Folien (PDF) Projekt (Git)

Sie können
- Testfälle für Klassen/Objekte schreiben,
- die Testabdeckung betrachten und
- dem Test-First-Ansatz folgen.

# 29 Testen (Objektorientiert)

## 29.1 Testen von Klassen / Objekten

Wir haben bereits das Testen einfacher (Klassen-)Methoden kennengelernt. Dort haben wir im Wesentlichen überprüft, ob Methoden für feste Eingabewerte die erwartete Rückgabe liefern. Was ist in der Objektorientierung nun anders?

Zunächst: Wir rufen Methoden auf, deren Verhalten gar nicht mehr (nur) von Parametern abhängen kann, sondern auch vom Wert der Instanzvariablen, also dem Zustand des Objektes. Und das führt gleich zum zweiten Problem: Die Instanzvariablen sind im buchstäblich *Privatsache* der Klasse – auch beim Testen sollten wir nicht direkt darauf zugreifen.

Daher ändert sich auf das Testen ein wenig. Statt zu testen, dass eine Methode sich *für gegebene Parameter* auf eine erwartete Art verhält, müssen wir jetzt die Vergangenheit des Objektes betrachten: *Wenn vorher mit dem Objekt folgendes passiert ist, dann sollte beim Aufruf dieser Methode folgendes passieren.*

Nehmen wir als ganz einfaches Beispiel eine Klasse `Counter` zum Zählen von Ereignissen mit Methoden `void countUp()` und `int getCount()`. Ein sinnvoller Testfall wäre, dass man, wenn man ein neues Objekt erstellt (Ausgangssituation) und dann zweimal `countUp()` aufruft, beim Aufruf von `getCount()` den Wert 2 erhält.

```
Count sut = new Count(); //sut: subject under test
sut.countUp();
sut.countUp();
System.out.println(sut.getCount()); // sollte 2 ausgeben!
```

Auch hier ist der Testfall sehr konkret, aber er besteht nicht mehr nur aus einem Methodenaufruf. Trotzdem lässt er sich wieder mit Assertions als JUnit-Test schreiben (hier als einziger Test einer Testklasse):

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class CounterTest {

  @Test
  public void sollteZweimalKorrektZaehlen() {

    Counter sut = new Counter();
    sut.countUp();
    sut.countUp();
```

```
    int value = sut.getCount();

    Assertions.assertEquals(2, value, "Counter zählt nicht korrekt hoch.");
  }
}
```

https://gitlab.lrz.de/ib-dev1/dev1-e29.git

## 29.2 Struktur von Testfällen

Als Struktur für Testfälle empfehlen sich die Aufteilung in drei (sehr kurze) Abschnitte: *Arrange*/*Act*/*Assert*[1].

Dabei wird der Testfall (Wie ein Experiment) zerlegt in den Versuchsaufbau (*Arrange*, z.B. Erstellen benötigter Objekte), die Durchführung (*Act*, z.B. einzelner Aufruf oder zusammenhängende Folge von Aufrufen) und die Beobachtung und Überprüfung (*Assert*, also der Vergleich von Erwartung mit tatsächlichem Verhalten).

Dieser „Dreisprung“ ist nicht immer trennscharf, aber er verhindert, dass man mehrere Testfälle miteinander in einer Testmethode vermischt. Nutzen Sie ruhig Kommentare in Testmethoden, um die Struktur zu verdeutlichen:

```
  @Test
  public void sollteZweimalKorrektZaehlen() {

    // Arrange
    Counter sut = new Counter();

    //Act
    sut.countUp();
    sut.countUp();
    int value = sut.getCount();

