

I. PHPUnit

1. Installation
 01. alias
2. Premier test
 01. Quelques règles
 02. Lancer le test
3. Ajouter un test
4. Configuration de PHPUnit
5. Auto chargement des classes
 01. Première approche
 02. Meilleure approche
6. Tester la classe Queue
 01. Définition de dépendances
 02. Fixtures
7. Fixtures globales
 - α. Exemple
8. Tester une exception
 01. Exception personnalisée
9. Les providers
 01. Exemple
10. Les assertions les plus utiles
11. Exercices
 01. EmailValidator
 02. PrimeChecker
 03. PasswordStrength
 04. Autres tests
12. Test doubles, les Stubs
 01. Un exemple simple
 02. Limitation
 03. Un exemple plus complexe
 04. Configurer un stub
 05. Tester différents arguments
 06. Exercices sur les stubs
13. Tester des méthodes privées
14. Tester une base de données
 01. Un exemple
15. TDD
 01. Principe du TDD
 02. Avantages du TDD
 03. Un exemple simple
 - α. Première étape
 - β. La suite
 04. Exercices
 - α. Un validateur de mot de passe
 - β. Un convertisseur de chiffres romains en décimal

PHPUnit

Installation

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

alias

Sur MacOS et Linux `alias phpunit="./vendor/bin/phpunit"`

Sur Windows `doskey phpunit=./vendor/bin/phpunit`

Premier test

Dans un dossier `tests`, créer un fichier `SimpleTest.php`

```
<?php

use PHPUnit\Framework\TestCase;

class SimpleTest extends TestCase
{
    public function testAddingTwoPlusOneIsThree()
    {
        $expected = 3;
        $actual   = 2 + 1;
        $this->assertEquals($expected, $actual);
    }
}
```

Quelques règles

- Les classes de test héritent de `PHPUnit\Framework\TestCase`
- Le nom de la classe et de son fichier sont identiques (pour faciliter l'auto chargement)
- Le nom de la classe doit se terminer par `Test`
- Le nom des méthodes de test doit commencer par `test`
- Les méthodes de test doivent contenir au moins une assertion

Lancer le test

```
./vendor/bin/phpunit/ tests/SimpleTest.php
```

Ajouter un test

Créer une classe `StupidTest` avec une méthode `testThatTrueIsTrue` .

Lancer tous les tests avec.

```
./vendor/bin/phpunit/ tests
```

Configuration de PHPUnit

Créer un fichier `phpunit.xml`

```
<?xml version="1.0" encoding="UTF-8" ?>
<phpunit colors="true">
  <testsuites>
    <testsuite name="my application test suites">
      <directory>./tests</directory>
    </testsuite>
  </testsuites>
</phpunit>
```

On peut désormais lancer tous les tests du dossier `tests` avec la commande suivante :

```
./vendor/bin/phpunit/
```

Auto chargement des classes

Première approche

Inclure les classe requises dans la classe de test

```
require 'src/Queue.php';
```

ou mieux encore inclure l'autoloader de Composer

```
require 'vendor/autoload.php';
```

Meilleure approche

Indiquer à PHPUnit l'autoloader à utiliser pour chacune des classes de test

```
./vendor/bin/phpunit --bootstrap='vendor/autoload.php'
```

On peut simplifier encore en déclarant le fichier bootstrap dans phpunit.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<phpunit colors="true"
    bootstrap="vendor/autoload.php">
    <testsuites>
        <testsuite name="my application test suites">
            <directory>./tests</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

Tester la classe Queue

```
use PHPUnit\Framework\TestCase;
use Smaloron\App\Queue;

class QueueTest extends TestCase
{
    final public function testNewQueueIsEmpty(): void{
        $queue = new Queue();
        $this->assertTrue($queue->isEmpty(), "La file est vide");
    }
    final public function testQueueOperations(): void{
        $queue = new Queue();
        $queue->enqueue(1);
        $queue->enqueue(2);
        $queue->enqueue(3);
        $this->assertEquals(3, $queue->getSize(), "La taille de la file doit être 3");
        $this->assertEquals(1, $queue->dequeue(), "Le premier élément dépilé doit être 1");
        $this->assertEquals(2, $queue->getSize(), "La taille de la file doit être 2");
    }
}
```

Définition de dépendances

Les deux tests ont besoin d'une instance de Queue, on peut donc établir une dépendance du second test vers le premier.

