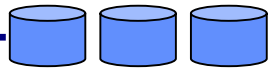
Software Engineering 1

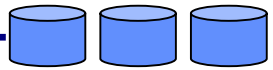
Übung: (Unit) Testing und Mocking

Kristof Böhmer
Fakultät für Informatik
Universität Wien

Inhalt

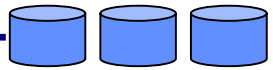


4.6 Überblick

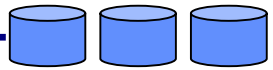


- **Limitierungen dieses Foliensatzes**
  - Der Fokus liegt auf den notwendigen Kenntnissen für die Übung.
  - Die Vorlesung liefert weitere Informationen.
- **Systemtest**
  - Testen des gesamten Systems, Abnahmetests.
  - Beispielsweise durch UI Interaktionen mit dem Programm.
- **Integrationstest**
  - Integration (Zusammenspiel) mehrere Komponenten.
  - Prüft die korrekte Einbindung von Schnittstellen (Client <-> Server).
- **Unit Test**
  - Eher kleinteilig, einzelne Klassen, Methoden, Komponenten.
  - Wird im Rahmen der Übung von Ihnen verlangt.

Inhalt

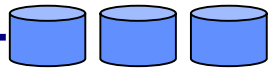


4.7 Unit Testing: Generell



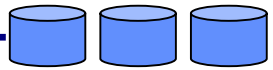
- **Fokus von objektorientierten Unit Tests**
  - Testen einzelner Klassen bzw. einzelner Methoden.
  - Schnittstellen werden häufig abstrahiert.
- **Mock:** Simulieren in kontrollierter Art und Weise das Verhalten realer Objekte.
  - Ermöglicht komplexes Systemverhalten abzubilden.
  - Simulation von Netzwerken, Datenbanken, Filesystem, etc.
- **Stub:** Rückgabe fester Werte bzw. Anwendung einfacher Regeln.
  - Zumeist einfacher als Mocks.
- **Mockingframework:** Erleichtern die Entwicklung von Mocks/Stubs, z.B. JMock, Mockito, …
  - Dependency Injection und testfokussierte Entwicklung ist hilfreich.

4.7 Unit Testing: Test Automatisierung

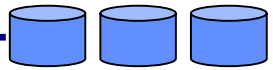


- **Regressionstest:** Automatisierte Tests bzw. automatisiertes ausführen von Unit Tests.
  - Werden häufig durchgeführt, beispielsweise bevor oder nach Änderungen im Code (Fragestellung: hat mein letzter Commit etwas „zerstört“?).
  - Automatisch beim Daily Build, bei Git Commits, etc.
- **Refactoring:** Überarbeitung bestehender Funktionalität um z.B. die Wartbarkeit zu verbessern.
  - Initial muss eine umfassende Bibliothek an Testfällen erstellt werden, z.B. der Einsatz von Fuzzing Libraries hilft hier.
  - Sicherstellen, dass nach und vor Änderungen vergleichbares Verhalten gegeben ist.
- **Dokumentation:** Tests dokumentieren erwartetes Verhalten.
  - Generierung der „normalen“ Dokumentation aus den Tests und Code.
  - „Code is Law“

Inhalt



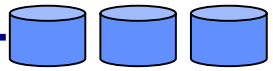
4.8 Definieren von Unit Tests – Beispiele für jUnit 5



