Manuel PHPUnit

Sebastian Bergmann

Manuel PHPUnit

Sebastian Bergmann

Date de publication Edition pour PHPUnit 6.5 mise à jour le 2017-12-07.

Table des matières

Liste des tableaux

Liste des exemples

Chapitre 1. Installer PHPUnit

Pré-requis

PHPUnit 6.5 nécessite PHP 7; utiliser la dernière version de PHP est fortement recommandé.

PHPUnit nécessite les extensions dom [http://php.net/manual/en/dom.setup.php] et json [http://php.net/manual/en/json.installation.php] qui sont traditionnellement activées par défaut.

PHPUnit nécessite aussi les extensions pcre [http://php.net/manual/en/pcre.installation.php], reflection [http://php.net/manual/en/reflection.installation.php], et spl [http://php.net/manual/en/spl.installation.php]. Ces extensions standard sont activées par défaut et ne peuvent être désactivées sans patcher le système de construction de PHP ou/et les sources en C.

La fonctionnalité de couverture de code nécessite l'extension Xdebug [http://xdebug.org/] (2.5.0 ou ultérieur) et tokenizer [http://php.net/manual/en/tokenizer.installation.php]. La génération des rapports XML nécessite l'extension xmlwriter [http://php.net/manual/en/xmlwriter.installation.php].

PHP Archive (PHAR)

La manière la plus simple d'obtenir PHPUnit est de téléchargerl'Archive PHP (PHAR) [http://php.net/phar] qui contient toutes les dépendances requises (ainsi que certaines optionnelles) de PHPUnit archivées en un seul fichier.

L'extension phar [http://php.net/manual/en/phar.installation.php] est requise pour utilisé les archives PHP (PHAR).

Si l'extension Suhosin [http://suhosin.org/] est activé, vous devez autoriser l'exécution des PHAR dans votre `php.ini`:

```
suhosin.executor.include.whitelist = phar
```

Pour installer le PHAR de manière globale :

```
$ wget https://phar.phpunit.de/phpunit-6.5.phar
$ chmod +x phpunit-6.5.phar
$ sudo mv phpunit-6.5.phar /usr/local/bin/phpunit
$ phpunit --version
PHPUnit x.y.z by Sebastian Bergmann and contributors.
```

Vous pouvez également utiliser directement le fichier PHAR téléchargé:

```
$ wget https://phar.phpunit.de/phpunit-6.5.phar
$ php phpunit-6.5.phar --version
PHPUnit x.y.z by Sebastian Bergmann and contributors.
```

Windows

L'installation globale du PHAR implique la même procédure que l'installation manuelle de Composer sous Windows [https://getcomposer.org/doc/00-intro.md#installation-windows]:

1. Créer un répertoire pour les binaires PHP; ex. : `C:\bin`
2. Ajouter **`;C:\bin`** à votre variable d'environnement `PATH` (related help [http://stackoverflow.com/questions/6318156/adding-python-path-on-windows-7])

3. Télécharger https://phar.phpunit.de/phpunit-6.5.phar et sauvegarder le fichier sous `C:\bin\phpunit.phar`

4. Ouvrir une ligne de commande (par exemple, appuyez **Windows**+**R** » et tapez **`cmd`** » **ENTER**)

5. Créer un script batch (dans `C:\bin\phpunit.cmd`):

```
C:\Users\username> cd C:\bin
C:\bin> echo @php "%~dp0phpunit.phar" %* > phpunit.cmd
C:\bin> exit
```

6. Ouvrez une nouvelle ligne de commande et confirmez que vous pouvez exécuter PHPUnit à partir de n'importe quel chemin:

```
C:\Users\username> phpunit --version
PHPUnit x.y.z by Sebastian Bergmann and contributors.
```

Pour les environments shell Cygwin et/ou MingW32 (ex: TortoiseGit), vous passer l'étape 5. ci-dessus, il suffit de sauvegarder le fichier `phpunit` (sans l'extension `.phar`), et de le rendre executable via **`chmod 775 phpunit`**.

Vérification des versions PHAR de PHPUnit

Toutes les versions officielles de code distribuées par le projet PHPUnit sont signées par le responsable de publication de la version. Les signatures PGP et les hachages SHA1 sont disponibles pour vérification sur phar.phpunit.de [https://phar.phpunit.de/].

L'exemple suivant détaille le fonctionnement de la vérification de version. Nous commençons par tlécharger `phpunit.phar` ainsi que sa signature PGP détachée `phpunit.phar.asc`:

```
wget https://phar.phpunit.de/phpunit.phar
wget https://phar.phpunit.de/phpunit.phar.asc
```

Nous voulons vérifier l'archive PHP Phar de PHPUnit (`phpunit.phar`) par rapport à sa signature détachée (`phpunit.phar.asc`):

```
gpg phpunit.phar.asc
gpg: Signature made Sat 19 Jul 2014 01:28:02 PM CEST using RSA key ID 6372C20A
gpg: Can't check signature: public key not found
```

Nous n'avons pas la clé publique du responsable de la publication (`6372C20A`) dans notre système local. Afin de procéder à la vérification, nous devons récupérer la clé publique du gestionnaire de versions à partir d'un serveur de clés. Un de ces serveurs est `pgp.uni-mainz.de`. Les serveurs de clés publiques sont liés entre eux, vous devriez donc pouvoir vous connecter à n'importe quel serveur de clés.

```
gpg --keyserver pgp.uni-mainz.de --recv-keys 0x4AA394086372C20A
gpg: requesting key 6372C20A from hkp server pgp.uni-mainz.de
gpg: key 6372C20A: public key "Sebastian Bergmann <sb@sebastian-bergmann.de>" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
```

Nous avons maintenant reçu une clé publique pour une entité appelée "Sebastian Bergmann <sb@sebastian-bergmann.de>". Cependant, nous n'avons aucun moyen de vérifier que cette clé a été créée par la personne connue sous le nom de Sebastian Bergmann. Mais, essayons de vérifier à nouveau la signature de la version délivrée.

```
gpg phpunit.phar.asc
gpg: Signature made Sat 19 Jul 2014 01:28:02 PM CEST using RSA key ID 6372C20A
```

```
gpg: Good signature from "Sebastian Bergmann <sb@sebastian-bergmann.de>"
gpg:                 aka "Sebastian Bergmann <sebastian@php.net>"
gpg:                 aka "Sebastian Bergmann <sebastian@thephp.cc>"
gpg:                 aka "Sebastian Bergmann <sebastian@phpunit.de>"
gpg:                 aka "Sebastian Bergmann <sebastian.bergmann@thephp.cc>"
gpg:                 aka "[jpeg image of size 40635]"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: D840 6D0D 8294 7747 2937  7831 4AA3 9408 6372 C20A
```

À ce stade, la signature est bonne, mais nous ne faisons pas confiance à cette clé. Une bonne signature signifie que le fichier n'a pas été falsifié. Cependant, en raison de la nature de la cryptographie à clé publique, vous devez également vérifier que la clé `6372C20A` a été créée par le vrai Sebastian Bergmann.

Tout attaquant peut créer une clé publique et l'uploader sur les serveurs de clés publiques. Ils peuvent ensuite créer une version malveillante signée par cette fausse clé. Ensuite, si vous essayiez de vérifier la signature de cette version corrompue, cela réussirait car la clé n'était pas la "vraie" clé. Par conséquent, vous devez valider l'authenticité de cette clé. La validation de l'authenticité d'une clé publique est toutefois hors de la portée de cette documentation.

Il peut être prudent de créer un script shell pour gérer l'installation de PHPUnit qui vérifie la signature de GnuPG avant d'exécuter votre suite de tests. Par exemple:

```
#!/usr/bin/env bash
clean=1 # Delete phpunit.phar after the tests are complete?
aftercmd="php phpunit.phar --bootstrap bootstrap.php src/tests"
gpg --fingerprint D8406D0D82947747293778314AA394086372C20A
if [ $? -ne 0 ]; then
    echo -e "\033[33mDownloading PGP Public Key...\033[0m"
    gpg --recv-keys D8406D0D82947747293778314AA394086372C20A
    # Sebastian Bergmann <sb@sebastian-bergmann.de>
    gpg --fingerprint D8406D0D82947747293778314AA394086372C20A
    if [ $? -ne 0 ]; then
        echo -e "\033[31mCould not download PGP public key for verification\033[0m"
        exit
    fi
fi

if [ "$clean" -eq 1 ]; then
    # Let's clean them up, if they exist
    if [ -f phpunit.phar ]; then
        rm -f phpunit.phar
    fi
    if [ -f phpunit.phar.asc ]; then
        rm -f phpunit.phar.asc
    fi
fi

# Let's grab the latest release and its signature
if [ ! -f phpunit.phar ]; then
    wget https://phar.phpunit.de/phpunit.phar
fi
if [ ! -f phpunit.phar.asc ]; then
    wget https://phar.phpunit.de/phpunit.phar.asc
fi

# Verify before running
gpg --verify phpunit.phar.asc phpunit.phar
if [ $? -eq 0 ]; then
    echo
    echo -e "\033[33mBegin Unit Testing\033[0m"
    # Run the testing suite
```

```
    `$after_cmd`
    # Cleanup
    if [ "$clean" -eq 1 ]; then
        echo -e "\033[32mCleaning Up!\033[0m"
        rm -f phpunit.phar
        rm -f phpunit.phar.asc
    fi
else
    echo
    chmod -x phpunit.phar
    mv phpunit.phar /tmp/bad-phpunit.phar
    mv phpunit.phar.asc /tmp/bad-phpunit.phar.asc
    echo -e "\033[31mSignature did not match! PHPUnit has been moved to /tmp/bad-phpunit
    exit 1
fi
```

Composer

Ajoutez simplement une dépendance (au développement) à `phpunit/phpunit` au fichier `composer.json` de votre projet si vous utilisez Composer [https://getcomposer.org/] pour gérer les dépendances de votre projet :

```
composer require --dev phpunit/phpunit ^6.5
```

Paquets optionnels

Les packages optionnels suivants sont disponibles:

`PHP_Invoker`

Une classe d'utilitaire pour invoquer des appelables avec un délai d'expiration. Ce package est requis pour appliquer les délais d'attente de test en mode strict.

