

# 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** (`testCompile group:'junit', name:'junit', version:'4.11'`)
- 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 Unit Tests

- 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<sup>1</sup>

**@Test**

```
public void testAddToList(){  
    // Arrange  
    List<Integer> list = new ArrayList<>();  
  
    // Act  
    list.add(1);  
  
    // Assert  
    assertEquals(list.size(), 1)  
}
```

---

<sup>1</sup>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 Unit Tests

- 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(243290200817L, 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
    public int input;
    @Parameter
    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

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

# Test-driven development

Erst Tests, dann Methoden schreiben

- Unterstützt gutes Design
- Feedback: Was ist schon fertig?
- Aufteilung auf mehrere Programmierer
- Einfaches Debuggen