PHPUnit, fiabiliser vos développements avec les tests unitaires

Première journée

Par Fabien Salles | Clever Age

Fabien Salles

Consultant Web - Clever Age

Formateur (Symfony/PHPUnit) - Clever Institut

- fsalles@clever-age.com
- GitHub
- Twitter

Cette formation n'est pas une simple formation à PHPUnit

La documention du frameworks le fait très bien !

Le but de cette formation est de comprendre :

- ce qu'est un test unitaire et ce que cela remet en cause
- comment écrire un test unitaire
- comment tester une application afin de répondre à différents besoins
- des bonnes pratiques et des courants de pensée

Première journée

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

Pourquoi tester une application ?

1. Améliorer la couverture de code
2. Répondre à la demande de son manager, du client
3. Vérifier que le code répond aux spécifications
4. Empécher les regressions
5. Refactorer plus facilement
6. Feeback plus rapide
7. Documentation
8. 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 acteur 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'un cas d'utilisation.

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

Classification des tests

- Test manuel et test automatisé
- Static / Dynamic testing
- Par niveau d'accessibiltié
- Par niveau de test
- Par type de test

Test manuel

Exécution de cas de tests manuellement

- 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, langues différentes.

Test automatisé

Utilisation d'un outil d'automatisation pour exécuter les cas de tests

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

Static / Dynamic testing

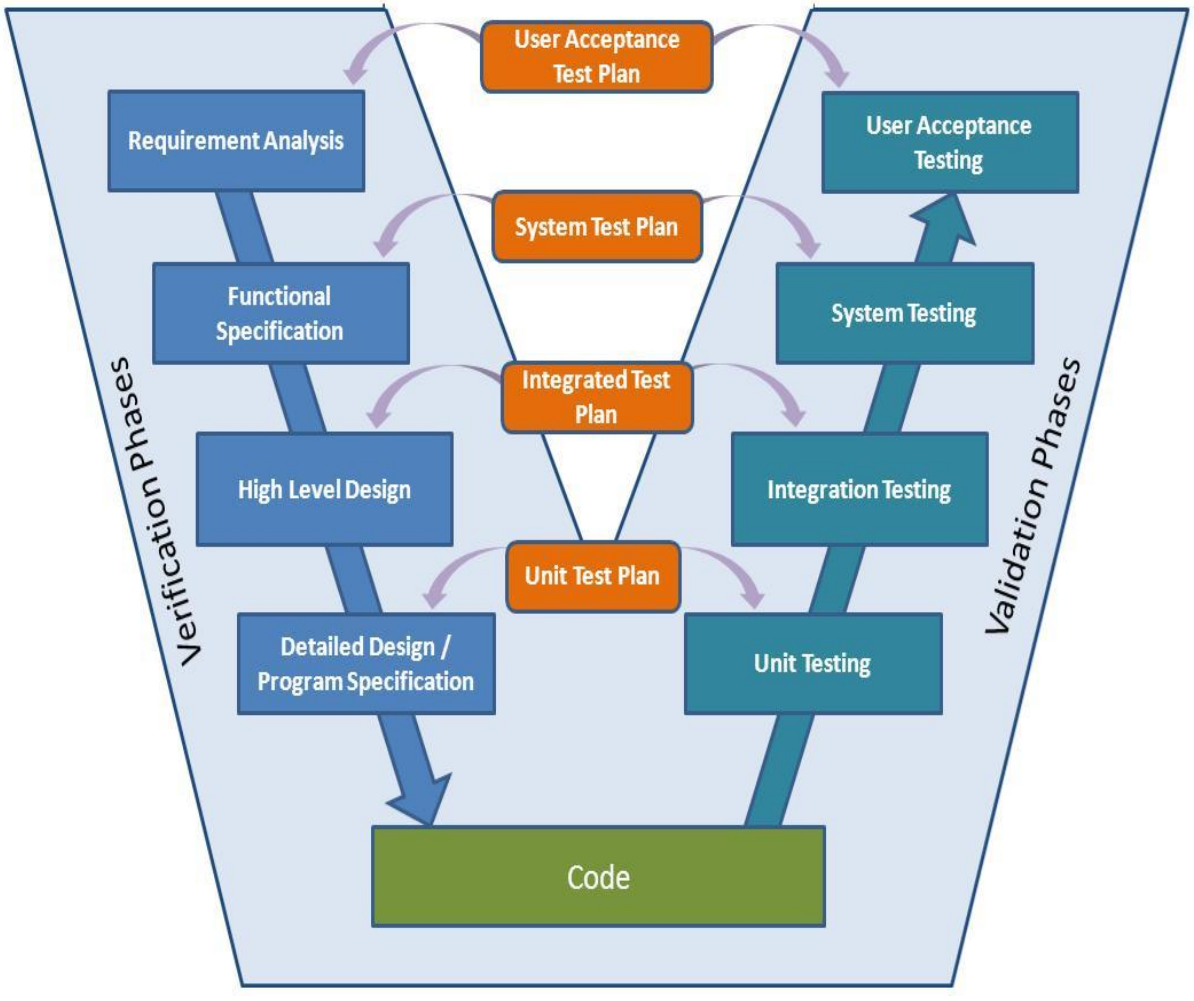
- Static testing : revue de code, de specs techniques, fonctionnels
- Dynamic testing : test unitaire, d'intégration, fonctionnel, de performance ect...

