PHPUnit, maîtriser ses tests unitaires

Fabien Salles

- Freelance spécialisé dans l'artisanat logiciel
- Contact :
  - Linkedin
  - training-phpunit@fabiensalles.com

Nous n'allons pas juste apprendre PHPUnit

La documentation officielle le fait très bien !

Par contre, elle n'explique pas :

- par où commencer
- comment bien écrire nos tests.

Première journée

- Les tests : vision d’ensemble
- Des notions à connaître
- PHPUnit
- Exercices
- Les doublures de tests

Pourquoi tester une application ?

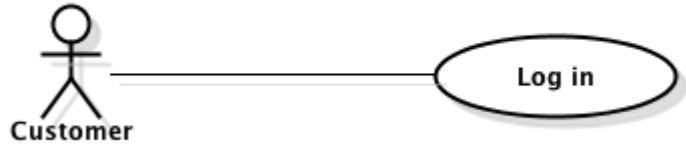
- Améliorer la couverture de code
- Répondre à la demande de son manager, du client
- Vérifier que le code répond au besoin
- Empêcher les regressions
- Refactorer plus facilement
- Feeback plus rapide
- Documentation
- Améliorer le design

Il est important de comprendre l'intérêt des tests que l'on écrit

> *Any fool can write a test that helps them today. Good programmers write tests that help the entire team in the future. - Jay Fields, Working effectively with unit tests.*

Use case (cas d'utilisation)

C'est la spécification d'une fonctionnalité, une action réalisée par un acteu
et qui mène à un résultat.



AS A	Customer
I WANT TO	Be able to log into the site
SO THAT	I can use premium features.

Test case (cas de test)

Un test case est l'instanciation d'un use case dans un contexte défini.

```
GIVEN
    I browse the login page
WHEN
    I fill the « User » field with my username
AND
    I fill the « Password » field with my password
AND
    I click on the « Submit » button
THEN
    I access a welcome page where my name is mentionned.
```

Il n'y a pas de test case sans use case ! Un cas de test découle toujours d'u cas d'utilisation.

Vous devez être en accord avec la façon dont vous écrivez les tests

Classification des tests

- Test manuel et automatisé
- Test statique et dynamique
- Niveau de test
- Niveau d'accessibiltié
- Type de test

Test manuel

- Rapide et peu coûteux en début de projet
- Devient de plus en plus long, coûteux répétitif et ennuyant par la suite
- Nécessite une intervention humaine
- Couvre difficilement les cas de test sur des systèmes, navigateurs, langue différentes.

