Programmierpraktikum 1

Nachklausurtutorium Sommersemester 2025

Paul Christian Dötsch

Institut für Informatik
Heinrich-Heine-Universität Düsseldorf

18. September 2025

Überblick

Heinrich Heine
Universität Düsseldorf

Nachklausurtutorium Programmierpraktikum 1

Heinrich Heine Universität Düsseldorf

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-propra1-ss25/nachklausurtutorium`
- **GitHub Organisation:** `https://github.com/hhu-propra1-ss25/Organisation`
- **RocketChat:** `https://rocketchat.hhu.de/group/Propra-SoSe-25-NKT`

Tagesplanung - Überblick

Heinrich Heine Universität Düsseldorf

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

Was diese Folien sind

Heinrich Heine Universität Düsseldorf

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 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

Wichtiger Hinweis

hhu
Heinrich Heine
Universität Düsseldorf

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!

Überblick - Java-Basics & Klassenbibliothek

Heinrich Heine
Universität Düsseldorf

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()`

List Interface - Grundlagen

Heinrich Heine Universität Düsseldorf

Eigenschaften:

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

Wichtige Methoden:

```
get(int index)
set(int index, E element)
add(int index, E element)
remove(int index)
indexOf(Object o)
lastIndexOf(Object o)
```

Beispiel:

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

ArrayList vs LinkedList

Heinrich Heine Universität Düsseldorf

ArrayList

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

```
List<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(2);
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)

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

Set Interface - Einzigartige Elemente

Heinrich Heine Universität Düsseldorf

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
```

Set Implementierungen im Detail

Heinrich Heine Universität Düsseldorf

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 Interface - FIFO Prinzip

Heinrich Heine Universität Düsseldorf

Queue-Operationen:

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

```java
Queue<String> q = new LinkedList<>();
q.offer("first");
q.offer("second");
q.offer("third");

System.out.println(q.peek()); // "first"
System.out.println(q.poll()); // "first"
System.out.println(q.poll()); // "second"
```

Deque (Double Ended Queue):

```java
Deque<Integer> deque = new ArrayDeque<>();
deque.offerFirst(1);  // [1]
deque.offerLast(2);   // [1, 2]
deque.offerFirst(0);  // [0, 1, 2]

// Stack-Operationen:
deque.push(5);        // [5, 0, 1, 2]
Integer top = deque.pop(); // 5
```

Map Interface - Key-Value Paare

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.getOrDefault("Charlie", 0)); // 0
8 System.out.println(ages.containsKey("Bob")); // true
```

Map-Operationen im Detail

Heinrich Heine Universität Düsseldorf

Grundoperationen:

```java
Map<String, Integer> map = new HashMap<>();

// Einfügen und Abrufen
map.put("key1", 100);
Integer value = map.get("key1");

// Sichere Operationen
map.putIfAbsent("key2", 200);
Integer safe = map.getOrDefault("key3", -1);

// Prüfungen
boolean hasKey = map.containsKey("key1");
boolean hasValue = map.containsValue(100);
```

Iteration über Maps:

```java
// Über Schlüssel iterieren
for (String key : map.keySet()) {
    System.out.println(key + ": " + map.get(key)
        );
}

// Über Werte iterieren
for (Integer value : map.values()) {
    System.out.println(value);
}

// Über Entry-Paare iterieren
for (Map.Entry<String, Integer> entry : map.
     entrySet()) {
    System.out.println(
      entry.getKey() + " = " + entry.getValue()
    );
}
```

Comparable vs Comparator

Heinrich Heine Universität Düsseldorf

Comparable Interface

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

```java
class Student implements Comparable<Student> {
    private String name;
    private int grade;

    @Override
    public int compareTo(Student other) {
        // ACHTUNG: Overflow-Gefahr!
        // return this.grade - other.grade;

        // BESSER:
        return Integer.compare(this.grade, other.grade);
    }
}
```

Comparator Interface

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

```java
// Als separate Klasse
class ReverseOrder implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2.compareTo(o1);
    }
}

// Verwendung
TreeSet<Integer> reversed = new TreeSet<>(new ReverseOrder());

// Mit Lambda (später mehr dazu)
TreeSet<Integer> desc = new TreeSet<>((a, b) -> b.compareTo(a));
```

Collections Utility-Klasse

Heinrich Heine Universität Düsseldorf

Wichtige statische Methoden:

```
List<String> list = new ArrayList<>(
    List.of("c", "a", "b"));

// Sortieren
Collections.sort(list);         // [a, b, c]

// Umdrehen
Collections.reverse(list);      // [c, b, a]

// Mischen
Collections.shuffle(list);      // zufällige Reihenfolge

// Rotieren
Collections.rotate(list, 1);    // [?, c, b] (je nach shuffle)
```

```
// Füllen
Collections.fill(list, "x");    // [x, x, x]

// Min/Max finden
List<Integer> numbers = List.of(3, 1, 4, 1, 5);
Integer min = Collections.min(numbers); // 1
Integer max = Collections.max(numbers); // 5

// Häufigkeit zählen
int count = Collections.frequency(numbers, 1); // 2

// Unveränderliche Views
List<String> immutable = Collections.unmodifiableList(list);
```

Optionals - Einführung

Problem mit null-Werten:

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

Optional als Lösung:

```java
// Traditionell - fehleranfällig
public String findUserName(int id) {
    User user = database.findUser(id);
    if (user != null) {
        return user.getName();
    }
    return null; // Könnte vergessen werden!
}
```

```java
// Mit Optional - explizit
public Optional<String> findUserName(int id) {
    Optional<User> user = database.findUser(id);
    return user.map(User::getName);
}

// Verwendung
String name = findUserName(123)
    .orElse("Unknown User");
```

Sichtbarkeiten in Java

Modifier	Klasse	Package	Subklasse	Überall
`private`	✓			
package-private	✓	✓		
`protected`	✓	✓	✓	
`public`	✓	✓	✓	✓

Besonderheit: private zwischen Instanzen

```java
class Outer {
    private int value;

    class Inner {
        void method(Outer other) {
            other.value = 42;  // OK! Private zwischen Instanzen derselben Klasse
        }
    }
}
```

Überblick - Generics

Heinrich Heine
Universität Düsseldorf

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!
```

Statische Typisierung mit Generics

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

```
ArrayList<String> list = new ArrayList<>();  // Diamond Operator (Java 7+)
list.add("String");
list.add(42);  // Kompilier-Fehler!

String s = list.get(0);  // Kein Cast nötig
```

Pattern Matching mit instanceof (Java 17+)

Heinrich Heine Universität Düsseldorf

Traditionell:

```java
if (obj instanceof String) {
    String str = (String) obj;
    System.out.println(str.toUpperCase());
}

if (obj instanceof Integer) {
    Integer num = (Integer) obj;
    System.out.println(num * 2);
}
```

Mit Pattern Matching:

```java
if (obj instanceof String str) {
    System.out.println(str.toUpperCase());
}

if (obj instanceof Integer num) {
    System.out.println(num * 2);
}

// Auch in switch möglich (Java 21+)
```

Generische Datentypen definieren

Heinrich Heine Universität Düsseldorf

Einfacher generischer Typ:

```java
public class Box<T> {
    private T content;

    public Box(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }

    public void setContent(T content) {
        this.content = content;
    }
}
```

Mehrere Typparameter:

```java
public class Pair<T, U> {
    private final T first;
    private final U second;

    public Pair(T first, U second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() { return first; }
    public U getSecond() { return second; }
}

// Verwendung:
Pair<String, Integer> nameAge =
    new Pair<>("Alice", 25);
```

Type Constraints mit extends

Heinrich Heine Universität Düsseldorf

Einfacher Constraint:

```java
// Nur Number und Subtypen erlaubt
public class NumberBox<T extends Number> {
    private T value;

    public NumberBox(T value) {
        this.value = value;
    }

    public double getDoubleValue() {
        return value.doubleValue(); // Möglich wegen Number
    }
}

// Verwendung:
NumberBox<Integer> intBox = new NumberBox<>(42);
NumberBox<String> strBox = new NumberBox<>("text"); // Fehler!
```

Self-referencing Constraint:

```java
// T muss mit sich selbst vergleichbar sein
public class SortedList<T extends Comparable<T>> {
    private List<T> elements = new ArrayList<>();

    public void add(T element) {
        elements.add(element);
        Collections.sort(elements); // Funktioniert!
    }

    public T getMin() {
        return Collections.min(elements);
    }
}

// Verwendung:
SortedList<String> strings = new SortedList<>();
SortedList<Object> objects = new SortedList<>(); // Fehler!
```

Generische Methoden

Heinrich Heine Universität Düsseldorf

```java
public class ArrayUtils {

    // Generische statische Methode
    public static <T> List<T> arrayToList(T[] array) {
        List<T> list = new ArrayList<>();
        for (T element : array) {
            list.add(element);
        }
        return list;
    }

    // Mit Constraint
    public static <T extends Comparable<T>> T findMax(T[] array) {
        if (array.length == 0) {
            throw new IllegalArgumentException("Array empty");
        }

        T max = array[0];
        for (T element : array) {
            if (element.compareTo(max) > 0) {
                max = element;
            }
        }
        return max;
    }
}
```

Verwendung:

```java
// Type Inference - Compiler leitet Typen ab
String[] names = {"Alice", "Bob", "Charlie"};
List<String> nameList = ArrayUtils.arrayToList(names);

Integer[] numbers = {3, 1, 4, 1, 5};
List<Integer> numberList = ArrayUtils.arrayToList(numbers);

// Explizite Typangabe möglich
List<String> explicit = ArrayUtils.<String>arrayToList(names);

// findMax Beispiel
String longest = ArrayUtils.findMax(names); // "Charlie"
Integer biggest = ArrayUtils.findMax(numbers); // 5
```

Überblick - Funktionale Programmierung

Heinrich Heine Universität Düsseldorf

2. Tag 1 - Grundlagen & Werkzeuge
   - Java-Basics & Klassenbibliothek
   - Generics
   - Funktionale Programmierung
   - Streams
   - Werkzeuge für Softwareentwicklung

Funktionale Interfaces - Überblick

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
`Consumer<T>`	`T → void`	Seiteneffekt ausführen
`Supplier<T>`	`() → T`	Wert erzeugen
`Function<T,R>`	`T → R`	Transformation
`Predicate<T>`	`T → boolean`	Bedingung prüfen
`BiFunction<T,U,R>`	`(T,U) → R`	Zwei-Parameter Funktion

