
Java User Group Darmstadt

Community der Java Anwender in und um Darmstadt

# API-Design für Java-Entwickler

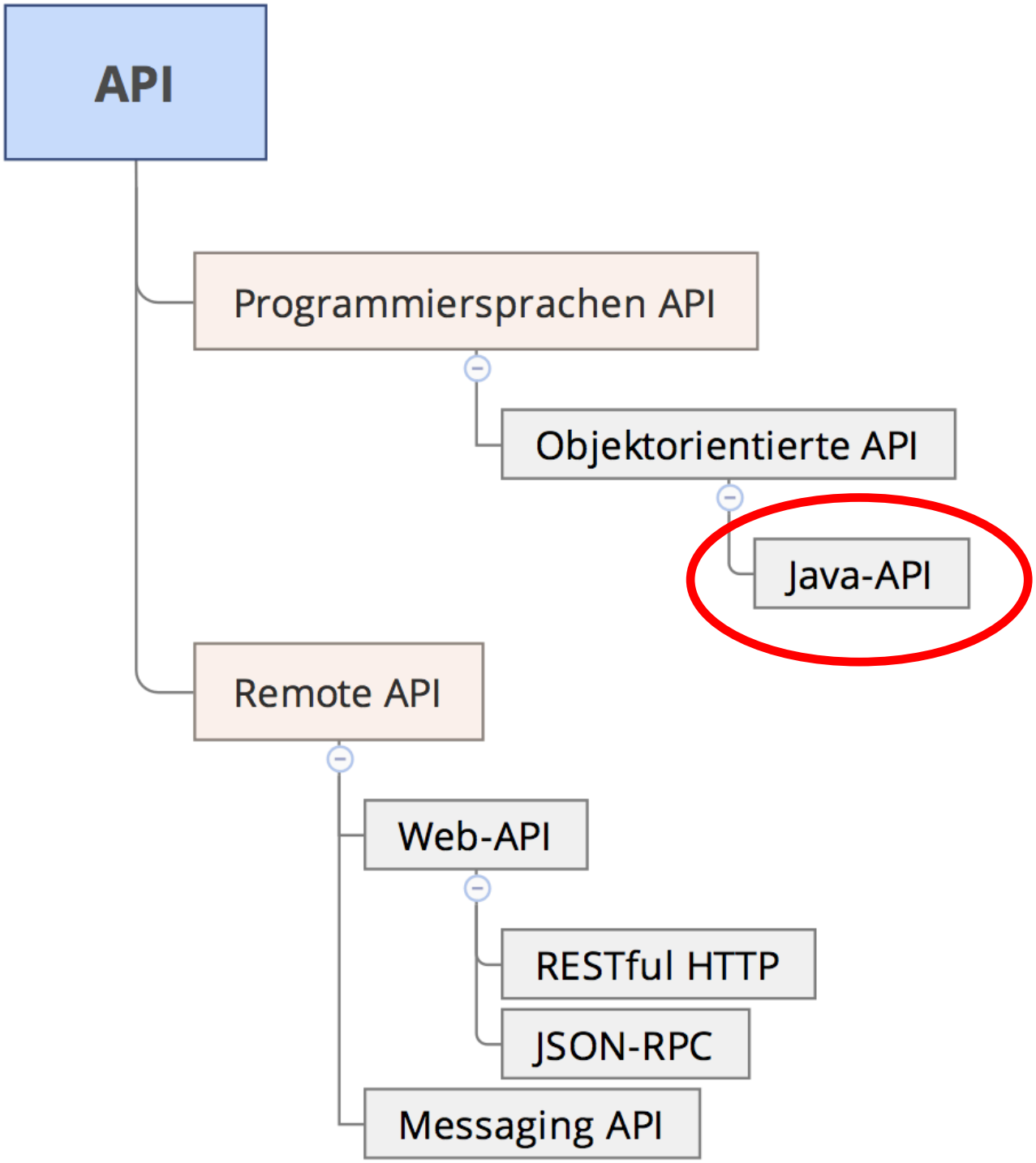
Kai Spichale

@kspichale

14.07.2016

# Demnächst im Handel ☺

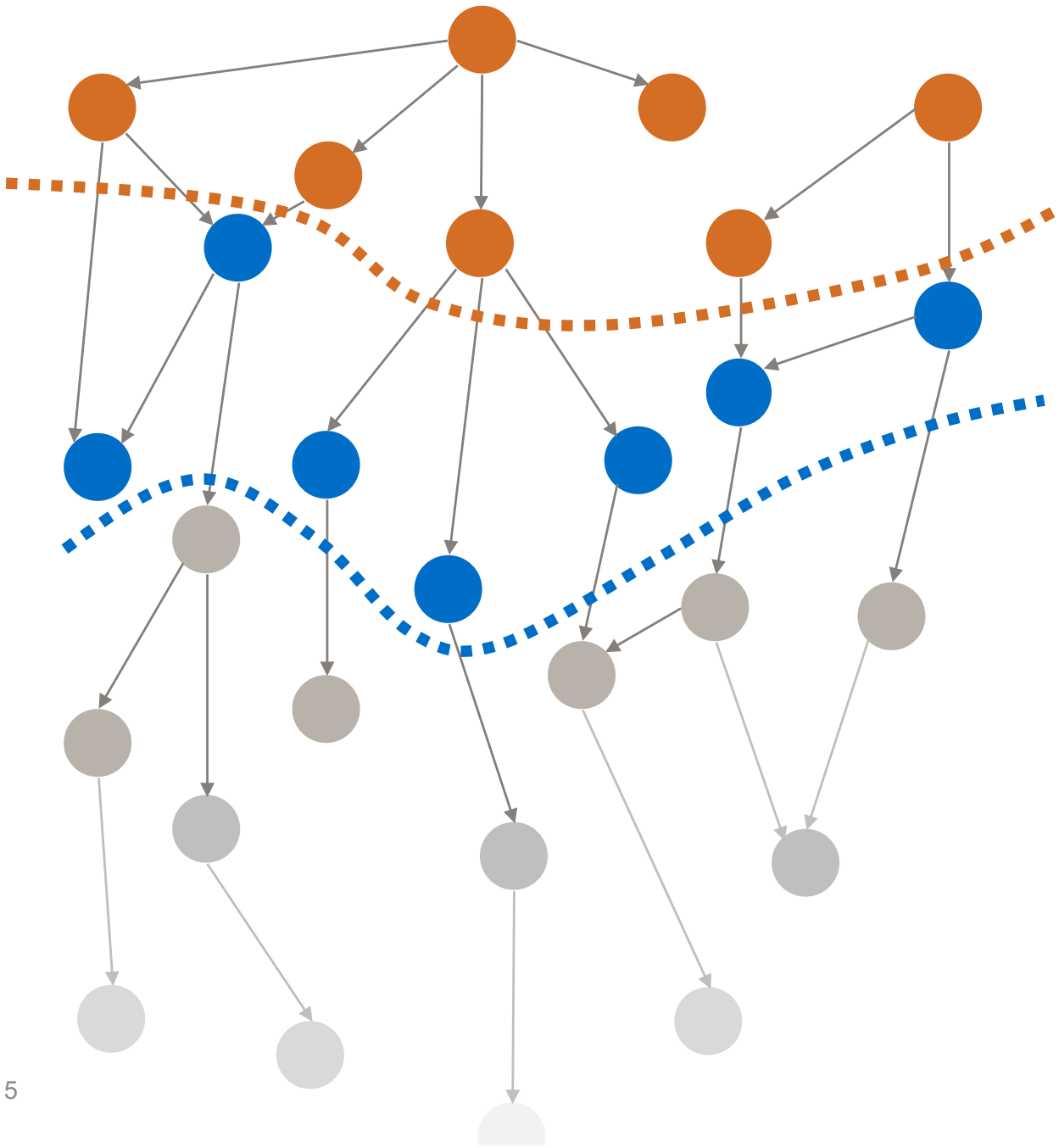
- Java-APIs
- RESTful HTTP, Web-APIs
- Messaging
- Skalierbarkeit
- API-Management


