1.– 4. September 2014
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Auf die inneren Werte kommt es an

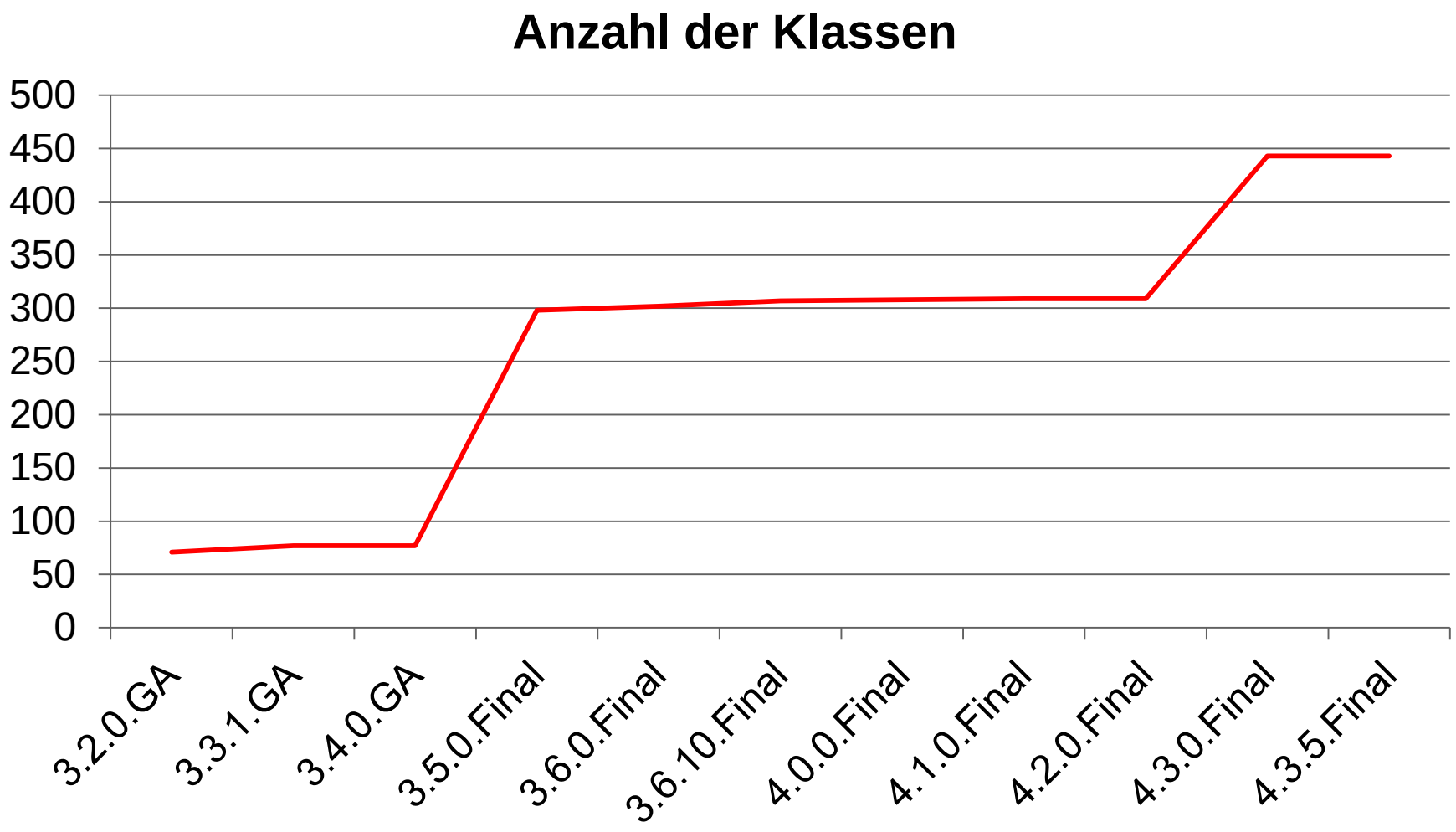
Kontinuierliche Architekturvalidierung

Kai Spichale

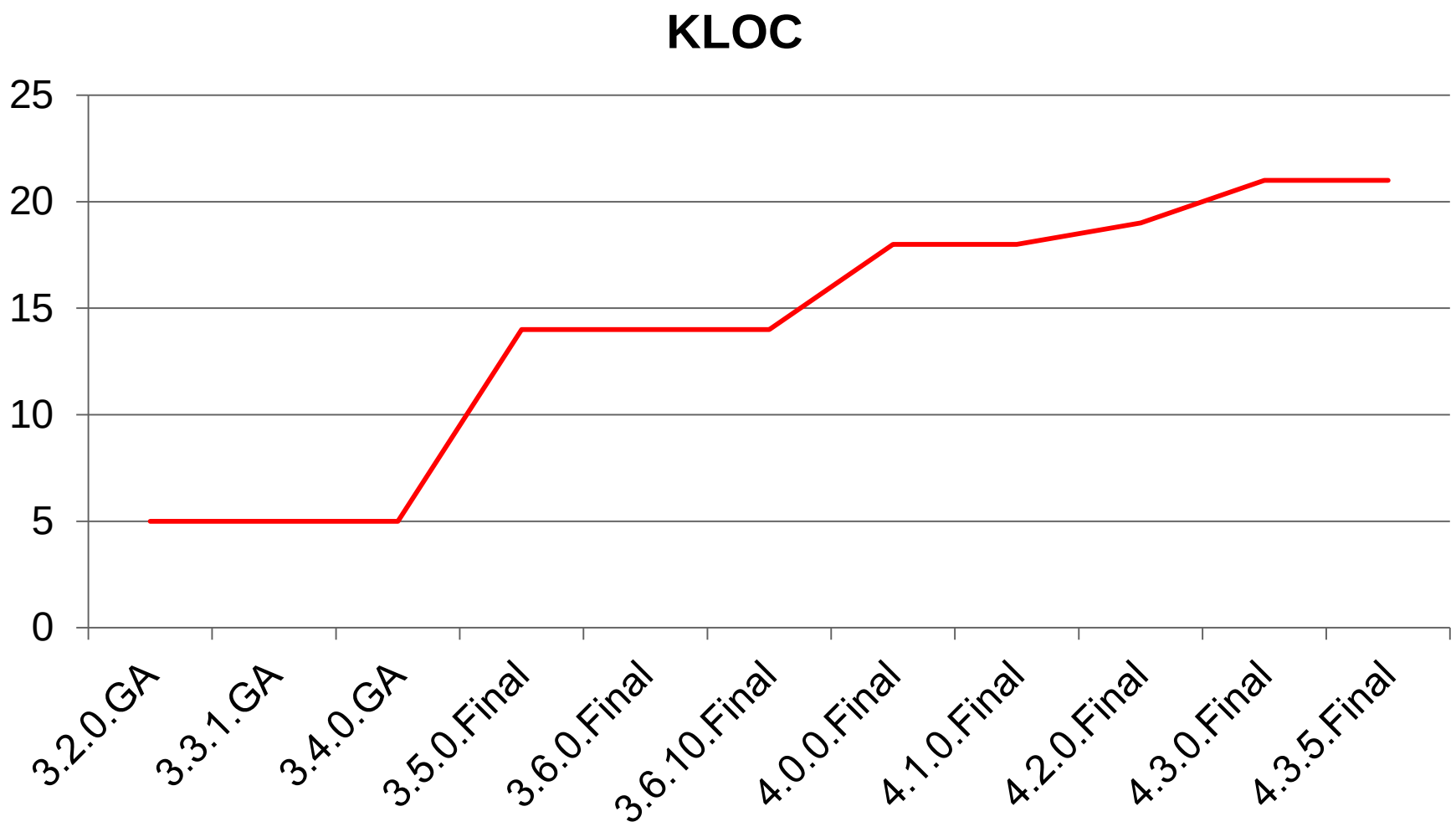
adesso AG

Wie äußert sich Software-Erosion bei Weiterentwicklung einer Softwarelösung?

