Software Product Lines

Concepts, Analysis and Implementation

Programmier-Paradigmen für Software-Produktlinien (1/3)

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

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

III. Produktlinien-Analyse

- Feature-Interaktion
- Testen von Produktlinien
- Verifikation von Produktlinien

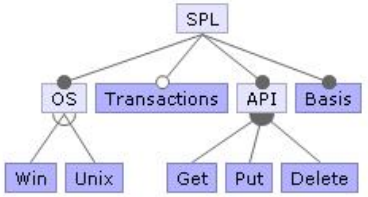
IV. Fallbeispiele und aktuelle Forschungsthemen

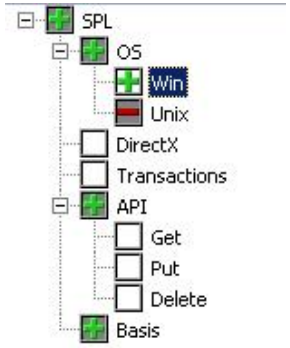
Software-Product-Line Engineering

Domain Eng.

Application Eng.

Feature-Modell

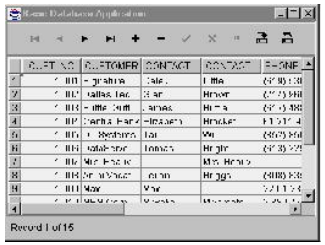




Feature-Auswahl

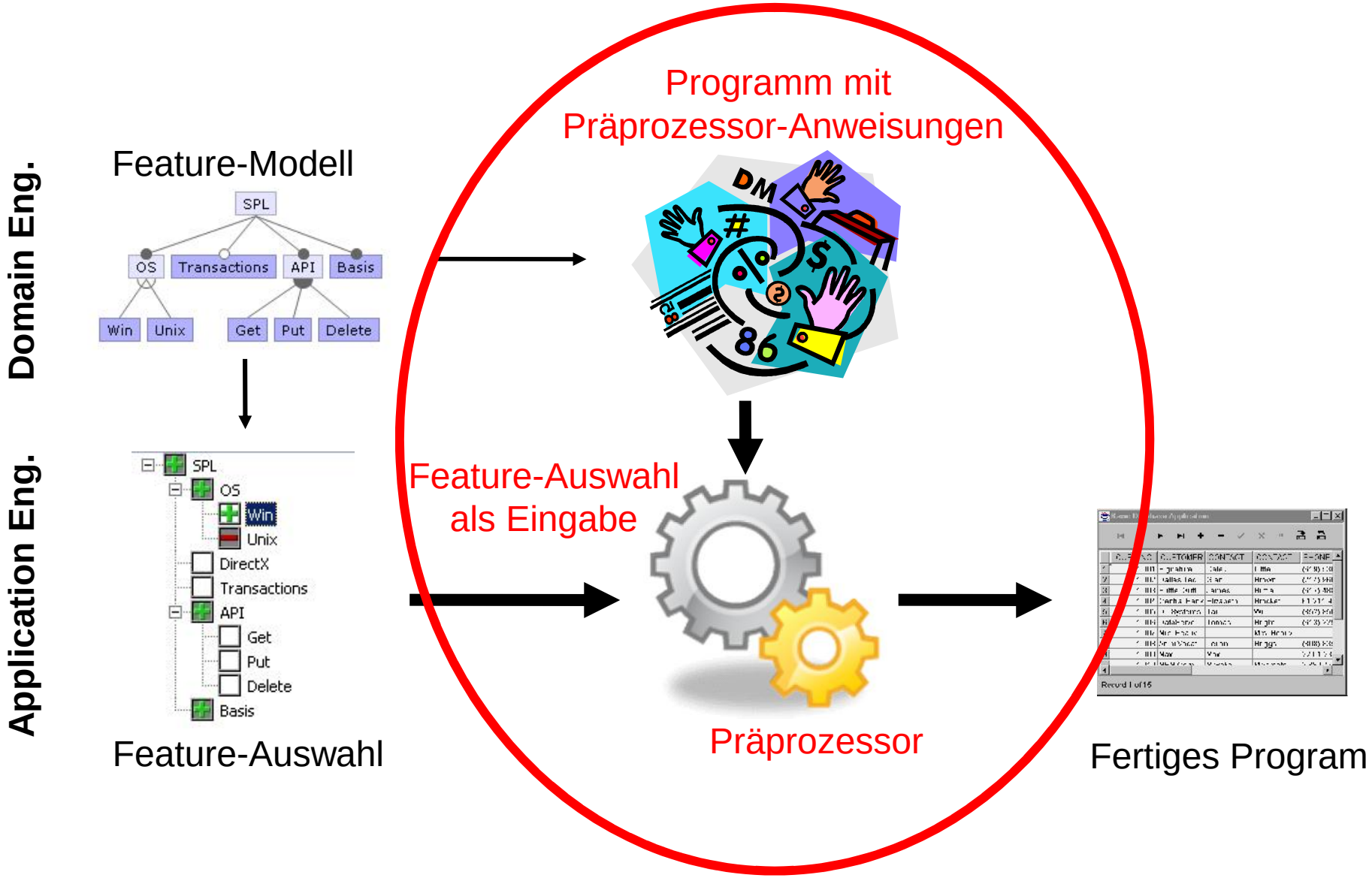
Wiederverwendbare Implementierungs-artefakte

Generator

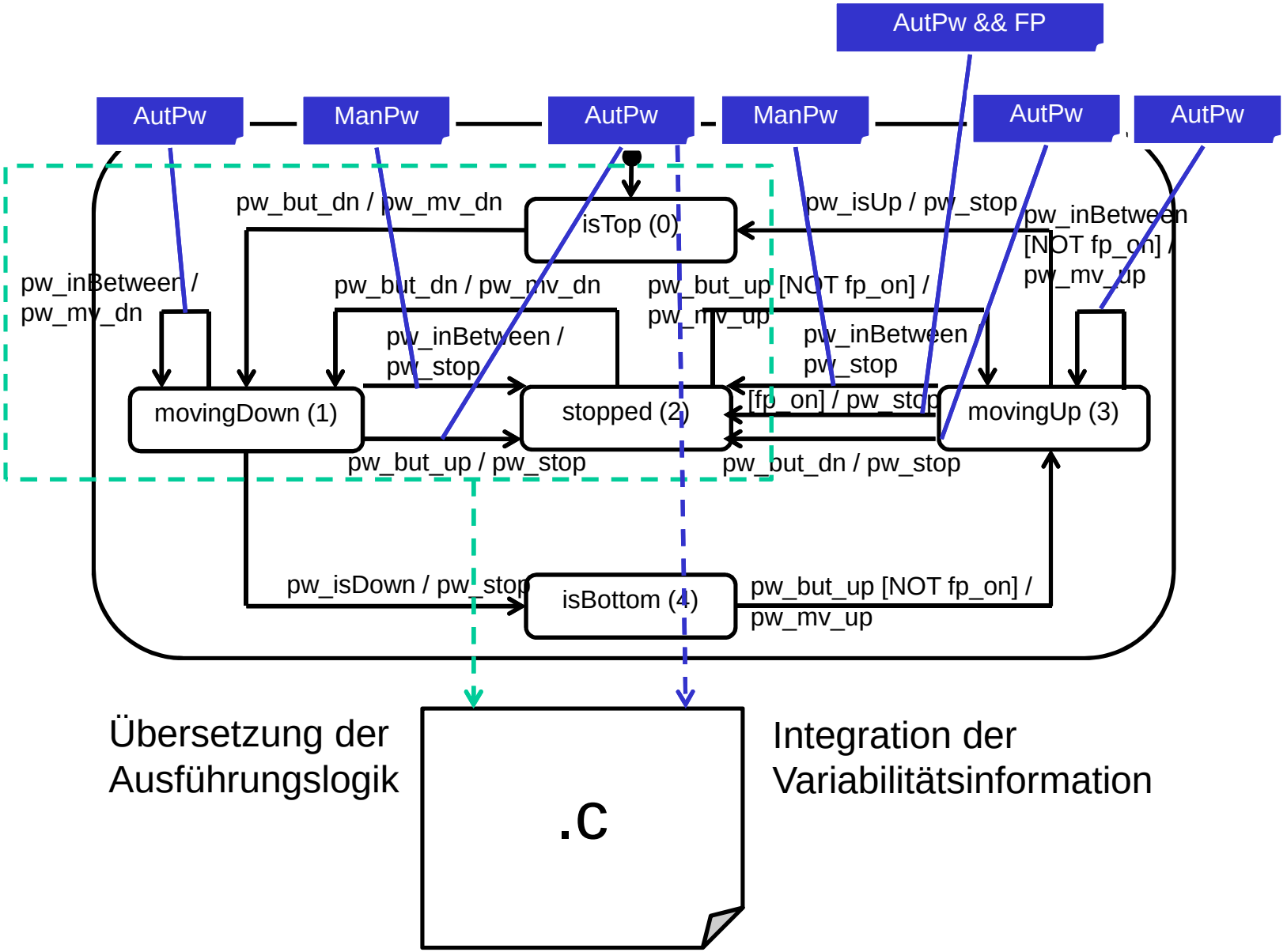


Fertiges Program

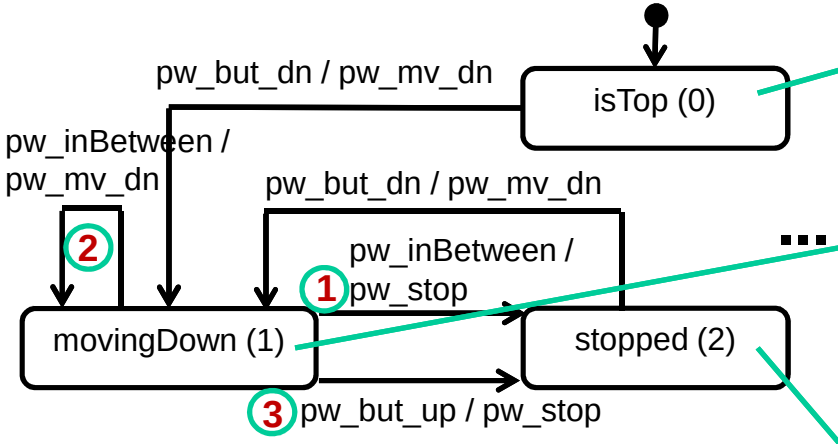
Präprozessoren



Implementierung von SPLs



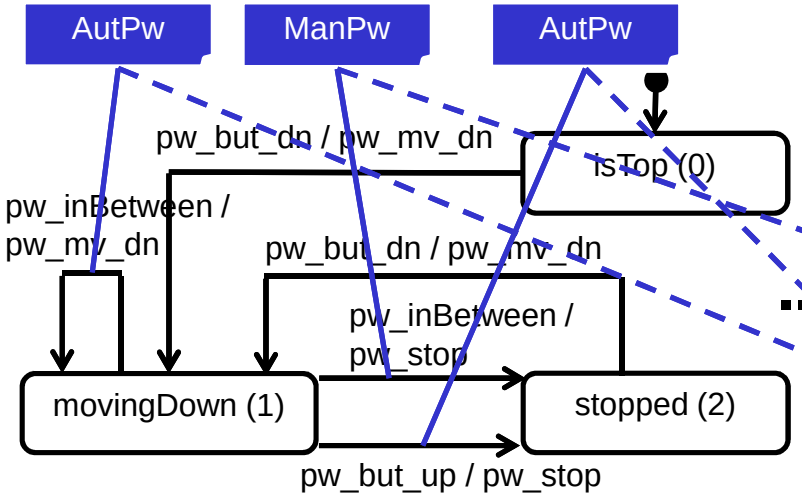
Implementierung von State Machines




```
int state = 0;
while(true) {
    Event ev = eventInQueue.getNext();
    switch(state) {
        case 0:
            if (ev == pw_but_dn) {
                eventOutQueue.put(pw_mv_dn);
            state = 1; break; }
        case 1:
          1 if (ev == pw_inBetween) {
                eventOutQueue.put(pw_stop);
                state = 2; break; }
          2 if (ev == pw_inBetween) {
                eventOutQueue.put(pw_mv_dn);
                break; }
          3 if (ev == pw_but_up) {
                eventOutQueue.put(pw_stop);
                state = 2; break; }
            ...
        case 2:
            if (ev == pw_but_dn) {
                eventOutQueue(pw_mv_dn);
                state = 1; break; }
        ...
    }
}
```

- Switch-case über aktuellen Zustand (Variable `state`)
- If-Statement für jede ausgehende Transition

Implementierung mit Variabilität

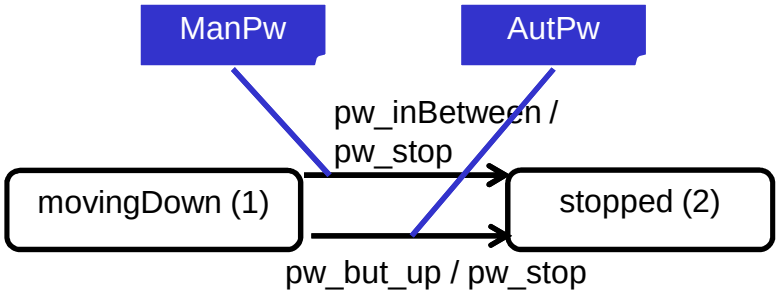



```
int state = 0;
while(true) {
    Event ev = eventInQueue.getNext();
    switch(state) {
        case 0:
            if (ev == pw_but_dn) {
                eventOutQueue.put(pw_mv_dn);
            state = 1; break; }
        case 1:
            if (ev == pw_inBetween) {
                eventOutQueue.put(pw_stop);
                state = 2; break; }
            if (ev == pw_inBetween) {
                eventOutQueue.put(pw_mv_dn);
                break; }
            if (ev == pw_but_up) {
                eventOutQueue.put(pw_stop);
                state = 2; break; }
            ...
        case 2:
            if (ev == pw_but_dn) {
                eventOutQueue(pw_mv_dn);
                state = 1; break; }
            ...
    }
}
```

- Mapping zwischen annotierten Modellfragment und Code-Fragmenten?
- Variabilität von Code-Fragmenten?

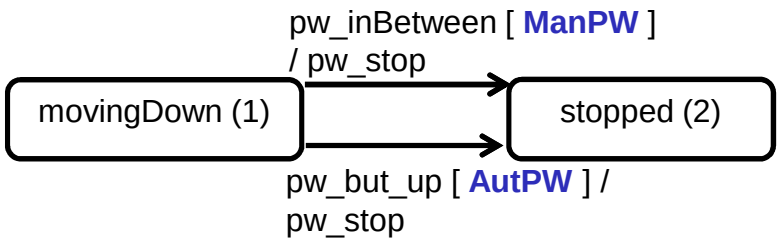
Übersetzung von Variabilität

1. Modell-Annotationen ⇒ Code-Annotationen



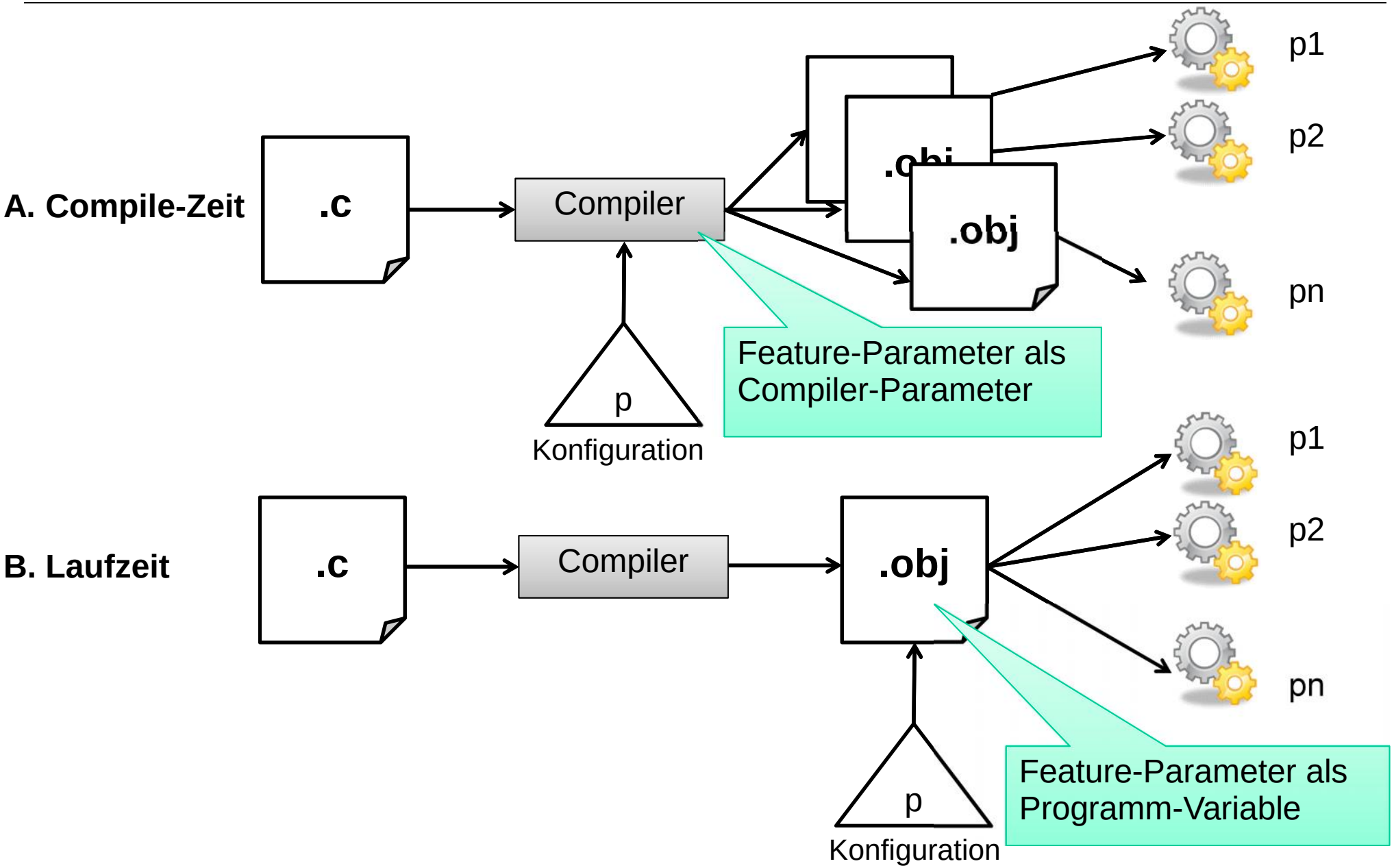

```
case 1:
    #ifdef(ManPW)
        if (ev == pw_inBetween) {
            eventOutQueue.put(pw_stop);
            state = 2; break; }
    #endif
    #ifdef(AutPW)
        if (ev == pw_but_up) {
            eventOutQueue.put(pw_stop);
            state = 2; break; }
    #endif
```

