Testen mit Pfefferminzgeschmack

Birgit Kratz
MT AG
Ratingen

Schlüsselworte

Unit-Tests, Mock-Objekte, TDD, BDD, Java

Einleitung
In Gesprächen mit Kollegen zum Thema Testen kommt immer wieder die Aussage, dass Tests generell als gut und notwendig angesehen werden. Jedoch wäre der Aufwand solche Tests zu schreiben unverhältnismäßig hoch und komplex, da eine Abhängigkeit zu vielen anderen System oder Klassen besteht.

In Unit-Tests sollen die einzelnen Testobjekte jedoch unabhängig und isoliert von ihrer Umgebung getestet werden, was es erforderlich macht, diese Umgebung durch Mock-Objekte zu ersetzen. Mock-Objekte fungieren dabei als Platzhalter (Test-Double) für die echten Objekte.

Mock-Frameworks bieten hervorragende Hilfestellungen um die Erstellung von Mock-Objekten zu vereinfachen. Eines dieser Mock-Frameworks ist „Mockito“, welches ich in diesem Vortrag näher vorstellen möchte.

Mockito
Mockito ist ein kleines und sehr leicht zu benutzendes Mock-Framework. Es besticht durch seine intuitive API, die das Erstellen aussagekräftiger Unit-Tests zum Kinderspiel macht. Es hat eine ähnliche Syntax wie EasyMock, ein Umstieg ist daher leicht.

Es gibt nur 2 Regeln:

- Stubbing kommt vor der Ausführung
- Verifizierung von Interaktionen kommt nach der Ausführung

Mocks erstellen
Im Gegensatz zu anderen Mock-Framworks kann Mockito auch Mock-Objekte zu konkreten Klassen und nicht nur (aber auch) zu Interfaces erstellen.

```
import static org.mockito.Mockito.*
MyClass mockedClass = mock(MyClass.class);
```

Mit Hilfe von Annotationen geht das sogar noch einfacher.

```
@Mock
MyClass mockedClass;
```

Mocks injizieren
Angenommen man hat eine zu testende Klasse, die per Injektion Abhängigkeiten zu anderen Klassen hat. Um diese Abhängigkeiten zu mocken, musste man bisher Konstruktoren bzw. getter- oder setter-Methoden schreiben, die nur in den Testklassen gebraucht wurden. Man schrieb also Klassen mit testspezifischem Code. Mit Mockito fällt diese Notwendigkeit weg.

Zu testende Klasse:

```
public class MyClass {

   @Inject
   private SomeService someService;

   public String doSomething () {
      boolean isItTrue = someService.isItTrue();

      if (isItTrue) {
         return "It's true";
      }
      return "It's false";
   }
}
```

Testklasse:

```
@RunWith(MockitoJUnitRunner.class)
public class MyClassTest {

   @Mock
   private SomeService someServiceMock;

   @InjectMocks
   private MyClass classUnderTest;

   @Test
   public void testDoSomething() {
      when(someServiceMock.istItTrue()).thenReturn(true);

      String answer = classUnderTest.doSomething();

      verify(someServiceMock).isItTrue();
      assertEquals("It's true", answer);
   }
}
```

Mockito wird nun versuchen `@InjectMocks` mit Hilfe von Konstruktor-Injection, Setter-Injection oder Field-Injection zu instanziieren. Im vorliegenden Beispiel wird Field-Injection benutzt.

Stubbing
Im letzten Beispiel sieht man schon ein kleines Stubbing Beispiel mit anschließender Überprüfung. Beim Stubbing wird festgelegt, wie das Mock-Objekt im jeweiligen Test-Kontext reagieren soll. Dazu gibt es folgend Konstrukte.

Methoden-Stubbing:

```
// wenn mock.someMethod() aufgerufen wird, soll der Mock den Wert 10
// zurück liefern.
when(mock.someMethod()).thenReturn(10);
```

Stubbing void-Methoden mit Exceptions:

```
// wenn mock.someVoidMethod() aufgerufen wird, soll eine RuntimeException
// geworfen werden.
doThrow(new RuntimeException()).when(mock).someVoidMethod();
```

Iterator-Style stubbing:

```
// Der erste Aufruf von mock.someMethod() liefert 1 zurück,
// der zweite 2
// und der dritte Aufruf sowie alle weiteren wirft eine
// RuntimeException (das letzte Stubbing gilt).
when(mock.someMethod())
   .thenReturn(1)
   .thenReturn(2)
   .thenThrow(new RuntimeException());
```

Weitere Stubbing Möglichkeiten:

- Stubbing mit Callbacks (erlaubt Stubbing mit dem generischen `Answer` interface)
- `doReturn()`, `doThrow()`, `doAnswer()`, `doNothing()`, `doCallRealMethod()`

