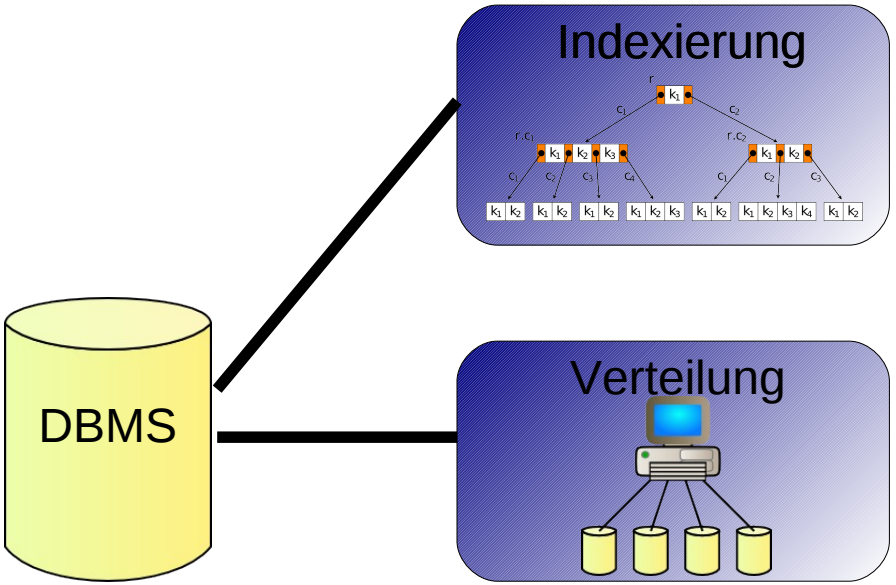
Statische Erkennung semantischer Feature-Interaktionen

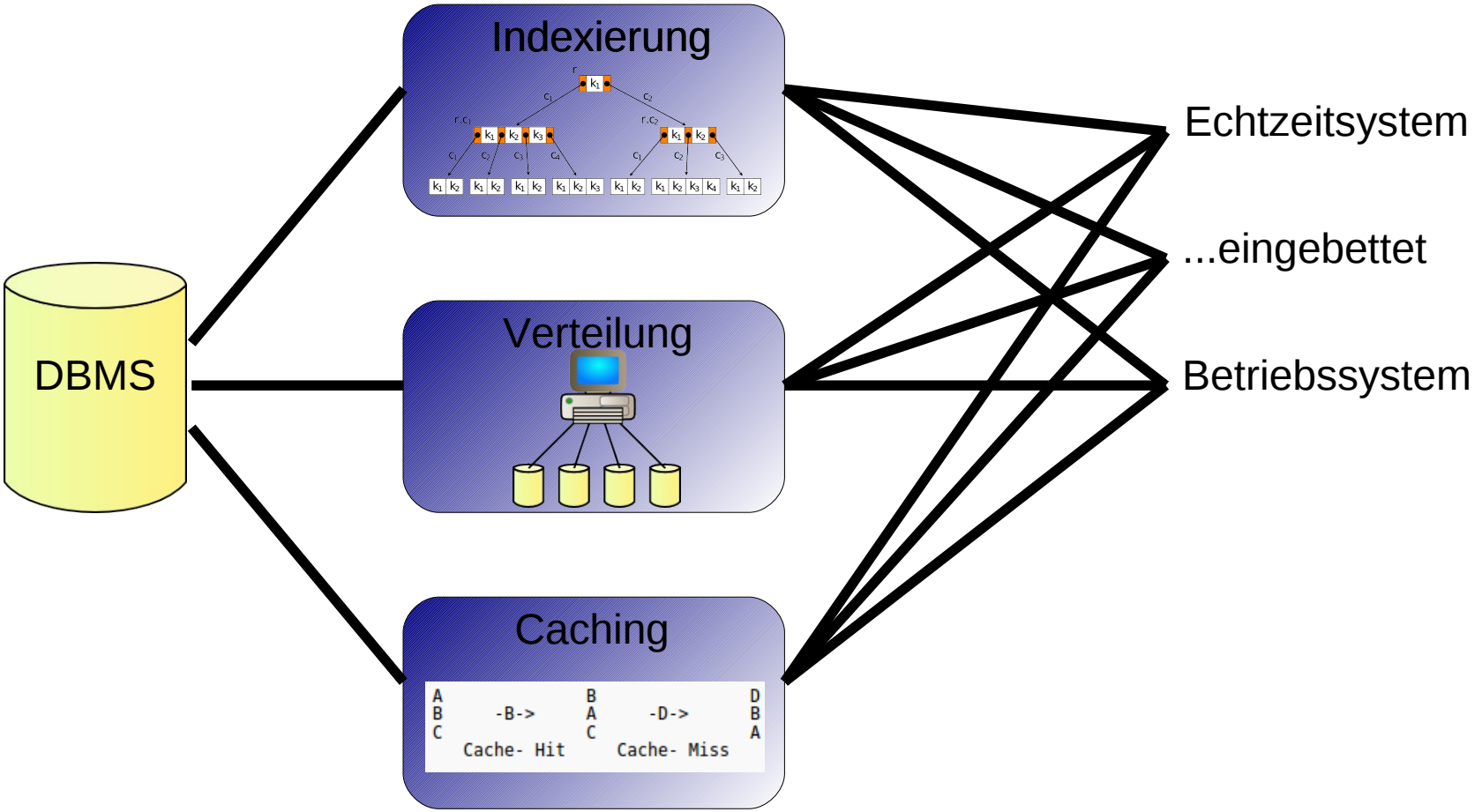
KPS'09
Maria Taferl
12.-14. Oktober 2009