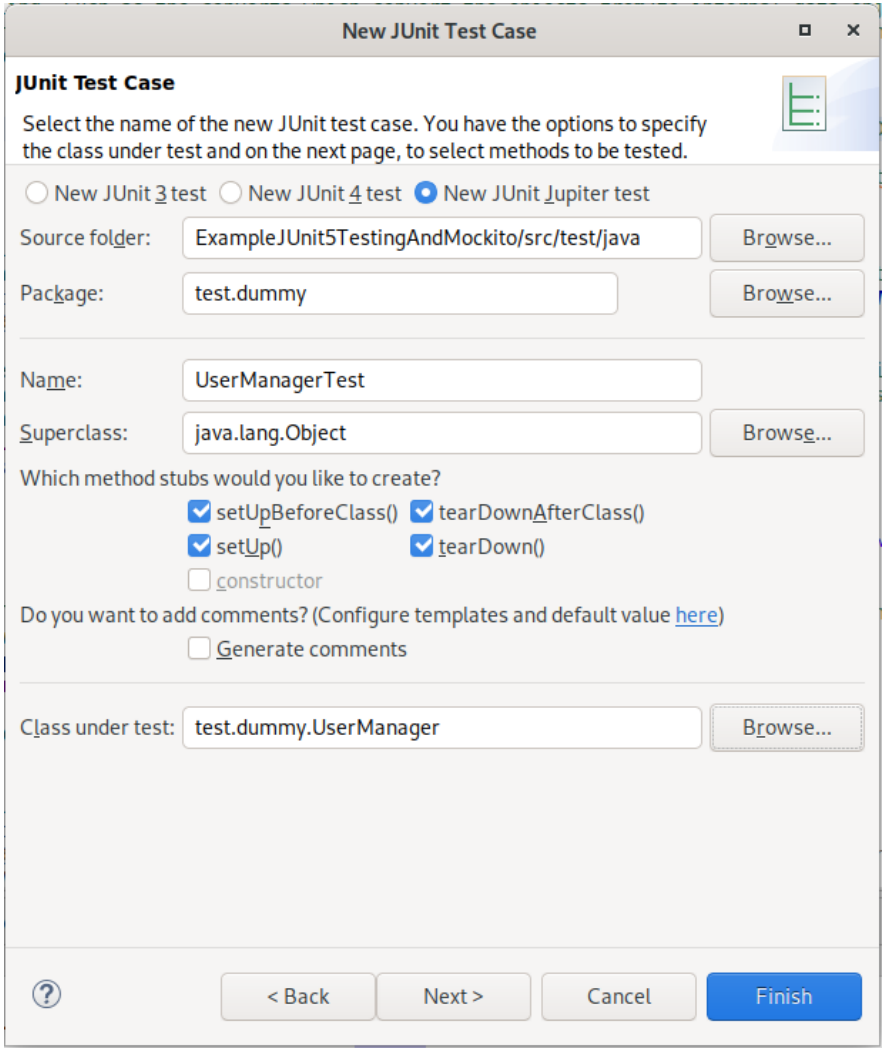
- **jUnit Testklassen erzeugen**
  - Test Framework für Java, C#, C++, Ruby, etc.
  - Erleichtert die Entwicklung und Automatisierung von Tests.
  - Erstellen in Eclipse ⇨ File ⇨ New ⇨ jUnit Test Case
  - Ausführen in Eclipse ⇨ Run ⇨ Run as jUnit Test

- **jUnit5 Testklassen (Testfixture) beinhalten zumeist**
  - Einen oder mehrere Testfälle (Annotation mit `@Test`).
  - Methoden die vor Testfällen ausgeführt werden z.B. um Mocks und Stubs vorzubereiten (Annotation mit `@BeforeEach`).
  - Methoden die nach Testfällen ausgeführt werden z.B. um nach Tests „aufzuräumen“ (Annotation mit `@AfterEach`).
  - Methoden die vor bzw. nach allen Testfällen ausgeführt werden (Annotation mit `@BeforeAll` und `@AfterAll`).

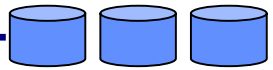
4.8 Definieren von Unit Tests – Beispiele für jUnit 5



- **Definieren von Unit Tests in Eclipse**
  - Beispielsweise Rechtsklick auf eine Klasse
  - New □ Other... □ jUnit Test Case
  - Einige Daten werden dann bereits vorausgefüllt (z.B. *Class under test*).
  - Einstellung "*New JUnit Jupiter test*" wählen um jUnit 5 zu nützen.