Pour cela, on commence par retourner l'objet dans le premier test

```
final public function testNewQueueIsEmpty(): Queue{
    $queue = new Queue();
    $this->assertTrue($queue->isEmpty(), "La file est vide");
    return $queue;
}
```

Puis dans le second test, on indique la dépendance dans un docblock avec l'annotation `@depends`. La dépendance sera injectée lors de l'appel de la méthode, il faut donc la déclarer en argument

```
/**
 * @depends testNewQueueIsEmpty
 * @param Queue $queue
 * @return void
 */
final public function testQueueOperations(Queue $queue): void{
    $queue->enqueue(1);
    $queue->enqueue(2);
    $queue->enqueue(3);
    $this->assertEquals(3, $queue->getSize(), "La taille de la file doit être 3");
    $this->assertEquals(1, $queue->dequeue(), "Le premier élément défilé doit être 1");
    $this->assertEquals(2, $queue->getSize(), "La taille de la file doit être 2");
}
```

Fixtures

Les fixtures sont une alternative aux dépendances. Il s'agit de définir dans une méthode `setUp` le code qui sera exécuté avant chaque test. La méthode `tearDown` permet de faire le ménage et de s'assurer que chaque test démarre dans un état connu et maîtrisé.

Dans ce cas précis la méthode `tearDown` n'est pas indispensable, elle est implémentée uniquement en vue de démonstration.

```
<?php
use PHPUnit\Framework\TestCase;
use Smaloron\App\Queue;

class QueueTest extends TestCase
{
    protected Queue $queue;
    protected function setUp(): void{
        $this->queue = new Queue();
    }
    final public function testNewQueueIsEmpty(): void{
        $this->assertTrue($this->queue->isEmpty(), "La file est vide");
    }

    final public function testQueueOperations(): void{
        $this->queue->enqueue(1);
        $this->queue->enqueue(2);
        $this->queue->enqueue(3);
        $this->assertEquals(3, $this->queue->getSize(), "La taille de la file doit être 3");
        $this->assertEquals(1, $this->queue->dequeue(), "Le premier élément défilé doit être 1");
        $this->assertEquals(2, $this->queue->getSize(), "La taille de la file doit être 2");
    }

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

Fixtures globales

Les méthodes `setUpBeforeClass` et `tearDownBeforeClass` peuvent contenir du code qui sera exécuté une seule fois juste après l'instanciation de la classe de test. Cela permet d'initialiser le test avec des informations partagées pour l'ensemble des méthodes de test

Example

On veut tester la classe suivante

```
class UserRepository
{
    private $pdo;

    public function __construct(PDO $pdo)
    {
        $this->pdo = $pdo;
    }

    public function create($name, $email)
    {
        $stmt = $this->pdo->prepare('INSERT INTO users (name, email) VALUES (?, ?)');
        $stmt->execute([$name, $email]);
        $id = $this->pdo->lastInsertId();
        return ['id' => $id, 'name' => $name, 'email' => $email];
    }

    public function getById($id)
    {
        $stmt = $this->pdo->prepare('SELECT * FROM users WHERE id = ?');
        $stmt->execute([$id]);
        return $stmt->fetch(PDO::FETCH_ASSOC);
    }
}
```

Pour ce faire, on ne va pas utiliser la base de données en production ni même installer un SGBDR. Le mieux est de travailler avec une base de données temporaire en mémoire vive. SQLite est parfait pour cela

```
<?php

use PHPUnit\Framework\TestCase;
use PDO;

class UserRepositoryTest extends TestCase
{
    private static $pdo;
    private $userRepository;

    public static function setUpBeforeClass(): void
    {
```

```

        self::$pdo = new PDO('sqlite::memory:');
        self::$pdo->exec('CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT, email TEXT)');
    }

    protected function setUp(): void
    {
        $this->userRepository = new UserRepository(self::$pdo);
        self::$pdo->exec('DELETE FROM users');
    }

    public function testCreateUser()
    {
        $user = $this->userRepository->create('John Doe', 'john@example.com');