2. Variability Encoding ⇒ Teil der Programmlogik




```
case 1:
    if (ManPW && ev == pw_inBetween) {
        eventOutQueue.put(pw_stop);
        state = 2; break;
    }
    if (AutPW && ev == pw_but_up) {
        eventOutQueue.put(pw_stop);
        state = 2; break;
    }
```

Binde-Zeitpunkt von Variabilität



Variabilität zur Compile-Zeit vs. Laufzeit

Variabilität zur Laufzeit

- Gesamter Quelltext aller Konfigurationen wird kompiliert
- Belegung der Feature-Parameter variiert den Kontrollfluss der Programmausführung

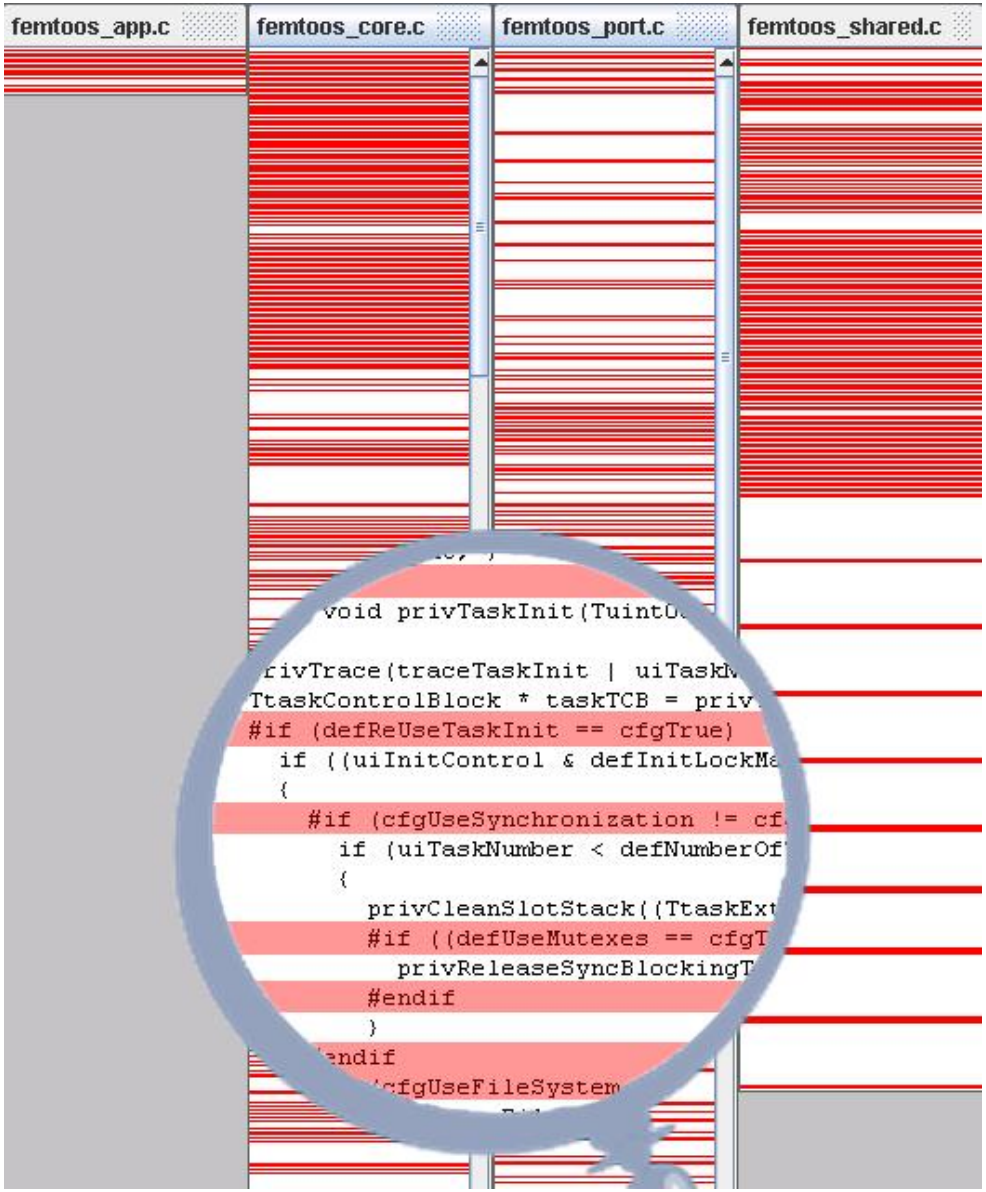
Variabilität zur Compile-Zeit

- Nur für die Konfiguration benötigter Quelltext wird kompiliert
- Belegung der Feature-Parameter variiert den übersetzten Quelltext

Beispiel: #ifdef in Berkley DB

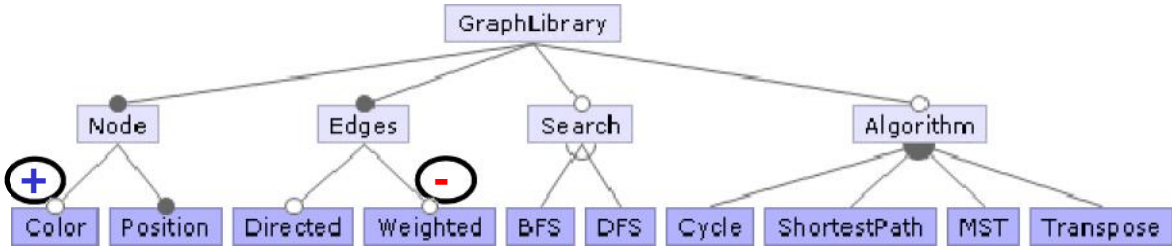
```
static int __rep_queue_filedone(dbenv, rep, rfp)
        DB_ENV *dbenv;
        REP *rep;
        __rep_fileinfo_args *rfp; {
#ifndef HAVE_QUEUE
        COMPQUIET(rep, NULL);
        COMPQUIET(rfp, NULL);
        return (__db_no_queue_am(dbenv));
#else
        db_pgno_t first, last;
        u_int32_t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

Beispiel: #ifdef in Femto OS



http://www.femtoos.org/

Beispiel: Graph SPL in Java




```
class Conf {
  public static boolean COLORED = true;
  public static boolean WEIGHTED = false;
}
```

```
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);
    if (Conf.WEIGHTED) e.weight = new Weight();
    return e;
  }
  Edge add(Node n, Node m, Weight w)
    if (!Conf.WEIGHTED) throw RuntimeException();
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m); ev.add(e);
    e.weight = w; return e;
  }
  void print() {
    for(int i = 0; i < ev.size(); i++) {
      ((Edge)ev.get(i)).print();
    }
  }
}
```

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

```
class Edge {
  Node a, b;
  Color color = new Color();
  Weight weight;
  Edge(Node _a, Node _b) { a = _a; b = _b; }
  void print() {
    if (Conf. COLORED) Color.setDisplayColor(color);
    a.print(); b.print();
    if (!Conf.WEIGHTED)  weight.print();
  }
}
```

```
class Color {
  static void setDisplayColor(Color c) { ... }
}
```

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

Präprozessoren

- Präprozessoren transformieren den Quelltext vor dem eigentlichen Compileraufruf
- Funktionsumfang: von einfachen **`#include`** Befehlen und bedingter Übersetzung bis zu komplexen Makrosprachen und Regeln
- Grundlage für **generische Programmierung / konfigurierbare Programme / Meta-Programmierung / …**
- Fester Bestandteil vieler Programmiersprachen
  - C, C++, Fortran, Erlang mit eigenem Präprozessor
  - C#, Visual Basic, D, PL/SQL, Adobe Flex
  - Java: externe Tools

C-Präprozessor

Präprozessor-Direktiven sind spezielle Instruktionen im Programm-Code, die nicht Teil des endgültigen (durch den Compiler übersetzten) Programms sind, sondern vom Präprozessor vor der Kompilierung verarbeitet (und dabei entfernt) werden.

- Der C-Präprozessor (CPP) wird vor der eigentlichen Kompilierung aufgerufen und transformiert das Programm gemäß der im Programm enthaltenen Direktiven
- CPP ist Teil des ANSI C Standards
- Für Präprozessor-Direktiven in C gilt
  - bestehen aus Zeilen beginnend mit **#** gefolgt vom Namen und (optionalen) weiteren Argumenten für die Direktive
  - dürfen sich nicht über eine Zeile hinaus erstrecken

Bestandteile von CPP

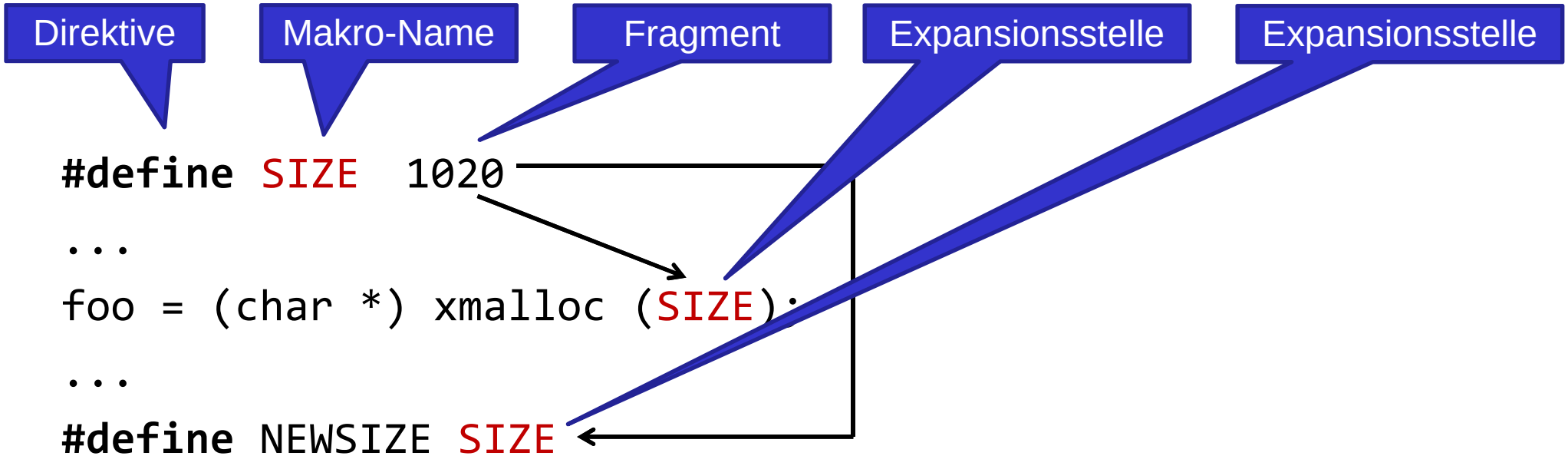
Der CPP umfasst vier Mechanismen

- Auslagern von Code und Einbinden durch Header-Files
- Makros zum Definieren/Verwenden und Finden/Ersetzen (Expandieren) von Namen für beliebige Code-Fragmente
- Bedingte Kompilierung: **optionale/alternative** Code-Fragmente vor der Kompilierung einblenden/ausblenden bzw. auswählen.

Generische Programmierung

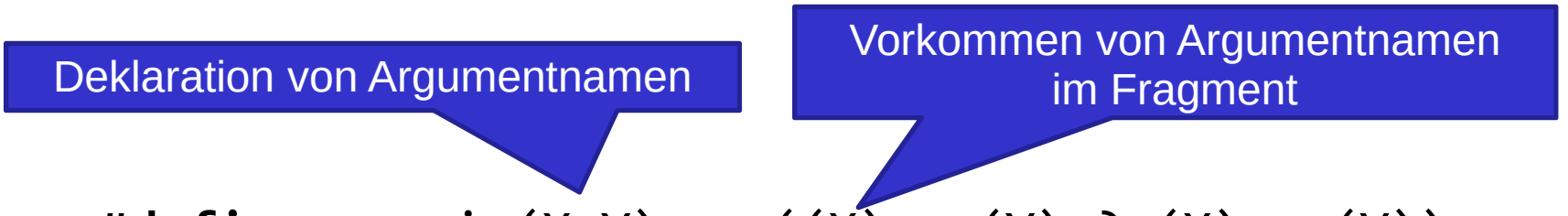
- Explizite Deklaration und Kennzeichnung von Programmzeilen, z.B. für Tracing, Debugging etc.

Definition einfacher Makros



- Definition von Namen für beliebige Code-Fragmente
- Jedes Auftreten des Namens im nachfolgenden Programmtext wird durch das Fragment ersetzt (expandiert)
- Fragmente können selbst wieder Makro-Namen enthalten
- Fragmente werden nicht interpretiert, sondern als einfache Strings „kopiert“

Makros mit Argumenten




```
#define     min(X,Y)      ((X) < (Y) ? (X) : (Y))
...
min(1,2)    ⟶             ((1) < (2) ? (1) : (2))
...
min(x+42,*p) ⟶            ((x+42) < (*p) ? (x+42) : (*p))
...
min(min(a,b),c) ⟶         ... // => Übung
```

- Argumente können an jeder Expansionsstelle neu gesetzt werden
- Genau wie Fragmente sind Argumente uninterpretierte Strings
- Schachtelung von Makro-Expansionen und/oder Argumenten beliebig möglich

Makros un-/umdefinieren

```
#define FOO 4
x = FOO;          ⟶    x = 4;
#define FOO 5                    // CPP WARNING
#undef  FOO
x = FOO;          ⟶    x = FOO; // COMPILE-TIME ERROR?
#define FOO 5
x = FOO;          ⟶    x = 5;
```

- Anwendung auf einfache Makros und Makros mit Argumenten
- **#undef** ohne Auswirkungen, falls der Name nicht zuvor als Makro definiert wurde
- **#undef** sollte zwischen Definition und Neu-Definition eines Makro-Namens aufgerufen werden, andernfalls gibt CPP eine Warnung aus

Bedingte Kompilierung

- Markieren von Quelltextabschnitten, die beim Kompilieren unter bestimmten Bedingungen ignoriert werden, d.h. nicht in das Zielprogramm übersetzt werden
- Bedingungen müssen zur Kompilierzeit auswertbar sein
- Mögliche Bedingungen:
  - Prüfen, ob ein Name als Makro definiert/undefiniert ist
  - (eingeschränkte) Boole‘sche Ausdrücke über Makro-Namen und Konstanten

#if Direktive

```
#if expression
    // conditional code
#endif
```

```
#if SIZE < 1020
    foo = (char *) xmalloc(SIZE);
#endif
```

- Der *conditional code* kann weitere Makros enthalten
- In der *expression* dürfen vorkommen:
  - Literale vom Typ *Integer* und *Character*
  - Makro-Namen
  - Arithmetische, relationale und logische Operatoren
  - Namen, die keine Makros sind, werden als Konstante 0 interpretiert

#else Direktive

```
#if expression
    // conditional code
#else
    // alternative conditional code
#endif
```

```
#if SIZE < 1020
    foo = (char *) xmalloc(SIZE);
#else
    foo = (char *) xmalloc(1020)
#endif
```

- Auswahl zwischen zwei alternativen Quelltext-Abschnitten
- Es wird immer genau einer der beiden *conditional code* Abschnitte kompiliert

#elif Direktive

```
#if expression
    // 1. alternative conditional code
#elif  expression
    // 2. alternative conditional code
…
#elif  expression
    // n. alternative conditional code
#else
    // default conditional code
#endif
```

Auswertungsreihenfolge

- Auswahl zwischen *n* alternativen Quelltextabschnitten
- Die *expressions* müssen sich nicht zwangsläufig gegenseitig ausschließen (ähnlich zu switch-case)

#ifdef und #ifndef