Static	**Dynamic**
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

Classement par 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

Classement par niveau de test



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 composants 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 de composant.

Correspond au Grey Box testing

Test d'intégration (2/3)

Avantages :

- Plus fiable que les tests unitaire étant donné qu'on effectue aucunes isolations
- Couvre plus de cas de figures (test 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 un test en échec
- L'écriture peut également être complexe lorsque l'on doit préparé l'environnement pour le test

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

Les 2 sont en black box testing effectués sans connaissance technique en utilisant l'application

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

Régression : on parle de régression lorsque l'on trouve un bug sur quelque chose qui fonctionnait correctement avant. Une régression apparaît suite à une modification.

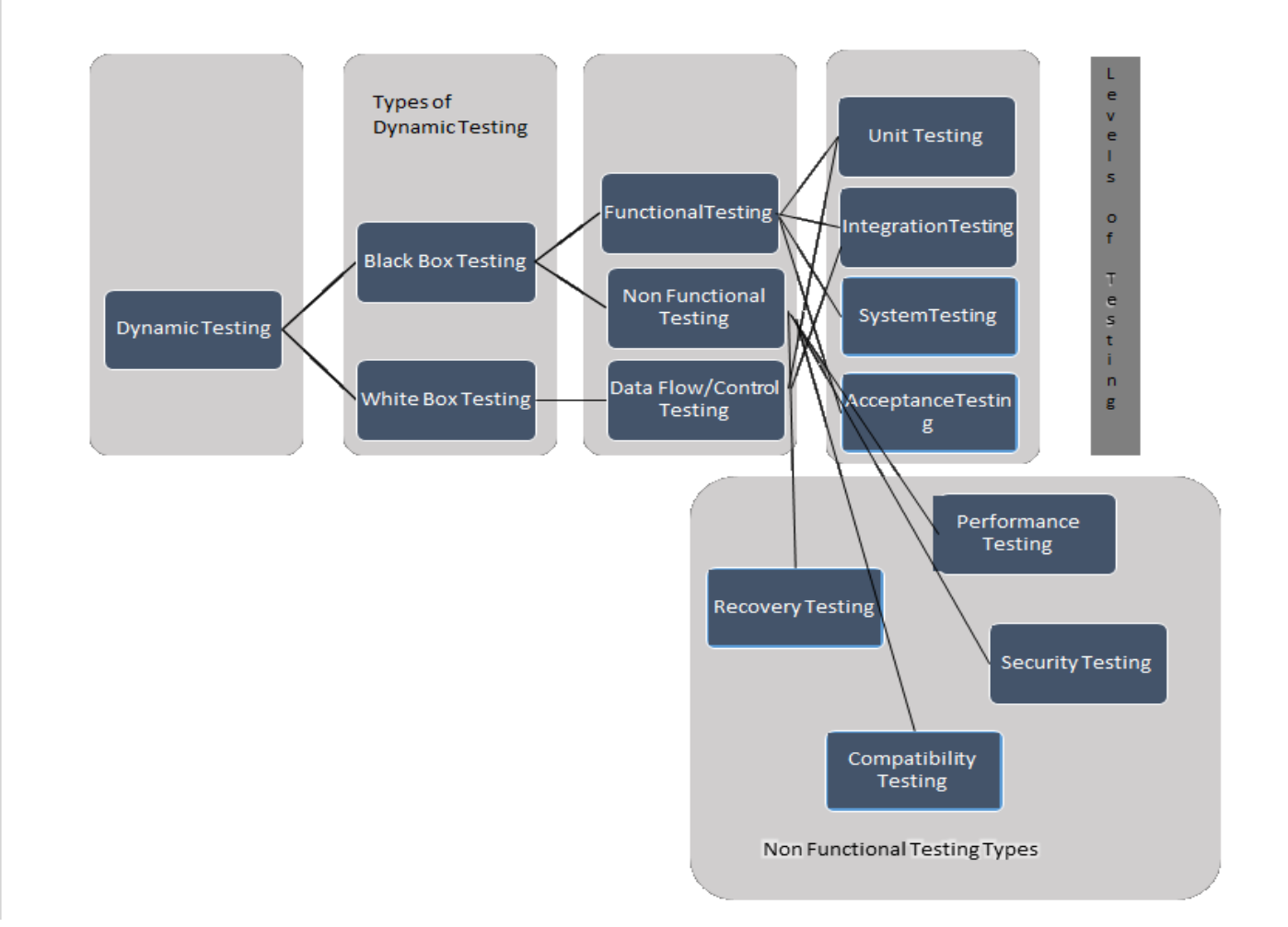
Un **test de régression** est donc le fait de lancer les tests suite à des changements afin de vérifier qu'une régression n'a pas été introduite.

Et encore bien d'autres

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 dans sa documentation qu'un test non isolé ou utilisant une base de données n'est plus unitaire
- Certaines personnes indiquent qu'un test fonctionnel est effectué en black box par la MOA alors que d'autres expliquent comment réaliser des testes fonctionnels en White/Grey Box par la MOE

