

# Introduction aux tests unitaires avec PHPUnit 3.1

par Jean-Pierre Grossglauser ([Page personnelle](#))

Date de publication : 15/08/2007

Dernière mise à jour : 15/08/2007

Ce tutoriel fournit une introduction aux tests unitaires PHP avec le framework de tests PHPUnit.

I - Introduction.....	3
I-A - Remerciements.....	3
I-B - À qui s'adresse ce tutoriel ?.....	3
I-C - Prérequis.....	3
I-D - A propos des tests unitaires dans PHP.....	3
I-E - Méthodologie.....	4
II - PHPUnit.....	6
II-A - Installation.....	6
II-B - Utilitaire de ligne de commande.....	6
II-C - Écriture des tests.....	9
II-C-1 - Générateur de gabarit.....	9
II-D - Organisation des tests.....	11
II-E - Exécution des tests.....	12
II-E-1 - Lancer une suite de tests.....	12
II-E-2 - Indicateurs de résultat.....	13
II-F - Exemple récapitulatif.....	13
III - Conclusion.....	17
III-A - Références.....	17

## I - Introduction

### I-A - Remerciements

Mes remerciements à **Guillaume Rossolini** pour ses critiques et sa relecture.

### I-B - À qui s'adresse ce tutoriel ?

Pour aborder ce tutoriel dans les meilleures conditions, vous devez :

- Maîtriser **le paradigme orienté objet PHP 5**.

### I-C - Prérequis

- Bibliothèque PEAR ( **Guide d'installation de PEAR**).
- Pouvoir exécuter PHP en ligne de commande.

### I-D - A propos des tests unitaires dans PHP

Le langage PHP est un langage de programmation flexible, permissif et facile à appréhender. Les principes et contraintes de conception généralement imposées dans les langages à vocation industrielle (C/C++, Java, etc.) ne s'appliquent pas nécessairement avec PHP.

Le programmeur est seul maître à bord, il choisit la précision avec laquelle il souhaite intégrer un paradigme, que ce soit impératif, fonctionnel ou orienté objet, tout comme il décide de structurer ou non son développement par rapport à des processus éprouvés. Cette grande liberté est à l'origine du succès du langage, mais c'est également son plus grand défaut.

Beaucoup de développeurs PHP qui n'ont pas de formation spécifique en programmation ou qui sont peu expérimentés ne perçoivent pas l'importance d'un processus de développement, et de test en particulier.

L'absence d'une forme structurée de tests engendre notamment les problématiques suivantes :

**Le code source n'est pas testé en profondeur** : cela a pour conséquence des aléas de « post-publication », plus ou moins critiques. Le plus souvent il s'agit d'instabilités dans l'application ou des problèmes de sécurité *classiques*.

**Le code source n'est pas robuste** : toute modification du code source (refactorisation, ajout de fonctionnalités) est susceptible d'engendrer des régressions.

**Le code source n'est pas réutilisable, pas transmissible** : si un autre développeur doit vous assister ou reprendre votre travail, il sera confronté d'une part à votre code source et d'autre part à l'absence d'un protocole de test uniformisé.

**Le code source n'est pas évolutif** : il va sans dire, plus votre application aura une structure complexe et plus vous peinerez à déceler des erreurs et problèmes de conception de manière *empirique*. Vous serez contraint à moyen terme, de reprogrammer entièrement votre application.