```
#ifdef makro-name
   // conditional code
#endif
```

```
#ifdef SIZE
    foo = (char *) xmalloc(SIZE);
#endif
```

```
#ifndef makro-name
   // conditional code
#endif
```

```
#ifndef SIZE
    foo = (char *) xmalloc(1020);
#endif
```

- Der *conditional code* wird kompiliert, wenn ein Makro-Name definiert / nicht definiert (bzw. undefiniert) ist

#if defined

`#if defined (foo) ... #endif`

- äquivalent zu **`#ifdef foo`**

`#if ! defined (foo) ... #endif`

- äquivalent zu **`#ifndef foo`**

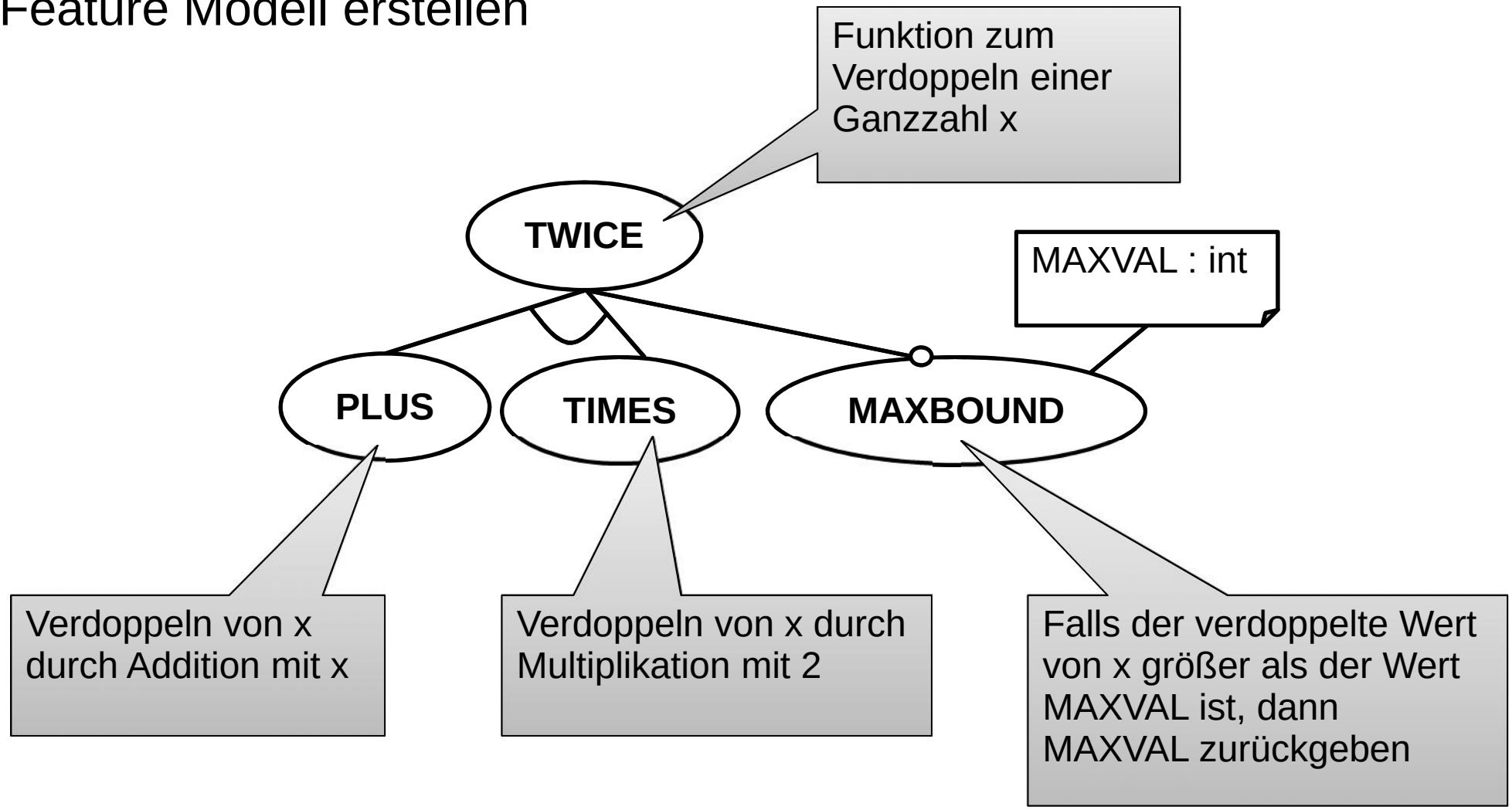
`#if defined (foo) && defined (bar) ... #endif`

- äquivalent zu **`#ifdef foo { #ifdef bar...#endif } #endif`**

- Verallgemeinerung von **`#ifdef`** und **`#ifndef`**
- Formulierung beliebiger logischer Bedingungen über Definition von Makro-Namen

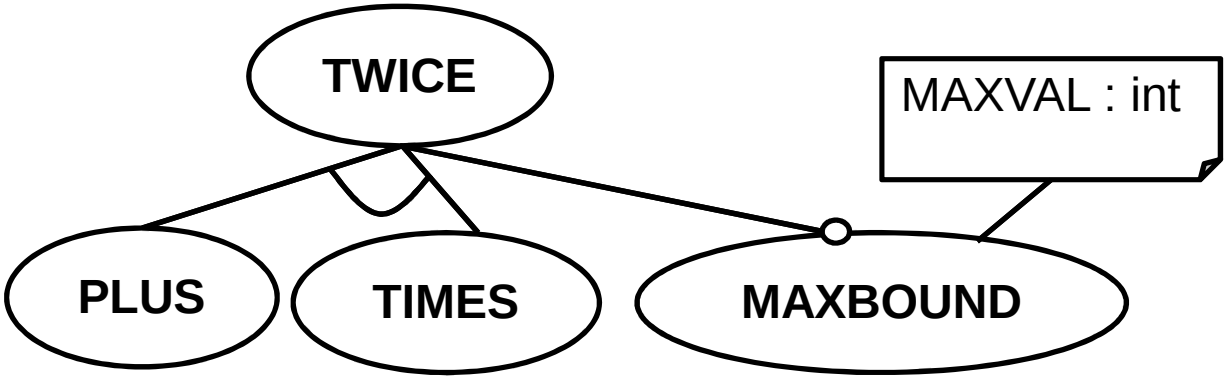
SPL Implementierung mit CPP

1. Feature Modell erstellen



SPL Implementierung mit CPP

2. Feature-Modell als Makro kodieren



Mandatory Root Feature

Feature-Makros

Alternativ-Gruppe

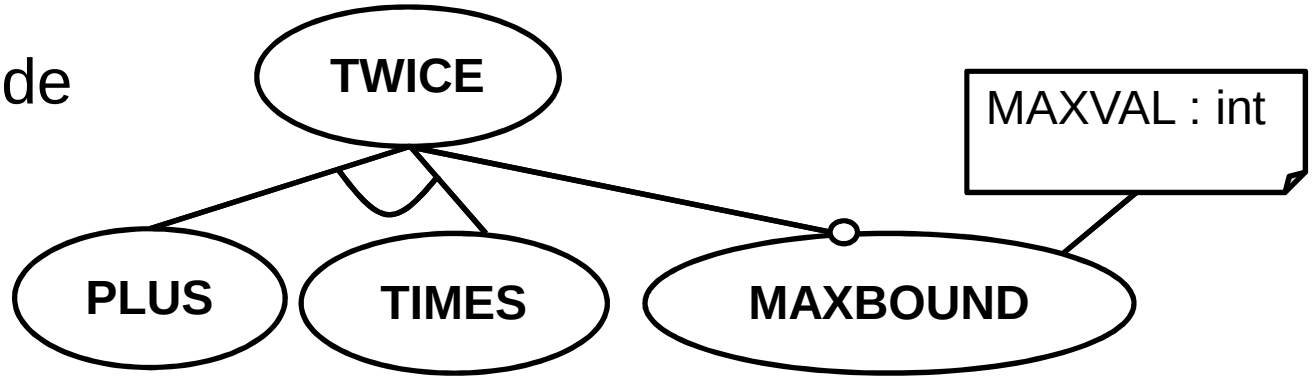
```
#if      ( defined (TWICE)  &&
          ((defined (PLUS)   && ! defined (TIMES)) ||
           (defined (TIMES)  && ! defined (PLUS) ))
           (defined (MAXVAL) || ! defined (MAXBOUND))
#define VALIDCONFIG
#endif
```

Makro gesetzt, falls Konfiguration valide

Attributwert für MAXVAL gesetzt, wenn MAXBOUND gewählt

SPL Implementierung mit CPP

3. Parametrisierter C-Code

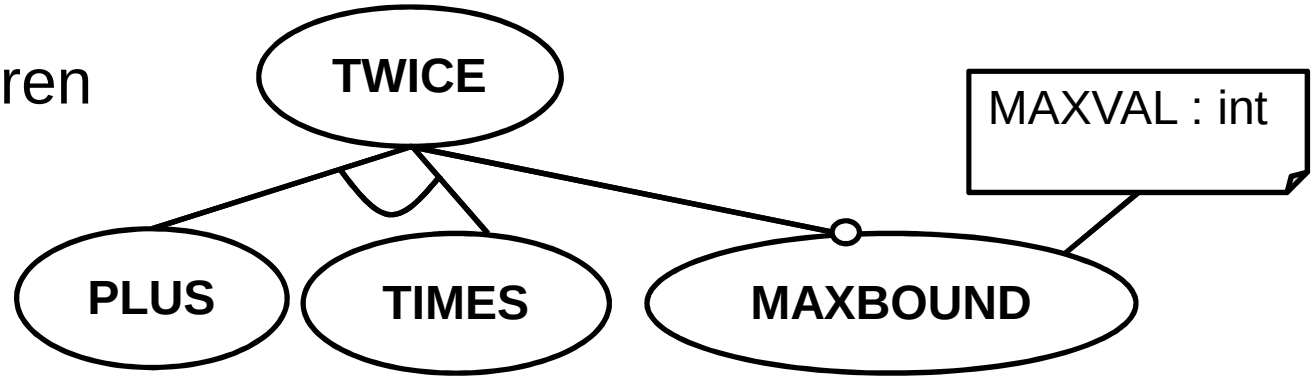


Nur für valide Konfigurationen wird ein Programmvariante erzeugt

```
#ifdef VALIDCONFIG
int twice(int x) {
    int y = #ifdef    PLUS    x+x;
            #elif     TIMES   2*x;
            #endif
    #ifdef MAXBOUND
    if (y > MAXVAL) return MAXVAL;
    else
    #endif
    return y;
}
#endif
```

SPL Implementierung mit CPP

4. Konfigurationen kodieren



Beliebiger Wert

P1 = { TWICE, PLUS }

```
#define TWICE  1
#define PLUS   1
```

P2 = { TWICE, TIMES }

```
#define TWICE  1
#define TIMES  1
```

P3 = { TWICE, PLUS, MAXBOUND, MAXVAL=1020 }

```
#define TWICE      1
#define PLUS       1
#define MAXBOUND   1
#define MAXVAL     1020
```

P4 = { TWICE, TIMES, MAXBOUND, MAXVAL=1020 }

```
#define TWICE      1
#define TIMES      1
#define MAXBOUND   1
#define MAXVAL     1020
```

SPL Implementierung mit CPP

5. Programmvarianten

twice_spl.c

```
#ifdef VALIDCONFIG
int twice(int x) {
    int y =   #ifdef PLUS   x+x;
              #elif  TIMES 2*x;
              #endif
    #ifdef   MAXBOUND
    if (y > MAXVAL) return MAXVAL;
    else
    #endif
    return y;
}
#endif
```

+

config1.h

```
#define TWICE 1
#define PLUS  1
```

→

```
int twice(int x) {
    int y = x+x;
    return y;
}
```

+

config2.h

```
#define TWICE 1
#define TIMES 1
```

→

```
int twice(int x) {
    int y = 2*x;
    return y;
}
```

+

config3.h

```
#define TWICE     1
#define PLUS      1
#define MAXBOUND  1
#define MAXVAL    1020
```

→

```
int twice(int x) {
    int y = x+x;
    if (y > 1020) return 1020;
    else return y;
}
```

+

config4.h

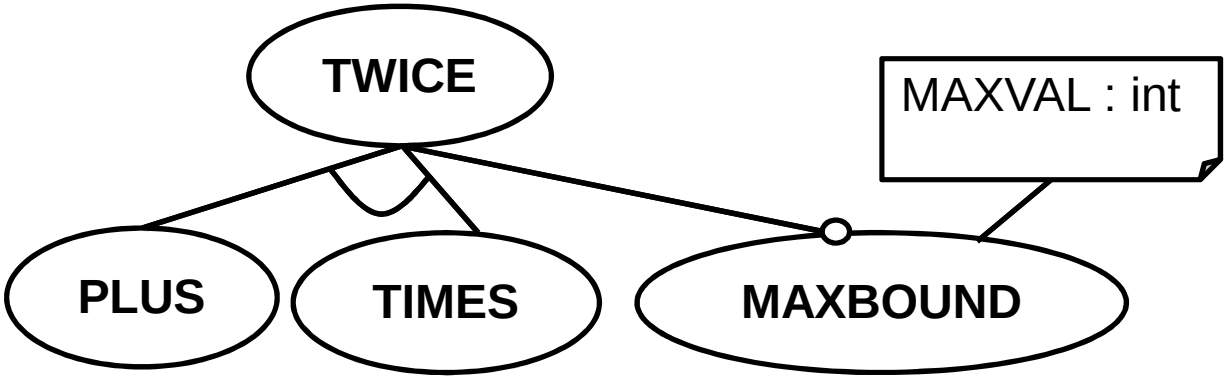
```
#define TWICE     1
#define TIMES     1
#define MAXBOUND  1
#define MAXVAL    1020
```

→

```
int twice(int x) {
    int y = 2*x;
    if (y > 1020) return 1020;
    else return y;
}
```

SPL Implementierung mit Variability Encoding in C

1. Feature-Modell als Makro mit Argumenten kodieren




```
#define FM(TWICE,PLUS,TIMES,MAXBOUND,MAXVAL)
          (TWICE && ((PLUS && !TIMES) || (TIMES  && !PLUS))
          (MAXVAL || !MAXBOUND))
```

Annahme: MAXVAL soll einen Wert größer 0 haben, wenn MAXBOUND ausgewählt wird

SPL Implementierung mit Variability Encoding in C

```
int twice(int x) {
    int y =  #ifdef   PLUS     x+x;
             #elif    TIMES    2*x;
             #endif
    #ifdef   MAXBOUND
    if (y > MAXVAL) return MAXVAL;
    else
    #endif
    return y;
}
```

distribute

```
int twice(int x) {
    #ifdef   PLUS  int y = x+x;
    #elif    TIMES int y = 2*x;
    #endif
    #ifdef   MAXBOUND
    if (y > MAXVAL) return MAXVAL;
    else
    #endif
    return y;
}
```

distribute + split

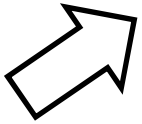
```
int twice(int x) {
    int y = #ifdef   PLUS     x+x;
            #elif    TIMES    2*x;
            #endif
    #ifdef   MAXBOUND
    if (y > MAXVAL) return MAXVAL;
    else return y;
    #else
    return y;
    #endif
}
```

```
int twice(int x) {
    #ifdef   PLUS  int y = x+x;
    #elif    TIMES int y = 2*x;
    #endif
    #ifdef   MAXBOUND
    if (y > MAXVAL) return MAXVAL;
    else return y;
    #else
    return y;
    #endif
}
```

2. Granularität verringern, bis Variabilität durch if-Statements kodierbar ist

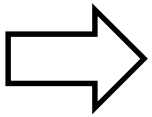
SPL Implementierung mit Variability Encoding in C

3. Makros als if-Statements kodieren



```
int main() {
    if(!FM(TWICE,PLUS,TIMES,MAXBOUND,MAXVAL))
        return -1;
    else
        // Code mit Aufruf von twice
}
```

```
#ifdef VALIDCONFIG
int twice(int x) {
    #ifdef   PLUS  int y = x+x;
    #elif    TIMES int y = 2*x;
    #endif
    #ifdef   MAXBOUND
    if (y > MAXVAL) return MAXVAL;
    else return y;
    #else
    return y;
    #endif
}
#endif
```

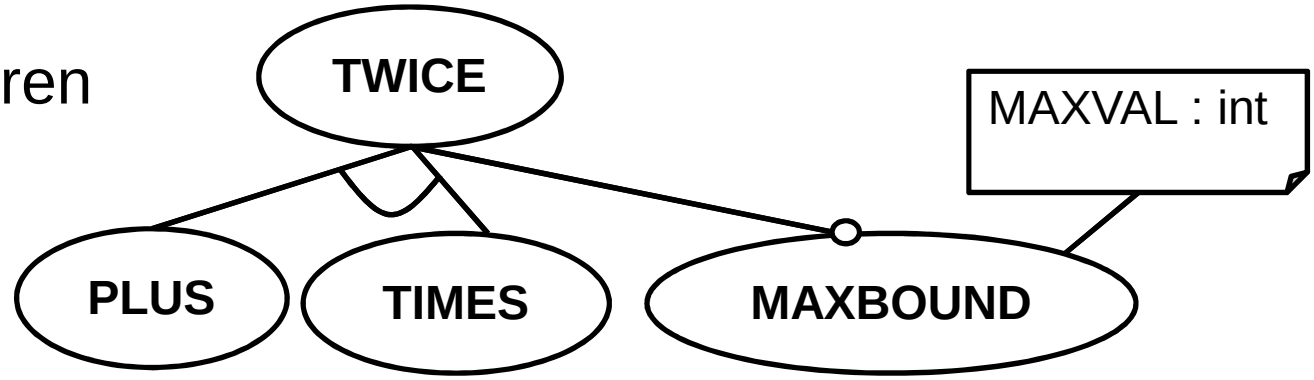


```
int twice(int x) {
    if(PLUS)  int y = x+x;
    else if(TIMES) int y = 2*x;

    if(MAXBOUND) {
      if (y > MAXVAL) return MAXVAL;
      else return y;
    }
    else {
      return y;
    }
}
```

SPL Implementierung mit CPP

4. Konfigurationen kodieren



P1 = { TWICE, PLUS }

```
int TWICE      = 1;
int PLUS       = 1;
int TIMES      = 0;
int MAXBOUND   = 0;
int MAXVAL     = 0;
```

P2 = { TWICE, TIMES }

```
int TWICE      = 1;
int PLUS       = 0;
int TIMES      = 1;
int MAXBOUND   = 0;
int MAXVAL     = 0;
```

P3 = { TWICE, PLUS, MAXBOUND, MAXVAL=1020 }

```
int TWICE      = 1;
int PLUS       = 1;
int TIMES      = 0;
int MAXBOUND   = 1;
int MAXVAL     = 1020;
```

P4 = { TWICE, PLUS, MAXBOUND, MAXVAL=1020 }

```
int TWICE      = 1;
int PLUS       = 0;
int TIMES      = 1;
int MAXBOUND   = 1;
int MAXVAL     = 1020;
```

Laufzeitvariabilität durch Variability Encoding

5. Programmvarianten

twice_spl.c

```
int main() { ... }
int twice(int x) {
    if(PLUS)  int y = x+x;
    else if(TIMES) int y = 2*x;

    if(MAXBOUND) {
    if (y > MAXVAL) return MAXVAL;
    else return y;
    }
    else {
    return y;
    }
}
```

Notation: ≃ gilt, wenn sich das Programm zur Laufzeit äquivalent zur Variante verhält.

\+

config1.h

```
int TWICE    = 1;
int PLUS     = 1;
int TIMES    = 0;
int MAXBOUND = 0;
int MAXVAL   = 0;
```

≃

```
int twice(int x) {
    int y = x+x;
    return y;
}
```

\+

config2.h

```
int TWICE    = 1;
int PLUS     = 0;
int TIMES    = 1;
int MAXBOUND = 0;
int MAXVAL   = 0;
```

≃

```
int twice(int x) {
    int y = 2*x;
    return y;
}
```

\+

config3.h

```
int TWICE    = 1;
int PLUS     = 1;
int TIMES    = 0;
int MAXBOUND = 1;
int MAXVAL= 1020;
```

≃

```
int twice(int x) {
    int y = x+x;
    if (y > 1020) return
1020;
    else return y;
}
```

\+

config4.h

```
int TWICE    = 1;
int PLUS     = 0;
int TIMES    = 1;
int MAXBOUND = 1;
int MAXVAL= 1020;
```

≃

```
int twice(int x) {
    int y = 2*x;
    if (y > 1020) return
1020;
    else return y;
}
```

Präprozessoren für Java

- Nicht nativ vorhanden
- Bedingte Kompilierung im Java-Compiler nur auf Statement-Ebene, nicht für Klassen, Methoden und Felder

```
class Example {
    public static final boolean DEBUG = false;

    void main() {
        System.out.println("immer");
        if (DEBUG)
            System.out.println("debug info");
    }
}
```

„Dead“ Code wird durch Java-Compiler entfernt

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

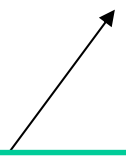
MUNGE

- Einfacher Präprozessor für Java Code
- Ursprünglich für Swing in Java 1.2 eingeführt

http://weblogs.java.net/blog/tball/archive/2006/09/munge_swings_se.html

```
class Example {
    void main() {
        System.out.println("immer");
        /*if[DEBUG]*/
        System.out.println("debug info");
        /*end[DEBUG]*/
    }
}
```

