



ZooRoyal IT

# PHPUnit Tutorial

Sebastian Knott

9. April 2018



## Bevor es los geht

- Was ist PHPUnit
- Voraussetzungen
- Exkurs: PSR-4

## Installation

- Composer
- Verzeichnisse
- PHPUnit konfigurieren

## Der erste Test

- Grundgerüst
- Testphasen

## Fortgeschrittene Techniken

- Template Methods
- Data Provider
- Exceptions
- Output testen
- Mark test as ...
- Hamcrest
- Mocks und Mockery

## Komplexes Beispiel Code Dojo

# Bevor es los geht

Was ist PHPUnit



PHPUnit is a **unit testing framework** [↗](#) for the PHP programming language. It is an instance of the **xUnit** [↗](#) architecture for unit testing frameworks that originated with **SUnit** [↗](#) and became popular with **JUnit** [↗](#). PHPUnit was created by Sebastian Bergmann and its development is hosted on **GitHub** [↗](#).

– [Wikipedia.org](#) [↗](#)



### Zweck

PHPUnit is based on the idea that developers should be able to **find mistakes** in their newly committed code quickly and assert that no **code regression** [↗](#) has occurred in other parts of the code base. Much like other **unit testing frameworks** [↗](#), PHPUnit uses **assertions** [↗](#) to verify that the behavior of the specific component - or unit - being tested behaves as expected.

– [Wikipedia.org](#) [↗](#)



- ▶ Für viele Features wird Xdebug benötigt
- ▶ Version von PHPUnit hängt von PHP Version ab


PHPUnit	PHP	Release	Support
7	7.1 – 7.3	2.2.2018	Ends on 7.2.2020
6	7.0 – 7.2	3.2.2017	Ends on 1.2.2019
5	5.6 – 7.0	2.10.2015	Ends on 2.2.2018



- ▶ Einfachere Installation
- ▶ Versionsmanagement
- ▶ Abhängigkeiten verwalten



**PSR** steht für PHP Standards Recommendation. Diese werden von der Framework Interoperability Group rausgegeben. Sie verfolgen das Ziel **PHP-Code möglichst kompatibel** zueinander zu gestalten

**PSR-4**  ist ein Standard, der das **Autoloading** in PHP regelt. Sie definiert den **Zusammenhang zwischen Namespace und Verzeichnissen**.



Voll qualifizierter Klassenname ...

`\\<NamespaceName>(\\<SubNamespaceName>)*\\<ClassName>`

- ▶ ...muss einen Vendor Namespace haben
- ▶ ...kann Subnamespaces haben
- ▶ ...muss mit einem Klassennamen enden
- ▶ ...darf keine Unterstriche mit spezieller Bedeutung enthalten
- ▶ ...muss case-sensitive sein





- ▶ Eine beliebiger gültiger Namespace kann auf einen Basisordner mappen (Präfix)
- ▶ Eine beliebige Menge Subnamespaces nach einem Namespacepräfix muss mit den Verzeichnissen unter dem Basisordner übereinstimmen
- ▶ Der Classname entspricht dem Dateinamen mit der Endung .php
- ▶ Eine Class.php Datei darf nur Class deklarieren

Classname	Präfix	Basisordner	Verzeichnispfad
\ZooRoyal\Class	ZooRoyal\	/ZooRoyalweb/src	/ZooRoyalweb/src/Class.php



Bevor wir Composer benutzen können muss unser Arbeitsverzeichnis als composer-Projekt initialisiert werden.

- ▶ Verzeichnis erstellen und rein wechseln
- ▶ `composer init -q`
- ▶ `composer require --dev "PHPUnit/PHPUnit"`  
`"mockery/mockery"`

Nach der Initialisierung enthält die `composer.json` folgende Einträge.

```
2  "require": {},  
3  "require-dev": {  
4      "phpunit/phpunit": "^6.5",  
5      "mockery/mockery": "^1"  
6  },
```



```
└─ vendor/bin/phpunit
PHPUnit 6.5.6 by Sebastian Bergmann and contributors.

Usage: phpunit [options] UnitTest [UnitTest.php]
       phpunit [options] <directory>

Code Coverage Options:

--coverage-clover <file>   Generate code coverage report in Clover XML format.
--coverage-crap4j <file>   Generate code coverage report in Crap4J XML format.
--coverage-html <dir>      Generate code coverage report in HTML format.
```

