FAKTOR ZEHN

Einführung in Mockito

Faktor-IPS Workshop 14.01.2011

Alexander Weickmann

Agenda

1. **Überblick**
2. **Mocks & Co**
3. **Verwendung**
4. **Weitere Beispiele**
5. **Best Practices & Common Pitfalls**
6. **Limitierungen**
7. **Fazit**

Agenda

1. **Überblick**
2. **Mocks & Co**
3. **Verwendung**
4. **Weitere Beispiele**
5. **Best Practices & Common Pitfalls**
6. **Limitierungen**
7. **Fazit**

1. Überblick

- **Was ist Mockito?**

  Framework dessen Einsatz

  - die Erstellung von Unit-Testfällen vereinfacht und Test-Code lesbarer macht
  - dem Programmierer beim Erstellen von Tests repetitive Tätigkeiten abnimmt
  - die Test-Performance erheblich steigern kann

1. Überblick

- **Herkunft**
  - Projekt erreichbar unter: http://www.mockito.org
  - Entwickelt von Szczepan Faber „und Freunden“
  - Release: 2008
  - Zu diesem Zeitpunkt gab es bereits Mocking-Frameworks, aber der Entwickler war mit diesen nicht zufrieden:

    „They spoil my TDD experience.“

1. Überblick

- **Features**
  - Intuitives API

    → keine Konstrukte wie *record*, stattdessen *when* … *then* und *verify*
  - Kann Interfaces und konkrete Klassen mocken
  - Verständliche Verifikations-Fehler und Stack-Traces
  - Argument Matchers für erwartete Parameterwerte (z.B. anyInt())
  - Tests folgen *setup* – *exercise* – *verify* Schema

1. Überblick

- **Firmen, die Mockito einsetzen**

  Liste im Wiki auf http://mockito.org, hier eine kleine Auswahl:

  - Sonar
  - Apache
  - RedHat
  - SpringSource

Agenda

1. **Überblick**
2. **Mocks & Co**
3. **Verwendung**
4. **Weitere Beispiele**
5. **Best Practices & Common Pitfalls**
6. **Limitierungen**
7. **Fazit**

2. Mocks & Co

- **Was ist überhaupt ein „Mock“?** *(1 / 2)*
  - Unter einem „Mock“ versteht man im Allgemeinen ein Dummy- oder Attrappen-Objekt, welches in Testfällen als Platzhalter für reale Objekte dient.
  - Englisch: to mock = etwas vortäuschen

2. Mocks & Co

- **Was ist überhaupt ein „Mock“?** *(2 / 2)*

```
public class A {
    public void methodWithB(B b) {
        …
    }
}
public interface B {
    public void anyMethod();
}
```

```
public void testMethodWithB() {
    A a = new A();
    B bMock = new B() {
        public void anyMethod() {
        }
    };
    a.methodWithB(bMock);
}
```

2. Mocks & Co

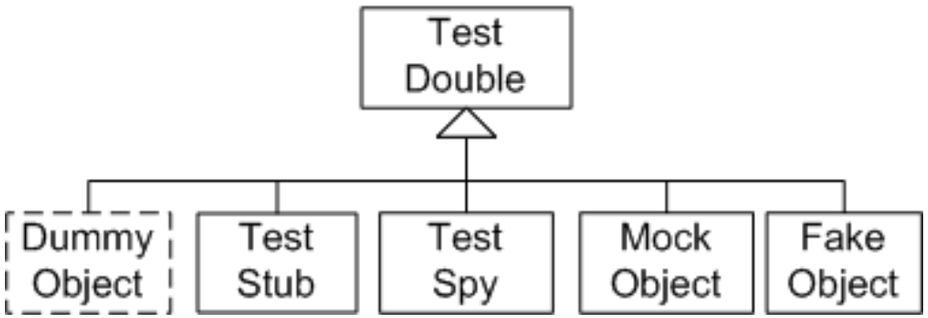
- **Wann braucht man Mocks?**

  Das echte Objekt

  - liefert nicht-deterministische Ergebnisse (z.B. Datum)
  - birgt Schwierigkeiten bei der Vorbereitung oder Ausführung (z.B. GUI)
  - soll Verhalten zeigen, das nur schwer auszulösen ist (z.B. IOException)
  - müsste seine Schnittstelle lediglich zu Testzwecken ändern
  - ist langsam (z.B. Zugriff auf Dateisystem, Datenbank)
  - existiert noch nicht

2. Mocks & Co

- **Unterschiedliche Arten von Mocks**
  - Die gegebene Definition für den Begriff Mock ist ungenügend, da sich verschiedene Begriffe für unterschiedliche Varianten herausgebildet haben.
  - Diese Begriffe wurden im Buch XUnit Test Patterns von Gerard Meszaro definiert und werden z.B. auch von Martin Fowler verwendet.



