

Spickzettel zu testgetriebener Entwicklung mit PHPUnit

Oliver Klee | typo3-coding@oliverklee.de | @oliklee
<https://github.com/oliverklee/tdd-reader>

Version 3.0.0, 8. Mai 2017, für PHP ≥ 5.6 und TYPO3 CMS 7.6

Lizenz

Dieser Reader ist unter einer *Creative-Commons*-Lizenz lizenziert, und zwar konkret unter der *Namensnennung- Weitergabe unter gleichen Bedingungen 4.0 (CC BY-SA 4.0)*. Das bedeutet, dass ihr den Reader unter diesen Bedingungen für euch kostenlos verbreiten, bearbeiten und nutzen könnt (auch kommerziell):

Namensnennung. Ihr müsst den Namen des Autors (Oliver Klee) nennen. Wenn ihr dabei zusätzlich auch noch die Quelle¹ nennt, wäre das nett. Und wenn ihr mir zusätzlich eine Freude machen möchtet, sagt mir per E-Mail Bescheid.

Weitergabe unter gleichen Bedingungen. Wenn ihr diesen Inhalt bearbeitet oder in anderer Weise umgestaltet, verändert oder als Grundlage für einen anderen Inhalt verwendet, dann dürft ihr den neu entstandenen Inhalt nur unter Verwendung identischer Lizenzbedingungen weitergeben.

Lizenz nennen. Wenn ihr den Reader weiter verbreitet, müsst ihr dabei auch die Lizenzbedingungen nennen oder beifügen.

Die ausführliche Version dieser Lizenz findet ihr online.²

¹<https://github.com/oliverklee/tdd-reader>

²<http://creativecommons.org/licenses/by-sa/4.0/>

Inhaltsverzeichnis

1	Test-Level und -Typen	4
1.1	Test-Level	4
1.2	Test-Typen	4
2	Testphasen	6
3	PhpStorm-Plugins	6
4	Ein Wort zu TYPO3 CMS 8.7	6
5	Benennung von Dateien und Klassen	6
5.1	Dateinamen	6
5.2	Klassennamen	6
6	Struktur von Testklassen	7
6.1	Extbase-Extensions	7
6.2	Nicht-TYPO3-PHP-Projekt mit Composer	8
6.2.1	composer.json	8
6.2.2	Testcase	9
7	Tests ausführen	10
7.1	Nicht-TYPO3-Projekte	10
7.1.1	Auf der Kommandozeile	10
7.1.2	In PhpStorm	10
7.2	TYPO3-Core	10
7.3	TYPO3-Extensions	10
7.3.1	In PhpStorm	10
7.3.2	Mit dem Backend-Modul der PHPUnit-Extension	13
8	Mocks	14
8.1	Warum mocken?	14
8.2	Tools für Mocks	14
9	Auf Exceptions testen	15
9.1	Nur auf die Klasse testen	15
9.2	Auf Klasse, Nachricht und Code testen	15
10	Protected- und Private-Methoden testen	15
10.1	Indirekt testen (empfohlen)	15
10.2	Testing-Unterklasse erstellen	15
11	Abstrakte Klassen testen	16
11.1	Den PHPUnit-Mock-Builder benutzen	16
11.2	Eine konkrete Unterklasse erstellen	16
12	Das Test-Framework der PHPUnit-TYPO3-Extension benutzen	18

13 Gemockte Dateisystem mit vfsStream benutzen	19
13.1 Lauffähige Beispiele	19
13.2 Einrichten	19
13.3 Die Dateien benutzen	19
14 Extbase-Controller testen	21
14.1 Repositories mocken und injecten	21
14.2 Datenfluss vom Repository zum View testen	21
15 PHPUnit-Assertions	23

1 Test-Level und -Typen

Dank für viele dieser Konzepte geht an Filip Defar, der in seinem Blog³ darüber geschrieben hat.

1.1 Test-Level

Unit-Tests sind klein und laufen schnell. Sie sind außerdem normalerweise am schnellsten zu schreiben. Ihr benutzt sie, um alle Details eurer Anwendung zu testen – einschließlich Fälle, die auf GUI-Ebene nicht sinnvoll testbar sind. Es gibt einige Ausnahmen, bei denen Unit-Tests nicht sinnvoll sind, z.,B. Tests für Queries in Extbase-Repositories. Manchmal wird der Begriff *Unit-Tests* auch für alle Tests benutzt, die mit einem Unit-Testing-Framework ausgeführt werden (also auch etwa Systemtests).

