Software Tests

Objektorientiertes Programmieren

Lukas Wais

Codersbay

Version: 4. April 2023

Inhaltsverzeichnis

Arten von Software Tests

- **Unit Testing** testet einzelne Komponenten, unabhängig voneinander.
  - Eine Unit ist der kleinste testbare Teil der Software, der kompiliert, gelinkt, geladen und in einen Testrahmen gebracht werden kann, z.B. ein Java Objekt.
  - Diese Tests werden in der Regel von Programmieren im Rahmen vom Entwicklungsprozess umgesetzt.
- **Integration Testing** deckt Fehler in Schnittstellen und in den Interaktionen zwischen unterschiedlichen und separat getesteten Einheiten auf. Sobald diese integriert wurden.
  - Beispiele für Integration Bugs sind: inkorrekte Call oder Return Sequenzen, oder auch inkonsistente Datenvalidierungs Kriterien, ...
  - Sie werden in der Regel von Entwickler, Testern, oder beiden gemeinsam durchgeführt.
- **System Testing** testet ob integrierte Systeme die Anforderungen der Anwender/Kunden erfüllen.
  - Hier werden nicht funktionale Anforderungen wie: Performance, Sicherheit, oder Zuverlässigkeit getestet.
  - Getestet wird meist von speziellen Test Teams, aus Sicht der Endbenutzer.
- **Acceptance Tests** stellen Fest ob ein System die Anfoderungen erfüllt und ob der Kunde es abnimmt.

Blackbox Tests

Hier werden nur die Ein und Ausgabe betrachtet. Die interne Struktur und Funktionsweise ist irrelevant. Ich habe x als Eingabe und erwarte mir y als Ausgabe.

Whitebox Tests

Hier ist die interne Struktur und Funktionsweise bekannt und wird auch berücksichtigt. Hier wird Sichergestellt ob Arbeitsabläufe und Implementierungen korrekt und effizient sind.

Beispiele: Statement-, Decision-, Condition-, Path- und Mutation Coverage Tests.

Unittests

In Java wird dafür JUnit verwendet

- Der Name einer Testklasse endet mit *Test*. Beispiel: *VectorTest*.
  - Man kann spezielle Bedingungen zum Klassennamen hinzufügen. Beispiel: *EmptyVectorTest*.
- Testfälle (cases) sind annotierte Methoden in einer Testklasse.
- Der Name einer solchen Methode startet mit test und sollte aufschlussreich sein. Er soll das Verhalten beschreiben. Beispiel: *testRemoveElement()*.
  - Man fügt spezielle Testbedingungen zum Namen hinzu. Beispiel: *testAddNull()*.
- In einem Maven Projekt werden Tests im gleichnamigen Ordner gespeichert. Dort gibt es auch eine resources Ordner.

Eigenschaften Guter JUnit Tests

1. Namenskonventionen einhalten
2. Isolation: Jeder Test sollte nur eine Funktion oder Methode testen und nicht von anderen Tests oder der Umgebung beeinflusst werden.
3. Verständlichkeit: Der Test sollte einfach zu lesen und zu verstehen sein.
4. Reproduzierbarkeit: Der Test sollte jederzeit und unter allen Bedingungen das gleiche Ergebnis liefern. Das ist sehr wichtig für CI/CD.
5. Korrektheit: Die Assertions im Test sollten alle möglichen Fälle abdecken und sicherstellen, dass das Ergebnis dem erwarteten Ergebnis entspricht.
6. Lesbarkeit.
7. Schnelligkeit: Tests können die Test-Suite verlangsamen und den Feedback-Zyklus verlangsamen.

Beispiel JUnit Test klasse

```
class MyFirstJUnitJupiterTests {

  private final Calculator calculator = new Calculator();

  @Test
  void testAddition() {
    assertEquals(/* expected value */ 2, /* actual value */
    ↪  calculator.add(1, 1));
  }
}
```

Plattform Konzepte

- **Container:** ein Knoten in der Teststruktur, der andere Container oder Tests als seine Kinder enthält (z.B. eine Testklasse).
- **Test:** ein Knoten in der Teststruktur, der das erwartete Verhalten bei der Ausführung überprüft (z. B. eine @Test-Methode).