2. Mocks & Co

- **Test Double**
  - Gilt als Oberbegriff für alle Arten von vorgetäuschten Objekten
- **Dummy Object**
  - Wird als Parameterwert herumgereicht, aber niemals wirklich verwendet
- **Test Stub**
  - Enthält speziell für den Test vorprogrammiertes Verhalten
  - Reagiert auf nichts was nicht speziell für den Test definiert wurde

2. Mocks & Co

- **Mock Object**
  - Kann dynamisch vorkonfiguriert werden
    → falls Methode a mit Argumenten b und c aufgerufen wird, dann reagiere folgendermaßen
  - Die vorkonfigurierten Methoden werden auch als *Expectations* bezeichnet und werden in der Regel genau so im Testfall dann auch aufgerufen

2. Mocks & Co

- **Test Spy**
  - Mock Object, welches sich merkt welche Methoden an ihm aufgerufen wurden → ermöglicht Behavior Verification (*verify*)
- **Fake Object**
  - Speziell für den Test implementiert
  - Stellt eine einfachere Variante des echten Objekts dar
  - Sinnvoll z.B. wenn die echte Implementierung noch nicht existiert, aber bereits Funktionalität simuliert werden soll

Agenda

1. **Überblick**
2. **Mocks & Co**
3. **Verwendung**
4. **Weitere Beispiele**
5. **Best Practices & Common Pitfalls**
6. **Limitierungen**
7. **Fazit**

3. Verwendung

- **Generelle Infos**
  - Mockito arbeitet ausschließlich mit Test Spies
  - Im Folgenden wird dennoch der Begriff Mock verwendet, da dies auch die Namensgebung im Code ist
  - Aufgabe von Mockito ist es, Mocks automatisch zu generieren und ein komfortables API zur Konfiguration und Verifikation zur Verfügung zu stellen
  - Es können sowohl Interfaces als auch Klassen gemockt werden

3. Verwendung

- **Vorgehensweise** *(1 / 2)*
  - `org.mockito.Mockito.*` statisch importieren
  - In Testfällen können Mocks ganz einfach mit der statischen Methode

    ```
    mock(Class<T> classToMock)
    ```

    erzeugt werden
  - Mocks mit der statischen Methode

    ```
    when(mockInstance.methodCall()).thenXXX(...)
    ```

    konfigurieren

    → Im Fachjargon: *„Stubbing“*

3. Verwendung

- **Vorgehensweise** *(2 / 2)*
  - Test durchführen
  - Verhalten verifizieren

```
IDatabase database = mock(IDatabase.class);
Bean bean = new Bean(database);

bean.setName(„MyBean“); // Setze Property und aktualisiere DB

assertEquals(„MyBean“, bean.getName()); // State
verify(database).updateName(); // Behaviour
```

3. Verwendung

- **Offizielles Beispiel**

```
// Setup
List mockedList = mock(List.class)

// Exercise
mockedList.add(„one“);
mockedList.clear();

// Verify
verify(mockedList).add(„one“);
verify(mockedList).clear();
```

Feststellungen Behaviour Verification:

- Alle Methodenaufrufe werden am Ende des Tests überprüft

  → Lesbarkeit?
- Kein Wissen über get(...) Methode von List notwendig

  „Mir ist egal was List macht, ich habe durch den Aufruf von add(...) meinen Teil erledigt.“

  → Fokus auf SUT
- Welchen State sollte man testen?
  - Element in Liste?
  - Position in Liste?
  - size()?

3. Verwendung

- **Variante**
  - SUT: State Verification
  - Collaborators: Behavior Verification
  - Vorschlag für die Praxis:

    Verständnis für beide Methoden entwickeln und selbst entscheiden, wann welche Technik am Besten passt

    → Einstellung „Das ist eh nur Test-Code“ vermeiden

    → Test-Code minimieren

    → Tests lesbar und verständlich halten

3. Verwendung

- **Argument Matchers**

  Wie konfiguriert man die Parameterwerte eines Methodenaufrufs?

  - Direkt:

    ```
    when(mock.getEntry(3)).thenXXX(...);
    ```

  - ArgumentMatcher:

    ```
    when(mock.getEntry(anyInt()).thenXXX(...);
    when(mock.getEntry(eq(new Integer(3)).thenXXX(...);
    ```

    Dazu `org.mockito.Matchers.*` statisch importieren

    Achtung: Wenn ein ArgumentMatcher verwendet wird, müssen dies alle Parameter der Methode tun

Agenda

1. **Überblick**
2. **Mocks & Co**
3. **Verwendung**
4. **Weitere Beispiele**
5. **Best Practices & Common Pitfalls**
6. **Limitierungen**
7. **Fazit**

4. Weitere Beispiele

- **Exakte Anzahl an Aufrufen verifizieren**
  - Wie verifiziert man, dass eine Methode exakt 3 mal aufgerufen wurde?

```
mock.methodCall();
mock.methodCall();
mock.methodCall();
verify(mock, times(3)).methodCall();
```

  - Außerdem:

```
atLeastOnce(), atMost(n), never(), once()
```

    Default ist `once()`

4. Weitere Beispiele

- **Verifizieren, dass sonst nichts geschehen ist**
  - Wie verifiziert man, dass außer den verifizierten Methoden keine anderen Methoden an dem Mock aufgerufen wurden?

```
mock.methodCall();
verify(mock).methodCall();
verifyNoMoreInteractions(mock);
```

