

Software Product Lines

Part 8: Feature Interactions

Daniel Strüber, Radboud University

with courtesy of: **Sven Apel**, **Christian Kästner**, **Gunter Saake**

Introduction

- ▶ Not considered yet:
 - ▶ What happens if features are not independent?
 - ▶ How do features interact?
 - ▶ How to keep variability despite dependencies?
 - ▶ How much variability is reasonable?

Agenda

- ▶ Feature interactions
- ▶ „Optional feature“ problem
- ▶ Solutions for managing of feature interactions
- ▶ Discussion: variability in practice



Feature interactions

Feature interactions

- ▶ Up to now, we considered features in isolation.
But: features interact, for example, via access to a shared resource or by providing a behaviour in combination
- ▶ **Feature interaction:**
 - ▶ Behaviour arising from the interplay of two or more features
 - ▶ Types: Intended or Unintended



Feature interactions: 1

- ▶ A phone operation system
- ▶ Shall support “**call waiting**” and “**forward if busy**”
- ▶ What happens if both features are active?
 - ▶ Not busy: no problem
 - ▶ Busy: waiting or forwarding?
- ▶ If intended behaviour not specified: both features try to “win”
- ▶ Can such problems be detected automatically?

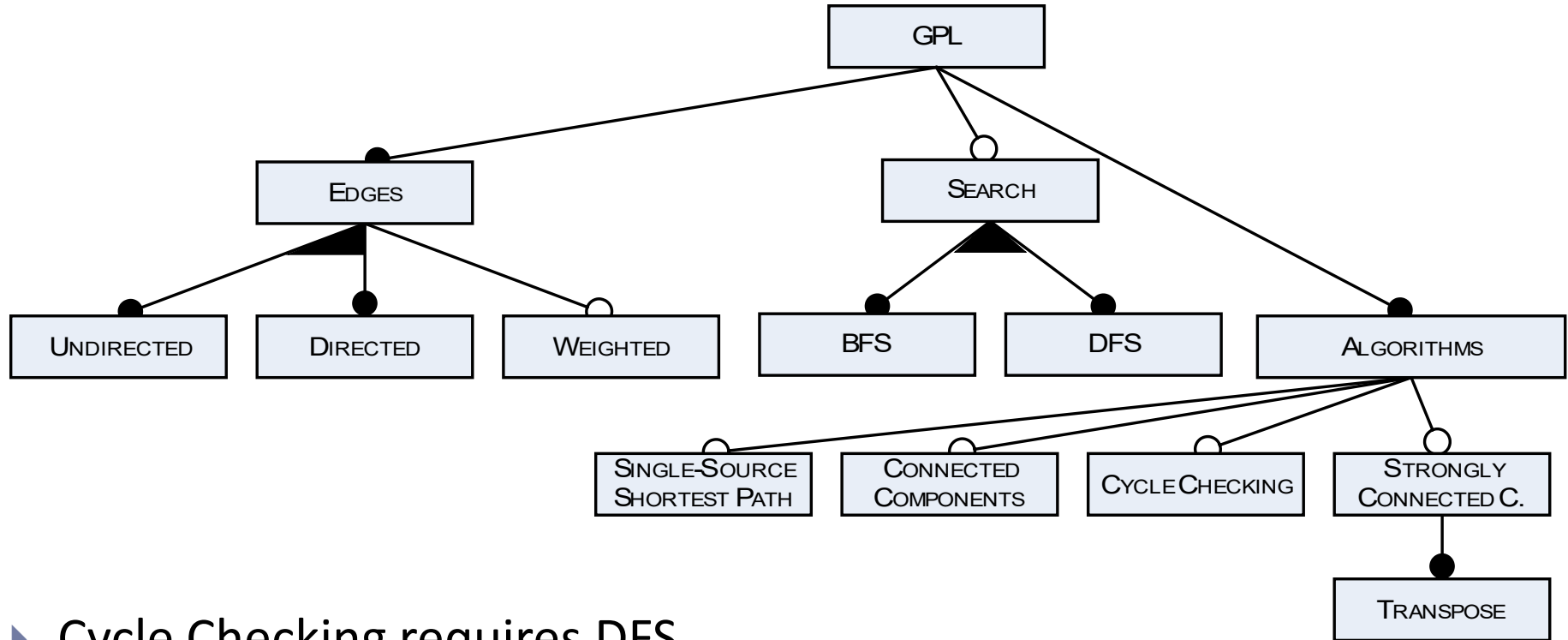


Feature interactions: 2

- ▶ Flood Control
 - ▶ avoid flooding by turning off water automatically
- ▶ Fire Control
 - ▶ if fire detected, sprinkle water
- ▶ What can go wrong?



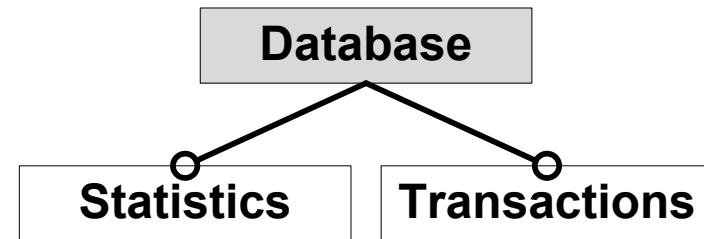
Feature interactions: 3



- ▶ Cycle Checking requires DFS
- ▶ Shortest Path requires special implementation for unweighted graphs
- ▶ Strongly Connected only works on directed graphs and requires DFS

Feature interactions: 4

- ▶ Database product line with two features
 - ▶ Collections of *statistics* like buffer-hit ratio, table size, transactions
 - ▶ *Transactions* to guarantee ACID properties
- ▶ Both are supposed to be optional
 - ▶ But: statistics collects information about transactions, transactions can use statistic information
- ▶ How to implement so that all variants are supported?

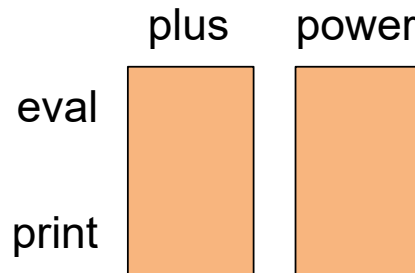


Feature interactions: 5

- ▶ Database product line with 2 features
 - ▶ Index: accelerates access through B-tree
 - ▶ Update: allows database updates, otherwise read-only
- ▶ Both supposed to be optional
 - ▶ Efficient read index
 - ▶ Writing to database without index simple
 - ▶ But: both features active → index needs to affect updates
- ▶ How to implement so that all variants are supported?

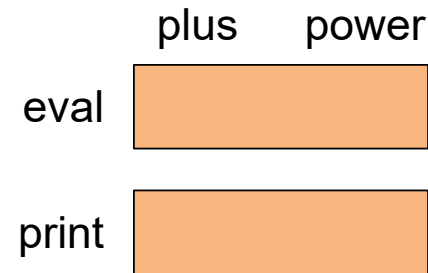
Feature interactions: 6

Data centric

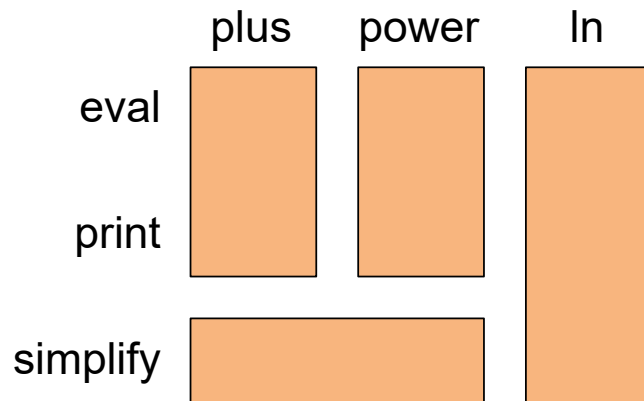


(a)

**Method centric
(visitor)**



(b)



(c)