Integrationstests laufen langsamer. Ihr benutzt sie, um zu testen, wie Komponenten zusammenarbeiten, und testet normalerweise nicht jedes Detail damit. In der TYPO3-Welt werden die Integrationstests **funktionale Tests** genannt.

Systemtests sind die größten und langsamsten Tests. Ihr benutzt sie, um eure Anwendung als Ganzes zu testen. Systemtests werden normalerweise für relativ wenige, große Usecases benutzt.

1.2 Test-Typen

Blackbox-Tests testen, wie sich eine Einheit oder API nach außen verhält. Sie gehen davon aus, dass ihr kein Wissen darüber habt, wie die Einheit innen arbeitet. Blackbox-Tests erleichtern Refactoring. Sie sind das eine Ende eines Kontinuums, bei dem Whitebox-Tests das andere Extrem bilden.

Whitebox-Tests testen, wie eine Einheit innen arbeitet. Dies erschwert Refactoring und wird nicht empfohlen. Blackbox-Tests erleichtern Refactoring. Sie sind das eine Ende eines Kontinuums, bei dem Blackbox-Tests das andere Extrem bilden. Tests sollten normalerweise Blackbox-Tests sein (oder sehr dunkelgrau).

Funktionale Tests testen das Verhalten einer Einheit. Die allermeisten Tests sind funktionale Tests. Hinweis: In der TYPO3-Welt versteht man unter funktionalen Tests eigentlich *Integrationstests* (und manchmal *Systemtests*).

Akzeptanztests testen, wie sich eine Anwendung auf GUI-Ebene (oder im Web-Interface) verhält. Manchmal wird dieser Begriff als Synonym für *Story-Tests* benutzt.

Regressionstests testen, dass Bugs nicht erneut auftreten. Man schreibt sie optimalerweise direkt zusammen mit dem Bugfix.

Smoketests testen, dass die Anwendung überhaupt irgendwie läuft.

³<http://filipdefar.com/2015/06/tested-be-thy-name.html>

Storytests testen Usecases für User-Stories. Sie werden normalerweise in der Sprache *Gherkin* für Behavior-driven-Design (BDD) geschrieben. Dieser Begriff wird manchmal als synonym für *Akzeptanztests* benutzt.

2 Testphasen

Innerhalb der einzelnen Testmethoden sollte Code für die Testphasen durch Leerzeilen getrennt sein, damit die Struktur direkt sichtbar wird.

<i>Setup</i>	(aufbauen)	<code>setUp()</code> und Code am Anfang der Testmethode (falls nötig)
<i>Exercise</i>	(ausführen)	Methodenaufruf
<i>Verify</i>	(prüfen)	<code>self::assert...()</code>
<i>Teardown</i>	(abbauen)	<code>tearDown()</code>

3 PhpStorm-Plugins

Diese beiden PhpStorm-Plugins sind bei Unit-Tests besonders hilfreich:

- PHPUnit Autocomplete Assistant⁴
- DynamicReturnTypePlugin⁵

4 Ein Wort zu TYPO3 CMS 8.7

Die TYPO3-spezifischen Teile dieses Handouts passen zu TYPO3 CMS 7.6. Für die Version 8.7 gibt es ein paar Unterschiede:

- Benutzt das TYPO3-Testing-Framework⁶ anstelle des Testing-Frameworks des PHPUnit-Extension.
- Die Test-Basisklasse kommt jetzt aus dem TYPO3-Testing-Framework und nicht mehr aus dem TYPO3-Core.
- Da TYPO3 CMS 8.7 kein PHP 5.6 mehr unterstützt, könnt ihr PHPUnit 6 benutzen (das PHP 7 erfordert).

