Cheatsheet for test-driven development with PHPUnit

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

Version 3.1.0, September 8, 2017, for TYPO3 PHP >= 7.0 and CMS 7.6-8.7

License

This handout is licensed under a *Creative Commons* license, in this case under an *Attribution-ShareAlike 4.0 (CC BY-SA 4.0)*. This means that you can use, edit and distribute this handout (even commercially) under the following conditions:

Attribution. You need to give credit to the author (me) by listing my name (Oliver Klee). If you also list the source[1], that would be nice. And if you want to make me happy, please drop me an e-mail if you use this document.

ShareAlike. If you edit or change this document or use it as a basis for some other document, you must use the same license for the resulting document.

Name the license. If you distribute this document, you will need to mention or enclose the license.

You can find a more comprehensive version of this license online.[2]

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

Contents

1 Test levels and types

Thanks for of these concepts go to Filip Defar who wrote about this in his blog[3].

1.1 Test levels

Unit tests are small and execute fast. They also usually are the fastest to write. You use them to test every detail of your application, including things that cannot be tested at the GUI level. There are a few cases where unit tests do not make much sense, e. g., queries in Extbase repositories. Please note that *unit tests* also sometimes is used for all tests that are executed with a unit-testing framework (including system tests).

Integration tests are slower to execute. You use them to test how components work together. Usually, you do not test every detail with them. In the TYPO3 world, these tests are called **functional tests**.

System tests are the biggest and the slowest to execute. You use those to test you application as a whole. They are used to test few bigger use cases.

1.2 Test types

Black-box tests test how a unit or API behaves on the outside. No knowledge about the way it works inside is assumed. Black-box tests facilitate refactoring. They are one extreme of a continuum where white-box tests are the other extreme.

White-box tests test how a unit works on the inside. They obstruct refactoring and are not recommended. They are one extreme of a continuum where black-box tests are the other extreme. Usually, tests should be black-box (or a very dark gray).

Functional tests test the behavior of a unit. Most tests are functional tests. Note: In the TYPO3 world, *functional test* actually means *integration test* (and sometimes *system test*).

Acceptance tests test how the application works on the GUI level (or in the web interface). Sometimes this term is used as a synonym for *story tests*.

Regression tests test that bugs to not reappear. In the optimal case, they are written together with the corresponding bug fix.

Smoke tests test that your application runs at all.

Story tests test use cases for user stories. Usually, they are written in the *Gherkin* language for behavior-driven design (BDD). These test sometimes also are called *acceptance tests*.

[3] `http://filipdefar.com/2015/06/tested-be-thy-name.html`

2 Test phases

Within the test methods, the code for the test phases should be separated by blank lines to make the structure easily visible.

Setup	`setUp()` and code at the beginning of the method (if needed)
Exercise	Method call
Verify	`self::assert...()`
Teardown	`tearDown()`

3 PhpStorm plug-ins

These two plug-ins for PhpStorm are particularly helpful when creating unit tests:

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

4 File and class naming

4.1 File names

Production code file name	Test file name
`Classes/Domain/Model/Shoe.php`	`Tests/Unit/Domain/Model/ShoeTest.php`
`Classes/Service/BaristaService.php`	`Tests/Unit/Service/BaristaServiceTest.php`

4.2 Class names

Production code class name	Test class name
`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 Test class structure

5.1 Test base class

5.1.1 Non-TYPO3 project

PHPUnit <= 5.7 `\PHPUnit_Framework_TestCase`

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

5.1.2 TYPO3 project

5.1.3 With `nimut/testing-framework`

Unit tests `\Nimut\TestingFramework\TestCase\UnitTestCase`

ViewHelper tests `\Nimut\TestingFramework\TestCase\ViewHelperBaseTestcase`

Functional tests `\Nimut\TestingFramework\TestCase\FunctionalTestCase`

5.1.4 With `typo3/testing-framework`

Unit tests `\TYPO3\TestingFramework\Core\Unit\UnitTestCase`

Functional tests `\TYPO3\TestingFramework\Core\Unit\FunctionalTestCase`

5.2 Non-TYPO3 PHP projects with Composer

There is an empty starter project for this on GitHub:
`https://github.com/oliverklee/tdd-seed`

