

Software Craftmanship und Clean Code

Organisation

2

- Vorlesung
 - Notebook / Gruppenarbeit / Sessions
- Prüfung
 - 90 Minuten
 - schriftlich, keine Hilfsmittel
- Code
 - <https://github.com/mnhock/swcs>
- Buch
 - <https://leanpub.com/clean-code-fundamentals>
- Kontakt
 - Martin Hock
 - martin.hock.de@gmail.com

Zielsetzung

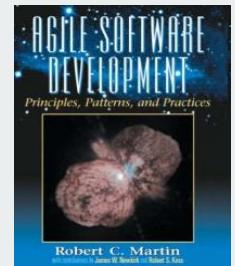
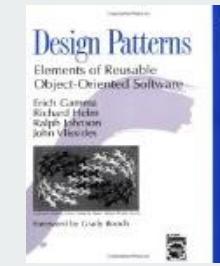
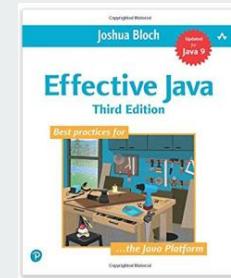
- Verständnis wichtiger Konzepte von Softwarequalität in Java Projekten
 - Grundlagen der Softwarequalität
 - Best Practices
 - Funktions- und Technologieumfang
 - Entwicklungs- und Programmiermodelle
 - Tools
- Vorgehen
 - Technologiebetrachtungen
 - Querbezüge und Vergleiche
 - Demos
 - Präsentationen
- Lernziele
 - Erwerb von Kenntnissen und Fähigkeiten zur Analyse, Beurteilung und Verbesserung von Softwarequalität.
 - Erlernen von Prinzipien, Patterns, Techniken und Werkzeugen
 - Aufbau von Technologiewissen

Struktur der Vorlesung

- Grundlagen der Softwarequalität
- Grundlagen von Softwaredesign
- Java Best Practices
- Werkzeuge zur Sicherung der Softwarequalität
- Prinzipien Objektorientierten Designs
- Design Patterns

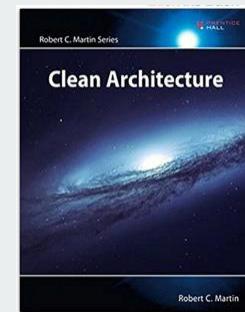
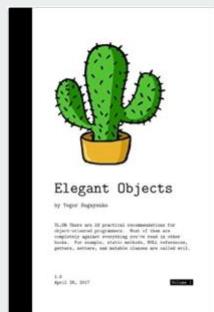
Literatur (1/2)

- Schneider, Kurt: *Abenteuer Software Qualität – Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement*, dpunkt.verlag, 2007
- Robert, Martin: *Clean Code – Refactoring, Patterns, Testen und Techniken für sauberen Code*, mitp-Verlag, 2009
- Lilienthal, Carola: *Langlebige Software-Architekturen*, Dpunkt Verlag, 2015
- Bloch, Joshua: *Effective Java – Third Edition*, Addison Wesley, 2017
- Roock, Stefan: *Refactorings in grossen Softwareprojekten*, Dpunkt Verlag, 2004
- Gamma, Erich: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994
- Robert C. Martin: *Agile Software Development: Principles, Patterns and Practices*, Prentice Hall, 2003



Literatur (2/2)

- Bugayenko, Yegor: *Elegant Objects Volume 1*, CreateSpace Independent Publishing Platform, 2016
- Bugayenko, Yegor: *Elegant Objects Volume 2*, CreateSpace Independent Publishing Platform, 2017
- Harrer, Simon: *Java by Comparison: Become a Java Craftsman in 70 Examples*, O'Reilly UK Ltd., 2018
- Robert, Martin: *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Prentice Hall, 2017
- Robert, Martin: *The Clean Coder: A Code of Conduct for Professional Programmers*, Prentice Hall, 2011



10 Dinge über Software-Entwicklung, die man nicht an der Hochschule lernt (1/2)

7

1. Entwickler haben immer unrecht – nur der Grad unterscheidet sich, in dem verschiedene Entwickler in ihren endlosen Architekturdiskussionen falsch liegen!
2. Wenn etwas kaputt gehen kann, wird es auch kaputt gehen – also bitte alles daran setzen, dass ein Projekt narrensicher ist!
3. Jeder Code ist schlecht – auch hier gibt es nur graduelle Abstufungen der Mangelhaftigkeit eines Stücks Software!
4. Es gibt immer irgendwo einen Bug – man muss nur lang genug danach suchen!
5. Das wichtigste ist der Kunde – und den interessieren die verwendeten Technologien und Prozesse nicht die Bohne!

10 Dinge über Software-Entwicklung, die man nicht an der Hochschule lernt (2/2)

6. Projektentwicklung auf dem Papier funktioniert nicht – Probleme erkennt man erst während des Entwicklungsprozesses!
7. Weniger ist mehr – wenn etwas nicht nötig ist, einfach weglassen (Keep it simple, stupid)!
8. Nur 20% unserer Arbeit besteht aus Kodieren – der Rest ist Konzipieren, Debuggen, Testen, Meetings, Besprechen etc.
9. Der Kunde weiß NIE, was er will – alle haben nur eine vage Idee von dem gewünschten Ergebnis!
10. Irgendjemand hat es bereits vorher gemacht – also: nicht das Rad neu erfinden wollen!

Grundlagen von Software Craftmanship und Clean Code

Software Craftsmanship Manifest

10

Als engagierte Software-Handwerker heben wir die Messlatte für professionelle Softwareentwicklung an, indem wir üben und anderen dabei helfen, das Handwerk zu erlernen. Durch diese Tätigkeit haben wir diese Werte zu schätzen gelernt:

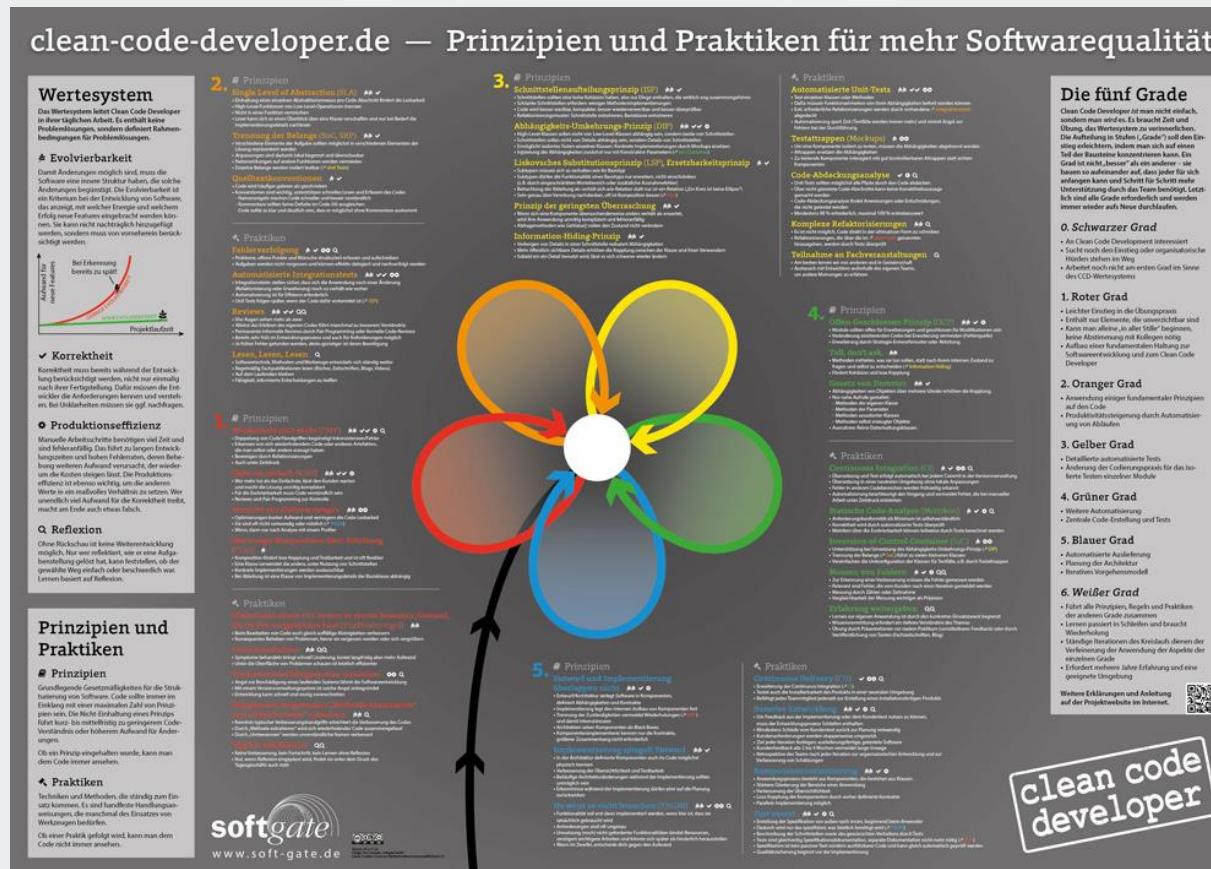
- Nicht nur funktionierende Software, sondern **auch gut gefertigte Software**
- Nicht nur auf Veränderung zu reagieren, sondern **stets Mehrwert zu schaffen**
- Nicht nur Individuen und Interaktionen, sondern **auch eine Gemeinschaft aus Experten**
- Nicht nur Zusammenarbeit mit dem Kunden, sondern **auch produktive Partnerschaften**

Das heißt, beim Streben nach den Werten auf der linken Seite halten wir die Werte auf der rechten Seite für unverzichtbar.

- <http://manifesto.softwarecraftsmanship.org/>

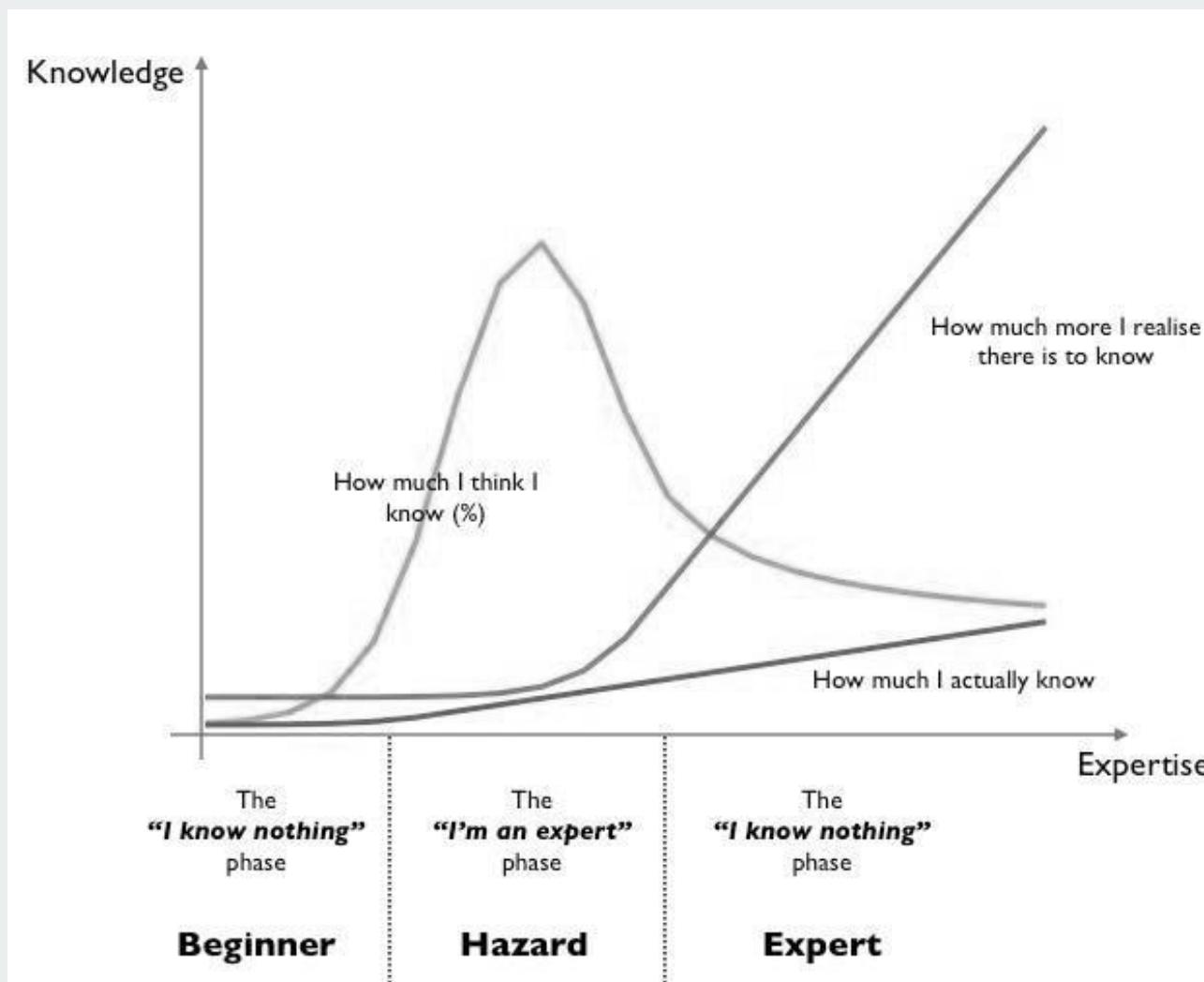
Clean Code Developer

- Eine Initiative für mehr Professionalität in der Softwareentwicklung
- <http://clean-code-developer.de/>



Wissen - Expertise

12



Boy Scout Rule

- *"Always leave the campground cleaner than you found it."*
- *"Always check a module in cleaner than when you checked it out."*



Broken windows theory (1/2)

Broken-Windows-Theorie

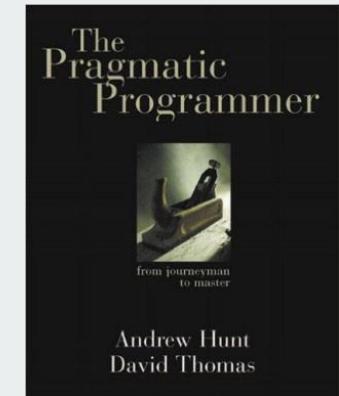
Die Broken-Windows-Theorie geht auf ein Experiment im Jahr 1969 zurück, in dem der Psychologe Philip Zimbardo ein Auto ohne Nummernschild und mit hochgeklappter Motorhaube in einem sozialen Brennpunkt abstellte, um die Reaktionen von Passanten zu untersuchen [1]. Zum Vergleich stellte er ein ähnliches Auto in einer besseren Wohngegend ab. Dem ersten Auto wurden nach etwa fünfzehn Minuten von Passanten Teile wie die Batterie ausgebaut. Es dauerte nicht lange, bis das Auto vollständig zerstört war und von Kindern als Spielplatz genutzt wurde. Das zweite Auto in der besseren Wohngegend wurde eine Woche lang nicht angerührt, bis die Forscher selbst ein Fenster einschlugen. Danach dauerte es nicht lange, bis dem zweiten Auto eine sehr ähnliche Geschichte widerfuhr wie dem ersten. In beiden Fällen waren die meisten Täter gut gekleidet und passten somit nicht in das erwartete Bild eines

Vandalen. Aus diesem Experiment ist die Broken-Windows-Theorie hervorgegangen, die aussagt, dass das Verhalten von Menschen direkt mit ihrer Umgebung zusammenhängt. Sind in der Umgebung eingeschlagene Fenster üblich, ist die Schwelle, selbst das Fenster eines verlassenen Autos einzuschlagen sehr gering. Ist das Auto bereits beschädigt, wird die Schwelle weiter gesenkt.

Die Broken-Windows-Theorie lässt sich gut in die Softwareentwicklung übertragen. Gibt es in einer Codebasis bereits schlechten Code (Broken Windows), kann man davon ausgehen, dass die Entwickler auch weiterhin schlechten Code produzieren. Ein konkretes Beispiel ist eine Codebasis, in der es mehrere Tausend Verstöße gegen Checkstyle-Regeln gibt. Den meisten Entwicklern wird es relativ egal sein, wenn der eigene Code ein paar neue Verstöße hinzufügt.

Broken windows theory (2/2)

- Beschrieben in *The Pragmatic Programmer*
von Andy Hunt und Dave Thomas
- *"The importance of fixing the small problems in your code,
the "broken windows," so they don't grow into large problems."*



Cargo cult programming

- Cargo cult programming is a style of computer programming characterized by the ritual inclusion of code or program structures that serve no real purpose.
- The term cargo cult programmer may apply when an unskilled or novice computer programmer (or one inexperienced with the problem at hand) copies some program code from one place to another with little or no understanding of how it works or whether it is required in its new position.
- Cargo cult programming can also refer to the results of applying a design pattern or coding style blindly without understanding the reasons behind that design principle.
- https://en.wikipedia.org/wiki/Cargo_cult_programming

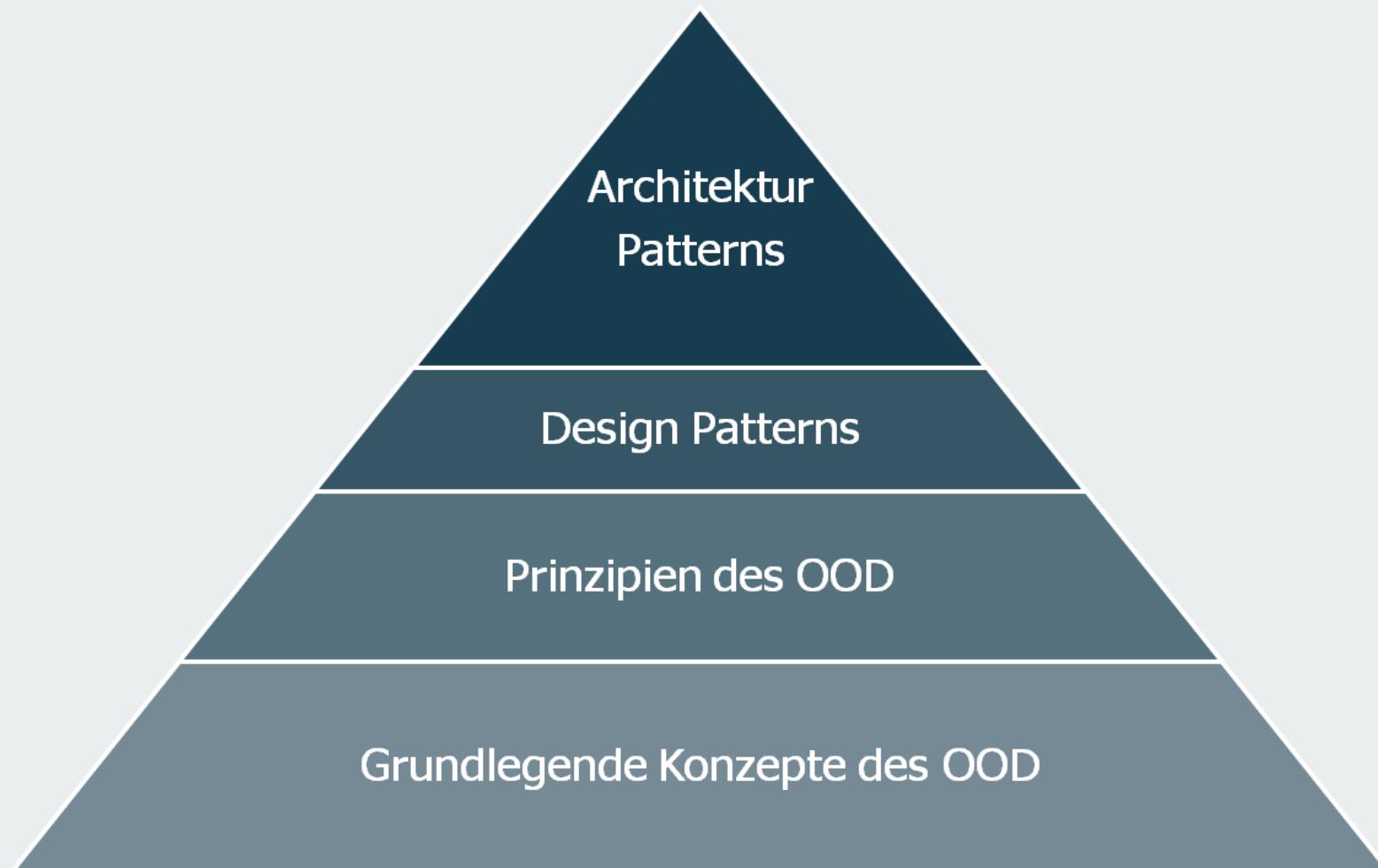
Fazit

"Nothing is more expensive than hiring cheap developers!"

Grundlagen von Softwaredesign

Softwaredesign Pyramide

19



Ziele des Softwaredesigns

- Design ist unsichtbar
 - Software, die alle funktionalen Anforderungen genügt, kann trotzdem schlecht entworfen sein.
- Design ist gut wenn
 - es die Komplexität der Software in handbare und einfache Probleme herunterbricht
 - schlanke Schnittstellen definiert wurden
 - die Komponenten entkoppelt sind
 - die Komponenten klar definierte Verantwortlichkeiten besitzen
 - die Software wartbar ist
 - die Software leicht geändert und erweitert werden kann
 - die Software stabil ist
 - Bugs schnell behebbar sind
 - die Software in anderen Softwareprojekten wiederverwendbar ist
 - der Code verständlich ist

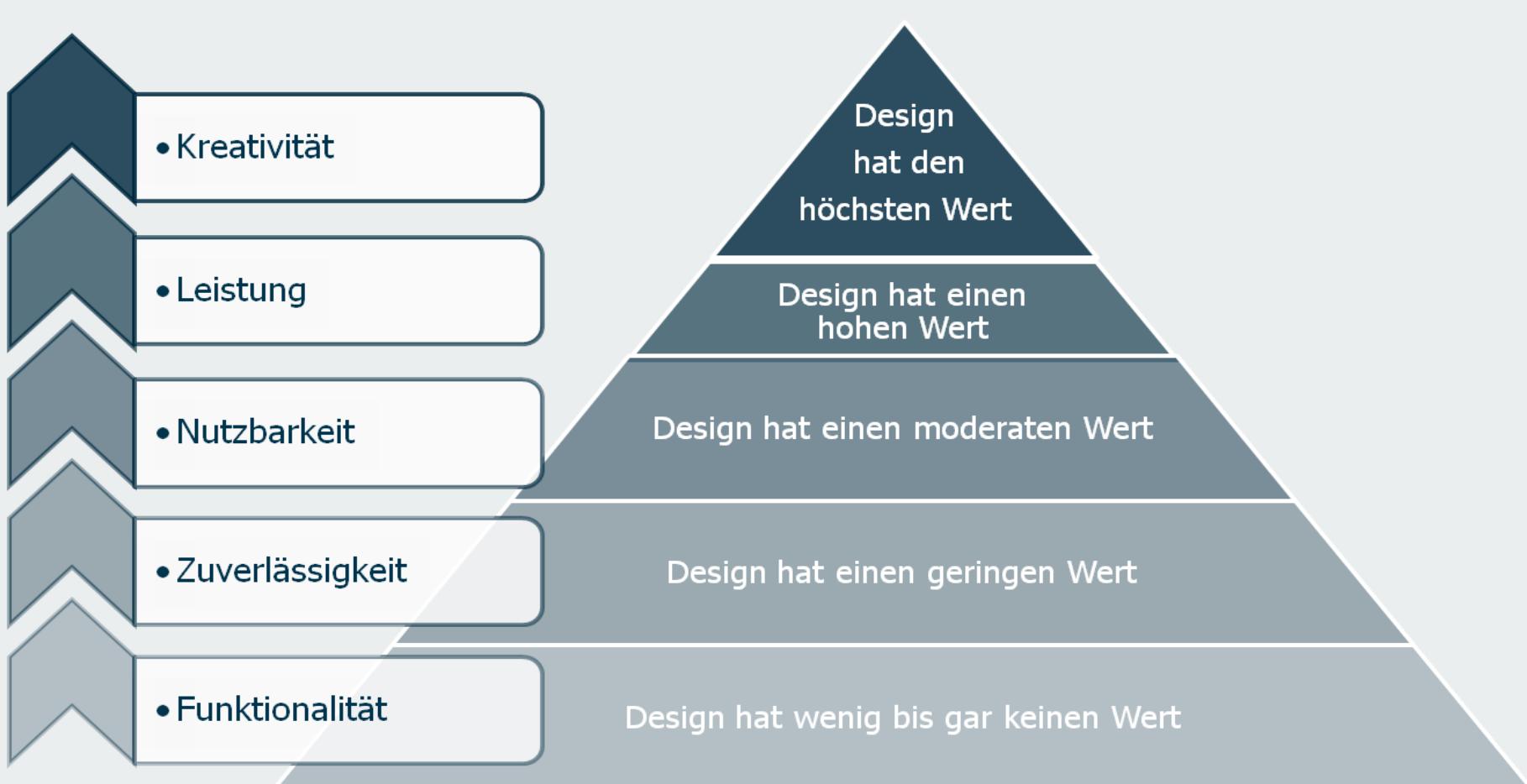
Symptome schlechten Designs

- Never-touch-running-code Syndrom
 - Entwickler haben Angst undurchschaubaren Code zu ändern.
 - Workarounds, es wird Code darum herum entwickelt.
- Änderungen haben unbekannte und unerkannte Seiteneffekte
 - Bugfixing produziert weitere Bugs, die erst viel später gefunden werden.
- Kleine Änderung in den Anforderungen führt zu großen Änderungen im Code
- Wiederverwendung durch Codeduplikation (Copy-Paste)
 - Entwickler jagen den Stellen hinterher die geändert werden müssen.
 - Je mehr Code desto schwieriger wird es die Übersicht zu behalten.
 - Fehler müssen an verschiedenen Stellen mehrfach geflickt werden.
- Zyklische Beziehungen zwischen Artefakten
 - Artefakte die zyklisch gekoppelt sind können nicht einzeln getestet werden.
 - Artefakte die in verschiedenen Zyklen gebraucht werden spielen häufig mehrere Rollen, was sie schlecht verständlich macht.
 - Artefakte die in verschiedenen Zyklen gebraucht werden können nicht mehr einfach ausgetauscht werden.

Kriterien für “gutes” Design

- Korrektheit
 - Erfüllung der Anforderungen
 - Wiedergabe aller Funktionen des Systemmodells
 - Sicherstellung der nichtfunktionalen Anforderungen
- Verständlichkeit
 - Selbsterklärender Code, Design
 - Gute Dokumentation
- Anpassbarkeit
- Hohe Kohäsion innerhalb der Komponenten
- Entkopplung der Komponenten
- Wiederverwendung
- Zyklendifreiheit
- Diese Kriterien sind fraktal, d. h. sie gelten auf allen Ebenen des Entwurfs (Architektur, Subsysteme, Komponenten).

Design-Hierarchie der Bedürfnisse

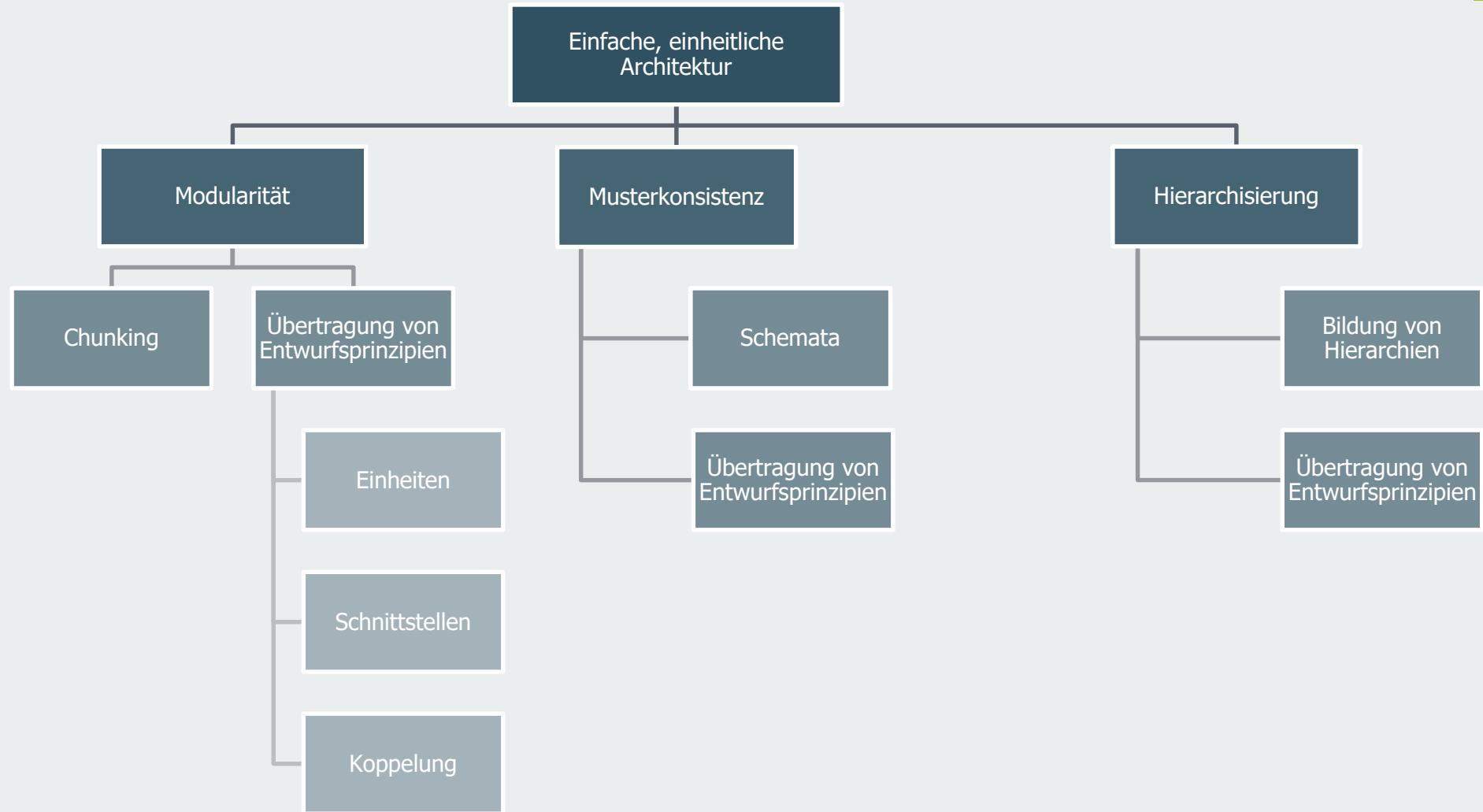


Grundlegende Konzepte des OOD

- Abstraktion (*abstraction*)
 - Ignorieren irrelevanter Details
- Kapselung (*encapsulation*)
 - Geheimnisprinzip (*information hiding*)
 - Trennung von Implementation und Schnittstelle
- Vererbung (*inheritance*)
 - Erweiterung und Spezialisierung
- Polymorphismus (*polymorphism*)
 - Austauschbarkeit (*substitutability*)
 - Gleiche Schnittstelle, anderes Verhalten

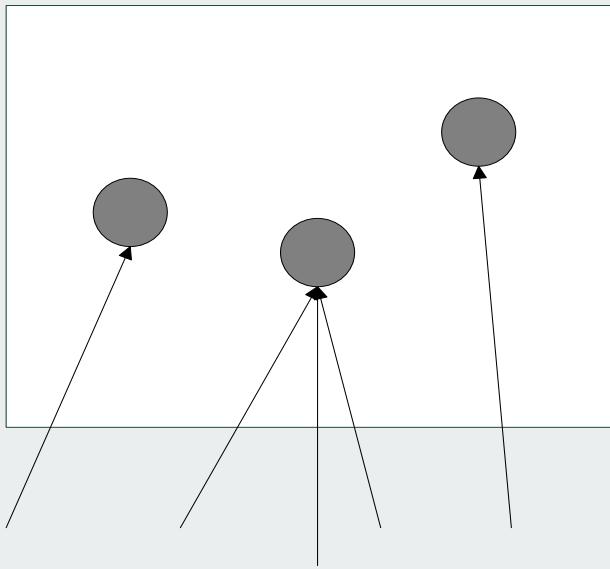
Kognitive Psychologie und Architekturprinzipien

25



Information Hiding (1/3)

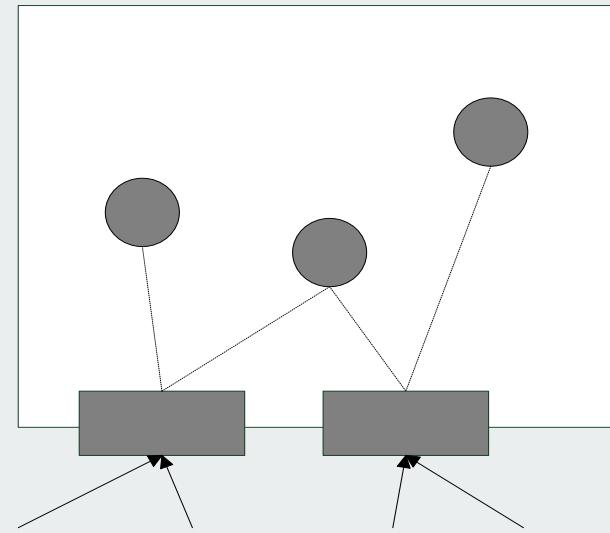
Globale Variablen



Direkter Zugriff auf Daten

**Schlechtes Information
Hiding!**

Klasse mit private Attributen



Zugriff nur über öffentliche
Methoden

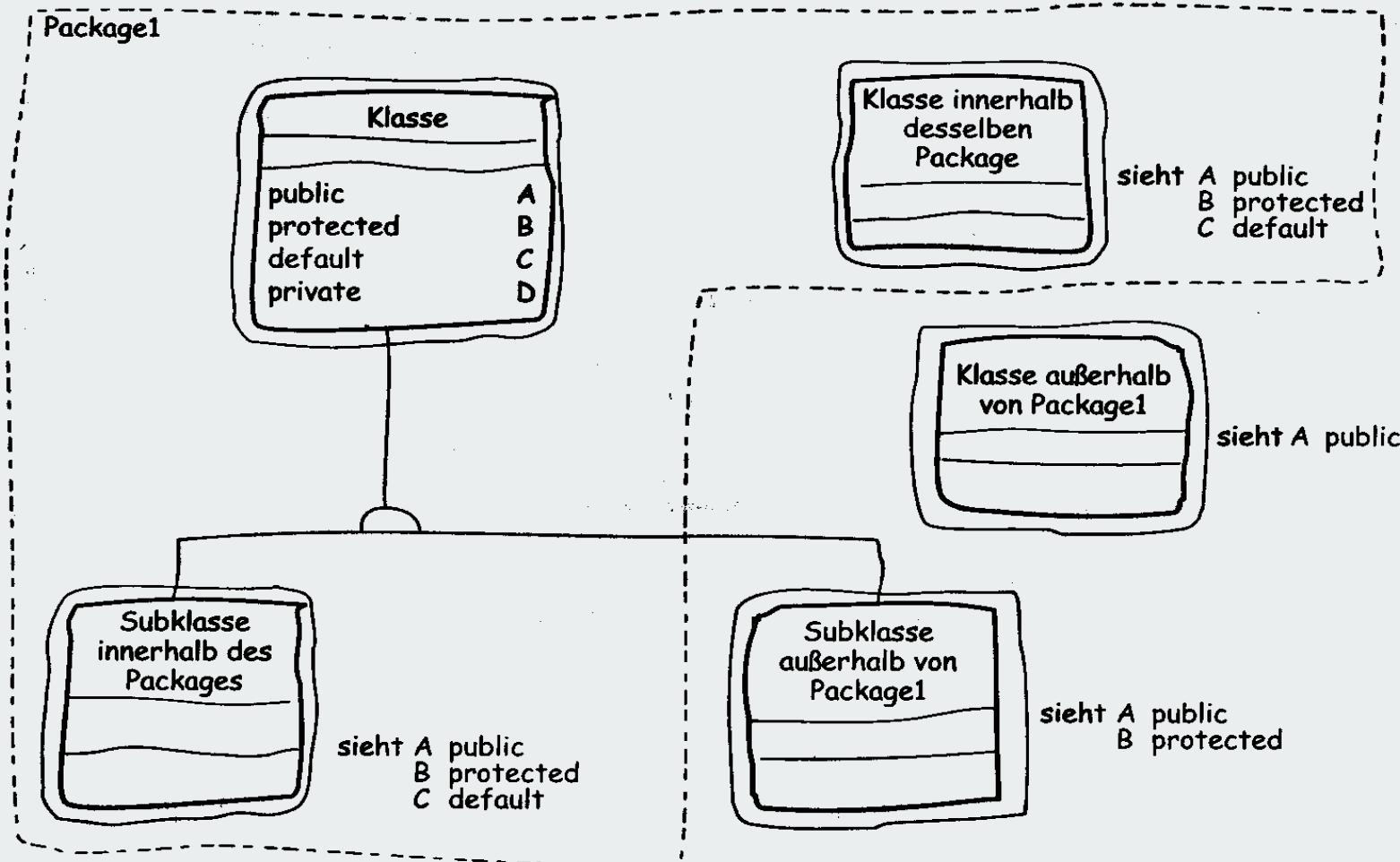
**Gutes Information
Hiding!**

Information Hiding (2/3)

- Sichtbarkeit in Java

Sichtbar für:	public	protected	(default)	private
Gleiche Klasse	ja	Ja	ja	ja
Andere Klasse, gleiches Paket	ja	ja	ja	nein
Andere Klasse, anderes Paket	ja	nein	nein	nein
Unterklasse, gleiches Paket	ja	ja	ja	nein
Unterklasse, anderes Paket	ja	ja	nein	nein

Information Hiding (3/3)



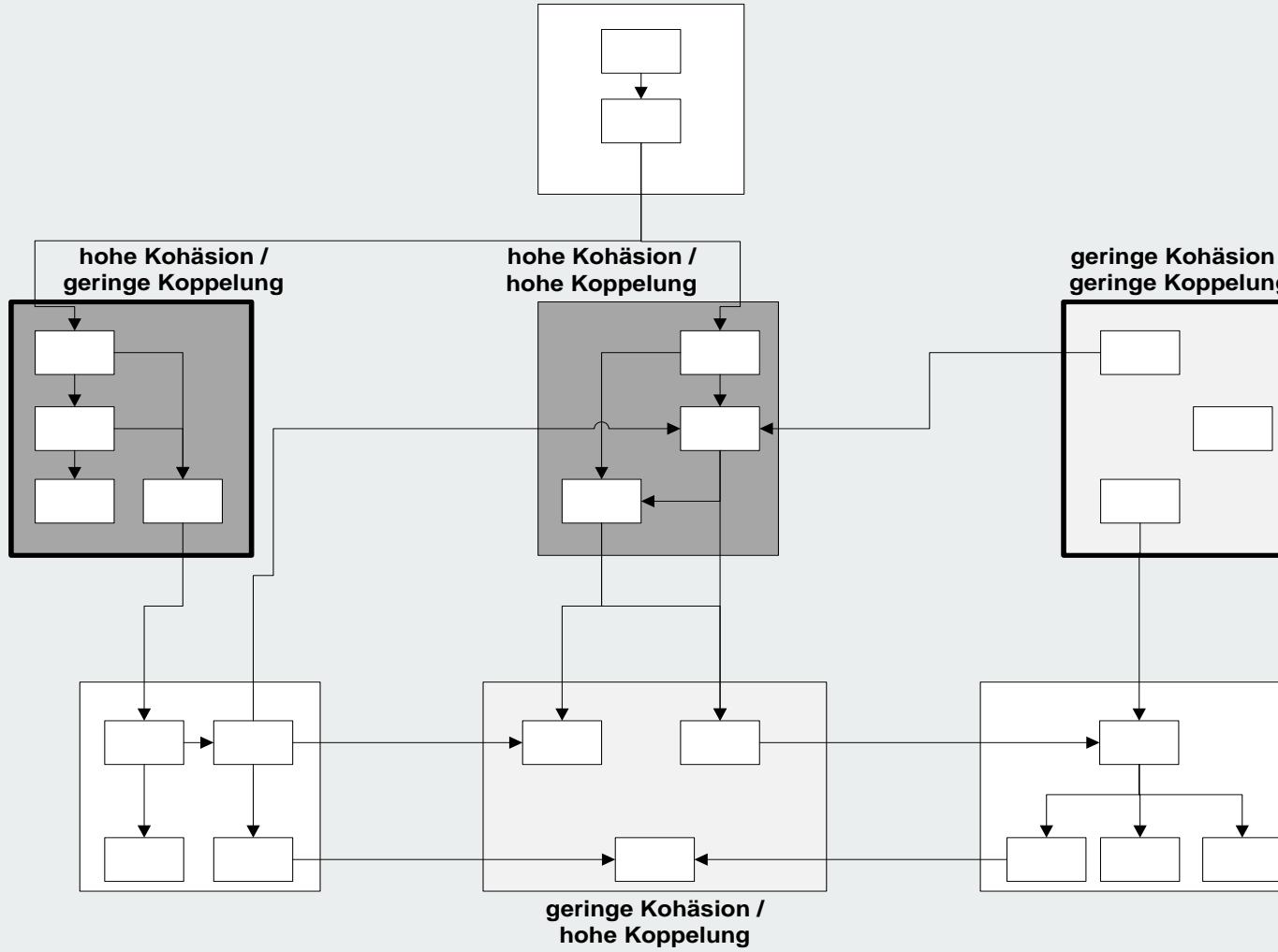
Kohäsion

- Maß für die **Zusammengehörigkeit** der Bestandteile einer Komponente.
- Hohe Kohäsion einer Komponente erleichtert Verständnis, Wartung und Anpassung.
- Grade der Kohäsion
 - Funktionale Kohäsion (*functional cohesion*)
 - Aufruf-Kohäsion (*sequential cohesion*)
 - Datenkohäsion (*communicational cohesion*)
 - Ablaufkohäsion (*procedural cohesion*)
 - Zeitkohäsion (*temporal cohesion*)
 - Logische Kohäsion (*logical cohesion*)
 - Zufällige Kohäsion (*coincidental cohesion*)
- Objektorientierte Klassen-Kohäsion
 - Keine Partitionierung in Untergruppen von zusammengehörigen Operationen und Attributen.

Kopplung

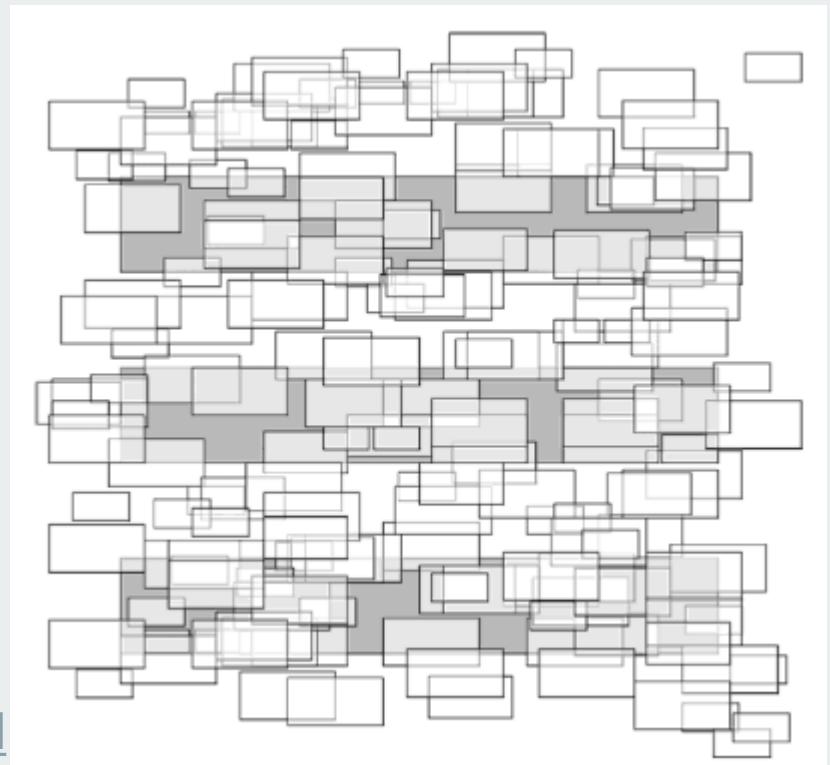
- Maß für die **Abhängigkeiten** zwischen Komponenten.
- Geringe Kopplung erleichtert die Wartbarkeit und macht das System stabiler.
- Folgen hoher Kopplung
 - schwer zu warten, schwer anpassbar
 - monolithisch
 - Austausch von Komponenten kaum möglich
 - System wird „zerbrechlich“
 - Erschwert die Wiederverwendung einzelner Komponenten
- Arten der Kopplung
 - Datenkopplung
 - Schnittstellenkopplung
 - Strukturkopplung
- Reduktion der Kopplung
 - Kopplung kann nie auf Null reduziert werden!
 - Höhere Schnittstellenkopplung erhöht Flexibilität
- Hohe Kohäsion ermöglicht geringe Kopplung

Kohäsion - Kopplung



Big Ball of Mud

- Programm, das keine erkennbare Softwarearchitektur besitzt
 - „großer Matschklumpen“
- Anti-Pattern der Softwarearchitektur
- häufigsten Ursachen sind:
 - ungenügende Erfahrung
 - fehlendes Bewusstsein für Softwarearchitektur
 - nicht modularisiert
 - Entspricht nicht den Prinzipien eines guten Entwurfs
 - Druck auf die Umsetzungsmannschaft
 - Ständig wechselnde Anforderungen
 - Hoher Zeitdruck
 - Geringes Budget
 - Fluktuation der Mitarbeiter
- https://de.wikipedia.org/wiki/Big_Ball_of_Mud



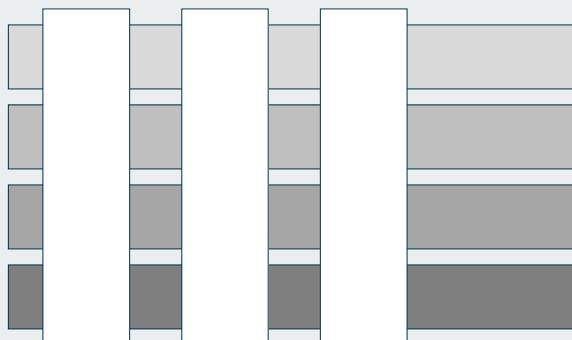
Anwendung von Schichtenarchitekturen

33

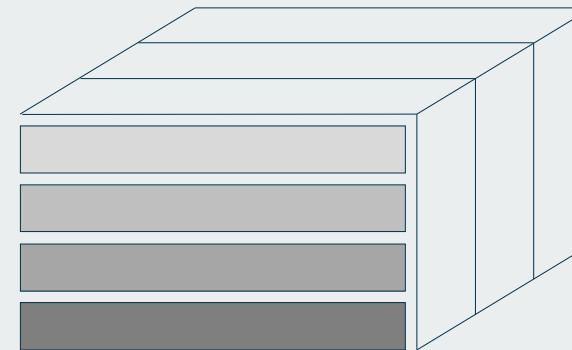
Schichtenmodell



Vertikale Schichten z. B. für Produktlinien



Prozess- und Library-Schichtenmodelle

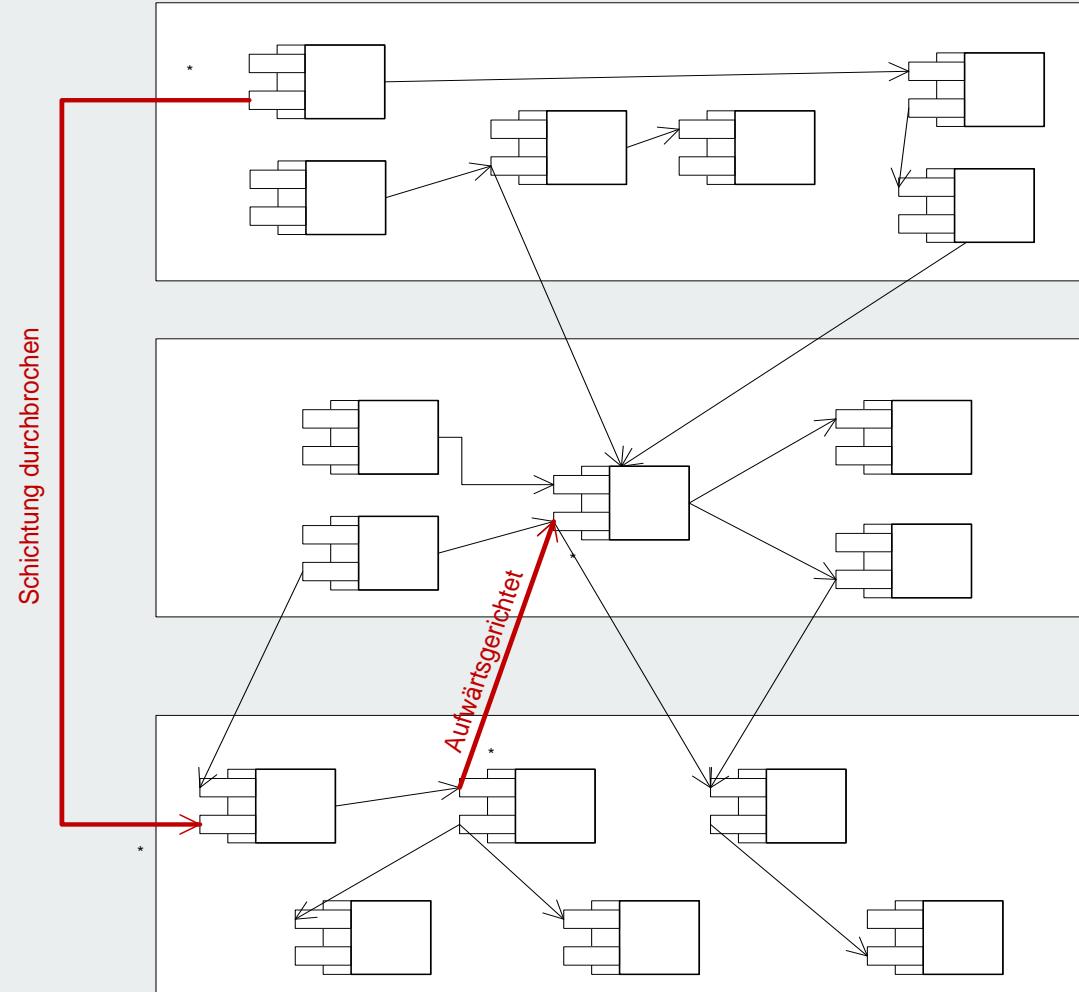


Verfeinern einer Schicht



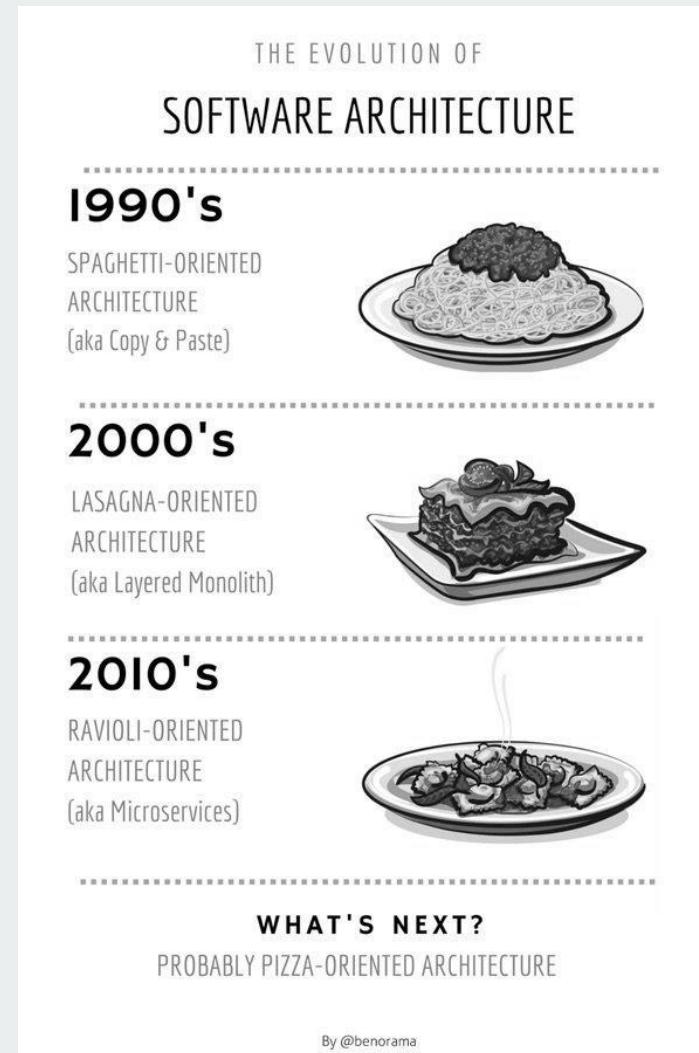
Schichtenarchitekturverletzung

34



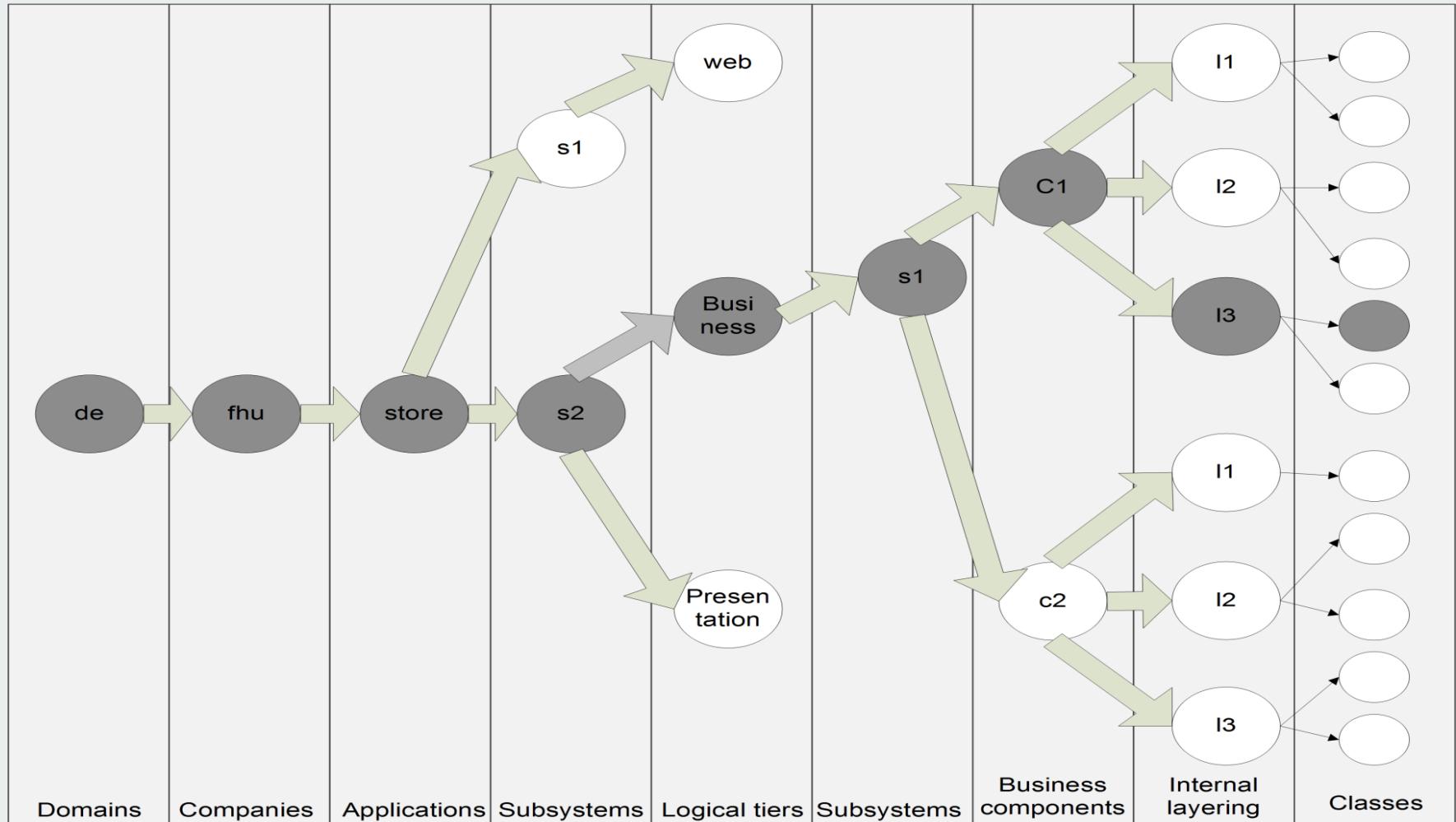
Evolution von Softwarearchitektur

35



Architekturbaum

36



Quellen

- Bien, Adam: *Enterprise Architekturen*, Entwickler.Press, 2006
- <http://www-st.inf.tu-dresden.de/Lehre/WS00-01/st2/>
- http://swt-www.informatik.uni-hamburg.de/attachments/LVTermine/STE06-07_VL-10_Softwareentwurf-Architektur.pdf/
- http://swt-www.informatik.uni-hamburg.de/attachments/LVTermine/STE06-07_VL-16_SWT-Qualitätssicherung.pdf/

Java Best Practices

Java Best Practices - Kommunikation durch Code (1/3)

- Anwendung der Java Code Conventions
- Fehlinformationen vermeiden

```
Set hobbyList;
```



```
Set hobbies;
```



- Zweckbeschreibende Namen wählen

```
int d; //elapsed time in days
```



```
int elapsedTimeInDays;
int fileAgeInDays;
int daysSinceLastModification;
```



Java Best Practices - Kommunikation durch Code (2/3)

```
public List<int[]> getThem() {  
    List<int[]> list1 = new ArrayList<int[]>();  
    for (int[] x : theList)  
        if (x[0] == 4)  
            list1.add(x);  
    return list1;  
}
```



```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells = new ArrayList<int[]>();  
    for (int[] cell : gameBoard)  
        if (cell[STATUS_VALUE] == FLAGGED)  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```



```
public List<Cell> getFlaggedCells() {  
    List<Cell> flaggedCells = new ArrayList<Cell>();  
    for (Cell cell : gameBoard)  
        if (cell.isFlagged())  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```



Java Best Practices - Kommunikation durch Code (3/3)

- Unterschiede deutlich machen

```
public void copy(String s1, String s2);
```



```
public void copy(String destination, String source);
```



- Aussprechbare Name verwenden

```
class DtaRcrd102 {  
    private Date genymdhms;  
    private Date modymdhms;  
    private final String pszqint = „102“;  
}
```



```
class Customer {  
    private static final String RECORD_ID = „102“;  
    private Date generationTimestamp;  
    private Date modificationTimestamp;  
}
```



Java Best Practices - Gruppierung durch Zeilenumbruch (1/2)

```
public class DistanceUnits {  
    enum DistanceUnit { MILES, KILOMETERS}  
    private static final double MILE_IN_KILOMETERS = 1.60934d;  
    private static final double IDENTITY = 1;  
    private static final double KILOMETER_IN_MILES = 1 / MILE_IN_KILOMETERS;  
  
    private DistanceUnits() {  
    }  
    public static final double conversionFactor(DistanceUnit from, DistanceUnit to) {  
        Objects.requireNonNull(from);  
        Objects.requireNonNull(to);  
        if (from == to) {  
            return IDENTITY;  
        }  
        if (from == DistanceUnit.MILES && to == DistanceUnit.KILOMETERS) {  
            return MILE_IN_KILOMETERS;  
        }  
        return KILOMETER_IN_MILES;  
    }  
}
```



Java Best Practices - Gruppierung durch Zeilenumbruch (2/2)

```
public class DistanceUnits {  
  
    enum DistanceUnit { MILES, KILOMETERS }  
  
    private static final double MILE_IN_KILOMETERS = 1.60934d;  
    private static final double KILOMETER_IN_MILES = 1 / MILE_IN_KILOMETERS;  
  
    private static final double IDENTITY = 1;  
  
    private DistanceUnits() {  
    }  
  
    public static final double conversionFactor(DistanceUnit from, DistanceUnit to) {  
        Objects.requireNonNull(from);  
        Objects.requireNonNull(to);  
  
        if (from == to) {  
            return IDENTITY;  
        }  
  
        if (from == DistanceUnit.MILES && to == DistanceUnit.KILOMETERS) {  
            return MILE_IN_KILOMETERS;  
        }  
  
        return KILOMETER_IN_MILES;  
    }  
}
```



Java Best Practices - Umwandlung von Kommentaren in Code (1/2)

- „Kommentieren Sie schlechten Code nicht – schreiben Sie ihn um“
- Kommentare sind kein Ersatz für schlechten Code
- Erklären Sie im und durch den Code

```
class InchToPointConvertor {  
  
    // convert the quantity in inches to points  
    static float parseInch(float inch) {  
        return inch * 72; // one inch contains 72 points  
    }  
}
```



```
class InchToPointConvertor {  
  
    private final static int POINTS_PER_INCH = 72;  
  
    static float convertToPoints(float inch) {  
        return inch * POINTS_PER_INCH;  
    }  
}
```



```
// Check to see if the employee is eligible for full benefits  
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
```



```
if (employee.isEligibleForFullBenefits())
```



Java Best Practices - Umwandlung von Kommentaren in Code (2/2)

```
class Account {  
    ...  
    //check if the password is complex enough, i.e., contains letter and digit/symbol  
    boolean isComplexPassword(String password) {  
        boolean dg_sym_found = false; //found a digit or symbol?  
        boolean letter_found = false; //found a letter?  
  
        for (int i = 0; i < password.length(); i++) {  
            char c = password.charAt(i);  
            if (Character.isLowerCase(c) || Character.isUpperCase(c))  
                letter_found = true;  
            else  
                dg_sym_found = true;  
        }  
        return (letter_found) && (dg_sym_found);  
    }  
}
```



```
class Account {  
    ...  
    boolean isComplexPassword(String password) {  
        return containsLetter(password) && (containsDigit(password) || containsSymbol(password));  
    }  
  
    boolean containsLetter(String password) { ... }  
  
    boolean containsDigit(String password) { ... }  
  
    boolean containsSymbol(String password) { ... }  
}
```



Java Best Practices - Schlechte Kommentare (1/7)

■ Redundante Kommentare

```
// Utility method that returns when this.closed is true. Throws an exception
// if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis) throws Exception {
    if (!closed) {
        wait(timeoutMillis);
        if (!closed)
            throw new Exception("MockResponseSender could not be closed");
    }
}
```



- Den Kommentar zu lesen, dauert wahrscheinlich länger, als den Code selbst zu lesen.
- Der Kommentar ist sicher nicht informativer als der Code.
- Weder rechtfertigt er den Code noch nennt er seinen Zweck oder Existenzgrund.
- Der Kommentar ist weniger präzise als der Code und verführt den Leser, diesen Mangel an Präzision zu akzeptieren.