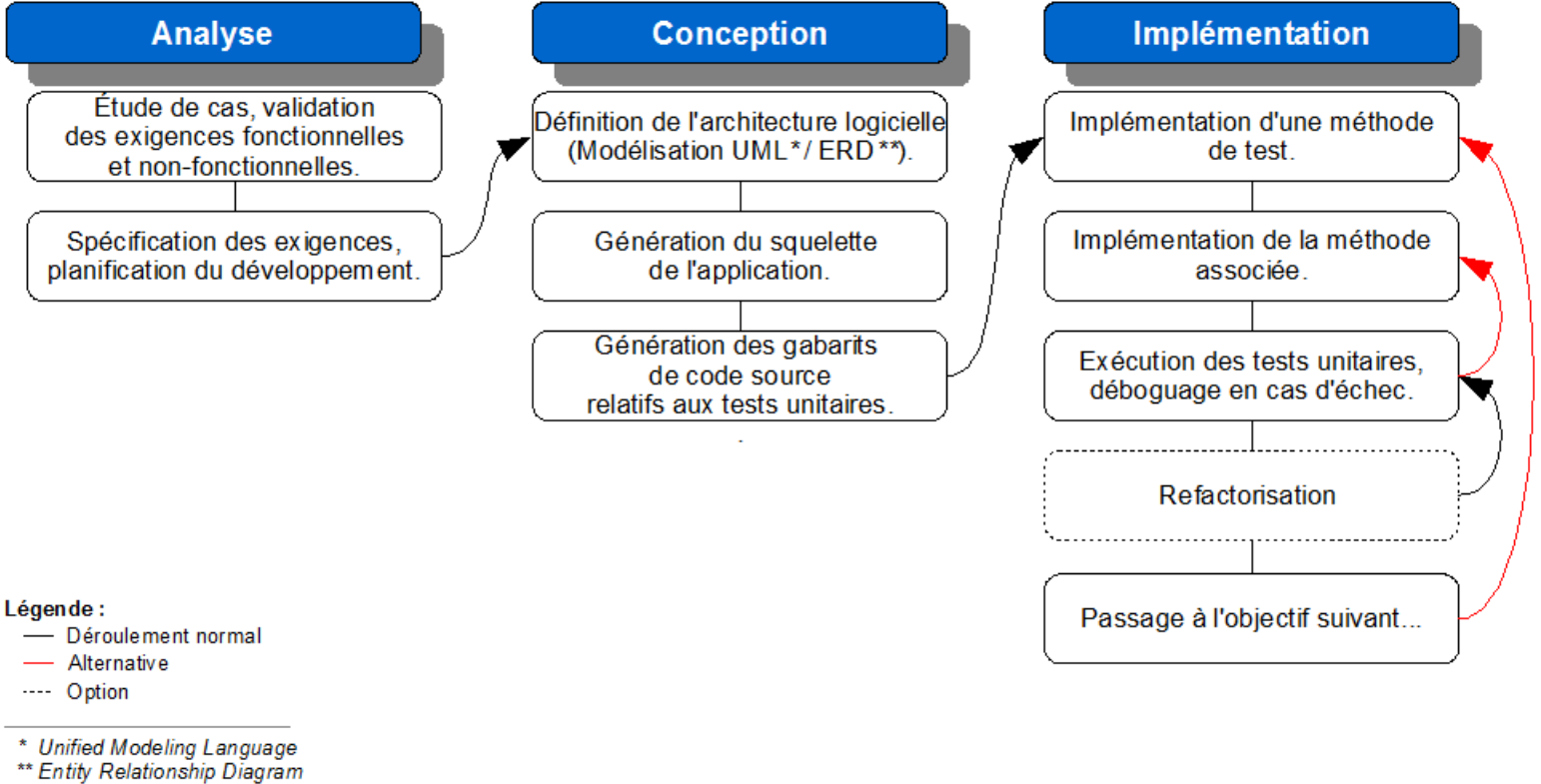
Tout ceci aura non seulement un impact sur la fiabilité de votre programme (et vos prestations de service), mais également sur le temps alloué pour le développement et la maintenance du projet, avec les tracas que cela implique pour vous et vos utilisateurs finaux.

Bien que les tests unitaires soient primordiaux, ils ne résoudre pas les problèmes d'analyse et conception (au mieux, ils les mettront en évidence), c'est pourquoi vous devez garder en tête que la qualité finale de votre produit dépend de votre méthodologie de travail dans son ensemble.

