# Mockito für Interaction Testing

## **Neue Dependency**

|  |
| --- |
| <dependency>  <groupId>org.mockito</groupId>  <artifactId>mockito-all</artifactId>  <version>1.10.19</version>  <scope>test</scope>  </dependency> |

## **Demo: Der Typ eines Mockito -test spy (mock)**

Ein gemocktes Objekt behält seinen Typ.

Gegeben: DOC **ch10.x01MockitoIntro.Car**

Test: **ch10.x01MockitoIntro.FirstMockitoTest**

## **Demo: Default-Rückgabewerte eines Mockito -test spy (mock)**

Der Sinn eines gemockten Objektes ist es, dass man ihm sagen kann, was seine Methoden zurückgeben sollen. Ein gemocktes Objekt hat default-Rückgabewerte. Hier werden nur 2 demonstriert. Die komplette Liste ist in der Mockito Javadoc zu package org.mockito.internal.stubbing.defaultanswers.

Gegeben: DOC **ch10.x01MockitoIntro.Car**

Test: **ch10.x01MockitoIntro.SecondMockitoDefaultValuesTest**

## **Übung: Kann man auch Klassen mocken und nicht nur Interfaces?**

Probieren Sie es aus: Schreiben Sie eine Klasse Ferrari, welche wiederum Interface Car implementiert.

Lassen Sie die Methoden dieser Klasse Werte zurückgeben, welche nicht den gezeigten default-Werten von Mockito entsprechen.

Schreiben Sie einen Test, der dennoch die Rückgabewerte eines gemockten Ferrari-Objektes validiert, sowie dass dieses gemockte Objekt von den Typen Car und Ferrari ist.

Gegeben: DOC **ch10.x01MockitoIntro.Car**

Lösung:

DOC-Implementierung **ch10.x02MockitoMockingClassesLösung.Ferrari**

Test **ch10.x02MockitoMockingClassesLösung.MockitoMockingClasses**

## **Demo: Festlegen des Rückgabetyps eines Mockito -test spy (mock)**

Gegeben: DOC **ch10.x01MockitoIntro.Car**

Test: **ch10.x01MockitoIntro.ThirdMockitoReturningDesiredValuesTest**

## **Demo: Festlegen der zu werfenden Exception eines Mockito -test spy (mock)**

Dieses feature von Mocking Frameworks, das werfen von Exceptions “on demand” ist sehr nützlich besonders für Integrationstests, wenn Szenarien getestet werden sollen, bei denen third-party components z.B. Datenbanken oder Web Services mit dabei sind.

Gegeben: DOC **ch10.x01MockitoIntro.Car**

Test: **ch10.x01MockitoIntro.FourthMockitoThrowingExceptionsTest**

## **Demo: Verification – stelle sicher, dass die SUT bestimmte Methoden ihrer (gemockten) DOCs ausgeführt hat.**

Dies ist eine technische Voraussetzung für echtes Interaction Testing.

Gegeben: DOC **ch10.x01MockitoIntro.Car**

Test: **ch10.x01MockitoIntro.FifthMockitoVerificationTest**

**Regel: Schreibe einen Interaktionstest nur, wenn sich das Verhalten des SUT *nicht* durch State Testing verifizieren lässt (vgl. Beispiel Cache in 2.3 ).**

Grund: Durch Interaction Testing sind die Tests zu sehr an die Implementierung gebunden und weniger an die Spezifikation. Wird die Implementierung geändert ohne dass sich das Verhalten ändert, so schlägt ein Interaction Test fehl.

## **Übung: Verification – Was verifiziert nicht?**

Überprüfen Sie folgende Szenarien mit geeigneten Tests:

* + - 1. Was passiert, wenn eine andere Methode gerufen wird als danach verifiziert wird?
      2. Was passiert, wenn zwar jene Methode gerufen wird, die danach auch verifiziert wird, jedoch mit anderen Parametern?

Gegeben: DOC **ch10.x01MockitoIntro.Car**

Lösungen:

**ch10.x02MockitoIntroLösungen.MockitoFailedVerificationTest  
ch10.x02MockitoIntroLösungen.MockitoFailedVerificationArgumentsTest**

## **Demo: Mockito Matchers**

Wie kann man eine Mockito-Verifikation unabhängig von den übergebenen Parametern machen?

