Programmierpraktikum 1

Nachklausurtutorium Sommersemester 2025

Paul Christian Dötsch

Institut für Informatik
Heinrich-Heine-Universität Düsseldorf

16. September 2025

Überblick - Programmierpraktikum 1

Heinrich Heine
Universität Düsseldorf

1. Tag 2 - Testing & Codequalität

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

Heinrich Heine Universität Düsseldorf

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

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:

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

Strings:

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

Collections:

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

Booleans und null:

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

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

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!

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 $\rightarrow$ komplexere UI
- **Kosten vs. Qualität**: Höhere Qualität braucht mehr Zeit/Ressourcen

Wartbarkeit - Warum so wichtig?

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

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 ïm Kleinen"

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

Mehrere Verantwortlichkeiten / Aufgaben

Problem: Eine Methode oder Klasse macht zu viele verschiedene Dinge

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() { /* ... */ }
}
```

Vorteile der Aufteilung:

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;
```

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
}
```

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
```

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

hhu
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

Fragen & Diskussion

Heinrich Heine
Universität Düsseldorf

Vielen Dank für eure Aufmerksamkeit!

Fragen?

Diskussion und Erfahrungsaustausch