## I-E - Méthodologie

Cette brève introduction méthodologique fournit un processus et quelques conseils vous permettant d'utiliser les tests unitaires dans vos projets PHP. Elle n'a pas pour but d'être exhaustive, si vous souhaitez avoir d'amples informations sur les sujets abordés, consultez les références en fin de chapitre.

### Processus de développement dirigé par les tests en orienté objet (résumé)




Le processus qui vous est proposé ci-dessus fait abstraction des contraintes liées à la gestion de projet (cahier des charges, organisation interne, etc.) ainsi que de la phase d'analyse. Il est basé sur le développement piloté par les tests (*TDD*) et s'adresse particulièrement aux développeurs PHP autonomes.




Si vous développez en équipe, la base restera sensiblement la même mais vous devrez fournir des précisions au niveau de l'organisation et du partage des tâches, notamment.

Le développement piloté par les tests (Test-Driven Development, ou *TDD*) est une méthode de développement mettant les tests unitaires au premier plan, on peut résumer son processus en cinq étapes :

- 1 Ajouter un test,
- 2 vérifier qu'il ne passe pas,
- 3 implémenter la fonctionnalité,
- 4 exécuter les tests unitaires - déboguer en cas d'échec,
- 5 refactoriser le code source - exécuter à nouveau les tests unitaires, passer à l'objectif suivant...

En reprenant le processus ci-dessus, vous observerez que l'approche TDD débute dès la phase de conception. Dans l'idéal, la structure de votre application devrait être modélisée dans un éditeur UML puis son squelette





généralisé directement dans des fichiers PHP. Plusieurs éditeurs UML libres fournissent un générateur de code PHP 5 (nativement ou sous forme de plug-in), notamment  **ArgoUML** (multi-plateforme, notation *UML 1.4*) et  **StarUML** (pour Windows uniquement, notation *UML 2.0*).

Parallèlement, vous pouvez modéliser des diagrammes entité-relation (ERD) et générer les schémas de vos bases de données à la volée. Pour ce faire il existe notamment les logiciels  **DBDesigner 4** (logiciel libre et optimisé pour MySQL),  **MySQL Workbench** (version BETA) ainsi que  **Toad Data Modeler** (version freeware ou commerciale).

Dès que le squelette de votre application sera défini, vous pourrez en faire de même avec les tests unitaires, grâce à PHPUnit et son générateur de gabarits.

Enfin en observant la partie *implémentation* du processus, vous retrouvez les étapes du TDD en bonne et due forme. La refactorisation a été marquée optionnelle car elle pourrait être effectuée durant l'implémentation de la méthode, mais aussi dans le cadre d'une maintenance générale du code source. La refactorisation peut être considérée comme un objectif de développement à part entière.

### D'amples informations :

-  **Cours de conception.**
-  **Tutoriel sur le développement dirigé par les tests** (tutoriel avancé).
-  **Refactorisation.**
-  **Développement de logiciel.**

## II - PHPUnit

PHPUnit est un framework de test unitaires open source.


### Quelques qualités...

- Une syntaxe simple, facile à comprendre et à retenir.
- Un grand nombre de méthodes de tests.
- Organisation et exécution des tests flexibles.
- Un utilitaire en ligne de commande complet.

### Quelques fonctionnalités avancées

- Support des objets "mock" (simulateur d'objets)
- Analyse de la couverture de code (code coverage analyse).
- Support de Selenium RC (tests fonctionnels)
- Journalisation des tests aux formats XML, JSON, TAP ou dans une base de données.

PHPUnit est pris en charge nativement dans les IDE suivants :


- NuSphere PHPEd.
- PHPEdit.
- Zend Studio.
- PHPEclipse (voir plugin  **Eclipse SimpleTest** ).
- Eclipse PDT (le support a été prévu).


