

# Zusammenfassung: Testen & Fehlerbeseitigung

---

Beim Programmieren muss man immer damit rechnen, dass Fehler auftreten. Deshalb gehören Testen und Fehlerbeseitigung **Debugging** fest zum Entwicklungsprozess dazu. Dieses Kapitel behandelt, wie man Fehler findet, Tests schreibt und wie Tools wie **JUnit** dabei helfen.

## ◇ 1. Testen

**Testen** überprüft, ob Software das gewünschte Verhalten zeigt — egal ob eine **Methode**, eine **Klasse** oder ein komplettes Programm.

Arten von Tests:

- **Modultests**
  - Testen einzelner **Methoden** oder **Klassen**
- **Positives Testen**
  - Fälle testen, die funktionieren sollen
- **Negatives Testen**
  - Fälle testen, die nicht funktionieren sollen (z. B. ungültige Eingaben)
- **Regressionstests**
  - **Automatisierte Tests**, die überprüfen, dass frühere Fehler nicht zurückkehren

## ◇ 2. Fehlerarten

### ◇ Syntaxfehler

Code kann nicht kompiliert werden.

Meist Tippfehler, vergessene Zeichen, falsche Syntax.

### ◇ Logische Fehler

Programm läuft, macht aber das Falsche.

Werden nur durch **Testen** oder **Debugging** gefunden.

### ◇ 3. Automatisiertes Testen mit JUnit

JUnit ermöglicht das automatische Ausführen von Tests.

Beispiel (vereinfachter JUnit-Test):

```
@Test
public void testAddiere() {
    Rechner r = new Rechner();
    assertEquals(7, r.addiere(3, 4));
}
```

Warum JUnit?

- Tests laufen automatisch.
- Fehler werden früh gefunden.
- Hilft zu sehen, wo ein Fehler auftritt.
- Ideal für Regressionstests.

### ◇ 4. Zusicherungen Assertions

Eine Zusicherung prüft, ob ein Ausdruck wahr ist.

Beispiel in JUnit:

```
assertTrue(konto.getSaldo() >= 0);
```

Wenn die Bedingung falsch ist → Test schlägt fehl.

### ◇ 5. Testgerüst Fixture

Eine Sammlung von Objekten, die vor jedem Test vorbereitet wird.

Beispiel:

```
@BeforeEach
public void setup() {
    konto = new Konto();
    konto.einzahlen(100);
}
```

Tests basieren dann auf einem definierten Startzustand.

## ◇ 6. Fehlerbeseitigung (Debugging)

- Fehlerbeseitigung bedeutet:
  - Fehler finden
  - Ursache verstehen
  - Korrigieren
- Techniken:
  - Debugger nutzen → **Breakpoints** setzen, Programm schrittweise ausführen
  - Zustandsänderungen beobachten
  - manuelle Ausführung → Code gedanklich oder schriftlich Zeile für Zeile durchgehen
  - Aufrufsequenzen analysieren → Welche **Methode** ruft welche andere in welcher Reihenfolge auf?

## ◇ 7. Manuelle Ausführung

Das bewusste, zeilenweise Durchgehen eines Codes:

- Welche **Variablen** ändern sich?
- Welche **Methode** wird als nächstes aufgerufen?
- Welcher Zustand entsteht?

Dies ist extrem hilfreich bei logischen Fehlern.

## ◇ 8. Aufrufsequenzen

Wichtig für das Verständnis, wie ein Programm intern arbeitet.

Beispiel:

```
main() → berechne() → validiere() → speichere()
```

Fehler lassen sich oft auf eine bestimmte Station der Aufrufkette zurückführen.

## 📌 Kleines Beispiel: **Positives** + **negatives** Testen

```
@Test
public void testAbhebenPositiv() {
    Konto k = new Konto(100);
    assertEquals(50, k.abheben(50));
}

@Test
public void testAbhebenNegativ() {
    Konto k = new Konto(20);
    assertEquals(0, k.abheben(50)); // darf nicht klappen
}
```

## 🧠 Merksätze

Jeder Code enthält Fehler — Tests gehören zum Alltag.

**Positiv** testen: Was funktionieren soll, muss funktionieren.

**Negativ** testen: Was nicht funktionieren soll, darf nicht funktionieren.

Automatisierte Tests **JUnit** sparen Zeit und verhindern Rückschritte.

**Debugging** ist ein aktiver Prozess: analysieren, verstehen, korrigieren.

Testgerüste sorgen für gleichbleibende Testbedingungen.

**Regressionstests** stellen sicher, dass alte Fehler nicht zurückkehren.