



# Programmation orientée objet en PHP

[Accueil](#) ► [Mes cours](#) ► [Développement logiciel](#) ► [POO PHP](#) ► [Les tests avec PHPUnit](#) ► [Premier test](#)

## Premier test

**PHPUnit** est une bibliothèque qui va nous permettre de tester nos composants logiciels. Nous pourrons ainsi vérifier le bon fonctionnement de notre application à plusieurs niveaux :

- au niveau du composant lui-même : on parle ici de test unitaire, on essaie de tester la plus petite unité de code possible (souvent une fonction)
- au niveau de plusieurs composants : on vérifie ici que l'intégration de nouveaux composants n'entraîne pas un dysfonctionnement des autres composants : on parle ici de test d'intégration
- au niveau de l'application elle-même : on vérifie le bon fonctionnement d'une fonctionnalité entière : on parle ici de test fonctionnel

Les tests permettent aussi de mettre en évidence des problèmes en terme de performance, de robustesse, de fiabilité, ou encore de vulnérabilité.

L'installation de **PHPUnit** sera assurée par un outil de gestion de dépendances appelé **Composer**. Pour le dire vite, Composer va nous permettre de télécharger les bibliothèques dont nous aurons besoin dans notre application. Ces bibliothèques seront renseignées dans un fichier JSON. L'installation de Composer peut se faire de plusieurs façons, soit sur votre système, soit dans votre projet sous forme d'une archive phar. Dans notre projet nous utiliserons l'archive phar.

Nous créons un projet TestProject dans notre répertoire personnel, et nous mettons à la racine de notre projet l'archive composer.phar téléchargée à partir du site :



Dans un terminal, nous nous plaçons dans le répertoire de notre projet et exécutons la commande *php composer.php*.

```
Terminal
Fichier Edition Affichage Rechercher Terminal Aide
patrice@patrice-laptop:~/TestProject$ php composer.phar

Composer version 1.0.0-dev (c33c5196b19c77ea89b05a81c573d88543d3a93d) 2014-10-15 13:11:07

Usage:
  [options] command [arguments]

Options:
  --help                -h Display this help message.
  --quiet               -q Do not output any message.
  --verbose             -v|vv|vvv Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug
  --version             -V Display this application version.
  --ansi               -A Force ANSI output.
  --no-ansi             -A Disable ANSI output.
  --no-interaction      -n Do not ask any interactive question.
  --profile             -p Display timing and memory usage information
  --working-dir         -d If specified, use the given directory as working directory.

Available commands:
  about                Short information about Composer
  archive              Create an archive of this composer package
  browse              Opens the package's repository URL or homepage in your browser.
  clear-cache          Clears composer's internal package cache.
  config              Set config options
  create-project       Create new project from a package into given directory.
  depends             Shows which packages depend on the given package
  diagnose            Diagnoses the system to identify common errors.
  dump-autoload        Dumps the autoloader
  dumpautoload         Dumps the autoloader
  global              Allows running commands in the global composer dir ($COMPOSER_HOME).
  help                Displays help for a command
  home                Opens the package's repository URL or homepage in your browser.
  init                Creates a basic composer.json file in current directory.
  install             Installs the project dependencies from the composer.lock file if present, or falls back on the composer.json.
  licenses            Show information about licenses of dependencies
  list                Lists commands
  remove              Removes a package from the require or require-dev
  require             Adds required packages to your composer.json and installs them
  run-script          Run the scripts defined in composer.json.
  search              Search for packages
  self-update          Updates composer.phar to the latest version.
  selfupdate          Updates composer.phar to the latest version.
  show               Show information about packages
  status             Show a list of locally modified packages
  update             Updates your dependencies to the latest version according to composer.json, and updates the composer.lock file.
  validate            Validates a composer.json

patrice@patrice-laptop:~/TestProject$
```

Nous obtenons ainsi la liste des commandes de Composer.

Nous éditons à la racine de notre projet un fichier composer.json qui indiquera toutes les dépendances que devra

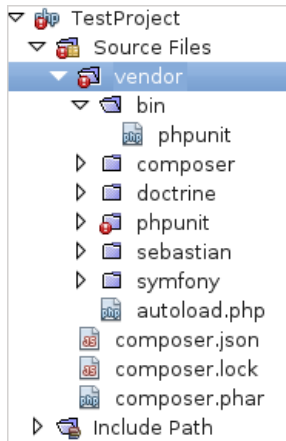
gérer Composer :

```

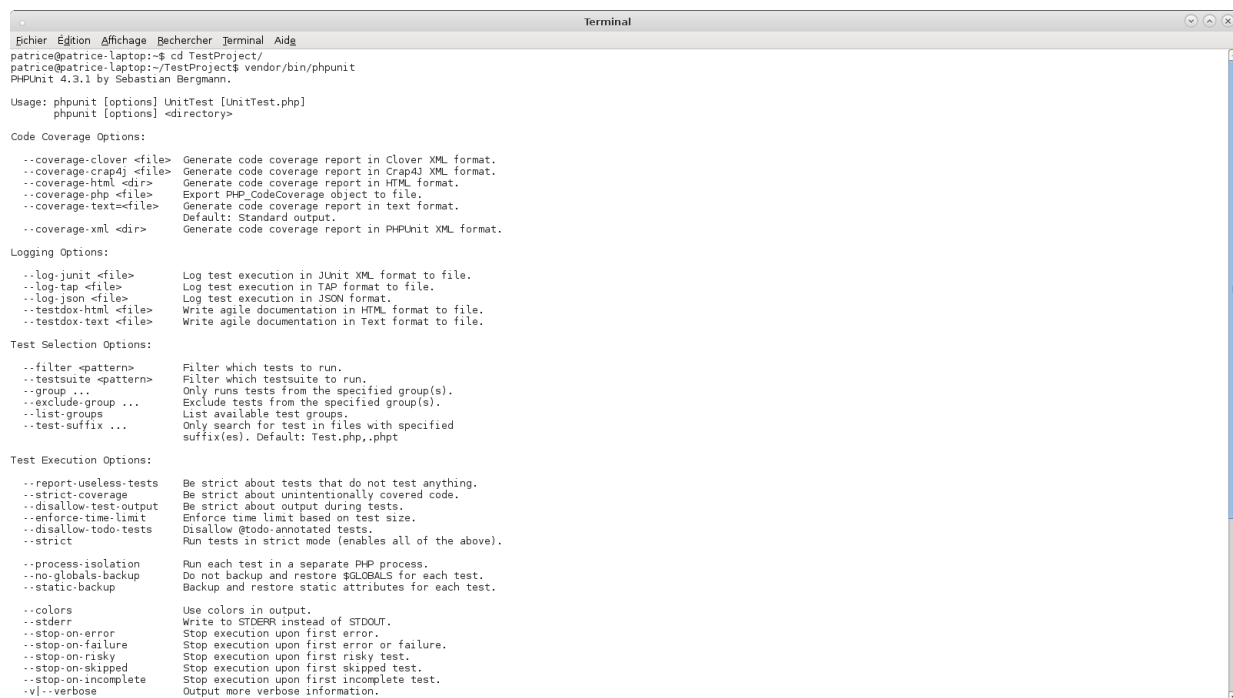
1  {
2      "require-dev": {
3          "phpunit/phpunit": "4.3.*"
4      }
5  }

```

Dans le fichier `composer.json`, le wildcard `"*"` signifie la version la plus élevée. Dans un terminal nous exécutons la commande `php composer.phar install` et laissons Composer installer nos dépendances. Un répertoire `vendor` contenant nos dépendances a été créé au sein de notre projet :



Dans le répertoire `vendor/bin` nous trouvons un exécutable `phpunit`. C'est cet exécutable qui va nous permettre de lancer nos tests. Si, dans le terminal, nous exécutons la commande `vendor/bin/phpunit` alors s'afficheront les commandes de `phpunit`. Quand nous exécutons cette commande nous sommes bien à la racine de notre projet. À chaque fois que nous lancerons nos tests nous veillerons à être à la racine de notre projet.



