

Programmierpraktikum 1

Nachklausurtutorium Sommersemester 2025

Paul Christian Dötsch

Institut für Informatik
Heinrich-Heine-Universität Düsseldorf

18. September 2025



- 1 Organisation
- 2 Tag 1 - Grundlagen & Werkzeuge
- 3 Tag 2 - Testing & Codequalität
- 4 Tag 3 - Architektur & Prinzipien
- 5 Zusammenfassung

Zeitraum: Montag 15.09.2025 bis Freitag 19.09.2025

Zeit: Täglich 08:30 - 12:30 Uhr

Ort: Hörsaal 5M

Wichtige Links

- **GitHub für dieses Tutorium:**
<https://github.com/hhu-propa1-ss25/nachklausurtutorium>
- **GitHub Organisation:** <https://github.com/hhu-propa1-ss25/Organisation>
- **RocketChat:** <https://rocketchat.hhu.de/group/Propa-SoSe-25-NKT>

Tag 1 - Grundlagen & Werkzeuge

- Java-Basics & Collections
- Generics
- Funktionale Programmierung
- Streams
- Gradle & Git

Tag 2 - Testing & Codequalität

- Testing Basics & TDD
- Code Smells im Kleinen
- Wartbarkeit

Tag 3 - Architektur & Prinzipien

- SOLID-Prinzipien
- Vererbung & Polymorphismus
- Code Smells im Großen

Tag 4 - Fortgeschrittene Themen

- Mocking & Test-Doubles
- Spring Framework
- Git im Detail

Tag 5 - Klausurvorbereitung

- Altklausuren lösen

Diese Präsentation IST:

- **Zentrale Lernhilfe** - Alles Wichtige an einem Ort
- **Nachschlagewerk** - Ctrl+F für Themen
- **Spickzettel-freundlich** - Einfach kopieren
- **Klausurvorbereitung** - Was ihr wahrscheinlich braucht
- **Meine Interpretation** der Modulinhalte
- Was ich mir als Student gewünscht hätte!

Diese Präsentation ist NICHT:

- **Offiziell** - nicht von Jens Bendisposto oder Markus Brenneis
- **Fehlerfrei** - kann kleine Fehler enthalten
- **Vollständig** - ersetzt nicht die Vorlesung
- **Einzige Quelle** - immer gegenchecken!
- **Garantie** für Klausurerfolg

Disclaimer

- Diese Präsentation ist **meine persönliche Interpretation** der Modulinhalte
- Sie ist **nicht offiziell** und kann Fehler enthalten
- **Immer gegen offizielle Quellen prüfen:**
 - Vorlesungsfolien von Prof. Brenneis
 - Online-Dokumentation (Java Docs, etc.)
 - Offizielle Tutorials und Referenzen
- Bei Unsicherheiten: Fragen stellen!
- Mein Ziel: Euch beim Bestehen helfen, nicht die Vorlesung ersetzen

Diese Folien dienen als Lernhilfe!

② Tag 1 - Grundlagen & Werkzeuge

Java-Basics & Klassenbibliothek

Generics

Funktionale Programmierung

Streams

Werkzeuge für Softwareentwicklung

Java Collections Framework - Überblick

- Zentrale Datenstrukturen für moderne Java-Entwicklung
- Drei Hauptkategorien: **List**, **Set**, **Map**, **Queue**
- Gemeinsame Basis: `Collection<E>` Interface
- Utility-Klassen: `Collections`, `Arrays`

Kernmethoden des Collection Interface

- `add(E)`, `addAll(Collection)`
- `remove(Object)`, `contains(Object)`
- `size()`, `isEmpty()`, `clear()`

Eigenschaften:

- Geordnete Sammlung mit Index-Zugriff
- Duplikate erlaubt
- Erweitert Collection um Index-basierte Operationen

Wichtige Methoden:

```
1 get(int index)
2 set(int index, E element)
3 add(int index, E element)
4 remove(int index)
5 indexOf(Object o)
6 lastIndexOf(Object o)
```

Beispiel:

```
1 List<String> names = new ArrayList<>();
2 names.add("Java");
3 names.add(1, "Python");
4 System.out.println(names.get(0)); // Java
5 System.out.println(names); // [Java,
    Python]
```

ArrayList

- Array-basierte Implementierung
- Schneller Zugriff: $O(1)$
- Langsames Einfügen in der Mitte: $O(n)$
- Automatische Größenerweiterung (um 50%)

```
1 List<Integer> numbers = new ArrayList<>();  
2 numbers.add(1);  
3 numbers.add(2);  
4 numbers.add(1, 5); // [1, 5, 2]
```

LinkedList

- Doppelt verkettete Liste
- Implementiert auch Queue und Deque
- Effizienter für häufiges Einfügen/Löschen
- Kein Index-Zugriff in $O(1)$

```
1 LinkedList<String> words = new LinkedList  
    <>();  
2 words.add("first");  
3 words.addFirst("start"); // [start, first]  
4 words.addLast("end");    // [start, first,  
    end]
```

Set Interface - Einzigartige Elemente

Eigenschaften:

- Keine Duplikate (basiert auf equals() Methode)
- Kein Index-basierter Zugriff
- Drei wichtige Implementierungen

HashSet - Häufigste Implementierung

```
1 Set<Integer> numbers = new HashSet<>();  
2 numbers.add(5);  
3 numbers.add(5); // Wird ignoriert - Set bleibt unverändert  
4 System.out.println(numbers.size()); // 1  
5 System.out.println(numbers.contains(5)); // true
```

HashSet

- Schnellste Implementierung
- Keine Reihenfolge garantiert
- Basiert auf Hash-Tabelle

```
1 Set<String> words = new HashSet<>();  
2 words.addAll(List.of("c", "a", "b"));  
3 // Reihenfolge nicht vorhersagbar
```

TreeSet

- Elemente automatisch sortiert
- Implementiert SortedSet
- Erfordert Comparable oder Comparator

```
1 Set<Integer> sorted = new TreeSet<>();  
2 sorted.addAll(List.of(7,3,9,1));  
3 System.out.println(sorted);  
4 // [1, 3, 7, 9]
```

LinkedHashSet

- Behält Einfügereihenfolge bei
- Kombination aus Hash und Linked List
- Etwas langsamer als HashSet

```
1 Set<String> ordered = new LinkedHashSet<>();  
2 ordered.addAll(List.of("c", "a", "b"));  
3 System.out.println(ordered);  
4 // [c, a, b]
```

Queue-Operationen:

- offer(E) - Element einfügen
- poll() - Element entfernen und zurückgeben (null wenn leer)
- peek() - Element anschauen ohne zu entfernen (null wenn leer)

```
1 Queue<String> q = new LinkedList<>();
2 q.offer("first");
3 q.offer("second");
4 q.offer("third");
5
6 System.out.println(q.peek()); // "first"
7 System.out.println(q.poll()); // "first"
8 System.out.println(q.poll()); // "second"
```

Deque (Double Ended Queue):

```
1 Deque<Integer> deque = new ArrayDeque<>();
2 deque.offerFirst(1); // [1]
3 deque.offerLast(2); // [1, 2]
4 deque.offerFirst(0); // [0, 1, 2]
5
6 // Stack-Operationen:
7 deque.push(5); // [5, 0, 1, 2]
8 Integer top = deque.pop(); // 5
```

Eigenschaften:

- Speichert Key-Value Paare
- Schlüssel sind eindeutig (keine Duplikate)
- Nicht Teil der Collection-Hierarchie

HashMap - Standard-Implementierung

```
1 Map<String, Integer> ages = new HashMap<>();
2 ages.put("Alice", 25);
3 ages.put("Bob", 30);
4 ages.put("Alice", 26); // Überschreibt vorherigen Wert
5
6 System.out.println(ages.get("Alice")); // 26
7 System.out.println(ages.getDefault("Charlie", 0)); // 0
8 System.out.println(ages.containsKey("Bob")); // true
```

Grundoperationen:

```
1 Map<String, Integer> map = new HashMap<>();
2
3 // Einfügen und Abrufen
4 map.put("key1", 100);
5 Integer value = map.get("key1");
6
7 // Sichere Operationen
8 map.putIfAbsent("key2", 200);
9 Integer safe = map.getOrDefault("key3", -1);
10
11 // Prüfungen
12 boolean hasKey = map.containsKey("key1");
13 boolean hasValue = map.containsValue(100);
```

Iteration über Maps:

```
1 // Über Schlüssel iterieren
2 for (String key : map.keySet()) {
3     System.out.println(key + ": " + map.get(key));
4 }
5
6 // Über Werte iterieren
7 for (Integer value : map.values()) {
8     System.out.println(value);
9 }
10
11 // Über Entry-Paare iterieren
12 for (Map.Entry<String, Integer> entry : map.
13     entrySet()) {
14     System.out.println(
15         entry.getKey() + " = " + entry.getValue()
16     );
17 }
```

Comparable Interface

- *Natürliche* Ordnung definieren
- Implementiert von der Klasse selbst
- Methode: `compareTo(T o)`

```
1 class Student implements Comparable<Student> {  
2     private String name;  
3     private int grade;  
4  
5     @Override  
6     public int compareTo(Student other) {  
7         // ACHTUNG: Overflow-Gefahr!  
8         // return this.grade - other.grade;  
9  
10        // BESSER:  
11        return Integer.compare(this.grade, other.grade);  
12    }  
13 }
```

Comparator Interface

- *Alternative* Ordnungen definieren
- Externe Klasse oder Lambda
- Methode: `compare(T o1, T o2)`

```
1 // Als separate Klasse  
2 class ReverseOrder implements Comparator<Integer> {  
3     @Override  
4     public int compare(Integer o1, Integer o2) {  
5         return o2.compareTo(o1);  
6     }  
7 }  
8  
9 // Verwendung  
10 TreeSet<Integer> reversed = new TreeSet<>(new ReverseOrder());  
11  
12 // Mit Lambda (später mehr dazu)  
13 TreeSet<Integer> desc = new TreeSet<>((a, b) -> b.compareTo(a));
```


Wichtige statische Methoden:

```
1 List<String> list = new ArrayList<>(  
2     List.of("c", "a", "b"));  
3  
4 // Sortieren  
5 Collections.sort(list);           // [a, b, c]  
6  
7 // Umdrehen  
8 Collections.reverse(list);        // [c, b, a]  
9  
10 // Mischen  
11 Collections.shuffle(list);         // zufällige Reihenfolge  
12  
13 // Rotieren  
14 Collections.rotate(list, 1);       // [?, c, b] (je nach shuffle)
```

```
1 // Füllen  
2 Collections.fill(list, "x");      // [x, x, x]  
3  
4 // Min/Max finden  
5 List<Integer> numbers = List.of(3, 1, 4, 1, 5);  
6 Integer min = Collections.min(numbers); // 1  
7 Integer max = Collections.max(numbers); // 5  
8  
9 // Häufigkeit zählen  
10 int count = Collections.frequency(numbers, 1); // 2  
11  
12 // Unveränderliche Views  
13 List<String> immutable = Collections.unmodifiableList(list);
```

Problem mit null-Werten:

- NullPointerException - häufigste Fehlerquelle
- Implizite null-Checks überall nötig
- Code wird unlesbar und fehleranfällig

Optional als Lösung:

```
1 // Traditionell - fehleranfällig
2 public String findUserName(int id) {
3     User user = database.findUser(id);
4     if (user != null) {
5         return user.getName();
6     }
7     return null; // Könnte vergessen werden!
8 }
```

```
1 // Mit Optional - explizit
2 public Optional<String> findUserName(int id) {
3     Optional<User> user = database.findUser(id);
4     return user.map(User::getName);
5 }
6
7 // Verwendung
8 String name = findUserName(123)
9     .orElse("Unknown User");
```

Modifier	Klasse	Package	Subklasse	Überall
private	✓			
<i>package-private</i>	✓	✓		
protected	✓	✓	✓	
public	✓	✓	✓	✓

Besonderheit: private zwischen Instanzen

```
1 class Outer {  
2     private int value;  
3  
4     class Inner {  
5         void method(Outer other) {  
6             other.value = 42; // OK! Private zwischen Instanzen derselben Klasse  
7         }  
8     }  
9 }
```

② Tag 1 - Grundlagen & Werkzeuge

Java-Basics & Klassenbibliothek

Generics

Funktionale Programmierung

Streams

Werkzeuge für Softwareentwicklung

Warum Generics? - Das Problem

Ohne Generics (Java < 5):

- Alles war Object
- Keine Typsicherheit zur Compile-Zeit
- Casts überall nötig
- ClassCastException zur Laufzeit möglich

Problematischer Code

```
1 ArrayList list = new ArrayList(); // Raw Type
2 list.add("String");
3 list.add(new Integer(42)); // Gemischte Typen!
4
5 String s = (String) list.get(1); // ClassCastException!
```

Mit Generics (Java 5+):

- Typsicherheit zur Compile-Zeit
- Keine Casts nötig
- Klarere APIs und bessere Dokumentation
- Bessere Performance (keine Boxing bei primitiven Collections)

Typsicherer Code

```
1 ArrayList<String> list = new ArrayList<>(); // Diamond Operator (Java 7+)
2 list.add("String");
3 list.add(42); // Kompilier-Fehler!
4
5 String s = list.get(0); // Kein Cast nötig
```

Traditionell:

```
1 if (obj instanceof String) {  
2     String str = (String) obj;  
3     System.out.println(str.toUpperCase());  
4 }  
5  
6 if (obj instanceof Integer) {  
7     Integer num = (Integer) obj;  
8     System.out.println(num * 2);  
9 }
```

Mit Pattern Matching:

```
1 if (obj instanceof String str) {  
2     System.out.println(str.toUpperCase());  
3 }  
4  
5 if (obj instanceof Integer num) {  
6     System.out.println(num * 2);  
7 }  
8  
9 // Auch in switch möglich (Java 21+)
```

Einfacher generischer Typ:

```
1 public class Box<T> {  
2     private T content;  
3  
4     public Box(T content) {  
5         this.content = content;  
6     }  
7  
8     public T getContent() {  
9         return content;  
10    }  
11  
12    public void setContent(T content) {  
13        this.content = content;  
14    }  
15 }
```

Mehrere Typparameter:

```
1 public class Pair<T, U> {  
2     private final T first;  
3     private final U second;  
4  
5     public Pair(T first, U second) {  
6         this.first = first;  
7         this.second = second;  
8     }  
9  
10    public T getFirst() { return first; }  
11    public U getSecond() { return second; }  
12 }  
13  
14 // Verwendung:  
15 Pair<String, Integer> nameAge =  
16     new Pair<>("Alice", 25);
```


Einfacher Constraint:

```
1 // Nur Number und Subtypen erlaubt
2 public class NumberBox<T extends Number> {
3     private T value;
4
5     public NumberBox(T value) {
6         this.value = value;
7     }
8
9     public double getDoubleValue() {
10         return value.doubleValue(); // Möglich wegen Number
11     }
12 }
13
14 // Verwendung:
15 NumberBox<Integer> intBox = new NumberBox<>(42);
16 NumberBox<String> strBox = new NumberBox<>("text"); // Fehler!
```