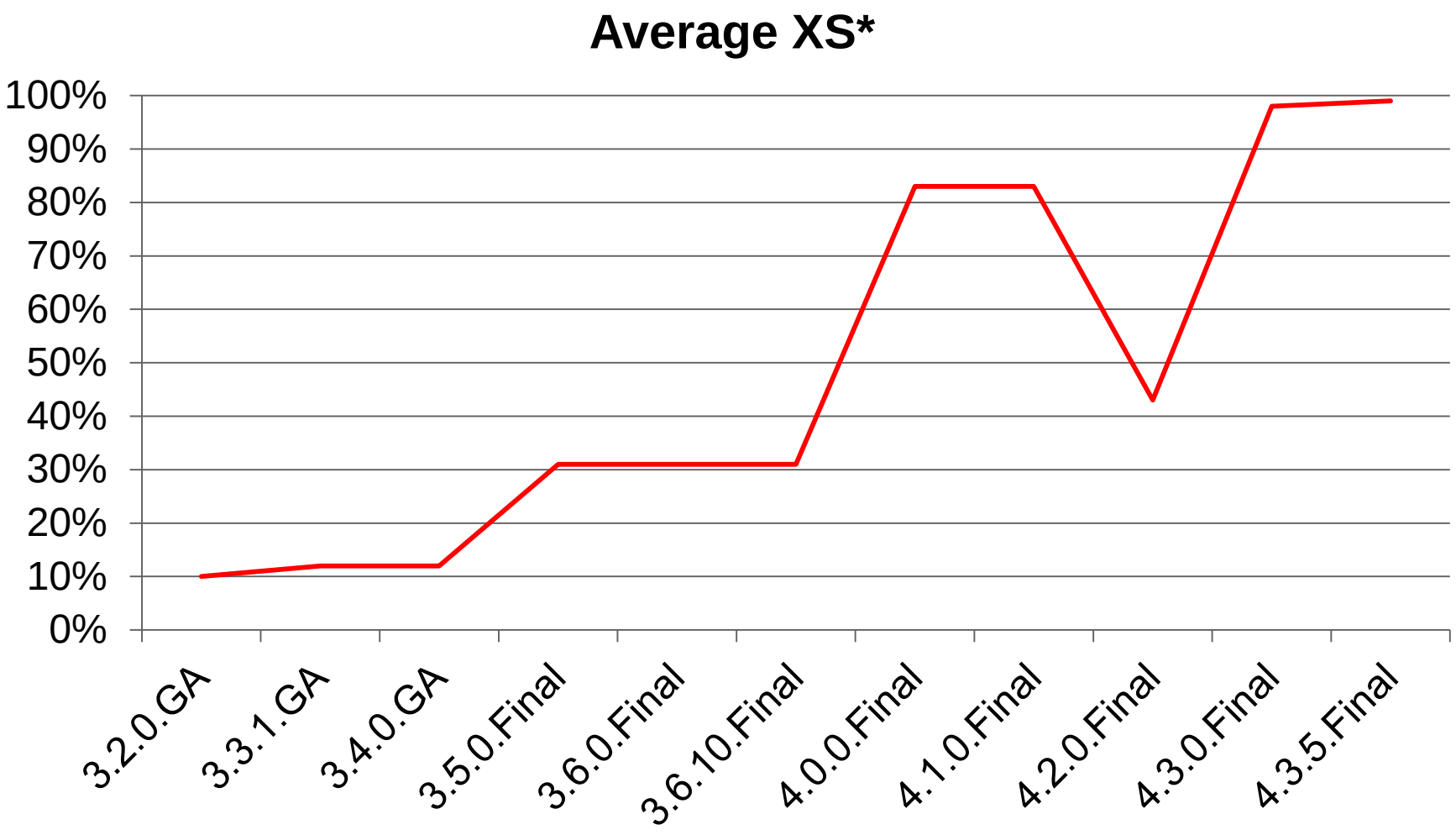
hibernate-entitymanager.jar



hibernate-entitymanager.jar

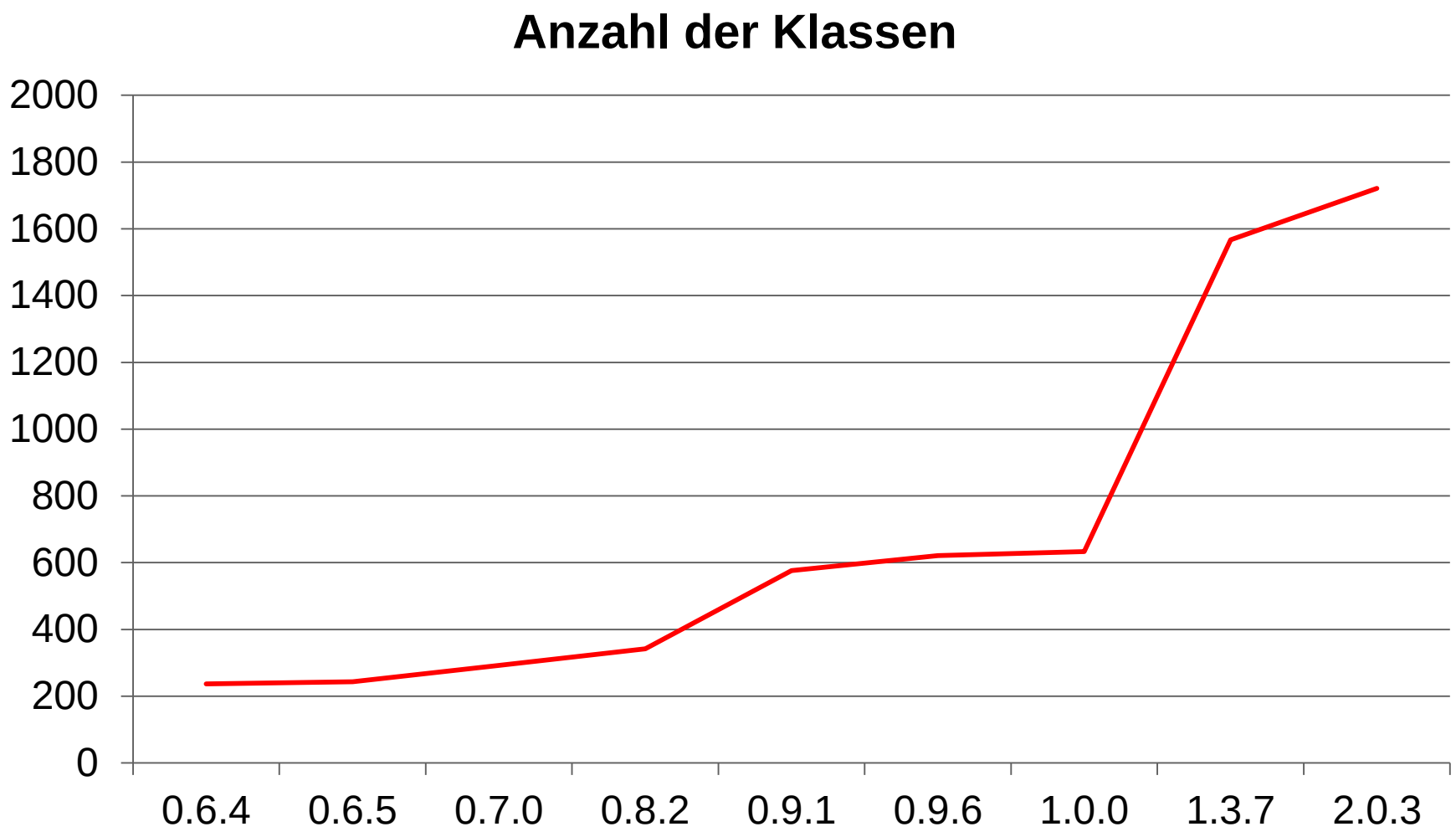


hibernate-entitymanager.jar

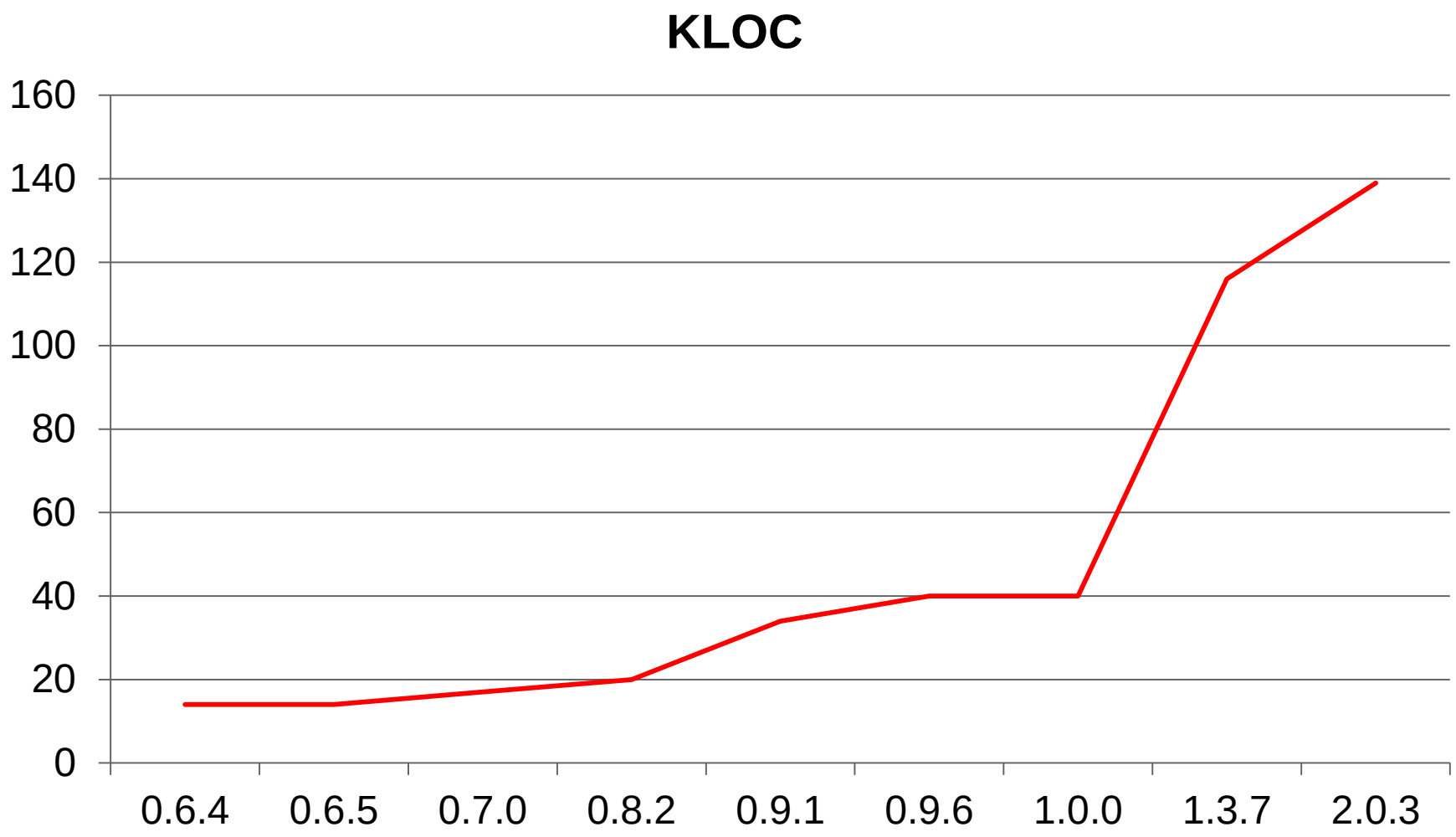


*Komplexitätsmetrik von Structure101

findbugs.jar

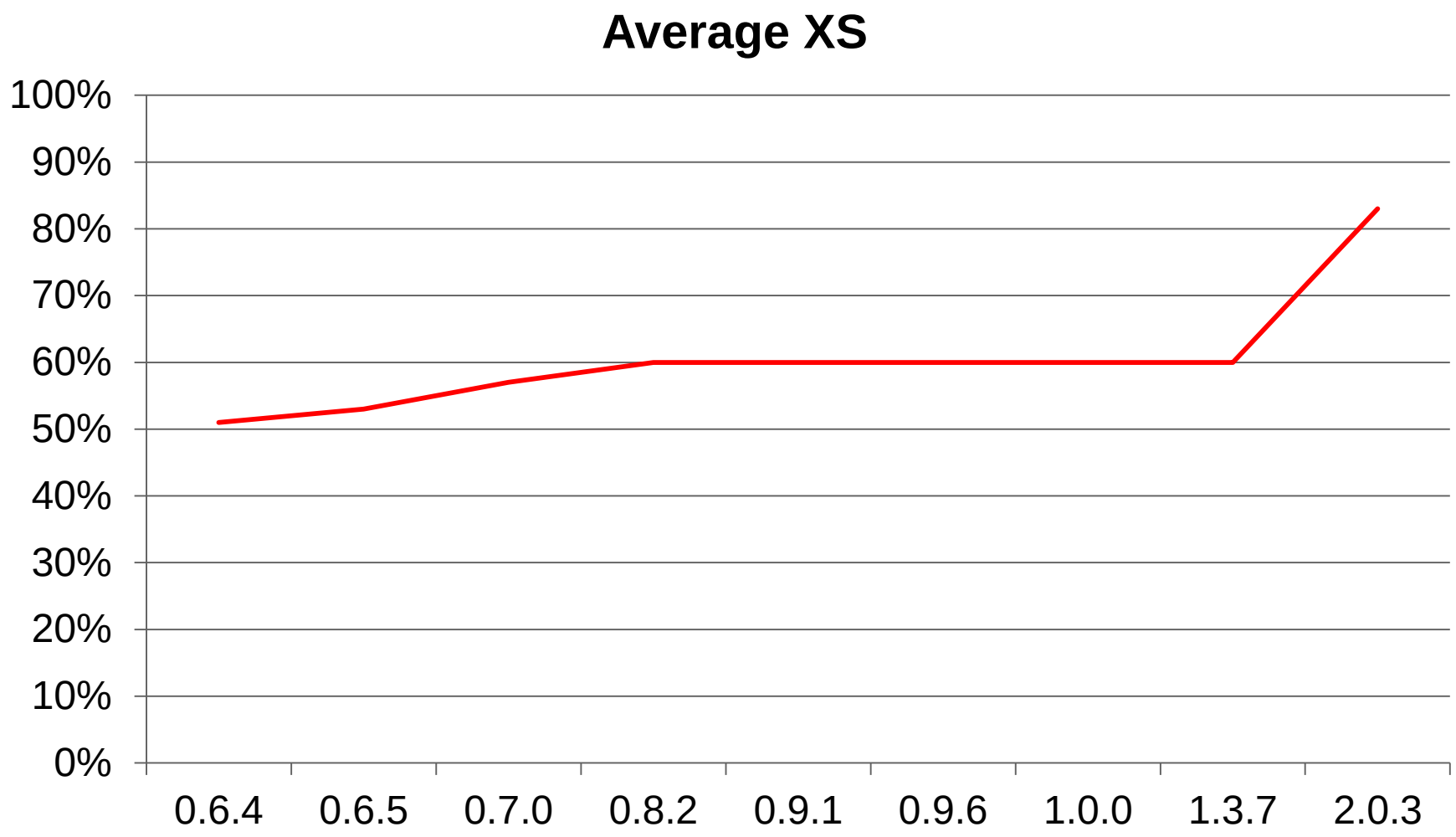


findbugs.jar

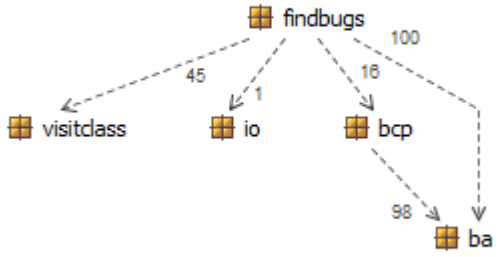


adesso

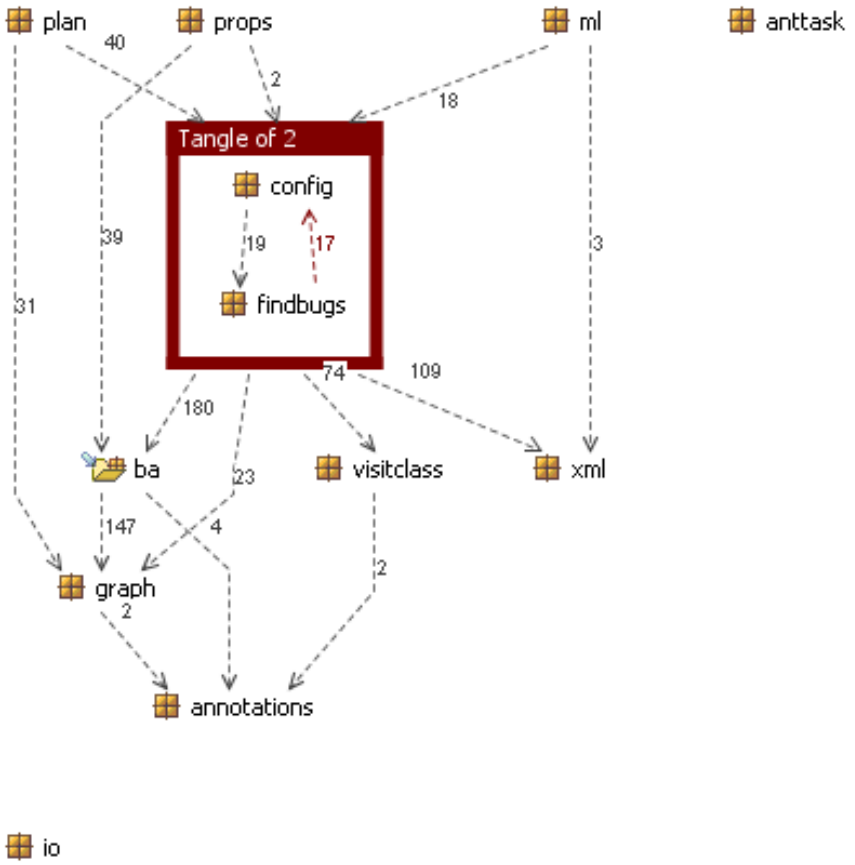
findbugs.jar



findbugs-0.6.4.jar

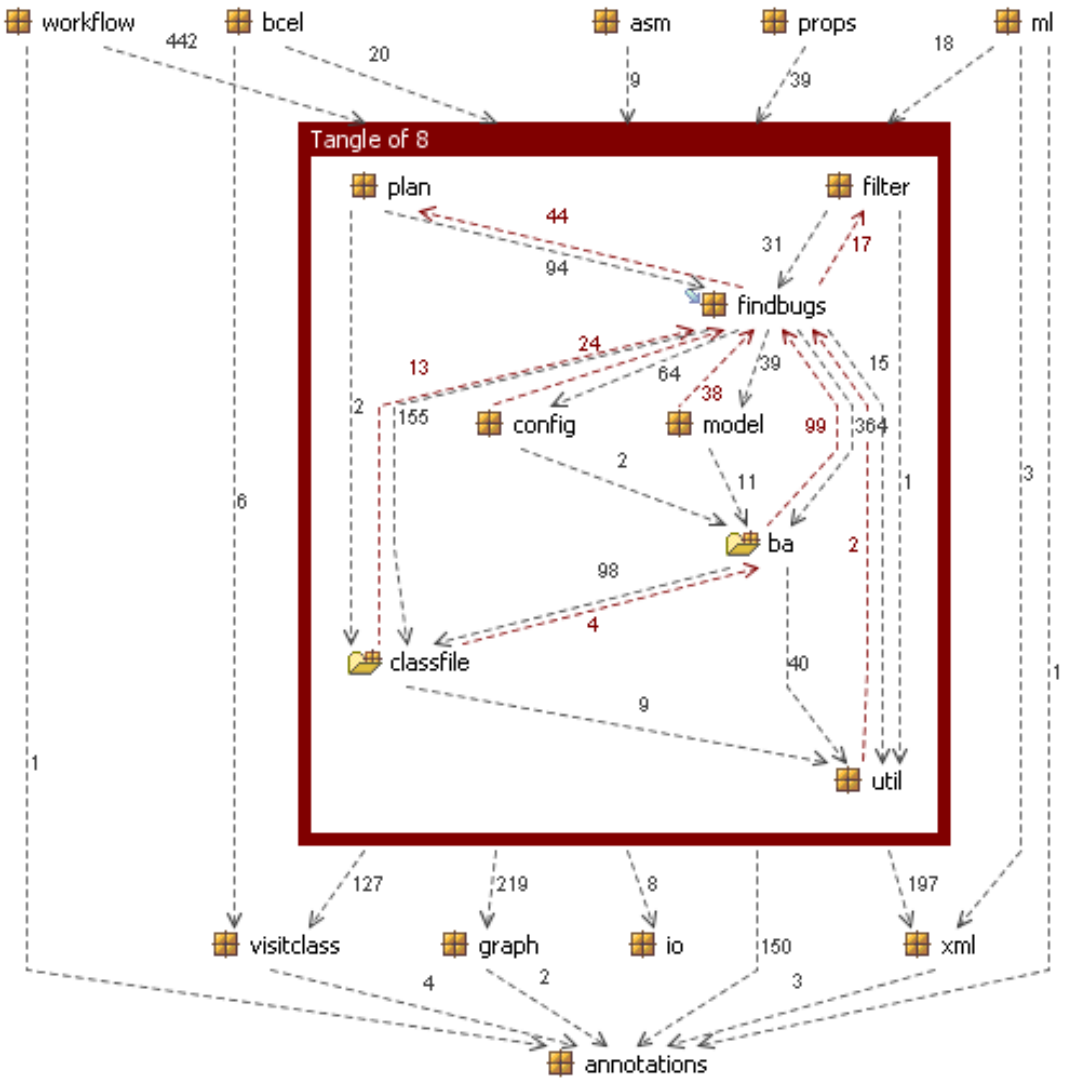


findbugs-0.8.7.jar



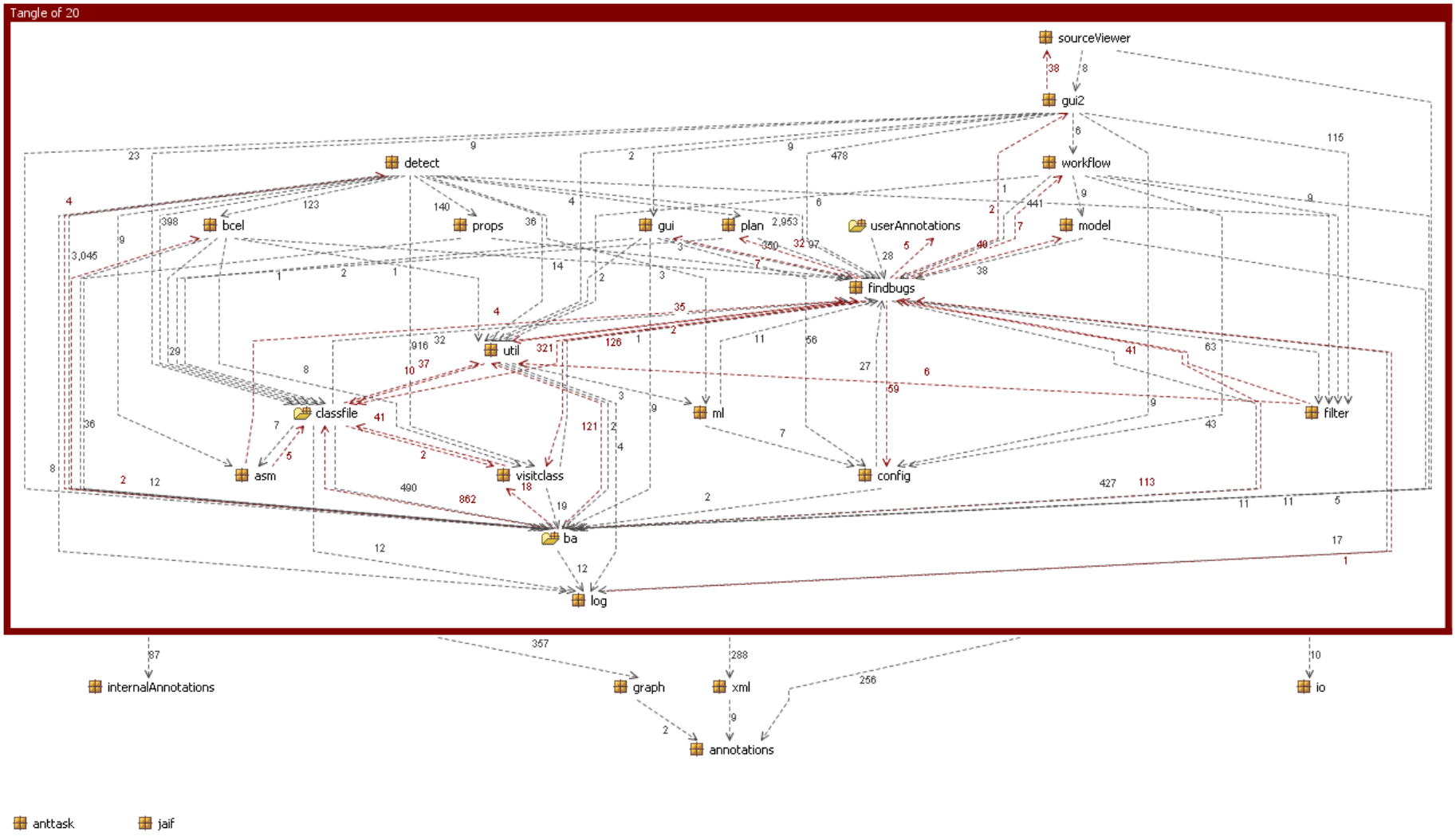
adesso

findbugs-1.0.0.jar

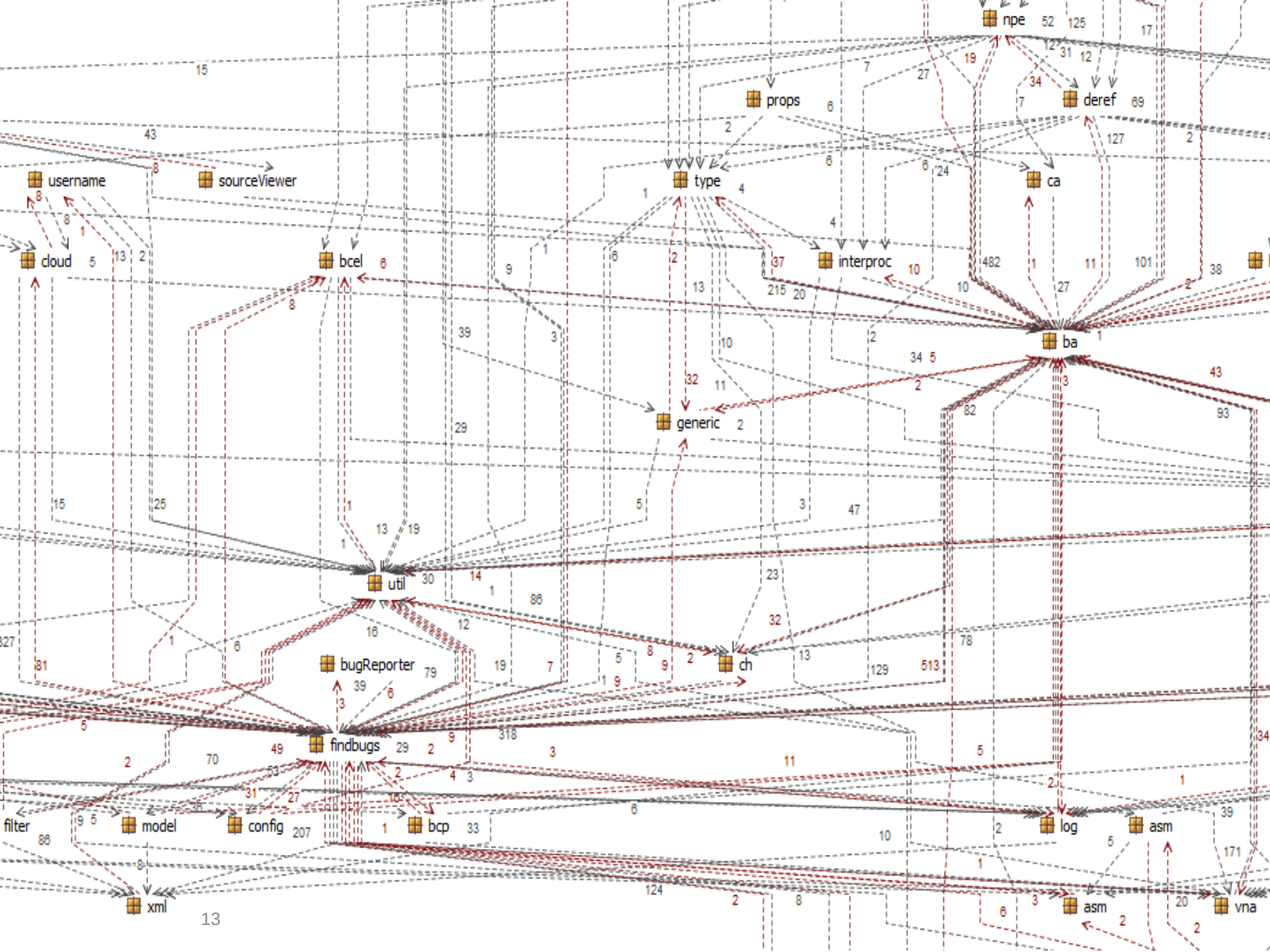


adesso