Consumer<T> - Seiteneffekte

Heinrich Heine Universität Düsseldorf

Nimmt einen Parameter, gibt nichts zurück:

```java
// Consumer definieren
Consumer<String> printer = s -> System.out.println(s);
Consumer<String> upperPrinter = s -> System.out.println(s.
      toUpperCase());

// Verwenden
printer.accept("Hello World");
upperPrinter.accept("Hello World");

// Mit forEach
List<String> words = List.of("Java", "Python", "JavaScript");
words.forEach(printer);
words.forEach(upperPrinter);
```

```java
// Consumer verketten mit andThen
Consumer<String> print = s -> System.out.print(s);
Consumer<String> newline = s -> System.out.println();

Consumer<String> printWithNewline = print.andThen(newline);
printWithNewline.accept("Hello");

// Komplexeres Beispiel
Consumer<Person> emailSender = person ->
    emailService.sendWelcome(person.getEmail());
Consumer<Person> logger = person ->
    log.info("Processed: " + person.getName());

Consumer<Person> processNewUser = emailSender.andThen(logger);
```

Supplier<T> - Werte erzeugen

Nimmt keine Parameter, gibt einen Wert zurück:

```java
// Einfache Supplier
Supplier<Double> randomValue = () -> Math.random();
Supplier<String> greeting = () -> "Hello World";
Supplier<LocalDateTime> currentTime = LocalDateTime::now;

// Verwenden
double rand = randomValue.get();
String msg = greeting.get();
LocalDateTime now = currentTime.get();
```

```java
// Lazy Initialization
public class ExpensiveObject {
    private Supplier<String> lazyValue = () -> {
        System.out.println("Computing expensive value...");
        return "Expensive Result";
    };

    public String getValue() {
        return lazyValue.get();
    }
}

// Factory Pattern
Supplier<List<String>> listFactory = ArrayList::new;
List<String> newList = listFactory.get();
```

Function<T,R> - Transformationen

Heinrich Heine Universität Düsseldorf

Nimmt einen Parameter, gibt transformierten Wert zurück:

```java
// Einfache Funktionen
Function<String, Integer> length = s -> s.length();
Function<String, String> toUpper = s -> s.toUpperCase();
Function<Integer, Integer> square = x -> x * x;

// Verwenden
Integer len = length.apply("Hello");  // 5
String upper = toUpper.apply("hello"); // "HELLO"
Integer squared = square.apply(4);    // 16
```

```java
// Funktionen verketten
Function<String, String> normalize = s -> s.trim().toLowerCase();
Function<String, Integer> wordCount = s -> s.split("\\s+").length;

Function<String, Integer> countNormalized =
    normalize.andThen(wordCount);

Integer count = countNormalized.apply("  Hello World  "); // 2

// Compose (umgekehrte Richtung)
Function<String, Integer> lengthOfUpper =
    length.compose(toUpper);
// Entspricht: s -> length.apply(toUpper.apply(s))
```

Predicate<T> - Bedingungen prüfen

Heinrich Heine Universität Düsseldorf

Nimmt einen Parameter, gibt boolean zurück:

```java
// Einfache Prädikate
Predicate<String> isEmpty = s -> s.isEmpty();
Predicate<String> isLong = s -> s.length() > 10;
Predicate<Integer> isEven = n -> n % 2 == 0;
Predicate<Integer> isPositive = n -> n > 0;

// Verwenden
boolean empty = isEmpty.test("");      // true
boolean long = isLong.test("Short");   // false
boolean even = isEven.test(42);        // true
```

```java
// Prädikate kombinieren
Predicate<Integer> isPositiveEven = isPositive.and(isEven);
Predicate<Integer> isNegativeOrOdd = isPositive.negate().or(isEven.
      negate());

List<Integer> numbers = List.of(-2, -1, 0, 1, 2, 3, 4);

// Mit filter verwenden
List<Integer> positiveEvens = numbers.stream()
    .filter(isPositiveEven)
    .toList(); // [2, 4]

// Statische Utility-Methoden
Predicate<Object> isNull = Objects::isNull;
Predicate<String> isEqual = Predicate.isEqual("test");
```

Lambda-Ausdrücke - Syntax

Heinrich Heine Universität Düsseldorf

Verschiedene Syntaxformen:

Parameter-Syntax:

```
// Mit Typdeklaration
Consumer<String> c1 = (String s) -> System.out.println(s);

// Ohne Typdeklaration (Type Inference)
Consumer<String> c2 = (s) -> System.out.println(s);

// Ohne Klammern (bei einem Parameter)
Consumer<String> c3 = s -> System.out.println(s);

// Mehrere Parameter
BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;

// Keine Parameter
Supplier<String> greeting = () -> "Hello";
```

Body-Syntax:

```
// Expression (kein return nötig)
Function<Integer, Integer> square = x -> x * x;

// Statement Block (return erforderlich)
Function<String, String> process = s -> {
    String trimmed = s.trim();
    String upper = trimmed.toUpperCase();
    return upper;
};

// Void Block
Consumer<String> debug = s -> {
    System.out.println("Processing: " + s);
    // Kein return
};
```

Methodenreferenzen

Heinrich Heine Universität Düsseldorf

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;
```

Funktionen höherer Ordnung

Heinrich Heine Universität Düsseldorf

Methoden, die Funktionen als Parameter nehmen oder zurückgeben:

forEach implementieren:

```java
public class MyList<T> {
    private List<T> elements = new ArrayList<>();

    public void forEach(Consumer<T> action) {
        for (T element : elements) {
            action.accept(element);
        }
    }

    public void addIf(T element, Predicate<T> condition) {
        if (condition.test(element)) {
            elements.add(element);
        }
    }
}
```

Funktionen zurückgeben:

```java
public class FunctionFactory {

    public static Predicate<Integer> greaterThan(int threshold) {
        return value -> value > threshold;
    }

    public static Function<String, String> addPrefix(String prefix)
            {
        return text -> prefix + text;
    }
}

// Verwendung
Predicate<Integer> greaterThan10 = FunctionFactory.greaterThan(10);
Function<String, String> addHello = FunctionFactory.addPrefix("
      Hello, ");

boolean result = greaterThan10.test(15); // true
String greeting = addHello.apply("World"); // "Hello, World"
```

Überblick - Streams

Heinrich Heine
Universität Düsseldorf

2. Tag 1 - Grundlagen & Werkzeuge
   - Java-Basics & Klassenbibliothek
   - Generics
   - Funktionale Programmierung
   - Streams
   - Werkzeuge für Softwareentwicklung

Stream-API Motivation

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:

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

Stream API - Überblick

Heinrich Heine Universität Düsseldorf

Stream erzeugen:

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

Filtern und Transformieren:

```
stream.filter(x -> x > 5);
stream.map(String::toUpperCase);
stream.mapToInt(String::length);
stream.flatMap(s -> Arrays.stream(s.split(",")));
stream.distinct();
stream.sorted();
stream.sorted(Comparator.comparing(Person::getName));
stream.limit(10);
stream.skip(5);
stream.peek(System.out::println);
stream.takeWhile(x -> x < 100);
stream.dropWhile(x -> x < 10);
stream.mapToObj(i -> "Item " + i);
stream.mapToDouble(account -> account.getBalance());
```

Aggregation:

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

Sammeln:

```
stream.toList();
stream.collect(Collectors.toSet());
stream.collect(Collectors.joining(", "));
stream.collect(Collectors.groupingBy(String::length));
stream.collect(Collectors.partitioningBy(x -> x > 0));
stream.collect(Collectors.counting());
stream.collect(Collectors.summarizingInt(String::length));
```

Prüfen und Finden:

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

Streams erzeugen

Heinrich Heine Universität Düsseldorf

Aus Collections:

```
List<String> words = List.of("Java", "Python", "C++");
Stream<String> wordStream = words.stream();

Set<Integer> numbers = Set.of(1, 2, 3, 4, 5);
Stream<Integer> numberStream = numbers.stream();

// Parallele Streams
Stream<String> parallelStream = words.parallelStream();
```

Aus Arrays:

```
String[] array = {"a", "b", "c"};
Stream<String> arrayStream = Arrays.stream(array);

int[] intArray = {1, 2, 3, 4, 5};
IntStream intStream = Arrays.stream(intArray);
```

Statische Factory-Methoden:

```
// Endliche Streams
Stream<String> of = Stream.of("a", "b", "c");
Stream<Integer> empty = Stream.empty();

// Unendliche Streams
Stream<Double> randoms = Stream.generate(Math::random);
Stream<Integer> naturals = Stream.iterate(0, n -> n + 1);

// Mit Bedingung (Java 9+)
Stream<Integer> limited = Stream.iterate(0, n -> n < 100, n -> n +
      2);
```

Primitive Streams:

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

Intermediate Operations

Heinrich Heine Universität Düsseldorf

Geben einen neuen Stream zurück - lazy evaluation:

Filterung und Transformation:

```
List<String> words = List.of("Java", "Python", "JavaScript", "C++")
      ;

Stream<String> filtered = words.stream()
    .filter(s -> s.length() > 4)
    .map(String::toUpperCase)
    .distinct();

// Noch nicht ausgeführt!
```

FlatMap - Streams "flach klopfen":

```
List<List<String>> nested = List.of(
    List.of("a", "b"),
    List.of("c", "d"),
    List.of("e")
);

List<String> flattened = nested.stream()
    .flatMap(Collection::stream)
    .toList(); // [a, b, c, d, e]
```

Limiting und Skipping:

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

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

Sortierung:

```
List<String> sorted = words.stream()
    .sorted()                          // Natural order
    .toList();

List<String> byLength = words.stream()
    .sorted(Comparator.comparing(String::length))
    .toList();

List<String> reversed = words.stream()
    .sorted(Comparator.reverseOrder())
    .toList();
```

Terminal Operations

Heinrich Heine Universität Düsseldorf

Beenden die Stream-Pipeline und lösen Berechnung aus:

Sammeln:

```java
List<String> words = List.of("Java", "Python", "C++");

List<String> list = words.stream()
    .map(String::toUpperCase)
    .toList();

Set<String> set = words.stream()
    .collect(Collectors.toSet());

String joined = words.stream()
    .collect(Collectors.joining(", "));
// "Java, Python, C++"
```

Aggregation:

```java
List<Integer> numbers = List.of(1, 2, 3, 4, 5);

long count = numbers.stream().count();
Optional<Integer> max = numbers.stream().max(Integer::compareTo);
Optional<Integer> min = numbers.stream().min(Integer::compareTo);
```

Matching:

```java
boolean anyEven = numbers.stream()
    .anyMatch(n -> n % 2 == 0);    // true

boolean allPositive = numbers.stream()
    .allMatch(n -> n > 0);         // true

boolean noneNegative = numbers.stream()
    .noneMatch(n -> n < 0);        // true
```

Finden:

```java
Optional<String> first = words.stream()
    .filter(s -> s.startsWith("J"))
    .findFirst();                  // Optional["Java"]

Optional<String> any = words.stream()
    .filter(s -> s.startsWith("P"))
    .findAny();                    // Optional["Python"]
```

Reduce-Operationen

Heinrich Heine Universität Düsseldorf

Reduziert Stream auf einen einzigen Wert:

Mit Initialwert:

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);

// Summe
Integer sum = numbers.stream()
    .reduce(0, (a, b) -> a + b);        // 15

// Oder kürzer mit Integer::sum
Integer sum2 = numbers.stream()
    .reduce(0, Integer::sum);           // 15

// Produkt
Integer product = numbers.stream()
    .reduce(1, (a, b) -> a * b);        // 120

// String konkatenation
String concatenated = List.of("A", "B", "C").stream()
    .reduce("", (a, b) -> a + b);       // "ABC"
```

Ohne Initialwert (Optional):

```
Optional<Integer> sum = numbers.stream()
    .reduce((a, b) -> a + b);

Optional<Integer> max = numbers.stream()
    .reduce(Integer::max);

// Komplexeres Beispiel
Optional<String> longest = List.of("Java", "Python", "C++").stream
    ()
    .reduce((s1, s2) -> s1.length() >= s2.length() ? s1 : s2);
// Optional["Python"]
```

Primitive Streams:

```
IntStream.range(1, 6)
    .sum();                             // 15

OptionalDouble average = IntStream.range(1, 6)
    .average();                         // OptionalDouble[3.0]
```

Collectors - Vorgefertigte Sammler

Heinrich Heine Universität Düsseldorf

Basis-Collectors:

```java
List<String> words = List.of("Java", "Python", "Java", "C++");

List<String> toList = words.stream()
    .collect(Collectors.toList());

Set<String> toSet = words.stream()
    .collect(Collectors.toSet());

Map<String, Integer> toMap = words.stream()
    .distinct()
    .collect(Collectors.toMap(
        Function.identity(),     // Key: das Wort selbst
        String::length           // Value: Länge des Worts
    ));
```

Statistiken:

```java
long count = words.stream()
    .collect(Collectors.counting());

IntSummaryStatistics stats = words.stream()
    .collect(Collectors.summarizingInt(String::length));
```

Gruppierung:

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

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

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

Primitive Streams - Performance

Heinrich Heine Universität Düsseldorf

Spezialisierte Streams für bessere Performance:

IntStream, LongStream, DoubleStream:

```java
// Vermeidet Boxing/Unboxing
IntStream numbers = IntStream.range(1, 1000000);

int sum = numbers.sum();                    // Optimiert
OptionalDouble avg = numbers.average();
IntSummaryStatistics stats = numbers.summaryStatistics();

// Von Stream<Integer> zu IntStream
List<Integer> integers = List.of(1, 2, 3, 4, 5);
int sumFromStream = integers.stream()
    .mapToInt(Integer::intValue)
    .sum();

// Von IntStream zu Stream<Integer>
Stream<Integer> boxed = IntStream.range(1, 10)
    .boxed();
```

Praktisches Beispiel:

```java
// Summe aller geraden Quadrate von 1-100
int result = IntStream.rangeClosed(1, 100)
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .sum();

// Mit Double-Werten
double[] values = {1.1, 2.2, 3.3, 4.4, 5.5};
OptionalDouble max = Arrays.stream(values)
    .max();

// String-Längen als IntStream
String[] words = {"Java", "Python", "C++"};
IntSummaryStatistics lengthStats = Arrays.stream(words)
    .mapToInt(String::length)
    .summaryStatistics();
```

Parallele Streams

Heinrich Heine Universität Düsseldorf

Automatische Parallelisierung für bessere Performance:

Erzeugung:

```java
// Aus Collections
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
Stream<Integer> parallel = numbers.parallelStream();

// Aus sequenziellen Streams
Stream<Integer> sequential = numbers.stream();
Stream<Integer> parallelized = sequential.parallel();

// Zurück zu sequenziell
Stream<Integer> backToSeq = parallel.sequential();
```

Wann sinnvoll:

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

Wichtige Einschränkungen:

```java
// GEFÄHRLICH - Race Condition!
List<String> result = new ArrayList<>();
words.parallelStream()
    .forEach(word -> result.add(word.toUpperCase())); // FEHLER!

// SICHER - Verwendung von Collectors
List<String> safe = words.parallelStream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());    // OK!

// GEFÄHRLICH - Shared Mutable State
AtomicInteger counter = new AtomicInteger(0);
numbers.parallelStream()
    .forEach(n -> counter.incrementAndGet()); // Problematisch
```

Regeln:

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

Überblick - Werkzeuge für Softwareentwicklung

Heinrich Heine Universität Düsseldorf

2. Tag 1 - Grundlagen & Werkzeuge
   - Java-Basics & Klassenbibliothek
   - Generics
   - Funktionale Programmierung
   - Streams
   - Werkzeuge für Softwareentwicklung

Gradle - Build Automation Tool

Heinrich Heine Universität Düsseldorf

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

Gradle Projektstruktur

Standard-Layout (Convention over Configuration):

```
myproject/
|-- build.gradle
|-- settings.gradle
|-- gradle/
|   +-- wrapper/
|-- gradlew         (Unix)
|-- gradlew.bat     (Windows)
+-- src/
    |-- main/
    |   |-- java/
    |   |   +-- com/example/
    |   |       +-- Main.java
    |   +-- resources/
    +-- test/
        +-- java/
            +-- com/example/
                +-- 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)

Gradle Build-Datei Beispiel

Heinrich Heine
Universität Düsseldorf

```
plugins {
    id 'java'                    // Java-Plugin für Kompilierung
    id 'application'             // Anwendungs-Plugin für run-Task
}

repositories {
    mavenCentral()              // Standard Java-Repository
}

dependencies {
    // Produktions-Dependencies
    implementation 'com.google.guava:guava:33.1.0-jre'
    implementation 'org.apache.commons:commons-collections4:4.4'

    // Test-Dependencies
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.12.2'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.12.2'
    testImplementation 'org.assertj:assertj-core:3.27.3'
}

application {
    mainClass = 'com.example.Main'  // Hauptklasse für run-Task
}

test {
    useJUnitPlatform()          // JUnit 5 verwenden
}
```

Wichtige Gradle Tasks

Heinrich Heine Universität Düsseldorf

Build-Tasks:

```
gradle compile          # Nur kompilieren
gradle assemble         # JAR erstellen (ohne Tests)
gradle build            # Vollständiger Build mit Tests
gradle clean            # Build-Verzeichnis löschen
gradle jar              # JAR-Datei erstellen
```

Run-Tasks:

```
gradle run              # Anwendung ausführen
gradle test             # Tests ausführen
gradle distZip          # ZIP-Distribution erstellen
gradle installDist      # Lokale Installation
```

Task-Abkürzungen:

```
gradle b                # build
gradle cJ               # compileJava
gradle dZ               # distZip
gradle iD               # installDist
```

Info-Tasks:

```
gradle tasks            # Alle verfügbaren Tasks
gradle dependencies     # Dependency-Tree anzeigen
gradle projects         # Multi-Project Info
gradle properties       # Projekt-Properties
```

Tipp: Gradle Wrapper verwenden

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

Packages und Classpath

Heinrich Heine Universität Düsseldorf

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

JAR-Dateien

Heinrich Heine Universität Düsseldorf

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

JAR erstellen:

```
# Einfache JAR
jar cf myapp.jar *.class

# Mit Manifest
jar cfm myapp.jar MANIFEST.MF *.class

# Mit Gradle
gradle jar  # Erstellt JAR in build/libs/
```

JAR ausführen:

```
# Mit Klassenname
java -cp myapp.jar com.example.Main

# Ausführbare JAR (mit Main-Class im Manifest)
java -jar myapp.jar
```

MANIFEST.MF Beispiel:

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

JAR inspizieren:

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

Git - Versionskontrolle Grundlagen

Heinrich Heine Universität Düsseldorf

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

Git Basis-Workflow

Heinrich Heine Universität Düsseldorf

Projekt starten:

```
# Neues Repository
git init

# Existierendes Repository klonen
git clone https://github.com/user/repo.git

# Status anzeigen
git status
```

Änderungen committen:

```
# Dateien zur Staging Area hinzufügen
git add file1.java file2.java
git add .                    # Alle Änderungen

# Commit erstellen
git commit -m "Add new feature"

# Add und Commit kombiniert (nur für tracked files)
git commit -am "Fix bug in calculation"
```

Mit Remote arbeiten:

```
# Änderungen hochladen
git push

# Änderungen herunterladen
git pull

# Remote Repository hinzufügen
git remote add origin https://github.com/user/repo.git
git push -u origin main
```

Historie anzeigen:

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

Gute Commit-Messages

Anatomie einer guten Commit-Message

```
Kurze Zusammenfassung (< 50 Zeichen)

Detaillierte Beschreibung, wenn nötig.
Erkläre WARUM, nicht WAS.
Verwende Imperativ: "Add" statt "Added".

- Kann Aufzählungen enthalten
- 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)

.gitignore Datei

```
# Build-Ordner
build/
target/
out/

# IDE-Dateien
.idea/
.vscode/

# OS-spezifische Dateien
.DS_Store
Thumbs.db
*~

# Sensitive Daten
.env
config/local.properties
```

Patterns:

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

Überblick - Testing Basics

Heinrich Heine Universität Düsseldorf

3. Tag 2 - Testing & Codequalität
   - Testing Basics
   - TDD-Zyklus
   - Qualität von Software
   - Code Smells im Kleinen

Warum Unit Testing?