        $this->assertIsInt($user['id']);
        $this->assertEquals('John Doe', $user['name']);
        $this->assertEquals('john@example.com', $user['email']);
    }

    public function testGetUserById()
    {
        $this->userRepository->create('Jane Smith', 'jane@example.com');
        $user = $this->userRepository->getById(1);

        $this->assertEquals('Jane Smith', $user['name']);
        $this->assertEquals('jane@example.com', $user['email']);
    }
}

```

Tester une exception

Nous souhaitons tester la classe suivante

```

class Calculator
{
    public function divide($a, $b)
    {
        if ($b == 0) {
            throw new InvalidArgumentException("Cannot divide by zero");
        }
        return $a / $b;
    }
}

```

Voici le code du test

```

class CalculatorTest extends TestCase
{
    public function testDivideByZeroThrowsException()
    {
        $calculator = new Calculator();
    }
}

```

```

        $this->expectException(InvalidArgumentException::class);
        $this->expectExceptionMessage("Cannot divide by zero");

        $calculator->divide(10, 0);
    }
}

```

Exception personnalisée

```

class InsufficientFundsException extends Exception
{
    protected $balance;
    protected $withdrawAmount;

    public function __construct(
        string $message,
        protected int $balance,
        protected int $withdrawAmount)
    {
        parent::__construct($message);
    }

    public function getBalance(): int
    {
        return $this->balance;
    }

    public function getWithdrawAmount():int
    {
        return $this->withdrawAmount;
    }
}

```

La class BankAccount qui utilise la nouvelle exception

```

class BankAccount
{
    private int $balance;

    public function __construct(int $initialBalance)
    {
        $this->balance = $initialBalance;
    }

    public function withdraw(int $amount): int
    {
        if ($amount > $this->balance) {
            throw new InsufficientFundsException(
                "Insufficient funds for withdrawal",
                $this->balance,
                $amount
            );
        }
        $this->balance -= $amount;
    }
}

```



```
        return $this->balance;
    }
}
```

Et enfin le test

```
class BankAccountTest extends TestCase
{
    public function testWithdrawThrowsExceptionWhenInsufficientFunds()
    {
        $account = new BankAccount(100);

        $this->expectException(InsufficientFundsException::class);
        $this->expectExceptionMessage("Insufficient funds for withdrawal");

        $account->withdraw(150);
    }

    public function testExceptionProvidesCorrectBalanceAndWithdrawAmount()
    {
        $account = new BankAccount(50);

        try {
            $account->withdraw(100);
            $this->fail("Expected InsufficientFundsException was not thrown");
        } catch (InsufficientFundsException $e) {
            $this->assertEquals(50, $e->getBalance());
            $this->assertEquals(100, $e->getWithdrawAmount());
        }
    }
}
```

Les providers

Un provider est une méthode qui fournit une suite de données à une méthode de test. On peut ainsi limiter la duplication du code et rendre les tests plus digestes et plus faciles à maintenir.

- La méthode provider retourne un tableau de tableau, chacun de ces tableaux internes sera passé en argument à la fonction de test.
- La méthode de test déclare qu'elle utilise le provider avec une annotation ou un attribut `DataProvider`
- La signature de la méthode de test doit inclure les arguments définis dans le provider

Exemple

```
class Calculator
{
    public function add(int $a, int $b): int
    {
        return $a + $b;
    }
}

class CalculatorTest extends TestCase
{
    #[DataProvider('additionProvider')]
    public function testAddition(int $a, int $b, int $expected): void
    {
        $calculator = new Calculator();
        $result = $calculator->add($a, $b);
        $this->assertSame($expected, $result);
    }

    public static function additionProvider(): array
    {
        return [
            [1, 2, 3],
            [0, 0, 0],
            [-1, 1, 0],
            [100, 200, 300],
            [-5, -7, -12]
        ];
    }
}
```

Les assertions les plus utiles

Les deux assertions absolument indispensables sont `assertTrue` et `assertException` mais PHPUnit en fournit bien d'autres qui peuvent simplifier le code de nos tests.

Méthode	Description
assertEquals	Vérifie si deux valeurs sont égales sans prendre en compte le type
assertSame	Vérifie l'identité des valeurs et des types
assertArrayHasKey	Vérifie la présence d'une clef dans un tableau
assertCount	Vérifie le nombre d'éléments dans un iterable
assertGreaterThan	
assertLessThan	
assertNull	
assertInstanceOf	
assertFileExists	
assertStringStartsWith	
assertStringMatchesFormat	Vérifie qu'une chaîne correspond à une expression régulière

Exercices

Écrire les tests pour les classes suivantes.

EmailValidator