Ce package est inclus dans la distribution PHAR de PHPUnit. Il peut être installée via Composer en utilisant la commande suivate:

```
composer require --dev phpunit/php-invoker
```

`DbUnit`

Portage DbUnit pour PHP/PHPUnit pour prendre en charge le test d'interaction de base de données.

Ce package n'est pas inclus dans la distribution PHAR de PHPUnit. Il peut être installée via Composer en utilisant la commande suivante:

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

Chapitre 2. Écrire des tests pour PHPUnit

Exemple 2.1, « Tester des opérations de tableau avec PHPUnit » montre comment nous pouvons écrire des tests en utilisant PHPUnit pour contrôler les opérations PHP sur les tableaux. L'exemple introduit les conventions et les étapes de base pour écrire des tests avec PHPUnit:

1. Les tests pour une classe `Class` vont dans une classe `ClassTest`.
2. `ClassTest` hérite (la plupart du temps) de `PHPUnit\Framework\TestCase`.
3. Les tests sont des méthodes publiques qui sont appelées `test*`.

   Alternativement, vous pouvez utiliser l'annotation `@test` dans le bloc de documentation d'une méthode pour la marquer comme étant une méthode de test.
4. A l'intérieur des méthodes de test, des méthodes d'assertion telles que `assertEquals()` (voir Annexe A, *Assertions*) sont utilisées pour affirmer qu'une valeur constatée correspond à une valeur attendue.

Exemple 2.1. Tester des opérations de tableau avec PHPUnit

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

class StackTest extends TestCase
{
    public function testPushAndPop()
    {
        $stack = [];
        $this->assertEquals(0, count($stack));

        array_push($stack, 'foo');
        $this->assertEquals('foo', $stack[count($stack)-1]);
        $this->assertEquals(1, count($stack));

        $this->assertEquals('foo', array_pop($stack));
        $this->assertEquals(0, count($stack));
    }
}
?>
```

> A chaque fois que vous avez la tentation de saisir quelque chose dans une instruction `print` ou dans une expression de débogage, écrivez le plutôt dans un test.
>
> —Martin Fowler

Dépendances des tests

> Les tests unitaires sont avant tout écrits comme étant une bonne pratique destinée à aider les développeurs à identifier et corriger les bugs, à refactoriser le code et à servir de documentation pour une unité du logiciel testé. Pour obtenir ces avantages, les tests unitaires doivent idéalement couvrir tous les chemins possibles du programme. Un test unitaire couvre usuellement un unique chemin particulier d'une seule fonction ou méthode. Cependant, une méthode de test n'est pas obligatoirement une entité encapsulée et indépendante. Souvent, il existe des dépendances implicites entre les méthodes de test, cachées dans l'implémentation du scénario d'un test.
>
> —Adrian Kuhn et. al.

PHPUnit gère la déclaration de dépendances explicites entre les méthodes de test. De telles dépendances ne définissent pas l'ordre dans lequel les méthodes de test doivent être exécutées mais elles permettent de renvoyer une instance de la fixture de test par un producteur à des consommateurs qui en dépendent.

- Un producteur est une méthode de test qui produit ses éléments testées comme valeur de retour.
- Un consommateur est une méthode de test qui dépend d'un ou plusieurs producteurs et de leurs valeurs de retour.

Exemple 2.2, « Utiliser l'annotation `@depends` pour exprimer des dépendances » montre comment utiliser l'annotation `@depends` pour exprimer des dépendances entre des méthodes de test.

Exemple 2.2. Utiliser l'annotation `@depends` pour exprimer des dépendances

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

class StackTest extends TestCase
{
    public function testEmpty()
    {
        $stack = [];
        $this->assertEmpty($stack);

        return $stack;
    }

    /**
     * @depends testEmpty
     */
    public function testPush(array $stack)
    {
        array_push($stack, 'foo');
        $this->assertEquals('foo', $stack[count($stack)-1]);
        $this->assertNotEmpty($stack);

        return $stack;
    }

    /**
     * @depends testPush
     */
    public function testPop(array $stack)
    {
        $this->assertEquals('foo', array_pop($stack));
        $this->assertEmpty($stack);
    }
}
?>
```

Dans l'exemple ci-dessus, le premier test, `testEmpty()`, crée un nouveau tableau et affirme qu'il est vide. Le test renvoie ensuite la fixture comme résultat. Le deuxième test, `testPush()`, dépend de `testEmpty()` et reçoit le résultat de ce test dont il dépend comme argument. Enfin, `testPop()` dépend de `testPush()`.

Note

La valeur de retour produite par un producteur est passée "telle quelle" à son consommateur par défaut. Cela signifie que lorsqu'un producteur renvoie un objet, une référence vers cet objet est passée a son consommateur. Lorsqu'une copie doit être utilisée au lieu d'une référence, alors `@depends clone` doit être utilisé au lieu de `@depends`.

Pour localiser rapidement les défauts, nous voulons que notre attention soit retenue par les tests en échecs pertinents. C'est pourquoi PHPUnit saute l'exécution d'un test quand un test dont il dépend a échoué. Ceci améliore la localisation des défauts en exploitant les dépendances entre les tests comme montré dans Exemple 2.3, « Exploiter les dépendances entre les tests ».

Exemple 2.3. Exploiter les dépendances entre les tests

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

class DependencyFailureTest extends TestCase
{
    public function testOne()
    {
        $this->assertTrue(false);
    }

    /**
     * @depends testOne
     */
    public function testTwo()
    {
    }
}
?>
```

```
phpunit --verbose DependencyFailureTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

FS

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) DependencyFailureTest::testOne
Failed asserting that false is true.

/home/sb/DependencyFailureTest.php:6

There was 1 skipped test:

1) DependencyFailureTest::testTwo
This test depends on "DependencyFailureTest::testOne" to pass.


FAILURES!
Tests: 1, Assertions: 1, Failures: 1, Skipped: 1.
```

Un test peut avoir plusieurs annotations `@depends`. PHPUnit ne change pas l'ordre dans lequel les tests sont exécutés, vous devez donc vous assurer que les dépendances d'un test peuvent effectivement être utilisables avant que le test ne soit lancé.

Un test qui a plusieurs annotations `@depends` prendra une fixture du premier producteur en premier argument, une fixture du second producteur en second argument, et ainsi de suite. Voir Exemple 2.4, « Test avec plusieurs dépendances »

Exemple 2.4. Test avec plusieurs dépendances

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

```
class MultipleDependenciesTest extends TestCase
{
    public function testProducerFirst()
    {
        $this->assertTrue(true);
        return 'first';
    }

    public function testProducerSecond()
    {
        $this->assertTrue(true);
        return 'second';
    }

    /**
     * @depends testProducerFirst
     * @depends testProducerSecond
     */
    public function testConsumer()
    {
        $this->assertEquals(
            ['first', 'second'],
            func_get_args()
        );
    }
}
?>
```

```
phpunit --verbose MultipleDependenciesTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

...

Time: 0 seconds, Memory: 3.25Mb

OK (3 tests, 3 assertions)
```

Fournisseur de données

Une méthode de test peut recevoir des arguments arbitraires. Ces arguments doivent être fournis par une méthode fournisseuse de données (`additionProvider()` dans Exemple 2.5, « Utiliser un fournisseur de données qui renvoie un tableau de tableaux »). La méthode fournisseuse de données à utiliser est indiquée dans l'annotation `@dataProvider`.

Une méthode fournisseuse de données doit être `public` et retourne, soit un tableau de tableaux, soit un objet qui implémente l'interface `Iterator` et renvoie un tableau pour chaque itération. Pour chaque tableau qui est une partie de l'ensemble, la méthode de test sera appelée avec comme arguments le contenu du tableau.

Exemple 2.5. Utiliser un fournisseur de données qui renvoie un tableau de tableaux

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

class DataTest extends TestCase
{
    /**
     * @dataProvider additionProvider
```

```
     */
    public function testAdd($a, $b, $expected)
    {
        $this->assertEquals($expected, $a + $b);
    }

    public function additionProvider()
    {
        return [
            [0, 0, 0],
            [0, 1, 1],
            [1, 0, 1],
            [1, 1, 3]
        ];
    }
}
?>
```

```
phpunit DataTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

...F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

1) DataTest::testAdd with data set #3 (1, 1, 3)
Failed asserting that 2 matches expected 3.

/home/sb/DataTest.php:9

FAILURES!
Tests: 4, Assertions: 4, Failures: 1.
```

Lorsque vous utilisez un grand nombre de jeux de données, il est utile de nommer chacun avec une clé en chaine de caractère au lieu de la valeur numérique par défaut. La sortie sera plus verbeuse car elle contiendra le nom du jeu de données qui casse un test.