Nous allons créer une classe `Calculatrice` dans un répertoire `fr/fyigrane`, et tester ses méthodes. Nous allons tester les plus petites unités de code possible, nous faisons donc du test unitaire.

```

1  <?php
2
3  // fr/fyigrane/Calculatrice.php
4
5  namespace fr\fyligrane;
6
7  class Calculatrice {
8

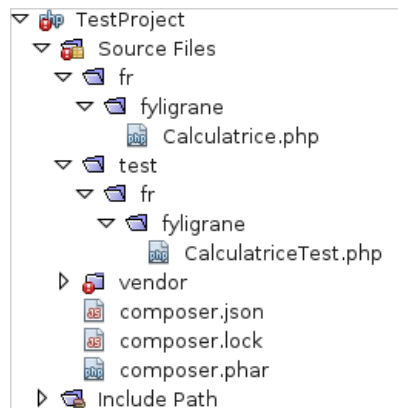
```

```

9      public function additionner($nb1, $nb2) {
10          return $nb1 + $nb2;
11      }
12
13  }
```

Cette classe ne contient pour l'instant qu'une méthode `additionner()` qui prend en argument deux entiers et qui renverra bien sûr la somme de ces deux entiers. Tester le bon fonctionnement de cette méthode revient à vérifier que l'entier retourné est bien égal à la somme des deux entiers passés en argument.

Nous allons créer dans un répertoire `test/fr/fyligrane`, une classe `CalculatriceTest` héritant de `PHPUnit_Framework_TestCase`.



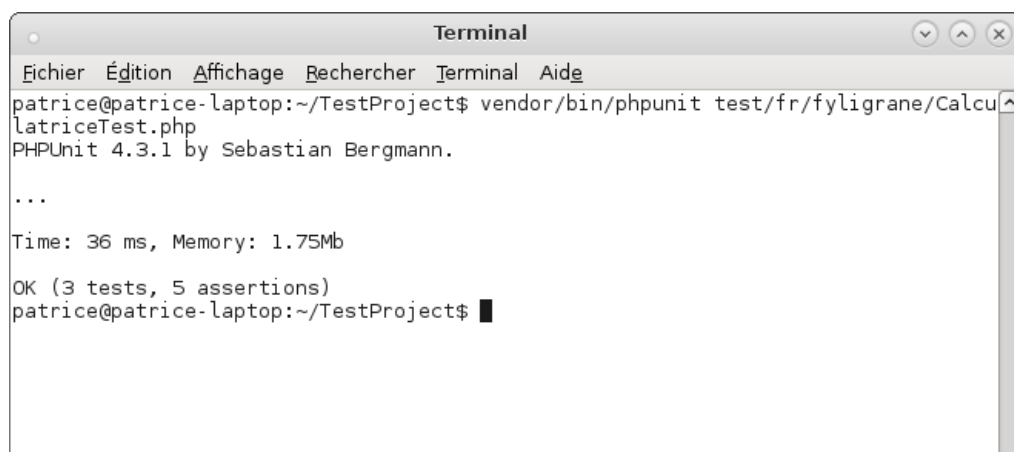
Dans cette classe nous allons créer une méthode `testAdditionner()` qui effectuera le test de notre méthode `additionner()`. La classe `PHPUnit_Framework_TestCase` possède une méthode `assertEquals()` qui permet de vérifier l'égalité de deux valeurs ou variables : `assertEquals(mixed $expected, mixed $actual[, string $message = ''])`.

Pour exécuter la fonction `additionner()` nous aurons évidemment besoin d'une instance de `Calculatrice`.

```

1  <?php
2
3  // test/fr/fyligrane/CalculatriceTest.php
4
5  include_once './fr/fyligrane/Calculatrice.php';
6
7  use fr\fyligrane\Calculatrice;
8
9  class CalculatriceTest extends PHPUnit_Framework_TestCase {
10
11      public function testAdditionner() {
12          $calc = new Calculatrice();
13          $result = $calc->additionner(15, 25);
14          $this->assertEquals(40, $result);
15      }
16
17  }
```

Pour exécuter ce test nous allons utiliser l'exécutable `phpunit` et lui passer en paramètre le fichier à tester :





En cas d'erreur, phpunit affiche quelques informations :

```

1 <?php
2
3 // test/fr/fyigrane/CalculatriceTest.php
4
5 include_once './fr/fyigrane/Calculatrice.php';
6
7 use fr\fyigrane\Calculatrice;
8
9 class CalculatriceTest extends PHPUnit_Framework_TestCase {
10
11     public function testAdditionner() {
12         $calc = new Calculatrice();
13         $result = $calc->additionner(15, 25);
14         $this->assertEquals("azerty", $result);
15     }
16
17 }

```

```

Terminal
Fichier Édition Affichage Rechercher Terminal Aide
patrice@patrice-laptop:~/TestProject$ vendor/bin/phpunit test/CalculatriceTest.php
PHPUnit 4.3.1 by Sebastian Bergmann.

F

Time: 34 ms, Memory: 2.00Mb

There was 1 failure:

1) CalculatriceTest::testAdditionner
Failed asserting that 40 matches expected 'azerty'.

/home/patrice/TestProject/test/CalculatriceTest.php:14

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
patrice@patrice-laptop:~/TestProject$

```

Un test peut comporter plusieurs "assertions" ; dans ce cas le test ne sera concluant que si toutes les assertions se vérifient :

```

1 <?php
2
3 // test/fr/fyigrane/CalculatriceTest.php
4
5 include_once './fr/fyigrane/Calculatrice.php';
6
7 use fr\fyigrane\Calculatrice;
8
9 class CalculatriceTest extends PHPUnit_Framework_TestCase {
10
11     public function testAdditionner() {
12         $calc = new Calculatrice();
13         $result = $calc->additionner(15, 25);
14         $this->assertEquals(40, $result);

```

```

15         // stupide... Juste pour l'exemple...
16         $this->assertNotFalse($result);
17     }
18
19 }

```

```

Terminal
Fichier Édition Affichage Rechercher Terminal Aide
patrice@patrice-laptop:~/TestProject$ vendor/bin/phpunit test/CalculatriceTest.php
PHPUnit 4.3.1 by Sebastian Bergmann.

.

Time: 33 ms, Memory: 1.75Mb

OK (1 test, 2 assertions)
patrice@patrice-laptop:~/TestProject$

```

La classe `PHPUnit_Framework_TestCase` possède un nombre important d'assertions sous la forme de méthode `assert*****()`.

Pour que phpunit sache quelles méthodes correspondent à un test, il faut que le nom de ces méthodes commence par `test` (ici `testAdditionner()`), ou alors que la méthode porte l'annotation `@test` (les annotations se situent toujours dans des commentaires de la forme `/**.....*/`).

```

1 <?php
2
3 // test/fr/fyigrane/CalculatriceTest.php
4
5 include_once './fr/fyigrane/Calculatrice.php';
6
7 use fr\fyligrane\Calculatrice;
8
9 class CalculatriceTest extends PHPUnit_Framework_TestCase {
10
11     /**
12      * @test
13      */
14     public function additionner() {
15         $calc = new Calculatrice();
16         $result = $calc->additionner(15, 25);
17         $this->assertEquals(40, $result);
18         // stupide... Juste pour l'exemple...
19         $this->assertNotFalse($result);
20     }
21
22 }

```

Fin

## NAVIGATION

[Accueil](#)