findbugs-1.3.7.jar



adesso


npe
props
deref
username
sourceViewer
type
ca
cloud
bcel
interproc
ba
generic
util
ch
bugReporter
findbugs
filter
model
config
bcp
log
asm
xml
asm
vna

David Parnas – Pionier der Softwaretechnik

Idee

- Geheimnisprinzip als Modularisierungs-kriterium

Ziel

- Flexibilität, Verständlichkeit verbessern
- Entwicklungszeiten verkürzen

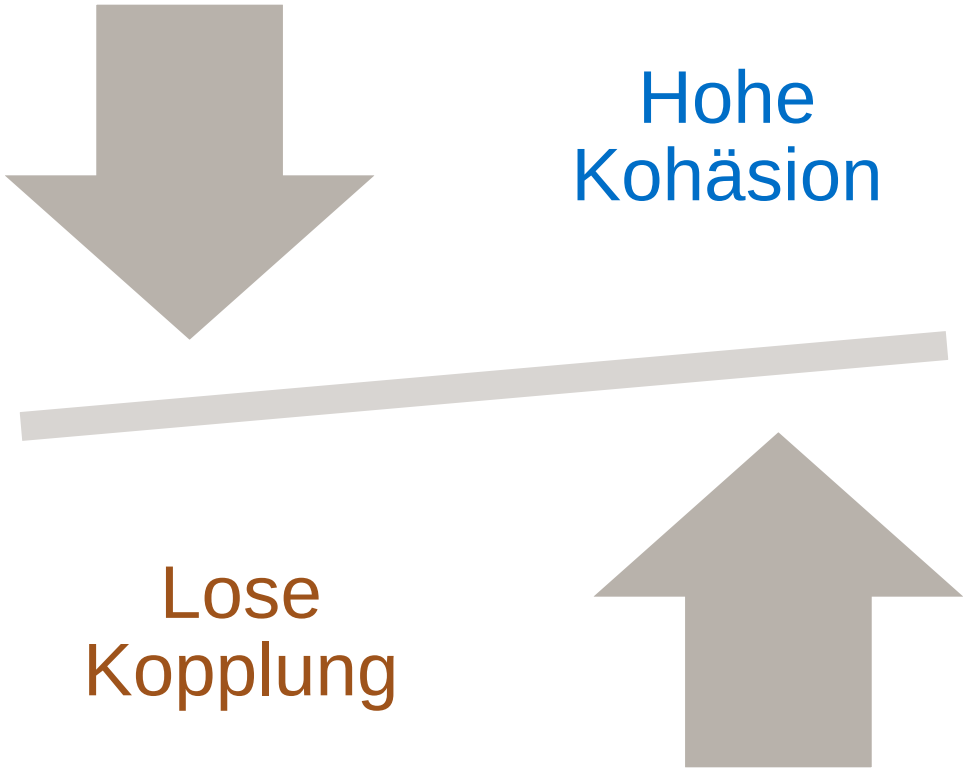
Larry Constantine – Pionier der Softwaretechnik

Herausforderung

- Objektive Bewertung alternativer Entwürfe

Lösungsansatz

- Kopplung: Grad der intermodularen Abhängigkeit
- Kohäsion: Grad der intramodularen funktionalen Beziehungen