1. Lesen Sie dazu die Beispiele aus der Mockito Javadoc unter Kapitel “Argument Matcher“ <https://static.javadoc.io/org.mockito/mockito-core/2.11.0/org/mockito/Mockito.html#3>
2. Betrachten Sie den Beispiel-Code unter **ch10.x02MockitoMatchers.MockitoMatchersTest**(mit den zu testenden Klassen/Interfaces aus package **ch10.x02MockitoMatchers**)

## **Übung: Verification – Wie Matcher dabei helfen**

„Reparieren“ Sie Klasse **ch10.x02MockitoIntroLösungen.MockitoFailedVerificationArgumentsTest**

so, dass ein Aufruf mit beliebigem Parameter verifiziert.

Gegeben:

DOC **ch10.x01MockitoIntro.Car**

Test **ch10.x02MockitoIntroLösungen.MockitoFailedVerificationArgumentsTest**

Lösung: **ch10.x02MockitoIntroLösungen.MockitoMatcherLösung**

## **Typen von Test Doubles**

|  |  |  |
| --- | --- | --- |
| **Test-Double Typ** | **AKA** | **Beschreibung** |
| dummy object | dummy | muss nur existieren; keine echte collaboration nötig |
| test stub | stub | Wird benutzt, um eine SUT mit indirect inputs zu versorgen. |
| test spy | spy | Wird benutzt, um zu verifizieren, dass die SUT spezifische Methoden der collaborators ruft ggf. mit indirect outputs. |
| mock object | mock |

Beobachtungen:

Dummies und stubs benutzt man um die Testumgebung zu definieren. Sie werden nicht für Verifikation benutzt. Ein dummy wird benutzt, um als Parameter an eine Methode übergeben zu werden. Ein stub ersetzt einen collaborator und gibt möglicherweise einen Wert an die SUT zurück (indirect input). Der Zweck von test spies und mocks ist es die Korrektheit der Kommunikation zwischen SUT und DOCs zu verifizieren. Keines dieser Test-Doubles wird dazu benutzt, die SUT einen bestimmten Wert zurückgeben zu lassen. Genau das soll ja bei state testing von der SUT beobachtet und getestet werden.

In Stein gemeißelt sind diese Rollen allerdings nicht.

## **Demo: Beispiel-Code für Typen von test-Doubles**

siehe package **ch10.x03MockitoMockingTypenVonDoubles**

|  |
| --- |
| **public** **class** Messenger {  **private** TemplateEngine templateEngine;  **private** MailServer mailServer;  **public** Messenger(MailServer mailServer,  TemplateEngine templateEngine) {  **this**.mailServer = mailServer;  **this**.templateEngine = templateEngine;  }  **public** **void** sendMessage(Client client, Template template) {  String msgContent =  templateEngine.prepareMessage(template, client);  mailServer.send(client.getEmail(), msgContent);  }  } |

Was kann an dieser Klasse überhaupt Unit-getestet werden?

Zunächst: *Nicht* getestet werden kann:

* State testing der SUT, da die sendMessage()-Methode nichts zurückgibt.
* Zusandsänderung der collaborators client, template, templateEngine und mailServer, da diese gar keinen sichtbaren Zusandswechsel durchlaufen, rsp. da auf diese nur lesend zugegriffen wird. Hätten sie einen Zustandswechsel, so müsste dies in Unit Test dieser Klassen jeweils getestet werden.
* Integration testing mit echter TemplateEngine und echtem MailServer, da es ein Unit Test sein soll.

Es kann nur getestet/verifiziert werden:

Die Interaktionen der SUT mit ihren DOCs.