- ▶ `vendor/bin/PHPUnit` kann im Terminal ausgeführt werden
- ▶ Aufruf gibt eine Übersicht über Parameter



```
/.
├── src
│   └── ExamplePackage
│       └── Example.php
├── tests
│   └── Unit
│       └── ExamplePackage
│           └── ExampleTest.php
└── vendor
```

Bei PHPUnit hat sich folgende Konvention durchgesetzt

- ▶ Source und Tests in verschiedene Ordner
- ▶ Tests passend zu Testsubjekt benennen
- ▶ PSR-4 einhalten
- ▶ Nur mit PSR-4 adressierte Dateien und Ordner werden groß geschrieben

# Installation

## Composer PSR-4 einrichten



In der **composer.json** müssen die PSR-4 Präfixe für **src** und **tests** angelegt werden.

```
1 {
2     "require": {},
3     "require-dev": {
4         "phpunit/phpunit": "^6.5",
5         "mockery/mockery": "^1"
6     },
7     "autoload": {
8         "psr-4": {
9             "ExampleProject\\": "src",
10            "ExampleProject\\Tests\\": "tests"
11        }
12    }
13 }
```



- ▶ PHPUnit generell vollständig aus dem Terminal nutzbar
- ▶ Mit wachsenden Ansprüchen sehr viele Parameter
- ▶ Konfiguration kann in `PHPUnit.xml` vorgenommen werden

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <phpunit
3      colors="true"
4      bootstrap="vendor/autoload.php">
5      <testsuites>
6          <testsuite name="AllUnitTests">
7              <directory>./tests/Unit/</directory>
8          </testsuite>
9      </testsuites>
10     <filter>
11         <whitelist processUncoveredFilesFromWhitelist="true">
12             <directory suffix=".php">./src</directory>
13         </whitelist>
14     </filter>
15 </phpunit>
```



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <phpunit
3   colors="true"
4   bootstrap="vendor/autoload.php">
5     <testsuites>
6       <testsuite name="AllUnitTests">
```

Das colors-Attribute macht die Ausgabe farbig.

color=true

```
PHPUnit 6.5.6 by Sebastian Bergmann and contributors.
.
Time: 36 ms, Memory: 4.00MB
OK (1 test, 1 assertion)
```

color=false

```
PHPUnit 6.5.6 by Sebastian Bergmann and contributors.
.
Time: 32 ms, Memory: 4.00MB
OK (1 test, 1 assertion)
```



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <phpunit
3   colors="true"
4   bootstrap="vendor/autoload.php">
5   <testsuites>
6     <testsuite name="AllUnitTests">
```

- ▶ bootstrap-Attribute bestimmt eine PHP-Datei, die vor PHPUnit ausgeführt wird
- ▶ vendor/autoload lädt den Autoloader von PHPUnit
- ▶ Macht composer-Libraries verfügbar (PHPUnit-Libraries)
- ▶ Lädt PSR-4 Settings aus der composer.json





```
4 bootstrap="vendor/autoload.php">
5   <testsuites>
6     <testsuite name="AllUnitTests">
7       <directory>./tests/Unit/</directory>
8     </testsuite>
9   </testsuites>
10  <filter>
11    <whitelist processUncoveredFilesFromWhitelist="true">
12      <directory suffix=".php">./src</directory>
13    </whitelist>
14  </filter>
15 </phpunit>
```

**testsuite** markiert eine Menge von Dateien als auszuführende Tests

**directory** lässt PHPUnit im Verzeichnis nach Tests suchen

**file** fügt genau eine Datei als Test hinzu

**exclude** schließt ein bestimmtes Verzeichnis aus