Vue d'ensemble



Ce qu'il faut retenir

Pour qu'un test soit dit **unitaire**, il faut qu'il se trouve sur les éléments les plus atomiques et être isolé de toutes dépendances.

Rappels autour de PHP

- Namespace
- Autoloading : PSR 0 et 4
- Composer

Namespaces

```php
<?php

namespace CleverAge\Training;

use General\Talk\Training;

class Symfony2 extends Training
{
    public function runPresentation()
    {
        // talk to you guys...
    }
}
```

Evite les **collisions** !

```php
use My\Name\Space\Classname as Niceclass;
use Another\Name\Space\Classname;

$obj1 = new Niceclass();  // == My\Name\Space\Classname
$obj2 = new Classname();  // == Another\Name\Space\Classname
$obj3 = new \Classname(); // == Classname (root namespace)
```

Autoloading (1/2)

```
class Training {
    // ...
}
```

Dans un autre fichier :

```
$symfony = new Training();
```

Boom ! On obtient l'erreur classique :

```
Fatal error: Class 'Training' not found in /path/to/file.php
```

Afin d'éviter ça (et d'éviter les `require_once` partout dans le code), PHP 5.3 à introduit le mécanisme d'**autoload**.

Autoloading (2/2)

PSR-0

```
\Doctrine\ORM\EntityManager
// => /path/to/project/lib/vendor/Doctrine/ORM/EntityManager.php
```

Déprecié en octobre 2014 en faveur de PSR-4

PSR-4

Pareil que PSR-0 mais en mieux :

- permet une structure de fichier plus claire;
- supprime le support du standard d'autoloading PEAR.