```
> java Munge -DDEBUG -DFEATURE2 Datei1.java Datei2.java ... Zielverzeichnis
```



Konfiguration: Feature-Auswahl aus Feature-Modell

Beispiel: Graph SPL

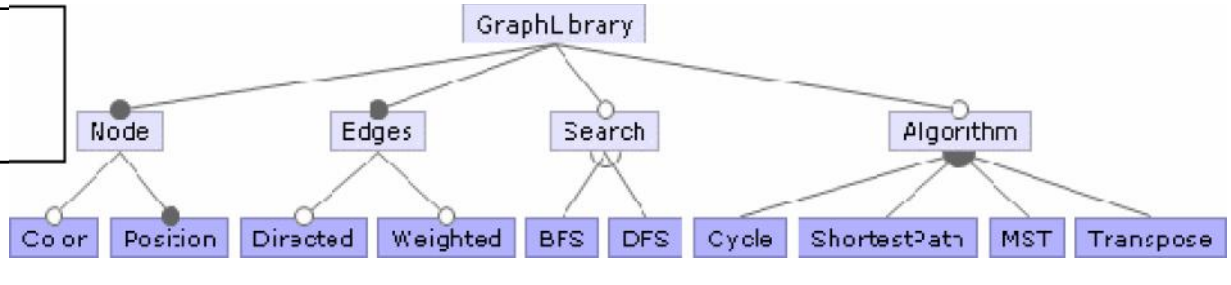
```
class Graph {
  Vector nv = new Vector(); Vector ev = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m); ev.add(e);
    e.weight = new Weight();
    return e;
  }
  Edge add(Node n, Node m, Weight w)
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m);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

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

```
class Color {
  static void setDisplayColor(Color c) { ... }
}
```

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



Graph SPL mit MUNGE

```
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);
    /*if[WEIGHT]*/
    e.weight = new Weight();
    /*end[WEIGHT]*/
    return e;
  }
  /*if[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;
  }
  /*end[WEIGHT]*/
  void print() {
    for(int i = 0; i < ev.size(); i++) {
      ((Edge)ev.get(i)).print();
    }
  }
}
```

```
class Edge {
  Node a, b;
  /*if[COLOR]*/
  Color color = new Color();
  /*end[COLOR]*/
  /*if[WEIGHT]*/
  Weight weight;
  /*end[WEIGHT]*/
  Edge(Node _a, Node _b) { a = _a; b = _b; }
  void print() {
    /*if[COLOR]*/
    Color.setDisplayColor(color);
    /*end[COLOR]*/
    a.print(); b.print();
    /*if[WEIGHT]*/
    weight.print();
    /*end[WEIGHT]*/
  }
}
```

```
class Node {
  int id = 0;
  /*if[COLOR]*/
  Color color = new Color();
  /*end[COLOR]*/
  void print() {
    /*if[COLOR]*/
    Color.setDisplayColor(color);
    /*end[COLOR]*/
    System.out.print(id);
  }
}
```

```
/*if[COLOR]*/
class Color {
  static void setDisplayColor(Color c) { ... }
}
/*end[COLOR]*/
```

```
/*if[WEIGHT]*/
class Weight { void print() { ... } }
/*end[WEIGHT]*/
```

XVCL

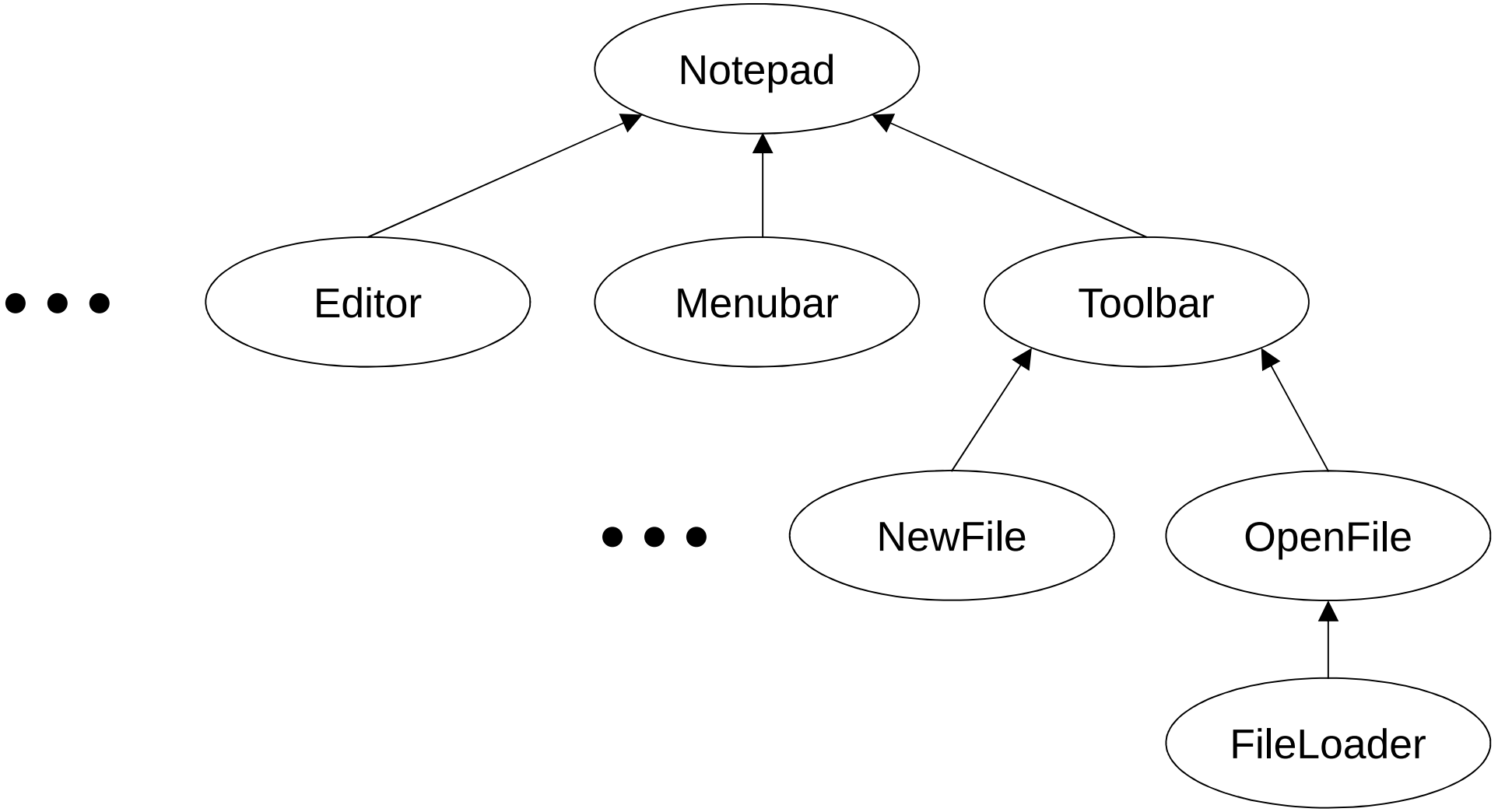
- XML-basierter Präprozessor für Java
- Schwerpunkt GUI-Entwicklung

```
<x-frame name="Notepad">
import java.awt.*;
class Notepad extends JPanel {
   Notepad() {
      super();
      …
   }
   public static void main(String[] args) {
      JFrame frame = new JFrame();
      frame.setTitle("<value-of expr="?@TITLE?"/>");
      frame.setBackground(Color.<value-of
expr="?@BGCOLOR?"/>);
      frame.show();
   }
   <adapt x-frame="Editor.XVCL"/>
   <adapt x-frame="Menubar.XVCL"/>
   <adapt x-frame="Toolbar.XVCL"/>
   …
}
</x-frame>
```

```
<x-frame name="Toolbar">
<set-multivar="ToolbarBtns" value="New,Open,Save, -,
Exit"/>
private Component createToolbar() {
         JToolBar toolbar = new JToolBar();
         JButton button;
<while using-items-in="ToolbarBtns">
   <select option="ToolbarBtns">
     <option value="-">
toolbar.add(Box.createHorizontalStrut(5));//separator
     </option>
     <otherwise>
button = new JButton(new ImageIcon("<value-of
expr="?@Gif@ToolbarBtns?"/> "));
toolbar.add(button);
      </otherwise>
   </select>
</while>
   toolbar.add(Box.createHorizontalGlue());
   return toolbar;
}
</x-frame>
```

XVCL

- XVCL basiert auf Frame-Hierarchie



Antenna

- Präprozessor-Direktive **`#ifdef`** wie in *cpp*
- Sammlung von Ant-Tasks für Java ME
- In vielen Java ME Projekten eingesetzt

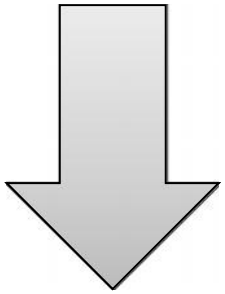
Bedingte Kompilierung objektorienter Programm-Elemente in Java

- Klassen und Interfaces
- Methoden und Member-Variablen
- Klassenvariablen und statische Methoden
- Statements
- …

```
        /** Read HTML and if it has links, redirect and parse the XML. */
        protected String parseHTMLRedirect(String url, InputStream is)
    throws IOException, Exception {
                //#ifdef DSMALLMEM
//#             throw new IOException("Error HTML not supported with this version.");
                //#else
                if (m_redirect) {
                        //#ifdef DLOGGING
//#                     logger.severe("Error 2nd redirect url:  " + url);
                        //#endif
                        System.out.println("Error 2nd redirect url:  " + url);
                        throw new IOException("Error url " + m_redirectUrl +
                                              " to 2nd redirect url:  " + url);
                }
                m_redirect = true;
                m_redirectUrl = url;
                com.substanceofcode.rssreader.businessentities.RssItunesFeed[] feeds =
                                HTMLLinkParser.parseFeeds(new EncodingUtil(is),
                                                                        url, null, null, true
                                                                        //#ifdef DLOGGING
//#                                                                     ,logger,
//#                                                                     fineLoggable,
//#                                                                     finerLoggable,
//#                                                                     finestLoggable
                                                                        //#endif
                                                                        );
                if ((feeds == null) || (feeds.length == 0)) {
```

Antenna - Beispiel

```
//#ifdef midp20
//# import javax.microedition.lcdui.game.Sprite;
//#endif
//#ifdef siemens
//# import com.siemens.mp.game.Sprite
//#endif
```



```
<wtkpreprocess
srcdir="src“ destdir="siemens“
symbols="siemens“ verbose="true"
/>
```

(build.xml)

```
//#ifdef midp20
//# import javax.microedition.lcdui.game.Sprite;
//#endif
//#ifdef siemens
import com.siemens.mp.game.Sprite
//#endif
```

Vorteile von Präprozessoren

- In vielen Sprachen bereits enthalten / einfach verwendbare Tools
- Den meisten Entwicklern bereits bekannt
- Sehr einfaches Programmierkonzept: *markieren und entfernen*
- Sehr flexibel / ausdrucksstark, beliebige Granularität
- Nachträgliche Einführung von Variabilität in bestehendes Projekt einfach

Nachteile: Unleserlicher Code

- Vermischung von zwei Sprachen (C und #ifdef, oder Java und Munge, …)
- Kontrollfluss schwer nachvollziehbar
- Lange Annotationen schwer zu finden
- Zusätzliche Zeilenumbrüche zerstören Layout
- Vermischung von syntaktischer Variation und Programmsemantik

Alternativ: Feature-Code separieren/modularisieren?

```
class Stack {
    void push(Object o
#ifdef SYNC
    , Transaction txn
#endif
    ) {
        if (o==null
#ifdef SYNC
            || txn==null
#endif
            ) return;
#ifdef SYNC
        Lock l=txn.lock(o);
#endif
        elementData[size++] = o;
#ifdef SYNC
        l.unlock();
#endif
        fireStackChanged();
    }
}
```

Weitere Nachteile

- Hohe Komplexität durch beliebige Schachtelung
- Fehleranfälligkeit durch Komplexität und unstrukturierten (**„undisziplinierten“)** Einsatz

Beispiele:

- Variabler Rückgabetyp => Typanalyse?

```
Edge/*if[WEIGHT]*/Weight/*end[WEIGHT]*/ add(Node n, Node m /*if[WEIGHT]*/, int w/*end[WEIGHT]*/) {
   return new Edge/*if[WEIGHT]*/Weight/*end[WEIGHT]*/ (n, m /*if[WEIGHT]*/, w/*end[WEIGHT]*/ );
}
```

- „Derivate“-Kommata bei Separierung variabler Parameter

```
Edge set(/*if[WEIGHT]*/int w/*if[COLOR]*/, /*end[COLOR]/*end[WEIGHT]*/
        /*if[COLOR]*/int c/*end[COLOR]*/) {
   …
}
```

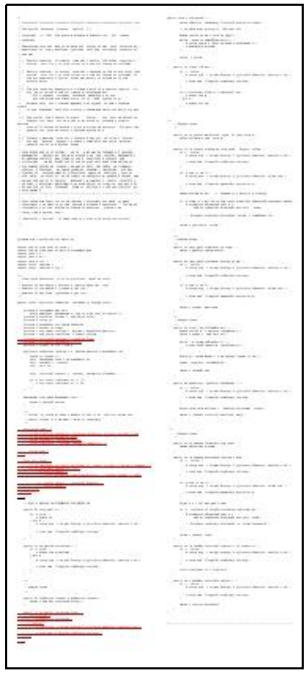
Weitere Nachteile

- Feature-Code ist komplett verstreut
  - Feature-Traceability-Problem
  - Beispiel Graph-SPL: Wie findet man einen Fehler in der Implementierung des *Weight* Features?

- Verhindert/erschwert Tool Support
  - Negative Erfahrungen bereits belannt von der Analyse von C/C++ (Refactoring, Typ-Analyse, …)
  - MUNGE und andere: Definition in Kommentaren

A Matter of Size…

ApplicationSession



StandardSession

ServerSession

Beispiel: Zeitmanagement von Sessions im Apache Tomcat Server als Bestandteil der Session-Verwaltung

SessionInterceptor

ServerSessionManager

StandardManager

StandardSessionManager

Kritik an Präprozessoren

Designed in the 70th and hardly evolved since

“#ifdef considered harmful”

“#ifdef hell”

“maintenance becomes a ‘hit or miss’ process”

“is difficult to determine if the code being viewed is actually compiled into the system”

“incomprehensible source texts”

“programming errors are easy to make and difficult to detect”

“CPP makes maintenance difficult”

“source code rapidly becomes a maze”

“preprocessor diagnostics are poor”

Traceability-Problem

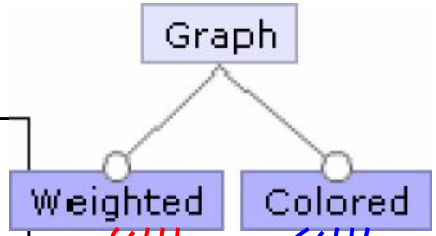
```
class Graph {
  Vector nv = new Vector(); Vector ev = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m); ev.add(e);
    e.weight = new Weight();
    return e;
  }
  Edge add(Node n, Node m, Weight w)
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m);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

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

```
class Color {
  static void setDisplayColor(Color c) { ... }
}
```

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



Sicherheitsrisiken