5 Benennung von Dateien und Klassen

5.1 Dateinamen

Dateiname des Produktionscodes	Name der Testdatei
Classes/Domain/Model/Shoe.php	Tests/Unit/Domain/Model/ShoeTest.php
Classes/Service/BaristaService.php	Tests/Unit/Service/BaristaServiceTest.php

5.2 Klassennamen

Name der Klasse im Produktionscode	Name der Testklasse
Shoes\Shop\Domain\Model\Shoe	Shoes\Shop\Tests\Unit\Domain\Model\ShoeTest
Shoes\Shop\Service\BaristaService	Shoes\Shop\Tests\Unit\Service\BaristaServiceTest

⁴<https://plugins.jetbrains.com/plugin/7722-phpunit-autocomplete-assistant>

⁵<https://plugins.jetbrains.com/plugin/7251-dynamicreturntypeplugin>

⁶<https://github.com/TYPO3/testing-framework>

6 Struktur von Testklassen

6.1 Extbase-Extensions

Es gibt auf GitHub dazu auch ein Beispielprojekt (das *Tea-Example*):
https://github.com/oliverklee/ext_tea

```
1 namespace \OliverKlee\Shop\Tests\Unit\Domain\Model;
2
3 use OliverKlee\Shop\Domain\Model\Article;
4
5 class ArticleTest extends \TYPO3\CMS\Core\Tests\UnitTestCase
6 {
7     /**
8      * @var Article;
9      */
10    protected $subject = null;
11
12    protected function setUp()
13    {
14        $this->subject = new Article;
15        $this->subject->initializeObject();
16    }
17
18    /**
19     * @test
20     */
21    public function getNameInitiallyReturnsEmptyString()
22    {
23        self::assertSame('', $this->subject->getName());
24    }
25
26    /**
27     * @test
28     */
29    public function setNameSetsName()
30    {
31        $name = 'foo bar';
32
33        $this->subject->setName($name);
34
35        self::assertSame($name, $this->subject->getName());
36    }
37
38    // ...
39 }
```

6.2 Nicht-TYPO3-PHP-Projekt mit Composer

Es gibt auf GitHub dazu auch ein leeres Startprojekt:
<https://github.com/oliverklee/tdd-seed>

6.2.1 composer.json

Diese composer.json installiert PHPUnit und vfsStream für **PHP bis 5.6**:

```
1 {
2     "require-dev": {
3         "phpunit/phpunit": "^5.7.19",
4         "mikey179/vfsStream": "^1.6.4"
5     },
6     "autoload": {
7         "psr-4": {
8             "...": ""
9         }
10    },
11    "autoload-dev": {
12        "psr-4": {
13            "...": ""
14        }
15    }
16 }
```

Diese composer.json installiert PHPUnit und vfsStream für **PHP ab Version 7**:

```
1 {
2     "require-dev": {
3         "phpunit/phpunit": "^6.0.13",
4         "mikey179/vfsStream": "^1.6.4"
5     },
6     "autoload": {
7         "psr-4": {
8             "...": ""
9         }
10    },
11    "autoload-dev": {
12        "psr-4": {
13            "...": ""
14        }
15    }
16 }
```

6.2.2 Testcase

```
1 namespace OliverKlee\Books\Tests\Unit\Domain\Model;
2
3 use OliverKlee\Books\Domain\Model;
4
5 class BookTest extends \PHPUnit_Framework_TestCase
6 {
7     /**
8      * @var Book
9      */
10    protected $subject = null;
11
12    protected function setUp()
13    {
14        $this->subject = new Book();
15    }
16
17    /**
18     * @test
19     */
20    public function getTitleInitiallyReturnsEmptyString()
21    {
22        self::assertSame('', $this->subject->getTitle());
23    }
24
25    /**
26     * @test
27     */
28    public function setTitleSetsTitle()
29    {
30        $title = 'foo bar';
31
32        $this->subject->setTitle($title);
33
34        self::assertSame($title, $this->subject->getTitle());
35    }
36 }
```

7 Tests ausführen

7.1 Nicht-TYPO3-Projekte

7.1.1 Auf der Kommandozeile

```
1 vendor/bin/phpunit Test/
```

7.1.2 In PhpStorm

1. Settings > Languages & Frameworks > PHP > PHPUnit
2. PHPUnit library > Use Composer autoloader
3. PHPUnit library > Path to script: `vendor/autoload`
4. OK
5. auf den Order `Tests/` rechtsklicken (oder einen anderen Ordner oder eine Testdatei)
6. Run 'Tests'

7.2 TYPO3-Core

Im TYPO-Wiki gibt es Anleitungen, wie man die Unit-Tests⁷ und die funktionale Tests⁸ des Core ausführt.