[Ma page](#)[Pages du site](#)[Mon profil](#)[Cours actuel](#)[POO PHP](#)[Participants](#)[Généralités](#)[La programmation orientée objet : premiers pas](#)[L'héritage](#)[Les interfaces](#)[Le typage](#)[Les namespaces](#)[Les exceptions](#)[Les bases de données avec PDO](#)[Les tests avec PHPUnit](#)[Premier test](#)[L'environnement du test : test fixtures](#)[Test des exceptions](#)[T.P. premier test](#)[Test des dépendances](#)[Test des bases de données](#)[Petite application version 2](#)[Petite application version 3](#)[Mes cours](#)**ADMINISTRATION**[Administration du cours](#)[Réglages de mon profil](#)Connecté sous le nom « [Arnaud Lemais](#) » ([Déconnexion](#))[POO PHP](#)



# Programmation orientée objet en PHP

[Accueil](#) ▶ [Mes cours](#) ▶ [Développement logiciel](#) ▶ [POO PHP](#) ▶ [Les tests avec PHPUnit](#) ▶ [L'environnement du test : test fixtures](#)

## L'environnement du test : test fixtures

Pour que les tests soient concluants, il faut qu'ils aient lieu dans un environnement "sain". L'environnement du test est constitué de toutes les "ressources" qui vont être impliquées au cours du test. Avant chaque test, chaque "ressource" doit être dans un état défini. Par exemple, si j'implémente une fonction qui écrit un haiku dans un fichier si ce fichier est vide, je vais pouvoir écrire deux tests :

1. un test à partir d'un fichier vide, et qui vérifie si la fonction a bien écrit dans ce fichier
2. un test à partir d'un fichier non vide, et qui vérifie si la fonction n'a pas écrit dans ce fichier

Nous avons donc deux tests avec deux environnements différents : un test nécessitera un fichier vide, l'autre un fichier non vide.

L'environnement du test peut aussi se composer d'objets. Dans certains tests nous aurons besoin d'objets dans un état défini, c'est-à-dire dont les attributs possèdent certaines valeurs définies. Par exemple, nous implémentons une fonction qui vérifie si une instance de `Personne` est majeure, en vérifiant la valeur de l'attribut `age` de `Personne`. Pour tester cette fonction, nous pourrions écrire deux tests :

1. un test à partir d'une instance de `Personne` dont l'attribut `age` vaut 10
2. un autre test à partir d'une autre instance de `Personne` dont l'attribut `age` vaut 35

□

Il est toujours difficile dans un test de choisir les "bonnes valeurs" à tester. Bien choisir ces valeurs c'est s'assurer qu'un test sera bien représentatif du comportement du composant.

La classe `PHPUnit_Framework_TestCase` possède des méthodes qui permettent de modifier l'environnement d'un ou des tests, et ceci à plusieurs moments :

- `setUp()` : permet de déterminer l'environnement avant chaque test, cette méthode est appelée avant chaque test (permet par exemple d'initialiser un objet pour le test)
- `tearDown()` : permet de déterminer l'environnement après chaque test, cette méthode est appelée après chaque test (permet par exemple de libérer une connexion...)
- `setUpBeforeClass()` : permet de déterminer l'environnement avant les tests, cette méthode de classe est appelée avant l'exécution du [premier test](#)
- `tearDownAfterClass()` : permet de déterminer l'environnement après les tests, cette méthode de classe est appelée après l'exécution du dernier test

Il existe des annotations correspondant aux méthodes :

- `@before` pour `setUp()`
- `@after` pour `tearDown()`
- `@beforeClass` pour `setUpBeforeClass()`
- `@afterClass` pour `tearDownAfterClass()`

Ajoutons à notre classe `Calculatrice` une méthode `soustraire()` :

```
1 <?php
2
3 // fr/fyligrane/Calculatrice.php
4
5 namespace fr\fyligrane;
6
7 class Calculatrice {
8
9     public function additionner($nb1, $nb2) {
10         return $nb1 + $nb2;
11     }
12 }
```

```

13     public function soustaire($nb1, $nb2) {
14         return $nb1 - $nb2;
15     }
16
17 }

```

Testons cette méthode dans notre classe de test CalculatriceTest. Dans notre classe de test CalculatriceTest, si nous souhaitons posséder une nouvelle instance de Calculatrice avant chaque test, nous pouvons utiliser la méthode setUp(), ainsi qu'un attribut d'instance pour maintenir une référence vers l'objet Calculatrice :

```

1  <?php
2
3  // test/fr/fyigrane/CalculatriceTest.php
4
5  include_once './fr/fyigrane/Calculatrice.php';
6
7  use fr\fyligrane\Calculatrice;
8
9  class CalculatriceTest extends PHPUnit_Framework_TestCase {
10
11     protected $calc;
12
13     protected function setUp() {
14         $this->calc = new Calculatrice();
15     }
16
17     public function testAdditionner() {
18         $result = $this->calc->additionner(15, 25);
19         $this->assertEquals(40, $result);
20         // stupide... Juste pour l'exemple...
21         $this->assertNotFalse($result);
22     }
23
24     public function testSoustraire() {
25         $result = $this->calc->soustaire(15, 25);
26         $this->assertEquals(-10, $result);
27     }
28
29 }

```

Avec les annotations :

```

1  <?php
2
3  // test/fr/fyigrane/CalculatriceTest.php
4
5  include_once './fr/fyigrane/Calculatrice.php';
6
7  use fr\fyligrane\Calculatrice;
8
9  class CalculatriceTest extends PHPUnit_Framework_TestCase {
10
11     protected $calc;
12
13     /**
14      * @before
15      */
16     protected function init() {
17         $this->calc = new Calculatrice();
18     }
19
20     /**
21      * @test
22      */
23     public function additionner() {

```



```

24         $result = $this->calc->additionner(15, 25);
25         $this->assertEquals(40, $result);
26         // stupide... Juste pour l'exemple...
27         $this->assertNotFalse($result);
28     }
29
30     /**
31      * @test
32      */
33     public function soustraire() {
34         $result = $this->calc->soustraire(15, 25);
35         $this->assertEquals(-10, $result);
36     }
37
38 }

```

Dans notre cas nous n'avons pas réellement besoin d'une nouvelle instance avant chaque test. Une seule instance serait suffisante pour tous les tests, nous pouvons créer un objet Calculatrice en attribut de classe et l'initialiser avec la méthode de classe setUpBeforeClass() :

```

1  <?php
2
3  // test/fr/fyigrane/CalculatriceTest.php
4
5  include_once './fr/fyigrane/Calculatrice.php';
6
7  use fr\fyligrane\Calculatrice;
8
9  class CalculatriceTest extends PHPUnit_Framework_TestCase {
10
11     protected static $calc;
12
13     public static function setUpBeforeClass() {
14         CalculatriceTest::$calc = new Calculatrice();
15     }
16
17     public function testAdditionner() {
18         $result = CalculatriceTest::$calc->additionner(15, 25);
19         $this->assertEquals(40, $result);
20         // stupide... Juste pour l'exemple...
21         $this->assertNotFalse($result);
22     }
23
24     public function testSoustraire() {
25         $result = CalculatriceTest::$calc->soustraire(15, 25);
26         $this->assertEquals(-10, $result);
27     }
28
29 }

```

Avec les annotations :

```

1  <?php
2
3  // test/fr/fyigrane/CalculatriceTest.php
4
5  include_once './fr/fyigrane/Calculatrice.php';
6
7  use fr\fyligrane\Calculatrice;
8
9  class CalculatriceTest extends PHPUnit_Framework_TestCase {
10
11     protected static $calc;
12
13     /**

```

```
14      * @beforeClass
15      */
16      public static function initBeforeClass() {
17          CalculatriceTest::$calc = new Calculatrice();
18      }
19
20      /**
21       * @test
22       */
23      public function additionner() {
24          $result = CalculatriceTest::$calc->additionner(15, 25);
25          $this->assertEquals(40, $result);
26          // stupide... Juste pour l'exemple...
27          $this->assertNotFalse($result);
28      }
29
30      /**
31       * @test
32       */
33      public function soustraire() {
34          $result = CalculatriceTest::$calc->soustraire(15, 25);
35          $this->assertEquals(-10, $result);
36      }
37
38  }
```