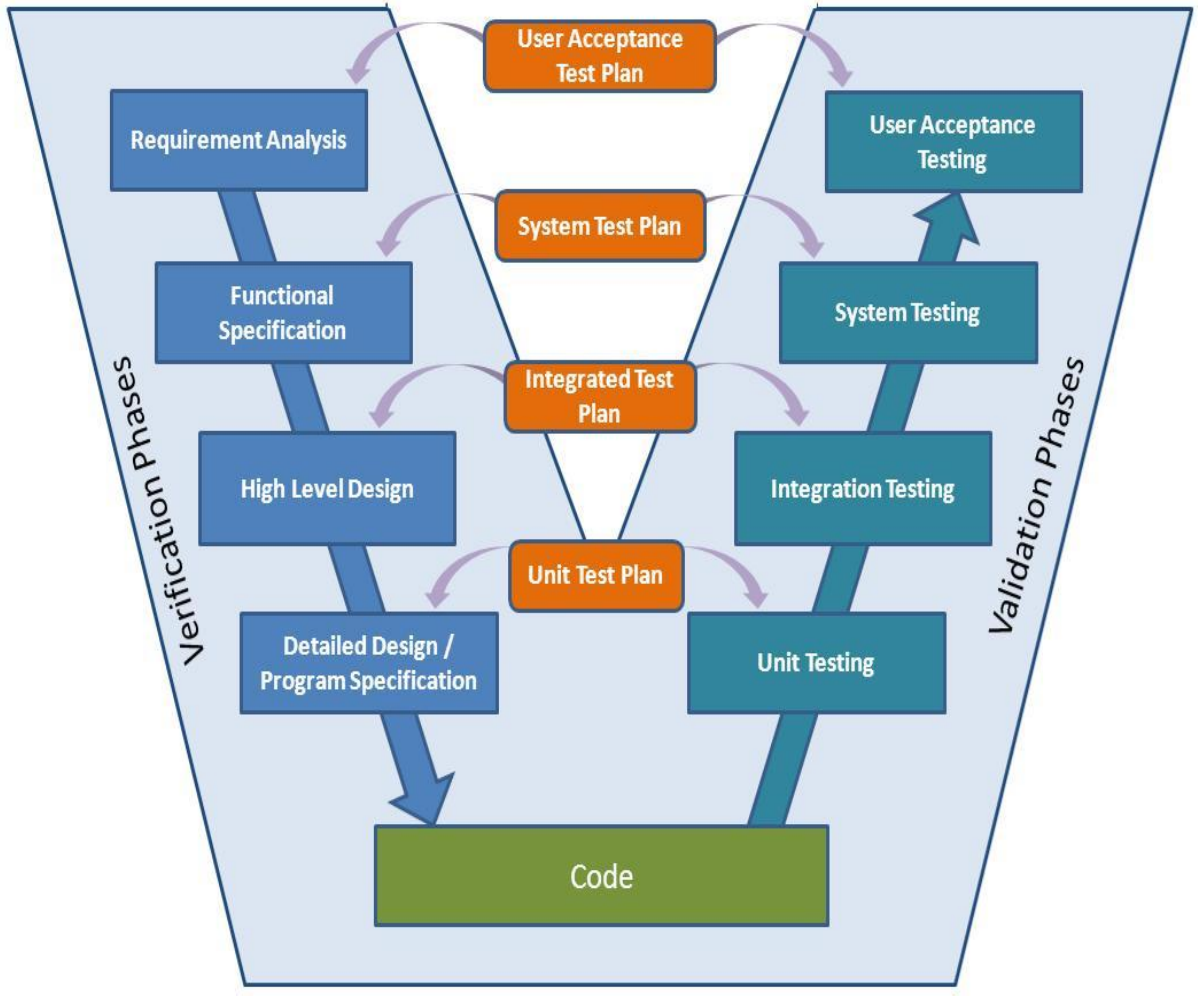
Test automatisé

- Économise du temps, de l'argent de la main d'oeuvre et plus précis que l
  tests manuels
- Une fois créer permet d'avoir un retour rapide sur tous les tests à effectu

Test statique / dynamique

Statique	**Dynamique**
Test sans exécution	Test avec exécution
Test dans un process de vérification	Test dans un process de validation
Le coût est généralement léger	Coût plus important

Niveau de test



Niveau d'accessibilité : Box testing

- White Box : test la structure interne, l'architecture
- Black Box : test uniquement ce qui est visible par l'utilisateur
- Grey Box ou Gray Box : test l'application avec une vue interne partielle

Test unitaire (1/3)

Un test est dit unitaire à partir du moment où il est réalisé en isolation complète par rapport aux autres composants.

Correspond en principe au White box testing

Test unitaire (2/3)

Avantages :

- Les tests s’exécutent très rapidement
- Une erreur sur un composant n'affecte généralement pas les composant qui l'entoure
- Ils permettent de localiser très facilement les problèmes.

Test unitaire (3/3)

Inconvénients :

- Les tests sont dissociés des cas d'utilisation réels et ne couvrent généralement pas tous les cas de figures
- Ils ne vérifient pas que les dépendances externes fonctionnent
- L'isolation peut rendre les tests un peu complexe a écrire

Test d'intégration (1/3)

Un test d'intégration est un test qui n'est pas isolé et qui test un ensemble d
composant.

Test d'intégration (2/3)

Avantages :

- Peut mieux couvrir une fonctionnalité (dans sa globalité)
- Peut tester plus de paramètres :
  - la liaison avec les autres composants,
  - les dépendances externes (base de données, web services ...)

Test d'intégration (3/3)

Inconvénients :

- Le temps d'exécution est plus long
- Il peut être difficile de localiser l'anomalie d'un test en échec
- On doit préparer l'environnement pour le test
- Il peut être plus fragile

Test système et d'acceptation

Test système : test à plus haut niveau cherchant à tester un système dans son ensemble

Test d'acceptation : vérifie que le logiciel est conforme aux spécifications

Classement par type de test

- Test de performance
- Test de charge
- Test fonctionnel
- Test de regression

Test fonctionnel

Test permettant de vérifier les fonctionnalités de l'application.