Self-referencing Constraint:

```
1 // T muss mit sich selbst vergleichbar sein
2 public class SortedList<T extends Comparable<T>> {
3     private List<T> elements = new ArrayList<>();
4
5     public void add(T element) {
6         elements.add(element);
7         Collections.sort(elements); // Funktioniert!
8     }
9
10    public T getMin() {
11        return Collections.min(elements);
12    }
13 }
14
15 // Verwendung:
16 SortedList<String> strings = new SortedList<>();
17 SortedList<Object> objects = new SortedList<>(); // Fehler!
```

```
1 public class ArrayUtils {
2
3     // Generische statische Methode
4     public static <T> List<T> arrayToList(T[] array) {
5         List<T> list = new ArrayList<>();
6         for (T element : array) {
7             list.add(element);
8         }
9         return list;
10    }
11
12    // Mit Constraint
13    public static <T extends Comparable<T>> T findMax(T[] array) {
14        if (array.length == 0) {
15            throw new IllegalArgumentException("Array empty");
16        }
17
18        T max = array[0];
19        for (T element : array) {
20            if (element.compareTo(max) > 0) {
21                max = element;
22            }
23        }
24        return max;
25    }
26 }
```

Verwendung:

```
1 // Type Inference - Compiler leitet Typen ab
2 String[] names = {"Alice", "Bob", "Charlie"};
3 List<String> nameList = ArrayUtils.arrayToList(names);
4
5 Integer[] numbers = {3, 1, 4, 1, 5};
6 List<Integer> numberList = ArrayUtils.arrayToList(numbers);
7
8 // Explizite Typangabe möglich
9 List<String> explicit = ArrayUtils.<String>arrayToList(names);
10
11 // findMax Beispiel
12 String longest = ArrayUtils.findMax(names); // "Charlie"
13 Integer biggest = ArrayUtils.findMax(numbers); // 5
```

② Tag 1 - Grundlagen & Werkzeuge

Java-Basics & Klassenbibliothek

Generics

Funktionale Programmierung

Streams

Werkzeuge für Softwareentwicklung

Funktionale Interfaces seit Java 8:

- Interface mit **genau einer** abstrakten Methode
- Verwendbar mit Lambda-Ausdrücken
- Annotation `@FunctionalInterface` zur Dokumentation
- Vordefinierte Interfaces im Package `java.util.function`

Interface	Signatur	Zweck
<code>Consumer<T></code>	$T \rightarrow \text{void}$	Seiteneffekt ausführen
<code>Supplier<T></code>	$() \rightarrow T$	Wert erzeugen
<code>Function<T,R></code>	$T \rightarrow R$	Transformation
<code>Predicate<T></code>	$T \rightarrow \text{boolean}$	Bedingung prüfen
<code>BiFunction<T,U,R></code>	$(T,U) \rightarrow R$	Zwei-Parameter Funktion

Nimmt einen Parameter, gibt nichts zurück:

```
1 // Consumer definieren
2 Consumer<String> printer = s -> System.out.println(s);
3 Consumer<String> upperPrinter = s -> System.out.println(s.
    toUpperCase());
4
5 // Verwenden
6 printer.accept("Hello World");
7 upperPrinter.accept("Hello World");
8
9 // Mit forEach
10 List<String> words = List.of("Java", "Python", "JavaScript");
11 words.forEach(printer);
12 words.forEach(upperPrinter);
```

```
1 // Consumer verketteten mit andThen
2 Consumer<String> print = s -> System.out.print(s);
3 Consumer<String> newline = s -> System.out.println();
4
5 Consumer<String> printWithNewline = print.andThen(newline);
6 printWithNewline.accept("Hello");
7
8 // Komplexeres Beispiel
9 Consumer<Person> emailSender = person ->
10     emailService.sendWelcome(person.getEmail());
11 Consumer<Person> logger = person ->
12     log.info("Processed: " + person.getName());
13
14 Consumer<Person> processNewUser = emailSender.andThen(logger);
```

Nimmt keine Parameter, gibt einen Wert zurück:

```
1 // Einfache Supplier
2 Supplier<Double> randomValue = () -> Math.random();
3 Supplier<String> greeting = () -> "Hello World";
4 Supplier<LocalDateTime> currentTime = LocalDateTime.now;
5
6 // Verwenden
7 double rand = randomValue.get();
8 String msg = greeting.get();
9 LocalDateTime now = currentTime.get();
```

```
1 // Lazy Initialization
2 public class ExpensiveObject {
3     private Supplier<String> lazyValue = () -> {
4         System.out.println("Computing expensive value...");
5         return "Expensive Result";
6     };
7
8     public String getValue() {
9         return lazyValue.get();
10    }
11 }
12
13 // Factory Pattern
14 Supplier<List<String>> listFactory = ArrayList::new;
15 List<String> newList = listFactory.get();
```

Nimmt einen Parameter, gibt transformierten Wert zurück:

```
1 // Einfache Funktionen
2 Function<String, Integer> length = s -> s.length();
3 Function<String, String> toUpper = s -> s.toUpperCase();
4 Function<Integer, Integer> square = x -> x * x;
5
6 // Verwenden
7 Integer len = length.apply("Hello"); // 5
8 String upper = toUpper.apply("hello"); // "HELLO"
9 Integer squared = square.apply(4); // 16
```

```
1 // Funktionen verketteten
2 Function<String, String> normalize = s -> s.trim().toLowerCase();
3 Function<String, Integer> wordCount = s -> s.split("\\s+").length;
4
5 Function<String, Integer> countNormalized =
6     normalize.andThen(wordCount);
7
8 Integer count = countNormalized.apply(" Hello World "); // 2
9
10 // Compose (umgekehrte Richtung)
11 Function<String, Integer> lengthOfUpper =
12     length.compose(toUpper);
13 // Entspricht: s -> length.apply(toUpper.apply(s))
```

Nimmt einen Parameter, gibt boolean zurück:

```
1 // Einfache Prädikate
2 Predicate<String> isEmpty = s -> s.isEmpty();
3 Predicate<String> isLong = s -> s.length() > 10;
4 Predicate<Integer> isEven = n -> n % 2 == 0;
5 Predicate<Integer> isPositive = n -> n > 0;
6
7 // Verwenden
8 boolean empty = isEmpty.test(""); // true
9 boolean long = isLong.test("Short"); // false
10 boolean even = isEven.test(42); // true
```

```
1 // Prädikate kombinieren
2 Predicate<Integer> isPositiveEven = isPositive.and(isEven);
3 Predicate<Integer> isNegativeOrOdd = isPositive.negate().or(isEven.
    negate());
4
5 List<Integer> numbers = List.of(-2, -1, 0, 1, 2, 3, 4);
6
7 // Mit filter verwenden
8 List<Integer> positiveEvens = numbers.stream()
9     .filter(isPositiveEven)
10    .toList(); // [2, 4]
11
12 // Statische Utility-Methoden
13 Predicate<Object> isNull = Objects::isNull;
14 Predicate<String> isEqual = Predicate.isEqual("test");
```


Verschiedene Syntaxformen:

Parameter-Syntax:

```
1 // Mit Typdeklaration
2 Consumer<String> c1 = (String s) -> System.out.println(s);
3
4 // Ohne Typdeklaration (Type Inference)
5 Consumer<String> c2 = (s) -> System.out.println(s);
6
7 // Ohne Klammern (bei einem Parameter)
8 Consumer<String> c3 = s -> System.out.println(s);
9
10 // Mehrere Parameter
11 BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
12
13 // Keine Parameter
14 Supplier<String> greeting = () -> "Hello";
```

Body-Syntax:

```
1 // Expression (kein return nötig)
2 Function<Integer, Integer> square = x -> x * x;
3
4 // Statement Block (return erforderlich)
5 Function<String, String> process = s -> {
6     String trimmed = s.trim();
7     String upper = trimmed.toUpperCase();
8     return upper;
9 };
10
11 // Void Block
12 Consumer<String> debug = s -> {
13     System.out.println("Processing: " + s);
14     // Kein return
15 };
```

Kompakte Alternative zu Lambda-Ausdrücken:

Statische Methoden:

```
1 // Lambda
2 Function<String, Integer> parseInt1 = s -> Integer.parseInt(s);
3
4 // Methodenreferenz
5 Function<String, Integer> parseInt2 = Integer::parseInt;
6
7 // Verwendung mit Collections
8 List<String> numbers = List.of("1", "2", "3");
9 List<Integer> parsed = numbers.stream()
10     .map(Integer::parseInt)
11     .toList();
```

Instanzmethoden:

```
1 // Auf bestimmter Instanz
2 List<String> words = List.of("hello", "world");
3 words.forEach(System.out::println);
4
5 // Auf Parameter (unbound reference)
6 Function<String, String> toUpper1 = s -> s.toUpperCase();
7 Function<String, String> toUpper2 = String::toUpperCase;
8
9 // Konstruktor-Referenzen
10 Supplier<ArrayList<String>> listSupplier = ArrayList::new;
11 Function<String, StringBuilder> sbFactory = StringBuilder::new;
```

Methoden, die Funktionen als Parameter nehmen oder zurückgeben: forEach implementieren:

```
1 public class MyList<T> {  
2     private List<T> elements = new ArrayList<>();  
3  
4     public void forEach(Consumer<T> action) {  
5         for (T element : elements) {  
6             action.accept(element);  
7         }  
8     }  
9  
10    public void addIf(T element, Predicate<T> condition) {  
11        if (condition.test(element)) {  
12            elements.add(element);  
13        }  
14    }  
15 }
```

Funktionen zurückgeben:

```
1 public class FunctionFactory {  
2  
3     public static Predicate<Integer> greaterThan(int threshold) {  
4         return value -> value > threshold;  
5     }  
6  
7     public static Function<String, String> addPrefix(String prefix)  
8     {  
9         return text -> prefix + text;  
10    }  
11  
12    // Verwendung  
13    Predicate<Integer> greaterThan10 = FunctionFactory.greaterThan(10);  
14    Function<String, String> addHello = FunctionFactory.addPrefix("  
15        Hello, ");  
16  
17    boolean result = greaterThan10.test(15); // true  
18    String greeting = addHello.apply("World"); // "Hello, World"
```

② Tag 1 - Grundlagen & Werkzeuge

Java-Basics & Klassenbibliothek

Generics

Funktionale Programmierung

Streams

Werkzeuge für Softwareentwicklung

Problem mit traditioneller Programmierung:

- Viel Boilerplate-Code für einfache Operationen
- Schleifen vermischen "Was" und "Wie"
- Schwer parallelisierbar
- Wenig deklarativ

Beispiel: Erste 3 gerade Zahlen > 5 , quadriert

Traditional: 15+ Zeilen mit for-Schleifen, if-Statements, temporären Variablen

Mit Streams:

```
1 numbers.stream()  
2     .filter(x -> x > 5)  
3     .filter(x -> x % 2 == 0)  
4     .limit(3)  
5     .map(x -> x * x)  
6     .toList();
```

Stream erzeugen:

```
1 Stream.of("a", "b", "c");
2 Arrays.stream(array);
3 list.stream();
4 IntStream.range(1, 10);
5 Stream.generate(Math::random);
6 Stream.iterate(0, n -> n + 1);
7 "Hello,World,Java".chars().mapToObj(c -> (char) c);
8 map.entrySet().stream();
```

Filtern und Transformieren:

```
1 stream.filter(x -> x > 5);
2 stream.map(String::toUpperCase);
3 stream.mapToInt(String::length);
4 stream.flatMap(s -> Arrays.stream(s.split(",")));
5 stream.distinct();
6 stream.sorted();
7 stream.sorted(Comparator.comparing(Person::getName));
8 stream.limit(10);
9 stream.skip(5);
10 stream.peek(System.out::println);
11 stream.takeWhile(x -> x < 100);
12 stream.dropWhile(x -> x < 10);
13 stream.mapToObj(i -> "Item " + i);
14 stream.mapToDouble(account -> account.getBalance());
```

Aggregation:

```
1 stream.count();
2 stream.min(Comparator.naturalOrder());
3 stream.max(String::compareTo);
4 stream.reduce(0, Integer::sum);
5 stream.reduce((a, b) -> a + b);
```

Sammeln:

```
1 stream.toList();
2 stream.collect(Collectors.toSet());
3 stream.collect(Collectors.joining(", "));
4 stream.collect(Collectors.groupingBy(String::length));
5 stream.collect(Collectors.partitioningBy(x -> x > 0));
6 stream.collect(Collectors.counting());
7 stream.collect(Collectors.summarizingInt(String::length));
```

Prüfen und Finden:

```
1 stream.anyMatch(x -> x > 10);
2 stream.allMatch(x -> x >= 0);
3 stream.noneMatch(String::isEmpty);
4 stream.findFirst();
5 stream.findAny();
```

Aus Collections:

```
1 List<String> words = List.of("Java", "Python", "C++");
2 Stream<String> wordStream = words.stream();
3
4 Set<Integer> numbers = Set.of(1, 2, 3, 4, 5);
5 Stream<Integer> numberStream = numbers.stream();
6
7 // Parallele Streams
8 Stream<String> parallelStream = words.parallelStream();
```

Aus Arrays:

```
1 String[] array = {"a", "b", "c"};
2 Stream<String> arrayStream = Arrays.stream(array);
3
4 int[] intArray = {1, 2, 3, 4, 5};
5 IntStream intStream = Arrays.stream(intArray);
```

Statische Factory-Methoden:

```
1 // Endliche Streams
2 Stream<String> of = Stream.of("a", "b", "c");
3 Stream<Integer> empty = Stream.empty();
4
5 // Unendliche Streams
6 Stream<Double> randoms = Stream.generate(Math::random);
7 Stream<Integer> naturals = Stream.iterate(0, n -> n + 1);
8
9 // Mit Bedingung (Java 9+)
10 Stream<Integer> limited = Stream.iterate(0, n -> n < 100, n -> n + 2);
```

Primitive Streams:

```
1 IntStream range = IntStream.range(0, 10); // 0-9
2 IntStream rangeClosed = IntStream.rangeClosed(1, 10); // 1-10
3 DoubleStream doubles = DoubleStream.of(1.1, 2.2, 3.3);
```

Geben einen neuen Stream zurück - lazy evaluation:

Filterung und Transformation:

```
1 List<String> words = List.of("Java", "Python", "JavaScript", "C++")  
  ;  
2  
3 Stream<String> filtered = words.stream()  
  .filter(s -> s.length() > 4)  
  .map(String::toUpperCase)  
  .distinct();  
4  
5 // Noch nicht ausgeführt!
```

FlatMap - Streams "flach klopfen":

```
1 List<List<String>> nested = List.of(  
2   List.of("a", "b"),  
3   List.of("c", "d"),  
4   List.of("e")  
5 );  
6  
7 List<String> flattened = nested.stream()  
  .flatMap(Collection::stream)  
  .toList(); // [a, b, c, d, e]
```

Limiting und Skipping:

```
1 List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
2  
3 List<Integer> middle = numbers.stream()  
4   .skip(3)           // 4, 5, 6, 7, 8, 9, 10  
5   .limit(4)          // 4, 5, 6, 7  
6   .toList();
```

Sortierung:

```
1 List<String> sorted = words.stream()  
2   .sorted()           // Natural order  
3   .toList();  
4  
5 List<String> byLength = words.stream()  
6   .sorted(Comparator.comparing(String::length))  
7   .toList();  
8  
9 List<String> reversed = words.stream()  
10  .sorted(Comparator.reverseOrder())  
11  .toList();
```


Beenden die Stream-Pipeline und lösen Berechnung aus:

Sammeln:

```
1 List<String> words = List.of("Java", "Python", "C++");
2
3 List<String> list = words.stream()
4     .map(String::toUpperCase)
5     .toList();
6
7 Set<String> set = words.stream()
8     .collect(Collectors.toSet());
9
10 String joined = words.stream()
11     .collect(Collectors.joining(", "));
12 // "Java, Python, C++"
```