Java Best Practices - Schlechte Kommentare (2/7)

■ Irreführende Kommentare

```
// Utility method that returns when this.closed is true. Throws an exception
// if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis) throws Exception {
    if (!closed) {
        wait(timeoutMillis);
        if (!closed)
            throw new Exception("MockResponseSender could not be closed");
    }
}
```



- Dieser Kommentar ist redundant und irreführend!
- Diese Methode kehrt nicht zurück, wenn `this.closed` den Wert `true` annimmt. Sie kehrt zurück, falls `this.closed` den Wert `true` hat; andernfalls wartet sie auf einen Timeout und löst dann eine Ausnahme aus, falls `this.closed` immer noch nicht den Wert `true` hat.
- Dieser Kommentar könnte einen armen Programmierer veranlassen, die Funktion unbekümmert aufzurufen und zu erwarten, dass sie zurückkehrt, sobald `this.closed` den Wert `true` annimmt.

Java Best Practices - Schlechte Kommentare (3/7)

- Vorgeschriebene Kommentare

```
/**  
 * Returns the day of the month.  
 *  
 * @return the day of the month.  
 */  
public int getDayOfMonth() {  
    return this.dayOfMonth;  
}
```



```
/**  
 * @param title The title of the CD  
 * @param author The author of the CD  
 * @param tracks The number of tracks on the CD  
 * @param durationInMinutes The duration of the CD in minutes  
 */  
public void addCD(String title, String author, int tracks, int durationInMinutes)
```



- Dieser Wust von Kommentar liefert keine zusätzlichen Informationen, macht den Code nur unklarer und bringt das Potenzial für Irreführungen.
- Es ist schlecht, per Regel vorzuschreiben, dass jede Funktion einer Javadoc oder jede Variable einen Kommentar haben müsste.
- Derartige Kommentare machen den Code nur unübersichtlicher und führen zu einer allgemeinen Verwirrung und Unordnung.

Java Best Practices - Schlechte Kommentare (4/7)

■ Tagebuch-Kommentare

```
* Changes (from 11-Oct-2001)
* -----
* 11-Oct-2001 : Re-organised the class and moved it to new package
*           com.jrefinery.date (DG);
* 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
*           class (DG);
* 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
*           class is gone (DG); Changed getPreviousDayOfWeek(),
*           getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
*           bugs (DG);
* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
* 29-May-2002 : Moved the month constants into a separate interface
*           (MonthConstants) (DG);
* 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
* 13-Mar-2003 : Implemented Serializable (DG);
* 29-May-2003 : Fixed bug in addMonths method (DG);
* 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG);
* 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
```



- Vor langer Zeit gab es einen Grund für die Erstellung und Pflege solcher Protokolleinträge.
- Heutzutage gibt es dafür Versionsverwaltungssysteme.
- Diese Kommentare sind Stördaten, die Unübersichtlichkeit verbreiten und sollten entfernt werden.

Java Best Practices - Schlechte Kommentare (5/7)

■ Geschwätz

```
/**  
 * Default constructor.  
 */  
protected AnnualDateRule() {  
}
```



```
/** The day of the month. */  
private int dayOfMonth;
```



```
private void startSending() {  
    try {  
        doSending();  
    } catch (SocketException e) {  
        // normal. someone stopped the request. (GOOD COMMENT)  
    } catch (Exception e) {  
        try {  
            response.add(ErrorResponder.makeExceptionString(e));  
            response.closeAll();  
        } catch (Exception e1) {  
            //Give me a break!  
        }  
    }  
}
```



- Diese Kommentare sind so geschwätzig, dass wir lernen, sie zu ignorieren.
- Wenn wir den Code lesen, springen unsere Augen einfach über sie hinweg.
- Irgendwann fangen die Kommentare an zu lügen, wenn sich der umliegende Code ändert.

Java Best Practices - Schlechte Kommentare (6/7)

- Positionsbezeichner

```
// Methods /////////////////////////////////
```



- Zuschreibungen und Nebenbemerkungen

```
// Added by Rick
```



- Im Versionsverwaltungssystem ist es ersichtlich!
- Auskommentierter Code

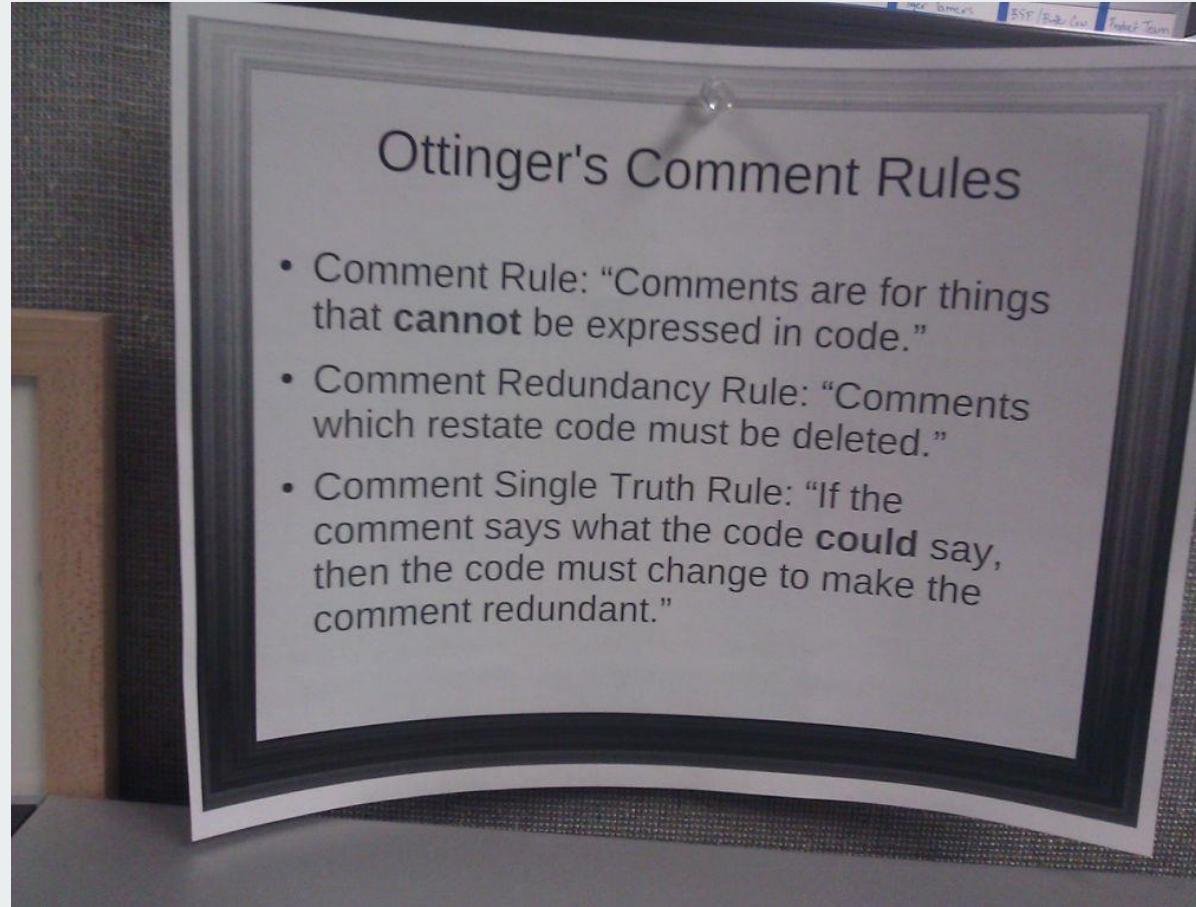
```
...
this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;
writeHeader();
writeResolution();
//dataPos = bytePos;
...
```



- Andere, die diesen auskommentierten Code sehen, werden nicht den Mut aufbringen, ihn zu löschen.
- Sie werden glauben, dass es einen Grund gibt, warum der Code dort steht, und das er zu wichtig ist, um gelöscht zu werden.

Java Best Practices - Schlechte Kommentare (7/7)

52



- <https://agileinaflash.blogspot.com/2009/04/rules-for-commenting.html>

Java Best Practices - Instanziierung vermeiden

53

```
public final class UtilityClass {  
    public static void foo() {  
    }  
  
    public static void bar() {  
    }  
}
```



```
public final class UtilityClass {  
  
    //Noninstantiable utility class  
    private UtilityClass() {  
    }  
  
    public static void foo() {  
    }  
  
    public static void bar() {  
    }  
}
```



- java.lang.Math
- java.util.Arrays
- java.util.Collections

Java Best Practices - Unveränderliche Objekte (1/4)

54

- Unveränderbarkeit spielt immer dann eine Rolle, wenn Objekte gemeinsam verwendet werden.
- Voraussetzungen, um den synchronisierten Zugriff zu vermeiden
 - Der Object state darf sich nach der Erzeugung nicht mehr ändern.
 - Alle Felder (die Klasse selbst) sind `final` (und `private`)
- keine Setter, weil die Felder als `final` deklariert sind
- Immutable Objects sind immer thread safe
- Ein unveränderliches Objekt kann gefahrlos an nicht vertrauenswürdigen Code übergeben werden.
- Technik:
 - Erzeugen neuer Instanzen anstelle der Veränderung des Object-State .

Java Best Practices - Unveränderliche Objekte (2/4)

■ final Variablen

- Primitive Typen
 - Der Inhalt einer Variablen, die als final deklariert ist, kann nicht geändert werden.
 - Bei Variablen vom primitivem Typ bedeutet es, dass sich der Wert der Variablen nach der Initialisierung nicht mehr ändert.

```
final int max = 256;  
max = 0; // error: does not compile
```

■ Referenzvariablen

- Bei Referenzvariablen bedeutet es, dass sich die in der Referenzvariablen gespeicherte Adresse nicht mehr ändert.
- Eine final Referenzvariable verweist auf das Objekt, das ihr bei der Initialisierung zugewiesen wurde und kann niemals auf ein anderes Objekt verweisen.
- Es bedeutet aber nicht, dass das referenzierte Objekt vor Veränderungen geschützt ist.

```
final Date deadline = new Date();  
deadline = new Date(100,0,1,0,0,0); // error: does not compile  
deadline.set(new Date(100,0,1,0,0,0)); // fine: compiles
```

Java Best Practices - Unveränderliche Objekte (3/4)

```
public final class ImmutableRGB {  
    private final int red, green, blue;  
  
    public ImmutableRGB(int red, int green, int blue) {  
        this.red = red;  
        this.green = green;  
        this.blue = blue;  
    }  
    public ImmutableRGB invert() {  
        return new ImmutableRGB(255 - this.red, 255 - this.green, 255 - this.blue);  
    }  
}
```



```
public final class Stamp {  
    private final Date date;  
    private final String author;  
  
    public Stamp(Date date, String author) {  
        this.date = (Date) date.clone();  
        this.author = author;  
    }  
  
    public Date getDate() {  
        return (Date) this.date.clone();  
    }  
  
    public String getAuthor() {  
        return this.author;  
    }  
}
```



Java Best Practices - Unveränderliche Objekte (4/4)

```
public static final String[] VALUES = {"Potential", "security", "hole!"};
```



```
private static final String[] PRIVATE_VALUES = {"Not", "potential", "security", "hole!"};  
public static final List<String> VALUES = Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```



```
private static final String[] PRIVATE_VALUES = {"Not", "potential", "security", "hole!"};  
  
public static final String[] values() {  
    return PRIVATE_VALUES.clone();  
}
```



```
// Returns an immutable list containing one element.  
public static final List<String> VALUES = List.of("Not", "potential", "security", "hole!"); // Java 9
```



Java Best Practices - public constants (1/3)

```
public final class Constants {  
    private Constants() {}  
  
    public static final String CRLF = "\r\n"; // Windows  
}
```



```
public class Records {  
    public void write(Writer out) {  
        for (Record record: this.records) {  
            out.write(record.data());  
            out.write(Constants.CRLF);  
        }  
    }  
}
```



```
public class Rows {  
    public void print(PrintStream ps) {  
        for (Row row: this.rows) {  
            ps.print(row.cells());  
            ps.print(Constants.CRLF);  
        }  
    }  
}
```



Java Best Practices - public constants (2/3)

```
public class LineEnding {  
    private final String origin;  
  
    public LineEnding(String origin) {  
        this.origin = origin;  
    }  
  
    @Override  
    public String toString() {  
        return String.format("%s\r\n", this.origin); //Windows  
    }  
}
```



```
public class Records {  
    public void write(Writer out) {  
        for (Record record: this.records) {  
            out.write(new LineEnding(record.data()));  
        }  
    }  
}
```



```
public class Rows {  
    public void print(PrintStream ps) {  
        for (Row row: this.rows) {  
            ps.print(new LineEnding(row.cells()));  
        }  
    }  
}
```



Java Best Practices - public constants (3/3)

- Verhalten nun änderbar:

```
public class LineEnding {  
    private static final String WINDOWS_LINE_ENDING = "\r\n";  
    private static final String UNIX_LINE_ENDING = "\n";  
  
    private final String origin;  
  
    public LineEnding(String origin) {  
        this.origin = origin;  
    }  
  
    @Override  
    public String toString() {  
        if (isUnix()) {  
            return String.format("%s%s", this.origin, UNIX_LINE_ENDING);  
        }  
  
        return String.format("%s%s", this.origin, WINDOWS_LINE_ENDING)  
    }  
}
```



Java Best Practices – Datenkapselung vs. Vererbung (1/2)

```
public class Document {  
    public int length() {  
        return this.content().length();  
    }  
  
    public byte[] content() {  
        //Loads the raw content of the document as an array of bytes  
    }  
}
```



```
public EncryptedDocument extends Document {  
  
    @Override  
    public byte[] content() {  
    }  
}
```



- `length()`
- EncryptedDocument verwendet die `length()` Methode von Document, ist das so beabsichtigt?

Java Best Practices – Datenkapselung vs. Vererbung (2/2)

```
interface Document {  
    int length();  
    byte[] content();  
}
```



```
public final DefaultDocument implements Document {  
  
    @Override  
    public int length() { /* Same code */ }  
  
    @Override  
    public byte[] content() { /* Same code */ }  
}
```



```
public final EncryptedDocument implements Document {  
    private final Document plain;  
  
    public EncryptedDocument(Document plain) {  
        this.plain = plain;  
    }  
  
    @Override  
    public int length() {  
        return this.plain.length();  
    }  
  
    @Override  
    public byte[] content() {  
        byte [] raw = this.plain.content();  
        return /* Decrypt the raw content */  
    }  
}
```



Java Best Practices - @Override Annotation

63

```
public class User {  
    private final String name;  
  
    public User(String name) {  
        this.name = name;  
    }  
  
    public int hashCode() {  
    }  
  
    public boolean equals(Object obj) {  
    }  
}
```



```
public class User {  
    private final String name;  
  
    public User(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public int hashCode() {  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
    }  
}
```



Java Best Practices - @FunctionalInterface Annotation

64

```
@FunctionalInterface  
public interface Runnable {  
  
    * When an object implementing interface <code>Runnable</code> is used  
    * to create a thread, starting the thread causes the object's  
    * <code>run</code> method to be called in that separately executing  
    * thread.  
    * <p>  
    * The general contract of the method <code>run</code> is that it may  
    * take any action whatsoever.  
    *  
    * @see     java.lang.Thread#run()  
    */  
    public abstract void run();  
}
```



Java Best Practices - Null-Object-Muster (1/2)

- Vermeidung von null-Rückgaben
- Null-Objekte sind unveränderlich (immutable)
- Für Datenstrukturen der Collection-API gibt es bereits Implementierungen:
 - `Collections.EMPTY_LIST`, `EMPTY_MAP`, `EMPTY_SET`
 - `Collections.emptyList()`, `emptyMap()`, `emptySet()`

```
public List<Book> getBooks(String author) {  
    if (author != null && !author.isEmpty()) {  
        return null;  
    }  
    ...  
}
```



```
public List<Book> getBooks(String author) {  
    if (author != null && !author.isEmpty()) {  
        return Collections.emptyList();  
    }  
    ...  
}
```



Java Best Practices - Null-Object-Muster (2/2)

```
public class Action {  
    private Session session;  
    public boolean isVisible() {  
        User user = session.getUser();  
  
        return user != null && user.isSuperUser();  
    }  
}
```



```
public class Action {  
    private Session session;  
    public boolean isVisible() {  
        return session.getUser().isSuperUser();  
    }  
}
```



```
public class NullUser extends User {  
    @Override  
    public boolean isSuperUser () {  
        return false;  
    }  
}
```



```
public class Session {  
    private User user = new NullUser();  
    public User getUser() {  
        return user;  
    }  
}
```



Java Best Practices - null als Methodenparameter

```
// Looks like a convenient alternatives to these two methods
// public Iterable<File> findAll()
// public Iterable<File> find(String fileExtension)

public Iterable<File> find(String fileExtension) {
    if (fileExtension == null) {
        // find all files
    } else {
        // find files by file extension
    }
}
```



```
interface FileFilter {
    boolean accept(File file)
}
```



```
public class AllFileFilter implements FileFilter {
    @Override
    public boolean accept(File file) {
        return true;
    }
}
```



```
public Iterable<File> find(FileFilter filter) {
    // find all files by filter
    if (filter.accept(file)) {
        foundFiles.add(file);
    }
}
```



Java Best Practices - Enhanced Loops

```
for (Iterator i = c.iterator(); i.hasNext();) {  
    doSomething((Element) i.next());  
}
```



```
for (int i = 0; i < a.length; i++) {  
    doSomething(a[i]);  
}
```



```
for (Element e : elements) {  
    doSomething(e);  
}
```



```
elements.forEach(e -> doSomething(e)); //Java 8 Lambdas
```



Java Best Practices - Interface / Implementierung

```
// Bad - uses class as type  
ArrayList<Subscriber> subscribers = new ArrayList<Subscriber>();
```



```
// Good - uses Interface as type  
List<Subscriber> subscribers = new ArrayList<Subscriber>();
```



Java Best Practices - Vorhandene Exceptions

70

- Die Java-API bietet eine große Anzahl von Exception-Klassen, so muss nicht für jeden Fall eine eigene Exception-Klasse deklariert werden.

IllegalArgumentException	Non-null parameter value is inappropriate
IllegalStateException	Object state is inappropriate for method invocation
NullPointerException	Parameter value is null where prohibited
IndexOutOfBoundsException	Index parameter value is out of range
ConcurrentModificationException	Concurrent modification of an object has been detected where it is prohibited
UnsupportedOperationException	Object does not support method

Java Best Practices - Parametervalidierung

```
public void myMethod(String str, int index, Object[] arr) {  
    if (str == null) {  
        throw new IllegalArgumentException("str cannot be null");  
    }  
    if (index >= arr.length || index < 0) {  
        throw new IllegalArgumentException("index exceeds bounds of array");  
    }  
    ...  
}
```



- `Objects.requireNonNull(obj);`
- `Objects.requireNonNull(obj, message);`
- `Objects.requireNonNull(obj, messageSupplier);`
- `Objects.requireNonNullElse(obj, defaultObj)`
- `Objects.requireNonNullElseGet(obj, supplier);`
- `Objects.checkIndex(index, length);`
- `Objects.checkFromIndexSize(fromIndex, size, length);`
- `Objects.checkFromToIndex(fromIndex, toIndex, length);`

Java Best Practices - Vergleich von Strings

72

```
private static final String COMPARE_VALUE = "Bad programmer";

public boolean compareIt(String input) {
    if (input.equals(COMPARE_VALUE)) {
        return true;
    } else {
        return false;
    }
}
```



```
private static final String COMPARE_VALUE = "Rockstar programmer";

public boolean compareIt(String input) {
    return COMPARE_VALUE.equals(input);
}
```



Java Best Practices - Klasse für Konstanten

73

```
public final class Constants {  
    private Constants() {  
    }  
  
    public static final String NEW_SEPARATOR = System.getProperty("line.separator");  
    public static final String FILE_SEPARATOR = System.getProperty("file.separator");  
}
```



Java Best Practices - Casting long to int

74

```
long foo = 10L;  
int bar = (int) foo;
```



```
long foo = 10L;  
int bar = Math.toIntExact(foo);
```



- Math.incrementExact(long)
- Math.subtractExact(long, long)
- Math.decrementExact(long)
- Math.negateExact(long)
- Math.subtractExact(int, int)

Java Best Practices - Generics

- Sicherstellung der Typsicherheit
 - Fehler beim Einstellen eines Elements von einem falschen Typ
 - Typ eines Elements beim Herausnehmen bekannt
- Vermeidung von ClassCastException
- Wegfall von Casts macht den Code übersichtlicher
- Selbstdokumentation des Codes

```
List data = new ArrayList();
```



```
List<String> data = new ArrayList<String>();
```



```
List list = new ArrayList();
list.add(new Integer(0));
Integer i = (Integer) list.get(0); // possible ClassCastException at runtime
```



```
List<Integer> list = new ArrayList<Integer>();
list.add(new Integer(0));
Integer i = list.get(0);
```



Java Best Practices - Enums (1/2)

76

```
public static final int APPLE_FUJI = 0;
public static final int APPLE_PIPPIN = 1;
public static final int APPLE_GRANNY_SMITH = 2;

public static final int ORANGE_NAVEL = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD = 2;
```



```
public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
public enum Orange { NAVEL, TEMPLE, BLOOD }
```



Java Best Practices - Enums (2/2)

```
public enum Ensemble {  
    SOLO, DUET, TRIO, QUARTET, QUINTET, SEXTET, SEPTET, OCTET, NONET, DECTET;  
  
    public int numberOfMusicians() {  
        return ordinal() + 1;  
    }  
}
```



```
public enum Ensemble {  
    SOLO(1), DUET(2), TRIO(3), QUARTET(4),  
    QUINTET(5), SEXTET(6), SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),  
    NONET(9), DECTET(10), TRIPLE_QUARTET(12);  
  
    private final int numberOfMusicians;  
  
    Ensemble(int size) {  
        this.numberOfMusicians = size;  
    }  
  
    public int numberOfMusicians() {  
        return this.numberOfMusicians;  
    }  
}
```

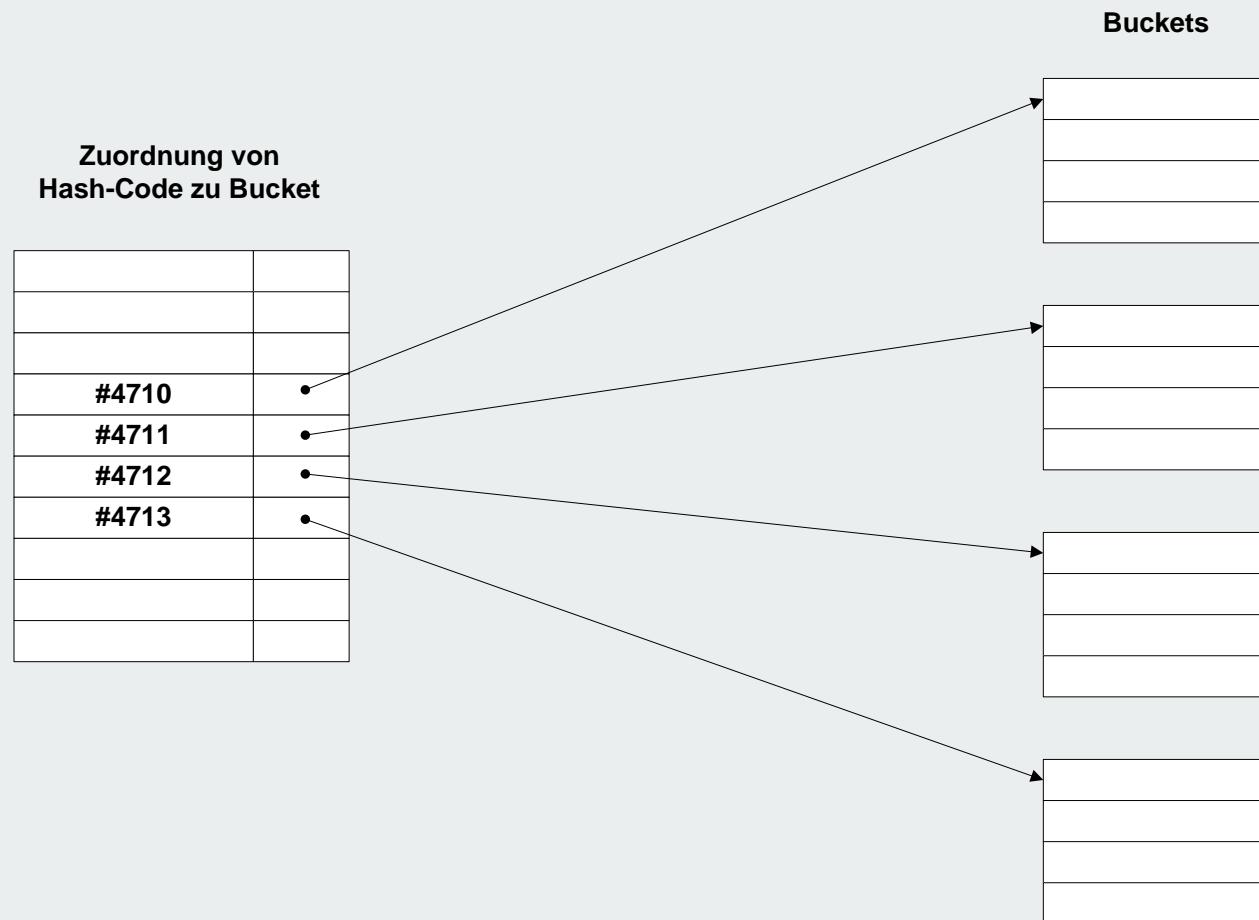


Java Best Practices - equals / hashCode (1/5)

- Hash-basierte Container in Java
 - Ein Hash-basierte Container ist so organisiert, dass er verschiedene Buckets anlegt, in denen die zu speichernden Objekte sequentiell abgelegt werden.
 - Einen Bucket kann man sich als ein Array oder Liste von Objekt-Referenzen vorstellen.
 - Der Zugriff auf die verschiedenen Buckets erfolgt über einen integralen Index und ist damit hochperformant; er erfolgt in konstanter Zeit, d. h. der Zugriff auf den Bucket dauert immer gleich lang unabhängig von der Zahl der Elemente im Container.
 - Innerhalb eines Buckets ist der Zugriff auf die Elemente allerdings langsam, weil er sequentiell erfolgt. Die Abhängigkeit ist hier linear, d. h. es dauert um so länger, je größer der Bucket ist. Deshalb ist ein hash-basierter Container mit vielen kleinen Buckets günstiger als einer mit wenigen großen Buckets.
 - `hashCode()` berechnet zu dem Objekt, auf dem sie gerufen wird, einen Hash-Code.
 - Der Hash-Code ist ein integraler Wert, der verwendet wird, um Objekte in einem hash-basierten Container abzulegen oder sie in einem solchen Container zu finden.
 - Die hash-basierten Container in Java sind `java.util.Hashtable`, `java.util.HashMap`, `java.util.HashSet` und deren Subklassen.

Java Best Practices - equals / hashCode (2/5)

- Aufbau eines Hash-basierte Containers



Java Best Practices - equals / hashCode (3/5)

- Implementierung von `equals()` / `hashCode()`
 - Die Default-Implementierung von `hashCode()` muss überschreiben werden, wenn man dasselbe für `equals()` tut.
 - Konsistenz zu `equals()`
 - Gleiche Objekte müssen gleiche Hash-Codes haben, sonst ist das Arbeiten mit hash-basierten Containern nicht möglich.
 - Performanz
 - Der Zugriff auf Elemente im einem hash-basierten Container geschieht über eine rasche Identifikation des Bucket mittels Index (= Hash-Code) gefolgt von der relativ langsamen sequentiellen Suche innerhalb des hoffentlich kleinen Bucket. Der Vorteil der hash-basierten Container ist daher der schnelle Zugriff auf den Bucket mittels Index (= Hash-Code). Wenn nun die Berechnung des Hash-Codes lange dauert, dann ist der Performance-Gewinn durch den schnellen Zugriff per Hash-Code zunichte gemacht. Deshalb sollten `HashCode`-Berechnungen möglichst effizient sein.
 - Die Hash-Code-Berechnung soll also nicht nur schnell sein, sondern auch zu einem Container mit vielen kleinen Buckets führen.
 - Ziel ist eine möglichst performante Implementierung, die eine möglichst gleichmäßige Verteilung der berechneten Hash-Codes im Intervall der möglichen Integer-Werte von -2147483648 bis 2147483647 erreicht.

Java Best Practices - equals / hashCode (4/5)

```
public class Name {  
    private final String name;  
  
    public Name(String first) {  
        this.name = first;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if (this == obj)  
            return true;  
        if (obj == null)  
            return false;  
        if (getClass() != obj.getClass())  
            return false;  
        Name other = (Name) obj;  
        if (this.name == null) {  
            if (other.name != null)  
                return false;  
        } else if (!this.name.equals(other.name))  
            return false;  
        return true;  
    }  
}
```



```
public static void main(String[] args) {  
    Set<Name> s = new HashSet<>();  
    s.add(new Name("Donald Duck"));  
    System.out.println(s.contains(new Name("Donald Duck"))); //prints false!!!  
}
```

Java Best Practices - equals / hashCode (5/5)

```
public class Name {  
    private final String name;  
  
    public Name(String first) {  
        this.name = first;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if (this == obj)  
            return true;  
        if (obj == null)  
            return false;  
        if (getClass() != obj.getClass())  
            return false;  
        Name other = (Name) obj;  
        if (this.name == null) {  
            if (other.name != null)  
                return false;  
        } else if (!this.name.equals(other.name))  
            return false;  
        return true;  
    }  
  
    @Override  
    public int hashCode() {  
        return 31 + ((this.name == null) ? 0 : this.name.hashCode());  
    }  
}
```



Java Best Practices - Performance

```
for (int i = 0; i < expensiveComputation(); i++) {  
    doSomething(i);  
}
```



```
for (int i = 0, n = expensiveComputation(); i < n; i++) {  
    doSomething(i);  
}
```



Java Best Practices - Performance

```
public String foo() {  
    String result = "";  
  
    for (int i = 0; i < numItems(); i++) {  
        result += lineForItem(i); // String concatenation  
    }  
  
    return result;  
}
```



```
public String foo() {  
    StringBuilder sb = new StringBuilder(numItems() * LINE_WIDTH);  
  
    for (int i = 0, n = numItems(); i < n; i++) {  
        sb.append(lineForItem(i));  
    }  
  
    return sb.toString();  
}
```



Java Best Practices - Performance

```
User user;

if (this.users.get(key) != null) {
    user = this.users.get(key);
    user.doSomething();
    ...
}
```



```
User user;

if ((user = this.users.get(key)) != null) {
    user.doSomething();
    ...
}
```



Java Best Practices - Performance

```
//slow instantiation  
String slow = new String("Yet another string object");
```



```
//fast instantiation  
String fast = "Yet another string object";
```



Java Best Practices - Performance

```
public class Person {  
    private final Date birthDate;  
  
    public Person(Date birthDate) {  
        this.birthDate = birthDate;  
    }  
  
    public boolean isBabyBoomer() {  
        Calendar gmtCal =  
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));  
  
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);  
        Date boomStart = gmtCal.getTime();  
  
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);  
        Date boomEnd = gmtCal.getTime();  
  
        return this.birthDate.compareTo(boomStart) >= 0  
            && this.birthDate.compareTo(boomEnd) < 0;  
    }  
}
```



```
class Person {  
    private static final Date BOOM_START;  
    private static final Date BOOM_END;  
  
    static {  
        Calendar gmtCal =  
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));  
  
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);  
        BOOM_START = gmtCal.getTime();  
  
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);  
        BOOM_END = gmtCal.getTime();  
    }  
  
    private final Date birthDate;  
  
    public Person(Date birthDate) {  
        this.birthDate = birthDate;  
    }  
  
    public boolean isBabyBoomer() {  
        return this.birthDate.compareTo(BOOM_START) >= 0  
            && this.birthDate.compareTo(BOOM_END) < 0;  
    }  
}
```



Java Best Practices - Performance

```
new Boolean(true);  
new Long(1);  
new Integer(2);
```



```
Boolean.valueOf(true);  
Long.valueOf(1);  
Integer.valueOf(2);
```



```
// 4 objects in memory  
Boolean b1 = new Boolean(true);  
Boolean b2 = new Boolean(false);  
Boolean b3 = new Boolean(false);  
Boolean b4 = new Boolean(false);
```



```
// 2 objects in memory  
Boolean b1 = Boolean.TRUE;  
Boolean b2 = Boolean.FALSE;  
Boolean b3 = Boolean.FALSE;  
Boolean b4 = Boolean.FALSE;
```



Java Best Practices - Performance

```
Map<String, String> table = new HashMap<String, String>();  
  
table.put("FirstKey", "FirstValue");  
table.put("SecondKey", "SecondValue");  
  
for (String key : table.keySet()) {  
    System.out.println(table.get(key));  
}
```



```
Map<String, String> table = new HashMap<String, String>();  
  
table.put("FirstKey", "FirstValue");  
table.put("SecondKey", "SecondValue");  
  
for (Entry<String, String> entry: table.entrySet()) {  
    System.out.println(entry.getValue());  
}
```



Java Best Practices - Performance

90

```
if (text != null && !text.equals("")) {  
    // actions  
}
```



```
if (text != null && !text.isEmpty()) {  
    // actions  
}
```



Java Best Practices - Performance

```
Integer io = new Integer(0);
Object obj = (Object) io;

for (int i = 0; i < 100000; i++) {
    if (obj instanceof Integer) {
        byte x = ((Integer) obj).byteValue();
        double d = ((Integer) obj).doubleValue();
        float f = ((Integer) obj).floatValue();
    }
}
```



```
Integer io = new Integer(0);
Object obj = (Object) io;

for (int i = 0; i < 100000; i++) {
    if (obj instanceof Integer) {
        Integer icast = (Integer) obj;
        byte x = icast.byteValue();
        double d = icast.doubleValue();
        float f = icast.floatValue();
    }
}
```



Java Best Practices

$$e = mc^2$$

errors = (more code)²

Quellen

- Bloch, Joshua: *Effective Java – Second Edition*, Addison Wesley, 2008
- Robert, Martin: *Clean Code – Refactoring, Patterns, Testen und Techniken für sauberen Code*, mitp-Verlag, 2009
- <http://www.angelikalanger.com/Articles/EffectiveJava.html/>
- <http://www.tutego.de/blog/javainsel/2009/08/vorhandene-runtime-fehlertypen-kennen-und-nutzen/>

Werkzeuge zur Sicherung der Softwarequalität

Werkzeuge zur Sicherung der Softwarequalität

Eclipse

Eclipse - Java Compiler Errors / Warnings

96

The screenshot shows the Eclipse Preferences dialog with the title "Preferences" and the category "warnings". The left sidebar lists various compiler-related preferences under "Java", "Java Persistence", "JavaScript", "Plug-in Development", "Web", and "XML". Under "Java", the "Compiler" section has its "Errors/Warnings" tab selected. The main panel displays the "Errors/Warnings" configuration for Java compiler problems, divided into sections for "Code style" and "Potential programming problems". Each problem has a dropdown menu next to it where severity levels can be set. The "Configure Project Specific Settings..." link is visible at the top right.

Errors/Warnings

Select the severity level for the following optional Java compiler problems:

Code style

- Non-static access to static member: Warning
- Indirect access to static member: Warning
- Unqualified access to instance field: Warning
- Undocumented empty block: Ignore
- Access to a non-accessible member of an enclosing type: Warning
- Method with a constructor name: Warning
- Parameter assignment: Warning
- Non-externalized strings (missing/unused \$NON-NLS\$ tag): Ignore

Potential programming problems

- Serializable class without serialVersionUID: Warning
- Assignment has no effect (e.g. 'x = x'): Warning
- Possible accidental boolean assignment (e.g. 'if (a = b)'). Warning
- 'finally' does not complete normally: Warning
- Empty statement: Warning

[Configure Project Specific Settings...](#)

[Restore Defaults](#) [Apply](#)

Eclipse - HTML Validation

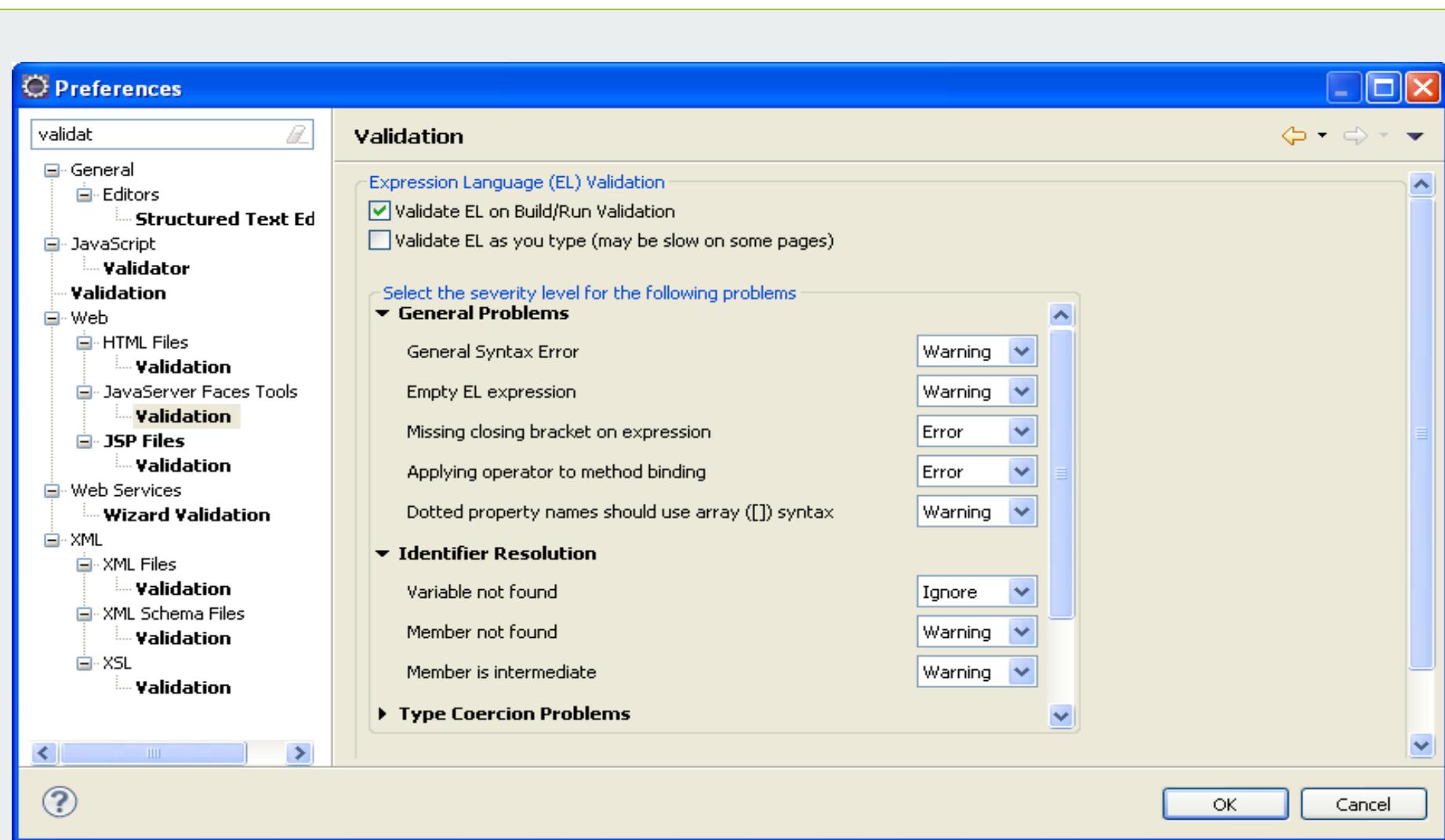
97

The screenshot shows the Eclipse Preferences dialog with the search bar set to "validat". The left sidebar lists various validation categories under the "Validation" section. The "HTML Files" category is currently selected, with its "Validation" sub-node highlighted. The main panel displays the "Validation" configuration for HTML files, specifically for "Elements". It lists ten validation problems with their corresponding severity levels:

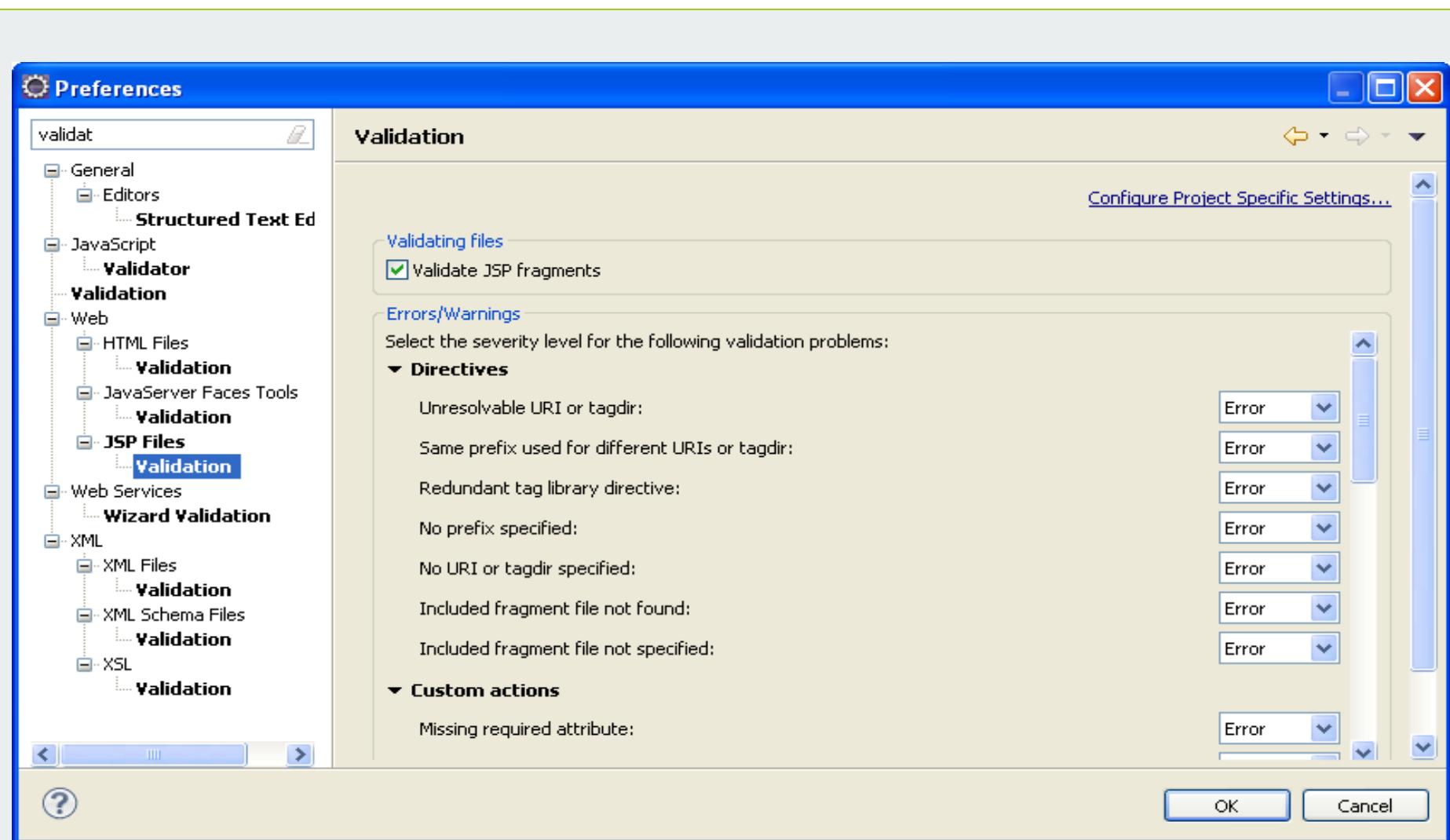
Validation Problem	Severity Level
Unknown tag name:	Warning
Invalid tag name:	Error
Start-tag uses invalid case:	Warning
End tag uses invalid case:	Error
Missing start tag:	Error
Missing end tag:	Warning
Unnecessary end tag:	Warning
Invalid directive:	Error
Invalid tag location:	Warning
Duplicate tag:	Warning
Coexistence:	Warning
Unclosed start tag:	Error

At the bottom right of the dialog are "OK" and "Cancel" buttons.

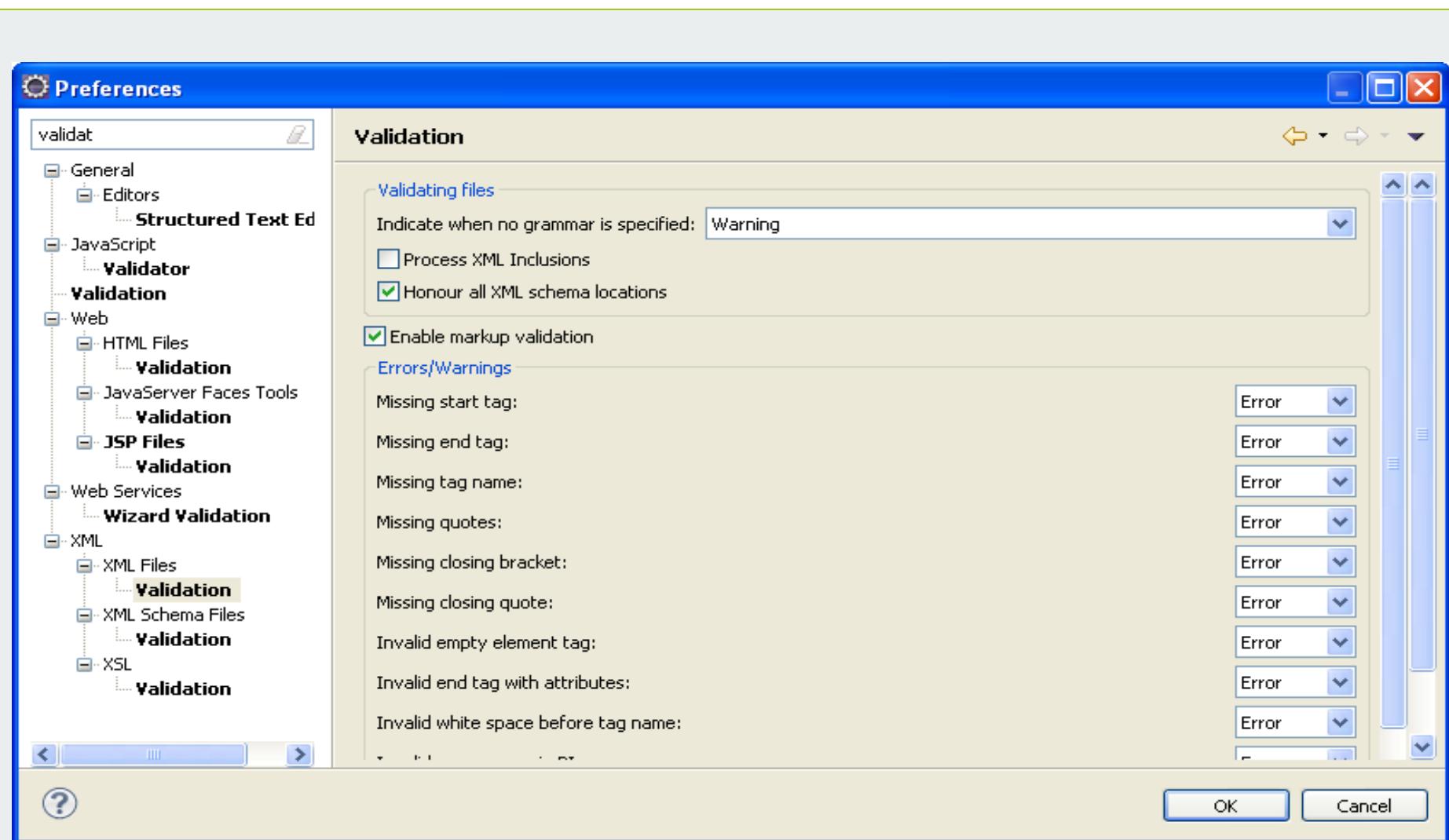
Eclipse - JSF Validation



Eclipse - JSP Validation

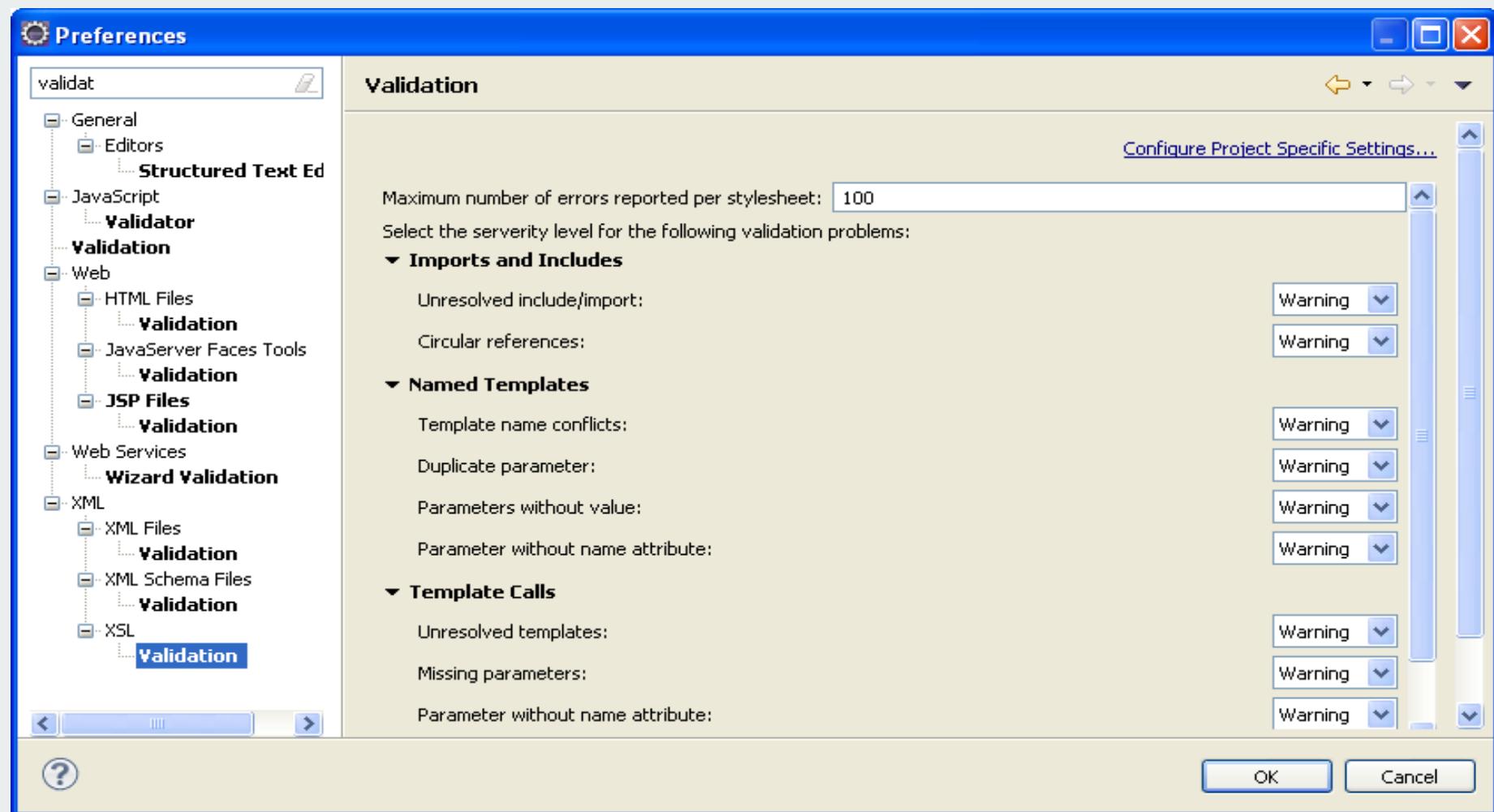


Eclipse - XML Validation

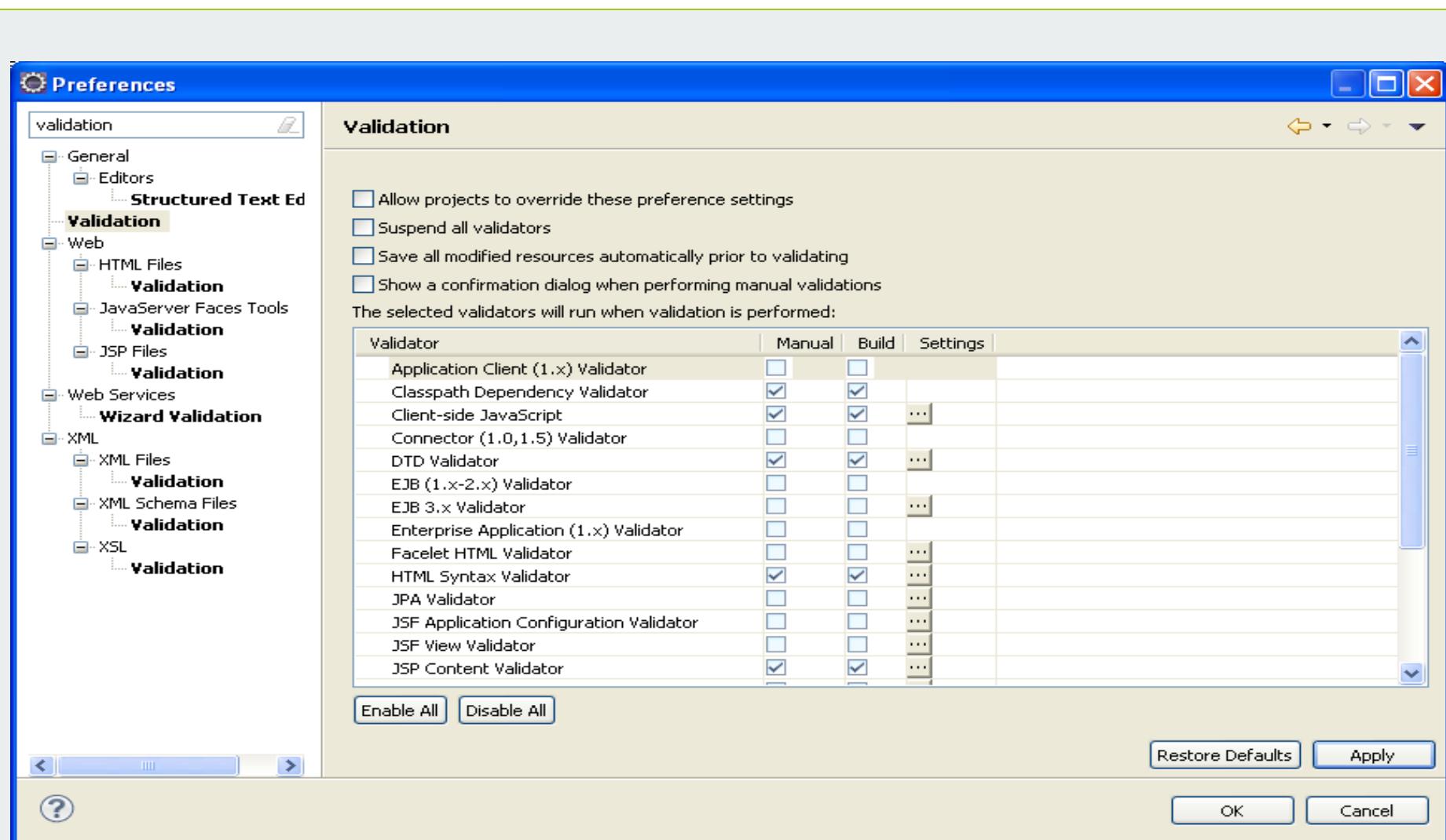


Eclipse - XSL Validation

101

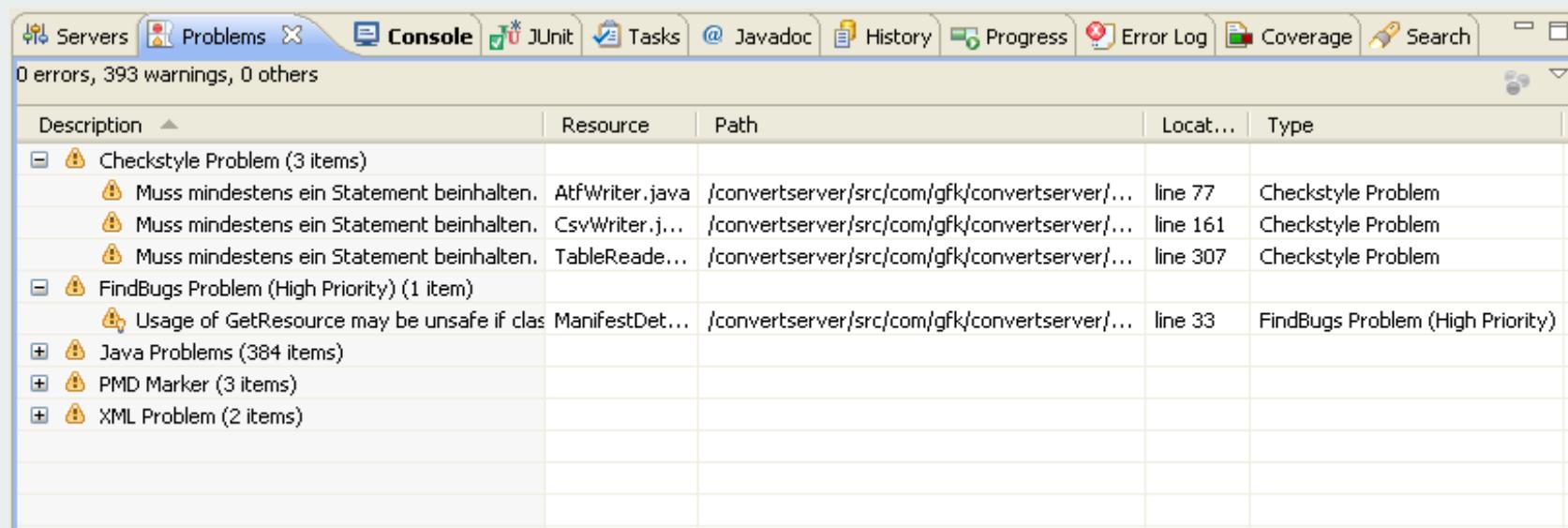


Eclipse - Errors / Warnings



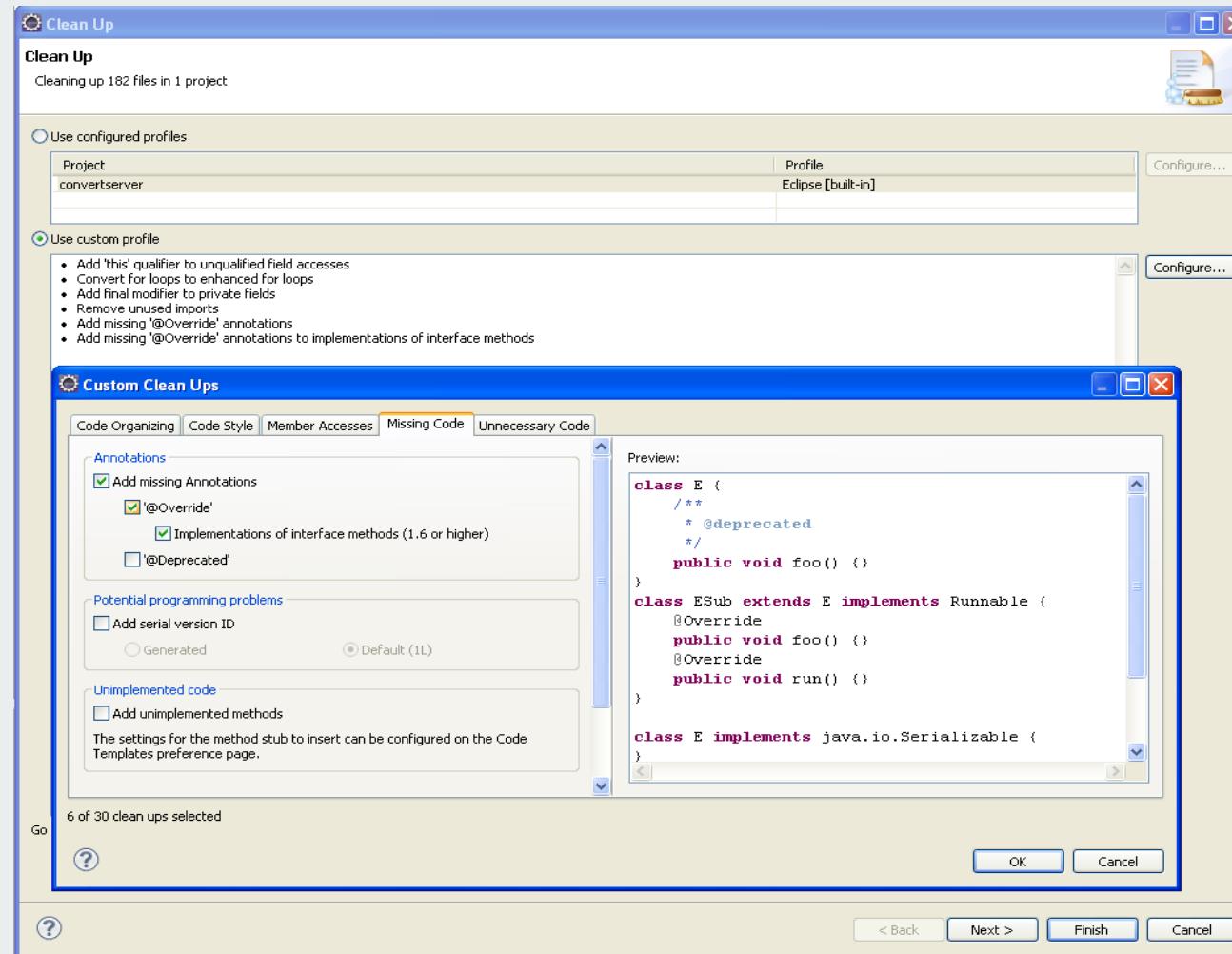
Eclipse - Errors / Warnings Ansicht

103



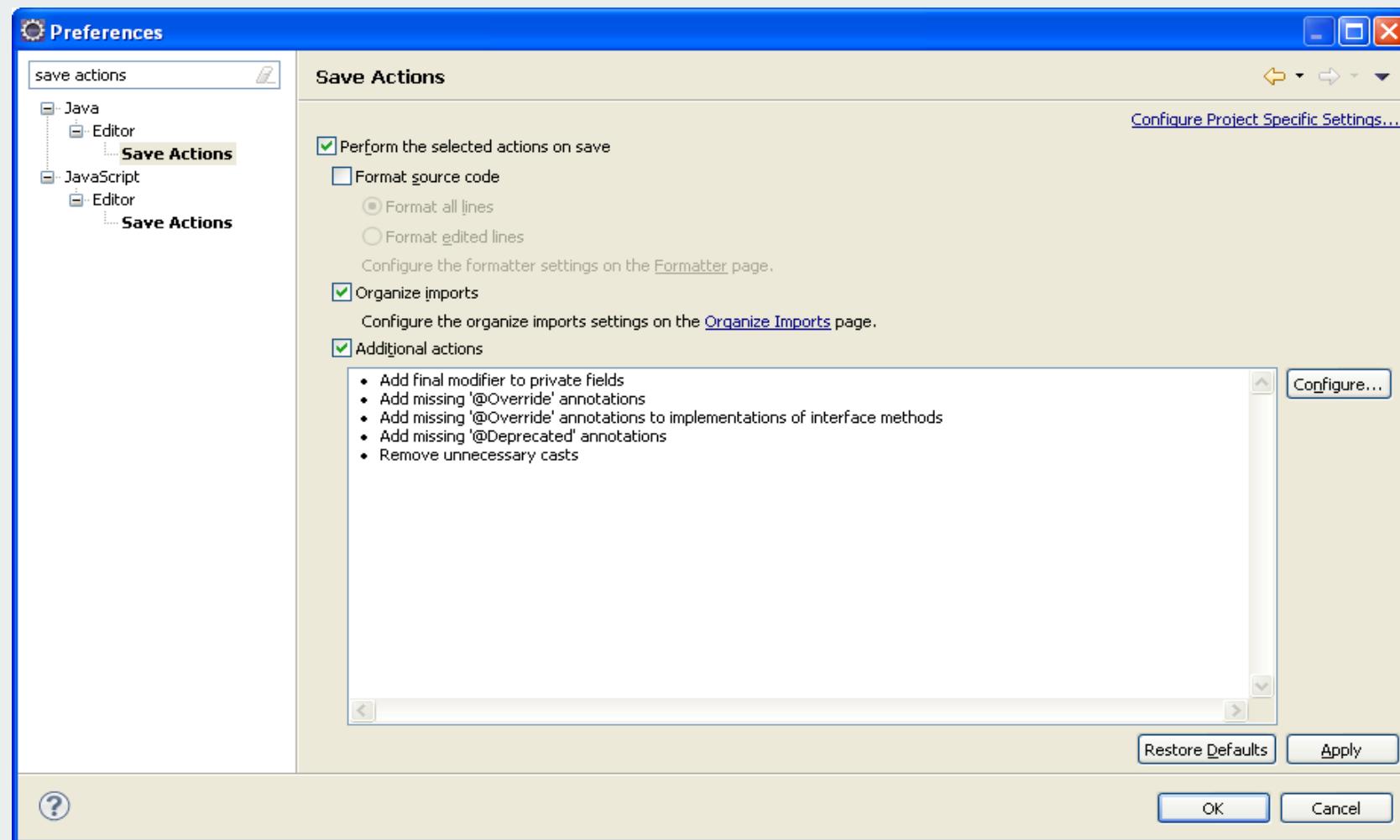
Eclipse - Source Clean Up

104



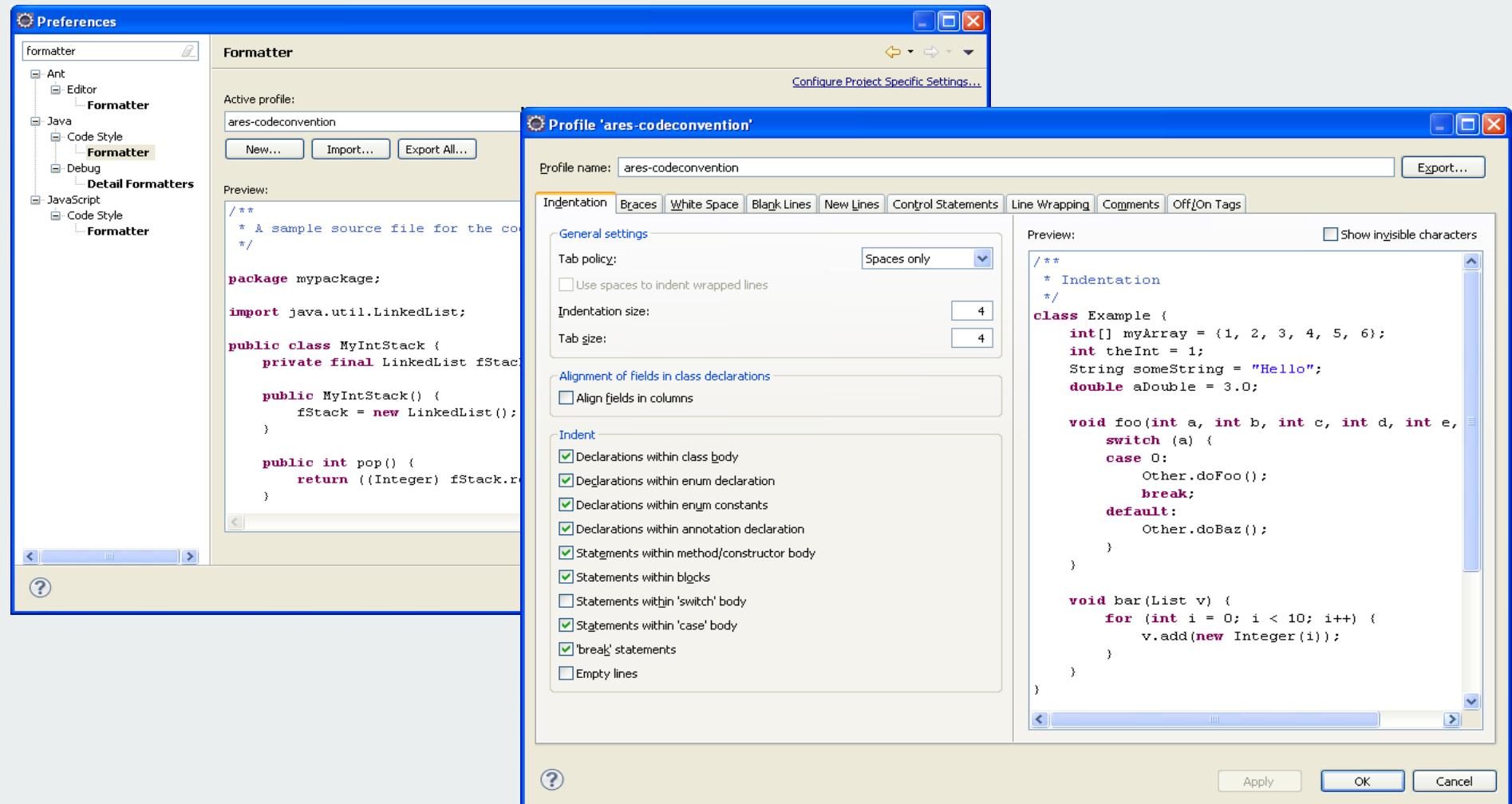
Eclipse - Save Actions

105

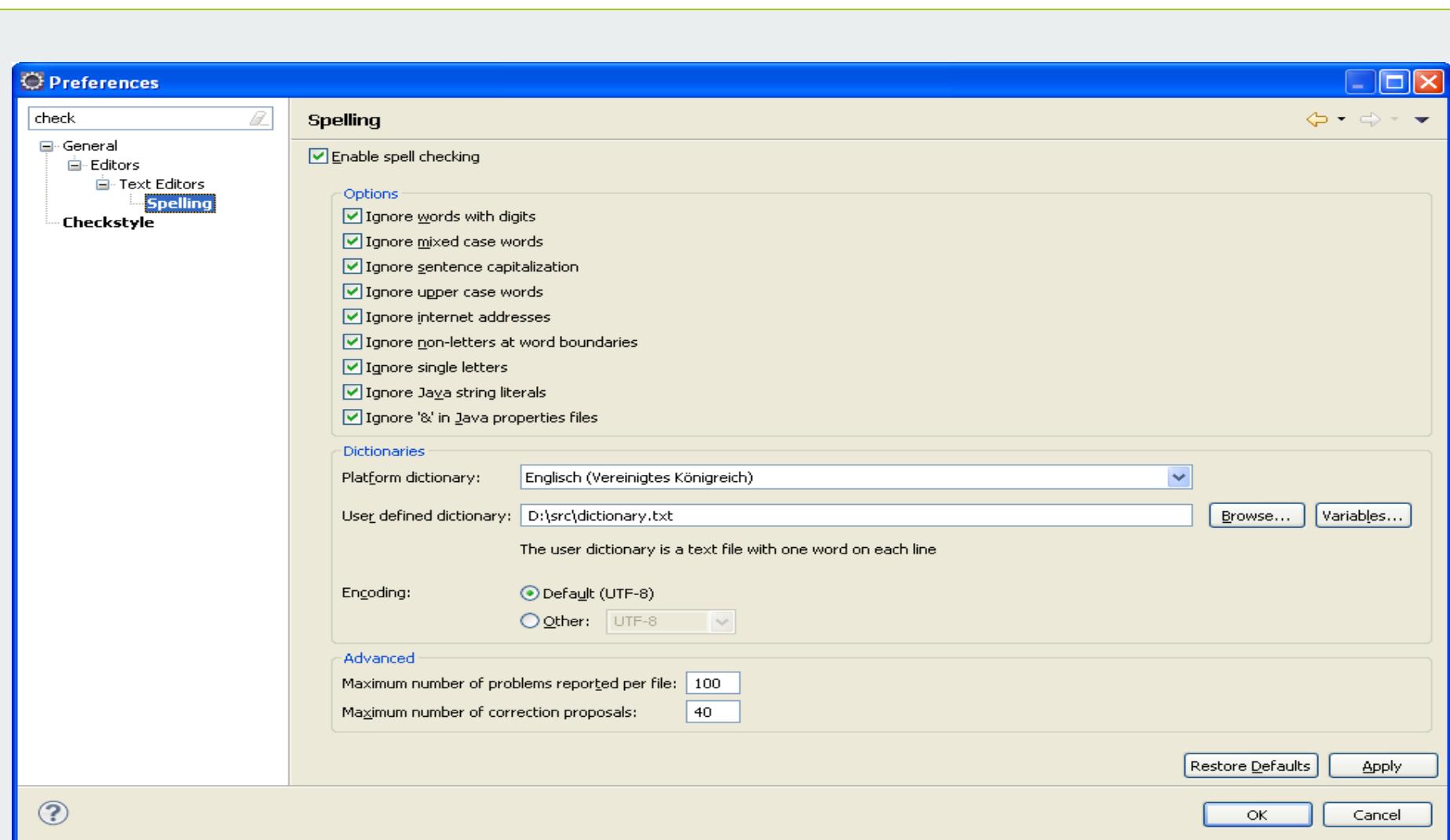


Eclipse - Source Code Formatter

106



Eclipse - Rechtschreibprüfung



Werkzeuge zur Sicherung der Softwarequalität

JUnit 5

Klassifikation nach Informationsstand

109

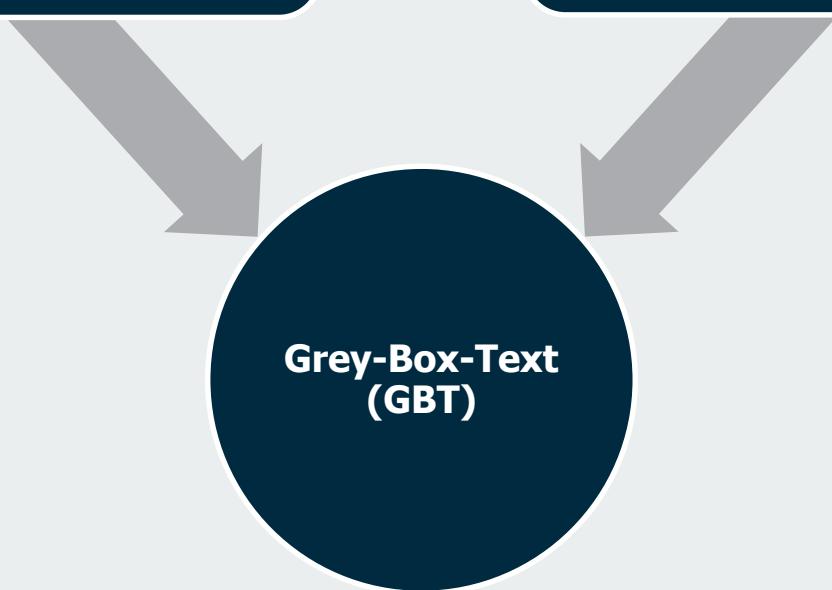
White-Box-Test (WBT)

- Code-nahe Tests
- Testet eher Struktur als Funktion
- Entwicklertest

Black-Box-Test (BBT)

- Entwicklung ohne Kenntnis des Codes
- Orientierung an Spezifikation bzw. Anforderungsdefinition
- funktionsorientiert
- kein Entwicklertest

Grey-Box-Text (GBT)



WBT - BBT im Vergleich

110

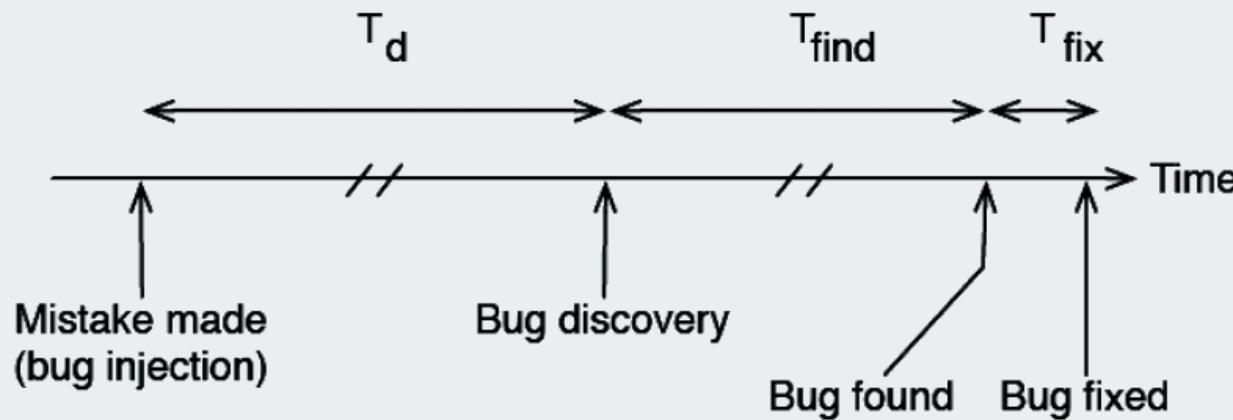
	WBT	BBT
Vorteile	<ul style="list-style-type: none">• Testen von Teilkomponenten und der internen Funktionsweise• Geringerer organisatorischer Aufwand• Automatisierung durch gute Tool-Unterstützung	<ul style="list-style-type: none">• bessere Verifikation des Gesamtsystems• Testen von semantischen Eigenschaften bei geeigneter Spezifikation• Portabilität von systematisch erstellten Testsequenzen auf plattformunabhängige Implementierungen
Nachteile	<ul style="list-style-type: none">• Erfüllung der Spezifikation nicht überprüft• Eventuelles Testen "um Fehler herum"	<ul style="list-style-type: none">• größerer organisatorischer Aufwand• zusätzlich eingefügte Funktionen bei der Implementierung werden nur durch Zufall getestet• Testsequenzen einer unzureichenden Spezifikation sind Unbrauchbar

- **GBT**
 - Vereint die Vorteile aus WBT und BBT
 - Unterstützt eine „testgetriebene Entwicklung“
 - Ohne Kenntnis der Implementierung entwickelt

Test Driven Development (TDD)

111

The Physics of Debug Later Programming (DLP)



- As T_d increases, T_{find} increases dramatically
- T_{fix} is usually short, but can increase with T_d

Test Driven Development (TDD)

- Was ist testgetriebene Entwicklung?
 - **Testgetriebene Programmierung**

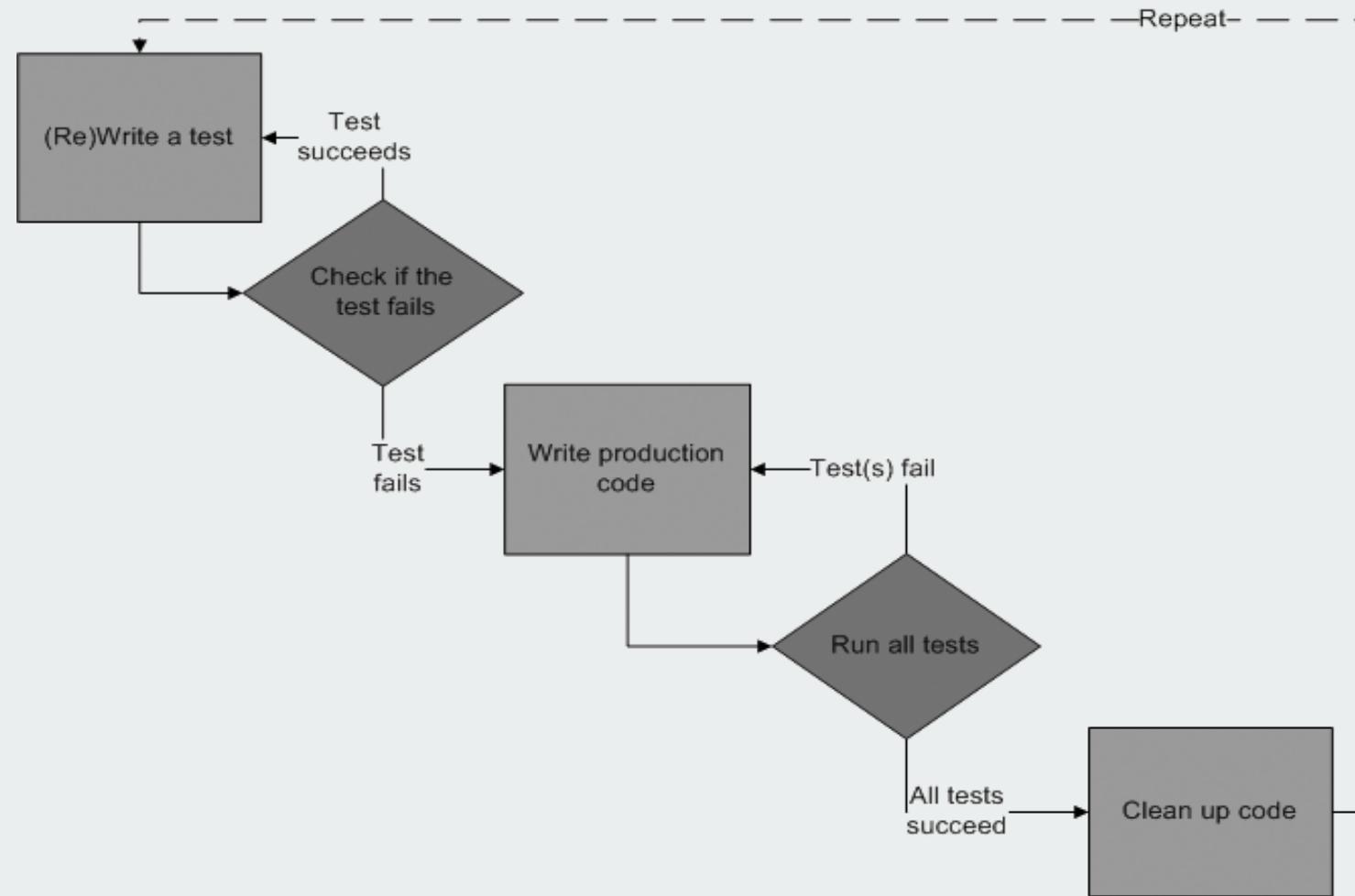
Motiviere jede Verhaltensänderung am Code durch einen automatisierten Test.
 - **Refactoring**

Bringe den Code ständig in die “einfache Form” und entwickle das Design Schritt für Schritt während der Programmierung.
 - **Häufige Integration**

Integriere den Code so häufig wie nötig.
- Warum testgetriebene Entwicklung?
 - Softwareentwicklung ohne Tests ist wie Klettern ohne Seil und Haken.
 - Tests sichern den Erhalt der vorhandenen Funktionen bei Erweiterung und Überarbeitung.
 - Refactoring verlängert die produktive Lebensdauer einer Software.
 - Code lässt sich im Nachhinein oft nur schlecht testen.
 - Test-First betont die Verwender-Sicht

Test Driven Development (TDD)

113



Unit Tests

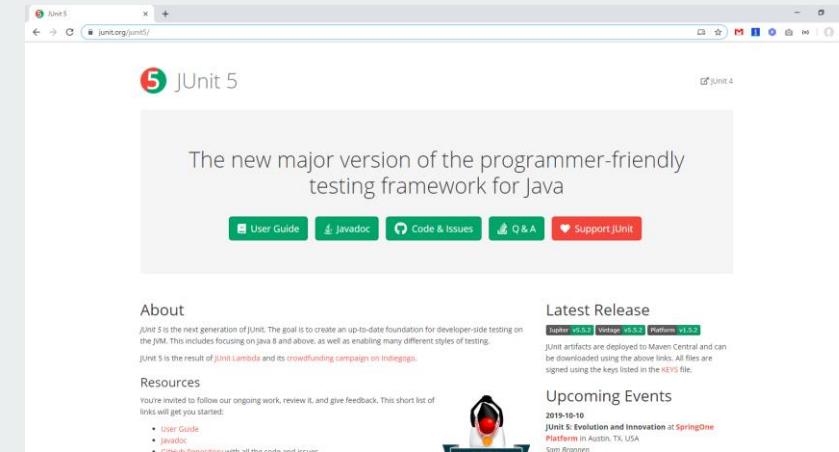
- Nutzen von Unit Tests
 - Änderungen am Code können sofort überprüft werden
 - Bessere Code-Qualität / -Stabilität bzw. Sicherheit
 - Weniger Zögern vor Änderungen an Kernkomponenten
 - Kürzere Entwicklungszeiten (trotz zusätzliche zu implementierenden Tests)
 - Einfacheres Refactoring
 - Vereinfachte Dokumentation der Implementierung
- Voraussetzungen für Unit Tests
 - Einfache Implementierung der Tests
 - Schnelle und automatisierte Ausführung der Tests
 - Eingängige Analyse der Testergebnisse

→ JUnit Testing Framework

JUnit 5

115

- Schlankes Java basiertes Unit-Testing Framework
- Autoren sind Kent Beck, Erich Gamma
- Version 3.x basiert auf Vererbung
- Seit JUnit 4 Annotation möglich (JDK 1.5)
- Seit JUnit 5 (JDK 8)
 - JUnit 5 ist die nächste Generation von JUnit.
 - Ziel ist es, eine aktuelle Grundlage für das Testen auf der JVM zu schaffen. Dazu gehört die Konzentration auf Java 8 und höher sowie die Möglichkeit, viele verschiedene Teststile zu ermöglichen.
- <https://junit.org/junit5/>



Ziele von JUnit

- Testerstellungsaufwand auf das Nötigste reduzieren
 - Leicht zu erlernen und benutzen
 - Redundante Arbeit vermeiden
 - Tests müssen
 - wiederholt anwendbar sein
 - separat erstellbar sein
 - inkrementell erstellbar sein
 - frei kombinierbar sein
 - auch von anderen als dem Autor durchführbar sein
 - auch von anderen als dem Autor auswertbar sein
 - Verwendung von Testdaten ermöglichen
 - Wiederverwendbarkeit von Testdaten
 - (Testdatenerstellung ist meist aufwendiger als der Test selbst)
- Erleichterung der Test-Erstellung, -Durchführung und -Auswertung

JUnit - Einfaches Beispiel

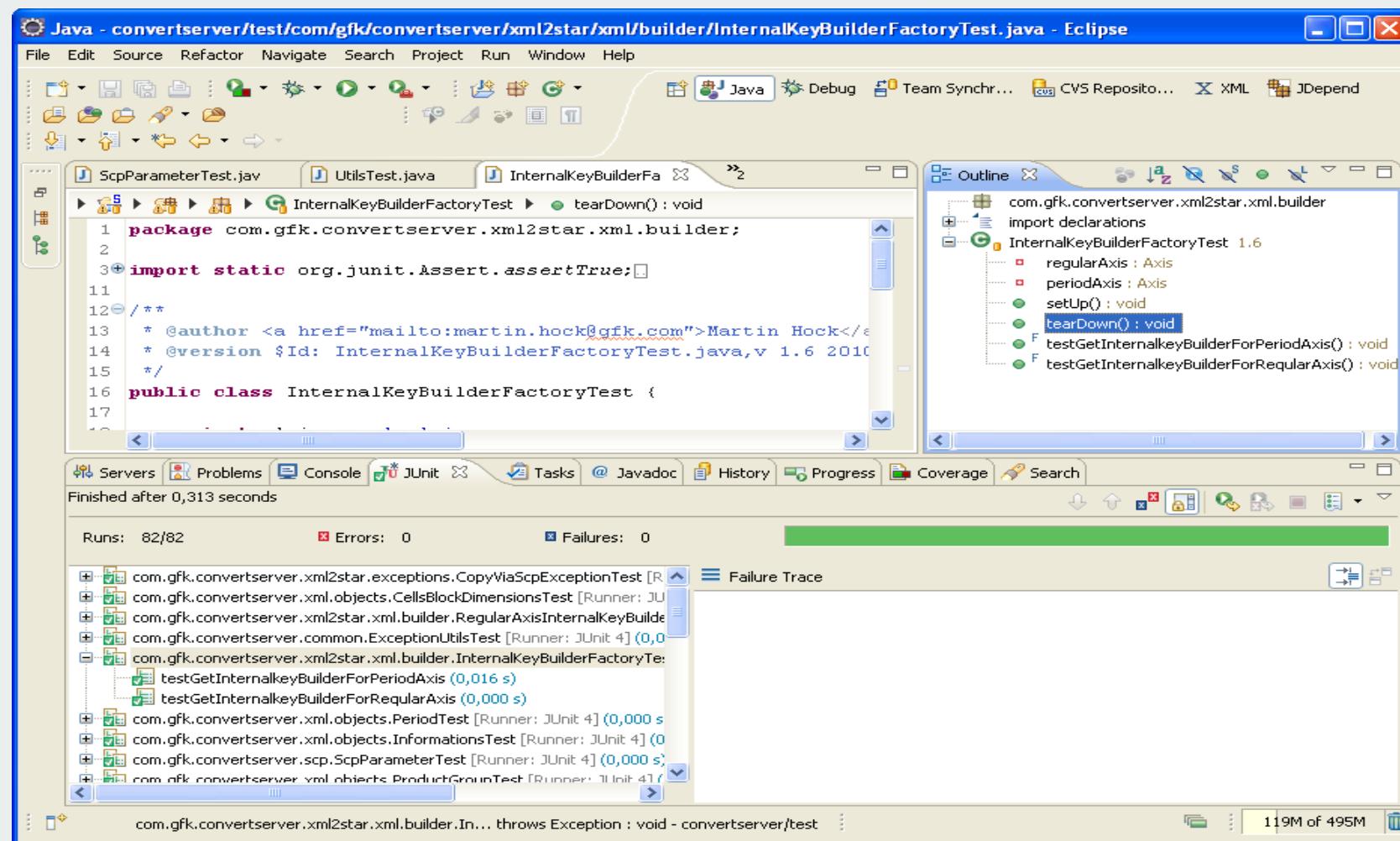
117