```
class BusinessClass
    //... Datenfelder
    //... Logging Stream
    //... Cache Status
    public void importantOperation(
                  Data data, User currentUser,
...){
        // prüfe Autorisierung
        // Objekt sperren für Synchronisation
        // Aktualität des Puffers prüfen
        // Start der Operation loggen
        // eigentliche Operation ausführen
        // Ende der Operation loggen
        // Sperre auf Objekt freigeben
    }
    public void alsoImportantOperation(
                  OtherData data, User currentUser,
...){
        // prüfe Autorisierung
        // Objekt sperren für Synchronisation
        // Aktualität des Puffers prüfen
        // Start der Operation loggen
        // eigentliche Operation ausführen
        // Ende der Operation loggen
        // Sperre auf Objekt freigeben
    }
}
```

- Welcher Code gehört zur Authentifizierung?
- Das Sperrverfahren soll geändert werden: welcher Code muss angepasst werden?
- Nutzer konnte ohne Anmeldung Daten löschen: wo Fehler suchen?

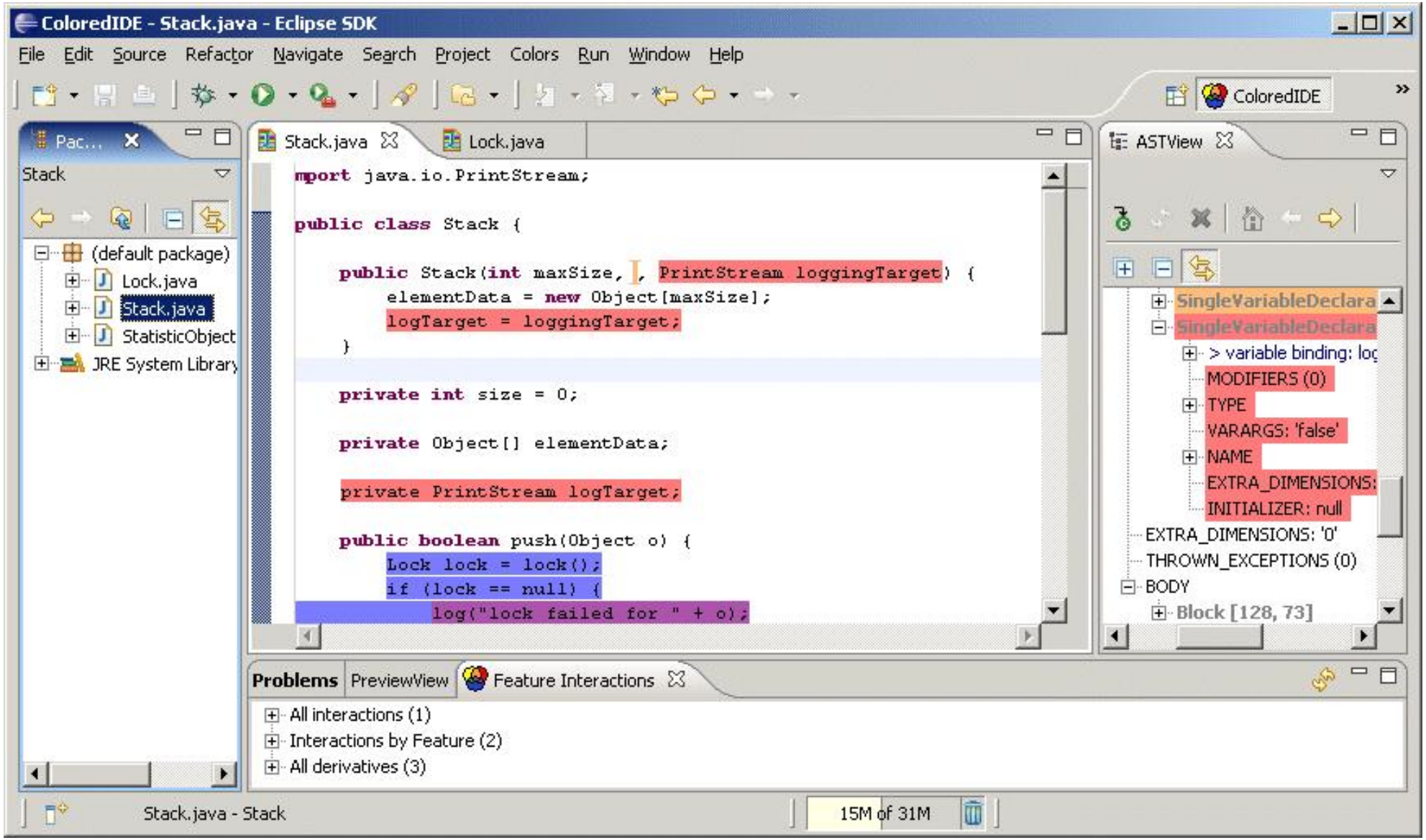
Verstreute Feature-Implementationen

- Feature „verschwinden“ in der Implementierung
  - Was „gehört“ direkt/indirekt zu einem Feature?
  - Welche Features interagieren zur Laufzeit?
  - Bei Wartungsaufgaben muss schlimmstenfalls der komplette Quelltext durchsucht werden
- Schwierige Arbeitsteilung
  - Für unterschiedliche Features kann es unterschiedliche Experten geben; alle müssen an den gleichen Code-Fragmenten arbeiten
- Geringere Produktivität, schwierige Evolution
  - Beim Hinzufügen neuer Features muss sich der Entwickler um viele andere Belange kümmern, die von der eigentlichen SPL-Evolution ablenken (Lesbarkeit, Erfassbarkeit)

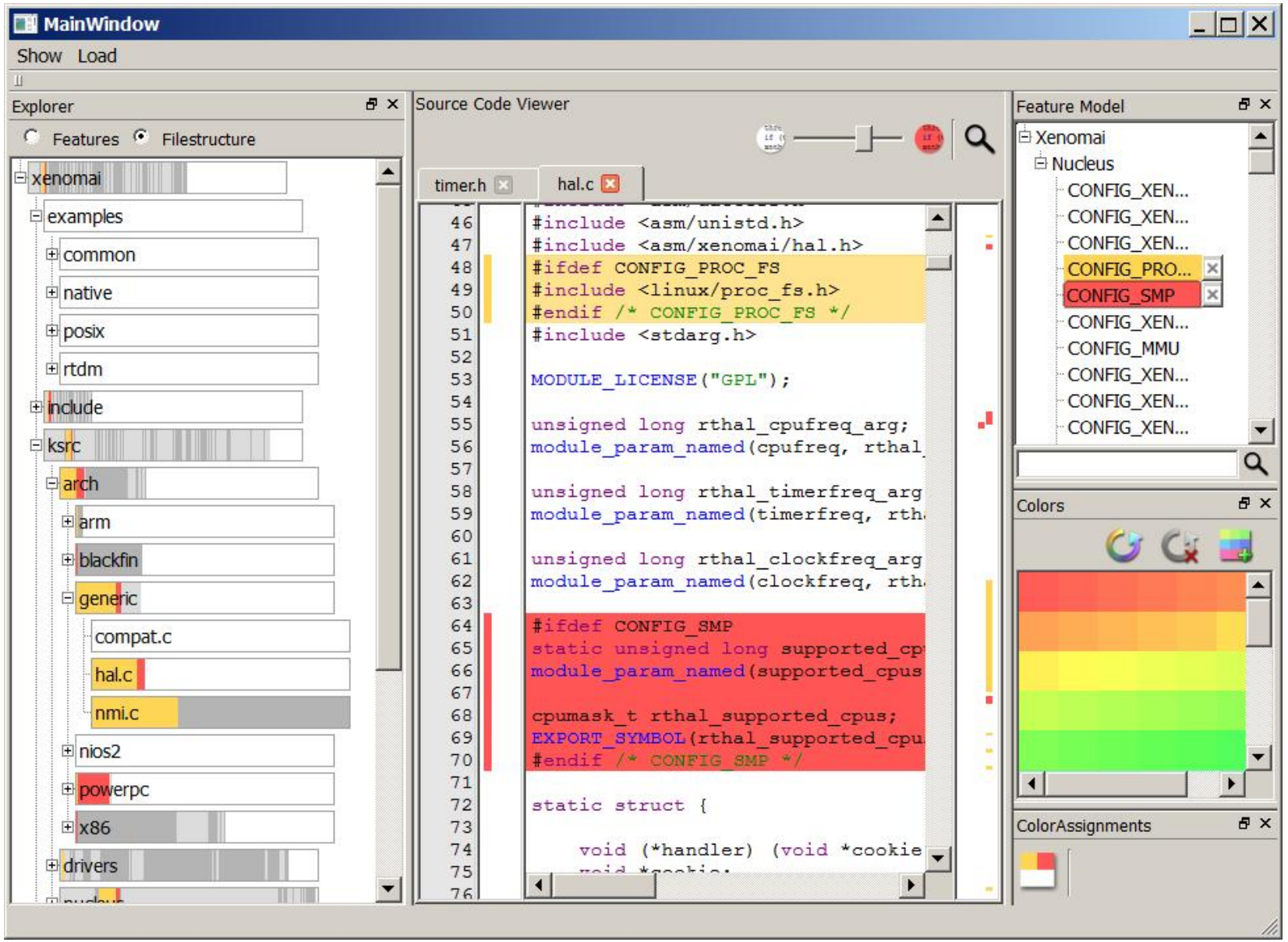
Feature-Traceability

- Ziel: Zusammenhang zwischen Code und Features explizit machen
- Im Idealfall: pro Feature ein (virtuelles) **Modul**
- Andernfalls sind Umwege und Notlösungen unausweichlich, wenn Modularisierung nicht möglich
  - Kommentare oder Annotationen im Quelltext (z. B. jeglicher Authentifizierungscode wird mit „**`//auth`**“ markiert)
  - Namenskonventionen (z. B. alle Authentifizierungsmethoden fangen mit „**`auth_`**“ an)
  - Zusätzliche Werkzeuge, z. B. IDE-Unterstützung (Highlightning, Views, Filter, Abhängigkeitsgraphen etc.)

CIDE


ColoredIDE - Stack.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Colors Run Window Help
ColoredIDE
Pac...
Stack
(default package)
Lock.java
Stack.java
StatisticObject
JRE System Library
Stack.java
Lock.java
mport java.io.PrintStream;
public class Stack {
public Stack(int maxSize, , PrintStream loggingTarget) {
elementData = new Object[maxSize];
logTarget = loggingTarget;
}
private int size = 0;
private Object[] elementData;
private PrintStream logTarget;
public boolean push(Object o) {
Lock lock = lock();
if (lock == null) {
log("lock failed for " + o);
ASTView
SingleVariableDeclara
SingleVariableDeclara
> variable binding: loc
MODIFIERS (0)
TYPE
VARARGS: 'false'
NAME
EXTRA_DIMENSIONS:
INITIALIZER: null
EXTRA_DIMENSIONS: '0'
THROWN_EXCEPTIONS (0)
BODY
Block [128, 73]
Problems
PreviewView
Feature Interactions
All interactions (1)
Interactions by Feature (2)
All derivatives (3)
Stack.java - Stack
15M of 31M

FeatureCommander

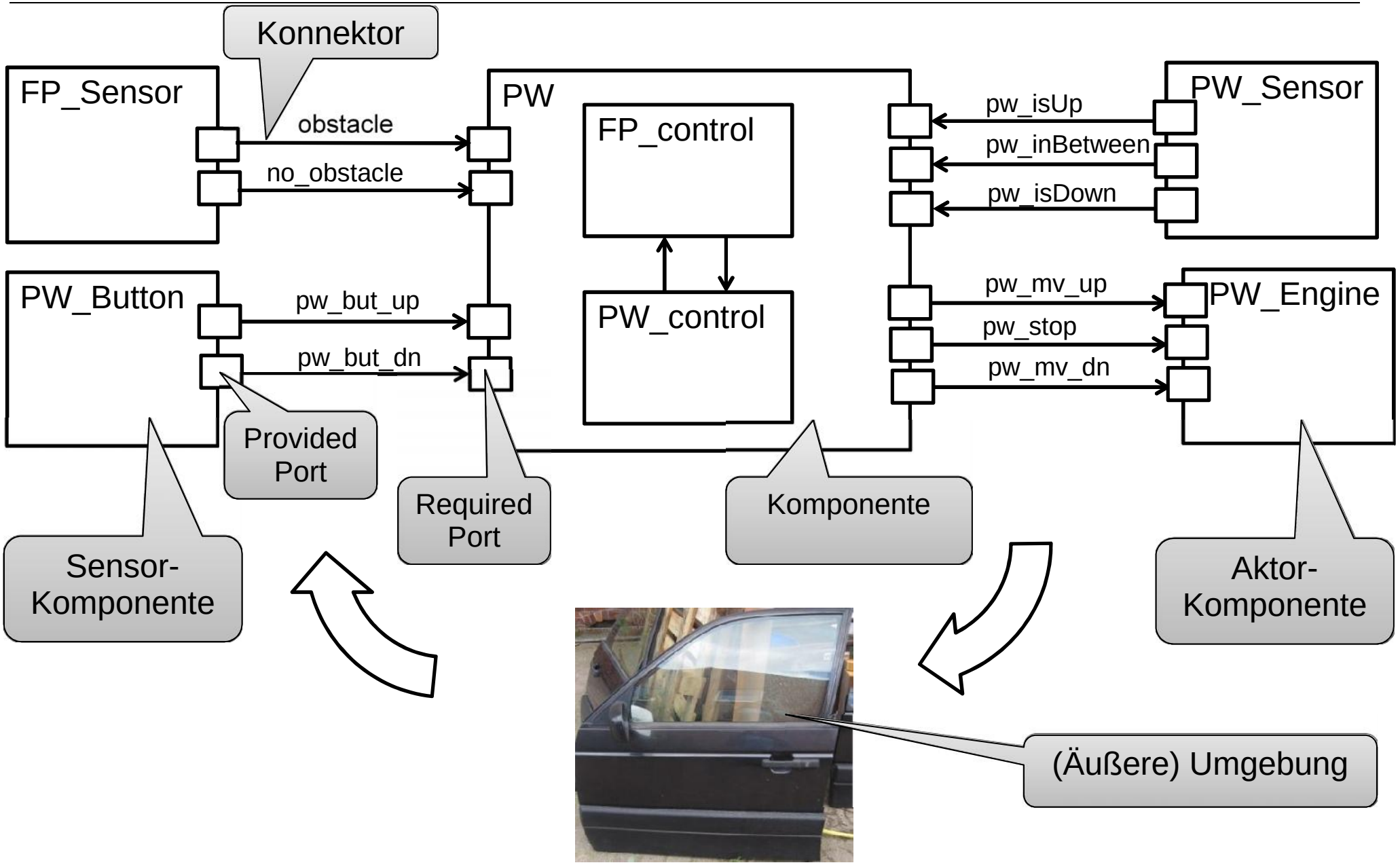


Komponenten-Systeme

Komponenten sind abgeschlossene, modulare Implementierungseinheiten mit fester **Schnittstelle** (Black-Box Abstraktionssicht)

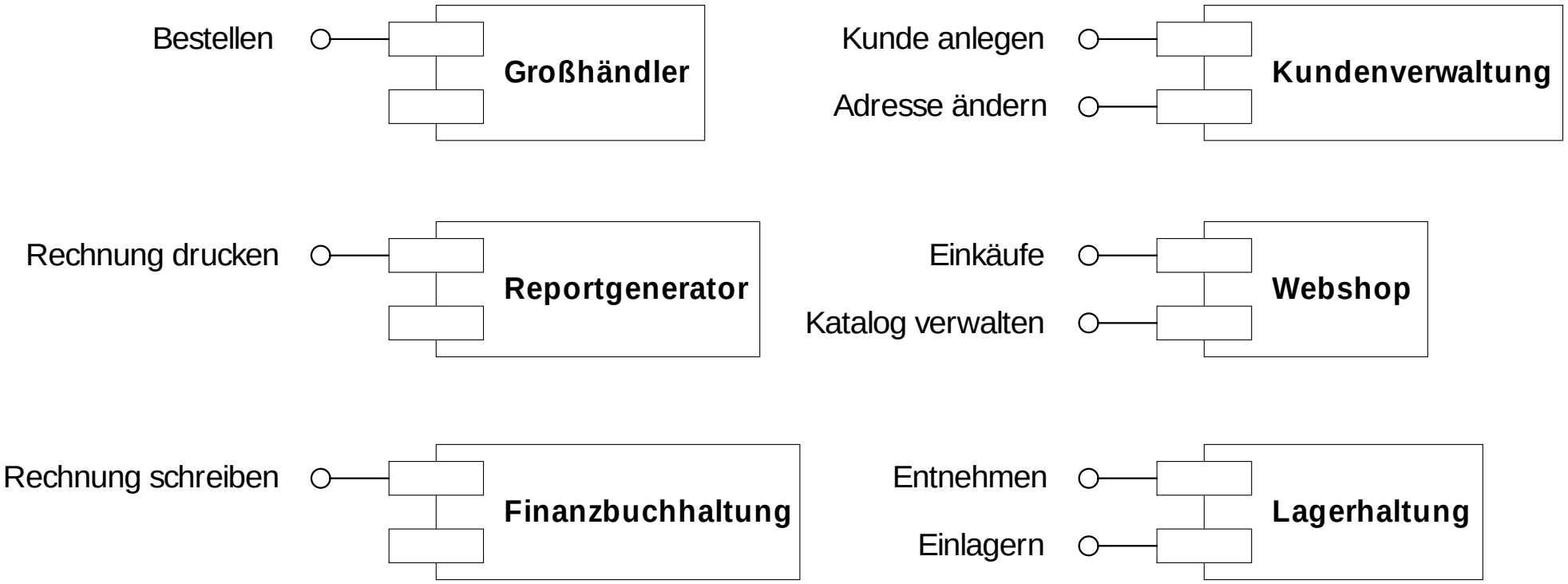
- Komponenten bieten einen „**Dienst**“ an die Umgebung an
- Komponenten werden zusammen mit anderen **kompatiblen** Komponenten – auch von anderen Herstellern – „zusammengesetzt“ zu Softwaresystemen (**Komposition und Aggregation**)
- Der **Kontext** (z. B. JavaEE, CORBA, COM+/DCOM, OSGi) und die **Abhängigkeiten** (imports, exports) von Komponenten werden explizit spezifiziert
- Klein genug für Erzeugung und Wartung in einem Stück, groß genug um sinnvolle Funktion bieten zu können

Beispiel: BCS-Komponenten



Beispiel: Komponenten eines Webshops

Komponentnbeschreibung in UML-Notation:



Szenario: Kunde anlegen -> Einkauf -> Rechnung schreiben -> Rechnung drucken

Komponenten vs. Objekte/Klassen

- Ähnliche Konzepte: Kapselung, Interfaces, Geheimnisprinzip
  - Objekte strukturieren ein Problem
  - Komponenten bieten anwendbare Teilfunktionalität
- Objekte sind üblicherweise „kleiner“ als Komponenten: „Komponenten skalieren OOP“
- Objekte haben häufig viele (implizite) Abhängigkeiten zu anderen Objekten; Komponenten haben wenige, explizite Abhängigkeiten (lose Kopplung)
- Interfaces von Objekten sind häufig implementierungsnah; Komponenten sind auf einer höheren Abstraktionsebene

Vision: Marketplaces for Components

- Komponenten (Hardware und/oder Software) können einzeln gekauft und beliebig in eigenen Produkte kombiniert / integriert werden
- *Best of Breed Prinzip*: Entwickler kann für jedes Teilsystem den besten/billigsten Anbieter auswählen
- Anbieter können sich auf Kernkompetenz konzentrieren und diese als Komponente anbieten
- Siehe z.B. AUTOSAR Standard im automotive Bereich: http://www.autosar.org/

Produktlinien als Komponenten-Systeme

- Jedes Feature wird jeweils in einer Komponenten implementiert (1:1 Mapping)
- Eine Produktvariante entspricht einer **Komposition**
  - Durch Feature-Auswahl werden Komponenten **selektiert**
  - Der Entwickler muss ggf. Komponenten „von Hand“ verbinden (sog. **Glue Code**)

- Beispiel: Datenbank-Produktlinie
  - Komponenten: Transaktionsverwaltung, Log/Recovery, Pufferverwaltung, Optimierer, …
  - Komponenten können ggf. Laufzeitvariabilität enthalten

Beispiel: Component Color in Java

```
package modules.colorModule;
// public interface
public class ColorModule {
    public Color createColor(r:Int,g:Int,b:Int) { …}
    public void printColor(color: Color) {colorPrint… }

    public void mapColor(elem: Object, col: Color)
            { colorMapping…}
     public Color getColor(elem: Object)
            { colorMapping…}

    // just one module instance
    public static ColorModule getInstance()
            { return module; }
    private static ColorModule module =
            new ColorModule();
     private ColorModule() { super(); }
}
public interface Color { … }