Aggregation:

```
1 List<Integer> numbers = List.of(1, 2, 3, 4, 5);
2
3 long count = numbers.stream().count();
4 Optional<Integer> max = numbers.stream().max(Integer::compareTo);
5 Optional<Integer> min = numbers.stream().min(Integer::compareTo);
```

Matching:

```
1 boolean anyEven = numbers.stream()
2     .anyMatch(n -> n % 2 == 0);    // true
3
4 boolean allPositive = numbers.stream()
5     .allMatch(n -> n > 0);         // true
6
7 boolean noneNegative = numbers.stream()
8     .noneMatch(n -> n < 0);        // true
```

Finden:

```
1 Optional<String> first = words.stream()
2     .filter(s -> s.startsWith("J"))
3     .findFirst();                // Optional["Java"]
4
5 Optional<String> any = words.stream()
6     .filter(s -> s.startsWith("P"))
7     .findAny();                  // Optional["Python"]
```

Reduziert Stream auf einen einzigen Wert:

Mit Initialwert:

```
1 List<Integer> numbers = List.of(1, 2, 3, 4, 5);
2
3 // Summe
4 Integer sum = numbers.stream()
5     .reduce(0, (a, b) -> a + b);           // 15
6
7 // Oder kürzer mit Integer::sum
8 Integer sum2 = numbers.stream()
9     .reduce(0, Integer::sum);             // 15
10
11 // Produkt
12 Integer product = numbers.stream()
13     .reduce(1, (a, b) -> a * b);          // 120
14
15 // String konkatenation
16 String concatenated = List.of("A", "B", "C").stream()
17     .reduce("", (a, b) -> a + b);         // "ABC"
```

Ohne Initialwert (Optional):

```
1 Optional<Integer> sum = numbers.stream()
2     .reduce((a, b) -> a + b);
3
4 Optional<Integer> max = numbers.stream()
5     .reduce(Integer::max);
6
7 // Komplexeres Beispiel
8 Optional<String> longest = List.of("Java", "Python", "C++").stream()
9     .reduce((s1, s2) -> s1.length() >= s2.length() ? s1 : s2);
10 // Optional["Python"]
```

Primitive Streams:

```
1 IntStream.range(1, 6)
2     .sum();                               // 15
3
4 OptionalDouble average = IntStream.range(1, 6)
5     .average();                           // OptionalDouble[3.0]
```

Basis-Collectors:

```
1 List<String> words = List.of("Java", "Python", "Java", "C++");
2
3 List<String> toList = words.stream()
4     .collect(Collectors.toList());
5
6 Set<String> toSet = words.stream()
7     .collect(Collectors.toSet());
8
9 Map<String, Integer> toMap = words.stream()
10    .distinct()
11    .collect(Collectors.toMap(
12        Function.identity(), // Key: das Wort selbst
13        String::length       // Value: Länge des Worts
14    ));
```

Statistiken:

```
1 long count = words.stream()
2     .collect(Collectors.counting());
3
4 IntSummaryStatistics stats = words.stream()
5     .collect(Collectors.summarizingInt(String::length));
```

Gruppierung:

```
1 // Gruppierung nach Länge
2 Map<Integer, List<String>> byLength = words.stream()
3     .collect(Collectors.groupingBy(String::length));
4 // {4=[Java, Java], 6=[Python], 3=[C++]}
```



```
5 // Mit Downstream-Collector
6 Map<Integer, Long> countByLength = words.stream()
7     .collect(Collectors.groupingBy(
8         String::length,
9         Collectors.counting()
10    ));
11 // {4=2, 6=1, 3=1}
```



```
12 // Partitionierung (boolean-Kriterium)
13 Map<Boolean, List<String>> partition = words.stream()
14     .collect(Collectors.partitioningBy(
15         s -> s.length() > 4
16    ));
17 // {false=[Java, Java, C++], true=[Python]}
```

Spezialisierte Streams für bessere Performance:

IntStream, LongStream, DoubleStream:

```
1 // Vermeidet Boxing/Unboxing
2 IntStream numbers = IntStream.range(1, 1000000);
3
4 int sum = numbers.sum(); // Optimiert
5 OptionalDouble avg = numbers.average();
6 IntSummaryStatistics stats = numbers.summaryStatistics();
7
8 // Von Stream<Integer> zu IntStream
9 List<Integer> integers = List.of(1, 2, 3, 4, 5);
10 int sumFromStream = integers.stream()
11     .mapToInt(Integer::intValue)
12     .sum();
13
14 // Von IntStream zu Stream<Integer>
15 Stream<Integer> boxed = IntStream.range(1, 10)
16     .boxed();
```

Praktisches Beispiel:

```
1 // Summe aller geraden Quadrate von 1-100
2 int result = IntStream.rangeClosed(1, 100)
3     .filter(n -> n % 2 == 0)
4     .map(n -> n * n)
5     .sum();
6
7 // Mit Double-Werten
8 double[] values = {1.1, 2.2, 3.3, 4.4, 5.5};
9 OptionalDouble max = Arrays.stream(values)
10     .max();
11
12 // String-Längen als IntStream
13 String[] words = {"Java", "Python", "C++"};
14 IntSummaryStatistics lengthStats = Arrays.stream(words)
15     .mapToInt(String::length)
16     .summaryStatistics();
```

Automatische Parallelisierung für bessere Performance:

Erzeugung:

```
1 // Aus Collections
2 List<Integer> numbers = List.of(1, 2, 3, 4, 5);
3 Stream<Integer> parallel = numbers.parallelStream();
4
5 // Aus sequenziellen Streams
6 Stream<Integer> sequential = numbers.stream();
7 Stream<Integer> parallelized = sequential.parallel();
8
9 // Zurück zu sequenziell
10 Stream<Integer> backToSeq = parallel.sequential();
```

Wann sinnvoll:

- Große Datenmengen
- CPU-intensive Operationen
- Unabhängige Operationen
- Mehrere CPU-Kerne verfügbar

Wichtige Einschränkungen:

```
1 // GEFÄHRlich - Race Condition!
2 List<String> result = new ArrayList<>();
3 words.parallelStream()
4     .forEach(word -> result.add(word.toUpperCase())); // FEHLER!
5
6 // SICHER - Verwendung von Collectors
7 List<String> safe = words.parallelStream()
8     .map(String::toUpperCase)
9     .collect(Collectors.toList()); // OK!
10
11 // GEFÄHRlich - Shared Mutable State
12 AtomicInteger counter = new AtomicInteger(0);
13 numbers.parallelStream()
14     .forEach(n -> counter.incrementAndGet()); // Problematisch
```

Regeln:

- Keine Seiteneffekte
- Thread-safe Operationen
- Assoziative Funktionen bei reduce

② Tag 1 - Grundlagen & Werkzeuge

Java-Basics & Klassenbibliothek

Generics

Funktionale Programmierung

Streams

Werkzeuge für Softwareentwicklung

Warum Build Tools?

- Automatisierung von Kompilierung, Tests, Packaging
- Abhängigkeitsverwaltung (Dependencies)
- Standardisierte Projektstruktur
- Integration mit IDEs und CI/CD

Gradle vs. Alternatives:

- **Maven**: XML-basiert, sehr verbreitet
- **Gradle**: Groovy/Kotlin DSL, flexibler, schneller
- **SBT**: Hauptsächlich für Scala

Gradle Vorteile

- Inkrementelle Builds (nur geänderte Teile)
- Build Cache für bessere Performance
- Flexible Projektstrukturen möglich

Standard-Layout (Convention over Configuration):

```
1 myproject/  
2 |-- build.gradle  
3 |-- settings.gradle  
4 |-- gradle/  
5 |   +-- wrapper/  
6 |-- gradlew      (Unix)  
7 |-- gradlew.bat  (Windows)  
8 +-- src/  
9     |-- main/  
10      |-- java/  
11      |   +-- com/example/  
12      |   +-- Main.java  
13      +-- resources/  
14 +-- test/  
15     +-- java/  
16         +-- com/example/  
17         +-- MainTest.java
```

Wichtige Dateien:

- build.gradle - Build-Konfiguration
- settings.gradle - Projekt-Einstellungen
- gradlew - Gradle Wrapper (empfohlen)

Verzeichnisse:

- src/main/java - Produktions-Code
- src/main/resources - Ressourcen (Konfiguration, etc.)
- src/test/java - Test-Code
- build/ - Generierte Dateien (nicht versioniert)


```
1 plugins {  
2     id 'java'                // Java-Plugin für Kompilierung  
3     id 'application'         // Anwendungs-Plugin für run-Task  
4 }  
5  
6 repositories {  
7     mavenCentral()           // Standard Java-Repository  
8 }  
9  
10 dependencies {  
11     // Produktions-Dependencies  
12     implementation 'com.google.guava:guava:33.1.0-jre'  
13     implementation 'org.apache.commons:commons-collections4:4.4'  
14  
15     // Test-Dependencies  
16     testImplementation 'org.junit.jupiter:junit-jupiter-api:5.12.2'  
17     testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.12.2'  
18     testImplementation 'org.assertj:assertj-core:3.27.3'  
19 }  
20  
21 application {  
22     mainClass = 'com.example.Main' // Hauptklasse für run-Task  
23 }  
24  
25 test {  
26     useJUnitPlatform()        // JUnit 5 verwenden  
27 }
```

Build-Tasks:

1	<code>gradle compile</code>	# Nur kompilieren
2	<code>gradle assemble</code>	# JAR erstellen (ohne Tests)
3	<code>gradle build</code>	# Vollständiger Build mit Tests
4	<code>gradle clean</code>	# Build-Verzeichnis löschen
5	<code>gradle jar</code>	# JAR-Datei erstellen

Run-Tasks:

1	<code>gradle run</code>	# Anwendung ausführen
2	<code>gradle test</code>	# Tests ausführen
3	<code>gradle distZip</code>	# ZIP-Distribution erstellen
4	<code>gradle installDist</code>	# Lokale Installation

Task-Abkürzungen:

1	<code>gradle b</code>	# build
2	<code>gradle cJ</code>	# compileJava
3	<code>gradle dZ</code>	# distZip
4	<code>gradle iD</code>	# installDist

Info-Tasks:

1	<code>gradle tasks</code>	# Alle verfügbaren Tasks
2	<code>gradle dependencies</code>	# Dependency-Tree anzeigen
3	<code>gradle projects</code>	# Multi-Project Info
4	<code>gradle properties</code>	# Projekt-Properties

Tipp: Gradle Wrapper verwenden

`./gradlew build` statt `gradle build` - garantiert richtige Gradle-Version!

Java Package System:

- Organisiert Klassen in Namensräumen
- Vermeidet Namenskonflikte
- Ermöglicht Access Control (package-private)
- Spiegelt Verzeichnisstruktur wider

Classpath:

- Suchpfad für .class-Dateien und JAR-Archive
- Trenner: : (Unix/Linux) oder ; (Windows)
- Aktuelles Verzeichnis: .

Package-Regeln

- Package-Deklaration ist erste Zeile (außer Kommentaren)
- Dateipfad muss Package entsprechen
- Naming Convention: Reverse Domain (com.example.myapp)
- Keine Leerzeichen oder Sonderzeichen

Java Archive (JAR) - ZIP-Archive für Java:

JAR erstellen:

```
1 # Einfache JAR
2 jar cf myapp.jar *.class
3
4 # Mit Manifest
5 jar cfm myapp.jar MANIFEST.MF *.class
6
7 # Mit Gradle
8 gradle jar # Erstellt JAR in build/libs/
```

JAR ausführen:

```
1 # Mit Klassenname
2 java -cp myapp.jar com.example.Main
3
4 # Ausführbare JAR (mit Main-Class im Manifest)
5 java -jar myapp.jar
```

MANIFEST.MF Beispiel:

```
1 Manifest-Version: 1.0
2 Main-Class: com.example.Main
3 Class-Path: lib/dependency1.jar lib/dependency2.jar
```

JAR inspizieren:

```
1 jar tf myapp.jar      # Inhalt auflisten
2 jar xf myapp.jar      # Extrahieren
3 unzip -l myapp.jar     # Alternative mit unzip
```

Warum Versionskontrolle?

- Änderungen nachvollziehen (Wer? Wann? Was? Warum?)
- Zusammenarbeit im Team
- Backup und Wiederherstellung
- Experimente und Branches
- Release-Management

Git Konzepte:

- **Working Directory:** Aktuelle Dateien
- **Staging Area:** Vorbereitete Änderungen
- **Repository:** Vollständige Historie
- **Commit:** Snapshot zu einem Zeitpunkt
- **Branch:** Entwicklungszweig

Projekt starten:

```
1 # Neues Repository
2 git init
3
4 # Existierendes Repository klonen
5 git clone https://github.com/user/repo.git
6
7 # Status anzeigen
8 git status
```

Änderungen committen:

```
1 # Dateien zur Staging Area hinzufügen
2 git add file1.java file2.java
3 git add .           # Alle Änderungen
4
5 # Commit erstellen
6 git commit -m "Add new feature"
7
8 # Add und Commit kombiniert (nur für tracked files)
9 git commit -am "Fix bug in calculation"
```

Mit Remote arbeiten:

```
1 # Änderungen hochladen
2 git push
3
4 # Änderungen herunterladen
5 git pull
6
7 # Remote Repository hinzufügen
8 git remote add origin https://github.com/user/repo.git
9 git push -u origin main
```

Historie anzeigen:

```
1 git log           # Ausführliches Log
2 git log --oneline  # Kompakte Ansicht
3 git log --graph    # Mit Branching-Visualisierung
4 git diff           # Änderungen anzeigen
5 git blame file.java # Zeile für Zeile Autoreninfo
```

Anatomie einer guten Commit-Message

- 1 Kurze Zusammenfassung (< 50 Zeichen)
- 2
- 3 Detaillierte Beschreibung, wenn nötig.
- 4 Erkläre WARUM, nicht WAS.
- 5 Verwende Imperativ: "Add" statt "Added".
- 6
- 7 - Kann Aufzählungen enthalten
- 8 - Oder Links zu Issues: Fixes #123

Gute Beispiele:

- Add user authentication
- Fix memory leak in image processing
- Update Spring Boot to version 3.2

Schlechte Beispiele:

- fix
- changed some files
- WIP (außer für temporäre Commits)

```
1 # Build-Ordner
2 build/
3 target/
4 out/
5
6 # IDE-Dateien
7 .idea/
8 .vscode/
9
10 # OS-spezifische Dateien
11 .DS_Store
12 Thumbs.db
13 *~
14
15 # Sensitive Daten
16 .env
17 config/local.properties
```

Patterns:

- *.class - Alle .class Dateien
- build/ - Komplettes Verzeichnis
- !important.class - Ausnahme definieren
- docs/*.pdf - PDFs nur in docs/

③ Tag 2 - Testing & Codequalität

Testing Basics

TDD-Zyklus

Qualität von Software

Code Smells im Kleinen

Warum Unit Testing?