## II-A - Installation

Vous pouvez utiliser PEAR :

### Interpréteur de commandes


```
pear channel-discover pear.phpunit.de && pear install phpunit/PHPUnit
```

ou télécharger et installer manuellement le package via :  <http://pear.phpunit.de/get/>

 **PHPUnit 3.1.7** est utilisé dans ce tutoriel.

## II-B - Utilitaire de ligne de commande

L'utilitaire de ligne de commande **phpunit** est le principal outil dédié à la configuration et à l'exécution des tests unitaires.

 *Cet utilitaire est optimisé pour les systèmes Unix/Linux, si vous souhaitez l'utiliser avec **Windows**, vous devez :*





*Définir les variables d'environnement **PHP\_CLI** (chemin vers l'applicatif php.exe) et **PHPUNIT\_HOME** (chemin vers le répertoire de PHPUnit);*

*Créer un fichier \*.bat dénommé **phpunit.bat** et y ajouter l'instruction suivante : %PHP\_CLI % %PHPUNIT\_HOME%\PHPUnit\TextUI\Command.php %\*;*

Indiquer le chemin vers le fichier `phpunit.bat` dans votre variable d'environnement **PATH** (ex. `SET PATH=%PATH%;C:\chemin\vers\executable`).

#### Commande

```
phpunit --help
```

Option	Description
--log-graphviz	Journalise l'exécution du test au format  <b>GraphViz</b>
--log-json	Journalise l'exécution du test au format  <b>JSON</b> .
--log-tap	Journaliser l'exécution du test au format TAP.
--log-xml	Journalise l'exécution du test au format  <b>XML</b> .
--coverage-xml	Génère un rapport de couverture de code (nécessite l'extension PHP Xdebug) au format XML.
--report	Génère un rapport de couverture de code (nécessite l'extension PHP Xdebug) au format HTML.
--test-db-dsn	DSN (Paramètres de connexion) de la base de données de test (Ndlr : Cette option journalise le résultat des tests et éventuellement la couverture de code dans une base de données)
--test-db-log-rev	Information de révision pour la journalisation dans une base de données
--test-db-log-info	Informations additionnelles concernant la journalisation dans une base de données
--testdox-html	Génère  <b>la documentation Agile</b> au format HTML.
--testdox-text	Génère la documentation Agile au format texte.
--testdox-text	Génère un rapport de couverture de code (nécessite l'extension PHP Xdebug).
--filter	Filtre quels tests doivent être lancés (nom ou expression rationnelle).
--loader	Spécifie le chargeur de classe à utiliser.
--repeat	Nombre de répétitions des tests
--tap	Reporte la progression du test au format TAP
--testdox	Reporte la progression du test au format TestDox
--no-syntax-check	Désactive le contrôle syntaxique des fichiers sources.
--stop-on-failure	Interrompt l'exécution du test à la première erreur ou échec.
--verbose	Affiche toutes les informations.
--wait	Une touche doit être pressée entre chaque test.
--skeleton	Génère le gabarit d'une classe de test.
--help	Affiche le récapitulatif des options.
--version	Affiche la version de PHPUnit utilisée.
-d key[=value]	Définit une directive du php.ini

## II-C - Écriture des tests

### II-C-1 - Générateur de gabarit

Le générateur de gabarit (générateur de template, ou encore générateur de squelette) crée des gabarits de code source pour vos tests. Il s'agit de modèles de conception qui vous permettent d'augmenter votre productivité tout en diminuant considérablement les risques d'erreurs liés à l'organisation et à l'exécution des tests.

#### Interpréteur de commandes

```
phpunit --skeleton MyClass
```

Le générateur prendra en compte les méthodes publiques et éventuellement les annotations d'assertion (voir plus bas) qui leur sont associées.



