Testing-Kurs Übungen

Michael Inden
Freiberuflicher Consultant und Trainer
E-Mail: michael_inden@hotmail.com
Blog: https://jaxenter.de/author/minden

PART I: Einführung Testing — 15 — 30 min

Aufgabe 1: Brainstorming

Kläre die Frage: Was erschwert es uns, Tests zu schreiben?

Aufgabe 2: MoreUnit installieren und nutzen

Installiere das Plugin MoreUnit über den Eclipse Marketplace oder von der Seite

`http://moreunit.sourceforge.net/update-site/`

bzw. für IntelliJ unter

`https://plugins.jetbrains.com/plugin/7105-moreunit`

Aufgabe 3: Beispielprojekte installieren

Installiere die Beispielprogramme und Übungsaufgaben auf deinem Laptop.

PART II: JUnit 5 Intro — 45 — 60 min

Aufgabe 1: Calculator

Aufgabe 1a: Erstelle eine einfache Klasse `Ex01_Calculator` und eine Methode `add()` zum Addieren sowie `divide()` zum Dividieren zweier Zahlen.

Aufgabe 1b: Erstelle zugehörige Testmethoden:
a) für zwei beliebige positive Zahlen
b) den gleichen Wert positiv und negativ: Wie kann man das Naming schön machen?

Aufgabe 1c: Für Exceptions
a) eine Division durch 0.
b) Prüfe auf den Text der Exception

Aufgabe 1d: Erweitere den Calculator um folgende Methode und schreibe einen Test, der einen sprechenden Namen besitzt, etwa **0.1 + 0.1 + 0.1 = 0.3**

```
public double sumUp_0_1(final int factor)
{
	double value = 0.1;
	double result = 0;
	for (int i = 0; i < factor; i++)
	{
		result += value;
	}
	return result;
}
```

Aufgabe 2: Long Runner

Aufgabe 2a: Schaue dir die Klasse `Ex02_LongRunner` sowie den zugehörigen Test an, führe diesen aus. Wieso dauert das so lange? Verbessere die Situation, allerdings darf NUR im Test geändert werden.

Aufgabe 2b: Wie kann man trotz Timeout prüfen, ob das Resultat korrekt ist? Rufe dazu die Methode `calcFib30()` in einem Test auf und wähle 1 Sekunde als Timeout.

Tipp: **Erinnere dich an `assertThrows()` und wie man an das Ergebnis kommt.**

Aufgabe 3: Person Creation

Schaue die die Klasse `Ex03_PersonWithAddress` sowie den zugehörigen Test an, führe diesen aus. Behebe die Fehler! Verbessere sowohl im Test als auch im Java-Code. Wie erhältst du gleich alle Fehler?

Aufgabe 4: Test Inspection

Lies die folgenden als Annotations angegebenen Infos aus und überprüfe diese mit geeigneten Assert-Aufrufen.

```
class Ex04_TestInfoExampleTest
{
    @Test
    @Tag("Fast")
    @Tag("Cool")
    @DisplayName("5 + (-5) => 0 🤔")
    void mytestmethod(TestInfo ti)
    {
      // TODO
    }
 }
```

Aufgabe 5: Arrays und Collections

Aufgabe 5a: Die Utility-Klasse `Arrays` als auch `Collections` bieten Methoden zum Sortieren von Datenstrukturen. Verbessere und vereinfache die bereits bestehenden Tests in der Klasse `Ex05_ArraysSortTest`, die einige Werte sortieren und dann das sortierte Ergebnis prüfen.

Aufgabe 5b: Vereinfache den Test für `listRemoveDuplicates()` mit JUnit-5-Mitteln.

Aufgabe 6: Test Order

Aufgabe 6a: Analysiere die gegeben Klasse `Ex06_LRUCache`, die eine Last Recently User Cache basierend auf einer `LinkedHashMap<K,V>` bereitstellt. Ergänze passende Aufrufe an `assert`-Methoden, um die Funktionalität `testIntersectWithGets()` korrekt zu prüfen.

Aufgabe 6b: Vereinfache die Tests mit JUnit-5-Mitteln, etwa `@Order`.

PART III/IV: JUnit 5 Advanced / Migration — 60 — 75 min

Aufgabe 1: String Reverse

Gegeben sei eine Klasse `Ex01_StringUtils`, die ein Methode `reverse(String)` anbietet, die die Buchstaben in einem String umdrehen kann. Prüfe diese Methode mit entsprechenden Tests.

