Apprendre à tester unitairement son code en PHP

Support de cours pour la formation Apprendre à tester unitairement son code en PHP de Novembre 2024

Formateur : Dufrène Valérian

“Déclaration d’activité enregistrée sous le numéro 32800232680 auprès du préfet de région HAUTS-DE-FRANCE”

Déroulé et objectif

La formation "Apprendre à tester unitairement son code en PHP" est découpée en 4 chapitres et est accessible à toute personne ayant acquis des bases en PHP.

Le programme de formation a été conçu pour vous permettre d'effectuer une nouvelle montée en compétences de manière fluide, par rapport à l'apprentissage des bases.
Objectif final : Savoir tester unitairement son code, en PHP.

À la fin du premier chapitre, vous saurez :

➔ **Objectif du chapitre 1 :** Découvrir les tests unitaires, sélectionner un outil de conception de tests et créer nos premiers tests unitaires, en PHP.

➔ **Module 1 :**
- ◆ Expliquer l'utilité des tests unitaires.
- ◆ Différencier les types de tests possibles dans un projet.
- ◆ Déterminer les avantages des tests unitaires.
- ◆ Sélectionner votre outil de tests unitaires.
- ◆ **Objectif du Module 1 :** Découvrir les tests unitaires et les outils disponibles pour les créer.

➔ **Module 2 :**
- ◆ Installer un outil permettant la création de tests unitaires en PHP.
- ◆ Configurer l'outil sélectionné sur votre projet.
- ◆ Comprendre la structure de base d'un projet de tests.
- ◆ **Objectif du module 2 :** Sélectionner un outil de tests unitaires, l'installer et le configurer sur un projet PHP.

➔ **Module 3 :**
- ◆ Écrire des tests unitaires.
- ◆ Différencier les méthodes utilisables pour tester votre code.
- ◆ Lire et comprendre le résultat des tests unitaires.
- ◆ **Objectif du module 3 :** Découvrir la syntaxe de l'outil de test et créer ses premiers tests unitaires.

➔ **Module 4 :**

- Écrire des tests unitaires.
- Différencier les méthodes utilisables pour tester votre code.
- Lire et comprendre le résultat des tests unitaires.
- **Objectif du module 4 :** Découvrir la syntaxe de l'outil de test et créer ses premiers tests unitaires.

À la fin du deuxième chapitre, vous saurez :

➔ **Objectif du chapitre 2 :** Structurer ses tests et tester des cas plus complexes.

➔ **Module 1 :**

- Utiliser "setUp" et "tearDown" pour préparer et nettoyer les tests.
- Expliquer pourquoi ces étapes simplifient l'organisation des tests.
- Écrire des tests avec des étapes de préparation
- **Objectif du Module 1 :** Préparer et nettoyer ses tests.

➔ **Module 2 :**

- Utiliser "@dataProvider" pour tester plusieurs scénarios en un seul test.
- Comment structurer vos tests pour couvrir des cas variés.
- Créer un test qui vérifie différents scénarios d'une même fonction.
- **Objectif du module 2 :** Tester des cas multiples avec un seul test.

➔ **Module 3 :**

- Tester si une erreur ou une exception est bien déclenchée.
- Utiliser "expectException" pour tester les erreurs.
- Écrire des tests pour des cas d'erreurs dans votre code.
- **Objectif du module 3 :** Tester les erreurs et les exceptions dans un test unitaire.

À la fin du troisième chapitre, vous saurez :

➔ **Objectif du chapitre 3 :** Apprendre à tester du code ayant des dépendances.

➔ **Module 1 :**

- Comprendre les termes "mock" et "stub" (objets simulés pour les tests).
- Pourquoi on utilise des mocks pour isoler les tests.
- Comment créer des mocks simples avec PHPUnit.
- **Objectif du Module 1 :** Découvrir ce qu'est un mock et définir son utilité.

➔ **Module 2 :**

- Configurer un mock pour imiter le comportement de méthodes ou d'objets.
- Personnaliser les mocks pour qu'ils agissent comme on veut.
- Tester une fonction avec des mocks pour simuler des dépendances.
- **Objectif du module 2 :** Créer des tests avec des objets simulés.

➔ **Module 3 :**

- Comment gérer des dépendances comme des services, bases de données ou API.
- Isoler les tests et éviter les erreurs liées aux services externes.
- Écrire des tests unitaires avec des mocks pour simuler des services.
- **Objectif du module 3 :** Tester du code qui dépend de services externes.

À la fin du quatrième chapitre, vous saurez :

➔ **Objectif du chapitre 4 :** Organiser ses tests et projet final.

➔ **Module 1 :**

- Comment structurer ses tests dans un projet PHP.
- Définir les bonnes pratiques pour nommer et organiser vos fichiers de tests.
- Améliorer la rapidité et l'efficacité des tests.
- **Objectif du Module 1 :** Savoir organiser ses tests et éviter les erreurs courantes.

➔ **Module 2 :**

- Écrire des tests unitaires complets pour une classe ou un module PHP.
- Éviter les erreurs courantes et écrire des tests robustes.
- **Objectif du module 2 :** Créer des tests unitaires pour un projet réel.

Objectif pédagogique

À la fin de cette formation, les participants devraient être capables de :

- ➔ Comprendre ce qu'est un test unitaire et pourquoi c'est utile.
- ➔ Utiliser PHPUnit pour créer, exécuter et analyser des tests unitaires simples et avancés.
- ➔ Gérer les dépendances et tester des scénarios variés de manière isolée.
- ➔ Structurer un projet de tests de manière organisée, efficace et compréhensible.

Ce programme permet de progresser du niveau débutant vers des tests plus avancés.

Prérequis

- ➔ **Connaissances de base en PHP :**
  - ◆ Comprendre la syntaxe de PHP.
  - ◆ Savoir créer et utiliser des fonctions, des classes et des méthodes en PHP.
  - ◆ Connaissance des notions de base de la programmation orientée objet (POO), comme les objets, les classes, et les méthodes.
- ➔ **Expérience pratique avec un projet PHP :**
  - ◆ Avoir déjà développé une petite application ou des scripts en PHP.
  - ◆ Être à l'aise avec la structure de fichiers dans un projet PHP.
- ➔ **Familiarité avec Composer :**
  - ◆ Connaître les bases de Composer pour installer et gérer des bibliothèques PHP.
- ➔ **Environnement de développement installé :**
  - ◆ Disposer d'un environnement de développement PHP prêt à l'emploi (par exemple, un serveur local comme XAMPP, WAMP ou MAMP, ou un environnement Docker).
  - ◆ Avoir installé un éditeur de code (comme VS Code, PHPStorm, ou Sublime Text) et savoir l'utiliser pour écrire et exécuter du code.
- ➔ **Notions de base sur la ligne de commande :**
  - ◆ Connaître les commandes de base pour naviguer dans le terminal et exécuter des scripts PHP.

- Savoir comment lancer des commandes de Composer et PHPUnit via la ligne de commande.

Ce qui n'est pas nécessaire

Les participants n’ont pas besoin de connaître :

➔ Les tests unitaires ou PHPUnit en particulier (tout sera abordé depuis le début).
➔ Les concepts d’intégration continue ou les outils de CI/CD.
➔ Les concepts avancés en PHP, comme les design patterns ou la gestion des bases de données (bien que cela pourrait être un plus pour certains exercices).

Chapitre 1 - Module 1 - Qu’est-ce qu’un test unitaire et pourquoi c’est important ?

Objectifs

- ➔ Comprendre ce qu'est un test unitaire.
- ➔ Découvrir les avantages des tests unitaires dans le développement logiciel.
- ➔ Connaître les différents types de tests.
- ➔ Apprendre pourquoi les tests unitaires sont importants pour le développement.