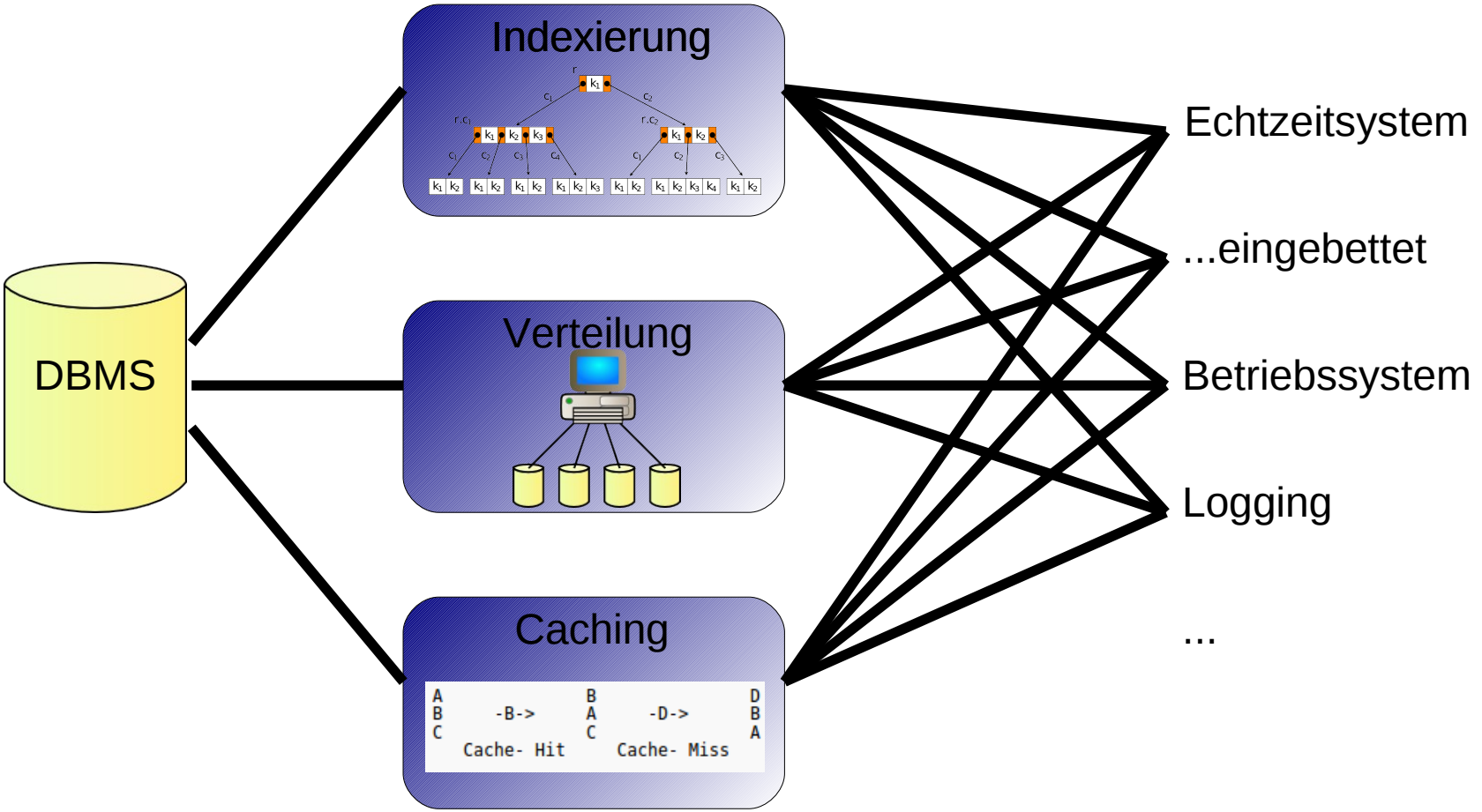
Wolfgang Scholz
Lehrstuhl für Programmierung
Universität Passau
wolfgang.scholz@uni-passau.de