*PHPUnit ne permet pas de tester des méthodes privées ou protégées. Vous devez donc changer explicitement la visibilité de ces dernières pour qu'elles soient prises en compte.*

#### Exemple de génération

```
<?php
// Call MyClassTest::main() if this source file is executed directly.
if (!defined('PHPUnit_MAIN_METHOD')) {
    define('PHPUnit_MAIN_METHOD', 'MyClassTest::main');
}

require_once 'PHPUnit/Framework.php';

require_once 'MyClass.php';

/**
 * Test class for MyClass.
 * Generated by PHPUnit on 2007-08-11 at 20:01:00.
 */
class MyClassTest extends PHPUnit_Framework_TestCase {
    /**
     * Runs the test methods of this class.
     *
     * @access public
     * @static
     */
    public static function main() {
        require_once 'PHPUnit/TextUI/TextRunner.php';

        $suite = new PHPUnit_Framework_TestSuite('MyClassTest');
        $result = PHPUnit_TextUI_TestRunner::run($suite);
    }

    /**
     * Sets up the fixture, for example, opens a network connection.
     * This method is called before a test is executed.
     *
     * @access protected
     */
    protected function setUp() {
    }

    /**
     * Tears down the fixture, for example, closes a network connection.
     * This method is called after a test is executed.
     *
     * @access protected
     */
    protected function tearDown() {
    }
}
```

### Exemple de génération


```

/**
 * @todo Implement testMyMethod().
 */
public function testMyMethod() {
    // Remove the following lines when you implement this test.
    $this->markTestIncomplete(
        'This test has not been implemented yet.'
    );
}

// Call MyClassTest::main() if this source file is executed directly.
if (PHPUnit_MAIN_METHOD == 'MyClassTest::main') {
    MyClassTest::main();
}
?>
    
```

Le générateur fournit une classe de test comportant des méthodes avec l'annotation `@todo Implement` (« À implémenter ») et un appel à `::markTestIncomplete()`, qui permet de marquer une méthode ou une partie de son implémentation comme « incomplète » tant qu'elle n'a pas été précisément remplie par le programmeur.

Vous noterez également les méthodes protégées `::setUp()` et `::tearDown()`; elles permettent de fournir une *fixture*, autrement dit un "contexte d'exécution" commun à toutes les méthodes de test. Dans la pratique, il peut s'agir d'une connexion à une base de données, ou toute autre instruction à vocation collective. La méthode `::setUp()` se comporte en constructeur et s'exécute avant la méthode de test, tandis que `::tearDown()` s'exécute après celle-ci, faisant office de destructeur.

 Les autres structures et notamment la méthode statique `::main()` sont relatives à l'organisation et l'exécution des tests. Elles sont expliquées plus en détails au chapitre suivant.

Il est possible d'intégrer des tests d'assertion directement dans les méthodes qui seront générées, pour ce faire vous devez annoter les méthodes de la classe concrète en utilisant la syntaxe suivante :

```
@assert (argument1 [, argument2, ...]) operator value
```

Structure	Description
@assert	Balise d'assertion
(argument1[,argument2, ...])	Paramètres effectifs de la méthode à tester. Il doit y en avoir au moins un.
operator	Opérateur de comparaison (sera remplacé par la méthode d'assertion qui lui est associé).;
value	La valeur qui sera comparée avec la valeur de retour de la méthode.

Exemple :

```


class MyClass extends PHPUnit_Framework_TestCase
{
    /**
     * @assert (1) > 2
     */
    public function myMethod($value)
    {
        return $value;
    }
}
    
```

Méthode de test générée :

```
/**
 * Generated from @assert (1) > 2.
 */
public function testMyMethod() {
    $constraint = $this->greaterThan(2);
    $object     = new myClass;
    $value      = $object->myMethod(1);
    $this->assertThat($value, $constraint);
}
```

Ces annotations se limitent aux assertions suivantes :

Opérateur	Assertion
==	Equals (égal à)
!=	NotEquals (différent de)
===	Same (strictement égal à)
!==	NotSame (strictement différent de)
>	greaterThan (plus grand que)
>=	greatThanOrEqual (plus grand ou égal à)
<	lessThan (plus petit que)
<=	lessThanOrEqual (plus petit ou égal à)

 *Utilisez cette fonctionnalité avec précaution !*

## II-D - Organisation des tests

PHPUnit offre une solution particulièrement flexible pour organiser vos tests, vous pouvez les exécuter indépendamment les uns des autres, les grouper en suites, ou rassembler les suites elles-mêmes afin d'exécuter le tout en une seule fois.