Tipp: **`@ParameterizedTest, @CsvSource`**

Aufgabe 2: Wohlgeformte Klammen

Gegeben sei eine Implementierung `Ex02_MatchingBracesChecker`, die prüft, ob Klammern wohlgeformt sind, also immer zunächst eine öffnende und dann eine passende schliessende Klammer auftritt, etwa für

```
String input1 = "()[]{}";
String input2 = "[((()[]{}))]";
String inputWrong1 = "(()";
String inputWrong2 = "(()}";
```

Aufgabe 2a: Schreibe entsprechende Tests und versuche zwei Fehler in der Implementierung zu finden und zu korrigieren. Erweitere die Tests, sofern nötig.

Aufgabe 2b: Verbessere die Implementierung der Tests mithilfe von Parameterized Tests.

Aufgabe 3: Schaltjahr: Umwandlung in Parameterized Test mit Hinweis

Gegeben sei eine Berechnung von Schaltjahren in einer Klasse `Ex03_LeapYear`.
Verbessere die Implementierung der Tests in der Klasse `Ex03_LeapYearTest`.

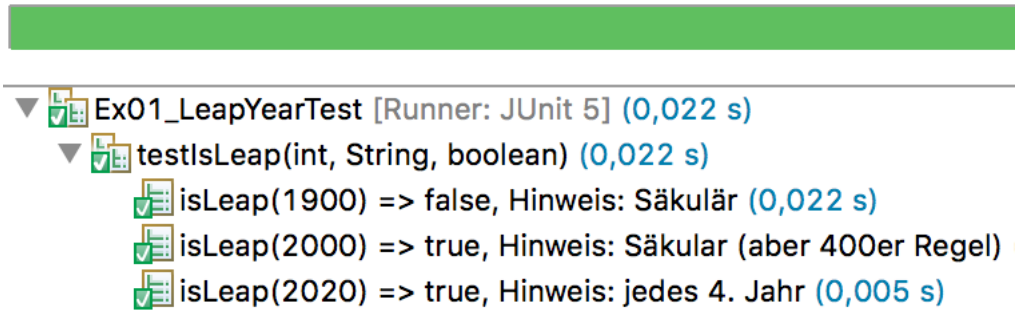
Dabei gibt es ein paar Spezialfälle sowie die 4er Regel:

* Ist eine Jahreszahl durch 4 teilbar, so ist es in der Regel ein Schaltjahr.
* Jahre, die durch 100 teilbar sind, werden Säkularjahre genannt und sind keine Schaltjahre.
* Allerdings sind Säkularjahre, die auch durch 400 teilbar sind, doch wieder Schaltjahre.

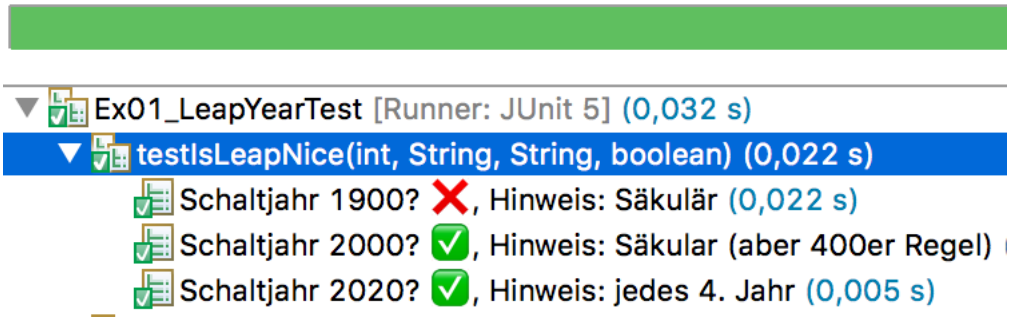
Aufgabe 3a: Diese Regeln kann man einzeln folgendermaßen prüfen, jedoch bietet sich seit JUnit 5 ein Parametrized Test geradezu an. Wandle diese in einen solchen um, beginne mit `@ValueSource(ints):`

```
@Test
void testIsLeap_4_Years_Rule()
{
      final boolean result = Ex01_LeapYear.isLeap(2020);

      assertTrue(result);
}
```