// hidden implementation
class ColorPrinter { … }
class ColorMapping {…}
```

- Facade Pattern
  - Versteckt Implementierungsdetails
  - Gemeinsames Interface für viele Klassen
- Singleton Pattern
  - Nur eine Instanz des Moduls

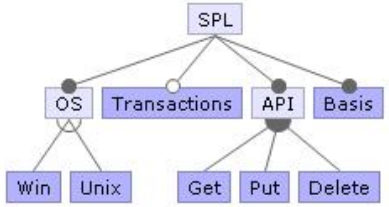
```
ColorModule.getInstance().createColor(…)
```

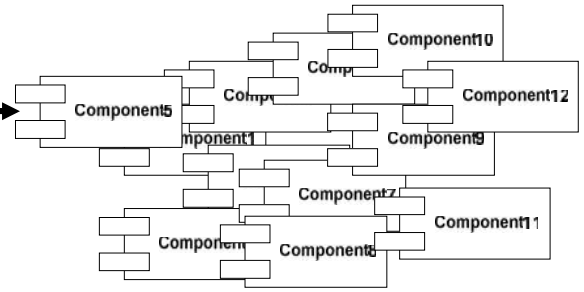
Produktlinien als Komponenten-Systeme

Mapping von Features zu Komponenten

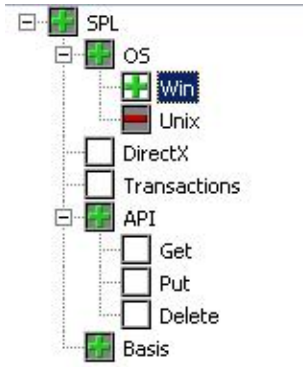
Komponentensammlung

Feature-Modell

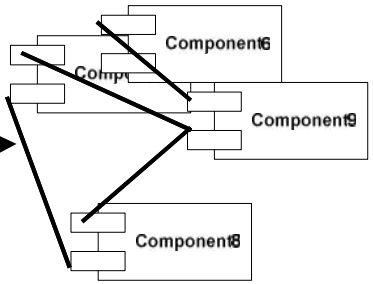


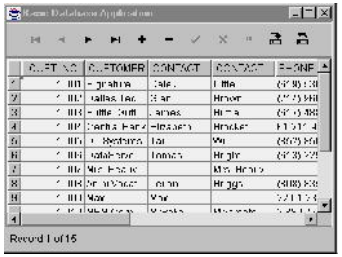


Programmvariante durch Komposition erzeugen



Feature-Auswahl als Eingabe





Feature-Auswahl

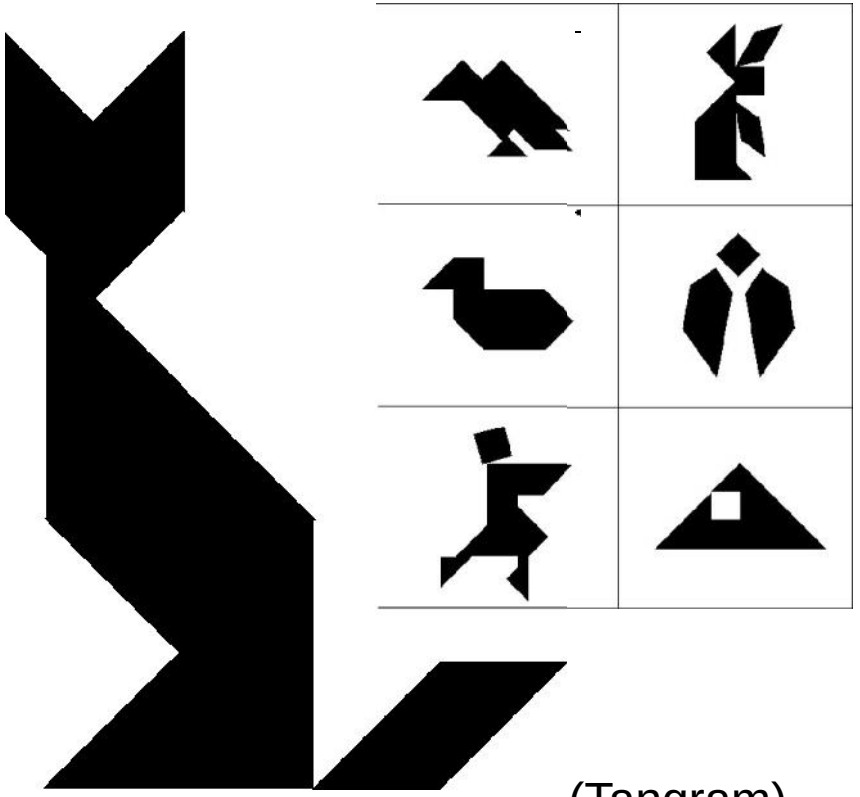
Komponentenauswahl

Menge von Komponenten

Entwurf von Komponenten

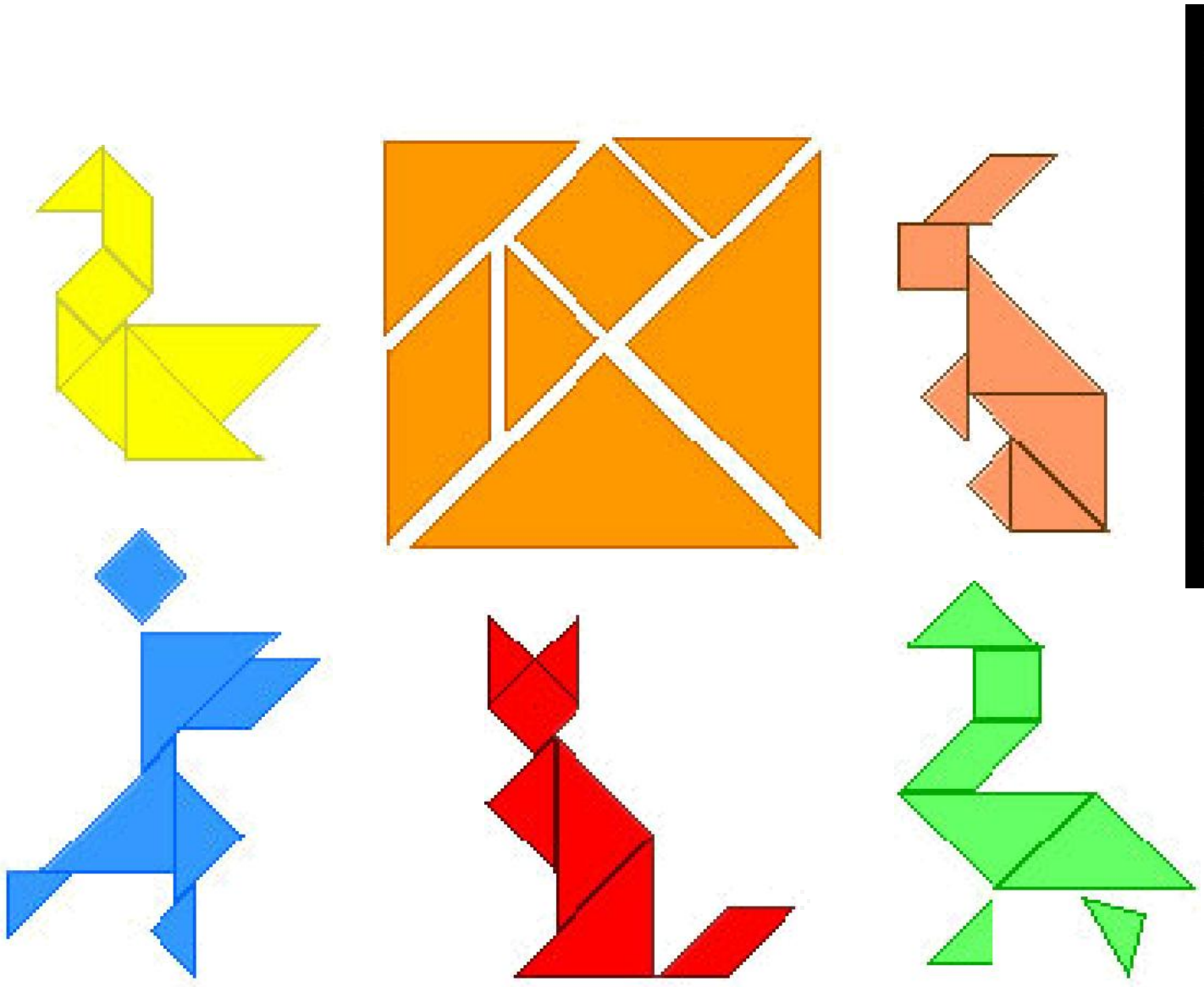
- Größe und Scope einer Komponente?
- Zu kleine Komponente → hoher Aufwand
- Zu große Komponente → kaum wiederverwendbar

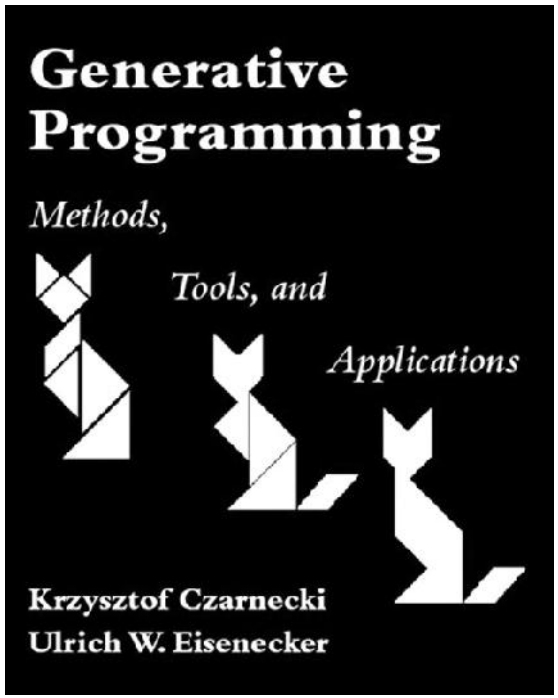
- Produktlinien liefern nötige **Domänenanalyse**
  - Welche Teilfunktionalität wird in welcher Granularität wiederverwendet?
  - Welche Abhängigkeiten bestehen?



(Tangram)

Literatur zum Thema





Diskussion: Produktlinien als Komponenten-Systeme

- In der Industrie weit verbreitet
  Beispiel: Heimelektronik mit Koala-Komponenten von Phillips
- Produktlinien-Ansätze zum Domain Engineering, zur Organisation der Entwicklung, …
- Systematische (geplante) Wiederverwendung von Komponenten durch Modularisierung
- Aber: Keine vollständige Automatisierung, hoher Entwicklungsaufwand (glue code) im Application Engineering
- Keine freie Feature-Kombination

Diskussion: Modularität

- Komponenten verbergen interne Implementierung
- Idealerweise haben Komponenten „kleine“ Interfaces
- **Ergebnis: Hohe Feature-Kohäsion, geringe Feature-Kopplung**
- Aber: Sehr geringe Granularität der Variabilität
  - Seitenwechselstrategien, Suchalgorithmen, Locking im B-Baum, oder VARCHAR als Komponente?
  - Farben oder gewichtete Kanten im Graph als Komp.?
- Funktionalität ggf. schwer als Komponente zu kapseln
  - Transaktionsverwaltungskomponente?

Frameworks

- Sammlung abstrakter und konkreter Klassen (*generischen Implementierung)*
- Abstrakte Struktur, die für einen bestimmten Zweck instanziiert/angepasst/erweitert werden kann
  - vgl. Template Method Pattern und Strategy Pattern
- Wiederverwendbare Lösung für eine Problemfamilie in einer Domäne
- Punkte, an denen Erweiterungen vorgesehen sind: **hot spots** (auch: variation points, extension points)
- Umkehrung der Kontrolle, das Framework bestimmt die Aufrufreihenfolge von Diensten
  - Hollywood Prinzip: „Don’t call us, we’ll call you.“

Plug-Ins

- Plug-Ins kapseln Erweiterungen eines Frameworks
- Ermöglichen es, spezielle Funktionen bei Bedarf hinzufügen
- Üblicherweise getrennt vom Framework kompilierbar (third-party provider)
- Beispiele: *Emailprogramme, Graphikprogramme, Media-Player, Webbrowser*

Beispiel: Web-Portal

- Webapplikation-Frameworks wie *Struts*, die grundlegende Konzepte vorgeben und (teil-)implementieren
- Entwickler konzentrieren sich auf Anwendungs-logik statt Navigation zwischen Webseiten

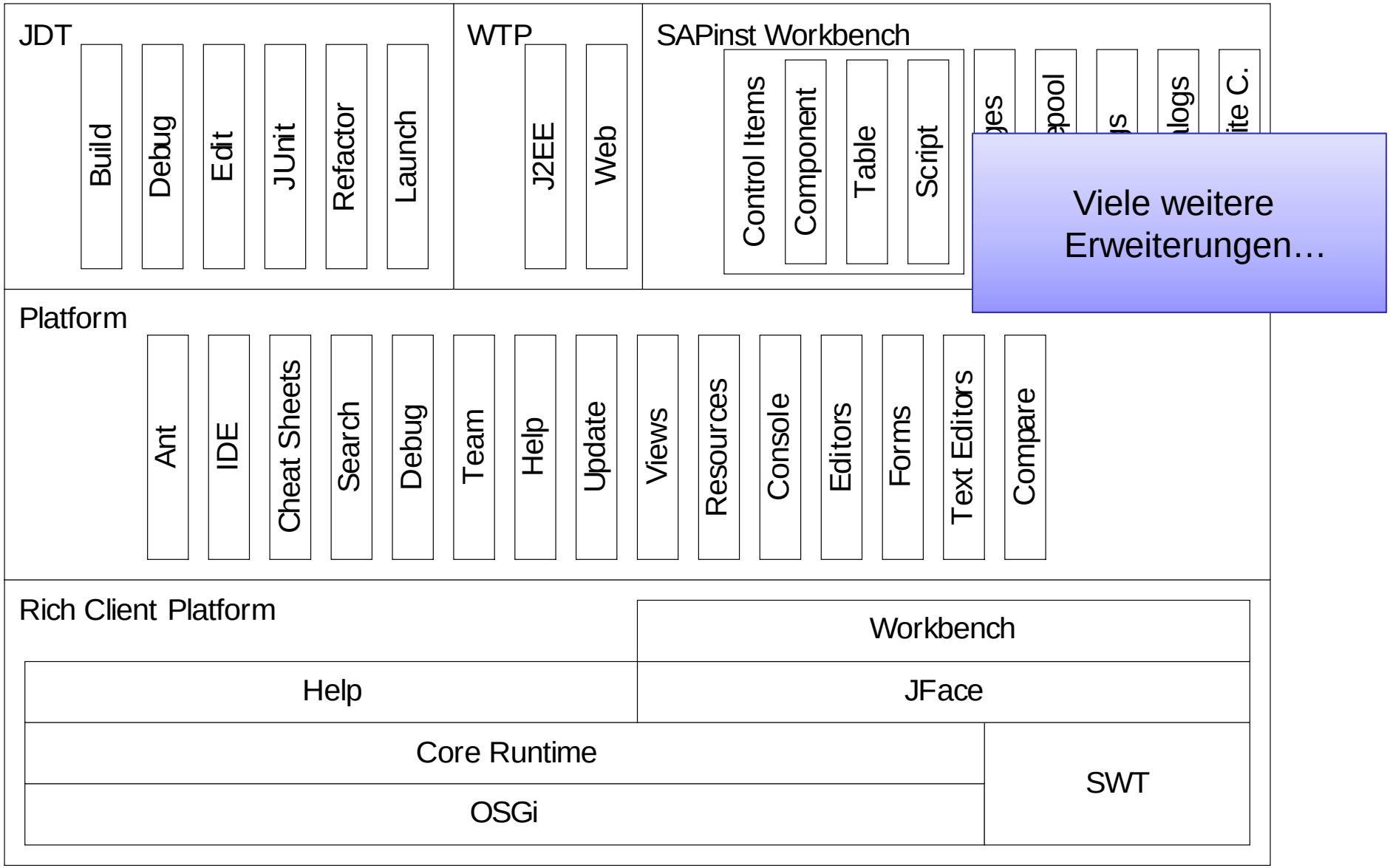
```
<?php
class WebPage {
        function getCSSFiles();
        function getModuleTitle();
        function hasAccess(User u);
        function printPage();
}
?>
```

```
<?php
class ConfigPage extends WebPage {
        function getCSSFiles() {…}
        function getModuleTitle() {
          return “Configuration”;
        }
        function hasAccess(User u) {
          return user.isAdmin();
        }
        function printPage() {
          print “<form><div>…”;
        }
}
?>
```

Beispiel: Eclipse

- Framework für IDEs
- Der gemeinsame Teil (Editoren, Menüs, Projekte, Verzeichnisbaum, Copy & Paste & Undo Operationen, CVS, uvm.) ist durch das Framework vorgegeben
- Nur sprachspezifische Erweiterungen (Syntax Highlighting, Compiler) müssen implementiert werden
- Eco-System: Framework bestehend aus vielen kleineren Frameworks

Eclipse: Architektur



Beispiel: Calculator

```
public class Calc extends JFrame {
        private JTextField textfield;
        public static void main(String[] args) { new Calc().setVisible(true); }
        public Calc() { init(); }
        protected void init() {
          JPanel contentPane = new JPanel(new BorderLayout());
          contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
          JButton button = new JButton();
          button.setText("calculate");
          contentPane.add(button, BorderLayout.EAST);
          textfield = new JTextField("");
          textfield.setText("10 / 2 + 6");
          textfield.setPreferredSize(new Dimension(200, 20));
          contentPane.add(textfield, BorderLayout.WEST);
          button.addActionListener(/* code zum berechnen */);
          this.setContentPane(contentPane);
          this.pack();
          this.setLocation(100, 100);
          this.setTitle("My Great Calculator");
          // code zum schliessen des fensters
        }
}
```

Quelltext für alle Varianten fast gleich, nur roter Quelltext unterscheidet sich (hot spots)

White-Box Frameworks

- Erweiterung durch überschreiben und hinzufügen von Methoden (vgl. Template Method Pattern)
- Interna des Framework müssen verstanden werden
  - schwierig zu erlernen
- Flexible Erweiterung
- Viele Subklassen nötig -> ggf. unübersichtlich
- Status direkt verfügbar durch Superklasse
- Keine Plug-ins, nicht getrennt kompilierbar

Beispiel: Calculator als Whitebox Framework

```
public abstract class Application extends JFrame {
        protected abstract String getApplicationTitle();        //Abstrakte Methoden
        protected abstract String getButtonText();
        protected String getInititalText() {return "";}
        protected void buttonClicked() { }
        private JTextField textfield;
        public Application() { init(); }
        protected void init() {
                JPanel contentPane = new JPanel(new BorderLayout());
                contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
                JButton button = new JButton();
                button.setText(getButtonText());
                contentPane.add(button, BorderLayout.EAST);
                textfield = new JTextField("");
                textfield.setText(getInititalText());
                textfield.setPreferredSize(new Dimension(200, 20));
                contentPane.add(textfield, BorderLayout.WEST);
                button.addActionListener(/* … buttonClicked(); … */);
                this.setContentPane(contentPane);
                this.pack();
                this.setLocation(100, 100);
                this.setTitle(getApplicationTitle());
                // code zum schliessen des fensters
        }
        protected String getInput() { return textfield.getText();}
}
```

Calculator as Whitebox framework

Modularität?