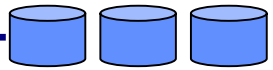
4.8 Unit Tests: Asserts – Beispiele für jUnit 5



- **Asserts:** Vergleich von Soll und Ist.
  - Bei Abweichungen gilt ein Test als fehlgeschlagen.
  - Vergleichbare Idee zu Java Asserts aber mächtigere Definitionen möglich.
  - „Hamcrest Matchern“ erhöhen die Lesbarkeit, siehe: `is`, `isNot`, `nullValue`, ...
- **Auszug möglicher Matcher**

```
assertThat(currentValue, is(expectedValue));

assertThat(contact.getName(), is("Kristof"));

assertThat(currentValue, isNot(expectedValue));

assertThat(currentValue, is(nullValue()));

assertThat(currentValue, is(notNullvalue()));

assertThat(currentValue, is(closeTo(expectedValue)));
```

4.8 jUnit Beispiel Test – Beispiele für jUnit 5

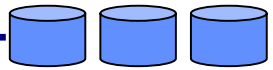


Zu testende Klasse

- Neuimplementierung einer einfachen Liste.
- Vergleichbar z.B. zur `ArrayList` Implementierung.
- Hinzufügen von Werten mit `add(value)` auslesen von Werten mit `get(index)`.

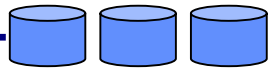
```
public class MyListTest {
   private MyList myList;
   private String testWord = "word";
   @BeforeAll // execute before each test
   public void init() {
     myList = new MyList();
   }
   @AfterAll // execute after each test
   public void cleanUp() {
     myList = null;
   }
   @Test
   public void emptyList_addValue_shouldContainAddedValue() {
      myList.add(testWord);
      assertEquals(testWord, myList.get(0));
   }
```

4.8 Best Practices für Unit Tests 1



- **Aufteilung der Testklassen**
  - Strukturierung und Aufteilung der Tests in Klassen und Packages.
  - Eine Testklasse sollte sich beispielsweise auf bestimmte Funktionsgruppen oder bestimmte (besser einzelne) Klassen konzentrieren.
  - Eine Testmethode sollte sich nur auf eine bestimmte Funktionalität konzentrieren.
  - Wiederverwendung von Code ist auch für Tests relevant (vor allem für Testvor- [Testdatenerstellung] und Nachbereitung [Ergebnisvalidierung]).
- **Benennung der Testmethoden**
  - Eine *einheitliche* Benennung die auch den *Sinn/Zweck* jedes Tests beschreibt erleichtert die Analyse von Ergebnissen und verbessert auch deren Lesbarkeit.
  - Verschiedene Benennungsschemata sind im Einsatz.
  - Beispiele, unter anderem, nach „*xUnit Test Patterns, Gerard Meszaros, 2007*“

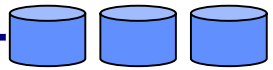
```
public void <UseCase/Method>_should<ExpectedPostState>
public void calendarToText_shouldReturnAppointmentsInDescendingOrder()
oder (empfohlen)
public void <OriginalBeforeState>_<Action>_<ExpectedPostState>
public void emptyList_addNewValue_ListContainsAddedValue()
```