Cela peut être via des tests d'intégration fonctionnelle, des tests système, d’acceptation...

Test de régression

Un **test de régression** est le fait de lancer les tests suite à des changements
afin de vérifier qu'une régression n'a pas été introduite.

Et bien d'autres encore

- End to end
- Sanity
- Smoke
- Monkey
- Mutation
- ...

Beaucoup de confusions et d'abus

- Documentation de la version 5.0 de laravel : Laravel is built with unit testing in mind
- PHPUnit n'indique pas qu'un test non isolé ou utilisant une base de données n'est plus unitaire
- Un test fonctionnel peut être aussi un smoke test, un test end to end, d'acceptation, système, d'intégration, en black boxe ou white boxe...

Ce qu'il faut retenir

Pour qu'un test soit dit **unitaire**, il faut qu'il n'effectue aucun appel I/O :

- aucune requête à une base de donnée
- aucun appel HTTP
- aucune dépendance avec le système (fichier, horloge...)

Composer

Gestionnaires de dépendances

Similaire à **Bundler** (Ruby), **npm** (Node), **Maven** (Java) etc.

Permet que tous les intervenants d'un projet travaillent avec les **mêmes versions** des dépendances.

Installation et mise à jour facile de nouvelles dépendances.

composer.json

```
{
    "name": "symfony/framework-standard-edition",
    "require": {
        "php": ">=5.5.9",
        "symfony/symfony": "3.1.*@dev",
        "doctrine/orm": "^2.5",
        "doctrine/doctrine-bundle": "^1.6",
        "symfony/swiftmailer-bundle": "^2.3",
        "symfony/monolog-bundle": "^2.8",
        ...
    },
    "autoload": {
        "psr-4": {
            "": "src/"
        },
        "classmap": [ "app/AppKernel.php", "app/AppCache.php" ]
    },
    ...
}
```

Syntaxe des versions

https://getcomposer.org/doc/articles/versions.md

Commandes utiles

```
composer install --no-dev
```

```
composer require doctrine/orm
```

```
composer update
```

composer.lock

Fichier de "lock" des dépendances.

Générer par `install`, il contient la version exacte des dépendances installées.

Si ce fichier existe, `install` se base dessus. Sinon se comporte comme un `update` : recalcul complet de l'arbre des dépendances.

Mis à jour par `update`.

Initialiser php

sudo apt-get install php-cli php-dom php-mbstring php-bcmath zip unzip php-zip

Installer composer

```
curl -sS https://getcomposer.org/installer | php
```

```
sudo mv composer.phar /usr/local/bin/composer
```

Initialiser le projet

composer init

```
Package name (<vendor>/<name>) [fsalles/phpunit-exercises]: conveycode/phpunit-exercices
Description []: phpunit training exercices
Author [Fabien Salles <github@fabiensalles.com>, n to skip]:
Minimum Stability []: stable
Package Type (e.g. library, project, metapackage, composer-plugin) []: project
License []:

Define your dependencies.

Would you like to define your dependencies (require) interactively [yes]? no
Would you like to define your dev dependencies (require-dev) interactively [yes]? no
Add PSR-4 autoload mapping? Maps namespace "Conveycode\PhpunitExercices" to the entered
relative path. [src/, n to skip]
```

PHPUnit

- Créé par Sebastian Bergmann
- Fait partie de la suite *xUnit*
- Framework de test référent en PHP
- Documentation

Installation de PHPUnit

- PHP Archive
- Composer

```
composer require --dev phpunit/phpunit
```

Lancer les tests

```
./vendor/bin/phpunit src
PHPUnit 10.5.38 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.1.2-1ubuntu2.19

No tests executed!
```

Exemple de test

```php
class StackTest extends TestCase
{
    public function testPushAndPop()
    {
        $stack = [];
        self::assertEquals(0, count($stack));

        array_push($stack, 'foo');
        self::assertEquals('foo', $stack[count($stack)-1]);
        self::assertEquals(1, count($stack));

        self::assertEquals('foo', array_pop($stack));
        self::assertEquals(0, count($stack));
    }
}
```