Heinrich Heine
Universität Düsseldorf

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

AAA-Schema - Test-Struktur

Heinrich Heine Universität Düsseldorf

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

```java
@Test
@DisplayName("Addition zweier positiver Zahlen")
void should_add_two_positive_numbers() {
    // Arrange
    Calculator calc = new Calculator();
    int a = 5, b = 3;

    // Act
    int result = calc.add(a, b);

    // Assert
    assertThat(result).isEqualTo(8);
}
```

FIRST-Prinzipien für gute Tests

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 $\rightarrow$ 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)

Test-Setup mit JUnit 5 und AssertJ

Heinrich Heine Universität Düsseldorf

Dependencies in build.gradle:

```
dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.12.2'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.12.2'
    testImplementation 'org.assertj:assertj-core:3.27.3'
}

test {
    useJUnitPlatform()
    testLogging {
        events "passed", "skipped", "failed"
    }
}
```

Imports in Test-Klassen:

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

Rückgaben testen - Pure Functions

Heinrich Heine Universität Düsseldorf

Pure Function: Ergebnis nur von Eingabeparametern abhängig

Eigenschaften:

- Keine Seiteneffekte
- Deterministisch
- Einfach zu testen
- Wiederverwendbar

```java
// Pure Function
public static double calculateCircleArea(double radius) {
    return Math.PI * radius * radius;
}

// Nicht pure (Seiteneffekt)
public void logAndCalculate(double radius) {
    System.out.println("Calculating...");  // Seiteneffekt!
    return Math.PI * radius * radius;
}
```

Test für Pure Function:

```java
@Test
@DisplayName("Kreisfläche für Radius 5")
void should_calculate_circle_area_for_radius_5() {
    // Act
    double area = MathUtils.calculateCircleArea(5.0);

    // Assert
    assertThat(area).isCloseTo(78.54, offset(0.01));
}

@Test
@DisplayName("Kreisfläche für Radius 0")
void should_return_zero_area_for_zero_radius() {
    // Act
    double area = MathUtils.calculateCircleArea(0.0);

    // Assert
    assertThat(area).isEqualTo(0.0);
}
```

Testen mit Zustand

Heinrich Heine Universität Düsseldorf

Counter-Klasse:

```java
public class Counter {
    private int count = 0;

    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }

    public void reset() {
        count = 0;
    }
}
```

Test-Klasse:

```java
public class CounterTest {
    private Counter counter = new Counter();
    @Test
    @DisplayName("Counter startet bei 0")
    void should_start_at_zero() {
        assertThat(counter.getCount()).isEqualTo(0);
    }
    @Test
    @DisplayName("Reset setzt auf 0 zurück")
    void should_reset_to_zero() {
        // Arrange
        counter.increment();
        // Act
        counter.reset();
        // Assert
        assertThat(counter.getCount()).isEqualTo(0);
    }
}
```

JUnit erzeugt für jeden Test eine neue Instanz!

Tests sind automatisch isoliert - kein geteilter Zustand.

AssertJ - Fluent Assertions

Heinrich Heine Universität Düsseldorf

Lesbare Assertions im natürlichen Sprachfluss:

Zahlen:

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

Strings:

```
assertThat("Hello World").isEqualTo("Hello World");
assertThat("Java").contains("av");
assertThat("").isEmpty();
assertThat("NotEmpty").isNotEmpty();
assertThat("Hello").startsWith("Hel");
assertThat("World").endsWith("rld");
assertThat("HELLO").isEqualToIgnoringCase("hello");
```

Collections:

```
List<String> list = List.of("a", "b", "c");

assertThat(list).hasSize(3);
assertThat(list).contains("b");
assertThat(list).doesNotContain("d");
assertThat(list).containsExactly("a", "b", "c");
assertThat(list).containsExactlyInAnyOrder("c", "a", "b");
assertThat(list).startsWith("a");
assertThat(list).isNotEmpty();
```

Booleans und null:

```
assertThat(true).isTrue();
assertThat(false).isFalse();
assertThat(object).isNull();
assertThat(object).isNotNull();
assertThat(object).isInstanceOf(String.class);
```

Testen auf Exceptions

Heinrich Heine Universität Düsseldorf

Exception-Tests mit assertThrows:

Einfacher Exception-Test:

```java
@Test
@DisplayName("Division durch Null wirft ArithmeticException")
void should_throw_exception_for_division_by_zero() {
    // Arrange
    Calculator calc = new Calculator();

    // Act & Assert
    ArithmeticException exception = assertThrows(
        ArithmeticException.class,
        () -> calc.divide(10, 0)
    );

    // Optional: Exception-Details prüfen
    assertThat(exception.getMessage())
        .contains("by zero");
}
```

Warum Lambda-Ausdruck?

```java
// FALSCH - Exception fliegt vor assertThrows
@Test
void wrong_exception_test() {
    Calculator calc = new Calculator();
    int result = calc.divide(10, 0);  // Exception hier!

    assertThrows(ArithmeticException.class,
        () -> { /* code never reached */ });
}

// RICHTIG - Code wird erst in Lambda ausgeführt
@Test
void correct_exception_test() {
    Calculator calc = new Calculator();

    assertThrows(ArithmeticException.class,
        () -> calc.divide(10, 0));  // Exception hier gefangen
}
```

Überblick - TDD-Zyklus

Heinrich Heine
Universität Düsseldorf

3. Tag 2 - Testing & Codequalität
   - Testing Basics
   - TDD-Zyklus
   - Qualität von Software
   - Code Smells im Kleinen

Test-Driven Development (TDD)

Heinrich Heine Universität Düsseldorf

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-Green-Refactor Zyklus

Heinrich Heine Universität Düsseldorf

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

TDD Beispiel: Roman Numbers (1/4)

Heinrich Heine Universität Düsseldorf

Anforderung: Arabische Zahlen in römische Zahlen umwandeln

1. RED - Erster fehlschlagender Test:

```java
@Test
@DisplayName("1 wird zu I")
void should_translate_1_to_I() {
    // Arrange & Act
    String result = RomanNumbers.translate(1);
    // Assert
    assertThat(result).isEqualTo("I");
}
```

Fehler: `RomanNumbers` Klasse existiert nicht → Kompiliert nicht

2. GREEN - Minimaler Code:

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

Test wird grün ✓

TDD Beispiel: Roman Numbers (2/4)

Heinrich Heine Universität Düsseldorf

3. REFACTOR - Noch nichts zu refactorieren

4. RED - Zweiter Test:

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

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

5. GREEN - Code erweitern:

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

Beide Tests grün ✓

TDD Beispiel: Roman Numbers (3/4)

Heinrich Heine Universität Düsseldorf

6. RED - Dritter Test:

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

7. GREEN - Pattern wird sichtbar:

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

8. REFACTOR - Code vereinfachen:

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

TDD Beispiel: Roman Numbers (4/4)

Heinrich Heine Universität Düsseldorf

9. RED - Nächster Test (Regel für 4):

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

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

10. GREEN - Spezialfall behandeln:

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

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!

Überblick - Qualität von Software

Heinrich Heine
Universität Düsseldorf

3. Tag 2 - Testing & Codequalität
   - Testing Basics
   - TDD-Zyklus
   - Qualität von Software
   - Code Smells im Kleinen

ISO 25010 - Software Quality Model

Heinrich Heine Universität Düsseldorf

- **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

Trade-offs zwischen Qualitätszielen 1

Heinrich Heine Universität Düsseldorf

Beispiel: Sicherheit vs. Usability

Sicherheit erhöhen:

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

Usability sinkt:

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

Trade-offs zwischen Qualitätszielen 2

Heinrich Heine Universität Düsseldorf

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

Wartbarkeit - Warum so wichtig?

hhu
Heinrich Heine
Universität Düsseldorf

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"

Eigenschaften wartbarer Software

Heinrich Heine Universität Düsseldorf

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

Wartbarkeit als Investition

Heinrich Heine Universität Düsseldorf

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

Überblick - Code Smells im Kleinen

Heinrich Heine
Universität Düsseldorf

3. Tag 2 - Testing & Codequalität
   - Testing Basics
   - TDD-Zyklus
   - Qualität von Software
   - Code Smells im Kleinen

Was sind Code Smells?

Heinrich Heine Universität Düsseldorf

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

Mehrere Verantwortlichkeiten 1

Heinrich Heine Universität Düsseldorf

Schlecht - Mars-Roboter Beispiel:

```java
public void controlRobot() {
    // Sensordaten lesen
    int distance = sensor.readDistance();
    boolean obstacle = distance < 10;

    // Route planen
    if (obstacle) {
        turnLeft();
        moveForward(5);
        turnRight();
    } else {
        moveForward(10);
    }

    // Energieverbrauch prüfen
    if (battery.getLevel() < 20) {
        sendLowBatteryAlert();
        activatePowerSaving();
    }

    // Daten komprimieren und senden
    String data = collectSensorData();
    String compressed = compress(data);
    radio.sendToEarth(compressed);
}
```

Besser - Aufgeteilt:

```java
public void controlRobot() {
    Environment env = sensorsystem.scan();
    Route route = navigator.planRoute(env);
    movement.execute(route);
    powerManager.checkAndOptimize();
    dataTransmitter.sendToEarth();
}

// Separate Klassen/Methoden:
class SensorSystem {
    public Environment scan() { /* ... */ }
}

class Navigator {
    public Route planRoute(Environment env) { /* ... */ }
}

class MovementController {
    public void execute(Route route) { /* ... */ }
}

class PowerManager {
    public void checkAndOptimize() { /* ... */ }
}
```

Mehrere Verantwortlichkeiten 2

Heinrich Heine
Universität Düsseldorf

Vorteile der Aufteilung:

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

Mysterious Name / Namenskonventionen

Heinrich Heine Universität Düsseldorf

Schlechte Namen:

```java
// Was macht diese Methode?
public static int fibo(int n) {
    int y = 0, z = 0;
    while(y < n) {
        z += 1 + 2*y++;
    }
    return z;
}

// Lügender Name!
private Set<String> kundenListe;

// Uninformativ
int d;
d += 30;
```

Gute Namen:

```java
// Klarerer Name enthüllt: es ist n^2!
public static int calculateSquareOfNumber(int n) {
    int result = 0;
    int iterator = 0;
    while(iterator < n) {
        result += 1 + 2 * iterator;
        iterator++;
    }
    return result;
}

// Wahrheitsgemäss
private Set<String> kunden;

// Aussagekräftig
int daysSinceLastUpdate;
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)

Java Namenskonventionen

Heinrich Heine Universität Düsseldorf

camelCase - Methoden und Variablen:

```java
// Methoden: Verben
public void calculateInterest()
public String getUserName()
public boolean isValid()
public void setCustomerAddress()