API
Programmiersprachen API
Objektorientierte API
Java-API
Remote API
Web-API
RESTful HTTP
JSON-RPC
Messaging API

Client-Code

De-Facto-API

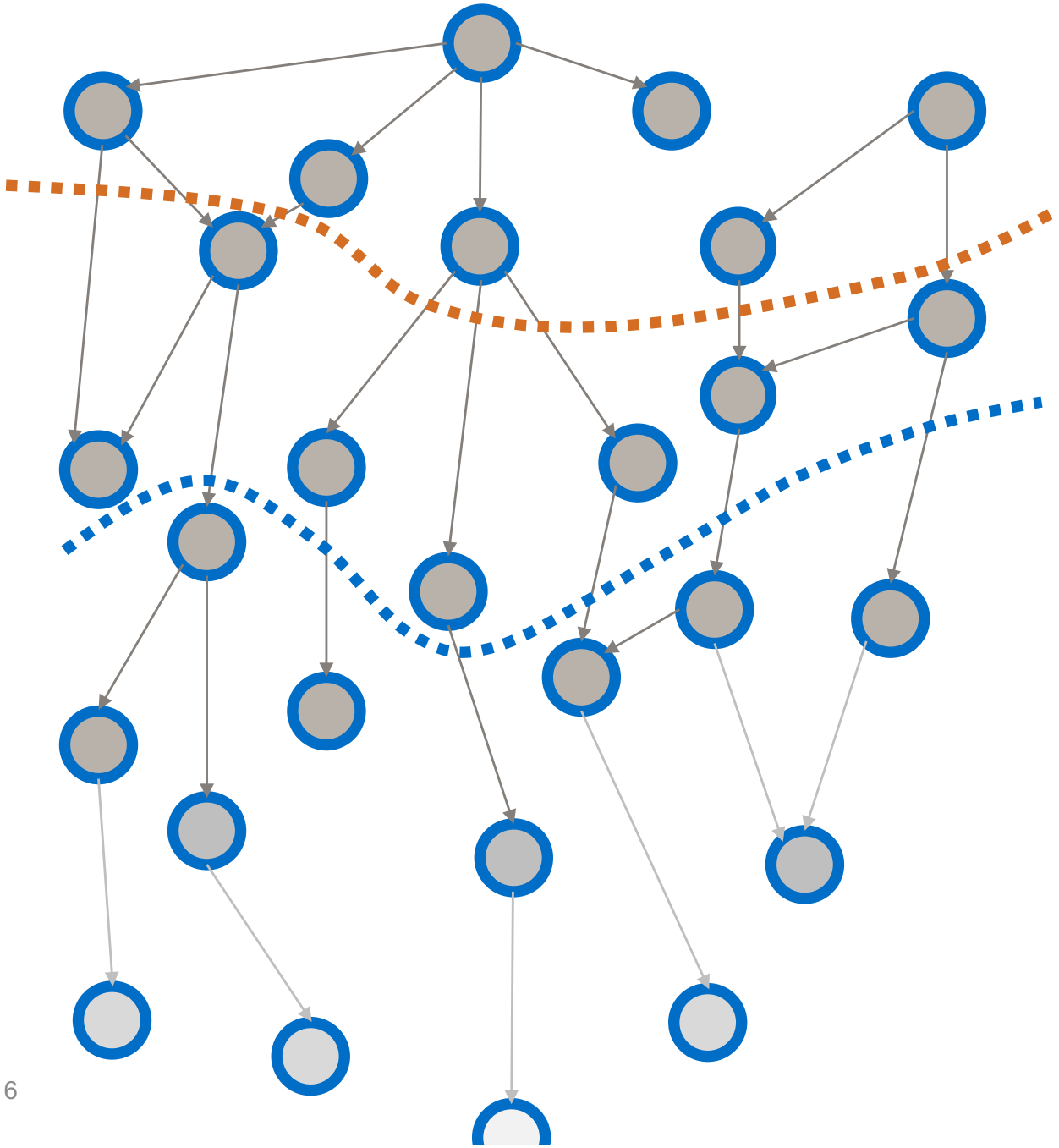
Potentielle API



Lesbarer Client-Code

Kompatible API, Geheimnisprinzip

Änderbarkeit



implizite
Objekt-API

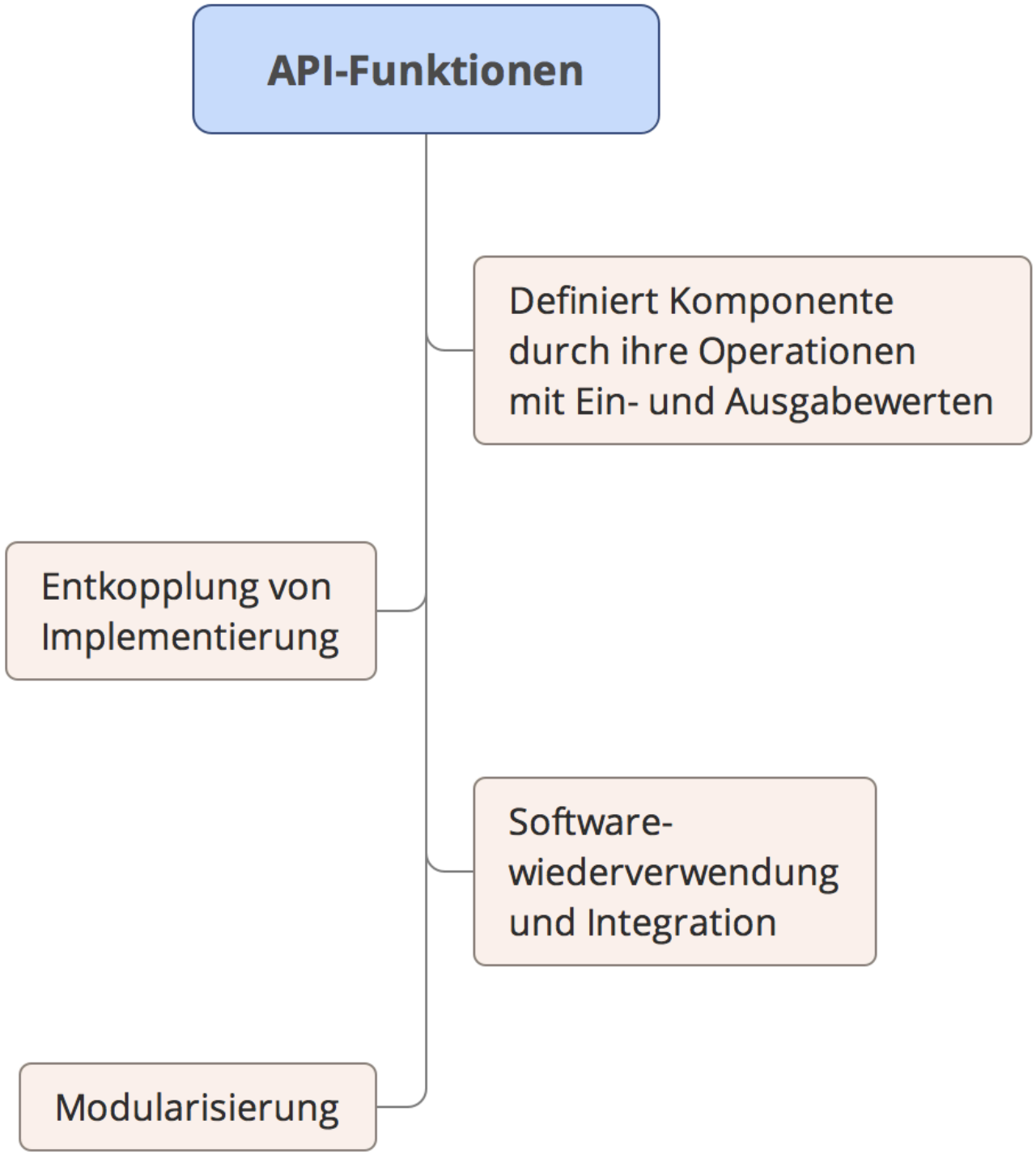
# Für wen ist API-Design relevant?