Hohe
Kohäsion
Lose
Kopplung

Softwaremetriken schaffen Vergleichs- und Bewertungsmöglichkeiten

"You can't control
what you can't measure“

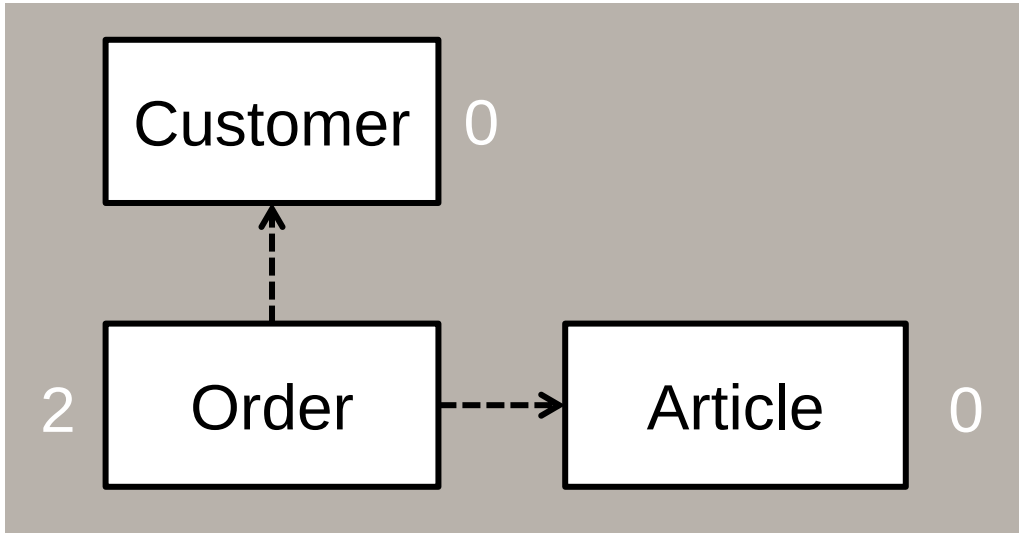
Tom DeMarco, 1982

Kopplungsmetriken

Kopplung messen

Coupling Between Object Classes (CBO)

von Chidamber & Kemerer

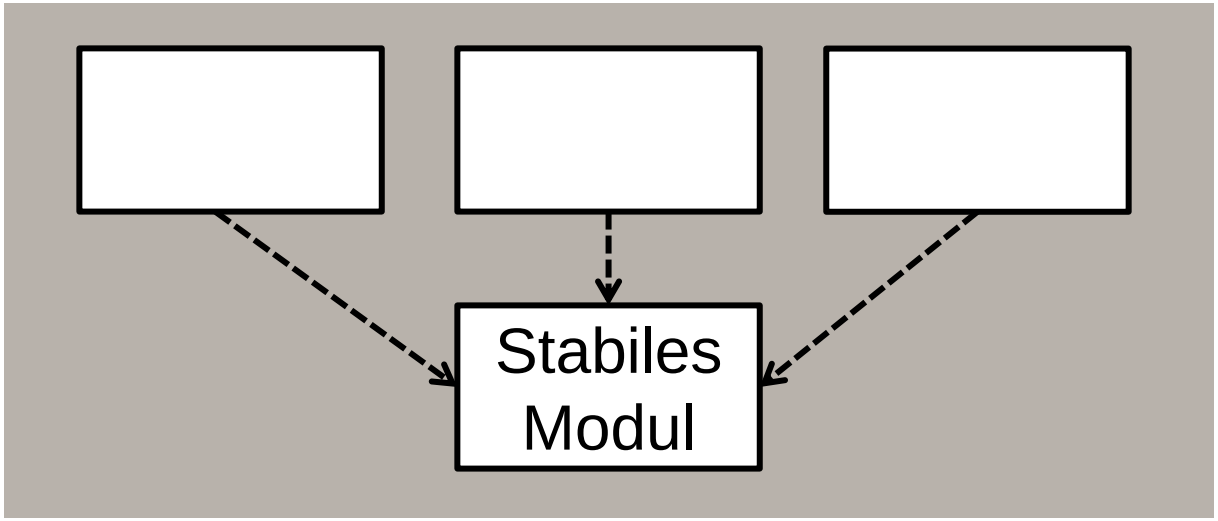


Alternativen

- Fan Out, Fan In
- Efferent Coupling (Ce), Afferent Coupling (Ca)

Stabilität erkennen

- Stabilität hängt vom Änderungsaufwand ab



Instabilität = Ce / (Ca + Ce)

= 0 / (3 + 0) = 0 => Modul ist stabil

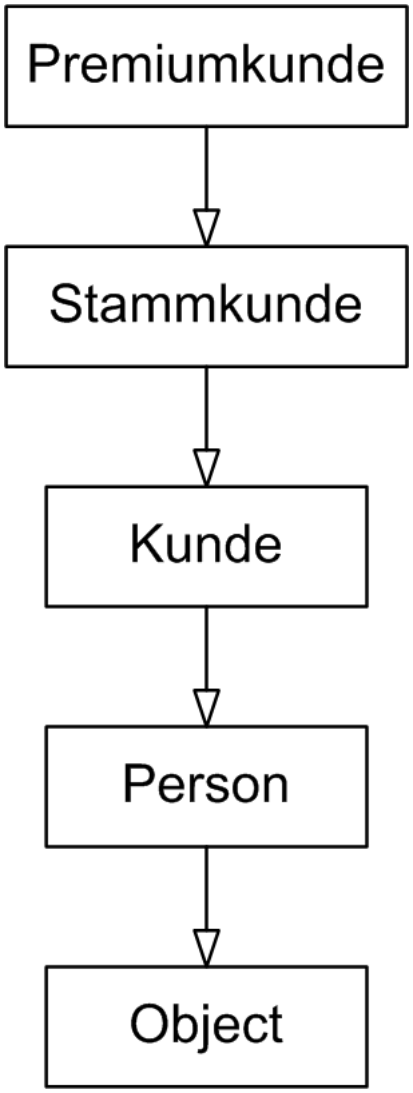
- Ein Package sollte nur von Packages abhängen, welche stabiler sind als es selbst.

Unnötig starke Kopplung

Depth of Inheritance Tree (DIT)

Chidamber & Kemerer

- Tiefe Vererbungsbäume deuten auf einen komplexen Entwurf
- Statistischer Zusammenhang zwischen Fehlerhäufigkeit und DIT



Unnötig starke Kopplung

Zeitliche Kopplung

- Zeitabhängige Verknüpfung von Code-Elementen
  - > API mit nicht offensichtlicher Reihenfolgeerwartung

- Beispiel: java.util.Iterator

```
it.remove();
it.next();
it.remove();
```

```
public class Compiler {
  protected JavaCompiler javac;
  public Compiler(JspCompilationContext ctxt) {
    this.ctxt = ctxt;
  }
  public boolean compile() {
    // Ohne Compiler wird hier abgebrochen
    if(javac == null) {
      return true;
    }
    …
  }
  public void setJavaCompiler(JavaCompiler javac) {
    this.javac = javac;
  }
}
```

Unnötig starke Kopplung

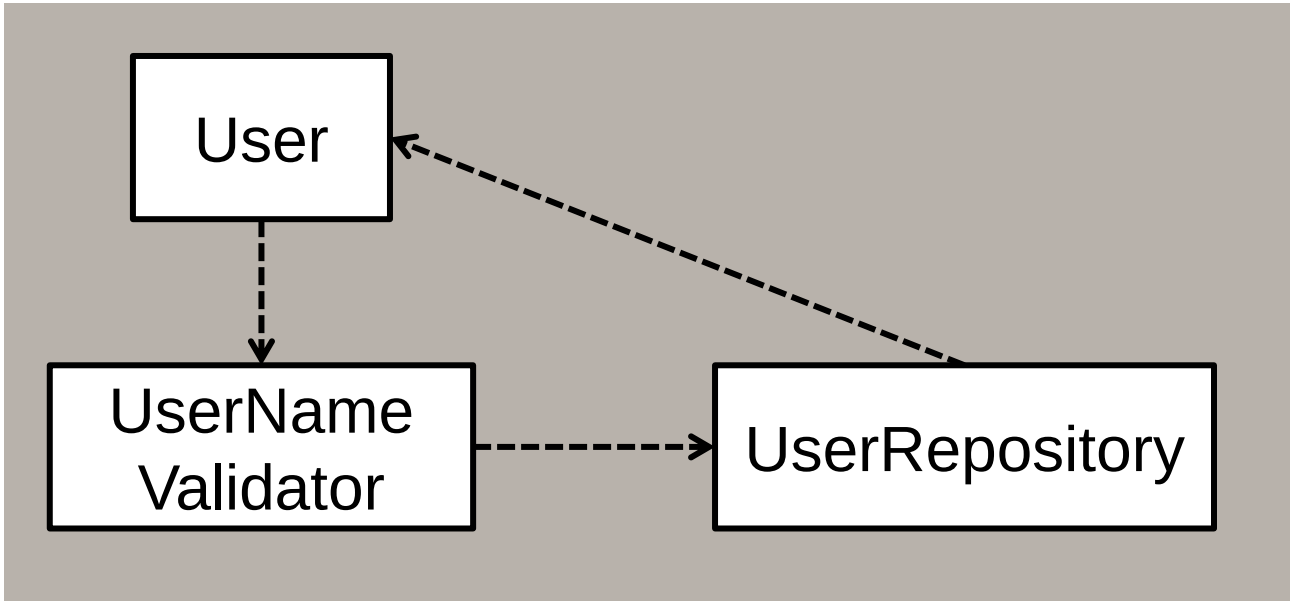
Law of Demeter

- Objekte sollen nur mit Objekten in „unmittelbarer Umgebung“ kommunizieren

```
String outputDir = user.getClient()
                        .getProperties()
                        .getReportDirectory()
                        .getAbsolutePath();
```

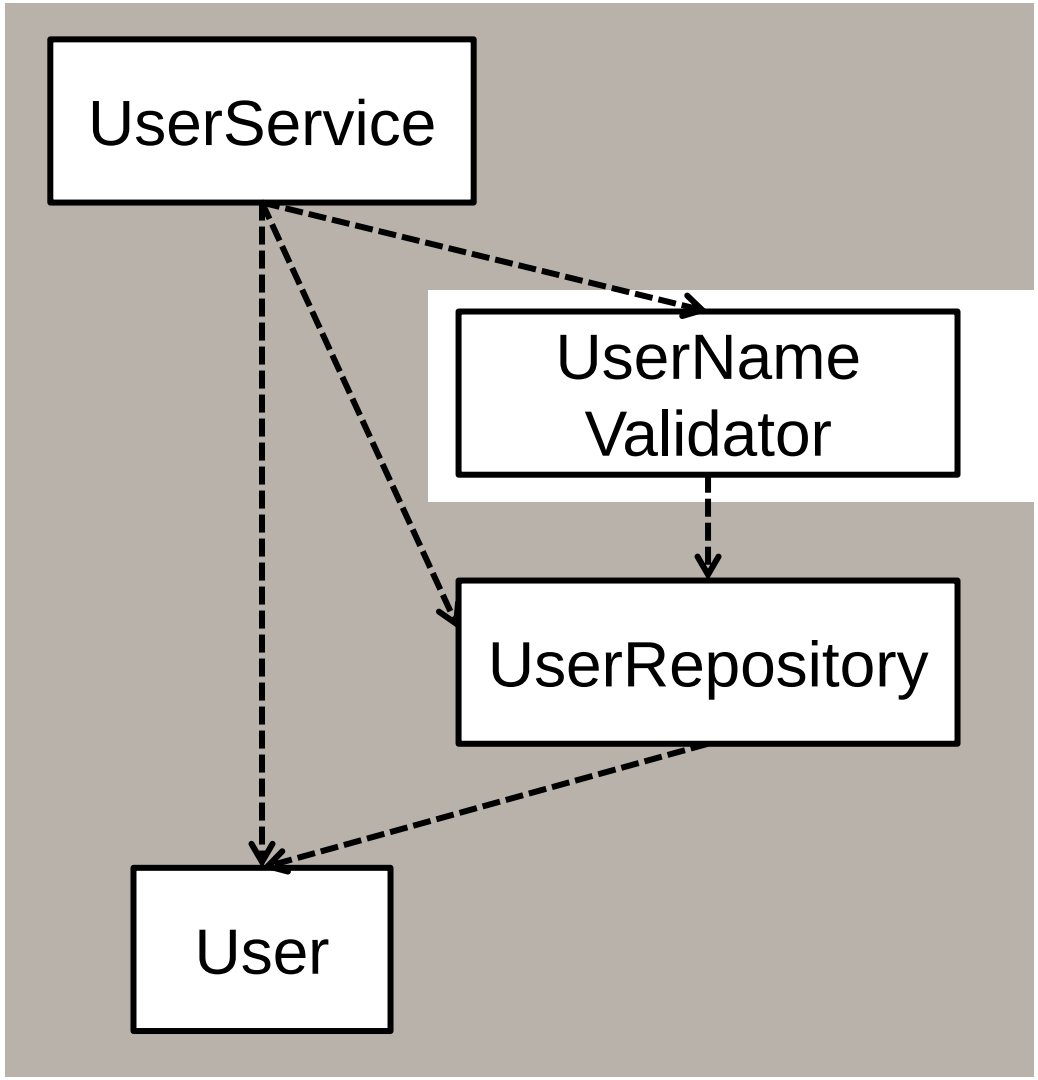
Unnötig starke Kopplung

Zyklische Abhängigkeiten auf Klassenebene



Unnötig starke Kopplung

Zyklische Abhängigkeiten auf Package-Ebene



Zyklische Abhängigkeiten

- Legacy-Projekt restrukturieren ist sehr aufwändig
  - Lohnt sich diese Investition wirtschaftlich?
- Von Anfang an einsetzen
  - Beispiel: Spring Framework hat schon früh Werkzeuge wie jdepend eingesetzt
  - In Continuous Integration integrieren

Kohäsionsmetriken

Wie kann man Kohäsion messen?