Testen privater Methoden - Ausgewählte Möglichkeiten

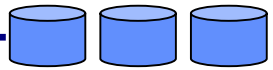
1. **Mittels `public` Methoden:** Hierbei werden private Methoden so getestet wie andere Entwickler diese auch verwenden würden, daher durch deren natürlichen Aufruf über `public` Methoden. Da hier white Box Testing eingesetzt wird, lässt sich erkennen ob und wie gut private Methoden von den Tests erfasst werden. Dieser Ansatz kann es erforderlich machen mehrere Tests pro `public` Methode zu erstellen um alle von dieser verwendeten private Methoden zu erreichen. Gründliches Testen erfordert dies aber in jedem Fall auch.
2. **Als `protected` Methoden:** Statt `private` den Modifier `protected` verwenden. Zugriffsschutz ist damit nur noch teilweise gegeben, Testmethoden im selben Package können die "private" Methoden dann aufrufen. Nicht empfohlen.
3. **Mittels Reflection:** Dadurch können Access Modifier während der Laufzeit ignoriert werden. Damit lassen sich private Methode so aufrufen wie als wären Sie public. Spring bietet hierfür z.B. die **`ReflectionTestUtils`** Klasse an.
   Die Wartbarkeit dieses Weges leidet etwas darunter, da hierbei Methodennamen, etc. als Text angegeben werden – welche im Falle von Änderungen nicht immer auch automatisch mit angepasst werden. Eine IDE kann solche Probleme ebenfalls nicht automatisch erkennen und melden.

Spezifische Empfehlungen werden auf der folgenden Seite gegeben.



Testen privater Methoden – Empfehlung

- Nützen Sie in der Regel den ersten Weg (`public` Methoden). Dies ist realistisch und in der Regel sollten nicht zu viele private Aufrufe hintereinander passieren die diesen Weg erschweren.
- Sollten in einer Klasse zu viele private Methoden sich gegenseitig aufrufen (`public->private->private->private->...`) deutet dies darauf hin, dass Sie zu viel Funktionalität in eine Klasse gepackt haben. Dies erschwert Verständnis, Erweiterung und Wiederverwendung von Funktionalität.
  Teilen Sie die Klassen weiter auf und geben sie jeder Klasse genau definierte public Schnittstellen sodass folgendes entsteht, erste Klasse (`public->private->private`), zweite Klasse (`public->private->private`). Die erste Klasse kann hierbei die zweite Klasse über die angebotenen public Methoden aufrufen.
- Für Fälle wo dies nicht möglich ist bietet sich ein Rückgriff auf Weg Drei an. Dies sollte aber nur spärlich verwendet werden da es die Wartung erschwert.