```
4 bootstrap="vendor/autoload.php">
5   <testsuites>
6     <testsuite name="AllUnitTests">
7       <directory>./tests/Unit/</directory>
8     </testsuite>
9   </testsuites>
10  <filter>
11    <whitelist processUncoveredFilesFromWhitelist="true">
12      <directory suffix=".php">./src</directory>
13    </whitelist>
14  </filter>
15 </phpunit>
```

**filter** ist eine Sammlung von whitelists

**whitelist** markiert Dateien für Code Coverage Analyse

**directory** fügt ein Verzeichnis der Whitelist hinzu

**file** fügt einzelnen File der Whitelist hinzu



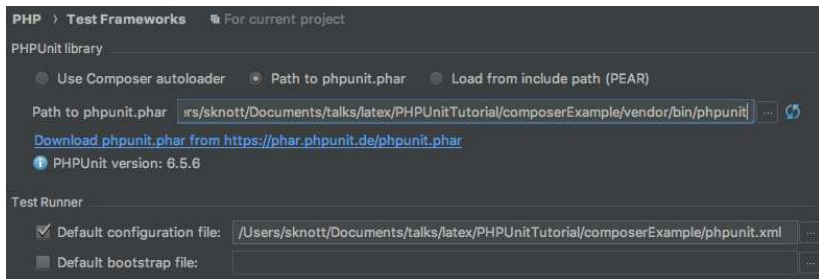
```
└─ vendor/bin/phpunit --configuration phpunit.xml --testsuite AllUnitTests
PHPUnit 6.5.6 by Sebastian Bergmann and contributors.

.                                                                    1 / 1 (100%)

Time: 34 ms, Memory: 4.00MB

OK (1 test, 1 assertion)
```

- `--confi PHPUnit.xml` übergibt PHPUnit die Konfiguration
- `--testsuite AllUnitTests` lässt eine bestimmte Test Suite prüfen  
(Optional: Sonst alle)
- `--filter <filter>` filtert alle Testname nach <filter>



Die Konfiguration kann auch direkt in PhpStorm geladen werden

Wir platzieren den ersten Test für unsere neue Klasse `MyFirstExample` im entsprechenden PSR-4 Pfad

```
/.  
├── src  
│   └── ExamplePackage  
│       └── MyFirstExample.php  
├── tests  
│   └── Unit  
│       └── ExamplePackage  
│           └── MyFirstExampleTest.php  
├── vendor  
└── phpunit.xml
```

```
1  <?php
2  namespace ExampleProject\ExamplePackage;
3
4  class MyFirstExample
5  {
6      /**
7       * This function returns the sum of its Parameters
8       *
9       * @param int $first
10      * @param int $second
11      *
12      * @return int
13      */
14      public function runExample(int $first, int $second) : int
15      {
16          return $first / $second;
17      }
18  }
```

```
1  <?php
2  namespace ExampleProject\Tests\Unit\ExamplePackage;
3
4  use ExampleProject\ExamplePackage\MyFirstExample;
5  use PHPUnit\Framework\TestCase;
6
7  class MyFirstExampleTest extends TestCase
8  {
9      /**
10       * @test
11       */
12     public function runExampleDividesParameters()
13     {
14         $firstParameter = 4;
15         $secondParameter = 4;
16         $expectedResult = $firstParameter / $secondParameter;
17
18         $subject = new MyFirstExample();
19
20         $result = $subject->runExample($firstParameter,
21             ↪ $secondParameter);
22
23         self::assertSame($expectedResult, $result);
24     }
25 }
```



```
1 <?php
2 namespace ExampleProject\Tests\Unit\ExamplePackage;
3
4 use ExampleProject\ExamplePackage\MyFirstExample;
5 use PHPUnit\Framework\TestCase;
6
7 class MyFirstExampleTest extends TestCase
8 {
9     /**
```

- ▶ Der Namespace muss nach PSR-4 korrekt sein
- ▶ Der Name der Testklasse und der Dateiname muss zum Klassennamen passen





```
8 {
9     /**
10      * @test
11      */
12     public function runExampleDividesParameters()
13     {
14         $firstParameter = 4;
```

Test-Methoden müssen als solche gekennzeichnet werden. Hierzu stehen zwei Varianten zur Verfügung

- ▶ @test-Annotation im Methodenkommentar
- ▶ Präfix am Methodennamen (testRunTest...)

Varianten können in einem Testcase nicht gemischt werden.



