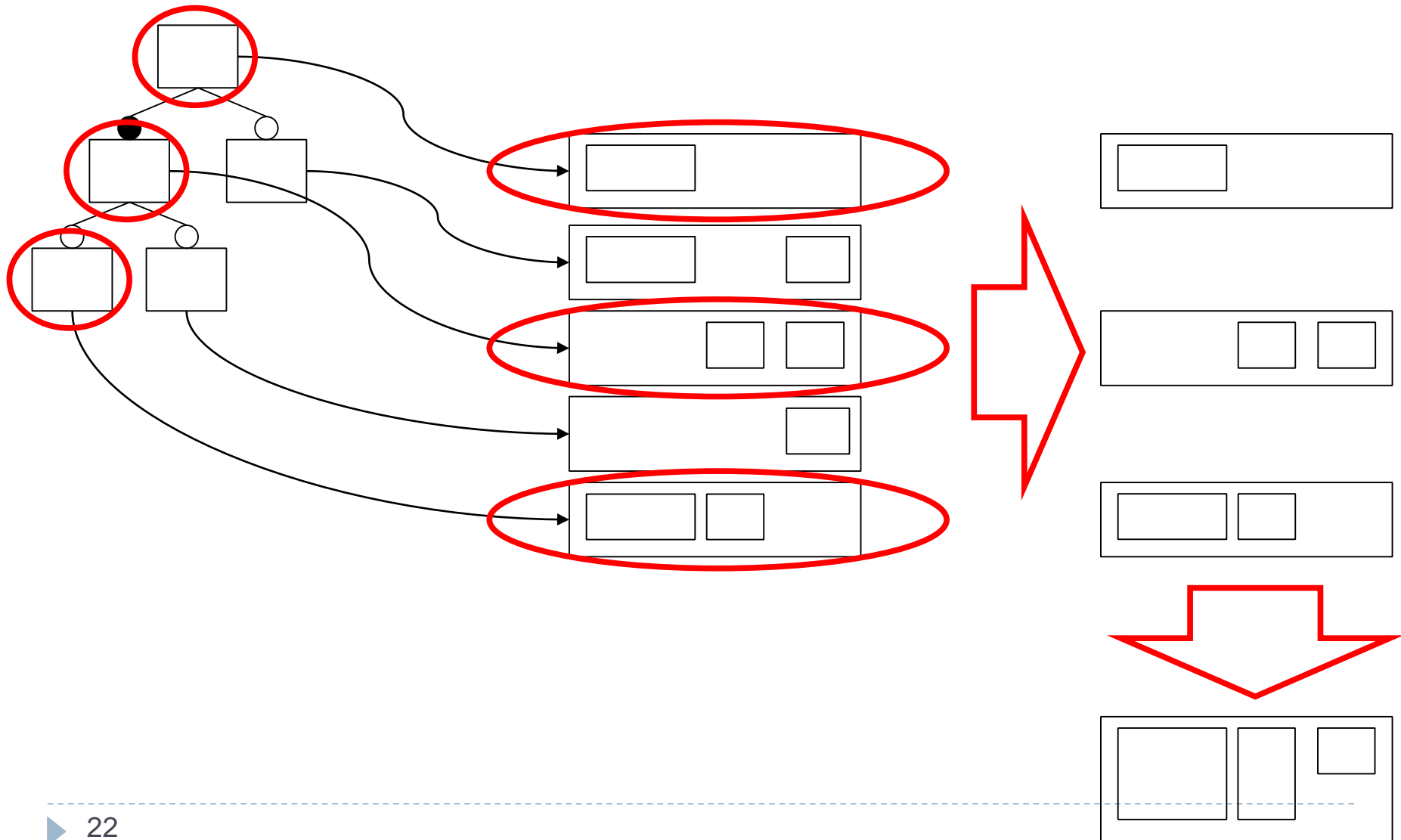
Feature-Komposition



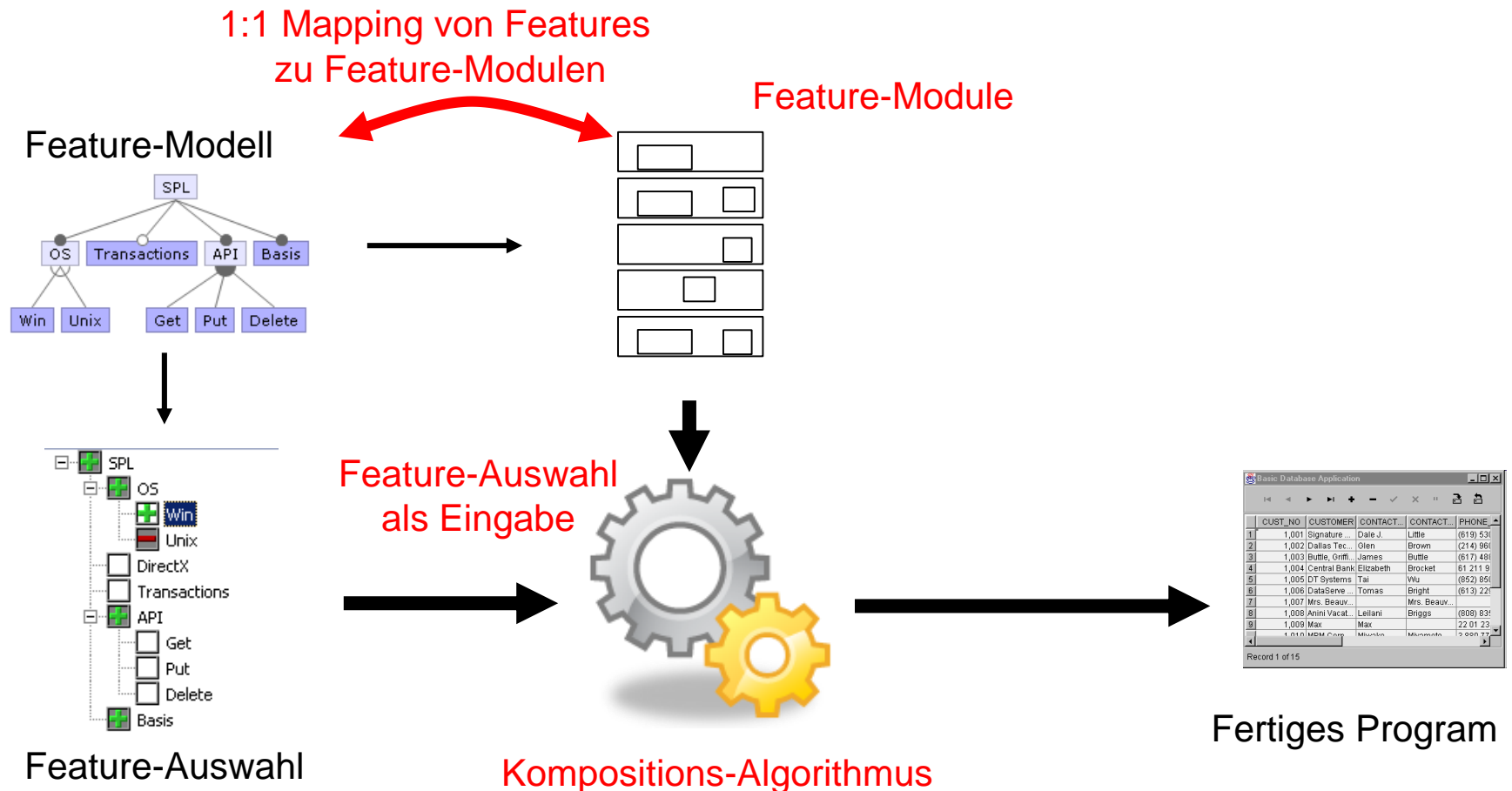
Feature-Komposition



Feature-Komposition



Produktlinien mit Feature-Modulen



Transformationale Ansätze

Ziele

- ▶ Idee: Änderungen zwischen Varianten modular beschreiben → *Deltas*
- ▶ Produkt-orientierte Sicht auf SPL
- ▶ Unterstützung aller Entwicklungsansätze für SPLs
- ▶ Nicht nur für Source Code → *Delta Modeling*
- ▶ Flexibles Mapping von Modulen auf Features