Software ohne Tests:

- Angst vor Änderungen - "never touch a running system"
- Manuelle Tests sind langsam und fehleranfällig
- Bugs werden erst in Produktion entdeckt
- Regression bei jeder Änderung möglich

Vorteile von automatisierten Tests:

- **Sicherheitsnetz** bei Refactoring
- **Dokumentation** des gewünschten Verhaltens
- **Frühzeitige** Fehlererkennung
- **Vertrauen** in Code-Änderungen
- **Besseres Design** durch Testbarkeit

Arrange-Act-Assert - Standardstruktur für Tests:

① Arrange: Test-Setup

- Objekte erzeugen
- Zustand vorbereiten
- Test-Daten erstellen

② Act: Code ausführen

- Die zu testende Methode aufrufen
- Meist nur eine Zeile

③ Assert: Ergebnis prüfen

- Erwartete mit tatsächlichen Werten vergleichen
- Mehrere Assertions möglich

Beispiel

```
1  @Test
2  @DisplayName("Addition zweier positiver Zahlen")
3  void should_add_two_positive_numbers() {
4      // Arrange
5      Calculator calc = new Calculator();
6      int a = 5, b = 3;
7
8      // Act
9      int result = calc.add(a, b);
10
11     // Assert
12     assertThat(result).isEqualTo(8);
13 }
```

Eigenschaften guter Unit-Tests:

Fast Tests sollen schnell laufen

- Keine Datenbankzugriffe, Netzwerk-Calls
- Hunderte Tests in Sekunden

Independent Tests sollen unabhängig voneinander sein

- Reihenfolge egal
- Kein geteilter Zustand zwischen Tests

Repeatable Tests sollen wiederholbar sein

- Gleiche Eingabe → gleiches Ergebnis
- Keine Abhängigkeit von aktueller Zeit, Zufallswerten

Self-evaluating Tests zeigen selbst an, ob sie bestanden haben

- Rot/Grün statt manuelle Interpretation

Timely Tests werden zeitnah geschrieben

- Idealerweise vor dem Code (TDD)

Dependencies in build.gradle:

```
1 dependencies {
2     testImplementation 'org.junit.jupiter:junit-jupiter-api:5.12.2'
3     testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.12.2'
4     testImplementation 'org.assertj:assertj-core:3.27.3'
5 }
6
7 test {
8     useJUnitPlatform()
9     testLogging {
10         events "passed", "skipped", "failed"
11     }
12 }
```

Imports in Test-Klassen:

```
1 import org.junit.jupiter.api.Test;
2 import org.junit.jupiter.api.DisplayName;
3 import static org.assertj.core.api.Assertions.*;
```

Pure Function: Ergebnis nur von Eingabeparametern abhängig

Eigenschaften:

- Keine Seiteneffekte
- Deterministisch
- Einfach zu testen
- Wiederverwendbar

```
1 // Pure Function
2 public static double calculateCircleArea(double radius) {
3     return Math.PI * radius * radius;
4 }
5
6 // Nicht pure (Seiteneffekt)
7 public void logAndCalculate(double radius) {
8     System.out.println("Calculating..."); // Seiteneffekt!
9     return Math.PI * radius * radius;
10 }
```

Test für Pure Function:

```
1 @Test
2 @DisplayName("Kreisfläche für Radius 5")
3 void should_calculate_circle_area_for_radius_5() {
4     // Act
5     double area = MathUtils.calculateCircleArea(5.0);
6
7     // Assert
8     assertThat(area).isCloseTo(78.54, offset(0.01));
9 }
10
11 @Test
12 @DisplayName("Kreisfläche für Radius 0")
13 void should_return_zero_area_for_zero_radius() {
14     // Act
15     double area = MathUtils.calculateCircleArea(0.0);
16
17     // Assert
18     assertThat(area).isEqualTo(0.0);
19 }
```

Counter-Klasse:

```
1 public class Counter {  
2     private int count = 0;  
3  
4     public void increment() {  
5         count++;  
6     }  
7  
8     public int getCount() {  
9         return count;  
10    }  
11  
12    public void reset() {  
13        count = 0;  
14    }  
15 }
```

Test-Klasse:

```
1 public class CounterTest {  
2     private Counter counter = new Counter();  
3     @Test  
4     @DisplayName("Counter startet bei 0")  
5     void should_start_at_zero() {  
6         assertThat(counter.getCount()).isEqualTo(0);  
7     }  
8     @Test  
9     @DisplayName("Reset setzt auf 0 zurück")  
10    void should_reset_to_zero() {  
11        // Arrange  
12        counter.increment();  
13        // Act  
14        counter.reset();  
15        // Assert  
16        assertThat(counter.getCount()).isEqualTo(0);  
17    }  
18 }
```

JUnit erzeugt für jeden Test eine neue Instanz!

Tests sind automatisch isoliert - kein geteilter Zustand.

Lesbare Assertions im natürlichen Sprachfluss:

Zahlen:

```
1 assertThat(42).isEqualTo(42);
2 assertThat(3.14159).isCloseTo(3.14, offset(0.01));
3 assertThat(100.0).isCloseTo(99.0, withPercentage(2));
4 assertThat(5).isGreaterThan(3);
5 assertThat(10).isBetween(5, 15);
6 assertThat(-5).isNegative();
7 assertThat(0).isZero();
```

Strings:

```
1 assertThat("Hello World").isEqualTo("Hello World");
2 assertThat("Java").contains("av");
3 assertThat("").isEmpty();
4 assertThat("NotEmpty").isNotEmpty();
5 assertThat("Hello").startsWith("Hel");
6 assertThat("World").endsWith("rld");
7 assertThat("HELLO").isEqualToIgnoringCase("hello");
```

Collections:

```
1 List<String> list = List.of("a", "b", "c");
2
3 assertThat(list).hasSize(3);
4 assertThat(list).contains("b");
5 assertThat(list).doesNotContain("d");
6 assertThat(list).containsExactly("a", "b", "c");
7 assertThat(list).containsExactlyInAnyOrder("c", "a", "b");
8 assertThat(list).startsWith("a");
9 assertThat(list).isNotEmpty();
```

Booleans und null:

```
1 assertThat(true).isTrue();
2 assertThat(false).isFalse();
3 assertThat(object).isNull();
4 assertThat(object).isNotNull();
5 assertThat(object).assertInstanceOf(String.class);
```


Exception-Tests mit assertThrows:

Einfacher Exception-Test:

```
1 @Test
2 @DisplayName("Division durch Null wirft ArithmeticException")
3 void should_throw_exception_for_division_by_zero() {
4     // Arrange
5     Calculator calc = new Calculator();
6
7     // Act & Assert
8     ArithmeticException exception = assertThrows(
9         ArithmeticException.class,
10        () -> calc.divide(10, 0)
11    );
12
13    // Optional: Exception-Details prüfen
14    assertThat(exception.getMessage())
15        .contains("by zero");
16 }
```

Warum Lambda-Ausdruck?

```
1 // FALSCH - Exception fliegt vor assertThrows
2 @Test
3 void wrong_exception_test() {
4     Calculator calc = new Calculator();
5     int result = calc.divide(10, 0); // Exception hier!
6
7     assertThrows(ArithmeticException.class,
8        () -> { /* code never reached */ });
9 }
10
11 // RICHTIG - Code wird erst in Lambda ausgeführt
12 @Test
13 void correct_exception_test() {
14     Calculator calc = new Calculator();
15
16     assertThrows(ArithmeticException.class,
17        () -> calc.divide(10, 0)); // Exception hier gefangen
18 }
```

③ Tag 2 - Testing & Codequalität

Testing Basics

TDD-Zyklus

Qualität von Software

Code Smells im Kleinen

Traditioneller Ansatz:

- ① Code schreiben
- ② Tests schreiben (wenn überhaupt)
- ③ Bugs finden und fixen

TDD-Ansatz:

- ① Test schreiben (der fehlschlägt)
- ② Minimalen Code schreiben (Test wird grün)
- ③ Code refactorieren (bei grünen Tests)

TDD-Vorteile

- Code ist garantiert testbar
- Tests als erste "Clients" des Codes
- Vollständige Testabdeckung
- Tests dokumentieren Anforderungen
- Besseres API-Design

RED Fehlschlagender Test

- Test schreiben BEVOR Code existiert
- Test MUSS fehlschlagen (sonst testet er nichts)
- Prüfen: Schlägt Test aus richtigem Grund fehl?

GREEN Minimaler Code

- Gerade genug Code bis Test grün wird
- Jeder Trick erlaubt: hardcoded returns, copy-paste
- Zeitrahmen: 30 Sekunden bis 2 Minuten

REFACTOR Code verbessern

- NUR bei grünen Tests!
- Externe Schnittstelle bleibt unverändert
- Interne Struktur/Lesbarkeit verbessern

Anforderung: Arabische Zahlen in römische Zahlen umwandeln

1. RED - Erster fehlschlagender Test:

```
1 @Test
2 @DisplayName("1 wird zu I")
3 void should_translate_1_to_I() {
4     // Arrange & Act
5     String result = RomanNumbers.translate(1);
6     // Assert
7     assertThat(result).isEqualTo("I");
8 }
```

Fehler: RomanNumbers Klasse existiert nicht → Kompiliert nicht

2. GREEN - Minimaler Code:

```
1 public class RomanNumbers {
2     public static String translate(int arabic) {
3         return "I"; // Hard-coded für ersten Test!
4     }
5 }
```

Test wird grün ✓

3. REFACTOR - Noch nichts zu refactorieren

4. RED - Zweiter Test:

```
1 @Test
2 @DisplayName("2 wird zu II")
3 void should_translate_2_to_II() {
4     String result = RomanNumbers.translate(2);
5     assertThat(result).isEqualTo("II");
6 }
```

Test schlägt fehl: erwartet "II", bekommt "I"

5. GREEN - Code erweitern:

```
1 public static String translate(int arabic) {
2     if (arabic == 2) return "II";
3     if (arabic == 1) return "I";
4     return "";
5 }
```

Beide Tests grün ✓

6. RED - Dritter Test:

```
1 @Test
2 @DisplayName("3 wird zu III")
3 void should_translate_3_to_III() {
4     String result = RomanNumbers.translate(3);
5     assertThat(result).isEqualTo("III");
6 }
```

7. GREEN - Pattern wird sichtbar:

```
1 public static String translate(int arabic) {
2     if (arabic == 3) return "III";
3     if (arabic == 2) return "II";
4     if (arabic == 1) return "I";
5     return "";
6 }
```

8. REFACTOR - Code vereinfachen:

```
1 public static String translate(int arabic) {
2     return "I".repeat(arabic); // Funktioniert für 1, 2, 3
3 }
```

9. RED - Nächster Test (Regel für 4):

```
1 @Test
2 @DisplayName("4 wird zu IV")
3 void should_translate_4_to_IV() {
4     String result = RomanNumbers.translate(4);
5     assertThat(result).isEqualTo("IV");
6 }
```

Schlägt fehl: erwartet "IV", bekommt "IIII"

10. GREEN - Spezialfall behandeln:

```
1 public static String translate(int arabic) {
2     if (arabic == 4) return "IV";
3     return "I".repeat(arabic);
4 }
```

TDD Fortsetzung: Weitere Tests für 5 ("V"), 9 ("IX"), 10 ("X"), etc.

TDD-Regel

Niemals mehr Code schreiben als nötig, um den aktuellen Test zum Laufen zu bringen!

③ Tag 2 - Testing & Codequalität

Testing Basics

TDD-Zyklus

Qualität von Software

Code Smells im Kleinen

- **Funktionale Angemessenheit**
 - Vollständigkeit
 - Korrektheit
 - Angemessenheit
- **Performance/Efficiency**
 - Zeit-Verhalten
 - Ressourcen-Verbrauch
- **Kompatibilität**
 - Interoperabilität
 - Koexistenz
- **Usability**
 - Bedienbarkeit
 - Lernbarkeit
 - Fehlertoleranz
- **Reliability**
 - Reife
 - Verfügbarkeit
 - Fehlertoleranz
 - Wiederherstellbarkeit
- **Security**
 - Vertraulichkeit
 - Integrität
 - Authentizität
- **Maintainability** ← **Fokus**
 - Modularität
 - Analysierbarkeit
 - Änderbarkeit
 - Testbarkeit
- **Portability**
 - Anpassbarkeit
 - Installierbarkeit
 - Austauschbarkeit

Beispiel: Sicherheit vs. Usability

Sicherheit erhöhen:

- 2-Faktor-Authentifizierung
- Komplexe Passwort-Regeln
- Häufige Re-Authentifizierung
- Detaillierte Eingabvalidierung

Usability sinkt:

- Längerer Login-Prozess
- Passwörter schwer zu merken
- Unterbrechungen im Workflow
- Mehr Schritte für Benutzer

Trade-offs zwischen Qualitätszielen 2

Nicht alle Qualitätsziele können gleichzeitig maximiert werden!

Weitere Trade-offs:

- **Performance vs. Maintainability:** Optimierter Code oft schwer lesbar
- **Funktionalität vs. Usability:** Mehr Features → komplexere UI
- **Kosten vs. Qualität:** Höhere Qualität braucht mehr Zeit/Ressourcen

Doug Bell's Erkenntnis

Ca. 70% der Software-Kosten entstehen NACH der initialen Entwicklung

Wartungsaufgaben:

- Neue Funktionalitäten hinzufügen
- Bugs beheben
- Performance optimieren
- Sicherheitslücken schließen
- An veränderte Anforderungen anpassen
- Auf neue Plattformen portieren

Wann ist Wartbarkeit weniger wichtig?

- Einmalige Skripte (wirklich einmalig!)
- Prototypen (werden nie produktiv)
- Wegwerf-Code

Achtung vor "temporären" Lösungen!

"Nothing is as permanent as a temporary solution"

Modularisierung

- Zerlegung in überschaubare Komponenten
- Änderungen bleiben lokal begrenzt
- Verstehen nur relevanter Teile nötig

Analysierbarkeit

- Code ist verständlich und nachvollziehbar
- Sprechende Namen und klare Struktur
- Gute Dokumentation durch Tests

Änderbarkeit

- Code kann ohne große Umstrukturierung angepasst werden
- Lose Kopplung zwischen Komponenten
- Klare Verantwortlichkeiten

Testbarkeit

- Automatisierte Tests als Sicherheitsnetz
- Tests zeigen sofort, wenn etwas kaputtgeht
- Modularisierung erleichtert isoliertes Testen

Kurzfristig (während Entwicklung):

- Wartbarer Code kostet mehr Zeit
- Mehr Nachdenken über Design
- Zusätzliche Tests schreiben
- Refactoring-Aufwand
- + Weniger Debugging
- + Einfacheres Erweitern

Langfristig (Wartungsphase):

- + Massive Kosteneinsparung
- + Schnellere Feature-Entwicklung
- + Weniger Bugs
- + Einfachere Einarbeitung neuer Entwickler
- + Weniger Stress bei Änderungen

Balance finden

- Nicht über-engineeren
- Ausreichend strukturieren für erwartete Änderungen
- Pragmatisch bleiben

③ Tag 2 - Testing & Codequalität

Testing Basics

TDD-Zyklus

Qualität von Software

Code Smells im Kleinen

Was sind Code Smells?