// Variablen: Nomen
String firstName;
int customerAge;
List<Order> pendingOrders;
boolean isReadyForProcessing;
```

PascalCase - Klassen und Interfaces:

```java
// Klassen: Nomen
public class CustomerManager
public class OrderProcessor
public class PaymentGateway

// Interfaces: oft Adjektive mit -able
public interface Serializable
public interface Comparable<T>
public interface Runnable
```

SCREAMING_SNAKE_CASE - Konstanten:

```java
public static final int MAX_RETRY_COUNT = 3;
public static final String DEFAULT_ENCODING = "UTF-8";
public static final double PI = 3.14159;

// Enums
public enum Status {
    PENDING,
    IN_PROGRESS,
    COMPLETED,
    CANCELLED
}
```

Boolean-Naming:

```java
// Präfixe: is, has, can, should
boolean isValid;
boolean hasPermission;
boolean canEdit;
boolean shouldRetry;

// Methoden genauso
public boolean isEmpty()
public boolean hasNext()
public boolean canAccess()
```

Kommentare als Code Smell

Heinrich Heine Universität Düsseldorf

Schlecht - Beschreibt WAS:

```
// Prüfen ob erste Argument gültige Mailadresse ist
if (Pattern.matches("^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0
        -9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\\.[a-zA-Z0-9](?:[a-
        zA-Z0-9-]{0,61}[a-zA-Z0-9])?)*$", args[0])) {
    // ...
}

// i um 1 erhöhen
i++;

// Schleife über alle Kunden
for (Customer customer : customers) {
    // ...
}
```

Besser - Code selbsterklärend machen:

```
// Statt Kommentar: Methode mit sprechendem Namen
if (isValidEmailAddress(args[0])) {
    // ...
}

private static boolean isValidEmailAddress(String email) {
    // Vereinfachter RegEx aus HTML5 Standard,
    // da RFC-konformer RegEx zu komplex für unseren Use-Case
    return Pattern.matches("^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@...",
            email);
}

// Keine Kommentare nötig:
customerIndex++;

for (Customer customer : customers) {
    processCustomerOrder(customer);
}
```

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

Long Method 1

Heinrich Heine
Universität Düsseldorf

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)

Long Method 2

Lösung: Extract Method

```java
// Vorher: Eine lange Methode mit 50+ Zeilen
public void processOrder(Order order) {
    // 10 Zeilen Validierung
    // 15 Zeilen Preisberechnung
    // 10 Zeilen Rabatt-Logik
    // 15 Zeilen E-Mail versenden
}

// Nachher: Aufgeteilt in kleinere Methoden
public void processOrder(Order order) {
    validateOrder(order);
    double total = calculateTotal(order);
    sendConfirmation(order, total);
}
```

SLAP - Single Level of Abstraction Principle

Heinrich Heine Universität Düsseldorf

Alle Anweisungen in einer Methode sollten den gleichen Detailgrad haben

Verletzt SLAP (Detail + Abstraktion):

```java
private void printReport() {
    // Hohe Abstraktion
    printHeader();

    // Niedrige Abstraktion (Details)
    int maxLength = 0;
    for (Product product : products) {
        int length = product.getName().length();
        if (length > maxLength) {
            maxLength = length + 1;
        }
    }

    // Wieder hohe Abstraktion
    printProductTable(maxLength);
    printFooter();
}
```

Erfüllt SLAP (gleiche Abstraktionsebene):

```java
private void printReport() {
    printHeader();
    int maxLength = calculateMaxProductNameLength();
    printProductTable(maxLength);
    printFooter();
}

private int calculateMaxProductNameLength() {
    int maxLength = 0;
    for (Product product : products) {
        int length = product.getName().length();
        if (length > maxLength) {
            maxLength = length + 1;
        }
    }
    return maxLength;
}
```

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

Long Parameter List

Heinrich Heine Universität Düsseldorf

Problem: Methoden mit zu vielen Parametern

Problematisch:

```
// Was bedeuten die Parameter?
public double calculateDistance(
    double x1, double y1, double z1,
    double x2, double y2, double z2) {
    return Math.sqrt(
        Math.pow(x2-x1,2) + Math.pow(y2-y1,2) + Math.pow(z2-z1,2)
    );
}

// Aufruf - fehleranfällig!
double dist = calculateDistance(
    1.0, 2.0, 3.0,    // Start
    4.0, 5.0, 6.0     // End - oder umgekehrt?
);
```

Besser - Parameter-Objekte:

```
public record Point3D(double x, double y, double z) {}

public double calculateDistance(Point3D start, Point3D end) {
    return Math.sqrt(
        Math.pow(end.x() - start.x(), 2) +
        Math.pow(end.y() - start.y(), 2) +
        Math.pow(end.z() - start.z(), 2)
    );
}

// Aufruf - selbsterklärend!
Point3D start = new Point3D(1.0, 2.0, 3.0);
Point3D end = new Point3D(4.0, 5.0, 6.0);
double distance = calculateDistance(start, end);
```

Boolean-Parameter vermeiden:

```
// Schlecht: Was bedeutet true/false?
public Package wrap(Product product, boolean isGift) { ... }

// Besser: Separate Methoden
public Package wrapAsGift(Product product) { ... }
public Package wrapNormally(Product product) { ... }
```

Duplicated Code / DRY-Prinzip 1

Heinrich Heine Universität Düsseldorf

Don't Repeat Yourself (DRY)

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

Problematisch - Gleiche Geschäftslogik:

```java
// Rabatt-Berechnung in OrderService
public double calculateOrderTotal(Order order) {
    double total = order.getSubtotal();
    if (order.getCustomer().isPremium()) {
        total *= 0.9;  // 10% Rabatt
    }
    return total;
}

// Gleiche Logik in InvoiceService
public double calculateInvoiceTotal(Invoice invoice) {
    double total = invoice.getAmount();
    if (invoice.getCustomer().isPremium()) {
        total *= 0.9;  // 10% Rabatt - DUPLIZIERT!
    }
    return total;
}
```

OK - Verschiedenes Wissen (trotz gleichem Code):

```java
// Alter validieren
public boolean validateAge(int age) {
    return age >= 0 && age <= 150;
}

// Menge validieren
public boolean validateQuantity(int quantity) {
    return quantity >= 0 && quantity <= 150;
}

// Gleicher Code, aber verschiedene Geschäftsregeln!
// Alter-Obergrenze könnte sich unabhängig von
// Mengen-Obergrenze ändern
```

Duplicated Code / DRY-Prinzip 2

Don't Repeat Yourself (DRY)
Vorteile:

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

Überblick - Bausteine & Strukturen

Heinrich Heine
Universität Düsseldorf

4. Tag 3 - Architektur & Prinzipien
   - Bausteine & Strukturen
   - Vererbung & Polymorphismus
   - SOLID-Prinzipien
   - Code Smells im Großen

Wartbare Software-Architektur

Heinrich Heine Universität Düsseldorf

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

LCHC: Das Fundament wartbarer Architektur

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.

Single Responsibility Prinzip (SRP) - Der Weg zu hoher Kohäsion

Heinrich Heine Universität Düsseldorf

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?

SRP Beispiel - Employee-Klasse Problem

Heinrich Heine Universität Düsseldorf

Problem - Mehrere Verantwortlichkeiten in einer Klasse:

```java
public class Employee {
    private String name;
    private double baseSalary;
    private List<EmployeeId> manages;

    // Lohnbuchhaltung
    public double calculateSalary() {
        return baseSalary + managementBonus();
    }
    private double managementBonus() {
        return manages.size() * 500.0;
    }

    // Datenbank-Persistierung
    public void save() {
        Connection conn = DriverManager.getConnection(...);
        PreparedStatement stmt = conn.prepareStatement(...);
        stmt.setDouble(1, calculateSalary());
        stmt.executeUpdate();
    }

    // Berichtswesen
    public String generateReport() {
        return String.format("Employee: %s, Salary: %.2f",
                             name, calculateSalary());
    }
}
```

SRP Beispiel - Analyse der Verantwortlichkeiten

Heinrich Heine Universität Düsseldorf

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

SRP Beispiel - Lösung durch Aufspaltung

Heinrich Heine Universität Düsseldorf

Geschäftslogik:

```java
public class Employee {
    private final String name;
    private final double baseSalary;
    private final List<EmployeeId> manages;
    public double calculateSalary() {
        return baseSalary + manages.size() * 500.0;
    }
    public String getName() { return name; }
}
```

Persistierung:

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

Berichtswesen:

```java
public class EmployeeReporter {
    public String generateReport(Employee employee) {
        return String.format(
            "Employee: %s\n" +
            "Salary: %.2f EUR\n" +
            "Status: %s",
            employee.getName(),
            employee.calculateSalary(),
            getStatus(employee));
    }
    private String getStatus(Employee emp) {
        // Report-spezifische Logik
        return "Active";
    }
}
```

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

Information Hiding Prinzip (IHP) - Definition 1

Heinrich Heine Universität Düsseldorf

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

Information Hiding Prinzip (IHP) - Definition 2

Heinrich Heine
Universität Düsseldorf

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

IHP Beispiel - Point-Klasse Evolution

Heinrich Heine Universität Düsseldorf

Schlecht - Keine Kapselung:

```java
public class Point {
    public double x;
    public double y;
}

// Client-Code:
Point p = new Point();
p.x = 3.0;
p.y = 4.0;
double distance = Math.sqrt(p.x * p.x + p.y * p.y);
```

Problem: Interne Darstellung ist fest verdrahtet!

Gut - Mit Kapselung:

```java
public class Point {
    private final double x;
    private final double y;

    private Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public static Point fromCartesian(double x, double y) {
        return new Point(x, y);
    }

    public double getX() { return x; }
    public double getY() { return y; }

    public double getRadius() {
        return Math.sqrt(x*x + y*y);
    }
}

// Client-Code:
Point p = Point.fromCartesian(3.0, 4.0);
double distance = p.getRadius();
```

IHP Beispiel - Umstellung auf Polarkoordinaten

Heinrich Heine Universität Düsseldorf

Jetzt kann die interne Darstellung geändert werden:

```java
public class Point {
    private final double radius;  // Jetzt polar!
    private final double theta;

    private Point(double radius, double theta) {
        this.radius = radius;
        this.theta = theta;
    }

    public static Point fromCartesian(double x, double y) {
        return new Point(Math.hypot(x, y), Math.atan2(y, x));
    }

    public double getX() { return radius * Math.cos(theta); }
    public double getY() { return radius * Math.sin(theta); }
    public double getRadius() { return radius; }
}
```

Vorteil

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

Zerlegungsstrategien: Fachlich vs. Technisch

Heinrich Heine Universität Düsseldorf

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.

Zerlegung: Visualisiert

Heinrich Heine Universität Düsseldorf

```java
// Benutzer - UI
class BenutzerUI {
    void anlegen() {
        String name = readInput();
        benutzerLogik.neuerBenutzer(name);
        showMessage("Benutzer erstellt");
    }
    void anzeigen() {
        benutzerLogik.alleBenutzer();
    }
}
```

```java
// Artikel - UI
class ArtikelUI {
    void anlegen() {
        String name = readInput();
        artikelLogik.neuerArtikel(name);
        showMessage("Artikel erstellt");
    }
    void suchen() {
        artikelLogik.findeArtikel();
    }
}
```

```java
// Benutzer - Logik
class BenutzerLogik {
    Benutzer neuerBenutzer(String name) {
        Benutzer b = new Benutzer(name);
        database.speichereBenutzer(b);
        return b;
    }
    List<Benutzer> alleBenutzer() {
        return database.ladeAlleBenutzer();
    }
}
```

```java
// Artikel - Logik
class ArtikelLogik {
    Artikel neuerArtikel(String name) {
        Artikel a = new Artikel(name);
        database.speichereArtikel(a);
        return a;
    }
    List<Artikel> findeArtikel() {
        return database.ladeAlleArtikel();
    }
}
```

Vertikal trennen bzw. Horizontal zusammenfassen → UI-Schicht + Logik-Schicht
Horizontal trennen bzw. Vertikal zusammenfassen → Benutzer-Modul + Artikel-Modul

Kopplung verstehen - Arten und Stärken

Heinrich Heine Universität Düsseldorf

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 → 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!

Kopplung durch Verletzung des IHP (1/2)

Heinrich Heine Universität Düsseldorf

Extrem problematisch - Geteilter, änderbarer Zustand:

```java
public class Interpolation {
    private final int[] array;

    public Interpolation(int[] eingabe) {
        Arrays.sort(eingabe);  // sortiert eingabe in-place!
        this.array = eingabe;  // Referenz geteilt!
    }

    public int search(int value) {
        return Arrays.binarySearch(array, value);
    }
}

// Client-Code kann array zerstören:
int[] foo = new int[] {1, -2, 3, 7};
Interpolation ip = new Interpolation(foo);

// Später im Code:
for(int i=0; i < foo.length; i++) {
    foo[i] = new Random().nextInt();  // Zerstört Sortierung!
}
// ip.search() funktioniert nicht mehr korrekt!
```

Kopplung durch Verletzung des IHP (2/2)

Heinrich Heine Universität Düsseldorf

Die Lösung - Defensive Kopie:

```java
public class Interpolation {
    private final int[] array;

    public Interpolation(int[] eingabe) {
        // Defensive Kopie erstellen
        this.array = Arrays.copyOf(eingabe, eingabe.length);
        Arrays.sort(this.array);  // Jetzt sicher!
    }

    public int search(int value) {
        return Arrays.binarySearch(array, value);
    }

    // Auch bei Rückgaben defensive Kopie:
    public int[] getArray() {
        return Arrays.copyOf(array, array.length);
    }
}
```

Wichtig

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

Law of Demeter - “Sprich nur mit direkten Freunden”

Heinrich Heine Universität Düsseldorf

Law of Demeter Regel

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

Verletzung - Rechnung kennt Produkt:

```java
class Rechnung {
    double berechneZeilenpreis(Position pos) {
        // BAD: Rechnung kennt Position UND Produkt
        return pos.getAnzahl() * pos.getProdukt().getPreis();
    }
}
class Position {
    private int anzahl;
    private Produkt produkt;
    public int getAnzahl() { return anzahl; }
    public Produkt getProdukt() { return produkt; }
}
class Produkt {
    private double preis;
    public double getPreis() { return preis; }
}
```

Korrekt - Position kapselt Details:

```java
class Rechnung {
    double berechneZeilenpreis(Position pos) {
        // GOOD: Rechnung kennt nur Position
        return pos.getZeilenpreis();
    }
}
class Position {
    private int anzahl;
    private Produkt produkt;
    public int getAnzahl() { return anzahl; }
    // Position kapselt Produktdetails
    public double getZeilenpreis() {
        return anzahl * produkt.getPreis();
    }
}
class Produkt {
    private double preis;
    public double getPreis() { return preis; }
}
```

Kopplung durch Aufruf und Konstruktion

Heinrich Heine Universität Düsseldorf

Problem - Hart verdrahtet:

```java
class CountingSummarizer {
    private Segmentation segmentation;

    public CountingSummarizer() {
        // Hard gekoppelt!
        this.segmentation = new AgeSegmentation();
    }

    public Map<String, Integer> summarize(List<Customer> customers)
            {
        Map<String, List<Customer>> segments =
            segmentation.getSegments(customers);

        return segments.entrySet().stream()
            .collect(toMap(
                Map.Entry::getKey,
                e -> e.getValue().size()));
    }
}
```

Problem: Kann nicht getestet oder erweitert werden!

Lösung - Dependency Injection:

```java
interface Segmentation {
    Map<String, List<Customer>> getSegments(
        List<Customer> customers);
}

class CountingSummarizer {
    private final Segmentation segmentation;

    // Injiziert via Konstruktor
    public CountingSummarizer(
            Segmentation segmentation) {
        this.segmentation = segmentation;
    }

    public Map<String, Integer> summarize(
            List<Customer> customers) {
        Map<String, List<Customer>> segs =
            segmentation.getSegments(customers);

        return segs.entrySet().stream()
            .collect(toMap(
                Map.Entry::getKey,
                e -> e.getValue().size()));
    }
}
```

Kohäsion verstehen - Das interne Bindemittel

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?

Kohäsion Beispiel - Schlechte Kohäsion

Heinrich Heine Universität Düsseldorf

Problem - Customer mit verschiedenen Verantwortlichkeiten:

```java
class Customer {
    // Verschiedene Attributsätze
    List<Order> orders;            // Order-Management
    String emailAddress;           // Kommunikation
    String name;                   // Basis-Daten
    String shippingAddress;        // Versand

    // Methoden greifen auf verschiedene Attributsätze zu
    List<Order> getOrders() { // -> orders
        return orders;
    }

    void sendWelcomeEmail() { // -> emailAddress, name
        EmailService.send(emailAddress, "Welcome " + name + "!");
    }

    void printShippingLabel() { // -> name, shippingAddress
        System.out.println("Ship to: " + name);
        System.out.println("Address: " + shippingAddress);
    }
}
```

Problem: Niedrige Kohäsion - jede Methode nutzt andere Attribute!

Kohäsion Beispiel - LCHC durch Aufspaltung erreichen

Heinrich Heine Universität Düsseldorf

```java
class Customer {
    private final String name;
    private final CustomerId id;
    public String getName() { return name; }
    public CustomerId getId() { return id; }
    // ...
}

class OrderManager {
    private final List<Order> orders;
    private final CustomerId customerId;
    public void addOrder(Order order) {
        orders.add(order);
    }
    public List<Order> getOrderHistory() {
        return List.copyOf(orders);
    }
    public double getTotalOrderValue() {
        return orders.stream()
            .mapToDouble(Order::getValue)
            .sum();
    }
}
```

```java
class NotificationService {
    private final String emailAddress;
    private final String customerName;
    public void sendWelcomeEmail() {
        EmailService.send(emailAddress,
            "Welcome " + customerName + "!");
    }
    public void sendOrderConfirmation(Order o) {
        EmailService.send(emailAddress,
            "Order " + o.getId() + " confirmed");
    }
}
class ShippingService {
    private final String customerName;
    private final Address shippingAddress;
    public void printShippingLabel() {
        System.out.println("Ship to: " + customerName);
        System.out.println("Address: " + shippingAddress);
    }
    public double calculateShippingCost() {
        return shippingAddress.getShippingZone().getBaseCost();
    }
}
```

LCHC → Code Smells: Erkennungsmuster

Heinrich Heine Universität Düsseldorf

Viele Code Smells verletzen das LCHC Prinzip!

Code Smell	Kohäsion	Kopplung	Erkennungsmuster
Large Class	niedrig	-	Klasse hat mehrere 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	`a.getB().getC().getD()`
Shotgun Surgery	-	hoch	Eine Änderung betrifft > 3 Klassen

Template für Refactoring-Aufgaben

1. Code Smell identifizieren
2. LCHC-Verletzung und ggf. Prinzip-Verletzung benennen
3. Lösung: Extract Class / Move Method / Hide Delegate / etc.

Von Prinzipien zu praktischer Anwendung

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

Überblick - Vererbung & Polymorphismus

Heinrich Heine
Universität Düsseldorf

4. Tag 3 - Architektur & Prinzipien
   - Bausteine & Strukturen
   - Vererbung & Polymorphismus
   - SOLID-Prinzipien
   - Code Smells im Großen

Arten des Polymorphismus in Java

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 vs. Overriding

Overloading (Überladen):

```java
public class Calculator {
    // Verschiedene add-Methoden
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }

    public String add(String a, String b) {
        return a + b;
    }

    // Compiler wählt basierend auf Parametern
}

// Verwendung:
Calculator calc = new Calculator();
int result1 = calc.add(5, 3);         // int-Variante
double result2 = calc.add(5.5, 3.2); // double-Variante
String result3 = calc.add("Hi", "!"); // String-Variante
```

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

Overloading vs. Overriding

Heinrich Heine Universität Düsseldorf

Overriding (Überschreiben):

```java
abstract class Animal {
    public abstract String makeSound();
    public void sleep() {
        System.out.println("Zzz...");
    }
}
class Dog extends Animal {
    @Override
    public String makeSound() {
        return "Woof!";
    }
}
class Cat extends Animal {
    @Override
    public String makeSound() {
        return "Meow!";
    }
}

// Verwendung:
Animal animal = new Dog();  // Polymorphie
System.out.println(animal.makeSound()); // "Woof!" - Runtime
```

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

Interface-Vererbung vs. Klassen-Vererbung

Heinrich Heine Universität Düsseldorf

Interface-Vererbung (BEVORZUGT):

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