En reprenant une partie de *l'exemple 1.0*, vous remarquerez que le générateur de gabarit propose plusieurs éléments dédiés à la composition de tests :

```
L'en-tête
if (!defined('PHPUnit_MAIN_METHOD')) {
    define('PHPUnit_MAIN_METHOD', 'MyClassTest::main');
}
```

Cette structure définit la première méthode qui sera exécutée si le fichier est appelé directement. Si la classe de test est incluse dans une suite, la définition sera ignorée.

```
Méthode statique ::main()
public static function main() {
    require_once 'PHPUnit/TextUI/TestRunner.php';

    $suite = new PHPUnit_Framework_TestSuite('MyClassTest');
    $result = PHPUnit_TextUI_TestRunner::run($suite);
}
```

Par défaut, la méthode statique `::main()` fournit une implémentation minimale permettant de lancer le test et éventuellement d'y inclure d'autres éléments (cas de test, suites ou autre instructions).

```
Appel
if (PHPUnit_MAIN_METHOD == 'MyClassTest::main') {
```

### Appel

```
MyClassTest::main();
}
?>
```


Finalement, la structure ci-dessus appellera la méthode définie en en-tête (`::main`, dans le cas précis) si le fichier est exécuté directement.

Pour créer une suite de tests, vous devez instancier la classe `PHPUnit_Framework_TestSuite`, puis ajouter les tests ou suites de tests qui vous intéresse avec les méthodes `::addTest` et (ou) `::addTestSuite`, comme le montre l'exemple ci-dessous :

### Méthode statique ::main()

```
public static function main() {
    require_once 'PHPUnit/TextUI/TestRunner.php';

    $suite = new PHPUnit_Framework_TestSuite('MyClassTest');
    $suite->addTest('MyOtherClassTest');
    $suite->addTestSuite('MyTestSuite');
    $result = PHPUnit_TextUI_TestRunner::run($suite);
}
```

 *En pratique, les instructions relatives aux suites ne devraient pas figurer dans `::main()` mais dans une méthode statique nommée (par convention) `::suite()`; comme dans l'exemple ci-dessous :*

### Méthode statique ::suite()

```
public static function main()
{
    PHPUnit_TextUI_TestRunner::run(self::suite());
}

public static function suite()
{
    $suite = new PHPUnit_Framework_TestSuite('All Tests');
    $suite->addTest(MyOtherClassTest::suite());
}
```

De cette manière, vous obtiendrez un code source mieux décomposé et plus lisible.

## II-E - Exécution des tests

Pour lancer un seul cas de test ou une suite :

### Lancement du cas de test MyClass

```
phpunit MyClass
```

### II-E-1 - Lancer une suite de tests

Pour lancer tous les tests unitaires séquentiellement, vous devez créer une classe principale comme ci-dessous, en y indiquant tous les cas de test et suites qui vous intéressent :

### Classe AllTests

```
<?php

require_once 'PHPUnit/Framework.php';
require_once 'PHPUnit/TextUI/TestRunner.php';
require_once 'MyClass.php';
```