```
class SimpleTest {  
  
    private Collection<Object> collection;  
  
    @BeforeEach  
    void setUp() {  
        collection = new ArrayList<Object>();  
    }  
  
    @Test  
    void testEmptyCollection() {  
        assertTrue(collection.isEmpty());  
    }  
  
    @Test  
    void testOneItemCollection() {  
        collection.add("itemA");  
  
        assertEquals(1, collection.size());  
    }  
}
```

```
setUp()  
testEmptyCollection()  
setUp()  
testOneItemCollection()
```

JUnit - Eclipse Integration

118



Zusicherungen mit Assertions - (1/4)

119

- Zusicherung
 - <https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html>
 - Statische Funktionen in der Klasse `Assertions`
 - Methoden `Assertions.assert<BEDINGUNG>()`
 - Ergebnis eines Tests liefert bestimmten Wert

```
assertEquals(0, JobStatus.NOT_AVAILABLE.getId());
assertEquals(1, JobStatus.WAITING.getId());
assertEquals(2, JobStatus.RUNNING.getId());
assertEquals(3, JobStatus.COMPLETED.getId());
assertEquals(4, JobStatus.ERROR.getId());
assertEquals(5, JobStatus.KILLED.getId());
```

assertX (Erwartungswert, Tatsächlicher Wert)

- Auslösen von `AssertionFailedError` bei Ungleichheit

Zusicherungen mit Assertions - (2/4)

120

- **Gleichheit**
 - `assertEquals()` / `assertArrayEquals()`
- **Identität**
 - `assertSame()` / `assertNotSame()`
- **Wahrheit**
 - `assertTrue()` / `assertFalse()`
 - `assertAll()`
- **Existenz**
 - `assertNull()` / `assertNotNull()`
 - `assertThrows()`
- **Sonstige**
 - `assertTimeout()`
 - `fail()`

Zusicherungen mit Assertions - (3/4)

```
class AssertionsDemo {  
    private final Calculator calculator = new Calculator();  
    private final Person person = new Person("Jane", "Doe");  
  
    @Test  
    void standardAssertions() {  
        assertEquals(2, calculator.add(1, 1));  
        assertEquals(4, calculator.multiply(2, 2), "The optional failure message is now the last parameter");  
        assertTrue('a' < 'b', () -> "Assertion messages can be lazily evaluated" + "to avoid constructing complex messages unnecessarily.");  
    }  
  
    @Test  
    void groupedAssertions() {  
        // In a grouped assertion all assertions are executed, and all failures will be reported together.  
        assertAll("person",  
            () -> assertEquals("Jane", person.getFirstName()),  
            () -> assertEquals("Doe", person.getLastName())  
        );  
    }  
  
    @Test  
    void failingTest() {  
        fail("a failing test");  
    }  
  
    @Test  
    void exceptionTesting() {  
        Exception exception = assertThrows(ArithmeticException.class, () -> calculator.divide(1, 0));  
        assertEquals("/ by zero", exception.getMessage());  
    }  
}
```

Zusicherungen mit Assertions - (4/4)

```
class AssertionsDemo {

    @Test
    void timeoutNotExceeded() {
        // The following assertion succeeds.
        assertEquals("a result", assertTimeout(ofMinutes(2), () -> {
            // Perform task that takes less than 2 minutes.
        }));
    }

    @Test
    void timeoutNotExceededWithResult() {
        // The following assertion succeeds, and returns the supplied object.
        String actualResult = assertEquals("a result", assertTimeout(ofMinutes(2), () -> {
            return "a result";
        }));
    }

    @Test
    void timeoutExceeded() {
        // The following assertion fails with an error message similar to:
        // execution exceeded timeout of 10 ms by 91 ms
        assertEquals("a result", assertTimeout(ofMillis(10), () -> {
            // Simulate task that takes more than 10 ms.
            Thread.sleep(100);
        }));
    }
}
```

Annotations

Annotation	Beschreibung
<code>@BeforeAll</code>	Methoden, die beim Starten der Testsuite, vor allen Tests ausgeführt werden (<code>static void</code>).
<code>@AfterAll</code>	Methoden, die nach allen Tests, vor dem Schließen der Testsuite ausgeführt werden. Hier können z. B. Ressourcen freigegeben werden.
<code>@BeforeEach</code>	Methoden, die vor jedem Test ausgeführt werden.
<code>@AfterEach</code>	Methoden, die nach jedem Test ausgeführt werden.
<code>@Test</code>	Die tatsächlichen Testmethoden. Nur Methoden mit dieser Annotation werden als Tests ausgeführt.
<code>@Disabled</code>	Temporäre Deaktivierung von Test-Methoden. Parameter: Meldungstext
<code>@Tag</code>	Dient zur Deklaration von Tags für das Filtern, entweder auf Klassen- oder Methodenebene; analog zu <code>Categories</code> in JUnit 4.
<code>@DisplayName</code>	Angabe von benutzerdefinierten Namen für Testklassen und Methoden.
<code>@Timeout</code>	Wird verwendet um einen Test fehlzuschlagen, wenn seine Ausführung eine bestimmte Dauer überschreitet.

Annotation - @Test

- Markiert die zu testende Methode

```
@Test
void simple() {
    Collection<Object> collection = new ArrayList<Object>();
    assertTrue(collection.isEmpty());
}

@Test
void lambdaExpressions() {
    assertTrue(Stream.of(1, 2, 3)
        .stream()
        .mapToInt(i -> i)
        .sum() > 5, () -> "Sum should be greater than 5");
}

@Test
void groupAssertions() {
    int[] numbers = {0, 1, 2, 3, 4};

    assertAll("numbers",
        () -> assertEquals(numbers[0], 1),
        () -> assertEquals(numbers[3], 3),
        () -> assertEquals(numbers[4], 1)
    );
}
```

Annotation - @BeforeEach / @AfterEach

125

```
class SimpleTest {  
    private Collection collection;  
  
    @BeforeEach  
    void setUp() {  
        collection = new ArrayList();  
    }  
  
    @AfterEach  
    void tearDown() {  
        collection.clear();  
    }  
  
    @Test  
    void testEmptyCollection() {  
        assertTrue(collection.isEmpty());  
    }  
  
    @Test  
    void testOneItemCollection() {  
        collection.add("itemA");  
        assertEquals(1, collection.size());  
    }  
}
```

setUp()	tearDown()
testEmptyCollection()	
setUp()	tearDown()
testOneItemCollection()	

Annotation - @BeforeAll / @AfterAll

126

```
class SimpleTest {  
    private Collection collection;  
  
    @BeforeAll  
    static void oneTimeSetUp() {}  
  
    @AfterAll  
    static void oneTimeTearDown() {}  
  
    @BeforeEach  
    void setUp() {  
        collection = new ArrayList();  
    }  
  
    @AfterEach  
    void tearDown() {  
        collection.clear();  
    }  
  
    @Test  
    void testEmptyCollection() {  
        assertTrue(collection.isEmpty());  
    }  
  
    @Test  
    void testOneItemCollection() {  
        collection.add("itemA");  
        assertEquals(1, collection.size());  
    }  
}
```

oneTimeSetUp()

setUp()
testEmptyCollection()
tearDown()

setUp()
testOneItemCollection()
tearDown()

oneTimeTearDown()

Annotation - @Disabled

127

- Ausschluss von Methoden

```
@Disabled  
@Test  
void something() {  
}
```

- Ausschluss von Methoden mit Kommentar

```
@Disabled("not ready yet")  
@Test  
void something() {  
}
```

- Ausschluss von Klassen

```
@Disabled  
class IgnoreMe {  
    @Test  
    void test1() {  
    }  
  
    @Test  
    void test2() {  
    }  
}
```

Annotation - @DisplayName

128

- Vergabe von Namen auf Klassen- und Methodenebene

Annotation - @Tag

129

- Testklassen und -methoden können über die @Tag-Annotation gekennzeichnet werden.
- Diese Tags können später verwendet werden, um die Erkennung und Ausführung von Tests zu filtern.

```
@Tag("fast")
@Tag("model")
class TaggingDemo {

    @Test
    @Tag("taxes")
    void testingTaxCalculation() {
    }

}
```

Annotation - @Timeout

130

```
class TimeoutDemo {

    @BeforeEach
    @Timeout(5)
    void setUp() {
        // fails if execution time exceeds 5 seconds
    }

    @Test
    @Timeout(value = 100, unit = TimeUnit.MILLISECONDS)
    void failsIfExecutionTimeExceeds100Milliseconds() {
        // fails if execution time exceeds 100 milliseconds
    }
}
```

Annahmen mit Assumptions - (1/2)

131

- <https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assumptions.html>
- Falls die Bedingung nicht zutrifft, wird die Methode verlassen.
- Annahmen werden verwendet, um Tests nur dann durchzuführen, wenn bestimmte Bedingungen erfüllt sind. Dies wird typischerweise für externe Bedingungen verwendet, die für den ordnungsgemäßen Ablauf des Tests erforderlich sind.
- Methoden:
 - `assumeFalse()`
 - `assumeTrue()`
 - `assumingThat()`

```
// only provides information if database is reachable.
@Test
void calculateTotalSalary() {
    DBConnection dbc = Database.connect();
    assumeTrue(dbc != null);
    // ...
}
```

Annahmen mit Assumptions - (2/2)

```
class AssumptionsDemo {

    @Test
    void trueAssumption() {
        assumeTrue(5 > 1);

        assertEquals(7, 5 + 2);
    }

    @Test
    void falseAssumption() {
        assumeFalse(5 < 1);

        assertEquals(7, 5 + 2);
    }

    @Test
    void assumptionThat() {
        String someString = "Just a string";

        assumingThat(
            someString.equals("Just a string"),
            () -> assertEquals(2 + 2, 4)
        );
    }
}
```

Parameterized Tests (1/4)

133

- Parametisierte Tests ermöglichen es, einen Test mehrfach mit unterschiedlichen Argumenten durchzuführen.
- Sie werden genau wie normale `@Test` Methoden deklariert, verwenden aber stattdessen die `@ParameterizedTest` Annotation.
- Unterstützt verschiedene Sourcen als Argument:
 - `@ValueSource`
 - `@EnumSource`
 - `@MethodSource`
 - `@CsvSource`
 - `@CsvFileSource`
 - `@ArgumentsSource`

Parameterized Tests - @ValueSource (1/5)

134

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}

@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testWithValueSource(int argument) {
    assertTrue(argument > 0 && argument < 4);
}
```

Parameterized Tests - @MethodSource (2/5)

135

```
@ParameterizedTest
@MethodSource("stringProvider")
void testWithExplicitLocalMethodSource(String argument) {
    assertNotNull(argument);
}

static Stream<String> stringProvider() {
    return Stream.of("apple", "banana");
}
```

```
@ParameterizedTest
@MethodSource("range")
void testWithRangeMethodSource(int argument) {
    assertEquals(9, argument);
}

static IntStream range() {
    return IntStream.range(0, 20).skip(10);
}
```

Parameterized Tests - @MethodSource (3/5)

```
@ParameterizedTest
@MethodSource("stringIntAndListProvider")
void testWithMultiArgMethodSource(String str, int num, List<String> list) {
    assertEquals(5, str.length());
    assertTrue(num >=1 && num <=2);
    assertEquals(2, list.size());
}

static Stream<Arguments> stringIntAndListProvider() {
    return Stream.of(
        arguments("apple", 1, Arrays.asList("a", "b")),
        arguments("lemon", 2, Arrays.asList("x", "y"))
    );
}
```

Parameterized Tests - @CsvSource (4/5)

```
@ParameterizedTest
@CsvSource({
    "apple,      1",
    "banana,     2",
    "'lemon, lime', 0xF1"
})
void testWithCsvSource(String fruit, int rank) {
    assertNotNull(fruit);
    assertNotEquals(0, rank);
}
```

Parameterized Tests - @CsvFileSource (5/5)

138

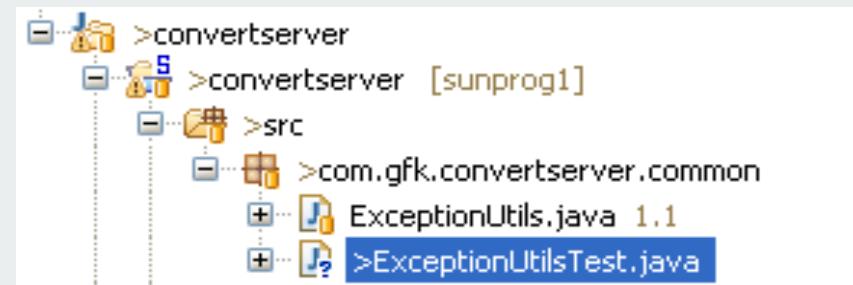
```
@ParameterizedTest
@CsvFileSource(resources = "/two-column.csv", numLinesToSkip = 1)
void testWithCsvFileSource(String country, int reference) {
    assertNotNull(country);
    assertNotEquals(0, reference);
}
```

```
#two-column.csv
Country, reference
Sweden, 1
Poland, 2
"United States of America", 3
```

JUnit - Organisation der Testklassen

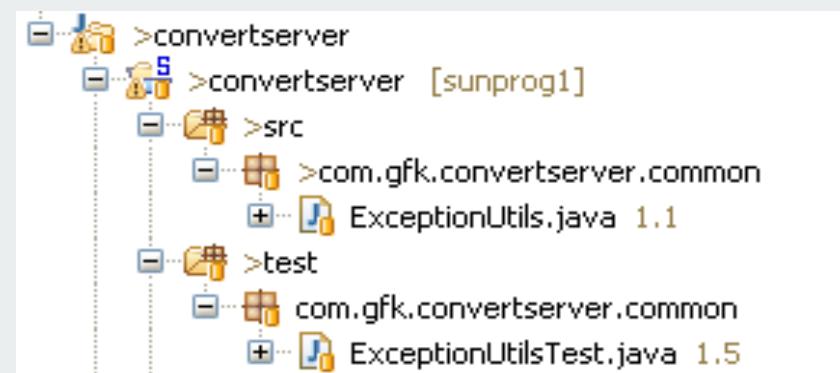
139

- Gleiche Verzeichnisse wie die zu testenden Klassen



- Standard

- zwei Verzeichnisse `src` und `test`
- Klasse und Testklasse im gleichen Paket
- Zugriff auf protected and package private Klassen



JUnit - Heuristik (1/4)

140

- Features testen, nicht Methoden

- Testfälle orientieren sich an Anforderungen und Features, nicht an den zu testenden Methoden.
- Es gibt keine 1:1 Entsprechung von Testfallmethode zu getesteter Methode.
- Testfälle sollen am Beispiel dokumentieren, wie die Klassen korrekt zu verwenden sind.

```
class EuroTest {  
  
    @Test void newEuro() {...}  
    @Test void getIntAmount() {...}  
    @Test void getAmount() {...}  
    @Test void plus() {...}  
    @Test void minus() {...}  
  
}
```



```
class EuroTest {  
  
    @Test void createFromInt() {...}  
    @Test void createFromString() {...}  
    @Test void rounding() {...}  
    @Test void simpleAddition() {...}  
    ...  
  
}
```



JUnit - Heuristik (2/4)

141

- An den Rändern testen
 - Die meisten algorithmischen Fehler treten an den Rändern der erlaubten Wertebereiche auf: Auswahl der Testexemplare dort, anstatt "irgendwo"!
 - Lässt sich der Wertebereich in mehrere für den Test äquivalente Proben unterteilen? Pro Äquivalenzklasse mindestens eine Probe!
 - Was nicht getestet ist, funktioniert möglicherweise auch nicht.
- Implementierungsunabhängigkeit
 - Tests sind gegen die öffentliche Klassenschnittstelle gerichtet.
 - Tests, die auf den Innereien einer Klasse basieren, sind äußerst fragil.
 - Sich Zugriff auf Variablen oder private Methoden zu wünschen zeigt, dass dem Code noch eine entscheidende Designidee fehlt.
- Orthogonale Testfälle
 - Unabhängig voneinander sind Testfälle dann, wenn sie sich auf orthogonale Aspekte beziehen.
 - Häufig lässt sich ein Test extrem vereinfachen, indem er Annahmen macht, die ein anderer Test bereits verifiziert hat.
 - Kommt man in die Not, zu viele Tests anpassen zu müssen, nur um eine Codeänderung machen zu können, sind die Tests nicht orthogonal.

JUnit - Heuristik (3/4)

142

- Ergebnisse im Test festschreiben
 - Erwartete Werte werden als Konstanten kodiert, nicht noch mal im Test berechnet.
 - Reproduzieren wir Anwendungslogik im Test, reproduzieren wir auch ihre Fehler.
 - Performance

```
class EuroTest {  
  
    @Test  
    void multiYearInterest() {  
        double amount = 100.0;  
        double interest = 5.0;  
        double expectedInterest = amount * Math.pow((1 + interest / 100.0), 3.0);  
        assertEquals(expectedInterest, calculator.interest(amount, interest, 3), 0.001);  
    }  
}
```



```
class EuroTest {  
  
    @Test  
    void multiYearInterest() {  
        double amount = 100.0;  
        double interest = 5.0;  
        assertEquals(115.76, calculator.interest(amount, interest, 3), 0.001);  
    }  
}
```



JUnit - Heuristik (4/4)

- Vergiss Exceptions und Fehlerfälle nicht!
- Entferne Redundanz in Testcode!
- Halte Testfälle kurz und verständlich!
- Wähle aussagekräftige Testfallnamen!

JUnit - Namesgebung von Testmethoden

144

- Ansatz 1: Beschreibe den Sachverhalt

```
@Test void newAccount() {}
@Test void withdraw() {}
@Test void cannotWithdrawNegativeAmount() {}
@Test void cannotWithdrawUncoveredAmount() {}
```

- Ansatz 2: Beschreibe das gewünschte Verhalten

```
@Test newAccountShouldReturnCustomer() {}
@Test newAccountShouldHaveZeroBalance() {}
@Test withdrawShouldReduceBalanceByAmount() {}
@Test withdrawNegativeAmountShouldThrowException() {}
@Test withdrawNegativeAmountShouldNotChangeBalance() {}
```

JUnit - Object Mother

145

- Klassische Vorgehensweise

```
Invoice invoice = new Invoice(  
    new Recipient("Sherlock Holmes",  
        new Address("222b Baker Street", "London", new PostCode("NW1", "3RX"))),  
    new InvoiceLines(  
        new InvoiceLine("Deerstalker Hat", new PoundsShillingsPence(0, 3, 10)),  
        new InvoiceLine("Tweed Cape", new PoundsShillingsPence(0, 4, 12))));
```



- Object Mother Pattern

```
Invoice invoice = TestInvoices.newDeerstalkerAndCapeInvoice();  
Invoice invoice1 = TestInvoices.newDeerstalkerAndCapeAndSwordstickInvoice();  
Invoice invoice2 = TestInvoices.newDeerstalkerAndBootsInvoice();  
...
```



JUnit - Test Data Builder (1/2)

- Aufruf

```
Invoice anInvoice = new InvoiceBuilder().build();
```

- Builder Pattern

```
public class InvoiceBuilder {  
  
    Recipient recipient = new RecipientBuilder().build();  
    InvoiceLines lines = new InvoiceLines(new InvoiceLineBuilder().build());  
    PoundsShillingsPence discount = PoundsShillingsPence.ZERO;  
  
    public InvoiceBuilder withRecipient(Recipient recipient) {  
        this.recipient = recipient;  
        return this;  
    }  
    public InvoiceBuilder withInvoiceLines(InvoiceLines lines) {  
        this.lines = lines;  
        return this;  
    }  
    public InvoiceBuilder withDiscount(PoundsShillingsPence discount) {  
        this.discount = discount;  
        return this;  
    }  
    public Invoice build() {  
        return new Invoice(recipient, lines, discount);  
    }  
}
```

JUnit - Test Data Builder (2/2)

- Builder kombinieren

```
Invoice invoiceWithNoPostcode = new InvoiceBuilder()  
    .withRecipient(new RecipientBuilder()  
        .withAddress(new AddressBuilder()  
            .withNoPostcode()  
            .build())  
        .build())  
    .build();
```

- Builder kombinieren, besser

```
Invoice invoice = new InvoiceBuilder()  
    .withRecipient(new RecipientBuilder().withAddress(new AddressBuilder().withNoPostcode()))  
    .build();
```



```
Invoice invoice = anInvoice()  
    .fromRecipient(aRecipient().withAddress(anAddress().withNoPostcode()))  
    .build();
```



```
Invoice invoice = anInvoice()  
    .from(aRecipient().with(anAddress().withNoPostcode()))  
    .build();
```



JUnit - F.I.R.S.T (1/2)

- Saubere Tests folgen fünf Regeln:
 - Fast (schnell)
 - Tests sollten schnell sein
 - Wenn Tests langsam laufen, werden sie seltener ausgeführt und finden daher Probleme nicht früh genug.
 - Independent (unabhängig)
 - Tests sollten nicht voneinander abhängen, sondern unabhängig ausführbar sein.
 - Ein Test sollte keine Bedingungen für den nächsten Test setzen und in beliebiger Reihenfolge ausführbar sein.
 - Wenn Tests voneinander abhängen, dann löst der erste, der scheitert, eine Kaskade weiterer nicht bestandener Tests stromabwärts aus.
 - Repeatable (wiederholbar)
 - Tests sollten in jeder Umgebung wiederholbar sein.
 - Die Tests sollten in der Produktionsumgebung, in der QA-Umgebung und auf dem Laptop im Zug ohne Netzwerk ausführbar sein.

JUnit - F.I.R.S.T (2/2)

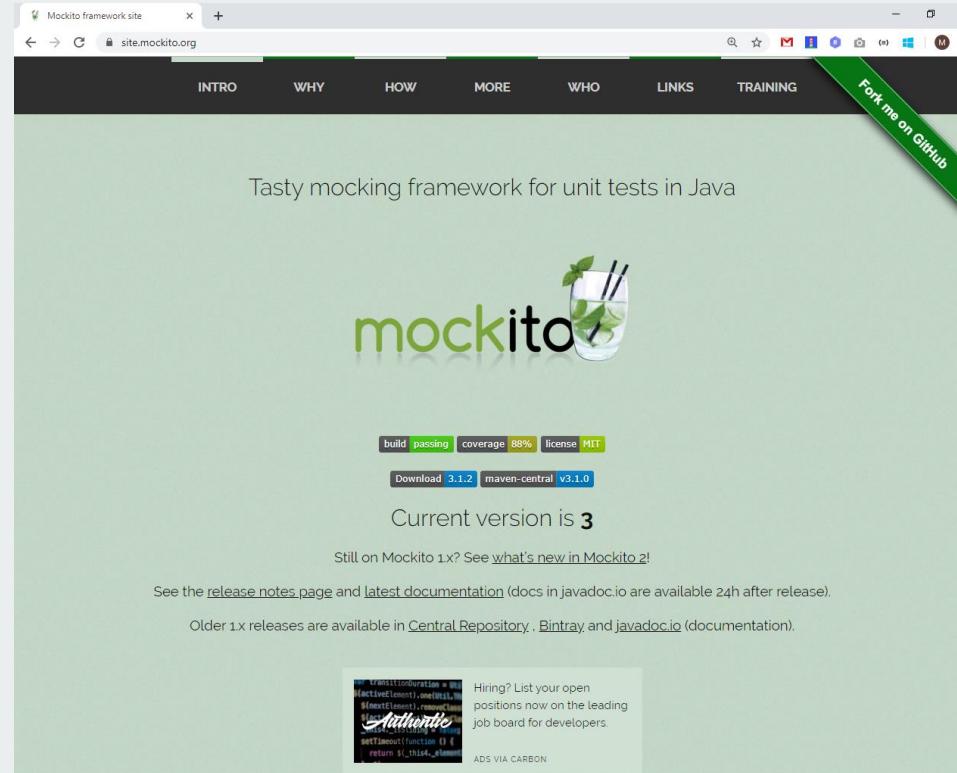
- Self-Validating (selbst-validierend)
 - Tests sollten einen booleschen Output haben.
 - Entweder sie werden bestanden oder sie scheitern.
 - Das manuelle Studieren von Protokolldateien soll vermieden werden.
 - Wenn die Tests sich nicht selbst validieren, kann das Scheitern subjektiv werden, und für die Ausführung der Tests kann eine lange manuelle Auswertung erfordern.
- Timely (zeitgerecht)
 - Tests müssen rechtzeitig geschrieben werden.
 - Unit-Tests sollten kurz vor dem Produktionscode geschrieben werden, der dafür sorgt, dass sie bestanden werden.
 - Wenn Tests nach dem Produktionscode entwickelt werden, dann kann es passieren, dass der Produktionscode möglicherweise schwer zu testen ist.

Werkzeuge zur Sicherung der Softwarequalität

Mockito

Mockito (1/5)

- Mockito ist ein Java-Mocking-Framework zum Erstellen von Mock-Objekten für Unit-Tests von Java-Programmen.
- <https://site.mockito.org/>



Mockito – Aktivierung (2/5)

152

```
// Activating with Junit 5 ExtendWith Annotation
// MockitoExtension Extension that initializes mocks and handles strict stubbings.
@ExtendWith(MockitoExtension.class)
class MockitoAnnotationTest {
    ...
}
```

```
// Alternatively, we can enable these annotations programmatically as well, by invoking MockitoAnnotations.initMocks()
@BeforeEach
void init() {
    MockitoAnnotations.initMocks(this);
}
```

Mockito – @Mock (3/5)

- Manuell mit Mockito.mock()

```
@Test
void whenNotUsingMockAnnotation() {
    List mockList = Mockito.mock(ArrayList.class);

    mockList.add("one");
    Mockito.verify(mockList).add("one");
    assertEquals(0, mockList.size());

    Mockito.when(mockList.size()).thenReturn(100);
    assertEquals(100, mockList.size());
}
```

- Mit @Mock Annotation

```
@Mock
List<String> mockedList;

@Test
void whenUsingMockAnnotation() {
    mockedList.add("one");
    Mockito.verify(mockedList).add("one");
    assertEquals(0, mockedList.size());

    Mockito.when(mockedList.size()).thenReturn(100);
    assertEquals(100, mockedList.size());
}
```

Mockito – @Spy (4/5)

154

- Manuell mit Mockito.spy()

```
@Test
void whenNotUsingSpyAnnotation() {
    List<String> spyList = Mockito.spy(new ArrayList<String>());

    spyList.add("one");
    Mockito.verify(spyList).add("one");
    assertEquals(1, spyList.size());

    Mockito.doReturn(100).when(spyList).size();
    assertEquals(100, spyList.size());
}
```

- Mit @Spy Annotation

```
@Spy
List<String> spiedList = new ArrayList<String>();

@Test
void whenUsingSpyAnnotation() {
    spiedList.add("one");
    Mockito.verify(spiedList).add("one");
    assertEquals(2, spiedList.size());

    Mockito.doReturn(100).when(spiedList).size();
    assertEquals(100, spiedList.size());
}
```

Mockito – @Captor (5/5)

```
@Mock
List mockedList;

@Captor
ArgumentCaptor argCaptor;

@Test
void whenUseCaptorAnnotation() {
    mockedList.add("one");
    Mockito.verify(mockedList).add(argCaptor.capture());

    assertEquals("one", argCaptor.getValue());
}
```

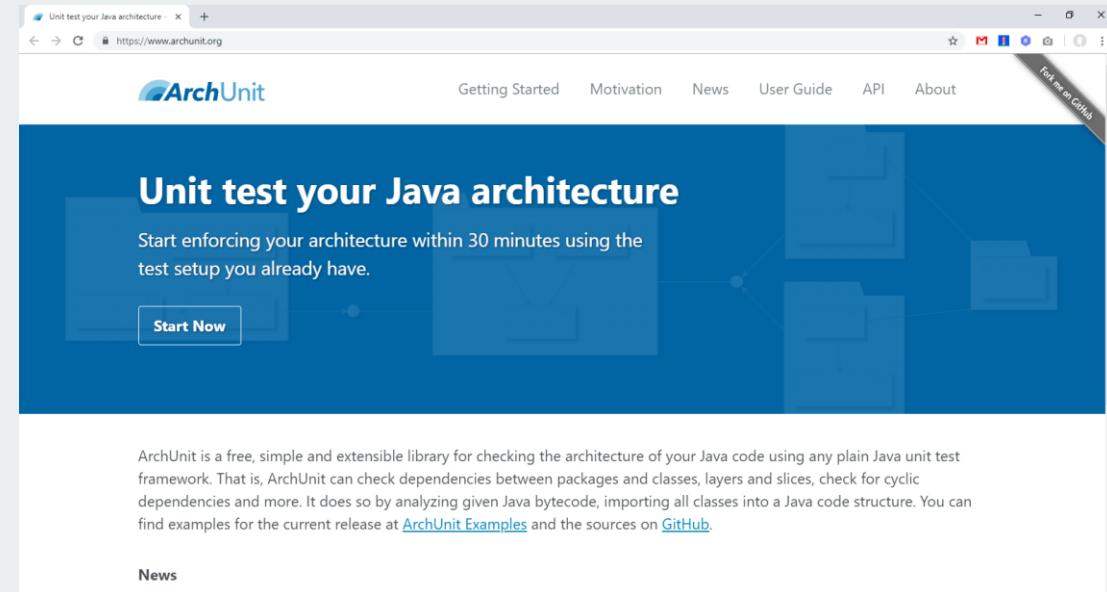
Werkzeuge zur Sicherung der Softwarequalität

ArchUnit

ArchUnit (1/7)

157

- ArchUnit ist eine kostenlose, einfache und erweiterbare Bibliothek zur Überprüfung der Architektur von Java-Code
- Es kann Abhängigkeiten zwischen Paketen und Klassen, Schichten und Slices, zyklischen Abhängigkeiten und mehr überprüfen
- Ermöglicht kontinuierlichen Architekturerhalt
- <https://www.archunit.org/>

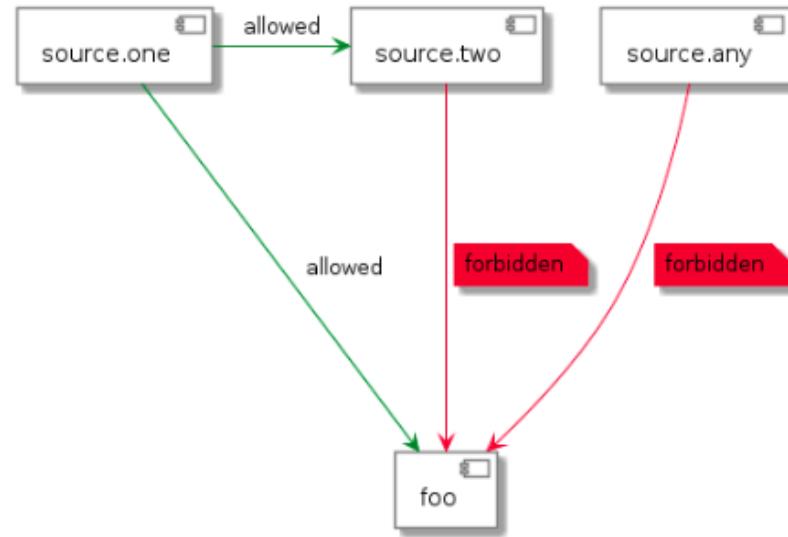


ArchUnit - Package Dependency Checks (2/7)

158



```
noClasses().that().resideInAPackage("..source..")  
    .should().dependOnClassesThat().resideInAPackage("..foo..")
```



```
classes().that().resideInAPackage("..foo..")  
    .should().onlyHaveDependentClassesThat().resideInAnyPackage("..source.one..", "..foo..")
```

ArchUnit - Class Dependency Checks (3/7)

159



ArchUnit - Class and Package Containment Checks (4/7)

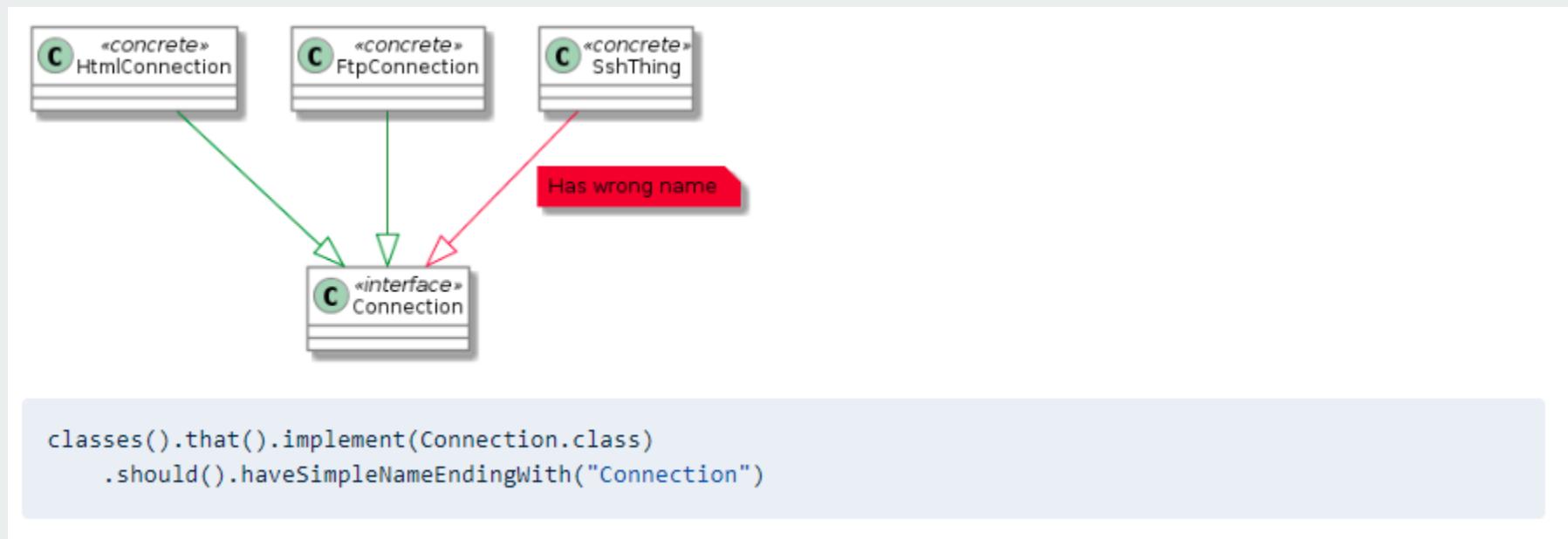
160

The diagram illustrates a package structure with two packages: `com.wrong` and `com.foo`. Inside `com.wrong`, there is a class named `FooController`. Inside `com.foo`, there is a class named `FooService`. A red arrow points from `FooController` to a red box at the bottom containing the text "resides in wrong package".

```
classes().that().haveSimpleNameStartingWith("Foo")
    .should().resideInAPackage("com.foo")
```

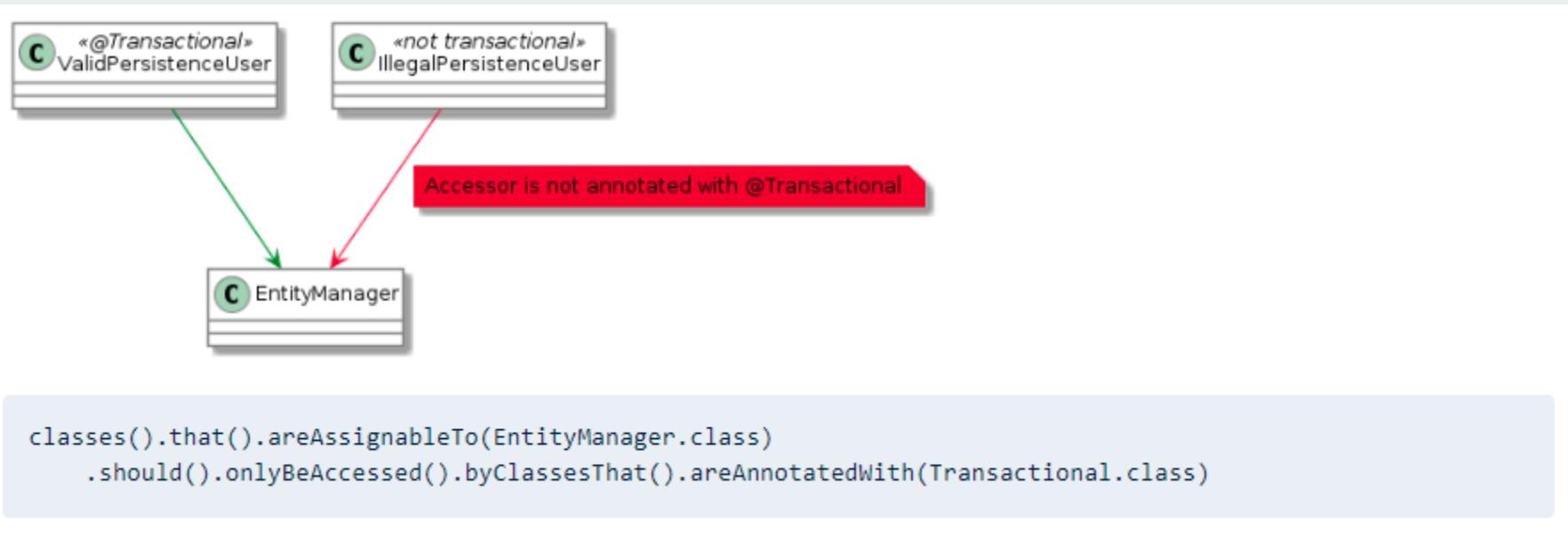
ArchUnit - Inheritance Checks (5/7)

161



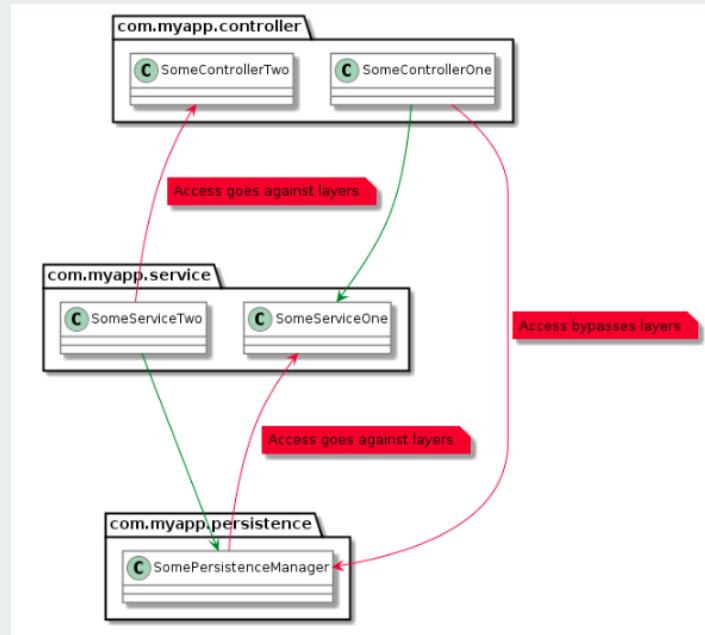
ArchUnit - Annotation Checks (6/7)

162



ArchUnit - Layer Checks (7/7)

163



```
layeredArchitecture()  
    .layer("Controller").definedBy(..controller..)  
    .layer("Service").definedBy(..service..)  
    .layer("Persistence").definedBy(..persistence..)  
  
    .whereLayer("Controller").mayNotBeAccessedByAnyLayer()  
    .whereLayer("Service").mayOnlyBeAccessedByLayers("Controller")  
    .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Service")
```

10 Gründe, warum Test Driven Development die Lebensqualität verbessert (1/4)

164

1. If you can't write a test for something you don't understand it

Die Praxis, Tests vor der Implementierung zu schreiben, zwingt dazu, sich intensiver mit dem zu lösenden Problem auseinanderzusetzen und führt so zu einer höheren Qualität des Codes, der auch für andere Entwickler besser nachvollziehbar ist. Als Nebeneffekt wird man bei TDD außerdem dazu genötigt, sich mit der Frage zu beschäftigen, was die Kunden eigentlich genau von der zu schreibenden Software erwarten. TDD ist also im Grunde keine Test- sondern eine Designmethode.

