JUnit 5 im Überblick

Mehr Freude beim Testen

Jeder Entwickler kennt es aus leidvoller Erfahrung. Die gerade realisierte Funktionalität sträubt sich beharrlich, fehlerfrei zu sein. Selbst wenn wir uns sehr viel Mühe geben und gewissenhaft entwickeln, lassen sich Fehler nicht immer ausschließen. Zudem kann eine Fehlersuche anstrengend, mühsam und nervenaufreibend sein. Glücklicherweise existiert mit JUnit ein Framework, mit dem das Erstellen von Tests richtig Freude bereiten kann.

Dieser Artikel soll einen Eindruck der vielfältigen Möglichkeiten der neuesten Version 5.7.2 von JUnit vermitteln – tatsächlich gibt es noch einiges mehr zu entdecken, aber wir konzentrieren uns hier auf wesentliche, praxisrelevante Dinge.

Einstieg JUnit 5: Ein erster Blick auf Unit-Tests

Rekapitulieren wir zunächst ein wenig: Zum Testen einer Applikationsklasse schreiben wir mit JUnit normalerweise eine korrespondierende Testklasse. Häufig beginnt man damit, die wesentliche und zentrale Anwendungsfunktionalität (die relevanten Methoden) durch Tests zu überprüfen. Dazu besitzt eine Testklasse statt einer *main()*-Methode verschiedene Testmethoden, die mit der Annotation *@Test* gekennzeichnet sind.

assert()-Methoden im Überblick

Um die gewünschte Funktionalität zu prüfen, lassen sich Testbehauptungen mit Hilfe von speziellen *assertXyz()*-Methoden aufstellen, wie in Listing 1 gezeigt.

Listing 1

```
public class SomeAssertsInActionTest
{
  @Test
  void assertMethodsInAction()
  {
    String expected = "Tim";
    String actual = "Tim";

    assertEquals(expected, actual);
    assertEquals(expected, "XYZ", "Hint if wrong");
    assertTrue(true);
    assertTrue(true, "Always true");
    assertFalse(false);
    assertNull(null);
    assertNotNull(new Object());
    assertSame(null, null);
    assertNotSame(null, new Object());
  }
}
```

Das ist eher eine Machbarkeitsstudie bzw. Demonstration der vielfältigen Möglichkeiten, in der Praxis sollte man sich pro Testmethode auf eine oder wenige Prüfungen beschränken. Das trägt zur Kürze und Verständlichkeit bei. Zudem hält man Unit-Tests so wartbar und übersichtlich. Ebenso wird dann nur eine Funktionalität geprüft, wodurch die Aussagekraft gesteigert wird und es idealerweise nur einen Grund gibt, warum ein Testfall fehlschlägt.

Strukturierung von Testfällen: AAA-Stil

Als weitere Faustregel gilt, dass für eine zu testende Methode zumindest eine, oft aber besser mehrere Testmethoden existieren, die jeweils einzelne Testfälle repräsentieren. Gewöhnlich erfolgen in jeder Testmethode dazu drei Schritte (AAA-Stil):

- *Arrange:* Zunächst wird der gewünschte Startzustand, die Ausgangsbasis des Testfalls, auch Test-Fixture genannt, hergestellt.
- *Act:* Danach folgen dann idealerweise ein Aufruf oder selten auch mehrere Aufrufe von Methoden der zu testenden Klasse.
- *Assert:* Abschließend prüfen wir, ob das erwartete Ergebnis erzielt wurde. Dabei helfen verschiedene Prüfmethoden, unter anderem die bereits im einführenden Beispiel genutzten *assertEquals()*, *assertTrue()* und *assertNotNull()*, die man zur besseren Lesbarkeit der Testfälle statisch importieren sollte.

Statt Arrange-Act-Assert spricht man mitunter auch von Given-When-Then (GWT). Damit ergibt sich die in Listing 2 gezeigte Struktur einer Testmethode.