5.2.1 composer.json

This setup installs PHPUnit and vfsStream:

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

```
    }
}
```

5.2.2 Test case

```
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('foo bar', $this->subject->getTitle());
    }
}
```

5.3 Extbase extensions

There is an example project (the tea example) for this on GitHub:
`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 Executing the tests

6.1 Non-TYPO3 projects

6.1.1 On the command line

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

6.1.2 Within PhpStorm

1. Settings > Languages & Frameworks > PHP > PHPUnit
2. PHPUnit library > Use Composer autoloader
3. PHPUnit library > Path to script: `vendor/autoload`
4. OK
5. right-click on the `Tests/` folder (or any test file or folder)
6. Run 'Tests'

6.2 TYPO3 core

In the TYPO3 wiki, there are how-tos on how to run the unit tests[6] and the functional tests[7] of the TYPO3 core.

6.3 TYPO3 extensions

6.3.1 Within PhpStorm

For an existing TYPO3 installation in Composer mode: This will also load all existing extensions (including the PHPUnit extension), making it possible to use the features of the PHPUnit extension.

1. Settings > Languages & Frameworks > PHP > PHPUnit
2. PHPUnit library > Use Composer autoloader
3. PHPUnit library > Path to script: `vendor/autoload` within the TYPO3 document root
4. OK
5. Run > Edit Configurations
6. Defaults > PHPUnit
7. Use alternative configuration file: `.../vendor/nimut/testing-framework/res/Configuration/UnitT`
   or `.../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. add four variables (only necessary for functional tests):
    - `typo3DatabaseUsername`
    - `typo3DatabasePassword`
    - `typo3DatabaseHost` (usually `localhost`)
    - `typo3DatabaseName`
10. OK
11. right-click on the `Tests/` folder (or any test file or folder)
12. Run 'Tests'

For an existing TYPO3 installation in classic mode (non-Composer mode): In this case, you will not be able to autoload any classes from other extensions, i. e., you will not be able to use any features of the PHPUnit extension (or of any other extension dependencies).

1. If you have downloaded the TYPO3 source via git instead of as a TAR package, you will need to do a `composer install` in the TYPO3 source directory.
2. Settings > Languages & Frameworks > PHP > PHPUnit
3. PHPUnit library > Use Composer autoloader
4. PHPUnit library > Path to script: `vendor/autoload` within the TYPO3 source
5. OK
6. Run > Edit Configurations
7. Defaults > PHPUnit
8. Use alternative configuration file: `.../vendor/nimut/testing-framework/res/Configuration/UnitT`
or `.../vendor/nimut/testing-framework/res/Configuration/FunctionalTests.xml`
9. Command Line > Environment variables
10. add four variables (only necessary for functional tests):
    - `typo3DatabaseUsername`
    - `typo3DatabasePassword`
    - `typo3DatabaseHost` (usually `localhost`)
    - `typo3DatabaseName`
11. right-click on the `Tests/` folder (or any test file or folder)
12. Run 'Tests'

Without using an existing TYPO3 installation This is the approach used in the TYPO3 extension skeleton[8] by Helmut Hummel and Nicole Cordes.

Add the following sections the `composer.json` of your extension:

```
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 }
```

You'll need to replace `tea` in line 18 with the key of your extension.