[Fin](#)

## NAVIGATION



### Accueil

#### ■ Ma page

[Pages du site](#)[Mon profil](#)[Cours actuel](#)

#### POO PHP

[Participants](#)[Généralités](#)[La programmation orientée objet : premiers pas](#)[L'héritage](#)[Les interfaces](#)[Le typage](#)[Les namespaces](#)[Les exceptions](#)[Les bases de données avec PDO](#)[Les tests avec PHPUnit](#)[Premier test](#)[L'environnement du test : test fixtures](#)[Test des exceptions](#)[T.P. premier test](#)[Test des dépendances](#)[Test des bases de données](#)[Petite application version 2](#)[Petite application version 3](#)[Mes cours](#)

## ADMINISTRATION

[Administration du cours](#)[Réglages de mon profil](#)

---

Connecté sous le nom « [Arnaud Lemais](#) » ([Déconnexion](#))  
[POO PHP](#)



# Programmation orientée objet en PHP

[Accueil](#) ► [Mes cours](#) ► [Développement logiciel](#) ► [POO PHP](#) ► [Les tests avec PHPUnit](#) ► [Test des exceptions](#)

## Test des exceptions

Dans la méthode `additionner()` de notre classe `Calculatrice`, nous allons jeter une exception si un des deux arguments n'est pas de type entier.

```
1 <?php
2
3 // fr/fyligrane/Calculatrice.php
4
5 namespace fr\fyligrane;
6
7 use Exception;
8
9 class Calculatrice {
10
11     public function additionner($nb1, $nb2) {
12         if (!is_integer($nb1) || !is_integer($nb2)) {
13             throw new Exception('les arguments doivent être de type entier');
14         }
15         return $nb1 + $nb2;
16     }
17
18     public function soustraire($nb1, $nb2) {
19         return $nb1 - $nb2;
20     }
21
22 }
```

Nous devons donc tester, dans le cas où un des deux arguments ne serait pas de type entier, que notre méthode jette bien une exception. Pour cela nous allons utiliser l'annotation `@expectedException`, qui va nous permettre de préciser le type d'Exception attendue :

```
1 <?php
2
3 // test/fr/fyligrane/CalculatriceTest.php
4
5 include_once './fr/fyligrane/Calculatrice.php';
6
7 use fr\fyligrane\Calculatrice;
8
9 class CalculatriceTest extends PHPUnit_Framework_TestCase {
10
11     protected $calc;
12
13     /**
14      * @before
15      */
16     protected function init() {
17         $this->calc = new Calculatrice();
18     }
19
20     /**
21      * @test
22      */
23     public function additionner() {
24         $result = $this->calc->additionner(15, 25);
```

```

25     $this->assertEquals(40, $result);
26     // stupide... Juste pour l'exemple...
27     $this->assertNotFalse($result);
28 }
29
30 /**
31  * @test
32  * @expectedException Exception
33  */
34 public function additionnerWithException() {
35     $this->calc->additionner(10, "azerty");
36 }
37
38 /**
39  * @test
40  */
41 public function soustraire() {
42     $result = $this->calc->soustraire(15, 25);
43     $this->assertEquals(-10, $result);
44 }
45
46 }

```

Il est possible de tester le message de l'exception avec l'annotation `@expectedExceptionMessage` :

```

1 <?php
2
3 // test/fr/fyigrane/CalculatriceTest.php
4
5 include_once './fr/fyigrane/Calculatrice.php';
6
7 use fr\fyligrane\Calculatrice;
8
9 class CalculatriceTest extends PHPUnit_Framework_TestCase {
10
11     protected $calc;
12
13     /**
14      * @before
15      */
16     protected function init() {
17         $this->calc = new Calculatrice();
18     }
19
20     /**
21      * @test
22      */
23     public function additionner() {
24         $result = $this->calc->additionner(15, 25);
25         $this->assertEquals(40, $result);
26         // stupide... Juste pour l'exemple...
27         $this->assertNotFalse($result);
28     }
29
30     /**
31      * @test
32      * @expectedException Exception
33      * @expectedExceptionMessage les arguments doivent être de type entier
34      */
35     public function additionnerWithException() {
36         $this->calc->additionner(10, "azerty");
37     }
38
39     /**
40      * @test
41      */

```

```

42     public function soustraire() {
43         $result = $this->calc->soustraire(15, 25);
44         $this->assertEquals(-10, $result);
45     }
46
47 }

```

Ou encore l'annotation , en utilisant une expression rationnelle (aussi dite régulière) :

```

1  <?php
2
3  // test/fr/fyigrane/CalculatriceTest.php
4
5  include_once './fr/fyigrane/Calculatrice.php';
6
7  use fr\fyligrane\Calculatrice;
8
9  class CalculatriceTest extends PHPUnit_Framework_TestCase {
10
11     protected $calc;
12
13     /**
14      * @before
15      */
16     protected function init() {
17         $this->calc = new Calculatrice();
18     }
19
20     /**
21      * @test
22      */
23     public function additionner() {
24         $result = $this->calc->additionner(15, 25);
25         $this->assertEquals(40, $result);
26         // stupide... Juste pour l'exemple...
27         $this->assertNotFalse($result);
28     }
29
30     /**
31      * @test
32      * @expectedException Exception
33      * @expectedExceptionMessageRegExp /type entier/
34      */
35     public function additionnerWithException() {
36         $this->calc->additionner(10, "azerty");
37     }
38
39     /**
40      * @test
41      */
42     public function soustraire() {
43         $result = $this->calc->soustraire(15, 25);
44         $this->assertEquals(-10, $result);
45     }
46
47 }

```

Fin

## NAVIGATION



[Accueil](#)

■ [Ma page](#)

[Pages du site](#)

Mon profil

Cours actuel

**POO PHP**

Participants

Généralités

La programmation orientée objet : premiers pas

L'héritage

Les interfaces

Le typage


Les namespaces

Les exceptions

Les bases de données avec PDO

Les tests avec PHPUnit

 **Premier test**

 L'environnement du test : test fixtures

 **Test des exceptions**

 T.P. premier test

 Test des dépendances

 Test des bases de données

Petite application version 2

Petite application version 3

[Mes cours](#)

## ADMINISTRATION



Administration du cours

Réglages de mon profil

Connecté sous le nom « [Arnaud Lemais](#) » ([Déconnexion](#))  
[POO PHP](#)



# Programmation orientée objet en PHP

[Accueil](#) ▶ [Mes cours](#) ▶ [Développement logiciel](#) ▶ [POO PHP](#) ▶ [Les tests avec PHPUnit](#) ▶ [T.P. premier test](#)

## T.P. premier test

Ajouter à la classe Calculatrice une méthode diviser() qui prendra en argument deux entiers.

Si un des deux arguments n'est pas de type entier alors jeter une exception avec le message "les arguments doivent être de type entier".

Si le premier argument est égal à 0 alors jeter une exception avec le message "division par zero impossible".

Ajouter à la classe de test les tests qui permettront de vérifier le bon fonctionnement de cette méthode.

Les fichiers seront à remettre dans une archive zip dont le nom sera de la forme "premier\_test\_*nom\_prenom*.zip".