Delta-Oriented Programming (DOP)

Transformational Approach

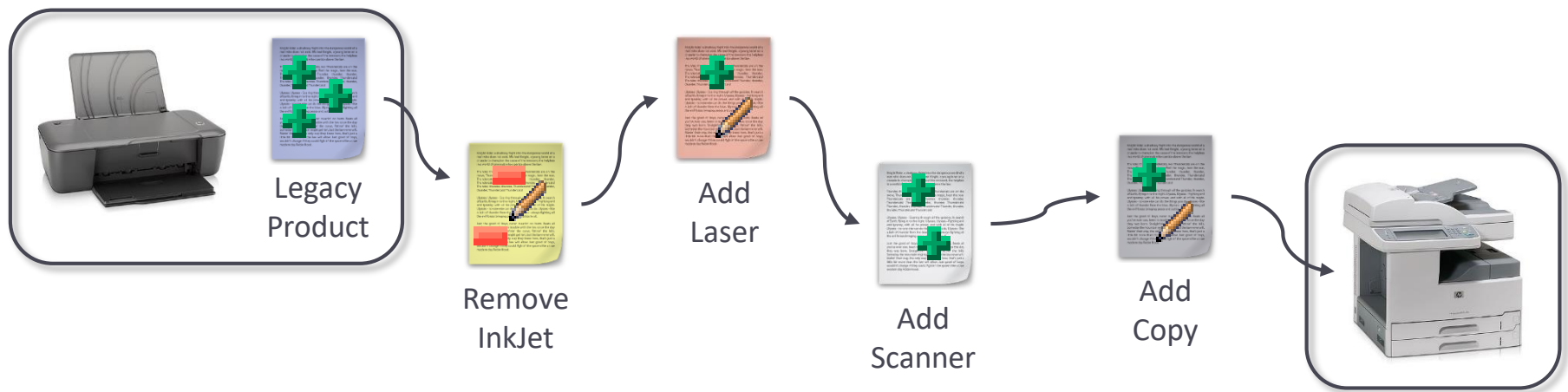
- Transform legacy product to specific variant
- Efficiently implement SPLs
- Combine expressiveness & cohesion