Exemple 2.6. Utiliser un fournisseur de données avec des jeux de données nommés

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

class DataTest extends TestCase
{
    /**
     * @dataProvider additionProvider
     */
    public function testAdd($a, $b, $expected)
    {
        $this->assertEquals($expected, $a + $b);
    }

    public function additionProvider()
    {
        return [
            'adding zeros'  => [0, 0, 0],
            'zero plus one' => [0, 1, 1],
            'one plus zero' => [1, 0, 1],
            'one plus one'  => [1, 1, 3]
```

```
        ];
    }
}
?>
```

```
phpunit DataTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

...F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

1) DataTest::testAdd with data set "one plus one" (1, 1, 3)
Failed asserting that 2 matches expected 3.

/home/sb/DataTest.php:9

FAILURES!
Tests: 4, Assertions: 4, Failures: 1.
```

Exemple 2.7. Utiliser un fournisseur de données qui renvoie un objet Iterator

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

require 'CsvFileIterator.php';

class DataTest extends TestCase
{
    /**
     * @dataProvider additionProvider
     */
    public function testAdd($a, $b, $expected)
    {
        $this->assertEquals($expected, $a + $b);
    }

    public function additionProvider()
    {
        return new CsvFileIterator('data.csv');
    }
}
?>
```

```
phpunit DataTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

...F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

1) DataTest::testAdd with data set #3 ('1', '1', '3')
Failed asserting that 2 matches expected '3'.

/home/sb/DataTest.php:11

FAILURES!
Tests: 4, Assertions: 4, Failures: 1.
```

Exemple 2.8. La classe CsvFileIterator

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

class CsvFileIterator implements Iterator {
    protected $file;
    protected $key = 0;
    protected $current;

    public function __construct($file) {
        $this->file = fopen($file, 'r');
    }

    public function __destruct() {
        fclose($this->file);
    }

    public function rewind() {
        rewind($this->file);
        $this->current = fgetcsv($this->file);
        $this->key = 0;
    }

    public function valid() {
        return !feof($this->file);
    }

    public function key() {
        return $this->key;
    }

    public function current() {
        return $this->current;
    }

    public function next() {
        $this->current = fgetcsv($this->file);
        $this->key++;
    }
}
?>
```

Quand un test reçoit des entrées à la fois d'une méthode `@dataProvider` et d'un ou plusieurs tests dont il `@depends`, les arguments provenant du fournisseur de données arriveront avant ceux des tests dont il dépend. Les arguments des tests dépendants seront les mêmes pour chaque jeux de données Voir Exemple 2.9, « Combinaison de @depends et @dataProvider dans le même test »

Exemple 2.9. Combinaison de @depends et @dataProvider dans le même test

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

class DependencyAndDataProviderComboTest extends TestCase
{
    public function provider()
    {
        return [['provider1'], ['provider2']];
    }

    public function testProducerFirst()
    {
        $this->assertTrue(true);
```

```
        return 'first';
    }

    public function testProducerSecond()
    {
        $this->assertTrue(true);
        return 'second';
    }

    /**
     * @depends testProducerFirst
     * @depends testProducerSecond
     * @dataProvider provider
     */
    public function testConsumer()
    {
        $this->assertEquals(
            ['provider1', 'first', 'second'],
            func_get_args()
        );
    }
}
?>
```

```
phpunit --verbose DependencyAndDataProviderComboTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

...F

Time: 0 seconds, Memory: 3.50Mb

There was 1 failure:

1) DependencyAndDataProviderComboTest::testConsumer with data set #1 ('provider2')
Failed asserting that two arrays are equal.
--- Expected
+++ Actual
@@ @@
Array (
-    0 => 'provider1'
+    0 => 'provider2'
1 => 'first'
2 => 'second'
)

/home/sb/DependencyAndDataProviderComboTest.php:31

FAILURES!
Tests: 4, Assertions: 4, Failures: 1.
```

Note

Quand un test dépend d'un test qui utilise des fournisseurs de données, le test dépendant sera exécuté quand le test dont il dépend réussira pour au moins un jeu de données. Le résultat d'un test qui utilise des fournisseurs de données ne peut pas être injecté dans un test dépendant.

Note

Tous les fournisseurs de données sont exécutés avant le premier appel à la méthode statique `setUpBeforeClass` et le premier appel à la méthode `setUp`. De ce fait, vous ne pouvez accéder à aucune variable créée à ces endroits depuis un fournisseur de données. Ceci est requis pour que PHPUnit puisse calculer le nombre total de tests.

Tester des exceptions

Exemple 2.10, « Utiliser la méthode expectException() » montre comment utiliser la méthode `expectException()` pour tester si une exception est levée par le code testé.

Exemple 2.10. Utiliser la méthode expectException()

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

class ExceptionTest extends TestCase
{
    public function testException()
    {
        $this->expectException(InvalidArgumentException::class);
    }
}
?>
```

```
phpunit ExceptionTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ExceptionTest::testException
Expected exception InvalidArgumentException

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

En complément à la méthode `expectException()` les méthodes `expectExceptionCode()`, `expectExceptionMessage()` et `expectExceptionMessageRegExp()` existent pour établir des attentes pour les exceptions levées par le code testé.

Alternativement, vous pouvez utiliser les annotations `@expectedException`, `@expectedExceptionCode`, `@expectedExceptionMessage` et `@expectedExceptionMessageRegExp` pour établir des attentes pour les exceptions levées par le code testé. Exemple 2.11, « Utiliser l'annotation @expectedException » montre un exemple.

Exemple 2.11. Utiliser l'annotation @expectedException

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

class ExceptionTest extends TestCase
{
    /**
     * @expectedException InvalidArgumentException
     */
    public function testException()
    {
    }
}
?>
```

```
phpunit ExceptionTest
```

```
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ExceptionTest::testException
Expected exception InvalidArgumentException

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Tester les erreurs PHP

Par défaut, PHPUnit convertit les erreurs, avertissements et remarques PHP qui sont émises lors de l'exécution d'un test en exception. En utilisant ces exceptions, vous pouvez, par exemple, attendre d'un test qu'il produise une erreur PHP comme montré dans Exemple 2.12, « Attendre une erreur PHP en utilisant @expectedException ».

Note

La configuration d'exécution PHP `error_reporting` peut limiter les erreurs que PHPUnit convertira en exceptions. Si vous rencontrez des problèmes avec cette fonctionnalité, assurez-vous que PHP n'est pas configuré pour supprimer le type d'erreurs que vous testez.

Exemple 2.12. Attendre une erreur PHP en utilisant @expectedException

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

class ExpectedErrorTest extends TestCase
{
    /**
      @expectedException PHPUnit\Framework\Error
     */
    public function testFailingInclude()
    {
        include 'not_existing_file.php';
    }
}
?>
```

```
phpunit -d error_reporting=2 ExpectedErrorTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

.

Time: 0 seconds, Memory: 5.25Mb

OK (1 test, 1 assertion)
```

`PHPUnit\Framework\Error\Notice` et `PHPUnit\Framework\Error\Warning` représentent respectivement les remarques et les avertissements PHP.

Note

Vous devriez être aussi précis que possible lorsque vous testez des exceptions. Tester avec des classes qui sont trop génériques peut conduire à des effets de bord indésirables. C'est

pourquoi tester la présence de la classe `Exception` avec `@expectedException` ou `setExpectedException()` n'est plus autorisé.

Quand les tests s'appuient sur des fonctions php qui déclenchent des erreurs comme `fopen`, il peut parfois être utile d'utiliser la suppression d'erreur lors du test. Ceci permet de contrôler les valeurs de retour en supprimant les remarques qui auraient conduit à une `PHPUnit\Framework\Error\Notice` de phpunit.

Exemple 2.13. Tester des valeurs de retour d'un code source qui utilise des erreurs PHP

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

class ErrorSuppressionTest extends TestCase
{
    public function testFileWriting() {
        $writer = new FileWriter;
        $this->assertFalse(@$writer->write('/is-not-writeable/file', 'stuff'));
    }
}
class FileWriter
{
    public function write($file, $content) {
        $file = fopen($file, 'w');
        if($file == false) {
            return false;
        }
        // ...
    }
}

?>
```

```
phpunit ErrorSuppressionTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

.

Time: 1 seconds, Memory: 5.25Mb

OK (1 test, 1 assertion)
```

Sans la suppression d'erreur, le test échouerait à rapporter `fopen(/is-not-writeable/file): failed to open stream: No such file or directory`.

Tester la sortie écran

Quelquefois, vous voulez confirmer que l'exécution d'une méthode, par exemple, produit une sortie écran donnée (via `echo` ou `print`, par exemple). La classe `PHPUnit\Framework\TestCase` utilise la fonctionnalité de mise en tampon de la sortie écran [http://www.php.net/manual/en/ref.outcontrol.php] de PHP pour fournir la fonctionnalité qui est nécessaire pour cela.

Exemple 2.14, « Tester la sortie écran d'une fonction ou d'une méthode » montre comment utiliser la méthode `expectOutputString()` pour indiquer la sortie écran attendue. Si la sortie écran attendue n'est pas générée, le test sera compté comme étant en échec.

