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.

🧩 Schablone: Äquivalenzklassen beim Testen

Diese Schritte kannst du 1:1 in Prüfungen anwenden.

1️⃣ Methode & Parameter identifizieren

✏️ Frage:

- **Was soll getestet werden?**
- **Welche Eingaben gibt es?**

📌 Schablone:

```
Zu testende Methode: <Methodenname>
Parameter:
- <Name> (<Typ>)
```

📌 Beispiel (Uhr):

```
Zu testende Methode: addiereMinuten(int minuten)
Parameter:
- minuten (int)
```

2️⃣ Einschränkungen & Regeln aus der Beschreibung extrahieren

✏️ Frage:

- **Welche Bedingungen gelten für die Eingaben?**

🔎 Suche nach Wörtern wie:

- „muss“
- „darf“
- „mindestens“
- „höchstens“
- „nur wenn“
- „ansonsten Exception“

📌 Schablone:

```
Einschränkungen:
- <Regel 1>
- <Regel 2>
```

📌 Beispiel:

```
Einschränkungen:
- minuten ≥ 1
- minuten ≤ 1440
```

3️⃣ Wertebereich in Äquivalenzklassen aufteilen

✏️ Frage:

- **Welche Bereiche werden gleich behandelt?**

📌 Schablone:

```
Äquivalenzklassen:

EK1 (gültig):     ...
EK2 (ungültig):   ...
EK3 (ungültig):   ...
```

📌 Beispiel:

```
EK1 (gültig):     1 ≤ minuten ≤ 1440
EK2 (ungültig):   minuten < 1
EK3 (ungültig):   minuten > 1440
```

4️⃣ Repräsentative Testwerte wählen

✏️ Frage:

- **Welche einzelnen Werte stehen stellvertretend für die Klasse?**

📌 Regeln:

- Pro Äquivalenzklasse mindestens ein Wert
- Grenzwerte bevorzugen

📌 Schablone:

```
Testwerte:
- EK1: <Wert(e)>
- EK2: <Wert(e)>
- EK3: <Wert(e)>
```

📌 Beispiel:

```
Testwerte:
- EK1: 1, 10, 59, 60, 1440
- EK2: 0, -1
- EK3: 1441
```

5️⃣ Erwartetes Verhalten festlegen

📝 Frage:

- **Was passiert bei jeder Klasse?**

📌 Schablone:

```
Erwartetes Verhalten:
- EK1: Korrekte Berechnung
- EK2: Exception
- EK3: Exception
```

📌 Beispiel:

```
EK1 → Uhrzeit wird korrekt angepasst
EK2 → IllegalArgumentException
EK3 → IllegalArgumentException
```

6️⃣ Testfälle formulieren (optional Code)

📌 Schablone (JUnit):

Gültiger Test

```
@Test
public void test<Beschreibung>() {
objekt.methode(wert);
assertEquals(...);
}
```

Ungültiger Test

```
@Test(expected = IllegalArgumentException.class)
public void test<Beschreibung>() {
objekt.methode(wert);
}
```

7️⃣ Prüfungs-Merksätze 🧠

💡 Kurz & effektiv merken:

- Eine Äquivalenzklasse = ein Bereich mit gleichem Verhalten
- Pro Klasse reicht ein repräsentativer Test
- Grenzen sind besonders wichtig
- Ungültige Klassen → Exception-Tests

Mini-Spickzettel für Prüfungen

```
1. Parameter bestimmen
2. Regeln lesen (muss / darf / min / max)
3. Bereiche bilden
4. Gültig vs. ungültig trennen
5. Grenzwerte testen
6. Exception nicht vergessen
```