2. Without a regression test you can't clean the code

Ohne die Möglichkeit des schnellen Testens traut man sich oft nicht, existierenden lauffähigen Code zu verändern. Man kopiert den Code einer Klasse z. B. lieber in eine neue Klasse, die man dann um zusätzliche Features erweitert. Ergebnis ist ein überfrachtetes, unübersichtliches Projekt mit großen Mengen überflüssigen Codes.

10 Gründe, warum Test Driven Development die Lebensqualität verbessert (2/4)

165

3. Fast feedback cycles save time and money

Wenn man zuerst den Test schreibt, lässt sich die Korrektheit der Implementierung eines Features sofort überprüfen. Das führt dazu, dass man immer Gewissheit darüber hat, wann man mit der Entwicklung eines Features tatsächlich fertig ist. Ohne diese schnellen Feedback-Zyklen verliert man sich oft in der Suche nach den Ursachen für einen Bug. TDD hilft dabei, ein großes Problem in kleinere, überschaubare und testbare Probleme zu zerlegen.

4. You are going to test it anyway, spend the time to do it right

Regressionstests sind ohnehin unabdingbar. Ohne dass man diese regelmäßig durchführt, wird das Testen von großen Applikationen fast unmöglich. Sicherlich lassen sich Regressionstests auch nachträglich schreiben, doch dann kommt man erstes nicht in den Genuss der besseren Design-Qualität durch TDD. Zweitens tendiert die Zeit dazu, mit dem Vorrücken einer Deadline knapp zu werden, sodass nachträgliches Testen oft zu kurz kommt.

10 Gründe, warum Test Driven Development die Lebensqualität verbessert (3/4)

166

5. Without tests the next developer will need a word doc and a lot of luck

Unit-Tests sind auch eine gute Form der Dokumentation, da sie die Inputs und erwarteten Outputs anzeigen. TDD bringt den Testprozess und damit diese Form der Dokumentation ins Zentrum der Aufmerksamkeit.

6. Evolutionary design is possible without fear

Ohne TDD hat man oft Angst, die bestehende Codebasis zu verändern, da beim Auftreten eines Fehlers lange Ursachenforschung nötig wird. Mit TDD wird die Möglichkeit der Änderungen an der Codebasis quasi zum Prinzip erhoben, was einen wirklich evolutionären Entwicklungsprozess ermöglicht.

7. You will actually write less code

Mit Hilfe moderner IDEs und Refactoring-Werkzeugen schreibt man mit TDD weniger Code als ohne. Wenn man z. B. mit dem Tool [ReSharper](#) zunächst einen Test für eine noch nicht existierende Klasse schreibt, wirft ReSharper das Grundgerüst der Klasse auf Knopfdruck automatisch aus. Lästiger Boilerplate-Code muss nicht händisch geschrieben werden.

10 Gründe, warum Test Driven Development die Lebensqualität verbessert (4/4)

167

8. Job security

Bug-freie Software ist auch das Ziel eines jeden Chefs!

9. It actually makes my work more enjoyable

Bei jedem erfolgreichen Test hat man das Gefühl, etwas Wertvolles geschaffen zu haben: ein Stück lauffähiger Software, die man stolz auch anderen Entwicklern zeigen kann.

10. Wrapping it up

Billups' Fazit: TDD stellt die Qualität und Lauffähigkeit von Software sicher. In Verbindung mit Akzeptanztests kommt man so zu einem System, welches den Anforderungen der Kunden tatsächlich entspricht.

Quellen

- Robert, Martin: *Clean Code – Refactoring, Patterns, Testen und Techniken für sauberen Code*, mitp-Verlag, 2009
- <http://junit.org/>
- <http://www.aifb.uni-karlsruhe.de/CoM/projects/EPP/EPP02-03/crashkurs/JUnit.pdf/>
- <http://www.iai.uni-bonn.de/III//lehre/praktika/xp/WS2002/material/unitTests.pdf/>
- http://www.dinkla.net/slides/20090320_TDD_JUnit4_4s.pdf/
- <http://de.wikipedia.org/wiki/Software-Test/>
- http://en.wikipedia.org/wiki/Test-driven_development/
- <http://de.wikipedia.org/wiki/White-Box-Test/>
- <http://it-republik.de/jaxenter/news/10-Gruende-warum-Test-Driven-Development-die-Lebensqualitaet-verbessert-055109.html/>

Werkzeuge zur Sicherung der Softwarequalität

Code Coverage mit EMMA

Code Coverage (1/2)

170

- Problem
 - Wie kann man die Qualität von Unit-Tests beurteilen und verbessern?
- Ziel
 - Mit möglichst geringem Aufwand einen möglichst umfassenden Testfall implementieren!
- Lösung
 - Messen der Codeabdeckung von Testfällen
 - während der Ausführung der Tests wird von der Laufzeitumgebung gemessen, welche Statements / Codezeilen tatsächlich abgearbeitet wurden
 - statistische Aufbereitung der Daten
 - nach Testfall, Komponente, Package, Teilsystem etc.
 - z. T. auch historisiert, d. h. zeitliche Entwicklung der Werte

Code Coverage (2/2)

171

- Code Coverage ist eine Metrik welche zur Laufzeit misst, welche Quellcodezeilen abgearbeitet wurde.
- Grad in welchem der Quellcode getestet wurde.
- Art von White-Box-Tests
- Messung mittels Ausführung der Unit-Testfälle (z. B. mit JUnit)
- Somit kann eine Aussage gemacht werden, wie umfassend der Code tatsächlich getestet wurde!
 - Verbesserung der SW-Qualität durch besser Qualität der Testfälle
 - Effizienzsteigerung der Testfälle
- Die Testabdeckung sollte NIE als Metrik über die Qualität der Tests oder des Codes verstanden werden!
 - Ist nur ein Indikator für Problembereiche.
 - Kann also nur negative Aussagen machen, keine positiven.

Code Coverage - Was wird gemessen? (1/2)

172

- Messtechniken
 - Statement Coverage / Line Coverage
 - Misst, ob und wie häufig eine einzelne Codezeile durchlaufen wurde.
 - Problem: Handelt es sich bei der Zeile um einen logischen Vergleich, ist ein einmaliger Durchlauf nicht repräsentativ.
 - Decision Coverage / Branch Coverage
 - Bei Fallunterscheidungen (`if`, `while` etc.) wird zusätzlich geprüft, dass beide Fälle (`true` und `false`) aufgetreten sind.
 - Path Coverage
 - Bei der Path Coverage wird gemessen, ob alle möglichen Kombinationen von Programmablaufpfäden durchlaufen wurden.
 - Problem: Die Anzahl der Möglichkeiten steigt exponentiell mit der Anzahl Entscheidungen.
 - Function Coverage
 - Misst auf der Basis der Funktionen ob sie aufgerufen wurden.
 - Race Coverage
 - Konzentriert sich auf Codestellen die parallel ablaufen.

Code Coverage - Was wird gemessen? (2/2)

173

- Unterschiedlich aufwändig und aussagekräftig
- Die meisten Code Coverage-Werkzeuge auf dem Markt unterstützen nur Statement / Decision Coverage, teilweise noch Path Coverage.
 - Dennoch liefern sie damit schon sehr wertvolle Aussagen, z. B.:
 - Codeteile die beim Test schlicht vergessen wurden
 - Ausnahmen, die nicht berücksichtigt wurden

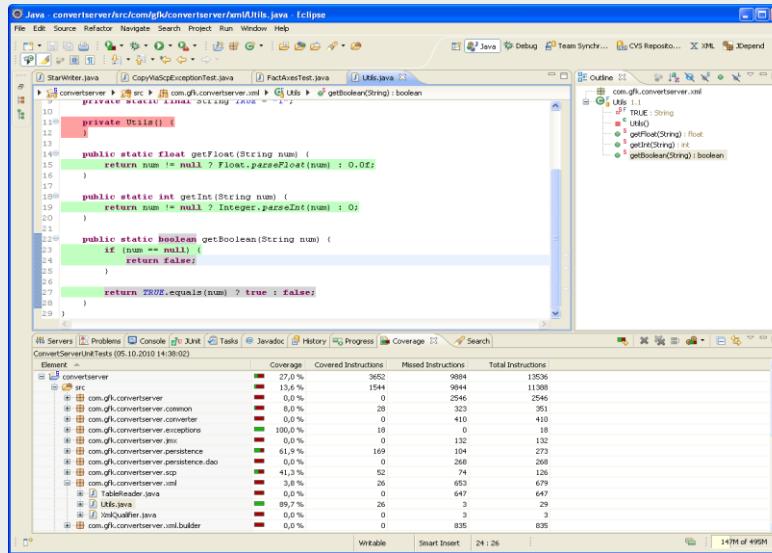
Code Coverage - Herausforderung

174

- Eine 100%-ige Abdeckung ist eine Illusion!
- Erreichbare Abdeckung stark abhängig von Projekttyp
 - Automatisierbare Testfälle als Grundlage
- Realistische Werte liegen im Bereich von 60-80%
 - Projektspezifisch festlegen
 - Kann im Build-Prozess automatisch geprüft werden

EclEmma

- Java basiertes Code Coverage Framework
- Unterstützung von Statement / Decision Coverage
- Generierung von Text, HTML und XML Reports
- Bestandteil von Eclipse
- <http://www.eclemma.org/>



EclEmma 3.0.0 | Java Code Coverage for Eclipse

Overview

EclEmma is a free Java code coverage tool for [Eclipse](#), available under the [Eclipse Public License](#). It brings code coverage analysis directly into the Eclipse workbench.

- **Fast develop/test cycle:** Launches from within the workbench like JUnit test runs can directly be analyzed for code coverage.
- **Rich coverage analysis:** Coverage results are immediately summarized and highlighted in the Java source code editors.
- **Non-invasive:** EclEmma does not require modifying your projects or performing any other setup.

Since version 2.0 EclEmma is based on the [JaCoCo](#) code coverage library. The Eclipse integration has its focus on supporting the individual developer in an highly interactive way. For automated builds please refer to [JaCoCo documentation for integrations with other tools](#).

Originally EclEmma was inspired by and technically based on the great [EMMA](#) library developed by Vlad Roubtsov.

The update site for EclEmma is <http://update.eclemma.org/>. EclEmma is also available via the Eclipse [Marketplace Client](#). simply search for "EclEmma".

Features

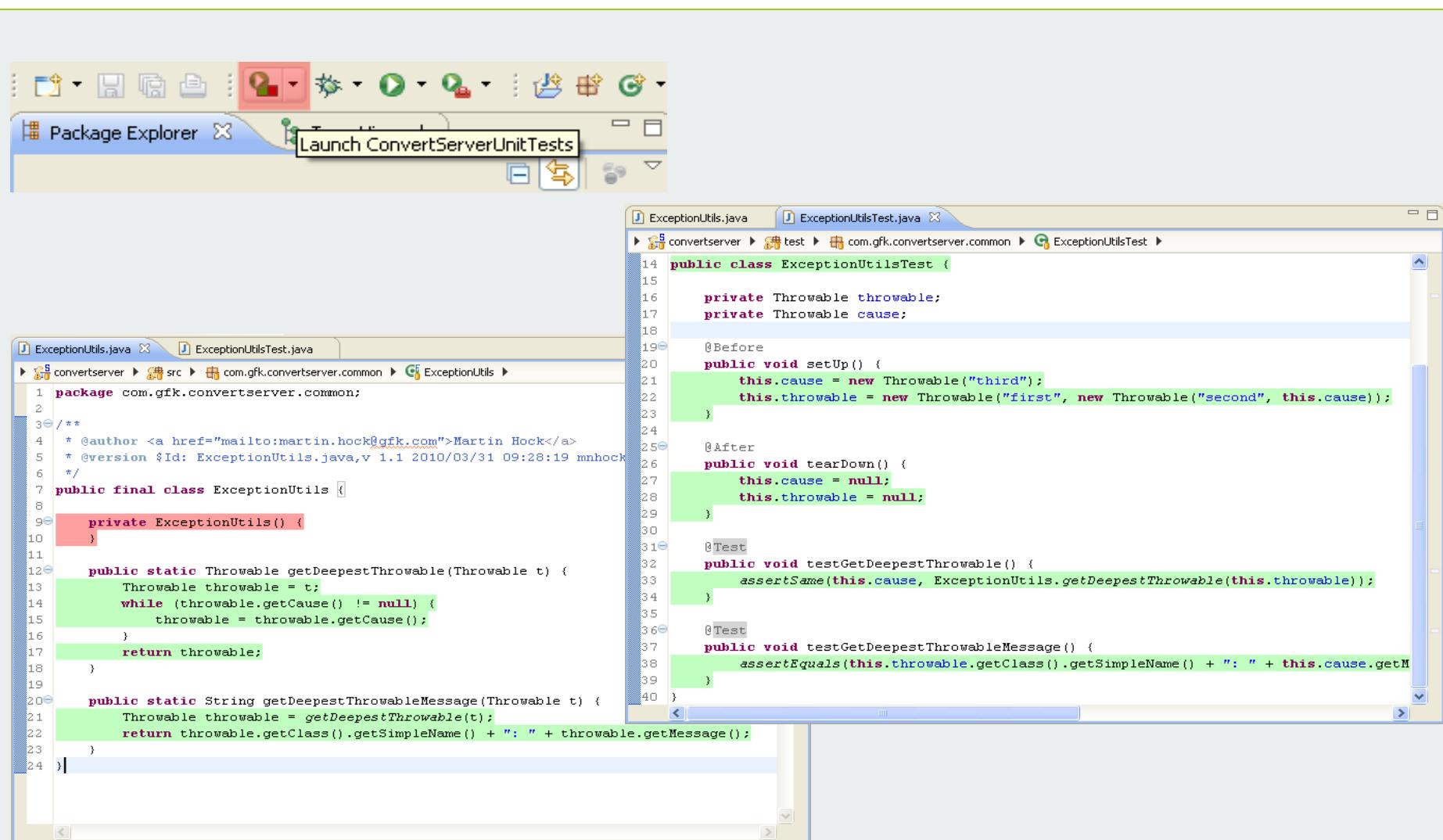
Launching

EclEmma adds a so called *launch mode* to the Eclipse workbench. It is called *Coverage mode* and works exactly like the existing *Run* and *Debug* modes. The *Coverage* launch mode can be activated from the *Run* menu or the workbench's toolbar.

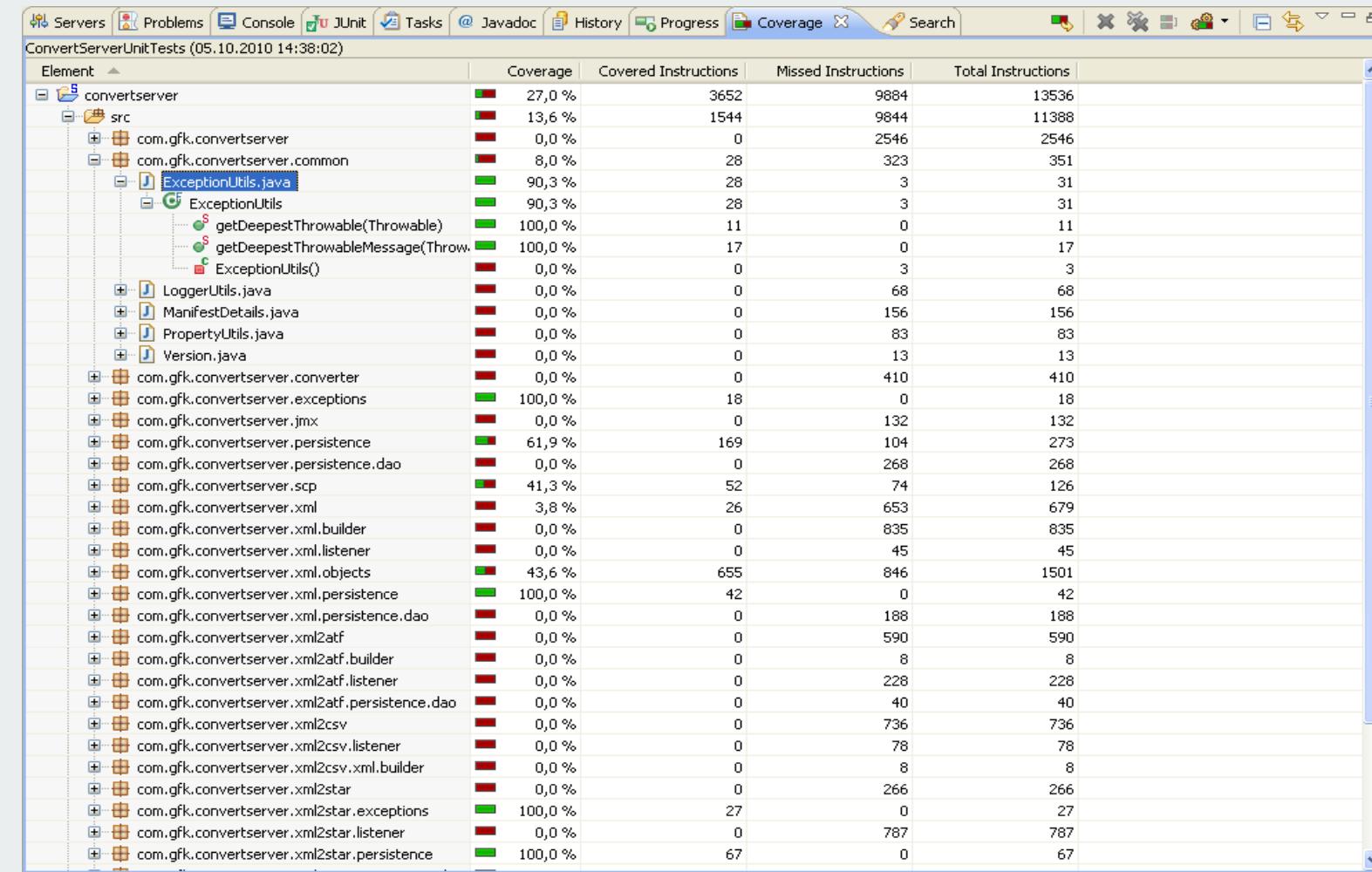
Simply [launch](#) your applications or unit tests in the *Coverage* mode to collect coverage information. Currently the following launch types are supported:

- Local Java application
- Eclipse/RCP application

EclEmma und Eclipse (1/2)



EclEmma und Eclipse (2/2)



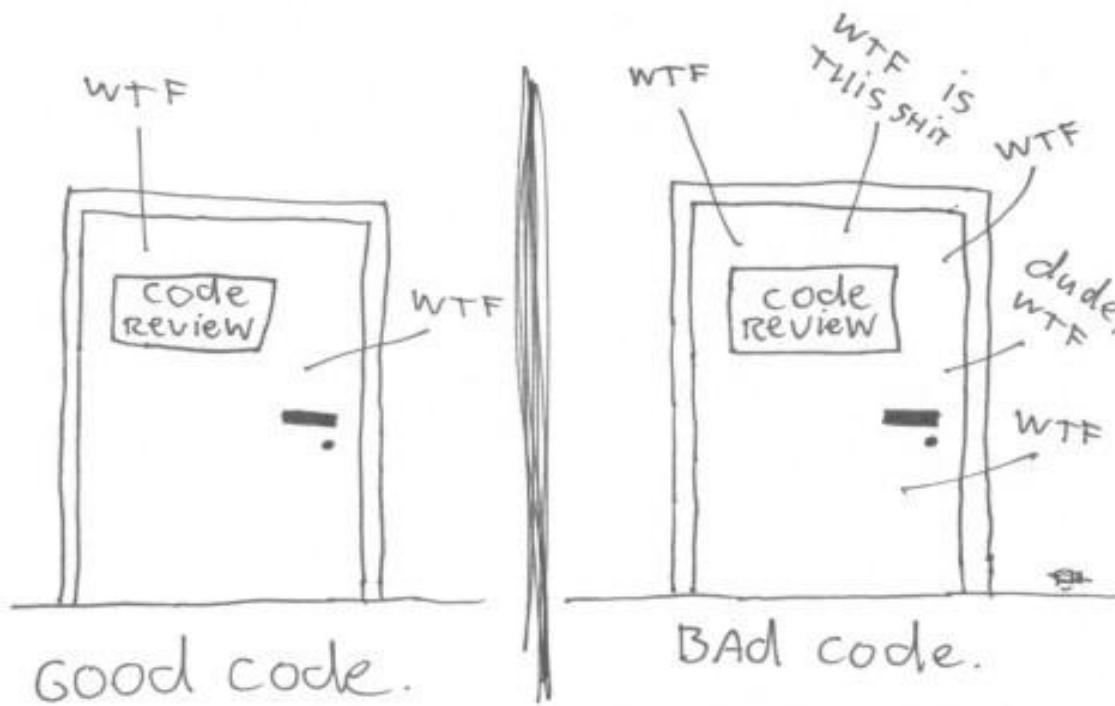
Quellen

- <http://emma.sourceforge.net/>
- <http://de.wikipedia.org/wiki/Überdeckungstest>
- http://www.stuber.info/hslu/SWK/Unterlagen/08_Testing.pdf/

Werkzeuge zur Sicherung der Softwarequalität

Tools zur statischen Codeanalyse

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



Statische Codeanalyse

- Statische Codeanalyse ist ein statisches Software-Testverfahren.
- Im Gegensatz zu dynamischen Testverfahren muss die zu testende Software nicht ausgeführt werden.
 - Es wird also nur der Quelltext benötigt
 - Lauffähigkeit des Systems nicht erforderlich
 - Keine spezielle Testumgebung
- Kann durch manuelle Inspektion erfolgen, aber auch automatisch durch ein Programm.
- White-Box-Test
- Statische Codeanalyse wird auch bei Compilern und in IDEs verwendet.
 - Typprüfung (Type Check), z. B. korrekte Verwendung von Typen
 - Stilprüfung (Style Check), z. B. Verwendung von Coderichtlinien
 - Fehlersuche (Bug Finding), z. B. Ermittlung der Verwendung nicht initialisierter Variablen
- Sollte so früh wie möglich in den Entwicklungsprozess verankert werden.

Statische Codeanalyse - Grundprinzipien

182

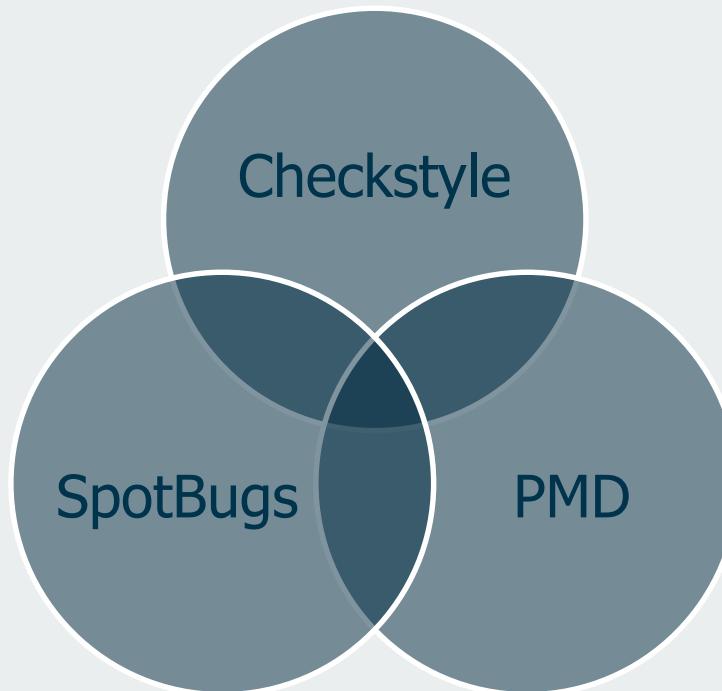
- Source Code oder kompilierte .class Dateien werden analysiert.
- Code kommt nicht zur Ausführung.
- Einfache Ausführung der Tools als Batch Datei, ANT Task oder Maven Goal.
- Ausgabedaten sind XML oder HTML Dateien.



Welches Tool nutzen?

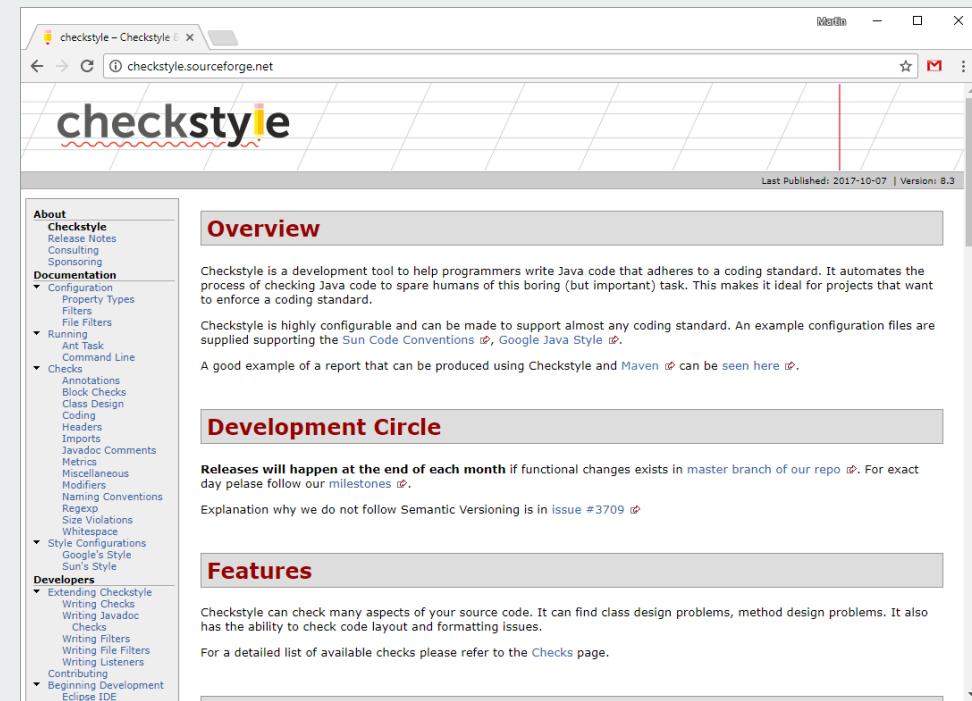
183

- Jedes Tool hat seine Stärken und Schwächen.
- Regeln der Tools überdecken sich teilweise.
- Optimale Qualität bei Einsatz von mehreren Tools.



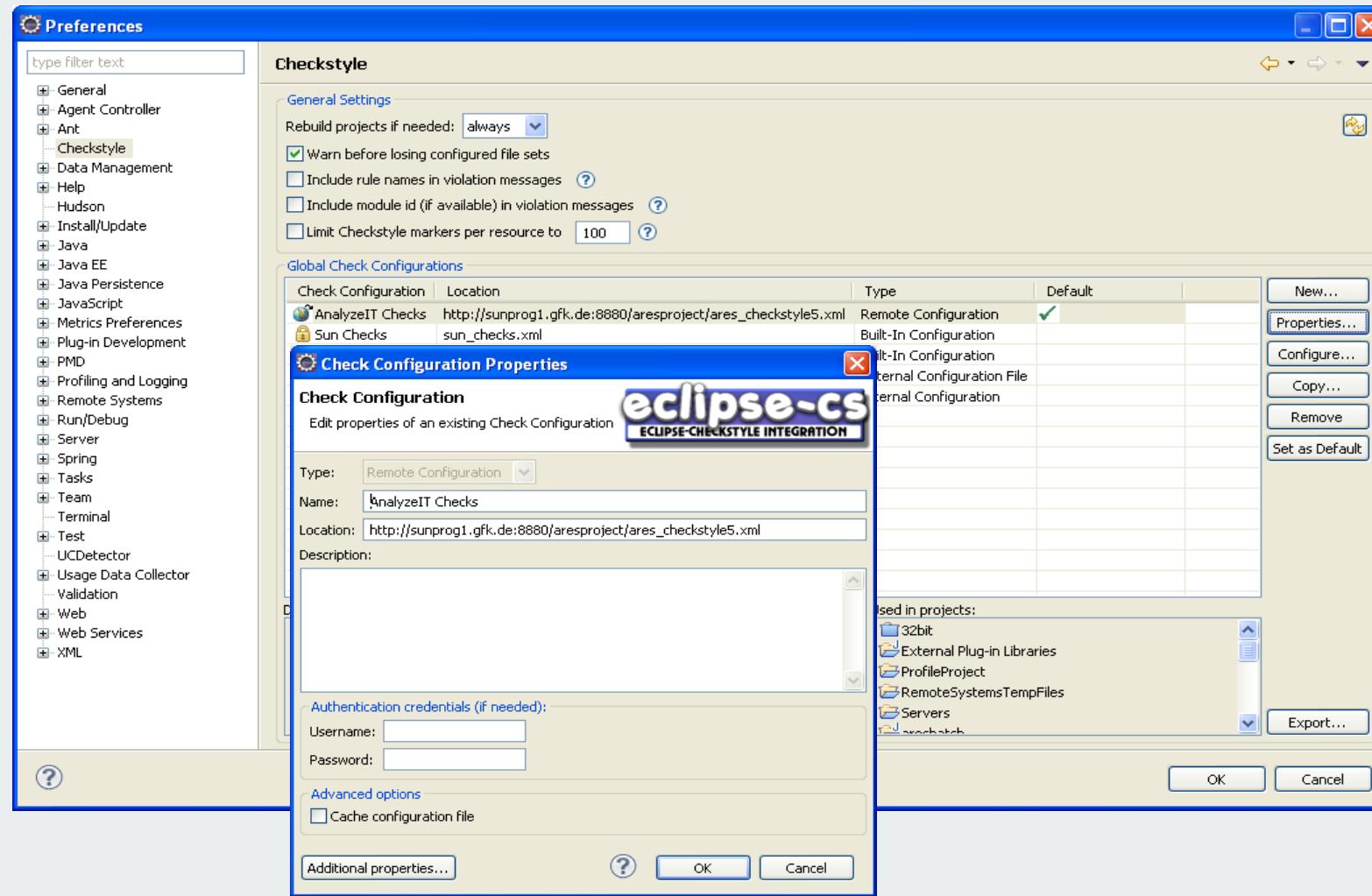
Checkstyle

- Checkstyle ist ein Open Source Programm, das die Einhaltung von Coding Conventions in Java Programmen unterstützt.
- Unterstützung für ANT, Maven, Eclipse, Netbeans, Standalone Version
- <http://checkstyle.sourceforge.net/>



Checkstyle - Eclipse

185



Checkstyle - Eclipse

186

Checkstyle Configuration

Remote Configuration "AnalyzeIT Checks"
The configuration can not be edited.

eclipse-cs
ECLIPSE-CHECKSTYLE INTEGRATION

Known modules

Input filter text here

- Naming Conventions
 - Abstract Class Name
 - Class Type Parameter Name
 - Constant Names**
 - Local Final Variable Names
 - Local Variable Names
 - Member Names
 - Method Names
 - Method Type Parameter Name
 - Package Names
 - Parameter Names
 - Static Variable Names
 - Type Names
- Headers
- Imports

Add... -> <- Remove Open...

Configured modules for group "Naming Conventions"

Enabled	Module	Severity	Comment
<input type="checkbox"/>	Local Variable Names	ignore	
<input type="checkbox"/>	Member Names	ignore	
<input type="checkbox"/>	Parameter Names	ignore	
<input checked="" type="checkbox"/>	Constant Names	warning	
<input checked="" type="checkbox"/>	Local Final Variable Names	warning	
<input checked="" type="checkbox"/>	Method Names	warning	
<input checked="" type="checkbox"/>	Package Names	error	
<input checked="" type="checkbox"/>	Parameter Names	warning	
<input checked="" type="checkbox"/>	Static Variable Names	warning	
<input checked="" type="checkbox"/>	Type Names	error	

Description:

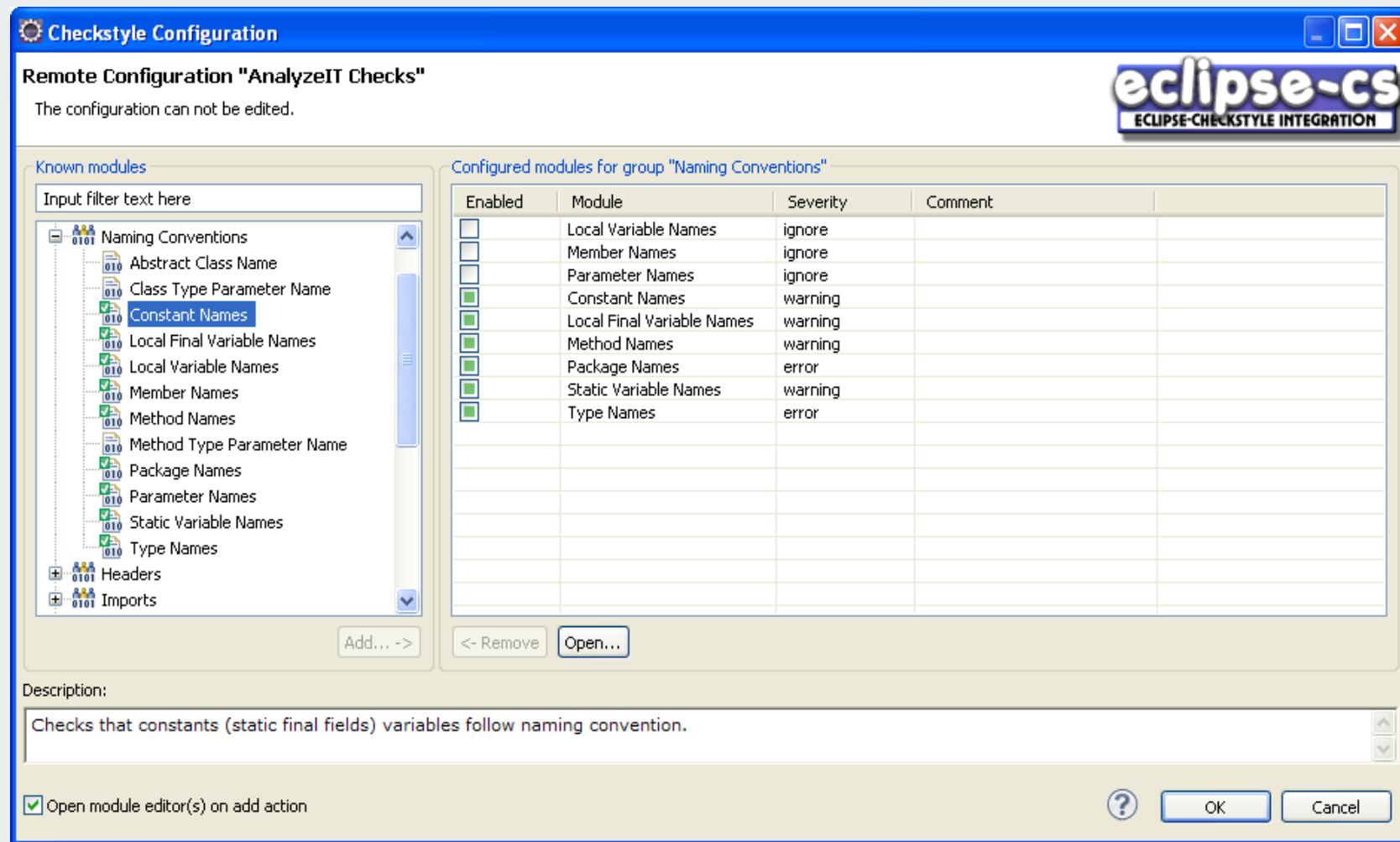
Checks that constants (static final fields) variables follow naming convention.

Open module editor(s) on add action

?

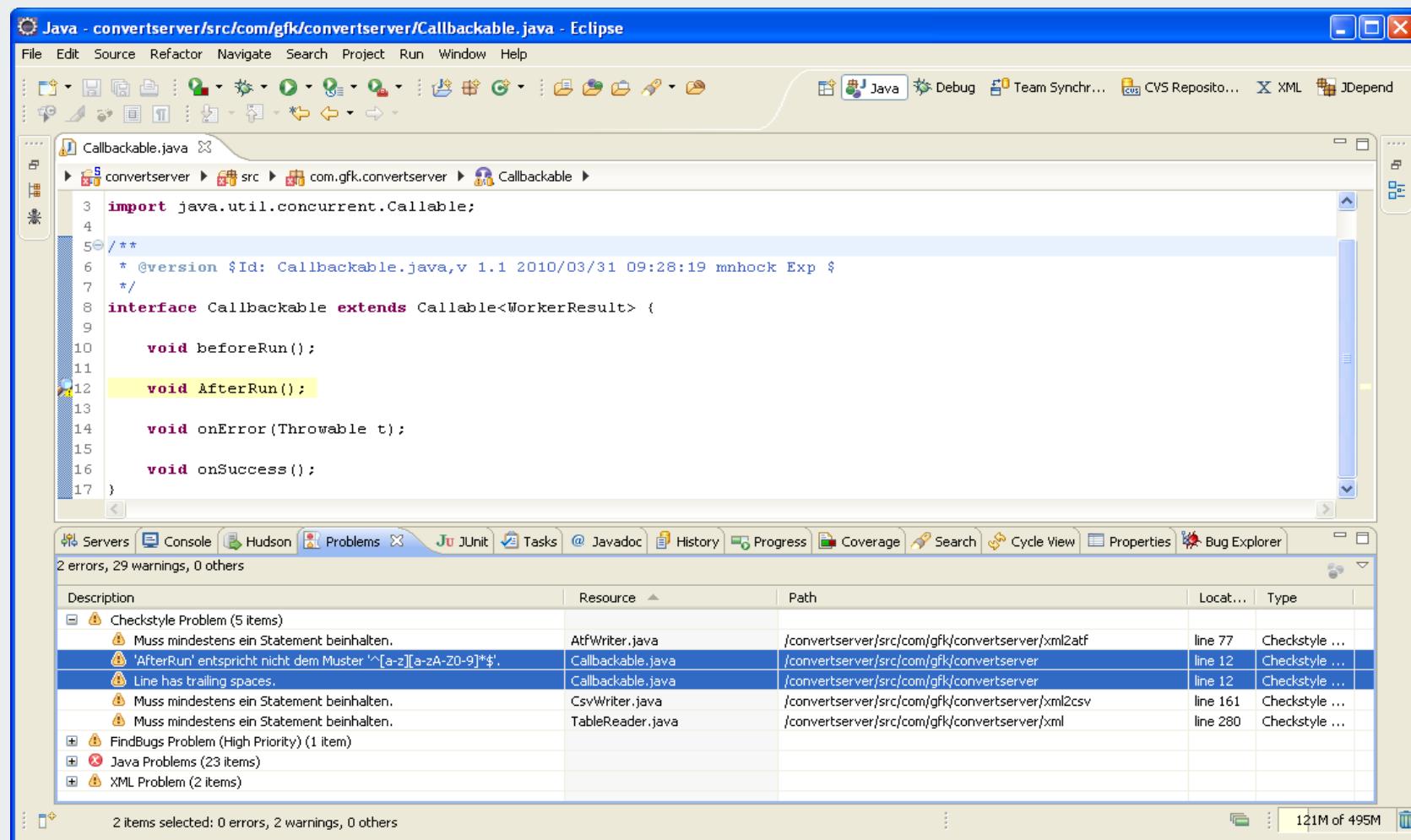
OK

Cancel



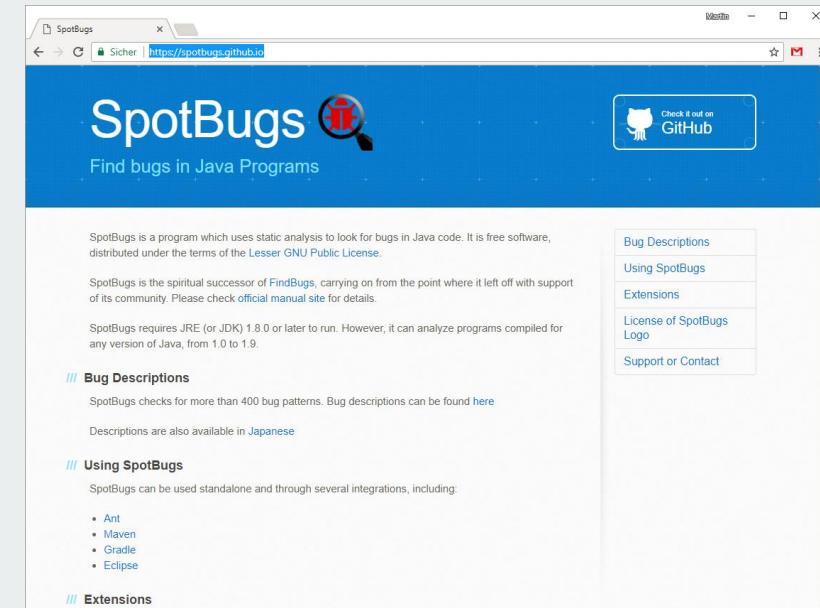
Checkstyle- Eclipse

187



SpotBugs

- Open Source Programm, das statische Code Analyse verwendet um Java Programme auf zahlreiche Bugs zu untersuchen.
- Nachfolger von Findbugs
- SpotBugs basiert auf dem Konzept der Bug Patterns
 - Muster von fehlerhaftem Programmverhalten in Verbindung mit Programmierfehlern.
 - Spotbugs untersucht nun Java Bytecode nach diesen Mustern.
 - Der Quelltext wird nicht benötigt, sondern nur das Kompilat.
 - Plugin Architektur für eigene Bug Detektoren
- <https://spotbugs.github.io/>



SpotBugs - Bug Kategorien

189

Correctness	Offensichtliche Fehler im Code
Bad Practice	Verletzung von empfohlener, grundlegender Codeanwendung
Dodgy	Fragwürdige, verwirrende Codestellen
Multithreaded Correctness	Wie Correctness, nur bezogen auf Multithreading
Performance	Codestellen, die die Performance verschlechtern
Security	Codestellen, die ein Sicherheitsrisiko darstellen
Malicious Code Vulnerability	Verletzbarkeit durch bösartigen Code
Internationalization	z. B. Probleme bei verschiedenen Encodings und internationalen Sonderzeichen
Experimental	Noch nicht endgültige Bug Patterns, werden gerade entwickelt

- Beschreibung von Bugpatterns

<https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html>

SpotBugs

190

- Nicht geworfene oder nicht behandelte Exceptions
- Vergleich von String mit == oder !=
- equals() / hashCode() Probleme
- irreführende Methodennamen
- NPEs
- ungelesene / ungenutzte Felder
- Zweifelhafte Konstrukte
 - instanceof, das immer true ist
 - Speichern in Variablen, die nicht mehr gelesen werden
 - komplizierte oder falsche Inkrementierung in for-Schleife
 - redundante Checks zwischen zwei null Objekten oder einem null Objekt und einem initialisierten
- Rückgabewerte von Methoden
 - Strings sind immutable, Funktionen wie trim() und toLowerCase() geben einen neuen String zurück.

SpotBugs

- NPE

```
// JDK1.6.0, b105, sun.awt.x11.XMSelection
if (listeners == null)
    listeners.remove(listener);
```

```
// com.sun.corba.se.impl.naming.cosnaming.NamingContextImpl
if (name != null || name.length > 0)
```

```
// com.sun.xml.internal.ws.wsdl.parser.RuntimeWSDLParser
if (part == null || part.equals(""))
```

```
// sun.awt.x11.ScrollPanePeer
if (g != null)
    paintScrollBars(g,colors);
g.dispose();
```

- Redundanter NPE Check

```
// java.awt.image.LookupOp
public final WritableRaster filter(Raster src, WritableRaster dst) {
    int dstLength = dst.getNumBands();
    // Create a new destination Raster, if needed
    if (dst == null)
        dst = createCompatibleDestRaster(src);
```

SpotBugs

192

- Endlosschleifen / Rekursive Endlosschleifen

```
// java.lang.annotation.AnnotationTypeMismatchException (Joshua Bloch)
public String foundType() {
    return this.foundType();
}
```

```
public WebSpider() {
    WebSpider w = new WebSpider();
}
```

- SQL Injection

```
String query = "SELECT cc_type, cc_number FROM user_data WHERE last_name = '" + user + "'";
```

- equals () (Gleichheit) / == (Identität)

```
//from Googles code
class MutableDouble {
    private double value;

    public boolean equals(final Object o) {
        return o instanceof MutableDouble && ((MutableDouble) o).doubleValue() == doubleValue();
    }

    public Double doubleValue() {
        return value;
    }
    ...
}
```

SpotBugs

193

- Cross-site scripting

```
public void doGet(HttpServletRequest req, HttpServletResponse res) {  
    ...  
    String target = req.getParameter("url");  
    InputStream in = this.getClass().getResourceAsStream("META-INF/resources/" + target);  
  
    if (in == null) {  
        res.getWriter().println("<p>Unable to locate resource: " + target);  
        return;  
    }  
}
```

- Rückgabe einer Reference auf ein veränderbares Objekt

```
// jdk1.7.0-b59 sun.security.x509.InvalidityDateExtension:  
public static final String DATE = "date";  
  
private Date date;  
  
public Object get(String name) {  
    if (name.equalsIgnoreCase(DATE)) {  
        return date;  
    } else {  
        ...  
    }  
}
```

- Ignorierte Rückgabewerte

```
// Eclipse 3.0.0M8  
String name= workingCopy.getName();  
name.replace('/', '.');
```

SpotBugs - Eclipse

194

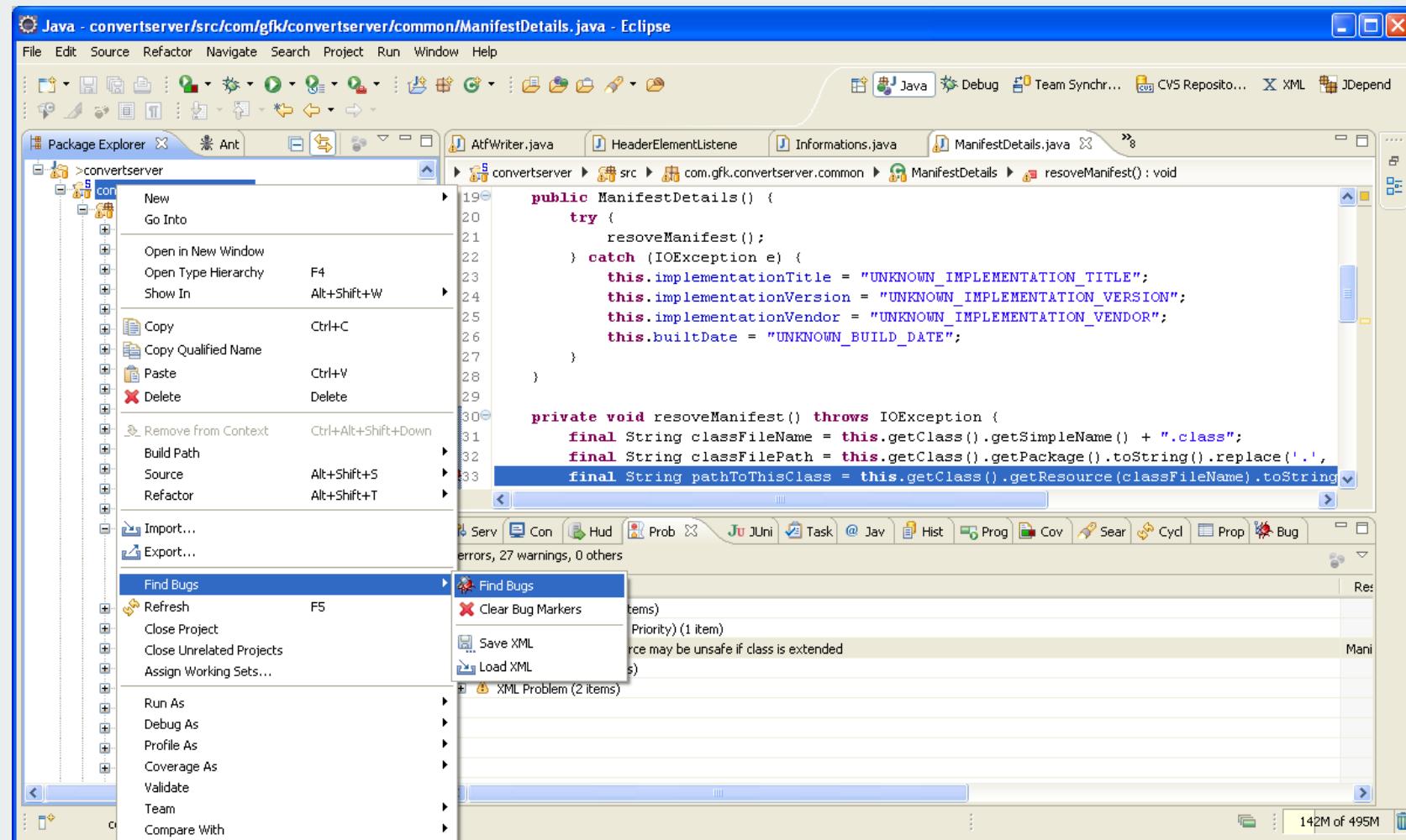
The screenshot shows the Eclipse Preferences dialog for the FindBugs plugin. The left sidebar lists various preferences categories, and the main panel is titled "FindBugs". The "Detector configuration" tab is selected, displaying a table of detectors with their status, pattern(s), speed, and category.

Detector id	Pattern(s)	Speed	Category
AppendingToAnObjectOutputStream	IO	fast	Correctness
BadAppletConstructor	BAC	fast	Correctness
BadResultSetAccess	SQL	fast	Correctness
BadSyntaxForRegularExpression	RE	fast	Correctness
BadUseOfReturnValue	RV	fast	Dodgy
BadlyOverriddenAdapter	BOA	fast	Correctness
BooleanReturnNull	NP	fast	Bad practice
CallToUnsupportedMethod	Dm	fast	Dodgy
CheckImmutableAnnotation	JCIP	fast	Bad practice
CheckTypeQualifiers	TQ	slow	Correctne...
CloneIdiom	CN	fast	Bad practice
ComparatorIdiom	Se	fast	Bad practice
ConfusedInheritance	CI	fast	Dodgy
ConfusionBetweenInheritedAndOuterMethod	IA	moderate	Dodgy
CrossSiteScripting	HRS XSS	fast	Security
DoInsideDoPrivileged	DP	fast	Bad practice
DontCatchIllegalMonitorStateException	IMSE	fast	Bad practice
DontIgnoreResultOfPutIfAbsent	RV	fast	Multithrea...
DontUseEnum	Nm	fast	Bad practice
DroppedException	DE	fast	Bad practice
DumbMethodInvocations	DMDI	fast	Security

Below the table, there is a "Detector details" section and buttons for "OK", "Cancel", and "Restore Defaults".

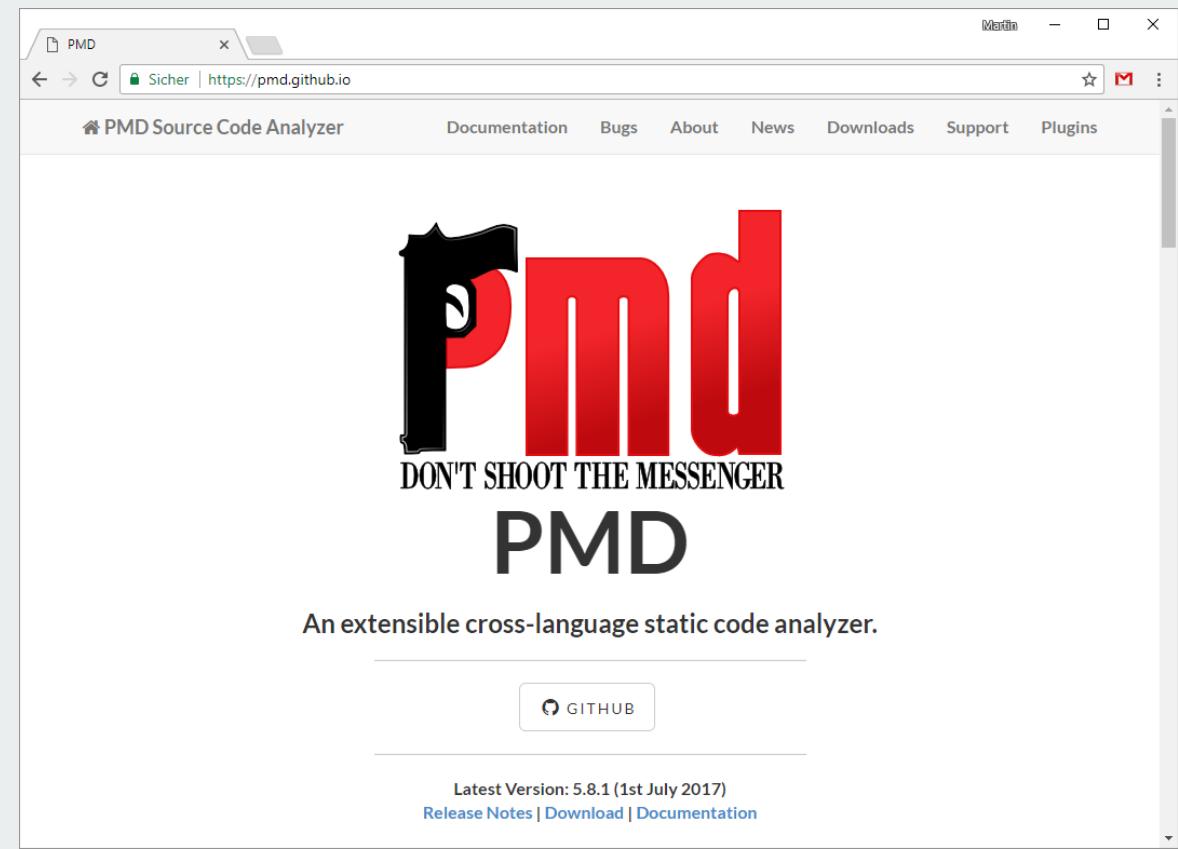
SpotBugs - Eclipse

195



PMD

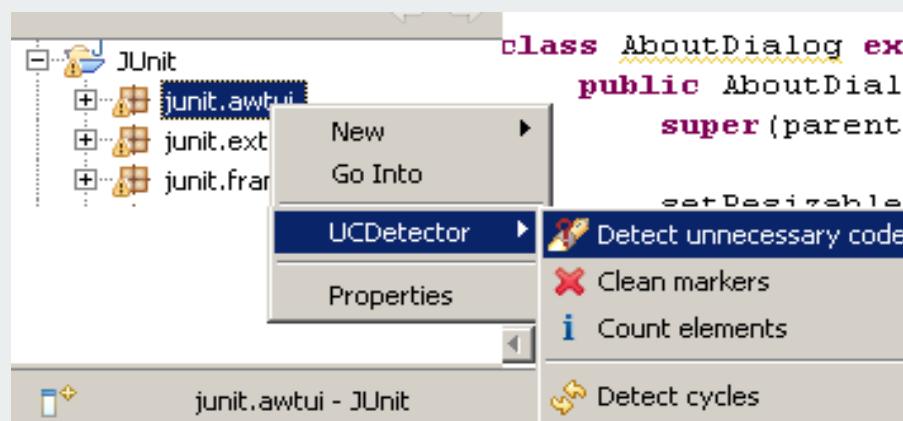
- Open Source Programm, das statische Code Analyse verwendet um Java Programme auf zahlreiche Bugs zu untersuchen.
- Copy / Paste Detector (CPD)
- Unterstützung für ANT, Maven, Eclipse, Netbeans
- <https://pmd.github.io/>



UCDetector - Unnecessary Code Detector

197

- Sucht nach nicht referenzierten Methoden und Klassen
- Ratschläge zur Sichtbarkeit von Methoden und Klassen
- Zeigt Methoden und Felder an, die final sein können
- <http://www.ucdetector.org/>

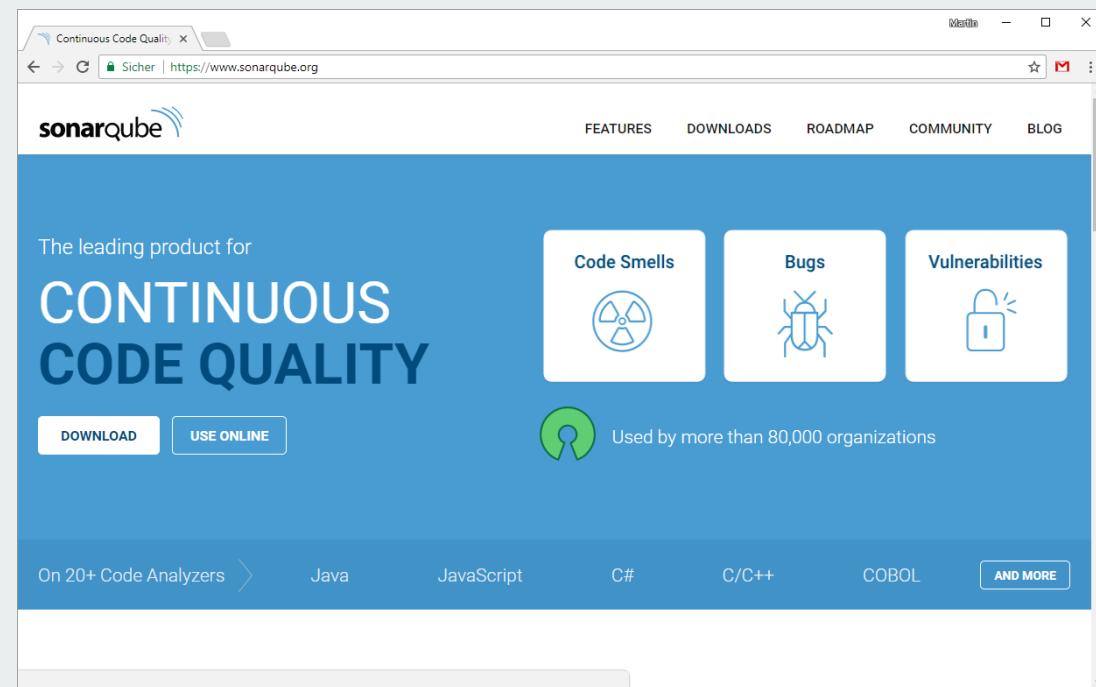


The screenshot shows the Eclipse IDE environment. On the left, the 'MixedExample.java' code editor displays Java code. On the right, the 'Problems' view lists 8 items, each with a warning icon and a detailed description related to UCDetector markers and visibility issues.