```
namespace Smaloron\App\Validators;
class EmailValidator {
    public function isValid(string $email):bool {
        return filter_var($email, FILTER_VALIDATE_EMAIL) !== false;
    }
}
```

PrimeChecker

```
namespace Smaloron\App\Utils;
class PrimeChecker {
    public function isPrime(int $number): bool {
        if ($number < 2) return false;
        for ($i = 2; $i <= sqrt($number); $i++) {
            if ($number % $i == 0) return false;
        }
        return true;
    }
}
```

PasswordStrength

```
namespace Smaloron\App\Validators;
class PasswordStrengthChecker {
    public function checkStrength(string $password): int {
        $score = 0;
        if (strlen($password) >= 8) $score++;
        if (preg_match('/[A-Z]/', $password)) $score++;
        if (preg_match('/[a-z]/', $password)) $score++;
        if (preg_match('/[0-9]/', $password)) $score++;
        if (preg_match('/^[A-Za-z0-9]/', $password)) $score++;
        return $score;
    }
}
```

Autres tests

- Tester la classe QueryBuilder
- Tester la classe FormBuilder

Test doubles, les Stubs

La plupart des classes possèdent des dépendances, d'autres classes qui fournissent des services, évitant ainsi la duplication du code et un trop grand nombre de responsabilités incombant à une seule classe.

Tester une classe et ses dépendances est possible et judicieux, cependant on quitte alors le domaine du test unitaire pour entrer dans celui du test d'intégration.

Mais alors, comment tester une classe en l'isolant de ses dépendances ?

Il faut remplacer la dépendance par une "fausse" classe de même type dont nous pourrions fixer le comportement. PHPUnit dispose d'un tel système

Un exemple simple

Ici la classe Calculator dépend d'une class RandomNumberGenerator injectée dans le constructeur

```

class Calculator
{
    private $randomGenerator;

    public function __construct(RandomNumberGenerator $randomGenerator)
    {
        $this->randomGenerator = $randomGenerator;
    }

    public function addRandomNumber($base)
    {
        $randomNumber = $this->randomGenerator->generate();
        return $base + $randomNumber;
    }
}

class CalculatorTest extends TestCase
{
    public function testAddRandomNumber()
    {
        // Créé un simulacre de la classe RandomNumberGenerator
        $stubRandomGenerator = $this->createStub(RandomNumberGenerator::class);

        // Configure la fausse classe pour que l'appel à la méthode generate retourne une va
        $stubRandomGenerator->method('generate')
            ->willReturn(5);

        // Injection de la fausse classe dans le constructeur
        $calculator = new Calculator($stubRandomGenerator);

        // Exécution du code
        $result = $calculator->addRandomNumber(10);

        // Assertion
        $this->assertEquals(15, $result);
    }
}

```

Cette pratique fonctionne également avec les interfaces, il est même recommandé de simuler des interfaces plutôt que des classes concrètes autant que faire se peut.

Limitation

- Les méthodes possédant les attributs `private` ou `final` ne peuvent être simulées, le code d'origine sera donc exécuté.
- Les méthodes possédant l'attribut `static` ne peuvent être simulée et tout appel de ces méthodes depuis une classe simulée retournera une exception.
- Toute classe qui ne peut être étendue (`final` ou `readonly`) ne peut être simulée

Un exemple plus complexe

Ici, il s'agit de simuler une classe dont une méthode retourne une instance d'une autre classe. En l'occurrence, la méthode `prepare` de l'objet `PDO` retourne un objet `PDOStatement`.