Da diese Klasse eine typische Manager-Klasse ist, also die eigentliche Arbeit an die DOCs weiter delegiert (die ihrerseits typische worker-Klassen sind) besteht die einzige Interaktion hier aus dem weiterreichen von Parametern und Rückgabewerten.

|  |
| --- |
| Somit ist das einzig testenswerte die Frage, ob die send() Methode von mailServer mit denselben Argumenten aufgerufen wird, wie sie durch den client-Parameter und den Rückgabewert der templateEngine.prepareMessage()-Methode der SUT-Methode zur Verfügung gestellt wird. Mit anderen Worten: Ob die direct inputs (client-Parameter) und indirect inputs (msgContent ) unverändert weiter als indirect outputs weiter gereicht wurden. |

Dies ist der Test:

|  |
| --- |
| **public** **class** MessengerTest {  **private** **static** **final** String ***CLIENT\_EMAIL*** = "some@email.com";  **private** **static** **final** String ***MSG\_CONTENT*** = "Dear John! You are fired.";  @Test  **public** **void** shouldSendEmail() {  Template template = *mock*(Template.**class**);  Client client = *mock*(Client.**class**);  MailServer mailServer = *mock*(MailServer.**class**);  TemplateEngine templateEngine = *mock*(TemplateEngine.**class**);  Messenger sut = **new** Messenger(mailServer, templateEngine);  *when*(client.getEmail()).thenReturn(***CLIENT\_EMAIL***);  *when*(templateEngine.prepareMessage(template, client)).thenReturn(***MSG\_CONTENT***);  sut.sendMessage(client, template);  *verify*(mailServer).send(***CLIENT\_EMAIL***, ***MSG\_CONTENT***);  }  } |

Hierbei sind

|  |  |  |  |
| --- | --- | --- | --- |
| **Test-Double Typ** | **AKA** | **Beschreibung** | **Beispiel aus dem Code** |
| dummy object | dummy | muss nur existieren; keine echte collaboration nötig | Template **template** = *mock*(Template.**class**);  (…)  sut.sendMessage(client, **template**); |
| test stub | stub | Wird benutzt, um ein SUT mit indirect inputs zu versorgen. | TemplateEngine **templateEngine** = *mock*(TemplateEngine.**class**);  (…)  *when*(**templateEngine**.prepareMessage(template, client)).thenReturn(***MSG\_CONTENT***); |
| test spy | spy | Wird benutzt, um zu verifizieren, dass die SUT spezifische Methoden der collaborators ruft ggf. mit indirect outputs. **Hier:** **der eigentliche Test.** | MailServer **mailServer** = *mock*(MailServer.**class**);  (…)  *verify*(**mailServer**).send(***CLIENT\_EMAIL***, ***MSG\_CONTENT***); |
| mock object | mock |  |

## **Übung: dummy, test stub und test spy mit Mockito anwenden**

Gegeben: Klassen und Interfaces in package **ch10.x04MockitoExercise**

Gesucht: Ein Test, der die Interaktionen der SUT SomeClass mit seinen collaborators verifiziert.

Lösung: **ch10.x05MockitoTestExerciseLösung.SomeClassMockitoTestSpyTest**

## **Übung: Was ist falsch an diesem Test?**

Gegeben:

**ch10.x05MockitoWhatsWrongWithThisTest.QAService**

**ch10.x05MockitoWhatsWrongWithThisTest.QAServiceTest**

Dieser Test gelingt. Was ist jedoch der Denkfehler?

**Lösung:**

## **Übung: Mockito InteractionTesting nach Mockito Cookbook und Vergleich mit Mockitos eigenen Empfehlungen**

Gegeben: Klassen in package **ch10.x06MockitoByCookbook**

1. Schreiben Sie einen Test, der mit der catch-exception-Library die zu erwartende InvalidRequestException bestätigt.
2. Wenden Sie aus dem Mockito Cookbook unter <http://www.baeldung.com/mockito-verify> alle möglichen Arten an, um zu bestätigen, dass wegen der geworfenen Exception die Anweisung requestProcessor.process(request); tatsächlich nie aufgerufen wurde.
3. Wählen Sie nun Ihren Favoriten in Punkto Lesbarkeit des Testcodes
4. Vergleichen Sie nun die Empfehlungen von Mockito selbst in ihrer alten JavaDoc unter <https://mockito.googlecode.com/hg-history/1.5/javadoc/org/mockito/Mockito.html>   
   Hinweis: Die aktuelle JavaDoc ist unter <http://docs.mockito.googlecode.com/hg/1.9.5/org/mockito/Mockito.html>