### Classe AllTests

```

require_once 'MyClassTest.php';

if (!defined('PHPUnit_MAIN_METHOD')) {
    define('PHPUnit_MAIN_METHOD', 'AllTests::main');
}


class AllTests
{
    public static function main()
    {
        PHPUnit_TextUI_TestRunner::run(self::suite());
    }

    public static function suite()
    {
        $suite = new PHPUnit_Framework_TestSuite('All Tests');
        $suite->addTest(MyClassTest::main());
    }
}

if (PHPUnit_MAIN_METHOD == 'AllTests::main') {
    AllTests::main();
}

?>

```

 Pour lancer cette classe, vous devrez utiliser directement PHP :

### Interpréteur de commandes - Exécution d'un cas de test

```
php AllTests.php
```

## II-E-2 - Indicateurs de résultat

Un indicateur de résultat est fourni pour chaque méthode de test exécutée :

Indicateur	Description
.	Le test passe.
F	Le test a échoué (Failure).
E	Le test a généré une erreur PHP.
S	Le test est ignoré (Skipped).
I	Le test est marqué comme incomplet (Incomplete).

## II-F - Exemple récapitulatif

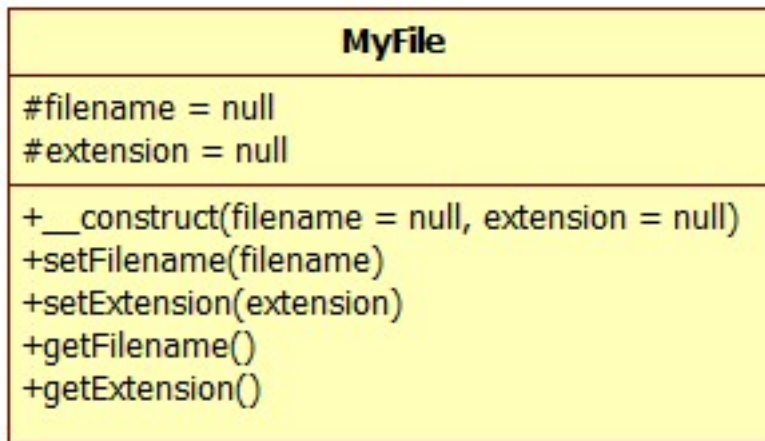
Voici un exemple récapitulatif de test unitaire.

La classe **MyFile** a pour attributs un nom (**fileName**) et une extension (**extension**) de fichier.

Le nom de fichier doit être une chaîne de caractères de 1 à 32 caractères alphanumériques (ASCII), trait d'union et sous-tiret compris. Tous les autres caractères sont proscrits.

L'extension de fichier doit être composée de 1 à 5 caractères alphanumériques (ASCII).

Diagramme de classe UML de *MyFile* :



- 1 Le squelette de la classe **MyFile** est généré à partir du diagramme de classe susmentionné.
- 2 Le gabarit de cas de test **MyFileTest** est généré à partir du squelette de la classe *MyFile*.

Implémentation du cas de test **MyFileTest** :

```

MyFileTest.php
<?php
// Call MyFileTest::main() if this source file is executed directly.
if (!defined('PHPUnit_MAIN_METHOD')) {
    define('PHPUnit_MAIN_METHOD', 'MyFileTest::main');
}

require_once 'PHPUnit/Framework.php';

require_once 'MyFile.php';

/**
 * Test class for MyFile.
 * Generated by PHPUnit on 2007-08-12 at 19:25:50.
 */
class MyFileTest extends PHPUnit_Framework_TestCase {
    /**
     * Runs the test methods of this class.
     *
     * @access public
     * @static
     */
    public static function main() {
        require_once 'PHPUnit/TextUI/TestRunner.php';

        $suite = new PHPUnit_Framework_TestSuite('MyFileTest');
        $result = PHPUnit_TextUI_TestRunner::run($suite);
    }

    public function testSetFileName()
    {
        $file = new MyFile;

        $this->assertTrue($file->setFileName('newfile'));
        $this->assertTrue($file->setFileName('newfile1'));
        $this->assertTrue($file->setFileName('new_file'));
        $this->assertTrue($file->setFileName('new-file'));

        $this->assertFalse($file->setFileName(null));
        $this->assertFalse($file->setFileName('ThisFileNameIsVeryLongTooLongReallyTooLong'));
        $this->assertFalse($file->setFileName('NewFileWith.ext'));
        $this->assertFalse($file->setFileName('àfile'));
        $this->assertFalse($file->setFileName(1));
    }
}