Définition d’un test unitaire

Les tests unitaires permettent de vérifier si une petite partie de code (souvent une fonction ou une méthode) fonctionne comme prévu, indépendamment des autres composants du programme.

Un test unitaire est un test qui vérifie le comportement d'une petite unité de code de manière isolée, typiquement une méthode ou une fonction. Ce test doit être rapide, indépendant des autres parties du système, et vise à vérifier un seul comportement à la fois.

Exemple : Si vous avez une méthode `add` dans une classe `Calculator`, le test unitaire vérifiera que l'addition de deux nombres donne le bon résultat.

Pourquoi faire des tests unitaires ?

- ➔ **Détection des bugs** : Les tests unitaires permettent de repérer rapidement les erreurs dans les petites unités de code, ce qui facilite la correction.
- ➔ **Maintenance simplifiée** : En cas de modification du code, les tests unitaires permettent de vérifier que le comportement du code n'a pas changé de manière inattendue.
- ➔ **Documenter le code** : Les tests servent également de documentation vivante pour expliquer comment une fonction ou une méthode est censée fonctionner.
- ➔ **Prévenir les régressions** : Lors de l'ajout de nouvelles fonctionnalités, les tests unitaires aident à s'assurer que les fonctionnalités existantes ne sont pas altérées.

Types de tests en développement

- ➔ **Test unitaire** : Vérifie une fonction ou une méthode isolée.
- ➔ **Test fonctionnel** : Vérifie le fonctionnement global d'une fonctionnalité.
- ➔ **Test d'intégration** : Vérifie que différents modules fonctionnent bien ensemble.

Exemple de code : Test simple d'une fonction add

Imaginons une classe `Calculator` qui contient une méthode `add` pour additionner deux nombres :

```
01.  <?php
02.  class Calculator
03.  {
04.      public function add($a, $b)
05.      {
06.          return $a + $b;
07.      }
08.  }
```

Le test unitaire associé pourrait ressembler à ceci :

```
01.  <?php
02.  use PHPUnit\Framework\TestCase;
03.
04.  class CalculatorTest extends TestCase
05.  {
06.      public function testAdd()
07.      {
08.          $calculator = new Calculator();
09.          $result = $calculator->add(2, 3);
10.          $this->assertEquals(5, $result); // On vérifie que
   l'addition de 2 et 3 donne bien 5.
11.      }
12.  }
```

➔ **Explication** : Le test crée un objet de la classe `Calculator`, appelle la méthode `add` avec les arguments 2 et 3, et vérifie que le résultat retourné est bien 5. La méthode `assertEquals` est utilisée pour comparer la valeur attendue avec la valeur obtenue.

Exercice

Imaginez une fonction de calcul permettant de multiplier 2 nombres et le test à lui associer.

Chapitre 1 - Module 2 - Installation et configuration de PHPUnit

Objectifs

- ➔ Apprendre à installer et configurer PHPUnit.
- ➔ Comprendre les bases de l'utilisation de PHPUnit pour écrire et exécuter des tests.
- ➔ Installer PHPUnit via Composer.
- ➔ Configurer un fichier de base `phpunit.xml`.

Installation de PHPUnit

- ➔ **PHPUnit** est une bibliothèque PHP qui permet d'écrire et d'exécuter des tests unitaires.
- ➔ L'installation peut se faire de plusieurs manières :
  - ◆ **Via Composer** : `composer require --dev phpunit/phpunit`
  - ◆ **Téléchargement manuel** : Vous pouvez télécharger PHPUnit directement depuis son site officiel.

Installation de PHPUnit via Composer

Dans votre projet PHP, lancez la commande :

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

Vérifiez l'installation avec :
`vendor/bin/phpunit --version`

Configuration de PHPUnit

Créez un fichier `phpunit.xml` à la racine de votre projet :

```
<phpunit bootstrap="vendor/autoload.php">

    <testsuites>

        <testsuite name="Application Test Suite">

            <directory>tests</directory>

        </testsuite>

    </testsuites>

</phpunit>
```

Après l'installation de PHPUnit via Composer, vous pouvez exécuter PHPUnit à l'aide de la commande suivante :
`vendor/bin/phpunit`

PHPUnit recherche automatiquement les fichiers de test dans le répertoire `tests/` (par défaut), créez un dossier `tests/` où vous allez placer tous vos fichiers de test.

Structure de base d'un test PHPUnit

- ➔ Un fichier de test est une classe qui étend la classe `PHPUnit\Framework\TestCase` (avec le mot clé **extends**).
- ➔ Chaque méthode de test commence par `test` et contient des assertions pour vérifier les comportements attendus.

Écrire son premier test avec PHPUnit

- ➔ Exemple d'un test simple avec PHPUnit :

```
<?php

use PHPUnit\Framework\TestCase;


class CalculatorTest extends TestCase

{

    public function testAdd()

    {

        $calculator = new Calculator();

        $result = $calculator->add(1, 2);

        $this->assertEquals(3, $result);

    }

}
```

Exécution des tests

Pour exécuter les tests, vous pouvez lancer la commande suivante dans votre terminal : `vendor/bin/phpunit --testdox`

➔ Cette commande exécute tous les tests et affiche des résultats lisibles pour chaque test effectué.

Exercice

En suivant les étapes de ce chapitre, installez PHPUnit et créez le fichier de configuration de votre projet, puis :

➔ **Créez une classe Calculator** avec une méthode add et écrire un test unitaire pour cette méthode.
➔ Exécutez les tests pour vérifier que tout fonctionne correctement.

Chapitre 1 - Module 3 - Écrire ses premiers tests unitaires

Objectifs

➔ Écrire des tests unitaires de base en utilisant PHPUnit.
➔ Découvrir les assertions de base.
➔ Comprendre l'importance des assertions dans les tests unitaires.
➔ Apprendre à utiliser différentes assertions pour vérifier les résultats des tests.

Qu'est-ce qu'une assertion ?

➔ Une assertion est une instruction qui vérifie qu'une condition spécifique est vraie. Si l'assertion échoue, le test échoue également.

➔ Exemple : Si vous voulez vérifier qu'une méthode `add` renvoie bien 5, vous utilisez `assertEquals(5, $result)`.

Principales assertions dans PHPUnit

- **`assertEquals($expected, $actual)`** : Vérifie que les valeurs attendues et réelles sont identiques.
- **`assertTrue($condition)`** : Vérifie qu'une condition est vraie.
- **`assertFalse($condition)`** : Vérifie qu'une condition est fausse.
- **`assertNull($variable)`** : Vérifie que la variable est nulle.
- **`assertNotNull($variable)`** : Vérifie que la variable n'est pas nulle.
- **`assertCount($expectedCount, $array)`** : Vérifie que le nombre d'éléments dans un tableau ou une collection est égal au nombre attendu.

Exemples d'utilisation des assertions

➔ **Test avec `assertTrue`**

```
01.  public function testIsEven()
02.  {
03.      $this->assertTrue(4 % 2 == 0); // Vérifie que 4 est pair
04.  }
```

➔ **Test avec `assertNull`**

```
01.  public function testIsNull()
02.  {
03.      $value = null;
04.      $this->assertNull($value); // Vérifie que la valeur est
   nulle
05.  }
```

➔ **Test avec `assertCount`**

```
01.  public function testArrayCount()
02.  {
03.      $array = [1, 2, 3];
04.      $this->assertCount(3, $array); // Vérifie que le tableau
   contient 3 éléments
05.  }
```

Écrire un premier test avec PHPUnit

Créez un fichier `CalculatorTest.php` dans le dossier `tests/`, puis écrivez le code suivant :

```
01.<?php
02.  use PHPUnit\Framework\TestCase;
03.
04.  class CalculatorTest extends TestCase
05.  {
06.      public function testAddition()
07.      {
08.          $result = 2 + 3;
09.          $this->assertEquals(5, $result, "L'addition devrait
   être correcte");
```

```
10.       }
11.   }
```

Exécutez le test :

```
vendor/bin/phpunit
```

Les assertions les plus courantes

- ➔ `assertEquals(expected, actual)`
- ➔ `assertTrue(condition)`
- ➔ `assertFalse(condition)`

Exemples d'utilisation :

```
01. public function testAssertions()
02. {
03.     $this->assertTrue(true);
04.     $this->assertFalse(false);
05.     $this->assertEquals(5, 5);
06. }
```

Analyser les résultats de PHPUnit

- ➔ Lorsque le test passe, vous verrez un message de succès.
- ➔ Si un test échoue, PHPUnit vous montrera l'erreur et la ligne où le test a échoué.