# APIs sind allgegenwärtig!

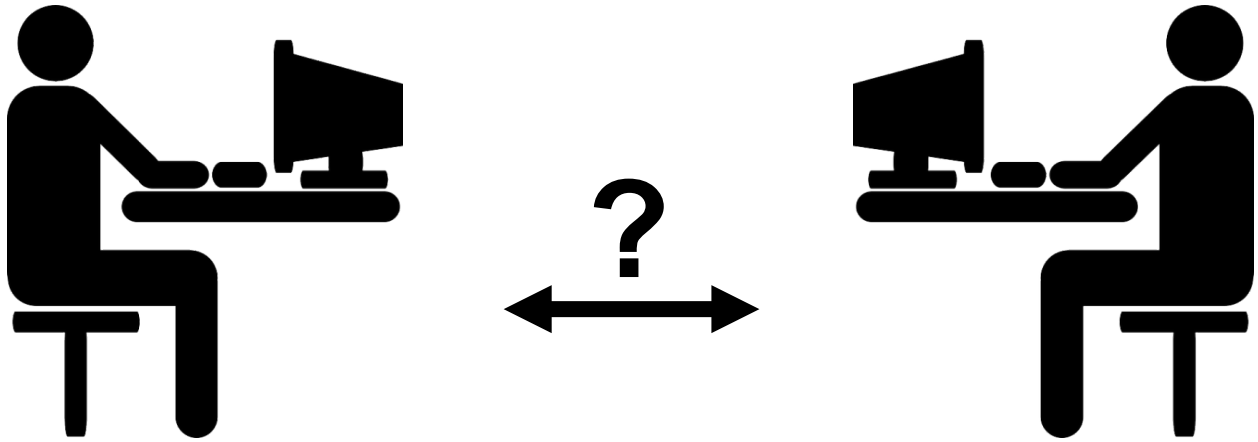
Anti-Pattern:

# API-Design aus Versehen

API-Design ist eine Frage der **Priorität**!


API-Funktionen
Definiert Komponente durch ihre Operationen mit Ein- und Ausgabewerten
Entkopplung von Implementierung
Software-wiederverwendung und Integration
Modularisierung

# Kommunikationsproblem bei Softwarewiederverwendung

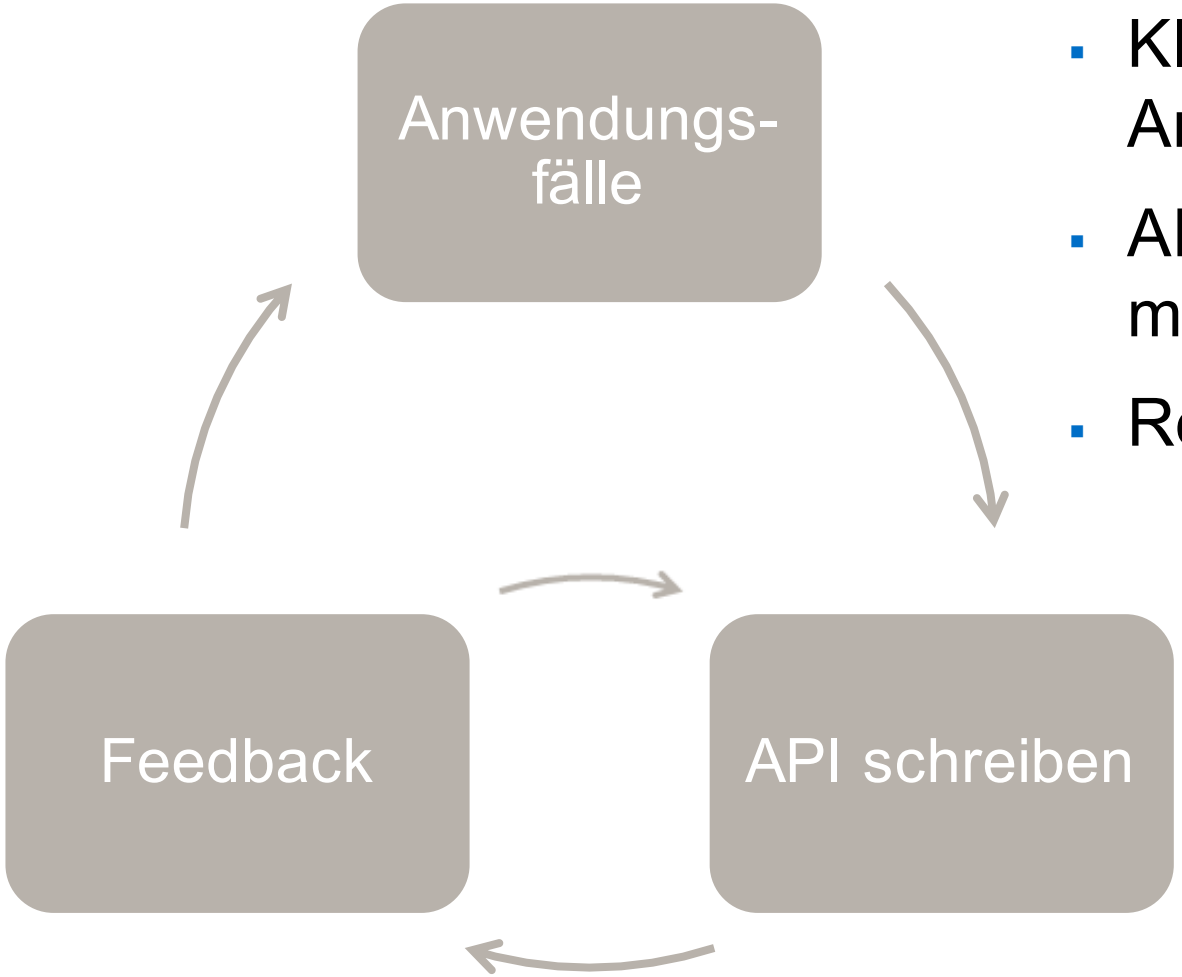



## Lösung: Kommunikation durch API

- APIs werden für Menschen geschrieben
- Perspektive ist beim API-Design entscheidend