Les assertions (1/3)

```php
// Check 1 === 1 is true
self::assertTrue(1 === 1);

// Check 1 === 2 is false
self::assertFalse(1 === 2);

// Check 'Hello' equals 'Hello'
self::assertEquals('Hello', 'Hello');

// Check array has key 'lang'
self::assertArrayHasKey('lang', ['lang' => 'php', 'size' => '1024']);

// Check array contains value 'php'
self::assertContains('php', ['php', 'ruby', 'c++', 'JavaScript']);
```

Documentation : https://docs.phpunit.de/en/11.4/assertions.html

Les assertions (2/3) : assertEquals

assertEquals vérifie que 2 variables sont égales

Exemple :

```php
public function testEqualSuccess()
{
    self::assertEquals(1, true);
    self::assertEquals(2, true);
    self::assertEquals(new stdClass(), new stdClass());
    self::assertEquals(0, false);
    self::assertEquals(0, null);
}
```

Les assertions (3/3) : assertSame

assertSame vérifie que le type et la valeur de 2 variables sont identiques

```php
public function testSameFailure()
{
    self::assertSame(1, true);
    self::assertSame(2, true);
    self::assertSame(new stdClass(), new stdClass());
    self::assertSame(0, false);
    self::assertSame(0, null);
}
```

Fixtures

Une fixture d'écrit l'état initial requis lors de l'exécution d'un test

Documentation : https://docs.phpunit.de/en/11.4/fixtures.html

setUp() et tearDown()

```php
class StackTest extends TestCase
{
    protected $stack;

    protected function setUp()
    {
        self::stack = [];
    }

    public function testPush()
    {
        array_push(self::stack, 'foo');
        self::assertEquals('foo', self::stack[count(self::stack)-1]);
        self::assertFalse(empty(self::stack));
    }

    protected function tearDown()
    {
        // useless here
        unset(self::stack);
    }
```

setUpBeforeClass() et tearDownAfterClass()

```php
class DatabaseTest extends TestCase
{
    protected static $dbh;

    public static function setUpBeforeClass()
    {
        self::$dbh = new PDO('sqlite::memory:');
    }

    public static function tearDownAfterClass()
    {
        self::$dbh = null;
    }
}
```

Les annotations

Les annotations sont des méta-données apportant des informations complémentaires et permettant de documenter son code.

Documentation : https://docs.phpunit.de/en/11.4/annotations.html

Les fixtures

```php
class MyTest extends TestCase
{
    /**
     * @before
     */
    public function setupSomeFixtures()
    {
        // ...
    }
}
```

Il existe aussi **@beforeClass**, **@after**, **@afterClass**

Les dataProvider

```php
class DataTest extends TestCase
{
    /**
     * @dataProvider additionProvider
     */
    public function testAdd($a, $b, $expected)
    {
        self::assertEquals($expected, $a + $b);
    }

    public function additionProvider()
    {
        return [
            [0, 0, 0],
            [0, 1, 1],
            [1, 1, 2],
        ];
    }
}
```

testWith

```
class MyTest extends TestCase
{
    /**
     * @testWith    [0, 0, 0]
     *              [0, 1, 1]
     *              [1, 1, 2]
     */
   public function testAdd($a, $b, $expected)
   {
       self::assertEquals($expected, $a + $b);
   }
}
```

Les groupes

```php
/**
 * @group unit
 */
public function testSomething()
{
}

/**
 * @group integration
 */
public function testSomethingElse()
{
}
```

Les groupes sont utilisés avec les options **--group** et **--exclude-group** de la ligne de commande PHPUnit

Les attributs

Depuis PHPUnit 10 et PHP 8

Documentation : https://docs.phpunit.de/en/11.4/attributes.html

Les fixtures

```php
use PHPUnit\Framework\Attributes\Before;

class MyTest extends TestCase
{
    #[Before]
    public function setupSomeFixtures()
    {
        // ...
    }
}
```

Les dataProvider

```php
use PHPUnit\Framework\Attributes\DataProvider;

class DataTest extends TestCase
{
    #[DataProvider('additionProvider')]
    public function testAdd($a, $b, $expected)
    {
        self::assertEquals($expected, $a + $b);
    }

    public static function additionProvider()
    {
        return [
            [0, 0, 0],
            [0, 1, 1],
            [1, 1, 2],
        ];
    }
}
```