Exemple 2.14. Tester la sortie écran d'une fonction ou d'une méthode

```
<?php
```

```
use PHPUnit\Framework\TestCase;

class OutputTest extends TestCase
{
    public function testExpectFooActualFoo()
    {
        $this->expectOutputString('foo');
        print 'foo';
    }

    public function testExpectBarActualBaz()
    {
        $this->expectOutputString('bar');
        print 'baz';
    }
}
?>
```

```
phpunit OutputTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

.F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

1) OutputTest::testExpectBarActualBaz
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'bar'
+'baz'


FAILURES!
Tests: 2, Assertions: 2, Failures: 1.
```

Tableau 2.1, « Méthodes pour tester les sorties écran » montre les méthodes fournies pour tester les sorties écran

Tableau 2.1. Méthodes pour tester les sorties écran

Méthode	Signification
`void expectOutputRegex(string $regularExpression)`	Indique que l'on s'attend à ce que la sortie écran corresponde à une expression régulière `$regularExpression`.
`void expectOutputString(string $attenduString)`	Indique que l'on s'attend que la sortie écran soit égale à une chaine de caractère `$expectedString`.
`bool setOutputCallback(callable $callback)`	Configure une fonction de rappel (callback) qui est utilisée, par exemple, formater la sortie écran effective.
`string getActualOutput()`	Renvoi la sortie écran courrante.

Note

En mode strict, un test qui produit une sortie écran échouera.

Sortie d'erreur

Chaque fois qu'un test échoue, PHPUnit essaie de vous fournir le plus de contexte possible pour identifier le problème.

Exemple 2.15. Sortie d'erreur générée lorsqu'un échec de comparaison de tableau

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

class ArrayDiffTest extends TestCase
{
    public function testEquality() {
        $this->assertEquals(
            [1, 2,  3, 4, 5, 6],
            [1, 2, 33, 4, 5, 6]
        );
    }
}
?>
```

```
phpunit ArrayDiffTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) ArrayDiffTest::testEquality
Failed asserting that two arrays are equal.
--- Expected
+++ Actual
@@ @@
 Array (
     0 => 1
     1 => 2
-    2 => 3
+    2 => 33
     3 => 4
     4 => 5
     5 => 6
 )

/home/sb/ArrayDiffTest.php:7

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Dans cet exemple, une seule des valeurs du tableau diffère et les autres valeurs sont affichées pour fournir un contexte sur l'endroit où l'erreur s'est produite.

Lorsque la sortie générée serait longue à lire, PHPUnit la divisera et fournira quelques lignes de contexte autour de chaque différence.

Exemple 2.16. Sortie d'erreur quand une comparaison de long tableaux échoue

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

```
class LongArrayDiffTest extends TestCase
{
    public function testEquality() {
        $this->assertEquals(
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2,  3, 4, 5, 6],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 33, 4, 5, 6]
        );
    }
}
?>
```

```
phpunit LongArrayDiffTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) LongArrayDiffTest::testEquality
Failed asserting that two arrays are equal.
--- Expected
+++ Actual
@@ @@
     13 => 2
-    14 => 3
+    14 => 33
     15 => 4
     16 => 5
     17 => 6
 )


/home/sb/LongArrayDiffTest.php:7

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Cas limite

Quand une comparaison échoue, PHPUnit crée une représentation textuel des valeurs d'entrées et les compare. A cause de cette implémentation un diff peut montrer plus de problèmes qu'il n'en existe réellement.

Cela arrive seulement lors de l'utilisation de assetEquals ou d'autres fonction de comparaison "faible" sur les tableaux ou les objets.

Exemple 2.17. Cas limite dans la génération de la différence lors de l'utilisation de comparaison faible

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

class ArrayWeakComparisonTest extends TestCase
{
    public function testEquality() {
        $this->assertEquals(
            [1, 2, 3, 4, 5, 6],
            ['1', 2, 33, 4, 5, 6]
        );
```

```
    }
}
?>
```

```
phpunit ArrayWeakComparisonTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) ArrayWeakComparisonTest::testEquality
Failed asserting that two arrays are equal.
--- Expected
+++ Actual
@@ @@
 Array (
-    0 => 1
+    0 => '1'
     1 => 2
-    2 => 3
+    2 => 33
     3 => 4
     4 => 5
     5 => 6
 )


/home/sb/ArrayWeakComparisonTest.php:7

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Dans cet exemple, la différence dans le premier indice entre `1` et `'1'` est signalée même si AssertEquals considère les valeurs comme une correspondance.

Chapitre 3. Le lanceur de tests en ligne de commandes

Le lanceur de tests en ligne de commandes de PHPUnit peut être appelé via la commande `phpunit`. Le code suivant montre comment exécuter des tests avec le lanceur de tests en ligne de commandes de PHPUnit:

```
phpunit ArrayTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

..

Time: 0 seconds


OK (2 tests, 2 assertions)
```

Lorsqu'il est appelé comme indiqué ci-dessus, le contrôleur de ligne de commande PHPUnit recherchera un fichier source `ArrayTest.php` dans le répertoire de travail courant, le chargera et s'attendra à trouver une classe de cas de test `ArrayTest`. Il exécutera alors les tests de cette classe.

Pour chaque test exécuté, l'outil en ligne de commandes de PHPUnit affiche un caractère pour indiquer l'avancement:

`.` Affiché quand le test a réussi.

`F` Affiché quand une assertion échoue lors de l'exécution d'une méthode de test.

`E` Affiché quand une erreur survient pendant l'exécution d'une méthode de test.

`R` Affiché quand le test a été marqué comme risqué (voir Chapitre 6, *Tests risqués*).

`S` Affiché quand le test a été sauté (voir Chapitre 7, *Tests incomplets et sautés*).

`I` Affiché quand le test est marqué comme incomplet ou pas encore implémenté (voir Chapitre 7, *Tests incomplets et sautés*).

PHPUnit différencie les *échecs* et les *erreurs*. Un échec est une assertion PHPUnit violée comme un appel en échec de `assertEquals()`. Une erreur est une exception inattendue ou une erreur PHP. Parfois cette distinction s'avère utile car les erreurs tendent à être plus faciles à corriger que les échecs. Si vous avez une longue liste de problèmes, il vaut mieux éradiquer d'abord les erreurs pour voir s'il reste encore des échecs uen fois qu'elles ont été corrigées.

Options de la ligne de commandes

Jetons un oeil aux options du lanceur de tests en ligne de commandes dans le code suivant :

```
phpunit --help
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

Usage: phpunit [options] UnitTest [UnitTest.php]
       phpunit [options] <directory>

Code Coverage Options:

  --coverage-clover <file>    Generate code coverage report in Clover XML format.
  --coverage-crap4j <file>    Generate code coverage report in Crap4J XML format.
  --coverage-html <dir>       Generate code coverage report in HTML format.
  --coverage-php <file>       Export PHP_CodeCoverage object to file.
  --coverage-text=<file>      Generate code coverage report in text format.
```

```
                              Default: Standard output.
  --coverage-xml <dir>        Generate code coverage report in PHPUnit XML format.
  --whitelist <dir>           Whitelist <dir> for code coverage analysis.
  --disable-coverage-ignore   Disable annotations for ignoring code coverage.

Logging Options:

  --log-junit <file>          Log test execution in JUnit XML format to file.
  --log-teamcity <file>       Log test execution in TeamCity format to file.
  --testdox-html <file>       Write agile documentation in HTML format to file.
  --testdox-text <file>       Write agile documentation in Text format to file.
  --testdox-xml <file>        Write agile documentation in XML format to file.
  --reverse-list              Print defects in reverse order

Test Selection Options:

  --filter <pattern>          Filter which tests to run.
  --testsuite <name,...>      Filter which testsuite to run.
  --group ...                 Only runs tests from the specified group(s).
  --exclude-group ...         Exclude tests from the specified group(s).
  --list-groups               List available test groups.
  --list-suites               List available test suites.
  --test-suffix ...           Only search for test in files with specified
                              suffix(es). Default: Test.php,.phpt

Test Execution Options:

  --dont-report-useless-tests Do not report tests that do not test anything.
  --strict-coverage           Be strict about @covers annotation usage.
  --strict-global-state       Be strict about changes to global state
  --disallow-test-output      Be strict about output during tests.
  --disallow-resource-usage   Be strict about resource usage during small tests.
  --enforce-time-limit        Enforce time limit based on test size.
  --disallow-todo-tests       Disallow @todo-annotated tests.

  --process-isolation         Run each test in a separate PHP process.
  --globals-backup            Backup and restore $GLOBALS for each test.
  --static-backup             Backup and restore static attributes for each test.

  --colors=<flag>             Use colors in output ("never", "auto" or "always").
  --columns <n>               Number of columns to use for progress output.
  --columns max               Use maximum number of columns for progress output.
  --stderr                    Write to STDERR instead of STDOUT.
  --stop-on-error             Stop execution upon first error.
  --stop-on-failure           Stop execution upon first error or failure.
  --stop-on-warning           Stop execution upon first warning.
  --stop-on-risky             Stop execution upon first risky test.
  --stop-on-skipped           Stop execution upon first skipped test.
  --stop-on-incomplete        Stop execution upon first incomplete test.
  --fail-on-warning           Treat tests with warnings as failures.
  --fail-on-risky             Treat risky tests as failures.
  -v|--verbose                Output more verbose information.
  --debug                     Display debugging information.