Jupiter Konzepte

- **Lifecycle Method:** jede Methode, die annotiert ist mit @BeforeAll, @AfterAll, @BeforeEach, oder @AfterEach.
- **Test Class:** jede Klasse die mindestens eine Testmethode enthält. Sie dürfen nicht abstract sein und müssen einen einzige Konstruktor haben.
- **Test Method:** jede Instanzmethode, die direkt mit @Test, @RepeatedTest, @ParameterizedTest, @TestFactory oder @TestTemplate annotiert ist. Mit Ausnahme von @Test erstellen diese einen Container in der Teststruktur, oder andere Container für @TestFactory.

JUnit 5 User Guide: https://junit.org/junit5/docs/current/user-guide/

Test Driven Development (TDD)

Zitat von Edwards Deming

"Quality cannot be tested into the product at the end. Quality must be built in or it will be absent. "

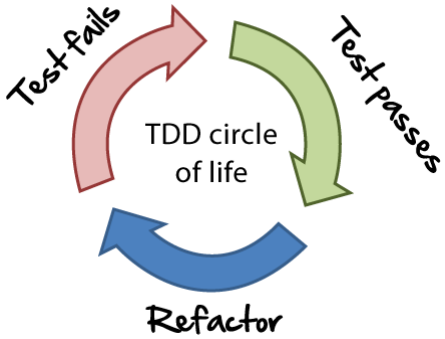
Phasen von TDD

1. Red: Schreibe einen kleinen Test der scheitert. Er wird vermutlich anfangs nicht kompilieren.

2. Green: Der Test soll funktionieren (schnell).

3. Refactor: der Code der grünen Phase wird aufgeräumt und verbessert. Die Funktionalität wird beibehalten und nicht verändert.

TDD Circle of Life



Source: https://neutrondev.com/can-you-become-a-programmer-without-learning-testing/test-driven-development-circle-of-life/

Beispiel TDD Prozess

Wir möchten eine Methode schreiben, welche zwei Integer addiert.

RED Phase

Test schreiben.

```
public class CalculatorTest {

  @Test
  public void testSum() {
    assertEquals(5, Calculator.add(2, 3));
  }
}
```

GREEN Phase

Funktion implementieren.

```
public class Calculator {
  public static int addAAndB(int a, int b) {
    int c = a + b;
    return c;
  }
}
```

Refactoring Phase

```java
public class Calculator {
  public static int add(int a, int b) {
    return a + b;
  }
}
```

Prüfen ob eine Exception geworfen wurde

```
@Test
public void testFooThrowsIndexOutOfBoundsException() {
        Throwable exception =
        ↪  assertThrows(IndexOutOfBoundsException.class, () ->
        ↪  foo.doStuff());
        assertEquals("expected messages", exception.getMessage());
}
```

Implementiere folgende Methoden in der Klasse StringCalculator und wende TDD an

- int add(String numbers): Diese Methode soll eine Zeichenkette von durch Kommas getrennten Zahlen entgegennehmen und die Summe aller Zahlen zurückgeben. Zum Beispiel sollte add("1,2,3") 6 zurückgeben. Die Methode muss auch Zeichenketten akzeptieren, die neue Zeilen enthalten, wie z.B. "1\n2,3". Das Ergebnis ist immer noch 6.

- int subtract(String numbers): Diese Methode soll das Ergebnis der Subtraktion aller Zahlen in der Zeichenkette von links nach rechts berechnen. Zum Beispiel sollte subtract("10,2,1") das Ergebnis von (10 - 2 - 1) = 7 zurückgeben.

- int multiply(String numbers): Diese Methode soll das Ergebnis der Multiplikation aller Zahlen in der Zeichenkette von links nach rechts berechnen. Zum Beispiel sollte multiply("2,3,4") das Ergebnis von (2 * 3 * 4) = 24 zurückgeben.

- boolean checkBrackets(String brackets): Diese Methode überprüft ob jede öffnende Klammer auch eine schließende besitzt. Zum Beispiel checkBrackets("{{{}}}") gibt true zurück. Wird ein anderes Zeichen als "{" oder "}"übergeben soll eine InvalidArgumentException mit dem Text "Wrong parameter, only { or } is allowed"geworfen werden.