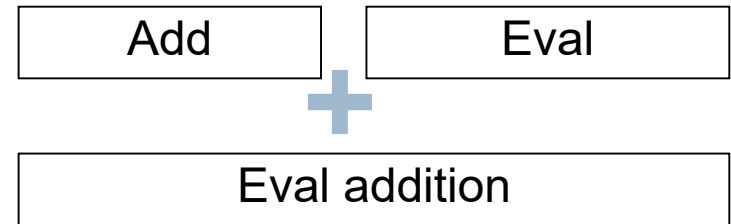
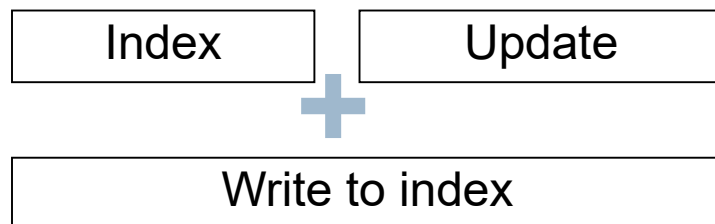
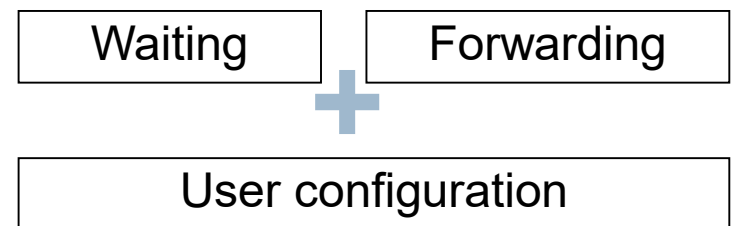
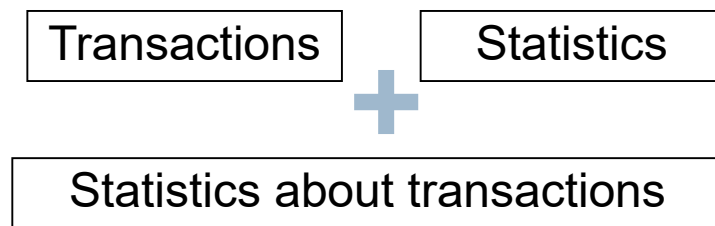
Interactions lead to dependencies

- ▶ Features using other features' methods
 - ▶ **Cycle** uses search function *Graph.search()* from **DFS**
 - ▶ **Shortest Path** expects that method *Edge.getWeight()* is available
- ▶ Features extending other features
 - ▶ Feature **Weighted** implements weights by overriding method *addEdge()* from **Base**
- ▶ Features rely on a certain structure or behaviour defined in another feature
 - ▶ **Strongly Connected** assumes that edges are directed

“Optional feature” problem

“Optional feature” problem

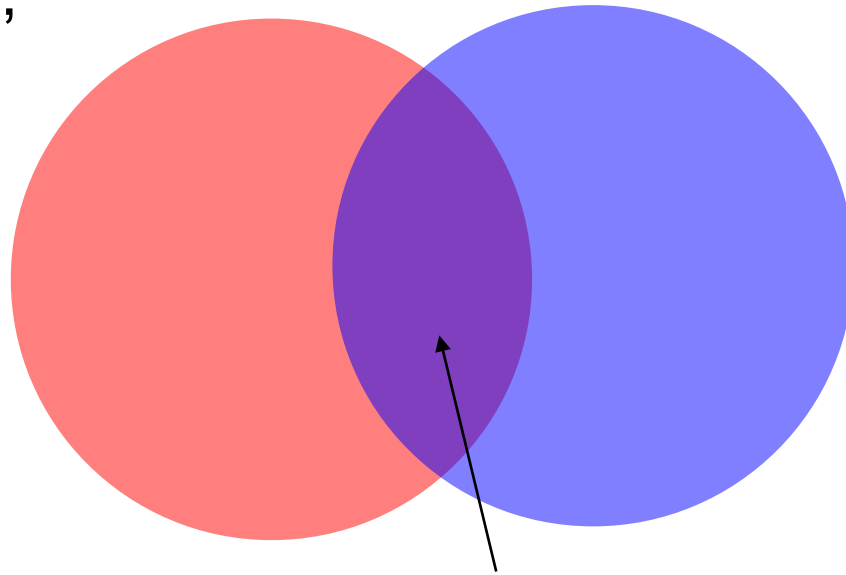
- ▶ An optional features behaves correctly in isolation
- ▶ Combination with another feature leads to problems
- ▶ Additional code necessary to coordinate behavior



“Optional feature” problem: Transactions and statistics

Statistics

(buffer hit ratio,
table size and
cardinality, ...)

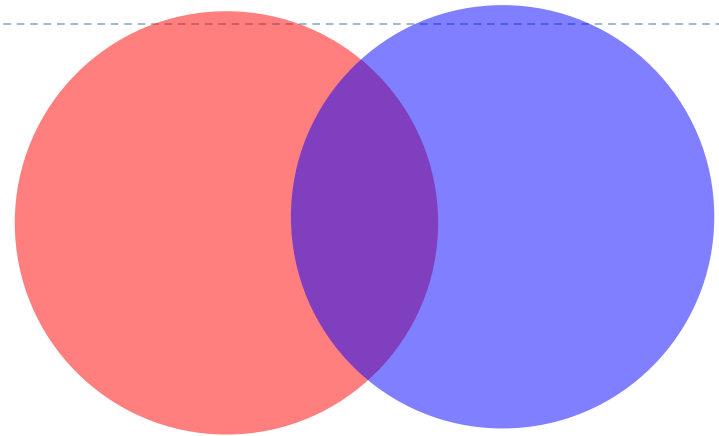


Transactions
(locks, commit,
rollback, ...)

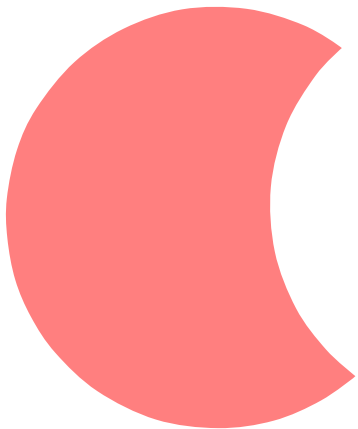
Throughput measurement
("Transactions per second")

Desired products

Database with transactions
and stats



Database with stats, without transactions

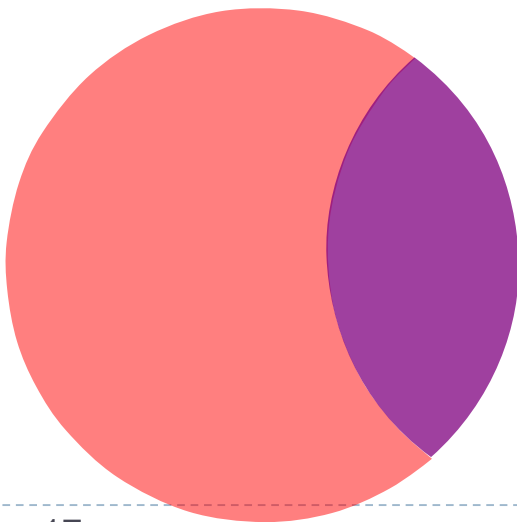
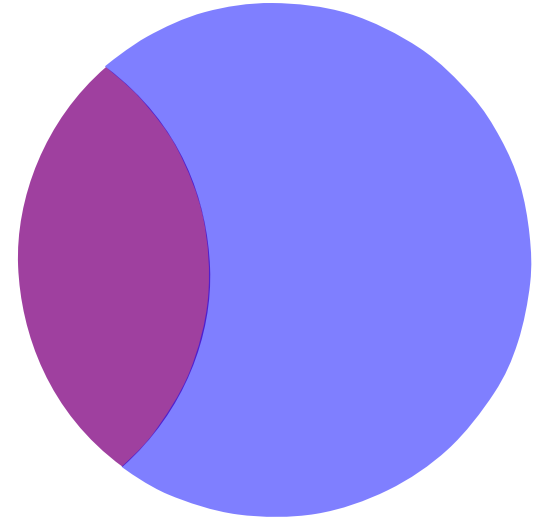


Database with transactions, without stats



Undesired/Ill-defined products

Database with transactions, without stats that still measures the throughput
(*slower and unnecessary code*)



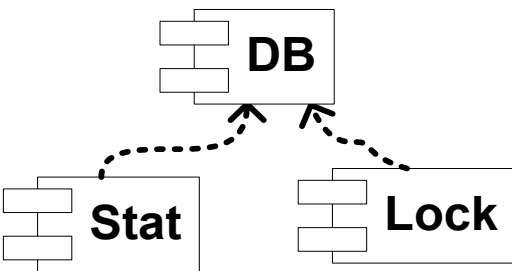
Database with stats, without transactions that still measures throughput
(???)