7.3 TYPO3-Extensions

7.3.1 In PhpStorm

Für eine existierende TYPO3-Installation im Composer-Modus: Bei diesem Ansatz werden alle installierten Extensions geladen, sodass ihr auch die Features der PHPUnit-Extension nutzen könnt.

1. Settings > Languages & Frameworks > PHP > PHPUnit
2. PHPUnit library > Use Composer autoloader
3. PHPUnit library > Path to script: `vendor/autoload` aus dem Document-Root der TYPO3-Installation
4. Test runner > Default configuration file: `typo3/sysext/core/Build/UnitTests.xml` innerhalb des Document-Roots der TYPO3-Installation
5. OK
6. Run > Edit Configurations
7. Defaults > PHPUnit

⁷https://wiki.typo3.org/Unit_Testing_TYPO3

⁸https://wiki.typo3.org/Functional_testing

8. Command Line > Environment variables

9. zwei Variablen hinzufügen:

- `TYPO3_CONTENT = Development`
- `TYPO3_PATH_WEB` = der absolute Pfad zum Document-Root der TYPO3-Installation (ohne den Slash am Ende)

10. auf den Order **Tests/** rechtsklicken (oder einen anderen Ordner oder eine Testdatei)

11. Run 'Tests'

Für eine existierende TYPO3-Installation im Klassik-Modus (Nicht-Composer-Modus):

In diesem Fall werdet ihr keine Klassen aus anderen Extensions autoloade können, d. h., ihr werdet auch keine Features der PHPUnit-Extension nutzen können (und auch nicht von anderen Extension-Abhängigkeiten).

1. Wenn ihr den TYPO3-Source per git statt als TAR-Paket heruntergeladen habt, braucht ihr ein `composer install` im TYPO3-Source-Verzeichnis.

2. Settings > Languages & Frameworks > PHP > PHPUnit

3. PHPUnit library > Use Composer autoloader

4. PHPUnit library > Path to script: `vendor/autoload` innerhalb des TYPO3-Source

5. Test runner > Default configuration file: `typo3/sysext/core/Build/UnitTests.xml` innerhalb des TYPO3-Document-Roots

6. OK

7. Run > Edit Configurations

8. Defaults > PHPUnit

9. Command Line > Environment variables

10. zwei Variablen hinzufügen:

- `TYPO3_CONTENT = Development`
- `TYPO3_PATH_WEB` = der absolute Pfad zum Document-Root der TYPO3-Installation (ohne den Slash am Ende)

11. auf den Order **Tests/** rechtsklicken (oder einen anderen Ordner oder eine Testdatei)

12. Run 'Tests'

Ohne eine existierende TYPO3-Installation Dieser Ansatz benutzt das TYPO3-Extension-Skelett⁹ von Helmut Hummel und Nicole Cordes.

Fügt die folgenden Abschnitte zur `composer.json` eurer Extension hinzu:

⁹https://github.com/helhum/ext_scaffold

```

1  "require": {
2      "typo3/cms": "~7.6.0"
3  },
4  "require-dev": {
5      "namelesscoder/typo3-repository-client": "^1.2",
6      "nimut/testing-framework": "^1.0",
7      "mikey179/vfsStream": "^1.4",
8      "phpunit/phpunit": "^4.7 || ^5.0"
9  },
10 "config": {
11     "vendor-dir": ".Build/vendor",
12     "bin-dir": ".Build/bin"
13 },
14 "scripts": {
15     "post-autoload-dump": [
16         "mkdir -p .Build/Web/typo3conf/ext/",
17         "[ -L .Build/Web/typo3conf/ext/tea ] || ln -snvf ../../../../.Build/Web/typo3conf/ext/tea"
18     ]
19 },
20 "extra": {
21     "typo3/cms": {
22         "cms-package-dir": "${vendor-dir}/typo3/cms",
23         "web-dir": ".Build/Web"
24     }
25 }

```

Ersetzt dabei **tea** in Zeile 18 durch den Schlüssel eurer Extension.