Listing 2

```
public class AAAStyleTest
{
  @Test
  void listAdd_AAAStyle()
  {
    // GIVEN: An empty list
    final List<String> names = new ArrayList<>();

    // WHEN: adding 2 elements
    names.add("Tim");
    names.add("Mike");

    // THEN: list should contain 2 elements
    assertEquals(2, names.size(), "list should contain 2 elements");
  }
}
```

Verbesserungen gegenüber Version JUnit 4.x

Praktischerweise bietet JUnit 5 eine Rückwärtskompatibilität zur Vorgängerversion JUnit 4. Um das zu ermöglichen, liegen die Definitionen der Klassen, Interfaces und Annotations in unterschiedlichen Packages: BeiJUnit 5 im Package *org.junit.jupiter.api* und bei JUnit 4 in *org.junit* bzw. *org.junit.Assert*.

Wenn man noch ein wenig genauer schaut, müssen die Testmethoden in JUnit 4 *public* sein. Zudem dürfen diese Methoden in JUnit 4 keine Parameter besitzen. Beide Einschränkungen fallen mit JUnit 5 weg. Es gibt nun sogar diverse Parameter, über die man sich Kontextinformationen von JUnit bereitstellen lassen kann. Schauen wir uns eins nach dem anderen an.

Variation der Testnamen: @DisplayName

Standardmäßig ergibt sich der Name eines Testfalls aus dem Namen der Testmethode. Mitunter ist das eine ziemliche Einschränkung, unter anderem weil Sonderzeichen oder Leerzeichen nicht möglich sind. MitJUnit 5 kann man diese Beschränkung durch Einsatz der Annotation *@DisplayName* umgehen. Dadurch lassen sich auch Zeichen verwenden, die im Methodennamen nicht gültig wären, etwa Leerzeichen, spezielle Unicode-Zeichen oder auch die Zeichen *!* sowie +, -, / und *, die man etwa zur Benennung bei mathematischen Operationen nutzen kann (Listing 3). Dieser Test produziert die in **Abbildung 1** gezeigten Ausgaben.

Listing 3

```
public class DisplayNameTest
{
  @Test
  @DisplayName("This is a name with unicode \u1234")
  public void changeMyName()
  {
    assertTrue(true, "name is changed to that of @DisplayName");
  }

  @Test
  @DisplayName("Simple Math: (1 * 2 * 3) / 4 = 6/4 = 3/2 = 1.5")
  void divideResultOfMultiplication()
  {
    assertEquals((1 * 2 * 3) / 4d, 1.5, 0.1);
  }

  @Test
  @DisplayName("GET 'http://localhost:8080/products/4711'" +
                  " user: Peter Müller")
  public void getProductFor4711()
  {
    // ..
  }
}
```