Implementation example

```
class Database {
    List locks;
    void lock() { /*...*/ }
    void unlock() { /*...*/ }
    void put(Object key, Object data) {
        lock();
        /*...*/
        unlock();
    }
    Object get(Object key) {
        lock();
        /*...*/
        unlock();
    }
    int getOpenLocks() {
        return locks.size();
    }
    int getDbSize() {
        return calculateDbSize();
    }
    static int calculateDbSize() {
        lock();
        /*...*/
        unlock();
    }
}
18
```

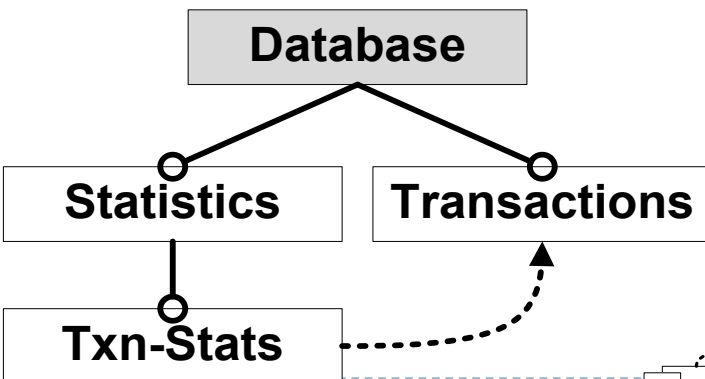
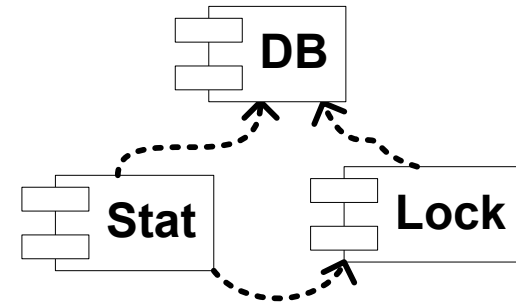
- ▶ Locking (blue)
- ▶ Stats (red)
- ▶ Features overlap in two places (violet)
 - ▶ Stats about locks
 - ▶ Synchronisation of statistics method

Divide into modules?

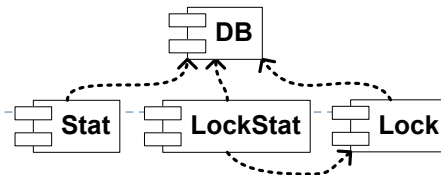


Where do we implement the throughput measurement?

How to create product with stats, without locking?



Is the mere throughput measurement really a feature *(on the domain level)*?

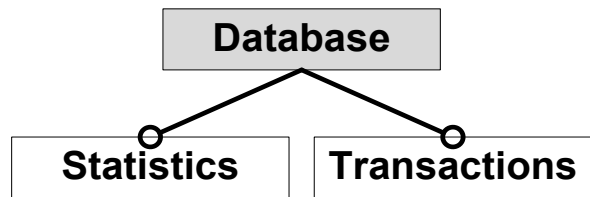


Variability

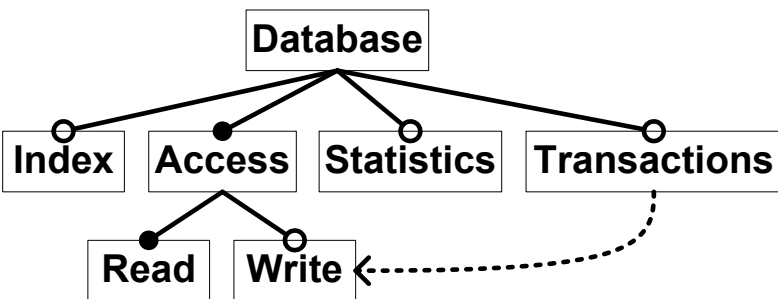
- ▶ Describes *which* and *how many* products can be derived from a software product line
- ▶ A product line with n independent, optional features allows 2^n products
- ▶ Dependencies between features restrict variability
- ▶ One dependency like „Feature A requires B“ reduces the number of possible products by 25%

Restricted variability because of unsuitable modularization

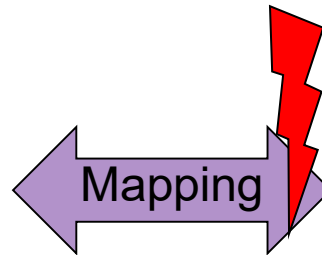
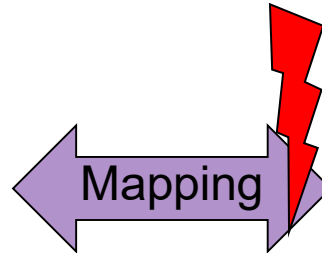
Feature model



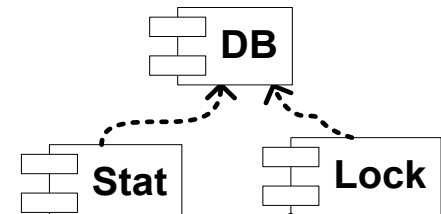
Desired:
4 products



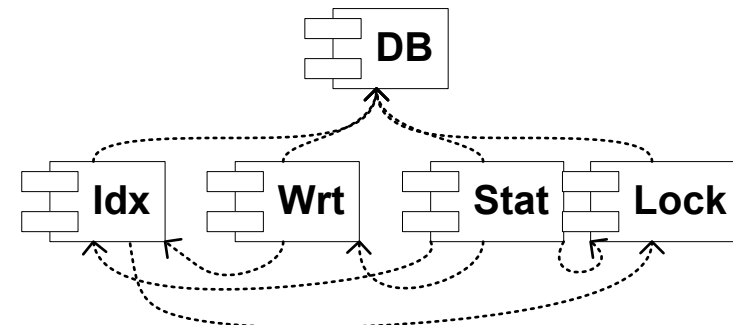
Desired:
12 products



Implementation



Actually supported:
3 products



Actually supported:
5 products

Case study: Berkeley DB

Java version

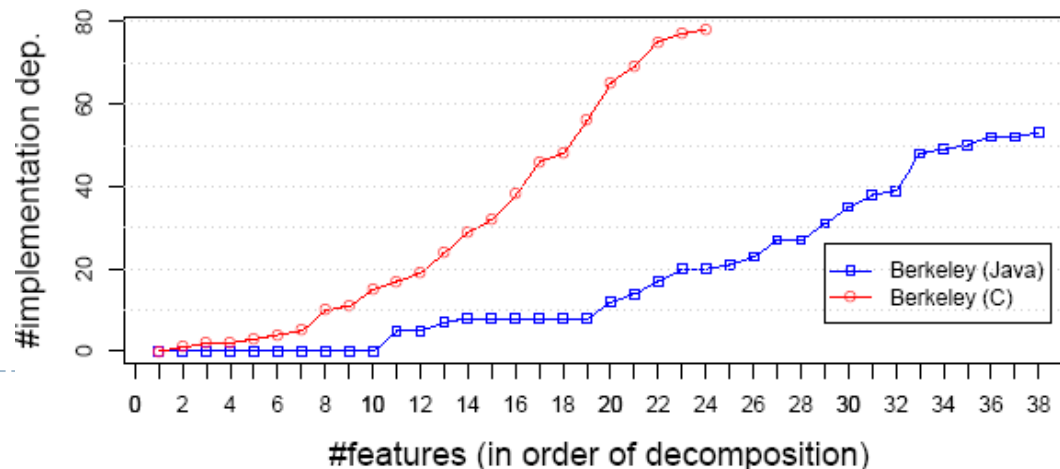
- Feature model
 - 38 features
 - 16 “req.” dependencies
 - 3.6 billion variants
- Implementation
 - 53 dependencies

(restricted variability!)

C version

- Feature model
 - 24 features
 - 8 “req.” dependencies
 - 1 million variants
- Implementation
 - 78 dependencies

(restricted variability!)



Dependencies are transitive

- ▶ „A requires B“,
„B requires C“
→ „A requires B and C“
- ▶ Consequence: single features can enforce the selection of many other features, considerably reducing variability
- ▶ Example: Berkeley DB
 - ▶ *Statistics* feature collects stats about different concerns, e.g., memory consumption, transactions, write accesses, buffer hit ratio and many more
 - ▶ Selecting *statistics* enforces selection of 14 (out of 37) additional features, including *transactions* and *caches*

Resolving feature interactions

Classification of feature interactions

▶ Dynamic vs. static

- ▶ **Dynamic:** during product execution; can lead to unexpected behavior, crashes, race conditions
- ▶ **Static:** during product generation/compilation; for example, calling an undefined method

▶ Domain vs. implementation

- ▶ **Domain dependency:** conceptually defined by domain; alternative implementations have the same dependencies
- ▶ **Implementation dependency:** comes from implementation decisions, alternative implementations may lead to different dependencies

Dynamic feature interactions

- ▶ Hard to detect
- ▶ Massively researched especially in telecommunications domain
 - ▶ M. Calder, M. Kolberg, E.H. Magill, S. Reiff-Marganiec.
Feature interaction: A critical review and considered forecast.
Computer Networks, Volume 41, Issue 1, 2003, pp. 115-141
- ▶ Approaches rely on massive testing or specialized representations of requirements
 - ▶ Formal specification, model checking, ...
- ▶ If found → “optional feature” problem

Focus: implementation dependencies

- ▶ Implementation dependencies are unpleasant
- ▶ Reduced variability, even if variants would in principle be desirable in domain
- ▶ Example: transactions vs. statistics
 - ▶ Solution 1: in the feature model, have statistics require transactions → reduced variability
 - ▶ Solution 2: no statistics about transactions → less useful implementation
- ▶ Therefore we seek options to resolve implementation dependencies