Aufgabe 3b: Es soll nicht nur eine einfache Prüfung erfolgen, sondern auch ein Hinweis, warum derart entscheiden wurde. Schreibe dazu eine Testmethode, die Ausgaben wie etwa die folgenden produziert:



Aufgabe 3c: Verbessere die Lesbarkeit wie folgt:



Aufgabe 4: addOne

Gegeben sei ein Array von Zahlen, die die Ziffern einer Zahl darstellen, etwa `[1, 2, 3, 4]` für den Wert 1234. Die Methode `int[] addOne(int[])` addiert den Wert eins auf diese Zahl und produziert für das Beispiel damit folgendes Ergebnis `[1, 2, 3, 5]`.

Aufgabe 4a: Der ursprüngliche Entwickler hat noch folgenden Unit Test hinterlassen und sich schnell aus dem Staub gemacht, weil dieser **ROT** war. Analysiere und behebe das Problem.

```
@Test
@DisplayName("[1, 2, 3, 4] + 1 => [1, 2, 3, 5]")
void testAddOne()
{
      final int[] values = { 1, 2, 3, 4 };
      final int[] expected = { 1, 2, 3, 5 };

      final int[] result = Ex04_AddOneToAnArray.addOne(values);

      assertEquals(expected, result);
}
```

Aufgabe 4b: Nachdem nun ein erster Test als Basis besteht, sollten weitere Testfälle ergänzt werden. Überlege, welche Besonderheiten die Addition besitzt und welche Testfälle benötigt werden, um diese zu prüfen. und wie man das `int[]` geeignet dem Test übergeben kann.

Tipp: Verwende eine `@MethodSource`.

Aufgabe 5: Add Roman Numbers

Gegeben sei eine Klasse `Ex05_RomanNumbers` zur Umwandlung in und aus römischen Zahlen mit den zwei Methoden:

* `int fromRomanNumber(String)`
* `String toRomanNumber(int)`

Praktischerweise sind beide schon gut mit Unit Tests geprüft.

Nun kommt aber das Business und möchte eine Additionsfunktionalität anbieten. Entwickle diese in einer Klasse `Ex05_RomanNumberAdder` und stütze dich auf Unit Tests ab. Das Business ist bereit, eine CSV-Datei `roman-addition.csv` beizusteuern, die gekürzt wie folgt aussieht:

```
Roman      Roman      Summe      Berechnung = Wert
-------------------------------------------------
I,         I,         II,        1 + 1 = 2
I,         II,        III,       1 + 2 = 3
I,         III,       IV,        1 + 3 = 4
I,         IV,        V,         1 + 4 = 5
V,         II,        VII,       5 + 2 = 7
V,         IV,        IX,        5 + 4 = 9
X,         VII,       XVII,      10 + 7 = 17
X,         XX,        XXX,       10 + 20 = 30
```

Aufgabe 6: Gehaltszahltag mit Hinweis

Gegeben sei eine Berechnung des nächsten Gehaltszahltags basierend auf einem `LocalDate`. Dabei gelten folgende Regeln:

1) Das Gehalt wird am 25. Des Monats ausgezahlt, im Dezember in der Mitte des Monats.
2) Fällt der Gehaltszahltag auf ein Wochenende, so ist der Freitag davor der Tag der Auszahlung, im Dezember jedoch der darauffolgende Montag.

Dazu wurde eine Klasse `NextPaydayAdjuster` geschrieben. **Überlege dir die notwendigen Testfälle und erstelle passende Unit Tests** mit einem verständlichen Hinweis, etwa „Freitag, falls 25. am Wochenende“.

Tipp 1: Korrigiere die Implementierung, falls durch Testfälle Fehler entdeckt werden.
Tipp 2: Verwende eine `@CsvSource` und modifiziere das Trennzeichen.

Aufgabe 7: Rabattberechnung

Es soll für Artikel der Rabatt basierend auf der Menge der Bestellung ermittelt werden. Gegeben sei folgende Anforderung aus dem Business:

Wertebereich	Rabatt
count < 50	0 %
50 <= count <= 1000	4 %
count > 1000	7 %

Zudem hat ein fleißiger Entwickler schon einmal folgende Implementierung bereitgestellt:

```
public class Ex07_DiscountCalculator
{
	// ACHTUNG: Enthält bewusst ein paar kleine Fehler
	public int calcDiscount(final int count)
	{
		if (count < 50)
			return 0;
		if (count > 50 && count < 1000)
			return 4;
		if (count > 1000)
			return 7;

		throw new IllegalStateException("programming " +
			"problem: should never reach this line.“ +
		    "value " + count + " is not handled!");
	}
}
```

Aufgabe 7a: Prüfe die Implementierung der Äquivalenzklassentests und vereinfache diese mithilfe von Parameterized Tests.

