Spickzettel zu testgetriebener Entwicklung mit PHPUnit

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

Version 3.1.0, 8. September 2017, für PHP >= 7.0 und TYPO3 CMS 7.6-8.7

Lizenz

[1] https://github.com/oliverklee/tdd-reader
[2] http://creativecommons.org/licenses/by-sa/4.0/

Inhaltsverzeichnis

1 Test-Level und -Typen

Dank für viele dieser Konzepte geht an Filip Defar, der in seinem Blog[3] 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 einen 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 Test 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 schreib 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.

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

3 PhpStorm-Plugins

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

- PHPUnit Autocomplete Assistant[4]
- DynamicReturnTypePlugin[5]

4 Benennung von Dateien und Klassen

4.1 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`

4.2 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`

5 Struktur von Testklassen

5.1 Test-Basisklasse

5.1.1 Nicht-TYPO3-Projekt

PHPUnit <= 5.7 `\PHPUnit_Framework_TestCase`

PHPUnit >= 6.0 `\PHPUnit\Framework\TestCase`

5.1.2 TYPO3-Projekt

5.1.3 Mit `nimut/testing-framework`

Unit-Tests `\Nimut\TestingFramework\TestCase\UnitTestCase`

ViewHelper-Tests `\Nimut\TestingFramework\TestCase\ViewHelperBaseTestcase`

Functional-Tests `\Nimut\TestingFramework\TestCase\FunctionalTestCase`

5.1.4 Mit `typo3/testing-framework`

Unit-Tests `\TYPO3\TestingFramework\Core\Unit\UnitTestCase`

Functional-Tests `\TYPO3\TestingFramework\Core\Unit\FunctionalTestCase`

5.2 Nicht-TYPO3-PHP-Projekt mit Composer

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

5.2.1 composer.json

Diese composer.json installiert PHPUnit und vfsStream:

```
{
    "require-dev": {
        "phpunit/phpunit": "^6.3.0",
        "mikey179/vfsStream": "^1.6.0"
    },
    "autoload": {
        "psr-4": {
            "..."
        }
    },
    "autoload-dev": {
        "psr-4": {
            "..."
        }
```

```
    }
}
```

5.2.2 Testcase

```
namespace OliverKlee\Books\Tests\Unit\Domain\Model;

use OliverKlee\Books\Domain\Model;

class BookTest extends \PHPUnit\Framework\TestCase
{
    /**
     * @var Book
     */
    protected $subject = null;

    protected function setUp()
    {
        $this->subject = new Book();
    }

    /**
     * @test
     */
    public function getTitleInitiallyReturnsEmptyString()
    {
        self::assertSame('', $this->subject->getTitle());
    }

    /**
     * @test
     */
    public function setTitleSetsTitle()
    {
        $title = 'foo bar';

        $this->subject->setTitle($title);

        self::assertSame($title, $this->subject->getTitle());
    }
}
```

5.3 Extbase-Extensions

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

```php
namespace \OliverKlee\Shop\Tests\Unit\Domain\Model;

use OliverKlee\Shop\Domain\Model\Article;

class ArticleTest extends \Nimut\TestingFramework\TestCase\UnitTestCase
{
    /**
     * @var Article;
     */
    protected $subject = null;

    protected function setUp()
    {
        $this->subject = new Article;
        $this->subject->initializeObject();
    }

    /**
     * @test
     */
    public function getNameInitiallyReturnsEmptyString()
    {
        self::assertSame('', $this->subject->getName());
    }

    /**
     * @test
     */
    public function setNameSetsName()
    {
        $name = 'foo bar';

        $this->subject->setName($name);

        self::assertSame($name, $this->subject->getName());
    }

    // ...
}
```

6 Tests ausführen

6.1 Nicht-TYPO3-Projekte

6.1.1 Auf der Kommandozeile

```
vendor/bin/phpunit Tests/
```

6.1.2 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'

6.2 TYPO3-Core

Im TYPO-Wiki gibt es Anleitungen, wie man die Unit-Tests[6] und die funktionale Tests[7] des Core ausführt.