Code Smells sind:

- Hinweise auf Wartbarkeitsprobleme
- Nicht automatisch Fehler (Code funktioniert)
- Indikatoren für schlechtes Design
- Kandidaten für Refactoring

Wichtig:

- Nicht jeder Smell muss behoben werden
- Kontext und Aufwand beachten
- Pragmatische Entscheidungen treffen

Heute: Code Smells "im Kleinen"

- Auf Methoden- und Klassen-Ebene
- Direkt sichtbare Probleme
- Einfach zu behebende Smells

Schlecht - Mars-Roboter Beispiel:

```
1 public void controlRobot() {
2     // Sensordaten lesen
3     int distance = sensor.readDistance();
4     boolean obstacle = distance < 10;
5
6     // Route planen
7     if (obstacle) {
8         turnLeft();
9         moveForward(5);
10        turnRight();
11    } else {
12        moveForward(10);
13    }
14
15    // Energieverbrauch prüfen
16    if (battery.getLevel() < 20) {
17        sendLowBatteryAlert();
18        activatePowerSaving();
19    }
20
21    // Daten komprimieren und senden
22    String data = collectSensorData();
23    String compressed = compress(data);
24    radio.sendToEarth(compressed);
25 }
```

Besser - Aufgeteilt:

```
1 public void controlRobot() {
2     Environment env = sensorsystem.scan();
3     Route route = navigator.planRoute(env);
4     movement.execute(route);
5     powerManager.checkAndOptimize();
6     dataTransmitter.sendToEarth();
7 }
8
9 // Separate Klassen/Methoden:
10 class SensorSystem {
11     public Environment scan() { /* ... */ }
12 }
13
14 class Navigator {
15     public Route planRoute(Environment env) { /* ... */ }
16 }
17
18 class MovementController {
19     public void execute(Route route) { /* ... */ }
20 }
21
22 class PowerManager {
23     public void checkAndOptimize() { /* ... */ }
24 }
```

Vorteile der Aufteilung:

- Jede Klasse hat eine klare Verantwortlichkeit
- Einfacher zu testen (einzelne Komponenten)
- Einfacher zu verstehen und zu ändern

Schlechte Namen:

```
1 // Was macht diese Methode?  
2 public static int fibo(int n) {  
3     int y = 0, z = 0;  
4     while(y < n) {  
5         z += 1 + 2*y++;  
6     }  
7     return z;  
8 }  
9  
10 // Lügender Name!  
11 private Set<String> kundenListe;  
12  
13 // Uninformativ  
14 int d;  
15 d += 30;
```

Gute Namen:

```
1 // Klarerer Name enthüllt: es ist n^2!  
2 public static int calculateSquareOfNumber(int n) {  
3     int result = 0;  
4     int iterator = 0;  
5     while(iterator < n) {  
6         result += 1 + 2 * iterator;  
7         iterator++;  
8     }  
9     return result;  
10 }  
11  
12 // Wahrheitsgemäss  
13 private Set<String> kunden;  
14  
15 // Aussagekräftig  
16 int daysSinceLastUpdate;  
17 daysSinceLastUpdate += DAYS_IN_APRIL;
```

- Namen beschreiben Zweck/Aufgabe
- Namen dürfen NIEMALS lügen
- Spezifisch sein (customers statt data)
- Kontextabhängige Länge (kurze Namen bei kleinem Scope OK)

camelCase - Methoden und Variablen:

```
1 // Methoden: Verben
2 public void calculateInterest()
3 public String getUsername()
4 public boolean isValid()
5 public void setCustomerAddress()
6
7 // Variablen: Nomen
8 String firstName;
9 int customerAge;
10 List<Order> pendingOrders;
11 boolean isReadyForProcessing;
```

PascalCase - Klassen und Interfaces:

```
1 // Klassen: Nomen
2 public class CustomerManager
3 public class OrderProcessor
4 public class PaymentGateway
5
6 // Interfaces: oft Adjektive mit -able
7 public interface Serializable
8 public interface Comparable<T>
9 public interface Runnable
```

SCREAMING_SNAKE_CASE - Konstanten:

```
1 public static final int MAX_RETRY_COUNT = 3;
2 public static final String DEFAULT_ENCODING = "UTF-8";
3 public static final double PI = 3.14159;
4
5 // Enums
6 public enum Status {
7     PENDING,
8     IN_PROGRESS,
9     COMPLETED,
10    CANCELLED
11 }
```

Boolean-Naming:

```
1 // Präfixe: is, has, can, should
2 boolean isValid;
3 boolean hasPermission;
4 boolean canEdit;
5 boolean shouldRetry;
6
7 // Methoden genauso
8 public boolean isEmpty()
9 public boolean hasNext()
10 public boolean canAccess()
```

Schlecht - Beschreibt WAS:

```
1 // Prüfen ob erste Argument gültige Mailadresse ist
2 if (Pattern.matches("[a-zA-Z0-9.!#$%&'*/+=?^_`{|}~-]+@[a-zA-Z0-9-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\\.[a-zA-Z0-9-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)*$", args[0])) {
3     // ...
4 }
5
6 // i um 1 erhöhen
7 i++;
8
9 // Schleife über alle Kunden
10 for (Customer customer : customers) {
11     // ...
12 }
```

Besser - Code selbsterklärend machen:

```
1 // Statt Kommentar: Methode mit sprechendem Namen
2 if (isValidEmailAddress(args[0])) {
3     // ...
4 }
5
6 private static boolean isValidEmailAddress(String email) {
7     // Vereinfachter RegEx aus HTML5 Standard,
8     // da RFC-konformer RegEx zu komplex für unseren Use-Case
9     return Pattern.matches("[a-zA-Z0-9.!#$%&'*/+=?^_`{|}~-]+@...", email);
10 }
11
12 // Keine Kommentare nötig:
13 customerIndex++;
14
15 for (Customer customer : customers) {
16     processCustomerOrder(customer);
17 }
```

- Erklären WARUM (nicht was)
- Rechtliche Hinweise, Copyrights
- Warnung vor Konsequenzen
- TODO-Kommentare (temporär)

Problem: Methoden mit zu vielen Zeilen Code

Probleme langer Methoden:

- Schwer zu verstehen ("Wo war ich gerade?")
- Meist mehrere Verantwortlichkeiten vermischt
- Schwer zu testen (viele Code-Pfade)
- Schwer wiederzuverwenden
- Schwer zu debuggen

Faustregel: Methode sollte auf eine Bildschirmseite passen (ca. 20-30 Zeilen)

Lösung: Extract Method

```
1 // Vorher: Eine lange Methode mit 50+ Zeilen
2 public void processOrder(Order order) {
3     // 10 Zeilen Validierung
4     // 15 Zeilen Preisberechnung
5     // 10 Zeilen Rabatt-Logik
6     // 15 Zeilen E-Mail versenden
7 }
8
9 // Nachher: Aufgeteilt in kleinere Methoden
10 public void processOrder(Order order) {
11     validateOrder(order);
12     double total = calculateTotal(order);
13     sendConfirmation(order, total);
14 }
```


Alle Anweisungen in einer Methode sollten den gleichen Detailgrad haben

Verletzt SLAP (Detail + Abstraktion):

```
1 private void printReport() {
2     // Hohe Abstraktion
3     printHeader();
4
5     // Niedrige Abstraktion (Details)
6     int maxLength = 0;
7     for (Product product : products) {
8         int length = product.getName().length();
9         if (length > maxLength) {
10             maxLength = length + 1;
11         }
12     }
13
14     // Wieder hohe Abstraktion
15     printProductTable(maxLength);
16     printFooter();
17 }
```

Erfüllt SLAP (gleiche Abstraktionsebene):

```
1 private void printReport() {
2     printHeader();
3     int maxLength = calculateMaxProductNameLength();
4     printProductTable(maxLength);
5     printFooter();
6 }
7
8 private int calculateMaxProductNameLength() {
9     int maxLength = 0;
10    for (Product product : products) {
11        int length = product.getName().length();
12        if (length > maxLength) {
13            maxLength = length + 1;
14        }
15    }
16    return maxLength;
17 }
```

- Jede Methode ist auf einer "Zoom-Stufe"
- Bei Bedarf in Details "hineinzoomen"
- Übersicht bleibt erhalten

Problem: Methoden mit zu vielen Parametern

Problematisch:

```
1 // Was bedeuten die Parameter?
2 public double calculateDistance(
3     double x1, double y1, double z1,
4     double x2, double y2, double z2) {
5     return Math.sqrt(
6         Math.pow(x2-x1,2) + Math.pow(y2-y1,2) + Math.pow(z2-z1,2)
7     );
8 }
9
10 // Aufruf - fehleranfällig!
11 double dist = calculateDistance(
12     1.0, 2.0, 3.0,    // Start
13     4.0, 5.0, 6.0    // End - oder umgekehrt?
14 );
```

Besser - Parameter-Objekte:

```
1 public record Point3D(double x, double y, double z) {}
2
3 public double calculateDistance(Point3D start, Point3D end) {
4     return Math.sqrt(
5         Math.pow(end.x() - start.x(), 2) +
6         Math.pow(end.y() - start.y(), 2) +
7         Math.pow(end.z() - start.z(), 2)
8     );
9 }
10
11 // Aufruf - selbsterklärend!
12 Point3D start = new Point3D(1.0, 2.0, 3.0);
13 Point3D end = new Point3D(4.0, 5.0, 6.0);
14 double distance = calculateDistance(start, end);
```

Boolean-Parameter vermeiden:

```
1 // Schlecht: Was bedeutet true/false?
2 public Package wrap(Product product, boolean isGift) { ... }
3
4 // Besser: Separate Methoden
5 public Package wrapAsGift(Product product) { ... }
6 public Package wrapNormally(Product product) { ... }
```

Don't Repeat Yourself (DRY)

Nicht der identische Code ist das Problem, sondern doppeltes WISSEN!

Problematisch - Gleiche Geschäftslogik:

```
1 // Rabatt-Berechnung in OrderService
2 public double calculateOrderTotal(Order order) {
3     double total = order.getSubtotal();
4     if (order.getCustomer().isPremium()) {
5         total *= 0.9; // 10% Rabatt
6     }
7     return total;
8 }
9
10 // Gleiche Logik in InvoiceService
11 public double calculateInvoiceTotal(Invoice invoice) {
12     double total = invoice.getAmount();
13     if (invoice.getCustomer().isPremium()) {
14         total *= 0.9; // 10% Rabatt - DUPLIZIERT!
15     }
16     return total;
17 }
```

OK - Verschiedenes Wissen (trotz gleichem Code):

```
1 // Alter validieren
2 public boolean validateAge(int age) {
3     return age >= 0 && age <= 150;
4 }
5
6 // Menge validieren
7 public boolean validateQuantity(int quantity) {
8     return quantity >= 0 && quantity <= 150;
9 }
10
11 // Gleicher Code, aber verschiedene Geschäftsregeln!
12 // Alter-Obergrenze könnte sich unabhängig von
13 // Mengen-Obergrenze ändern
```

Don't Repeat Yourself (DRY)

Vorteile:

- Änderungen nur an einer Stelle
- Konsistenz automatisch gewährleistet
- Weniger Vergesslichkeit bei Änderungen

4 Tag 3 - Architektur & Prinzipien

Bausteine & Strukturen

Vererbung & Polymorphismus

SOLID-Prinzipien

Code Smells im Großen

Warum Architektur wichtig ist:

- 70% der Software-Kosten entstehen NACH der initialen Entwicklung
- Wartungsaufgaben: Neue Features, Bugs beheben, Performance optimieren
- Schlechte Architektur = exponentiell steigende Kosten

Zentrale Herausforderung:

- Komplexe Software in verständbare Teile zerlegen
- Abhängigkeiten zwischen Komponenten minimieren
- Änderungen lokal begrenzen

Von LCHC zu praktischen Prinzipien

- **LCHC**: Das Fundament guter Architektur
- Single Responsibility & Information Hiding
- Zerlegungsstrategien: Fachlich vs. Technisch
- SOLID-Prinzipien & Code Smells

Low Coupling, High Cohesion - Die Goldene Regel

Ziel: Lose Kopplung zwischen Komponenten + Hohe Kohäsion innerhalb von Komponenten

Niedrige Kopplung (Low Coupling):

- Wenige Abhängigkeiten zwischen Komponenten
- Änderungen bleiben lokal begrenzt
- Komponenten können unabhängig entwickelt werden
- Bessere Testbarkeit & Wiederverwendung

Hohe Kohäsion (High Cohesion):

- Starker innerer Zusammenhalt
- Alle Teile arbeiten für gemeinsames Ziel
- Klarer, eindeutiger Zweck
- Änderungen betreffen meist die ganze Komponente

Warum LCHC funktioniert

Hohe Kohäsion und lose Kopplung ergänzen sich perfekt: Starke interne Bindung bei schwachen externen Abhängigkeiten.

SRP Kernaussage

“Eine Klasse sollte nur einen Grund haben, sich zu ändern.”

Oder genauer: Dinge, die sich aus demselben Grund ändern, gehören zusammen. Dinge, die sich aus unterschiedlichen Gründen ändern, gehören getrennt.

SRP → Hohe Kohäsion

Das SRP ist der Schlüssel zu hoher Kohäsion: Wenn eine Komponente nur eine Verantwortlichkeit hat, arbeiten alle ihre Teile für dasselbe Ziel zusammen.

Praktisches Vorgehen - Frage stellen:

- ① Welche Personengruppen könnten Änderungen an dieser Klasse anfordern?
- ② Welche unterschiedlichen Geschäftszweige sind betroffen?
- ③ Welche verschiedenen fachlichen Bereiche werden behandelt?

Problem - Mehrere Verantwortlichkeiten in einer Klasse:

```
1 public class Employee {
2     private String name;
3     private double baseSalary;
4     private List<EmployeeId> manages;
5
6     // Lohnbuchhaltung
7     public double calculateSalary() {
8         return baseSalary + managementBonus();
9     }
10    private double managementBonus() {
11        return manages.size() * 500.0;
12    }
13
14    // Datenbank-Persistierung
15    public void save() {
16        Connection conn = DriverManager.getConnection(...);
17        PreparedStatement stmt = conn.prepareStatement(...);
18        stmt.setDouble(1, calculateSalary());
19        stmt.executeUpdate();
20    }
21
22    // Berichtswesen
23    public String generateReport() {
24        return String.format("Employee: %s, Salary: %.2f",
25                               name, calculateSalary());
26    }
27 }
```

SRP Beispiel - Analyse der Verantwortlichkeiten

Wer würde Änderungen an der Employee-Klasse anfordern?

- **Lohnbuchhaltung**: Gehaltsberechnung ändern
- **Datenbankadministration**: Persistierung ändern
- **Controlling**: Report-Format ändern

Die resultierenden Probleme:

- Jede Änderung kann andere, unbeteiligte Bereiche beeinflussen
- Verschiedene Teams müssen an derselben Klasse arbeiten
- Merge-Konflikte sind vorprogrammiert
- Tests sind schwer zu isolieren und zu schreiben

Geschäftslogik:

```
1 public class Employee {
2     private final String name;
3     private final double baseSalary;
4     private final List<EmployeeId> manages;
5     public double calculateSalary() {
6         return baseSalary + manages.size() * 500.0;
7     }
8     public String getName() { return name; }
9 }
```

Persistierung:

```
1 public class EmployeeRepository {
2     public void save(Employee employee) {
3         Connection conn = getConnection();
4         PreparedStatement stmt = conn.prepareStatement( "UPDATE
5             employees SET salary=? WHERE name=?");
6         stmt.setDouble(1, employee.calculateSalary());
7         stmt.setString(2, employee.getName());
8         stmt.executeUpdate();
9     }
10 }
```

Berichtswesen:

```
1 public class EmployeeReporter {
2     public String generateReport(Employee employee) {
3         return String.format(
4             "Employee: %s\n" +
5             "Salary: %.2f EUR\n" +
6             "Status: %s",
7             employee.getName(),
8             employee.calculateSalary(),
9             getStatus(employee));
10    }
11    private String getStatus(Employee emp) {
12        // Report-spezifische Logik
13        return "Active";
14    }
15 }
```

Vorteile: Jede Klasse hat genau eine Verantwortlichkeit und einen Grund zur Änderung!

IHP Kernaussage

"Komponenten so schneiden, dass sie Entscheidungen über die Umsetzung kapseln und nach außen verstecken."

IHP → Niedrige Kopplung

Das IHP reduziert Kopplung, indem es Client-Code von Implementierungsdetails abschirmt.

