Objektorientierte Programmierung

Testen mit JUnit 5

Prof. Dr. Ulrike Hammerschall
Fakultät für Informatik und Mathematik

Was ist Testen?

- Ziel ist das Auffinden von Fehlern in Programmen mit Hilfe geeigneter Testfälle.
- Testfälle:
  - Erlauben die Ausführen des Programms mit ausgewählten Testdaten.
  - Die erwarteten Ergebnisse (expected) für die konkreten Testdaten sind bekannt
  - Testmethoden vergleichen tatsächliche (actual) mit erwarteten Ergebnissen.
- Entscheidend für die Qualität des Ergebnisses ist eine gute Auswahl von Testfällen.
- Tests sind notwendig. Sie erhöhten das Vertrauen in die Korrektheit der Software, liefern aber keinen endgültigen Nachweis der Fehlerfreiheit.

Das JUnit Framework

HM

- Testframework zum Blackbox-Test von Java Klassen. Aktuelle Version JUnit 5.
- Entstand im Umfeld der agilen Methoden und der Entwicklung von Eclipse.
- Eigenständige Test-Bibliothek. Kein Bestandteil von Java oder einer Entwicklungsumgebung.
- Wird aber von allen gängigen Entwicklungsumgebungen (IntelliJ, Eclipse, …) unterstützt.
- Die Konzepte im Überblick:
  - Es gibt (grob) eine Testklasse pro zu testender Java-Klasse
  - Name der Testklasse laut Konvention:
    <Klasse unter Test>Test (Beispiel: *RationalTest*, AccountTest, etc.)
  - In der Testklasse i eine beliebige Menge von Testmethoden definiert
- Dokumentation und Tutorial unter:
  https://junit.org/junit5/docs/current/user-guide/

Aus der Vogelperspektive - Aufbau von JUnit-Testklassen

HM

- JUnit Testklassen sind normale Java-Klassen, die vom JUnit-Testframework ausgeführt werden.
- Annotationen (z.B. `@Test`) in der Testklasse liefern dem Framework alle Meta-Informationen, die zum Ausführen der Tests notwendig sind.
- Eine JUnit-Testklasse enthält:
  - Definierte Hilfsmethoden zum Vor- bzw. zur Nachbereiten der Testumgebung. Die Hilfsmethoden sind optional.
  - Beliebig viele Testmethoden zur Durchführung der eigentlichen Tests.
- Getestet wird mit Hilfe vorgegebener, vom Framework definierter statischer Hilfsmethoden (assert-Methoden) nach dem Blackbox-Prinzip.

Methoden und Annotationen der JUnit-Testklasse

```
class AllTestMethods {
    @BeforeAll
    static void initAll() {…} // Wird einmal zu Beginn des Tests ausgeführt.
    @BeforeEach
    void init() {…} // Wird vor jeder einzelnen Testmethode ausgeführt.
    @Test
    void succeedingTest() {…} // Normale Testmethode
    @AfterEach
    void tearDown() {…} // Wird nach jeder einzelnen Testmethode ausgeführt.
    @AfterAll
    static void tearDownAll() {…} // Wird einmal am Ende des Tests ausgeführt
}
```


JUnit 4 nach JUnit 5 – die wichtigsten Änderungen

- *JUnit 5 Annotationen:*
  - *@BeforeAll*
  - *@BeforeEach*
  - *@Test*
  - *@AfterEach*
  - *@AfterAll*
- *Die Klasse Assertions*
  - *org.junit.jupiter.api.Assertions*

- *JUnit 4 Annotationen:*
  - *@BeforeAll*
  - *@Before*
  - *@Test*
  - *@After*
  - *@AfterAll*

Zum Vergleich. Viele Testbeispiele Im Internet nutzen noch JUnit4

- *Die Klasse Assert*
  - *org.junit.Assert*

Schnittstellendokumentation der Klasse Assertions:
https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html