Description
UCDetector Marker (Few references) (1 item) Method "MixedExample.usedOnceMethod()" has 1 references
UCDetector Marker (Final) (2 items) Use "final" for Field "MixedExample.readOnlyField" Use "final" for Method "MixedExample.helper()" - apply carefully!
UCDetector Marker (References) (1 item) Constant "MixedExample.UNUSED" has 0 references
UCDetector Marker (Test only) (1 item) Method "MixedExample.usedOnlyByTests()" is only called from tests
UCDetector Marker (Visibility: Use private) (2 items) Change visibility of Field "MixedExample.localField" to private Change visibility of Method "MixedExample.localMethod()" to private

SonarQube

- Plattform für statische Codeanalyse der technischen Qualität von Sourcecode
- Unterstützt neben Java auch die Analyse von andern Programmiersprachen
- Analysiert den Code hinsichtlich verschiedener Qualitätsbereiche und stellt die Ergebnisse grafisch da
 - Softwarearchitektur/design
 - Doppelter Code
 - Modultests
 - Komplexität
 - Potentielle Fehler
 - Kodierrichtlinien
 - Kommentare
- Lokale Installation möglich
- In Java programmiert
- <http://www.sonarqube.org/>



Quellen

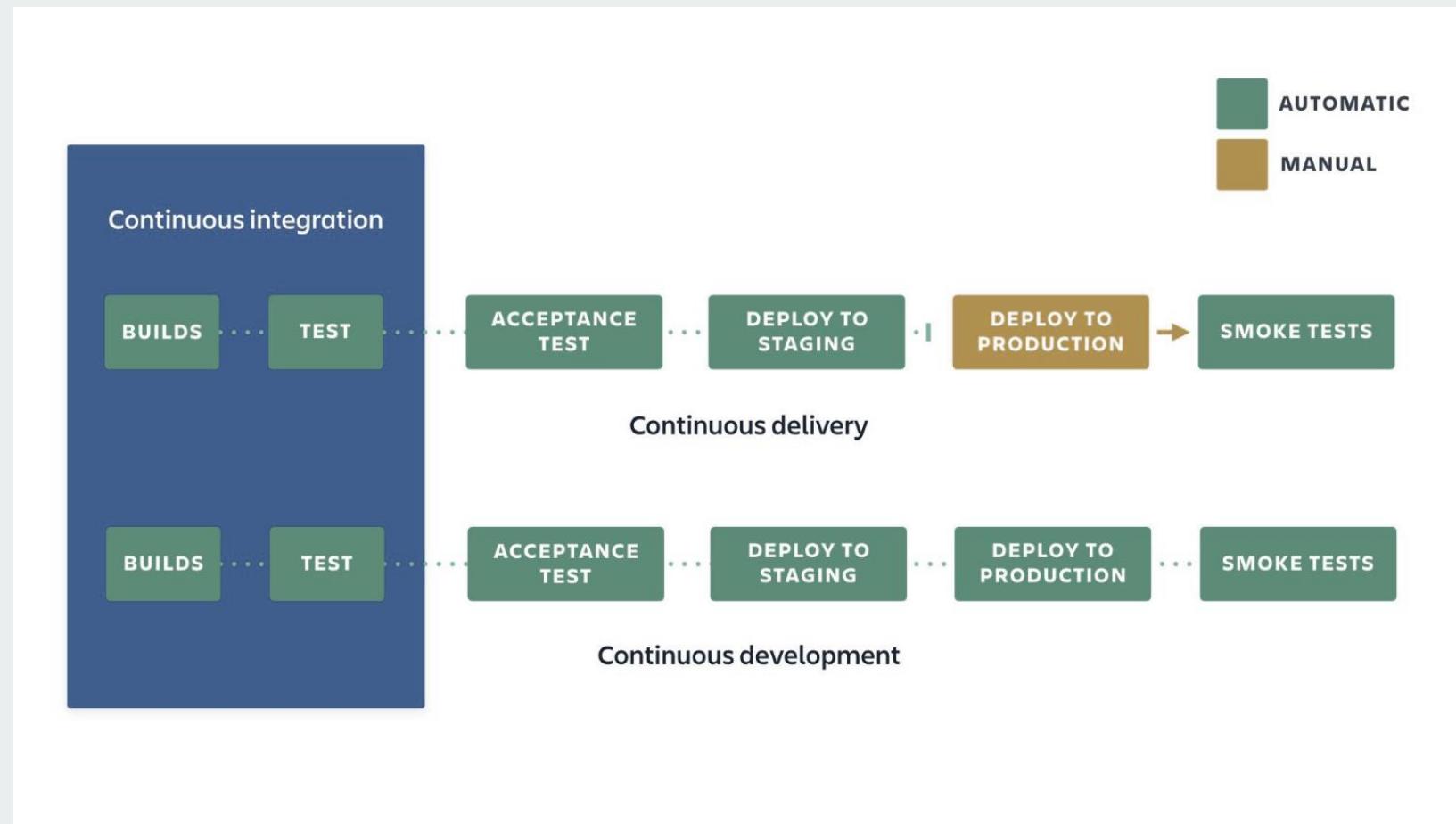
- http://de.wikipedia.org/wiki/Statische_Code-Analyse/
- <http://checkstyle.sourceforge.net/>
- <http://findbugs.sourceforge.net/>
- <http://pmd.sourceforge.net/>
- <https://de.wikipedia.org/wiki/SonarQube>
- <http://www.sonarqube.org/>

Werkzeuge zur Sicherung der Softwarequalität

Continuous Integration (CI)

Continuous Integration / Continuous Delivery / Continuous Deployment (1/4)

201

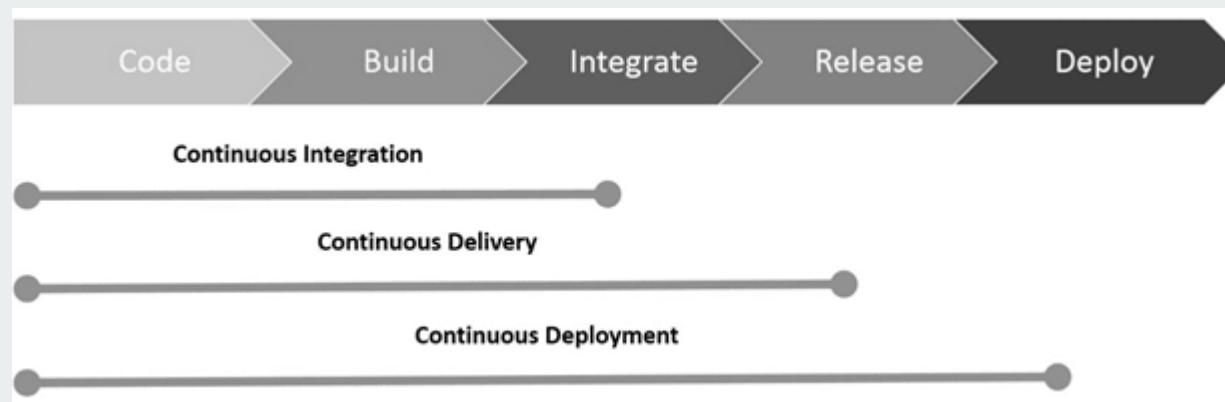


Continuous Integration / Continuous Delivery / Continuous Deployment (2/4)

202

■ Continuous Integration

- Bei der CI werden die Code-Änderungen unmittelbar nach dem Einchecken getestet, so dass die neu aufgenommenen Änderungen kontinuierlich getestet werden können.
- Die Grundidee ist es, den Code im Laufe des Entwicklungsprozesses häufig zu testen, um mögliche Probleme frühzeitig zu erkennen und zu beheben.
- Beim CI wird der Großteil der Arbeit durch automatisierte Tests und einem Build-Server geleistet.
- Da CI eine ständige Rückmeldung über den Softwarestatus gibt, sind die auftretenden Code-Probleme in der Regel weniger komplex und viel einfacher zu lösen.

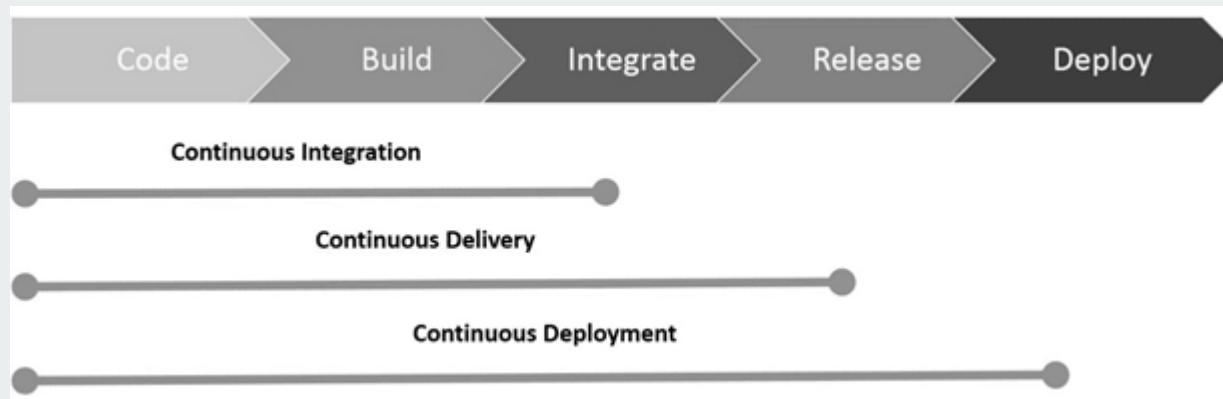


Continuous Integration / Continuous Delivery / Continuous Deployment (3/4)

203

■ Continuous Delivery

- Continuous Delivery ist eine Erweiterung des CI. Dabei wird der entwickelte Code kontinuierlich auf eine Staging Umgebung ausgeliefert, sobald der Entwickler ihn entwickelt hat.
- Es beinhaltet kontinuierliche Integrations-, Testautomatisierungs- und Bereitstellungsautomatisierungsprozesse, die eine schnelle und zuverlässige Entwicklung und Bereitstellung von Software mit geringstem manuellen Aufwand ermöglichen.
- Die Kernidee ist es, den Code an QA, Kunden oder einer beliebigen Benutzerbasis auszuliefern, so dass er ständig und regelmäßig überprüft werden kann.
- Dies ermöglicht es, Probleme frühzeitig im Entwicklungszyklus zu erkennen und sicherzustellen, dass die Behebung frühzeitig erfolgt kann.

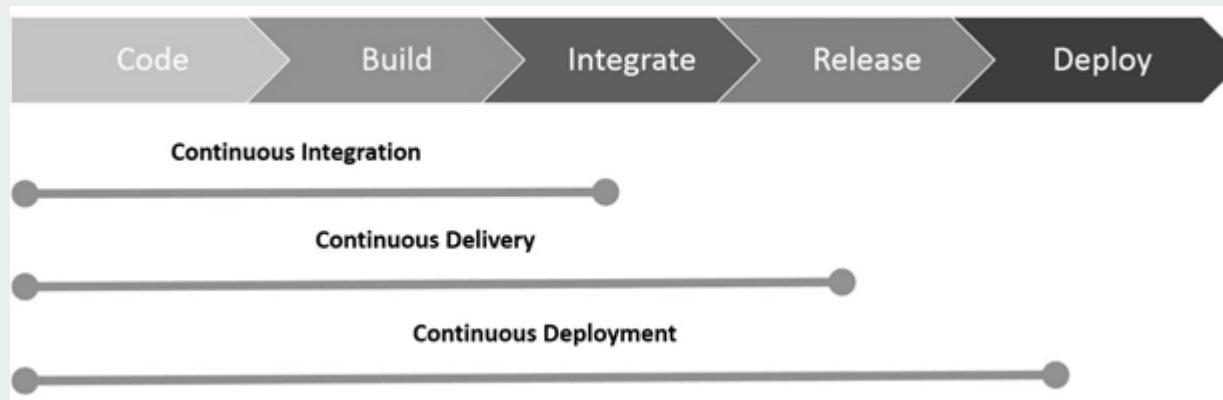


Continuous Integration / Continuous Delivery / Continuous Deployment (4/4)

204

■ Continuous Deployment

- Continuous Deployment ist der nächste logische Schritt nach Continuous Delivery.
- Es ist der Prozess der Bereitstellung des Codes direkt in die Produktion, sobald er entwickelt wurde.
- Bei Continuous Deployment werden alle Änderungen, die die automatisierten Tests bestehen, automatisch in die Produktion übertragen.
- Die erfolgreiche Implementierung erfordert die Automatisierung von Continuous Integration, sowie die Automatisierung von Continuous Delivery in die Staging Umgebung. Schließlich erfordert es auch die Fähigkeit zur automatischen Bereitstellung in die Produktion.

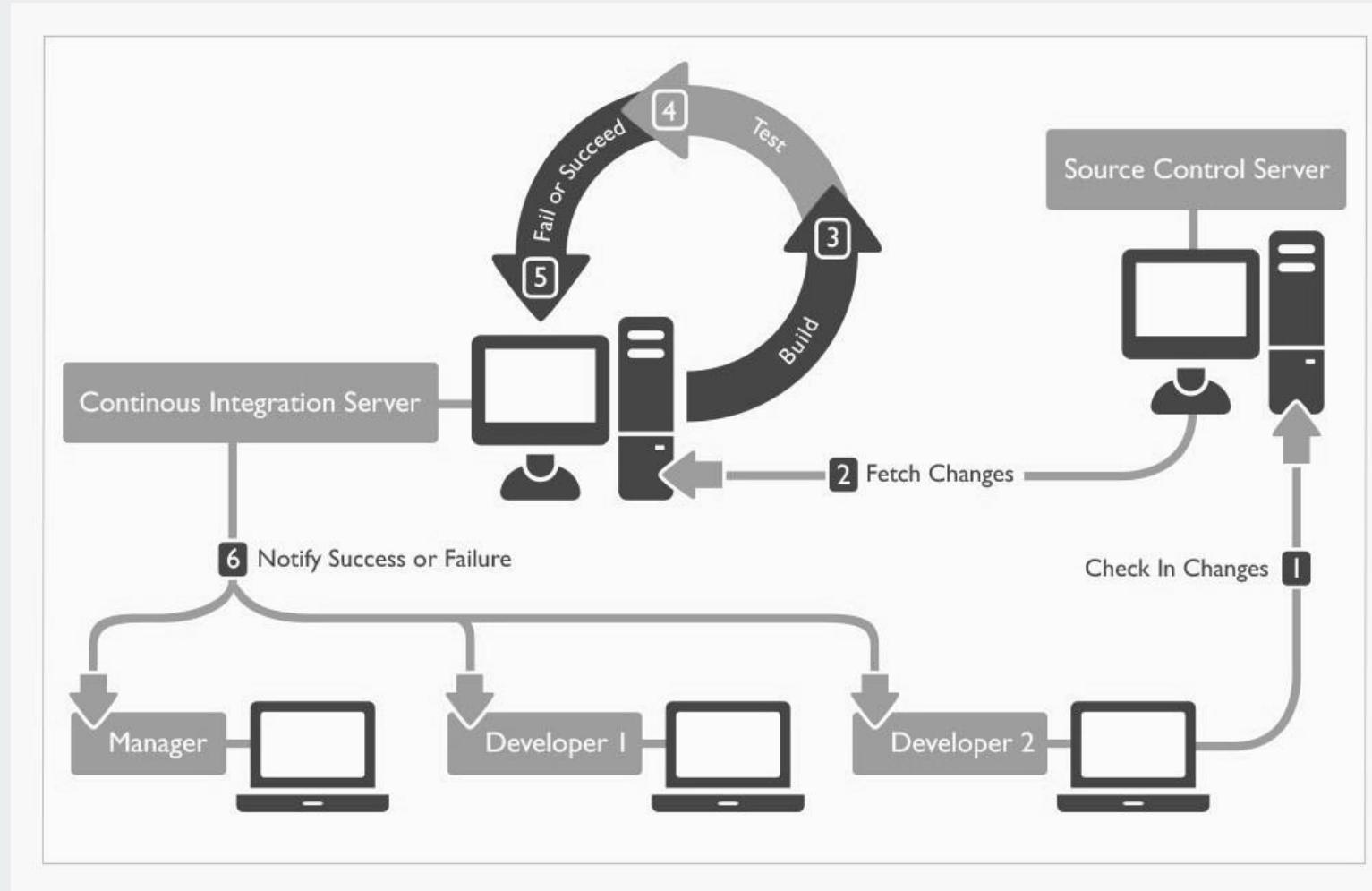


Continuous Integration

- **Kontinuierliche Integration** beschreibt einen Prozess des regelmäßigen, vollständigen Neubildens, Testens und Verteilen einer Anwendung.
- Die Idee:
 - Frühzeitiges und häufigeres Einchecken von Änderungen in die Versionsverwaltung.
 - Ersatz von großen Änderungen durch inkrementelle, funktionsfähige kleine Änderungen
 - Beim Einchecken von Änderungen in die Versionsverwaltung, wird das Gesamtsystem neu gebaut und automatisch getestet.
 - Versuch dem Entwickler so schnell wie möglich Feedback auf seine Änderungen zu geben.
 - Verbesserung der Qualität in dem der traditionelle Ansatz des Testens durch ein kontinuierliches Testen und ein schnelles Feedback ersetzt wird.
- Bestandteile:
 - Kompilierung
 - Test Ausführung (Unit Tests, Akzeptanztests, etc.)
 - Integration (Datenbank, Drittsysteme)
 - Statische Analysen (Code & Architektur)
 - Automatisches Deployment auf Staging / Produktiv Server
 - Generierung von Dokumentation

Vorgehensweise mit Continuous Integration

206



Continuous Integration - Adressierte Risiken

207

- Späte Fehlerbehebung
- Mangelnde Teamabstimmung
 - „Deine Änderung passen nicht mit meinen zusammen“
 - „Hattest Du das nicht bereits vor 2 Monaten gefixt“
- Schlechte Code-Qualität
 - „Wieso machen drei verschiedene Klassen das Gleiche?“
 - „Der Code von Team XY schaut ja ganz anders aus“
- Mangelnde Transparenz / Sichtbarkeit
 - „Welche Tests laufen nicht?“
 - „Was beinhaltet Build 1.2.3?“
 - „Wo stehen wir mit der Code-Abdeckung?“
- Nicht verfügbare Software
 - „Bei mir geht's“
 - „Eigentlich läuft's“
 - „Ich brauche noch einen Build zum Testen“
 - „Morgen kommt der Chef-Chef, wir brauchen eine Demo“

Continuous Integration - Bedingungen (1/2)

208

- Gemeinsame Codebasis
 - Um innerhalb eines Teams sinnvoll integrieren zu können, muss eine Versionsverwaltung existieren, in die alle Entwickler ihre Änderungen kontinuierlich integrieren können.
- Automatisierte Übersetzung
 - Jede Integration muss einheitlich definierte Tests durchlaufen, bevor die Änderungen integriert werden.
 - Dafür ist automatisierte Übersetzung notwendig.
- Kontinuierliche Test-Entwicklung
 - Jede Änderung sollte möglichst mit einem dazugehörigen Test entwickelt werden.
 - Neu entwickelte Tests sollten Bestandteil der automatisierten Test-Suite werden.
 - Einsatz von Code Coverage um die Testabdeckung zu dokumentieren und kontrollieren.
- Häufige Integration
 - Änderungen so oft wie möglich in die gemeinsame Code-Basis integrieren.
 - Risiko von fehlschlagenden Integrationen wird mit kleinen Inkrementen minimiert.
 - Arbeit der Entwickler wird häufig in die gemeinsame Code-Basis gesichert.

Continuous Integration - Bedingungen (2/2)

- Kurze Testzyklen
 - Test-Zyklus vor der Integration sollte kurz gehalten sein, um häufige Integrationen zu fördern.
- Gespiegelte Produktionsumgebung
 - Die Änderungen sollten in einem Abbild der realen Produktionsumgebung getestet werden.
- Einfacher Zugriff
 - Auch Nicht-Entwickler brauchen einfachen Zugriff auf die Ergebnisse der Software-Entwicklung
 - Qualitätszahlen für Qualitäts-Verantwortliche
 - Dokumentation
 - fertig paketiertes Abbild für Release Manager
- Automatisiertes Reporting
 - Wann wurde die letzte erfolgreiche Integration ausgeführt?
 - Welche Änderungen wurden seit der letzten Lieferung eingebracht?
 - Welche Qualität hat die Version?
- Automatisierte Verteilung

Continuous Integration

210

- Vorteile
 - Integrations-Probleme werden laufend entdeckt und behoben, nicht erst kurz vor einem Meilenstein.
 - Frühe Warnungen bei nicht zusammenpassenden Bestandteilen
 - Schnelles Entdecken von Fehlern durch sofortiges Ausführen von Unit-Tests
 - Konstante Verfügbarkeit eines lauffähigen Standes für Demo-, Test- oder Vertriebszwecke
 - Schnelles Feedback auf das Einchecken von fehlerhaften oder unvollständigen Code
 - Frühzeitiges und häufiges Testen, Fehler werden früher gefunden
 - Tests finden parallel zur Entwicklung statt
 - Behebung der Fehler zum frühesten, günstigsten Zeitpunkt
 - Möglichkeit auf Schlüsselmetriken sofort zu reagieren
 - Überprüfung von „Coding Standards“ und „Best Practices“
- Nachteile
 - Einrichten der Infrastruktur
 - Kosten für Hardware, Software

Continuous Integration - Best Practices

211

- Verwendung einer Versionsverwaltung
- Früh und oft einchecken
- Keinen Code einchecken, der nicht läuft
- Build Fehler sofort beheben
- Probleme früh angehen und schnell scheitern
- Aufgrund von Metriken (re)agieren
- Auf allen Zielplattformen bauen
- Artefakte für jeden Build erstellen
- Schnelle Builds

Quellen

- <http://www.martinfowler.com/articles/continuousIntegration.html/>
- http://de.wikipedia.org/wiki/Kontinuierliche_Integration/
- http://www.infomar.de/de/consulting/Im_Fluss_bleiben_JAX_2009.pdf/

Prinzipien Objektorientierten Designs

Warum Designprinzipien?

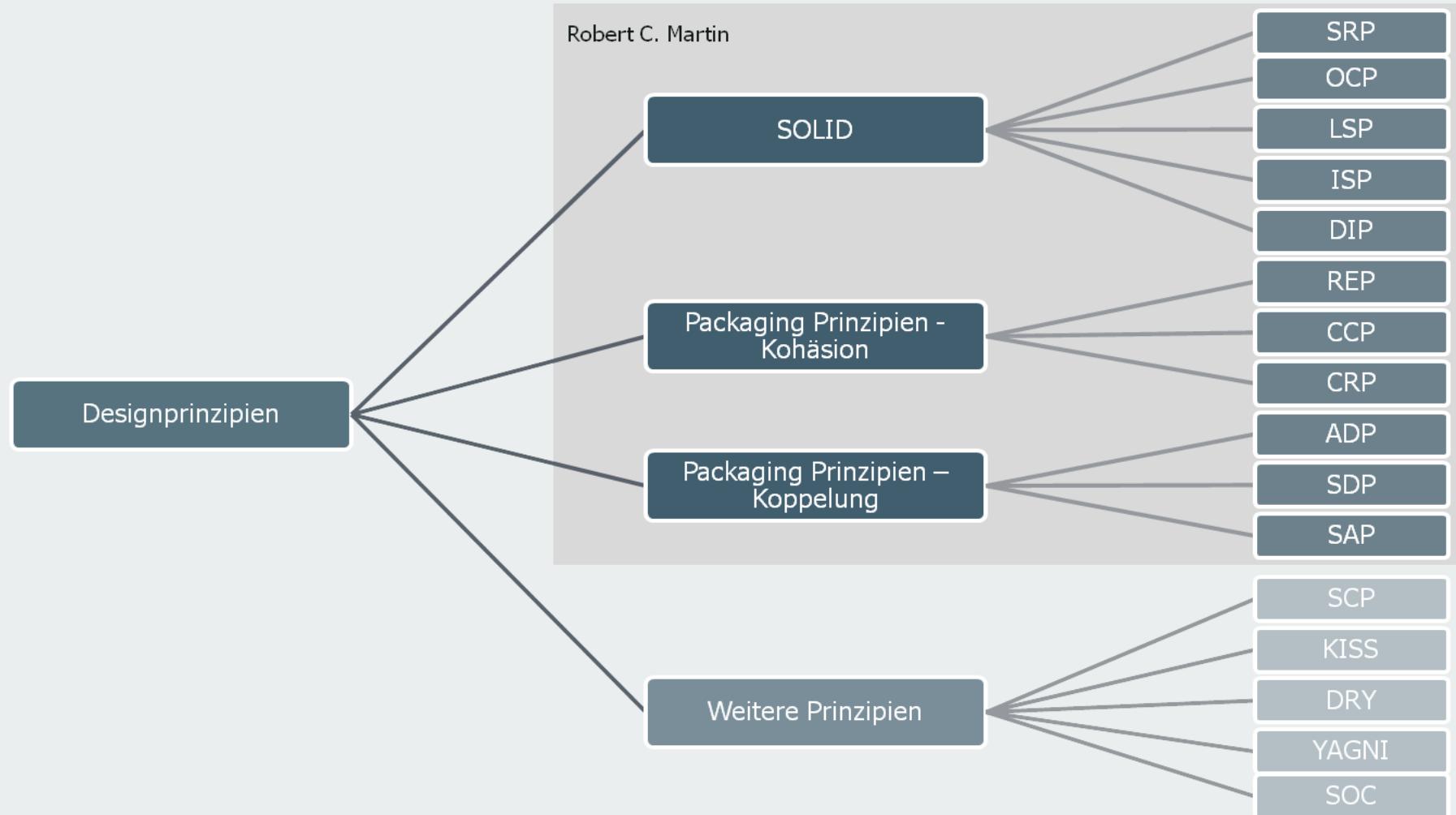
214

„Code muss **KISS**, **DRY** und **YAGNI** sein!“

- Prinzipien Objektorientierten Designs sollen zu gutem objektorientierten Design führen.
- Code- und Architektur-Smells beruhen auf der Verletzung von anerkannten Entwurfsprinzipien.
- Liefern wichtige Hinweise wie Code- und Architektur-Smells behoben werden können.
- Kann das verletzte Entwurfsprinzip identifiziert werden, gibt es das einen ersten Hinweis darauf, wie eine bessere Struktur für das System aussehen könnte.
- Wurden neben anderen von Robert C. Martin, Bertrand Meyer und Barbara Liskov publiziert und propagiert.
- <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

Überblick der Designprinzipien

215



SOLID Prinzipien

216

- Für eine Gruppe dieser Prinzipien wurde von Robert C. Martin das Akronym **SOLID** geprägt.
- Diese Prinzipien gemeinsam angewandt führt lt. Robert C. Martin zu einer höheren Wartbarkeit und somit Lebensdauer von Software.

S	SRP	Single Responsibility Principle	<i>A class should have one, and only one, reason to change.</i>
O	OCP	Open Closed Principle	<i>You should be able to extend a classes behavior, without modifying it.</i>
L	LSP	Liskov Substitution Principle	<i>Derived classes must be substitutable for their base classes.</i>
I	ISP	Interface Segregation Principle	<i>Make fine grained interfaces that are client specific.</i>
D	DIP	Dependency Inversion Principle	<i>Depend on abstractions, not on concretions.</i>

Packaging Prinzipien

217

- Beschäftigen sich mit der Frage, wie man Klassen zu Modulen (Packages) zusammenführen sollte.
- Führen insbesondere zu einer **hohen Kohäsion** innerhalb der Module und **geringen Kopplung** zwischen den Modulen.
- Packaging Prinzipien - **Kohäsion**

REP	Release Reuse Equivalency Principle	<i>The granule of reuse is the granule of release.</i>
CCP	Common Closure Principle	<i>Classes that change together are packaged together.</i>
CRP	Common Reuse Principle	<i>Classes that are used together are packaged together.</i>

- Packaging Prinzipien - **Kopplung**

ADP	Acyclic Dependencies Principle	<i>The dependency graph of packages must have no cycles.</i>
SDP	Stable Dependencies Principle	<i>Depend in the direction of stability.</i>
SAP	Stable Abstractions Principles	<i>Abstractness increases with stability.</i>

Weitere Prinzipien des Objektorientierten Designs

218

SCP	Speaking Code Principle	<i>Writing code that speaks for itself and that does not need a comment.</i>
KISS	Keep It Simple (and) Stupid	<i>Simplicity should be a key goal in design, and unnecessary complexity should be avoided.</i>
DRY	Don't Repeat Yourself	<i>Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.</i>
YAGNI	You Ain't Gonna Need It!	<i>Always implement things when you actually need them, never when you just foresee that you need them.</i>
SOC	Separation Of Concerns	<i>Process of separating a program into distinct features that overlap in functionality as little as possible.</i>

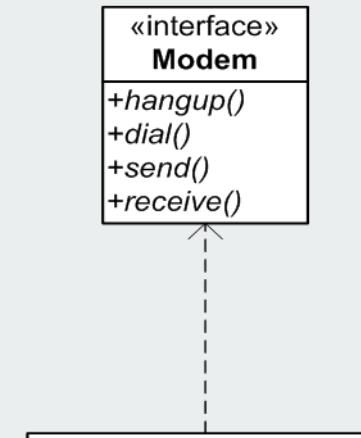
SOLID Designprinzipien - SRP (1/3)

- SRP: Single Responsibility Principle
 - „*A class should have one, and only one, reason to change.*“
 - Jede Klasse hat nur **eine** fest definierte Aufgabe zu erfüllen.
 - In einer Klasse sollten lediglich Funktionen vorhanden sein, die direkt zur Erfüllung dieser Aufgabe beitragen.
 - Es sollte einen und nur **einen Grund** geben, eine Klasse zu ändern.
 - Eine der einfachsten Möglichkeiten, Klassen zu schreiben, die nie geändert werden müssen, besteht darin, Klassen zu schreiben, die nur einem Zweck dienen. Dann muss eine Klasse nur dann geändert werden, wenn sich genau das, was die Klasse tut, ändert.
 - Jede Klasse sollte einen zusammenhängenden Satz verwandter Funktionen implementieren.
 - Eine einfache Möglichkeit, dem Prinzip der einzigen Verantwortung zu folgen, besteht darin, sich immer wieder zu fragen, ob jede Methode und jeder Vorgang einer Klasse in direktem Zusammenhang mit dem Namen dieser Klasse steht.
 - Bei Methoden die nicht zum Namen der Klasse passen, diese Methoden in andere Klassen verschieben.

SOLID Designprinzipien - SRP (2/3)

220

```
public interface Modem {  
  
    // Connection management responsibilities  
    void dial(String phoneNumber);  
    void hangup();  
  
    // Data management responsibilities  
    void send(char c);  
    char receive();  
}
```

A small, simple sad face icon consisting of two dots for eyes and a downward-curving line for a mouth, positioned to the right of the implementation class.

Ohne SRP

SOLID Designprinzipien - SRP (3/3)

221

```
public interface DataChannel {  
    void send(char c);  
    char receive();  
}
```

```
public class Modem implements Connection, DataChannel {  
    private final Connection connection;  
    private final DataChannel dataChannel;  
  
    public Modem(Connection connection, DataChannel dataChannel) {  
        this.connection = connection;  
        this.dataChannel = dataChannel;  
    }  
  
    @Override  
    public void dial(String phoneNumber) {  
        this.connection.dial(phoneNumber);  
    }  
  
    @Override  
    public void hangup() {  
        this.connection.hangup();  
    }  
  
    @Override  
    public void send(char c) {  
        this.dataChannel.send(c);  
    }  
  
    @Override  
    public char receive() {  
        return dataChannel.receive();  
    }  
}
```

```
public interface Connection {  
    void dial(String phoneNumber);  
    void hangup();  
}
```



Mit SRP

SOLID Designprinzipien - OCP (1/7)

222

- OCP: Open Closed Principle
 - „*You should be able to extend a classes behavior, without modifying it.*“
 - Klassen, die dem OCP gehorchen, verfügen über zwei wesentliche Eigenschaften:
 - **offen gegenüber Erweiterungen**
 - ➔ Das Verhalten solcher Klasse kann erweitert werden, um neue Anforderungen einer bestehenden oder gar neuen Anwendung zu erfüllen.
 - **geschlossen gegenüber nachträglicher Veränderung**
 - ➔ Müssen nicht mehr modifiziert werden, um als Basis für neue Anforderungen dienen zu können.
 - Funktionen die unterschiedliche Aktionen aufgrund von **Typswitches** durchführen, sind gute Beispiele für die Verletzung des OCP. Solche Funktionen sind **nie gegen Veränderungen geschlossen**, da die Hinzunahme eines neuen Typs die Änderung des Quellcodes der Funktion nötig macht.
 - Es ist eleganter und robuster, einen Klassenverbund durch Hinzufügen einer Klasse zu erweitern, als den bestehenden Quellcode zu modifizieren.
 - Durch die Verwendung von z. B. **abstrakten Basisklassen** können Softwarebauelemente erstellt werden, die zwar eine feste unveränderliche Implementation besitzen, deren Verhalten aber durch Vererbung und Polymorphie unbegrenzt veränderbar ist.

SOLID Designprinzipien - OCP (2/7)

223

Verletzung gegen OCP

```
public class LoanRequestHandler {  
    private int balance;  
    private int period;  
  
    public LoanRequestHandler(int balance, int period) {  
        this.balance = balance;  
        this.period = period;  
    }  
  
    public void approveLoan(PersonalLoanValidator validator) {  
        if(validator.isValid(balance))  
            System.out.println("Loan approved...");  
        else  
            System.out.println("Sorry not enough balance...");  
    }  
}
```



```
public class PersonalLoanValidator {  
  
    public boolean isValid(int balance) {  
        return balance > 1000 ? true : false;  
    }  
}
```

SOLID Designprinzipien - OCP (3/7)

224

Keine Verletzung gegen OCP

```
public class LoanRequestHandler {  
    private int balance;  
    private int period;  
  
    public LoanRequestHandler(int balance, int period) {  
        this.balance = balance;  
        this.period = period;  
    }  
  
    public void approveLoan(Validator validator) {  
        if(validator.isValid(balance))  
            System.out.println("Loan approved...");  
        else  
            System.out.println("Sorry not enough balance...");  
    }  
}
```



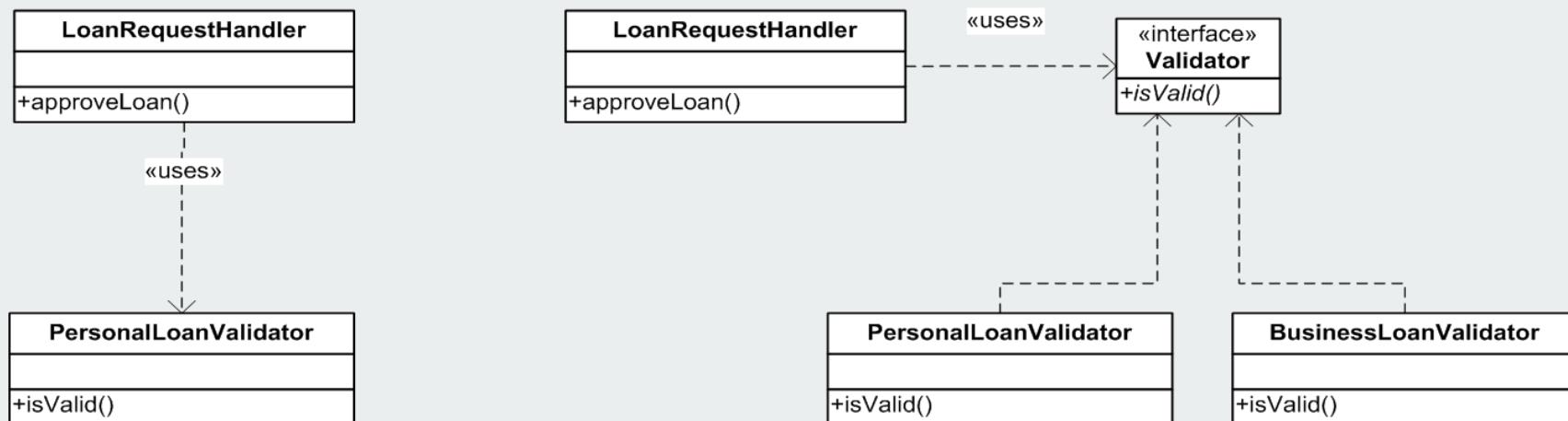
```
public interface Validator {  
    public boolean isValid(int balance);  
}
```

```
public class PersonalLoanValidator implements Validator {  
    public boolean isValid(int balance) {  
        return balance > 1000 ? true : false;  
    }  
}
```

```
public class BusinessLoanValidator implements Validator {  
    public boolean isValid(int balance) {  
        return balance > 5000 ? true : false;  
    }  
}
```

SOLID Designprinzipien - OCP (4/7)

225



Verletzung gegen OCP

Verletzung gegen OCP aufgelöst mit Strategy Pattern



SOLID Designprinzipien - OCP (5/7)

226

Verletzung gegen OCP

```
public void draw(Shape[] shapes) {  
    for(Shape shape : shapes) {  
        switch (shape.getType()) {  
            case Shape.SQUARE:  
                draw((Square) shape);  
                break;  
            case Shape.CIRCLE:  
                draw((Circle) shape);  
                break;  
        }  
    }  
}
```



Keine Verletzung des OCP

```
public void draw(Shape[] shapes) {  
    for(Shape shape : shapes) {  
        shape.draw();  
    }  
}
```



SOLID Designprinzipien - OCP (6/7)

227

```
public class HumanResourceDepartment {  
  
    private List<Developer> developers;  
    private List<Manager> managers;  
  
    public void hire(Developer developer) {  
        developer.signContract();  
        developers.add(developer);  
    }  
  
    public void hire(Manager manager) {  
        manager.signContract();  
        managers.add(manager);  
    }  
}
```



SOLID Designprinzipien - OCP (7/7)

228

```
interface Employee {  
    void signContract();  
}
```

```
class Developer implements Employee {  
    public void signContract() {  
    }  
}
```

```
class Manager implements Employee {  
    public void signContract() {  
    }  
}
```

```
class Secretary implements Employee {  
    public void signContract() {  
    }  
}
```



```
public class HumanResourceDepartment {  
    private List<Employee> employees;  
  
    public void hire(Employee employee) {  
        employee.signContract();  
        this.employees.add(employee);  
    }  
}
```

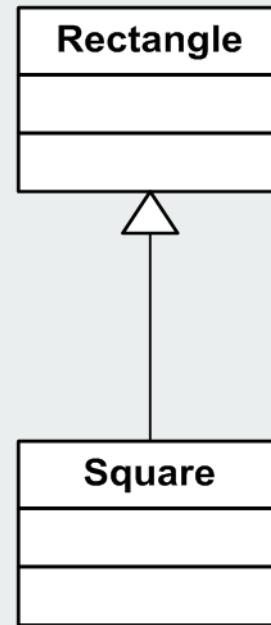
SOLID Designprinzipien - LSP (1/2)

- LSP: Liskov Substitution Principle
 - „*Derived classes must be substitutable for their base classes.*“
 - Bei einem sauberen OOD repräsentiert die Ableitungsbeziehung zwischen Super- und Subklasse eine sogenannte „**is-a**“-Beziehung, d. h. ein Objekt der Subklasse ist vom Typ her kompatibel zu einem Superklassenobjekt und kann überall dort verwendet werden, wo ein Objekt der Superklasse verlangt wird.
 - ➔ Liskov Substitution Principle
 - „*Sei $q(x)$ eine Eigenschaft des Objektes x vom Typ T , dann sollte $q(y)$ für alle Objekte y des Typs S gelten, wo S ein Subtyp von T ist.*“
 - ➔ Damit ist garantiert, dass Operationen vom Typ Superklasse, die auf ein Objekt des Typs Subklasse angewendet werden, auch korrekt ausgeführt werden.
 - ➔ Dann lässt sich stets bedenkenlos ein Objekt vom Typ Superklasse durch ein Objekt vom Typ Subklasse ersetzen.
 - Bei Vererbung nicht nur auf die Daten achten, sondern auch das Verhalten der Methoden einer Klasse berücksichtigen.

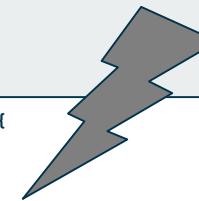
SOLID Designprinzipien - LSP (2/2)

230

```
public class Rectangle {  
    private double height;  
    private double width;  
  
    public double getHeight() {  
        return this.height;  
    }  
  
    public double getWidth() {  
        return this.width;  
    }  
  
    public void setHeight(double height) {  
        this.height = height;  
    }  
  
    public void setWidth(double width) {  
        this.width = width;  
    }  
  
    public double getArea() {  
        return this.width * this.height;  
    }  
}  
  
public class Square extends Rectangle {  
  
    @Override  
    public double getArea() {  
        return this.getHeight() * this.getHeight();  
    }  
}
```



```
public void do(Rectangle rectangle) {  
    rectangle.setWidth(5);  
    rectangle.setHeight(4);  
  
    if (rectangle.getArea() != 20)  
        throw new IllegalStateException("Error in area calculation!");  
}
```



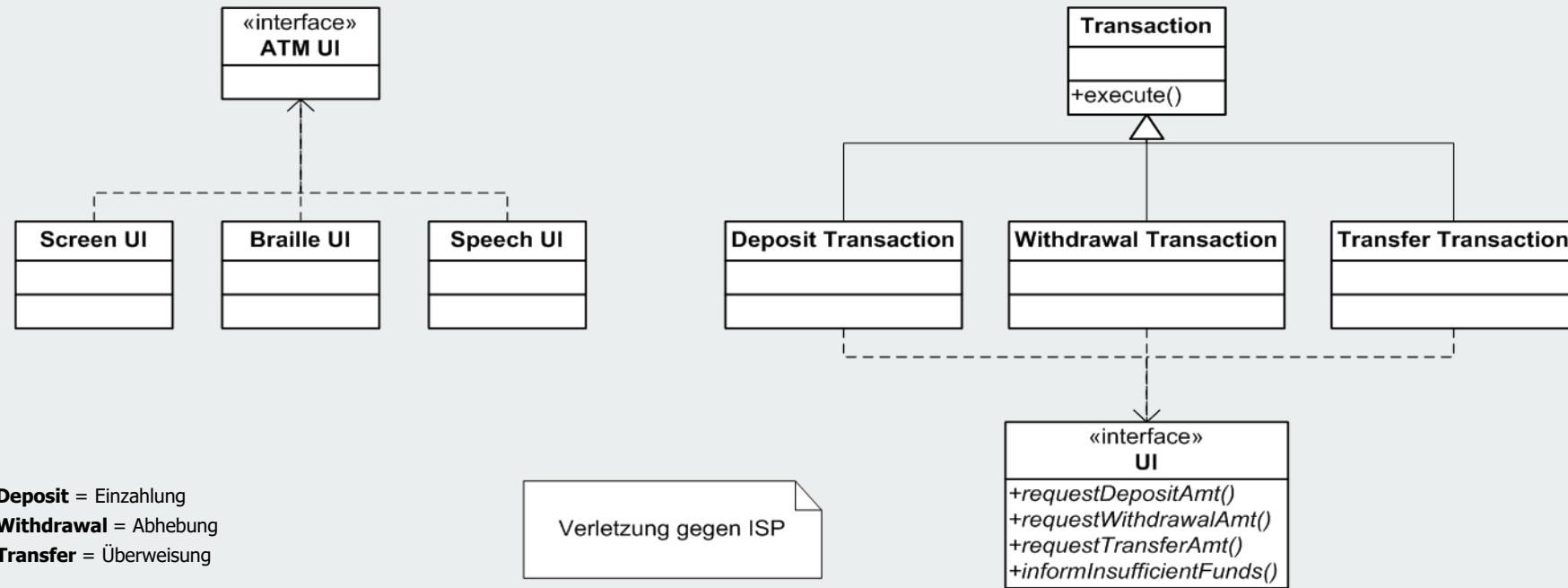
SOLID Designprinzipien - ISP (1/5)

231

- ISP: Interface Segregation Principle
 - „*Make fine grained interfaces that are client specific.*“
 - Vermeidung von „Interface Pollution“
 - schmale Schnittstellen
 - Methoden einzelner Schnittstellen sollten eine **geringe Kopplung** besitzen.
 - Clients sollten nicht von Methoden abhängen, die gar nicht gebraucht werden.
 - Aufteilung von zu große Interfaces
 - Aufteilung soll gemäß der Anforderungen der Clients an die Interfaces gemacht werden .
 - Clients müssen also nur mit Interfaces agieren, die nur das können, was die sie benötigen.
 - Ermöglicht eine Software in entkoppelte und somit flexible Klassen aufzuteilen.
 - Zukünftige fachliche oder technische Anforderungen benötigen somit nur geringe Änderungen an der Software selbst.

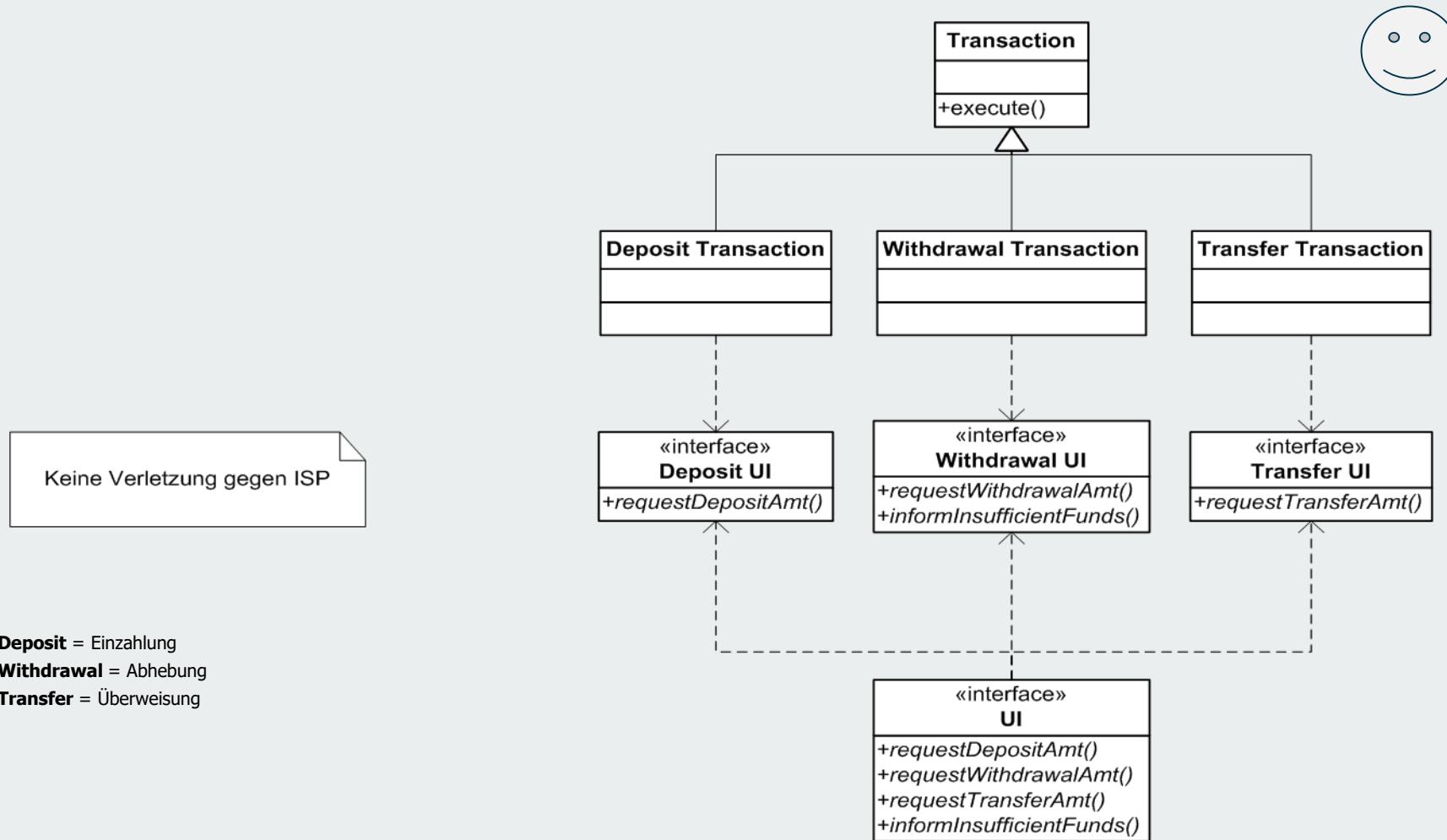
SOLID Designprinzipien - ISP (2/5)

232



SOLID Designprinzipien - ISP (3/5)

233



SOLID Designprinzipien - ISP (4/5)

234

```
interface MultiFunctionDevice {  
  
    void print();  
  
    void fax();  
  
    void scan();  
}
```



```
class AllInOnePrinter implements MultiFunctionDevice {  
  
    @Override  
    public void print() {  
    }  
  
    @Override  
    public void fax() {  
    }  
  
    @Override  
    public void scan() {  
    }  
}
```

```
class InkjetPrinter implements MultiFunctionDevice {  
  
    @Override  
    public void print() {  
    }  
  
    @Override  
    public void fax() {  
        throw new NotSupportedException();  
    }  
  
    @Override  
    public void scan() {  
        throw new NotSupportedException();  
    }  
}
```

SOLID Designprinzipien - ISP (5/5)

235

```
interface Printer {  
    void print();  
}
```

```
interface Fax {  
    void fax();  
}
```

```
interface Scanner {  
    void scan();  
}
```



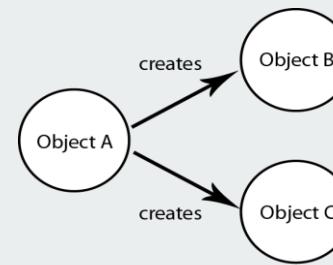
```
class InkjetPrinter implements Printer {  
  
    @Override  
    public void print() {  
    }  
}
```

```
class AllInOnePrinter implements Printer, Fax, Scanner {  
  
    @Override  
    public void print() {  
    }  
  
    @Override  
    public void fax() {  
    }  
  
    @Override  
    public void scan() {  
    }  
}
```

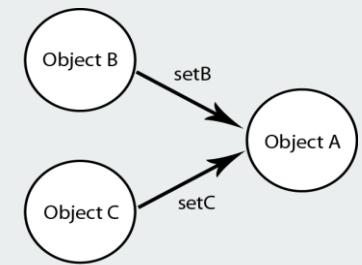
SOLID Designprinzipien - DIP (1/5)

236

- DIP: Dependency Inversion Principle
 - „*Depend on abstractions, not on concretions.*“
 - Abhängigkeiten sollten immer von konkreteren Modulen **niedriger Ebenen** zu abstrakten Modulen **höherer Ebenen** gerichtet sein.
 - Keine Klasse sollte fremde Klassen instanziieren, sondern diese als Abstraktionen (z. B. Interfaces) in Form eines Parameters bekommen.
 - Damit ist sichergestellt, dass die Abhängigkeitsbeziehungen immer in eine Richtung verlaufen, von den konkreten zu den abstrakten Modulen, von den abgeleiteten Klassen zu den Basisklassen.
 - Hollywood-Principle „*Don't call us, we'll call you*“
 - Abhängigkeiten zwischen den Objekten werden von **außen** gesteuert.
 - ➔ Reduktion von Abhängigkeiten zwischen den Modulen
 - ➔ Vermeidung von zyklische Abhängigkeiten
 - ➔ Entkopplung der Komponente von ihrer Umgebung
 - ➔ Reduktion des notwendigen Wissens



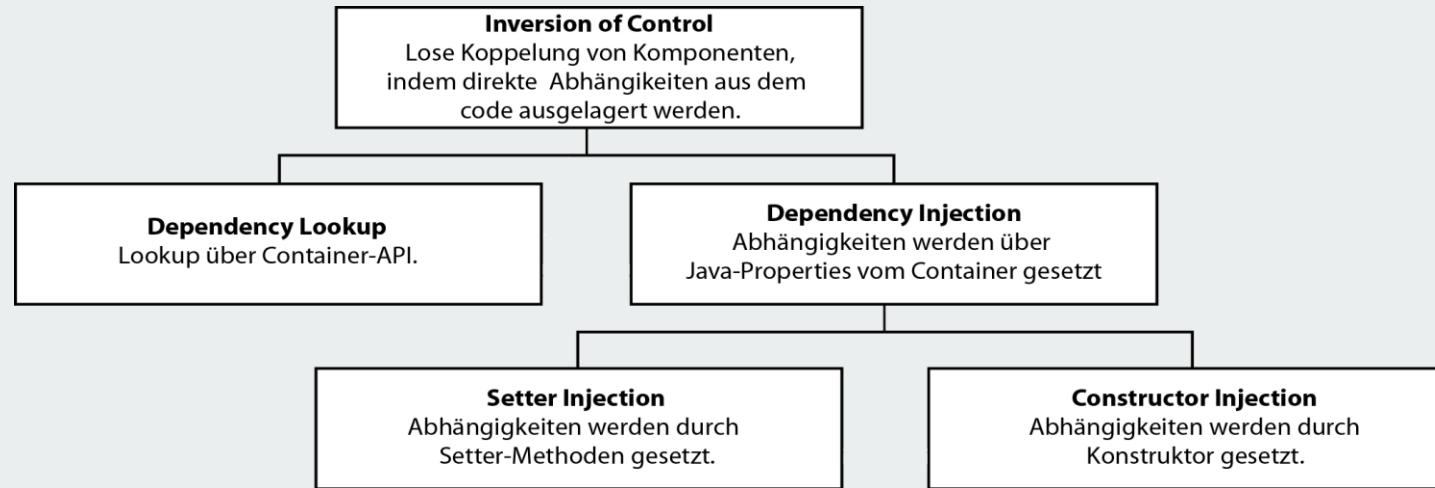
Objects arranged without IoC



Objects arranged with IoC

SOLID Designprinzipien - DIP (2/5)

237



- **Dependency Lookup**
 - Container stellt den Komponenten einen Lookup-Kontext zur Verfügung.
 - Komponenten benutzen diesen, um ihre abhängigen Ressourcen selbst zu lokalisieren.
 - Container beschränkt sich hierbei lediglich auf die Initialisierung des Kontextes.
- **Dependency Injection**
 - Abhängigkeiten werden in die Komponenten von außen injiziert.
 - Komponenten stellen hierfür setter()-Methoden (Setter Injection) oder spezifischen Konstruktoren (Constructor Injection) bereit.

