Cheat Sheet: Testing mit JUnit + Mockito

Refresher JUnit

Vorarbeiten im Gradle-Build

```
apply plugin: 'java'
apply plugin: 'eclipse'

sourceCompatibility=1.8
targetCompatibility=1.8

repositories
{
      jcenter();
}

dependencies
{
      // JUnit 4 Support
      testCompile "junit:junit:4.12"
      testRuntimeOnly("org.junit.vintage:junit-vintage-engine:5.6.2")

      // JUnit 5
      testRuntimeOnly "org.junit.platform:junit-platform-launcher:1.6.2"
      testCompile "org.junit.jupiter:junit-jupiter-engine:5.6.2"

      // Assert J
      testCompile 'org.assertj:assertj-core:3.17.1'

      // Parameterized tests
      testCompile("org.junit.jupiter:junit-jupiter-params:5.6.2")

      // Migration Support to Enable Rules in JUnit 5
      testCompile "org.junit.jupiter:junit-jupiter-migrationsupport:5.6.2"
}
```

Gewünschte Eigenschaften von Unit Tests

- ⇨ **F** - Fast
- ⇨ **A** - Automated
- ⇨ **I** - Isolated
- ⇨ **R** – Reliable, Repeatable

Empfohlene Strukturierung: Arrange Act Assert

```
@Test
public void add_should_calc_sum() {

    // Arrange
    int[] inputPair = { 1, 1 };
    int expectedResult = 2;

    // Act
    final int calculatedResult = new Adder().add(inputPair[0], inputPair[1]);

    // Assert
    assertEquals(expectedResult, calculatedResult);
}
```

```
@Test
public void add_should_calc_zero_for_same_positive_and_negative_value() {

    // Arrange
    int[] inputPair = { 7, -7 };
    int expectedResult = 0;

    // Act
    final int calculatedResult = new Adder().add(inputPair[0], inputPair[1]);

    // Assert
    assertEquals(expectedResult, calculatedResult);
}
```

Basis-Technik: Extract and Override

Die Technik Extract and Override dient dazu Sollbruchstellen in den Code einzufügen, um danach die Testbarkeit durch Ableiten und Überschreiben zu erreichen.

Betrachten wir als Beispiel eine Klasse `PizzaService`, die bei Bestellungen eine SMS-Benachrichtigung versendet. Das wird wie folgt realisiert – wobei vereinfachend der SMS-Versand durch die Darstellung eines Dialogs simuliert wird:

```
public class PizzaService
{
	private final INotificationService notificationService;

	public PizzaService()
	{
		notificationService = new SmsNotificationService();
	}

	public void orderPizza(final String name)
	{
		notificationService.notify("Pizza " + name +
					" wird in Kürze geliefert.");
	}
}

public class SmsNotificationService
{
	public void notify(String msg)
	{
		// send sms
		JOptionPane.showConfirmDialog(null, msg);
	}
}
```

Wollten wir dieses Konstrukt testen, so ist dies ziemlich problematisch:

1) Sicherlich soll nicht bei jedem Testlauf eine SMS versendet werden.
2) Auch das Darstellen eines Dialogs behindert die Automatisierung von Tests.

Versuchen wir trotzdem einen Test zu schreiben:

```
public class PizzaServiceTest
{
	@Test
	public void orderPizza_should_send_sms()
	{
		// Arrange
		final PizzaService service = new PizzaService();

		// Act
		service.orderPizza("Diavolo");
		service.orderPizza("Surprise");

		// Assert ???
	}
}
```

An dem Test sieht man, dass wir die Auswirkungen nicht prüfen können und dass zudem immer noch die oben genannten Negativpunkte existieren. Was können wir also machen?

Sollbruchstelle einführen

Um dem Problem Herr zu werden, muss man folgende Dinge tun:

1) Wir müssen eine Sollbruchstelle einführen.
2) Wir müssen eine Stub-/Fake-Implementierung für den `SmsNotificationService` erstellen.
3) Anpassungen im JUnit-Test vornehmen

Sollbruchstelle einführen

```
public class PizzaServiceV2
{
	private final SmsNotificationService notificationService;

	public PizzaServiceV2()
	{
		notificationService = getNotificationService();
	}

	public void orderPizza(final String name)
	{
		notificationService.notify("Pizza " + name +
					" wird in Kürze geliefert.");
	}

	protected SmsNotificationService getNotificationService()
	{
		return new SmsNotificationService();
	}
}
```

Stub-/Fake-Implementierung erstellen

Zum Protokollieren und Nachvollziehen der Änderungen erstellen wir folgenden Stub/Fake, der die zu versendenden Nachrichten in einer Liste speichert und diese für die Prüfung in Testfällen wieder bereitstellen kann.

```
public class StubNotificationService extends SmsNotificationService
{
	private final List<String> messages = new ArrayList<>();

	public void notify(String msg)
	{
		messages.add(msg);
	}

	public List<String> getMessages()
	{
		return Collections.unmodifiableList(messages);
	}
}
```