Aufbau und Regeln für Testmethoden

- Eine JUnit-Testklasse kann beliebig viele Testmethoden definieren.
- Testmethoden haben keine Parameter, der Ergebnistyp ist immer void.
- Die Namen der Testmethoden sind frei wählbar. Der Name sollte die Art / das Ziel des Tests kennzeichnen.
- Eine JUnit-Testklasse kann zusätzlich beliebig viele andere Methoden definieren (diese sind in der Regel *private* und dienen als reine Hilfsmethoden).
- Eine JUnit-Testklasse hat keinen Konstruktor.
- Es gibt keine main-Methode. Die Testmethoden werden direkt vom JUnit-Framework aufgerufen.
- Die Ausführungsreihenfolge der Methoden ist nicht festgelegt!

```
// Klasse unter Test
public class MathUtil {

    public int add(int x, int y) {
        return x + y;
    }

}
```

```
// JUnit Testklasse
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.Assertions;

class MathUtilTest {
    @Test
    void testAdd() {
        MathUtil mUtil = new MathUtil();
        Assertions.assertEquals(12, mUtil.add(4,
8));
    }
}
```


Ermittlung sinnvoller Testfälle

- Äquivalenzklassen-Test
  - ➢ Äquivalenzklassen pro Eingabeparameter und pro Ergebnis bilden:
    - Klasse aller gültigen Eingabewerte mit gleichem erwarteten Verhalten
    - Klasse aller ungültige Eingabewerte mit gleichem erwarteten Verhalten
  - ➢ Minimierung der erforderlichen Testfälle durch Auswahl eines Stellvertreters einer Äquivalenzklasse.
- Grenzwertanalyse
  - ➢ Auswahl der Vertreter von Äquivalenzklassen an Grenzen (Annahme: die meisten Fehler entstehen durch Randwerte).

Grundlage für gute und aussagekräftige Tests ist die Qualität der Testfälle.

Beispiel

- Die Methode;

```
int getMaxIndex(String[] array)
```

- erhält als Eingabe ein String-Array und liefert als Ergebnis den Index des längsten Strings im Array zurück.
- Falls mehrere Strings die gleiche maximale Länge haben, wird der Index des letzten längsten Strings zurückgegeben.
- Falls kein Element im Array enthalten oder das Array selbst nicht initialisiert (null) ist, wird -1 zurückgegeben.

Ableitung der Testfälle

- Äquivalenzklassen für das zu übergebende Array:

```
{null}
{Länge des Arrays 0}
{Länge > 0, einmaliges Vorkommen des längsten Strings}
{Länge > 0, mehrmaliges Vorkommen des längsten Strings}
```

- Ergebnismenge:

```
{-1, gültiger Index}
```

- Eine minimale Menge an Testfällen mit Grenzwertanalyse wäre beispielsweise:

```
null
[]
["T"]
["T", "T"]
```

- Weitere sinnvolle Testfälle könnten beispielsweise sein:

```
["das", "ist", "ein", "sinnvoller", "Testfall"]
["das", "ist", "ein", "besserer", "Testfall"]
```

Testabdeckung (Coverage)

- Anweisungsabdeckung (Line Coverage):
  - Jede Anweisung wird mindestens einmal ausgeführt. Bei if-Anweisung beispielsweise nur if- oder nur else-Zweig.
- Zweigabdeckung (Branch Coverage):
  - Anweisungsabdeckung ist erfüllt.
  - Jeder Zweig (Branch) wird mindestens einmal ausgeführt. Bei if-Anweisung werden sowohl if- als auch else-Zweig getestet.
- Pfadabdeckung
  - Jeder Pfad (Path) wird mindestens einmal durchlaufen (realistisch häufig nicht umsetzbar).
- Umsetzung:
  - Automatische Prüfung der Testabdeckung in der IDE (Run with Coverage). In der Regel wird damit Anweisungsabdeckung, manchmal auch Zweigabdeckung erreicht (abhängig vom verwendeten Coverage Plugin).
  - Hilfreich um eine minimale Abdeckung des Codes durch Tests sicherzustellen.