6.3 TYPO3-Extensions

6.3.1 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. OK
5. Run > Edit Configurations
6. Defaults > PHPUnit
7. Use alternative configuration file: `.../vendor/nimut/testing-framework/res/Configuration/UnitTe`
   bzw. `.../vendor/nimut/testing-framework/res/Configuration/FunctionalTests.xml`

[6] `https://wiki.typo3.org/Unit_Testing_TYPO3`
[7] `https://wiki.typo3.org/Functional_testing`

8. Command Line > Environment variables
9. vier Variablen hinzufügen (nur notwendig für functionale Tests):
    - `typo3DatabaseUsername`
    - `typo3DatabasePassword`
    - `typo3DatabaseHost` (normalerweise `localhost`)
    - `typo3DatabaseName`
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 autoloaden 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. OK
6. Run > Edit Configurations
7. Defaults > PHPUnit
8. Use alternative configuration file: `.../vendor/nimut/testing-framework/res/Configuration/UnitTe`
   bzw. `.../vendor/nimut/testing-framework/res/Configuration/FunctionalTests.xml`
9. Command Line > Environment variables
10. vier Variablen hinzufügen (nur notwendig für functionale Tests):
    - `typo3DatabaseUsername`
    - `typo3DatabasePassword`
    - `typo3DatabaseHost` (normalerweise `localhost`)
    - `typo3DatabaseName`
11. OK
12. auf den Order `Tests/` rechtsklicken (oder einen anderen Ordner oder eine Testdatei)
13. Run 'Tests'

Ohne eine existierende TYPO3-Installation Dieser Ansatz benutzt das TYPO3-Extension-Skelett[8] von Helmut Hummel und Nicole Cordes.

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

```
1  "require": {
2    "typo3/cms": "~7.6.0"
3  },
4  "require-dev": {
5    "namelesscoder/typo3-repository-client": "^1.2.0",
6    "nimut/testing-framework": "^2.0.0",
7    "phpunit/phpunit": "^5.7.0",
8    "mikey179/vfsStream": "^1.6.0"
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):

```
"repositories": [
{
  "type": "composer",
  "url": "https://composer.typo3.org/"
}
],
"require-dev": {
  "typo3-ter/phpunit": "*"
},
```

Führt danach diese Schritte in PhpStorm aus:

1. Settings > Languages & Frameworks > PHP > PHPUnit

[8] `https://github.com/helhum/ext_scaffold`

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. OK
5. Run > Edit Configurations
6. Defaults > PHPUnit
7. Command Line > Environment variables
8. vier Variablen hinzufügen (nur notwendig für functionale Tests):
   - `typo3DatabaseUsername`
   - `typo3DatabasePassword`
   - `typo3DatabaseHost` (normalerweise `localhost`)
   - `typo3DatabaseName`
9. auf den Order `Tests/` rechtsklicken (oder einen anderen Ordner oder eine Testdatei)
10. Run 'Tests'

7 Mocks

7.1 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

7.2 Tools für Mocks

Prophecy: Das empfohlene, einfach benutzbare, aktuelle Mocking-Framework. Allerdings kann es keine partiellen Mocks erzeugen.[9]

PHPUnit-Mocks: Die alte Art, Mocks zu erzeugen. Mocking ist damit etwas unhandlich, aber dafür kann es auch partielle Mocks erzeugen.[10]

Mockery: Auch sehr elegant.[11]

[9] Prophecy-Cheatsheet:
`https://github.com/oliverklee/tdd-reader/blob/master/AdditionalDocuments/prophecy-cheatsheet.pdf`

[10] PHPUnit-Mocking-Cheatsheet:
`https://github.com/oliverklee/tdd-reader/blob/master/AdditionalDocuments/mocking-cheatsheet.pdf`

[11] `https://github.com/mockery/mockery`

8 Auf Exceptions testen

Laut einem Blogpost[12] von Sebastian Bergmann (dem Autor von PHPUnit) ist dies die aktuelle empfohlene Praxis.

8.1 Nur auf die Klasse testen

```php
/**
 * @test
 */
public function createBreadWithNegativeSizeThrowsException()
{
    $this->expectException(\UnexpectedValueException::class);

    $this->subject->createBread(-1);
}
```

8.2 Auf Klasse, Nachricht und Code testen

```php
/**
 * @test
 */
public function createBreadWithNegativeSizeThrowsException()
{
    $this->expectException(\UnexpectedValueException::class);
    $this->expectExceptionMessage('$size must be > 0.');
    $this->expectExceptionCode(1323700434);

    $this->subject->createBread(-1);
}
```

9 Protected- und Private-Methoden testen

9.1 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.

9.2 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.

[12] `https://thephp.cc/news/2016/02/questioning-phpunit-best-practices`

10 Abstrakte Klassen testen

10.1 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     }
```

10.2 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 usen und instanziieren.