Anpassungen im JUnit-Test vornehmen

```
public class PizzaServiceV2Test
{
	@Test
	public void orderPizza_should_send_sms()
	{
		// Arrange
		final StubNotificationService stub =
						new StubNotificationService();
		final PizzaServiceV2 service = new PizzaServiceV2()
		{
		     @Override
		     protected SmsNotificationService getNotificationService()
		     {
		         return stub;
		     }
		};

		// Act
		service.orderPizza("Diavolo");
		service.orderPizza("Surpirse");

		// Assert
		assertTrue(stub.getMessages().contains(
				"Pizza Diavolo wird in Kürze geliefert."));
		assertThat(stub.getMessages(), hasItem(
				"Pizza Surpirse wird in Kürze geliefert."));
		assertThat(stub.getMessages(), hasItems(
				"Pizza Surpirse wird in Kürze geliefert.",
				"Pizza Diavolo wird in Kürze geliefert."));
	}
}
```

Hier sehen wir verschiedene Varianten mit `assertTrue()` und `assertThat()`, letztere sind zwar besser lesbar, aber die verwendeten Matcher deprecated – es gibt oftmals einen Konflikt mit Hamcrest.

Was ist an diesem Test aber noch unschön und insbesondere fragil?

- ⇨ Die textuellen Vergleiche arbeiten auf exakter Übereinstimmung
- ⇨ Die Tests brechen bei jeder kleinen Änderung

Was kann man dagegen tun?

- ⇨ Der Service um eine Methode ergänzen:

```
public String createNotificationMsg(final String name)
{
	return "Pizza " + name + " wird in Kürze geliefert.";
}
```

Damit wird der Test unabhängig von Text-Änderungen:

```
assertTrue(stub.getMessages().contains(service.createNotificationMsg("Diavolo")));
assertThat(stub.getMessages(),hasItem(service.createNotificationMsg("Surpirse")));
assertThat(stub.getMessages(), hasItems(service.createNotificationMsg("Surpirse"),
                                        service.createNotificationMsg("Diavolo")));
```

MOCKITO und Junit 5

Vorarbeiten im Gradle-Build

```
dependencies
{
	// JUnit Dependencies
	// ..

	// Mockito
	testCompile group: 'org.mockito:mockito-core:3.5.10'
	testCompile group: 'org.mockito:mockito-junit-jupiter:3.5.10'

}
```

Mockito Basics

- Mock() => Mock / Stub erstellen
- When().thenReturn() => Expectations einer Rückgabe formulieren

```
import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

public class MockitoHelloWorldExample
{
   @Test
   public void testGreetingReturnValue()
   {
	// Arrange
	final Greeting greeting = mock(Greeting.class);
	when(greeting.greet()).thenReturn("Hello Mockito");

	// Act
	final String result = greeting.greet();

	// Assert
	assertEquals("Hello Mockito", result);
   }

   private class Greeting
   {
	public String greet()
	{
		return "Hello world!";
	}
   }
}
```

Exceptions bei ungewünschten Eingaben auslösen (when().thenThrow())

```
public class MockitoThrowingExample
{
    @Test
    public void testGreetingReturnValue()
    {
        // Arrange
        final Greeting greeting = mock(Greeting.class);
        // Achtung: Reihenfolge wichtig
        when(greeting.greet(anyString())).thenReturn("Hello Mockito");
        when(greeting.greet("ERROR")).thenThrow(
                                        new IllegalArgumentException());
        // when(greeting.greet(anyString())).thenReturn("Hello Mockito");

        // Act
        final String result1 = greeting.greet("Mike");
        final String result2 = greeting.greet("ERROR");
    }

    private class Greeting
    {
        public String greet(final String name)
        {
            return "Hello " + name;
        }
    }
}
```

Mehrere Rückgaben + Häufigkeit

- `thenReturn(…)` => mehrere Rückgaben vorgeben
- `thenReturn()`-Chaining => mehrere Rückgaben vorgeben
- `times(), atLeast(), atMost()` => Häufigkeit abfragen
- `verifyNoMoreInteractions()` => weitere Aktionen unterbinden

```
public class MockitoCallTimesExample
{
    @Test
    public void testGreetingCallCount()
    {
        // Arrange
        final Greeting greeting = mock(Greeting.class);
        when(greeting.greet()).thenReturn("Hello Mockito1", "Hello Mockito2");

        // Act
        final String result1 = greeting.greet();
        final String result2 = greeting.greet();
        final String result3 = greeting.greet();
        // greeting.additional();

        // Assert
        assertEquals("Hello Mockito1", result1);
        assertEquals("Hello Mockito2", result2);
        assertEquals("Hello Mockito2", result3);
```

```
        Mockito.verify(greeting, atLeast(1)).greet();
        Mockito.verify(greeting, atMost(3)).greet();
        Mockito.verify(greeting, times(3)).greet();
        // Mockito.verifyNoMoreInteractions(greeting);
    }

    private class Greeting
    {
        public String greet()
        {
            return "Hello world!";
        }

        public String additional()
        {
            return "Hello world!";
        }
    }
}
```