Aufgabe 7b: Ergänze die exemplarisch angegebenen Grenzwerttests. Damit sollte man Fehler an Rändern entdecken können. Korrigiere die Implementierung, sofern nötig. Vereinfache die Grenzwerttests durch Einsatz von Parameterized Tests.

Aufgabe 7c: Ergänze eine Prüfung auf positive Werte und sichere dies durch einen passenden Test ab.

Aufgabe 7d: Variiere die Rückgabe, sodass statt einer Zahl Werte aus dem Enum Discounts als Rückgabe dient. Erstelle die Klasse `Ex07_DiscountCalculator_WithEnum`. Passe die Tests so an, dass jeweils Enum-Werte für Parametrized Tests genutzt werden.

Tipp: **`@MethodSource`**

Aufgabe 8: Argument Converter

Manchmal sind die von JUnit 5 bereitgestellten Konvertierungen nicht ausreichend. Es lassen sich aber als Abhilfe auf einfache Weise eigene Konverter erstellen.

Aufgabe 8a: Schreibe einen einfachen eigenen HexConverter, der die Angabe hexadezimaler Zahlen für folgenden Test ermöglicht — wenn du Binärzahlen magst, dann ergänze noch den zweiten Konverter.

```
@ParameterizedTest
@CsvSource({ "F, 15", "10, 16", "AA, 170", "FF, 255" })
void hexConverter(@ConvertWith(HexToInt.class) int input,
                  int expected)
{
      assertEquals(expected, input);
}

@ParameterizedTest
@CsvSource({ "1, 1", "10, 2", "111, 7", "11111111, 255" })
void binaryConverter(@ConvertWith(BinaryToInt.class) int input,
                     int expected)
{
      assertEquals(expected, input);
}
```

Aufgabe 8b: Schreibe einen ArgumentConverter, der eine String-Repräsentation eines Arrays oder einer Liste, also `[Wert1, Wert2, Wert3]` in eine Liste von gewünschten Werten transformiert, im nachfolgenden Beispiel aus den String-basierten Datumswerten in eine `List<LocalDate>`:

```
@ParameterizedTest(name = "sundays between {0} and {1} => {2}")
@MethodSource("startAndEndDateAndArrayResults")
void sundaysBetween(LocalDate start, LocalDate end,
                    @ConvertWith(FromStringArrayConverter.class)
List<LocalDate> expected)
{
    final List<LocalDate> result =
          SundayCalculator.allSundaysBetween(start, end);

    assertEquals(expected, result);
}

private static Stream<Arguments> startAndEndDateAndArrayResults()
{
    return Stream.of(Arguments.of(LocalDate.of(2020, 1, 1),
                                  LocalDate.of(2020, 3, 1),
          "[2020-01-05, 2020-01-12, 2020-01-19, 2020-01-26," +
          " 2020-02-02, 2020-02-09, 2020-02-16,2020-02-23]"));
}
```

Aufgabe 9: JSON Argument Converter

Manchmal sind die von JUnit 5 bereitgestellten Konvertierungen nicht ausreichend. Es lassen sich aber als Abhilfe auf einfache Weise eigene Konverter erstellen. In dieser Aufgabe soll ein JSON-Konverter erstellt werden.

Tipp: Nutze die externe Bibliothek GSON.
https://mvnrepository.com/artifact/com.google.code.gson/gson

```
@ParameterizedTest
@CsvSource(value = {
"{ name:'Peter', dateOfBirth: '2012-12-06', homeTown : 'Köln'} |
false",
"{ name:'Mike', dateOfBirth: '1971-02-07', homeTown : 'Zürich'} | true"
}, delimiter = '|')
void jsonPersonAdultTest(@ConvertWith(JsonToPerson.class)
                         Person person, boolean expected)
{
      final long age = ChronoUnit.YEARS.between(person.dateOfBirth,
                                                LocalDate.now());

      assertEquals(expected, age >= 18);
}

static class JsonToPerson extends SimpleArgumentConverter
{
      @Override
      protected Person convert(Object source, Class<?> targetType)
      {
            // TODO
            return null;
      }
}
```