Inhalt

- Feature-orientierte Software-Entwicklung
- Feature-Interaktionen
- Statische Erkennung von Feature-Interaktionen
- Programmsemantik
- Zusammenfassung


Indexierung
DBMS
Verteilung


Indexierung
Verteilung
Caching
DBMS
Echtzeitsystem
...eingebettet
Betriebssystem
A
B
C
-B->
B
A
C
D
B
A
Cache- Hit
Cache- Miss

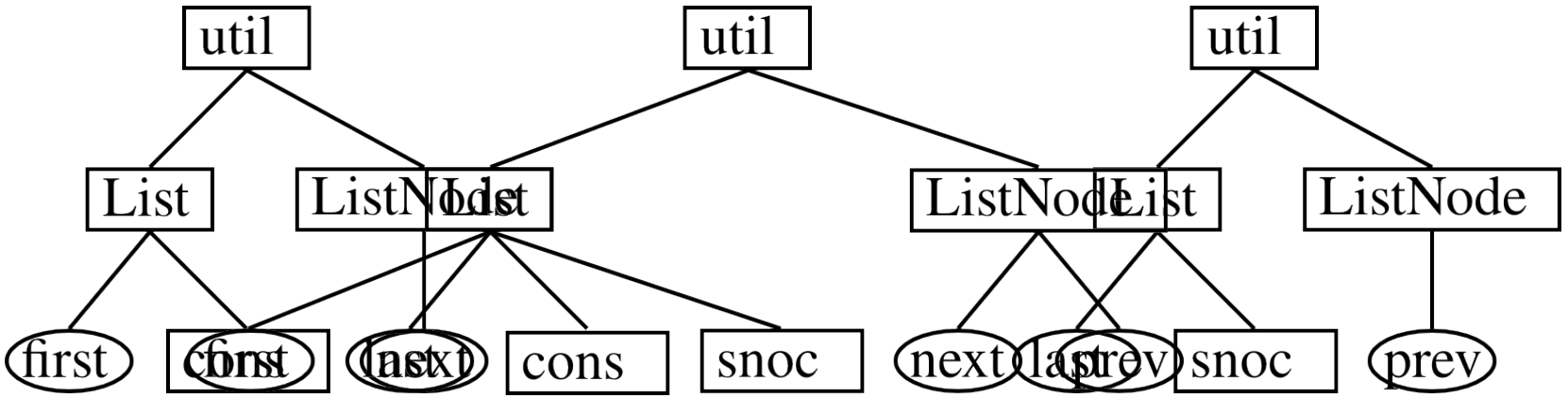

Indexierung
Verteilung
Caching
DBMS
A
B
C
-B->
B
A
C
D
B
A
-D->
Cache- Hit
Cache- Miss
Echtzeitsystem
...eingebettet
Betriebssystem
Logging
...