# Entwurf mit Benutzerperspektive

# Empfohlene Vorgehensweise

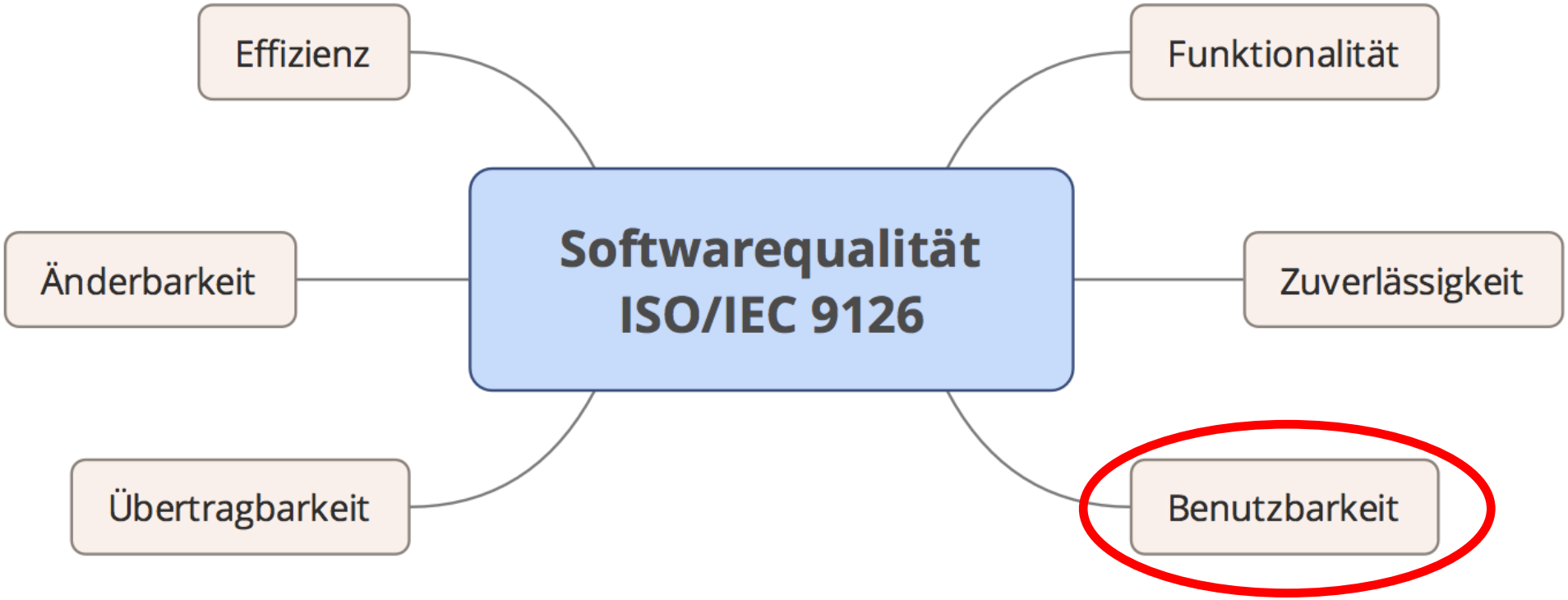



- Klein mit den wichtigsten Anforderungen beginnen
- API häufig schreiben, mit Beispielen entwerfen
- Regelmäßiges Feedback

# Empfohlene Vorgehensweise

- Tests für Entwurf, Dokumentation und Kompatibilitätsprüfung nutzen
- Consumer-driven Contract Testing

# Qualitätsmerkmale


Effizienz
Funktionalität
Änderbarkeit
Softwarequalität
ISO/IEC 9126
Zuverlässigkeit
Übertragbarkeit
Benutzbarkeit

# Entwurfsziele

- Konsistent
- Intuitiv verständlich
- Dokumentiert
- Einprägsam und leicht zu lernen
- Lesbaren Code fördernd
- Minimal
- Stabil
- Einfach erweiterbar

**Grundvoraussetzung:**
**Nützlich und korrekt implementiert :-)**

# Konsistent

- **Konzeptionelle Integrität** bedeutet
  kohärentes Design
  mit der Handschrift eines Architekten

# Konsistent

```
java.util.zip.GZIPInputStream
java.util.zip.ZipOutputStream
```

```
java.awt.TextField.setText();
java.awt.Label.setText();
javax.swing.AbstractButton.setText();
java.awt.Button.setLabel();
java.awt.Frame.setTitle();
```

# Inkonsistent vs semantisch genau

```
arr = [1, 2, 3]

arr.length # => 3
arr.size   # => 3
arr.count  # => 3
```

- length: Anzahl der Elemente im Array in konstanter Zeit (unabhängig der Länge), z.B. String
- size: z.B. Collections
- count: Aufruf typischerweise mit Parameter, traversiert das Objekt und zählt die Matches

# Intuitiv verständlich

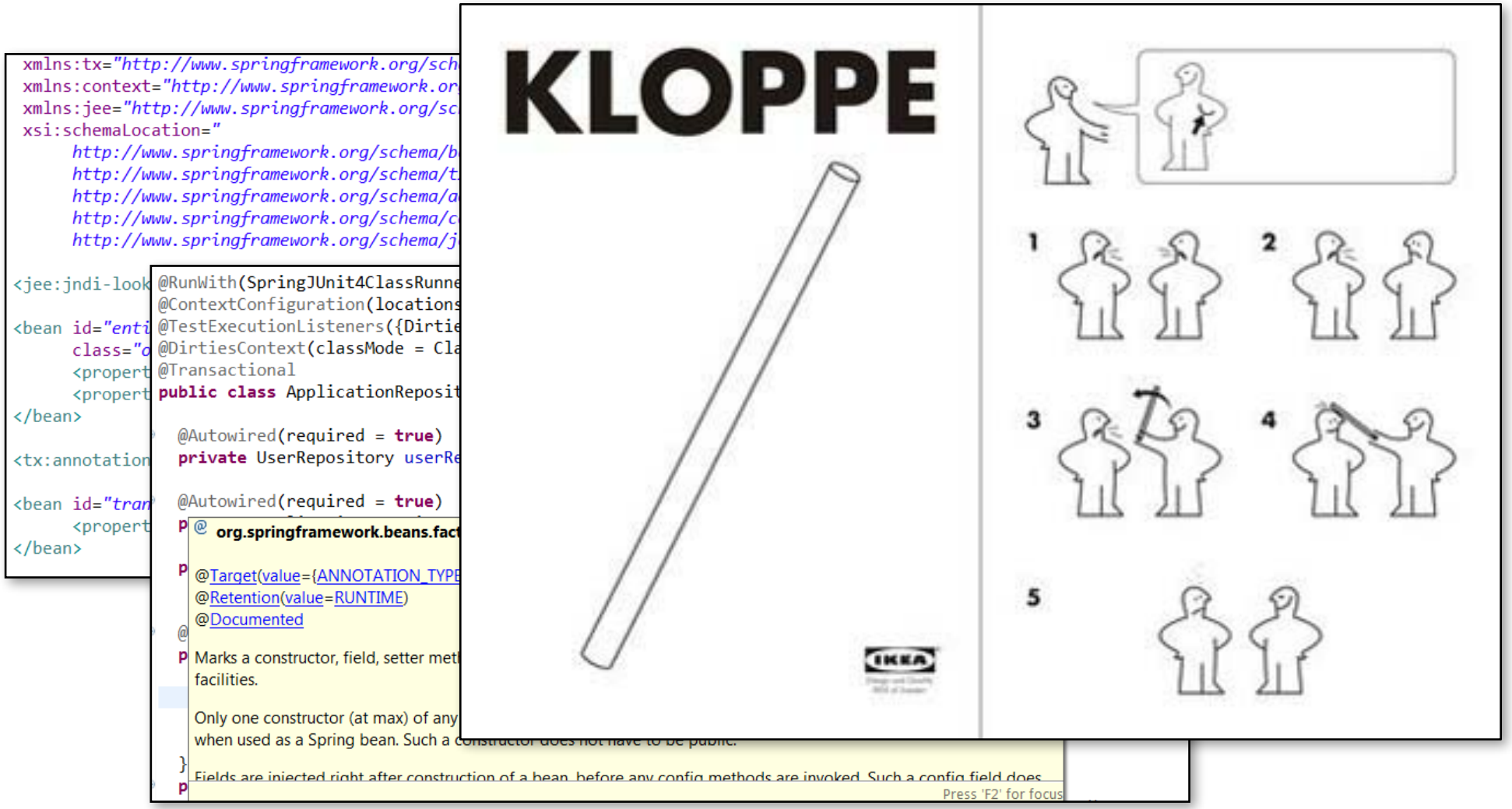
- Idealerweise ist Client-Code ohne Dokumentation verständlich
- Vorwissen ausnutzen und bekannte Konzepte wiederverwenden
- Starke Begriffe etablieren und diese konsequent wiederverwenden