Les autres standards PSR

> *PHP Standards Recommendations*

Composer

Gestionnaires de dépendances

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

Permet que tout les intervenants d'un projet travaille avec les **mêmes versions** des dépendances.

Installation facile de nouvelles dépendances.

Mise à jour des dépendances de façon simple.

Génère un fichier d'autoloading automatiquement !

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 --prefer-dist --optimize-autoloader
```

```
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`.

Doit être versionné !

Package Development Standards

- PHP PDS Skeleton
- PHP League Skeleton

Initialiser notre projet

> *https://github.com/FabienSalles/training-phpunit*

PHPUnit

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

Installation de PHPUnit

- PHP Archive
- Composer

> *Installation*

Exemple de test

```
class StackTest extends TestCase
{
    public function testPushAndPop()
    {
        $stack = [];
        $this->assertEquals(0, count($stack));

        array_push($stack, 'foo');
        $this->assertEquals('foo', $stack[count($stack)-1]);
        $this->assertEquals(1, count($stack));

        $this->assertEquals('foo', array_pop($stack));
        $this->assertEquals(0, count($stack));
    }
}
```

Les assertions (1/3)

```php
// Check 1 === 1 is true
$this->assertTrue(1 === 1);

// Check 1 === 2 is false
$this->assertFalse(1 === 2);

// Check 'Hello' equals 'Hello'
$this->assertEquals('Hello', 'Hello');

// Check array has key 'lang'
$this->assertArrayHasKey('lang', ['lang' => 'php', 'size' => '1024']);

// Check array contains value 'php'
$this->assertContains('php', ['php', 'ruby', 'c++', 'JavaScript']);
```

> *Documentation*

Les assertions (2/3) : assertEquals

assertEquals vérifie que 2 variables sont égales

Exemple :

```
public function testEqualSuccess()
{
    $this->assertEquals(1, true);
    $this->assertEquals(2, true);
    $this->assertEquals(new stdClass(), new stdClass());
    $this->assertEquals(0, false);
    $this->assertEquals(0, null);
}
```

Les assertions (3/3) : assertSame

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

```
public function testSameFailure()
{
    $this->assertSame(1, true);
    $this->assertSame(2, true);
    $this->assertSame(new stdClass(), new stdClass());
    $this->assertSame(0, false);
    $this->assertSame(0, null);
}
```

Fixtures

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

> *Documentation des fixtures PHPUnit*

setUp() et tearDown()

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

    protected function setUp()
    {
        $this->stack = [];
    }

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

    protected function tearDown()
    {
        // useless here
```

setUpBeforeClass() et tearDownAfterClass()

```
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 (1/5)

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

On peut aller plus loin en interprétant les annotations à l'éxécution du code :

- **Doctrine** : pour le mapping entre la BDD et les classes PHP;
- **Symfony 2** : pour le routing ou encore la sécurité ;
- **PhpUnit** : https://phpunit.de/manual/current/en/appendixes.annotations.html