Exercice

- ➔ Créez une classe `Calculator` avec une méthode `add` qui additionne deux nombres. Puis, écrivez un test pour vérifier cette méthode.
- ➔ Écrivez une méthode dans la classe `Calculator` pour soustraire deux nombres et créer un test unitaire pour cette méthode.
- ➔ Utiliser différentes assertions pour tester des comportements variés dans votre classe `Calculator`.

Exemple :

Classe `Calculator.php` :

```php
<?php
class Calculator
{
    public function add($a, $b)
    {
        return $a + $b;
    }
}
```

Test `CalculatorTest.php` :

```php
<?php
use PHPUnit\Framework\TestCase;

class CalculatorTest extends TestCase
{
    public function testAdd()
    {
        $calculator = new Calculator();
        $result = $calculator->add(2, 3);
        $this->assertEquals(5, $result, "La méthode add()
```

```
    devrait retourner 5");
11.     }
12. }
```

Chapitre 2 - Module 1 - Préparer et nettoyer ses tests avec “setUp” et “tearDown”

Objectifs

01. Comprendre l'utilité de `setUp` et `tearDown` pour configurer des conditions de test.
02. Utiliser ces méthodes pour préparer et nettoyer l’environnement de test.
03. Apprendre à configurer les ressources nécessaires avant chaque test.
04. Nettoyer et libérer les ressources après chaque test pour éviter les interférences.

Méthodes setUp et tearDown

➔ La méthode `setUp` est exécutée automatiquement avant chaque test. Elle permet d'initialiser des objets ou des ressources qui seront utilisés dans plusieurs tests, évitant ainsi la duplication de code.

➔ La méthode `tearDown`, quant à elle, est exécutée après chaque test pour libérer les ressources ou réinitialiser l’état de l’application.

Exemple d'utilisation de setUp et tearDown

Prenons l'exemple de la classe `Calculator`. Nous allons créer une instance de la classe dans `setUp` pour pouvoir la réutiliser dans chaque test.

La classe de tests `CalculatorTest` vérifiera les fonctionnalités de `Calculator`, en utilisant `setUp` pour créer une instance de `Calculator` avant chaque test et `tearDown` pour la nettoyer après chaque test.

Exemple de code :

```
01.  class CalculatorTest extends TestCase
02.  {
03.      private $calculator;
04.
05.      protected function setUp(): void
06.      {
07.          // Initialiser une nouvelle instance de Calculator
   avant chaque test
08.          $this->calculator = new Calculator();
09.      }
10.
11.      public function testAddReturnsCorrectSum()
12.      {
13.          $result = $this->calculator->add(2, 3);
14.          $this->assertEquals(5, $result);
15.      }
16.
17.      public function testSubtractReturnsCorrectDifference()
18.      {
19.          $result = $this->calculator->subtract(10, 5);
20.          $this->assertEquals(5, $result);
21.      }
22.
```

```
23.      protected function tearDown(): void
24.      {
25.          // Nettoyer la ressource pour libérer la mémoire
   après chaque test
26.          unset($this->calculator);
27.      }
28.  }
```

Dans cet exemple, `setUp` crée une instance de `Calculator` qui est ensuite utilisée dans chaque test. `tearDown` nettoie cette instance une fois le test terminé.

Exercice

Ajoutez une méthode `subtract` dans la classe `Calculator` et créez un test pour cette méthode en utilisant `setUp` et `tearDown`.

Chapitre 2 - Module 2 - Tester différents cas avec un seul test grâce à “@dataProvider”

Objectifs

- ★ Gagner du temps en testant plusieurs cas en une seule méthode de test.
- ★ Réduire le code dupliqué en testant plusieurs cas avec un même test.
- ★ Apprendre à utiliser l’annotation `@dataProvider` pour passer plusieurs jeux de données à un seul test.

Introduction au @dataProvider

- ➔ `@dataProvider` permet de spécifier plusieurs ensembles de données pour un même test, qui sera exécuté pour chaque ensemble.
- ➔ Vous pouvez créer une méthode de fournisseur de données qui retourne un tableau de valeurs à tester.
- ➔ Cela évite d'écrire des tests similaires pour des cas différents, ce qui rend le code plus maintenable et concis.

Exemple d'utilisation du @dataProvider

Ajoutons un `dataProvider` pour tester différents cas d'addition avec la méthode `add`.

Dans cet exemple, nous allons tester différentes additions dans un seul test grâce à `@dataProvider`.

Exemple de code :

Dans cet exemple, `additionProvider` fournit des paires de valeurs et leur résultat attendu, et `testAdd` les teste toutes en une seule méthode.

```
01. class CalculatorTest extends TestCase
02. {
03.     private $calculator;
04.
05.     protected function setUp(): void
06.     {
07.         $this->calculator = new Calculator();
08.     }
09.
10.     /**
11.      * @dataProvider additionProvider
12.      */
13.     public function testAdd($a, $b, $expected)
```

```
14.     {
15.         $result = $this->calculator->add($a, $b);
16.         $this->assertEquals($expected, $result);
17.     }
18.
19.     public function additionProvider()
20.     {
21.         return [
22.             [1, 2, 3],   // Cas 1 : 1 + 2 = 3
23.             [0, 0, 0],   // Cas 2 : 0 + 0 = 0
24.             [-1, 1, 0],  // Cas 3 : -1 + 1 = 0
25.             [3, 3, 6]    // Cas 4 : 3 + 3 = 6
26.         ];
27.     }
28.
29.     protected function tearDown(): void
30.     {
31.         unset($this->calculator);
32.     }
33. }
```

Avantages de @dataProvider

- ➔ **Réduction de la redondance** : Pas besoin de créer plusieurs tests pour des cas similaires.
- ➔ **Maintenabilité** : Les cas de test peuvent être ajoutés ou modifiés dans une seule méthode (le fournisseur de données).
- ➔ **Clarté** : La méthode de test est centrée sur la logique de vérification, tandis que les données de test sont isolées dans la méthode de fournisseur.

Exercice

Créez un `dataProvider` pour la méthode `subtract` et testez différents scénarios de soustraction.

Chapitre 2 - Module 3 - Tester les erreurs et les exceptions

Objectifs

- ➔ Tester si des erreurs ou des exceptions sont correctement levées par le code.
- ➔ Utiliser l'annotation `@expectedException` pour tester les exceptions.
- ➔ Utiliser `expectException` et `expectExceptionMessage` pour gérer les erreurs dans PHPUnit.

Détection des erreurs avec expectException

- ➔ `expectException` permet de s'assurer qu'une exception spécifique est levée par la méthode testée.
- ➔ Exemple : Testons la méthode `divide` de `Calculator`, qui lève une exception `DivisionByZeroError` si le dénominateur est zéro.

```
01. class CalculatorTest extends TestCase
02. {
03.     private $calculator;
04.
```

```
    protected function setUp(): void
    {
        $this->calculator = new Calculator();
    }

    public function testDivideByZeroThrowsException()
    {
        $this->expectException(DivisionByZeroError::class);
        $this->calculator->divide(10, 0);
    }

    protected function tearDown(): void
    {
        unset($this->calculator);
    }
}
```

Vérifier le message d'exception avec expectExceptionMessage

- ➔ `expectExceptionMessage` permet de vérifier que le message d'erreur de l'exception est celui attendu.
- ➔ Exemple : Vérifions que la méthode `divide` retourne le message "Division by zero".

```
public function testDivideByZeroThrowsCorrectMessage()
{
    $this->expectException(DivisionByZeroError::class);
    $this->expectExceptionMessage("Division by zero");
    $this->calculator->divide(10, 0);
}
```

Pourquoi tester les erreurs et les exceptions ?