SOLID Designprinzipien - DIP (3/5)

Ohne DI müsste das UserRepository direkt im UserService erzeugt werden

```
public class UserService {  
    private UserRepository userRepository;  
  
    public UserService() {  
        userRepository = new UserRepositoryHibernate();  
    }  
}
```

UserRepository wird per Setter Injection bei der Erzeugung des Objektes gesetzt

```
public class UserService {  
    private UserRepository userRepository;  
  
    public void setUserRepository(UserRepository userRepository) {  
        userRepository = userRepository;  
    }  
}
```

Konfigurationsdatei für das Spring Framework

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"  
"http://www.springframework.org/dtd/springbeans.dtd">  
<beans>  
    <bean id="userService" class="UserService" singleton="true">  
        <property name="userRepository">  
            <refbean = "userRepository" />  
        </property>  
    </bean>  
    <bean id="userRepository" class="dao.hibernate.UserRepositoryHibernate" />  
</beans>
```

SOLID Designprinzipien - DIP (4/5)

239

```
class Logger {  
    private final FileSystem fileSystem;  
  
    public Logger() {  
        this.fileSystem = new FileSystem();  
    }  
  
    public void log(String message) {  
        this.fileSystem.write(message);  
    }  
}
```



SOLID Designprinzipien - DIP (5/5)

240

```
public interface Loggable {  
    void log(String message);  
}
```

```
class FileSystem implements Loggable {  
    @Override  
    public void log(String message) {  
        //file handling and writing  
    }  
}
```

```
class Database implements Loggable {  
    @Override  
    public void log(String message) {  
        //writing in database  
    }  
}
```

```
class Logger {  
    private final Loggable loggable;  
  
    public Logger(Loggable loggable) {  
        this.loggable = loggable;  
    }  
  
    public void log(String message) {  
        this.loggable.log(message);  
    }  
}
```



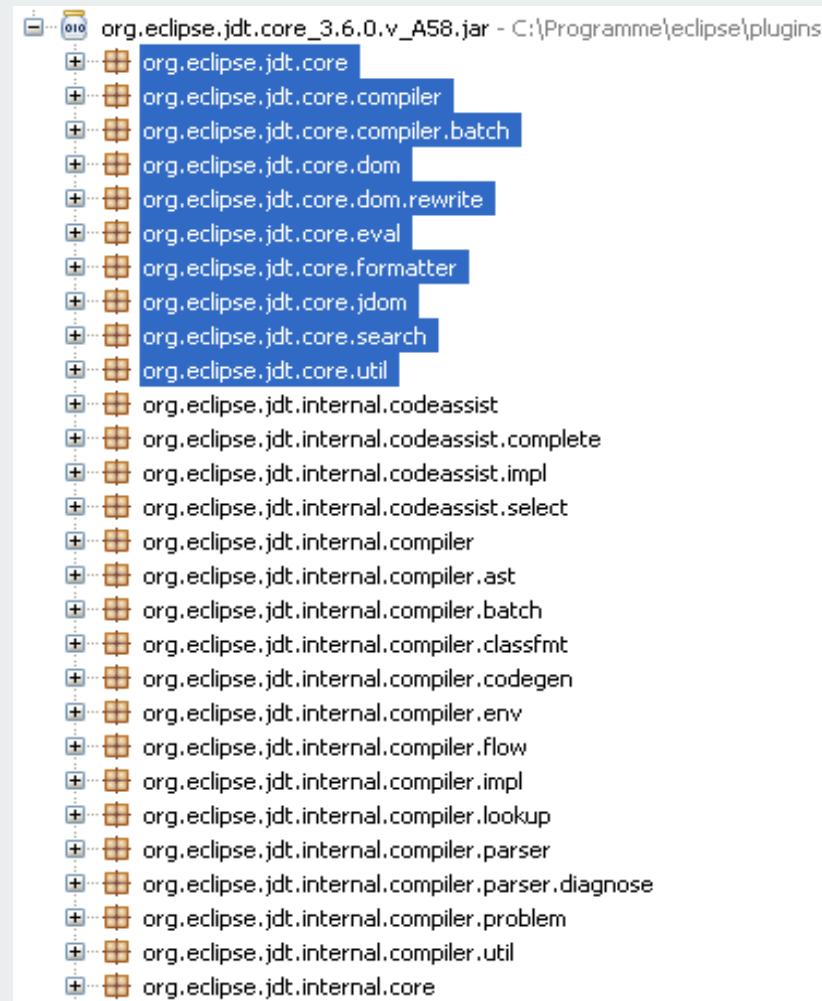
```
class Tester {  
    public static void main(String[] args) {  
        Logger fsLogger = new Logger(new FileSystem());  
        Logger dbLogger = new Logger(new Database());  
  
        fsLogger.log("some text");  
        dbLogger.log("other text");  
    }  
}
```

Packaging Designprinzipien (Kohäsion) - REP (1/2)

241

- REP: Release Reuse Equivalency Principle
 - *"The granule of reuse is the granule of release."*
 - Die Elemente der Wiederverwendung sind die Elemente des Release.
 - Klassen, die miteinander wiederverwendet werden, sollten im gleichen Paket sein, damit sie auch gemeinsam freigegeben werden können.
 - Reduzierung der Anzahl der zu verteilenden Module, was wiederum die Aufwände der Softwareverteilung reduziert.
 - Vermeidung von Versionskonflikten bei der Wiederverwendung, die nur durch die gleichzeitige Freigabe mehrerer Pakete entstehen können.
 - Die Benutzer von Klassen, die wiederum von anderen Klassen abhängen, wollen sich auf eine gemeinsame Freigabe neuer Versionen verlassen.

Packaging Designprinzipien (Kohäsion) - REP (2/2)



`org.eclipse.jdt.core.*` beinhaltet
den wiederverwendbaren Teil von JDT core

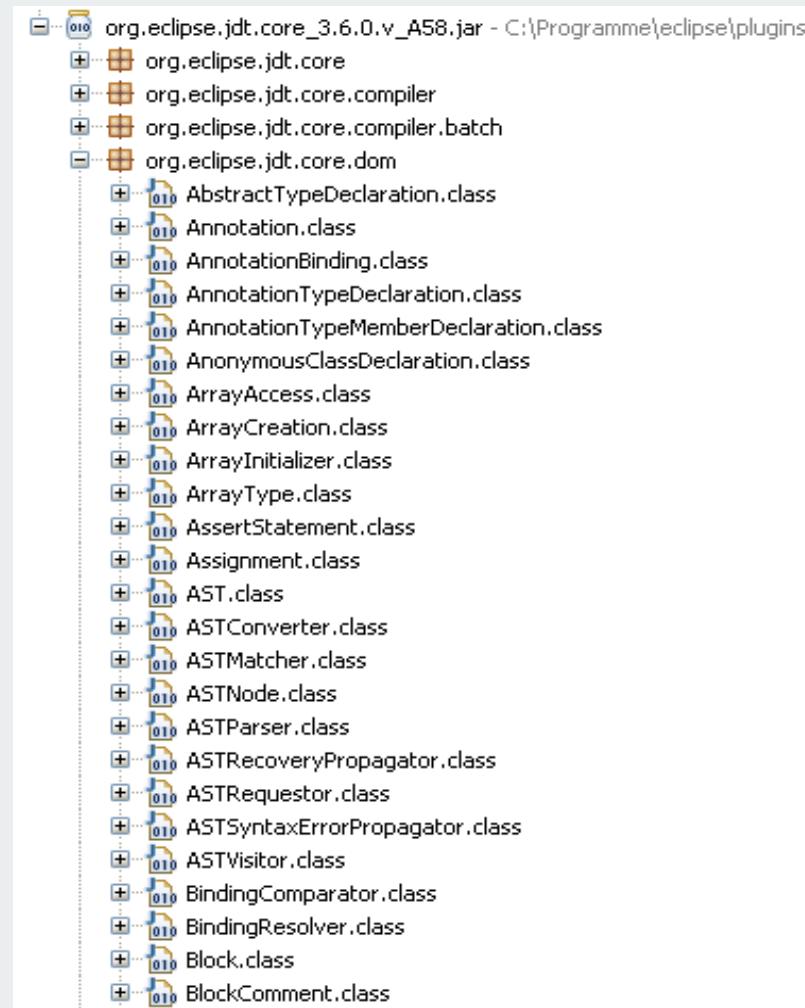
`org.eclipse.jdt.internal.*`
beinhaltet die internen Klassen, sind nicht
für die Wiederverwendung gedacht

Packaging Designprinzipien (Kohäsion) - CCP (1/2)

243

- CCP: Common Closure Principle
 - „*Classes that change together are packaged together.*“
 - Alle Klassen in einem Package sollten gemäß dem Open Closed Prinzip geschlossen gegenüber derselben Art von Veränderungen sein.
 - Entspricht dem Single Responsibility Prinzip angewendet auf Packages.
 - Änderungen an den Anforderungen einer Software, welche Änderungen an einer Klasse eines Moduls benötigen, sollten auch nur die Klassen des Moduls betreffen.
 - Die Einhaltung dieses Prinzips ermöglicht es die Software derart in Packages zu zerlegen, dass (zukünftige) Änderungen in nur wenigen Modulen umgesetzt werden können.
 - Damit ist sichergestellt, dass Änderungen an der Software ohne große Seiteneffekte und somit relativ kostengünstig gemacht werden können.

Packaging Designprinzipien (Kohäsion) - CCP (2/2)



org.eclipse.jdt.core.dom.*
beinhaltet die Repräsentation von JAVA AST
(Abstract Syntax Tree). Falls sich die
Sprachspezifikation ändert, dann befinden
sich alle entsprechenden Änderungen in
diesem Packet.

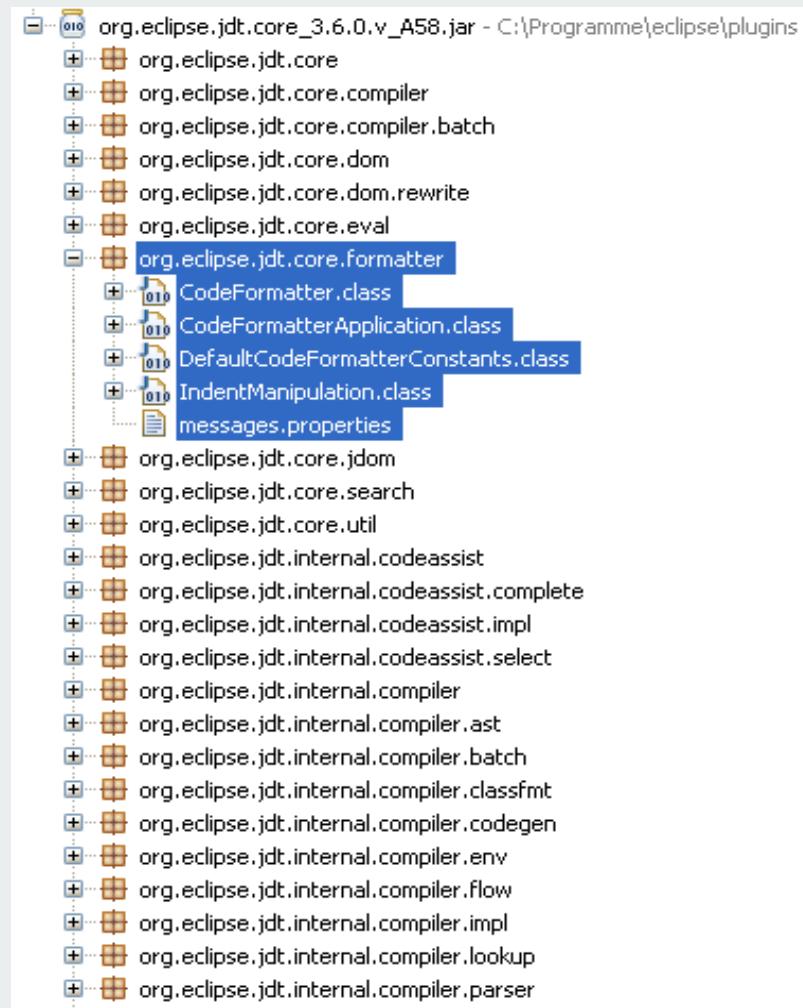
Packaging Designprinzipien (Kohäsion) - CRP (1/2)

245

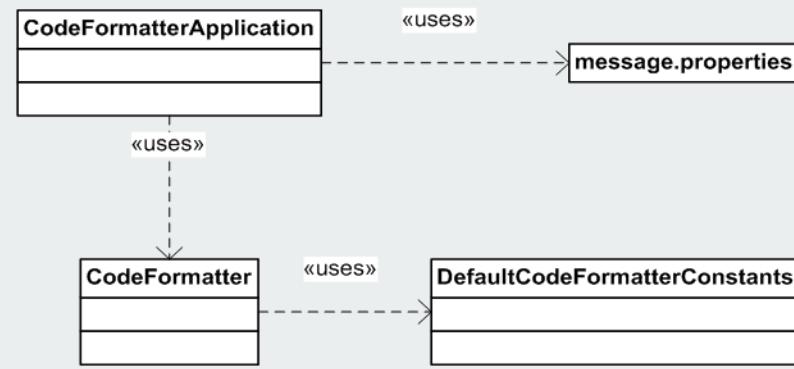
- CRP: Common Reuse Principle
 - „*Classes that are used together are packaged together.*“
 - Zusammenfassung von Klassen, welche gemeinsam verwendet werden.
 - Durch die Einhaltung dieses Prinzips wird eine Unterteilung zusammengehörende Einheiten sichergestellt.

Packaging Designprinzipien (Kohäsion) - CRP (2/2)

246



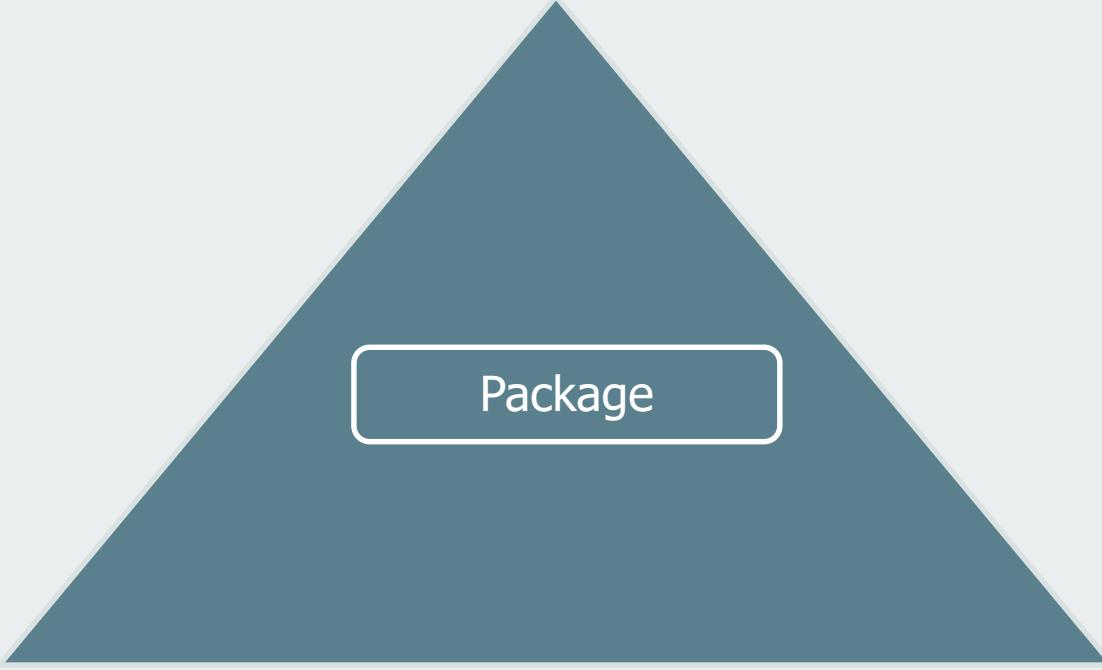
`org.eclipse.jdt.core.formatter.*`
beinhaltet Funktionalität für die Source
Code Formatierung



Packaging Designprinzipien (Kohäsion)

247

**CRP (Common Reuse Principle):
Benutzer**



Package

**REP (Release Reuse Equivalency Principle):
Benutzer**

**CCP (Common Closure Principle):
Maintainer**

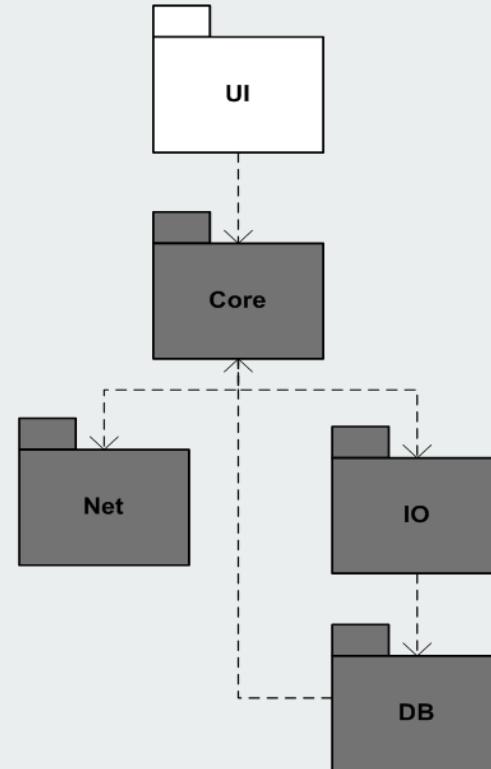
Packaging Designprinzipien (Koppelung) - ADP (1/4)

248

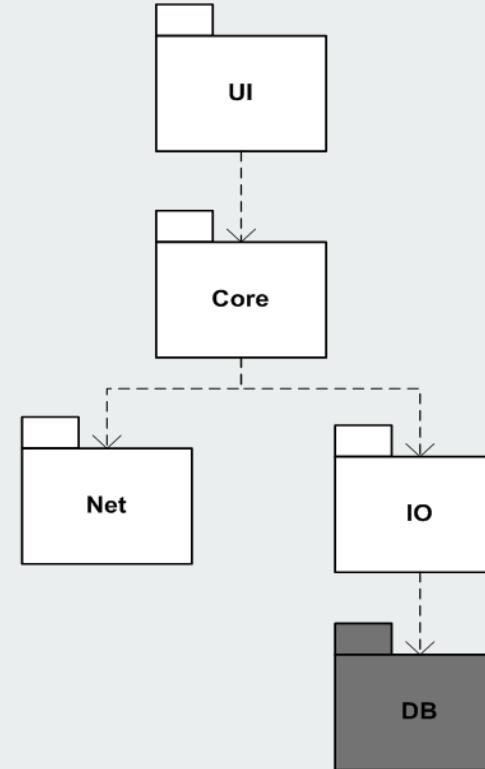
- ADP: Acyclic Dependencies Principle
 - „*The dependency graph of packages must have no cycles.*“
 - Klassen und Pakete sollten **keine zyklischen Abhängigkeiten** enthalten.
 - Erschweren die Wartbarkeit und verringern die Flexibilität, unter anderem, weil Zyklen nur als Ganzes testbar sind.
 - Aufbrechen derartige Zyklen ist immer möglich, dafür gibt es prinzipiell zwei Möglichkeiten:
 - Wenn die Abhängigkeit von Package A auf Package B invertiert werden soll.
→ **Anwendung des DIP:**
 - Einführen eines Interfaces im Package A, welches die von B benötigten Methoden hält.
 - Implementierung dieser Interfaces in den entsprechenden Klassen von Package B.
 - Restrukturierung der Packages:
 - Sammeln aller Klassen eines Zyklus in einem eigenen Package.
 - Einführen ein oder mehrerer neuer Packages, mit den Klassen von denen die Klassen außerhalb des Zyklus abhängen.

Packaging Designprinzipien (Koppelung) - ADP (2/4)

249



Zyklische Abhängigkeit
Unit Tests für DB Package wird
IO, Core und Net Package benötigt

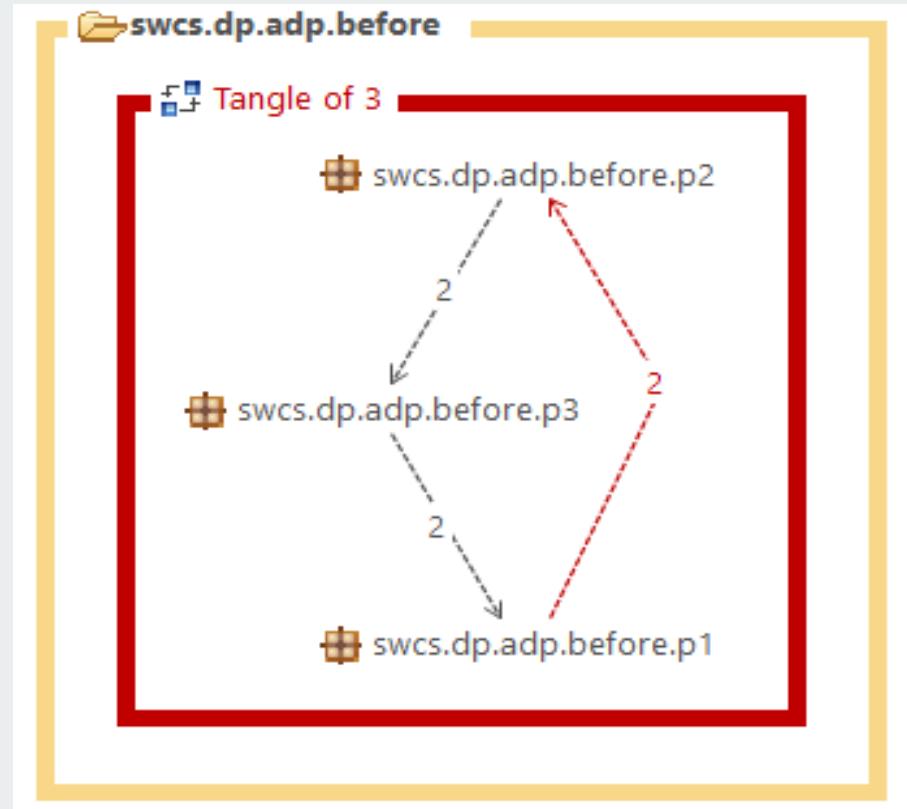
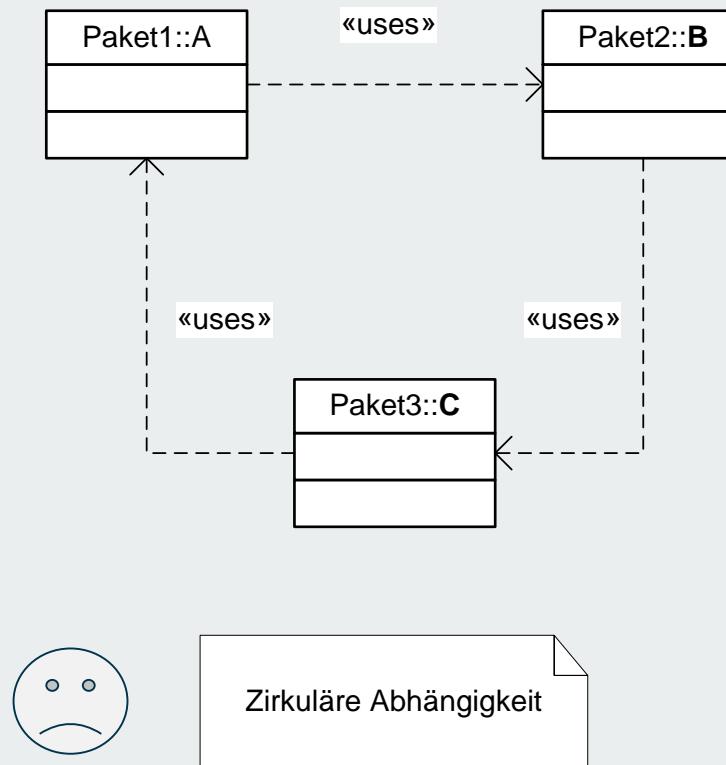


Zykliefrei



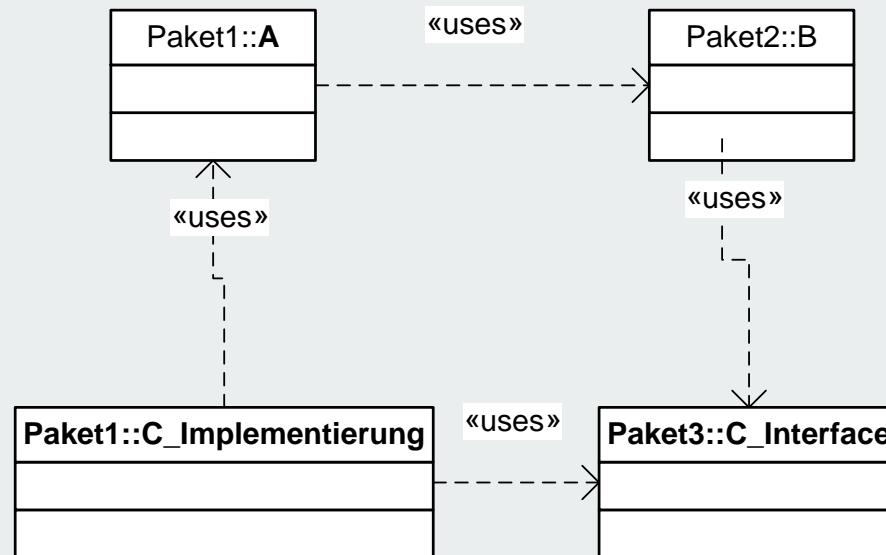
Packaging Designprinzipien (Koppelung) - ADP (3/4)

250

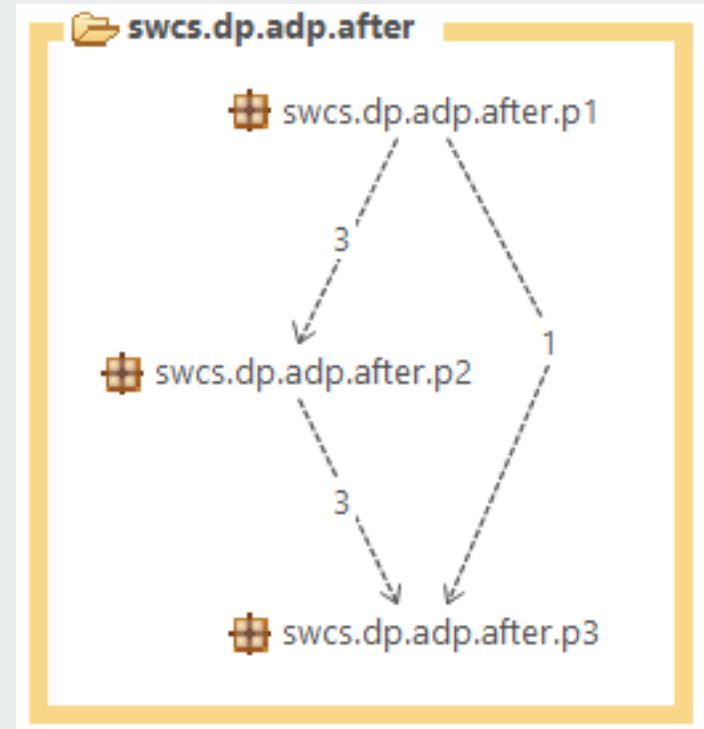


Packaging Designprinzipien (Koppelung) - ADP (4/4)

251



Aufgelöster Zirkel



Packaging Designprinzipien (Koppelung) - SDP (1/3)

252

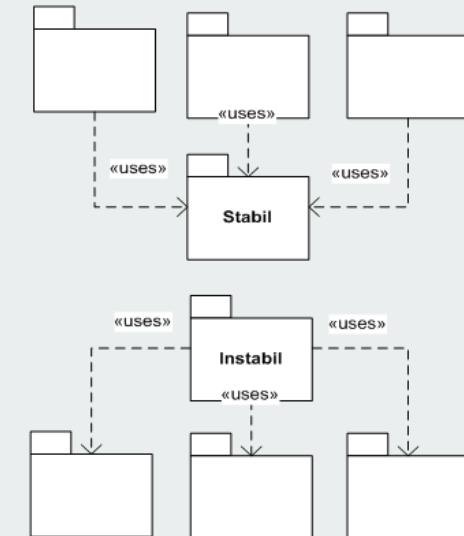
- SDP: Stable Dependencies Principle
 - „*Depend in the direction of stability.*“
 - Die Abhängigkeiten zwischen Packages sollten in der selben Richtung wie die Stabilität verlaufen.
 - Ein Package sollte nur von Packages abhängen, welche stabiler als es selbst sind.
 - Je weniger Abhängigkeiten ein Package zu anderen hat und je mehr Abhängigkeiten andere Packages zu diesem Package haben, umso stabiler ist es.
 - Unter Stabilität versteht man das Verhältnis von ausgehender Koppelung (C_e) zur totalen Koppelung ($C_a + C_e$).
 - $I = 0$, stabiles Package
 - $I = 1$, instabiles Package

$$I = \frac{C_e}{C_a + C_e}$$

$I = \text{Instabilität}, 0 \leq I \leq 1$

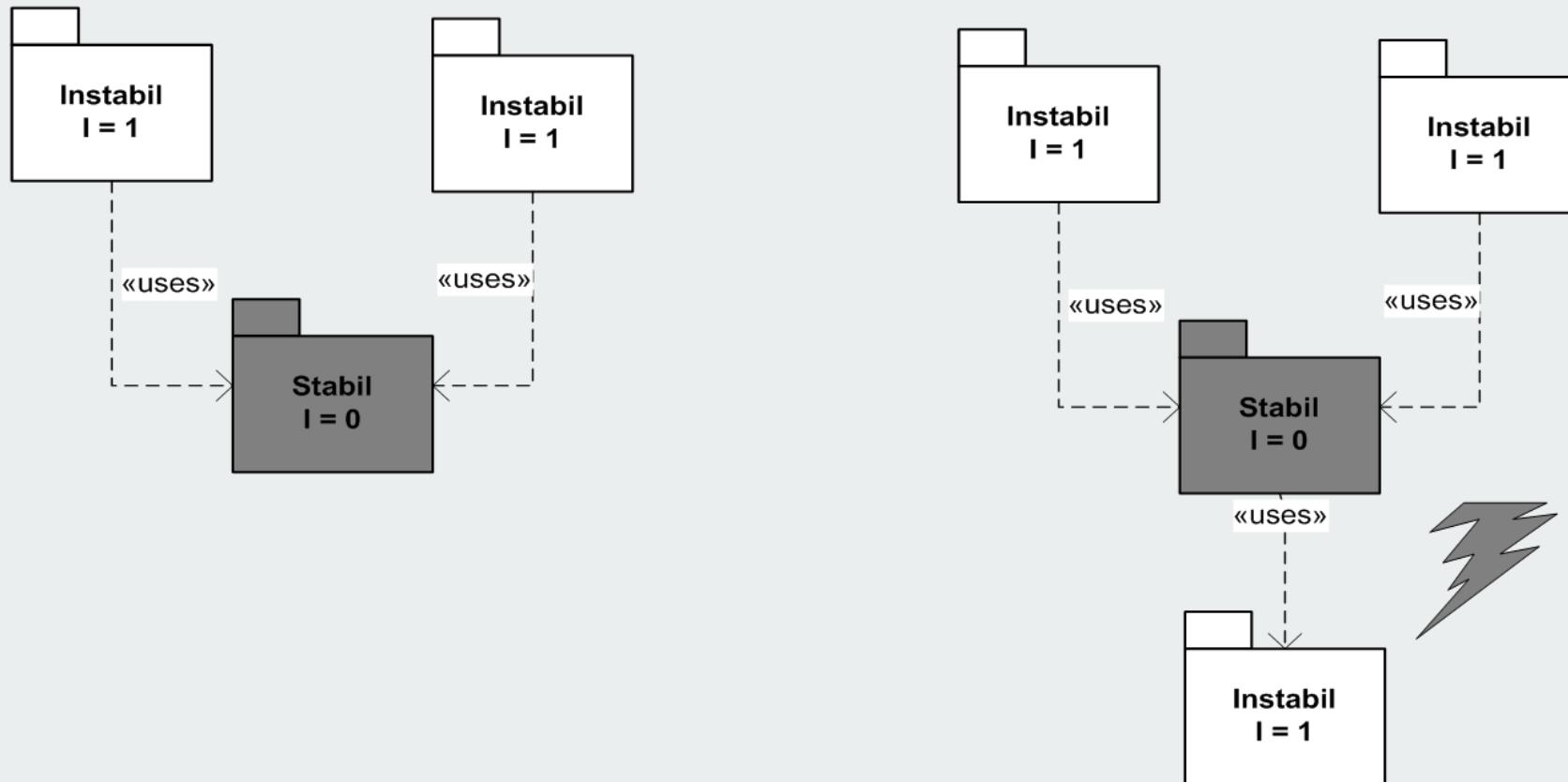
$C_a = \text{eingehende Abhängigkeiten (engl. afferent couplings)}$

$C_e = \text{ausgehende Abhängigkeiten (engl. efferent couplings)}$



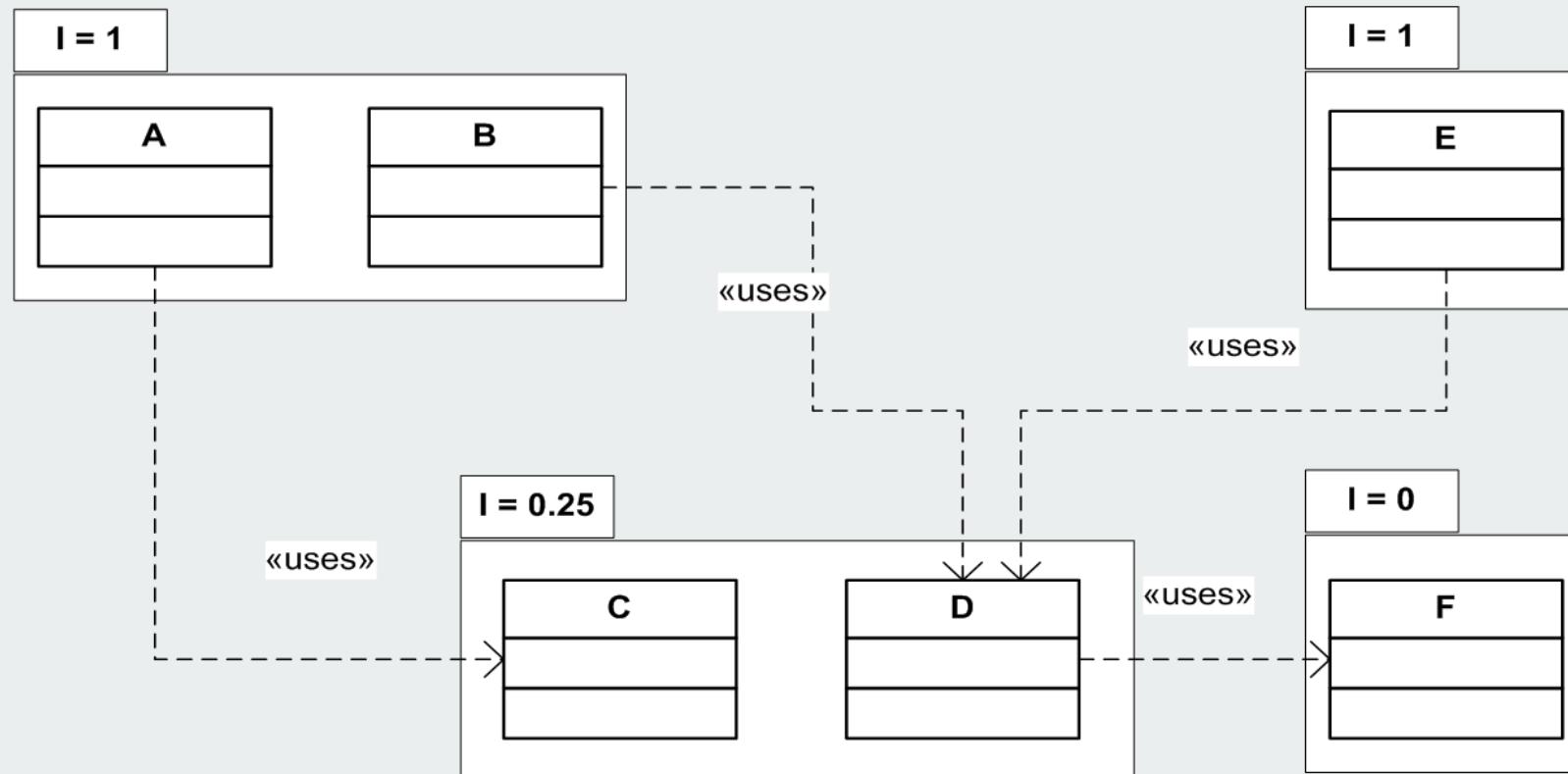
Packaging Designprinzipien (Koppelung) - SDP (2/3)

253



Packaging Designprinzipien (Koppelung) - SDP (3/3)

254



Packaging Designprinzipien (Koppelung) - SAP (1/3)

255

- SAP: Stable Abstractions Principles
 - „*Abstractness increases with stability.*“
 - Packages die maximal stabil sind sollten maximal abstrakt sein.
 - Instabile Packages sollten konkret sein
 - Die Abstraktheit eines Packages sollte proportional zu seiner Stabilität sein.
 - A = 0, komplett konkretes Package
 - A = 1, komplett abstraktes Package

$$A = \frac{N_a}{N_c}$$

A = Abstraktheit , $0 \leq A \leq 1$

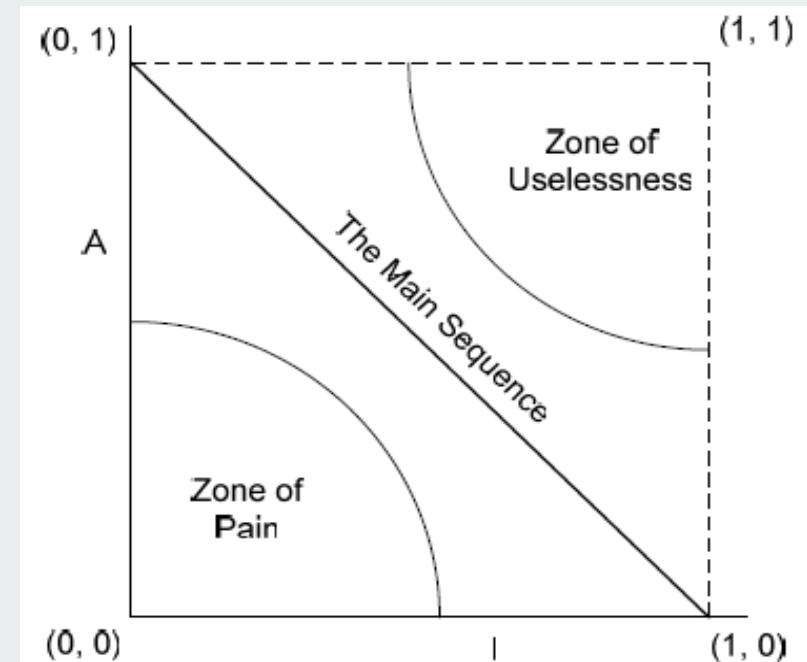
N_a = Anzahl abstrakter Klassen, Interfaces

N_c = Gesamtanzahl der Klassen

Packaging Designprinzipien (Koppelung) - SAP (2/3)

256

- Zone of Pain
 - Stabile nicht abstrakte Klassen
 - Schwer zu verändern, da stabil, schlecht erweiterbar da nicht abstrakt
 - Ausnahmen sind Utility-Klassen wie z. B. String Klasse
- Zone of Uselessness
 - Instabil sehr abstrakte Klassen
 - Sind abstrakt aber werden nirgends erweitert
- The Main Sequence
 - Ideallinie
 - Noch besser wenn Package bei (0,1) oder (1,0) liegt.
 - ➔ In der Praxis kaum umsetzbar
 - **Interfacepackage** bei (0,1)
 - **Implementierungspackage** bei (1,0)



Packaging Designprinzipien (Koppelung) - SAP (3/3)

257

- Für jedes Package lässt sich die Distanz zur idealen Linie zwischen maximaler Stabilität und Abstraktheit und maximaler Instabilität und Konkretheit errechnen.
- Je größer die Distanz ist, desto schlechter ist das SAP erfüllt.

$$D = \left| \frac{A + I - 1}{\sqrt{2}} \right|$$

oder normalisiert

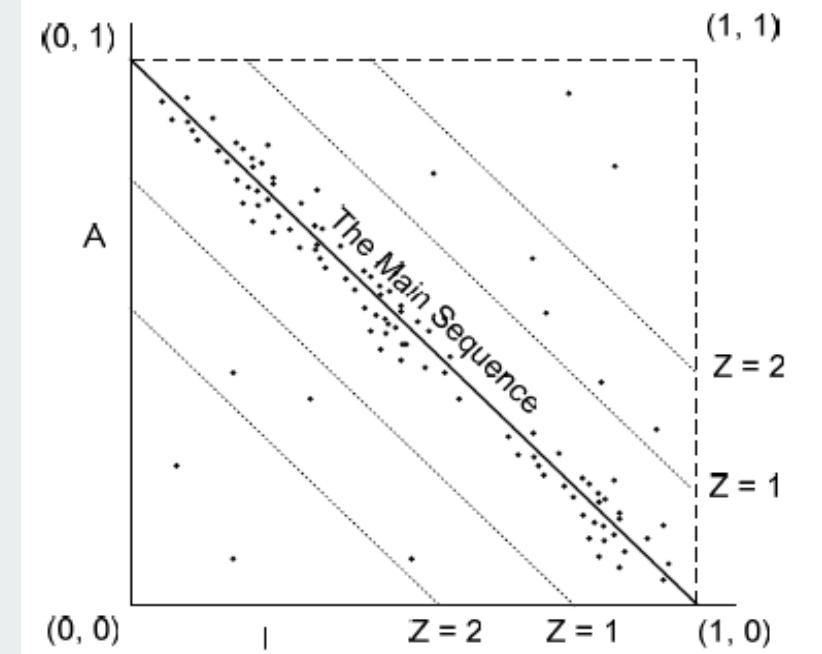
$$D' = |A + I - 1|$$

D = Distanz zur idealen Linie, $0 \leq D \leq 0,707$

D' = normalisierte Distanz zur idealen Linie, $0 \leq D' \leq 1$

A = Abstraktheit eines Packages

I = Instabilität eines Packages



Weitere Designprinzipien (1/3)

258

- SCP: Speaking Code Principle
 - Code soll seinen Zweck kommunizieren, auch ohne Kommentar und Dokumentation.
 - Kommentare sind kein Ersatz für schlechten Code.
 - Klarer und ausdrucksstarker Code mit wenigen Kommentaren ist viel besser als unordentlicher und komplexer Code mit zahlreichen Kommentaren.
 - Anstatt Zeit mit dem Schreiben von Kommentaren zu verbringen, die das Chaos erklären, lieber Zeit darauf verwenden, das Chaos zu beseitigen.
 - Kommentar muss bei Codeänderung angepasst werden.

```
if ((employee.flags & HOURLY_FLAG) && employee.age > 65) { ... }
```



```
if (employee.isEligibleForFullBenefits()) { ... }
```



Weitere Designprinzipien (2/3)

- KISS: Keep It Simple (and) Stupid!
 - Besagt, dass stets die **einfachste Lösung** eines Problems gewählt werden sollte.
 - Vermeiden von zu komplexem und dadurch zu kompliziertem Code.
 - Das Suchen von einfachen Lösungen ist eine Regel, die dabei hilft, diverse Fehler zu vermeiden.
 - Eine gesunde Ablehnung von nicht-simplen Lösungen ist ein Zeichen für das Suchen nach einem guten Code.
 - Je schwieriger Code zu erklären ist, desto wahrscheinlicher ist es, dass er komplizierter als nötig ist und nicht die eleganteste Lösung darstellt.
 - Mache die Dinge so einfach wie möglich, aber nicht einfacher!
- DRY: Don't Repeat Yourself / Once and Only Once
 - Bezieht sich auf die Vermeidung von **dupliziertem Code**, also Code Fragmenten, die an mehreren Stellen gleich oder sehr ähnlich umgesetzt sind.
 - Redundant vorhandener Quellcode ist schwierig zu pflegen, da die Konsistenz zwischen den einzelnen Duplikaten gewährleistet sein muss. Bei Systemen, die dem DRY-Prinzip treu bleiben, müssen hingegen Änderungen nur an einer Stelle vorgenommen werden.

Weitere Designprinzipien (3/3)

260

- YAGNI: You Ain't Gonna Need It!
 - Ein Programm soll erst dann Funktionalität implementieren, wenn klar ist, dass diese Funktionalität tatsächlich gebraucht wird.
 - Entgegen diesem Vorgehen wird in der Praxis oft versucht Programme durch zusätzlichen oder allgemeineren Code auf mögliche künftige Änderungsanforderungen und Features vorzubereiten.
"...das werden DIE sowieso bald fordern..."
 - Solcher Code ist u. U. sehr störend beim Lesen, Verstehen, Ändern von bestehenden Funktionalitäten und kostet Zeit und Geld!
 - Wo keine Anforderung, auch keine Umsetzung!
- SOC: Separation Of Concerns
 - Eine Komponente soll genau eine Aufgabe haben.
 - Mischen von mehreren Verantwortlichkeiten in z. B. einer Klasse vermeiden.
 - Komponenten werden einfacher, verständlicher und wartbarer.
 - Bessere Wiederverwendbarkeit
 - Der Aufgabenkomplex einer Einheit soll in sich geschlossen sein (**hohe Kohäsion**).
 - Die Einheit soll so wenig wie möglich von anderen Einheiten abhängen (**niedrige Kopplung**).

Quellen

- Roock, Stefan: *Refactorings in grossen Softwareprojekten*, Dpunkt Verlag, 2004
- Robert C. Martin: *Agile Software Development: Principles, Patterns and Practices*, Prentice Hall, 2003
- <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>
- <http://javaboutique.internet.com>
- <http://www.slideshare.net/intellizhang/the-oo-design-principles>
- http://www.st.informatik.tu-darmstadt.de/pages/lectures/se/ws06-07/design/lecture/10_principlesOfPackageDesign.pdf
- http://de.wikipedia.org/wiki/Prinzipien_Objektorientierten_Designs

Design Patterns

Design Patterns - Geschichte (1/2)

263

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way Twice."

Christopher Alexander 1977, A Pattern Language: Towns, Buildings, Construction

- **Muster in der Architektur: Christopher Alexander**
- Eingeführt in der Architektur
- In seinem Buch „A Pattern Language“ (1977) beschreibt Alexander ein System von Mustern zur Architektur auf den Ebenen:
 - Landschaftsplanung
 - Stadtplanung
 - einzelne Gebäude
 - einzelne Räume

Design Patterns - Geschichte (2/2)

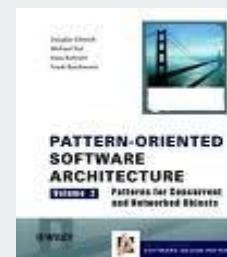
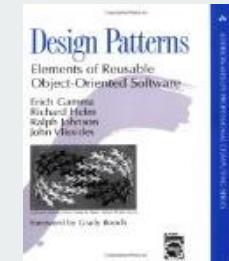
264

- **Muster in der Softwareentwicklung**
- Kent und Cunningham bezogen von Alexander ihre Inspiration für Entwurfsmuster in der Software-Entwicklung (etwa 1987).
- Die Viererbande (**Gang of Four**) übertrug Idee auf die Softwaretechnik und entwarf ein Musterkatalog.
- Gamma, Vlissides, Johnson und Helm veröffentlichen 1995 das Buch „Design Patterns“ und machen damit Entwurfsmuster populär.
 - Lösungsmuster für immer wieder auftretende Entwurfsprobleme im Softwareentwurf
 - Beruhen auf Erfahrung, sie sind häufig nur mit entsprechendem Hintergrundwissen und eigener Erfahrung verständlich und nachvollziehbar.
 - Bilden ein Vokabular (Pattern Language), das Entwickler benutzen um über Design zu diskutieren.
 - Vorteile:
 - Beschleunigung des Entwicklungsprozesses
 - Erhöhung der Lesbarkeit im Code, Wiedererkennungswert
 - Erleichterung der Kommunikation

Gang of Four and POSA

265

- Der einflussreichste Musterkatalog ist das Buch der sogenannten „Gang of Four“ (GoF):
 - E. Gamma, R. Helm, R. Johnson, und J. Vlissides:
Design Patterns - Elements of Reusable Object-Oriented Software,
(Addison-Wesley, 1995)
 - Grundvokabular für objektorientiertes Design
- Ebenfalls einflussreich ist POSA:
 - F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: Pattern-Oriented Software Architecture – A System of Patterns, (Addison-Wesley, 1996)
 - Schwerpunkt bei POSA sind Architekturmuster



Design Patterns der Gang of Four

266

- Die 23 GoF Pattern sind in 3 Gruppen eingeteilt:

Erzeugungsmuster (Creational)

- Singleton
- Builder
- Factory Method
- Abstract Factory
- Prototype

Strukturmuster (Structural)

- Facade
- Decorator
- Adapter
- Composite
- Bridge
- Flyweight
- Proxy

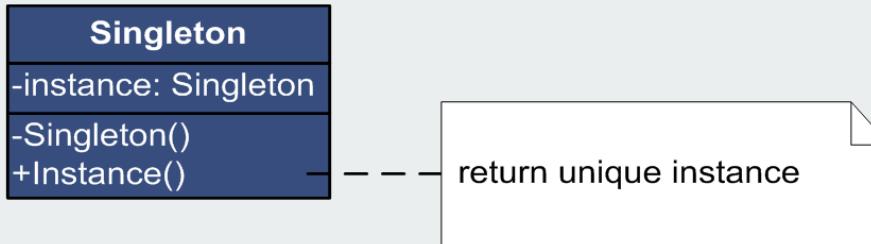
Verhaltensmuster (Behavioral)

- State
- Template Method
- Strategy
- Observer
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Visitor

Erzeugungsmuster - Singleton (1/4)

267

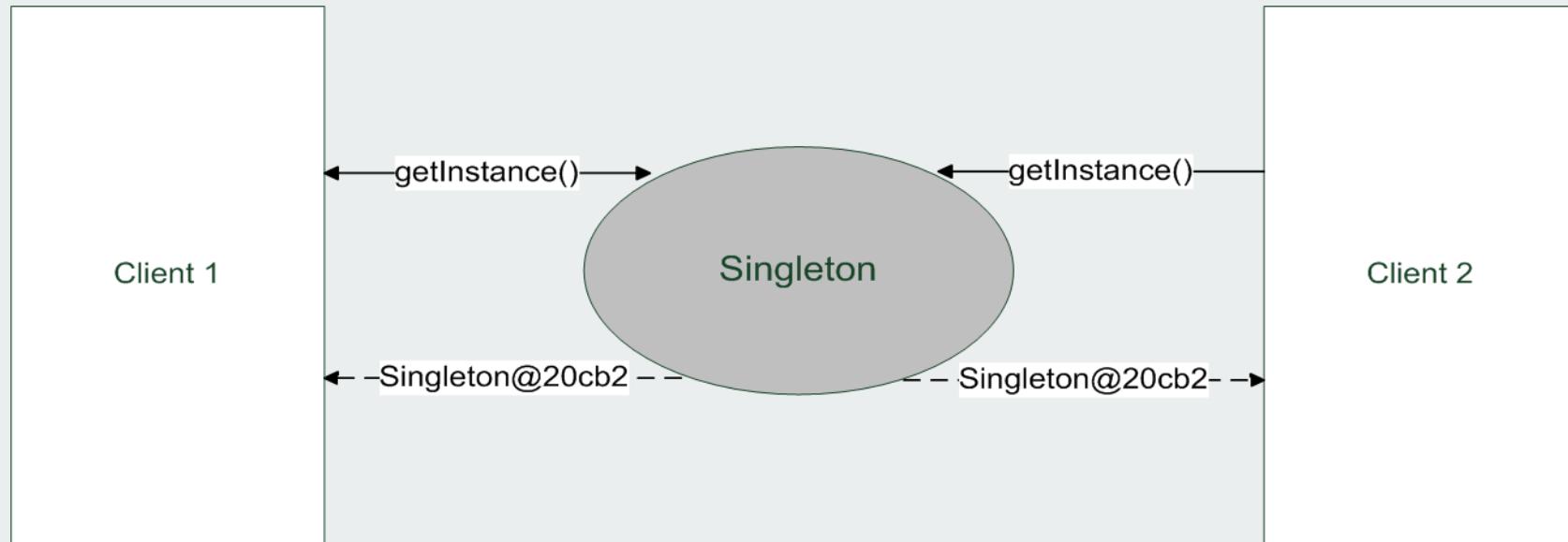
- „Ensure a class only has one instance, and provide a global point of access to it.“



Erzeugungsmuster - Singleton (2/4)

268

- Das Singleton Entwurfsmuster sorgt dafür, dass es von einer Klasse nur eine einzige Instanz gibt.



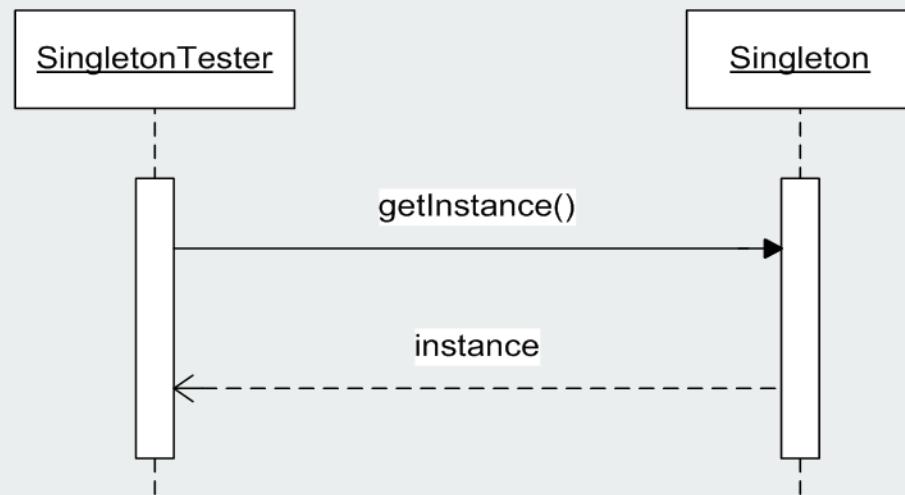
Erzeugungsmuster - Singleton (3/4)

269

```
public class Singleton {  
    private static Singleton instance;  
    private Singleton() {}  
  
    //Lazy Loading  
    public static synchronized Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
  
        return instance;  
    }  
}
```

```
public class Singleton {  
    private static Singleton instance = new Singleton();  
    private Singleton() {}  
  
    //Eager Loading  
    public static Singleton getInstance() {  
        return instance;  
    }  
}
```

```
public class SingletonTester {  
  
    public static void main(String[] args) {  
        Singleton singleton = Singleton.getInstance();  
    }  
}
```



Erzeugungsmuster - Singleton (4/4)

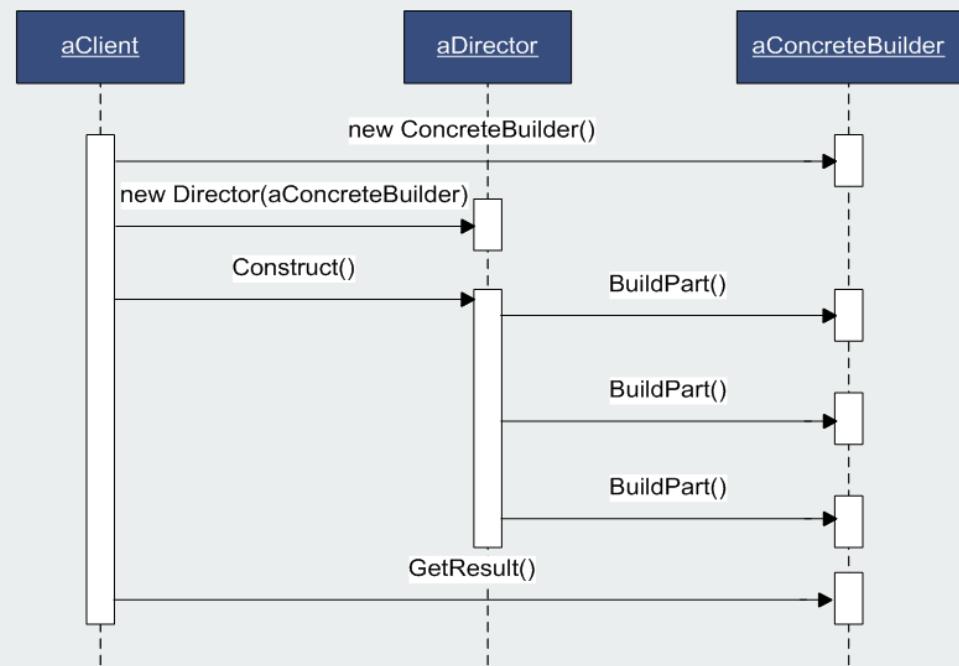
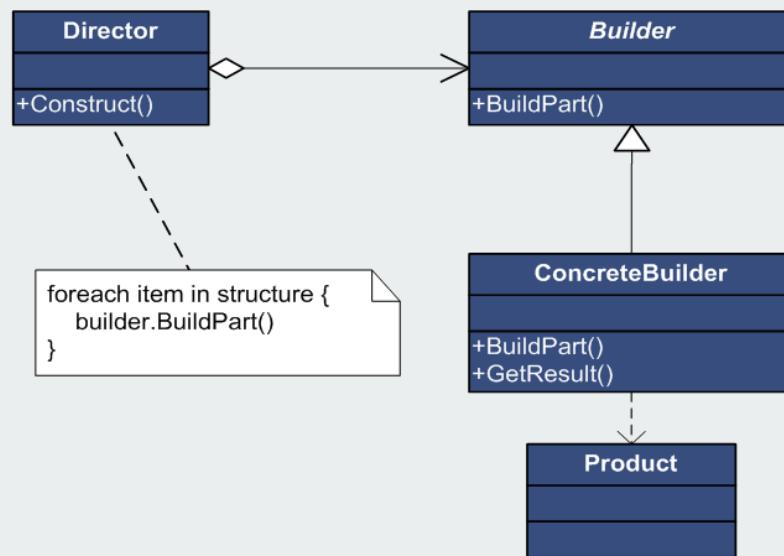
270

- Vorteile
 - **Einfache Anwendung.** Eine Singletonklasse ist schnell und unkompliziert geschrieben.
 - Garantie, dass nur eine Instanz gleichzeitig in einer JVM existiert.
 - Gegenüber *globalen Variablen* ergeben sich eine Reihe von Vorteilen:
 - **Zugriffskontrolle.** Das Singleton kapselt seine eigene Erstellung und kann damit genau kontrollieren, wann und wie Zugriff auf das Singleton erlaubt wird.
 - **Lazy-Loading.** Singletons können erst erzeugt werden, wenn sie auch wirklich gebraucht werden.
- Nachteile
 - **Problematisches Zerstören.** Um in Sprachen mit Garbage Collection Objekte zu zerstören, darf ein Objekt nicht mehr referenziert werden. Dies ist bei Singletons schwierig sicherzustellen. Durch die globale Verfügbarkeit, passiert es sehr schnell, dass Codeteile noch eine Referenz auf das Singleton halten. Besonders bei Mehrbenutzeranwendungen kann ein Singleton die **Performance** senken, da er - besonders in der synchronisierten Form - ein Flaschenhals darstellt.
 - **Einmaligkeit über physikalische Grenzen.** Die Einzigartigkeit eines Singletons über physikalische Grenzen hinweg (JVM) zu gewährleisten, ist schwierig.

Erzeugungsmuster - Builder (1/5)

271

- „Separate the construction of a complex object from its representation so that the same construction process can create different representations.“*



Erzeugungsmuster - Builder (2/5)

```
public abstract class MealBuilder {
    protected Meal meal = new Meal();

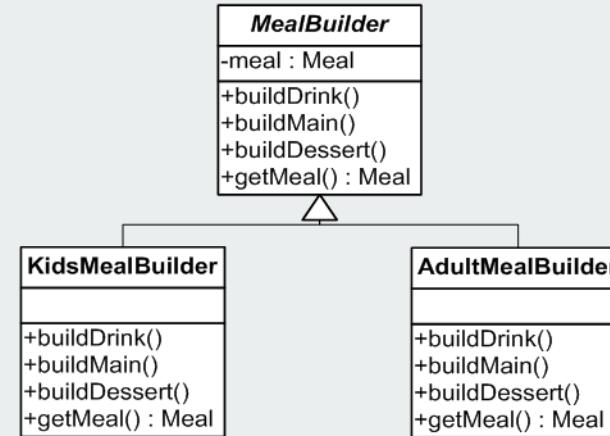
    public abstract void buildDrink();
    public abstract void buildMain();
    public abstract void buildDessert();
    public abstract Meal getMeal();
}
```

```
public class KidsMealBuilder extends MealBuilder {
    public void buildDrink() {
        // add drinks to the meal
    }

    public void buildMain() {
        // add main part of the meal
    }

    public void buildDessert() {
        // add dessert part to the meal
    }

    public Meal getMeal() {
        return this.meal;
    }
}
```



```
public class AdultMealBuilder extends MealBuilder {
    public void buildDrink() {
        // add drinks to the meal
    }

    public void buildMain() {
        // add main part of the meal
    }

    public void buildDessert() {
        // add dessert part to the meal
    }

    public Meal getMeal() {
        return this.meal;
    }
}
```

Erzeugungsmuster - Builder (3/5)

273

```
public class MealDirector {  
    public Meal createMeal(MealBuilder builder) {  
  
        builder.buildDrink();  
        builder.buildMain();  
        builder.buildDessert();  
  
        return builder.getMeal();  
    }  
}
```

```
public class BuilderTester {  
  
    public static void main(String[] args) {  
  
        MealDirector director = new MealDirector();  
        MealBuilder builder = null;  
        boolean isKid = true;  
  
        if (isKid) {  
            builder = new KidsMealBuilder();  
        } else {  
            builder = new AdultMealBuilder();  
        }  
  
        Meal meal = director.createMeal(builder);  
    }  
}
```

Erzeugungsmuster - Builder (4/5)

274

```
public class ImmutablePerson {  
    private final String name;  
    private final List<String> favoriteDishes;  
  
    private ImmutablePerson(Builder builder) {  
        this.name = builder.name;  
        this.favoriteDishes = builder.favoriteDishes;  
    }  
  
    public String getName() {  
        return this.name;  
    }  
  
    public List<String> getFavoriteDishes() {  
        return this.favoriteDishes != null ?  
            new ArrayList<>(this.favoriteDishes) : null;  
    }  
  
    public Builder toBuilder() {  
        return new Builder(this);  
    }  
  
    public static Builder builder() {  
        return new Builder();  
    }  
}
```

```
public static final class Builder {  
    private String name;  
    private List<String> favoriteDishes;  
  
    public Builder() {}  
  
    public Builder(ImmutablePerson person) {  
        this.name = person.name;  
        this.favoriteDishes = person.favoriteDishes != null ?  
            new ArrayList<>(person.favoriteDishes) : null;  
    }  
  
    public Builder withName(String name) {  
        this.name = name;  
        return this;  
    }  
  
    public Builder withFavoriteDishes(List<String> dishes) {  
        this.favoriteDishes = dishes != null ?  
            new ArrayList<>(dishes) : null;  
        return this;  
    }  
  
    public ImmutablePerson build() {  
        return new ImmutablePerson(this);  
    }  
}
```

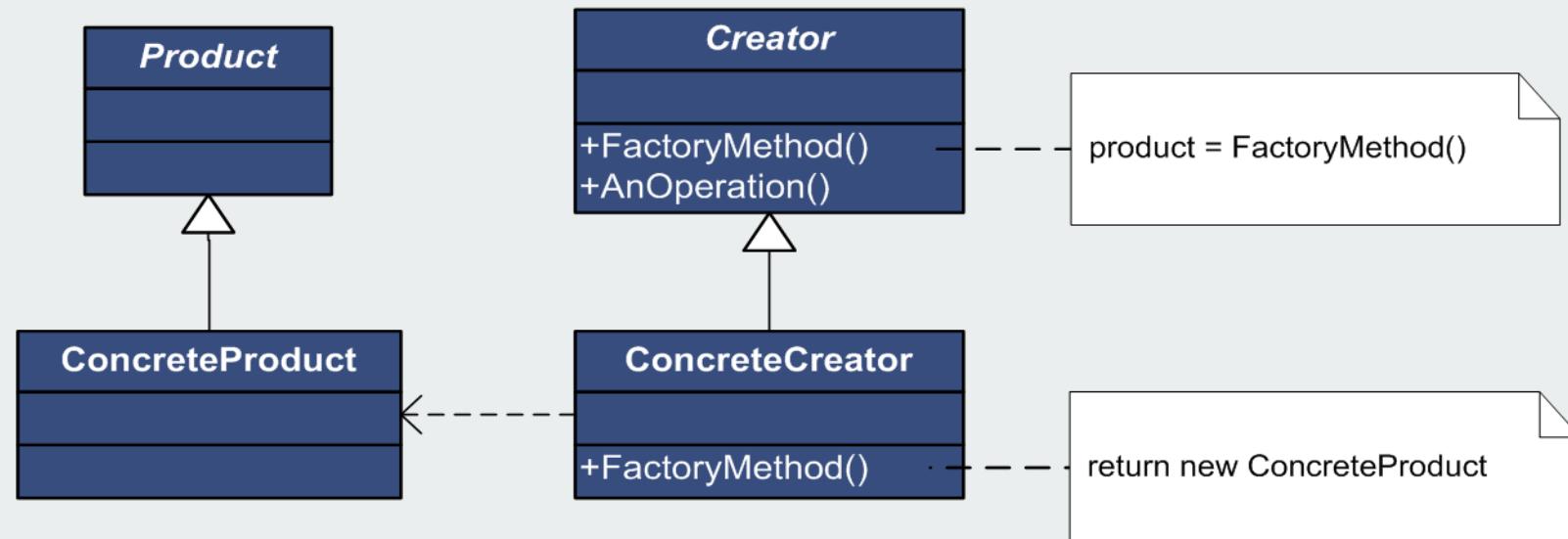
Erzeugungsmuster - Builder (5/5)

```
public class ImmutablePersonTester {  
  
    public static void main(String[] args) {  
        List<String> dishes = new ArrayList<>();  
        dishes.add("Nudeln");  
        dishes.add("Müsli");  
        dishes.add("Pfannkuchen");  
  
        ImmutablePerson immutablePerson1 = ImmutablePerson.builder()  
            .withName("Timo")  
            .withCity("Nürnberg")  
            .withFavoriteDishes(dishes)  
            .build();  
  
        List<String> dishes2 = immutablePerson1.getFavoriteDishes();  
        dishes2.add("Brei");  
  
        ImmutablePerson immutablePerson2 = ImmutablePerson.builder()  
            .withName("Lia")  
            .withCity("Nürnberg")  
            .withFavoriteDishes(dishes2)  
            .build();  
  
        ImmutablePerson updatedPerson = immutablePerson2.toBuilder()  
            .withCity("Schönau")  
            .build();  
  
    }  
}
```

Erzeugungsmuster - Factory Method (1/8)

276

- „Define an interface for creating an object, but let the subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.“



Erzeugungsmuster - Factory Method (2/8)

277

- Anwendungsfälle
 - Trennung der Objektverarbeitung (**Wie?**) von der konkreten Objekterstellung (**Instanziierung; Was?**)
 - Delegation der Objektinstanziierung an Unterklasse
 - Fälle, in denen mit einer wachsenden Anzahl und Ausformung von Produkten zu rechnen ist. Sowie Szenarien in denen alle Produkte einen allgemeinen Herstellungsprozess durchlaufen müssen, egal was für ein Produkt sie konkret sind.
 - Wenn die konkret zu erstellenden Produkte nicht bekannt sind oder nicht von vorne herein festgelegt werden sollen.

Erzeugungsmuster - Factory Method (3/8)

278

```
public interface Logger {  
    public void log(String message);  
}
```

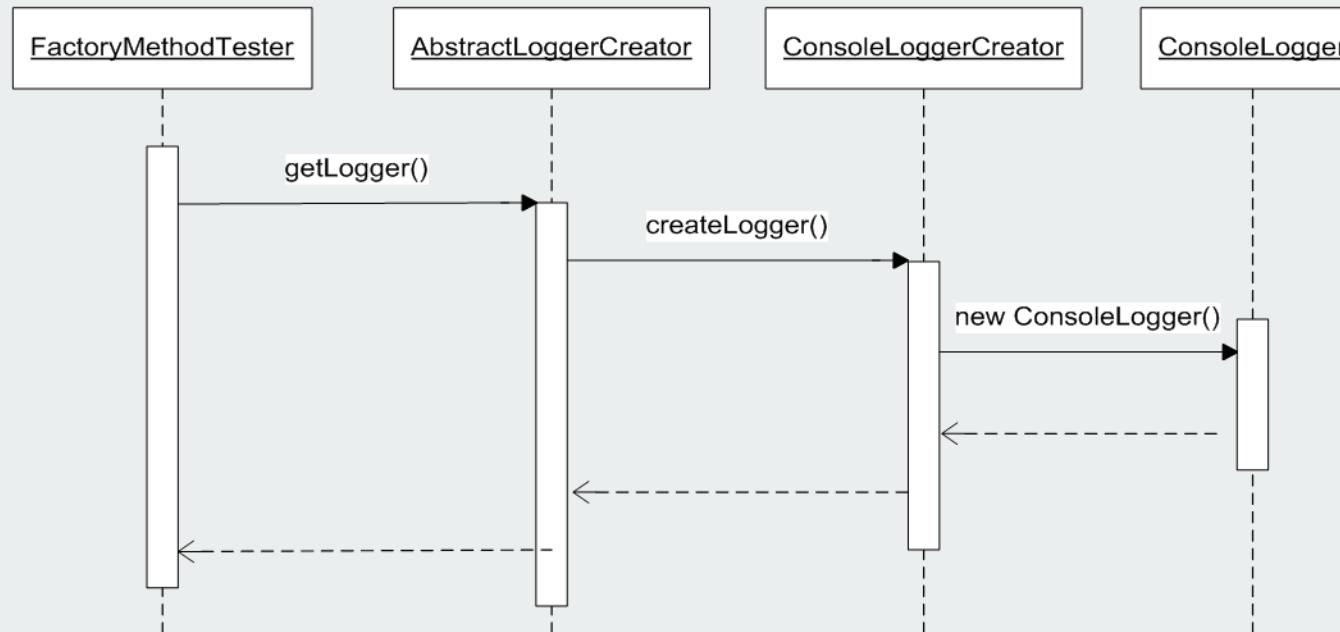
```
public class ConsoleLogger implements Logger {  
  
    public void log(String message) {  
        System.err.println(message);  
    }  
}
```

```
public abstract class AbstractLoggerCreator {  
  
    public Logger getLogger() {  
        // depending on the subclass, we'll get a particular logger.  
        Logger logger = createLogger();  
  
        //could do other operations on the logger here  
        return logger;  
    }  
  
    public abstract Logger createLogger();  
}
```

```
public class ConsoleLoggerCreator extends AbstractLoggerCreator {  
  
    @Override  
    public Logger createLogger() {  
        return new ConsoleLogger();  
    }  
}
```

Erzeugungsmuster - Factory Method (4/8)

279



```
public class FactoryMethodTester {  
  
    public static void main(String[] args) {  
        AbstractLoggerCreator creator = new ConsoleLoggerCreator();  
        Logger logger = creator.getLogger();  
        logger.log("message");  
    }  
}
```

Erzeugungsmuster - Factory Method (5/8)

280

```
public interface Button {  
    void render();  
    void onClick();  
}
```

```
public class HtmlButton implements Button {  
  
    @Override  
    public void render() {  
        System.out.println("<button>Say hello world!</button>");  
    }  
  
    @Override  
    public void onClick() {  
        System.out.println("Hello World!");  
    }  
}
```

```
public class WindowsButton implements Button {  
  
    @Override  
    public void render() {  
        JFrame frame = new JFrame();  
        JPanel panel = new JPanel();  
  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        panel.setLayout(new FlowLayout(FlowLayout.CENTER));  
        frame.getContentPane().add(panel);  
  
        JButton button = new JButton("Say hello world!");  
        button.addActionListener(e -> onClick());  
        panel.add(button);  
        frame.setSize(200, 100);  
        frame.setVisible(true);  
    }  
  
    @Override  
    public void onClick() {  
        System.out.println("Hello World!");  
    }  
}
```

Erzeugungsmuster - Factory Method (6/8)

281

```
public abstract class Dialog {  
  
    public void renderWindow() {  
        // ... other code ...  
  