Les annotations (2/5) : before

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

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

Les annotations (3/5) : dataProvider

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

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

Les annotations (5/5) : les groupes

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

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

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

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 bootstrap="vendor/autoload.php"
         backupGlobals="false"
         backupStaticAttributes="false"
         colors="true"
         verbose="true"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         processIsolation="false"
         stopOnFailure="false">
    <testsuites>
        <testsuite name="FabienSalles Test Suite">
            <directory>tests</directory>
        </testsuite>
    </testsuites>
    <filter>
        <whitelist>
            <directory suffix=".php">src/</directory>
        </whitelist>
```

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

Exercice Math

1. Créer une classe `Math` disposant d'une propriété `number` et vérifier que la 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'annotation `@dataProvider` pour enrichir les tests

Exercice ProductCart 1/3

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

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

Exercice ProductCart 2/3

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 vos jeux de tests afin de prendre en compte ces paramètres
2. Vérifier que les tests sont en échecs 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.

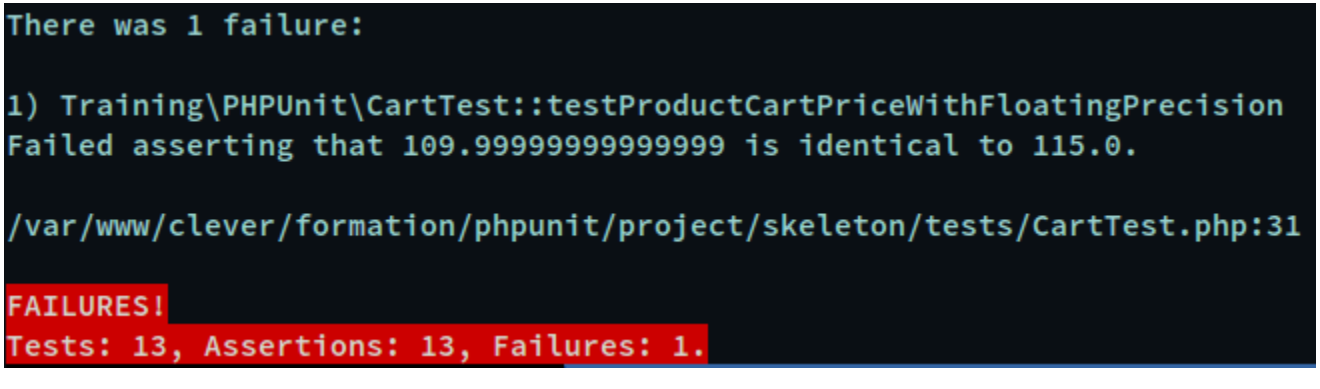
Que devons nous faire ?

> *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

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());
}
```



Exercice ProductCart 3/3

3 solutions s'offrent 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 ?

Les tests vous ont-ils indiqués clairement où se situai(en)t le(s) problème(s) ?

Comment pouvons nous améliorer nos tests et simplifier le débuggage ?

Tester en premier la classe que l'on modifie indépendament des autres

> */bin/phpunit tests/MathTest.php*

Ne pas passer par la méthode `setUp` pour initialiser l'objet Math

L'initialisation de l'objet peut faire partie du test et le débuggage peut être plus difficile à comprendre.

Dans notre cas en utilisant les fonctions **BC Math** sans préciser le nombre de décimale après la virgule a mis en avant une **anomalie**.

En initialisant notre propriété number à 0 et un jeu de test essayant d'additionner 2,5 + 4,5 on aura **0 + 2,5 + 4,5 = 6**

Si on n'aurait pas eu d'initialisation à 0 mais a 2,5 on aurait eu **2,5 + 4,5 = 7**

L'ajout des frais de ports ne devrait pas être dans la classe `Cart`

La méthode `getProductCartPrices` ne devrait faire que retourner le prix. L'ajout de tout nouveau traitement devrait faire suite à la création d'une méthode ou classe. La classe `Cart` devant gérer le panier, il faudrait créer une nouvelle classe (`Pricing` par exemple) pour gérer le prix.

Nous somme passés par la classe `Cart` pour tester une anomalie de calcule issue de la classe `Math`

La classe `Cart` devrait s'occuper de tester uniquement ses propres méthodes.

Elle devrait se préoccuper de son comportement et pas celui des classes qu'elle utilise

Quand je veux tester quelque chose pourquoi devrais-je éviter de tester/utiliser des dépendances en même temps ?