```java
interface Drawable { void draw(); }
interface Movable { void move(int dx, int dy); }

class GameCharacter
        implements Drawable, Movable {
    @Override
    public void draw() { /* ... */ }
    @Override
    public void move(int dx, int dy) { /* ... */ }
}
```

Klassen-Vererbung (VORSICHTIG):

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

```java
abstract class Vehicle {
    protected int speed = 0;
    public void accelerate() {
        speed += 10;
    }
    public abstract void startEngine();
}

class Car extends Vehicle {
    @Override
    public void startEngine() { /* ... */ }
    public void brake() {
        speed -= 15;
    }
}
```

Polymorphismus zur Entkopplung - Vorher

Heinrich Heine Universität Düsseldorf

```java
public class Employee {
    private boolean management;
    private List<EmployeeId> manages;
    private Salary baseSalary;
    public Salary computeSalary() {
        if (management) {
            return baseSalary.addManagementBonus(
                manages.size());
        } else {
            return baseSalary;
        }
    }
    public String getDescription() {
        if (management) {
            return "Manager with " + manages.size() +
                   " reports";
        } else {
            return "Regular employee";
        }
    }
}
```

Problem

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

Polymorphismus zur Entkopplung - Nachher

Heinrich Heine Universität Düsseldorf

```java
interface Employee {
    Salary computeSalary();
    String getDescription();
}
class RegularEmployee implements Employee {
    private final Salary baseSalary;
    public Salary computeSalary() { return baseSalary; }
    public String getDescription() { return "Regular employee"; }
}
class Manager implements Employee {
    private final Salary baseSalary;
    private final List<EmployeeId> manages;
    public Salary computeSalary() {
        return baseSalary.addManagementBonus(manages.size());
    }
    public String getDescription() {
        return "Manager with " + manages.size() + " reports";
    }
}
```

Vorteil

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

Kopplung durch Vererbung - InstrumentedHashSet Problem

Heinrich Heine Universität Düsseldorf

```java
public class InstrumentedHashSet<E> extends HashSet<E> {
    private int addCount = 0;
    @Override
    public boolean add(E e) {
        addCount++;
        return super.add(e);
    }
    @Override
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();  // PROBLEM: Doppeltes Zählen!
        return super.addAll(c);
    }
    public int getAddCount() { return addCount; }
}
```

Folgender Test schlägt fehl:

```java
@Test
void testAddCount() {
    InstrumentedHashSet<Integer> set = new InstrumentedHashSet<>();
    set.addAll(List.of(1, 2, 3));

    // Erwartet: 3, Tatsächlich: 6
    assertThat(set.getAddCount()).isEqualTo(3);
}
```

Warum schlägt er fehl?

- `HashSet.addAll()` ruft intern `add()` auf
- Unsere `add()`-Methode wird aufgerufen → Jedes Element wird zweimal gezählt
- **Kopplung an interne Implementierung!**
- Bricht Kapselung und erzeugt unsichtbare Abhängigkeiten

Lösung durch Komposition statt Vererbung

Heinrich Heine Universität Düsseldorf

Bessere Lösung - Komposition verwenden:

```java
public class InstrumentedSet<E> implements Set<E> {
    private int addCount = 0;
    private final Set<E> set;  // Komposition statt Vererbung
    public InstrumentedSet(Set<E> set) {
        this.set = set;
    }
    @Override
    public boolean add(E e) {
        addCount++;
        return set.add(e);  // Delegation
    }
    @Override
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return set.addAll(c);  // Delegation - kein doppeltes Zählen!
    }
    public int getAddCount() { return addCount; }
    // Alle anderen Set-Methoden an set delegieren...
    @Override public int size() { return set.size(); }
    // ... usw.
}
```

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

Refused Bequest Code Smell

Heinrich Heine Universität Düsseldorf

Refused Bequest Definition

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

Problematisches Beispiel:

```java
public interface Bird {
    void fly();
    void eat();
}

public class Ostrich implements Bird {
    @Override
    public void fly() {
        throw new UnsupportedOperationException();
    }

    @Override
    public void eat() {
        // ...
    }
}
```

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)

Refused Bequest: Lösung durch Interface Segregation

Heinrich Heine Universität Düsseldorf

Lösung: Kleinere, spezifischere Interfaces

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

```java
public interface Bird {
    void eat();
}
public interface FlyingBird extends Bird {
    void fly();
}
public class Ostrich implements Bird {
    @Override
    public void eat() { /* ... */ }
    // Keine fly() Methode mehr nötig!
}
public class Swallow implements FlyingBird {
    @Override
    public void fly() { /* ... */ }
    @Override
    public void eat() { /* ... */ }
}
```

Überblick - SOLID-Prinzipien

Heinrich Heine
Universität Düsseldorf

4 Tag 3 - Architektur & Prinzipien
- Bausteine & Strukturen
- Vererbung & Polymorphismus
- SOLID-Prinzipien
- Code Smells im Großen

SOLID-Prinzipien Überblick

Heinrich Heine Universität Düsseldorf

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

Open/Closed Principle (OCP) - Verletzung

Heinrich Heine Universität Düsseldorf

OCP Definition

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

```java
public class AreaCalculator {
    public double calculateArea(Shape shape) {
        switch (shape.getType()) {
            case RECTANGLE:
                Rectangle rect = (Rectangle) shape;
                return rect.getWidth() * rect.getHeight();
            case CIRCLE:
                Circle circle = (Circle) shape;
                return Math.PI * circle.getRadius() * circle.getRadius();
            default:
                throw new IllegalArgumentException("Unknown shape type");
        }
    }
}
```

Problem

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

Open/Closed Principle (OCP) - Lösung

```java
interface Shape {
    double calculateArea();
}
class Rectangle implements Shape {
    private final double width, height;
    // ... constructor ...
    @Override
    public double calculateArea() { return width * height; }
}
class Circle implements Shape {
    private final double radius;
    // ... constructor ...
    @Override
    public double calculateArea() { return Math.PI * radius * radius; }
}
// Neue Shapes können hinzugefügt werden ohne bestehenden Code zu ändern!
class Point implements Shape {
    @Override
    public double calculateArea() { return 0.0; }
}
```

Vorteil

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

OCP in der Praxis - Template Method Pattern

Heinrich Heine Universität Düsseldorf

```java
abstract class DataProcessor {
    // Template Method - definiert Algorithmus-Struktur (geschlossen)
    public final void processData() {
        Data data = readData();
        Data transformed = transformData(data);
        writeData(transformed);
    }
    // Hooks für Subklassen - offen für Erweiterung
    protected abstract Data readData();
    protected abstract Data transformData(Data data);
    protected abstract void writeData(Data data);
}
class CSVProcessor extends DataProcessor {
    @Override protected Data readData() { /* CSV-spezifisch */ return null; }
    @Override protected Data transformData(Data data) { /* ... */ return null; }
    @Override protected void writeData(Data data) { /* ... */ }
}
class XMLProcessor extends DataProcessor {
    @Override protected Data readData() { /* XML-spezifisch */ return null; }
    @Override protected Data transformData(Data data) { /* ... */ return null; }
    @Override protected void writeData(Data data) { /* ... */ }
}
```

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

Liskov Substitution Principle (LSP) - Verletzung

Heinrich Heine Universität Düsseldorf

LSP Definition

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

```java
class Rectangle {
    // ... (Rechteck with width, height und den jeweiligen settern)
    public int getArea() { return width * height; }
}

// Test zeigt LSP-Verletzung:
public void testRectangle(Rectangle rect) {
    rect.setWidth(5);
    rect.setHeight(4);
    assert rect.getArea() == 20;  // Failt für Square! (=16)
}
```

```java
class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        this.width = width;
        this.height = width;  // Quadrat-Eigenschaft
    }
    @Override
    public void setHeight(int height) {
        this.width = height;
        this.height = height; // Quadrat-Eigenschaft
    }
}
```

Problem

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

Liskov Substitution Principle (LSP) - Lösung

Heinrich Heine Universität Düsseldorf

LSP-konforme Lösung durch gemeinsames Interface:

```java
interface Shape {
    int getArea();
}
class Rectangle implements Shape {
    private final int width, height;
    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }
    public int getArea() { return width * height; }
}
class Square implements Shape {
    private final int side;
    public Square(int side) { this.side = side; }
    public int getArea() { return side * side; }
}
```

Ergebnis

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

Interface Segregation Principle (ISP) - Verletzung

Heinrich Heine Universität Düsseldorf

ISP Definition

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

ISP-Verletzung durch ein zu großes Interface:

```java
interface Machine {
    void print();
    void staple();
}
class SimplePrinter implements Machine {
    public void print() { /* OK */ }
    // UNSINNIG für einen einfachen Drucker!
    public void staple() {
        throw new UnsupportedOperationException();
    }
}
```

Problem

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

Interface Segregation Principle (ISP) - Lösung

Heinrich Heine Universität Düsseldorf

ISP-konforme Lösung durch kleine, fokussierte Interfaces:

```java
interface Printer {
    void print();
}
interface Stapler {
    void staple();
}
// Implementiert nur, was es kann
class SimplePrinter implements Printer {
    @Override public void print() { /* ... */ }
}
// Kombiniert Fähigkeiten nach Bedarf
class MultifunctionPrinter implements Printer, Stapler {
    @Override public void print() { /* ... */ }
    @Override public void staple() { /* ... */ }
}
```

Vorteil

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

Dependency Inversion Principle (DIP) - Verletzung

Heinrich Heine Universität Düsseldorf

DIP Definition

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

```java
class EmailService { // Konkrete Implementierung
    public void sendEmail(String to, String message) {
        System.out.println("Sending email via SMTP to " + to);
    }
}
class NotificationService {
    private EmailService emailService;  // Konkrete Abhängigkeit!
    public NotificationService() {
        this.emailService = new EmailService();  // Hard gekoppelt!
    }
    public void sendNotification(String user, String message) {
        emailService.sendEmail(user, message);
    }
}
```

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

Heinrich Heine
Universität Düsseldorf

```java
interface MessageSender {  // Abstraktion
    void sendMessage(String to, String message);
}
class EmailService implements MessageSender {
    @Override
    public void sendMessage(String to, String message) {
        System.out.println("Sending email via SMTP to " + to);
    }
}
class SMSService implements MessageSender { // Weitere Implementierung
    @Override
    public void sendMessage(String to, String message) {
        System.out.println("Sending SMS to " + to);
    }
}
class NotificationService {
    private MessageSender messageSender;  // Abhängigkeit von Abstraktion!
    // Abhängigkeit wird von außen injiziert (Dependency Injection)
    public NotificationService(MessageSender messageSender) {
        this.messageSender = messageSender;
    }
    public void sendNotification(String user, String message) {
        messageSender.sendMessage(user, message);
    }
}
```

Überblick - Code Smells im Großen

Heinrich Heine
Universität Düsseldorf

4. Tag 3 - Architektur & Prinzipien
   - Bausteine & Strukturen
   - Vererbung & Polymorphismus
   - SOLID-Prinzipien
   - Code Smells im Großen

Code Smells im Großen - Überblick

Heinrich Heine Universität Düsseldorf

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

Code Smells im Großen - Überblick

Heinrich Heine Universität Düsseldorf

Wichtig für Klausur

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

Large Class Code Smell

Definition: Klassen mit sehr vielen Methoden und Verantwortlichkeiten.