- ➔ **Les erreurs font partie de la réalité du code :** il est crucial de s'assurer qu'elles sont correctement gérées.
- ➔ **Prévenir les erreurs critiques** : Garantit que le code gère correctement les cas d'erreur.
- ➔ **Améliorer la robustesse** : S'assurer que le code réagit de manière contrôlée aux entrées incorrectes ou aux situations exceptionnelles.
- ➔ **Documentation des comportements d'erreur** : Ces tests montrent clairement les conditions d'erreur attendues, ce qui est utile pour les autres développeurs et les mainteneurs du code.
- ➔ PHPUnit permet de vérifier que les exceptions sont bien levées dans certains cas.

Tester une exception avec expectException

- ➔ Si notre classe `Calculator` contient une méthode `divide` qui lance une exception en cas de division par zéro, nous pouvons tester ce comportement avec `expectException`.
- ➔

Exemple de code :

```
<?php
use PHPUnit\Framework\TestCase;

class CalculatorTest extends TestCase
{
    private $calculator;

    protected function setUp(): void
    {
        $this->calculator = new Calculator();
    }
```

```
★
★     public function testDivideByZeroThrowsException()
★     {
★         $this->expectException(DivisionByZeroError::class);
★         $this->calculator->divide(5, 0);
★     }
★ }
```

Dans cet exemple, nous nous attendons à ce que `divide` lève une exception de type `DivisionByZeroError` si le dénominateur est zéro.

- ➔ **Tester un message d'erreur spécifique**
  - ◆ Vous pouvez également vérifier que le message d'erreur est celui attendu en utilisant `expectExceptionMessage`.

Exercice

- ➔ **Créer une méthode `divide`** dans la classe `Calculator` qui lève une exception `DivisionByZeroError` avec le message "Division by zero" si le dénominateur est zéro.
- ➔ **Écrire un test avec `@dataProvider`** pour vérifier que `divide` retourne le résultat correct pour plusieurs paires de valeurs, sauf pour une division par zéro.
- ➔ **Utiliser `expectException` et `expectExceptionMessage`** pour tester les erreurs de division par zéro.

Chapitre 3 - Module 1 - Qu’est-ce qu’un mock et pourquoi en a-t-on besoin ?

Objectifs

- ➔ Comprendre le rôle des mocks dans les tests unitaires.
- ➔ Découvrir les différences entre les mocks et les stubs.
- ➔ Comprendre les concepts de doubles de test, comme les mocks et les stubs.
- ➔ Savoir quand et comment utiliser les doubles de test pour isoler le code testé.

Pourquoi utiliser des mocks ?

- ➔ Dans les tests unitaires, il est souvent nécessaire d’isoler le code testé de ses dépendances pour éviter les effets de bord.
- ➔ **Mocks** : Objets simulés qui imitent le comportement des dépendances et permettent de vérifier les interactions.
- ➔ **Stubs** : Versions simplifiées des dépendances qui retournent des valeurs prédéfinies, mais sans vérifier les interactions.

Introduction aux doubles de test

- ➔ Les doubles de test sont des objets simulés permettant d'isoler le code testé de ses dépendances.
- ➔ Ils sont utilisés pour imiter les comportements de classes ou d'objets externes sans exécuter leur logique réelle.

Types de doubles de test

- ➔ **Stub** : Fournit une réponse prédéfinie à une méthode sans implémenter sa logique.
- ➔ **Mock** : Permet de vérifier qu'une méthode a été appelée avec des arguments spécifiques.
- ➔ **Spy** : Enregistre des informations sur la façon dont une méthode a été utilisée, mais ne contrôle pas le flux de l'application.

Exemple d'utilisation des mocks avec PHPUnit

Imaginons que nous ayons une classe `UserService` qui dépend d'une classe `EmailService` pour envoyer un email. Nous allons utiliser un mock pour simuler `EmailService`.

```
01. class UserServiceTest extends TestCase
02. {
03.     public function testRegisterSendsWelcomeEmail()
04.     {
05.         $emailServiceMock = $this->createMock(EmailService::class);
06.
07.         // On configure le mock pour vérifier que sendEmail est appelé une fois avec
    l'email 'user@example.com'
08.         $emailServiceMock->expects($this->once())
09.                          ->method('sendEmail')
10.                          ->with($this->equalTo('user@example.com'));
```

```
11.
12.         $userService = new UserService($emailServiceMock);
13.         $userService->register('user@example.com');
14.     }
15. }
```

Avantages des doubles de test

- ➔ **Isolation** : Teste uniquement la logique de la classe cible, sans exécuter la logique des dépendances.
- ➔ **Rapidité** : Moins de ressources sont utilisées en évitant les interactions externes (comme les appels à la base de données).
- ➔ **Contrôle** : Permet de tester des situations spécifiques et de vérifier les interactions entre objets.

Exemple d'utilisation de mocks

Imaginons que notre classe `Calculator` ait besoin d'une classe `ExternalCalculatorService` pour réaliser des calculs avancés.

Lors des tests, nous n'avons pas besoin d'appeler réellement ce service externe, qui pourrait être lent ou coûteux.

Nous allons donc utiliser un mock pour `ExternalCalculatorService` afin de simuler les réponses du service et vérifier les interactions entre `Calculator` et `ExternalCalculatorService`.

```
01. class Calculator
02. {
03.     private $externalService;
04.
```

```
    public function __construct(ExternalCalculatorService $externalService)
    {
        $this->externalService = $externalService;
    }

    public function calculateAdvancedOperation($input)
    {
        // Appelle le service externe pour un calcul avancé
        return $this->externalService->performComplexCalculation($input);
    }
}

class CalculatorTest extends TestCase
{
    public function testCalculateAdvancedOperation()
    {
        // Création d'un mock pour ExternalCalculatorService
        $externalServiceMock = $this->createMock(ExternalCalculatorService::class);

        // On configure le mock pour retourner 42 quand performComplexCalculation est appelé avec l'argument 7
        $externalServiceMock->expects($this->once())
                            ->method('performComplexCalculation')
                            ->with($this->equalTo(7))
                            ->willReturn(42);

        // Instancie Calculator avec le mock comme dépendance
        $calculator = new Calculator($externalServiceMock);
        // Appel de la méthode testée
        $result = $calculator->calculateAdvancedOperation(7);

        // Vérification du résultat attendu
        $this->assertEquals(42, $result);
    }
}
```

Dans cet exemple, `performComplexCalculation` est simulé par le mock pour retourner une valeur prédéfinie, `42`. Nous vérifions également que la méthode est appelée avec le bon argument (`7`).

Exercice participatif

Discuter de situations où il serait utile d'utiliser des mocks, et créer une liste non exhaustive de cas d'utilisation.