```
public abstract class Application extends JFrame {
        protected abstract String getApplicationTitle();        //Abstrak
        protected abstract String getButtonText();
        protected String getInititalText() {return "";}
        protected void buttonClicked() { }
        private JTextField textfield;
        public Appli
        protected v
                JPa
                con
                JBu
                butt
                con
                text
                text
                text
                con
                button.addActionListener(/* … buttonClicked(); … */);
                this
                this
                this
                this
                // co
        }
    protected Strin
}
```

```
public class Calculator extends Application {
        protected String getButtonText() { return "calculate"; }
        protected String getInititalText() { return "(10 – 3) * 6"; }
        protected void buttonClicked() {
           JOptionPane.showMessageDialog(this, "The result of "+getInput()+
                 " is "+calculate(getInput())); }
        protected String getApplicationTitle() { return "My Great Calculator"; }
        public static void main(String[] args) {
           new Calculator().setVisible(true);
        }
}
```

```
public class Ping extends Application {
        protected String getButtonText() { return "ping"; }
        protected String getInititalText() { return "127.0.0.1"; }
        protected void buttonClicked() {  /* … */  }
        protected String getApplicationTitle() { return "Ping"; }
        public static void main(String[] args) {
           new Ping().setVisible(true);
        }
}
```

Black-Box Frameworks

- Einbinden des anwendungsspezifischen Verhaltens durch Komponenten mit speziellem Interfaces (Plug-ins)
  - vgl. Strategy Pattern, Observer Pattern
- Nur das Interface muss verstanden werden
  - einfacher zu erlernen, aber aufwendiger zu entwerfen
- Flexibilität durch bereitgestellte Hot Spots festgelegt, häufig in Form von Design Pattern
- Status der Applikation nur bekannt, wenn durch Interface verfügbar
- Insgesamt häufig besser wiederverwendbar

Beispiel: Calculator

```
public class Application extends JFrame {
        private JTextField textfield;
        private Plugin plugin;
        public Application(Plugin p) { this.plugin=p; p.setApplication(this); init(); }
        protected void init() {
                JPanel contentPane = new JPanel(new BorderLayout());
                contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
                JButton button = new JButton();
                if (plugin != null)
                        button.setText(plugin.getButtonText());
                else
                        button.setText("ok");
                contentPane.add(button, BorderLayout.EAST);
                textfield = new JTextField("");
                if (plugin != null)
                        textfield.setText(plugin.getInititalText());
                textfield.setPreferredSize(new Dimension(200, 20));
                contentPane.add(textfield, BorderLayout.WEST);
                if (plugin != null)
                        button.addActionListener(/* … plugin.buttonClicked();… */);
                this.setContentPane(contentPane);
                …
        }
        public String getInput() { return textfield.getText();}
}
```

```
public interface Plugin {
    String getApplicationTitle();
    String getButtonText();
    String getInititalText();
    void buttonClicked() ;
    void setApplication(Application app);
}
```

Beispiel: Calculator

Modularität?

Application kennt Plugins nicht

```
public class Application extends JFrame {
        private JTextField textfield;
        private Plugin plugin;
        public Application(Plugin p) { this.plugin=p; p.setApplication(this); init(); }
        protected void init() {
                JPanel contentPane = new JPanel(new BorderLayout());
                contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
                JButton button = new JButton();
                if (plugin != null)
                        button.setText(plugin.getButtonText());
                else
                        button.setText("ok");
                contentPane.add(button, BorderLayout.EAST);
                textfield = new JTextField("");
                if (plugin != nu
                        textfield.s
                textfield.setPre
                contentPane.a
                if (plugin != nu
                        button.ac
                this.setConten
                …
        }
        public String getInput()
}
```

```
public interface Plugin {
    String getApplicationTitle();
    String getButtonText();
    String getInititalText();
    void buttonClicked() ;
    void setApplication(Application app);
}
```

```
public class CalcPlugin implements Plugin {
        private Application application;
        public void setApplication(Application app) { this.application = app; }
        public String getButtonText() { return "calculate"; }
        public String getInititalText() { return "10 / 2 + 6"; }
        public void buttonClicked() {
          JOptionPane.showMessageDialog(null, "The result of "
                + application.getInput() + " is "
                + calculate(application.getText())); }
        public String getApplicationTitle() { return "My Great Calculator"; }
}
```

```
class CalcStarter {  public static void main(String[] args) { new Application(new CalcPlugin()).setVisible(true); }}
```

Mehr Separierung möglich?

Modularität?

Nur Plugin und InputProvider Interface

```
public class Application extends JFrame implements InputProvider {
        private JTextField textfield;
        private Plugin plugin;
        public Application(Plugin p) { this.plugin=p; p.setApplication(this); init(); }
        protected void init() {
                JPanel contentPane = new JPanel(new BorderLayout());
                contentPane.setBorder(new BevelBorder(BevelBorder.LOWE
                JButton button = new JButton();
                if (plugin != null)
                        button.setText(plugin.getButtonText());
                else
                        button.setText("ok");
                contentPane.add(button, BorderLayout.EAST);
                textfield = new JTextField("");
                if (plugin != null)
                        textfield.setText(plugin.getInititalText());
                textfield.setPre
                contentPane.a
                if (plugin != nu
                        button.ac
                this.setConten
                …
        }
        public String getInput()
}
```

```
public interface InputProvider {
    String getInput();
}
```

```
public interface Plugin {
    String getApplicationTitle();
    String getButtonText();
    String getInititalText();
    void buttonClicked() ;
    void setApplication(InputProvider app);
}
```

```
public class CalcPlugin implements Plu
        private InputProvider application;
        public void setApplication(InputProvider app) { this.application = app; }
        public String getButtonText() { return "calculate"; }
        public String getInititalText() { return "10 / 2 + 6"; }
        public void buttonClicked() {
           JOptionPane.showMessageDialog(null, "The result of "
                + application.getInput() + " is "
                + calculate(application.getInput())); }
        public String getApplicationTitle() { return "My Great Calculator"; }
}
```

```
class CalcStarter {  public static void main(String[] args) { new Application(new CalcPlugin()).setVisible(true); }}
```

Laden von Plug-Ins

- Häufig durch spezielle *Plug-In-Loader*…
  - … sucht in Verzeichnis nach DLL/Jar/XML Dateien
  - … testet ob eine Datei ein Plug-In implementiert
  - … prüft Abhängigkeiten zu anderen Plug-Ins
  - … initialisiert das Plug-In
- Dazu häufig ein GUI für Plug-In-Konfiguration
- Beispiele:
  - Eclipse (plugin-Verzeichnis + Jar)
  - Miranda (plugins-Verzeichnis + DLL)
- Alternativ: Plug-Ins in Konfigurationsdatei festlegen oder *StarterProgramm* generieren

Beispiel: Plug-In-Loader (Java Reflections)

```
public class Starter {

    public static void main(String[] args) {
        if (args.length != 1)
            System.out.println("Plugin name not specified");
        else {
            String pluginName = args[0];
            try {
                Class<?> pluginClass = Class.forName(pluginName);
                new Application((Plugin) pluginClass.newInstance())
                                                .setVisible(true);
            } catch (Exception e) {
                System.out.println("Cannot load plugin " + pluginName
                                                + ", reason: " + e);
            }
        }
    }
}
```

Produktlinien als Frameworks

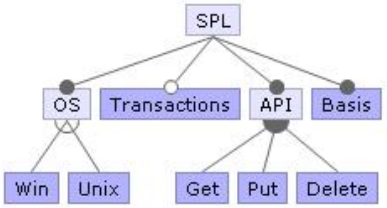
Domain Eng.

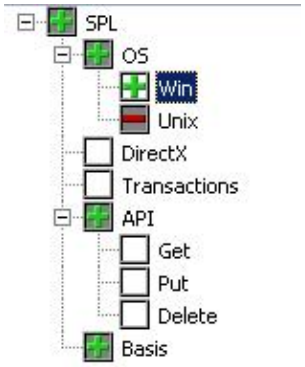
Application Eng.

Mapping von Features zu Plugins

Framework + Plug-Ins

Feature-Modell

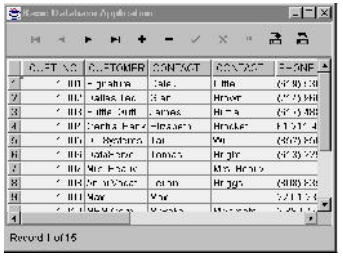




Feature-Auswahl als Eingabe

Feature-Auswahl

Plug-In-Auswahl (und ggf. Startkonfiguration generieren)



Anwendung = Framework mit passenden Plug-Ins

Produktlinien als Frameworks

- Vollautomatisierung möglich
- Modular
- In der Praxis weit verbreitet und bewährt
- Aber: Erstellungsaufwand und Laufzeit-Overhead für Framework/Architektur
- **Vorplanung** nötig, Frameworkdesign erfordert Erfahrung
- Schwierige Wartung, Evolution
- **Geringe Granularität** oder riesige Interfaces
  - Plug-In für Transaktionsverwaltung, VARCHAR oder gewichtete Kanten?

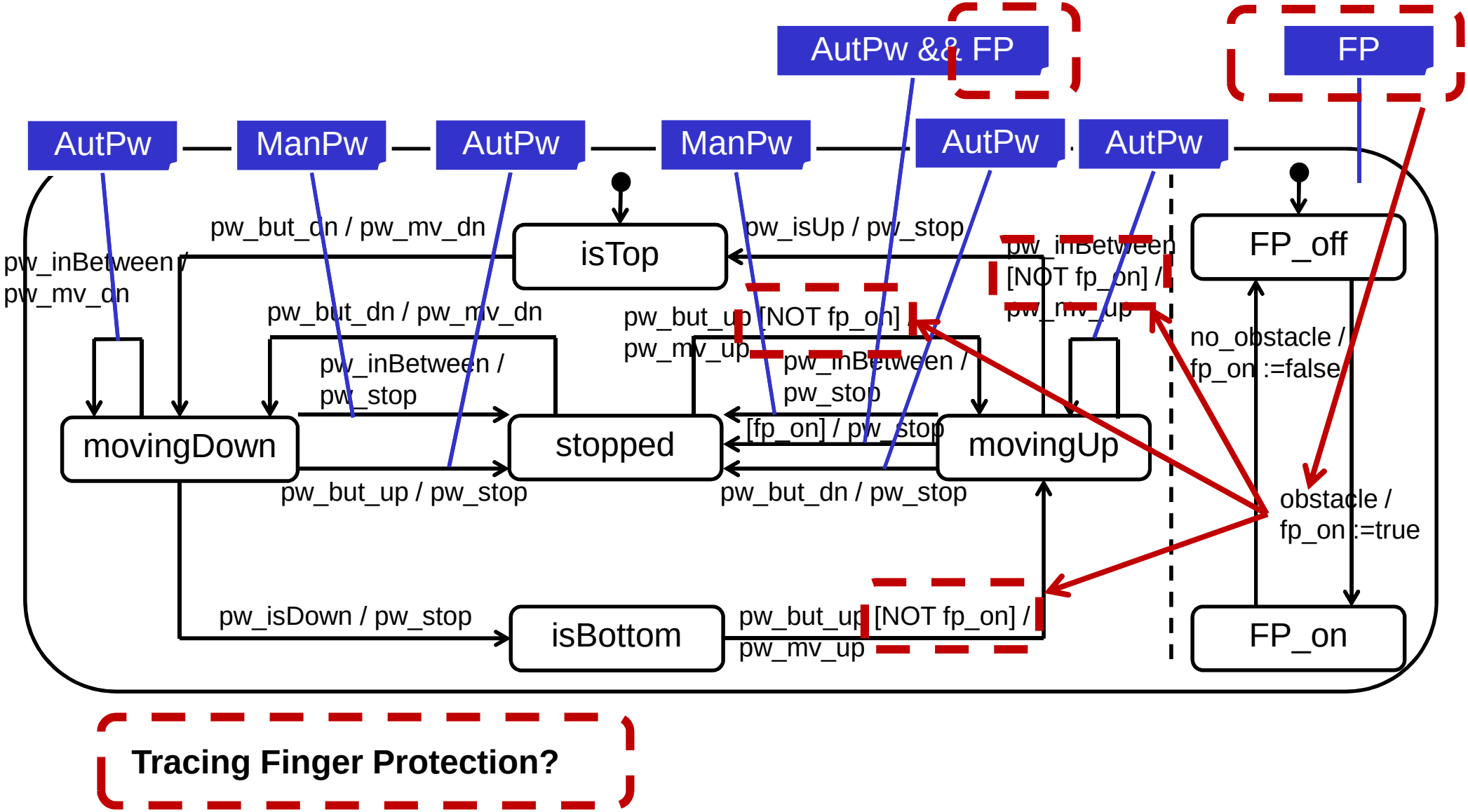
Crosscutting Concerns

- Beobachtung: Nicht alle Belange (Features) in einem Programm können mittels Objekten (Komponenten, Plugins) modularisiert werden

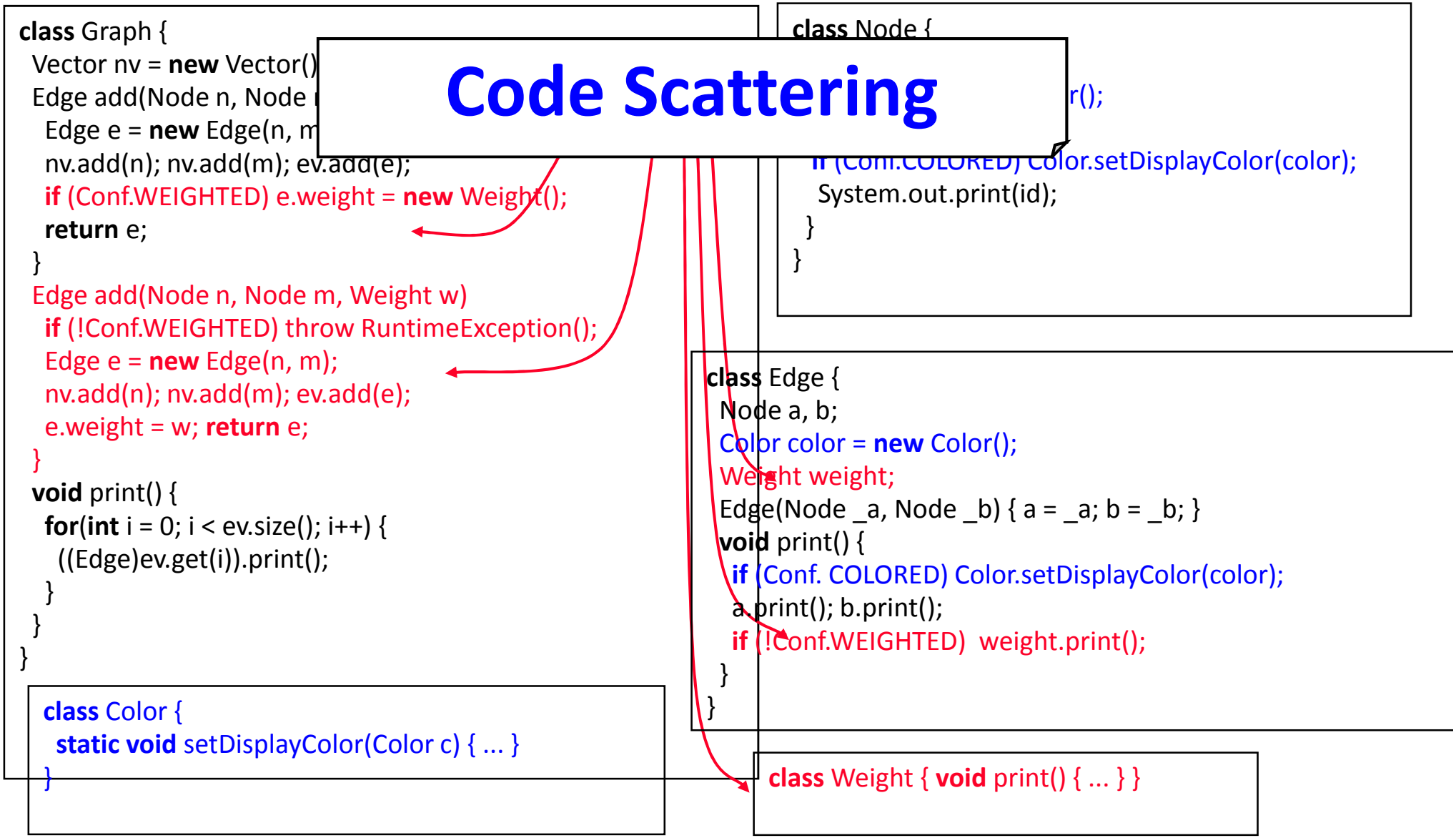
- **Belange** sind semantisch zusammenhängende Einheiten

- Aber ihre Implementierung ist häufig **verstreut, vermischt und repliziert** im Code

Beispiel: BCS



Distributed Code



Vermischter Code

```
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);
    if (Conf.WEIGHTED) e.weight = new Weight();
    return e;
  }
  Edge add(Node n, Node m, Weight w)
    if (!Conf.WEIGHTED) throw RuntimeException();
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m); ev.add(e);
    e.weight = w; return e;
  }
  void print() {
```

Code Tangling

```
  }
```

```
class Color {
  static void setDisplayColor(Color c) { ... }
}
```

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