Testwith

```php
use PHPUnit\Framework\Attributes\TestWith;

class MyTest extends TestCase
{
    #[TestWith([0, 0, 0])]
    #[TestWith([0, 1, 1])]
    #[TestWith([1, 1, 2])]
    public function testAdd($a, $b, $expected)
    {
        self::assertEquals($expected, $a + $b);
    }
}
```

Si vous ne voulez pas préfixer vos méthodes

```php
use PHPUnit\Framework\Attributes\Test;

class MyTest extends TestCase
{
    #[Test]
    public function itDoSomething()
    {
    }

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

La ligne de commande PHPUnit

```
phpunit [options] <directory>
```

Options :

- --converage-* : Génération de la couverture de code
- --testsuite : filtrer par une/des suite(s) de tests
- --group : filtrer par un/des groupe(s)
- --bootstrap : charger un fichier avant de lancer les tests
- -c|--configuration : selectionner un fichier de configuration
- -h|--help : pour l'aide

Fichier de configuration

```xml
<?xml version="1.0" encoding="UTF-8"?>
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="vendor/phpunit/phpunit/phpunit.xsd"
         bootstrap="vendor/autoload.php"
         backupGlobals="false"
         backupStaticAttributes="false"
         colors="true"
         verbose="true"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         convertDeprecationsToExceptions="true"
         stopOnFailure="false">
    <php>
        <ini name="display_errors" value="1" />
        <ini name="error_reporting" value="-1" />
        <server name="APP_ENV" value="test" force="true" />
        <server name="SHELL_VERBOSITY" value="-1" />
    </php>
    ...
</phpunit>
```

Fichier de configuration suite

```
<phpunit>
    ...
    <testsuites>
        <testsuite name="all">
            <directory>tests</directory>
        </testsuite>
        <testsuite name="unit">
            <directory>tests/unit</directory>
        </testsuite>
        <testsuite name="integration">
            <directory>tests/integration</directory>
        </testsuite>
    </testsuites>
    <extensions>
        <bootstrap class="DAMA\DoctrineTestBundle\PHPUnit\PHPUnitExtension" />
        <bootstrap class="Zalas\PHPUnit\Globals\AttributeExtension" />
    </extensions>
</phpunit>
```

Hiérarchie de la configuration

1. Le fichier indiqué en option `-c configuration-file.xml`
2. Le fichier phpunit.xml
3. Le fichier phpunit.xml.dist

Configurer composer

```
{
    "name": "conveycode/phpunit-exercices",
    "description": "phpunit training exercices",
    "type": "project",
    "autoload": {
        "psr-4": {
            "Conveycode\\PhpunitExercices\\": "src/"
        }
    },
    "autoload-dev": {
        "psr-4": {
            "Conveycode\\PhpunitExercices\\Tests\\": "tests/"
        }
    },
    "minimum-stability": "stable",
    "require-dev": {
        "phpunit/phpunit": "^10.5"
    }
}
```

Créer une configuration phpunit

```xml
<?xml version="1.0" encoding="UTF-8"?>
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="vendor/phpunit/phpunit/phpunit.xsd"
         bootstrap="vendor/autoload.php">
    <testsuites>
        <testsuite name="all">
            <directory>tests</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

Exercice1 : Math

1. Créer une classe `Math` disposant d'une propriété `number` et vérifier que l valeur est initialisé à 0 en type float.
2. Créer un test simple permettant d'ajouté une valeur à `number` (méthode `sum`). Vérifier que le test échoue, créer la méthode et vérifier que le test passe.
3. Faire la même chose pour soustraire un nombre `substract`, diviser `divide` et multiplier `multiply`
4. Utiliser l'attribut `DataProvider` pour enrichir les tests

Exercice2 : ProductCart

1. Créer une classe `Product` et la classe de test associée possédant un `nam`
   et un `price` toujours de type `float`
2. Créer une classe `Cart` et la classe de test associée qui contiendra une lis
   de produit et une méthode retournant le prix total
   `getProductCartPrice`

Attention : utiliser la classe `Math` pour tous les calculs à effectuer.

Exercice2 : 2ème partie

Ajout de frais de ports :

- Lorsque que le prix total est inférieur à 100 les frais de port sont de 15.5
- lorsque le prix est égale à 100, ceux-ci sont de 15
- lorsqu'il est supérieur, ils passent à 10

---

1. Modifier les jeu de test de la classe `Cart` afin de prendre en compte ces paramètres
2. Vérifier que les tests sont en échec et effectuer la modification au sein de la méthode `getProductCartPrices`

Un bug est remonté !