Lösung:

Zu a), b) **ch10.x06MockitoByCookbookLösung**

Zu c)

Zu d) Die JavaDoc bestätigt diese Auswahl. Auch ist dort eine Warnung bezüglich *verifyNoMoreInteractions*(requestProcessor); geben bezüglich des Stichwortes „overspecified tests“:

|  |
| --- |
| Some users who did a lot of classic, expect-run-verify mocking tend to use verifyNoMoreInteractions() very often, even in every test method.verifyNoMoreInteractions() is not recommended to use in every test method. verifyNoMoreInteractions() is a handy assertion from the interaction testing toolkit. Use it only when it's relevant. Abusing it leads to overspecified, less maintainable tests. You can find further reading [here](http://monkeyisland.pl/2008/07/12/should-i-worry-about-the-unexpected/). |

Zum Thema „overspecified tests“ und „test maintainability“ schreibt Tomek Kaczanowski:

|  |
| --- |
| In fact, we are in a trap, even if we have not yet seen it. The problem is as follows: as we already know,  if we just stick to state testing, we will not be able to test everything.  However, if we start testing interactions between objects, we will soon discover that even innocent  seeming refactorings may result in broken-down tests. And why is that? Well, with state testing all we  attend to is the outcome (of some method calls), and this gives us the freedom to refactor the tested code.  With interaction testing things are different, because it is all about methods being called on collaborators.  Interaction testing makes some assumptions about the implementation of an object, and thus makes it  harder to change this implementation.  (…)  **A test is overspecified if it verifies some aspects which are irrelevant to the scenario being tested.**  (…)  You could guard each and every one of the SUT’s collaborators with verifications that none of their methods have been called , but do not do so, unless they are important relative to the given scenario. Likewise, checking whether certain calls to collaborators happened in the order requested (using Mockito’s inOrder() method) will usually just amount to overkill.  We can find numerous examples of overspecified tests outside of the interactions testing domain, as  well. A common case is to expect a certain exact form of text, where what is in fact important is only  that it should contain several statements.  (…)  A good rule of thumb for writing decent, focused tests is as follows**: test only the minimally necessary set of features using each test method.** |

## **Demo: Mockito erweitern um Seiteneffekt zu testen/void-Methoden zu mocken**

Mockito hat Schnittstellen zur Erweiterung. Eine davon heißt **Answer.**

Gegeben:

SUT **ch10.x07MockitoVoidMethods.Holder** (hält nach außen hin Daten mit Set-Semantik, leitet Anfragen jedoch in Wirklichkeit weiter an:

DOC **ch10.x07MockitoVoidMethods.IRepo** (ein selbst zustandsloses Repository, welches eine externe Datenquelle mit Set-Semantik manipuliert)

Test **ch10.x07**

**MockitoVoidMethods.HolderTest**

Soll die Set-Semantik des Holders testen (und nicht den indirect output der SUT an das Repo wie bisher). Um zu vermeiden, das echte Repo mit vielleicht entferntem Datenbankzugriff einzubinden, soll es gemockt werden. Dazu muss aber sein Verhalten eine globale Datenquelle, also irgendeinen state manipulieren.

Und da die IRepo-Methoden überdies alle void zurückgeben, funktioniert das übliche when()…thenReturn() nicht.

Die JavaDoc zu doAnswer(): <http://site.mockito.org/mockito/docs/current/org/mockito/Mockito.html#12>

Basiert auf <http://stackoverflow.com/questions/32139252/mockito-how-to-stub-void-methods-to-run-some-code-when-called>

Der Frager schreibt:

|  |
| --- |
| I want to stub a repository class to test another class (Holder class) that has a repository. The repository interface supports CRUD operations, and has many methods, but my unit test on the Holder class only needs to call two of them. The repository interface:  public interface IRepo {  public void remove(String... sarr);  public void add(String... sarr);  //Lots of other methods I don't need now  }  **I want to create a repository mock that can store instances, define logic for add and removeonly, and also provide a means of checking what is stored on it after calling add and remove.** |