  --loader <loader>           TestSuiteLoader implementation to use.
  --repeat <times>            Runs the test(s) repeatedly.
  --teamcity                  Report test execution progress in TeamCity format.
  --testdox                   Report test execution progress in TestDox format.
  --testdox-group             Only include tests from the specified group(s).
  --testdox-exclude-group     Exclude tests from the specified group(s).
  --printer <printer>         TestListener implementation to use.

Configuration Options:
```

```
  --bootstrap <file>          A "bootstrap" PHP file that is run before the tests.
  -c|--configuration <file>   Read configuration from XML file.
  --no-configuration          Ignore default configuration file (phpunit.xml).
  --no-coverage               Ignore code coverage configuration.
  --no-extensions             Do not load PHPUnit extensions.
  --include-path <path(s)>    Prepend PHP's include_path with given path(s).
  -d key[=value]              Sets a php.ini value.
  --generate-configuration    Generate configuration file with suggested settings.

Miscellaneous Options:

  -h|--help                   Prints this usage information.
  --version                   Prints the version and exits.
  --atleast-version <min>     Checks that version is greater than min and exits.
```

`phpunit UnitTest`
: Exécute les tests qui sont fournis par la classe `UnitTest`. Cette classe est supposée être déclarée dans le fichier source `UnitTest.php`.

  `UnitTest` doit soit être une classe qui hérite de `PHPUnit\Framework\TestCase` soit une classe qui fournit une méthode `public static suite()` retournant un objet `PHPUnit_Framework_Test`, par exemple une instance de la classe `PHPUnit_Framework_TestSuite`.

`phpunit UnitTest UnitTest.php`
: Exécute les tests qui sont fournis par la classe `UnitTest`. Cette classe est supposée être déclarée dans le fichier source indiqué.

`--coverage-clover`
: Génère un fichier de log au format XML avec les informations de couverture de code pour les tests exécutés. Voir Chapitre 13, *Journalisation* pour plus de détails.

  Merci de noter que cette fonctionnalité n'est seulement disponible que lorsque les extensions tokenizer et Xdebug sont installées.

`--coverage-crap4j`
: Génère un raport de couverture de code au format Crap4j. Voir Chapitre 11, *Analyse de couverture de code* pour plus de détails.

  Merci de noter que cette fonctionnalité n'est seulement disponible que lorsque les extensions tokenizer et Xdebug sont installées.

`--coverage-html`
: Génère un rapport de couverture de code au format HTML. Voir Chapitre 11, *Analyse de couverture de code* pour plus de détails.

  Merci de noter que cette fonctionnalité n'est seulement disponible que lorsque les extensions tokenizer et Xdebug sont installées.

`--coverage-php`
: Génère un objet sérialisé PHP_CodeCoverage contenant les informations de couverture de code.

  Merci de noter que cette fonctionnalité n'est seulement disponible que lorsque les extensions tokenizer et Xdebug sont installées.

`--coverage-text`
: Génère un fichier de log ou une sortie écran sur la ligne de commandes dans un format lisible avec les informations de couver-

ture de code pour les tests exécutés. Voir Chapitre 13, *Journalisation* pour plus de détails.

Merci de noter que cette fonctionnalité n'est seulement disponible que lorsque les extensions tokenizer et Xdebug sont installées.

`--log-junit` — Génère un fichier de log au format JUnit XML pour les tests exécutés. Voir Chapitre 13, *Journalisation* pour plus de détails.

`--testdox-html` et `--testdox-text` — Génère la documentation agile au format HTML ou texte pur pour les tests exécutés. Voir Chapitre 12, *Autres utilisations des tests* pour plus de détails.

`--filter` — Exécute seulement les tests dont le nom correspond à l'expression régulière donnée. Si le motif n'est pas entouré de délimiteurs, PHPUnit inclura le motif dans les délimiteurs `/`.

Les noms de test à faire correspondre seront dans l'un des formats suivant :

`TestNamespace\TestCaseClass::testMethod` — Le format de nom de test par défaut est l'équivalent de l'utilisation de la constante magique `__METHOD__` dans la méthode de test.

`TestNamespace\TestCaseClass::testMethod with data set #0` — Lorsqu'un test a un fournisseur de données, chaque itération des données a l'index courant ajouté à la fin du nom de test par défaut.

`TestNamespace\TestCaseClass::testMethod with data set "my named data"` — Lorsqu'un test a un fournisseur de données qui utilise des ensembles nommés, chaque itération des données a le nom courant ajouté à la du nom de test par défaut. Voir Exemple 3.1, « Ensembles de données nommés » pour un exemple de fournisseurs de données avec des ensembles nommés.

Exemple 3.1 : Ensembles de données nommés

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

namespace TestNamespace;
```

```
class TestCaseClass extends Tes
{
    /**
     * @dataProvider provider
     */
    public function testMethod(
    {
        $this->assertTrue($data
    }

    public function provider()
    {
        return [
            'my named data' =>
            'my data'       =>
        ];
    }
}
?>
```

`/path/to/my/test.phpt` Le nom du test pour un test PHPT est le chemin du système de fichiers.

Voir Exemple 3.2, « Exemples de motif de filtre » pour des exemples de motifs de filtre valide.

Exemple 3.2. Exemples de motif de filtre

- `--filter 'TestNamespace\\TestCaseClass::testMethod'`
- `--filter 'TestNamespace\\TestCaseClass'`
- `--filter TestNamespace`
- `--filter TestCaseClass`
- `--filter testMethod`
- `--filter '/::testMethod .*"my named data"/'`
- `--filter '/::testMethod .*#5$/'`
- `--filter '/::testMethod .*#(5|6|7)$/'`

Voir Exemple 3.3, « Raccourcis de filtre » pour quelques raccourcis supplémentaires disponibles pour faire correspondre des fournisseurs de données.

Exemple 3.3. Raccourcis de filtre

- `--filter 'testMethod#2'`
- `--filter 'testMethod#2-4'`
- `--filter '#2'`
- `--filter '#2-4'`
- `--filter 'testMethod@my named data'`

	• `--filter 'testMethod@my.*data'` • `--filter '@my named data'` • `--filter '@my.*data'`
`--testsuite`	Exécute uniquement la suite de test dont le nom correspond au modèle donné.
`--group`	Exécute seulement les tests appartenant à un/des groupe(s) indiqué(s). Un test peut être signalé comme appartenant à un groupe en utilisant l'annotation `@group`. L'annotation `@author` est un alias pour `@group` permettant de filtrer les tests en se basant sur leurs auteurs.
`--exclude-group`	Exclut les tests d'un/des groupe(s) indiqué(s). Un test peut être signalé comme appartenant à un groupe en utilisant l'annotation `@group`.
`--list-groups`	Liste les groupes de tests disponibles.
`--test-suffix`	Recherche seulement les fichiers de test avec le(s) suffixe(s) spécifié(s).
`--report-useless-tests`	Être strict sur les tests qui ne testent rien. Voir Chapitre 6, *Tests risqués* pour plus de détails.
`--strict-coverage`	Être strict sur le code non-intentionnellemnt couvert. Voir Chapitre 6, *Tests risqués* pour plus de détails.
`--strict-global-state`	Être strict sur la manipulation de l'état global. Voir Chapitre 6, *Tests risqués* pour plus de détails.
`--disallow-test-output`	Être strict sur les sorties écran pendant les tests. Voir Chapitre 6, *Tests risqués* pour plus de détails.
`--disallow-todo-tests`	Ne pas executer les tests qui ont l'annotation `@todo` dans son docblock.
`--enforce-time-limit`	Appliquer une limite de temps basée sur la taille du test. Voir Chapitre 6, *Tests risqués* pour plus de détails.
`--process-isolation`	Exécute chaque test dans un processus PHP distinct.
`--no-globals-backup`	Ne pas sauvegarder et restaurer $GLOBALS. Voir la section intitulée « Etat global » pour plus de détails.
`--static-backup`	Sauvegarder et restaurer les attributs statiques des classes définies par l'utilisateur. Voir la section intitulée « Etat global » pour plus de détails.
`--colors`	Utiliser des couleurs pour la sortie écran. Sur Windows, utiliser ANSICON [https://github.com/adoxa/ansicon] ou ConEmu [https://github.com/Maximus5/ConEmu]. Il existe trois valeurs possible pour cette option: • `never` : Ne jamais afficher de couleurs dans la sortie écran. Il s'agit de la valeur par défaut lorsque l'option `--colors` n'est pas utilisée.