```

## MyFileTest.php

```

}

    public function testSetExtension()
    {
        $file = new MyFile('newFile');

        $this->assertTrue($file->setExtension('txt'));

        $this->assertFalse($file->setExtension(null));
        $this->assertFalse($file->setExtension('abcdef'));
        $this->assertFalse($file->setExtension('é'));
        $this->assertFalse($file->setExtension(1));
    }

    public function testGetFileName()
    {
        $expectedFileName = 'newFile';

        $file = new MyFile($expectedFileName);

        $this->assertEquals($expectedFileName, $file->getFileName());
    }

    public function testGetExtension()
    {
        $expectedExtension = 'txt';

        $file = new MyFile('newFile', $expectedExtension);

        $this->assertEquals($expectedExtension, $file->getExtension());
    }
}

// Call MyFileTest::main() if this source file is executed directly.
if (PHPUnit_MAIN_METHOD == 'MyFileTest::main') {
    MyFileTest::main();
}
?>
    
```

 Implémentation de la classe **MyFile**

## MyFile.php

```

<?php

class MyFile
{
    protected $fileName;

    protected $extension;

    public function __construct($fileName = null, $extension = null)
    {
        if (!is_null($fileName)) {
            $this->setFileName($fileName);
        }

        if (!is_null($extension)) {
            $this->setExtension($extension);
        }
    }

    /**
     * Affecte un nom de fichier (de 1 à 32 caractères alphanumériques, trait d'union et sous-tiret compris).
     * @param string $fileName
     * @return boolean
     */
    public function setFileName($fileName)
    {
        if (!is_string($fileName)) {
            return false;
        }
    }
}
    
```

## MyFile.php

```

    }

    if (!preg_match('/^[a-z0-9-]{1,32}$/i',$fileName)) {
        return false;
    }

    $this->fileName = $fileName;

    return true;
}

/**
 * Affecte une extension de fichier (de 1 à 5 caractères alphanumériques).
 * @param string $extension
 * @return boolean
 */
public function setExtension($extension)
{
    if (!is_string($extension)) {
        return false;
    }

    if (!preg_match('/^[a-z0-9]{1,5}$/i',$extension)) {
        return false;
    }

    $this->extension = $extension;

    return true;
}

/**
 * Retourne le nom de fichier.
 *
 * @return string
 */
public function getFilename()
{
    return $this->fileName;
}

/**
 * Retourne l'extension du fichier
 *
 * @return string
 */
public function getExtension()
{
    return $this->extension;
}
}

?>

```

Exécution du test unitaire :

## Interpréteur de commandes

```


phpunit MyFileTest
PHPUnit 3.1.7 by Sebastian Bergmann.

....

Time: 0 seconds

OK (4 tests)

```

 *PHPUnit fournit quelques exemples complets de test unitaires, ils sont disponibles dans le répertoire **PHPUnit/Samples**.*

## III - Conclusion

PHPUnit est un framework de test simple et efficace, son éventail de fonctionnalités permet de créer rapidement des tests unitaires complets et adaptés aux applications PHP professionnelles.

### III-A - Références

-  [Site officiel de PHPUnit.](#)

#### Ressources Developpez.com :

-  [PHPEclipse : Programmez librement pour le Web](#), par Jean-Pierre Grossglauser.
-  [PHPEdit, un IDE complet pour PHP](#), par Guillaume Rossolini.
-  [Compte-rendu de la Conférence Internationale PHP 2006](#), par Guillaume Rossolini.