Lack of Cohesion of Methods (LCOM2)

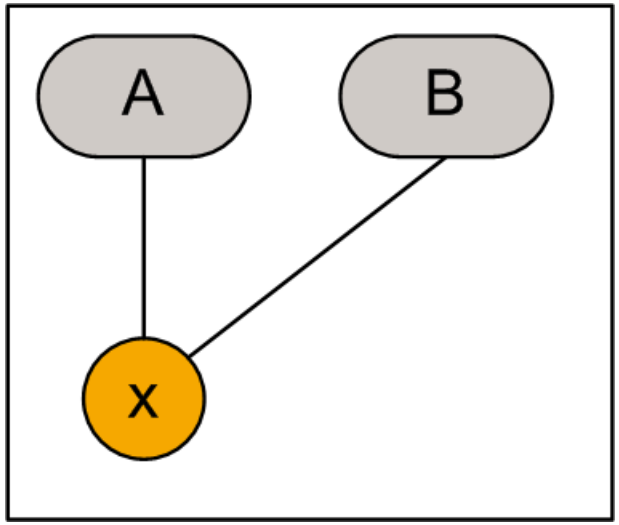
von Henderson-Sellers

m	#Methoden
a	#Attribute
mA	#Methoden, die ein Attribut nutzen
sum(mA)	Summer über mA aller Attribute

LCOM2 = 1 - sum(mA)/(m*a)

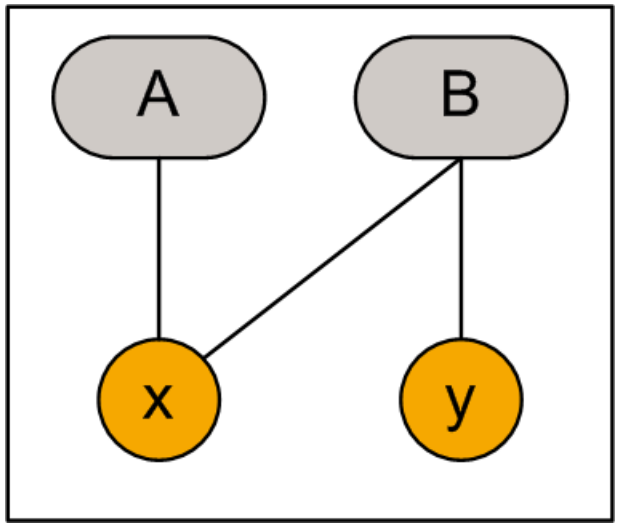
LCOM2-Beispiel mit hoher Kohäsion

LCOM2 = 1 - sum(mA)/(m*a) = 1 - 2/(2*1) = 0



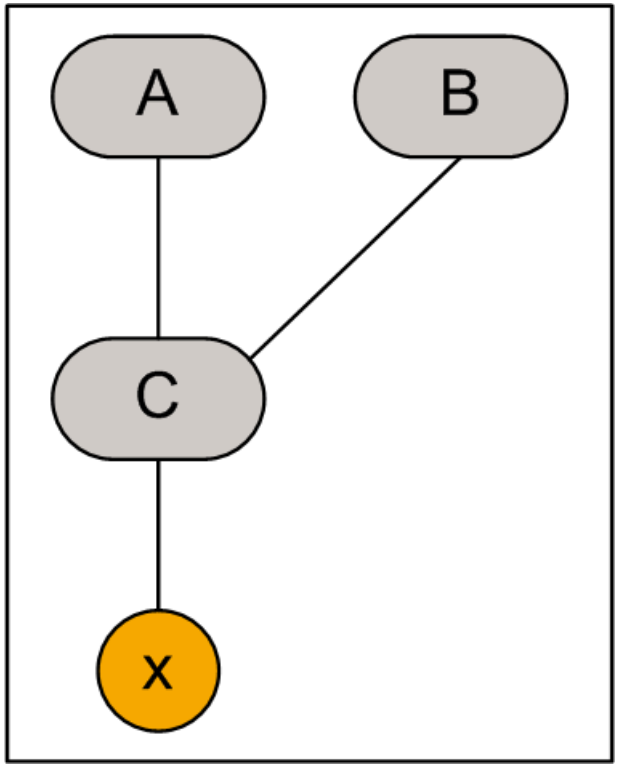
LCOM2-Beispiel mit mittlerer Kohäsion

LCOM2 = 1 - sum(mA)/(m*a) = 1 - 3/(2*2) = 0,25



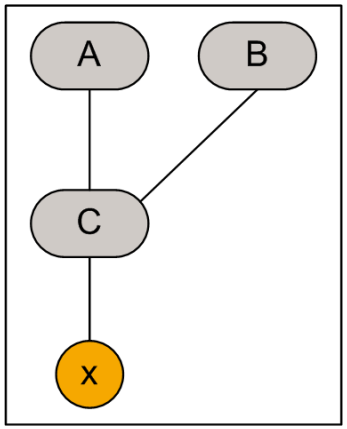
LCOM2-Beispiel mit geringer Kohäsion

LCOM2 = 1 - sum(mA)/(m*a) = 1 - 1/(3*1) = 0,67



```
class DateFormatter {
  private static final int DATE_FORMAT = DateFormat.MEDIUM;

  String format(Date date) {
    DateFormat df = DateFormat.getDateInstance(DATE_FORMAT);
    return df.format(date);
  }
  String format(long millis) {
    Date date = new Date(millis);
    return format(date);
  }
  String format(Calendar cal) {
    Date date = cal.getTime();
    return format(date);
  }
} A
```

X C A B



Wie kann man Kohäsion alternativ messen?

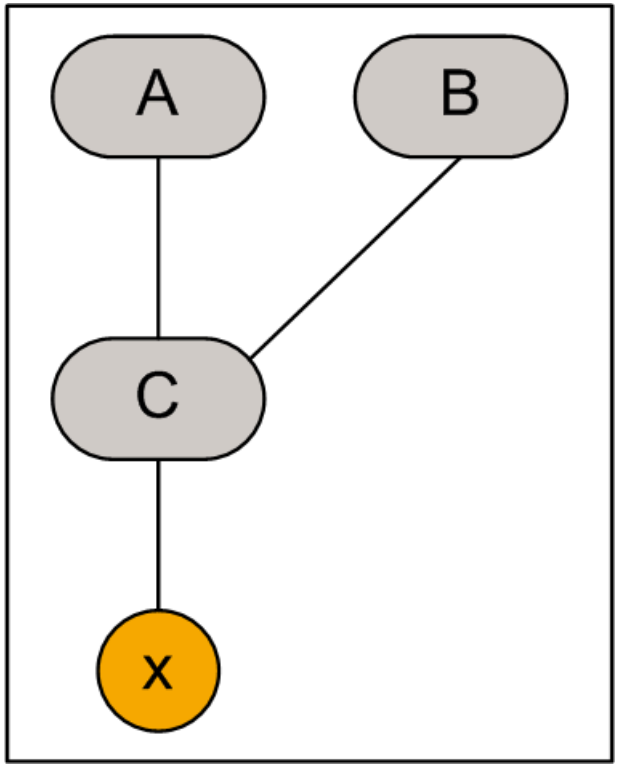
Lack of Cohesion of Methods (LCOM4)

von Hitz und Montazeri

LCOM4 = Anzahl der Zusammenhangskomponenten

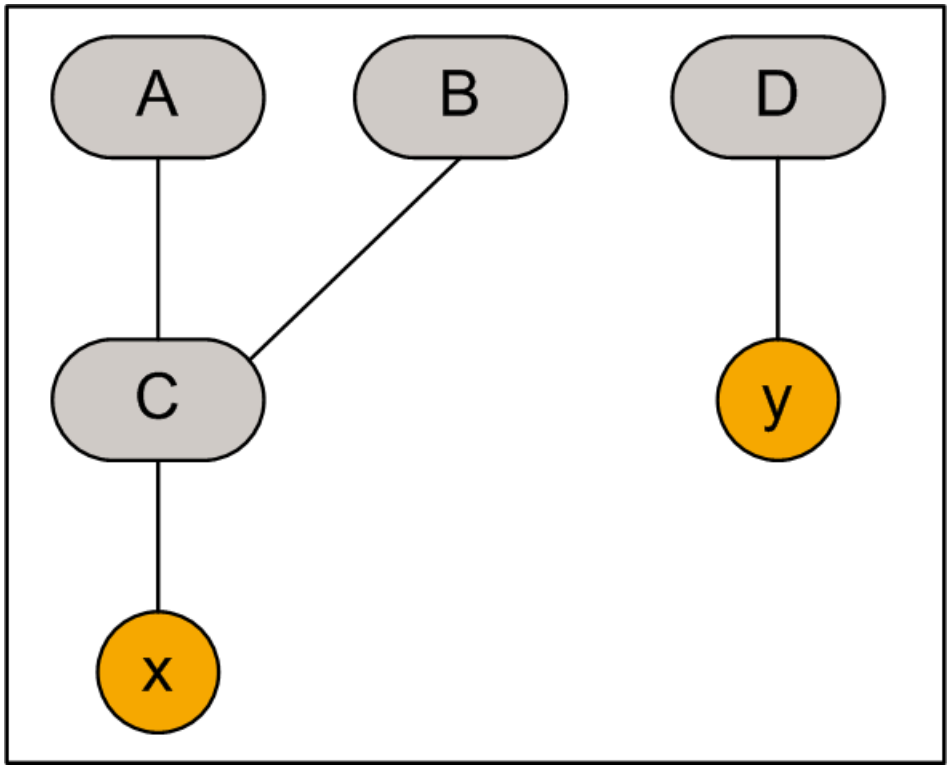
LCOM4-Beispiel mit hoher Kohäsion

LCOM4 = 1



LCOM4-Beispiel mit geringerer Kohäsion

LCOM4 = 2



Java Beispiel mit geringem LCOM4-Wert

```
class FinanceReport {

  void connectToDatabase() {}

  void generateFinanceReport() {}

  void saveToFile() {}

  void print() {}

}
```

Java Beispiel mit geringem LCOM4-Wert

```
class Math {

  public double sin(double a) {}

  public double cos(double a) {}

  public double tan(double a) {}

}
```

Hohe funktionale Kohäsion ist praktisch nicht erreichbar

- Übergeordnetes Ziel nicht aus den Augen verlieren:
  - Single Responsibility Prinzip

Zwischenbilanz

- Kleine Klassen mit einer Aufgabe einsetzen
- Unnötig starke Kopplung vermeiden

  (z.B. zyklische Abhängigkeiten)

An Idea Whose Time Has Come and Gone

"You can't control what you can't measure"

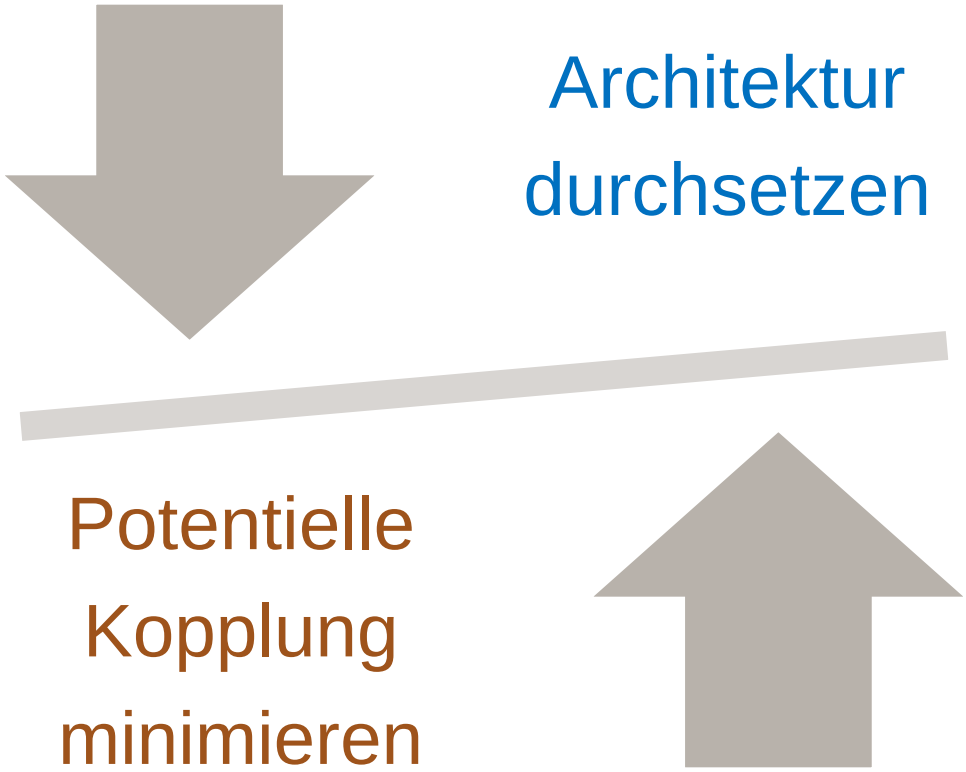
Tom DeMarco

"Was its advice correct at the time, is it still relevant, and do I still believe that metrics are a must for any successful software development effort? My answers are no, no, and no."