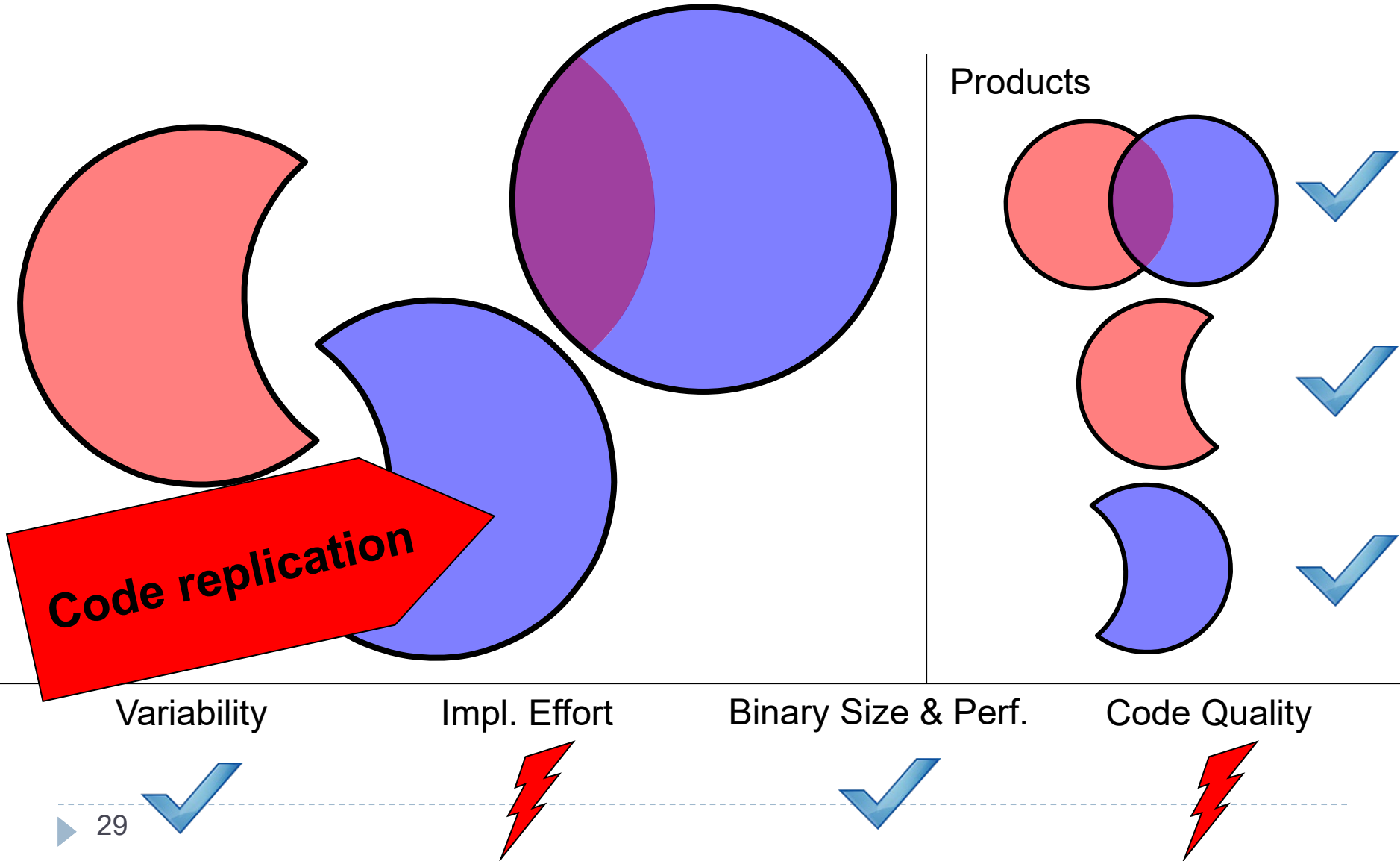
Solutions to the “optional feature” problem

- ▶ How to modularize two interacting features?

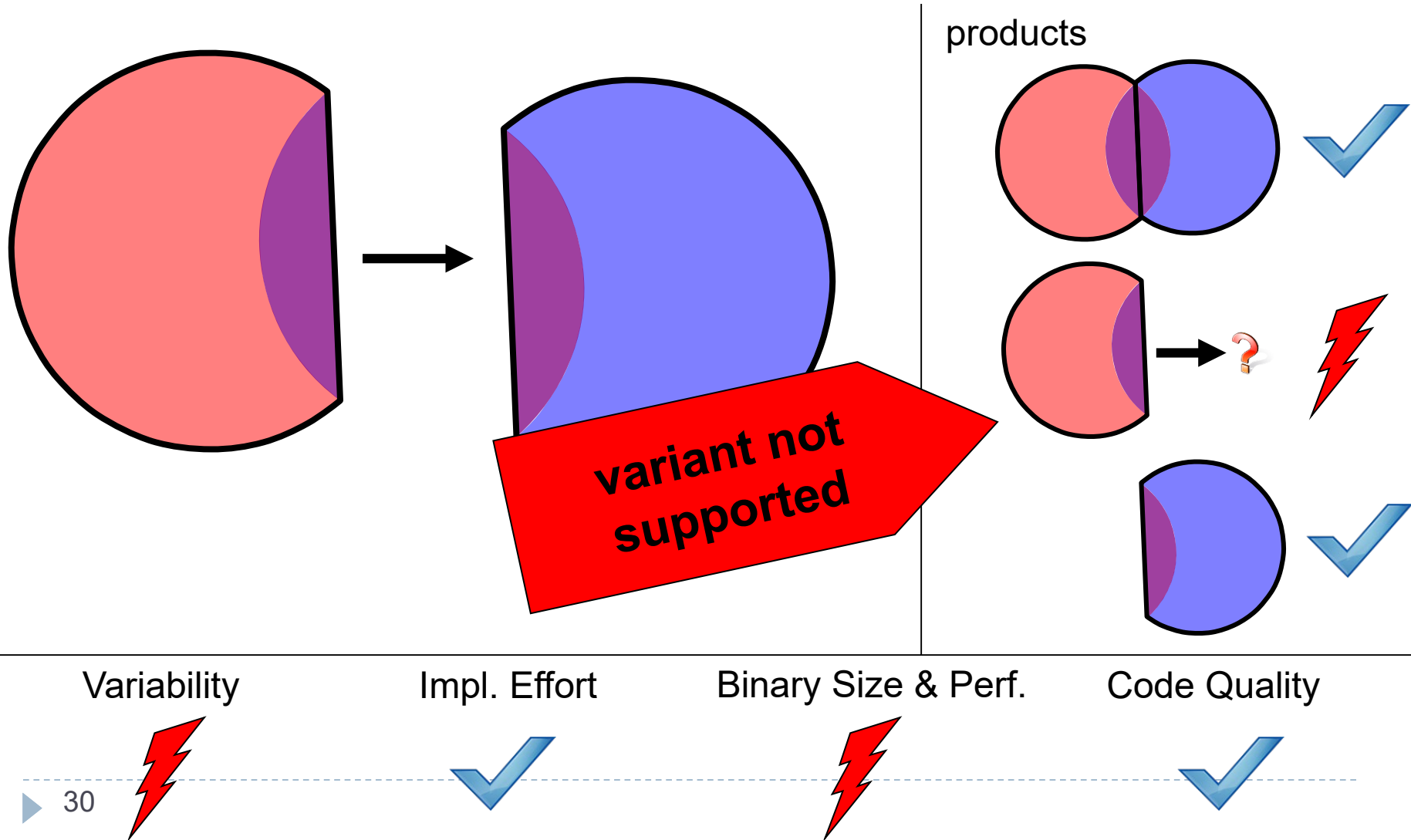


- ▶ Goals:
 - ▶ Support intended variability as expressed in feature model
 - ▶ Small implementation effort
 - ▶ Efficient implementation (code size, performance)
 - ▶ Code quality (separation of concerns, modularity)