Un panier comportant 3 produits avec comme prix respectif 80.1, 10.1 et 9.8
(égale à 100) devrait avoir un prix total à 115 mais celui-ci est à 110.

Que devons nous faire ?

Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead. -- Martin Fowler

Floating Precision

```php
public function testProductCartPriceWithFloatingPrecision()
{
    $cart = new Cart([
        new Product('un produit', 80.1),
        new Product('un 2ème produit',10.1),
        new Product('un 3ème', 9.8),
    ]);
    $this->assertSame((float) 115, $cart->getProductCartPrices());
}
```

```
./vendor/bin/phpunit --filter testProductCartPriceWithFloatingPrecision
PHPUnit 10.5.38 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.1.2-1ubuntu2.19
Configuration: /mnt/e/project/github/phpunit-exercises/phpunit.xml

F                                                                   1 / 1 (100%)

Time: 00:00.252, Memory: 8.00 MB

There was 1 failure:

1) Conveycode\PhpunitExercices\Tests\Exercice2\CartTest::testProductCartPriceWithFloatingPrecis
Failed asserting that 115.49999999999999 is identical to 115.0.

/mnt/e/project/github/phpunit-exercises/tests/Exercice2/CartTest.php:29

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Exercice2 : 3ème partie

3 solutions s'offre à nous :

1. Effectuer des arrondis avec la fonction round
2. Manipuler des entiers et effectuer une division à la fin lorsqu'on retourne le prix total
3. Utiliser les fonctions mathématiques de précisions

Modifions la classe `Math` en utilisant la 3ème solution et vérifions que les tests passent.

Attention Les méthodes de BC Math gèrent des `string` et non des `float` !

Avez vous rencontrez des problèmes ?

Ce qu'on aurait du faire

- Lancer uniquement le test en échec
- Déscendre d'un niveau et tester uniquement la classe `Math`
- Faire les modifications pas à pas
- Avoir des cas de tests plus exhaustifs
- Limiter le couplage entre la classe `Cart` et `Math`

SUT : System Under Test

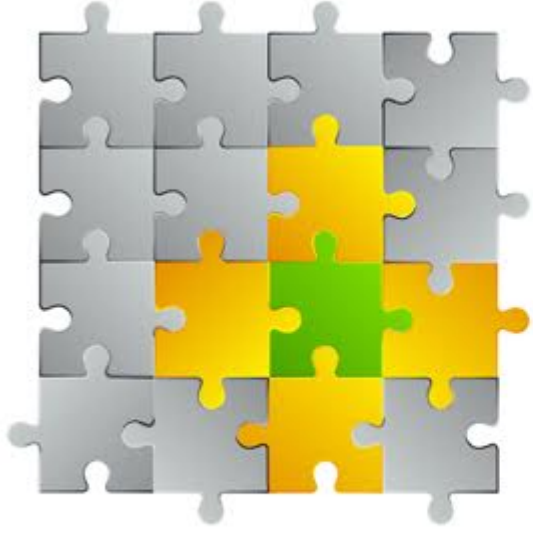
Fait référence au système qui est entrain d'être tester.