Verifikation

Ein Mock-Objekt merkt sich alle Interaktionen. So ist es möglich, diese Interaktionen hinterher zu verifizieren.

```
// überprüft, dass mock.someMethod() aufgerufen wurde
verify(mock).someMethod();
```

Zusätzlich kann man die genaue, minimale, maximale Anzahl von Aufrufen auswerten, oder ob ein Aufruf überhaupt erfolgte.

```
verify(mock, times(2)).someMethod();     // genau 2 Mal
verify(mock, atLeast(2)).someMethod();   // mindestens 2 Mal
verify(mock, atMost(5)).someMethod();    // nicht mehr als 5 Mal
verify(mock, never()).someMethod();      // niemals, Alias für times(0)

verifyZeroInteractions(mock);            // keinerlei Aufruf am Mock-Objekt
```

Weitere Möglichkeiten sind die Auswertung der Reihenfolge von Aufrufen am Mock-Objekt, das Auffinden nicht gewollter Aufrufe oder eine Verifizierung mit Timeout.

Argument Matcher

Argument Matcher erlauben flexibles Stubbing oder Verifizieren. Will man bspw. eine Methode stubben die Parameter hat, dann muss man keine realen Werte als Parameter übergeben, sondern kann Argument Matcher verwenden.

```
// wenn die get-Methode am Mock-Objekt mit irgendeinem Integer-Wert
// aufgerufen wird, dann gib "true" zurück.
when(mock.get(anyInt()).thenReturn(true);


// wenn die put-Methode am Mock-Objeckt mit einem Objekt der Klasse
// "MyClass" aufgerufen wird, dann gib „false" zurück.
when(mock.put(any(MyClass.class))).thenReturn(false);
```

Gleiches gilt für das Verifizieren von Aufrufen.

```
// überprüft, dass mock.get() mit irgendeinem Integer-Wert aufgerufen wurde
verify(mock).get(anyInt());
```

Folgende Matcher sind bei Mockito schon dabei:

```
any(), any(java.lang.Class<T> clazz), anyBoolean(), anyByte(), anyChar(),
anyCollection(), anyCollectionOf(java.lang.Class<T> clazz), anyDouble(),
anyFloat(), anyInt(), anyList(), anyListOf(java.lang.Class<T> clazz),
anyLong(), anyMap(), anyMapOf(java.lang.Class<T> clazz), anyObject(),
anySet(), anySetOf(java.lang.Class<T> clazz), anyShort(), anyString(),
anyVararg()
```

Zusätzlich kann man eigene Matcher mit Hilfe von Hamcrest Matchern erstellen.

Spy(ing)

Manchmal kann es nötig sein, ein Objekt nicht komplett zu mocken, sondern teilweise gemockte Aufrufe zu verwenden, teilweise aber auch die realen Methoden („Partial Mocking"). Dies wird durch sogenannte „Spies" ermöglicht.

Ein Spy wird analog zu einem Mock via Annotation erzeugt.

```
@Spy
MyClass spy;
```

oder

```
MyClass spy = spy(new MyClass());
```

Optional können dann Methoden gestubbt werden. Für alle anderen Aufrufe werden die realen Methoden verwendet.

Partial Mocking ist aber auch mit echten Mocks möglich:

```
when(mock.someMethod()).thenCallRealMethod();
```

Aufrufparameter aufzeichnen

Mit Hilfe von ArgumentCaptor ist es möglich aufzuzeichnen, mit welchen Argumenten eine Methode aufgerufen wurde, um diese hinterher auszuwerten.

```
ArgumentCaptor<Person> personCaptor =
ArgumentCaptor.forClass(Person.class);
```

```
verify(mock, times(2)).doSomethind(personCaptor.capture());

List<Person> capturedPerson = personCaptor.getAllValues();
assertEquals("Birgit", capturedPerson.get(0).getName());
assertEquals("Klaus", capturesPerson.get(0).getName());
```

BDD
Für Freunde des Behaviour Driven Developments hat Mockito auch eine Lösung: BDDMockito. Diese Klasse erzeugt einen Alias, so dass der BDD Style für Tests (//given, //when, //then) verwendet werden kann.

Limitations
Das Mocken statischer Methoden sowie das Mocken von Konstruktoren ist mit Mockito nicht möglich. Aber es gibt Rettung in Form von PowerMock (http://code.google.com/p/powermock/). Es erweitert Mock-Frameworks (unter anderem Mockito) um genau diese Fähigkeiten.

Mockito im WorldWideWeb
http://mockito.org
Hier findet man die gesamte Dokumentation als Javadocs.

Kontaktadresse:
Birgit Kratz
MT AG
Balcke-Dürr-Allee 9
D-40882 Ratingen

Telefon:	+49 (0) 2102-309 61 0
Fax:	+49 (0) 2102-309 61 101
E-Mail	Birgit.Kratz@mt-ag.com
Internet:	www.mt-ag.com