Software Engineering

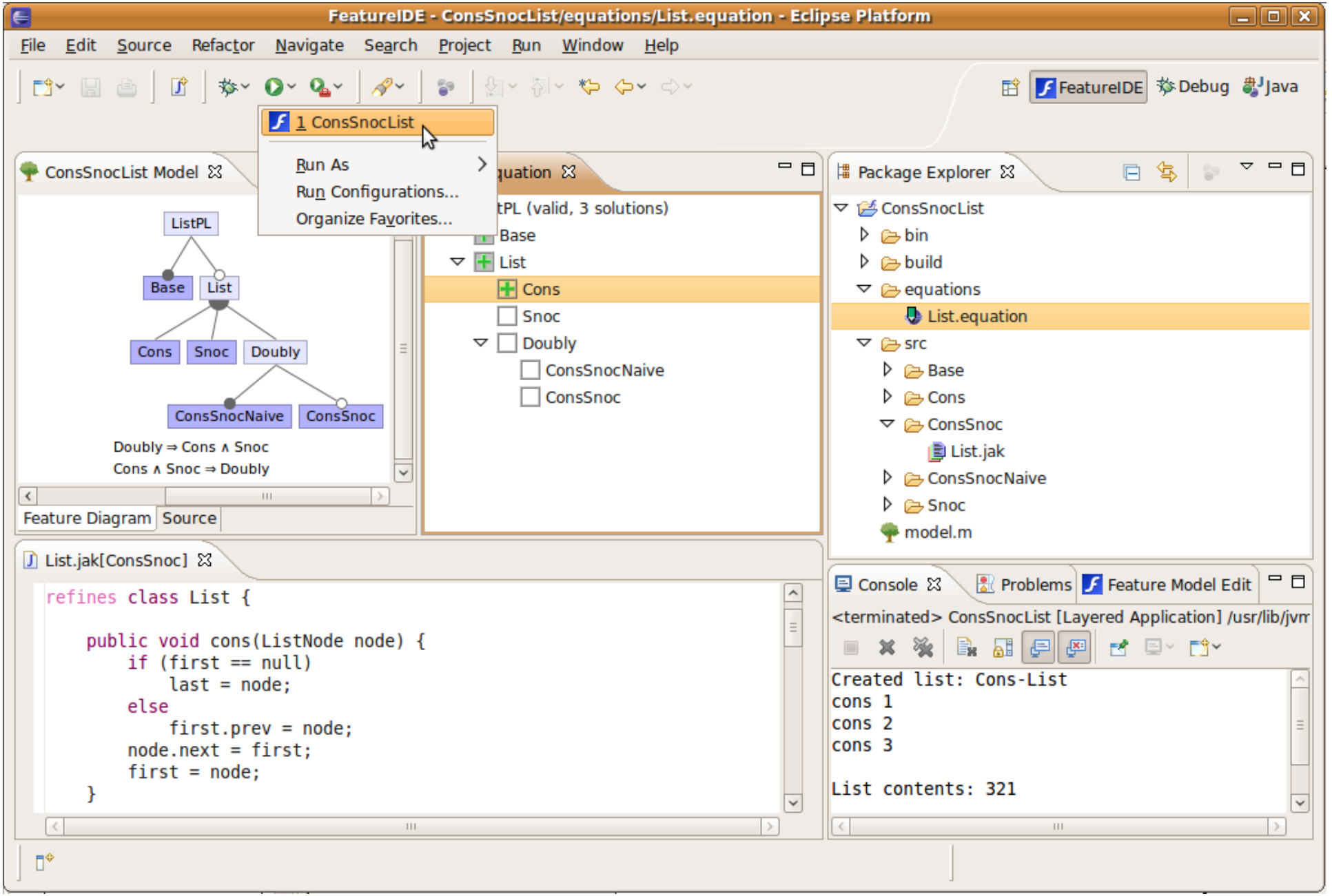
- Ziel: hohe Kohäsion, schwache Kopplung
- Kohäsive Einheiten über Modulgrenzen hinweg
- ➔ Dominante Zerlegung von Funktionalität reicht nicht aus

Feature-orientierte Software-Entwicklung

- Features (teilw. getrennt entwickelt)
- Superimposition
  - Zusätzlicher Kompilationsschritt
  - Kombiniert ausgewählte Features