```
5 use PHPUnit\Framework\TestCase;
6
7 class MyFirstExampleTest extends TestCase
8 {
9     /**
```

Unser Test erweitert die Klasse TestCase.

- ▶ Erfüllen des Interfaces für Test
- ▶ Zugriff auf asserts
- ▶ Template Pattern



Der theoretische Hintergrund von UnitTests wurden in einem anderen Vortrag behandelt. Die Unterlagen hierzu findet man im Wiki.

► [UnitTest-Präsentation](#)

Jeder Unittest läuft implizit immer in genau **drei Phasen** ab.

**Setup** In dieser Phase werden alle Daten zusammengestellt, die für den Test benötigt werden

**Execute** Die zu prüfende Codeunit wird aufgerufen und das Resultat wenn nötig aufgezeichnet

**Validate** Das Resultat wird im Hinblick auf die Äquivalenzklasse des Tests geprüft



```
14 $firstParameter = 4;  
15 $secondParameter = 4;  
16 $expectedResult = $firstParameter / $secondParameter;  
17  
18 $subject = new MyFirstExample();  
19  
20 $result = $subject->runExample($firstParameter, $secondParameter);  
21  
22 self::assertSame($expectedResult, $result);
```

- ▶ Alle benötigten Daten werden zusammengestellt
- ▶ Das erwartete Ergebnis wird festgelegt
- ▶ Das Subject wird erzeugt



```
14 $firstParameter = 4;
15 $secondParameter = 4;
16 $expectedResult = $firstParameter / $secondParameter;
17
18 $subject = new MyFirstExample();
19
20 $result = $subject->runExample($firstParameter, $secondParameter);
21
22 self::assertSame($expectedResult, $result);
```

- ▶ Code Unit wird aufgerufen (`$subject->runExample`)
- ▶ Ergebnis wird in `$result` gespeichert



```
14 $firstParameter = 4;  
15 $secondParameter = 4;  
16 $expectedResult = $firstParameter / $secondParameter;  
17  
18 $subject = new MyFirstExample();  
19  
20 $result = $subject->runExample($firstParameter, $secondParameter);  
21  
22 self::assertSame($expectedResult, $result);
```

- ▶ Abgleich ob das Ergebnis der Erwartung entspricht
- ▶ assert-Methoden lässt PHPUnit die Prüfung vornehmen



`assertSame(a,b)` Prüft ob `a === b`

`assertContains(a,b)` Prüft ob mixed `a` in Iterator|array `b` enthalten ist.

`assertCount(a,b)` Prüft ob `count(b) === a`

`assertEmpty(a)` Prüft ob `empty(a)`

`assertEquals(a,b)` Prüft ob `a == b`

`assertInstanceOf(a,b)` Prüft ob `b instanceof a`

`assertNull(a)` Prüft ob `isset(a)`

`assertRegExp(a,b)` Prüft ob `RegExp a` einen Match in `b` findet



## Assertion

```
self::assertEquals(['a', 's', 'd'], [1, 2, 'd']);
```

## Output

```
Failed asserting that two arrays are equal.
--- Expected
+++ Actual
@@ @@
    Array (
-      0 => 'a'
-      1 => 's'
+      0 => 1
+      1 => 2
```



Code, der zu bestimmten Phasen des **Lebenszyklus** eines Test ausgeführt werden soll kann in dazu vorgesehene Template-Methoden ausgelagert werden.

`setUp()` Wird vor jedem Test ausgeführt

`tearDown()` Wird nach jedem Test ausgeführt

`setUpBeforeClass()` Wird vor jedem TestCase ausgeführt

`tearDownAfterClass()` Wird nach jedem TestCase ausgeführt