# Java Collection API

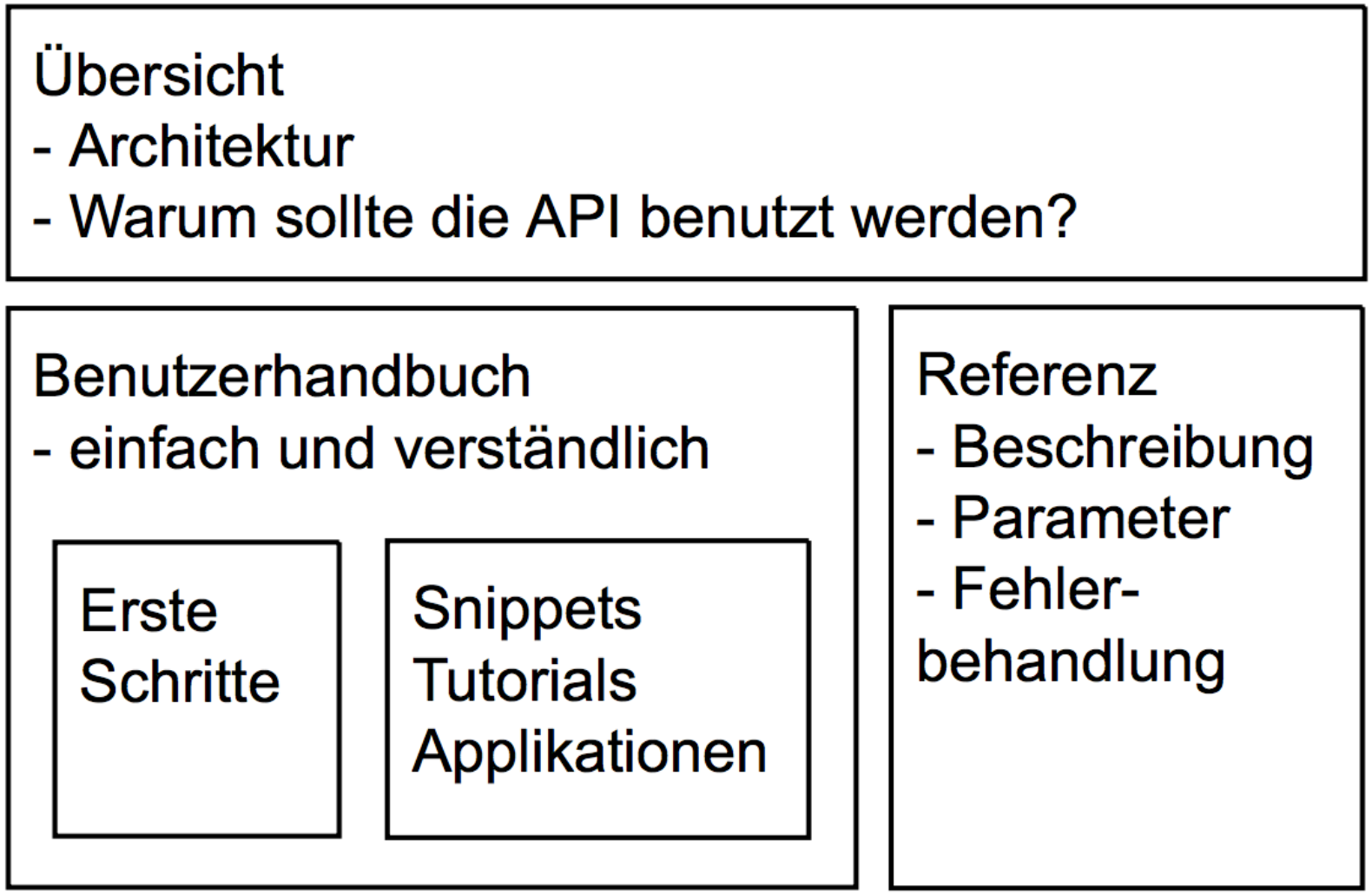
| java.util.List | java.util.Set | java.util.Map |
|---|---|---|
| add | add | put |
| addAll | addAll | putAll |
| contains | contains | containsKey, containsValue |
| containsAll | containsAll | - |
| remove | remove | remove |
| removeAll | removeAll | - |

# Dokumentiert

- Gute Dokumentation ist unverzichtbar



# Dokumentationsaufbau



*In Proc. OOPSLA 2006, ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. October 22-26, Portland, Oregon, USA.*

## Design Fragments Make Using Frameworks Easier

George Fairbanks, David Garlan, William Scherlis
Carnegie Mellon University
School of Computer Science
5000 Forbes Avenue
Pittsburgh, PA, 15213, USA
<george.fairbanks, david.garlan, william.scherlis>@cs.cmu.edu

### ABSTRACT

Object oriented frameworks impose additional burdens on programmers that libraries did not, such as requiring the programmer to understand the method callback sequence, respecting behavior constraints within these methods, and devising solutions within a constrained solution space. To overcome these burdens, we express the repeated patterns of engagement with the framework as a *design fragment*. We analyzed the 20 demo applets provided by Sun and created a representative catalog of design fragments of conventional best practice. By evaluating 36 applets pulled from the internet we show that these design fragments are common, many applets copied the structure of the Sun demos, and that creation of a catalog of design fragments is practical. Design fragments give programmers immediate benefit through tool-based conformance assurance and long-term benefit through expression of design intent.

### Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.13 [**Software Engineering**]: Reusable Software

### General Terms

Design, Documentation, Languages, Standardization

### Keywords

Object-oriented Programming, Frameworks, Patterns, Design Fragments

### 1. INTRODUCTION

Programmers use object oriented frameworks because they provide partially-complete and pre-debugged solutions to common problems, such as windowing systems and application servers. Popular examples of frameworks include Enterprise Java Beans[27], Microsoft .Net [4], and Java applets [26].



Frameworks differ from code libraries in that the framework, not the programmer, provides important parts of the architectural skeleton of the application [23] and, in doing so, places additional burdens on the programmer. The applet shown in Figure 1 is from the original Sun Java Development Kit (JDK) and it has a bug in it. No amount of code inspection can reveal the bug unless you also know how the applet framework will drive that code. The applet framework invokes the `start()` method then the `stop()` method on the programmer-provided code – perhaps this much could be guessed from the method names. Additionally, the runnable framework will invoke the `run()` method on the applet sometime after the `timer.start()` method is called. The bug is a race condition that can be reasoned about only if you know that the framework may call `start()` and `stop()` multiple times.

```
public class Clock2
  extends java.applet.Applet
  implements Runnable {
...
Thread timer = null;
...
public void start() {
  if (timer == null) {
      timer = new Thread(this);
      timer.start();
    }
}
public void stop() {
  timer = null;
}
public void run() {
  while (timer != null) {
    try {
      Thread.sleep(100);
    } catch (InterruptedException e){}
    repaint();
  }
  timer = null;
}
```

**Figure 1: Threaded applet example**

The programmer's intent during the `stop()` callback is to signal to the running thread that it should terminate. The signal is that the `timer` field is set to null. The race condition occurs when the framework invokes both `start()` and `stop()` before the thread

# Entwickler lernen und verstehen mit Beispielen

# Einprägsam und leicht zu lernen

## Einflussfaktoren

- Konsistenz, Verständlichkeit und Dokumentation
- Größe (Anzahl der Konzepte)
- Vorwissen der Benutzer
- Time to First Hello World

# Führt zu ~~leicht~~ lesbaren Code

```
EntityManager em = ...;
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Order> cq = builder
      .createQuery(Order.class);
Root<Order> order = cq.from(Order.class);
order.join(Order_.positions);
cq.groupBy(order.get(Order_.id)).having(
      builder.gt(builder.count(order), 1));
TypedQuery<Order> query = em.createQuery(cq);
List<Order> result = query.getResultList();
```

# Führt zu leicht lesbaren Code

```
EntityManager em = ...;


List<Order> list = new JPAQuery(em)
      .from(QOrder.order)
      .where(order.positions.size().gt(1))
      .list(order).getResults();
```

# Schwer ~~falsch~~ zu benutzen

```
new java.util.Date(2016, 11, 17);


int year = 2016 – 1900;
int month = 11 - 1;
new java.util.Date(year, month, 17);
```

# Minimal

- Hinzugefügte Elemente können nachträglich nur schwer entfernt werden
- Im Zweifel Elemente weglassen