Konkrete Vorteile:

- Implementierungsdetails können geändert werden, ohne Client-Code zu beeinträchtigen
- Klarere, stabilere Schnittstellen (APIs)
- Bessere Testbarkeit durch definierte Grenzen
- Weniger unbeabsichtigte Abhängigkeiten

Was wird versteckt:

- Datenstrukturen (private Felder)
- Algorithmen (private Methoden)
- Implementierungsentscheidungen
- Komplexität der internen Abläufe

Was wird offengelegt:

- Fachliche Operationen (public Methoden)
- Notwendige Konfigurationsmöglichkeiten
- Schnittstellen für Interaktion

Schlecht - Keine Kapselung:

```
1 public class Point {  
2     public double x;  
3     public double y;  
4 }  
5  
6 // Client-Code:  
7 Point p = new Point();  
8 p.x = 3.0;  
9 p.y = 4.0;  
10 double distance = Math.sqrt(p.x * p.x + p.y * p.y);
```

Problem: Interne Darstellung ist fest verdrahtet!

Gut - Mit Kapselung:

```
1 public class Point {  
2     private final double x;  
3     private final double y;  
4  
5     private Point(double x, double y) {  
6         this.x = x;  
7         this.y = y;  
8     }  
9  
10    public static Point fromCartesian(double x, double y) {  
11        return new Point(x, y);  
12    }  
13  
14    public double getX() { return x; }  
15    public double getY() { return y; }  
16  
17    public double getRadius() {  
18        return Math.sqrt(x*x + y*y);  
19    }  
20 }  
21  
22 // Client-Code:  
23 Point p = Point.fromCartesian(3.0, 4.0);  
24 double distance = p.getRadius();
```

Jetzt kann die interne Darstellung geändert werden:

```
1 public class Point {
2     private final double radius; // Jetzt polar!
3     private final double theta;
4
5     private Point(double radius, double theta) {
6         this.radius = radius;
7         this.theta = theta;
8     }
9
10    public static Point fromCartesian(double x, double y) {
11        return new Point(Math.hypot(x, y), Math.atan2(y, x));
12    }
13
14    public double getX() { return radius * Math.cos(theta); }
15    public double getY() { return radius * Math.sin(theta); }
16    public double getRadius() { return radius; }
17 }
```

Vorteil

Client-Code muss NICHT geändert werden! Die Schnittstelle bleibt identisch.

Wie sollten wir Software strukturieren?

Aspekt	Fachliche Zerlegung (BEVORZUGT)	Technische Zerlegung
Orientierung	Geschäftsprozesse & Domänenlogik	Technische Schichten & Infrastruktur
Teams	Domänenexperten, Product Owner	Entwickler, Architekten, DevOps
Kohäsion	Hoch (zusammengehörige Geschäftslogik)	Mittel (gemeinsame tech. Verantwortung)
Änderungen	Lokal begrenzt, team-autonom	Bei Technologie-Updates
Vorteile	Domäne-getrieben, Team-Autonomie	Technische Konsistenz, Wiederverwendung
Nachteile	Technische Duplikation möglich	Fachliche Logik verstreut
Beispiele	BestellService, PaymentService	DatabaseLayer, LoggingService

Best Practice

Primär fachlich zerlegen für hohe Kohäsion, technische Aspekte als Cross-Cutting Concerns für Wiederverwendung.

Kopplung Definition

Wenn eine Änderung Δ an Komponente A dazu führt, dass B geändert werden muss, ist B bezüglich Δ an A gekoppelt.

Kopplungsarten (schwach \rightarrow stark):

- 1 **Aufruf**: B ruft Methoden von A
- 2 **Konstruktion**: B erzeugt A-Instanzen
- 3 **Vererbung**: B erbt von A (stärkste)
- 4 **Unsichtbar**: Implizite Abhängigkeiten

Lösungsstrategien:

- **Aufruf**: Interface einführen (DIP)
- **Konstruktion**: Dependency Injection
- **Vererbung**: Komposition bevorzugen
- **Unsichtbar**: Defensive Kopien, Dokumentation

Merkregel

Stärkere Kopplung = größere Probleme. Vererbung ist fast immer die schlechteste Option!

Extrem problematisch - Geteilter, änderbarer Zustand:

```
1 public class Interpolation {
2     private final int[] array;
3
4     public Interpolation(int[] eingabe) {
5         Arrays.sort(eingabe); // sortiert eingabe in-place!
6         this.array = eingabe; // Referenz geteilt!
7     }
8
9     public int search(int value) {
10         return Arrays.binarySearch(array, value);
11     }
12 }
13
14 // Client-Code kann array zerstören:
15 int[] foo = new int[] {1, -2, 3, 7};
16 Interpolation ip = new Interpolation(foo);
17
18 // Später im Code:
19 for(int i=0; i < foo.length; i++) {
20     foo[i] = new Random().nextInt(); // Zerstört Sortierung!
21 }
22 // ip.search() funktioniert nicht mehr korrekt!
```

Die Lösung - Defensive Kopie:

```
1 public class Interpolation {
2     private final int[] array;
3
4     public Interpolation(int[] eingabe) {
5         // Defensive Kopie erstellen
6         this.array = Arrays.copyOf(eingabe, eingabe.length);
7         Arrays.sort(this.array); // Jetzt sicher!
8     }
9
10    public int search(int value) {
11        return Arrays.binarySearch(array, value);
12    }
13
14    // Auch bei Rückgaben defensive Kopie:
15    public int[] getArray() {
16        return Arrays.copyOf(array, array.length);
17    }
18 }
```

Wichtig

`final` bei Arrays verhindert nur Neuuzuweisung, nicht Inhaltsänderung!

Law of Demeter Regel

Ein Objekt sollte nur mit seinen “unmittelbaren Freunden” sprechen, nicht mit deren Freunden.

Verletzung - Rechnung kennt Produkt:

```
1 class Rechnung {
2     double berechneZeilenpreis(Position pos) {
3         // BAD: Rechnung kennt Position UND Produkt
4         return pos.getAnzahl() * pos.getProdukt().getPreis();
5     }
6 }
7 class Position {
8     private int anzahl;
9     private Produkt produkt;
10    public int getAnzahl() { return anzahl; }
11    public Produkt getProdukt() { return produkt; }
12 }
13 class Produkt {
14     private double preis;
15     public double getPreis() { return preis; }
16 }
```

Korrekt - Position kapselt Details:

```
1 class Rechnung {
2     double berechneZeilenpreis(Position pos) {
3         // GOOD: Rechnung kennt nur Position
4         return pos.getZeilenpreis();
5     }
6 }
7 class Position {
8     private int anzahl;
9     private Produkt produkt;
10    public int getAnzahl() { return anzahl; }
11    // Position kapselt Produktdetails
12    public double getZeilenpreis() {
13        return anzahl * produkt.getPreis();
14    }
15 }
16 class Produkt {
17     private double preis;
18     public double getPreis() { return preis; }
19 }
```

Problem - Hart verdrahtet:

```
1 class CountingSummarizer {
2     private Segmentation segmentation;
3
4     public CountingSummarizer() {
5         // Hard gekoppelt!
6         this.segmentation = new AgeSegmentation();
7     }
8
9     public Map<String, Integer> summarize(List<Customer> customers)
10    {
11        Map<String, List<Customer>> segments =
12            segmentation.getSegments(customers);
13
14        return segments.entrySet().stream()
15            .collect(toMap(
16                Map.Entry::getKey,
17                e -> e.getValue().size()));
18    }
19 }
```

Problem: Kann nicht getestet oder erweitert werden!

Lösung - Dependency Injection:

```
1 interface Segmentation {
2     Map<String, List<Customer>> getSegments(
3         List<Customer> customers);
4 }
5
6 class CountingSummarizer {
7     private final Segmentation segmentation;
8
9     // Injiziert via Konstruktor
10    public CountingSummarizer(
11        Segmentation segmentation) {
12        this.segmentation = segmentation;
13    }
14
15    public Map<String, Integer> summarize(
16        List<Customer> customers) {
17        Map<String, List<Customer>> segs =
18            segmentation.getSegments(customers);
19
20        return segs.entrySet().stream()
21            .collect(toMap(
22                Map.Entry::getKey,
23                e -> e.getValue().size()));
24    }
25 }
```

Kohäsion - Das **C** in LCHC

Interne Bindung innerhalb einer Komponente - alle Teile arbeiten zusammen!

Hohe Kohäsion erkennen:

- Methoden nutzen dieselben Attribute
- Klassen-Zweck in 1 Satz beschreibbar
- SRP eingehalten (ein Änderungsgrund)
- Zusammengehörige Daten gekapselt

Niedrige Kohäsion erkennen:

- Methoden nutzen verschiedene Attributsätze
- Klasse schwer beschreibbar
- Methoden könnten woanders stehen
- Utility-Klassen mit unzusammenhängenden Methoden

Quick-Check

Hohe Kohäsion? Kann ich den Zweck der Klasse in einem klaren Satz beschreiben?

Problem - Customer mit verschiedenen Verantwortlichkeiten:

```
1 class Customer {
2     // Verschiedene Attributsätze
3     List<Order> orders;           // Order-Management
4     String emailAddress;         // Kommunikation
5     String name;                 // Basis-Daten
6     String shippingAddress;      // Versand
7
8     // Methoden greifen auf verschiedene Attributsätze zu
9     List<Order> getOrders() { // -> orders
10         return orders;
11     }
12
13     void sendWelcomeEmail() { // -> emailAddress, name
14         EmailService.send(emailAddress, "Welcome " + name + "!");
15     }
16
17     void printShippingLabel() { // -> name, shippingAddress
18         System.out.println("Ship to: " + name);
19         System.out.println("Address: " + shippingAddress);
20     }
21 }
```

Problem: Niedrige Kohäsion - jede Methode nutzt andere Attribute!

Kohäsion Beispiel - LCHC durch Aufspaltung erreichen

```
1 class Customer {
2     private final String name;
3     private final CustomerId id;
4     public String getName() { return name; }
5     public CustomerId getId() { return id; }
6     // ...
7 }
8
9 class OrderManager {
10     private final List<Order> orders;
11     private final CustomerId customerId;
12     public void addOrder(Order order) {
13         orders.add(order);
14     }
15     public List<Order> getOrderHistory() {
16         return List.copyOf(orders);
17     }
18     public double getTotalOrderValue() {
19         return orders.stream()
20             .mapToDouble(Order::getValue)
21             .sum();
22     }
23 }
```

```
1 class NotificationService {
2     private final String emailAddress;
3     private final String customerName;
4     public void sendWelcomeEmail() {
5         EmailService.send(emailAddress,
6             "Welcome " + customerName + "!");
7     }
8     public void sendOrderConfirmation(Order o) {
9         EmailService.send(emailAddress,
10             "Order " + o.getId() + " confirmed");
11     }
12 }
13 class ShippingService {
14     private final String customerName;
15     private final Address shippingAddress;
16     public void printShippingLabel() {
17         System.out.println("Ship to: " + customerName);
18         System.out.println("Address: " + shippingAddress);
19     }
20     public double calculateShippingCost() {
21         return shippingAddress.getShippingZone().getBaseCost();
22     }
23 }
```


Jeder Code Smell verletzt LCHC! Erkennungsmuster:

Code Smell	Kohäsion	Kopplung	Erkennungsmuster
Large Class	niedrig	-	Klasse hat > 5 verschiedene Aufgaben
Divergent Change	niedrig	-	“Diese Klasse ändert sich, wenn...” hat > 1 Antwort
Data Clumps	niedrig	-	Parallele Arrays oder Parameter-Gruppen
Feature Envy	-	hoch	Methode nutzt andere Klasse mehr als eigene
Message Chains	-	hoch	<code>a.getB().getC().getD()</code>
Shotgun Surgery	-	hoch	Eine Änderung betrifft > 3 Klassen

Template für Refactoring-Aufgaben

- 1 Smell identifizieren
- 2 LCHC-Verletzung benennen
- 3 Lösung: Extract Class / Move Method / Hide Delegate

Zusammenhang der Konzepte:

Ebene	Prinzip/Konzept
Grundlage	LCHC - Low Coupling, High Cohesion
Kernprinzipien	SRP - Single Responsibility (für hohe Kohäsion)
	IHP - Information Hiding (für niedrige Kopplung)
Strategien	Fachliche Zerlegung (aus SRP abgeleitet)
	Law of Demeter (aus IHP abgeleitet)
Erkennung	Code Smells - Indikatoren für Verletzungen

Beziehungen:

- LCHC ist das **Grundprinzip** und der Bewertungsmaßstab für wartbare Architektur
- SRP und IHP sind die **praktischen Umsetzungen** von LCHC
- Fachliche Zerlegung und Law of Demeter sind **Strategien**
- Code Smells sind **Warnsignale** für Prinzipien-Verletzungen

4 Tag 3 - Architektur & Prinzipien

Bausteine & Strukturen

Vererbung & Polymorphismus

SOLID-Prinzipien

Code Smells im Großen

Polymorphismus = “Vielgestaltigkeit” - Ein Interface, verschiedene Implementierungen

- 1. Ad-hoc Polymorphismus (Overloading)**
 - Methoden-Overloading: Derselbe Name, verschiedene Parameter
 - Operator-Overloading: +-Operator für String und int
 - Compile-time Entscheidung
- 2. Parametrischer Polymorphismus (Generics)**
 - Typ-Parameter für Klassen und Methoden
 - `List<String>, <T> void process(T item)`
 - Nicht spezifisch für OOP
- 3. Vererbungspolymorphismus**
 - Spezifisch für objektorientierte Programmierung
 - Verschiedene Typen über gemeinsame Schnittstelle verwenden
 - Runtime-Entscheidung über dynamische Methodenbindung

Heute: Fokus auf Vererbungspolymorphismus

Wie Interfaces und Vererbung zur Entkopplung beitragen können.