FeatureIDE


FeatureIDE - ConsSnocList/equations/List.equation - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
FeatureIDE Debug Java
1 ConsSnocList
Run As
Run Configurations...
Organize Favorites...
ConsSnocList Model
ListPL
Base List
Cons Snoc Doubly
ConsSnocNaive ConsSnocNaive
Doubly ⇒ Cons ∧ Snoc
Cons ∧ Snoc ⇒ Doubly
Feature Diagram Source
tPL (valid, 3 solutions)
Base
List
Cons
Snoc
Doubly
ConsSnocNaive
ConsSnoc
Package Explorer
ConsSnocList
bin
build
equations
List.equation
src
Base
Cons
ConsSnoc
List.jak
ConsSnocNaive
Snoc
model.m
List.jak[ConsSnoc]
refines class List {
public void cons(ListNode node) {
if (first == null)
last = node;
else
first.prev = node;
node.next = first;
first = node;
}
Console Problems Feature Model Edit
<terminated> ConsSnocList [Layered Application] /usr/lib/jvm
Created list: Cons-List
cons 1
cons 2
cons 3
List contents: 321

Feature-Interaktionen

A

from: A
to: B

B

Feature-Interaktionen

A

from: A
to: B

Re:

from: B
to: A

B

Feature-Interaktionen

B

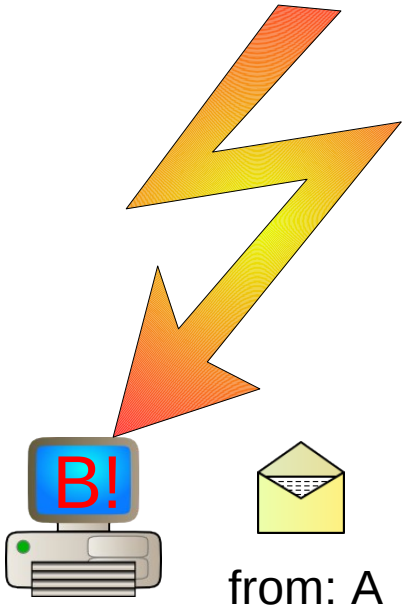
A

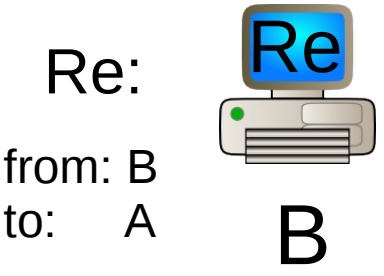
from: A
to: F

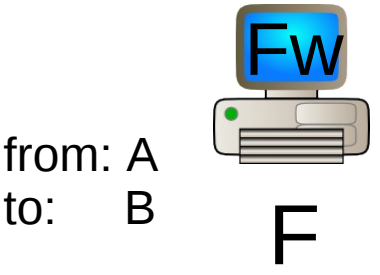
from: A
to: B

F

Feature-Interaktionen



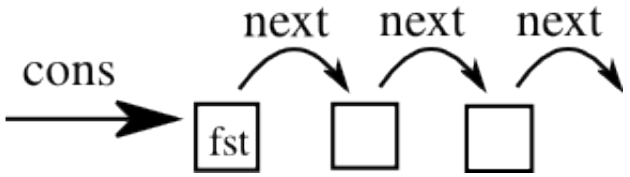


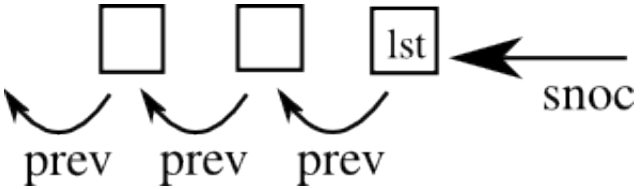


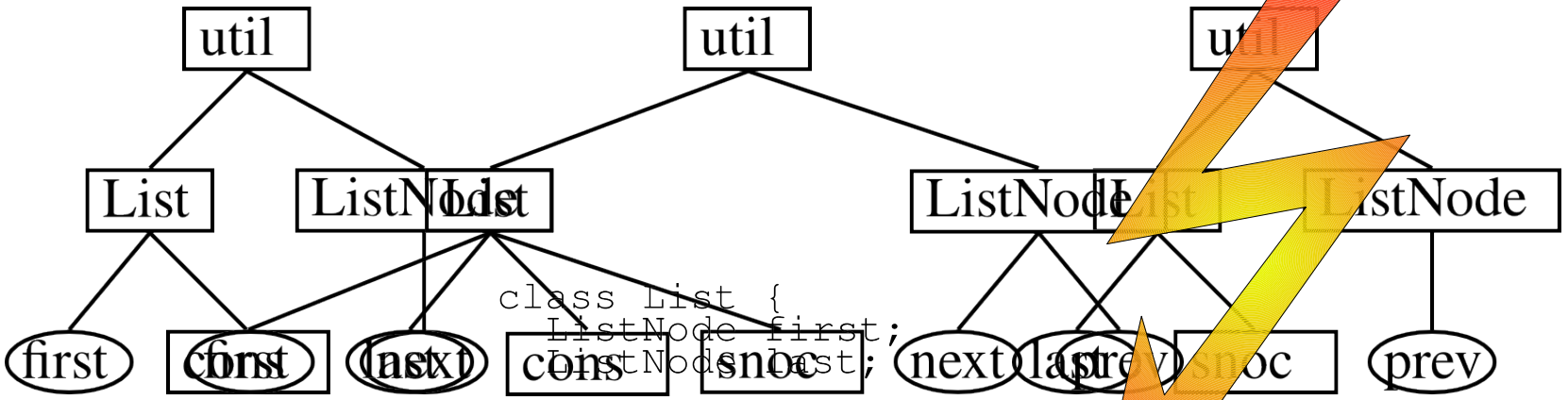
Feature-Interaktionen

- Mgl. getrennte Entwicklung von Features
- Funktionieren in Isolation
- Gemeinsam aber inkompatibel

Beispiel: cons-snoc-Liste








```
class List {
  ListNode first;

  void cons(ListNode n) {
    n.next = first;
    first = n;
  }
}

class ListNode {
  ListNode next;
}
```

```
class List {
  ListNode first;
  ListNode last;

  void cons(ListNode n) {
    n.next = first;
    first = n;
  }

  void snoc(ListNode n) {
    n.prev = last;
    last = n;
  }
}

class ListNode {
  ListNode next;
  ListNode prev;
}
```

```
class List {
  ListNode last;

  void snoc(ListNode n) {
    n.prev = last;
