

Mockito Unit Test mit Eclipse Maven

Ziel

In komplexen verteilten Architekturen sind Java Unit Tests nur schwierig realisierbar, da solche auf anderen teilweise nicht verfügbaren Systemen basieren. In diesen Fällen dienen uns Mock Frameworks als ideale Lösung. Mit Mocks werden Systeme (Klassen, Interfaces, ...) gemockt und damit als Fake Instanzen gebaut und verwendet. Das Spring Boot Framework bietet hier selber einige Lösungen und damit Abhilfe. Dieser Blog zeigt den minimalen Setup für den Einsatz des Mockito Framework zwecks Mocking einer nicht verfügbaren Umgebung.

Voraussetzungen

Für diesen Blog verwenden wir Java Version 11+, die Eclipse IDE mit Maven Plugin sowie Maven 3.6+.

Eclipse Maven Projekt Setup

Wir starten die Eclipse IDE in einem eigenen Workspace (z.b. /home/user/blog/mockito): Wir erstellen ein Maven Projekt: Das Maven Projekt ist nun erstellt: Achten Sie darauf, dass Sie die korrekt Java Version im Projekt verwenden.

Random Anwendung

Als Anwendung wollen wir 10 Random zahlen im Bereich 1 - 1000 generieren und Minimum/Maximum sowie Mittelwert bestimmen. Wir implementieren das folgende UML Diagramm: Die Random Quelle kapseln wir über das Interface IRandom: package ch.std.blog.random;

 import java.util.List;

 public interface IRandom

 {

 public abstract List

 newRandom(int size, int min, int max);

 } Das Interface wird von der Klasse RandomDoubleData verwendet als Quelle der Random Daten: package ch.std.blog.random;

 import java.util.DoubleSummaryStatistics;

 import java.util.List;

 public class RandomDoubleData {

 private IRandom

 Double

 random;

 private List

 Double

 doubleList;

 private DoubleSummaryStatistics stats;

 public RandomDoubleData(IRandom

 Double

 random) {

 this.random = random;

 this.doubleList = this.random.newRandom(10, 1, 1000);

 this.stats = doubleList.stream().mapToDouble(d -

 d).summaryStatistics();

 }

 public List

 Double

 getDoubleList() {

 return doubleList;

 }

 public Long getCount() {

 return this.stats.getCount();

 }

 public Double getMin() {

 return this.stats.getMin();

 }

 public Double getMax() {

 return this.stats.getMax();

 }

 public Double getAverage() {

 return this.stats.getAverage();

 }

 } Wir haben 3 Implementationen, welche Random Daten auf verschiedene Arten generieren. Die erste Implementation ClassicRandomImpl arbeitet mit der Java Standard Klasse java.util.Random: package ch.std.blog.random.impl;

 import java.util.ArrayList;

 import java.util.List;

 import java.util.Random;

 import ch.std.blog.random.IRandom;

 public class ClassicRandomImpl implements IRandom

 {

 public List

 Double

 newRandom(int size, int min, int max) {

 List

 Double

 randomList = new ArrayList

 ();

 Random r = new Random();

 for (int i = 0; i

 size; i++) {

 double randomValue = min + (max - min) * r.nextDouble();

 randomList.add(randomValue);

 }

 return randomList;

 }

 } Die Implementation LambdaRandomImpl arbeitet mit dem Java Lambda Ansatz: package ch.std.blog.random.impl;

 import java.util.List;

 import java.util.Random;

 import java.util.stream.Collectors;

 import ch.std.blog.random.IRandom;

 public class LambdaRandomImpl implements IRandom

 {

 public List

 Double

 newRandom(int size, int min, int max) {

 return new Random().doubles(size, min, max).boxed().collect(Collectors.toList());