## État du travail remis

Numéro de tentative	Ceci est la tentative 1.
Statut des travaux remis	Aucune tentative
Statut de l'évaluation	Pas évalué
Dernière modification	vendredi 27 mars 2015, 15:55

Ajouter un travail

Modifier votre travail remis

### NAVIGATION



#### Accueil

##### ■ Ma page

Pages du site

Mon profil

Cours actuel

#### POO PHP

Participants

Généralités

La programmation orientée objet : premiers pas

L'héritage

Les interfaces

Le typage

Les namespaces

Les exceptions

Les bases de données avec PDO

Les tests avec PHPUnit

Premier test

L'environnement du test : test fixtures

Test des exceptions

**T.P. premier test**

Test des dépendances



 [Test des bases de données](#)[Petite application version 2](#)[Petite application version 3](#)[Mes cours](#)**ADMINISTRATION**[Administration du cours](#)[Réglages de mon profil](#)

Connecté sous le nom « [Arnaud Lemais](#) » ([Déconnexion](#))  
[POO PHP](#)



# Programmation orientée objet en PHP

[Accueil](#) ▶ [Mes cours](#) ▶ [Développement logiciel](#) ▶ [POO PHP](#) ▶ [Les tests avec PHPUnit](#) ▶ [Test des dépendances](#)

## Test des dépendances

Nous allons créer une classe `Calculatrice2` qui possédera une méthode `additionner()` qui renverra la somme des valeurs contenues dans ce tableau dans un fichier texte. Nous aurons donc besoin d'une classe `FileReader` qui possédera un attribut d'instance `path` (le chemin vers le fichier txt), et une méthode `getIntegers()`. Cette méthode lira les valeurs du fichier et renverra un tableau d'entiers.

Pourquoi ne demandons-nous pas à la classe `Calculatrice2` de lire elle-même le fichier ? Simplement parce qu'il est un grand principe en conception objet qui consiste à n'accorder à une classe qu'une seule "responsabilité". Une responsabilité c'est simplement une tâche à accomplir. Chaque classe doit avoir une tâche bien déterminée : ici la classe `Calculatrice` fera les calculs, et la classe `FileReader` lira le fichier ; en somme c'est chacun son boulot. Une classe qui n'a qu'une seule responsabilité est plus facile à tester et à maintenir.

Prenons l'exemple de "la personne à tout faire" dans une entreprise, bien souvent cette personne a tellement de tâches qu'on ne sait plus ce qu'elle fait, et surtout ce qu'elle ne fait pas... Il est beaucoup plus difficile d'évaluer une personne qui fait 50 choses en même temps qu'une personne qui a une tâche unique. De plus "la personne multi-usage" est difficile à remplacer en cas de problème...

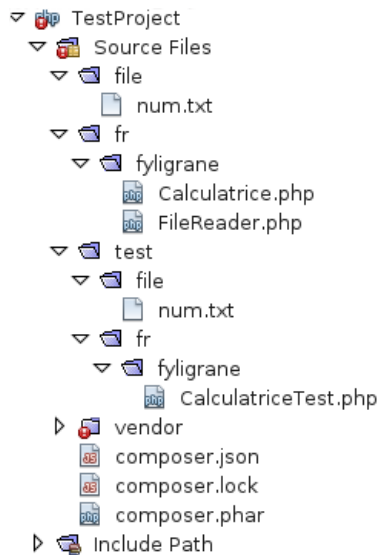
Dans le répertoire `fr/fyligrane`, nous créons la classe `FileReader` :

```
1 <?php
2
3 // fr/fyligrane/FileReader.php
4
5 namespace fr\fyligrane;
6
7 class FileReader {
8
9     private $path;
10
11     function __construct($path) {
12         $this->path = $path;
13     }
14
15     public function getIntegers() {
16         $text = file_get_contents($this->path);
17         return explode(' ', $text);
18     }
19
20 }
```

Le fichier `num.txt` contiendra les entiers à additionner, sur une seule ligne sans saut de ligne à la fin. Nous le placerons dans le répertoire `file`. Nous placerons aussi une copie de ce fichier dans le répertoire `test/file`.

```
1 10 54 12 89 65 23 54
```

L'arborescence de notre projet est donc :



Notre Calculatrice2 aura donc besoin de collaborer avec la classe FileReader : la classe Calculatrice2 sera donc dépendante de la classe FileReader. Pour régler ce problème de dépendance nous avons deux solutions :

1. donner à la classe Calculatrice2 un attribut d'instance de type FileReader, cet attribut d'instance sera utiliser dans la méthode additionner()
2. passer à la méthode additionner un argument de type FileReader

Voici l'implémentation de la première solution que nous appellerons Calculatrice2, c'est cette solution que nous utiliserons par la suite :

```

1  <?php
2
3  // fr/fyligrane/Calculatrice2.php
4
5  namespace fr\fyligrane;
6
7  class Calculatrice2 {
8
9      private $reader;
10
11     public function __construct(FileReader $reader) {
12         $this->reader = $reader;
13     }
14
15     public function additionner() {
16         $integers = $this->reader->getIntegers();
17         return array_sum($integers);
18     }
19
20 }
```

Et l'implémentation de la deuxième solution que nous appellerons Calculatrice2\_1 :

```

1  <?php
2
3  // fr/fyligrane/Calculatrice2_1.php
4
5  namespace fr\fyligrane;
6
7  class Calculatrice2_1 {
8
9      public function additionner(FileReader $reader) {
10         $integers = $reader->getIntegers();
11         return array_sum($integers);
12     }
13
14 }
```

Nous devons maintenant tester nos deux classes. Pour la classe `FileReader`, le test est simple, nous allons appeler la méthode `getIntegers()` et vérifier que cette méthode renvoie bien un tableau contenant les valeurs du fichier. Nous créons une classe `FileReaderTest` dans le répertoire `test/fr/fyigrane` et, dans le répertoire `test`, un fichier `num.txt`, copie du fichier `num.txt` contenu dans le répertoire `fr/fyigrane`.

```

1 <?php
2
3 // test/fr/fyigrane/FileReaderTest.php
4
5 include_once './fr/fyigrane/FileReader.php';
6
7 use fr\fyigrane\FileReader;
8
9 class FileReaderTest extends PHPUnit_Framework_TestCase {
10
11     public $reader;
12
13     public function setUp() {
14         $this->reader = new FileReader('test/file/num.txt');
15     }
16
17     public function testGetIntegers() {
18         $integers = $this->reader->getIntegers();
19         $expected = [10, 54, 12, 89, 65, 23, 54];
20         $this->assertEquals($expected, $integers);
21     }
22
23 }

```

Nous lançons le test dans un terminal :

```

Terminal
Fichier Édition Affichage Rechercher Terminal Aide
patrice@patrice-laptop:~/TestProject$ vendor/bin/phpunit test/fr/fyigrane/FileReaderTest.php
PHPUnit 4.3.1 by Sebastian Bergmann.

.

Time: 36 ms, Memory: 1.75Mb

OK (1 test, 1 assertion)
patrice@patrice-laptop:~/TestProject$

```

Lorsque vous testez une classe qui nécessite l'accès à une ressource (la lecture d'un fichier par exemple), il est préférable de créer des ressources spécifiques pour les tests. Par exemple un fichier qui ne sera utilisé que pour les tests.

Nous allons maintenant tester la classe `Calculatrice2`. Nous créons une classe de test `Calculatrice2Test`. Dans notre classe de test nous allons être obligés de fournir à la classe `Calculatrice2` une instance de `FileReader`, sans quoi la méthode `additionner()` ne pourra pas fonctionner. La classe `Calculatrice2` est donc dépendante de la classe `FileReader`, nous devons donc satisfaire cette dépendance.

Le fait de donner à une instance une autre instance d'une classe dont elle dépend s'appelle une injection de dépendance. Ici cette injection de dépendance aura lieu dans le constructeur de la classe `Calculatrice2`.

```

1  <?php
2
3  // test/fr/fyigrane/Calculatrice2Test.php
4
5  include_once './fr/fyigrane/FileReader.php';
6  include_once './fr/fyigrane/Calculatrice2.php';
7
8  use fr\fyigrane\FileReader;
9  use fr\fyigrane\Calculatrice2;
10
11 class Calculatrice2Test extends PHPUnit_Framework_TestCase {
12
13     public $calc;
14
15     public function setUp() {
16         $reader = new FileReader('test/file/num.txt');
17         $this->calc = new Calculatrice2($reader);
18     }
19
20     public function testAdditionner() {
21         $result = $this->calc->additionner();
22         $this->assertEquals(307, $result);
23     }
24
25 }

```

Le problème ici c'est que nous n'avons pas fait un test unitaire : nous n'avons pas tester la plus petite unité de code possible. Nous avons en fait tester une méthode d'un objet qui appelle en sous-main une méthode d'un autre objet don il dépend. En effet la méthode additionner de la classe Calculatrice2 appelle la méthode getIntegers() de la classe FileReader. En cas de problème ne nous pouvons pas savoir si c'est l'instance de FileReader ou de Calculatrice2 (ou même les deux) qui dysfonctionne.