Overloading (Überladen):

```
1 public class Calculator {  
2     // Verschiedene add-Methoden  
3     public int add(int a, int b) {  
4         return a + b;  
5     }  
6  
7     public double add(double a, double b) {  
8         return a + b;  
9     }  
10  
11     public String add(String a, String b) {  
12         return a + b;  
13     }  
14  
15     // Compiler wählt basierend auf Parametern  
16 }  
17  
18 // Verwendung:  
19 Calculator calc = new Calculator();  
20 int result1 = calc.add(5, 3);           // int-Variante  
21 double result2 = calc.add(5.5, 3.2);    // double-Variante  
22 String result3 = calc.add("Hi", "!");   // String-Variante
```

- Mehrere Methoden in einer Klasse
- Gleicher Name, verschiedene Parameter
- Compile-time Polymorphismus
- Statische Methodenauswahl

Overriding (Überschreiben):

```
1 abstract class Animal {
2     public abstract String makeSound();
3     public void sleep() {
4         System.out.println("Zzz...");
5     }
6 }
7 class Dog extends Animal {
8     @Override
9     public String makeSound() {
10         return "Woof!";
11     }
12 }
13 class Cat extends Animal {
14     @Override
15     public String makeSound() {
16         return "Meow!";
17     }
18 }
19
20 // Verwendung:
21 Animal animal = new Dog(); // Polymorphie
22 System.out.println(animal.makeSound()); // "Woof!" - Runtime
```

- Methode aus Superklasse überschreiben
- Gleiche Signatur (Name + Parameter)
- Runtime Polymorphismus
- Dynamische Methodenbindung

Interface-Vererbung (BEVORZUGT):

- Harmlose Kopplung
- Nur Namen und Signaturen
- Gutes Mittel zur Entkopplung
- Mehrfach-Vererbung möglich

```
1 interface Drawable { void draw(); }
2 interface Movable { void move(int dx, int dy); }
3
4 class GameCharacter
5     implements Drawable, Movable {
6     @Override
7     public void draw() { /* ... */ }
8     @Override
9     public void move(int dx, int dy) { /* ... */ }
10 }
```

Klassen-Vererbung (VORSICHTIG):

- Erzeugt starke Kopplung
- Bricht Kapselung von Objekten
- Gemeinsame Entwicklung nötig
- Nur Einfach-Vererbung möglich

```
1 abstract class Vehicle {
2     protected int speed = 0;
3     public void accelerate() {
4         speed += 10;
5     }
6     public abstract void startEngine();
7 }
8
9 class Car extends Vehicle {
10     @Override
11     public void startEngine() { /* ... */ }
12     public void brake() {
13         speed -= 15;
14     }
15 }
```

```
1 public class Employee {
2     private boolean management;
3     private List<EmployeeId> manages;
4     private Salary baseSalary;
5     public Salary computeSalary() {
6         if (management) {
7             return baseSalary.addManagementBonus(
8                 manages.size());
9         } else {
10            return baseSalary;
11        }
12    }
13    public String getDescription() {
14        if (management) {
15            return "Manager with " + manages.size() +
16                " reports";
17        } else {
18            return "Regular employee";
19        }
20    }
21 }
```

Problem

Jeder neue Employee-Typ erfordert Änderungen in ALLEN Methoden (OCP-Verletzung).


```
1 interface Employee {
2     Salary computeSalary();
3     String getDescription();
4 }
5 class RegularEmployee implements Employee {
6     private final Salary baseSalary;
7     public Salary computeSalary() { return baseSalary; }
8     public String getDescription() { return "Regular employee"; }
9 }
10 class Manager implements Employee {
11     private final Salary baseSalary;
12     private final List<EmployeeId> manages;
13     public Salary computeSalary() {
14         return baseSalary.addManagementBonus(manages.size());
15     }
16     public String getDescription() {
17         return "Manager with " + manages.size() + " reports";
18     }
19 }
```

Vorteil

Neue Typen können ohne Änderung bestehender Klassen hinzugefügt werden! Open/Closed Principle erfüllt - offen für Erweiterung, geschlossen für Modifikation!

Problem der starken Kopplung bei Klassen-Vererbung:

```
1 public class InstrumentedHashSet<E> extends HashSet<E> {
2     private int addCount = 0;
3     @Override
4     public boolean add(E e) {
5         addCount++;
6         return super.add(e);
7     }
8     @Override
9     public boolean addAll(Collection<? extends E> c) {
10        addCount += c.size(); // PROBLEM: Doppeltes Zählen!
11        return super.addAll(c);
12    }
13    public int getAddCount() { return addCount; }
14 }
```

Test schlägt fehl:

```
1 @Test
2 void testAddCount() {
3     InstrumentedHashSet<Integer> set = new InstrumentedHashSet<>();
4     set.addAll(List.of(1, 2, 3));
5
6     // Erwartet: 3, Tatsächlich: 6
7     assertThat(set.getAddCount()).isEqualTo(3);
8 }
```

Warum schlägt er fehl?

- `HashSet.addAll()` ruft intern `add()` auf
- Unsere `add()`-Methode wird aufgerufen
- Jedes Element wird zweimal gezählt
- **Kopplung an interne Implementierung!**

Bessere Lösung - Komposition verwenden:

```
1 public class InstrumentedSet<E> implements Set<E> {
2     private int addCount = 0;
3     private final Set<E> set; // Komposition statt Vererbung
4     public InstrumentedSet(Set<E> set) {
5         this.set = set;
6     }
7     @Override
8     public boolean add(E e) {
9         addCount++;
10        return set.add(e); // Delegation
11    }
12    @Override
13    public boolean addAll(Collection<? extends E> c) {
14        addCount += c.size();
15        return set.addAll(c); // Delegation - kein doppeltes Zählen!
16    }
17    public int getAddCount() { return addCount; }
18    // Alle anderen Set-Methoden an set delegieren...
19    @Override public int size() { return set.size(); }
20    // ... usw.
21 }
```

Vorteile: Keine Kopplung an interne Implementierung, jede Set-Implementierung funktioniert!

Refused Bequest Definition

Subklasse “verweigert” geerbte Methoden der Superklasse, meist durch `UnsupportedOperationException`.

Problematisches Beispiel:

```
1 public interface Bird {  
2     void fly();  
3     void eat();  
4 }  
5  
6 public class Ostrich implements Bird {  
7     @Override  
8     public void fly() {  
9         throw new UnsupportedOperationException();  
10    }  
11  
12    @Override  
13    public void eat() {  
14        // ...  
15    }  
16 }
```

Probleme:

- **Verletzung des Liskov Substitution Prinzips**
- Ein Strauß (`Ostrich`) ist kein vollwertiger Vogel (`Bird`) im Sinne des Interfaces.
- Interface zu groß (ISP-Verletzung)

Lösung:

- Interfaces aufteilen (Interface Segregation Principle)

Lösung: Kleinere, spezifischere Interfaces

Die Funktionalität wird in separate Interfaces aufgeteilt. Klassen implementieren nur, was sie wirklich können.

```
1 public interface Bird {  
2     void eat();  
3 }  
4 public interface FlyingBird extends Bird {  
5     void fly();  
6 }  
7 public class Ostrich implements Bird {  
8     @Override  
9     public void eat() { /* ... */ }  
10    // Keine fly() Methode mehr nötig!  
11 }  
12 public class Swallow implements FlyingBird {  
13     @Override  
14     public void fly() { /* ... */ }  
15     @Override  
16     public void eat() { /* ... */ }  
17 }
```

4 Tag 3 - Architektur & Prinzipien

Bausteine & Strukturen

Vererbung & Polymorphismus

SOLID-Prinzipien

Code Smells im Großen

SOLID - Fünf fundamentale Prinzipien für wartbaren objektorientierten Code:

Single Responsibility Eine Klasse sollte nur einen Grund zur Änderung haben

Open/Closed Offen für Erweiterungen, geschlossen für Modifikationen

Liskov Substitution Subtypen müssen durch Supertypen ersetzbar sein

Interface Segregation Clients sollen nicht von ungenutzten Methoden abhängen

Dependency Inversion Abhängigkeiten auf Abstraktionen, nicht auf Konkretionen

Zusammenhang mit vorherigen Themen

- SRP kennen wir bereits ausführlich
- OCP wird durch Polymorphismus erreicht
- LSP verhindert Refused Bequest
- ISP & DIP reduzieren Kopplung

OCP Definition

“Software-Entitäten sollten offen für Erweiterungen, aber geschlossen für Modifikationen sein.”

```
1 public class AreaCalculator {  
2     public double calculateArea(Shape shape) {  
3         switch (shape.getType()) {  
4             case RECTANGLE:  
5                 Rectangle rect = (Rectangle) shape;  
6                 return rect.getWidth() * rect.getHeight();  
7             case CIRCLE:  
8                 Circle circle = (Circle) shape;  
9                 return Math.PI * circle.getRadius() * circle.getRadius();  
10            default:  
11                throw new IllegalArgumentException("Unknown shape type");  
12            }  
13        }  
14    }
```

Problem

Jede neue Shape-Klasse erfordert eine **Änderung** in der AreaCalculator-Klasse.


```
1 interface Shape {  
2     double calculateArea();  
3 }  
4 class Rectangle implements Shape {  
5     private final double width, height;  
6     // ... constructor ...  
7     @Override  
8     public double calculateArea() { return width * height; }  
9 }  
10 class Circle implements Shape {  
11     private final double radius;  
12     // ... constructor ...  
13     @Override  
14     public double calculateArea() { return Math.PI * radius * radius; }  
15 }  
16 // Neue Shapes können hinzugefügt werden ohne bestehenden Code zu ändern!  
17 class Point implements Shape {  
18     @Override  
19     public double calculateArea() { return 0.0; }  
20 }
```

Vorteil

Das System ist nun offen für Erweiterungen (neue Shapes), aber geschlossen für Modifikationen (bestehender Code muss nicht geändert werden).

```
1 abstract class DataProcessor {
2     // Template Method - definiert Algorithmus-Struktur (geschlossen)
3     public final void processData() {
4         Data data = readData();
5         Data transformed = transformData(data);
6         writeData(transformed);
7     }
8     // Hooks für Subklassen - offen für Erweiterung
9     protected abstract Data readData();
10    protected abstract Data transformData(Data data);
11    protected abstract void writeData(Data data);
12 }
13 class CSVProcessor extends DataProcessor {
14     @Override protected Data readData() { /* CSV-spezifisch */ return null; }
15     @Override protected Data transformData(Data data) { /* ... */ return null; }
16     @Override protected void writeData(Data data) { /* ... */ }
17 }
18 class XMLProcessor extends DataProcessor {
19     @Override protected Data readData() { /* XML-spezifisch */ return null; }
20     @Override protected Data transformData(Data data) { /* ... */ return null; }
21     @Override protected void writeData(Data data) { /* ... */ }
22 }
```

Vorteil: Die Algorithmus-Struktur ist **geschlossen**, die Implementierungsdetails sind **offen** für Erweiterung.

LSP Definition

"Subtypen müssen durch Supertypen ersetzbar sein, ohne die Korrektheit des Programms zu beeinträchtigen."

```
1 class Rectangle {  
2     // ... (Rechteck with width, height und den jeweiligen settern)  
3     public int getArea() { return width * height; }  
4 }  
5  
6 // Test zeigt LSP-Verletzung:  
7 public void testRectangle(Rectangle rect) {  
8     rect.setWidth(5);  
9     rect.setHeight(4);  
10    assert rect.getArea() == 20; // Fails für Square! (=16)  
11 }
```

```
1 class Square extends Rectangle {  
2     @Override  
3     public void setWidth(int width) {  
4         this.width = width;  
5         this.height = width; // Quadrat-Eigenschaft  
6     }  
7     @Override  
8     public void setHeight(int height) {  
9         this.width = height;  
10        this.height = height; // Quadrat-Eigenschaft  
11    }  
12 }
```

Problem

Ein Square verhält sich anders als ein Rectangle erwartet wird. Der Verhaltensvertrag der Superklasse wird gebrochen.

LSP-konforme Lösung durch gemeinsames Interface:

```
1 interface Shape {  
2     int getArea();  
3 }  
4 class Rectangle implements Shape {  
5     private final int width, height;  
6     public Rectangle(int width, int height) {  
7         this.width = width;  
8         this.height = height;  
9     }  
10    public int getArea() { return width * height; }  
11 }  
12 class Square implements Shape {  
13     private final int side;  
14     public Square(int side) { this.side = side; }  
15     public int getArea() { return side * side; }  
16 }
```

Ergebnis

Beide Klassen verhalten sich korrekt als Shape. Es gibt keine vererbte Implementierung, die gebrochen werden kann.

ISP Definition

“Clients sollen nicht von Methoden abhängen, die sie nicht nutzen.”

ISP-Verletzung durch ein zu großes Interface:

```
1 interface Machine {  
2     void print();  
3     void staple();  
4 }  
5  
6 class SimplePrinter implements Machine {  
7     public void print() { /* OK */ }  
8     // UNSINNIG für einen einfachen Drucker!  
9     public void staple() {  
10         throw new UnsupportedOperationException();  
11     }  
12 }
```

Problem

Die SimplePrinter-Klasse wird gezwungen, eine staple-Methode zu implementieren, die sie nicht unterstützt. Dies ist ein Zeichen für ein schlechtes Interface-Design.

ISP-konforme Lösung durch kleine, fokussierte Interfaces:

```
1 interface Printer {  
2     void print();  
3 }  
4 interface Stapler {  
5     void staple();  
6 }  
7 // Implementiert nur, was es kann  
8 class SimplePrinter implements Printer {  
9     @Override public void print() { /* ... */ }  
10 }  
11 // Kombiniert Fähigkeiten nach Bedarf  
12 class MultifunctionPrinter implements Printer, Stapler {  
13     @Override public void print() { /* ... */ }  
14     @Override public void staple() { /* ... */ }  
15 }
```

Vorteil

Klassen implementieren nur die Interfaces, die sie wirklich benötigen. Der Code ist klarer und es gibt keine Notwendigkeit für leere Implementierungen oder Exceptions.

DIP Definition

“Abhängigkeiten sollten auf Abstraktionen zeigen, nicht auf Konkretionen.”

```
1 class EmailService { // Konkrete Implementierung
2     public void sendEmail(String to, String message) {
3         System.out.println("Sending email via SMTP to " + to);
4     }
5 }
6 class NotificationService {
7     private EmailService emailService; // Konkrete Abhängigkeit!
8     public NotificationService() {
9         this.emailService = new EmailService(); // Hard gekoppelt!
10    }
11    public void sendNotification(String user, String message) {
12        emailService.sendEmail(user, message);
13    }
14 }
```

Problem

NotificationService ist direkt an EmailService gekoppelt. Es können keine anderen Benachrichtigungsarten (SMS, Push) verwendet werden und die Klasse ist schwer zu testen.

Dependency Inversion Principle (DIP) - Lösung

```
1 interface MessageSender { // Abstraktion
2     void sendMessage(String to, String message);
3 }
4 class EmailService implements MessageSender {
5     @Override
6     public void sendMessage(String to, String message) {
7         System.out.println("Sending email via SMTP to " + to);
8     }
9 }
10 class SMSService implements MessageSender { // Weitere Implementierung
11     @Override
12     public void sendMessage(String to, String message) {
13         System.out.println("Sending SMS to " + to);
14     }
15 }
16 class NotificationService {
17     private MessageSender messageSender; // Abhängigkeit von Abstraktion!
18     // Abhängigkeit wird von außen injiziert (Dependency Injection)
19     public NotificationService(MessageSender messageSender) {
20         this.messageSender = messageSender;
21     }
22     public void sendNotification(String user, String message) {
23         messageSender.sendMessage(user, message);
24     }
25 }
```


4 Tag 3 - Architektur & Prinzipien

Bausteine & Strukturen

Vererbung & Polymorphismus

SOLID-Prinzipien

Code Smells im Großen

Code Smells “im Großen” betreffen die Struktur zwischen Klassen und Komponenten:

Strukturelle Probleme:

- **Large Class** - Zu viele Verantwortlichkeiten
- **Primitive Obsession** - Fehlende Fachobjekte
- **Data Clumps** - Zusammengehörige Daten getrennt

Änderungs-Probleme:

- **Divergent Change** - Eine Klasse, viele Änderungsgründe
- **Shotgun Surgery** - Eine Änderung, viele Stellen

Kopplungs-Probleme:

- **Feature Envy** - Klasse nutzt andere Klasse zu stark
- **Message Chains** - Lange Aufrufketten

Wichtig für Klausur

- Zusammenhang zu SOLID-Prinzipien verstehen
- Erkennungsmerkmale der verschiedenen Smells
- Wann Behebung sinnvoll ist und wann nicht

Definition: Klassen mit sehr vielen Methoden und Verantwortlichkeiten.

```
1 public class Webcrawler {
2     // Viele verschiedene Datenstrukturen
3     Queue<URL> toVisit; Map<URL, String> visited; Map<String, List<URL>> keywordIndex;
4     Set<String> blockedDomains; HttpClient httpClient;
5
6     // 50+ Methoden für verschiedene Aufgaben:
7     // URL-Management
8     void addUrl(URL url) { /* ... */ }
9     URL nextUrl() { /* ... */ }
10    // HTTP-Operationen
11    String downloadPage(URL url) { /* ... */ }
12    // Content-Parsing
13    List<URL> extractLinks(String content) { /* ... */ }
14    List<String> extractKeywords(String content) { /* ... */ }
15    // Indexing
16    void indexKeywords(String content, URL url) { /* ... */ }
17 }
```

Probleme:

- Verstößt gegen SRP
- Schwer zu verstehen & zu testen
- Hohes Konfliktpotential

Refactoring:

- **Extract Class** (z.B. PageDownloader, LinkExtractor, KeywordIndex)
- **Extract Superclass**

Definition: Verwendung primitiver Datentypen statt spezifischer Fachobjekte.