If you'd like to use other extensions that are available from the TER (e. g., the PHPUnit extension), you will need to add (or merge) these sections to your `composer.json`:

```
1  "repositories": [
2  {
3    "type": "composer",
4    "url": "https://composer.typo3.org/"
5  }
6  ],
7  "require-dev": {
8    "typo3-ter/phpunit": "^5.7.0"
9  },
```

Then do the following in PhpStorm:

1. Settings > Languages & Frameworks > PHP > PHPUnit
2. PHPUnit library > Use Composer autoloader

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

3. PHPUnit library > Path to script:
    - click on the *Show hidden files and directories* button `.Build/vendor/autoload.php` within the extension directory
4. OK
5. Run > Edit Configurations
6. Defaults > PHPUnit
7. Command Line > Environment variables
8. add four variables (only necessary for functional tests):
    - `typo3DatabaseUsername`
    - `typo3DatabasePassword`
    - `typo3DatabaseHost` (usually `localhost`)
    - `typo3DatabaseName`
9. right-click on the `Tests/` folder (or any test file or folder)
10. Run 'Tests'

7 Mocks

7.1 Why mock?

- to "disable" a method (to not write to the DB, or to not launch a cruise missile) and return null
- to have a method redurn a particular return value or throw an exception
- to test that a method gets called in a certain way

7.2 Tools for mocking

Prophecy: The recommended, state-of the art, easy-to-use mocking framework. It cannot create partial mocks, though.[9]

PHPUnit mocks: The old way of creating mocks. Creating mocks is a bit unwieldy, but it can create partial mocks.[10]

Mockery: Also very 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 Testing for Exceptions

According to a 2016 blog post[12] by Sebastian Bergmann (the author of PHPUnit), this is current best practise.

8.1 Test for the Exception class only

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

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

8.2 Test for the exception class, message and the code

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

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

9 Testing protected and private methods

9.1 Testing indirectly (recommended)

Tests should only test the behavior that has effects that can be observed from the outside. So protected methods should be tested by tests for public methods that call the protected methods.

9.2 Create a testing subclass

If you have protected methods that explicitly are part of the API to be used in subclasses, you can create a testing subclass located in `Fixtures/` that provides a public method calling the protected method.

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

10 Testing abstract classes

10.1 Using the PHPUnit mock builder

This will create an instance of the abstract class with all abstract methods mocked.

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

use OliverKlee\Coffee\Domain\Model\AbstractBeverage;

class AbstractBeverageTest
{
    /**
     * @var AbstractBeverage|\PHPUnit_Framework_MockObject_MockObject
     */
    protected $subject = null;

    protected function setUp()
    {
        $this->subject = $this->getMockForAbstractClass(
            AbstractBeverage::class
        );
    }
```

10.2 Creating a concrete subclass

This is recommended if you need to provide your subclass with some additional or specific behavior.

In `Tests/Unit/Domain/Model/Fixtures/`, create a subclass of the abstract class:

```
namespace OliverKlee\Coffee\Tests\Unit\Domain\Model\Fixtures;

class TestingBeverage extends \OliverKlee\Coffee\Domain\Model\AbstractBeverage
{
    // ...
}
```

Then you can use and instantiate the concrete subclass in your unit tests:

```
use OliverKlee\Coffee\Tests\Unit\Domain\Model\Fixtures\TestingBeverage;

class AbstractBeverageTest
{
    /**
     * @var TestingBeverage
     *
    protected $subject = null;

```

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

11 Using the testing framework of the PHPUnit TYPO3 extension

```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());
    }
```

11.1 Executable examples

The functional tests for the FileUtility class in the tea example[13] show what tests with vfsStream can look like.

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

12 Using mock file systems with vfsStream

12.1 Setting it all up

```
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 ...
}
```

12.2 Using the files

```
/**
 * @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));
}

/**
 * @test
 */
public function concatenateWithOneFileCopiesContentsFromSourceFileToTargetFile()
{
    // 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  }
```

13 Testing Extbase controllers

For controllers that are built in a clean way, you will mostly only need to test that data is passed correctly from the request to the repositories and from the repositories to the views. In addition, you will occasionally want to tests forwards and redirects.

13.1 Mocking and injecting repositories

```
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
    );
}
```

The `inject` method is a helper method from the test case base class.

If you are using explicit `inject...` method in the controller for better performance (instead the `@inject` annotations), you can use the inject methods:

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

13.2 Testing the data flow from the repository to the view

```
/**
  * @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()
```

References

[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.