    // Assert
    Assertions.assertEquals(2, value, "Counter zählt nicht korrekt hoch.");
  }
}
```

## 29.3 Testabdeckung

Ob für eine Programmeinheit ausreichende Tests vorliegen, wird häufig in Form einer **Testabdeckung** gemessen. Dahinter verbergen sich verschiedene Metriken/Messgrößen, die beschreiben, wie gut der Code durch Tests abgedeckt wird. Die einfachste Form, die in Entwicklungsumgebungen häufig ausgewiesen werden, ist die *Anweisungsüberdeckung*, die misst, wie groß ist der Anteil von Codezeilen, die bei der Ausführung der Tests ausgeführt wurden, ist.

[1] Alternativ: Given/When/Then

## 29.4 Test-Driven-Development (TDD)

Implementiert man zunächst eine Klasse oder Methode und dann die zugehörigen Tests, ist die Gefahr groß, dass man erst bei der Erstellung der Testfälle über Sonderfälle nachdenkt. Man „versteht“ Teile des Problems erst, nachdem man eine schlechte Lösung fertiggestellt hat. Hat man andersherum das Glück, dass man das Problem tatsächlich verstanden hatte, fühlt es sich sinnlos an, Testfälle zu schreiben, die scheinbar keinen Mehrwert liefern.

Eine Alternative ist das sogenannte **Test-Driven-Development** (TDD, test-getriebene Entwicklung). Hier schreibt man sukzessive Tests, die vorgeben, wie die Einheit funktionieren soll und ergänzt die Implementierung dann soweit, dass die Tests grün werden. Man beginnt also mit einem Testfall, der fehlschlagen sollte, entwickelt dann, bis der Testfall erfüllt wird und macht dann mit dem nächsten Testfall weiter, bis man keine Testfälle mehr findet, die fehlschlagen würden.

## 29.5 Übungen

Für die folgenden Aufgabe(n) gibt es wie gewohnt Vorlageklassen und Tests in folgendem GitLab-Projekt: `https://gitlab.lrz.de/courses/dev1/dev1-e29.git`.

### 29.5.1 Test-Driven Design

Im Projekt finden Sie im das Beispiel `Counter` von den Folien, in dem Sie Tests ergänzen können und die Implementierung nachziehen. Arbeiten Sie nach TDD und ergänzen Sie immer nur einen Test und implementieren dann nur, was notwendig ist, um den Test zu bestehen!

Eine Lösung finden Sie hier: Counter.java, CounterTest.java

## Folien

Hochschule München University of Applied Sciences

Fakultät für Mathematik und Informatik

29

v2023-01

# Softwareentwicklung I (W)

## Testen (Objektorientiert)

Bastian Katz

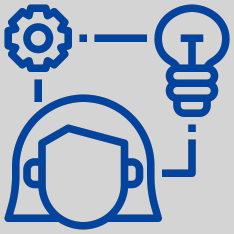
B. Sc. Wirtschaftsinformatik – Informationstechnologie
B. Sc. Wirtschaftsinformatik – Digitales Management

letzte Änderung: 2024-11-21 21:52:40



### Ziele

Sie testen das Verhalten von Objekten und testen vor allem frühzeitig!



Erläuterungen und Aufgaben zur Überprüfung finden Sie im Begleittext!

*Sie können*
- *Testfälle für Klassen/Objekte schreiben,*
- *die Testabdeckung betrachten und*
- *dem Test-First-Ansatz folgen.*



### Testfälle für Klassen/Objekte

*Ein **Testfall** beschreibt eine konkrete Situation, in der das Verhalten eines Objektes überprüft wird*
- *Ausgangssituation*
- *Konkrete Eingabe/Interaktion*
- *Konkrete Erwartung an Ausgabe oder Verhalten*

Unterschied zum Testen einfacher Methoden
- Erwartetes Verhalten abhängig von der Vergangenheit, nicht (nur) von Parametern!

Beispiele:
- Berechnung des Restkredites hängt von den Werten bei Erzeugung ab!
- Wert eines Bruches hängt von den durchgeführten Rechenoperationen ab.

## Warum fehlen eigentlich immer Tests?

Schreiben von Tests kostet Zeit
- „Der Kunde bezahlt nur Funktionen, nicht Tests“
- „Tests schreiben ist langweilig“

Der Mehrwert ist nicht immer erkennbar
- „Es funktioniert doch!“
- „Wofür haben wir Tester?“

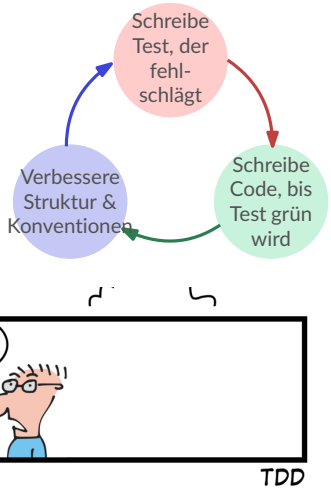
Die Lösung: Test First!
- Mit dem Schreiben von Tests beginnen
  - erst klären und aufschreiben, was funktionieren soll, dann implementieren!
- Motivierender und zielgerichteter
  - Erzwingt Klärung von Fragen zu Beginn
  - Tests geben Rückmeldung, definieren ein klares Ziel

Gute Projekte haben mehr Testcode als Programmcode!



## Test-Driven Development (TDD)

Test-Driven Development (TDD) ist ein Entwicklungsvorgehen, das sich konsequent an Tests orientiert
- 1. Neuen Testfall schreiben
- 2. Tests ausführen
  - Neuer Test sollte fehlschlagen
- 3. Code schreiben
- 4. Tests ausführen
  - Wiederhole 3 & 4, bis alle Tests erfolgreich sind
- 5. Code verbessern
  - Dabei müssen Tests erfolgreich bleiben
- Repeat






## Beispiel & Demo

Wir implementieren eine Klasse `Counter`, mit der wir Ereignisse zählen können!
- Initialisierung mit Konstruktor ohne Parameter auf Wert 0
- Methode `void countUp()` erhöht den Wert um `1`
- Methode `void reset()`, die den Wert auf 0 zurücksetzt
- Methode `int getCount()`, die den Wert zurückgibt.

Womit fangen Sie an?
- Test first! Erst die Tests!

## Struktur von Modultests (Unit-Tests) für Objekte

Gute Testfälle zerfallen in drei Schritte
- „Arrange–Act–Assert“

Arrange
- Erstelle die Situation in der der zu testende Funktionalität aufgerufen wird
  - Erzeugung benötigter abhängiger Objekte
  - Initialisierung des Testobjektes („subject under test“)

Act
- ruft die zu testende Methode / Funktion auf
  - manchmal: zusammenhängende Kombination von Aufrufen

Assert
- überprüft das Ergebnis / den Zustand des Testobjektes
  - keine ändernden Aufrufe mehr