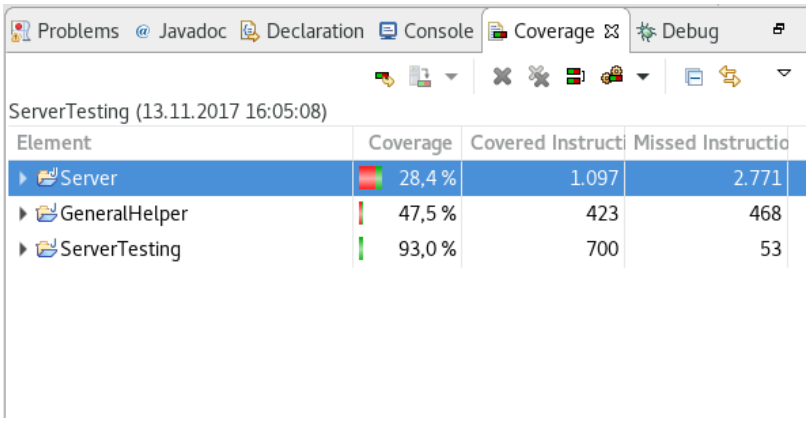


Umgang mit Abhängigkeiten

- Unit Tests und entsprechend auch der von Unit Tests ausgeführte Code sollte **keine Abhängigkeiten** zum "Umfeld", daher anderen Systemen, externen Verhalten (auch z.B. der Uhrzeit) und Datenquellen (z.B. Datenbanken) aufweisen. Solche Abhängigkeiten verlangsamen Tests, erschweren deren automatische Ausführung oder verhindern teilweise auch deren Ausführung komplett (z.B. weil externe Systeme zumeist nicht immer verfügbar sind).
- Generell gilt: Wenn es schwer fällt Tests zu schreiben und Klassen stark mit einander verzahnt sind sowie viele Abhängigkeiten aufweisen sollte die Architektur neu überdacht beziehungsweise überarbeitet werden.
- Vermeiden Sie Abhängigkeiten schon beim Design Ihrer Architektur. Achten Sie dafür darauf, dass single responsability principle, encapsulation sowie das interface segregation principle einzusetzen. Die refactoring Funktionalität in Ihrer IDE ermöglicht ihnen auch nachträglich Änderungen schnell vorzunehmen.
- Für Fälle wo Abhängigkeiten nicht vermieden werden können sollte Mocking eingesetzt werden. Darauf wird in den folgenden Abschnitten noch eingegangen.

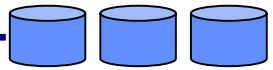
4.8 Test Coverage ermitteln mit Eclipse

- **Test Coverage:** „Misst“ die Abdeckung des Programmcodes mit Tests.
  - Ermöglicht einen einfachen und schnellen Überblick über die Testabdeckung.
  - Zeigt Fortschritte auf (z.B. zur Verdeutlichung des Aufwandes gegenüber des Managements).
  - Oft wenig aussagekräftig, 80/20 Regel gilt auch hier.
  - Mehrere Methoden: Branch-, Statement-, etc. Coverage
- **Eclipse:** Rechtsklick auf Testprojekt, Coverage As ⇨ jUnit Test

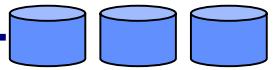



```
13          List<ETerrain> terrainToUse = new ArrayList<>();
14
15          //fill it up with grass
16          //then place montains on random positions
17          //then place water on fixed positions
18          for(int i=0;i<totalAmountRequired-minAmou
19          {
20              terrainToUse.add(ETerrain.Grass);
21          }
22          for(int i=0;i<minAmountMontain;i++)
23          {
24              terrainToUse.add(ETerrain.Montains);
25          }
26          Collections.shuffle(terrainToUse);
27
28          ETerrain[][] terraind2D = new ETerrain[halfMapSoue][mapSize];
29
30          int terrainIndex = 0;
31
32          for(int y=0;y<halfMapSoue;y++)
33          {
34              for(int x=0;x<mapSize;x++)
35              {
36                  terraind2D[y][x] = terrainToUse.get(terrainIndex);
37                  terrainIndex++;
38              }
```

Inhalt

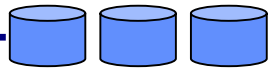


4.9 Unterschiede jUnit4 und 5 – Übersicht



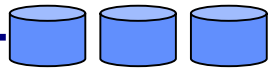
- **Identische Grundlagen**
  - Die Verfahren zur Messung der Code Coverage, Best Practices, Testklassenerstellung in Eclipse, etc. (siehe die vorangegangenen Inhalte) können unverändert in jUnit5 weiter angewendet werden.
- **jUnit5 – eine kurze Übersicht**
  - Schönere interne Strukturierung der Codebasis in *jUnit Platform* (Test Engine), *jUnit Jupiter* (Erweiterung um all die neuen schönen Java Features) und *jUnit Vintage* (ausführen von Testcode aus jUnit3, 4 und 5 basierend auf jUnit5).
  - jUnit5 benötigt Java 8 oder neuer, jUnit4 läuft ab Java 5.
  - Annotationen wurden teilweise angepasst, Asserts sind gleich geblieben (leicht erweitert). Im Folgenden gibt es hierzu Hinweise und Vergleiche.
  - Datengetriebene Tests werden von Beginn weg umfassend unterstützt.
  - Neue Features, z.B. repeated tests (z.B. `@RepeatedTest(100)`) um den gleichen Test z.B. 100 mal hintereinander auszuführen.
- **Bereitgestellte Codebeispiele:** In Moodle finden Sie ein Beispielprojekt wie für jUnit5 Tests implementiert werden können (z.B. datengetriebene Tests, Negativtests, positiv Tests, etc.)