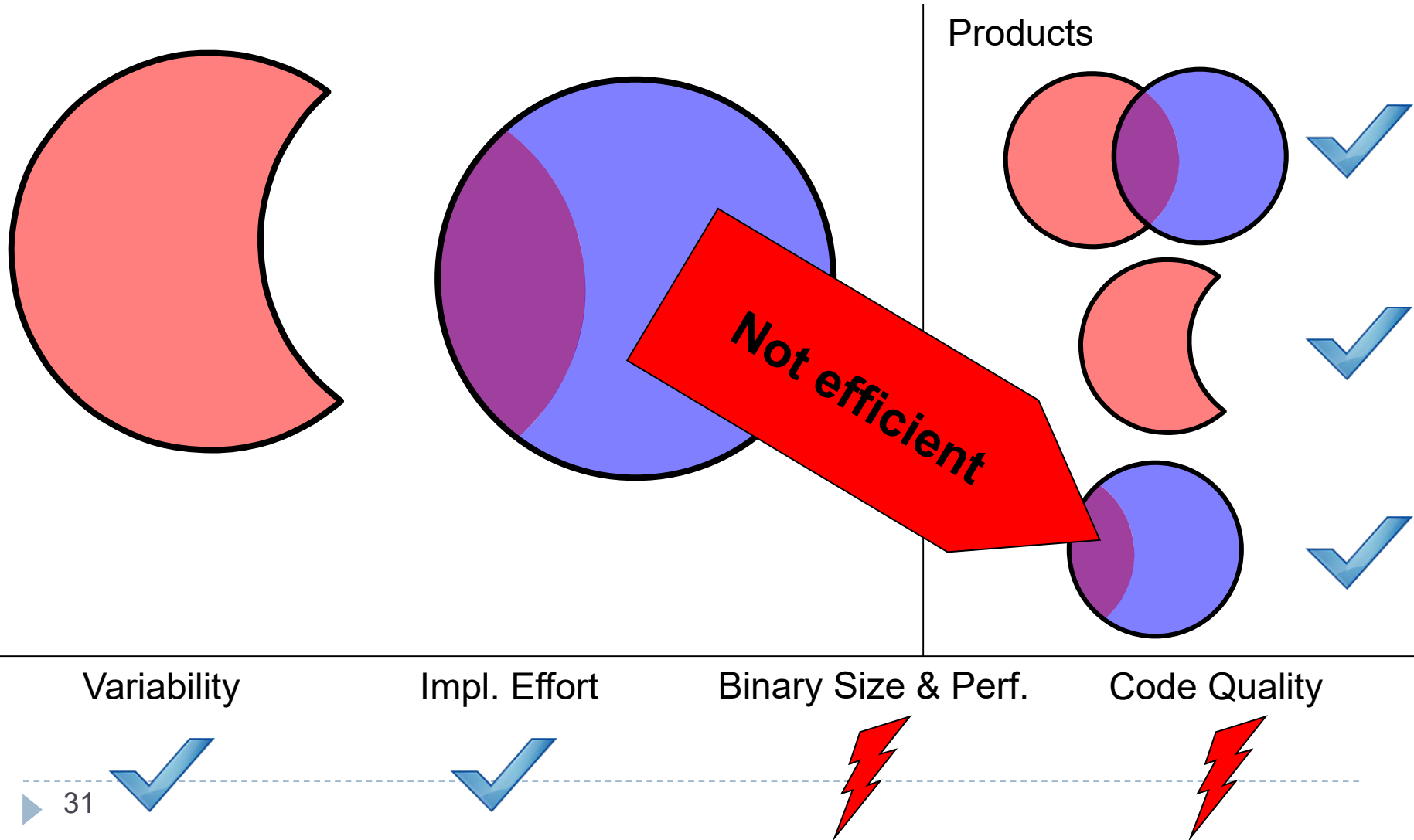
Solution 1: several implementations



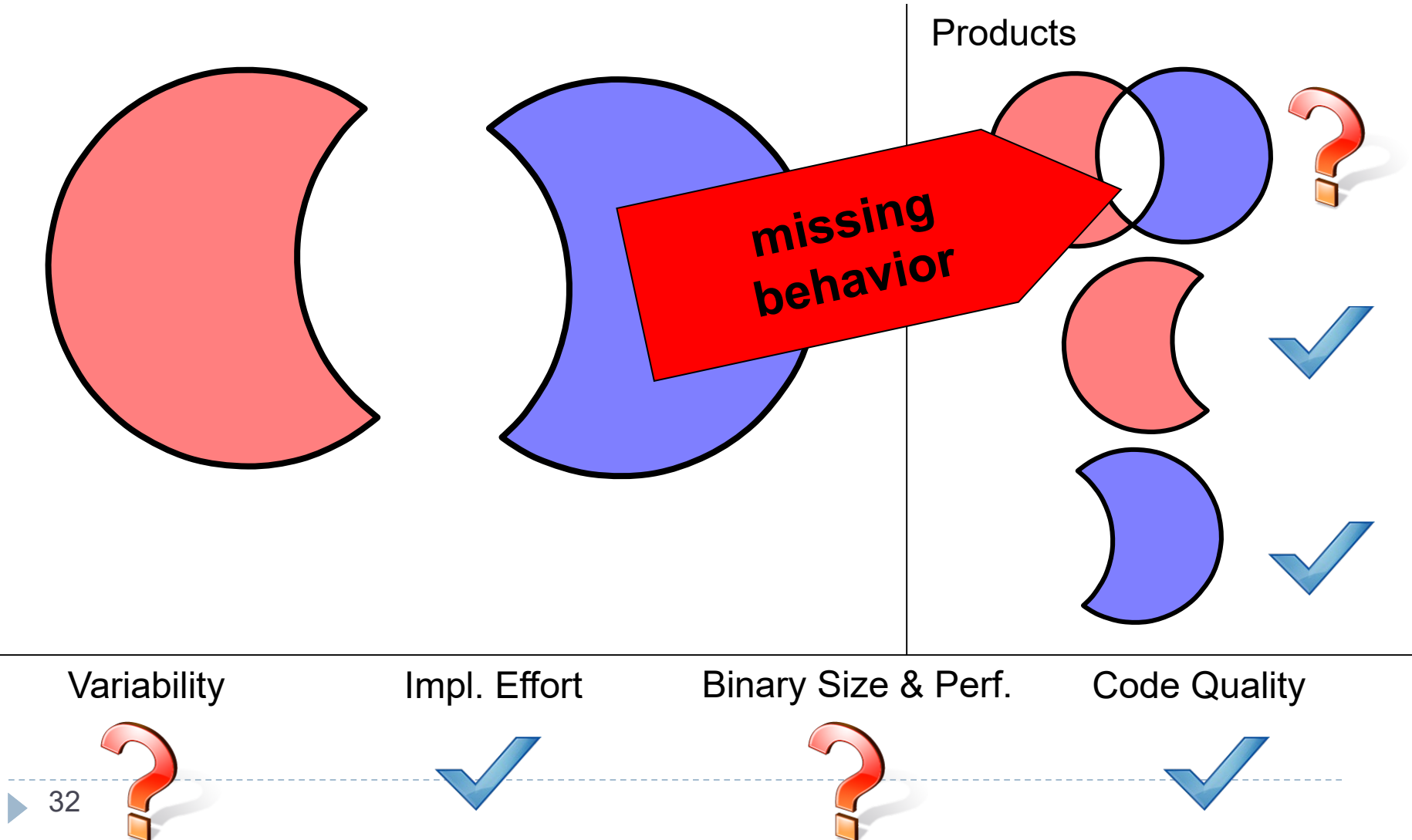
Solution 2: keep implementation dependencies, update feature model



Solution 3: move source code (until all dependencies removed)



Solution 4: change behavior (orthogonal implementations)