    last = n;
  }
}

class ListNode {
  ListNode prev;
}
```

Erkennung von Feature-Interaktionen

- Syntaktisch: Compiler
- Semantisch:
  - Zur Laufzeit
    - Tests
  - Statisch
    - Formale Methoden

Statische Erkennung semantischer Feature-Interaktionen

- Semantik notwendig
- Semantik aus Feature-Code extrahierbar, aber:
  - Einerseits: Semantik überbestimmt (Implementierungsdetails)
  - Andererseits: Semantik reicht nicht aus (cons-snoc Beispiel)
- Alternativ: Semantik angeben
  - Vorbedingungen, Nachbedingungen, Invarianten
  - Wie Design by Contract: intuitiv

Verifikationsplattform *Why*

- Verifikation imperativer Programme
- Programmannotationen
- Semantische Inferenz des Programmcodes
- Generierung von Beweispflichten
  - Automatische Beweiser (z.B. *Simplify)*
  - Beweisassistenten (z.B. *Coq)*

Beispiel: cons-snoc-Liste mit Why

live mit Eclipse...

Zusammenfassung

- Einbau einer Interaktionserkennung in *FeatureIDE* (mit *Why*)
  - Automatische Überprüfung einer gewählten Feature-Kombination auf Interaktionen
  - Nach Möglichkeit interagierende Methoden/Felder im Feature-Code anzeigen
- Untersuchung weiterer Methoden zur Erkennung von semantischen Feature-Interaktionen
- Möglichkeiten suchen, zusätzlich nötigen Code automatisch zu erzeugen

Referenzen

- FeatureIDE: *„FeatureIDE: Tool Framework for Feature-Oriented Software Development“*, Kästner, Thüm, Saake, Feigenspan, Leich, Wielgorz, Apel, 2009

- Why: *„Verification of Non-Functional Programs using Interpretations in Type Theory“*, Filliâtre, 2003

- e-Mail Beispiel: *„Fundamental Nonmodularity in Electronic Mail“*, Hall, 2005

- Model Checking: *„Modular Verification of Open Features Using Three-Valued Model Checking“*, Li, Krishnamurthi, Fisler, 2005

- Formale Methoden: *„Feature-Oriented Description, Formal Methods, and DFC“*, Zave

- Design by Contract: *„Applying Design by Contract“*, Meyer, 1992

Click to exit presentation...