        Button okButton = createButton();  
        okButton.render();  
    }  
  
    public abstract Button createButton();  
}
```

```
public class HtmlDialog extends Dialog {  
  
    @Override  
    public Button createButton() {  
        return new HtmlButton();  
    }  
}
```

```
public class WindowsDialog extends Dialog {  
  
    @Override  
    public Button createButton() {  
        return new WindowsButton();  
    }  
}
```

Erzeugungsmuster - Factory Method (7/8)

282

```
public class FactoryMethodTester {  
    private static Dialog dialog;  
  
    public static void main(String[] args) {  
        configure();  
        runBusinessLogic();  
    }  
  
    private static void configure() {  
        if (System.getProperty("os.name").equals("Windows 10")) {  
            dialog = new WindowsDialog();  
        } else {  
            dialog = new HtmlDialog();  
        }  
    }  
  
    private static void runBusinessLogic() {  
        dialog.renderWindow();  
    }  
}
```

Erzeugungsmuster - Factory Method (8/8)

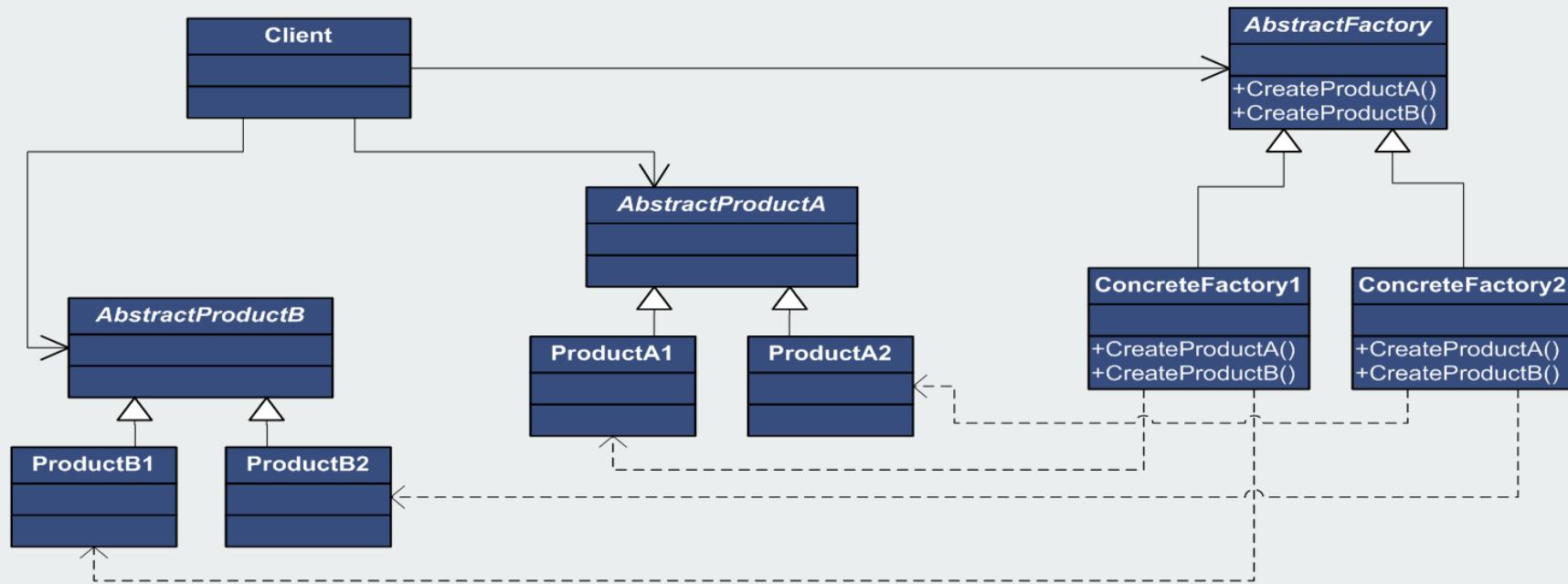
283

- Vorteile
 - Herstellungsprozess ist von einer konkreten Implementierung getrennt .
 - Unterschiedliche Produktimplementierungen können denselben Produktionsvorgang durchlaufen.
 - **Wiederverwendbarkeit und Konsistenz** durch:
 - Kapselung des Objekterstellungscodes in eigener Klasse. Dadurch entsteht eine **einheitliche und zentrale Schnittstelle** für den Client. Die Produktimplementierung ist von seiner Verwendung entkoppelt.
- Nachteile
 - Wenn viele simple Objekte erzeugt werden sollen ist der Aufwand unverhältnismäßig hoch, da immer vom Creator abgeleitet werden muss.

Erzeugungsmuster - Abstract Factory (1/8)

284

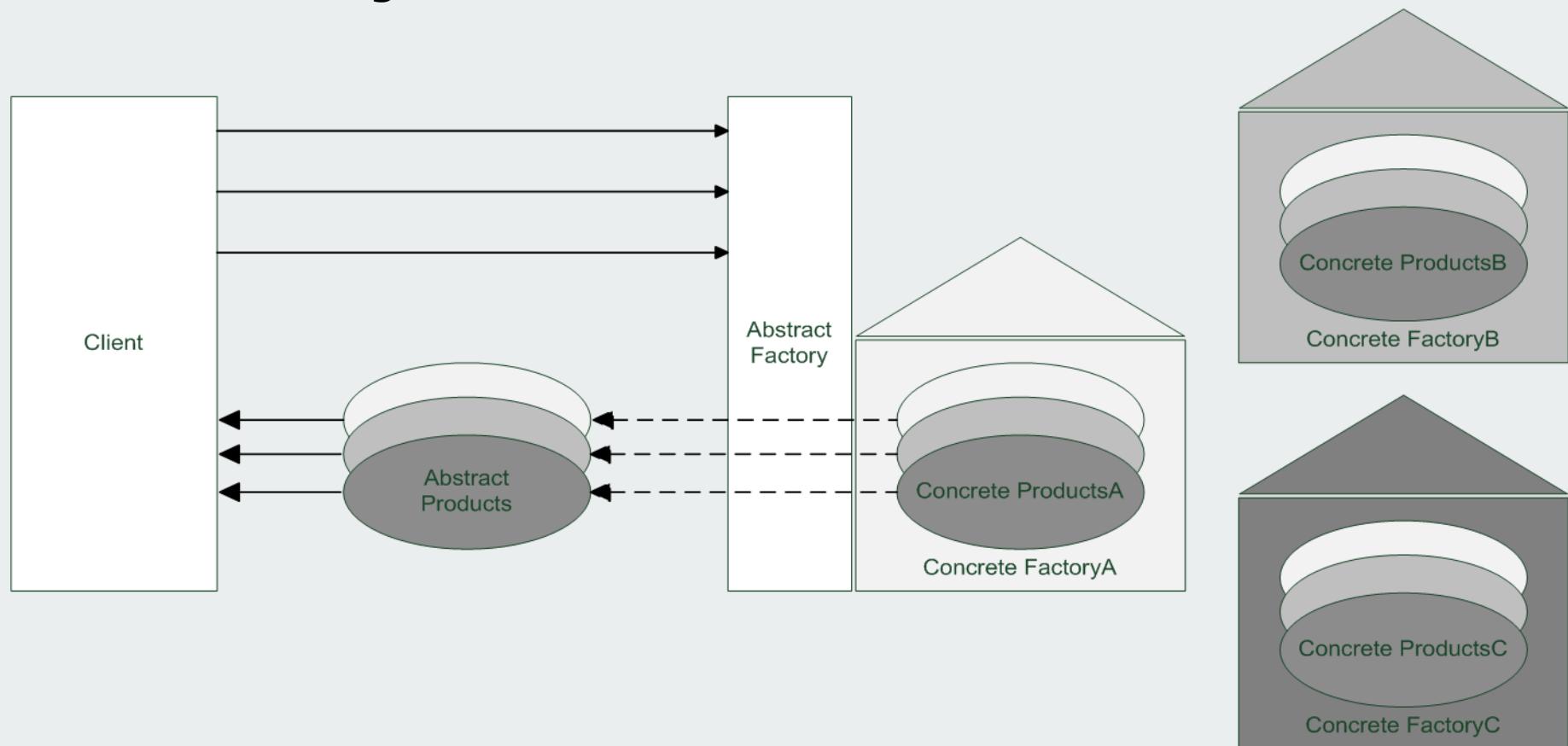
- „Provide an interface for creating families of related or dependent objects without specifying their concrete class.“



Erzeugungsmuster - Abstract Factory (2/8)

285

- Das Abstract Factory Design Pattern dient der Definition einer **zusammenhängenden Familie** aus Produkten.



Erzeugungsmuster - Abstract Factory (3/8)

286

- Anwendungsfälle
 - Wenn verschiedene Objekte zu einem Kontext erstellt werden und daher immer zusammenhängend erstellt werden müssen.
 - Wenn ein System mit unterschiedlichen Sets von Objekten konfiguriert werden muss oder dies ermöglicht sein soll.
 - Wenn ein System losgelöst davon sein soll, wie bestimmte Objekte erstellt werden.
 - Eine Objektfamilie bereitgestellt, aber noch keine Aussagen zu den konkreten Implementierungen gemacht werden kann oder soll. Stattdessen werden Interfaces bereitgestellt.
 - Typische Anwendung von Abstract Factories sind z. B. GUI-Bibliotheken, die unterschiedliche Look & Feels unterstützen.

Erzeugungsmuster - Abstract Factory (4/8)

287

Produktfamilie	Klassen
Luxury	LuxuryCar , LuxurySUV
Nonluxury	NonLuxuryCar , NonLuxurySUV

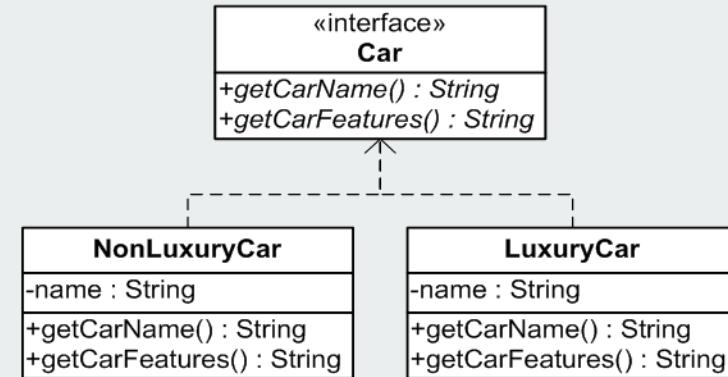
```
public interface Car {
    public String getCarName();
    public String getCarFeatures();
}
```

```
public class LuxuryCar implements Car {
    private final String name;

    public LuxuryCar(String name) {
        this.name = name;
    }

    @Override
    public String getCarName() {
        return this.name;
    }

    @Override
    public String getCarFeatures() {
        return "Luxury Car Features ";
    }
}
```



```
public class NonLuxuryCar implements Car {
    private final String name;

    public NonLuxuryCar(String name) {
        this.name = name;
    }

    @Override
    public String getCarName() {
        return this.name;
    }

    @Override
    public String getCarFeatures() {
        return "Non-Luxury Car Features ";
    }
}
```

Erzeugungsmuster - Abstract Factory (5/8)

288

Produktfamilie	Klassen
Luxury	LuxuryCar, LuxurySUV
Nonluxury	NonLuxuryCar, NonLuxurySUV

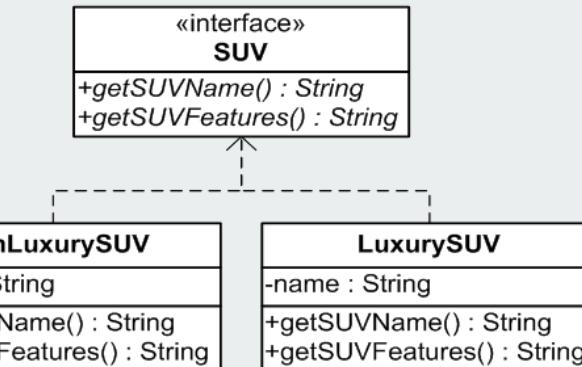
```
public interface SUV {
    public String getSUVName();
    public String getSUVFeatures();
}
```

```
public class LuxurySUV implements SUV {
    private final String name;

    public LuxurySUV(String name) {
        this.name = name;
    }

    @Override
    public String getSUVName() {
        return this.name;
    }

    @Override
    public String getSUVFeatures() {
        return "Luxury SUV Features ";
    }
}
```



```
public class NonLuxurySUV implements SUV {
    private final String name;

    public NonLuxurySUV(String name) {
        this.name = name;
    }

    @Override
    public String getSUVName() {
        return this.name;
    }

    @Override
    public String getSUVFeatures() {
        return "Non-Luxury SUV Features ";
    }
}
```

Erzeugungsmuster - Abstract Factory (6/8)

```
public abstract class VehicleFactory {
    public static final String LUXURY_VEHICLE = "Luxury";
    public static final String NON_LUXURY_VEHICLE = "Non-Luxury";

    public abstract Car getCar();
    public abstract SUV getSUV();

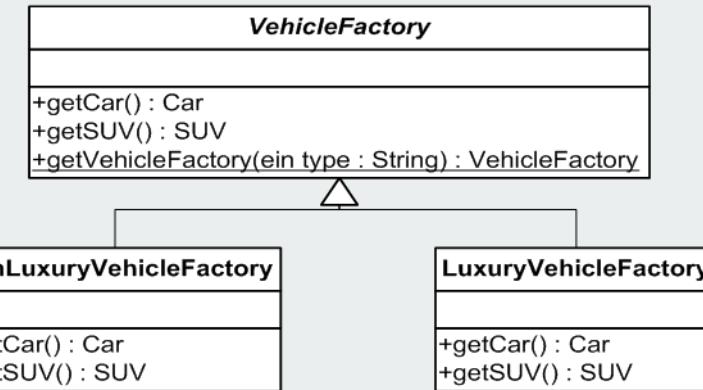
    public static VehicleFactory getVehicleFactory(String type) {
        if (VehicleFactory.LUXURY_VEHICLE.equals(type))
            return new LuxuryVehicleFactory();
        if (VehicleFactory.NON_LUXURY_VEHICLE.equals(type))
            return new NonLuxuryVehicleFactory();

        return new LuxuryVehicleFactory();
    }
}
```

```
public class NonLuxuryVehicleFactory extends VehicleFactory {

    @Override
    public Car getCar() {
        return new NonLuxuryCar("NL-C");
    }

    @Override
    public SUV getSUV() {
        return new NonLuxurySUV("NL-S");
    }
}
```



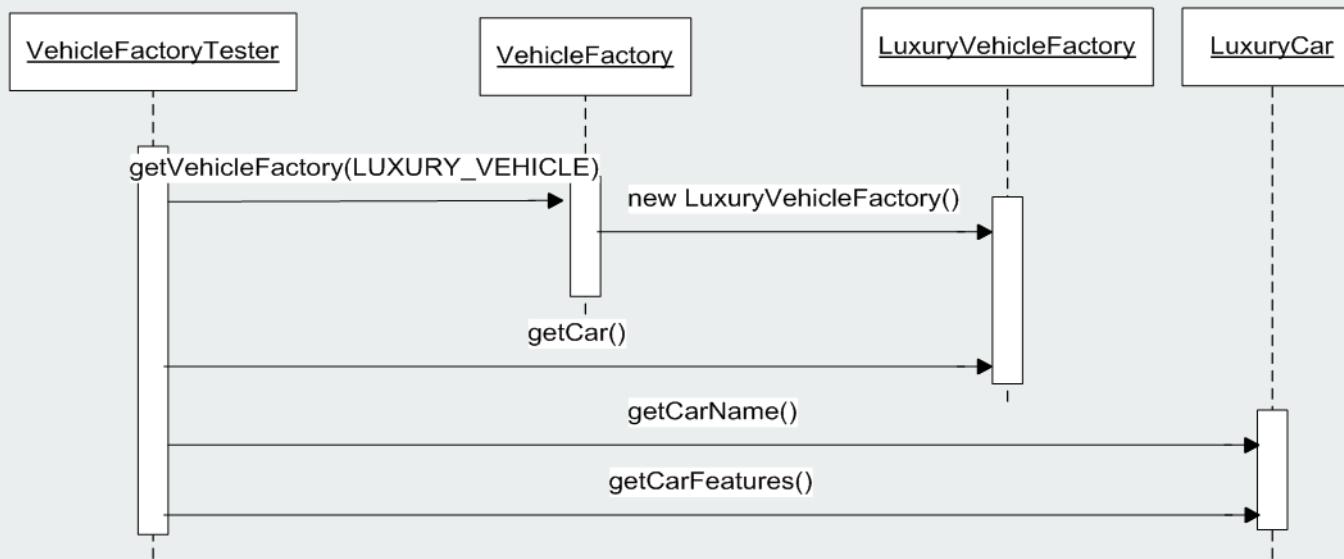
```
public class LuxuryVehicleFactory extends VehicleFactory {

    @Override
    public Car getCar() {
        return new LuxuryCar("L-C");
    }

    @Override
    public SUV getSUV() {
        return new LuxurySUV("L-S");
    }
}
```

Erzeugungsmuster - Abstract Factory (7/8)

290



```
public class VehicleFactoryTester {

    public static void main(String[] args) {
        VehicleFactory vf = VehicleFactory.getVehicleFactory(VehicleFactory.LUXURY_VEHICLE);

        Car car = vf.getCar();
        System.out.println("Name: " + car.getCarName() + " Features: " + car.getCarFeatures());

        SUV suv = vf.getSUV();
        System.out.println("Name: " + suv.getSUVName() + " Features: " + suv.getSUVFeatures());
    }
}
```

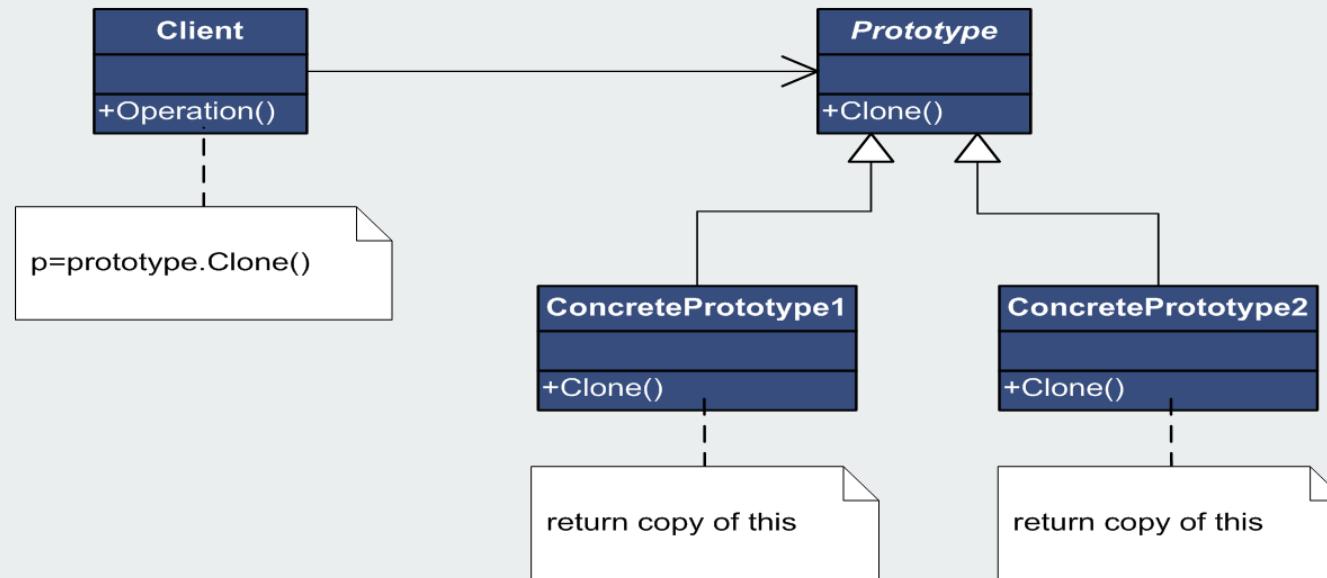
Erzeugungsmuster - Abstract Factory (8/8)

- Vorteile
 - Durch das **Abschirmen der konkreten Klassen** wird der Clientcode allgemeingültig. Es ist kein Code für spezielle Fälle notwendig.
 - **Konsistenz.** Es wird sichergestellt, dass nur jene Objekte zum Client gelangen, die auch zusammenpassen.
 - **Flexibilität.** Ganze Objektfamilien können ausgetauscht werden, da sich der Client nur auf Abstraktionen (Abstract Factory, Produktschnittstellen) stützt.
 - **Einfache Erweiterung mit neuen Produktfamilien.** Neue Produktfamilien können sehr einfach ins System integriert werden. Dazu ist lediglich das erneute Implementieren der Factoryschnittstelle nötig. Anschließend muss nur noch an einer zentralen Stelle im Client die neue Factory instanziert werden.
- Nachteile
 - **Inflexibilität hinsichtlich neuer Familienmitglieder.** Soll der Produktfamilie ein neues Produkt hinzugefügt werden, so ist eine Änderung der Schnittstelle der Abstract Factory notwendig. Dies führt aber zum Brechen von Code aller konkreten Factories. Der Änderungsaufwand ist groß.
 - Neue Implementierung mit großer Ähnlichkeit zu einer Bestehenden erfordert trotzdem eine neue Fabrik.

Erzeugungsmuster - Prototype

292

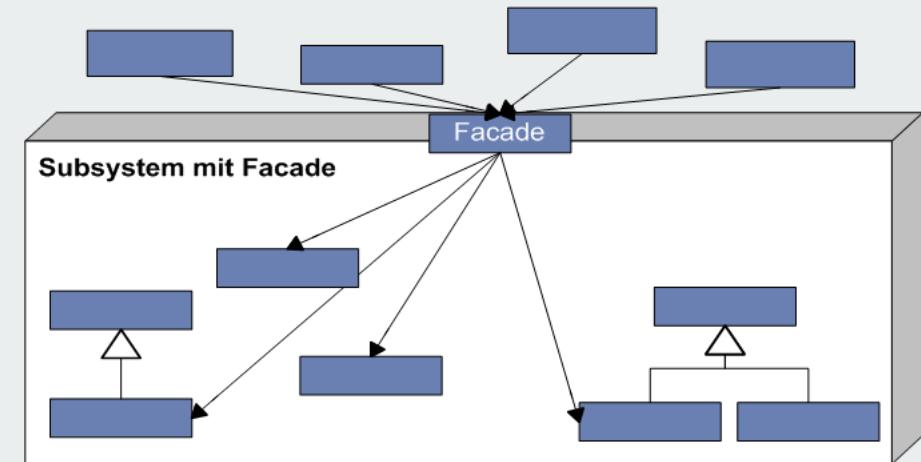
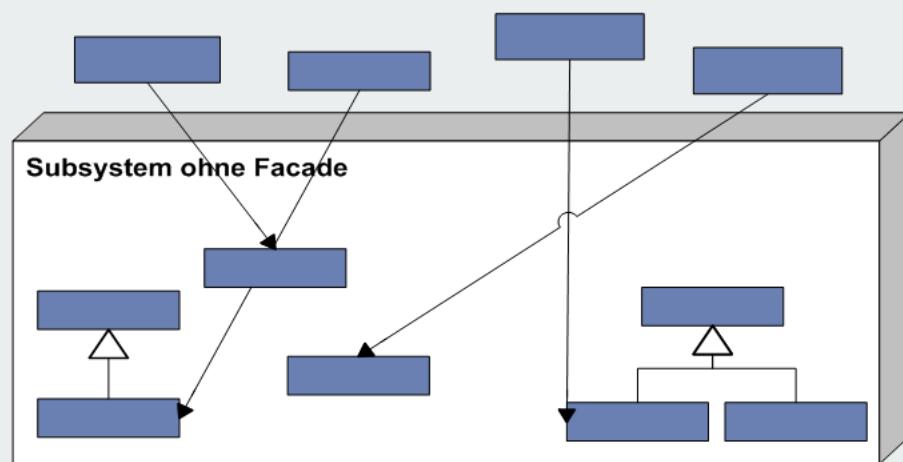
- „Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.“



Strukturmuster - Facade (1/5)

293

- „Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higherlevel interface that makes the subsystem easier to use.“

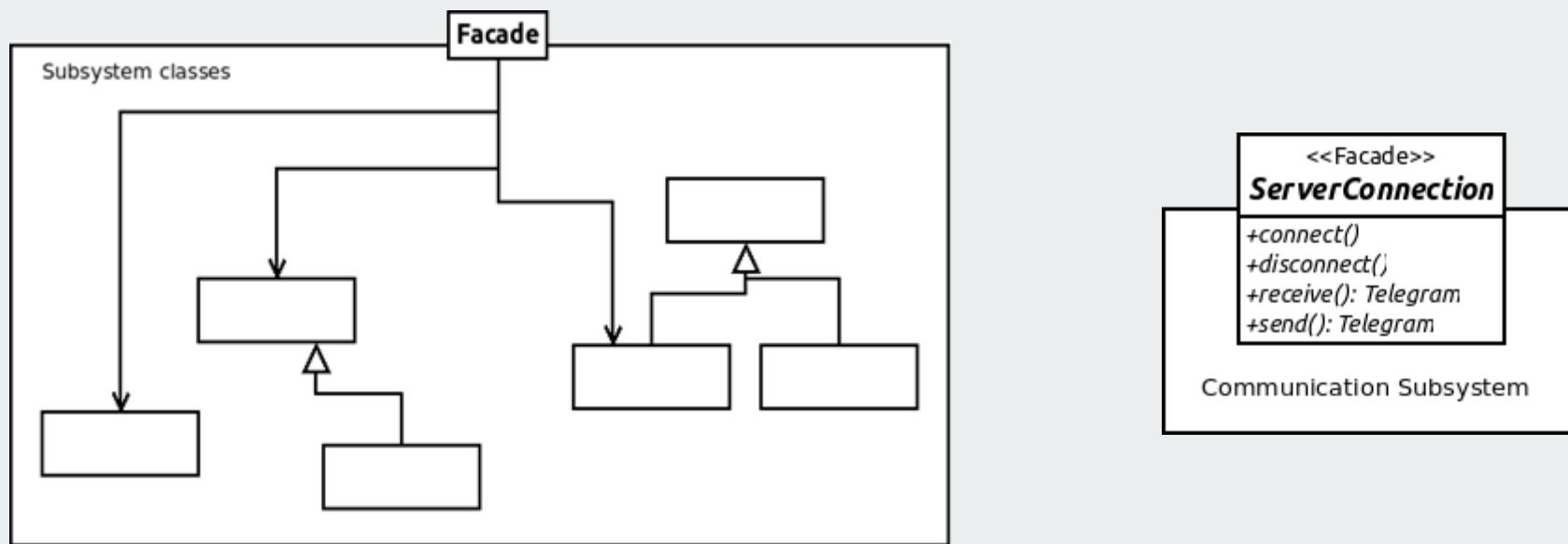


Strukturmuster - Facade (2/5)

294

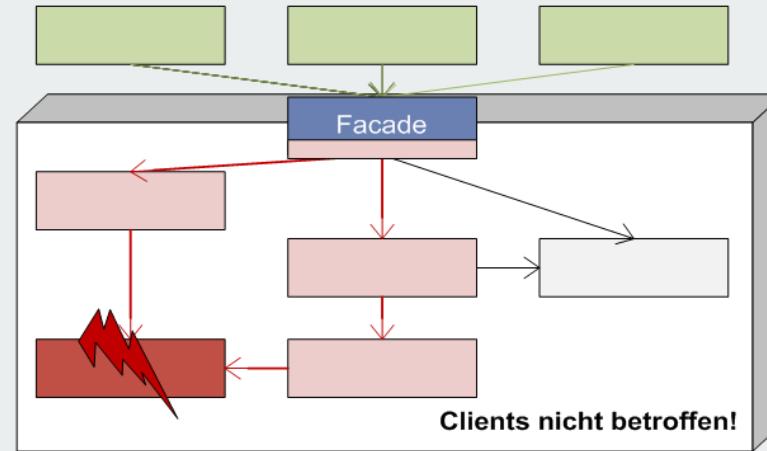
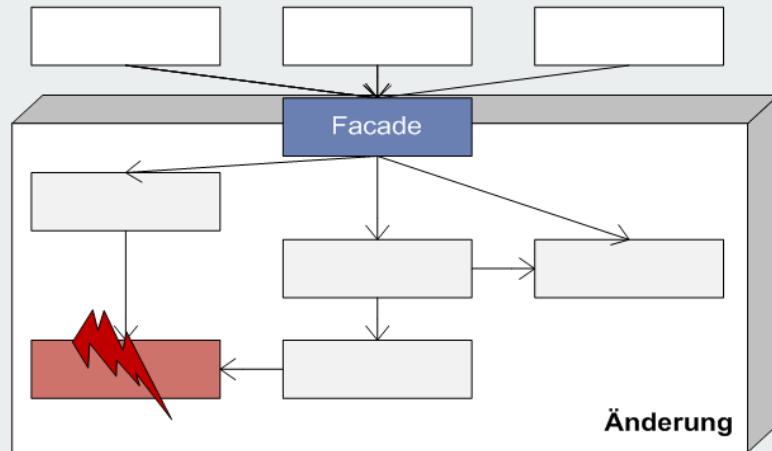
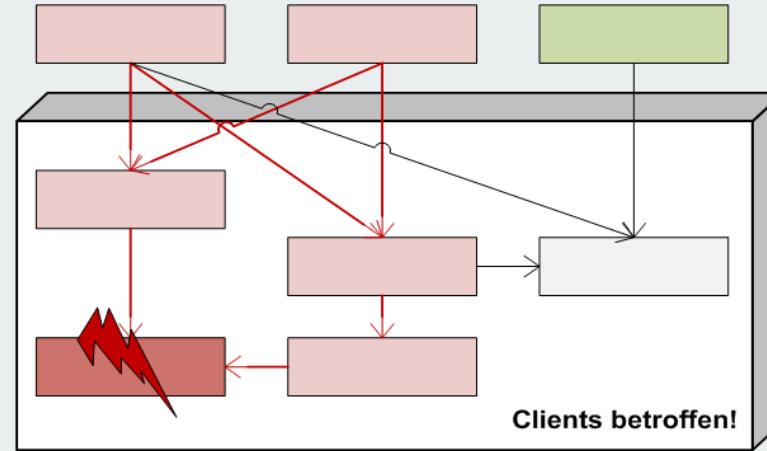
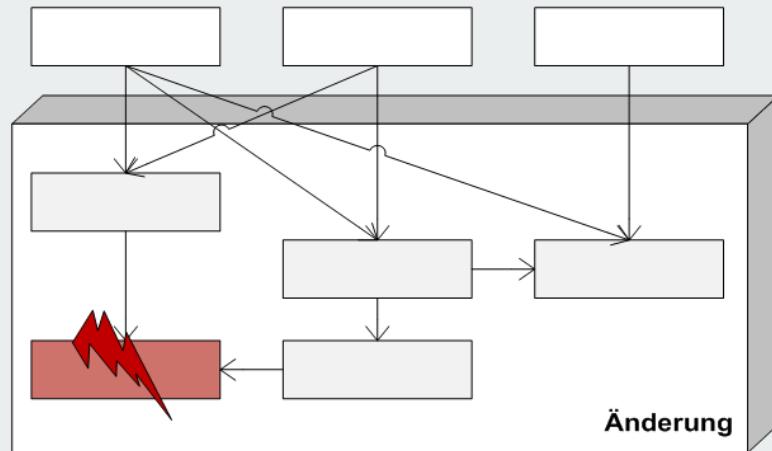
- Anwendungsfälle

- Unterteilung eines komplexen Systems in mehrere einfache Subsysteme.
- Fassade fördert die lose Kopplung.
- Unterteilung eines Systems in Schichten und der Kommunikation derer über Fassaden.
- Reduzierung der Abhängigkeiten zwischen dem Client und dem benutzten Subsystem.



Strukturmuster - Facade (3/5)

295



Strukturmuster - Facade (4/5)

296

```
public class HotelBooker {  
    public List<Hotel> getHotelNamesFor(Date from, Date to) {  
        //returns hotels available in the particular date range  
    }  
}
```

```
public class FlightBooker {  
    public List<Flight> getFlightsFor(Date from, Date to) {  
        //returns flights available in the particular date range  
    }  
}
```

```
public class TravelFacade {  
    private HotelBooker hotelBooker;  
    private FlightBooker flightBooker;  
  
    public Travels getAvailableTravels(Date from, Data to) {  
        List<Flight> flights = flightBooker.getFlightsFor(from, to);  
        List<Hotel> hotels = hotelBooker.getHotelsFor(from, to);  
  
        //process  
    }  
}
```

```
public class FacadeTester {  
    public static void main(String[] args) {  
        TravelFacade facade = new TravelFacade();  
        facade.getAvailableTravels(from, to);  
    }  
}
```

Strukturmuster - Facade (5/5)

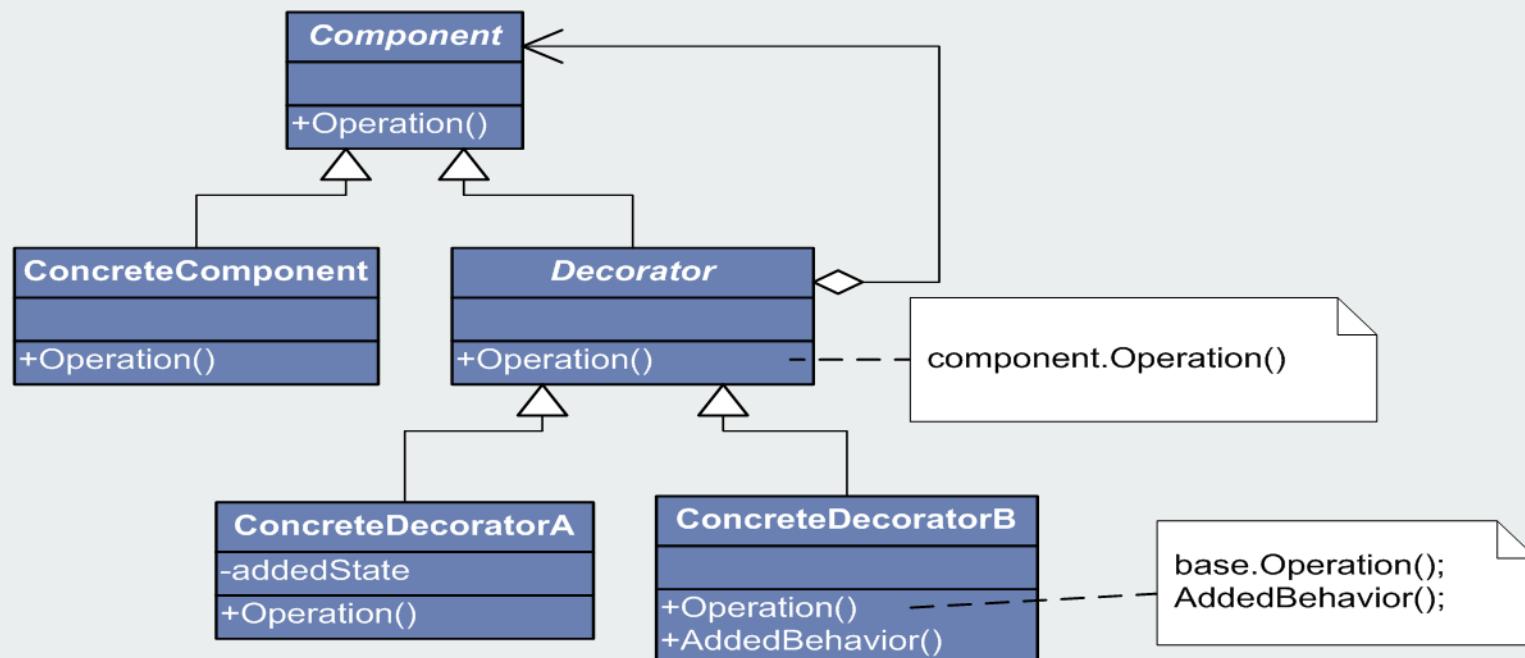
297

- Vorteile
 - **Vereinfachte Schnittstelle.** Der Client kann ein komplexes System einfacher verwenden, da die Anzahl der Objekte, die der Client kennen muss, reduziert wird.
 - **Entkopplung** des Client vom Subsystem. Da der Client nur noch gegen die Fassade arbeiten, ist er unabhängig von Änderungen im Subsystem.
 - **Wartungen** und Modifikationen am Subsystem bedeuten nur noch Änderungen innerhalb des Systems und höchstens der Fassadenimplementierung. Die Schnittstelle der Fassade nach außen bleibt davon unverändert.

Strukturmuster - Decorator (1/5)

298

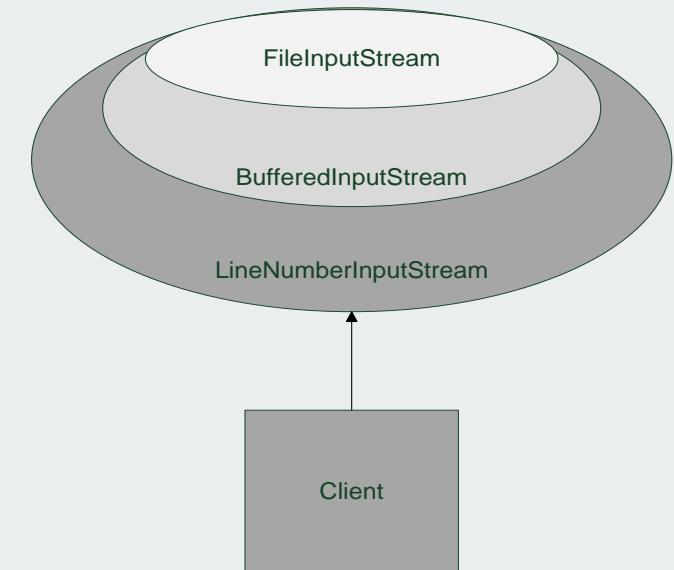
- „Attach additional responsibilities to an object dynamically . Decorators provide a flexible alternative to subclassing for extending functionality.“



Strukturmuster - Decorator (2/5)

299

- Anwendungsfälle
 - Alternative zur statischen Unterklassenbildung (Vererbung) von Objekten
 - Hinzufügen oder Entfernen von Funktionalitäten
 - Erweiterung von final Klassen, die somit nicht vererbbar sind
 - Wenn Funktionalitätserweiterung mittels statischer Vererbung impraktikabel ist
 - z. B. bei unüberschaubarer Anzahl von Klassen, die bei Beachtung jeder möglichen Erweiterungskombination entstehen würde
- Praktische Beispiele:
 - Java IO API, BufferedInputStream erweitert die Schnittstelle um einen Buffer
 - Verschiedene Autotypen mit verschiedenen Eigenschaften (Tieferlegung, Spoiler)
 - Gericht mit diversen Beilagen
 - Verschiedene Telefontypen (Handy, Smartphone, klassisches Telefon) mit variablen Verhalten (Vibration, Klingeln, Lautlos, Leuchten)



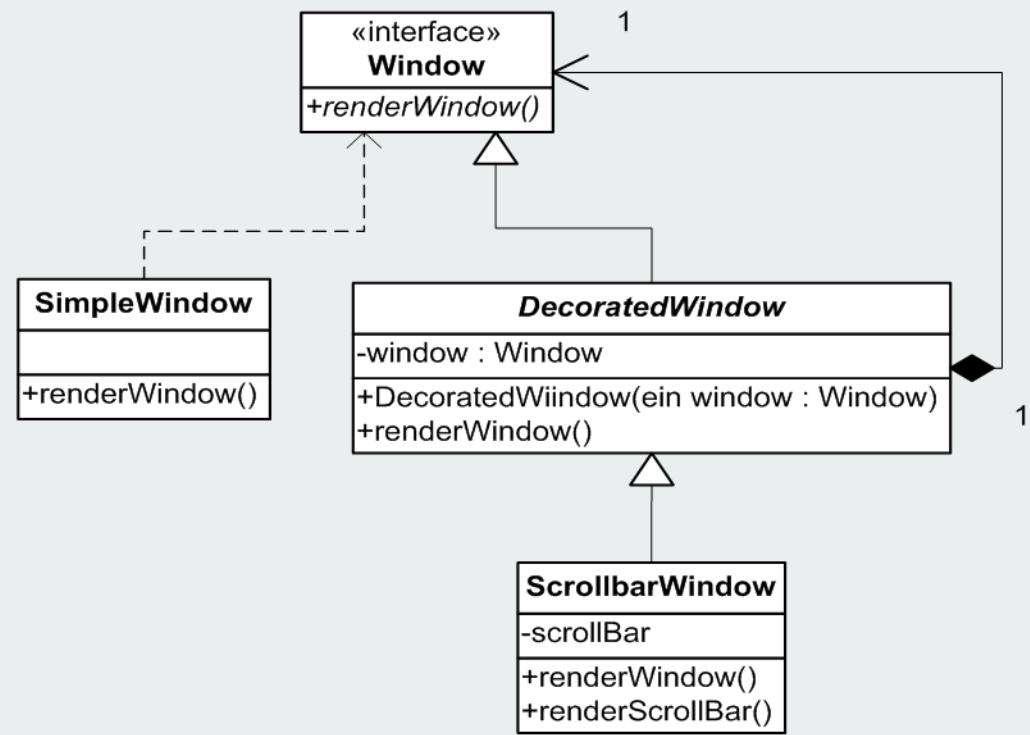
Strukturmuster - Decorator (3/5)

300

```
public interface Window {  
    public void renderWindow();  
}
```

```
public class SimpleWindow implements Window {  
    public void renderWindow() {  
        // implementation of rendering details  
    }  
}
```

```
public class DecoratedWindow implements Window {  
    private Window window = null;  
  
    public DecoratedWindow(Window window) {  
        this.window = window;  
    }  
  
    public void renderWindow() {  
        window.renderWindow();  
    }  
}
```



Strukturmuster - Decorator (4/5)

301

```
public class ScrollableWindow extends DecoratedWindow {  
    private Object scrollBar = null;  
  
    public ScrollableWindow(Window window) {  
        super(window);  
    }  
  
    public void renderWindow() {  
        super.renderWindow();  
        renderScrollBar();  
    }  
  
    private void renderScrollBar() {  
        scrollBar = new Object(); //prepare scrollbar  
        // render scrollbar  
    }  
}
```

```
public class DecoratorTester {  
    public static void main(String[] args) {  
        Window window = new SimpleWindow();  
        window.renderWindow();  
  
        // at some point later maybe text size becomes larger than the window thus the scrolling behavior must be added  
        window = new ScrollableWindow(window); // decorate old window, now window object has additional behavior / state  
  
        window.renderWindow();  
    }  
}
```

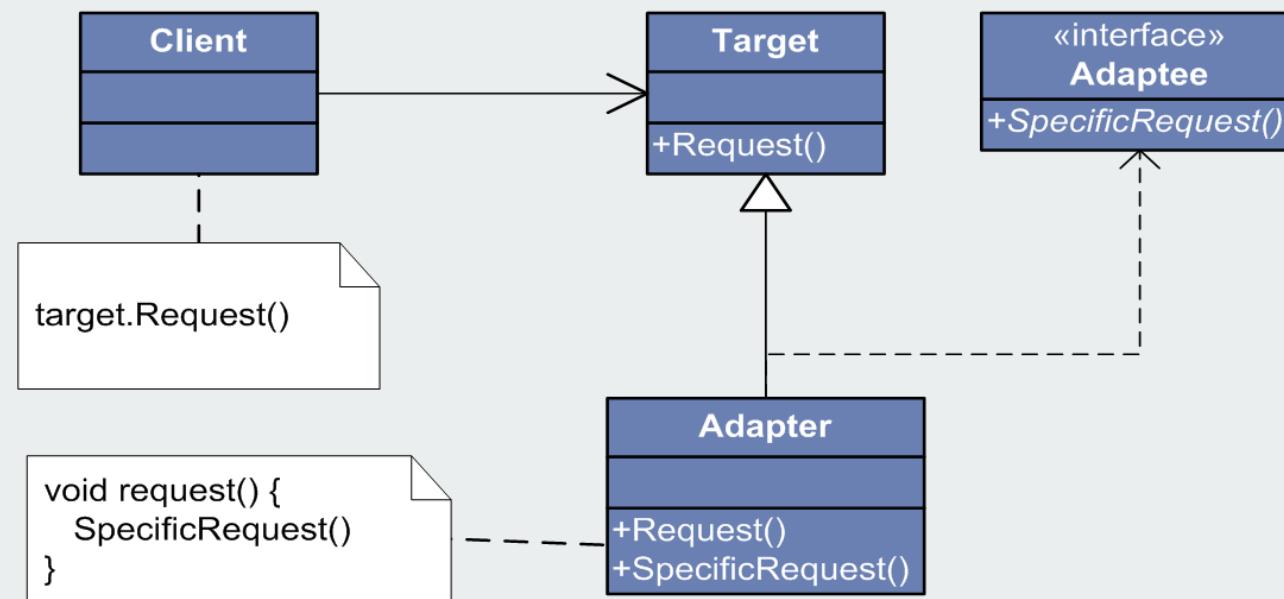
Strukturmuster - Decorator (5/5)

- Vorteile
 - Decorator bietet **mehr Möglichkeiten** als die Vererbung, da dieser nicht statisch und damit ein Verhalten der UnterkLASSE festlegt.
 - **Schlanke, kohäsive Klassen.** Jeder Decorator repräsentiert genau eine Funktion und nichts anderes. Dadurch steigt die Klassenkohäsion und der entsprechende Code wird leichter wart- und erweiterbar.
 - Vermeidung von langen und unübersichtlichen Vererbungshierarchien.
 - Durch Kombination von wenigen Decorator-Objekten kann man die Funktionalität des Objektes erheblich erweitern.
- Nachteile
 - Dekorierte Objekte werden vom eigentlichen Decorator umhüllt. Damit lässt sich die Identität der Objekte schwer feststellen.
 - **Hohe Objektanzahl.** Jedes zusätzliche Feature bedarf eines neuen Decoratorobjektes. Schnell kann die Anzahl der vielen kleinen, ähnlichen Objekten und ihr Initialisierungscode unübersichtlich werden. Kann durch eine Factory behoben werden.
 - Decorator-Pattern ist ein dynamisches Verfahren, damit ist im Allgemeinen das Laufzeitverhalten schlechter als bei der klassischen Vererbung.

Strukturmuster - Adapter (1/3)

303

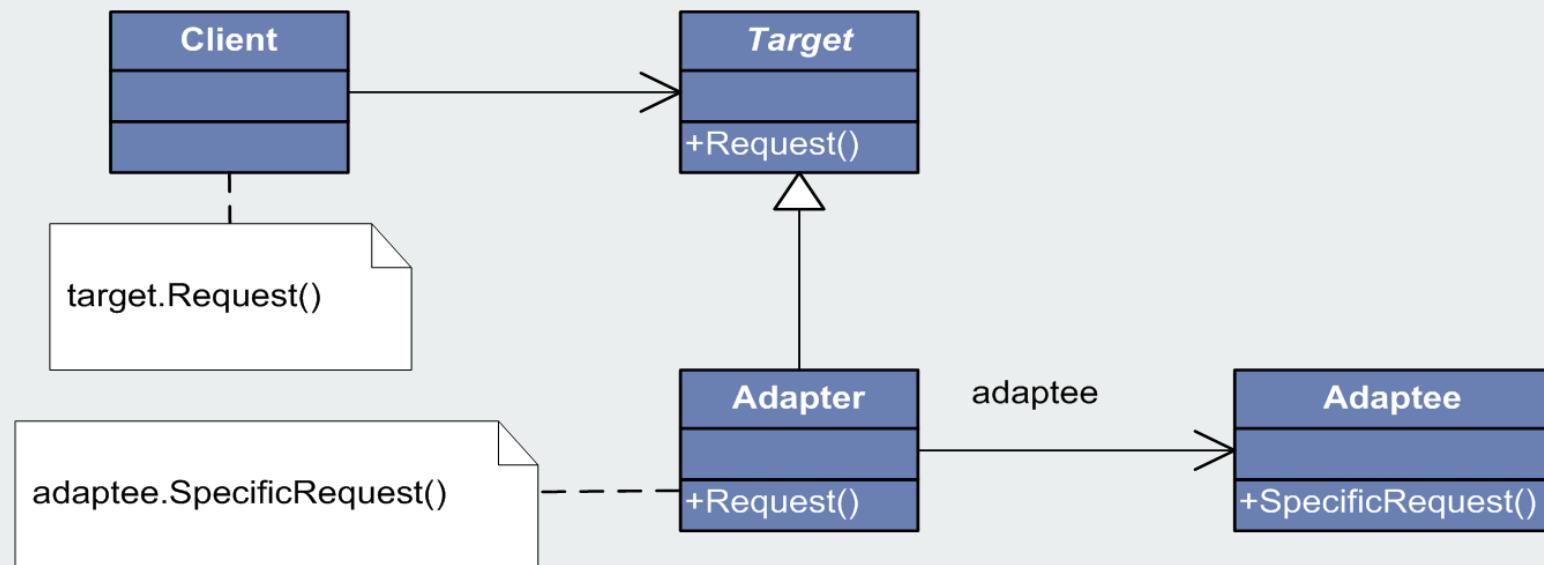
- „Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.“
- **Class Adapter** uses multiple inheritance.



Strukturmuster - Adapter (2/3)

304

- „Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.“
- **Object Adapter** relies on object composition.



Strukturmuster - Adapter (3/3)

305

```
public class AdapterTester {
    public static void main(String[] args) {
        Sorter sorter = new SortListAdapter();
        sorter.sort(new int[] { 34, 2, 4, 12, 1 });
    }
}
```

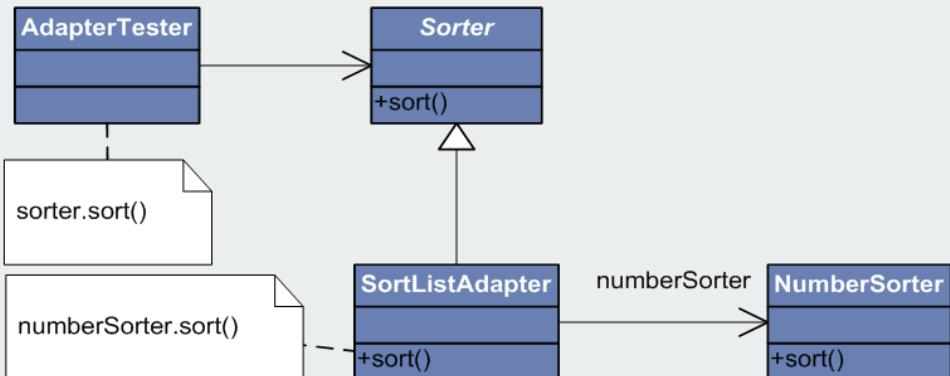
```
//this is our Target interface
public interface Sorter {
    int[] sort(int[] numbers);
}
```

```
public class SortListAdapter implements Sorter {

    public int[] sort(int[] numbers) {
        List<Integer> numberList = convertArrayToList(numbers);

        NumberSorter numberSorter = new NumberSorter();
        numberList = numberSorter.sort(numberList);

        return convertListToArray(numberList);
    }
}
```

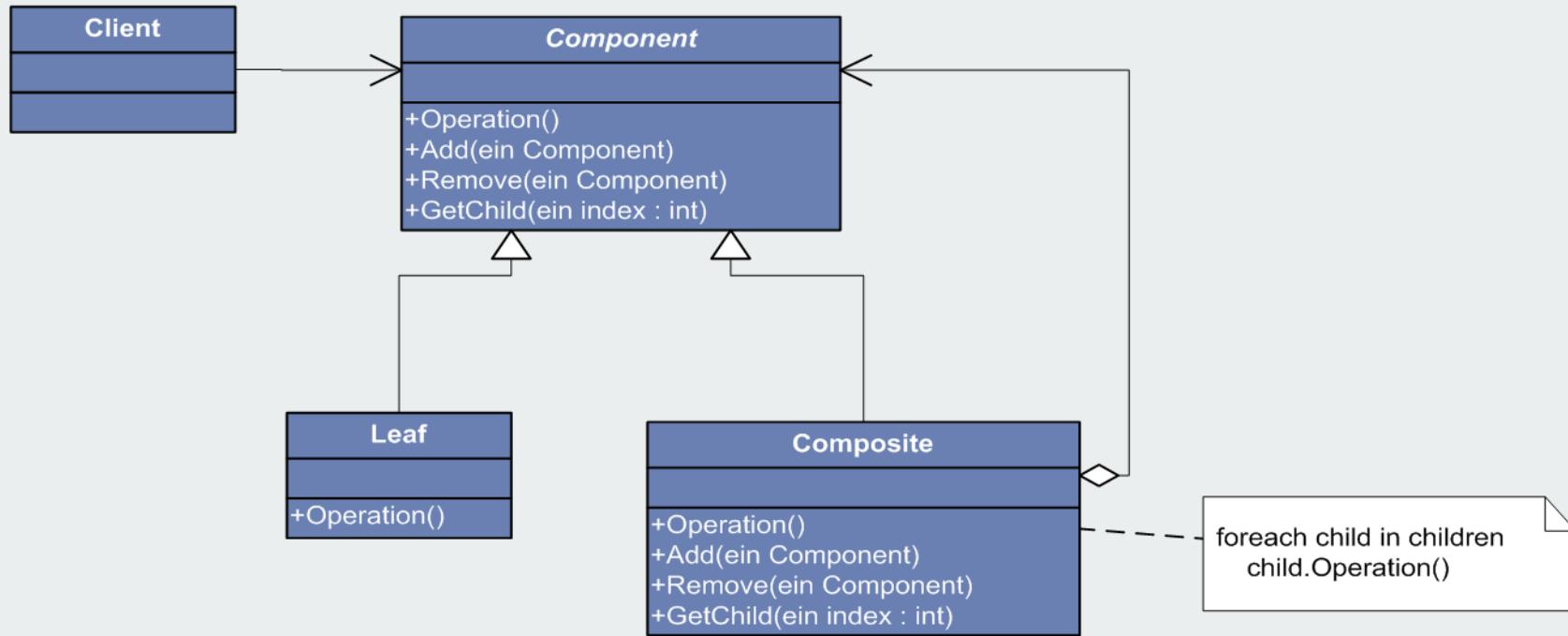


```
// This is our adaptee, a third party implementation of a number sorter that deals with Lists, not arrays.
public class NumberSorter {
    public List<Integer> sort(List<Integer> numbers) {
        return sort(numbers);
    }
}
```

Strukturmuster - Composite (1/5)

306

- „Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.“

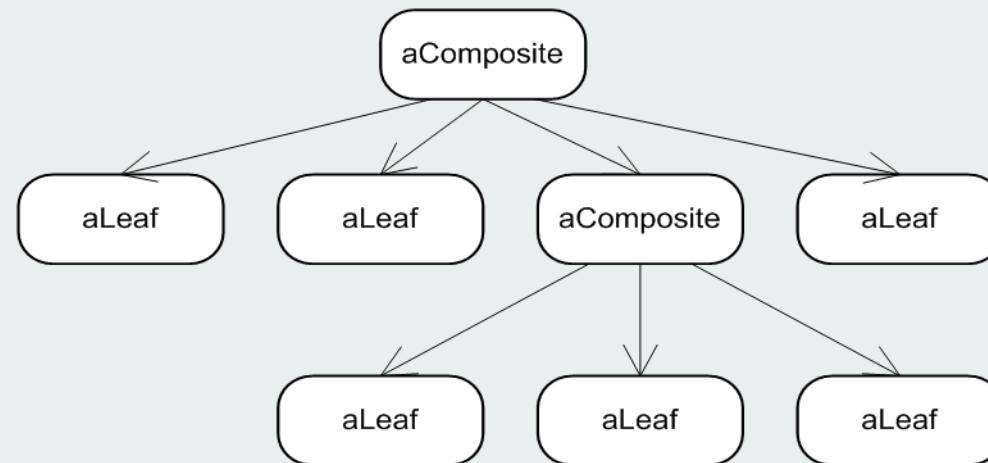


Strukturmuster - Composite (2/5)

307

- Anwendungsfälle

- Manipulation einer hierarchischen Sammlung von einfachen und komplexen Objekten.
- Modellierung einer baumartige Struktur aus Objekten
 - **Dateisystem.** Dateien und Ordner können mit dem Composite Pattern modelliert werden. Dateien repräsentieren die Leafs und die Ordner die Composites, da sie wiederum weitere Dateien und Ordnern enthalten können.
 - **Menüs.** Menüs bestehen bekanntlich aus einem Wurzeleintrag (Composite) und Befehlen (Leafs).
 - Hierarchie von Grafikobjekten z. B. in Java / AWT (Container & Component)



Strukturmuster - Composite (3/5)

308

```
public interface Component {  
    public void paint();  
}
```

```
public abstract class Composite implements Component {  
  
    private List<Component> components = new ArrayList<Component>();  
  
    @Override  
    public void paint() {  
        for (Component component : this.components) {  
            component.paint();  
        }  
    }  
  
    public void add(Component component) {  
        this.components.add(component);  
    }  
  
    public void remove(Component component) {  
        if (this.components.contains(component))  
            this.components.remove(component);  
    }  
  
    public Component get(int index) {  
        return this.components.get(index);  
    }  
}
```

Strukturmuster - Composite (4/5)

```
public class Sheet extends Composite {  
    public void paint() {  
        System.out.println("Sheet"); super.paint();  
    }  
}
```

```
public class Row extends Composite {  
    public void paint() {  
        System.out.println("Row"); super.paint();  
    }  
}
```

```
public class Column extends Composite {  
    public void paint() {  
        System.out.println("Column"); super.paint();  
    }  
}
```

```
public class CompositeTester {  
    public static void main(String[] args) {  
        Composite sheet = new Sheet();  
  
        Composite r1 = new Row();  
        r1.add(new Column());  
        r1.add(new Column());  
        sheet.add(r1);  
  