 }

 } Die Klasse UriFakeRandomImpl liest die Random Daten vom JSON Fake Service von der folgenden URL https://www.simtech-ag.ch/std-ajax/randomservice?min=0

 max=1000 package ch.std.blog.random.impl;

 import java.io.InputStream;

 import java.net.URL;

 import java.util.ArrayList;

 import java.util.List;

 import

```

org.json.JSONObject;&#xA;import org.json.JSONTokener;&#xA;&#xA;import
ch.std.blog.random.IRandom;&#xA;&#xA;public class UrlFakeRandomImpl implements
IRandom&lt;Double&gt; {&#xA;&#xA; private URL url;&#xA;&#xA; public
UrlFakeRandomImpl(URL url) {&#xA; this.url = url;&#xA; }&#xA;&#xA; public
List&lt;Double&gt; newRandom(int size, int min, int max) {&#xA;
List&lt;Double&gt; randomList = new ArrayList&lt;&gt;();&#xA; for (int i = 0; i
&lt; size; i++) {&#xA; JSONObject jsonObject = this.getJSONObject();&#xA; Double value
= jsonObject.getDouble(&#34;value&#34;);&#xA; randomList.add(value);&#xA; }&#xA; return
randomList;&#xA; }&#xA;&#xA; private JSONObject getJSONObject() {&#xA; try (InputStream is =
this.url.openStream()) {&#xA; JSONTokener tokenener = new JSONTokener(is);&#xA; JSONObject
root = new JSONObject(tokenener);&#xA; return root;&#xA; } catch (Exception e) {&#xA; return
null;&#xA; }&#xA; }&#xA;&#xA;}&#xA;

```

Unit Tests (Non Mock)

Random Zahlen sind nicht direkt testbar, weil das erwartete Ergebnis zufällig ist. Wir arbeiten ohne Mock mit einer Test Implementation, welche mit fixen Werten arbeitet:

```

package
ch.std.blog.random.test;&#xA;&#xA;import java.net.MalformedURLException;&#xA;import
java.net.URL;&#xA;import java.util.Arrays;&#xA;import java.util.List;&#xA;&#xA;import
org.junit.Assert;&#xA;import org.junit.jupiter.api.Test;&#xA;&#xA;import
ch.std.blog.random.IRandom;&#xA;import ch.std.blog.random.RandomDoubleData;&#xA;import
ch.std.blog.random.impl.UrlFakeRandomImpl;&#xA;&#xA;class TestRandom implements
IRandom&lt;Double&gt; {&#xA;&#xA; @Override&#xA; public List&lt;Double&gt;
newRandom(int size, int min, int max) {&#xA; return Arrays.asList(1.0, 2.0, 3.0);&#xA; }&#xA;
&#xA;&#xA;public class RandomDoubleUnitTest {&#xA;&#xA; @Test&#xA; public void
testDoubleRandom() {&#xA; RandomDoubleData rdd = new RandomDoubleData(new
TestRandom());&#xA; double expectedMin = 1.0;&#xA; double expectedMax = 3.0;&#xA; double
expectedAverage = 2.0;&#xA; &#xA; Assert.assertEquals(expectedMin, rdd.getMin(), 0.0);&#xA;
Assert.assertEquals(expectedMax, rdd.getMax(), 0.0);&#xA;
Assert.assertEquals(expectedAverage, rdd.getAverage(), 0.0);&#xA; &#xA; }&#xA; &#xA;
&#xA; @Test&#xA; public void testDoubleFakeRandom() throws MalformedURLException {&#xA;
RandomDoubleData rdd = new RandomDoubleData(new UrlFakeRandomImpl(new
URL(&#34;https://www.simtech-ag.ch/std-ajax/randomservice?min=0&max=1000&#34;)));
&#xA; Long expectedCount = 10L;&#xA; Assert.assertEquals(expectedCount,
rdd.getCount());&#xA; &#xA; }&#xA;&#xA;}