Voici quelques situations courantes où l'utilisation de mocks est particulièrement utile pour isoler les tests unitaires des dépendances externes :

- ➔ **Connexion à une base de données** : Les tests qui impliquent des opérations de base de données peuvent être lents et nécessitent un environnement configuré. En utilisant un mock pour simuler la connexion et les requêtes, on peut tester la logique de l'application sans exiger un accès à une base de données réelle.
- ➔ **Services externes** : Lorsque le code utilise une API externe pour récupérer des données, envoyer des notifications, ou effectuer des calculs avancés, il est pertinent d'utiliser des mocks pour éviter les appels réseau, qui peuvent être coûteux et ralentir les tests.
- ➔ **Envoi de courriels** : Lors des tests d'une fonctionnalité qui envoie des e-mails (par exemple, l'envoi d'un e-mail de bienvenue après inscription), un mock du service d'envoi d'e-mails permet de vérifier que l'e-mail est bien préparé et prêt à être envoyé sans envoyer de véritable message.
- ➔ **Traitements sur des fichiers** : Si le code manipule des fichiers ou enregistre des données sur le disque, un mock peut simuler ces opérations pour éviter d'interagir avec le système de fichiers, ce qui rend les tests plus rapides et prévisibles.
- ➔ **Notifications via des services de messagerie** : Dans une application qui utilise des notifications via SMS ou un service de messagerie instantanée, on peut se servir de mocks pour simuler l'envoi des notifications et vérifier les paramètres utilisés sans réellement envoyer de messages.

Chapitre 3 - Module 2 - Créer des tests avec des objets simulés

Objectifs

- ➔ Utiliser PHPUnit pour créer des mocks.
- ➔ Configurer un mock pour simuler le comportement d'une dépendance.

Contenu

- ➔ **Créer et configurer un mock en PHPUnit**
  - ◆ Pour créer un mock, utilisez `createMock` pour remplacer une dépendance.
  - ◆ Vous pouvez spécifier le comportement de méthodes spécifiques dans le mock.
- ➔ **Exemple d'utilisation d'un mock**
  - ◆ Ajoutons une méthode `advancedAdd` dans `Calculator` qui utilise un service externe pour ajouter deux nombres.

Code de `Calculator.php`

```
➔ <?php
➔ class Calculator
➔ {
➔     private $externalService;
➔
➔     public function __construct($externalService)
➔     {
➔         $this->externalService = $externalService;
```

```
➔     }
➔
➔     public function advancedAdd($a, $b)
➔     {
➔         return $this->externalService->performAddition($a,
  $b);
➔     }
➔ }
```

Test `CalculatorTest.php`

```
➔ <?php
➔ use PHPUnit\Framework\TestCase;
➔
➔ class CalculatorTest extends TestCase
➔ {
➔     public function testAdvancedAddUsesExternalService()
➔     {
➔         // Création d'un mock pour le service externe
➔         $mockService =
  $this->createMock(ExternalCalculatorService::class);
➔
➔         // Définition du comportement du mock
➔         $mockService->method('performAddition')->with(2,
  3)->willReturn(5);
➔
➔         // Création d'une instance de Calculator avec le mock
  en tant que dépendance
➔         $calculator = new Calculator($mockService);
➔
➔         // Test de la méthode advancedAdd
➔         $result = $calculator->advancedAdd(2, 3);
➔         $this->assertEquals(5, $result);
➔     }
```

➔ `}`

Dans cet exemple, nous spécifions que `performAddition(2, 3)` doit retourner `5` lorsque cette méthode est appelée sur le mock.

Exercice

Ajoutez une autre méthode utilisant le même service externe et créez un test pour cette méthode avec un mock.

Chapitre 3 - Module 3 - Tester du code ayant des dépendances externes

Objectifs

- ➔ Apprendre à tester le code qui utilise des dépendances complexes, comme des bases de données ou des API.
- ➔ S'assurer que les tests restent rapides et indépendants des services externes.

Contenu

- ➔ **Pourquoi éviter de tester directement les services externes ?**
  - ◆ Tester des services externes dans les tests unitaires peut ralentir les tests et rendre les résultats imprévisibles.
  - ◆ Utiliser des mocks permet de simuler les réponses des services sans les appeler réellement.
- ➔ **Simuler un service externe avec un mock**
  - ◆ Dans l'exemple suivant, nous ajoutons une méthode `fetchDataFromAPI` à la classe `Calculator`, qui utilise un service pour récupérer des données.

Exemple de code

Supposons que `Calculator` utilise un service `APIService`.

- ➔ `<?php`

```
➔ class Calculator
➔ {
➔     private $apiService;
➔
➔     public function __construct($apiService)
➔     {
➔         $this->apiService = $apiService;
➔     }
➔
➔     public function fetchDataFromAPI($endpoint)
➔     {
➔         return $this->apiService->fetch($endpoint);
➔     }
➔ }
```

Dans le test, nous mockons `APIService` pour simuler le comportement de `fetch`.

Test

```
➔ <?php
➔ use PHPUnit\Framework\TestCase;
➔
➔ class CalculatorTest extends TestCase
➔ {
➔     public function testFetchDataFromAPI()
➔     {
➔         // Création du mock pour le service d'API
➔         $mockAPIService =
  $this->createMock(APIService::class);
➔
➔         // Simulation de la réponse du service
➔
  $mockAPIService->method('fetch')->with('endpoint')->willReturn
  (['data' => 'value']);
➔
```

```
        $calculator = new Calculator($mockAPIService);
        $result = $calculator->fetchDataFromAPI('endpoint');

        $this->assertEquals(['data' => 'value'], $result);
    }
}
```

Exercice

Ajoutez une méthode `multiply` dans `Calculator` qui utilise le service `APIService`. Écrivez un test pour cette méthode en utilisant un mock pour `APIService`.

Chapitre 3 - Module 4 - Mesurer la couverture de code avec PHPUnit

Objectifs

- Comprendre la notion de couverture de code.
- Utiliser PHPUnit pour générer des rapports de couverture de code.

Définition de la couverture de code

- La couverture de code est un indicateur qui mesure le pourcentage de lignes de code exécutées par les tests.

➔ Elle permet d'identifier les parties du code qui ne sont pas couvertes par des tests, révélant ainsi des zones potentiellement risquées.

Configurer PHPUnit pour la couverture de code

➔ Pour générer un rapport de couverture de code, installez l'extension **Xdebug** (ou **phpdbg** sur certaines versions de PHP).
➔ Commande pour exécuter les tests avec un rapport de couverture : `phpunit --coverage-html coverage-report`
➔ Le rapport généré dans le dossier `coverage-report/` fournit une vue détaillée des classes et des méthodes couvertes par les tests, avec des pourcentages de couverture.

Interpréter les résultats de la couverture de code

➔ **Couverture à 100 %** : C'est l'objectif idéal, mais ce n'est pas toujours nécessaire ou réaliste. Un bon niveau de couverture dépend de l'application et du projet.
➔ **Analyser les zones non couvertes** : Identifiez les zones non testées pour ajouter des tests ou améliorer celles existantes.

Exercice pratique pour les doubles de test et la couverture de code

➔ **Créez un double de test** pour une classe `EmailService` utilisée dans `UserService`. Assurez-vous que la méthode `sendEmail` est bien appelée lors de l'enregistrement d'un nouvel utilisateur.
➔ **Exécutez les tests avec couverture de code** et analysez le rapport pour repérer les classes ou méthodes non couvertes. Ajustez les tests pour augmenter la couverture de code si nécessaire.

Chapitre 3 - Module 5 - Évaluer la qualité des tests et éviter les tests fragiles

Objectifs

- ➔ Comprendre ce qu'est un test fragile et comment éviter ce type de test.
- ➔ Apprendre les bonnes pratiques pour écrire des tests robustes.

Qu'est-ce qu'un test fragile ?

- ➔ Un test est dit fragile s'il échoue souvent pour des raisons qui ne sont pas liées à des erreurs dans le code testé.
- ➔ Les tests fragiles augmentent les coûts de maintenance et diminuent la fiabilité des tests.

Principales causes des tests fragiles

- ➔ **Dépendances sur des ressources externes** : Les tests qui dépendent de bases de données, de réseaux, ou de fichiers peuvent échouer si ces ressources sont indisponibles.
- ➔ **Tests trop spécifiques** : Tester des valeurs trop précises, comme des dates ou des horaires, peut rendre le test instable.