```java
@Test
public void testeObjekt() {
    // Arrange
    // ...

    // Act
    // ...

    // Assert
    // ...
}
```



## Initialisierung (Arrange)

Im *Arrange*-Teil erstellen wir ein neues Objekt der Klasse `Counter`
- Moment mal! Die Klasse gibt es doch gar nicht!
  - *Jetzt* erstellen wir *nur* die Klasse
  - nichts programmieren, was wir nicht brauchen, damit der Test *syntaktisch* korrekt wird!

```java
/**
 * Zähler-Klasse.
 */
public class Counter {

}
```

```java
@Test
public void sollteMitNullInitialisiertWerden() {
    // Arrange
    Counter sut = new Counter();

    // ...
}
```



## Ausführung (Act)

Im *Act*-Teil fragen wir den Zählerstand ab
- Das ist die Funktion, von der wir nicht wissen, ob sie unter dieser Voraussetzung schon funktioniert
- Fehlende Methode(n) wieder nur anlegen, nicht ausprogrammiert.

```java
/**
 * Zähler-Klasse.
 */
public class Counter {
    public int getCount() {
        return 0;
    }
}
```

```java
@Test
public void sollteMitNullInitialisiertWerden() {
    // Arrange
    Counter sut = new Counter();

    // Act
    int value = sut.getCount();
}
```

## Überprüfung (Assert)

Im *Assert*-Teil vergleichen wir Ergebnisse mit Erwartungen

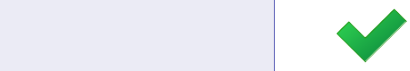
- Hier sollte das Objekt nocht mehr verändert werden!
- Hier sollten keine fehlenden Methoden ergänzt werden!