Wenn ihr außerdem noch andere, im TER verfügbarer Extensions in den Tests nutzen möchtet (z.B. die PHPUnit-Extension), fügt noch diese Abschnitte zu eurer `composer.json` hinzu (bzw. mergt sie):

```

1  "repositories": [
2  {
3      "type": "composer",
4      "url": "https://composer.typo3.org/"
5  }
6  ],
7  "require-dev": {
8      "typo3-ter/phpunit": "*"
9  },

```

Führt danach diese Schritte in PhpStorm aus:

1. Settings > Languages & Frameworks > PHP > PHPUnit
2. PHPUnit library > Use Composer autoloader
3. PHPUnit library > Path to script:
 - auf den Button *Show hidden files and directories* klicken `.Build/vendor/autoload.php` im Verzeichnis der Extension

-
4. Test runner > Default configuration file: `typo3/sysext/core/Build/UnitTests.xml` im TYPO3-Source-Verzeichnis
 5. OK
 6. Run > Edit Configurations
 7. Defaults > PHPUnit
 8. Command Line > Environment variables
 9. zwei Variablen hinzufügen:
 - `TYPO3_CONTENT` = `Development`
 - `TYPO3_PATH_WEB` = der absolute Pfad zu `.Build/Web` im Verzeichnis der Extension (ohne den Slash am Ende)
 10. auf den Order **Tests/** rechtsklicken (oder einen anderen Ordner oder eine Testdatei)
 11. Run 'Tests'

7.3.2 Mit dem Backend-Modul der PHPUnit-Extension

Dieser Ansatz funktioniert sowohl für TYPO3-Installationen im Composer-Modus als auch im Klassik-Modus.

Dabei werden alle installierten Extensions geladen (inklusive der PHPUnit-Extension), so dass ihr die Features der PHPUnit-Extension nutzen könnt.

Allerdings werden die Tests dann im Kontext der aktuellen Backend-Session ausgeführt, was die Tests sehr zehrbrechlich macht. Für die meisten Unit-Tests von Extensions funktioniert das, für funktionale Tests wird es hingegen meistens nicht funktionieren.

1. Admin > PHPUnit

8 Mocks

8.1 Warum mocken?

- um eine Methode „auszuschalten“ (damit sie nicht in die DB schreibt, kein Cruise-Missile abschießt etc.) und `null` zurückzugeben
- um einer Methode einen bestimmten Rückgabewert zu geben oder sie eine Exception werfen zu lassen
- um zu testen, dass eine Methode auf eine bestimmte Art und Weise aufgerufen wird

8.2 Tools für Mocks

Prophecy: Das empfohlene, einfach benutzbare, aktuelle Mocking-Framework. Allerdings kann es keine partiellen Mocks erzeugen.¹⁰

PHPUnit mocks: Die alte Art, Mocks zu erzeugen. Mocking ist damit etwas unhandlich, aber dafür kann es auch partielle Mocks erzeugen.¹¹

Mockery: Auch sehr elegant.¹²

¹⁰Prophecy-Cheatsheet:

<https://github.com/oliverklee/tdd-reader/blob/master/AdditionalDocuments/prophecy-cheatsheet.pdf>

¹¹PHPUnit-Mocking-Cheatsheet:

<https://github.com/oliverklee/tdd-reader/blob/master/AdditionalDocuments/mocking-cheatsheet.pdf>

¹²<https://github.com/mockery/mockery>

9 Auf Exceptions testen

9.1 Nur auf die Klasse testen

```
1  /**
2   * @test
3   * @expectedException \UnexpectedValueException
4   */
5  public function createBreadWithNegativeSizeThrowsException()
6  {
7      $this->subject->createBread(-1);
8  }
```

9.2 Auf Klasse, Nachricht und Code testen

```
1  /**
2   * @test
3   * @expectedException \UnexpectedValueException
4   * @expectedExceptionMessage size must be > 0.
5   * @expectedExceptionCode 1323700434
6   */
7  public function createBreadWithNegativeSizeThrowsException()
8  {
9      $this->subject->createBread(-1);
10 }
```

10 Protected- und Private-Methoden testen

10.1 Indirekt testen (empfohlen)

Tests sollten nur Verhalten testen, das von außen sichtbar ist. Daher solltet ihr Protected-Methoden über Test für Public-Methoden testen, die Gebrauch von den Protected-Methoden machen.