Tom DeMarco, 2009

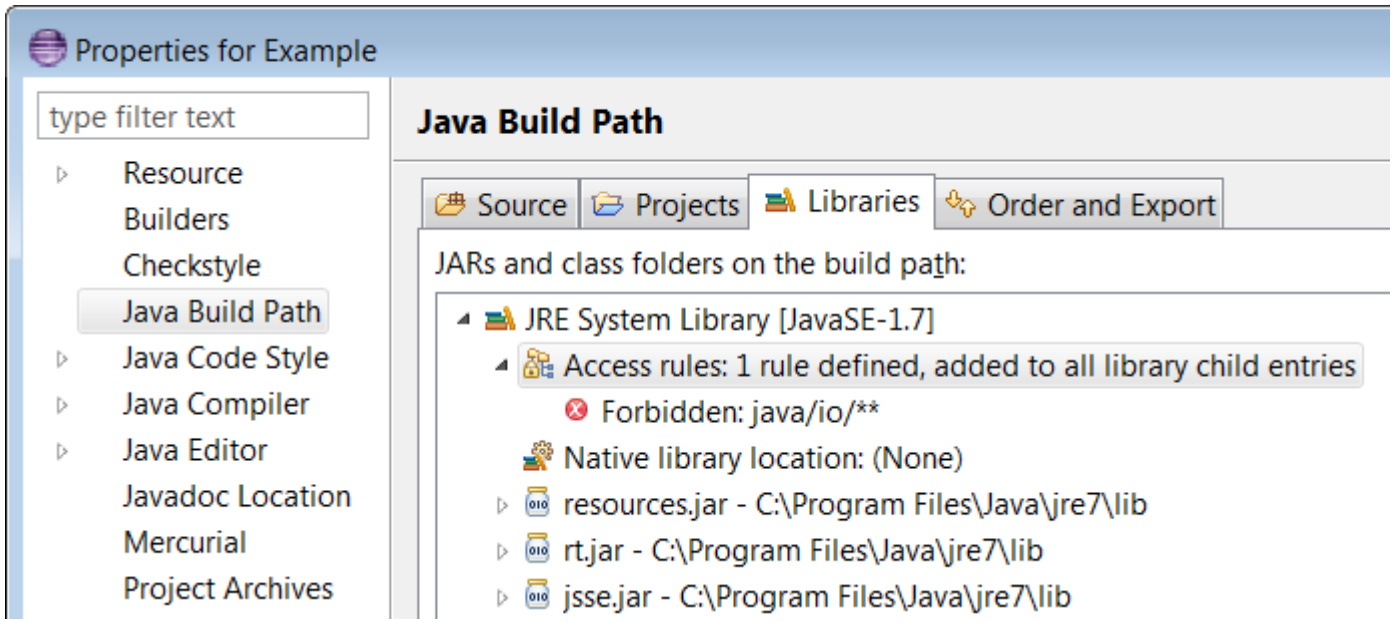
- Softwaremetriken und Coding Rules helfen starke Kopplung zu vermeiden

- Leitplanken für Größe und Komplexität von Klassen und Packages hilfreicher als Kohäsionsmetriken?


Architektur
durchsetzen
Potentielle
Kopplung
minimieren

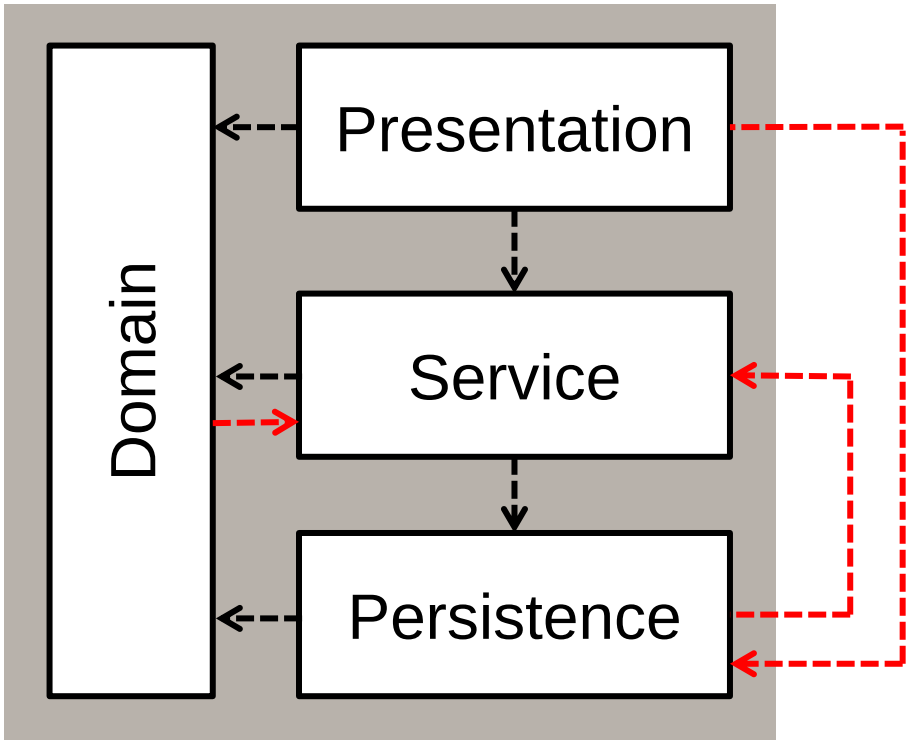
Eclipse Access Rules

- **Non-accessible Rules:** Fehler wird angezeigt, falls diese Typen referenziert werden
- **Discouraged Rules:** Warnung wird angezeigt
- **Accessible Rules:** Definiert welche Typen referenziert werden dürfen



Kontinuierliche Architekturvalidierung

Beispielarchitektur



erlaubte Abhängigkeiten

nicht erlaubte Abhängigkeiten

adesso

Architekturregeln in SonarQube definieren

- Coding Rule Architectural Constraint

Architectural constraint

A source code comply to an architectural model when it fully adheres to a set of architectural constraints. A constraint allows to deny references between classes by pattern.

You can for instance use this rule to :

- forbid access to **.web.** from **.dao.** classes
- forbid access to java.util.Vector, java.util.Hashtable and java.util.Enumeration from any classes
- forbid access to java.sql.** from **.ui.** and **.web.** classes

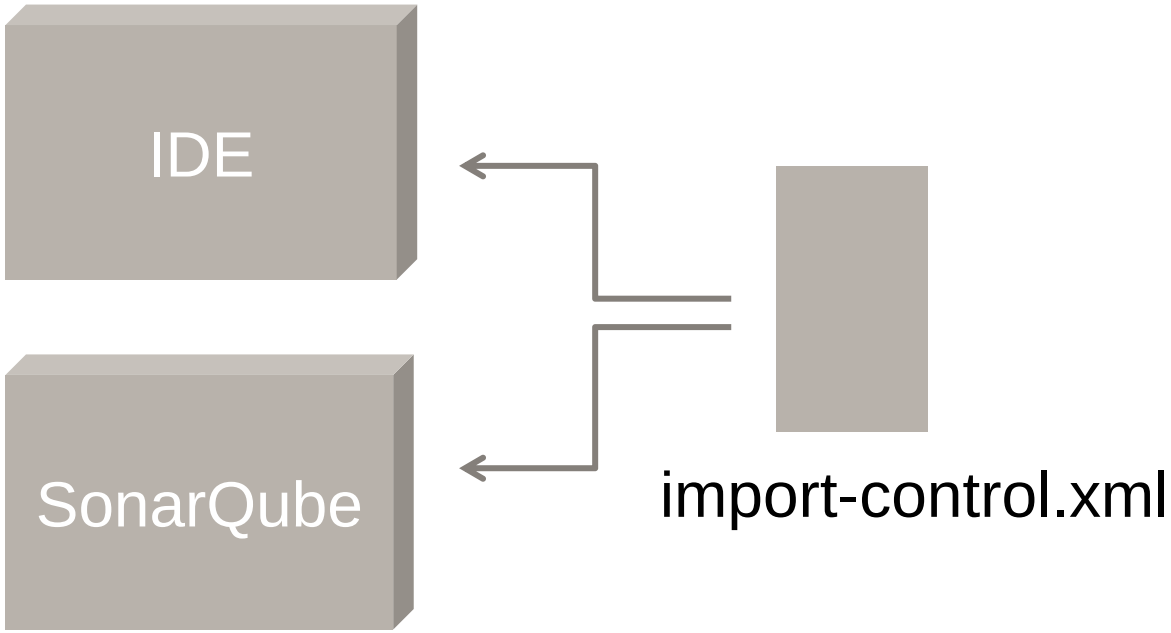
Extend Description

fromClasses	com.example.presentation.**	Update
	Optional. If this property is not defined, all classes should adhere to this constraint. Ex : **.web.**	
toClasses	com.example.persistence.**	Update
	Mandatory. Ex : java.util.Vector, java.util.Hashtable, java.util.Enumeration	

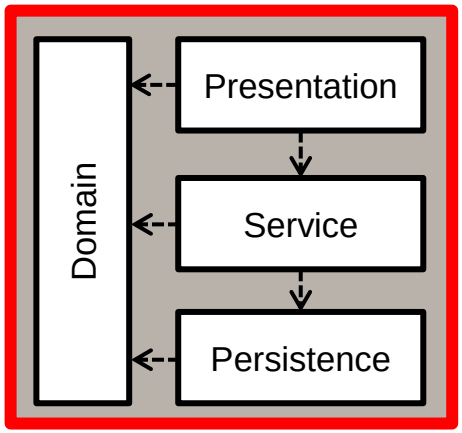
Add Note
Copy rule

Import-Kontrolle mit Checkstyle

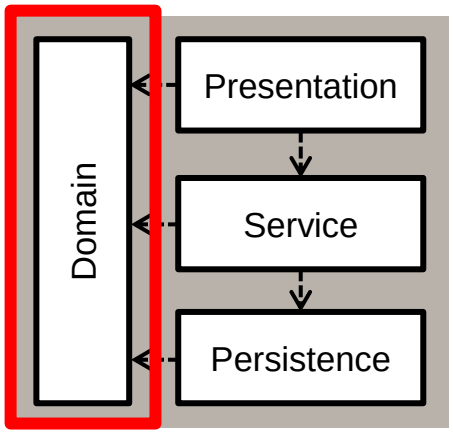
- Imports können durch Sonar oder direkt in der IDE überprüft werden



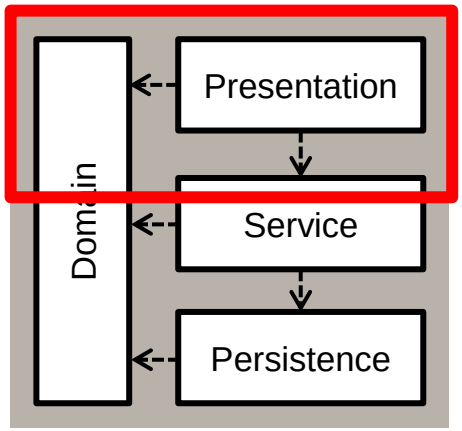
```
<import-control pkg="com.example">
 <allow pkg="java" />
 <allow pkg="javax" />
 <allow pkg="com.google.common" />
 <disallow pkg="javax.persistence" />
 <disallow pkg="javax.servlet" />

 <subpackage name="domain">
  <allow pkg="com.example.domain" />
 </subpackage>

 <subpackage name="presentation">
  <allow pkg="com.example.presentation" />
  <allow pkg="com.example.domain" />
  <allow pkg="com.example.services.api" />
  <allow pkg="javax.servlet" />
 </subpackage>
```



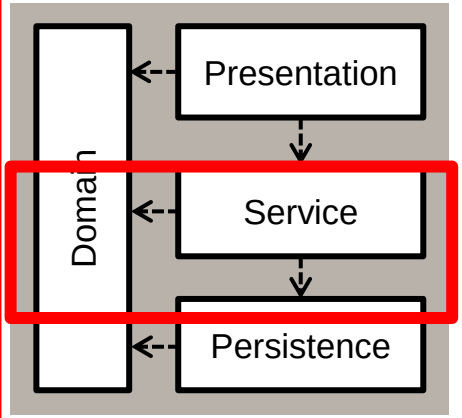
```
<import-control pkg="com.example">
 <allow pkg="java" />
 <allow pkg="javax" />
 <allow pkg="com.google.common" />
 <disallow pkg="javax.persistence" />
 <disallow pkg="javax.servlet" />

 <subpackage name="domain">
  <allow pkg="com.example.domain" />
 </subpackage>

 <subpackage name="presentation">
  <allow pkg="com.example.presentation" />
  <allow pkg="com.example.domain" />
  <allow pkg="com.example.services.api" />
  <allow pkg="javax.servlet" />
 </subpackage>
```



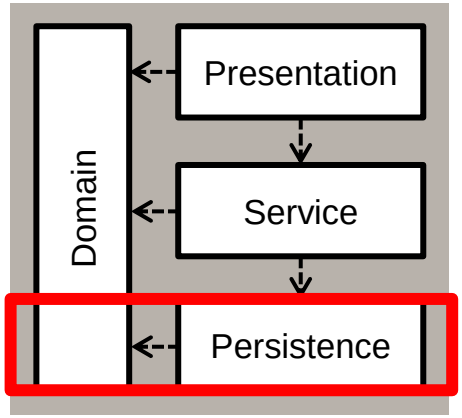
```
<import-control pkg="com.example">
 <allow pkg="java" />
 <allow pkg="javax" />
 <allow pkg="com.google.common" />
 <disallow pkg="javax.persistence" />
 <disallow pkg="javax.servlet" />

 <subpackage name="domain">
  <allow pkg="com.example.domain" />
 </subpackage>

 <subpackage name="presentation">
  <allow pkg="com.example.presentation" />
  <allow pkg="com.example.domain" />
  <allow pkg="com.example.services.api" />
  <allow pkg="javax.servlet" />
 </subpackage>
```



```
<subpackage name="services">
 <allow pkg="com.example.domain" />
 <subpackage name="impl">
  <allow pkg="com.example.services.impl" />
  <allow pkg="com.example.services.api" />
  <allow pkg="com.example.persistence" />
 </subpackage>
</subpackage>
 <subpackage name="persistence">
  <allow pkg="com.example.persistence" />
  <allow pkg="com.example.domain" />
  <allow pkg="javax.persistence" />
 </subpackage>
</import-control>
```