```
10  /** @var MySecondExample */
11  private $subject;
12
13  protected function setUp()
14  {
15      $this->subject = new MySecondExample();
16  }
17
18  /**
19   * @test
20   */
21  public function runExampleDividesParameters()
22  {
23      $firstParameter = 8;
24      $secondParameter = 4;
25      $expectedResult = $firstParameter / $secondParameter;
26
27      $result = $this->subject->runExample($firstParameter,
28          ↪ $secondParameter);
29
30      self::assertSame($expectedResult, $result);
31  }
```



Will man in einem Test die Menge der **Eingabeparameter und Resultate variieren** bieten sich **Data Provider** an. Dataprovider können auch für mehrere Tests genutzt werden.

- ▶ Data Provider sind einzelne **public Methoden**
- ▶ Data Provider geben ein **Array von Parametern** zurück mit denen die Testmethode aufgerufen wird
- ▶ Die Verknüpfung von Methode und Data Provider wird mittels **Annotation** hergestellt

```
9 public function runTestSumsUpParametersDataProvider()
10 {
11     return [
12         'division by one' => [4, 1, 4],
13         'division with negativ value' => [5, -5, -1],
14     ];
15 }
16
17 /**
18  * @test
19  * @dataProvider runTestSumsUpParametersDataProvider
20  *
21  * @param int $firstParameter
22  * @param int $secondParameter
23  * @param int $expectedResult
24  */
25 public function runExampleDividesParameters(int $firstParameter, int
    ↪ $secondParameter, int $expectedResult)
26 {
27     $subject = new MyThirdExample();
28
29     $result = $subject->runExample($firstParameter, $secondParameter);
30
31     self::assertSame($expectedResult, $result);
32 }
```

Unter gewissen Umständen möchte man, dass Methoden Exceptions werfen. Dieses Verhalten lässt sich auch mit PHPUnit prüfen.

- ▶ Mittels Annotationen können Eigenschaften von **Exceptions** asserted werden
- ▶ `@expectedException <Exception-Klasse>` legt die erwartete Klasse der Exception fest
- ▶ `@expectedExceptionCode <Code>` legt die erwartete Code fest
- ▶ `@expectedExceptionMessage <Message>` legt die erwartete Nachricht fest

```
17 public function runExample(int $first, int $second) : int
18 {
19     if ($second === 0) {
20         throw new RuntimeException('Division by zero not allowed.',
21             ↪ 1520867252);
22     }
23     return $first / $second;
24 }
```

```
34 /**
35  * @test
36  * @expectedException \RuntimeException
37  * @expectedExceptionCode 1520867252
38  * @expectedExceptionMessage Division by zero not allowed.
39  */
40 public function runExampleShouldthrowExceptionOnDevisionByZero()
41 {
42     $subject = new MyThirdExample();
43
44     $subject->runExample(3, 0);
45 }
```



PHPUnit ermöglicht es die **Ausgabe einer Code-Unit** zu prüfen, die beispielsweise über **echo** oder **print** erzeugt werden. PHPUnit nutzt PHP's **Output Buffering** um die Ausgabe abzufangen.

**expectOutputString()** erwartet, dass die Ausgabe nur aus dem übergebenen String besteht

**expectOutputRegex()** erwartet, dass die Ausgabe auf die übergebenen Regex passt



```
4  class MyFourthExample
5  {
6      const MEINE_AUSGABE = 'Meine Ausgabe';
7
8      /**
9       * This function writes a string to stdout.
10      */
11     public function runExample()
12     {
13         echo self::MEINE_AUSGABE;
14     }
15 }
```

```
7  class MyFourthExampleTest extends TestCase
8  {
9      public function testRunExample()
10     {
11         $this->expectOutputString(MyFourthExample::MEINE_AUSGABE);
12
13         $subject = new MyFourthExample();
14
15         $subject->runExample();
16     }
17 }
```



PHPUnit bietet die Möglichkeit das **Ergebnis eines Tests per Hand** zu setzen. Das ist besonders dann nützlich, wenn Tests unfertig sind, das Ergebnis eines Testlaufs trotzdem aussagekräftig sein soll.

**markTestIncomplete()** markiert einen Test als unvollständig. Optional kann eine Nachricht übergeben werden.

**markTestSkipped()** lässt PHPUnit diesen Test überspringen. Optional kann eine Nachricht übergeben werden.

**fail()** lässt den Test sofort fehlschlagen. Optional kann der Grund für den Fehlschlag übergeben werden.

# Mark test as ...