        Composite r2 = new Row();  
        r2.add(new Column());  
        r2.add(new Column());  
        sheet.add(r2);  
  
        sheet.paint();  
    }  
}
```

```
Sheet  
Row  
Column  
Column  
Row  
Column  
Column
```

Strukturmuster - Composite (5/5)

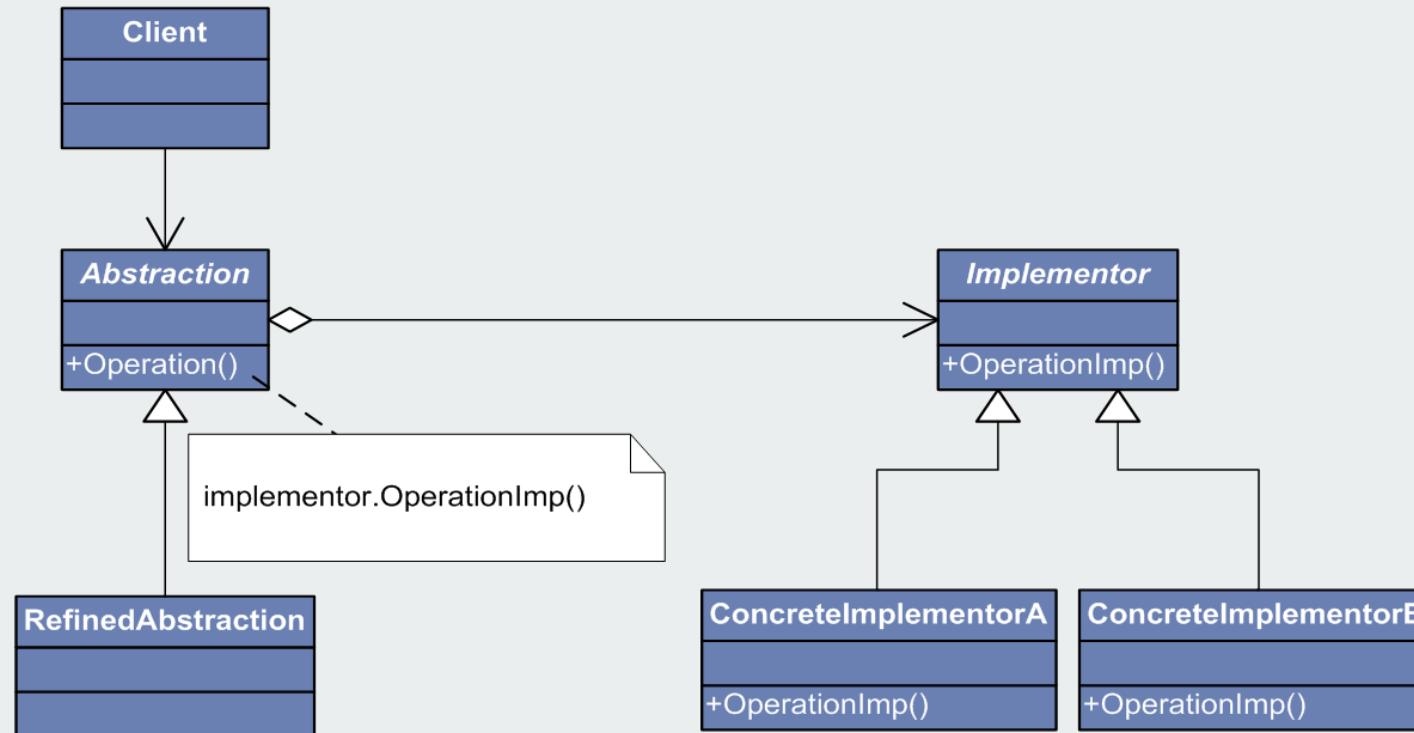
310

- Vorteile
 - **Einfache Implementierung von verschachtelten Strukturen**
 - **Elegantes Arbeiten mit der Baumstruktur.** Der Client kann leicht durch die Hierarchie traversieren, Operationen aufrufen und diese verwalten also neue Elemente hinzufügen und bestehende löschen.
 - **Flexibilität und Erweiterbarkeit.** Einfaches Hinzufügen von neuen Elementen (Leafs oder Composites) .
- Nachteile
 - **Verallgemeinerung des Entwurfs.** Soll ein bestimmtes zusammengesetztes Element nur eine feste Anzahl von Kindern haben, oder nur bestimmte Kinder, so kann dies erst zur Laufzeit (statt zur Übersetzungszeit) überprüft werden.

Strukturmuster - Bridge

311

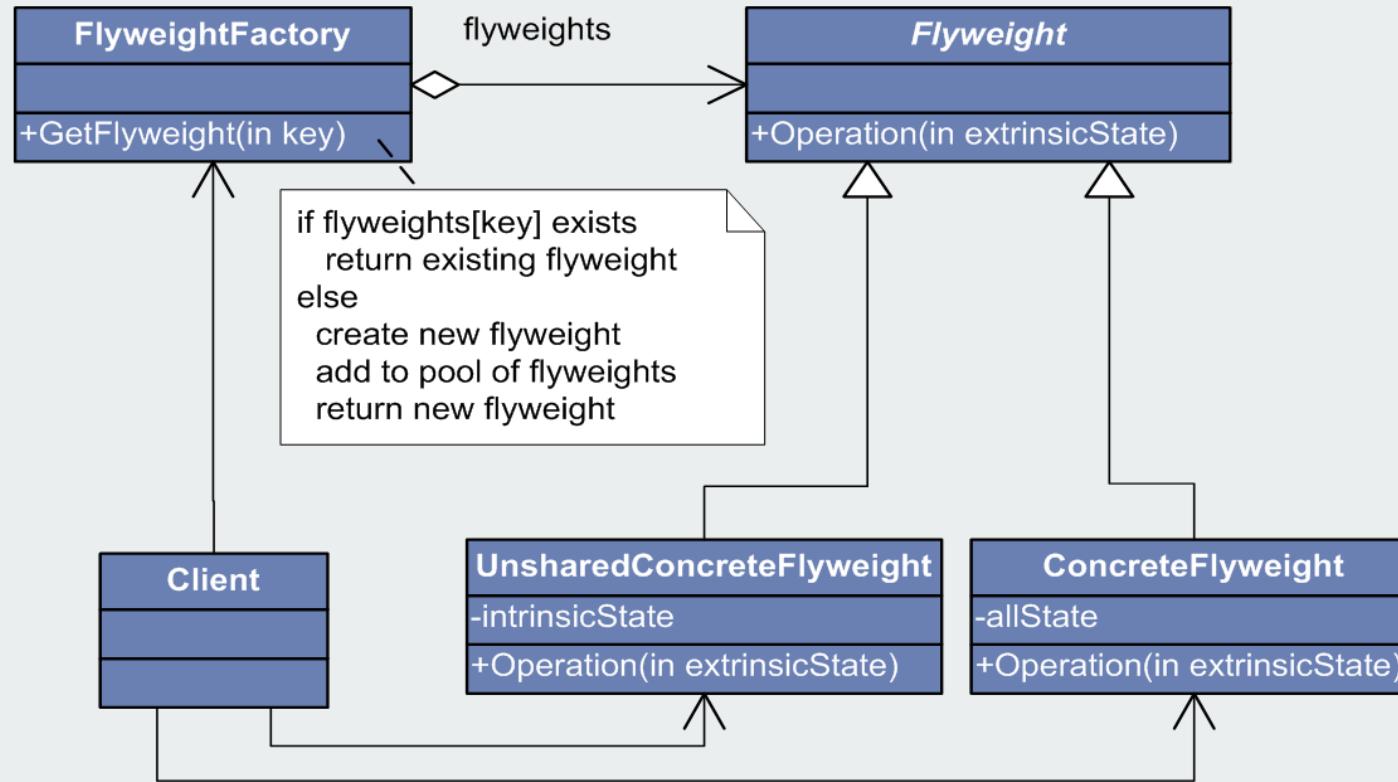
- „Decouple an abstraction from its implementation so that the two can vary independently.“



Strukturmuster - Flyweight

312

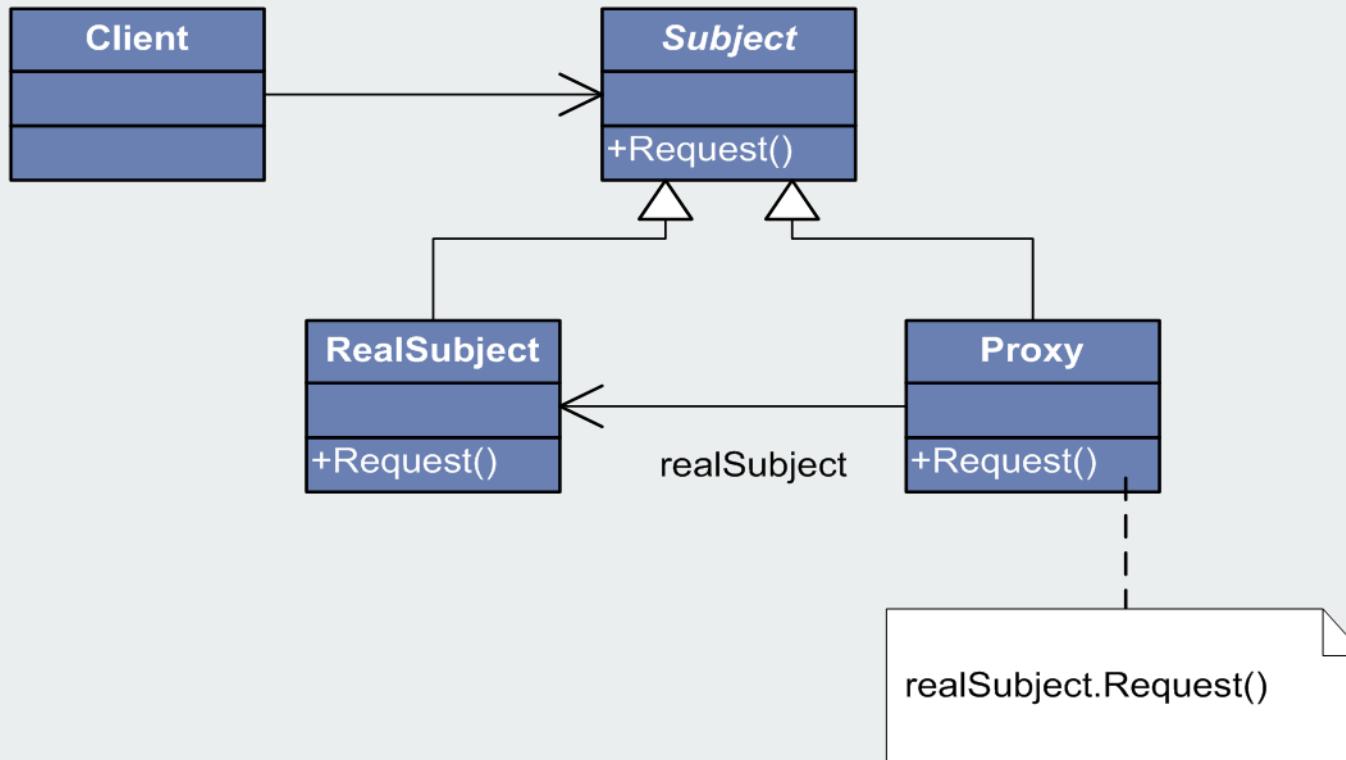
- „Use sharing to support large numbers of fine-grained objects efficiently.“



Strukturmuster - Proxy

313

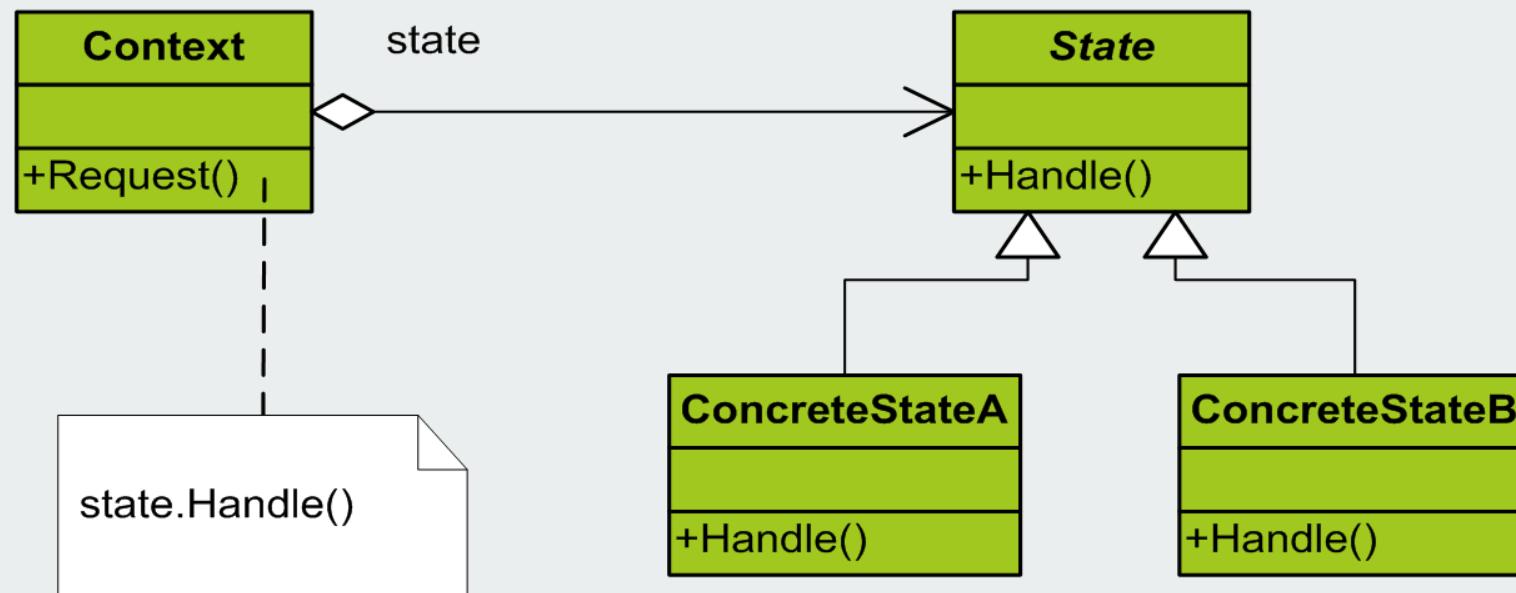
- „Provide a surrogate or placeholder for another object to control access to it.“



Verhaltensmuster - State (1/4)

314

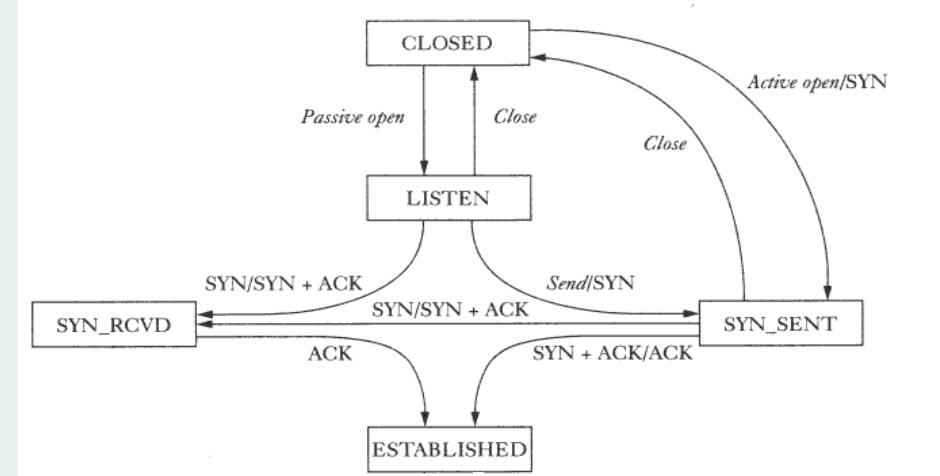
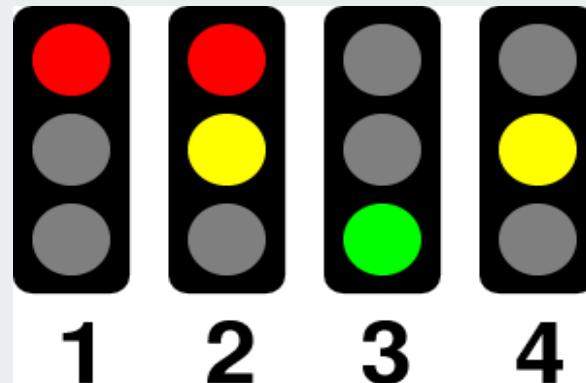
- „Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.“



Verhaltensmuster - State (2/4)

315

- Anwendungsfälle
 - Ein Objekt soll sein äußereres Verhalten zur Laufzeit in Abhängigkeit von seinen Zustand ändern.
 - **Zustandsautomaten** z. B. Parser. Die Funktionsweise vieler Textparser ist zwangsläufig zustandsbasiert. So muss ein Compiler, der Quellcode parst, ein Zeichen abhängig von den zuvor gelesen Zeichen interpretieren. So handelt es sich in vielen Programmiersprachen bei allen Zeichen, die einem "/*" folgen, um Kommentare bis zu einem "*/" (--> Kommentarzustand).
 - Repräsentation von Zuständen von **Netzwerkverbindungen**



Verhaltensmuster - State (3/4)

316

```
public class MP3PlayerContext {  
    private State state;  
    private MP3PlayerContext(State state) {  
        this.state = state;  
    }  
  
    public void play() {  
        state.pressPlay(this);  
    }  
  
    public void setState(State state) {  
        this.state = state;  
    }  
    public State getState() {  
        return state;  
    }  
}
```

```
public interface State {  
    public void pressPlay(MP3PlayerContext context);  
}
```

```
public class PlayingState implements State {  
    public void pressPlay(MP3PlayerContext context) {  
        context.setState(new StandByState());  
    }  
}
```

```
public class StandByState implements State {  
    public void pressPlay(MP3PlayerContext context) {  
        context.setState(new PlayingState());  
    }  
}
```

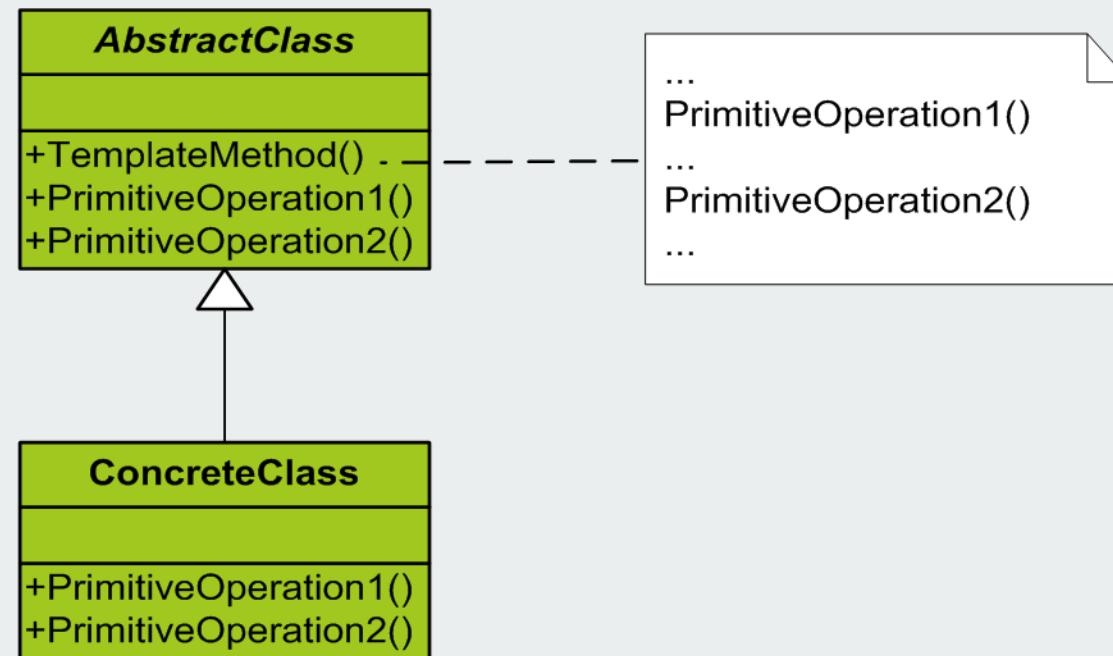
Verhaltensmuster - State (4/4)

- Vorteile
 - **Erweiterbarkeit und Änderungsstabilität.** Einfache Integration von neuen Zuständen, ohne dabei bestehenden Code ändern zu müssen. Änderungen am zustandsabhängigen Verhalten betreffen lediglich eine Zustandsklasse und nicht den Context.
 - **Verständlichkeit.** Durch die hohe Kohäsion und der Delegation von Verantwortlichkeiten (das Verhalten eines Zustands ist im entsprechenden Zustandsobjekt selbst gekapselt / lokalisiert) wird der Code leicht verständlich.
 - **Explizite Zustandsübergänge.** Durch die Einführung von Objekten für jeden Zustand wird der Zustandswechsel explizit. Außerdem werden inkonsistente Zustände verhindert, da ein Zustandswechsel ein atomarer Befehl ist (**Setzen einer Variablen**).
 - Der Entwurf ist hinsichtlich späterer Änderungen **weniger fehlerträchtig** als breite, sich wiederholende if-else-Konstrukte.
- Nachteile
 - **Erhöhte Klassenanzahl.** Ist weniger kompakt als eine einzige Klasse. Allerdings ist gerade die Aufteilung des Verhaltens von einer einzigen unübersichtlichen Klasse auf mehrere Zustandsobjekte das Ziel des Entwurfsmusters.

Verhaltensmuster - Template Method (1/4)

318

- „Define a skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.“



Verhaltensmuster - Template Method (2/4)

319

- Anwendungsfälle

- Definieren einer Schablone, die ähnliche Algorithmen implementiert und die einzelnen Ausführungsschritte an Unterklassen delegiert.
- Gemeinsames Verhalten von Subklassen in ein Superklasse verschoben werden sollen, um Code-Duplikation zu vermeiden.
- Modellierung von festen Abläufen und der Vorgabe von Variationspunkten.

Verhaltensmuster - Template Method (3/4)

320

```
public abstract class CrossCompiler {  
    public final void crossCompile() {  
        collectSource();  
        compileToTarget();  
    }  
  
    // Template methods  
    protected abstract void collectSource();  
    protected abstract void compileToTarget();  
}
```

```
public class IPhoneCompiler extends CrossCompiler {  
    protected void collectSource() {  
        // anything specific to this class  
    }  
  
    protected void compileToTarget() {  
        // iphone specific compilation  
    }  
}
```

```
public class AndroidCompiler extends CrossCompiler {  
    protected void collectSource() {  
        // anything specific to this class  
    }  
  
    protected void compileToTarget() {  
        // android specific compilation  
    }  
}
```

```
public class Client {  
    public static void main(String[] args) {  
        CrossCompiler iphone = new IPhoneCompiler();  
        iphone.crossCompile();  
  
        CrossCompiler android = new AndroidCompiler();  
        android.crossCompile();  
    }  
}
```

Verhaltensmuster - Template Method (4/4)

321

■ Vorteile

- Die Verwendung einer Template Method ermöglicht es erbenden Klassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne die Struktur des Algorithmus zu ändern.
- Gemeinsames Verhalten wird in einer Klasse gekapselt
- Alle Methoden müssen bei der Implementierung der Unterklasse überschrieben werden.
- Vermeidung von dupliziertem Code

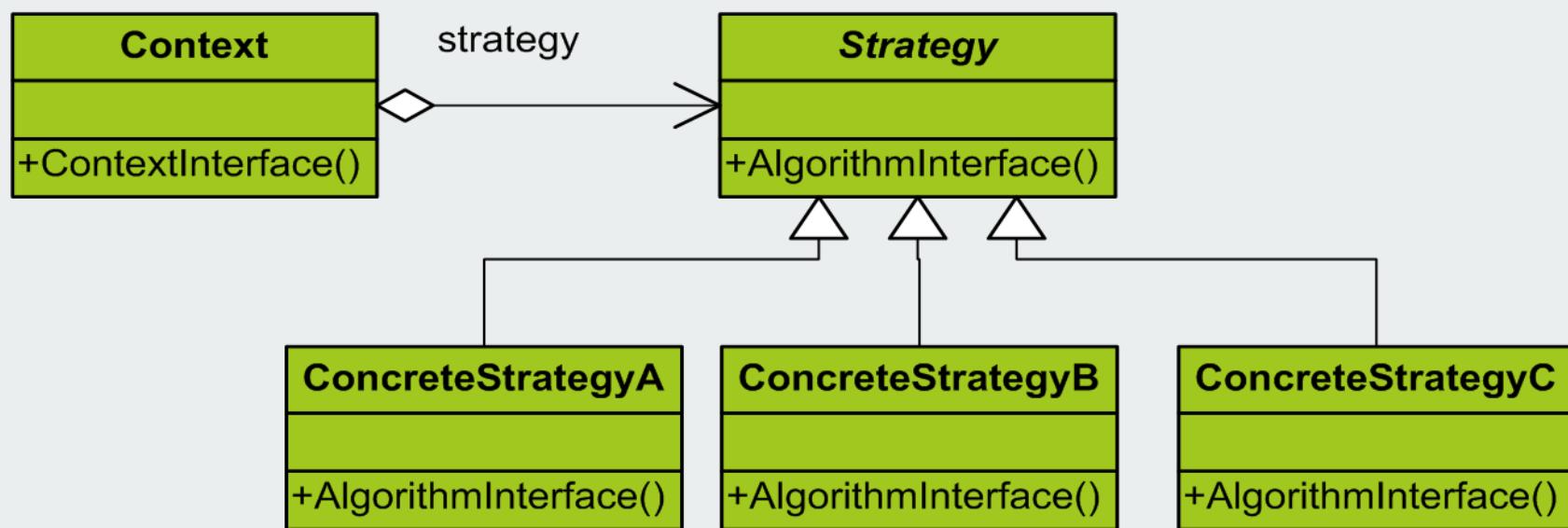
■ Nachteile

- Das Design kann unnötig komplizierter werden, wenn Unterklassen sehr viele Methoden implementieren müssen, um den Algorithmus zu konkretisieren.
- Es kann zum Nachteil werden, wenn der Algorithmus einer fixen Struktur folgt, die nicht mehr geändert werden kann.
- Es muss klar sein, welche Methoden überschrieben werden dürfen bzw. überschrieben werden müssen.

Verhaltensmuster - Strategy (1/6)

322

- „Define a family of algorithms, encapsulate each one, and make them interchangaeble. Strategy lets the algorithm vary independently from clients that use it.“



Verhaltensmuster - Strategy (2/6)

323

- Anwendungsfälle
 - Bei vielen ähnlichen Klassen, die sich **nur im Verhalten unterscheiden**
 - Bei Eingabemasken kann die Logik zur Validierung und Plausibilitätsprüfung von Benutzereingaben gekapselt werden.
 - Verhalten sollte auch dann vom Context entkoppelt werden, wenn unterschiedliche Ausformungen **ein und der selben Funktion** benötigt werden
 - Sortierung einer Collection (Array, List), wobei die konkreten Strategys verschiedene Sortierverfahren repräsentieren.
 - Speicherung in verschiedenen Dateiformaten
 - Packer mit verschiedenen Kompressionsalgorithmen
 - Entkopplung und **Verstecken von komplizierten Algorithmusdetails** vor dem Context
 - Es können Mehrfachverzweigungen (if (...) else if(...) else ...) vermieden werden und dies erhöht die Übersicht des Codes.

Verhaltensmuster - Strategy (3/6)

324

```
public interface CompressionStrategy {  
    public void compressFiles(List<File> files);  
}
```

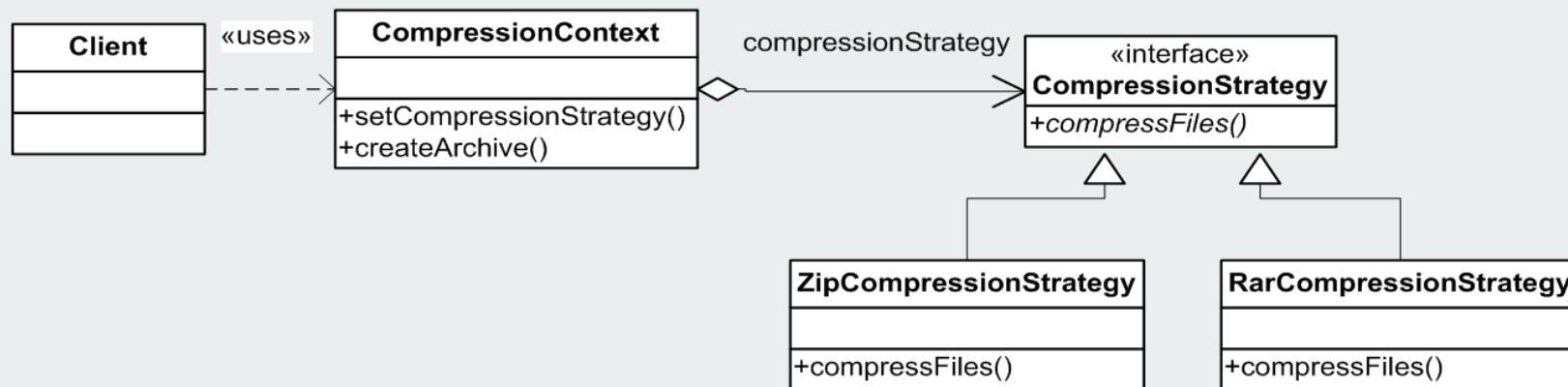
```
public class RarCompressionStrategy implements CompressionStrategy {  
    public void compressFiles(List<File> files) {  
        // using RAR approach  
    }  
}
```

```
public class ZipCompressionStrategy implements CompressionStrategy {  
    public void compressFiles(List<File> files) {  
        // using ZIP approach  
    }  
}
```

```
public class CompressionContext {  
    private CompressionStrategy strategy;  
  
    // this can be set at runtime by the application preferences  
    public void setCompressionStrategy(CompressionStrategy strategy) {  
        this.strategy = strategy;  
    }  
  
    // use the strategy  
    public void createArchive(List<File> files) {  
        this.strategy.compressFiles(files);  
    }  
}
```

Verhaltensmuster - Strategy (4/6)

325



```
public class Client {

    public static void main(String[] args) {
        CompressionContext ctx = new CompressionContext();

        // we could assume context is already set by preferences
        ctx.setCompressionStrategy(new RarCompressionStrategy());

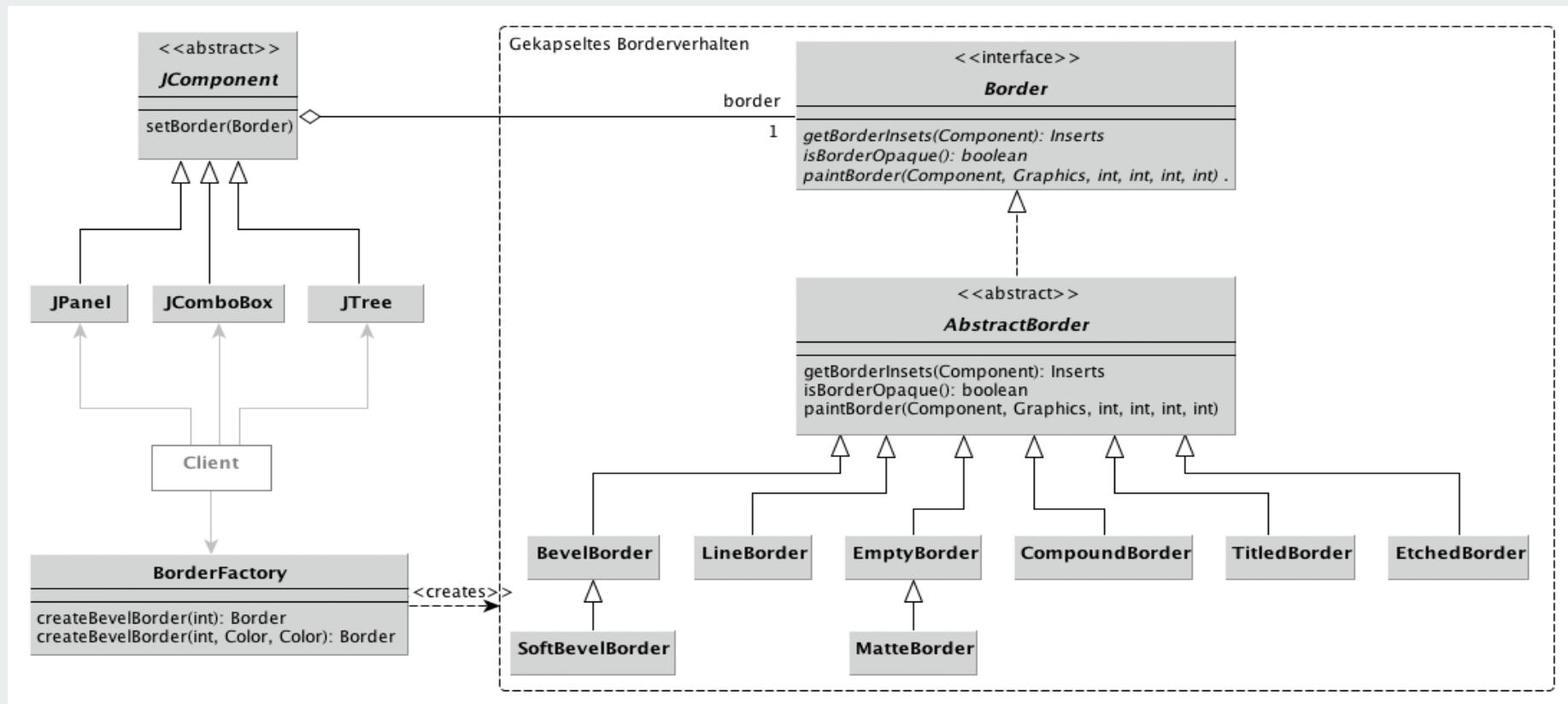
        // get a list of files

        ctx.createArchive(files);
    }
}
```

Verhaltensmuster - Strategy (5/6)

326

- In der Java API wird das Strategy Design Pattern an zahlreichen Stellen angewandt wie z. B. in den **Swing Border Classes**.



Verhaltensmuster - Strategy (6/6)

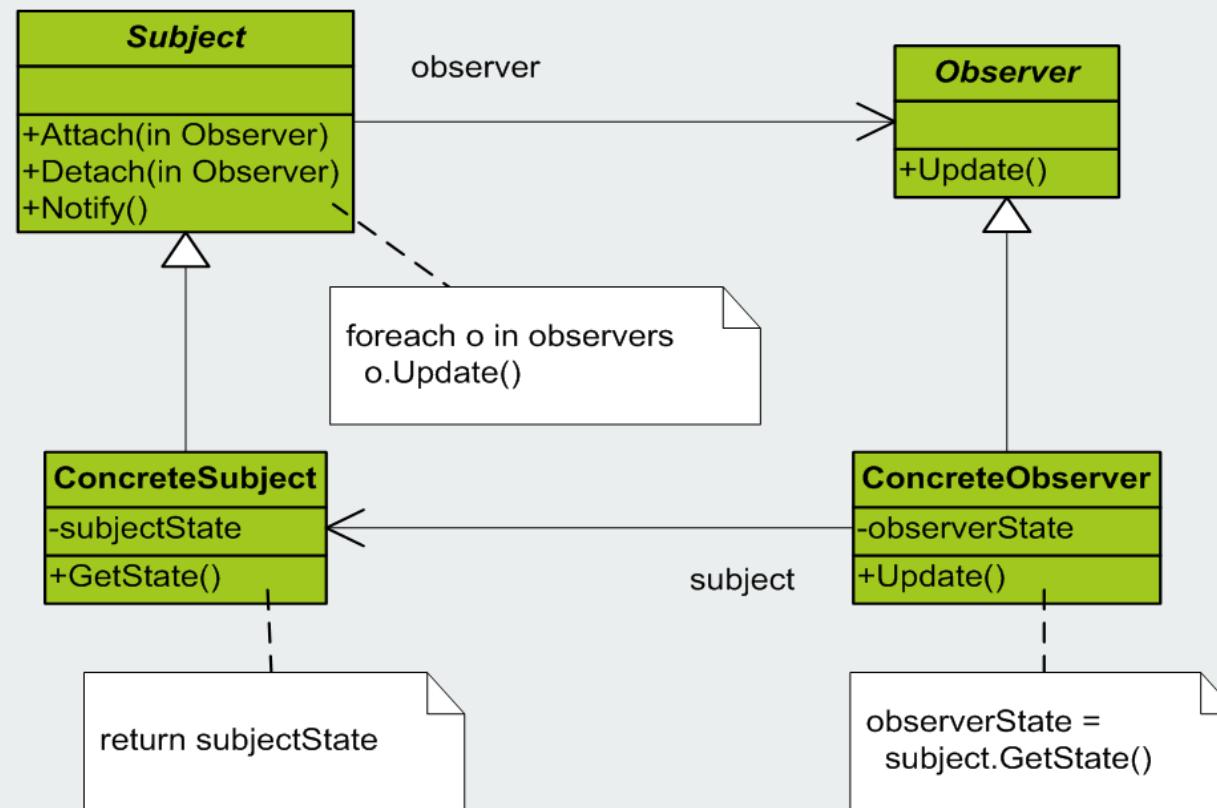
327

- Vorteile
 - **Wiederverwendbarkeit** und **Entkopplung von Context und Verhalten**. Es wird eine Familie von Algorithmen erstellt, welche contextunabhängig verwendet werden kann. Auf der einen Seite können neue Contextobjekte die bestehenden Strategien nutzen.
 - **Dynamisches Verhalten**. Das Verhalten des Contexts kann durch entsprechende Setter zur Laufzeit geändert werden.
 - Es wird die Auswahl aus verschiedenen Implementierungen ermöglicht und dadurch erhöht sich die **Flexibilität und die Wiederverwendbarkeit**.
 - Es können **Mehrfachverzweigungen** vermieden werden und dies erhöht die Übersicht des Codes.
 - **Alternativimplementierung**. Auch kann die selbe Funktion (z. B. Sortieren) durch verschiedene Implementierungen angeboten werden, die sich aber in nichtfunktionaler Hinsicht unterscheiden (Performance, Speicherbedarf).
- Nachteile
 - Clients müssen die unterschiedlichen Strategien kennen, um zwischen ihnen auswählen und den Kontext initialisieren zu können.
 - Problem kann mit einer Factory behoben werden. Damit lässt sich die Erstellungslogik aus dem Client in eine Factory auslagern.

Verhaltensmuster - Observer (1/4)

328

- „Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.“

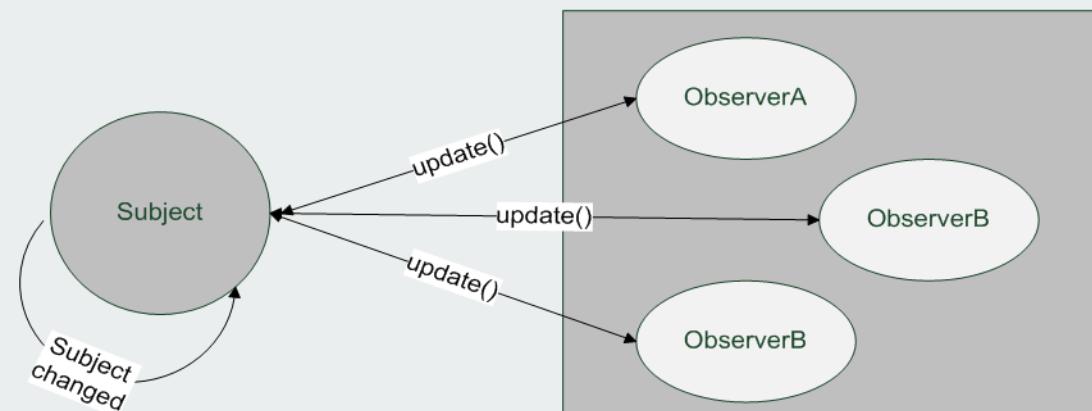


Verhaltensmuster - Observer (2/4)

329

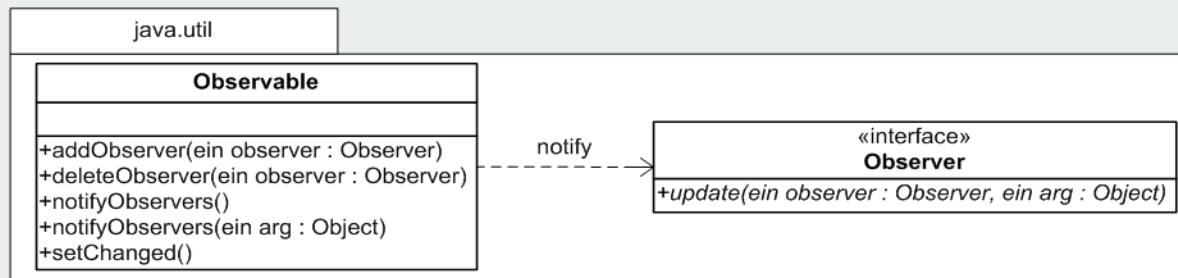
Anwendungsfälle

- Wenn Veränderung eines Objekts die Modifikation eines oder mehrerer Objekte nötig macht.
 - GUIs (User verändert Daten, neue Daten müssen in allen GUI-Komponenten aktualisiert werden)
 - MVC-Pattern (Model-View-Controller) bei der View-Model-Kommunikation
 - Bei jedem Sekundentakt eines Zeitgeber muss sowohl die Digitaluhr als auch die Analoguhr aktualisiert werden.
- Objekte andere Objekte benachrichtigen sollen, ohne dabei näheres über das zu benachrichtigende Objekt zu wissen.
 - Ermöglicht lose Kopplung der Objekte



Verhaltensmuster - Observer (3/4)

330



```
public class Screen implements Observer {  
    public void update(Observable o, Object arg) {  
        // act on the update  
    }  
}
```

```
public class ObserverTester {  
    public static void main(String[] args) {  
        Screen screen = new Screen();  
        DataStore dataStore = new DataStore();  
  
        // register observer  
        dataStore.addObserver(screen);  
  
        // do something with dataStore  
  
        // send a notification  
        dataStore.notifyObservers();  
    }  
}
```

```
public class DataStore extends Observable {  
    private String data;  
  
    public String getData() {  
        return data;  
    }  
  
    public void setData(String data) {  
        this.data = data;  
        // mark the observable as changed  
        setChanged();  
    }  
}
```

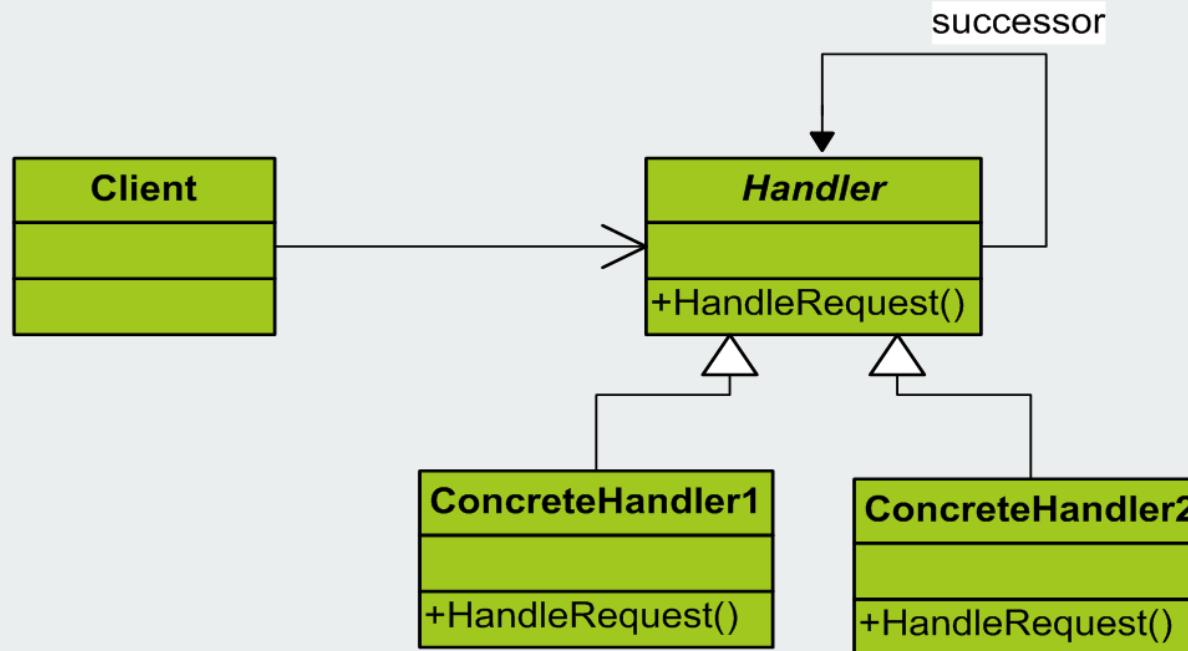
Verhaltensmuster - Observer (4/4)

- Vorteile
 - **Zustandskonsistenz.** Automatische Anpassung des Observerzustands bei Änderungen des Subjects.
 - **Wiederverwendbarkeit.** Durch das Observer Pattern lassen sich Subject und Observer unabhängig voneinander variieren.
 - **Einfache Erweiterung** von verschiedenen Observern, die ein einziges Subject beobachten.
- Nachteile
 - **Aktualisierungskaskaden und -zyklen.** In komplexen Systemen mit vielen Subjects und Observer kann es leicht zu Aktualisierungskaskaden kommen, da die Observer nichts von einander wissen und nicht abschätzen können, welche Folgen eine einzige Modifikation an einem Subject hat.

Verhaltensmuster - Chain of Responsibility

332

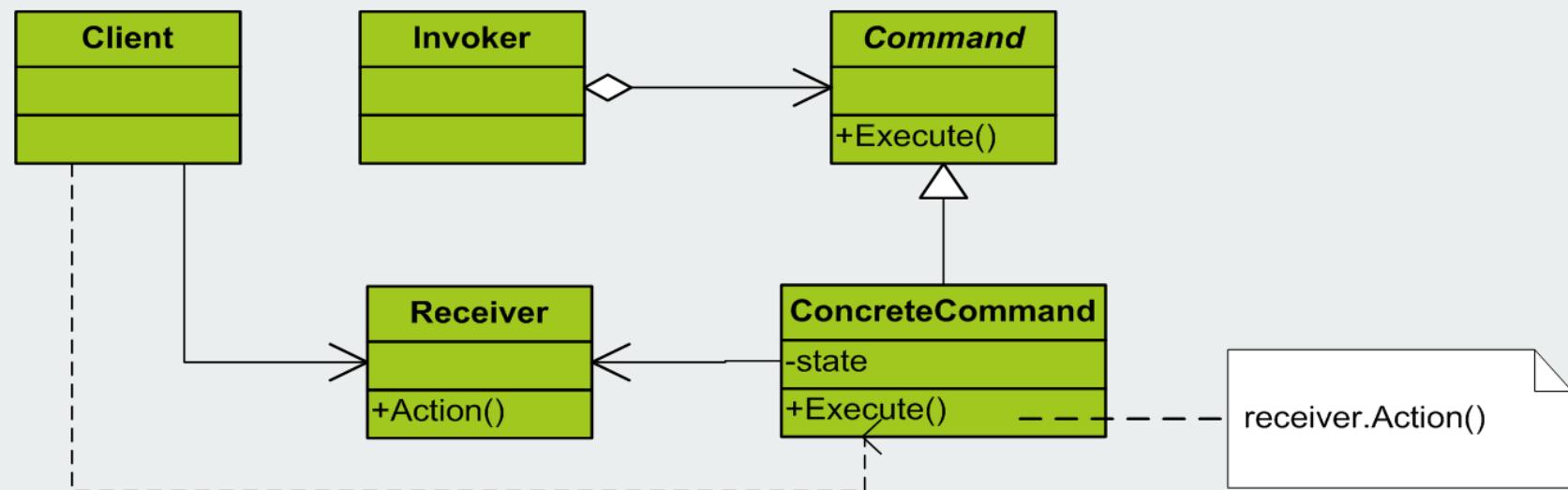
- „Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.“



Verhaltensmuster - Command

333

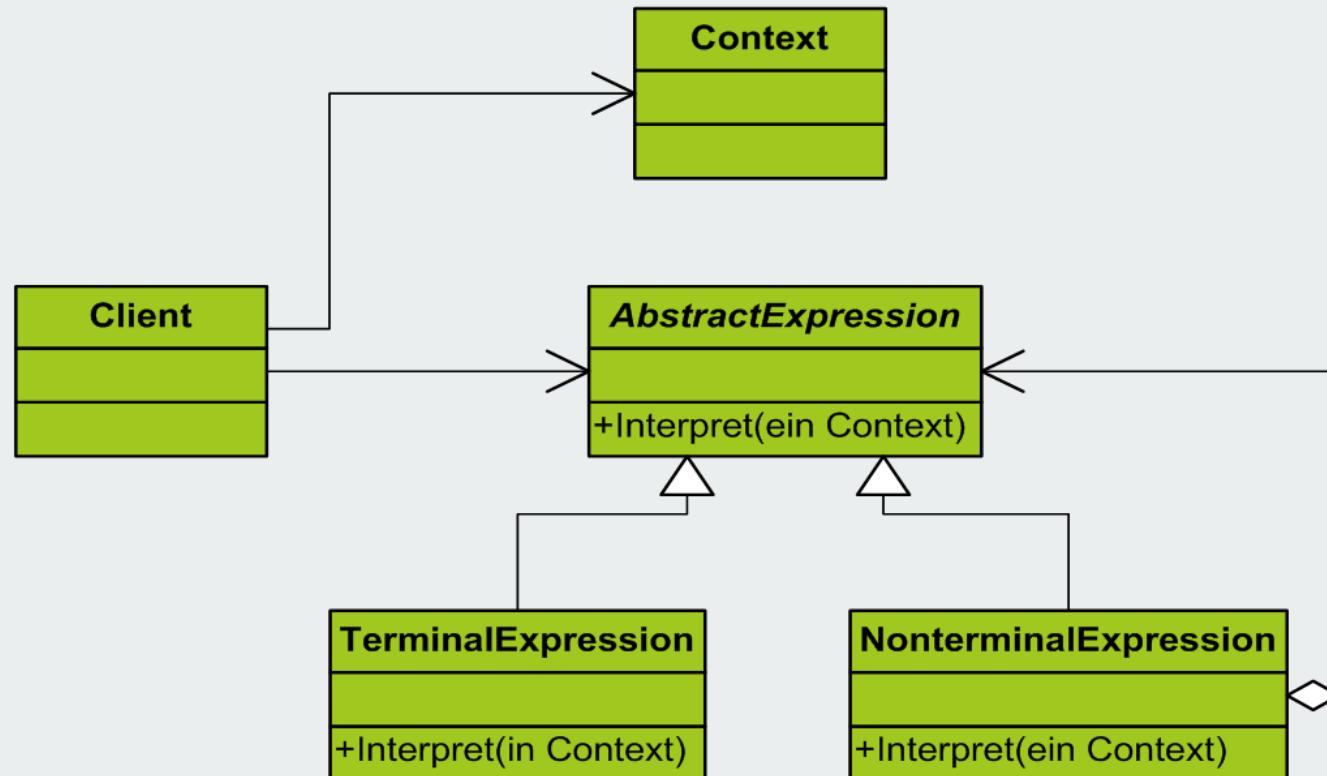
- „Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.“



Verhaltensmuster - Interpreter

334

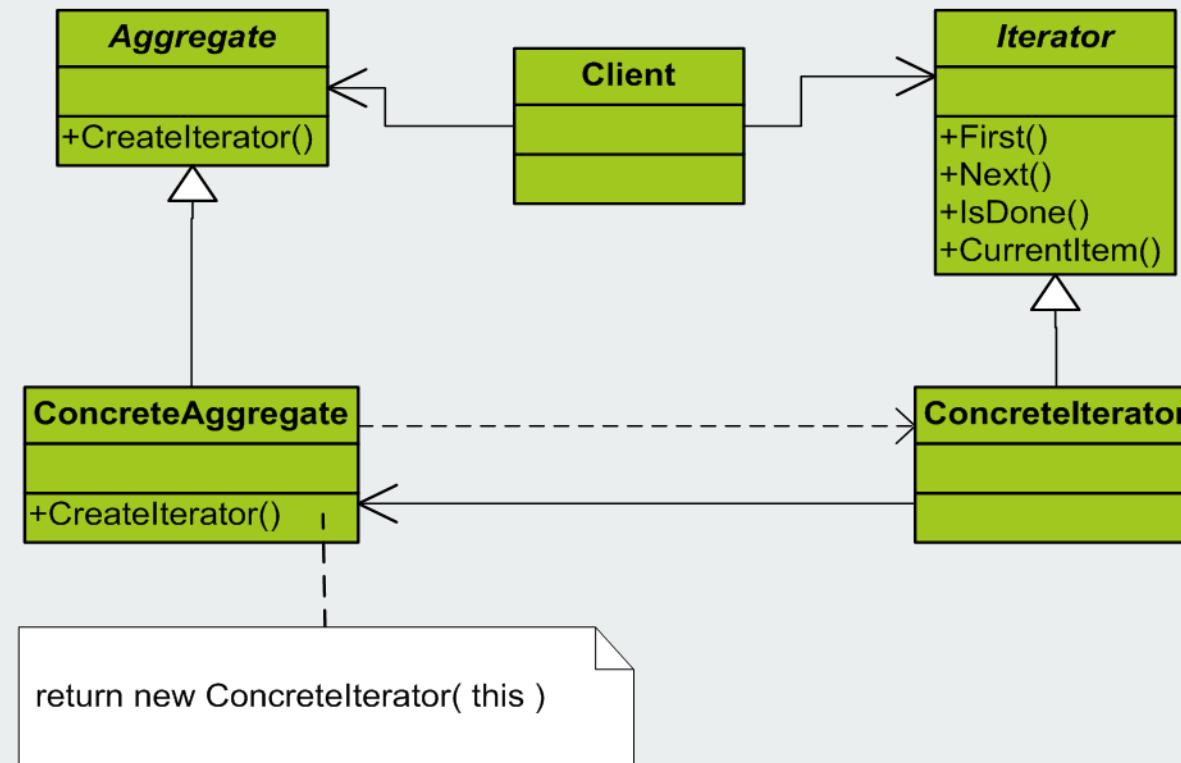
- „Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.“



Verhaltensmuster - Iterator

335

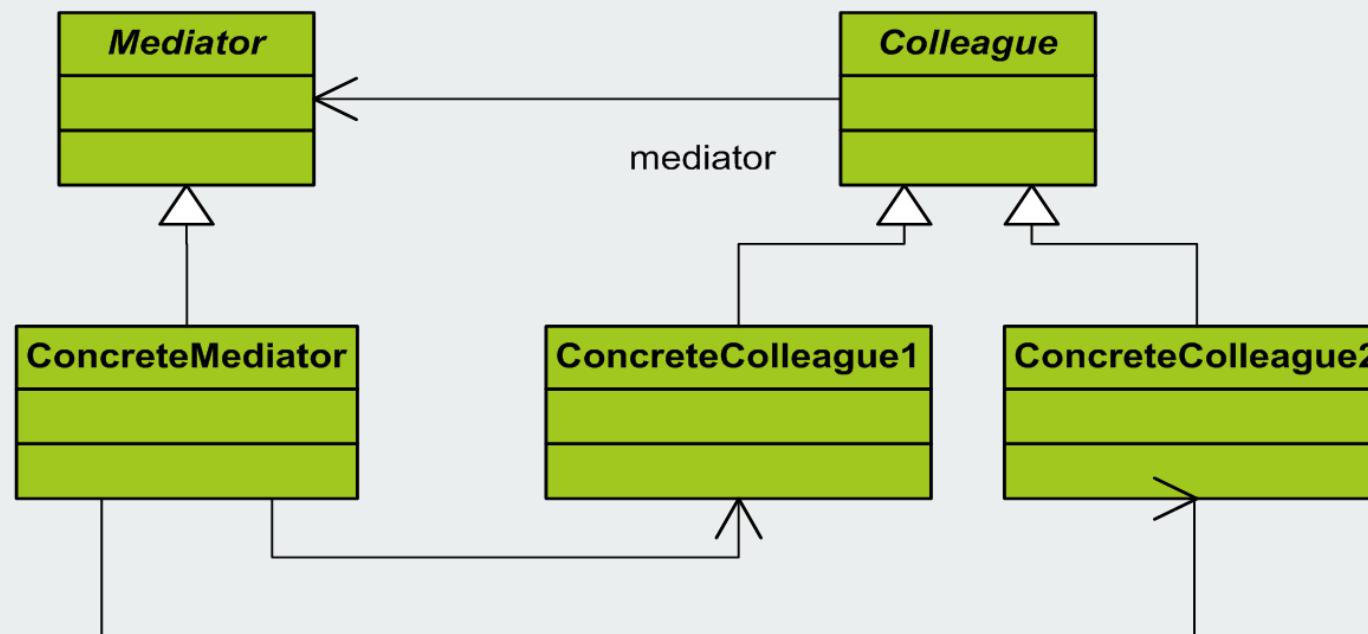
- „Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.“



Verhaltensmuster - Mediator

336

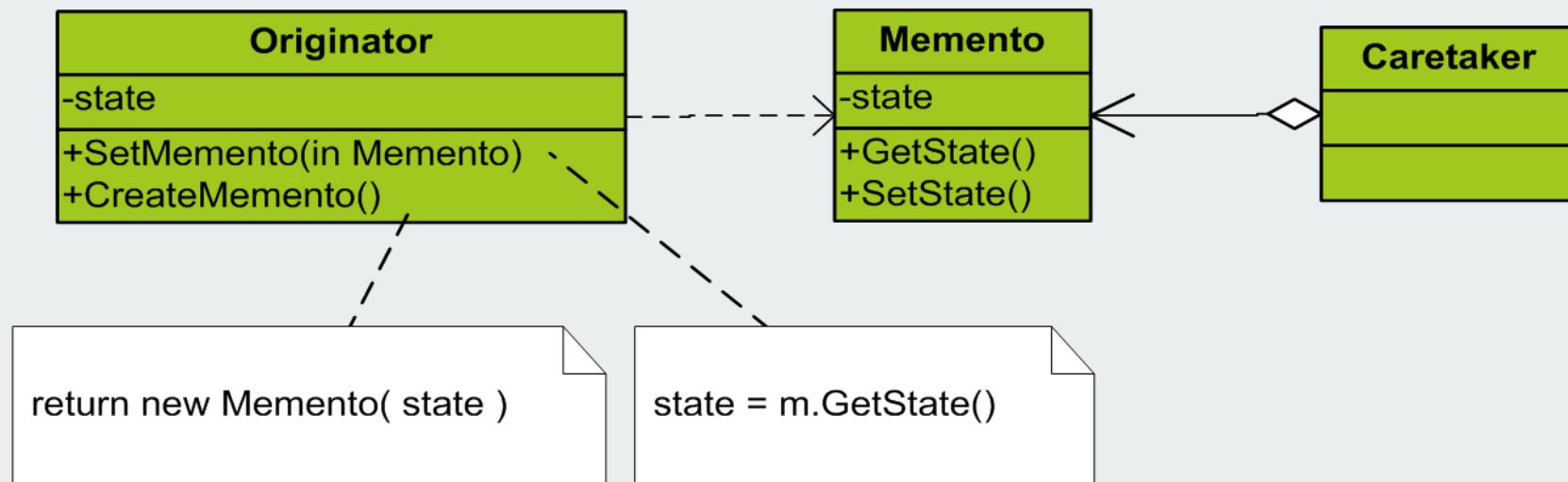
- „Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.“



Verhaltensmuster - Memento

337

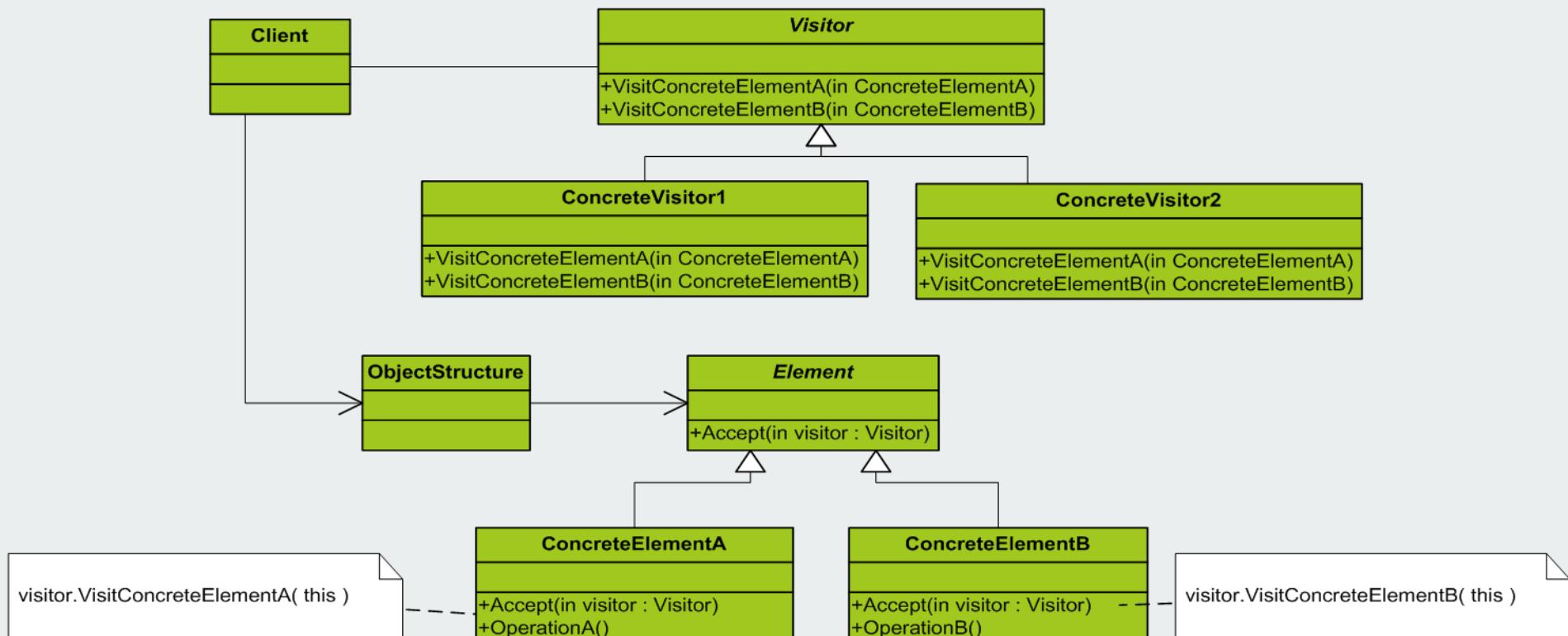
- „Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.“



Verhaltensmuster - Visitor

338

- „Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.“



Quellen

- Gamma, Erich: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994
- Steven J.Metsker: *The Design Patterns Java Workbook*, Addison-Wesley Longman, 2002
- Adam Bien: *Enterprise Java Frameworks - Java Technologien professionell einsetzen*, Addison-Wesley, 2001
- Partha Kuchana: *Software Architecture Design Patterns in Java*, Auerbach Publications, 2004
- <http://www.philippauer.de/study/se/design-pattern.php>
- <http://java.dzone.com/>
- <http://www.oodesign.com>
- <http://de.wikipedia.org/wiki/Entwurfsmuster>