Lernskript: Einführung in Mockito

Mockito ist ein "Mocking"-Framework, das es uns ermöglicht, "Test-Dummies" oder "Schauspieler" für die Abhängigkeiten einer Klasse zu erstellen. Dies ist entscheidend, um echte **Unit-Tests** zu schreiben, bei denen eine Klasse **isoliert** von ihren externen Abhängigkeiten (wie Datenbanken oder anderen Services) getestet wird.

Teil 1: Die grundlegende Analogie - Das "Warum"

- **Konzept:** Anstatt ein komplexes System (z.B. ein Auto-Airbag) mit all seinen echten, teuren und langsamen Abhängigkeiten (Aufprall-Sensor, Batterie) zu testen, schließt man es an eine Testbank an. Dort simuliert man die Signale der Abhängigkeiten mit einem Signal-Generator (dem **Mock**).
- **Zweck:** Mit **JUnit** stellen wir die Frage ("Hat der Airbag ausgelöst?"). Mit **Mockito** kontrollieren wir die Umgebung und simulieren die Antworten der Abhängigkeiten ("Der Sensor MUSS ein Aufprall-Signal senden.").

Teil 2: Praktische Anwendungsfälle - Das "Wofür"

Use Case 1: Verhalten einer Abhängigkeit simulieren (`when...thenReturn`)

- **Problem:** Eine Service-Klasse hängt von einem Repository ab, das auf eine Datenbank zugreift. Im Unit-Test wollen wir die Datenbank nicht starten.
- **Lösung:** Wir mocken das Repository und "trainieren" es, auf bestimmte Aufrufe mit vorhersehbaren Antworten zu reagieren.
- **Code-Beispiel:**

```
// 1. Erstelle den Mock
BenutzerRepository mockRepo = mock(BenutzerRepository.class);

// 2. Trainiere den Mock
// WENN die Methode 'existiert' mit "Peter" aufgerufen wird, DANN gib 'true'
zurück.
when(mockRepo.existiert("Peter")).thenReturn(true);

// 3. Gib den Mock in die zu testende Klasse
BenutzerService service = new BenutzerService(mockRepo);

// Jetzt wird der Aufruf von repository.existiert("Peter") im Service 'true'
zurückgeben,
// ohne jemals eine Datenbank zu berühren.
```

Use Case 2: Überprüfen, ob eine Methode aufgerufen wurde (verify)

```
Problem: Wir wollen nicht das Ergebnis einer Methode testen, sondern nur
sicherstellen, DASS sie aufgerufen wurde (z.B. das Senden einer E-Mail im
Fehlerfall).

Lösung: Wir mocken die Abhängigkeit und überprüfen (verify) nach der Aktion, ob
die gewünschte Methode auf dem Mock aufgerufen wurde.
```

NotificationService mockNotifier = mock(NotificationService.class); BestellService service = new BestellService(mockNotifier);

service.bestelleProdukt(...); // Aktion, die intern den Notifier aufrufen soll

// Überprüfe: Wurde die Methode 'sendeFehlerMail' auf dem Mock exakt 1x aufgerufen? verify(mockNotifier, times(1)).sendeFehlerMail(anyString());

Teil 3: Vertiefung (JavaMasta's Profi-Tipps)

```
Annotationen (@Mock, @InjectMocks): Dies ist der moderne, saubere Standard, um
Mocks zu erstellen und sie automatisch in die zu testende Klasse zu injizieren. Er
erfordert die @ExtendWith(MockitoExtension.class) über der Testklasse.
Generated java
```

@ExtendWith(MockitoExtension.class) class BenutzerServiceTest { @Mock // Erstellt automatisch einen Mock private BenutzerRepository mockRepo;

```
@InjectMocks // Erstellt ein echtes BenutzerService-Objekt und injiziert den Mock
private BenutzerService service;

// ...
```

}

Argument Matchers: Wenn der genaue Parameter eines Methodenaufrufs egal ist, verwendet man Matcher wie anyString(), anyInt(), any(Benutzer.class). Sie sind besonders nützlich bei when() und verify().

Unit-Test vs. Integrationstest:

```
Unit-Test (mit Mockito): Testet eine Klasse in kompletter Isolation. Schnell, aber
testet nicht das Zusammenspiel.

Integrationstest: Testet das Zusammenspiel mehrerer Komponenten, oft mit echten
(Test-)Abhängigkeiten wie einer In-Memory-Datenbank. Langsamer, aber näher an der
Realität.
```