```
class UserDAO
{
    private $pdo;

    public function __construct(PDO $pdo)
    {
        $this->pdo = $pdo;
    }

    public function getUserById($id)
    {
        $stmt = $this->pdo->prepare("SELECT * FROM users WHERE id = :id");
        $stmt->execute(['id' => $id]);
        return $stmt->fetch(PDO::FETCH_ASSOC);
    }
}
```

Voici le test

```
class UserDAOTest extends TestCase
{
    public function testGetUserById()
    {
        // Création des simulacres
        $pdoStub = $this->createStub(PDO::class);
        $pdoStatementStub = $this->createStub(PDOStatement::class);

        // Configuration de PDO
        $pdoStub->method('prepare')
            ->willReturn($pdoStatementStub);

        // configuration de PDOStatement
        $pdoStatementStub->method('execute')
            ->willReturn(true);
        $pdoStatementStub->method('fetch')
            ->willReturn(['id' => 1, 'name' => 'John Doe']);

        // Instanciation du DAO
        $userDAO = new UserDAO($pdoStub);

        // Appel de la méthode à tester
        $result = $userDAO->getUserById(1);

        // Assertion
        $this->assertEquals(['id' => 1, 'name' => 'John Doe'], $result);
    }
}
```

Configurer un stub

Définit un ensemble de méthodes et les valeurs de retour correspondantes en une seule fois. La méthode `createConfiguredStub` admet deux arguments :

- la classe à simuler
- Un tableau associatif dont les clefs sont les noms des méthodes et les valeurs les retours des appels de ces méthodes

La classe à tester

```
class UserService
{
    private $repository;

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

    public function getUserById($id)
    {
        return $this->repository->findById($id);
    }

    public function createUser($userData)
    {
        return $this->repository->create($userData);
    }

    public function deleteUser($id)
    {
        return $this->repository->delete($id);
    }
}
```

Le test

```
class UserServiceTest extends TestCase
{
    public function testUserServiceMethods()
    {
        $user = ['id' => 1, 'name' => 'John Doe', 'email' => 'john@example.com'];

        $repositoryStub = $this->createConfiguredStub(
            UserRepository::class,
            [
                'findById' => $user,
                'create' => true,
                'delete' => true
            ]
        );

        $userService = new UserService($repositoryStub);
    }
}
```

```

        // Test getUserById
        $this->assertEquals($user, $userService->getUserById(1));

        // Test createUser
        $this->assertTrue($userService->createUser($user));

        // Test deleteUser
        $this->assertTrue($userService->deleteUser(1));
    }
}

```

Tester différents arguments

La méthode `willReturnCallback` admet en argument un callable (souvent une fonction anonyme). Cela permet de faire varier la valeur de retour en fonction d'un argument passé à la fonction.

La classe à tester

```

class WeatherService
{
    private $apiClient;

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

    public function getTemperature($city)
    {
        $data = $this->apiClient->fetchWeatherData($city);
        return $data['temperature'];
    }
}

```

Le test

```

class WeatherServiceTest extends TestCase
{
    public function testGetTemperature()
    {
        // Create a stub for the ApiClient
        $apiClientStub = $this->createStub(ApiClient::class);

        // Configure the stub
        $apiClientStub->method('fetchWeatherData')
            ->willReturnCallback(function($city) {
                if ($city === 'London') {
                    return ['temperature' => 15];
                } elseif ($city === 'Paris') {
                    return ['temperature' => 20];
                } else {
                    throw new \Exception('City not found');
                }
            });
    }
}

```



```

    });

    // Create an instance of WeatherService with the stub
    $weatherService = new WeatherService($apiClientStub);

    // Test the getTemperature method
    $this->assertEquals(15, $weatherService->getTemperature('London'));
    $this->assertEquals(20, $weatherService->getTemperature('Paris'));

    // Test for an exception
    $this->expectException(\Exception::class);
    $weatherService->getTemperature('Berlin');
}
}

```

Exercices sur les stubs

- Tester quelques méthodes de votre classe DAO

Tester des méthodes privées

Généralement, on ne teste que les méthodes publiques. Cependant, avec l'API Reflection, il est possible d'accéder aux éléments privés ou protégés.

La classe

```

class PasswordManager
{
    // Private method to hash the password
    private function hashPassword($password)
    {
        return password_hash($password, PASSWORD_DEFAULT);
    }

    // Public method to create a user with a hashed password
    public function createUser($username, $password)
    {
        $hashedPassword = $this->hashPassword($password);
        // Here you would typically save the username and hashed password to a database
        return ['username' => $username, 'password' => $hashedPassword];
    }
}

```

Le test

```

class PasswordManagerTest extends TestCase
{
    public function testHashPassword()
    {
        // Une instance de PasswordManager
        $passwordManager = new PasswordManager();
    }
}

```