Problematisches Beispiel:

```
1 class Konto {
2     private final String iban;
3     private final String name;
4
5     public Konto(String name, String iban) {
6         // IBAN-Validierung in Konto-Klasse!
7         if (!gueltigeIBAN(iban)) {
8             throw new IllegalArgumentException();
9         }
10        this.iban = iban;
11        this.name = name;
12    }
13    private static boolean gueltigeIBAN(String iban) {
14        if (iban.length() != 22) return false;
15        // ...
16        return true;
17    }
18 }
```

Verbesserte Lösung:

```
1 record IBAN(String nummer) {
2     public IBAN(String nummer) {
3         if (!isValid(nummer)) {
4             throw new IllegalArgumentException();
5         }
6         this.nummer = nummer.toUpperCase();
7     }
8     private static boolean isValid(String iban) {
9         // ...
10    }
11 }
12 class Konto {
13     private final IBAN iban;
14     private final String name;
15     public Konto(String name, IBAN iban) {
16         this.iban = iban; // Bereits validiert!
17         this.name = name;
18     }
19 }
```

Vorteile: Höhere Kohäsion, Typ-sichere Schnittstellen, fachliche Semantik.

Definition: Mehrere Datenelemente werden immer zusammen verwendet, aber getrennt modelliert.

Problematisches Beispiel:

```
1 public void printStudent(  
2     String name, String email, int punkte) {  
3     // ...  
4 }  
5  
6 public static void main(String[] args) {  
7     // Parallele Arrays - ein Data Clump!  
8     int[] punkte = {79, 49, 90, 33};  
9     String[] namen = {"Susanne", "Domenica", ...};  
10    String[] emails = {"s@hhu.de", "d@hhu.de", ...};  
11  
12    for (int i = 0; i < namen.length; i++) {  
13        if (punkte[i] >= 45) {  
14            System.out.println(namen[i]);  
15        }  
16    }  
17 }
```

Verbesserte Lösung:

```
1 public record TeilnehmerIn(  
2     String name, String email, int punkte) {  
3     public boolean hatBestanden() {  
4         return punkte >= 45;  
5     }  
6 }  
7  
8 public void printStudent(TeilnehmerIn student) {  
9     // ...  
10 }  
11  
12 public static void main(String[] args) {  
13     List<TeilnehmerIn> teilnehmer = List.of(  
14         new TeilnehmerIn("Susanne", "s@hhu.de", 79),  
15         new TeilnehmerIn("Domenica", "d@hhu.de", 49)  
16     );  
17     teilnehmer.stream()  
18         .filter(TeilnehmerIn::hatBestanden)  
19         .forEach(t -> System.out.println(t.name()));  
20 }
```

Definition: Eine Komponente muss aus verschiedenen, unzusammenhängenden Gründen geändert werden.

Problematisches Beispiel - Active Record:

```
1 public class Person {
2     private String firstName, lastName;
3
4     // Geschäftslogik - Änderungsgrund 1
5     public String getFullName() {
6         return firstName + " " + lastName;
7     }
8
9     // Datenbankzugriff - Änderungsgrund 2
10    public void saveToDatabase() {
11        // SQL INSERT statement...
12    }
13    public static Person loadFromDb(int id) {
14        // SQL SELECT statement...
15    }
16 }
```

Bessere Lösung - Trennung:

```
1 // Nur Geschäftslogik
2 public class Person {
3     private final String firstName, lastName;
4     public String getFullName() {
5         return firstName + " " + lastName;
6     }
7 }
8 // Nur Datenzugriff (Repository Pattern)
9 public class PersonRepository {
10    public void save(Person person) {
11        // SQL INSERT statement...
12    }
13    public Person findById(int id) {
14        // SQL SELECT statement...
15    }
16 }
```

Problem: Die ursprüngliche Klasse verletzt das SRP, da sie sich sowohl bei Änderungen der Geschäftslogik als auch bei Änderungen am Datenbankschema ändern muss.

Definition: Eine logische Änderung erfordert viele kleine Änderungen an vielen Stellen im Code.

Problematisches Beispiel - Logging:

```
1 public class OrderService {
2     public void createOrder(Order order) {
3         System.out.println(LocalDate.now() +
4             " [INFO] Creating order: " + order.getId());
5         // ...
6     }
7 }
8
9 public class PaymentService {
10     public void processPayment(Payment p) {
11         System.out.println(LocalDate.now() +
12             " [INFO] Processing payment: " + p.getId());
13         // ...
14     }
15 }
16 // Problem: Log-Format ändern = ALLE Klassen ändern!
```

Verbesserte Lösung - Zentrale Abstraktion:

```
1 public class Logger { // Zentrale Logging-Klasse
2     public static void info(String message) {
3         System.out.println(LocalDate.now() +
4             " [INFO] " + message);
5     }
6 }
7 public class OrderService {
8     public void createOrder(Order order) {
9         Logger.info("Creating order: " + order.getId());
10        // ...
11    }
12 }
13 public class PaymentService {
14     public void processPayment(Payment p) {
15         Logger.info("Processing payment: " + p.getId());
16        // ...
17    }
18 }
```

Regel: Wenn "Suchen & Ersetzen" häufig für eine Änderung benötigt wird, ist das ein Hinweis auf Shotgun Surgery.

Definition: Eine Methode ist mehr an einer anderen Klasse interessiert als an ihrer eigenen.

Problematisches Beispiel:

```
1 public class Monitor {
2     public void sendeAlarm(String msg) { /*...*/ }
3     public int getSystemTemperatur() { /*...*/ }
4     public boolean isCpuFanRunning() { /*...*/ }
5     public int getMemoryUsage() { /*...*/ }
6 }
7
8 public class RechnerKontrolle {
9     public void healthCheck(Monitor monitor) {
10         // Feature Envy - viele Zugriffe auf Monitor!
11         if (monitor.getSystemTemperatur() > 93) {
12             monitor.sendeAlarm("CPU überhitzt");
13         }
14         if (!monitor.isCpuFanRunning() &&
15             monitor.getSystemTemperatur() > 85) {
16             monitor.sendeAlarm("CPU-Lüfter ausgefallen");
17         }
18     }
19 }
```

Verbesserte Lösung (Move Method):

```
1 public class Monitor {
2     public void sendeAlarm(String msg) { /*...*/ }
3     private int getSystemTemperatur() { /*...*/ }
4     private boolean isCpuFanRunning() { /*...*/ }
5     private int getMemoryUsage() { /*...*/ }
6     // Logik ist jetzt in der Klasse, zu der sie gehört
7     public void healthCheck() {
8         if (getSystemTemperatur() > 93) {
9             sendeAlarm("CPU überhitzt");
10        }
11        if (!isCpuFanRunning() &&
12            getSystemTemperatur() > 85) {
13            sendeAlarm("CPU-Lüfter ausgefallen");
14        }
15    }
16 }
17 public class RechnerKontrolle {
18     public void performHealthCheck(Monitor monitor) {
19         // "Tell, don't ask" - keine Feature Envy mehr!
20         monitor.healthCheck();
21     }
22 }
```

Definition: Lange Aufrufketten, die das Law of Demeter verletzen.

Problematisches Beispiel:

```
1 public class Rechnung {
2     public double berechneGesamtsumme() {
3         double summe = 0.0;
4         for (Position pos : positionen) {
5             // Message Chain:
6             summe += pos.getAnzahl() *
7                     pos.getProdukt().getPreis();
8         }
9         return summe;
10    }
11 }
12 // Noch schlimmer:
13 public void printCustomerCity(Order order) {
14     String city = order.getCustomer()
15                     .getAddress()
16                     .getCity()
17                     .getName();
18     System.out.println(city);
19 }
```

Verbesserte Lösung (Hide Delegate):

```
1 public class Position {
2     public double getGesamtpreis() {
3         return anzahl * produkt.getPreis();
4     }
5 }
6
7 public class Rechnung {
8     public double berechneGesamtsumme() {
9         return positionen.stream()
10            .mapToDouble(Position::getGesamtpreis)
11            .sum(); // Keine Chain mehr!
12    }
13 }
14
15 public class Order {
16     public String getCustomerCityName() {
17         return customer.getCityName(); // Delegieren
18     }
19 }
```

Fluent Interfaces sind KEINE Message Chains! (Selbes Objekt in allen Aufrufen)

Nicht jeder Code Smell muss automatisch behoben werden!

Entscheidungskriterien:

Wahrscheinlichkeit zukünftiger Änderungen Wie wahrscheinlich ist es, dass dieser Code-Teil geändert werden muss?

Trade-off-Analyse

- Entstehen durch die Behebung neue, schlimmere Probleme?
- Wird der Code komplexer oder einfacher?

Kontext berücksichtigen

- Manche Smells gehören zu bestimmten Patterns (z.B. Strategy → Feature Envy).
- Einmalige Scripts vs. langlebige Software.
- Prototypen vs. Produktionscode.

Pragmatismus

Ziel: Wartbarkeit verbessern, nicht perfekte Software erschaffen. Manchmal ist “Good Enough” wirklich gut genug!

Wichtiger Hinweis zur folgenden Folien

- Diese Zuordnungen sind **meine persönliche Interpretation**
- Nicht als absolute Wahrheit verstehen!
- Nur verwenden, wenn ihr **keine Ahnung habt** und besser als 50/50 raten wollt
- Kann als **Denkanstoß** dienen, um in die richtige Richtung zu kommen
- Maximal dann blind vertrauen, wenn das Ziel eine 4.0 ist

Diese Tabelle ist ein Notfallplan, kein Masterplan!

Code Smell	Verletztes Prinzip	Erklärung
Mehrere Verantwortlichkeiten	Single Responsibility	Ursache für viele andere Smells
Long Method	Single Responsibility	Mehrere Verantwortlichkeiten vermischt
Long Parameter List	Single Responsibility	Viele Parameter = oft multiple Aufgaben
Mysterious Name	Allgemeine Lesbarkeit	Behindert Verständlichkeit
Duplicated Code	DRY Principle	Gleiche Logik → verteilte Änderungen
SLAP-Verletzung	Single Level of Abstraction	Verschiedene Detailebenen vermischt

Prüfungsstrategie (basiert auf Altklausur-Analyse)

- 1 **Primitive Obsession** - Domain Types statt primitiver Typen
- 2 **SRP-Verletzungen** - Methoden/Klassen mit mehreren Aufgaben
- 3 **SLAP-Verletzungen** - Gemischte Abstraktionsebenen
- 4 **DRY-Verletzungen** - Duplizierte Geschäftslogik

Selten direkt geprüft: Long Method, Long Parameter List, Mysterious Name

Code Smell	Verletztes Prinzip	Erklärung
Large Class	Single Responsibility	Zu viele verschiedene Verantwortlichkeiten
Primitive Obsession	Single Responsibility	Validierung gehört in Fachobjekt
Data Clumps	High Cohesion	Zusammengehörige Daten getrennt
Divergent Change	Single Responsibility	Klasse ändert sich aus verschiedenen Gründen
Shotgun Surgery	DRY Principle	Oft (nicht immer) mit Code-Duplikation
Feature Envy	Law of Demeter / Tell, don't ask	Klasse nutzt andere mehr als sich selbst
Message Chains	Law of Demeter	Lange Aufrufketten: <code>a.getB().getC()</code>
Refused Bequest	Liskov Substitution / ISP	Subtyp nicht als Supertyp verwendbar

Prüfungsstrategie (basiert auf Altklausur-Analyse)

- 1 **Shotgun Surgery** - Eine Änderung betrifft viele Klassen
- 2 **Primitive Obsession/Data Clumps** - Domain Types fehlen bzw. Parameter-Gruppen wie Adressdaten existieren
- 3 **Large Class/SRP-Verletzung** - Mehrere Verantwortlichkeiten

Selten direkt geprüft: Feature Envy, Refused Bequest, Divergent Change

Code Smell	Erkennungsmerkmale
Mehrere Verantwortlichkeiten	Klasse/Methode hat mehrere Aufgaben, "und" im Namen
Long Method	Methode > 15 Zeilen, mehrere Abstraktionsebenen vermischt
Long Parameter List	> 3 Parameter, oft Hinweis auf SRP-Verletzung
Mysterious Name	Unklare Namen wie d, data, process, doStuff
Duplicated Code	Copy-Paste Code, gleiche Geschäftslogik mehrfach implementiert
SLAP-Verletzung	Verschiedene Detailebenen gemischt (Low-Level + High-Level)
Verdächtige Kommentare	"Der folgende Code tut xy", Entschuldigungen im Code
Large Class	Mehrere verschiedene Aufgaben, Zweck schwer in einem Satz beschreibbar
Primitive Obsession	String statt IBAN, int statt Temperatur, fehlende Fachobjekte
Data Clumps	Parallele Arrays, Parameter-Gruppen die immer zusammen auftreten
Divergent Change	"Klasse ändert sich, wenn..." hat mehr als eine Antwort
Shotgun Surgery	Eine logische Änderung betrifft > 3 Klassen/Stellen
Feature Envy	Methode nutzt andere Klasse intensiver als die eigene
Message Chains	a.getB().getC().getD() (NICHT Fluent APIs wie Stream)
Refused Bequest	UnsupportedOperationException in Override-Methoden

Struktur-Prinzipien:

- **Single Responsibility** - Ein Grund zur Änderung
 - **Information Hiding** - Implementierung verstecken
 - **High Cohesion** - Zusammengehöriges zusammen
 - **Low Coupling** - Abhängigkeiten minimieren
 - **Law of Demeter** - Nur mit Freunden sprechen
- Trade-offs bei Architektur-Entscheidungen bewerten

Vererbung & Polymorphismus:

- Interface-Vererbung > Klassen-Vererbung
- Komposition > Vererbung
- Polymorphismus zur Entkopplung
- Refused Bequest vermeiden

SOLID-Prinzipien:

- **Single Responsibility Principle** - *Nochmal*
 - **Open/Closed** - Erweiterbar, nicht änderbar
 - **Liskov Substitution** - Subtypen austauschbar
 - **Interface Segregation** - Kleine, fokussierte Interfaces
 - **Dependency Inversion** - Auf Abstraktionen setzen
- Die jeweiligen Refactoring Strategien wissen

Code Smells erkennen:

- Large Class, Primitive Obsession
- Data Clumps, Divergent Change
- Shotgun Surgery, Feature Envy
- Message Chains

Was haben wir (theoretisch) gelernt?

Technische Skills:

- Moderne Java-Features
- Funktionale Programmierung
- Testing mit TDD
- Git Workflow
- Spring Framework

Software-Design:

- Code Smells erkennen
- SOLID Prinzipien
- Clean Code
- Domain-Driven Design
- Refactoring

Empfohlene Ressourcen

- Clean Code - Robert C. Martin
- Effective Java - Joshua Bloch
- Spring Boot Documentation
- Git Pro Book (kostenlos online)

Vielen Dank für eure Aufmerksamkeit!

Fragen?