	• `auto` : Afficher les couleurs dans la sortie à moins que le terminal actuel ne supporte pas les couleurs, ou si la sortie est envoyée vers une commande ou redirigée vers un fichier. • `always` : Toujours affiche les couleurs dans la sortie écran, même lorsque le terminal en cours ne prend pas en charge les couleurs, ou lorsque la sortie est envoyée vers une commande ou redirigée vers un fichier. Lorsque `--colors` est utilisée sans aucune valeur, `auto` est la valeur choisie.
`--columns`	Définit le nombre de colonnes à utiliser pour la barre de progression. Si la valeur définie est `max`, le nombre de colonnes sera le maximum du terminal courant.
`--stderr`	Utilise optionnellement `STDERR` au lieu de `STDOUT` pour l'affichage.
`--stop-on-error`	Arrête l'execution à la première erreur.
`--stop-on-failure`	Arrête l'execution à la première erreur ou au premier échec.
`--stop-on-risky`	Arrête l'execution au premier test risqué.
`--stop-on-skipped`	Arrête l'execution au premier test sauté.
`--stop-on-incomplete`	Arrête l'execution au premier test incomplet.
`--verbose`	Affiche des informations plus détaillées, par exemple le nom des tests qui sont incomplets ou qui ont été sautés.
`--debug`	Affiche des informations de débogage telles que le nom d'un test quand son exécution démarre.
`--loader`	Indique l'implémentation de `PHPUnit_Runner_TestSuiteLoader` à utiliser. Le chargeur standard de suite de tests va chercher les fichiers source dans le répertoire de travail actuel et dans chaque répertoire qui est indiqué dans la directive de configuration PHP `include_path`. Le nom d'une classe tel que `Projet_Paquetage_Classe` est calqué sur le nom de fichier source `Projet/Paquetage/Classe.php`.
`--repeat`	Répéter l'exécution du(des) test(s) le nombre indiqué de fois.
`--testdox`	Rapporte l'avancement des tests sous forme de documentation agile. Voir Chapitre 12, *Autres utilisations des tests* pour plus de détails.
`--printer`	Indique l'afficheur de résultats à utiliser. Cette classe d'afficheur doit hériter de `PHPUnit_Util_Printer` et implémenter l'interface `PHPUnit\Framework\TestListener`.
`--bootstrap`	Un fichier PHP "amorce" ("bootstrap") est exécuté avant les tests.
`--configuration`, `-c`	Lit la configuration dans un fichier XML. Voir Annexe C, *Le fichier de configuration XML* pour plus de détails.

	Si `phpunit.xml` ou `phpunit.xml.dist` (dans cet ordre) existent dans le répertoire de travail actuel et que `--configuration` n'est *pas* utilisé, la configuration sera automatiquement lue dans ce fichier.
`--no-configuration`	Ignore `phpunit.xml` et `phpunit.xml.dist` du répertoire de travail actuel.
`--include-path`	Préfixe l'`include_path` PHP avec le(s) chemin(s) donné(s).
`-d`	Fixe la valeur des options de configuration PHP données.

Note

Notez qu'à partir de 4.8, les options peuvent être placées après le(s) argument(s).

Chapitre 4. Fixtures

L'une des parties les plus consommatrices en temps lors de l'écriture de tests est d'écrire le code pour configurer le monde dans un état connu puis de le remettre dans son état initial quand le test est terminé. Cet état connu est appelé la *fixture* du test.

Dans Exemple 2.1, « Tester des opérations de tableau avec PHPUnit », la fixture était simplement le tableau sauvegardé dans la variable `$stack`. La plupart du temps, cependant, la fixture sera beaucoup plus complexe qu'un simple tableau, et le volume de code nécessaire pour la mettre en place croîtra dans les mêmes proportions. Le contenu effectif du test sera perdu dans le bruit de configuration de la fixture. Ce problème s'aggrave quand vous écrivez plusieurs tests doté de fixtures similaires. Sans l'aide du framework de test, nous aurions à dupliquer le code qui configure la fixture pour chaque test que nous écrivons.

PHPUnit gère le partage du code de configuration. Avant qu'une méthode de test ne soit lancée, une méthode template appelée `setUp()` est invoquée. `setUp()` est l'endroit où vous créez les objets sur lesquels vous allez passer les tests. Une fois que la méthode de test est finie, qu'elle ait réussi ou échoué, une autre méthode template appelée `tearDown()` est invoquée. `tearDown()` est l'endroit où vous nettoyez les objets sur lesquels vous avez passé les tests.

Dans Exemple 2.2, « Utiliser l'annotation `@depends` pour exprimer des dépendances » nous avons utilisé la relation producteur-consommateur entre les tests pour partager une fixture. Ce n'est pas toujours souhaitable ni même possible. Exemple 4.1, « Utiliser setUp() pour créer les fixtures stack » montre comme nous pouvons écrire les tests de `StackTest` de telle façon que ce n'est pas la fixture elle-même qui est réutilisée mais le code qui l'a créée. D'abord nous déclarons la variable d'instance, `$stack`, que nous allons utiliser à la place d'une variable locale à la méthode. Puis nous plaçons la création de la fixture `tableau` dans la méthode `setUp()`. Enfin, nous supprimons le code redondant des méthodes de test et nous utilisons la variable d'instance nouvellement introduite. `$this->stack`, à la place de la variable locale à la méthode `$stack` avec la méthode d'assertion `assertEquals()`.

Exemple 4.1. Utiliser setUp() pour créer les fixtures stack

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

class StackTest extends TestCase
{
    protected $stack;

    protected function setUp()
    {
        $this->stack = [];
    }

    public function testEmpty()
    {
        $this->assertTrue(empty($this->stack));
    }

    public function testPush()
    {
        array_push($this->stack, 'foo');
        $this->assertEquals('foo', $this->stack[count($this->stack)-1]);
        $this->assertFalse(empty($this->stack));
    }

    public function testPop()
    {
```

```
        array_push($this->stack, 'foo');
        $this->assertEquals('foo', array_pop($this->stack));
        $this->assertTrue(empty($this->stack));
    }
}
?>
```

Les méthodes template `setUp()` et `tearDown()` sont exécutées une fois pour chaque méthode de test (et pour les nouvelles instances) de la classe de cas de test.

De plus, les méthodes template `setUpBeforeClass()` et `tearDownAfterClass()` sont appelées respectivement avant que le premier test de la classe de cas de test ne soit exécuté et après que le dernier test de la classe de test a été exécuté.

L'exemple ci-dessous montre toutes les méthodes template qui sont disponibles dans une classe de cas de test.

Exemple 4.2. Exemple montrant toutes les méthodes template disponibles

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

class TemplateMethodsTest extends TestCase
{
    public static function setUpBeforeClass()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

    protected function setUp()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

    protected function assertPreConditions()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

    public function testOne()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
        $this->assertTrue(true);
    }

    public function testTwo()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
        $this->assertTrue(false);
    }

    protected function assertPostConditions()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

    protected function tearDown()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

    public static function tearDownAfterClass()
    {
```

```
        fwrite(STDOUT, __METHOD__ . "\n");
    }

    protected function onNotSuccessfulTest(Exception $e)
    {
        fwrite(STDOUT, __METHOD__ . "\n");
        throw $e;
    }
}
?>
```

```
phpunit TemplateMethodsTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

TemplateMethodsTest::setUpBeforeClass
TemplateMethodsTest::setUp
TemplateMethodsTest::assertPreConditions
TemplateMethodsTest::testOne
TemplateMethodsTest::assertPostConditions
TemplateMethodsTest::tearDown
.TemplateMethodsTest::setUp
TemplateMethodsTest::assertPreConditions
TemplateMethodsTest::testTwo
TemplateMethodsTest::tearDown
TemplateMethodsTest::onNotSuccessfulTest
FTemplateMethodsTest::tearDownAfterClass


Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) TemplateMethodsTest::testTwo
Failed asserting that <boolean:false> is true.
/home/sb/TemplateMethodsTest.php:30

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.
```

Plus de setUp() que de tearDown()

`setUp()` et `tearDown()` sont sympathiquement symétriques en théorie mais pas en pratique. En pratique, vous n'avez besoin d'implémenter `tearDown()` que si vous avez alloué des ressources externes telles que des fichiers ou des sockets dans `setUp()`. Si votre `setUp()` ne crée simplement que de purs objets PHP, vous pouvez généralement ignorer `tearDown()`. Cependant, si vous créez de nombreux objets dans votre `setUp()`, vous pourriez vouloir libérer (`unset()`) les variables pointant vers ces objets dans votre `tearDown()` de façon à ce qu'ils puissent être récupérés par le ramasse-miettes. Le nettoyage des objets de cas de test n'est pas prévisible.

Variantes

Que se passe-t'il si vous avez deux tests avec deux setups légèrement différents ? Il y a deux possibilités :

- Si le code des `setUp()` ne diffère que légèrement, extrayez le code qui diffère du code de `setUp()` pour le mettre dans la méthode de test.
- Si vous avez vraiment deux `setUp()` différentes, vous avez besoin de classes de cas de test différentes. Nommez les classes selon les différences constatées dans les setup.

Partager les Fixtures

Il existe quelques bonnes raisons pour partager des fixtures entre les tests, mais dans la plupart des cas la nécessité de partager une fixture entre plusieurs tests résulte d'un problème de conception non résolu.

Un bon exemple de fixture qu'il est raisonnable de partager entre plusieurs tests est une connexion à une base de données : vous vous connectez une fois à la base de données et vous réutilisez cette connexion au lieu d'en créer une nouvelle pour chaque test. Ceci rend vos tests plus rapides.

Exemple 4.3, « Partager les fixtures entre les tests d'une série de tests » utilise les méthodes template `setUpBeforeClass()` et `tearDownAfterClass()` pour respectivement établir la connexion à la base de données avant le premier test de la classe de cas de test et pour de déconnecter de la base de données après le dernier test du cas de test.