Delta Operations

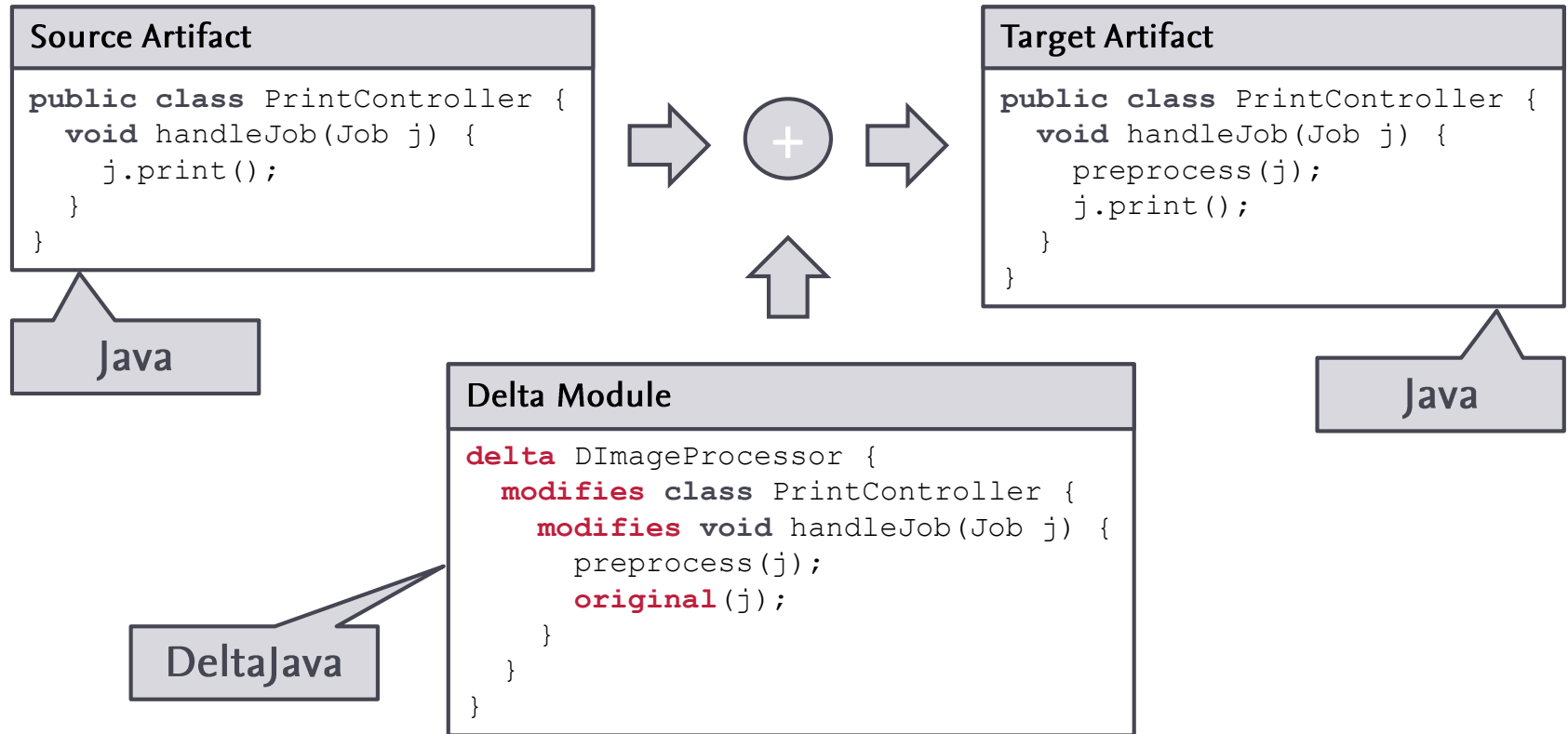
1. **Addition** of new elements
2. **Modification** of existing elements
3. **Removal** of existing elements