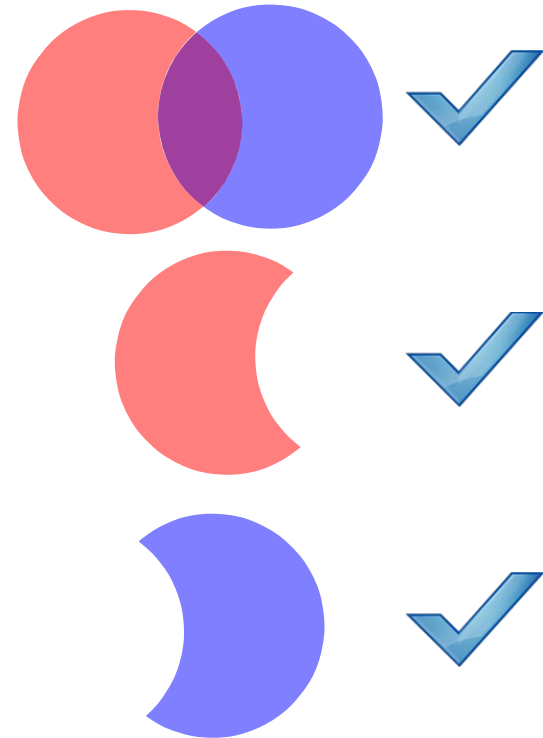
Solution 5: preprocessor

not modular



```
#ifdef TXN
lock();
#ifdef STAT
lockCount++;
#endif
#endif
```

Products



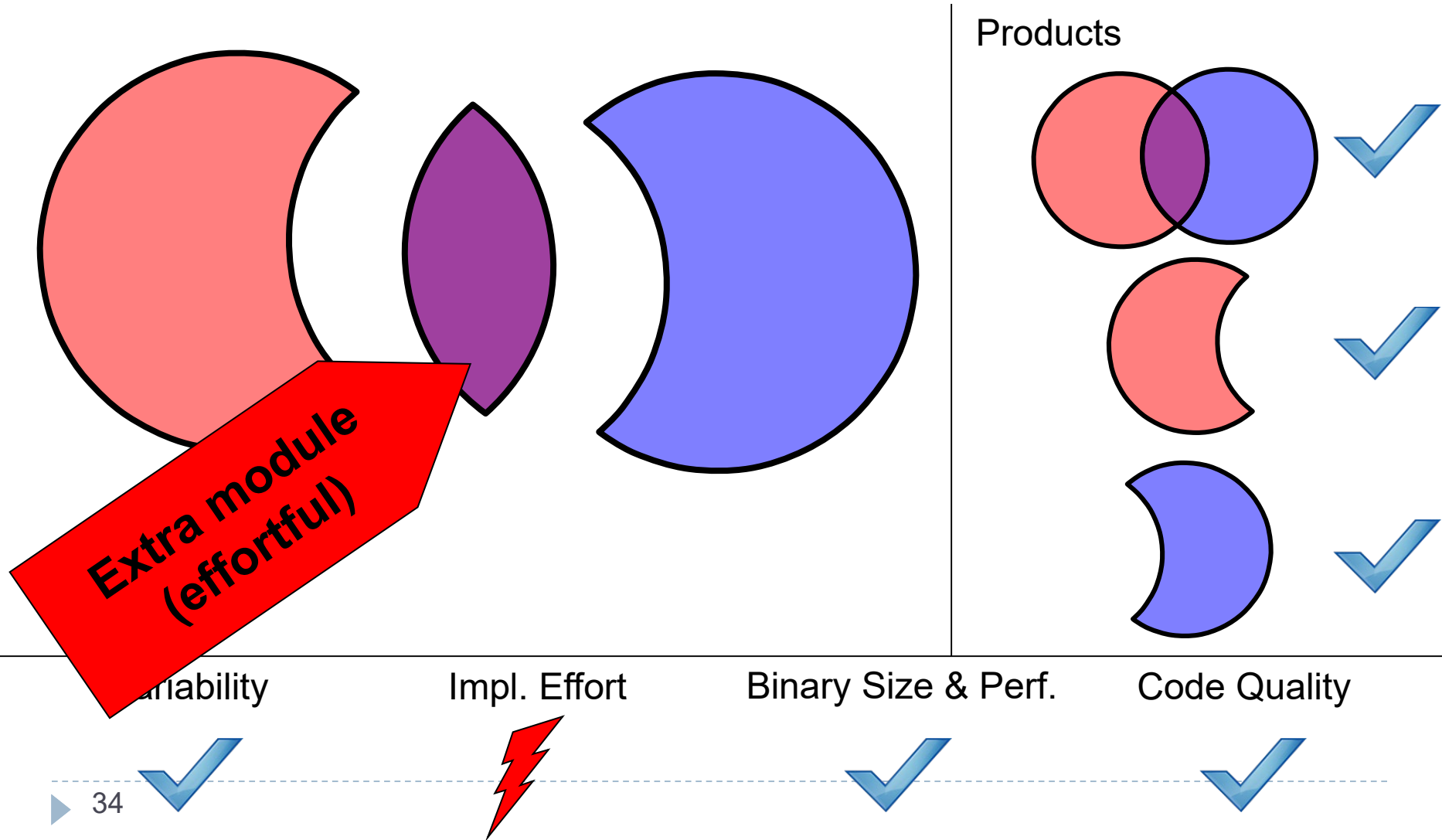
Variability

Impl. Effort
























Binary Size & Perf.

Code Quality

Solution 6: extract feature interactions (glue-code modules)



Overview of solutions

| Solution | Variability | Effort | Size, Performance | Quality |
|--------------------------|---|---|---|---|
| Multiple implementations |  |  |  |  |
| Keep dependencies |  |  |  |  |
| Move source code |  |  |  |  |
| Change behavior |  |  |  |  |
| Preprocessor |  |  |  |  |
| Extract interactions |  |  |  |  |

No single best solution

Example in detail: extracting interactions

- ▶ Feature modules for A and B (no interaction code)
- ▶ New module (A+B) that is automatically selected when A and B are selected
- ▶ A+B extends A or B if both are selected
- ▶ Then all 4 combinations possible
 - ▶ without A, without B
 - ▶ **with** A, without B
 - ▶ without A, **with** B
 - ▶ with A, with B (and with A+B)
- ▶ „Optimal“ implementation of all variants

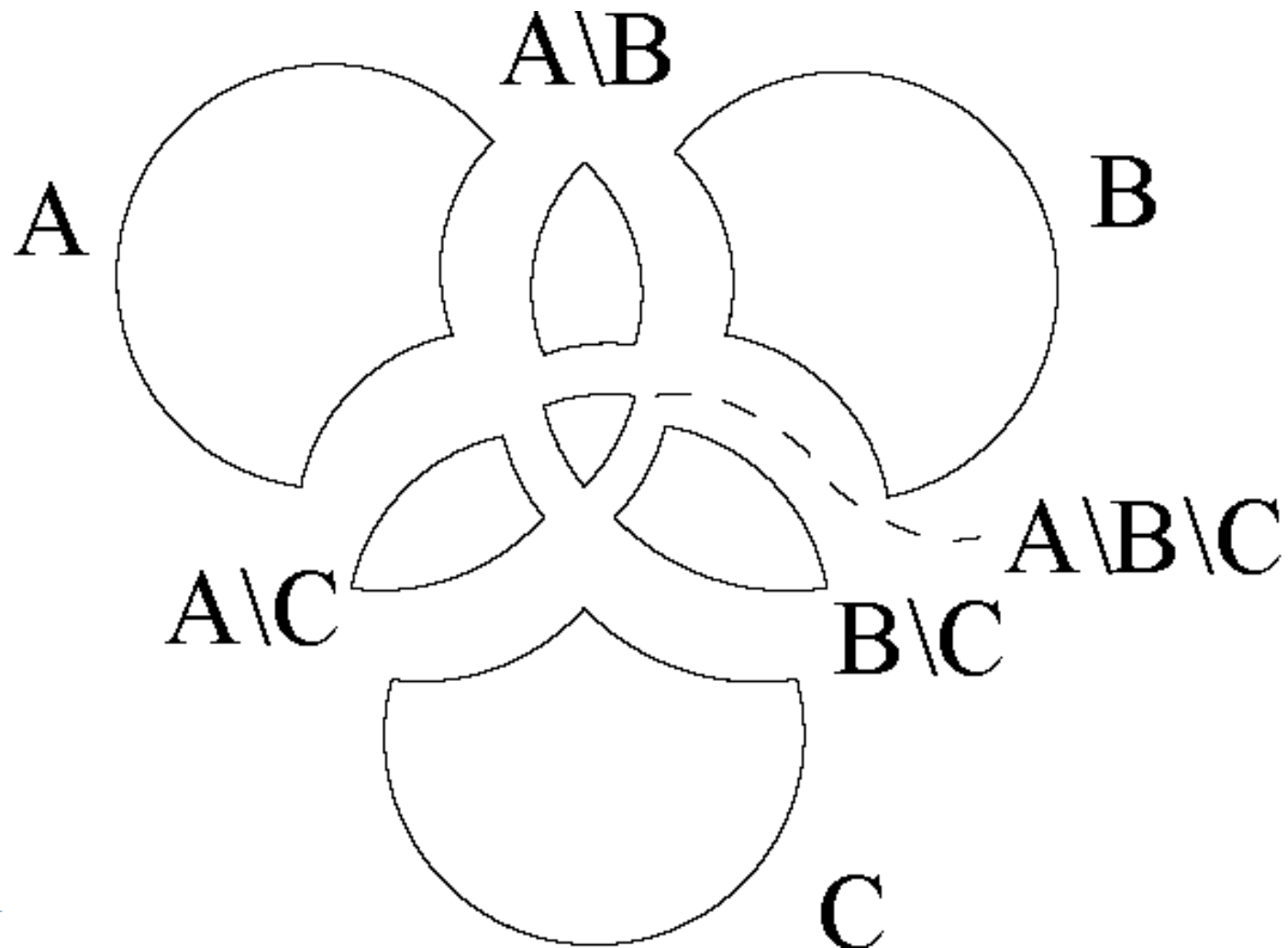
Problem

- ▶ Manual extraction is effortful
- ▶ Additional modules → higher complexity
- ▶ Interactions are often highly distributed, heterogeneous extensions
- ▶ Theoretically many extensions possible.
Number of pairwise extensions:

$$i_{\text{m}} \quad \text{a} \overline{\text{x}} \binom{n}{2} = \frac{n^2 - n}{2}$$

- ▶ n-wise (for $n > 2$) interactions possible, “higher-order interactions”

Higher-order interactions



Example for higher-order dependencies

```
class Stack {
  boolean push(Object o) {
    Lock lock=lock();
    if (lock==null) {
      log("lock failed for: "+o)';
      return false;
    }
    rememberValue ();
    elementData[size ++] = o;
    ...
  }

  void log(String msg) { /*...*/ }

  boolean undo () {
    Lock lock=lock();
    if (lock==null) {
      log("undo-lock failed")';
      return false;
    }
    restoreValue ();
    ...
    log("undone.")';
  }
}
```

Locking

Undo

Undo

Undo-Locking-Logging

Locking-Logging

Logging

Undo-Locking

Undo-Logging

Zoom quiz!