- Trade-off: Minimalität vs. Komfort
  - Gut: `java.util.List.removeAll()`
  - Schlecht: `java.util.List.removeAllEven()`

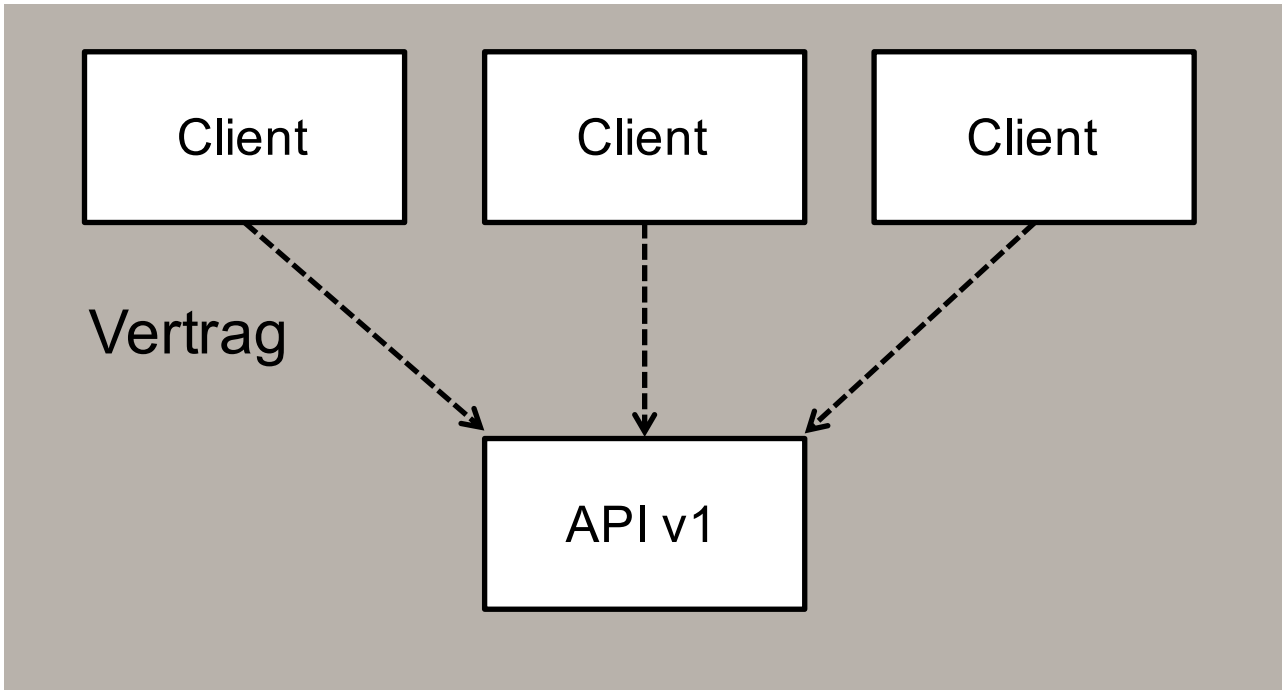
# Minimal

- Auslagerung in Hilfsklassen
  - z.B.: java.util.Arrays, java.util.Collections

- Nachteile:
  - Performance und Nebenläufigkeit
    (putIfAbsent() wäre in separater Hilfsklasse nicht threadsicher)
  - Schlechter OO-Stil
  - Hilfsklassen oft Sammelsuirum

# Stabil

- API-Änderungen sollten nur mit Sorgfalt durchgeführt werden



# Open Closed Principle

- Laut Bertrand Meyer sollen Module nicht verändert, sondern nur durch Vererbung erweitert werden (kein gutes OO-Design-Prinzip)

- Refactorings sind einfach, falls die Änderungen nur die eigene Codebasis betreffen

- Bessere Idee von Martin Fowler: Published Interface
  (http://martinfowler.com/ieeeSoftware/published.pdf)

# Einfach erweiterbar

- Welche Änderungen wird es in Zukunft geben?
- Erweiterung idealerweise abwärtskompatibel, sonst neue API-Version notwendig
- Anpassungsaufwand für Clients minimieren

# Abwärtskompatibilität

# Kompatibilität

- Trade-off: Abwärtskompatibilität vs. Flexibilität

- **Code-Kompatibilität**
- Schwächste Form der Kompatibilität
- Konflikte beim Entfernen und Verändern von API-Elementen
- Konflikte eventuell bei Vererbung

# Kompatibilität

```
/**
 * @since 1.0
 */
class ApiBaseClass {
}
```

```
class MyOwnClass extends ApiBaseClass {
      List<ResultItem> getResults() {
            ...
      }
}
```

# Kompatibilität

```
class ApiBaseClass {
      /**
       * @since 1.1
       */
      List<String> getResults() {
            ...
      }
}

class MyOwnClass extends ApiBaseClass {
      List<ResultItem> getResults() {
            ...
      }
}
```

# Kompatibilität

- Binär-kompatibel: Programm kann neue Version ohne erneute Kompilierung verwenden
- Beispiel:
  - > Kompilieren mit log4j 1.2.15
  - > Linken gegen log4j 1.2.17
- Grundvoraussetzung: Dynamisches Linken
  - > Einsprungadresse von virtuellen Methoden wird zur Laufzeit bestimmt

# Funktionale Kompatibilität

- Version kann gelinkt werden und erfüllt ihre erwartete Funktion



Erwartetes Verhalten

Tatsächliches Verhalten

Tatsächliches Verhalten der neuen Version

Bekannt als Amöben-Effekt (Jaroslav Tulach)

# Wie wird Kompatibilität sichergestellt?

# Versionierte Regressionstests

- Offizielles (dokumentiertes) API-Verhalten muss getestet werden
- Automatisierte Tests stellen Verhaltenskompatibilität sicher
- Niemals vollständig, Erweiterungen nach Rücksprache mit Clients

Was gehört zu API ?

```
com.company.foo.api
com.company.foo.internal
```

# Umgang mit inkompatiblen Änderungen

# API in Version 1.0

```
public interface FinanceService {
  /**
   * Returns brutto sum.
   * @since 1.0
   */
  int getSum();
}
```

# API in Version 1.0

```
public interface FinanceService {
  /**
   * Returns brutto sum.
   * @since 1.0
   */
  int getSum();
}
```

# API in Version 1.1 mit semantischer Änderung

```
public interface FinanceService {
  /**
   * Returns netto sum.
   * @since 1.0 // 1.1 ?
   */
  int getSum();
}
```

# API in Version 1.1

## Semantische und syntaktische Änderungen verbinden

```
public interface FinanceService {
   /**
    * Returns netto sum.
    * @since 1.1
    */
   int getNettoSum();
}
```

# API in Version 1.1

## Rückwärtskompatibilität beachten

```
public interface FinanceService {
  /**
   * Returns brutto sum.
   * @since 1.0
   */
  int getSum();

  /**
   * Returns netto sum.
   * @since 1.1
   */
  int getNettoSum();
}
```

```
public interface FinanceService {
  /**
   * Returns brutto sum.
   * @since 1.0
   * @deprecated As of version 1.1 for
   *             API improvements, use
   *             {@link #getBruttoSum()
   *             instead. Will be removed in
   *             version 1.2.
   */
  @Deprecated int getSum();

  /**
   * Returns brutto sum.
   * @since 1.1
   */
  int getBruttoSum();
}
```