> *Whatever thing we are testing. The SUT is always defined from the perspective of the test. - xUnit Test Patterns: Refactoring Test Code, by Gerard Meszaros*

On peut aussi parler de : **Application Under Test (AUT)**, **Method Under Test (MUT)** ou encore **Class Under Test (CUT)**.

Comment isoler nos test ?

REAL SYSTEM



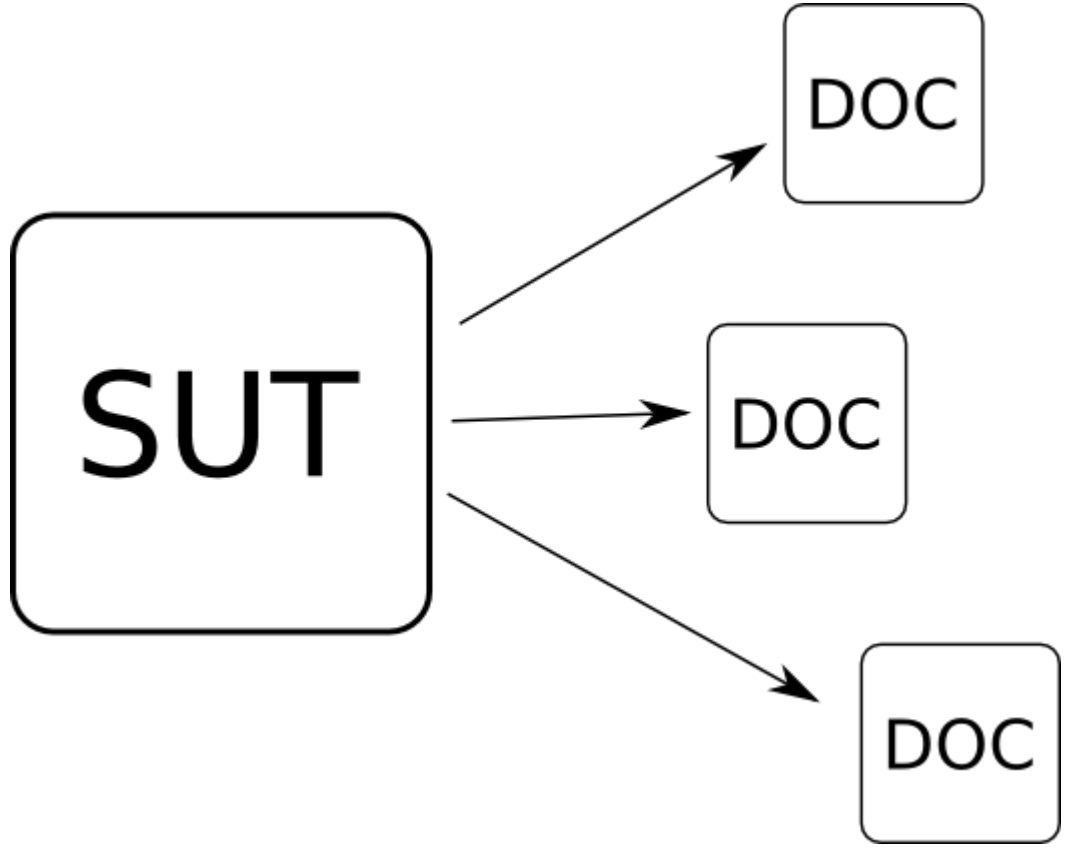
Green = class in focus
Yellow = dependencies
Grey = other unrelated classes

CLASS IN UNIT TEST

Green = class in focus
Yellow = mocks for the unit test

DOC : Depended-on component

Représente tout les éléments requis par le SUT pour remplir son rôle.



Aussi appelé **Collaborateur** ou **Dépendance**.

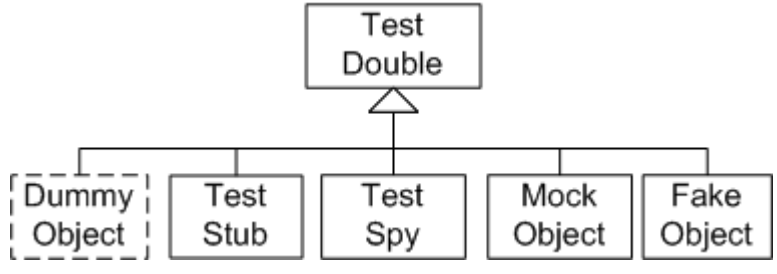
Test double

Terme défini par Gerard Meszaros dérivé de Stunt double représentant un ensemble de méthodes pour remplacer les dépendances d'un SUT

Les intérêts sont multiples :

- la dépendance ne retourne pas le résultat attendu
- elle ne peut pas être utilisée en environnement de test (ou difficilement)
- son utilisation peut engendrer des effets de bords (bugs, lenteurs...)
- elle ne nous appartient pas

Type de doublure de test



Dummy Object

Dépendance qui ne sert qu'à instancier notre SUT et que l'on ne souhaite p
tester.

Exemple de Dummy Object en PHP

```php
class LoggerDummy implements Logger
{
    public function debug()
    {
        return null;
    }

    ...
}
```

Stub Object

Dépendance qui dispose d'une réponse pré-configurée et fixe.

Un stub n'accorde pas d'importance :

- aux arguments qui lui sont fournis lorsqu'il est appelé
- à notre manière de l'appeler
- et à son implémentation

On ne fait qu'utiliser son retour sans se préoccuper de son état et comportement.

Exemple Stub Object

```
class InfoLevelLoggerStub implements Logger
{
    ...

    public function getLevel()
    {
        return self::INFO;
    }
}
```

Fake Object

Un fake est un objet fournissant une implémentation alternative souvent simplifiée de la méthode utilisée pour le test.