Nous avons en fait réalisé un test d'intégration en vérifiant que la collaboration entre deux objets se déroulaient correctement.

Pour effectuer un test unitaire de la méthode additionner() nous devons nous assurer qu'un dysfonctionnement ne sera pas provoqué par une méthode d'une dépendance, mais bien par la méthode elle-même. Pour cela nous allons remplacer toutes les dépendances par des dépendances dont les méthodes fonctionnent à coup sûr, en renvoyant exactement ce qu'elles renverraient dans une situation "normale". Dans notre cas, nous nous attendons à ce que la méthode getIntegers() renvoie un tableau d'entiers.

PHPUnit va nous permettre de créer des objets simulacres, appelés généralement mock objects. Le terme de mock recouvre plusieurs autres termes notamment les terme fake et stub. Il existe cependant des différences entre les fakes, les stubs et les mocks.

La classe PHPUnit\_Framework\_TestCase possède une méthode getMock() qui permet de récupérer un objet de type PHPUnit\_Framework\_MockObject\_MockObject. Par défaut, toutes les méthodes de l'objet simulacre renvoient null.

Le premier argument de la méthode getMock() correspond au nom de la classe à simuler.

Le deuxième argument, par défaut un tableau vide, est un tableau contenant les noms des méthodes redéfinies dans le simulacre, les autres méthodes garderont le comportement défini dans la classe originale.

Le troisième argument est un tableau contenant les arguments qui devront être passés au constructeur de la classe originale.

La classe MockObject possède une méthode expects() qui va permettre de préciser combien de fois la méthode concernée doit être appelée. Cette méthode renvoie un objet de type PHPUnit\_Framework\_MockObject\_Builder\_InvocationMocker et prend en argument un objet de type PHPUnit\_Framework\_MockObject\_Matcher\_Invocation, qui peut être récupéré grâce à des méthodes de PHPUnit\_Framework\_TestCase, notamment :

- any() : la méthode peut être appelée autant de fois que nécessaire
- once() : la méthode peut être appelée une fois
- atLeastOnce() : la méthode peut être appelée au moins une fois

- `exactly($n)` : la méthode peut être appelée exactement `$n` fois

La classe `InvocationMocker` possède plusieurs méthodes utiles dont :

- `method()` qui prend en argument le nom de la méthode à simuler
- `with()` : qui prend en argument les arguments à passer à la méthode à simuler
- `willReturn()` : qui prend en argument la valeur de retour de la méthode à simuler

Dans notre classe `Calculatrice2Test` nous allons créer un mock de `FileReader` :

```

1  <?php
2
3  // test/fr/fyigrane/Calculatrice2Test.php
4
5  include_once './fr/fyigrane/FileReader.php';
6  include_once './fr/fyigrane/Calculatrice2.php';
7
8  use fr\fyligrane\Calculatrice2;
9
10 class Calculatrice2Test extends PHPUnit_Framework_TestCase {
11
12     public $calc;
13
14     public function setUp() {
15         $reader = $this->getMock('fr\fyligrane\FileReader', [
16             , ['test/file/num.txt'];
17
18         $invocationMocker = $reader->expects($this->once());
19         $invocationMocker->method('getIntegers')->willReturn([10, 20, 30]);
20         $this->calc = new Calculatrice2($reader);
21     }
22
23     public function testAdditionner() {
24         $result = $this->calc->additionner();
25         $this->assertEquals(60, $result);
26     }
27 }
```

Fin

## NAVIGATION



### Accueil

#### ■ Ma page

Pages du site

Mon profil

Cours actuel

#### POO PHP

Participants

Généralités

La programmation orientée objet : premiers pas

L'héritage

Les interfaces

Le typage

Les namespaces

Les exceptions

Les bases de données avec PDO

Les tests avec PHPUnit

Premier test

L'environnement du test : test fixtures

Test des exceptions

T.P. premier test

**Test des dépendances**

Test des bases de données

Petite application version 2

Petite application version 3

[Mes cours](#)

## ADMINISTRATION



Administration du cours

Réglages de mon profil

Connecté sous le nom « [Arnaud Lemais](#) » ([Déconnexion](#))  
[POO PHP](#)



# Programmation orientée objet en PHP

[Accueil](#) ▶ [Mes cours](#) ▶ [Développement logiciel](#) ▶ [POO PHP](#) ▶ [Les tests avec PHPUnit](#) ▶ [Test des bases de données](#)

## Test des bases de données

Pour tester nos composants d'accès aux données, nous allons avoir besoin de DBUnit. DBUnit est une extension de PHPUnit qui permet d'intégrer plus facilement des tests de bases de données. En fait ce que nous allons tester ce ne sont pas les bases de données elles-même, mais les composants d'accès aux données (notre fameuse couche DAO).

Nous allons donc ajouter à notre fichier `composer.json` une dépendance vers DBUnit :

```
1 {
2     "require-dev": {
3         "phpunit/phpunit": "4.3.*",
4         "phpunit/dbunit": ">=1.2"
5     }
6 }
```

Puis nous exécutons, dans un terminal, la commande `php composer.phar update`.

```
Terminal
Fichier Édition Affichage Rechercher Terminal Aide
patrice@patrice-laptop:~/TestProject$ php composer.phar update
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Removing phpunit/phpunit (4.3.1)
- Installing phpunit/phpunit (4.3.3)
  Downloading: 100%
- Installing phpunit/dbunit (1.3.1)
  Downloading: 100%
Writing lock file
Generating autoload files
patrice@patrice-laptop:~/TestProject$
```

Une fois nos dépendances installées, nous allons récupérer notre classe `MysqlDao`. Nous mettons cette classe dans le répertoire `fr/fyligrane` de notre projet, et nous lui ajoutons un namespace `fr/fyligrane` :

```
1 <?php
2
3 // fr/fyligrane/MysqlDao.php
4
5 namespace fr\fyligrane;
6
7 use PDO;
8
9 class MysqlDao {
10
11     private $datasource;
12     private $user;
13     private $password;
14     private $conn;
15 }
```



```

16     function __construct($datasource, $user, $password) {
17         $this->datasource = $datasource;
18         $this->user = $user;
19         $this->password = $password;
20         $this->conn = new PDO($datasource, $user, $password
21             , [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION]);
22     }
23
24     public function getAllPersonnes() {
25         $query = 'SELECT * FROM personne';
26         $stmt = $this->conn->query($query);
27         $result = $stmt->fetchAll(PDO::FETCH_ASSOC);
28         return $result;
29     }
30
31     public function addPersonne($nom, $prenom) {
32         $query = "INSERT INTO personne (nom, prenom) VALUES "
33             . "('$nom', '$prenom')";
34         $result = $this->conn->exec($query);
35         return $result;
36     }
37
38 }

```

Pour effectuer des tests sur des bases de données nous allons devoir respecter certaines étapes entre chaque test :

1. clean-up : nettoyage de la base de données
2. set up : préparation de l'environnement de test et des données de test, avant chaque test la base doit être dans un état cohérent
3. run : exécution du test
4. tear down : au besoin, nettoyage et/ou libération de ressources