Delta Modules

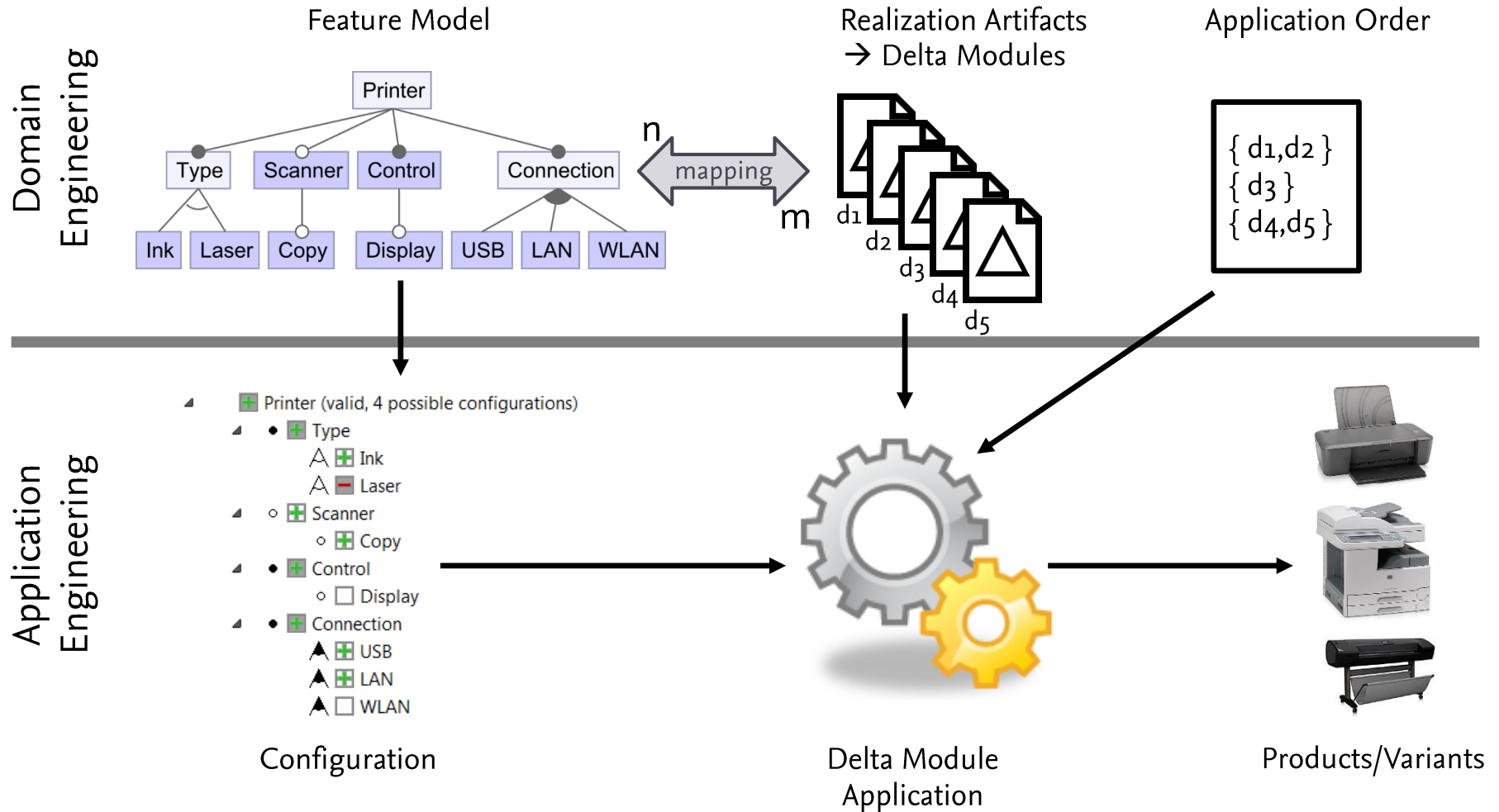
- Employ Delta Operations
- Modify legacy product
- Disjoint source code fragments



Delta-Oriented Programming (DOP) – Code Example



Produktlinien mit Delta-Modulen



Delta Operationen in DeltaJava

delta:

```
delta X <requires Y, ...> {  
    [deltaUnit]  
}
```

deltaUnits:

```
adds {  
    package [package]  
    import [...];  
    public class [...] { ... }  
}  
  
modifies [classifier] {  
    [deltaOperations]  
}  
  
removes [classifier];
```

deltaOperations:

```
adds [member];  
adds import [import];  
adds interfaces [interface1, interface2, ...];  
adds superclass [superclass];  
  
removes [member];  
removes import [import];  
removes [method] ([parameterTypes]);  
removes interfaces [interface1, interface2, ...];  
removes superclass [superclass];  
  
modifies [method] ([parameters]) {  
    [content]  
    original(...); // optional  
}  
  
modifies superclass [superclass];  
modifies constructor ([parameters]) {  
    [contents]  
}  
  
visibility [target] {[TypeOrModifiers]};  
    ex: visibility x {int !private public};
```

-
- ▶ Git: <https://github.com/TUBS-ISF/ProjectSPLDevelopment>
 - ▶ Checkout Project
 - ▶ Tutorial/de.tu_bs.cs.isf.deltatalk