Exemple de cas de figure

- Résoudre un problème lier à l'application et être distrait par des problèmes liés au système et à l'infrastructure
- Avoir un problème sur une page web et devoir se soucier des étapes pour arriver à cette page
- Avoir une anomalie dans l'affichage de données et devoir réfléchir à comment récupérer un jeu de données issue d'une base
- Avoir un comportement particulier avec une librairie ou un framework et devoir utiliser l'application et des objets métiers pour la/le tester

SUT : System Under Test

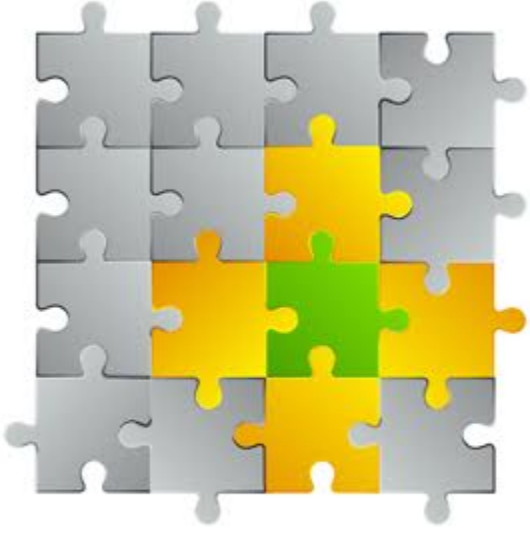
Fait référence au système qui est entrain d'être tester. Il dépend donc du test !

> *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*

En test unitaire on peut parler de **Class Under Test**

Comment isoler nos test ?

REAL SYSTEM



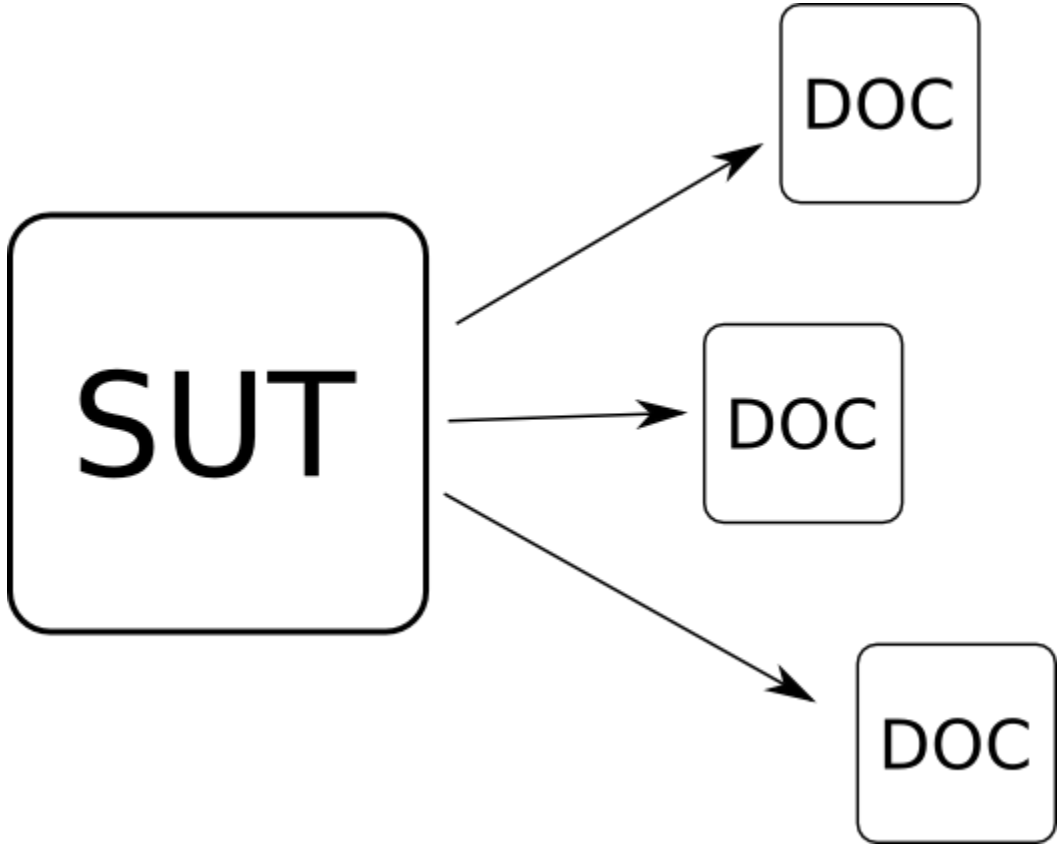
Green = class in focus
Yellow = dependencies
Grey = other unrelated classes

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. Il a en générale le même niveau de granularité que le SUT.