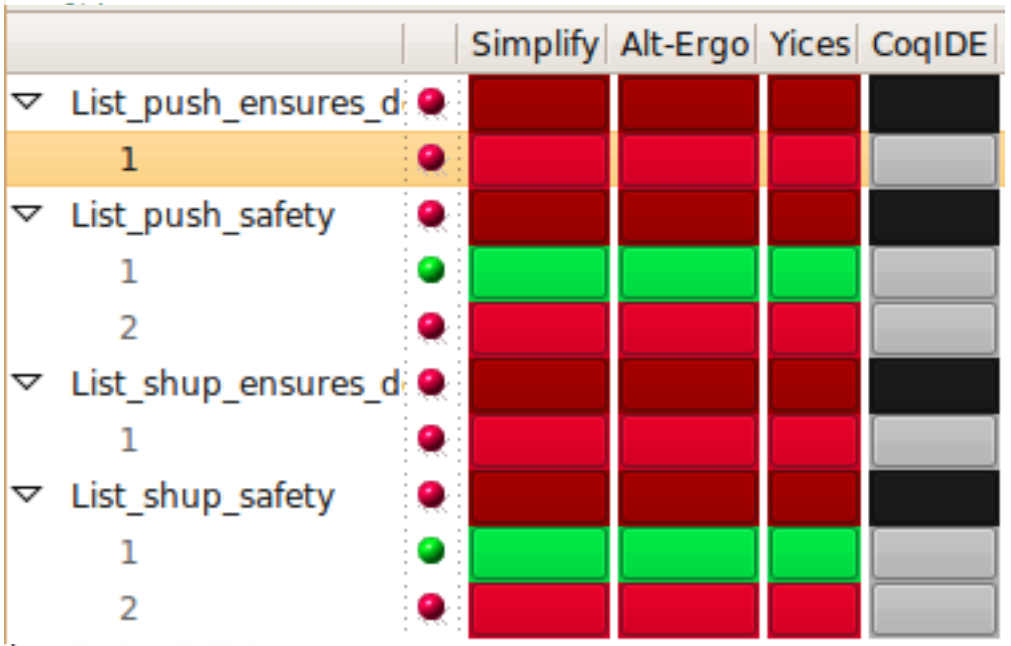
Semantik für cons-snoc

```
public class List {
    //@ invariant list_non_null: this != null;
    //@ invariant first_last_symmetry: first != null <==> last != null;

    ListNode first;
    ListNode last;

    /*@
    @ requires node != null && node.next == null
    @       && node.prev == null && node != first;
    @ ensures first == node && first.next == \old(first);
    @*/
    void cons(ListNode node) {
        node.next = first;
        first = node;
    }

    /*@
    @ requires node != null && node.next == null
    @       && node.prev == null && node != last;
    @ ensures last == node && last.prev == \old(last);
    @*/
    void snoc(ListNode node) {
        node.prev = last;
        last = node;
    }
}
```