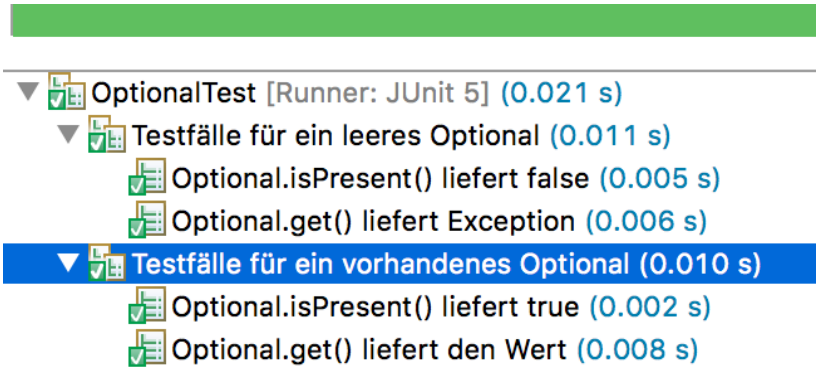
Aufgabe 10: Benchmark Extension

Erstelle eine Extension, um die Laufzeiten der einzelnen Testfälle zu ermitteln. Benutze beispielsweise Aufrufe an `FibonacciCalculator.fibRec(47)`, um längere Laufzeiten zu provozieren. Begrenze bei einem weiteren Testfall die Laufzeit auf maximal 2 Sekunden.

Tipp: **`@ExtendWith(BenchmarkExtension.class)`**

Aufgabe 11: Nested Test

Schreibe einen Test, der die wesentlichen Funktionalitäten der Klasse `Optional<T>` prüft (https://www.viadee.de/wp-content/uploads/JUnit5_javaspektrum.pdf).



Aufgabe 12: AssertJ

Gegeben sei eine Liste von Personen und eine Menge an Komparatoren.

```
private static final List<Person> persons =
                                    new ArrayList<>(List.of(
 new Person("Mike",  LocalDate.of(1971,  2,  7), "Bremen"),
 …
 new Person("Tom",   LocalDate.of(2011, 11, 11), "Aachen")));

private final Comparator<Person> byName =
              Comparator.comparing(Person::getName);
private final Comparator<Person> byBirthday =
              Comparator.comparing(Person::getDateOfBirth);
```

Um die jeweiligen Sortierungen zu testen, existieren drei Testfälle, die jeweils die gewünschten Ergebnisse definieren.

Aufgabe 12a: Wie lässt sich das Ganze mithilfe von AssertJ deutlich kürzer und einfacher gestallten?

Aufgabe 12b: Was ist der grosse Vorteil an der mit AssertJ umgesetzten Prüfung.

Aufgabe 13: Permutationen

In der Klasse `Ex09_StringUtils` gibt es eine Methode `calcPermutations(String)`, die zu einem gegebene Text alle Permutationen ermittelt, also etwa:

A	=>	A
AB	=>	AB, BA
ABC	=>	ABC, ACB, BAC, BCA, CAB, CBA

Aufgabe 13a: Schreibe Unit Tests für diese drei Fälle.

Aufgabe 13b: Vereinfache dies durch einen Parameterized Tests. Überlege, ob hier eine andere Form der Parameterbereitstellung in Frage kommt. (`@MethodSource`)

Aufgabe 13c: Schnell wird die Ergebnismenge recht groß, weil die Anzahl der Fakultät der Länge des Strings entspricht. Wie geht man etwa bei folgender Daten vor? Wie kann man alle Werte prüfen? Kann man das überhaupt? Was hat man für Alternativen