- ▶ How many feature interactions maximally possible for 4 features?



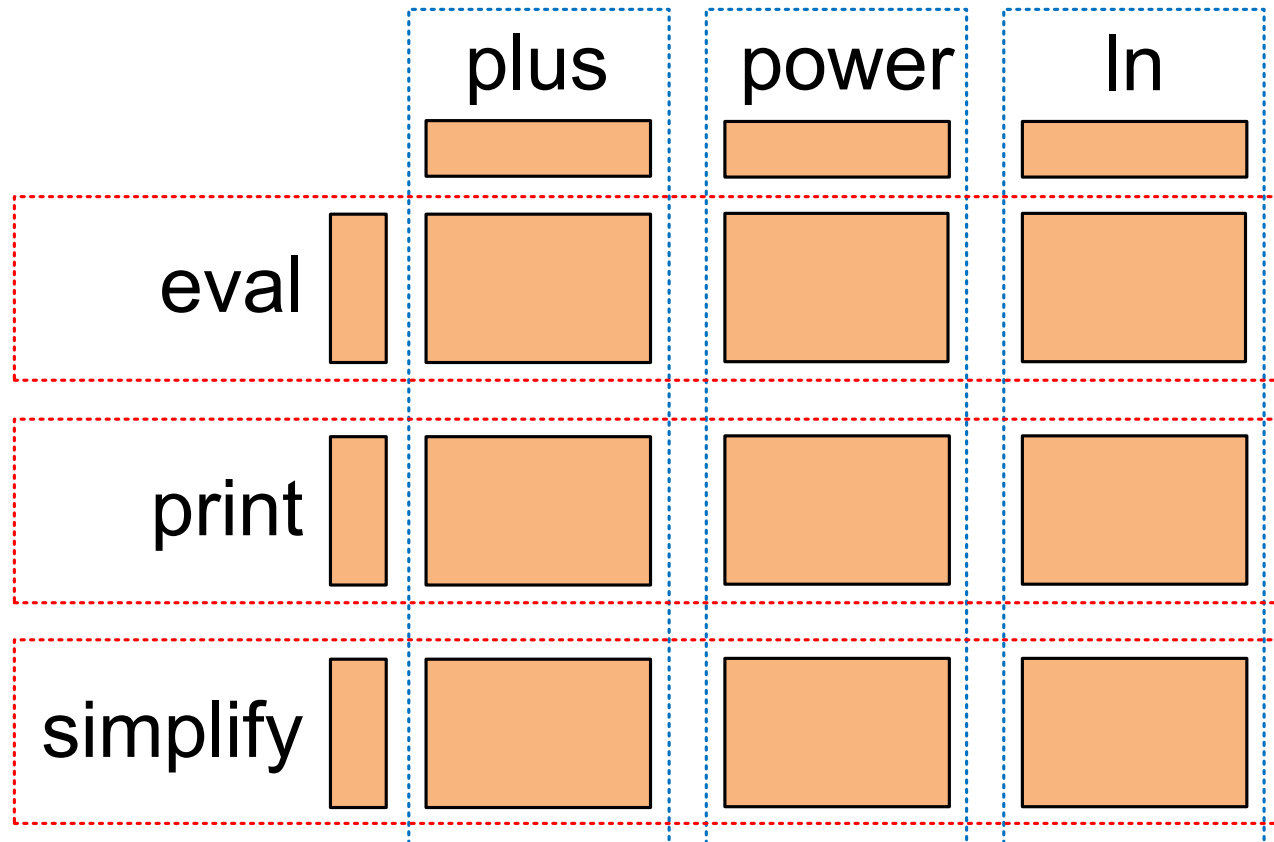
How many interactions?

- ▶ Theoretically possible
(with higher-order interactions)

$$h_m \stackrel{a}{=} \sum_{o=1}^{n-1} \binom{n}{o-1} = 2^n - n - 1$$

- ▶ Actual numbers in practice
 - ▶ From intuitive experiences, much lower
 - ▶ (Weak empirical support, more research required)

Expression problem: extracting interactions



- Almost all code extracted to new modules
- 6 features → 15 modules



Experiences

Refactoring Berkeley DB

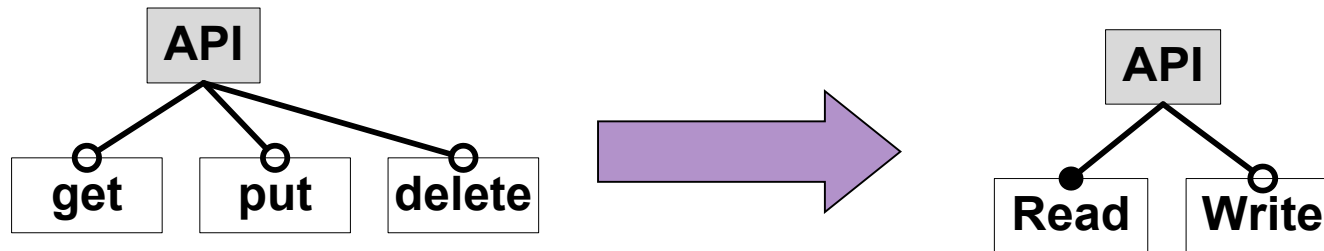
- ▶ Keep implementation dependencies?
 - ▶ Important features were de-facto mandatory (stats, transactions, memory management, ...)
- ▶ Change behavior?
 - ▶ Undesired, we do not want to compromise on usefulness
- ▶ Extract interactions?
 - ▶ 76% of statistics code extracted into 9 modules
 - ▶ → possible but effortful
- ▶ Preprocessor?
 - ▶ Much faster, simple
 - ▶ Extensive scattering and tangling



Experiences with FAME-DBMS

- ▶ Change feature model

- ▶ Avoiding 14 dependencies reduces product number by $\frac{3}{4}$



- ▶ Logging with preprocessor

- ▶ Avoids 11 dependencies, but scattered code

- ▶ B-tree can always be written

- ▶ Binary size increased by 5—13%

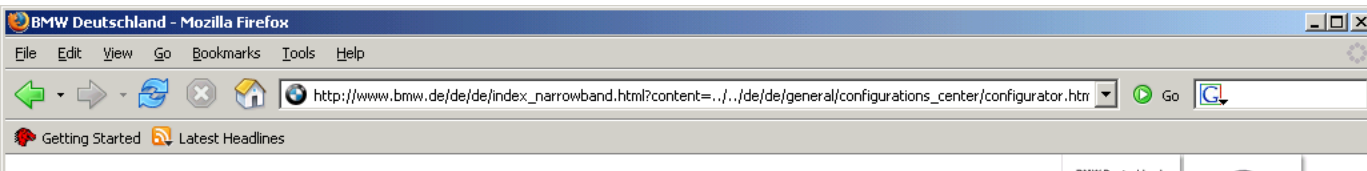
- ▶ 10 remaining interactions extracted

Discussion: variability in practice

Which interactions to resolve?

- ▶ Variability is no goal in itself
 - ▶ 33 optional independent features: one variant per person on Earth
 - ▶ 320 features: one variant per atom in the universe
 - ▶ Nobody can test all these variants; nobody needs all of them
- ▶ Therefore
 - ▶ Focus on required variants
 - ▶ Variability management, variability at the right place, domain analysis

Looking back: Car product lines (BMW, Audi)



Home 1 3 5 6 7 X3 X5 Z4 M Gebrauchte Automobile Service & Support Mein BMW Kontakt BMW Konfigurator BMW Händler & Service Partner Finanzieren & Versichern Shop

BMW Konfigurator

Zurück

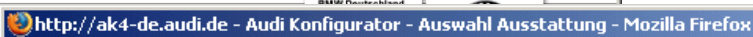
BMW Konfigurator

Tipps zum Car Configurator

| Modell | Grundausstattung | Farbe, Interieur + Felgen | Editionen + Pakete |
|--|------------------|---------------------------|--------------------|
| Getriebe Klima, Heizung Komfort/Nutzen Optik innen/außen Polsterung Kommunikation, Info Sicherheit Sportlichkeit | | | |
| Ausstattung | | | |
| Getriebe | | | |
| <input checked="" type="checkbox"/> Automatic Getriebe | | | |
| Klima, Heizung | | | |
| <input checked="" type="checkbox"/> Standheizung mit Fernbedienung | | | |
| <input type="checkbox"/> Sonnenschutzverglasung, Individual | | | |
| <input checked="" type="checkbox"/> Klimaautomatik mit Fondausströmern | | | |
| Komfort/Nutzen | | | |
| <input type="checkbox"/> Ablagenpaket | | | |
| <input type="checkbox"/> Armauflage vorn, verschiebbar | | | |
| <input type="checkbox"/> Außenspiegelpaket | | | |

Suche Sitemap Website Einstellungen

Waiting for ecom.bmwgroup.com...