Nous créons dans le répertoire test/fr/fyigrane une classe de test MysqlDaoTest, mais cette fois notre classe ne va pas hériter de PHPUnit\_Framework\_TestCase mais de PHPUnit\_Extensions\_Database\_TestCase. La classe abstraite PHPUnit\_Extensions\_Database\_TestCase nous impose l'implémentation de deux méthodes :

- getConnection() : qui renverra un objet de type PHPUnit\_Extensions\_Database\_DB\_IDatabaseConnection et qui représentera une connexion à notre base de tests
- getDataSet() : qui renverra un objet de type PHPUnit\_Extensions\_Database\_DataSet\_IDataSet et qui représentera un ensemble de données structurées à inclure dans notre base de données avant le lancement des tests

Le clean-up de la base sera assurée par DBUnit. Entre chaque test, la méthode setUp() de la classe PHPUnit\_Extensions\_Database\_TestCase effectuera un TRUNCATE sur les tables de la base. Le set up sera assuré par la méthode setUp() qui appellera en sous-main la méthode getDataSet(). Cette méthode getDataSet() va permettre le chargement de données dans la base à partir d'un fichier. Ce fichier, représentant notre ensemble de données, pourra être un fichier XML, CSV ou encore YAML. Pour ce projet, nous choisirons le format YAML. Voici le fichier correspondant au dataset de base, c'est à dire le dataset qui sera injecté dans la base avec chaque test. Nous appellerons ce fichier personne\_base.yml et nous le mettrons dans le répertoire test/dataset :

```

1  # test/dataset/personne_base.yml
2
3  personne:
4      -
5          id: 1
6          nom: "Sparrow"
7          prenom: "Jack"
8      -
9          id: 2
10         nom: "Wayne"
11         prenom: "Bruce"

```

Une fois ce dataset injecté nous trouverons donc dans notre base une table personne contenant les deux enregistrements décrits dans le fichier.

Le set up va aussi nécessiter une connexion à notre base de données de test. Cette connexion va être renvoyée par la méthode `getConnection()`.

Nous créerons aussi dans notre `setUp()` une instance de la classe que nous souhaitons tester, c'est-à-dire `MysqlDao`.

Ajoutons à cela un fichier de configuration de PHPUnit, que nous appellerons `phpunit.xml` (le nom n'a pas d'importance), et qui contiendra des configurations de PHPUnit. Nous mettrons ce fichier dans le répertoire `test`. Nous pourrions indiquer dans ce fichier, entre autres, un ensemble d'éléments `var` possédant un attribut `name` et un attribut `value`. Ces éléments seront récupérables dans nos classes de test via un tableau associatif `$GLOBAL`.

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!--test/phpunit.xml-->
3 <phpunit>
4   <php>
5     <var name="DB_DRIVER" value="mysql" />
6     <var name="DB_USER" value="root" />
7     <var name="DB_PASSWORD" value="root" />
8     <var name="DB_HOST" value="localhost" />
9     <var name="DB_DATABASE" value="test_pdo" />
10   </php>
11 </phpunit>

```

Nous pouvons maintenant implémenter les méthodes `getConnection()`, `getDataset()` et `setUp()` de notre classe :

```

1 <?php
2
3 // test/fr/fyigrane/MysqlDaoTest.php
4
5 include './fr/fyigrane/MysqlDao.php';
6
7 use fr\fyligrane\MysqlDao;
8
9 class MysqlDaoTest extends PHPUnit_Extensions_Database_TestCase {
10
11     protected $connection;
12     protected $dao;
13
14     protected function getConnection() {
15         if ($this->connection === null) {
16             $connectionString = $GLOBALS['DB_DRIVER'] . ':host=' .
17                 $GLOBALS['DB_HOST'] . ';dbname=' . $GLOBALS['DB_DATABASE'];
18             $this->connection = $this->createDefaultDBConnection(
19                 new PDO($connectionString, $GLOBALS['DB_USER'],
20                     $GLOBALS['DB_PASSWORD']));
21         }
22         return $this->connection;
23     }
24
25     protected function getDataSet() {
26         return new PHPUnit_Extensions_Database_DataSet_YamlDataSet(
27             './test/fyigrane/personne_base.yml');
28     }
29
30     protected function setUp() {
31         $conn = $this->getConnection();
32         // désactivation des contraintes de clés étrangères pour permettre
33         //le chargement des données via le dataset
34         $conn->getConnection()->query('set foreign_key_checks=0');
35         parent::setUp();
36         // activation des contraintes de clés étrangères
37         $conn->getConnection()->query('set foreign_key_checks=1');
38         $connectionString = $GLOBALS['DB_DRIVER'] . ':host=' . $GLOBALS['DB_HOST'] .
39             ';dbname=' . $GLOBALS['DB_DATABASE'];
40         $this->dao = new MysqlDao($connectionString, $GLOBALS['DB_USER'],

```

```

41         $GLOBALS['DB_PASSWORD']);
42     }
43
44 }

```

Nous allons maintenant tester la méthode `addPersonne()` de `MysqlDao`. Pour cela nous allons, dans une méthode de test, appeler la méthode à partir de l'attribut d'instance `$dao`. Nous ajouterons donc un enregistrement à notre base de données de test.

Puis nous récupérerons un dataset qui correspondra à l'état de notre base de test après l'insertion, et nous le comparerons à un dataset prédéfini qui représentera l'état attendu de la base après une insertion. Si les deux datasets coïncident cela signifie que le test est réussi. Dans un fichier `add_personne.yml` nous allons créer notre dataset "résultat attendu". Ce fichier se trouvera dans le répertoire `test/dataset` :

```

1  # test/dataset/add_personne.yml
2
3  personne:
4      -
5          id: 1
6          nom: "Sparrow"
7          prenom: "Jack"
8      -
9          id: 2
10         nom: "Wayne"
11         prenom: "Bruce"
12      -
13         id: 3
14         nom: "Kent"
15         prenom: "Clark"

```

Nous implémentons la méthode `testAddPersonne()` :

```

1  <?php
2
3  // test/fr/fyigrane/MysqlDaoTest.php
4
5  include './fr/fyigrane/MysqlDao.php';
6
7  use fr\fyligrane\MysqlDao;
8
9  class MysqlDaoTest extends PHPUnit_Extensions_Database_TestCase {
10
11     protected $connection;
12     protected $dao;
13
14     protected function getConnection() {
15         if ($this->connection === null) {
16             $connectionString = $GLOBALS['DB_DRIVER'] . ':host=' .
17                 $GLOBALS['DB_HOST'] . ';dbname=' . $GLOBALS['DB_DATABASE'];
18             $this->connection = $this->createDefaultDBConnection(
19                 new PDO($connectionString, $GLOBALS['DB_USER']
20                     , $GLOBALS['DB_PASSWORD']));
21         }
22         return $this->connection;
23     }
24
25     protected function getDataSet() {
26         return new PHPUnit_Extensions_Database_DataSet_YamlDataSet(
27             './test/dataset/personne_base.yml');
28     }
29
30     protected function setUp() {
31         $conn = $this->getConnection();
32         // désactivation des contraintes de clés étrangères pour permettre
33         // le chargement des données via le dataset