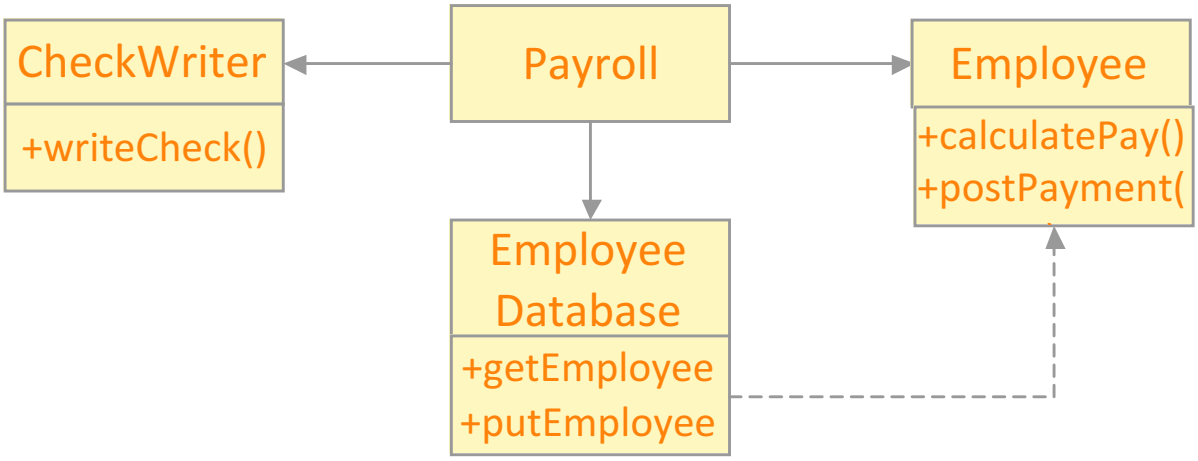
```
@Test
void testManyPermutations()
{
    final String input = "0123456789";
    final int expectedSizeBasedOnFaculty =
              MathUtils.fac(input.length());

    final Set<String> permutations =
                      Ex03_StringUtils.calcPermutations(input);

    assertEquals(expectedSizeBasedOnFaculty, permutations.size());
    // und nun??? wie alle die Werte prüfen?? Ideen
}
```

Aufgabe 13d: Bislang haben wir bei den Tests einen ziemlich wichtigen Spezialfall komplett außer Acht gelassen, nämlich, dass sich die Zeichenfolgen auch wiederholen können und sich dadurch die Anzahl der Kombinationen drastisch reduzieren kann. Tatsächlich gilt dann: Anzahl an Kombination = n! / k!, wobei n die Länge und k die Anzahl an Duplikaten ist. Schreiben Sie mit den gegebenen Hilfsmethoden beispielsweise `countDuplicates(String)` sowie `fac(int)` und den Information zur Anzahl der Kombinationen einen passenden Test für die Eingabe „AACAA“.

PART V/VI: Testweisen und Abhängigkeiten / Design For Testability — 60 — 75 min

Aufgabe 1: Sollbruchstellen — Design for Testability

Gegeben sei folgendes Klassendiagramm, dass noch einige direkte Abhängigkeiten enthält. Wie und wo muss man am Design etwas ändern, um die Klasse `Payroll` besser testbar zu machen: Modifiziere das Design so, dass die Klasse `Payroll` unabhängig von der Datenbank und anderen Klassen testbar wird. (Beispiel stammt aus „Agile Software Development“ von Robert C. Martin)



Aufgabe 2: Extract And Override

Gegeben sei die folgende Klasse `PizzaService`.

```
public class PizzaService
{
    private final SmsNotificationService notificationService;

    public PizzaService()
    {
        notificationService = new SmsNotificationService();
    }

    public void orderPizza(final String name)
    {
        notificationService.send("Pizza " + name +
                                 " wird in Kürze geliefert.");
    }
}
```

Durch die Art ihrer Implementierung ist sie aber nicht besonders gut testbar: Sie versendet bei Bestellungen eine SMS-Benachrichtigung – wobei vereinfachend der SMS-Versand durch die Darstellung eines Dialogs simuliert wird:

Wollten wir dieses Konstrukt testen, so ist dies ziemlich problematisch:

1) Sicherlich soll nicht bei jedem Testlauf eine SMS versendet werden.
2) Auch das Darstellen eines Dialogs behindert die Automatisierung von Tests.

Versuchen wir trotzdem einen Unit Test zu schreiben und erinnern uns an die ARRANGE-ACT-ASSERT-Struktur:

```
public class PizzaServiceTest
{
    @Test
    public void orderPizza_should_send_sms()
    {
        // Arrange
        final PizzaService service = new PizzaService();

        // Act
        service.orderPizza("Diavolo");
        service.orderPizza("Surprise");

        // Assert ???
    }
}
```

An dem Test sieht man Verschiedenes: Mit zustandsbasiertem Testen (also dem Abfragen von Daten) können wir nicht prüfen, ob es zu einem Versand der Nachrichten gekommen ist, oder nicht. Zudem bestehen die oben genannten Negativpunkte weiterhin. Was können wir also machen? Wir müssen ...