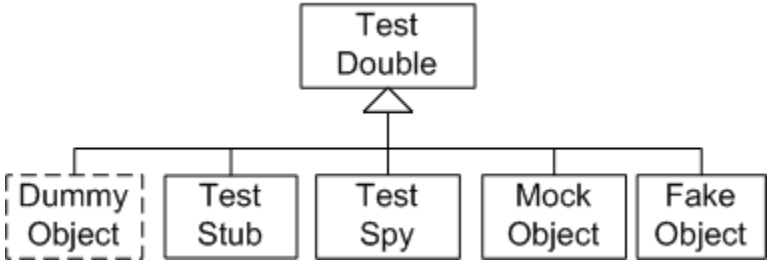
Test double

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

L’intérêt est de limiter les difficultés pour tester un SUT à cause de ses dépendances. Ces difficultés peuvent venir du fait que :

- le DOC ne peut pas être utiliser en environnement de test
- il n'est pas disponible
- il ne retourne pas/difficilement le résultat attendu
- son utilisation peut engendrer des effets de bords (bugs, lenteurs...)

Type de doublure de test



Dummy Object

Il peut arriver que notre SUT requière une dépendance inutile pour notre cas de test. Cela peut être par exemple une classe requise du constructeur de l'objet qu'on test. Une technique pour remplacer cette dépendance et d'en instancier un du même type (remplissant le même contrat) mais renvoyant null à chaque méthode.

Pourquoi null ?

- L'objet est inutile dans notre cas de test.
- Son seul objectif et de pouvoir construire notre SUT
- On ne veut pas que quelqu'un se mette à l'utiliser

Exemple de Dummy Object en PHP

```
class LoggerDummy implements Logger
{
    public function debug()
    {
        return null; // not even necessary here
    }

    ...
}
```

Stub Object

C'est un objet qui dispose d'une réponse pré-configurée et fixe. Un stub n'accorde pas d'importance à :

- quels arguments sont fournis lorsque l'une de ses méthodes est appelée
- combien de fois la méthode est appelée
- comment il a retourner cette valeur

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.

```
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

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

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

Mock Object

Un mock est une sorte d'objet préprogrammé avec des attentes sur son comportement

```
class AcceptingAuthorizerMock
{
  public $authorizeWasCalled = false;

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

  public boolean verify() {
    return $authorizedWasCalled;
  }
}
```

Spy vs Mock

- Un Spy est un mock partiel.
- On peut utiliser un objet réel et ne remplacer que quelques méthodes
- Il peut maintenir un état afin que l'on puisse le tester par la suite par des assertions
- Ces assertions sont effectués une fois que le Spy à été utilisé.

Mock vs Spy

- Un Mock défini un comportement avant d'être utilisé
- Il se teste de lui même
- C'est un objet totalement différent.
- Il ne peut pas utiliser les méthodes de l'objet qu'il remplace.
- Ne se préoccupe pas de son état mais vérifie que les appels des méthodes ont respecté certaines attentes
- Il est généralement plus simple d'utiliser un Mock qu'un Spy

Test double avec PHPUnit

> *Documentation*

Exemple Dummy avec PHPunit

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

Exemple de Stub avec PHPUnit

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

Exemple de Fake avec PHPUnit

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

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

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

Exemple de Mock avec PHPUnit

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

Exemple de Spy avec PHPUnit

```
$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]);
```

Des questions ?

Exercice Bonus #1

1. Faire une classe Pricing qui dispose d'un panier (Cart) et qui utilise la class Math pour calculer les frais de ports
2. Faire les tests sur la classe Pricing en créant des Mocks pour la classe (Cart)

Exercice Bonus #2

Allez plus loin en remplaçant tous les appels de la classe Math par des mocks

> *Refactoriser la classe Math et son utilisation en conséquence*