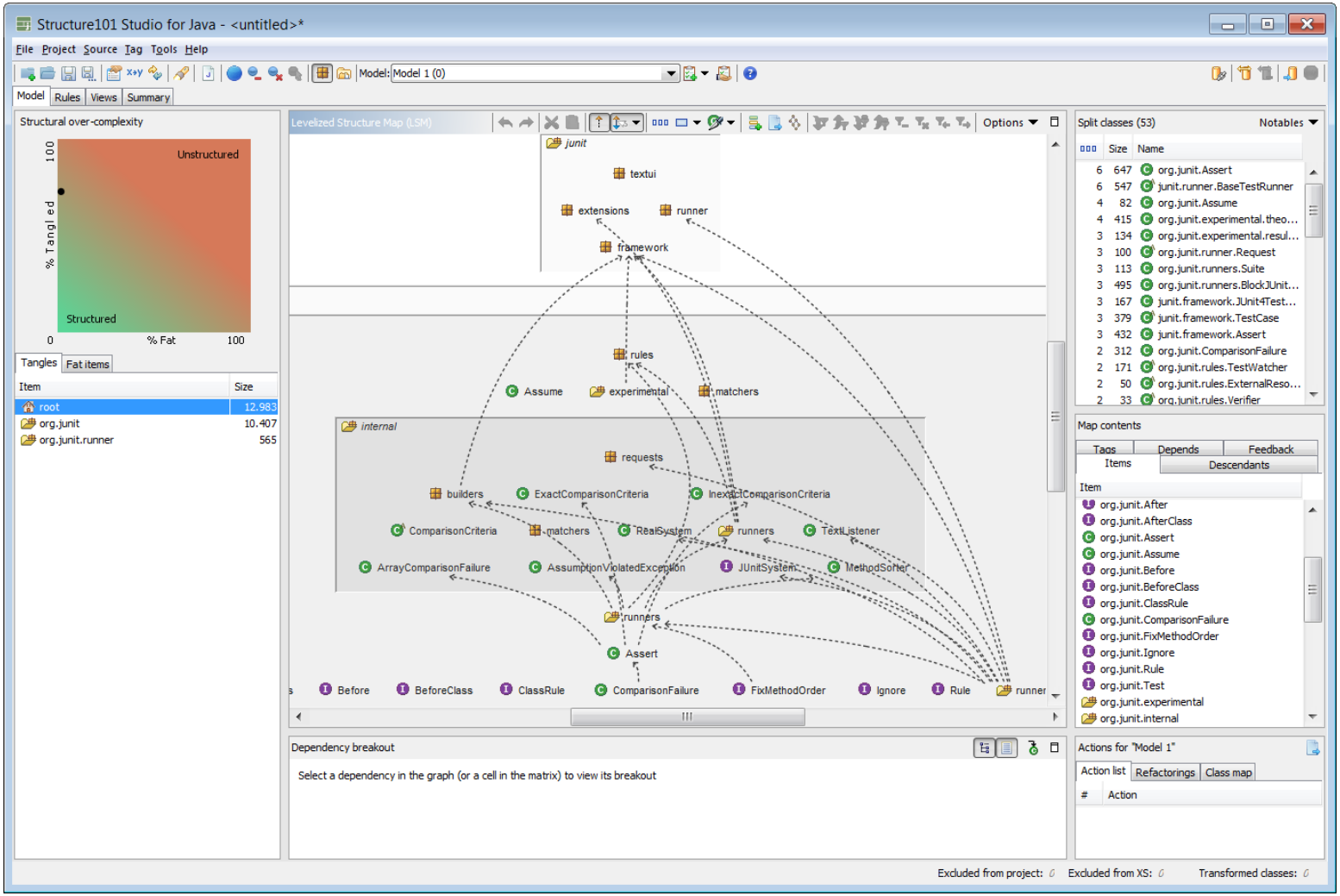


```
<subpackage name="services">
 <allow pkg="com.example.domain" />
 <subpackage name="impl">
  <allow pkg="com.example.services.impl" />
  <allow pkg="com.example.services.api" />
  <allow pkg="com.example.persistence" />
 </subpackage>
</subpackage>

<subpackage name="persistence">
 <allow pkg="com.example.persistence" />
 <allow pkg="com.example.domain" />
 <allow pkg="javax.persistence" />
</subpackage>
</import-control>
```



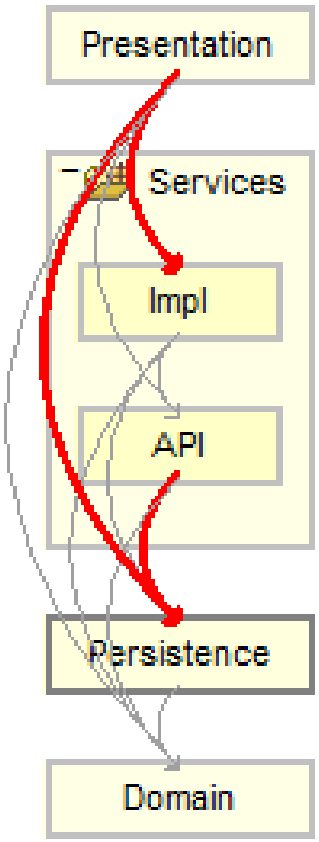
Structure101: Code verstehen und visualisieren

- Lücke zwischen Codebasis und Architektur schließen



adesso

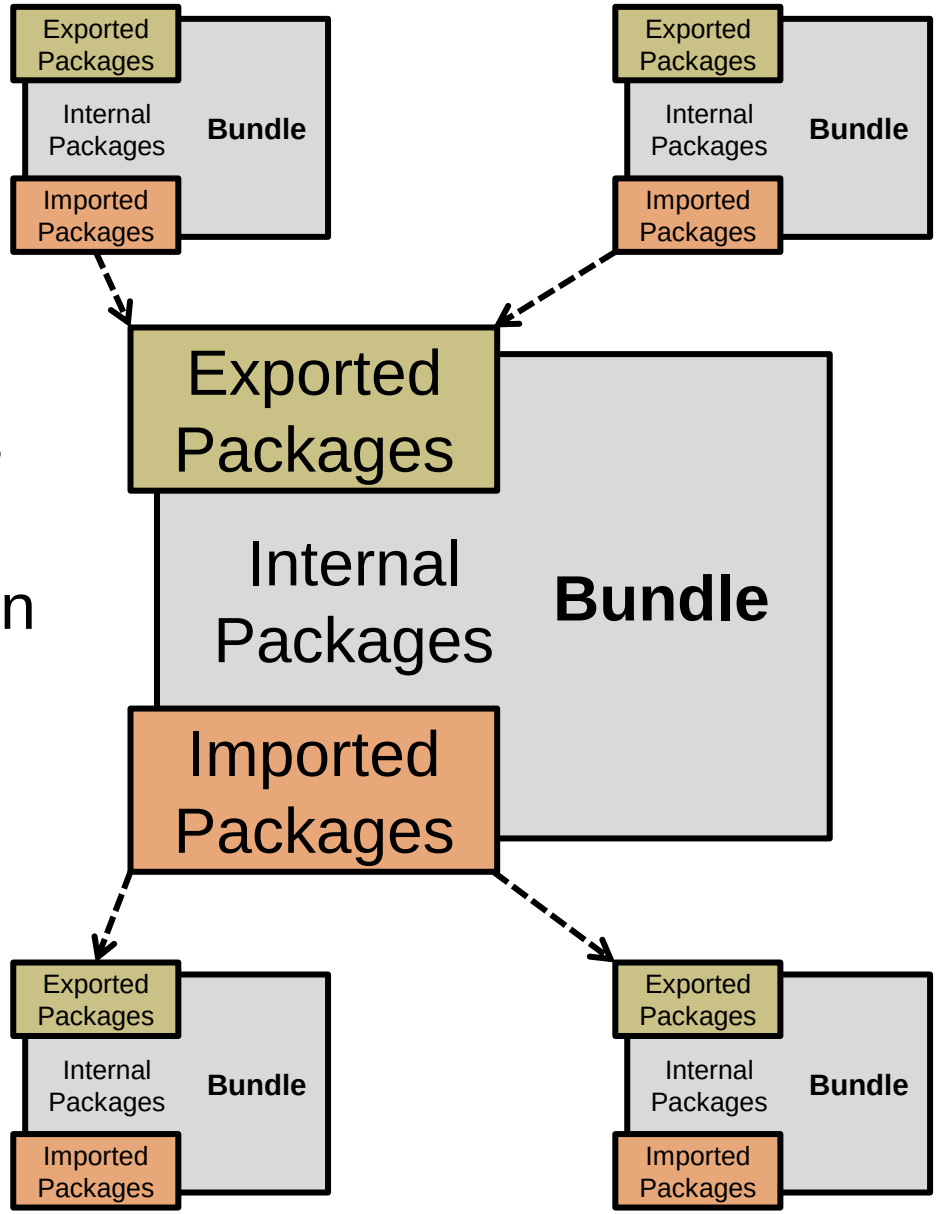
Structure101: Zielarchitektur definieren



- Abweichungen werden automatisch von der Zielarchitektur erkannt
- Abhängigkeiten sind implizit von oben nach unten erlaubt
- Mit „Overrides“ können Ausnahmen definiert werden
- Mit „Pattern“ werden visuelle Komponenten mit Klassen oder Packages verknüpft

OSGi Bundles

- Modularisierung mit Komponenten
- Nur exportierte Packages können von anderen Modulen importiert werden

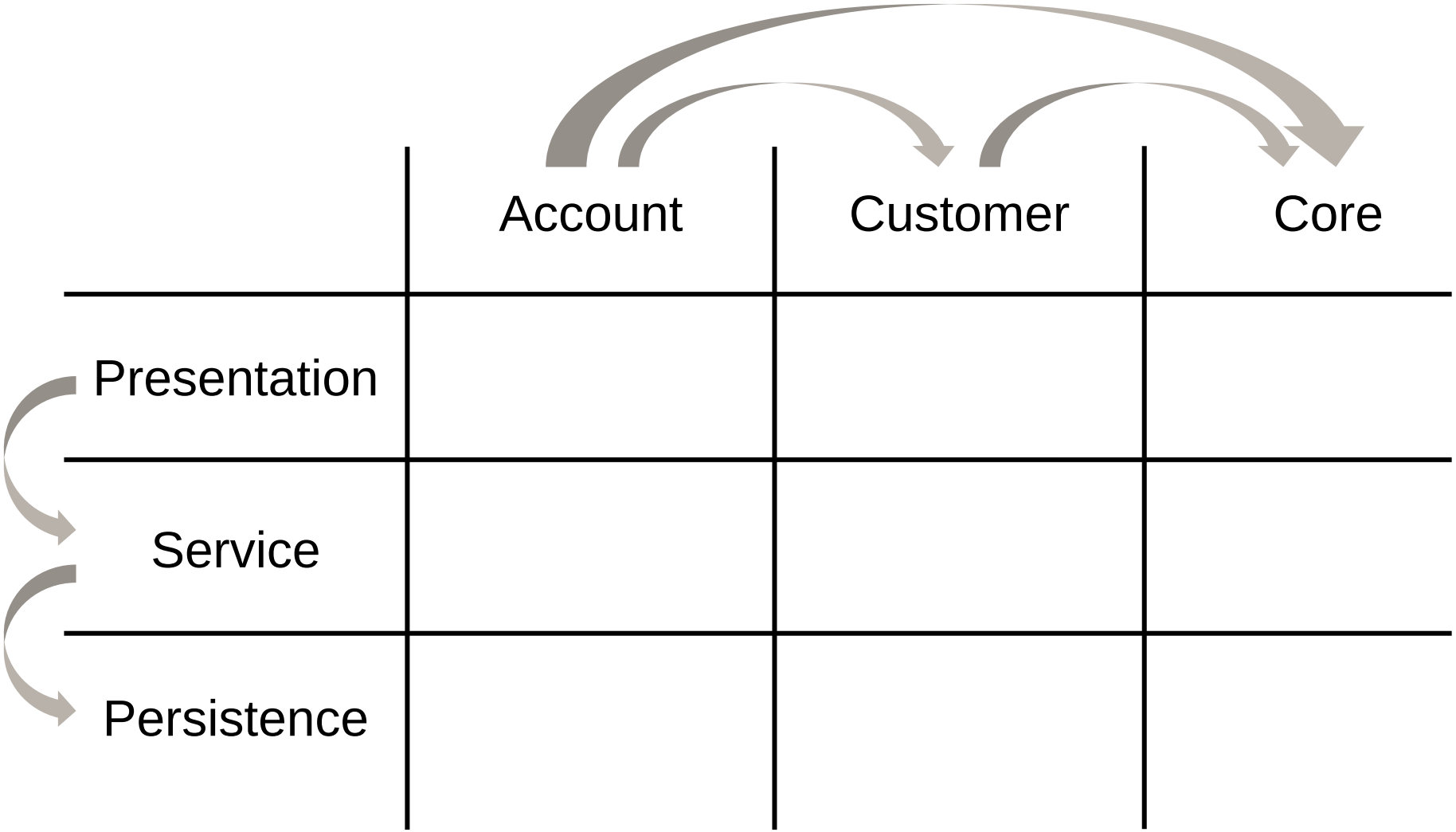


Layer vs. Slices

- Layer sind eine beliebte Form von Teile und Herrsche
  - Presentation Layer
  - Service Layer
  - Repository Layer

Nachteile:

- Technische Organisation der Codebasis
- Untere Layer sind für obere komplett sichtbar
- Hohe Komplexität bei großen Projekten



Vgl. Oliver Gierke: Whoops! Where did my architecture go?
https://speakerdeck.com/olivergierke/whoops-where-did-my-architecture-go-2?slide=35