1) eine Sollbruchstelle in der Klasse `PizzaService` einführen,
2) eine Stub-Implementierung für den `SMSNotificationService` erstellen und
3) einige Anpassungen im Unit Test vornehmen.

Aufgabe 3: Mockito First Steps

Gegeben sei eine einfache TestKlasse `Ex03_MockitoBasicsTest`, die an den jeweiligen Stellen erweitert werden muss, damit die Testfälle erfolgreich durchlaufen werden.

```
    @Test
    public void iterator_next_return_hello_world()
    {
        Iterator<String> it = Mockito.mock(Iterator.class);
        // TODO

        String result = it.next() + " " + it.next();

        assertEquals("Hello World", result);
    }
```

Aufgabe 4: Chat Applikation mit Mockito testen

Gegeben sei eine einfache Chat-Applikation mit folgenden Klassen:

```
public class MessageSender
{
     public String send(final String string)
     {
          return "- O K -";
     }
}

public class ChatEngine
{
    private final MessageSender messageSender;

    public ChatEngine(final MessageSender messageSender)
    {
        this.messageSender = messageSender;
    }

    public String say(final String message)
    {
        return messageSender.send(message);
    }
}
```

Die Klasse `ChatEngine` soll nun um einen Test erweitert werden, der beim Aufruf von `say("SECRET")` mit dem Text "`SERVICE`" antwortet.

Aufgabe 5: Datenbankzugriff mit Mockito abstrahieren

Gegeben sei die folgende Klasse `Ex05_PersonService`. Diese nutzt die Klasse `PersonDAO` zum Zugriff auf eine Datenbank und liefert die Domainklasse `Person`.

```
public class Ex05_PersonService
{
    private final PersonDao dao;

    public PersonService(PersonDao dao)
    {
        this.dao = dao;
    }

    public List<Person> findAll()
    {
        return dao.findAll();
    }

    public Person findById(int id)
    {
        return dao.findById(id);
    }
}


public class PersonDao
{
    public List<Person> findAll()
    {
        return Collections.emptyList();
    }

    public Person findById(int id)
    {
        return null;
    }
}
```

Um Unit Tests auch ohne Abhängigkeit oder laufende Verbindung zur Datenbank formulieren zu können, muss das `PersonDAO` geeignet mit Mockito simuliert werden.

Aufgabe RETURN-VALUE: Schreibe einen Test, der für einen Aufruf der Methode `findById(1)` eine gültige Person zurückgibt. Was passiert, wenn man nach der Id 2 abfragt?

Aufgabe EXCEPTION: Schreibe einen Test, der den Fehlerfall einer ungültigen Id prüfen soll. Für negative Ids soll eine `IllegalArgumentException` ausgelöst werden.

Aufgabe METHOD-CALLED: Schreibe einen Test, der prüft, ob eine spezielle Methode aufgerufen wurde. Das gewünschte Ergebnis ist eine Liste mit zwei Personen.

Aufgabe CALL-COUNT: Formuliere eine Überprüfung der Anzahl an Aufrufen für folgendes Code-Fragment für `findById()` und `findAll()` – prüfe nur für Id 1 auf exakte Übereinstimmung,

```
// Act
final Person person11 = service.findById(11);
if (person11 == null)
{
    final List<Person> result = service.findAll();
    final Person person1 = service.findById(1);
    final Person person2 = service.findById(2);
}
```

Tipp: Nutze `anyInt()` als Platzhalter für `findById()`.

Aufgabe 6: Mockito — Datumskonvertierung & Mock-Injection

Gegeben sei die folgende rudimentäre, unfertige Klasse `Arabic2RomanConverter`

```
public class Arabic2RomanConverter
{
      public String convert(int i)
      {
            return "" + i;
      }
}
```

Diese wird in einem Konvertierungsservice genutzt, der ein LocalDate in eine Darstellung mit römischen Zahlen umwandelt, also aus 7.2.1971 den Text VII-II-MCMLXXI erzeugt.

```
public class DateInRomanCharsService
{
   private Arabic2RomanConverter converter;

   public String getRomanDate(final LocalDate date)
   {
         String day = converter.convert(date.getDayOfMonth());
         String month = converter.convert(date.getMonthValue());
         String year = converter.convert(date.getYear());

        return String.format("%s-%s-%s", day, month, year);
   }
}
```

Schreibe einen Test `DateInRomanCharsServiceTest`, der sowohl den Konverter passend mocked als auch die entsprechenden Mocks mithilfe von Annotations erzeugt.