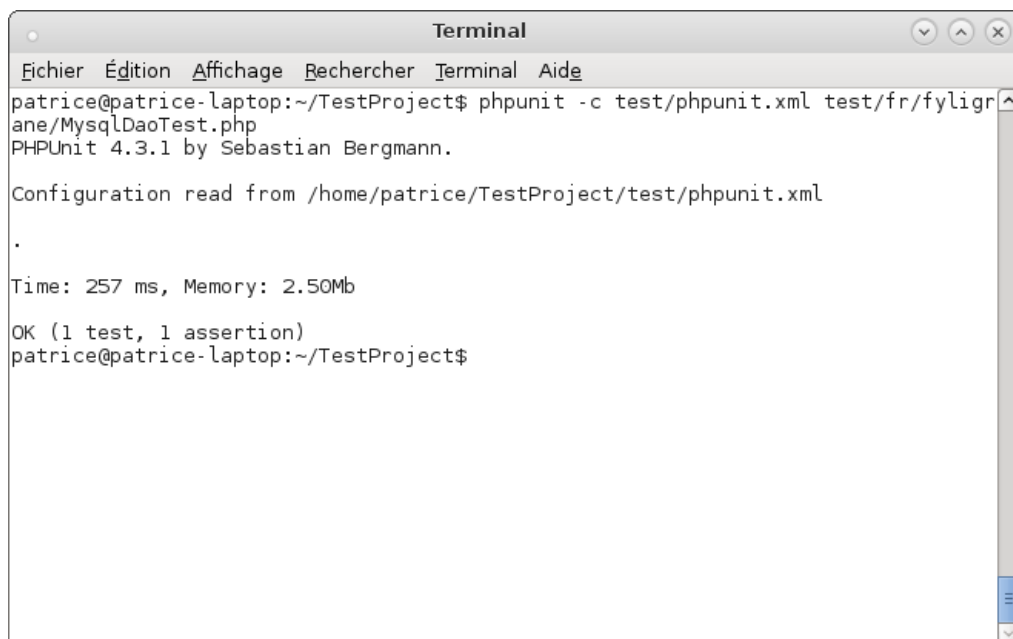
```

```

34     $conn->getConnection()->query('set foreign_key_checks=0');
35     parent::setUp();
36     // activation des contraintes de clés étrangères
37     $conn->getConnection()->query('set foreign_key_checks=1');
38     $connectionString = $GLOBALS['DB_DRIVER'] . ':host=' .
39         $GLOBALS['DB_HOST'] . ';dbname=' . $GLOBALS['DB_DATABASE'];
40     $this->dao = new MysqlDao($connectionString, $GLOBALS['DB_USER']
41         , $GLOBALS['DB_PASSWORD']);
42 }
43
44 public function testAddPersonne() {
45     $this->dao->addPersonne("Kent", "Clark");
46     // création d'un objet dataset à partir de la connexion
47     $actualDataset = new PHPUnit_Extensions_Database_DataSet_QueryDataSet(
48         $this->getConnection());
49     // ajout à ce dataset des enregistrements de la table personne
50     // de notre base de données de test
51     $actualDataset->addTable('personne');
52     // récupération d'un dataset à partir de notre fichier
53     $expectedDataset = new PHPUnit_Extensions_Database_DataSet_YamlDataSet(
54         './test/dataset/add_personne.yml');
55     // comparaison des deux datasets
56     $this->assertDataSetsEqual($expectedDataset, $actualDataset);
57 }
58
59 }

```

Pour lancer le test, il faut passer à la commande phpunit une option qui indique le chemin vers le fichier de configuration phpunit.xml :



```

Terminal
Fichier Édition Affichage Rechercher Terminal Aide
patrice@patrice-laptop:~/TestProject$ phpunit -c test/phpunit.xml test/fr/fyigr
ane/MysqlDaoTest.php
PHPUnit 4.3.1 by Sebastian Bergmann.

Configuration read from /home/patrice/TestProject/test/phpunit.xml

.

Time: 257 ms, Memory: 2.50Mb

OK (1 test, 1 assertion)
patrice@patrice-laptop:~/TestProject$

```

Une autre possibilité consiste à récupérer le résultat d'une requête SQL, et de comparer cette table, issue de la requête, à une table du dataset "résultat attendu". Voici la dataset "résultat attendu", dans un fichier add\_personne\_2.yml :

```

1 # test/dataset/add_personne_2.yml
2
3 personne:
4 -
5     id: 3
6     nom: "Kent"
7     prenom: "Clark"

```

Nous ajoutons une méthode testAddPersonne2() à notre classe de test :

```

1  <?php
2
3  // test/fr/fyigrane/MysqlDaoTest.php
4
5  include './fr/fyigrane/MysqlDao.php';
6
7  use fr\fyigrane\MysqlDao;
8
9  class MysqlDaoTest extends PHPUnit_Extensions_Database_TestCase {
10
11     protected $connection;
12     protected $dao;
13
14     protected function getConnection() {
15         if ($this->connection === null) {
16             $connectionString = $GLOBALS['DB_DRIVER'] . ':host=' .
17                 $GLOBALS['DB_HOST'] . ';dbname=' . $GLOBALS['DB_DATABASE'];
18             $this->connection = $this->createDefaultDBConnection(
19                 new PDO($connectionString, $GLOBALS['DB_USER']
20                     , $GLOBALS['DB_PASSWORD']));
21         }
22         return $this->connection;
23     }
24
25     protected function getDataSet() {
26         return new PHPUnit_Extensions_Database_DataSet_YamlDataSet(
27             './test/dataset/personne_base.yml');
28     }
29
30     protected function setUp() {
31         $conn = $this->getConnection();
32         // désactivation des contraintes de clés étrangères pour permettre
33         // le chargement des données via le dataset
34         $conn->getConnection()->query('set foreign_key_checks=0');
35         parent::setUp();
36         // activation des contraintes de clés étrangères
37         $conn->getConnection()->query('set foreign_key_checks=1');
38         $connectionString = $GLOBALS['DB_DRIVER'] . ':host=' .
39             $GLOBALS['DB_HOST'] . ';dbname=' . $GLOBALS['DB_DATABASE'];
40         $this->dao = new MysqlDao($connectionString, $GLOBALS['DB_USER']
41             , $GLOBALS['DB_PASSWORD']);
42     }
43
44     public function testAddPersonne() {
45         $this->dao->addPersonne("Kent", "Clark");
46         // création d'un objet dataset à partir de la connexion
47         $actualDataset = new PHPUnit_Extensions_Database_DataSet_QueryDataSet(
48             $this->getConnection());
49         // ajout à ce dataset des enregistrements de la table personne
50         // de notre base de données de test
51         $actualDataset->addTable('personne');
52         // récupération d'un dataset à partir de notre fichier
53         $expectedDataset = new PHPUnit_Extensions_Database_DataSet_YamlDataSet(
54             './test/dataset/add_personne.yml');
55         // comparaison des deux datasets
56         $this->assertDataSetsEqual($expectedDataset, $actualDataset);
57     }
58
59     public function testAddPersonne2() {
60         $this->dao->addPersonne("Kent", "Clark");
61         // création d'une table de résultat nommée personne
62         $queryTable = $this->getConnection()->createQueryTable(
63             'personne', 'SELECT * FROM personne where id=3'
64         );
65         $expectedDataset = new PHPUnit_Extensions_Database_DataSet_YamlDataSet(

```

```
66         './test/dataset/add_personne_2.yml');
67         // récupération de la table personne du dataset "résultat attendu"
68         $expectedTable = $expectedDataset->getTable("personne");
69         // comparaison des deux tables
70         $this->assertTablesEqual($expectedTable, $queryTable);
71     }
72 }
73 }
```

[Fin](#)

## NAVIGATION



### Accueil

#### ■ Ma page

[Pages du site](#)[Mon profil](#)[Cours actuel](#)

#### POO PHP

[Participants](#)[Généralités](#)[La programmation orientée objet : premiers pas](#)[L'héritage](#)[Les interfaces](#)[Le typage](#)[Les namespaces](#)[Les exceptions](#)[Les bases de données avec PDO](#)[Les tests avec PHPUnit](#) [Premier test](#) [L'environnement du test : test fixtures](#) [Test des exceptions](#) [T.P. premier test](#) [Test des dépendances](#) **[Test des bases de données](#)**[Petite application version 2](#)[Petite application version 3](#)[Mes cours](#)

## ADMINISTRATION

[Administration du cours](#)[Réglages de mon profil](#)

Connecté sous le nom « [Arnaud Lemais](#) » ([Déconnexion](#))  
[POO PHP](#)