Exemple 4.3. Partager les fixtures entre les tests d'une série de tests

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

class DatabaseTest extends TestCase
{
    protected static $dbh;

    public static function setUpBeforeClass()
    {
        self::$dbh = new PDO('sqlite::memory:');
    }

    public static function tearDownAfterClass()
    {
        self::$dbh = null;
    }
}
?>
```

On n'insistera jamais assez sur le fait que partager les fixtures entre les tests réduit la valeur de ces tests. Le problème de conception sous-jacent est que les objets ne sont pas faiblement couplés. Vous pourrez obtenir de meilleurs résultats en résolvant le problème de conception sous-jacent puis en écrivant des tests utilisant des bouchons (voir Chapitre 9, *Doublure de test*), plutôt qu'en créant des dépendances entre les tests à l'exécution et en ignorant l'opportunité d'améliorer votre conception.

Etat global

Il est difficile de tester du code qui utilise des singletons. [http://googletesting.blogspot.com/2008/05/tott-using-dependancy-injection-to.html] La même chose est vraie pour le code qui utilise des variables globales. Typiquement, le code que vous voulez tester est fortement couplé avec une variable globale et vous ne pouvez pas contrôler sa création. Un problème additionnel réside dans le fait qu'un test qui modifie une variable globale peut faire échouer un autre test.

En PHP, les variables globales fonctionnent comme ceci :

- Une variable globale `$foo = 'bar';` est enregistrée comme `$GLOBALS['foo'] = 'bar';`.
- La variable `$GLOBALS` est une variable appelée *super-globale*.
- Les variables super-globales sont des variables internes qui sont toujours disponibles dans toutes les portées.
- Dans la portée d'une fonction ou d'une méthode, vous pouvez accéder à la variable globale `$foo` soit en accédant directement à `$GLOBALS['foo']` soit en utilisant `global $foo;` pour créer une variable locale faisant référence à la variable globale.

A part les variables globales, les attributs statiques des classes font également partie de l'état global.

Avant la version 6, par défaut, PHPUnit exécute vos tests de façon à ce que des modifications aux variables globales et super-globales (`$GLOBALS`, `$_ENV`, `$_POST`, `$_GET`, `$_COOKIE`, `$_SERVER`, `$_FILES`, `$_REQUEST`) n'affectent pas les autres tests.

À partir de la version 6, PHPUnit n'effectue plus ces opérations de sauvegarde et restauration pour les variables globales et super-globales par défaut. Il peut être activé en utilisant l'option `--globals-backup` ou le paramètre `backupGlobals="true"` dans le fichier XML de configuration.

En utilisant l'option `--static-backup` ou le paramètre `backupStaticAttributes="true"` dans le fichier XML de configuration, cet isolation peut être étendue aux attributs statiques des classes.

Note

L'implémentation des opérations de sauvegarde et de restauration des variables globales et des attributs statiques des classes utilise `serialize()` et `unserialize()`.

Les objets de certaines classes (tel que `PDO` par exemple), ne peuvent pas être sérialisés si bien que l'opération de sauvegarde va échouer quand un tel objet sera e nregistré dans le tableau `$GLOBALS`, par exemple.

L'annotation `@backupGlobals` qui est discutée dans la section intitulée « @backupGlobals » peut être utilisée pour contrôler les opérations de sauvegarde et de restauration des variables globales. Alternativement, vous pouvez fournir une liste noire des variables globales qui doivent être exclues des opérations de sauvegarde et de restauration comme ceci :

```
class MyTest extends TestCase
{
    protected $backupGlobalsBlacklist = ['globalVariable'];

    // ...
}
```

Note

Paramétrer l'attribut `$backupGlobalsBlacklist` à l'intérieur de la méthode `setUp()`, par exemple, n'a aucun effet.

L'annotation `@backupStaticAttributes` qui est discutée dans la section intitulée « @backupStaticAttributes » peut être utilisée pour sauvegarder toutes les propriétés statiques dans toutes les classes déclarées avant chaque test et les restaurer ensuite.

Il traite toutes les classes déclarées au démarrage d'un test, et pas seulement la classe de test elle-même. Cela s'applique uniquement aux propriétés de classe statiques, pas aux variables statiques dans les fonctions.

Note

L'opération `@backupStaticAttributes` est exécutée avant une méthode de test, mais seulment si c'est activé. Si une valeur statique a été changée par un test exécuté précédement qui n'as activé `@backupStaticAttributes`, alors cette valeur sera sauvegardée et restaurée — pas la valeur par défaut originale. PHP n'enregistre pas la valeur par défaut déclarée à l'origine de toute variable statique.

La même chose s'applique aux propriétés statiques des classes nouvellement chargées/déclarées dans un test. Ils ne peuvent pas être réinitialisés à leur valeur par défaut déclarée à l'origine après le test, puisque cette valeur est inconnue. Quelle que soit la valeur définie, elle fuira dans les tests suivants.

Pour un test unitaire, il est recommandé de plutôt réinitialiser explicitement les valeurs des propriétés statiques testées dans le code de `setUp()` (et idéalement aussi `tearDown()`, de manière à ne pas affecter les prochains tests executés).

Vous pouvez fournir une liste noire d'attributs statiques qui doivent être exclus des opération de sauvegarde et de restauration:

```
class MyTest extends TestCase
{
    protected $backupStaticAttributesBlacklist = [
        'className' => ['attributeName']
    ];

    // ...
}
```

Note

Paramétrer l'attribut `$backupStaticAttributesBlacklist` à l'intérieur de la méthode `setUp()`, par exemple, n'a aucun effet.

Chapitre 5. Organiser les tests

L'un des objectifs de PHPUnit est que les tests soient combinables : nous voulons pouvoir exécuter n'importe quel nombre ou combinaison de tests ensembles, par exemple tous les tests pour le projet entier, ou les tests pour toutes les classes d'un composant qui constitue une partie du projet ou simplement les tests d'une seule classe particulière.

PHPUnit gère différente façon d'organiser les tests et de les combiner en une suite de tests. Ce chapitre montre les approches les plus communément utilisées.

Composer une suite de tests en utilisant le système de fichiers

La façon probablement la plus simple d'organiser une suite de tests est de mettre tous les fichiers sources des cas de test dans un répertoire de tests. PHPUnit peut automatiquement trouver et exécuter les tests en parcourant récursivement le répertoire test.

Jetons un oeil à la suite de tests de la bibliothèque sebastianbergmann/money [http://github.com/sebastianbergmann/money/]. En regardant la structure des répertoires du projet, nous voyons que les classes des cas de test dans le répertoire `tests` reflètent la structure des paquetages et des classes du système en cours de test (SCT, System Under Test ou SUT) dans le répertoire `src` :

```
src                                 tests
`-- Currency.php                    `-- CurrencyTest.php
`-- IntlFormatter.php               `-- IntlFormatterTest.php
`-- Money.php                       `-- MoneyTest.php
`-- autoload.php
```

Pour exécuter tous les tests de la bibliothèque, nous n'avons qu'à faire pointer le lanceur de tests en ligne de commandes de PHPUnit sur ce répertoire test :

```
phpunit --bootstrap src/autoload.php tests
PHPUnit 6.5.0 by Sebastian Bergmann.

.................................

Time: 636 ms, Memory: 3.50Mb

OK (33 tests, 52 assertions)
```

Note

Si vous pointez le lanceur de tests en ligne de commandes de PHPUnit sur un répertoire, il va chercher les fichiers `*Test.php`.

Pour n'exécuter que les tests déclarés dans la classe de cas de test `CurrencyTest` dans `tests/CurrencyTest`, nous pouvons utiliser la commande suivante :

```
phpunit --bootstrap src/autoload.php tests/CurrencyTest
PHPUnit 6.5.0 by Sebastian Bergmann.

........

Time: 280 ms, Memory: 2.75Mb

OK (8 tests, 8 assertions)
```

Pour un contrôle plus fin sur les tests à exécuter, nous pouvons utiliser l'option `--filter` :

```
phpunit --bootstrap src/autoload.php --filter testObjectCanBeConstructedForValidConstruc
PHPUnit 6.5.0 by Sebastian Bergmann.

..

Time: 167 ms, Memory: 3.00Mb

OK (2 test, 2 assertions)
```

Note

Un inconvénient de cette approche est que nous n'avons pas de contrôle sur l'ordre dans lequel les tests sont exécutés. Ceci peut conduire à des problèmes concernant les dépendances des tests, voir la section intitulée « Dépendances des tests ». Dans la prochaine section, nous verrons comment nous pouvons rendre l'ordre d'exécution des tests explicite en utilisant le fichier de configuration XML.

Composer une suite de tests en utilisant la configuration XML

Le fichier de configuration XML de PHPUnit (Annexe C, *Le fichier de configuration XML*) peut aussi être utilisé pour composer une suite de tests. Exemple 5.1, « Composer une suite de tests en utilisant la configuration XML » montre un exemple minimaliste d'un fichier `phpunit.xml` qui va ajouter toutes les classes `*Test` trouvées dans les fichiers `*Test.php` quand `tests` est parcouru récursivement.

Exemple 5.1. Composer une suite de tests en utilisant la configuration XML

```
<phpunit bootstrap="src/autoload.php">
  <testsuites>
    <testsuite name="money">
      <directory>tests</directory>
    </testsuite>
  </testsuites>
</phpunit>
```

Si un fichier `phpunit.xml` ou `phpunit.xml.dist` (dans cet ordre) existe dans le répertoire de travail courant et que l'option `--configuration` n'est *pas* utilisée, la configuration sera automatiquement lue depuis ce fichier.

L'ordre dans lequel les tests sont exécutés peut être rendu explicite :

Exemple 5.2. Composer une suite de tests en utilisant la configuration XML