➔ **Dépendance à l'ordre des tests** : Les tests qui modifient un état partagé ou qui dépendent d'un autre test peuvent échouer si l'ordre d'exécution change.

Bonnes pratiques pour éviter les tests fragiles

➔ **Utilisez des doubles de test** : Simulez les interactions avec des services externes.
➔ **Évitez les tests dépendants de l'heure** : Utilisez des dates fixes ou des fonctions pour contrôler le temps pendant les tests.
➔ **Assurez l'indépendance des tests** : Chaque test doit pouvoir s'exécuter indépendamment des autres.

Exemple de test fragile et comment le rendre robuste

➔ **Test fragile :** Test d'une méthode `generateReport` qui génère un fichier PDF dans un répertoire précis. Si le fichier est déjà présent ou verrouillé, le test échoue.
➔ **Solution :** Utilisez un double de test pour simuler l'écriture du fichier ou stockez le fichier dans un répertoire temporaire pour chaque test.

Chapitre 4 - Module 1 - Structurer et organiser ses tests

Objectifs

- ➔ Comprendre comment structurer les fichiers de test dans un projet.
- ➔ Apprendre les conventions de nommage et les meilleures pratiques pour garder des tests lisibles et maintenables.
- ➔ Apprendre à organiser les tests pour améliorer la lisibilité et la gestion de votre suite de tests.
- ➔ Utiliser des groupes et des annotations pour faciliter l'exécution de certains tests en fonction de leur importance ou de leur vitesse.

Organiser les fichiers de tests dans des répertoires

- ➔ **Convention** : Placer les tests dans un dossier `tests/` à la racine du projet.
- ➔ Créer des sous-dossiers qui correspondent à la structure des dossiers du code source pour une meilleure organisation.
- ➔ **Nommage des classes et méthodes de test**
  - ◆ **Conventions de nommage** : Utilisez un suffixe `Test` pour nommer les classes et fichiers de test, par exemple, `UserTest.php` pour tester `User.php`. Cela facilite la navigation dans les tests et permet aux outils de CI/CD de les reconnaître facilement.

  - ◆ Utiliser des noms de méthodes descriptifs, indiquant ce qui est testé et le résultat attendu.

- **Exemples de noms de méthodes :** `testAddReturnsCorrectSum`, `testDivideByZeroThrowsException`.

- **Un fichier par classe de test** : Chaque fichier de test devrait correspondre à une classe spécifique à tester. Par exemple, pour tester une classe `User`, créez un fichier `UserTest.php`.

➔ **Utiliser les annotations pour regrouper les tests**

- **Regrouper par fonctionnalités ou priorités** : Assigner des tests à des groupes peut aider à organiser une suite de tests importante et à exécuter seulement ceux dont vous avez besoin.

- **Annotation `@group`** : En ajoutant `@group` dans la docstring d'un test, vous pouvez créer des sous-ensembles pour exécuter uniquement certains tests.

- **Annotation `@depends` pour gérer les dépendances entre tests**
  - `@depends` permet à un test de s'appuyer sur les résultats d'un autre, ce qui est utile si vous souhaitez tester plusieurs étapes successives tout en gardant une indépendance.

➔ **Exécuter un groupe spécifique en utilisant `--group`** :
`vendor/bin/phpunit --group math`

➔ **Exécuter les tests d'un groupe spécifique :** En ligne de commande, utilisez **`phpunit --group critical`** pour n'exécuter que les tests critiques**.**

➔ **Séparer les tests unitaires des tests d'intégration**

- Les tests unitaires doivent tester des fonctions ou classes isolées et être rapides.

- Les tests d'intégration vérifient que des composants interagissent bien ensemble.

➔ **Avantages de structurer vos tests**

- **Lisibilité et maintenance** : Une organisation claire des tests simplifie la maintenance et rend la suite de tests plus intuitive.
- **Exécution ciblée** : La capacité de regrouper et de sélectionner certains tests améliore la gestion de grands projets.
- **Isolation et indépendance** : L'indépendance des tests permet une exécution plus fiable et réduit les échecs non pertinents.

Exemple d'utilisation de l'annotation @group

```
/**
 * @group critical
 */
public function testCriticalFeature()
{
    // Code du test pour une fonctionnalité critique
}
```

Exemple d'utilisation de l'annotation @depends

```
public function testAccountSetup()
{
    $account = new BankAccount(100);
    $this->assertEquals(100, $account->getBalance());
    return $account;
}

/**
 * @depends testAccountSetup
 */
public function testWithdraw(BankAccount $account)
{
```

```
    $account->withdraw(50);
    $this->assertEquals(50, $account->getBalance());
}
```

Exercice

Réorganisez le dossier `tests/` de votre projet pour le structurer comme le code source, et renommez les classes et méthodes de test en suivant les conventions.

Chapitre 4 - Module 2 - Bonnes pratiques et optimisation des tests

Objectifs

- ➔ Découvrir les meilleures pratiques pour écrire des tests unitaires.
- ➔ S'assurer que les tests restent rapides, lisibles, et utiles à long terme.
- ➔ Optimiser les tests pour les rendre plus rapides et éviter les erreurs fréquentes (tests fragiles).

Suivre la structure Arrange-Act-Assert pour une meilleure clarté

- ➔ **Arrange :** Préparer l'état initial ou les données nécessaires.
- ➔ **Act :** Exécuter l'action ou la méthode à tester.
- ➔ **Assert :** Vérifier que le résultat est celui attendu.

Exemple structuré de test pour `withdraw` dans une classe `BankAccount` :

```
01. public function testWithdrawDecreasesBalance()
02. {
03.     // Arrange : Préparer un compte avec un solde initial de 100
04.     $account = new BankAccount(100);
05.
06.     // Act : Effectuer un retrait de 50
07.     $account->withdraw(50);
08.
09.     // Assert : Vérifier que le solde est maintenant de 50
10.     $this->assertEquals(50, $account->getBalance());
11. }
```

Éviter les tests fragiles

- ➔ **Dépendances externes** : Évitez que les tests dépendent de ressources externes (comme une base de données ou une API).
- ➔ **Dépendance de l'ordre des tests** : Assurez-vous que chaque test peut s'exécuter indépendamment des autres.
- ➔ **Tests trop spécifiques** : Tester des valeurs précises (comme les dates actuelles) peut rendre les tests instables ; préférez des valeurs constantes ou simulées.

Optimiser les performances des tests

- ➔ **Utiliser des doubles de test (mocks, stubs)** : Les doubles de test simulent les dépendances, ce qui rend les tests plus rapides et plus fiables.
- ➔ **Exécution sélective** : Lancez seulement les tests importants (ex : les groupes critiques) en continu, et l'ensemble des tests pour les builds finaux.

Automatiser l'exécution des tests pour des retours rapides

- ➔ **Intégration continue (CI)** : Configurez un pipeline de CI pour exécuter automatiquement les tests à chaque push. Cela aide à détecter rapidement les erreurs, à maintenir la qualité du code, et à automatiser le suivi des nouvelles modifications.

[YAML - Exemple de workflow GitHub Actions pour PHPUnit :]

```yaml
name: Run Tests

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  phpunit:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

```

```
18.     - name: Set up PHP
19.       uses: shivammathur/setup-php@v2
20.       with:
21.         php-version: '8.0'
22.
23.     - name: Install dependencies
24.       run: composer install
25.
26.     - name: Run tests
27.       run: vendor/bin/phpunit --coverage-text
```

➔ **Écrire des tests atomiques**

- ◆ Chaque test doit être indépendant et se concentrer sur un seul comportement.
- ◆ Cela facilite le débogage et améliore la lisibilité.

➔ **Limiter les assertions par test**

- ◆ Un test idéal devrait contenir une seule assertion pour vérifier un comportement spécifique.