4.9 Unterschiede jUnit4 und 5 – Keywords



- **Unterschiede zwischen den Annotationen – Kurzübersicht**

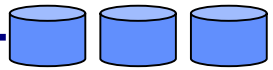
Funktionalität	jUnit4	jUnit5
Testmethode	`@Test`	`@Test`
Ausführen vor allen Tests	`@BeforeClass`	`@BeforeAll`
Ausführen nach allen Tests	`@AfterClass`	`@AfterAll`
Ausführen vor jeder Testmethode	`@Before`	`@BeforeEach`
Ausführen nach jeder Testmethode	`@After`	`@AfterEach`
Test deaktivieren	`@Ignore`	`@Disable`
Datengetriebene Tests (nur in jUnit5, siehe folgende Slides)	-	`@ParameterizedTest`
Wiederholende Tests (nur in jUnit5)	-	`@RepeatedTest`

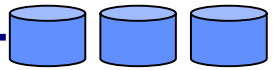


- **Datengetriebene Tests mit jUnit5 umsetzen**
  - Es wird eine Datenquelle benötigt, z.B. mit `@CsvSource` können die Daten mittels Annotation angegeben werden. Im Vergleich dazu können mit `@MethodSource` die Daten aus einem Methodenaufruf stammen.
  - Normalerweise dürfen Unit Tests in jUnit keine Parameter beinhalten, bei datengetriebenen Tests ändert sich dies. Hier werden die Testdaten über Parameter für den Testcode verfügbar => `int a, int b, int expected`.

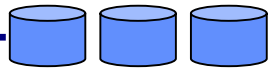
```
@ParameterizedTest
@CsvSource({"1,1,1", "2,2,1"})
public void Division_withParametervalues_shouldProduceExpectedResult(
int a, int b, int expected) {
        int result = dummy.div(a,b);
        Assertions.assertEquals(
                expected, result,
                MessageFormat.format(
                        "Expected {0} when dividing {1} by {2}",
                        expected, a, b));
}
```

Inhalt



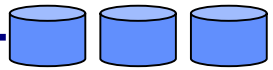


- **Mocking – Was, Warum, Wie?**
  - Unit Tests sollten immer nur die zu testende Funktionalität prüfen. Dazu müssen Sie von externen Abhängigkeiten möglichst unbeeinflusst arbeiten können. Eine gute Architektur berücksichtigt dies automatisch da hierdurch auch die Entwicklung und Wartung vereinfacht wird.
  - Ist dies nicht immer möglich dann können externe Abhängigkeiten mit simulierten Verhalten ersetzt werden. Je nach Umfang und Qualität der Simulation wird von einem Dummy, Stub oder Mock gesprochen.
  - Letzteres (Mock) simuliert externe Abhängigkeiten und deren Verhalten und ermöglicht auch die Verwendung solcher Abhängigkeiten zu prüfen. Beispielsweise ob Methoden in der (simulierten) Abhängigkeit in der passenden Reihenfolge, in der erwarteten Anzahl und den korrekten Methodenparametern aufgerufen wurden.
  - Um die Entwicklung von Mocks zu beschleunigen können Mocking Frameworks wie EasyMock, JMockit oder **Mockito** verwendet werden.



- **Mockito – Grundlagen**
  - **Mock:** Mockito ermöglicht es Verhalten anhand von Interfaces sowie abstrakten Klassen zu simulieren. Hierzu wird z.B. definiert was beim Aufruf einer simulierten Methode returniert werden soll: `Mockito.mock`
  - **Spy:** Zusätzlich ermöglicht Mockito Spies zu erzeugen. Diese basierend auf einer konkreten Implementierung einer Abhängigkeit und können diese überwachen. Beispielsweise welche Methoden wie aufgerufen wurden. Auch *partial Mocking* ist möglich. Dabei wird echte Implementierung und simulierte Implementierung gemischt: `Mockito.spy`
- **Mocks werden im Rahmen eines Unit Tests wie folgt erzeugt:**

```
// Simulation (mock) based on Java's List interface
List<String> mocked = Mockito.mock(List.class);

// Spying on a real implementation, here, Java's ArrayList
List<String> spied = Mockito.spy(new ArrayList<String>());
```