```java
public class Webcrawler {
    // Viele verschiedene Datenstrukturen
    Queue<URL> toVisit; Map<URL, String> visited; Map<String, List<URL>> keywordIndex;
    Set<String> blockedDomains; HttpClient httpClient;

    // 50+ Methoden für verschiedene Aufgaben:
    // URL-Management
    void addUrl(URL url) { /* ... */ }
    URL nextUrl() { /* ... */ }
    // HTTP-Operationen
    String downloadPage(URL url) { /* ... */ }
    // Content-Parsing
    List<URL> extractLinks(String content) { /* ... */ }
    List<String> extractKeywords(String content) { /* ... */ }
    // Indexing
    void indexKeywords(String content, URL url) { /* ... */ }
}
```

Probleme:

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

Refactoring:

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

Primitive Obsession Code Smell

Heinrich Heine Universität Düsseldorf

Definition: Verwendung primitiver Datentypen statt spezifischer Fachobjekte.

Problematisches Beispiel:

```
class Konto {
    private final String iban;
    private final String name;

    public Konto(String name, String iban) {
        // IBAN-Validierung in Konto-Klasse!
        if (!gueltigeIBAN(iban)) {
            throw new IllegalArgumentException();
        }
        this.iban = iban;
        this.name = name;
    }
    private static boolean gueltigeIBAN(String iban) {
        if (iban.length() != 22) return false;
        // ...
        return true;
    }
}
```

Verbesserte Lösung:

```
record IBAN(String nummer) {
    public IBAN(String nummer) {
        if (!isValid(nummer)) {
            throw new IllegalArgumentException();
        }
        this.nummer = nummer.toUpperCase();
    }
    private static boolean isValid(String iban) {
        // ...
    }
}
class Konto {
    private final IBAN iban;
    private final String name;
    public Konto(String name, IBAN iban) {
        this.iban = iban; // Bereits validiert!
        this.name = name;
    }
}
```

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

Data Clumps Code Smell

Heinrich Heine Universität Düsseldorf

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

Problematisches Beispiel:

```java
public void printStudent(String name, String email, int punkte) {
    System.out.println(name + " (" + email + "): " + punkte);
}

public static void main(String[] args) {
    // Parallele Arrays - ein Data Clump!
    int[] punkte = {79, 49, 90, 33};
    String[] namen = {"Susanne", "Domenica", "Majid", "Jonas"};
    String[] emails =
        {"sus@hhu.de", "dom@hhu.de", "maj@hhu.de", "jon@hhu.de"};

    // Fehleranfällig: Arrays müssen synchron bleiben!
    for (int i = 0; i < namen.length; i++) {
        if (punkte[i] >= 45) {
            printStudent(namen[i], emails[i], punkte[i]);
        }
    }
    // Beispielhaftes Problem:
    // Sortierung nach Punkten sehr umständlich
}
```

Verbesserte Lösung:

```java
public record TeilnehmerIn(
        String name, String email, int punkte) {
    public boolean hatBestanden() {
        return punkte >= 45;
    }
}
public void printStudent(TeilnehmerIn student) {
    System.out.println(student.name() + " (" +
        student.email() + "): " + student.punkte());
}
public static void main(String[] args) {
    List<TeilnehmerIn> teilnehmer = List.of(
        new TeilnehmerIn("Susanne", "sus@hhu.de", 79),
        new TeilnehmerIn("Domenica", "dom@hhu.de", 49),
        new TeilnehmerIn("Majid", "maj@hhu.de", 90),
        new TeilnehmerIn("Jonas", "jon@hhu.de", 33)
    );
    teilnehmer.stream()
        .filter(TeilnehmerIn::hatBestanden)
        .sorted((a, b) -> Integer.compare(b.punkte(), a.punkte()))
        .forEach(this::printStudent);
}
```

Divergent Change Code Smell

Heinrich Heine Universität Düsseldorf

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

Problematisches Beispiel - Active Record:

```java
public class Person {
    private String firstName, lastName;

    // Geschäftslogik - Änderungsgrund 1
    public String getFullName() {
        return firstName + " " + lastName;
    }
    // Datenbankzugriff - Änderungsgrund 2
    public void saveToDatabase() {
        // SQL INSERT statement...
    }
    // JSON-Serialisierung - Änderungsgrund 3
    public String toJson() {
        return "{\"firstName\":\"" + firstName + "\"}";
    }
}
```

Bessere Lösung - Trennung:

```java
public class Person {
    private final String firstName, lastName;
    public String getFullName() {
        return firstName + " " + lastName;
    }
}
public class PersonRepository {
    public void save(Person person) {
        // SQL INSERT statement...
    }
}
public class PersonJsonSerializer {
    public String toJson(Person person) {
        return "{\"firstName\":\"" + person.getFirstName() + "\"}";
    }
}
```

Problem: Die ursprüngliche Klasse verletzt das SRP, da sie sich bei Änderungen der Geschäftslogik, des Datenbankschemas UND des Serialisierungsformats ändern muss.

Shotgun Surgery Code Smell

Heinrich Heine Universität Düsseldorf

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

Problematisches Beispiel - Logging:

```java
public class OrderService {
    public void createOrder(Order order) {
        System.out.println(LocalDateTime.now() +
            " [INFO] Creating order: " + order.getId());
        // ...
    }
}

public class PaymentService {
    public void processPayment(Payment p) {
        System.out.println(LocalDateTime.now() +
            " [INFO] Processing payment: " + p.getId());
        // ...
    }
}
// Problem: Log-Format ändern = ALLE Klassen ändern!
```

Verbesserte Lösung - Zentrale Abstraktion:

```java
public class Logger { // Zentrale Logging-Klasse
    public static void info(String message) {
        System.out.println(LocalDateTime.now() +
            " [INFO] " + message);
    }
}
public class OrderService {
    public void createOrder(Order order) {
        Logger.info("Creating order: " + order.getId());
        // ...
    }
}
public class PaymentService {
    public void processPayment(Payment p) {
        Logger.info("Processing payment: " + p.getId());
        // ...
    }
}
```

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

Feature Envy Code Smell

Heinrich Heine Universität Düsseldorf

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

Problematisches Beispiel:

```java
public class Monitor {
    public void sendeAlarm(String msg) { /*...*/ }
    public int getSystemTemperatur() { /*...*/ }
    public boolean isCpuFanRunning() { /*...*/ }
    public int getMemoryUsage() { /*...*/ }
}

public class RechnerKontrolle {
    public void healthCheck(Monitor monitor) {
        // Feature Envy - viele Zugriffe auf Monitor!
        if (monitor.getSystemTemperatur() > 93) {
            monitor.sendeAlarm("CPU überhitzt");
        }
        if (!monitor.isCpuFanRunning() &&
            monitor.getSystemTemperatur() > 85) {
            monitor.sendeAlarm("CPU-Lüfter ausgefallen");
        }
    }
}
```

Verbesserte Lösung (Move Method):

```java
public class Monitor {
    public void sendeAlarm(String msg) { /*...*/ }
    private int getSystemTemperatur() { /*...*/ }
    private boolean isCpuFanRunning() { /*...*/ }
    private int getMemoryUsage() { /*...*/ }
    // Logik ist jetzt in der Klasse, zu der sie gehört
    public void healthCheck() {
        if (getSystemTemperatur() > 93) {
            sendeAlarm("CPU überhitzt");
        }
        if (!isCpuFanRunning() &&
            getSystemTemperatur() > 85) {
            sendeAlarm("CPU-Lüfter ausgefallen");
        }
    }
}
public class RechnerKontrolle {
    public void performHealthCheck(Monitor monitor) {
        // "Tell, don't ask" - keine Feature Envy mehr!
        monitor.healthCheck();
    }
}
```

Message Chains Code Smell

Heinrich Heine Universität Düsseldorf

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

Problematisches Beispiel:

```java
public class Rechnung {
    public double berechneGesamtsumme() {
        double summe = 0.0;
        for (Position pos : positionen) {
            // Message Chain:
            summe += pos.getAnzahl() *
                     pos.getProdukt().getPreis();
        }
        return summe;
    }
}
// Noch schlimmer:
public void printCustomerCity(Order order) {
    String city = order.getCustomer()
                       .getAddress()
                       .getCity()
                       .getName();
    System.out.println(city);
}
```

Verbesserte Lösung (Hide Delegate):

```java
public class Position {
    public double getGesamtpreis() {
        return anzahl * produkt.getPreis();
    }
}

public class Rechnung {
    public double berechneGesamtsumme() {
        return positionen.stream()
            .mapToDouble(Position::getGesamtpreis)
            .sum(); // Keine Chain mehr!
    }
}

public class Order {
    public String getCustomerCityName() {
        return customer.getCityName(); // Delegieren
    }
}
```

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

Wann Code Smells beheben?

Heinrich Heine Universität Düsseldorf

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!

Achtung: Subjektives Cheatsheet!

Heinrich Heine Universität Düsseldorf

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 Smells im Kleinen - Prinzipien-Verletzungen

Heinrich Heine Universität Düsseldorf

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 Smells im Großen - Prinzipien-Verletzungen

Heinrich Heine Universität Düsseldorf

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: `a.getB().getC()`
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 Smells - Erkennungstabelle

Heinrich Heine Universität Düsseldorf

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

Tag 3 - Zusammenfassung

Heinrich Heine
Universität Düsseldorf

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?

Heinrich Heine
Universität Düsseldorf

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)

Fragen & Diskussion

Heinrich Heine
Universität Düsseldorf

Vielen Dank für eure Aufmerksamkeit!

Fragen?