# Design-Repertoire

# Erzeugungsmuster

- Factories und Builder statt Konstruktoren

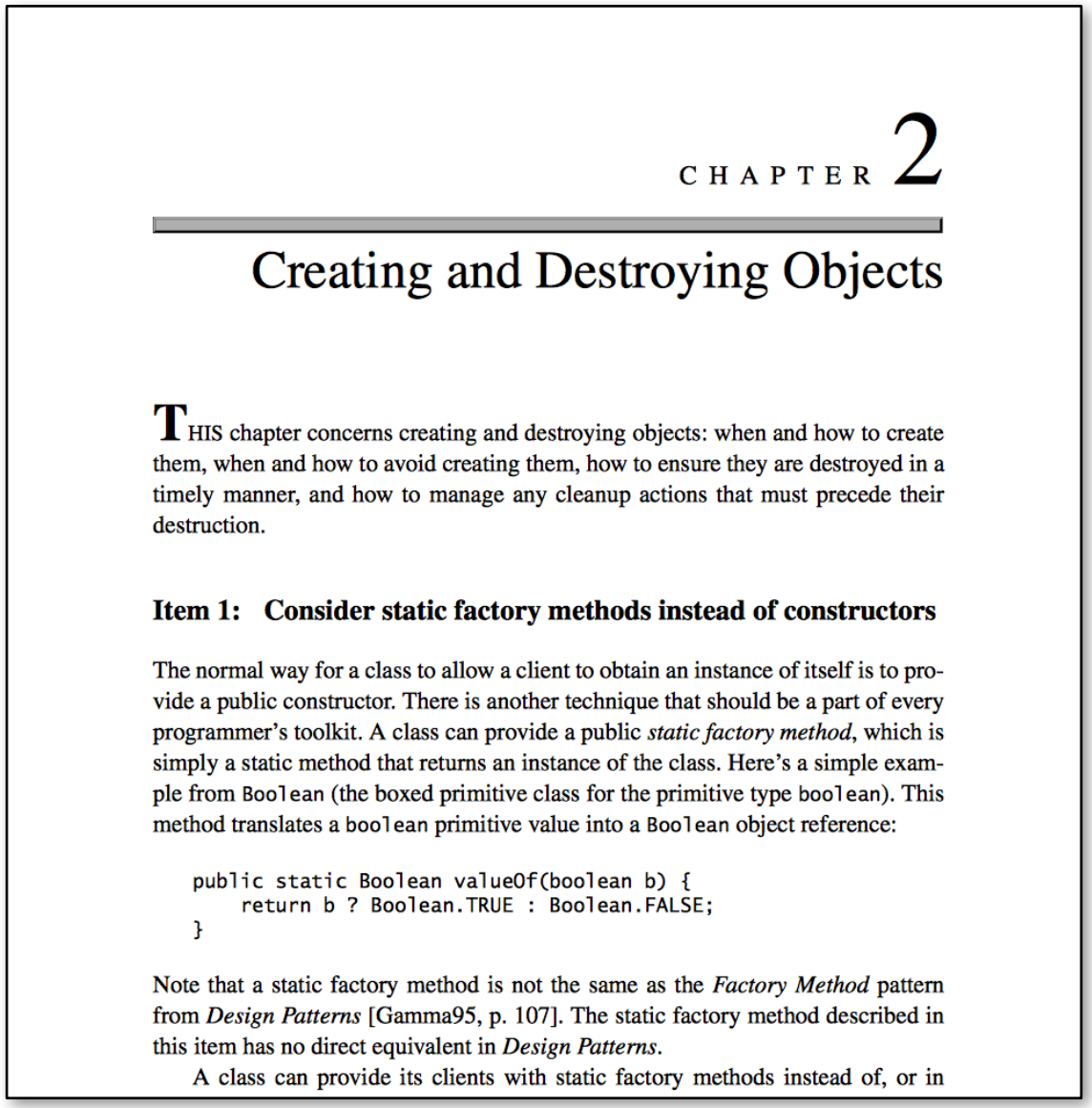
```
entityManager.createNamedQuery(queryName)
             .getResultList();
```

```
CalculatePriceCommand.new(order).run();
```

## Vorteil

- Verwendung unterschiedlicher Subtypen
- Erzeugung komplexer Objekte vereinfachen
- Keine Konstruktoren mit vielen Parametern

# Effective Java 2nd Edition



CHAPTER 2

## Creating and Destroying Objects

THIS chapter concerns creating and destroying objects: when and how to create them, when and how to avoid creating them, how to ensure they are destroyed in a timely manner, and how to manage any cleanup actions that must precede their destruction.

### Item 1: Consider static factory methods instead of constructors

The normal way for a class to allow a client to obtain an instance of itself is to provide a public constructor. There is another technique that should be a part of every programmer's toolkit. A class can provide a public *static factory method*, which is simply a static method that returns an instance of the class. Here's a simple example from `Boolean` (the boxed primitive class for the primitive type `boolean`). This method translates a `boolean` primitive value into a `Boolean` object reference:

```
public static Boolean valueOf(boolean b) {
    return b ? Boolean.TRUE : Boolean.FALSE;
}
```

Note that a static factory method is not the same as the *Factory Method* pattern from *Design Patterns* [Gamma95, p. 107]. The static factory method described in this item has no direct equivalent in *Design Patterns*.

A class can provide its clients with static factory methods instead of, or in

- Static Factory, statt Konstruktoren nutzen

# Alternative: Class Clusters

## The Factory Pattern in API Design: A Usability Evaluation

Brian Ellis, Jeffrey Stylos, and Brad Myers
Carnegie Mellon University
firebird@cs.cmu.edu, jsstylos@cs.cmu.edu, bam@cs.cmu.edu

### Abstract

*The usability of software APIs is an important and infrequently researched topic. A user study comparing the usability of the factory pattern and constructors in API designs found highly significant results indicating that factories are detrimental to API usability in several varied situations. The results showed that users require significantly more time ($p = 0.005$) to construct an object with a factory than with a constructor while performing both context-sensitive and context-free tasks. These results suggest that the use of factories can and should be avoided in many cases where other techniques, such as constructors or class clusters, can be used instead.*

### 1. Introduction

Whether creating a piece of desktop software, writing applications for handheld devices, or scripting the Web, the use of application programming interfaces (APIs) in modern software development is ubiquitous. These APIs, also called software development kits (SDKs) or libraries, are often large, complex, and broad in scope, containing many hundreds or thousands of classes and interfaces. A typical developer may use only a small portion of the total functionality of an API, but learning even that subset can be a daunting task for new programmers [1].

API designers must consider many different factors when creating an API, such as class granularity, level of abstraction, consistency with other APIs, etc. Research has also shown that designing APIs carefully for their intended audience improves usability [2]. To date,

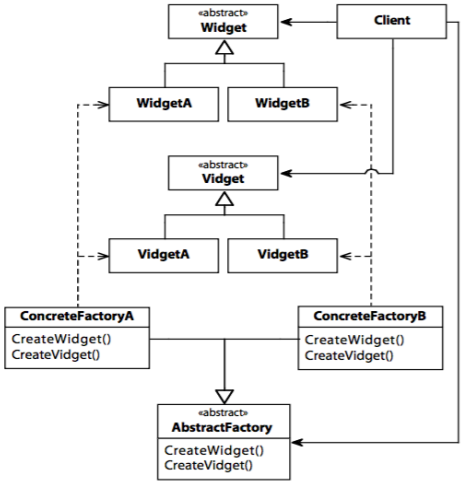



**Figure 1. The abstract factory pattern in UML**

consider the usability implications of one of the best-known object-oriented design patterns: the factory pattern [4]. Our new study shows that creating objects from factories used in APIs is significantly more time-consuming than from constructors, regardless of context or the level of experience of the programmer using the API. The reasons for this, as well as a discussion of specific stumbling blocks and possible alternative patterns, are discussed below.