10.2 Testing-Unterklasse erstellen

Wenn ihr Protected-Methoden habt, die explizit Teil der API für Unterklassen sind, könnt ihr in **Fixtures/** eine Testing-Unterklasse erstellen, die eine Public-Methode hat, die die entsprechende Protected-Methode aufruft.

11 Abstrakte Klassen testen

11.1 Den PHPUnit-Mock-Builder benutzen

Dies erzeugt eine Instanz der abstrakten Klassen, wobei alle abstrakten Methoden gemockt werden.

```
1 namespace OliverKlee\Coffee\Tests\Unit\Domain\Model;
2
3 use OliverKlee\Coffee\Domain\Model\AbstractBeverage;
4
5 class AbstractBeverageTest
6 {
7     /**
8      * @var AbstractBeverage|\PHPUnit\Framework\MockObject\MockObject
9      */
10    protected $subject = null;
11
12    protected function setUp()
13    {
14        $this->subject = $this->getMockForAbstractClass(
15            AbstractBeverage::class
16        );
17    }
18 }
```

11.2 Eine konkrete Unterklasse erstellen

Dies wird empfohlen, wenn die Subklasse zusätzliches oder spezifisches Verhalten haben soll.

Erzeugt in Tests/Unit/Unit/Domain/Model/Fixtures/ eine Unterklasse der abstrakten Klasse:

```
1 namespace OliverKlee\Coffee\Tests\Unit\Domain\Model\Fixtures;
2
3 class TestingBeverage extends \OliverKlee\Coffee\Domain\Model\AbstractBeverage
4 {
5     // ...
6 }
```

Dann könnt ihr die konkrete Unterklasse in euren Unit-Tests use und instanziiieren.

```
1 use OliverKlee\Coffee\Tests\Unit\Domain\Model\Fixtures\TestingBeverage;
2
3 class AbstractBeverageTest
4 {
5     /**
6      * @var TestingBeverage
7      */
8     protected $subject = null;
9 }
```

```
9
10     protected function setUp()
11     {
12         $this->subject = new TestingBeverage();
13     }
```

12 Das Test-Framework der PHPUnit-TYPO3-Extension benutzen

```
1 class DataMapperTest extends \Tx_Phpunit_TestCase
2 {
3     /**
4      * @var \Tx_Phpunit_Framework
5      */
6     protected $testingFramework = null;
7
8     protected $subject = null;
9
10    protected function setUp()
11    {
12        $this->testingFramework = new \Tx_Phpunit_Framework('tx_oelib');
13
14        $this->subject = new ...;
15    }
16
17    protected function tearDown()
18    {
19        $this->testingFramework->cleanUp();
20    }
21
22    /**
23     * @test
24     */
25    public function findWithUidOfExistingRecordReturnsModelDataFromDatabase()
26    {
27        $title = 'foo';
28        $uid = $this->testingFramework->createRecord(
29            'tx_oelib_test', ['title' => $title]
30        );
31
32        self::assertSame($title, $this->subject->find($uid)->getTitle());
33    }
```

13 Gemockte Dateisystem mit vfsStream benutzen

13.1 Lauffähige Beispiele

Die funktionalen Tests zur FileUtility-Klasse im Tea-Beispiel¹³ zeigen, wie Tests mit vfsStream aussehen können.

13.2 Einrichten

```
1 use org\bovigo\vfs\vfsStream;
2 use org\bovigo\vfs\vfsStreamDirectory;
3
4 /**
5  * @var |org\bovigo\vfs\vfsStreamFile
6  */
7 protected $moreStuff;
8
9 protected function setUp()
10 {
11     // This is the same as ::register and ::setRoot.
12     $this->root = vfsStream::setup('home');
13     $this->targetFilePath = vfsStream::url('home/target.txt');
14
15     $this->subject = new ...
16 }
```

13.3 Die Dateien benutzen

```
1 /**
2  * @test
3  */
4 public function concatenateWithOneEmptySourceFileCreatesEmptyTargetFile()
5 {
6     // This is one way to create a file with contents, using PHP's file functions.
7     $sourceFileName = vfsStream::url('home/source.txt');
8     // Just calling vfsStream::url does not create the file yet.
9     // We need to write into it to create it.
10    file_put_contents($sourceFileName, '');
11
12    $this->subject->concatenate($this->targetFilePath, [$sourceFileName]);
13
14    self::assertSame('', file_get_contents($this->targetFilePath));
15 }
16
```