  - *Achtung: Nicht in jedem Test einsetzen sondern nur, wenn es wirklich für den Test relevant ist!*

4. Weitere Beispiele

- **Aufruf-Reihenfolge verifizieren**
  - Wie verifiziert man, dass Methoden in einer bestimmten Reihenfolge aufgerufen wurden?

```
InOrder inOrder = inOrder(mock1, mock2);
inOrder.verify(mock1).methodCall1();
inOrder.verify(mock2).methodCall2();
inOrder.verify(mock1).methodCall3();
```

  - *Achtung: Nur dann einsetzen, wenn die Reihenfolge wirklich entscheidend ist!*

4. Weitere Beispiele

- **Methoden mit Rückgabetyp void konfigurieren**
  - `doXXX(...).when(...)` statt `when(...).thenXXX(...)` - Syntax verwenden

```
doThrow(new IOException()).when(mock).voidMethod();
```

4. Weitere Beispiele

- **Partial Mocks** *(1 / 2)*
  - Mock, welches auf einem echten Objekt basiert
  - Die echten Methoden des Objekts werden aufgerufen, außer es besteht eine Konfiguration
  - Generell als schlechter Stil klassifiziert
  - Verwendung in seltenen Situationen aber legitim (3rd Party Code)
  - Test von abstrakten Klassen?

4. Weitere Beispiele

- **Partial Mocks** *(2 / 2)*

```
MyObject object = new MyObject();
MyObject spy = spy(object);

when(spy.stubbedMethod()).thenReturn(2);

spy.stubbedMethod(); // Nicht-echte Methode aufgerufen
spy.anyOtherMethod(); // Echte Methode aufgerufen
```

  - Nicht mit „Test Spy“ aus Kapitel 2 zu verwechseln!

Agenda

1. **Überblick**
2. **Mocks & Co**
3. **Verwendung**
4. **Weitere Beispiele**
5. **Best Practices & Common Pitfalls**
6. **Limitierungen**
7. **Fazit**

5. Best Practices & Common Pitfalls

- **Best Practices**
  - *Setup* – *Exercise* – *Verify* Schema einhalten, Abschnitte mit Leerzeilen voneinander trennen
  - Wird ein Mock in der ganzen Test-Klasse gebraucht, dann **@Mock** Annotation an Instanzvariable verwenden, anstatt *mock*(...) - Aufruf in setup Methode
    → Test-Code minimieren
  - Mocks funktionieren am Besten, wenn gegen Interfaces programmiert wird

5. Best Practices & Common Pitfalls

- **Common Pitfalls**
  - `verifyNoMoreInteractions()` bei jedem Testfall
  - `InOrder` Verification bei jedem Testfall
  - Verwendung von *`reset`*`(mock)` deutet auf schlechten Stil hin
    (mit *`reset`*`(mock)` können Mocks auf ihren Anfangszustand zurückgesetzt werden)

FAKTOR ZEHN

Agenda

1. **Überblick**
2. **Mocks & Co**
3. **Verwendung**
4. **Weitere Beispiele**
5. **Best Practices & Common Pitfalls**
6. **Limitierungen**
7. **Fazit**

6. Limitierungen

- **Finale Klassen und Methoden**
  - Mockito kann keine Klassen mocken, die als **final** markiert sind

    → Intern erzeugt Mockito ein Mock einer Klasse natürlich über Vererbung
  - Eine Methode, die als **final** gekennzeichnet ist, kann nicht gestubbt werden
  - Verhindert insbesondere Effective Java Item #17:

    Design and document for inheritance or else prohibit it

6. Limitierungen

- **Statische Methoden**
  - Mockito kann keine statischen Methoden stubben
- **Statische Initializer**
  - Statische Initializer können Probleme beim Mocken verursachen
- **Evil Constructors**
  - Konstruktoren können Probleme beim Mocken verursachen

6. Limitierungen

- **PowerMock**
  - PowerMock ist ein Erweiterungs-API für Mockito und EasyMock
  - Durch massiven Einsatz von Reflection und Bytecode-Manipulation können alle genannten Limitierungen überwunden werden
  - Das ist natürlich nicht umsonst
    → Verhältnismäßig hohe Performance-Einbußen
    → Mehr Setup Code notwendig, reduziert Lesbarkeit und Wartbarkeit
  - Manchmal geht es aber nicht anders – vor allem wenn man nicht auf das Schlüsselwort **final** verzichten möchte

Agenda

1. **Überblick**
2. **Mocks & Co**
3. **Verwendung**
4. **Weitere Beispiele**
5. **Best Practices & Common Pitfalls**
6. **Limitierungen**
7. **Fazit**

7. Fazit

- **Fazit**
  - Subjektiv: Mockito bestes verfügbares Mocking-Framework
  - Bietet intuitives API
  - Sehr einfach zu erlernen
  - Schnelle Erfolge

    → Erhöhte Lesbarkeit und Performance
  - Ermöglicht Behaviour Verification
  - Limitierungen machen ggf. Einsatz von PowerMock notwendig

Noch Fragen?

Nein? Dann können wir jetzt ja einen

trinken gehen ;-)