PART VII/VIII: Test Smells / Test Coverage — 45 — 60 min

Aufgabe 1: Test Smells & Test-Eleganz

Untersuche die Klasse `Ex01_VersionNumberUtilsTest` auf mögliche Test Smells. Vereinfache auch zu komplexe Tests und nutze dazu auch mit JUnit 5 eingeführte Features. Versuche etwa AssertJ-Funktionalität ähnlich zu

```
assertThat(mike).usingComparator(byAge).isGreaterThan(tim);
```

selbst zu bauen, um ein besseres Gespür für Juni 5 und sauberes Testing zu bekommen.

Nutze dazu als Ausgangsbasis folgende Methodensignatur:

```
private void assertComparator(IntPredicate expectedResult,
                              int compareResult,
                              String v1, String v2, String resultHint)
{
    // TODO
}
```

Diese soll für die bewusst fehlerhaften Eingaben sprechende Fehlermeldung produzieren, etwa:

```
or: Multiple Failures (2 failures)
comparing 3.5 with 3.1.72 should result in 3.5 < 3.1.72 ==> expected: <true> but was: <false>
comparing 3.1.72 with 3.5 should result in 3.1.72 > 3.5 ==> expected: <true> but was: <false>
```

Tipp: Nutze folgende Predicate:

```
IntPredicate IS_SAME = n -> n == 0;
IntPredicate IS_SMALLER_THAN = n -> n < 0;
IntPredicate IS_BIGGER_THAN = n -> n > 0;
```

Bonus: Nutze Assert J und Wandel die Stils-Klasse in einem Comparator<E>.

Aufgabe 2: Test Order

Die gegebene Testklasse `Ex02_StudyGroupTest` prüft nach kurzer Analyse ganz offensichtlich einen Ablauf an Aktionen. Spalte diese in passende kleinere Testmethoden auf, sodass dieses jeweils einen Aspekt testen.

Tipp: Nutze `@TestMethodOrder`.

Aufgabe 3: Testabdeckung

Gegeben Sei folgende Klasse `Ex03_ScoreCalculator`. Schreibe dazu Testfälle, sodass eine Testabdeckung von mindestens 70 und schließlich 100% erreicht wird.

```
public class Ex03_ScoreCalculator
{
      public static String calcScore(int score)
      {
            if (score < 45)
                  return "failed";
            else
            {
                  if (score > 95)
                        return "passed with distinction";

                  return "passed";
            }
      }
}
```

Aufgabe 4: Testabdeckung

Gegeben Sei folgende Klasse `Ex04_ArrayUtils` mit einer fehlerhaften Implementierung. Schreibe dazu Testfälle, sodass eine Testabdeckung von mindestens 80 und 100% erreicht wird.

```
public class Ex04_ArrayUtils
{
    public static int indexOf(int[] values, int searchFor)
    {
        if (values == null)
           throw new IllegalArgumentException("Array is null");

        int index = -1;
        for (int i = 0; i <= values.length; i++)
        {
            if (values[i] == searchFor)
            {
                index = i;
                break;
            }
        }
        return index;
    }
}
```

Aufgabe 5: Testabdeckung

Gegeben Sei folgende Klasse `Ex05_GreetingCreator` mit einer fehlerhaften Implementierung. Schreibe dazu Testfälle, sodass eine Testabdeckung von mindestens 80 und schließlich 100% erreicht wird.

```
public class Ex05_GreetingCreator
{
    public String createGreeting(final LocalTime time)
    {
        String message = "Good ";
        if (time.isBefore(LocalTime.of(12, 0, 0)))
        {
            message += "Morning";
        }
        else if (time.isAfter(LocalTime.of(12, 0, 0)))
        {
            message += "Afternoon";
        }
        else if (time.isAfter(LocalTime.of(18, 0, 0)))
        {
            message += "Evening";
        }

        return message;
    }
}
```