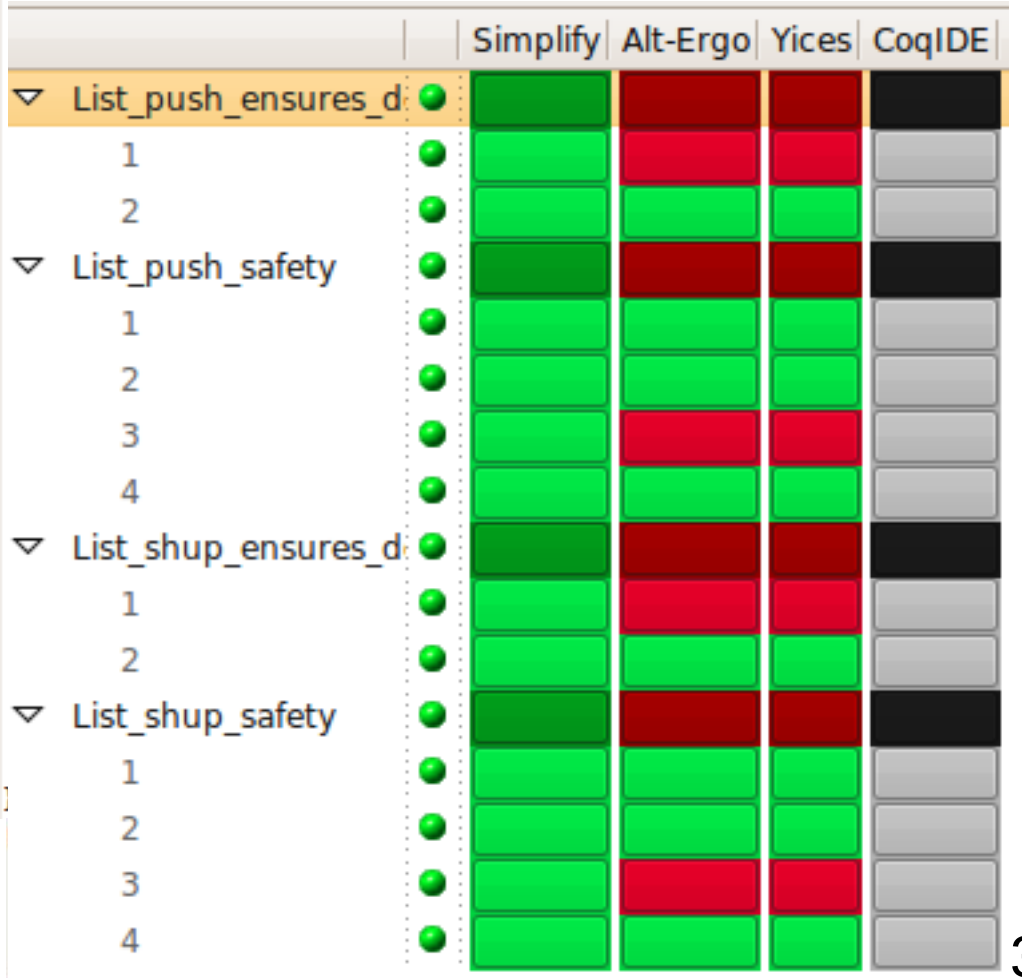
```
public class ListNode {
    //@ invariant list_symmetry: next != null ==> next.prev == this;

    ListNode next;
    ListNode prev;
}
```

Beispiel: cons-snoc-Liste

```
public class List {

    //@ invariant list_non_null: this != null;
    //@ invariant first_last_symmetry: first != null <==> last != null;

    ListNode first;
    ListNode last;

    /*@
    @ requires node != null && node.next == null
    @       && node.prev == null && node != first;
    @ ensures first == node && first.next == \old(first);
    @*/
    void push(ListNode node) {
        node.next = first;
        first = node;
    }

    /*@
    @ requires node != null && node.next == null
    @       && node.prev == null && node != last;
    @ ensures last == node && last.prev == \old(last);
```




```
public class List {
    //@ invariant list_non_null: this != null;
    //@ invariant first_last_symmetry: first != null <==> last != null;

    ListNode first;
    ListNode last;

    /*@
    @ requires node != null && node.next == null
    @       && node.prev == null && node != first;
    @ ensures first == node && first.next == \old(first);
```



weitere Ziele

- Granularität des Modells bestimmbar (Statement- / Methoden- / Featureebene)

Fazit

- Kombinatorische Explosion durch Feature-Komposition
- Annotationen zur Kompilationszeit geprüft
- Annotationen abstrahieren von Implementierungsdetails
- Einige semantische Feature-Interaktionen sind lokal erkennbar

nicht lokal erkennbare Interaktionen

- Erkennung: Annotationen im Derivat nötig
- im Derivat sind Annotationen „besser“ als Code:
  - Code berücksichtigt Implementierungsdetails aller beteiligten Features
  - Annotation berücksichtigt Semantik aller beteiligten Features
  - Semantik abstrahiert von Implementierungsdetails und ist daher „näher“ am intuitiven Verständnis