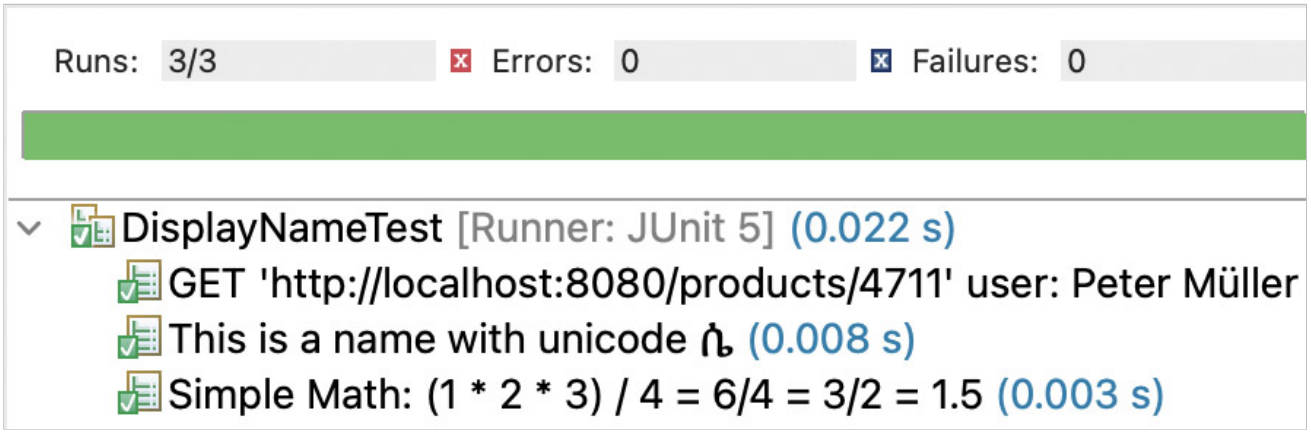

Abb. 1: Die Ausgaben des Tests aus Listing 3

Zugriff auf Metadaten: @TestInfo

Wie in Listing 4 gezeigt, lassen sich diverse Informationen zu einem Testfall durch JUnit 5 in Form des Interface *org.junit.jupiter.api.TestInfo* injizieren, etwa um daraus den Methodennamen oder weitere Metainformationen zu ermitteln. Betrachten wir in Listing 4 ein Beispiel, das unter anderem den Methodennamen, den Anzeigenamen sowie etwaige Tags ausliest und auf diese prüft.

Listing 4

```
public class InjectionOfMetaDataTest
{
  @Test
  @DisplayName("unicode \u1234")
  @Tag("very")
  @Tag("special")
  public void injectParams(final TestInfo testInfo)
  {
    assertEquals("injectParams", testInfo.getTestMethod().get().getName());
    assertEquals("unicode \u1234", testInfo.getDisplayName());
    assertEquals(Set.of("very", "special"), testInfo.getTags());
  }
}
```

Multi-Asserts: assertAll()

Bei der Formulierung von Testfällen sind immer mal wieder mehrere Bedingungen zu prüfen. Das scheint gegen die Regel „ein Testfall, ein Assert“ zu verstoßen. Allerdings sollte man das nicht immer allzu streng sehen, insbesondere dann nicht, wenn nur semantisch zusammengehörende Daten geprüft werden, wie etwa die Bestandteile einer Adresse (Listing 5).

Listing 5

```
@Test
void multipleAssertsforOneTopic_Diff1()
{
  Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");

  assertEquals("Tim", mike.name);
  assertEquals(LocalDate.of(1971, 3, 27), mike.dateOfBirth);
  assertEquals("Kiel", mike.homeTown);
}
```

Mit JUnit 5 lässt sich diese semantische Klammer durch die Methode *assertAll()* ausdrücken (Listing 6). Während wir uns im ersten Fall nach der Korrektur des ersten Fehlers weiter von Fehler zu Fehler hangeln müssen, erhalten wir im zweiten Fall mit *assertAll()* direkt eine Auflistung mit allen fehlschlagenden Asserts und können so das Ganze leichter korrigieren. Das verdeutlicht **Abbildung 2**.