```

// L'API Reflection permet d'accéder aux éléments privés de la classe
$reflection = new ReflectionClass(PasswordManager::class);
$method = $reflection->getMethod('hashPassword');
$method->setAccessible(true);

// Hachage
$password = 'securePassword123';
$hashedPassword = $method->invokeArgs($passwordManager, [$password]);

// Assertions
$this->assertNotEmpty($hashedPassword);
$this->assertTrue(password_verify($password, $hashedPassword));
}

```

Tester une base de données

Il va sans dire que les tests de la base de données ne doivent pas se faire sur le serveur de production. En outre, il vaut mieux isoler ces tests, car ils sont intrinsèquement bien plus lent que les autres tests et doivent être joués plus rarement.

```

<?xml version="1.0" encoding="UTF-8" ?>
<phpunit colors="true"
bootstrap="vendor/autoload.php">
  <testsuites>
    <testsuite name="app-tests">
      <directory>./tests</directory>
    </testsuite>
    <testsuite name="db-tests">
      <directory>./database-tests</directory>
    </testsuite>
  </testsuites>
</phpunit>

```

```
./vendor/bin/phpunit --testsuite="db-tests"
```

Ensuite, SQLite remplacera avantageusement le serveur de base de données. Ce SGBDR est léger et peut travailler en mémoire vive ce qui nous assure de maîtriser l'état des données en permettant la création d'une nouvelle structure de données à chaque test.

Un exemple

La classe à tester

```

namespace Smaloron\App;
use PDO;
class UserRepository
{

```

```

private PDO $pdo;

public function __construct(PDO $pdo)
{
    $this->pdo = $pdo;
}

public function createUser(string $name, string $email): int
{
    $stmt = $this->pdo->prepare("INSERT INTO users (name, email) VALUES (?, ?)");
    $stmt->execute([$name, $email]);
    return $this->pdo->lastInsertId();
}

public function getUserById(int $id): ?array
{
    $stmt = $this->pdo->prepare("SELECT * FROM users WHERE id = ?");
    $stmt->execute([$id]);
    $user = $stmt->fetch(PDO::FETCH_ASSOC);
    return $user ?: null;
}
}

```

Le test

```

use PHPUnit\Framework\TestCase;
use Smaloron\App\UserRepository;

class UserRepositoryTest extends TestCase
{
    private PDO $pdo;
    private UserRepository $userRepository;

    protected function setUp(): void
    {
        $this->pdo = new PDO('sqlite::memory:');
        $this->pdo->exec("CREATE TABLE users (id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT)");
        $this->userRepository = new UserRepository($this->pdo);
    }

    public function testCreateUser():void
    {
        $userId = $this->userRepository->createUser('John Doe', 'john@example.com');

        $this->assertIsInt($userId);
        $this->assertGreaterThan(0, $userId);

        $user = $this->userRepository->getUserById($userId);
        $this->assertIsArray($user);
        $this->assertEquals('John Doe', $user['name']);
        $this->assertEquals('john@example.com', $user['email']);
    }

    public function testGetUserById():void
    {
        $stmt = $this->pdo->prepare("INSERT INTO users (name, email) VALUES (?, ?)");
        $stmt->execute(['Jane Doe', 'jane@example.com']);
    }
}

```

```
$userId = $this->pdo->lastInsertId();

$user = $this->userRepository->getUserById($userId);

$this->assertIsArray($user);
$this->assertEquals($userId, $user['id']);
$this->assertEquals('Jane Doe', $user['name']);
$this->assertEquals('jane@example.com', $user['email']);
}

