Programmation orientée objet en PHP

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...

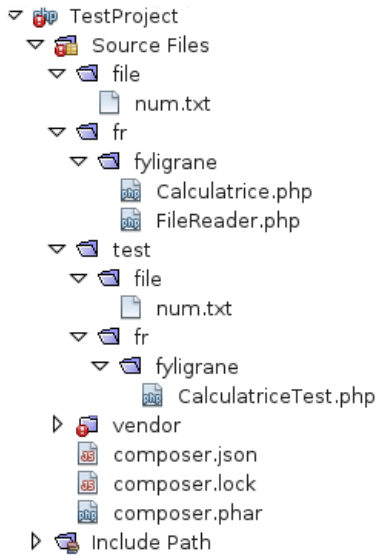
Dans le répertoire fr/fyligrane, nous créons la classe FileReader :

```php
<?php

// fr/fyligrane/FileReader.php

namespace fr\fyligrane;

class FileReader {

    private $path;

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

    public function getIntegers() {
        $text = file_get_contents($this->path);
        return explode(' ', $text);
    }

}
```

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.

```
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 :

```php
<?php

// fr/fyligrane/Calculatrice2.php

namespace fr\fyligrane;

class Calculatrice2 {

    private $reader;

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

    public function additionner() {
        $integers = $this->reader->getIntegers();
        return array_sum($integers);
    }

}
```

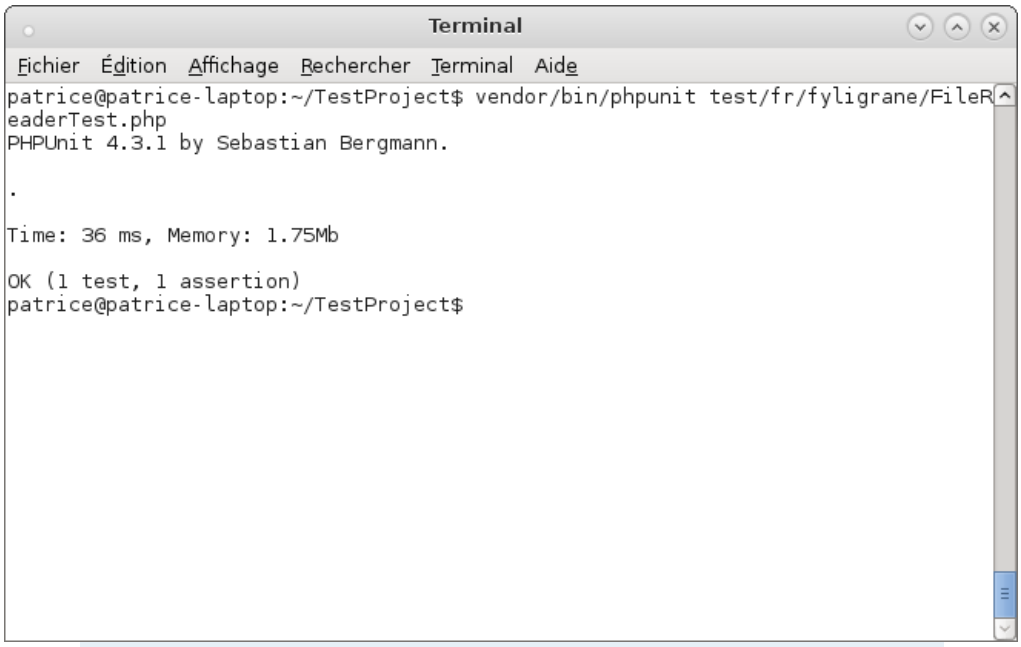
Et l'implémentation de la deuxième solution que nous appellerons Calculatrice2_1 :

```php
<?php

// fr/fyligrane/Calculatrice2_1.php

namespace fr\fyligrane;

class Calculatrice2_1 {

    public function additionner(FileReader $reader) {
        $integers = $reader->getIntegers();
        return array_sum($integers);
    }

}
```

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/fyligrane et, dans le répertoire test, un fichier num.txt, copie du fichier num.txt contenu dans le répertoire fr/fyligrane.

```php
<?php

// test/fr/fyligrane/FileReaderTest.php

include_once './fr/fyligrane/FileReader.php';

use fr\fyligrane\FileReader;

class FileReaderTest extends PHPUnit_Framework_TestCase {

    public $reader;

    public function setUp() {
        $this->reader = new FileReader('test/file/num.txt');
    }

    public function testGetIntegers() {
        $integers = $this->reader->getIntegers();
        $expected = [10, 54, 12, 89, 65, 23, 54];
        $this->assertEquals($expected, $integers);
    }

}
```

Nous lançons le test dans un terminal :



> 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 devrons 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.

```php
<?php

// test/fr/fyligrane/Calculatrice2Test.php

include_once './fr/fyligrane/FileReader.php';
include_once './fr/fyligrane/Calculatrice2.php';

use fr\fyligrane\FileReader;
use fr\fyligrane\Calculatrice2;

class Calculatrice2Test extends PHPUnit_Framework_TestCase {

    public $calc;

    public function setUp() {
        $reader = new FileReader('test/file/num.txt');
        $this->calc = new Calculatrice2($reader);
    }

    public function testAdditionner() {
        $result = $this->calc->additionner();
        $this->assertEquals(307, $result);
    }

}
```

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 devrons 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 :

```php
<?php

// test/fr/fyligrane/Calculatrice2Test.php

include_once './fr/fyligrane/FileReader.php';
include_once './fr/fyligrane/Calculatrice2.php';

use fr\fyligrane\Calculatrice2;

class Calculatrice2Test extends PHPUnit_Framework_TestCase {

    public $calc;

    public function setUp() {
        $reader = $this->getMock('fr\fyligrane\FileReader', []
                , ['test/file/num.txt']);
        $invocationMocker = $reader->expects($this->once());
        $invocationMocker->method('getIntegers')->willReturn([10, 20, 30]);
        $this->calc = new Calculatrice2($reader);
    }

    public function testAdditionner() {
        $result = $this->calc->additionner();
        $this->assertEquals(60, $result);
    }

}
```

Fin

Petite application version 2

Petitie application version 3

Mes cours

ADMINISTRATION

Administration du cours

Réglages de mon profil

Connecté sous le nom « Arnaud Lemais » (Déconnexion)

POO PHP