Mockito – Verhalten definieren

- Unabhängig von Mock oder Spy kann mit `Mockito.when` simuliertes Verhalten eines Methodenaufrufs definiert werden.

```
// Creating the mock based on the List interface
List<String> mocked = Mockito.mock(List.class);

// Defining simulated behaviour for the .get(int index) method
Mockito.when(mocked.get(anyInt())).thenReturn("SimulatedContent");
```

- Hierbei kann mit `any...` (`anyInt()`, `anyString()`, `anyObject()`, `any()`, etc.) ein beliebiger Parameterwert angenommen werden.
- Im Vergleich hierzu kann beispielsweise mit `eq(...)` ein spezifischer Wert definiert werden um je nach Parameterwert anderes Verhalten zu simulieren. Hierzu können auch mehrere individuelle `Mockito.when` kombiniert werden.
- Weitere und aufwändigere Beispiele hierzu (z.B. mit `Mockito.doAnswer`) finden Sie im Unit Testing und Mocking Beispielprojekt auf Moodle.

Mockito – Verhalten überprüfen

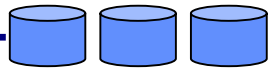
- Unabhängig von Mock oder Spy kann mit `Mockito.verify` überprüft werden ob Methoden wie erwartet aufgerufen werden. Schlägt ein `verify(…)` fehl wird auch der Unit Test als fehlgeschlagen erkannt.
- Wie oft wurde eine Methode aufgerufen:
  - Statt `times(…)` kann beispielsweise auch `atLeastOnce()`, `atMost(…)`, `never()`, `only()`, etc. verwendet werden.

```
// Creating the mock based on the List interface
List<String> mocked = Mockito.mock(List.class);
mocked.size();
// Checking if size() was called exactly once
Mockito.verify(mocked, times(1)).size();
```

- Mit welchen Parametern wurde eine Methode aufgerufen:

```
// Creating the mock based on the List interface
List<String> mocked = Mockito.mock(List.class);
mocked.add("SomeData");
// Checking if add(...) was called with the param value "SomeData"
Mockito.verify(mocked).add("SomeData");
```

Inhalt



- **Verwendung von Unit Testing (jUnit) mit Gradle**
  - Folgende Zeile in die bereits erstellte Datei `build.gradle` unter `dependencies` einfügen:
    - **Integration von Mockito** `(X.X.X => Version)`**:**

      ```
      testImplementation 'org.mockito:mockito-core:X.X.X'
      ```

      Bedeutung: Lädt Mockito als Mockingframework.
    - **Integration von jUnit5** `(X.X.X => Version)`**:**

      ```
      testImplementation 'org.junit.jupiter:junit-jupiter-api:X.X.X'
      testImplementation 'org.junit.jupiter:junit-jupiter-params:X.X.X'
      testImplementation 'org.junit.jupiter:junit-jupiter-engine:X.X.X'
      testImplementation 'org.junit.vintage:junit-vintage-engine:X.X.X'
      ```

      Bedeutung: Lädt das Unit Testing Framework jUnit5 samt der Erweiterungen für Date Driven Tests (`junit-jupiter-params`).
      - Bibliotheken für „Hamcrest Matcher“ lassen sich über `testImplementation 'org.hamcrest:hamcrest:2.1'` integrieren.
  - **Hinweis**: Das bereitgestellte Basic-Clientprojekt (siehe Tipps und Tricks in Moodle) umfasst bereits alle notwendigen Gradle-Konfigurationen mit Softwareversionen welche von uns erfolgreich getestet wurden.