```php
class PredefinedUrlsUrlGeneratorFake implements UrlGenerator
{
    private $urlsByRoute;

    public function __construct(array $urlsByRoute)
    {
        $this->urlsByRoute = $urlsByRoute;
    }

    public function generate($routeName)
    {
        if (isset($this->urlsByRoute[$routeName])) {
            return $this->urlsByRoute[$routeName];
        }

        throw new UnknownRouteException();
    }
}
```

Spy Object

C'est un stub qui enregistre certaines informations afin que l'on puisse vérifier son état. Ces informations peuvent être par ex :

- Est-ce que la/les méthode(s) a/ont étée(s) appelée(s) ?
- Combien de fois ?
- Dans quel ordre ?
- Les paramètres étaient-ils corrects ?

Exemple de Spy Object

```php
class AcceptingAuthorizerSpy
{
  public $authorizeWasCalled = false;

  public function authorize(string $username, string $password)
  {
    $authorizeWasCalled = true;
    return true;
  }
}
```

Mock Object

Un mock est un objet préprogrammé avec des attentes sur son comportement qui simule le comportement d'un objet réel.

Spy vs Mock

- Un Spy peut "espionner" un objet réel, un mock le remplace
- Un Spy vérifie un état après que le code ait été exécuté
- Un Mock définie un comportement avant que le code soit exécuté

Test double avec PHPUnit

Documentation : https://docs.phpunit.de/en/11.4/test-doubles.html

Exemple Dummy avec PHPunit

```php
// Logger is an interface or a class
$logger = $this->createMock(Logger::class);
```

Exemple de Stub avec PHPUnit

```php
$logger = $this->createStub(Logger::class);
$logger
    ->method('getLevel')
    ->willReturn(Logger::INFO);
```

Exemple de Fake avec PHPUnit

```php
$urlsByRoute = array(
    'index' => '/',
    'about_me' => '/about-me'
);

$urlGenerator = $this->createMock(UrlGenerator::class);
$urlGenerator
    ->method('generate')
    ->willReturnCallback(
        function ($routeName) use ($urlsByRoute) {
            if (isset($urlsByRoute[$routeName])) {
                return $urlsByRoute[$routeName];
            }

            throw new UnknownRouteException();
        }
    ));
```

Exemple de Mock avec PHPUnit

```php
$logger = $this->createMock(Logger::class);
$logger
    ->expects($this->once())
    ->method('getLevel')
    ->willReturn(Logger::INFO);
```

Exemple de Spy avec PHPUnit

```php
$citizen = $this->createMock(AverageCitizen::class);
$citizen->expects($spy = $this->any())
    ->method('spyOn');

$citizen->spyOn("foo");

$invocations = $spy->getInvocations();

$this->assertEquals(1, count($invocations));

// we can easily check specific arguments too
$last = end($invocations);
$this->assertEquals("foo", $last->parameters[0]);
```

Exercice3

Remplacer la dépendance `Math` de la classe `Cart` par un Stub.

Pour aller plus loin

- willThrowException(new Exception) : lever une exception
- willReturnArgument(0) : retourner le premier argument
- willReturnSelf() : retourner l'objet lui-même
- willReturnMap() : retourner une valeur en fonction d'un ensemble de paramètres

Stubber plusieurs méthodes à la fois

```
$stub = $this->createConfiguredStub(
    SomeInterface::class,
    [
    'doSomething'     => 'foo',
    'doSomethingElse' => 'bar',
    ]
);
```

Et plus encore

- getMockForAbstractClass : mocker une classe abstraite
- getMockForTrait : mocker un trait
- createStubForIntersectionOfInterfaces : stubber une intersection d'interfaces
- createMockForIntersectionOfInterfaces : mocker une intersection d'interfaces
- getMockBuilder(SomeClass::class) : pour aller plus loin dans la configuration du mock

Des questions ?