Unit Tests

Datamining und Sequenzanalyse

Kai Dührkop, Markus Fleischauer

Warum?

"Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse."

— Michael Feathers *Working Effectively With Legacy Code* (2004)

Warum?

- Wartbarkeit
  - Keine Fehler (wieder) einführen
  - Jeder Bug ein neuer Test
- Flexibilität
- Dokumentation, Einarbeitung
- Design-Hilfe
  - **Test-driven development**

Wie?

- **JUnit** `(testImplementation 'junit:junit:4.13.2')`
- TestNG
- Eigenes Verzeichnis für Tests
  - `/test`, oder
  - `/src/main` + `/src/test`
- Java + Annotations

Wie?

```
public class MyListTest {

    @Test
    public void testEmptyListConstructor(){
        List<Integer> list = new MyList<>();

        assertTrue(list.isEmpty())
    }
}
```

Verschiedene Arten von Tests

- Unit Tests
  - Funktion einer Klasse
- Integration Tests
  - Zusammenspiel der Klassen bzw. Programmteilen

Gute UnitTests

- Jede Methode Testen (außer triviale getter/setter u.ä.)
- Grenzfälle testen
  - null
  - leere Objekte
  - Container mit nur einem Element
  - Container mit maximal erlaubter Anzahl Elemente
- Komplizierte Fälle Testen
- *Optimal: 1 Assert pro Test*
  - **1 Operation pro Test überprüfen**
- Sollten schnell Durchlaufen

AAA-Pattern[1]

```
@Test
public void testAddToList(){
    // Arrange
    List<Integer> list = new ArrayList<>();

    // Act
    list.add(1);

    // Assert
    assertEquals(list.size(), 1)
}
```

[1] William C. Wake, 2003

Tests auf Fehlerverhalten

```
@Test(expected = IndexOutOfBoundsException.class)
public void testRemoveFromEmptyList() throws
                                IndexOutOfBoundsException {
    // Arrange
    List<Integer> list = new ArrayList<>();

    // Act
    list.get(1);
}
```

Gute UnitTests

- Unabhängige Tests
  - Undefinierte Reihenfolge
  - Abhängigkeiten über Mocks/Stubs
- @BeforeClass, @BeforeMethod
- @AfterClass, @AfterMethod
- Parametrisierte Tests

Parametrisierte Tests

```
public class CalculatorTest {
  private Calculator calc = new Calculator();

  @Test
  public void testNfac() {
    assertEquals(1, calc.nfac(0));
    assertEquals(1, calc.nfac(1));
    assertEquals(2, calc.nfac(2));
    assertEquals(3628800, calc.nfac(10));
    assertEquals(2432902008l7L, calc.nfac(20));
  }
}
```

Parametrisierte Tests 1

```
@RunWith(Parameterized.class)
public class ParameterizedCalculatorTest {
  private static Calculator calc;
  private int input;
  private long expected;

  public ParameterizedCalculatorTest(int input, long expected) {
    this.input = input;
    this.expected = expected;
    calc = new Calculator();
  }

  @Parameters
  public static List<Object[]> data() {
    return Arrays.asList(new Object[][] { { 0, 1 }, { 1, 1 }, { 2, 2 },
                { 10, 3628800 }, { 20, 243290200817 } });
  }

  @BeforeClass
  public static void setUp() throws Exception {
    calc = new Calculator;
    calc.doSomeConfigStuff();
  }

  @Test
  public void testNfac() {
    assertEquals(expected, calc.nfac(input));
  }
}
```

Parametrisierte Tests 2

```
@RunWith(Parameterized.class)
public class ParameterizedCalculatorTest {
  private static Calculator calc;

  @Parameter(0)
  public int input;
  @Parameter(1)
  public long expected;

  @Parameters
  public static List<Object[]> data() {
    return Arrays.asList(new Object[][] { { 0, 1 }, { 1, 1 }, { 2, 2 },
                { 10, 3628800 }, { 20, 243290200817 } });
  }

  @BeforeClass
  public static void setUp() throws Exception {
    calc = new Calculator;
    calc.doSomeConfigStuff();
  }

  @Test
  public void testNfac() {
    assertEquals(expected, calc.nfac(input));
  }
}
```

Algorithm Tests (Beispiele)

- Gegen Naiven Algorithmus vergleichen
- Spezialfälle, z.B.:
  - Pattern in Leerer Sequenz
  - Leeres Pattern in Sequenz
  - Pattern = Sequenz
  - Pattern am Anfang/Ende der Sequenz
- Größerer (Zufalls-) Datensatz
  - nicht jedes Fehlverhalten kann mit toy-examples aufgedeck werden

Test-driven development

Erst Tests, dann Features schreiben:
Test-Code $\rightarrow$ Fail(red) $\rightarrow$ Feature-Code $\rightarrow$ Pass(green)

- Unterstützt gutes Design (testbarer Code)
- Feedback: Was ist schon fertig? (Tests passing)
- Aufteilung auf mehrere Programmierer
- Einfaches Debuggen