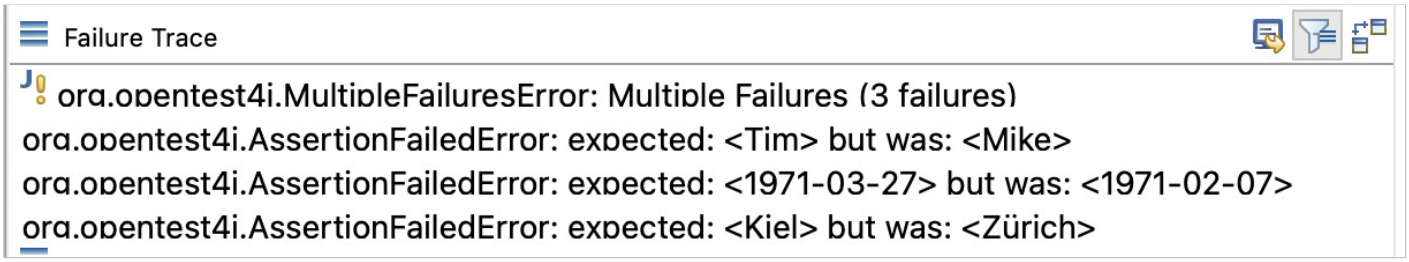

Abb. 2: Eine Auflistung aller fehlschlagenden Asserts

Listing 6

```
@Test
void multipleAssertsforOneTopic_Diff2()
{
  Person mike = new Person("Mike", LocalDate.of(1971, 2, 7), "Zürich");

  assertAll(() -> assertEquals("Tim", mike.name),
    () -> assertEquals(LocalDate.of(1971, 3, 27), mike.dateOfBirth),
    () -> assertEquals("Kiel", mike.homeTown));
}
```

Der Vollständigkeit halber zeige ich hier noch die Definition der Datenbehälterklasse bzw. des Records (mit modernem Java 17):

```
record Person(String name, LocalDate dateOfBirth, String homeTown) {}
```

Erwartete Exceptions prüfen: assertThrows()

Beim Testen sollte man nicht nur den Happy Path im Kopf haben. Um für Qualität und Zuverlässigkeit zu sorgen, sollte Businessfunktionalität auch mit ungültigen Werten, ungewöhnlichen Konstellationen usw. umgehen können. Genauer: Darauf adäquat reagieren, etwa durch das Auslösen einer Exception.

Möchte man nur Derartiges testen, wird das Auftreten von Exceptions während der Abarbeitung der Aktionen eines Testfalls erwartet und ein Ausbleiben würde einen Fehler darstellen. Ein Beispiel dafür ist die Prüfung des bewussten Zugriffs auf ein nicht existentes Element eines Arrays. Eine *ArrayIndexOutOfBoundsException* sollte die Folge sein. Um erwartete Exceptions in einem Testfall so zu behandeln, dass sie einen Testerfolg und keinen Fehlschlag darstellen, bietet JUnit 5 die Methode *assertThrows()*. Sie schlägt fehl (produziert einen Testfehler), wenn die ausgeführte Methode nicht den erwarteten Typ von Exception auslöst. Zudem gibt die Methode die ausgelöste Exception zurück, sodass man noch weitere Prüfungen vornehmen kann, beispielsweise, ob im Text der Exception die gewünschten und erwarteten Informationen bereitgestellt werden (Listing 7).

Listing 7

```
public class AssertThrowsTest
{
  @Test
  void arrayIndexOutOfBoundsExceptionExpected()
  {
    var numbers = new int[] { 1, 2, 3, 4, 5, 6, 7 };

    assertThrows(ArrayIndexOutOfBoundsException.class, () -> {
      numbers[1_000] = 13;
    });
  }

  @Test
  void indexOutOfBoundsExceptionExpected()
  {
    var text = "Some example text, but too short";

    IndexOutOfBoundsException ioe = assertThrows(IndexOutOfBounds-Exception.class, () -> {
      System.out.println(text.charAt(1_000));
    });

    assertTrue(ioe.getMessage().startsWith("String index out"));
  }
}
```

Good Practices

Nachdem zuvor gute Angewohnheiten angeklungen sind, möchte ich diese nachfolgend kurz explizit thematisieren.

Nicht nur beim Programmieren, sondern insbesondere auch beim Testen empfiehlt es sich, sauberen, gut lesbaren, verständlichen und leicht wartbaren Source Code zu erstellen. Dabei helfen ein paar gute Angewohnheiten, um Testmethoden kurz, klar und verständlich halten:

1. Eine Testmethode sollte genau eine Funktionalität (oder nur einen Teil davon) prüfen. Aus Gründen der Nachvollziehbarkeit und Fokussierung sollte man möglichst wenige, im besten Fall nur eine *assertXyz()*-Methode zur Prüfung aufrufen.
2. Arrange – Act – Assert: Hiermit ist gemeint, dass die einzelnen Testfälle immer dem gleichen Muster folgen:
   - *Arrange:* Vorbedingungen und Initialisierungen (auch Test-Fixture genannt)
   - *Act:* Danach wird/werden eine (oder wenige) Aktion(en) ausgeführt
   - *Assert:* Schließlich wird geprüft, ob der erwartete Zustand eingetreten ist