public function testGetUserByIdReturnsNullForNonexistentUser():void
{
    $user = $this->userRepository->getUserById(999);
    $this->assertNull($user);
}
}
```

TDD

Le TDD (Test Driven Development) est une méthode de développement qui consiste à écrire les tests avant d'écrire le code. Cette approche vise à créer un code plus robuste, maintenable et aligné sur les besoins fonctionnels.

Principe du TDD

Le TDD fonctionne par cycles, suivant ces étapes :

1. Écrire un test unitaire basé sur une hypothèse
2. Vérifier que le test échoue (car le code n'existe pas encore)
3. Écrire le code minimal pour faire passer le test
4. Vérifier que le test passe
5. Refactoriser le code tout en maintenant le succès du test

Avantages du TDD

Le Test Driven Development (TDD) offre plusieurs avantages significatifs par rapport aux méthodologies traditionnelles de développement :

1. Qualité du code améliorée : Le TDD encourage la production d'un code plus propre, plus fiable et mieux conçu. En écrivant les tests avant le code, les développeurs sont amenés à réfléchir davantage à la conception et à l'utilisation du code, ce qui réduit le nombre de bugs.
2. Meilleure maintenabilité : Le code produit par TDD est généralement plus facile à maintenir et à modifier, grâce à un couplage lâche et une cohésion forte. Les tests servent également de documentation vivante, facilitant la compréhension et la maintenance du code.
3. Productivité accrue : Bien que le TDD puisse sembler plus lent au début, il permet de gagner du temps à long terme en réduisant le temps passé à corriger des bugs et en facilitant les modifications ultérieures. Les développeurs peuvent refactoriser le code avec plus de confiance, sachant que les tests existants détecteront les régressions.

4. Conception orientée utilisateur : Le TDD encourage les développeurs à se concentrer sur les besoins fonctionnels et le comportement souhaité du logiciel avant de commencer à coder, ce qui conduit à des solutions mieux adaptées aux besoins des utilisateurs.
5. Détection précoce des problèmes : En testant continuellement pendant le développement, les problèmes sont identifiés et corrigés plus tôt dans le cycle de développement, ce qui réduit les coûts et le temps nécessaires pour les résoudre.
6. Automatisation facilitée : Le TDD produit naturellement une suite de tests unitaires automatisés, ce qui simplifie l'intégration continue et le déploiement continu.
7. Réduction du sur-engineering : En se concentrant sur l'écriture du code minimum nécessaire pour passer les tests, le TDD aide à éviter la création de fonctionnalités superflues.

Un exemple simple

L'objectif est de créer une classe `Calculator` fournissant une méthode `add` qui fait la somme de deux entiers passés en argument. Le code est trivial, mais il servira d'exemple pour la démonstration.

Première étape

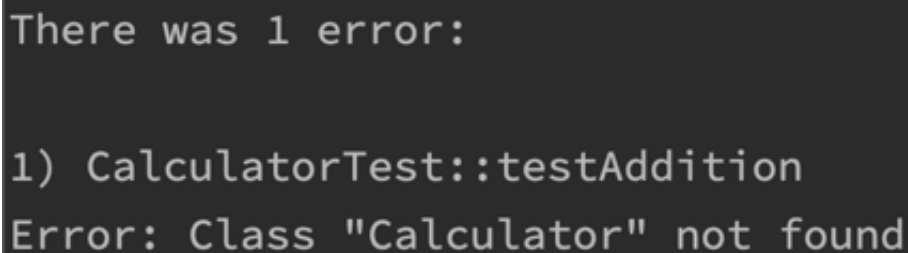
1. On écrit un test pour le cas nominal

```
<?php
declare(strict_types=1);

use PHPUnit\Framework\TestCase;
use Smaloron\App\Calculator;

class CalculatorTest extends TestCase {
    public function testAddition() {
        $calculator = new Calculator();
        $this->assertEquals(4, $calculator->add(2, 2));
    }
}
```

1. On lance le test qui doit échouer.



```
There was 1 error:

1) CalculatorTest::testAddition
Error: Class "Calculator" not found
```

3. On écrit le code minimum pour que le test réussisse.

```
class Calculator {
    public function add(int $a, int $b): int {
```

```
        return $a + $b;
    }
}
```

1. On relance le test qui cette fois-ci est un succès.

La suite

On peut continuer en testant les cas à la marge. Par exemple si l'on passe null en argument, on doit obtenir une exception `TypeError`.

```
public function testNullArgumentsShouldGiveTypeError()
{
    $this->expectException(TypeError::class);
    $calculator = new Calculator();
    $calculator->add(null, null);
}
```

Exercices

Un validateur de mot de passe

Retourne false si une des règles de validation est fausse :

- Au moins 8 caractères
- Au moins une majuscule
- Au moins une minuscule
- Au moins un caractère spécial

Pour chacune des règles :

- Rédiger un test
- Lancer le test
- Écrire le code qui fera passer le test

Un convertisseur de chiffres romains en décimal