¹³https://github.com/oliverklee/ext_tea

```
17  /**
18   * @test
19   */
20  public function concatenateWithOneFileCopiesContentsFromSourceFileToTargetFile()
21  {
22      // This is vfsStream's way of creating a file with contents.
23      $contents = 'Hello world!';
24      $sourceFileName = vfsStream::url('home/source.txt');
25      vfsStream::newFile('source.txt')->at($this->root)->setContent($contents);
26
27      $this->subject->concatenate($this->targetFilePath, [$sourceFileName]);
28
29      self::assertSame($contents, file_get_contents($this->targetFilePath));
30  }
```

14 Extbase-Controller testen

Bei sauber gebauten Controllern werdet ihr hauptsächlich testen, dass die Daten korrekt vom Request zu den Repositories gehen sowie von den Repositories zum View. Außerdem werdet ihr gelegentlich die Forwards und Redirects testen wollen.

14.1 Repositories mocken und injecten

```
1 protected function setUp()
2 {
3     $this->subject = new TestimonialController();
4
5     $this->viewProphecy = $this->prophesize(ViewInterface::class);
6     $this->view = $this->viewProphecy->reveal();
7     $this->inject($this->subject, 'view', $this->view);
8
9     $this->testimonialRepositoryProphecy
10         = $this->prophesize(TestimonialRepository::class);
11     $this->testimonialRepository
12         = $this->testimonialRepositoryProphecy->reveal();
13     $this->inject(
14         $this->subject,
15         'testimonialRepository',
16         $this->testimonialRepository
17     );
18 }
```

Die `inject`-Method ist eine Hilfsmethode der Testcase-Basisklasse.

Wenn ihr für bessere Performance im Controller explizite `inject...`-Methoden benutzt statt der `@inject`-Annotationen, könnt ihr diese benutzen:

```
1 $this->subject->injectTestimonialRepository($this->testimonialRepository);
```

14.2 Datenfluss vom Repository zum View testen

```
1 /**
2  * @test
3  */
4 public function indexActionPassesAllTestimonialsAsTestimonialsToView()
5 {
6     $allTestimonials = new ObjectStorage();
7     $this->testimonialRepositoryProphecy->findAll()
8         ->willReturn($allTestimonials);
9
10    $this->viewProphecy->assign('testimonials', $allTestimonials)
```

```
11         ->shouldBeCalled();
12
13     $this->subject->indexAction();
14 }
```

15 PHPUnit-Assertions

Diese Liste ist aktuell für PHPUnit 5.7.x.

```
assertArrayHasKey()  
assertClassHasAttribute()  
assertArraySubset()  
assertClassHasStaticAttribute()  
assertContains()  
assertContainsOnly()  
assertContainsOnlyInstancesOf()  
assertCount()  
assertDirectoryExists()  
assertDirectoryIsReadable()  
assertDirectoryIsWritable()  
assertEmpty()  
assertEqualXMLStructure()  
assertEquals()  
assertFalse()  
assertFileEquals()  
assertFileExists()  
assertFileIsReadable()  
assertFileIsWritable()  
assertGreaterThan()  
assertGreaterThanOrEqual()  
assertInfinite()  
assertInstanceOf()  
assertInternalType()  
assertIsReadable()  
assertIsWritable()  
assertJsonFileEqualsJsonFile()  
assertJsonStringEqualsJsonFile()  
assertJsonStringEqualsJsonString()  
assertLessThan()  
assertLessThanOrEqual()  
assertNan()  
assertNull()  
assertObjectHasAttribute()  
assertRegExp()  
assertStringMatchesFormat()  
assertStringMatchesFormatFile()  
assertSame()  
assertStringEndsWith()  
assertStringEqualsFile()  
assertStringStartsWith()  
assertThat()  
assertTrue()
```

```
assertXmlFileEqualsXmlFile()  
assertXmlStringEqualsXmlFile()  
assertXmlStringEqualsXmlString()
```