```

Mockito Unit Tests

Mit dem Mockito Framework können wir Interfaces als Mock implementieren. Das folgende Listing zeigt diesen Ansatz:

```

package ch.std.blog.random.test;&#xA;&#xA;import static
org.mockito.ArgumentMatchers.anyInt;&#xA;import static
org.mockito.Mockito.when;&#xA;&#xA;import java.util.Arrays;&#xA;&#xA;import
org.junit.Assert;&#xA;import org.junit.jupiter.api.Test;&#xA;import
org.junit.jupiter.api.extension.ExtendWith;&#xA;import org.mockito.Mock;&#xA;import
org.mockito.junit.jupiter.MockitoExtension;&#xA;&#xA;import
ch.std.blog.random.IRandom;&#xA;import
ch.std.blog.random.RandomDoubleData;&#xA;&#xA;@ExtendWith(MockitoExtension.class)
&#xA;public class MockRandomDoubleUnitTest {&#xA;&#xA; @Mock&#xA; private
IRandom&lt;Double&gt; mockRandom;&#xA; &#xA; @Test&#xA; public void
testDoubleMockRandom() {&#xA; &#xA; when(mockRandom.newRandom(anyInt(),anyInt(),
anyInt()))thenReturn(Arrays.asList(1.0, 2.0, 3.0));&#xA; RandomDoubleData rdd = new
RandomDoubleData(mockRandom);&#xA; double expectedMin = 1.0;&#xA; double expectedMax
= 3.0;&#xA; double expectedAverage = 2.0;&#xA; &#xA; Assert.assertEquals(expectedMin,
rdd.getMin(), 0.0);&#xA; Assert.assertEquals(expectedMax, rdd.getMax(), 0.0);&#xA;
Assert.assertEquals(expectedAverage, rdd.getAverage(), 0.0);&#xA; &#xA; }&#xA; &#xA;}

```

Damit dies kompilierte benötigen wir die Mockito Library inkl. JUnit 5 Jupiter Support:

```

&lt;project
xmlns=&#34;http://maven.apache.org/POM/4.0.0&#34;&#xA;
xmlns:xsi=&#34;http://www.w3.org/2001/XMLSchema-instance&#34;&#xA;
xsi:schemaLocation=&#34;http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd&#34;&#xA;
&lt;modelVersion&gt;4.0.0&lt;/modelVersion&gt;&#xA;

```

```
<groupId>blog.mockito</groupId>&#xA;
<artifactId>mockitoeclipsemaven</artifactId>&#xA;
<version>0.0.1-SNAPSHOT</version>&#xA;
<name>blog mockito eclipse maven</name>&#xA;&#xA;
<dependencies>&#xA; <!-- https://mvnrepository.com/artifact/org.json/json
-->&#xA; <dependency>&#xA;
<groupId>org.json</groupId>&#xA;
<artifactId>json</artifactId>&#xA;
<version>20210307</version>&#xA;
</dependency>&#xA;&#xA;&#xA; <!--
https://mvnrepository.com/artifact/org.mockito/mockito-core -->&#xA;
<dependency>&#xA;
<groupId>org.mockito</groupId>&#xA;
<artifactId>mockito-core</artifactId>&#xA;
<version>3.9.0</version>&#xA;
<scope>test</scope>&#xA;
</dependency>&#xA;&#xA; <dependency>&#xA;
<groupId>org.mockito</groupId>&#xA;
<artifactId>mockito-junit-jupiter</artifactId>&#xA;
<version>2.23.0</version>&#xA;
<scope>test</scope>&#xA; </dependency>&#xA;
</dependencies>&#xA;&#xA;</project>
```

Mit der @Mock Annotation wird die betroffene Klasse via Proxy Pattern gespiegelt und als Fake Instanz implementiert. Mit der when Anweisung wird die Mock Instanz für den Test vorbereitet, so dass die gesuchten Daten geliefert werden.

Komplettes Eclipse Projekt

Das komplette Eclipse Projekt findet man unter dem Link [mockitoeclipsemaven.zip](#).

Feedback

War dieser Blog für Sie wertvoll. Wir danken für jede Anregung und Feedback

Kontakt

Simtech AG
Finkenweg 23
3110 Münsingen
Schweiz

Impressum

Das Copyright für sämtliche Inhalte dieser Website liegt bei Simtech AG, Schweiz. Beachten Sie auch unsere Hinweise zum Urheberrecht, Datenschutz und Haftungsausschluss. Jeder Hinweis auf Fehler nehmen wir gerne entgegen.

Copyright

2024 Simtech AG, All rights reserved, Powered by stack.ch written in Golang by Daniel Schmutz

<https://www.simtech-ag.ch/blog/java/mockitoeclipsemaven>