```java
/**
 * Zähler-Klasse.
 */
public class Counter {
  public int getCount() {
    return 0;
  }
}
```

```java
@Test
public void sollteMitNullInitialisiertWerden() {
    // Arrange
    Counter sut = new Counter();

    // Act
    int value = sut.getCount();

    //Assert
    Assertions.assertEquals(0, value, "Fehlerhafte Initialisierung.");
}
```



Für diesen Testfall reicht das schon. Mehr interessiert uns noch nicht!



## Abgrenzungen Arrange / Act / Assert

Zuordnung ist nicht immer eindeutig

- gehört ein Schritt noch zur Vorbereitung oder schon zur Ausführung?
- Gehört ein lesender Zugriff auf das Testobjekt zur Überprüfung oder noch zur Ausführung?

Faustregeln

- Bei Aufrufen in *Arrange* und *Assert* gehe ich davon aus, dass die Funktionen korrekt sind
  - Werden von anderem Testfall getestet
- *Act*-Teil sollte einer denkbaren Verwendung entsprechen
  - Hierhin kommen Aufrufe, die einer Verwendungssequenz entsprechen



## Wichtige Regeln für Modultests

Modultests sind deterministisch / reproduzierbar

- frei von Zufallselementen → geht..geht nicht..geht

Modultests sind voneinander unabhängig

- keine Annahme über Reihenfolge der Ausführung
- einzeln ausführbar

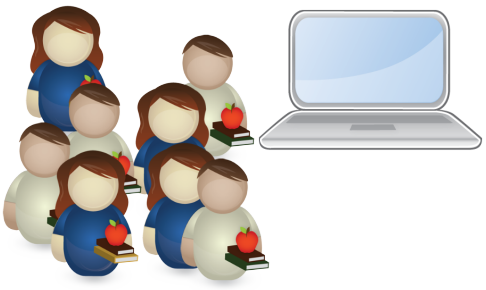
Tests sind kurz, schnell und überprüfen eine einzelne Funktionalität

- keine langen Abläufe, lieber viele Tests
- verständlich, decken erkennbare Fehler auf

Vermeiden Sie Objektvariablen in Testklassen

- können die Unabhängigkeit der Tests verletzen

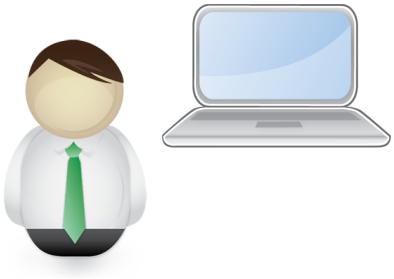
## Jetzt Sie – was wollen Sie testen?





## Jetzt ich - die Implementierung!

Vorsicht! Ich werde nicht besser sein als ihre Tests!





## Testabdeckung

***Testabdeckung*** *ist ein Oberbegriff für Größen, die messen, wie vollständig der Code durch die vorhandenen Tests durchlaufen wird.*

Anweisungsüberdeckung
- Werden alle Codezeilen im Rahmen der Tests ausgeführt?

Verzweigungsüberdeckung
- Treten bei jeder Verzweigung beide Fälle ein?

```
public class Counter {
  private int counter;

  public void countUp() {
    counter++;
  }

  public void reset() {
    counter = 0;
  }

  public int getCount() {
    return counter;
  }
}
```

100% classes, 33% lines covered in package 'edu.hm.cs.bka.dev1.testdemo'

| Element | Class, % | Method, % | Line, % |
|---|---|---|---|
| Counter | 100% (1/1) | 33% (1/3) | 33% (2/6) |

## Kann man alles testen? Sollte man?

Testfälle sind immer eine Auswahl an Verwendungen

- Noch nicht implementierte Funktion oder
- Beispiele für typische Verwendungsmuster oder
- Grenzfälle & Erfahrung oder
  - welche Eingaben sind problematisch?
  - welche Methoden sind kompliziert?
- bereits korrigierte Fehler
  - zu jedem behobenen Bug Tests schreiben!

Ziel: Dinge Testen, die fehlen, falsch sein könnten oder bei Änderungen übersehen werden könnten

- nicht versuchen, alle Werte zu testen!
- möglichst verschiedene Tests schreiben!