➔ **Prioriser la lisibilité**

- ◆ Utiliser des noms de méthode clairs et explicites.
- ◆ Ajouter des commentaires dans les tests pour expliquer les cas particuliers, si nécessaire.

➔ **Éviter les dépendances externes**

- ◆ Utiliser des mocks pour remplacer les bases de données, API, ou autres dépendances externes afin de garder les tests unitaires rapides et isolés.

➔ **Exécuter fréquemment les tests**

- ◆ Mettre en place des tests automatiques dans un système d'intégration continue (CI) pour détecter rapidement les régressions.

Exercice

Passez en revue un ensemble de tests existants et appliquez ces bonnes pratiques en simplifiant les tests, en réduisant les assertions multiples, et en utilisant des mocks si besoin.

- ➔ Organisez les tests existants dans votre projet en les regroupant par fonctionnalités ou en utilisant l'annotation `@group` pour exécuter des sous-ensembles de tests.
- ➔ Écrivez des tests clairs en appliquant la structure `Arrange-Act-Assert` pour rendre chaque test plus lisible.
- ➔ Configurez un pipeline CI pour automatiser l'exécution de votre suite de tests et vérifiez que le workflow fonctionne correctement à chaque modification.

Chapitre 4 - Module 3 - Exercice pratique non noté - Créer des tests unitaires pour une application PHP de liste de courses

Objectifs

- ➔ Mettre en pratique les concepts de tests unitaires dans un contexte de projet réel.
- ➔ Concevoir, organiser, et structurer les tests de manière professionnelle pour une application de liste de courses.

Présentation du Projet : Application de Liste de Courses

Dans cet exercice, vous travaillerez sur une application simple de liste de courses. Cette application permet à l'utilisateur de :

- ➔ Ajouter des articles à sa liste de courses.
- ➔ Mettre à jour et supprimer des articles.
- ➔ Recevoir une notification pour les articles marqués comme prioritaires.

Le projet contient deux composants principaux :

- ➔ Classe `ShoppingListManager` : Gère l'ajout, la mise à jour, et la suppression d'articles dans la liste de courses.
- ➔ Service `NotificationService` : Envoie des notifications, notamment lorsque des articles prioritaires sont ajoutés.

Structure de Projet et Détail des Fichiers

1. Fichier principal de l'application : `ShoppingListManager.php`

Ce fichier contient la logique principale de gestion de la liste de courses.

```
01. <?php
02. class ShoppingListManager
03. {
04.     private $notificationService;
05.
06.     public function __construct($notificationService)
07.     {
08.         $this->notificationService = $notificationService;
09.     }
10.
11.     public function addItem($item)
12.     {
13.         if (empty($item['name'])) {
14.             throw new InvalidArgumentException("Item name is required");
15.         }
16.
17.         // Code pour ajouter l'article à la liste
18.         if ($item['priority'] === 'high') {
19.             $this->notificationService->sendNotification("New high-priority item added to
    your shopping list");
20.         }
21.         return true;
```

```
    }

    public function updateItem($itemId, $itemData)
    {
        // Code pour mettre à jour un article de la liste
        return true;
    }

    public function deleteItem($itemId)
    {
        // Code pour supprimer l'article de la liste
        return true;
    }
}
```

➔ **`addItem($item)`** : Ajoute un article à la liste de courses. Si l'article est prioritaire, une notification est envoyée.

➔ **`updateItem($itemId, $itemData)`** : Met à jour les informations d'un article existant dans la liste.

➔ **`deleteItem($itemId)`** : Supprime un article de la liste.

2 . Service de notification : `NotificationService.php`

Ce fichier gère les notifications envoyées aux utilisateurs pour les articles prioritaires.

```
<?php
class NotificationService
{
    public function sendNotification($message)
    {
        // Code pour envoyer la notification
        return true;
    }
}
```

➔ **`sendNotification($message)`** : Envoie une notification avec un message spécifique. Dans un test, cette méthode sera simulée pour éviter l'envoi réel de notifications.

3. Fichier de tests : `ShoppingListManagerTest.php`

Ce fichier contient les tests unitaires pour `ShoppingListManager`, en utilisant PHPUnit.

```
01. <?php
02. use PHPUnit\Framework\TestCase;
03.
04. class ShoppingListManagerTest extends TestCase
05. {
06.     public function testAddItemWithoutNameThrowsException()
07.     {
08.         $mockNotificationService = $this->createMock(NotificationService::class);
09.         $shoppingListManager = new ShoppingListManager($mockNotificationService);
10.
11.         $this->expectException(InvalidArgumentException::class);
12.         $shoppingListManager->addItem(['name' => '']);
13.     }
14.
15.     public function testAddHighPriorityItemSendsNotification()
16.     {
17.         $mockNotificationService = $this->createMock(NotificationService::class);
18.         $mockNotificationService->expects($this->once())
19.             ->method('sendNotification')
20.             ->with("New high-priority item added to your shopping list");
21.
22.         $shoppingListManager = new ShoppingListManager($mockNotificationService);
23.
24.         $shoppingListManager->addItem(['name' => 'Urgent item', 'priority' => 'high']);
25.     }
26.
27.     public function testDeleteItem()
```

```
28.    {
29.        $mockNotificationService = $this->createMock(NotificationService::class);
30.        $shoppingListManager = new ShoppingListManager($mockNotificationService);
31.
32.        $this->assertTrue($shoppingListManager->deleteItem(1));
33.    }
34. }
```

➔ **`testAddItemWithoutNameThrowsException()`** : Vérifie que l'ajout d'un article sans nom génère une exception `InvalidArgumentException`.

➔ **`testAddHighPriorityItemSendsNotification()`** : Simule une notification pour les articles prioritaires et vérifie que le message attendu est bien envoyé.

➔ **`testDeleteItem()`** : Vérifie que la méthode `deleteItem()` supprime correctement un article de la liste.

4. Configuration PHPUnit : `phpunit.xml`

Ce fichier configure PHPUnit pour le projet.

```
01. <?xml version="1.0" encoding="UTF-8"?>
02. <phpunit bootstrap="vendor/autoload.php"
03.          colors="true"
04.          stopOnFailure="false">
05.     <testsuites>
06.         <testsuite name="Application Test Suite">
07.             <directory>./tests</directory>
08.         </testsuite>
09.     </testsuites>
10. </phpunit>
```

➔ **`bootstrap`** : Indique le fichier de chargement automatique pour inclure toutes les dépendances.

➔ **`colors`** : Active la colorisation des résultats pour plus de lisibilité.

➔ **`stopOnFailure`** : Continue l'exécution des tests même en cas d'échec d'un test.

Exécution du projet

Vous allez :

- ➔ Écrire des tests pour chaque méthode de `ShoppingListManager` en suivant les exemples fournis.
- ➔ Exécuter les tests avec PHPUnit pour s'assurer que tout fonctionne correctement.

Vous souhaitez en faire plus ?

Pour approfondir cet exercice :

- ➔ **Ajoutez une fonctionnalité** pour marquer un article comme acheté dans `ShoppingListManager`.
- ➔ **Créez un test unitaire** pour vérifier que cette fonctionnalité fonctionne correctement.

Code suggéré pour la nouvelle fonctionnalité :

Dans `ShoppingListManager.php` :

```
01. public function markItemAsPurchased($itemId)
02. {
03.     // Code pour marquer un article comme acheté
04.     return true;
05. }
```

Exemple de test pour la nouvelle fonctionnalité :

Dans `ShoppingListManagerTest.php` :

```
01. public function testMarkItemAsPurchased()
02. {
03.     $mockNotificationService = $this->createMock(NotificationService::class);
04.     $shoppingListManager = new ShoppingListManager($mockNotificationService);
```

```
05.
06.     $this->assertTrue($shoppingListManager->markItemAsPurchased(1));
07. }
```

Évaluation finale : Tests Unitaires avec PHPUnit en PHP

Note Maximale : 20 points.