Der Audi Konfigurator
Ihr A3 Ambiente

Außen

zu 'Ausstattungs Pakete' zu 'Räder/Reifen'

Für welche Ausstattung interessieren Sie sich?

☒ Alle ☐ Serienausstattung ☐ Sonderausstattung

☒ Alle Details anzeigen ☐ Ohne Details

Außenspiegel / Innenspiegel

Preis in EUR

| | |
|---|--------|
| <input checked="" type="radio"/> Außenspiegel elektrisch einstellbar, Gehäuse in Wagenfarbe lackiert, Spiegelglas links asphärisch, rechts konvex | 0,00 |
| <input type="radio"/> Außenspiegel beheizbar inklusive beheizbarer Scheibenwaschdüsen elektrisch einstellbar, links asphärisch, rechts konvex, Gehäuse in Wagenfarbe lackiert | 125,00 |

Ausstattungs Pakete

Außen

Räder/Reifen

Innen

Lenkräder

Sitze

Sicherheit/Technik

Infotainment

Fahrhilfen

Modell

Motor

Außenfarbe

Innenfarbe

Ausstattung

Finanzierung

Ihr Audi

Hilfe

Ihr Audi Ihre Möglichkeiten

| | |
|--|----------|
| Gesamtpreis: | 24.900,- |
| A3 Ambiente | |
| Motor Ambiente 1.4 TFSI 92(125) 22.900,- | |
| kW (PS) 6-Gang Kraftstoffverbrauch kombiniert: 6,5 l | |
| CO ₂ -Emission kombiniert: 154 g/km | |
| Außenfarbe Liquidblau Metallic | 0,- |
| Innenraum | |

Car product lines 20 years ago

- ▶ Choice was restricted to models and a few extras, like roof-mounted luggage rack or alternative CD player
- ▶ One variant (Audi 80, 1.3l, 55PS) = 40% of sales



Car product lines shortly ago

- ▶ 10^{20} possible variants of an Audi
 10^{32} possible variants of a BMW
- ▶ Hardly a car leaves the production plant with the same features as the one before it
- ▶ 100 different floor units for one model, based on engine and equipment
- ▶ 50 different steering wheels (3 vs. 4 spokes, wood vs. plastic vs. leather, heating, colors)



Variant management for car product lines

- ▶ First analyse which variants are actually desired
 - ▶ Exclude exotic variants
 - ▶ „We do not want to develop and produce parts that won't be used in the end“
- ▶ Variant management early in the development cycle
 - ▶ Audi was able to save 5 million € by reducing variability of roof: designed central user interface in a neutral way that fit all variants
 - ▶ BMW has reduced number of floor units from 100 to 4: left/right steering wheel; with and without sunroof. Differences were not user-visible anyways.

Variability in software product lines

- ▶ Offer required variability
- ▶ Avoid unnecessary variability
 - ▶ Reduce development cost
 - ▶ Reduce test cost
 - ▶ Reduce maintenance cost
- ▶ For example:
 - ▶ Decouple *shortest path* from *weighted* if demand exists, otherwise leave as is
 - ▶ Only separate stats and transactions with glue code if demand exists

Summary

- ▶ Dependencies between features through feature interactions
- ▶ Resolve implementation dependencies with additional modules
- ▶ Variant management makes sense

Outlook

- ▶ Feature interactions are an important variability problem in product lines and an open research field
- ▶ Dynamic interactions are hard to detect
- ▶ Can formalization and tool support help?



Future Solutions?

Current research:

Automated decomposition

- ▶ How to detect static interactions in source code?
- ▶ Is it possible to split up the code base into feature modules and interaction modules?
- ▶ Need to address cost of having too many modules

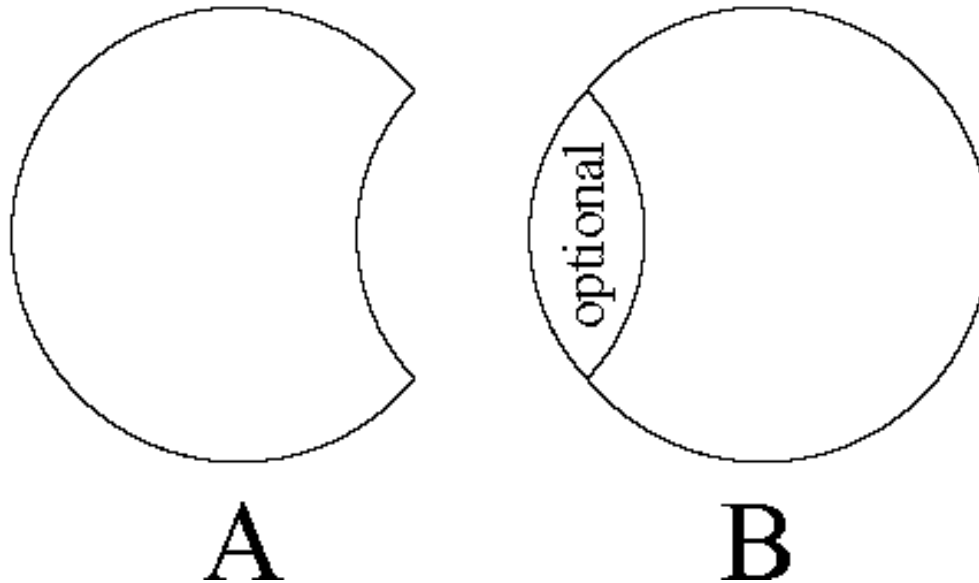
Current research: optional weaving

- ▶ AOP allows to define pointcuts regardless of if a corresponding join point is available
- ▶ If no join points available, code is not executed, but there is also no error

```
aspect Locking {  
    Object around():  
        execution(void Database.put(Object, Object)) ||  
        execution(Object Database.get(Object)) ||  
        execution(int Statistics.calculateDbSize()) {  
        lock();  
        Object r = proceed();  
        unlock();  
        return r;  
    }  
}
```

Optional weaving II

- ▶ Exploit AOP behaviour: implement interactions in optional code fragments
- ▶ If target feature not available, code fragment is just ignored

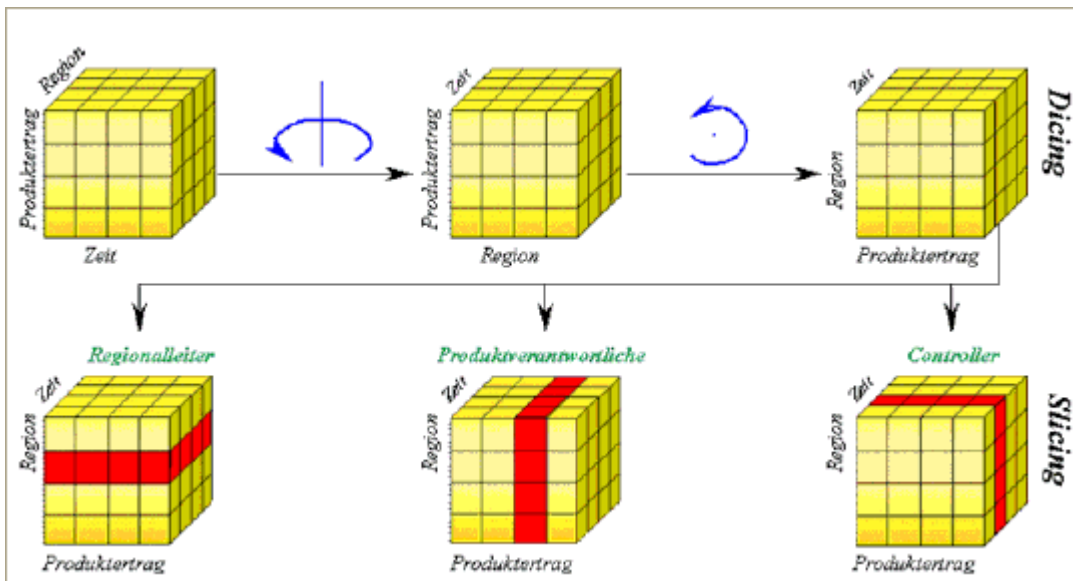


Optional weaving III

- ▶ Benefit: no additional modules
- ▶ Problems:
 - ▶ Hard to check correctness
 - ▶ AspectJ cannot be used for this due to technical limitations (for example, no optional inter-type declarations)
 - ▶ Not consequently implemented in available tools and cases

Program Cubes

- ▶ Tool-supported
- ▶ Views
- ▶ On-demand modularization



| | plus | power | In |
|----------|------|-------|----|
| eval | | | |
| print | | | |
| simplify | | | |

Literature

- ▶ J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In Proc. Int'l Conf. on Software Engineering, 2006.
[On resolution of interactions with additional modules]
- ▶ C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In Proc. Int'l Software Product Line Conference (SPLC), 2009.
[“Optional feature” problem and different solutions]
- ▶ M.-S. Andres. Die Optimale Varianz. brand eins, 2006(1)
[Variants in the automotive domain]