Package-Benennung

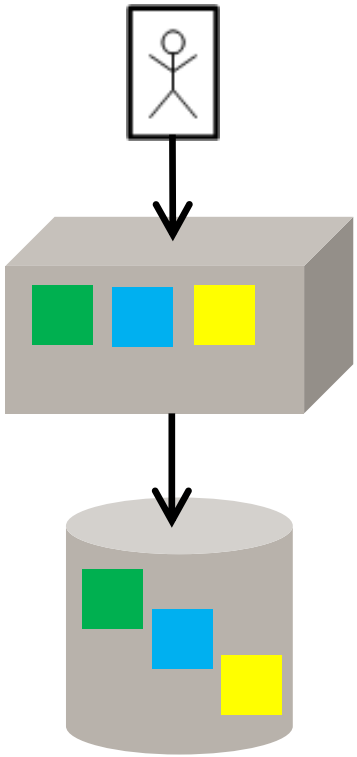
Layer-first Ansatz

com.foo.persistence.core
com.foo.service.core
com.foo.presentation.core

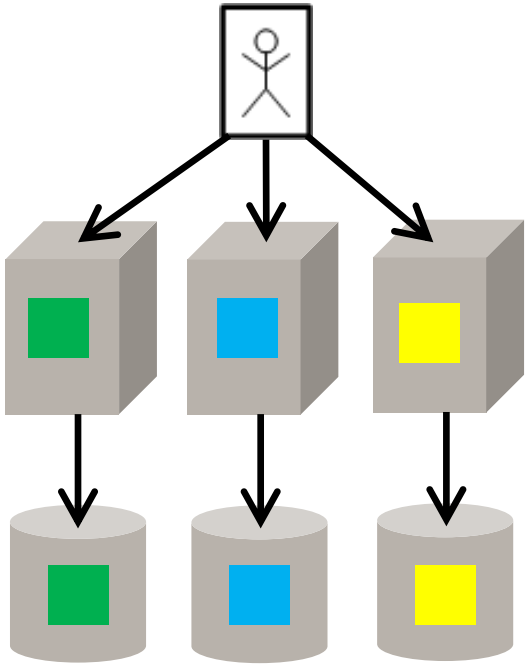
Slice-first Ansatz

com.foo.core.persistence
com.foo.core.service
com.foo.core.presentation

Microservices



Eine große Applikation
(monolithisch)



Microservices

Slices / Komponenten

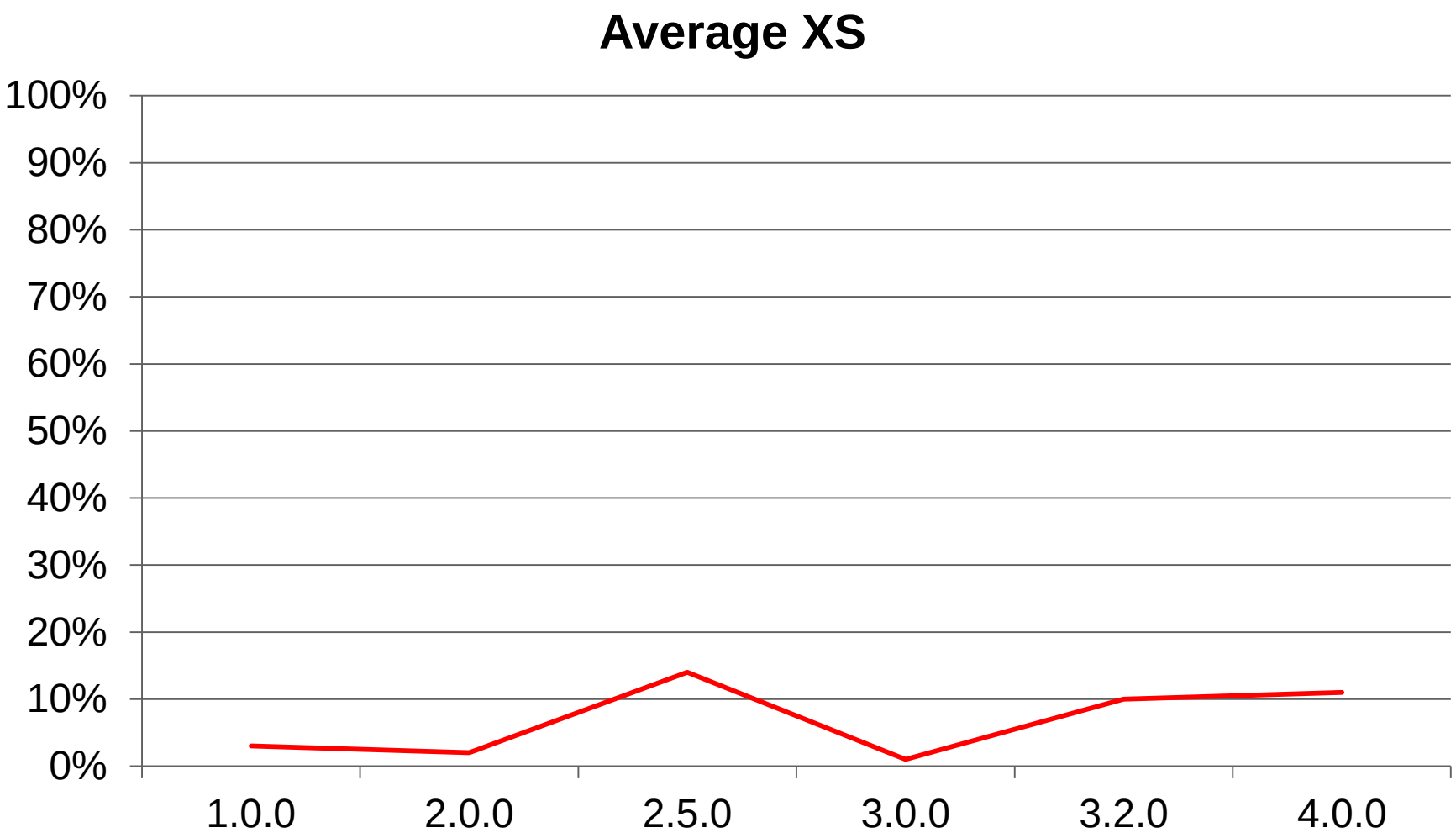
- Kommunikation innerhalb eines Prozesses
- Einfaches Deployment
- Werkzeuge zur Einhaltung der Komponentenschnittstellen

Microservices

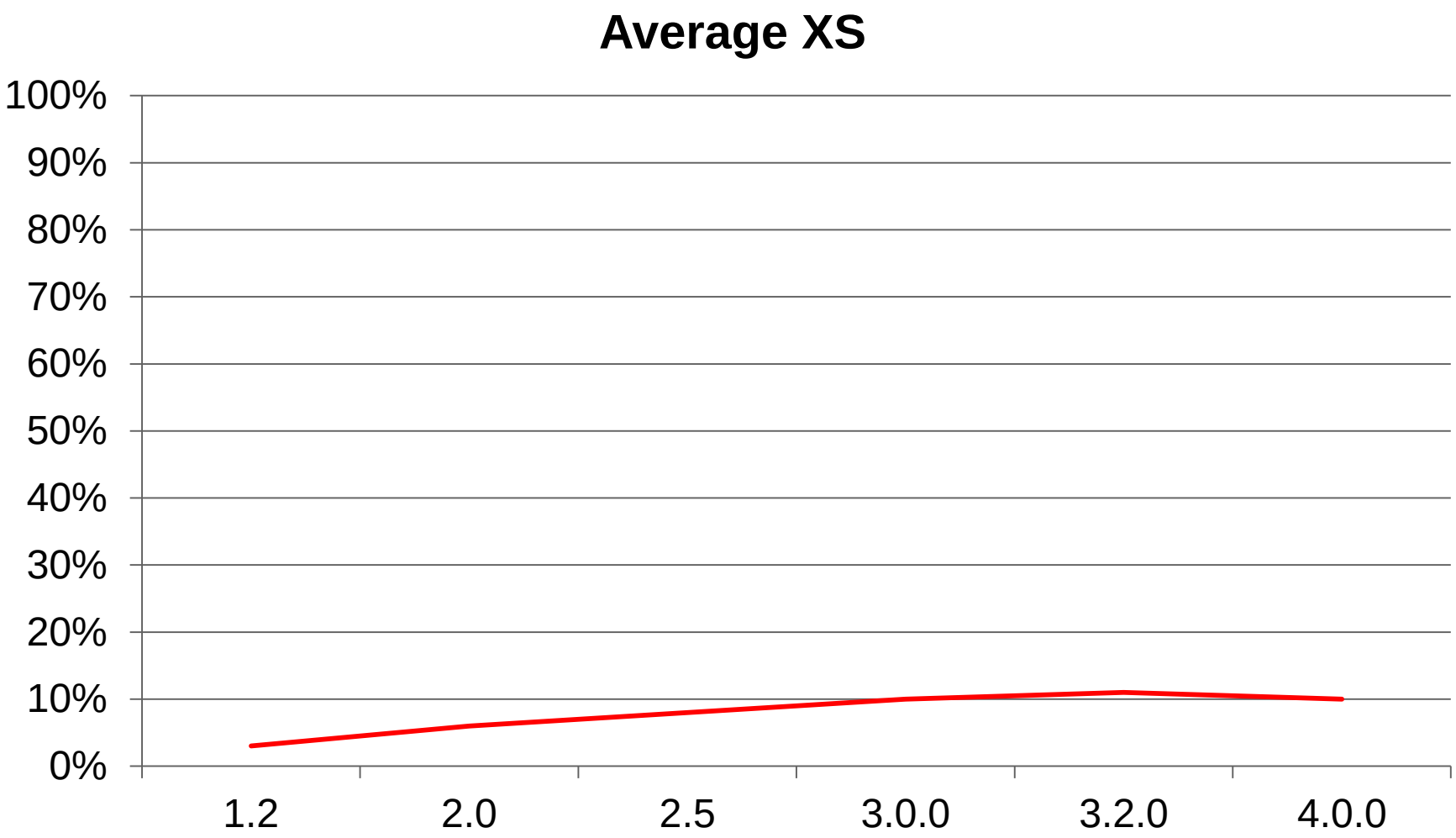
- Interprozesskommunikation, (ressourcenorientiert über HTTP)
- Eventuell sind verteilte Transaktionen notwendig
- Schnelles Deployment
- Skalierung einzelner Funktionen
- Fehlerisolierung (z.B.: Memory-Leak)

Peopleware

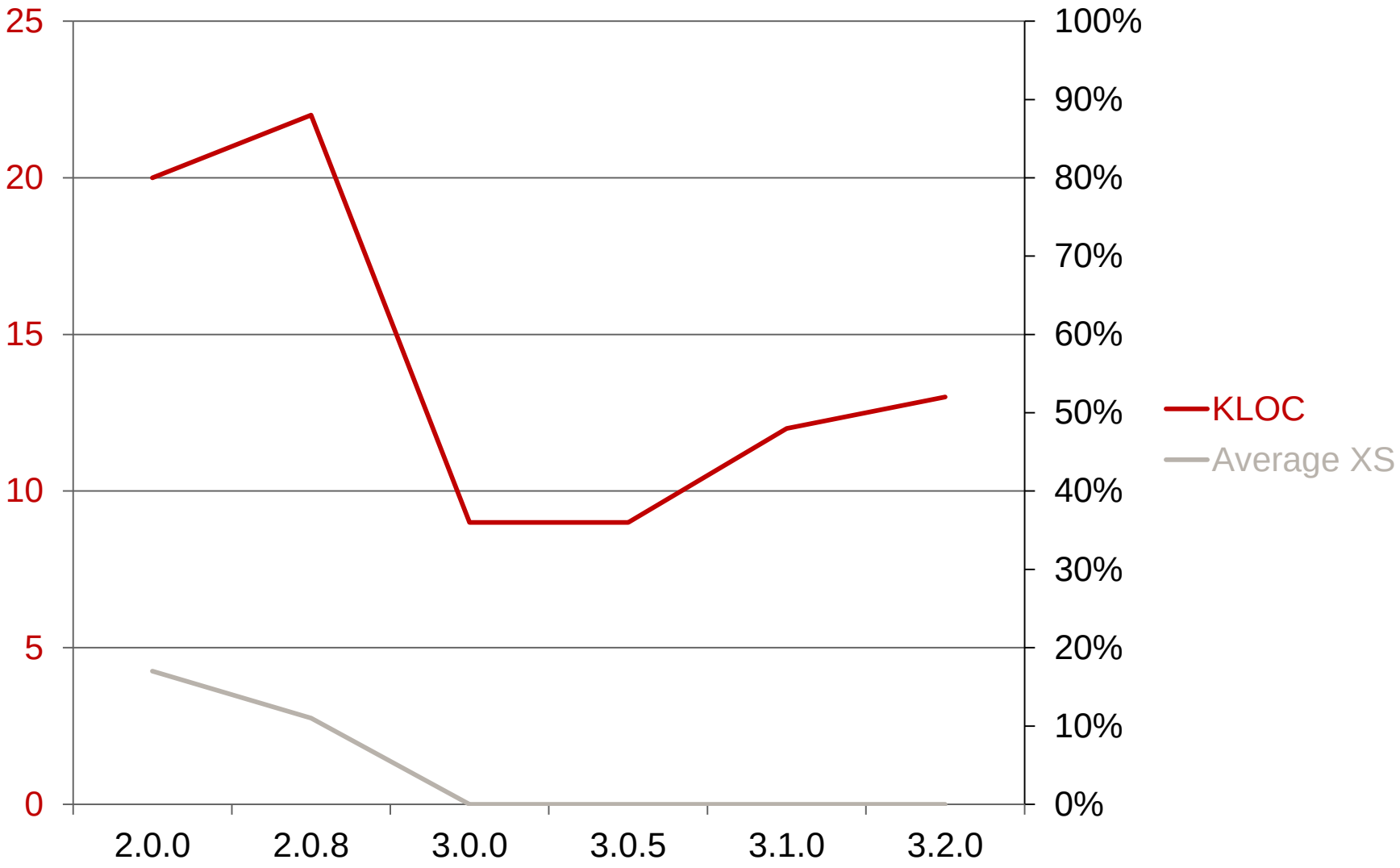
spring-core.jar



spring-beans.jar



spring-security-core.jar



adesso

Fazit

- Konzepte Kopplung und Kohäsion immer noch valide
- Softwaremetriken und Coding Rules helfen starke Kopplung zu vermeiden
- Architektur/Codebasis kontinuierlich validieren
  - Dependency Management und Soll-Architektur
  - Slices & Layer zur Quellcodestrukturierung
  - Microservices

Vielen Dank!

Kai Spichale
@kspichale
http://spichale.blogspot.de/
https://www.xing.com/profile/Kai_Spichale

adesso

IT MUSS NEU

GEDACHT WERDEN –

NEW SCHOOL OF IT

New-School-of-IT.de