```
7  class MyFifthExampleTest extends TestCase
8  {
9      public function testRunExampleIGuess()
10     {
11         $this->markTestIncomplete('Still not sure about the Parameters
12         ↳ of runExample()');
13
14         $subject = new MyFifthExample();
15
16         $subject->runExample(3, 0);
17     }
18 }
```

Matchers that can be combined to create flexible expressions of intent.

– [hamcrest.org](https://hamcrest.org) 

- ▶ Detaillierte Fehlerbeschreibung, wenn es keinen Match gibt
- ▶ Komplexe Strukturen mit einfachen Regeln matchen
- ▶ Matcher können wiederverwendet werden
- ▶ Alternative zu PHPUnit-asserts
- ▶ Flexibel im Aufbau

```
19 public function testHamcrestInvalid()  
20 {  
21     $string = 'Ich bin ein string!';  
22  
23     $matcher = H::containsString('bin kein');  
24  
25     MatcherAssert::assertThat($string, $matcher);  
26 }
```

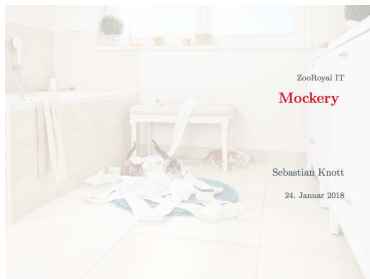
## Ausgabe

```
Hamcrest\AssertionError : Expected: a string containing "bin kein"  
    but: was "Ich bin ein string!"
```

- ▶ Jeder Parameter eines Matchers kann selbst wieder ein Matcher sein.

```
28 public function testHamcrestNestedMatcher()  
29 {  
30     $string = 'Ich bin ein string!';  
31     $array = [$string];  
32  
33     $matcher1 = H::containsString('bin ein');  
34     $matcher2 = H::hasItemInArray($matcher1);  
35  
36     MatcherAssert::assertThat($array, $matcher2);  
37 }
```

- ▶ Auf diese Weise können Asserts **präzise** und **semantisch** im Code formuliert werden



Zu diesem Thema gab es bereits einen Vortrag. Die Unterlagen finden sich in unserem Wiki.

► [Mockery-Präsentation](#)

# Komplexes Beispiel



The screenshot shows an IDE with a project structure on the left and a PHPUnit test run output at the bottom.

**Project Structure:**

- composerExample 0% files, 0% lines
  - src 100% files, 0% lines
    - ComplexExamplePackage 100% files, 0% lines
      - Game.php 100% lines
      - Match.php 80% lines
      - Player.php 100% lines
      - ScoreCounter.php 100% lines
      - Set.php 0% lines
      - Tennis.php 100% lines
  - tests 0% files
    - Integration 0% files
      - ComplexExamplePackage 0% files
        - TennisScoreTest.php
    - Unit 0% files
      - ComplexExamplePackage 0% files
        - GameTest.php
        - MatchTest.php
        - MyComplexExampleTest.php
        - ScoreCounterTest.php
        - SetTest.php
  - vendor 0% files
    - composer.json
    - composer.lock
    - phpunit.xml
  - External Libraries

**PHPUnit Configuration (phpunit.xml):**

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit
  colors="true"
  bootstrap="vendor/autoload.php"
  >
  <testsuite name="AllUnitTests">
    <directory>./tests/Unit/ </directory>
  </testsuite>
  </testsuites>
  <filters>
    <whitelist processUncoveredFilesFromWhitelist="true">
      <directory suffix=".php">./src </directory>
    </whitelist>
  </filters>
</phpunit>
```