3. Spezielle Namensgebung: Die Testfälle sollten sprechende Methodennamen besitzen, die etwas über die Eingabe und die erwarteten Resultate aussagen. Besonders wichtig ist es, dass der Testkontext gut beschrieben ist, also inklusive erwartetem Verhalten und möglichen Randbedingungen, in Etwa wie folgt:
   - *calcSum_WithValidInputs_ShouldSumUpAllValues()*
   - *calcSum_ThrowsException_WhenNullInput()*

Zwar weicht diese Namensgebung von den in Java sonst üblichen Konventionen ab, aber kann doch sehr zum Verständnis eines Testfalls beitragen. Obwohl diese Namenskonvention schon ein guter Schritt in die richtige Richtung ist, lassen sich mit JUnit 5 und der Annotation *@DisplayName* sehr aussagekräftige und auch ausgefallene Namen vergeben.

Positive Eigenschaften und Fazit

Kommen wir nach dieser kurzen Vorstellung einiger JUnit-5-Spezifika zu einem Fazit. Mit Unit-Tests lassen sich Einzelbausteine absichern und deren Qualität erhöhen. Das wiederum führt oft dazu, dass auch die Qualität des Gesamtsystems steigt. Das liegt unter anderem an Folgendem:

- Beim Schreiben von Unit-Tests werden auch Entwurfsentscheidungen überdacht, etwa solche zu Kohäsion, Kopplung und zum Design des API.
- Durch das Implementieren von Testfällen verwendet man das API der eigenen Klassen, wodurch die Beurteilung leichter fällt, ob die angebotenen Schnittstellen sinnvoll und handhabbar sind. Mögliche Schwachstellen der eigenen Klassen lassen sich noch vor einer Nutzung in anderen Komponenten aufdecken und man kann für gelungenere APIs sorgen.
- Die Unit-Tests dienen als Dokumentation des erwarteten Programmverhaltens, zudem ist diese Dokumentation automatisch immer aktuell, da die Testfälle ansonsten fehlschlagen würden.

Dadurch, dass die Unit-Tests noch während der Entwicklung geschrieben werden, lassen sich Fehler zudem frühzeitig finden und beheben. Das wird dadurch begünstigt, dass eine Ausführung immer ein eindeutiges Ergebnis liefert: Grün (bestanden) oder Rot (Fehler).

Weil sich zudem die Anzahl der Testfälle und die damit geprüften Programmteile (Testabdeckung) leicht auswerten lassen, ergibt sich ein recht gutes Bild über den Qualitätszustand der eigenen Klassen. Auch bilden die Unit-Tests ein gewisses Sicherheitsnetz, das vor Flüchtigkeitsfehlern bei Änderungen schützt.

Wie Sie sehen, sprechen viele gute Gründe dafür, mit dem Unit Testing zu beginnen oder es noch intensiver einzusetzen. Nach diesem Grundlagenartikel wenden wir uns im anschließenden Praxisartikel den sogenannten Parameterized Tests zu, die bei der Formulierung von Tests mit mehreren Wertebelegungen hilfreich sein können.

Michael Inden ist Oracle-zertifizierter Java-Entwickler mit über zwanzig Jahren Berufserfahrung bei diversen internationalen Firmen. Dort hat er in verschiedenen Rollen gearbeitet, etwa als Softwareentwickler, -architekt, Consultant, Teamleiter, CTO und Trainer. Derzeit ist er freiberuflich in Zürich tätig. Darüber hinaus schreibt er Fachbücher wie „Der Weg zum Java-Profi“, „Java – Die Neuerungen in Version 9 bis 14“ und das brandneue „Einfach Java“, alle im dpunkt.verlag erschienen.