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

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

```php
class DataMapperTest extends \Tx_Phpunit_TestCase
{
    /**
     * @var \Tx_Phpunit_Framework
     */
    protected $testingFramework = null;

    protected $subject = null;

    protected function setUp()
    {
        $this->testingFramework = new \Tx_Phpunit_Framework('tx_oelib');

        $this->subject = new ...;
    }

    protected function tearDown()
    {
        $this->testingFramework->cleanUp();
    }

    /**
     * @test
     */
    public function findWithUidOfExistingRecordReturnsModelDataFromDatabase()
    {
        $title = 'foo';
        $uid = $this->testingFramework->createRecord(
            'tx_oelib_test', ['title' => $title]
        );

        self::assertSame($title, $this->subject->find($uid)->getTitle());
    }
```

12 Gemockte Dateisystem mit vfsStream benutzen

12.1 Lauffähige Beispiele

Die funktionalen Tests zur FileUtility-Klasse im Tea-Beispiel[13] zeigen, wie Tests mit vfsStream aussehen können.

12.2 Einrichten

```
use org\bovigo\vfs\vfsStream;
use org\bovigo\vfs\vfsStreamDirectory;

/**
 * @var \org\bovigo\vfs\vfsStreamFile
 */
protected $moreStuff;

protected function setUp()
{
    // This is the same as ::register and ::setRoot.
    $this->root = vfsStream::setup('home');
    $this->targetFilePath = vfsStream::url('home/target.txt');

    $this->subject = new ...
}
```

12.3 Die Dateien benutzen

```
/**
 * @test
 */
public function concatenateWithOneEmptySourceFileCreatesEmptyTargetFile()
{
    // This is one way to create a file with contents, using PHP's file functions.
    $sourceFileName = vfsStream::url('home/source.txt');
    // Just calling vfsStream::url does not create the file yet.
    // We need to write into it to create it.
    file_put_contents($sourceFileName, '');

    $this->subject->concatenate($this->targetFilePath, [$sourceFileName]);

    self::assertSame('', file_get_contents($this->targetFilePath));
}

```

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

```php
/**
 * @test
 */
public function concatenateWithOneFileCopiesContentsFromSourceFileToTargetFile()
{
    // This is vfsStream's way of creating a file with contents.
    $contents = 'Hello world!';
    $sourceFileName = vfsStream::url('home/source.txt');
    vfsStream::newFile('source.txt')->at($this->root)->setContent($contents);

    $this->subject->concatenate($this->targetFilePath, [$sourceFileName]);

    self::assertSame($contents, file_get_contents($this->targetFilePath));
}
```

13 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.

13.1 Repositories mocken und injecten

```
protected function setUp()
{
    $this->subject = new TestimonialController();

    $this->viewProphecy = $this->prophesize(ViewInterface::class);
    $this->view = $this->viewProphecy->reveal();
    $this->inject($this->subject, 'view', $this->view);

    $this->testimonialRepositoryProphecy
        = $this->prophesize(TestimonialRepository::class);
    $this->testimonialRepository
        = $this->testimonialRepositoryProphecy->reveal();
    $this->inject(
        $this->subject,
        'testimonialRepository',
        $this->testimonialRepository
    );
}
```

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:

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

13.2 Datenfluss vom Repository zum View testen

```
/**
  * @test
  */
public function indexActionPassesAllTestimonialsAsTestimonialsToView()
{
    $allTestimonials = new ObjectStorage();
    $this->testimonialRepositoryProphecy->findAll()
        ->willReturn($allTestimonials);

    $this->viewProphecy->assign('testimonials', $allTestimonials)
```

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

14 PHPUnit-Assertions

This list is current for PHPUnit 6.3.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()
```

Literatur

[aSP11] Sebastian Bergmann Stefan Priebsch. *Real-World Solutions for Developing High-Quality PHP Frameworks and Applications.* Wiley Publishing, Indianapois, 2011.

[aSP13] Sebastian Bergmann Stefan Priebsch. *Softwarequalität in PHP-Projekten.* Carl Hanser, München, 2013.

[Bec03] Kent Beck. *Test-driven Development By Example.* Addison Wesley, Boston, 2003.

[Fea05] Michael C. Feathers. *Working Effectively wth Legacy Code.* Prentice Hall, Upper Saddle River, 2005.

[Fie14] Jay Fields. *Working Effectively wth Unit Tests.* Leanpub, 2014.

[Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code.* Addison Wesley, Boston, 2007.