```
class Edge {
  Node a, b;
  Color color = new Color();
  Weight weight;
  Edge(Node _a, Node _b) { a = _a; b = _b; }
  void print() {
    if (Conf. COLORED) Color.setDisplayColor(color);
    a.print(); b.print();
    if (!Conf.WEIGHTED)  weight.print();
  }
}
```

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

Code Scattering und Code Tangling

- Verstreuter Feature-Code (code scattering)
  - Code, der zu einem Feature gehört, ist nicht modularisiert, sondern im gesamten Programm verteilt
  - Häufig kopierter Code (auch wenn es je nur ein einzelner Methodenaufruf ist)
  - Oder stark verteilte Implementierung von (komplementären) Teilen eines Features

- Vermischter Feature-Code (code tangling)
  - Code, der zu verschiedenen Features gehört, ist in einem Modul (oder einer Methode) vermischt

Dominante Dekomposition

- Viele Belange können modularisiert werden, jedoch nicht immer gleichzeitig
  - Problemstellung nur in einer Richtung modularisierbar
  - Im Graph können Farben modularisiert werden…
  - …dann sind aber die Datenstrukturen (Node, Edge) verteilt (oder umgekehrt)

- Entwickler wählen eine „dominante“ Dekomposition aus (z.B. Operationen, Authentifizierung, Datenstrukturen), aber einige andere Belange „schneiden quer“

- Gleichzeitige Modularisierung entlang verschiedener Dimensionen nicht möglich

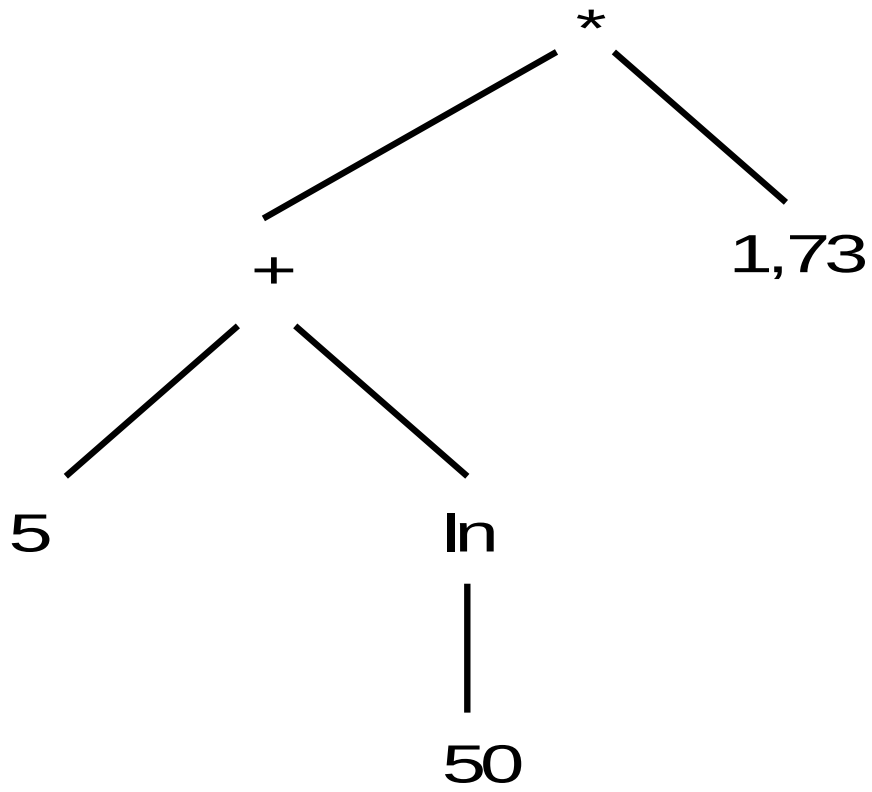
Beispiel: Expression Problem

Wie weit kann man von Datenstrukturen und Methoden abstrahieren, so dass beide unabhängig erweitert werden können…

- … ohne bestehenden Code zu ändern (oder sogar ohne Neukompilation des bestehenden Codes)
- … mehrfach und in beliebiger Reihenfolge und
- ohne (nicht-triviale) Code-Replikationen?

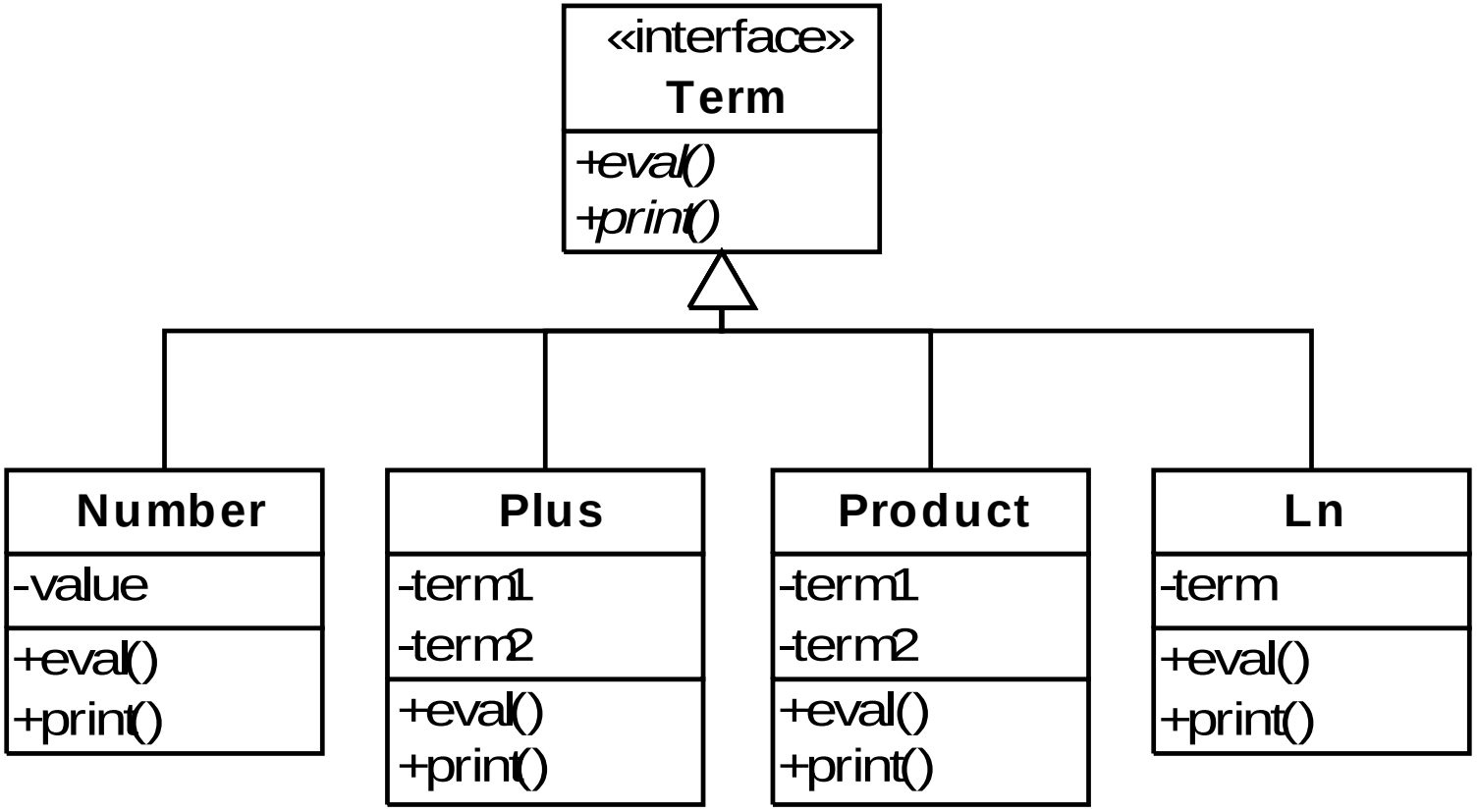
Beispiel: Expression Problem

- Beispiel: Mathematische Ausdrücke werden in einer Baumstruktur gespeichert und können ausgerechnet und ausgegeben werden



Implementation 1: Class-centered

- Rekursive Klassenstruktur (Composite Pattern)
- Für jede Operation eine Methode in jeder Klasse definiert

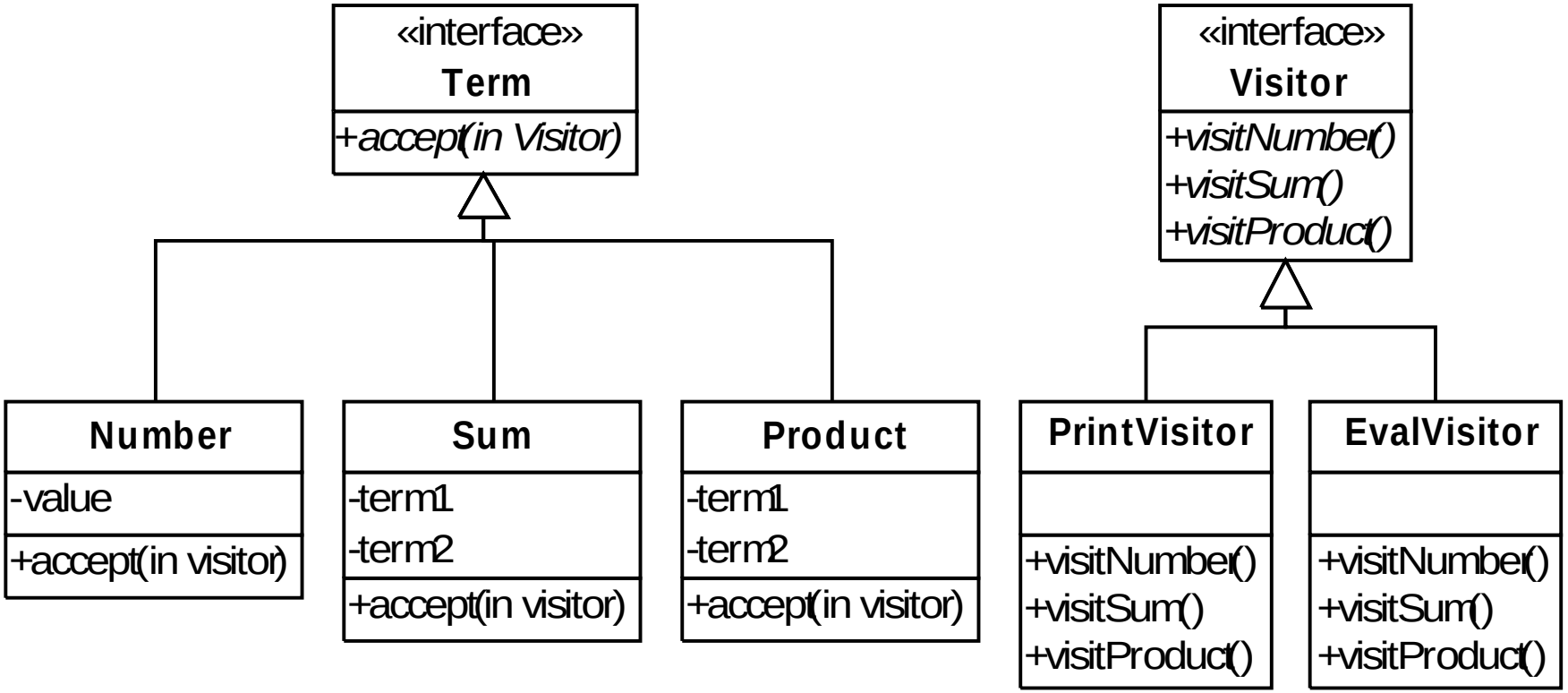


Implementation 1: Diskussion

- Ausdrücke sind modular
- Neue Operationen, z. B. drawTree oder simplify, können nicht einfach hinzugefügt werden
- Alle bestehenden Klassen müssen angepasst werden!
- Operationen sind querschneidend zu den Ausdrücken

Implementation 2: Method-centered

- Nur eine Methode accept pro Klasse
- Methoden werden mit dem Visitor-Pattern implementiert



(Ln Klasse aus Platzgründen ausgelassen)

Code: Method-centered

```
interface Term {
  void accept(Visitor v);
}
class Number {
  float value;
  void accept(Visitor v) {
    v.visitNumber(this);
  }
}
class Sum {
  Term term1, term2;
  void accept(Visitor v) {
    v.visitSum(this);
  }
}
class Product {
  Term term1, term2;
  void accept(Visitor v) {
    v.visitProduct(this);
  }
}
```

```
interface Visitor {
  void visitNumber(Number n);
  void visitSum(Sum s);
  void visitProduct(Product p);
}
class PrintVisitor {
  void visitNumber(Number n) {
    System.out.print(n.value);
  }
  void visitSum(Sum s) {
    System.out.print('(');
    s.term1.accept(this);
    System.out.print('+');
    s.term2.accept(this);
    System.out.print(')');
  }
  void visitProduct(Product p) {
    s.term1.accept(this);
    System.out.print('*');
    s.term2.accept(this);
  }
}

// Main:
// term.accept(new PrintVisitor());
```

Implementation 2: Diskussion

- Operationen sind modular
- Neue Ausdrücke, z. B. Min oder Power können nicht einfach hinzugefügt werden
- Für jede neue Klasse müssen alle Visitor-Klassen angepasst werden
- Ausdrücke sind querschneidend zu den Operationen

Expression Problem: Fazit

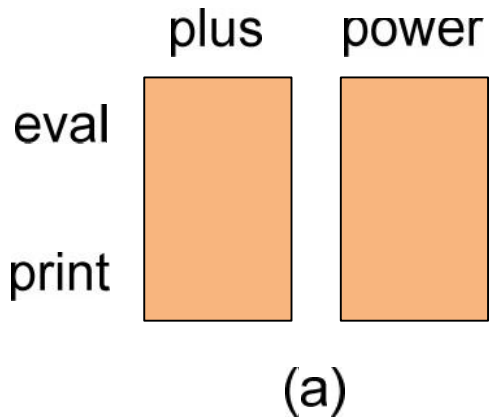
- Nur sehr schwer möglich, Ausdrücke und Operationen darauf **gleichzeitig** zu modularisieren
  (komplizierte Lösungen mit Java 1.5 Generics existieren)

- Daten-zentrierter Ansatz
  - Neue Ausdrücke können direkt hinzugefügt werden: modular
  - Neue Operationen müssen in alle Klassen eingefügt werden: nicht modular

- Methoden-zentrierter Ansatz
  - Neue Operationen können als weiterer Visitor hinzugefügt werden: modular
  - Für neue Ausdrücke müssen alle bestehenden Visitors erweitert werden: nicht modular

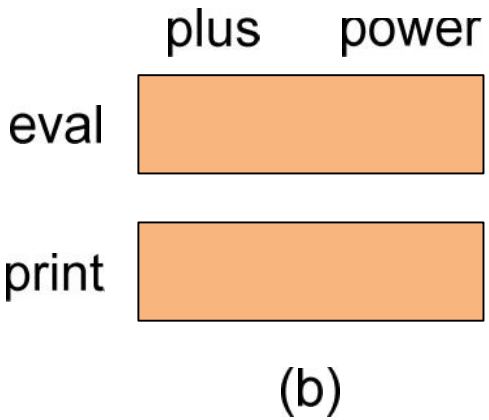
Expression Problems

Daten-zentriert (Composite)

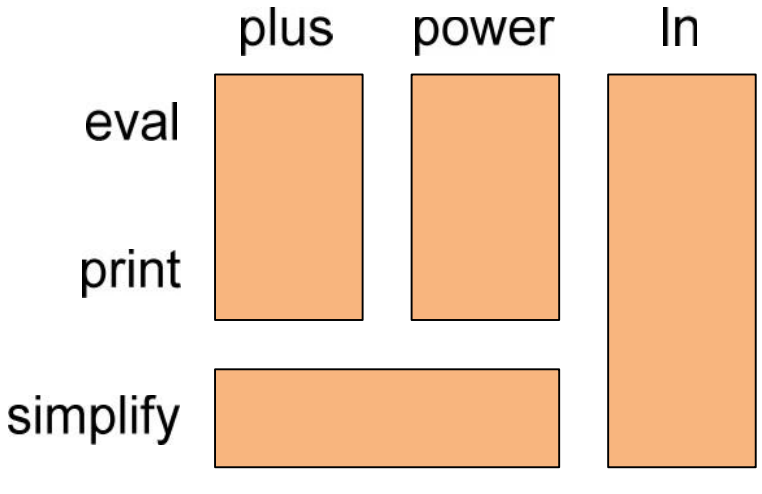


(a)

Methoden-zentriert (Visitor)

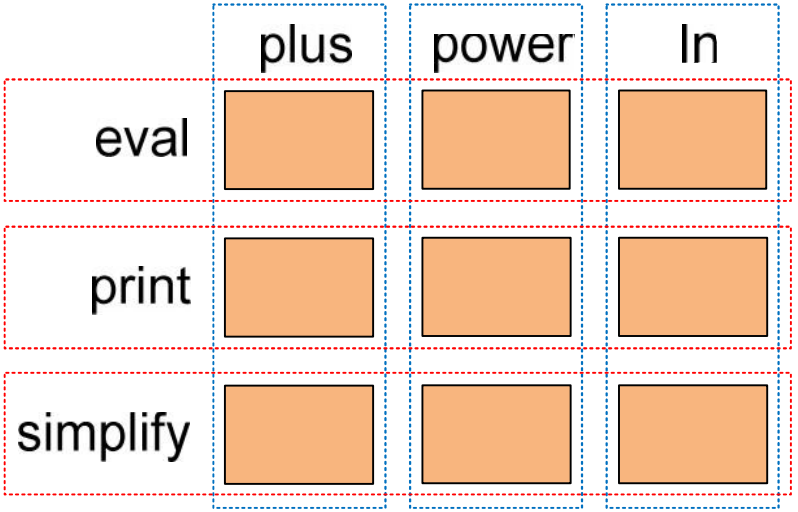


(b)



(c)

Ausblick: Neue Sprachansätze



(d)

Ausblick: Feature-Interaktionen

Crosscutting Concerns: Weitere Beispiele

- Logging: Meldung nach jeder Methode
- Caching/Pooling: Code bei jedem Erzeugen eines Objektes
- Synchronisierung/Locking: Erweiterung vieler Methoden mit lock/unlock-Aufrufen
- Und: Features in Produktlinien!

Dilemma

- Es ist nicht immer möglich, alle Belange zu modularisieren (Tyrannei der dominanten Dekomposition…)
- Ein **Grundmaß** an verstreutem und vermischtem Code in OOP-Implementierungen ist normal
- Einige Belange sind immer „orthogonal“ zu anderen: **querschneidende** Belange
- Features in Produktlinien sind (naturgemäß) häufig von diesem Phänomen betroffen

Preplanning Problem

- Erweiterungen sind nicht ad-hoc möglich, sondern müssen voraus geplant werden
- Es müssen explizit Erweiterungsmöglichkeiten vorgesehen werden
  - Extension Points in Frameworks
  - Interfaces/Parameter in Komponenten
- Ohne passende Extension Points sind modulare Erweiterungen nur mit sehr hohem Aufwand möglich

Preplanning Problem - Beispiel

- Stack-Methoden sollen synchronisiert werden
- Modulare Erweiterung mittels Subklasse oder Delegation

Basis-Code

```
class Stack { /* ... */ }
class Main {
  public static void main(
                          String[]
  args) {
    Stack stack = new Stack();
    stack.push('foo');
    stack.push('bar');
    stack.pop();
  }
}
```

Spätere ungeplante Erweiterung

```
class LockedStack extends Stack {
  private void lock() { /* ... /* }
  private void unlock() { /* ... /* }
  public void push(Object o) {
    lock();
    super.push(o);
    unlock();
  }
  public Object pop() {
    lock();
    Object result = super.pop();
    unlock();
    return result;
  }
}
```

Preplanning Problem – Beispiel 2

- Problem: Instantiierung des Stacks im Basiscode muss angepasst werden
  - Keine Möglichkeit, ohne Änderung des Basiscode (nicht-modular)
- Alternative
  - Design Pattern: Factory statt direkter Instantiierung (erlaubt modulare Erweiterung)
  - Framework mit passendem Extension Point
- Erweiterungsmöglichkeiten müssen antizipiert werden (preplanning) oder nachträglich dem Basiscode hinzugefügt werden (nicht-modular)

Zusammenfassung

- Modularisierung von Features mit Komponenten und Frameworks
- Keine Vollautomatisierung, Laufzeitoverhead, grobe Granularität
- Probleme bei querschneidenden Belangen und feiner Granularität
- Modularität erfordert vorherige Planung

Referenzen

- *Richard M. Stallman:* **The C Preprocessor.** Last Revised July 1992 for GCC v.2. https://gcc.gnu.org/onlinedocs/cpp/
- *Tom Ball:* **Munge: Swing's Secret Preprocessor.** https://weblogs.java.net/blog/tball/archive/2006/09/munge_swings_se.html
- *Jarzabek, S.; Bassett, P.; Hongyu Zhang; Weishan Zhang:* **XVCL: XML-based Variant Configuration Language.** *Proceedings. 25th International Conference on Software Engineering*, pp.810,811, 2003. http://sourceforge.net/projects/fxvcl/files/XVCL%20Processor/
- *Jörg Pleumann, Omry Yadan, Erik Wetterberg:* **Antenna – An Ant-To-End Solution for Wireless Java.** Version 1.2.1. http://antenna.sourceforge.net/
- *C. Kästner:* **CIDE: Decomposing Legacy Applications into Features**. *Proceedings of the 11st International Software Product Line Conference,* pp. 149--150, 2007. http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/
- *Janet Feigenspan, Maria Papendieck, Christian Kästner, Mathias Frisch, and Raimund Dachselt.* **FeatureCommander: colorful #ifdef world.** *Proceedings of the 15th International Software Product Line Conference*, 2011. http://wwwiti.cs.uni-magdeburg.de/~feigensp/xenomai/
- *Krzysztof Czarnecki and Ulrich W. Eisenecker:* **Generative Programming: Methods, Tools, and Applications.** ACM Press/Addison-Wesley Publ. Co., New York, NY, USA, 2000