### 2. The Factory Pattern

# Beispiel: Class Cluster

```
public class Widget {
    private Widget body;

    public Widget(Params p) {
         if (p.featureToggle().enabled()) {
             body = new WidgetA();
         } else {
             body = new WidgetB();
         }
     }

    public void performAction() {
         body.performAction(); }
    }
}
```

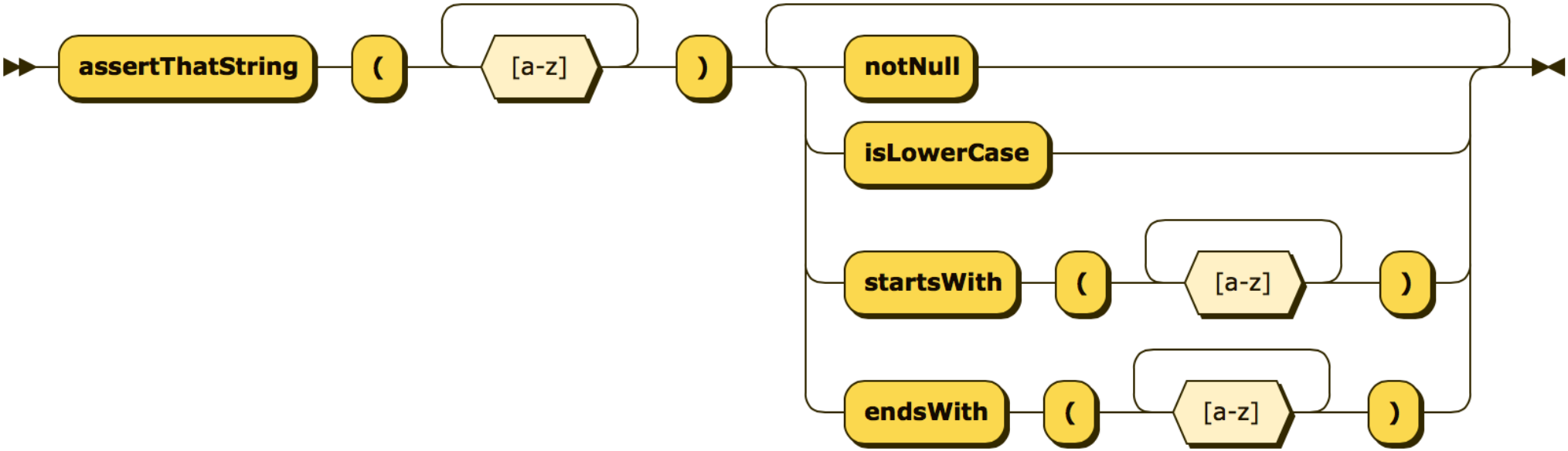
# Fluent Interface

- Interne DSL zur Verbesserung der Lesbarkeit des Quellcodes
- Umsetzung mit:
  - Methodenketten
  - (und Methodenschachtelung)

# Fluent Interface (Beispiel)

```
Grammar ::= 'assertThatString' '('[a-z]+')'
                    (  'notNull'
                     | 'isLowerCase'
                     | 'startsWith' '('[a-z]+')'
                     | 'endsWith' '('[a-z]+')'
                    )+
```

# Fluent Interface (Beispiel)



# Fluent Interface (Beispiel)

```
// ohne Fluent Interface
assertNotNull(text);
assertTrue(text.startsWith("a"));
assertTrue(text.endsWith("z"));
assertTrue(text.isLowerCase());

// mit Fluent Interface
assertThatString(text)
  .notNull()
  .startsWith("a")
  .endsWith("z")
  .isLowerCase();
```

# Vererbung

## Kontroverse Ansätze:

- Open Inheritance
- Designed Inheritance

## Standardmäßig alle konkreten Klassen final machen

- Vererbung verletzt Datenkapselung
- Klassen müssen speziell für Vererbung entworfen werden
- Für Wiederverwendung Kompositionen nutzen

## Beispiele:

- Richtig: Set extends Collection
- Falsch: Stack extends List

# Template-Methode

```
abstract class OrderSorter {

  public void sort(List<Order> orders) {
    …
    if(compare(o1, o2)) {
    …
  }

  abstract compare(Order o1, Order o2);

}
```

# Service Provider Interfaces (SPI) von der sonstigen API trennen

# Template-Methode

```
public class OrderSorter {

  private final Comparator<? super Order> comp;

  public OrderSorter(Comparator<? super Order> comp) {
    this.comp = comp;
  }

  public void sort(List<Order> orders) {
    Collections.sort(orders, comp);
  }

}
```

# Kontextobjekt als Parameter für SPI

- Schlechter Stil:

```
public interface MyServiceProviderInterface {
    void callbackMethod(String param);
}
```

- Besser:

```
public interface ServiceProviderInterface {
    void callbackMethod(Context param);
}
public class Context {
  String message();
  Integer id();
}
```

# Interface-Evolution

- Hinzufügen einer neuen abstrakten Methode war vor Java 8 eine inkompatible Änderung

```
interface NewInterface extends OldInterface {
   void newMethod();
}
```

- Ab Java 8 Default-Methoden
- Bevorzuge Klassen vor Interfaces

# Checked Exceptions

```
void submit( Abstract abstract )
    throws IOException {
        ...
}
```

Falsches Abstraktionsniveau

# Checked Exceptions

Kann diese Ausnahmesituation im „**korrekten**“ Programm auftreten?

```
void submit( Abstract abstract )
    throws TooLongAbstractException {
        ...
}
```

Kann sich ein Client davon **erholen**?

# Anwendungsfälle für Checked Exceptions sind selten

(konservative Formulierung :-)

# Unchecked Exceptions

**Gehört** eine Unchecked Exception **zur API?**

```
void submit( Abstract abstract ) {
    if( abstract == null ) {
        throw new NullPointerException(
            "abstract must not be null.");
    }
        ....
}
```

# Unchecked Exceptions

```
/**
 *
 * @throws  NullPointerException
 *     if abstract is null
 */
void submit( Abstract abstract ) {
    if( abstract == null ) {
        throw new NullPointerException(
            "abstract must not be null.");
    }
        ....
}
```

**Dokumentiere alle Exceptions, die von einer Methode geworfen werden (Vorbedingungen).**

# Invarianten und Bedingungen

- Klassen sind selbst für die Einhaltung ihrer **Invarianten** verantwortlich (nicht Client)
- Klassen sollten ihre **Vorbedingungen** möglichst gut überprüfen
- **Nachbedingungen** werden mit Tests überprüft

# Immutability / Minimale Änderbarkeit

- Klassen, die public und konkret sind, sollten standardmäßig final und immutable sein

**Vorteile:**

> Thread-safe
> Wiederverwendbar
> Einfach

**Nachteile:**

> Separates Objekt für jeden Wert

- Falls änderbar, dann sollte Zustandsraum klein sein

# Sinnvolle Defaults

- Vermeidung von Boilerplate-Code
- Geringe Einstiegshürden und Unterstützung für Neueinsteiger
- Defaults müssen dokumentiert sein
- Boolean sind standardmäßig false (entsprechende Namen wählen)

```
// Tests werden standardmäßig ausgeführt
// skipTests=false
```

# Globaler Zustand / Konfiguration

## Globaler Zustand ist problematisch weil:

- Schwer testbar, nicht flexibel
- Fehleranfällig, schwer verständlich

## Empfehlung:

- Sei funktional, vermeide Zustand
- Einbindung einer Bibliothek sollte keine Änderungen des globalen Zustand notwendig machen
- Falls doch, dann ist es eine Applikation und keine Bibliothek

## Beispiel:

- Log4j wird beim Laden automatisch konfiguriert
- Unterschiedliche Konfigurationen für Log4j in einem Prozess nicht möglich

# Bedeutung von Typen & Namen

Namen sind wie Kommentare (nützlich, aber schnell veraltet)

```
String findAddress( String userId );
Address foo( UserId blah );
```

Typen sind aussagekräftiger und verlässlicher als Namen und Kommentare

# Bedeutung von Typen

## Mit Typen Dokumentieren

Gibt -1, 0 oder 1 zurück

```
int compare( Object o1 );
```

```
Result compare( Object o1 );
```

```
enum Result {
  LESS_THEN,
  EQUAL,
  GREATER_THEN;
}
```

Rückagebtypen und Argumente mit benannten Alternativen einschränken

Kai Spichale

@kspichale

http://spichale.blogspot.de/

https://www.xing.com/profile/Kai_Spichale