**Test Run Output:**

```
All 32 tests passed - 0.00ms

Testing started at 14:18 ...
./usr/local/bin/php --debug-coverage-enabled /Users/sknoth/Documents/talks/late/PHPUnitTutorial/complexExample/vendor/bin/phpunit --coverage-clover
/Users/sknoth/Library/Caches/PhpStorm2817.3/coverage/composerExampleTests1...coverage --configuration
/Users/sknoth/Documents/talks/late/PHPUnitTutorial/complexExample/phpunit.xml /Users/sknoth/Documents/talks/late/PHPUnitTutorial/complexExample/tests --teamcity
PHPUnit 6.5.7 by Sebastian Bergmann and contributors.

Time: 1.11 seconds, Memory: 10.60MB

OK (32 tests, 24 assertions)

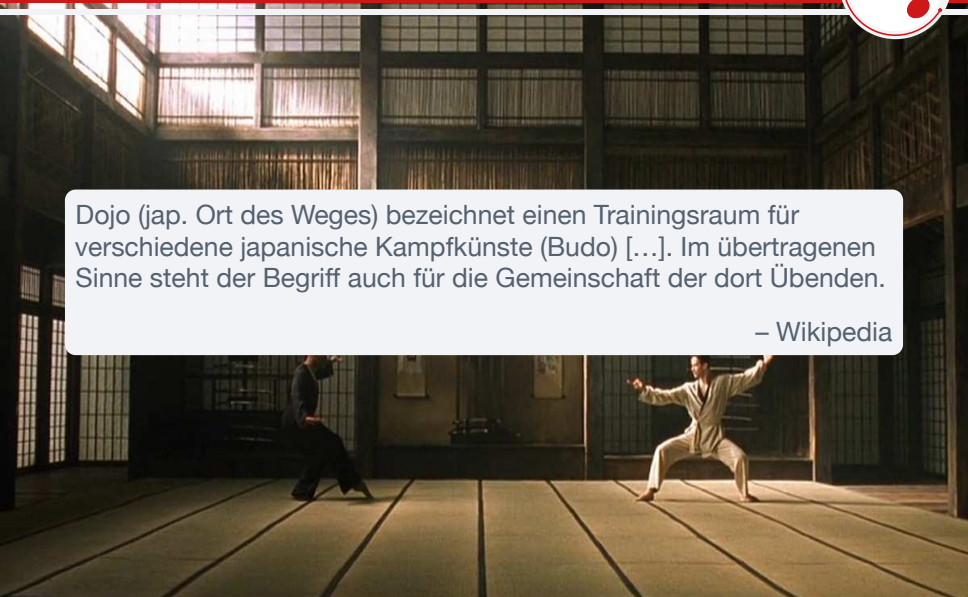
Generating code coverage report in Clover XML format ... done

Process finished with exit code 0
```



Dojo (jap. Ort des Weges) bezeichnet einen Trainingsraum für verschiedene japanische Kampfkünste (Budo) [...]. Im übertragenen Sinne steht der Begriff auch für die Gemeinschaft der dort Übenden.

– Wikipedia



Die Gemeinschaft führt gemeinsam Übungen – so genannte Katas – durch um ihr Wissen in der entsprechenden Disziplin zu vervollkommen.





- ▶ Kleine, unabhängige, fokussierte, in sich geschlossene Übung
- ▶ Übt die Ausführung und Herangehensweise
- ▶ Bietet Raum für gemeinsames Lernen
- ▶ Lösung der Aufgabe erklärtes Nicht-Ziel

## Fokus

- ▶ Pair Programming + TDD = TDD-Game
- ▶ Keine PHPMD Regel darf gebrochen werden
- ▶ Absichtsvolles Testen
  - ▶ Test wird zuallererst dem Pair-Partner erklärt, dann programmiert
  - ▶ Auswahl des Zwecks (use case, Erwartete Eingaben ...)
  - ▶ Auswahl der Kategorie (Black-, White-box)
  - ▶ Gegebenenfalls Auswahl der Äquivalenzklasse.

## Anforderung

- ▶ Eine Klasse muss die folgenden public Methoden haben
  - ▶ **addPointToPlayer(player:PlayerInterface):null** - Soll aufgerufen werden wenn ein Spieler einen Punkt erzielt. Wirft eine Exception, wenn das Spiel vorbei ist
  - ▶ **getScore():string** - Gibt eine Übersicht über alle gespielten Sätze, den aktuellen Punktestand und ob ein Spieler gewonnen hat aus. Es gelten die Tennisregeln mit 2 Gewinnsätzen.