Barème de points : 1 point par question.

Nombre de questions : 20.

1) **Qu'est-ce qu'un test unitaire ?**

   a) Un test pour valider un groupe de composants.

   b) Un test pour vérifier le bon fonctionnement d'une application entière.

   c) Un test pour vérifier le comportement isolé d'une fonction ou d'une méthode.

   d) Un test pour vérifier le rendu graphique d'une application.

2) **Pourquoi les tests unitaires sont-ils importants dans le développement logiciel ?**

   a) Ils augmentent la complexité du code.

   b) Ils permettent de vérifier rapidement si des modifications n'ont pas introduit de nouveaux bugs.

   c) Ils sont uniquement utilisés pour la documentation.

   d) Ils remplacent le travail de développement.

3) **Que fait l'annotation `@dataProvider` dans PHPUnit ?**

   a) Elle exécute plusieurs tests en parallèle.

   b) Elle fournit des données à un test pour exécuter plusieurs cas de test en une seule méthode.

   c) Elle documente le code testé.

   d) Elle vérifie automatiquement tous les cas de test d'un fichier.

4) **Quelle est la différence principale entre un mock et un stub dans les tests unitaires ?**

   a) Un mock vérifie les interactions, tandis qu'un stub fournit simplement des valeurs prédéfinies sans vérifier les interactions.

   b) Un stub vérifie les interactions, tandis qu'un mock fournit des valeurs prédéfinies.

   c) Un mock simule un service externe réel, tandis qu'un stub ne peut pas.

   d) Un mock est utilisé pour tester des interfaces graphiques, tandis qu'un stub teste des API.

5) **Dans quel cas est-il conseillé d'utiliser `setUp` et `tearDown` dans une classe de test ?**

   a) Pour configurer et nettoyer les ressources avant et après chaque test.

   b) Pour afficher des messages de log pendant les tests.

   c) Pour éviter d'écrire des assertions dans les tests.

   d) Pour exécuter des tests dans un ordre précis.

6) **Que retourne la méthode assertEquals si la valeur attendue et la valeur réelle ne correspondent pas ?**

   a) Elle retourne `null`.

   b) Elle lance une exception et marque le test comme échoué.

   c) Elle ne retourne rien.

   d) Elle continue le test en silence.

7) **Comment crée-t-on un mock en PHPUnit pour une classe Database ?**

   a) `$mock = new DatabaseMock();`

   b) `$mock = $this->createMock(Database::class);`

   c) `$mock = Database::mock();`

   d) `$mock = $this->mock(Database);`

8) **Quelle méthode est utilisée pour s'attendre à une exception spécifique dans un test ?**

   a) `throwException()`

   b) `setException()`

   c) `expectException()`

   d) `triggerException()`

9) **En utilisant un mock pour une méthode addUser, comment spécifie-t-on que cette méthode doit renvoyer true ?**

   a) `$mock->method('addUser')->returns(true);`

   b) `$mock->on('addUser')->returns(true);`

   c) `$mock->method('addUser')->willReturn(true);`

   d) `$mock->addUser(true);`

10) **Quelle est la sortie d'un test si aucune assertion n'a échoué ?**

   a) Le test échoue automatiquement.

   b) Le test est passé avec succès.

   c) Le test lance une alerte.

   d) Le test ne peut pas se terminer.

11) **Dans un test unitaire, pourquoi est-il déconseillé d'appeler directement des bases de données ou des services externes ?**

   a) Cela rend les tests trop rapides.

   b) Cela introduit des dépendances externes qui peuvent ralentir ou rendre les tests non fiables.

   c) Cela simplifie trop le code de test.

   d) Cela conduit à une couverture de code plus faible.

12) **Dans quel cas l'utilisation de `@dataProvider` est-elle avantageuse ?**

a) Lorsqu'on veut exécuter un test plusieurs fois avec différentes données sans copier le code du test.

b) Lorsqu'on souhaite valider uniquement les cas d'erreur d'une fonction.

c) Quand il est nécessaire de tester plusieurs méthodes simultanément.

d) Pour améliorer les performances d'un test unique en limitant son nombre d'assertions.

13) **Comment vérifier qu'une méthode `calculateDiscount` de la classe `Order` renvoie la bonne remise pour une commande de plus de 100 unités ?**

a) En écrivant un test qui vérifie que la méthode renvoie un tableau de réductions.

b) En appelant la méthode sans assertion et en vérifiant manuellement le résultat.

c) En passant une commande de plus de 100 unités dans le test et en utilisant une assertion assertEquals pour vérifier le pourcentage de remise attendu.

d) En ajoutant la méthode calculateDiscount directement dans le test et en exécutant le code.

14) **Pourquoi est-il important de structurer les tests unitaires dans des dossiers et de suivre les conventions de nommage ?**

a) Cela réduit le nombre de tests nécessaires.

b) Cela améliore la lisibilité et facilite la maintenance des tests.

c) Cela automatise l'exécution des tests.

d) Cela supprime les doublons dans le code source.

15) **Que doit-on tester pour vérifier le bon comportement de la méthode `deleteTask` dans une classe `TaskManager` ?**

a) Tester si une nouvelle tâche est créée après suppression.

b) Vérifier que la méthode deleteTask retourne une erreur lorsque la tâche n'existe pas.

c) Vérifier que la tâche n'est plus présente dans la liste des tâches après son appel.

d) Tester que toutes les tâches sont supprimées.

16) **Pourquoi et comment utiliser la méthode `tearDown` dans une classe de test ?**

a) Pour vérifier si les tests se déroulent dans l'ordre prévu.

b) Pour réinitialiser les dépendances ou ressources utilisées pendant chaque test, afin que chaque test démarre dans un état propre.

c) Pour accélérer l'exécution globale des tests.

d) Pour exécuter du code de nettoyage uniquement après le dernier test de la classe.

17) **Qu'est-ce que `expectExceptionMessage` et pourquoi est-il utile ?**

a) C'est une méthode qui enregistre tous les messages d'exception pendant les tests pour une analyse ultérieure.

b) C'est une méthode qui permet de vérifier que le message d'une exception levée correspond à celui attendu, afin de valider le type d'erreur spécifique.

c) Une méthode pour créer des logs de chaque exception levée dans les tests.

d) Une méthode qui permet d'éviter de lever des exceptions lors des tests.

18) **Que signifie "isoler les tests" et pourquoi est-ce important dans les tests unitaires ?**

a) Cela signifie regrouper plusieurs tests dans un même fichier pour simplifier le code.

b) Cela signifie exécuter tous les tests en même temps pour gagner du temps.

c) Cela signifie exécuter chaque test indépendamment des autres, pour éviter que les tests ne dépendent de l'état ou des données des autres tests.

d) Cela signifie inclure tous les tests dans la base de code principale.

19) **Comment vérifier qu'une exception `DivisionByZeroError` est bien levée par une méthode `divide` ?**

a) En exécutant la méthode et en ajoutant un message de log.

b) En utilisant assertEquals pour vérifier si DivisionByZeroError est levée.

c) En ajoutant expectException(DivisionByZeroError::class) avant d'exécuter la méthode divide.

d) En exécutant la méthode sans paramètre pour voir si une erreur est générée.

20) **Quelles sont deux bonnes pratiques pour écrire des tests unitaires en PHP avec PHPUnit ?**

a) Utiliser autant d'assertions que possible dans un même test et éviter les mocks pour simplifier le code.

b) Écrire des tests indépendants les uns des autres et limiter le nombre d'assertions par test pour cibler des comportements spécifiques.

c) Appeler des services externes dans les tests et créer des fichiers de test sans structure définie.

d) Utiliser des dépendances externes comme des API pour rendre les tests plus réalistes et regrouper tous les tests dans un même fichier.