```
<phpunit bootstrap="src/autoload.php">
  <testsuites>
    <testsuite name="money">
      <file>tests/IntlFormatterTest.php</file>
      <file>tests/MoneyTest.php</file>
      <file>tests/CurrencyTest.php</file>
    </testsuite>
  </testsuites>
</phpunit>
```

Chapitre 6. Tests risqués

PHPUnit peut effectuer les vérifications supplémentaires documentées ci-dessous pendant qu'il exécute les tests.

Tests inutiles

PHPUnit peut être strict sur les tests qui ne testent rien. Cette vérification peut être activée en utilisant l'option `--report-useless-tests` de la ligne de commande ou en définissant `beStrictAboutTestsThatDoNotTestAnything="true"` dans le fichier de configuration XML de PHPUnit.

Un test qui n'effectue pas d'assertion sera marqué comme risqué lorsque cette vérification est activée. Les attentes sur les objets bouchonnés ou les annotations telles que `@expectedException` comptent comme une assertion.

Code non-intentionnellement couvert

PHPUnit peut être strict sur le code couvert non-intentionnellement. Cette vérification peut être activée en utilisant l'option `--strict-coverage` de la ligne de commande ou en définissant `checkForUnintentionallyCoveredCode="true"` dans le fichier de configuration XML de PHPUnit.

Un test qui est annoté avec `@covers` et execute du code qui n'est pas listé avec les annotations `@covers` ou `@uses` sera marqué comme risqué quand cette vérification est activée.

Sortie d'écran lors de l'exécution d'un test

PHPUnit peut être strict sur la sortie écran pendant les tests. Cette vérification peut être activée en utilisant l'option `--disallow-test-output` de la ligne de commande ou en définissant `beStrictAboutOutputDuringTests="true"` dans le fichier de configuration XML de PHPUnit.

Un test qui émet une sortie écran, par exemple en appelant `print` dans dans le code du test ou dans le code testé, sera marqué comme risqué quand cette vérification est activée.

Délai d'execution des tests

Une limite de temps peut être appliquée pour l'exécution d'un test si le paquet `PHP_Invoker` est installé et que l'extension `pcntl` est disponible. L'application de cette limite de temps peut être activée en utilisant l'option `--enforce-time-limit` sur la ligne de commande ou en définissant `beStrictAboutTestSize="true"` dans le fichier de configuration XML de PHPUnit.

Un test annoté avec `@large` échouera s'il prend plus de 60 secondes à s'executer. Ce délai d'execution est configurable via l'attribut `timeoutForLargeTests` dans le fichier de configuration XML.

Un test annoté avec `@medium` échouera s'il prend plus de 10 secondes à s'executer. Ce délai d'execution est configurable via l'attribut `timeoutForMediumTests` dans le fichier de configuration XML.

Un test qui n'est pas annoté avec `@medium` ou `@large` sera traité comme s'il était annoté avec `@small`. Un test "small" échouera s'il prend plus de 1 seconde à s'executer. Ce délai d'execution est configurable via l'attribut `timeoutForMediumTests` dans le fichier de configuration XML.

Manipulation d'états globaux

PHPUnit peut être strict sur les tests qui manipulent l'état global. Cette vérification peut être activé en utilisant l'option `--strict-global-state` de la ligne de commande ou en définissant `beStrictAboutChangesToGlobalState="true"` dans le fichier de configuration XML de PHPUnit.

Chapitre 7. Tests incomplets et sautés

Tests incomplets

Quand vous travaillez sur une nouvelle classe de cas de test, vous pourriez vouloir commencer en écrivant des méthodes de test vides comme :

```
public function testQuelquechose()
{
}
```

pour garder la trace des tests que vous avez à écrire. Le problème avec les méthodes de test vides est qu'elles sont interprétées comme étant réussies par le framework PHPUnit. Cette mauvaise interprétation fait que le rapport de tests devient inutile -- vous ne pouvez pas voir si un test est effectivement réussi ou s'il n'a tout simplement pas été implémenté. Appeler `$this->fail()` dans une méthode de test non implémentée n'aide pas davantage, puisqu'alors le test sera interprété comme étant un échec. Ce serait tout aussi faux que d'interpréter un test non implémenté comme étant réussi.

Si nous pensons à un test réussi comme à un feu vert et à un échec de test comme à un feu rouge, nous avons besoin d'un feu orange additionnel pour signaler un test comme étant incomplet ou pas encore implémenté. `PHPUnit_Framework_IncompleteTest` est une interface de marquage pour signaler une exception qui est levée par une méthode de test comme résultat d'un test incomplet ou pas encore implémenté. `PHPUnit_Framework_IncompleteTestError` est l'implémentation standard de cette interface.

Exemple 7.1, « Signaler un test comme incomplet » montre une classe de cas de tests, `SampleTest`, qui contient une unique méthode de test, `testSomething()`. En appelant la méthode pratique `markTestIncomplete()` (qui lève automatiquement une exception `PHPUnit_Framework_IncompleteTestError`) dans la méthode de test, nous marquons le test comme étant incomplet.

Exemple 7.1. Signaler un test comme incomplet

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

class SampleTest extends TestCase
{
    public function testSomething()
    {
        // Optional: Test anything here, if you want.
        $this->assertTrue(true, 'This should already work.');

        // Stop here and mark this test as incomplete.
        $this->markTestIncomplete(
          'This test has not been implemented yet.'
        );
    }
}
?>
```

Un test incomplet est signalé par un `I` sur la sortie écran du lanceur de test en ligne de commandes PHPUnit, comme montré dans l'exemple suivant :

```
phpunit --verbose SampleTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.
```

```
I

Time: 0 seconds, Memory: 3.95Mb

There was 1 incomplete test:

1) SampleTest::testSomething
This test has not been implemented yet.

/home/sb/SampleTest.php:12
OK, but incomplete or skipped tests!
Tests: 1, Assertions: 1, Incomplete: 1.
```

Tableau 7.1, « API pour les tests incomplets » montre l'API pour marquer des tests comme incomplets.

Tableau 7.1. API pour les tests incomplets

Méthode	Signification
`void markTestIncomplete()`	Marque le test courant comme incomplet.
`void markTestIncomplete(string $message)`	Marque le test courant comme incomplet en utilisant `$message` comme message d'explication.

Sauter des tests

Tous les tests ne peuvent pas être exécutés dans tous les environnements. Considérez, par exemple, une couche d'abstraction de base de données qui possède différents pilotes pour les différents systèmes de base de données qu'elle gère. Les tests pour le pilote MySQL ne peuvent bien sûr être exécutés que si un serveur MySQL est disponible.

Exemple 7.2, « Sauter un test » montre une classe de cas de tests, `DatabaseTest`, qui contient une méthode de tests `testConnection()`. Dans le patron de méthode `setUp()` de la classe du cas de test, nous pouvons contrôler si l'extension MySQLi est disponible et utiliser la méthode `markTestSkipped()` pour sauter le test si ce n'est pas le cas.

Exemple 7.2. Sauter un test

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

class DatabaseTest extends TestCase
{
    protected function setUp()
    {
        if (!extension_loaded('mysqli')) {
            $this->markTestSkipped(
              'The MySQLi extension is not available.'
            );
        }
    }

    public function testConnection()
    {
        // ...
    }
}
?>
```

Un test qui a été sauté est signalé par un S dans la sortie écran du lanceur de tests en ligne de commande PHPUnit, comme montré dans l'exemple suivant.

```
phpunit --verbose DatabaseTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

S

Time: 0 seconds, Memory: 3.95Mb

There was 1 skipped test:

1) DatabaseTest::testConnection
The MySQLi extension is not available.

/home/sb/DatabaseTest.php:9
OK, but incomplete or skipped tests!
Tests: 1, Assertions: 0, Skipped: 1.
```

Tableau 7.2, « API pour sauter des tests » montre l'API pour sauter des tests.

Tableau 7.2. API pour sauter des tests

Méthode	Signification
`void markTestSkipped()`	Marque le test courant comme sauté.
`void markTestSkipped(string $message)`	Marque le test courant comme étant sauté en utilisant `$message` comme message d'explication.

Sauter des tests en utilisant @requires

En plus des méthodes ci-dessus, il est également possible d'utiliser l'annotation `@requires` pour exprimer les préconditions communes pour un cas de test.

Tableau 7.3. Usages possibles de @requires

Type	Valeurs possibles	Exemple	Autre exemple
`PHP`	Tout identifiant de version PHP	@requires PHP 5.3.3	@requires PHP 7.1-dev
`PHPUnit`	Tout identifiant de version PHPUnit	@requires PHPUnit 3.6.3	@requires PHPUnit 4.6
`OS`	Une expression regulière qui match PHP_OS [http://php.net/manual/en/reserved.constants.php#constant.php-os]	@requires OS Linux	@requires OS WIN32|WINNT
`function`	Tout paramètre valide pour function_exists [http://php.net/function_exists]	@requires function imap_open	@requires function ReflectionMethod::setAccessible
`extension`	Tout nom d'extension	@requires extension mysqli	@requires extension redis 2.2.0

Exemple 7.3. Sauter des cas de tests en utilisant @requires

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

/**
```

```
 * @requires extension mysqli
 */
class DatabaseTest extends TestCase
{
    /**
     * @requires PHP 5.3
     */
    public function testConnection()
    {
        // Test requires the mysqli extension and PHP >= 5.3
    }

    // ... All other tests require the mysqli extension
}
?>
```

Si vous utilisez une syntaxe qui ne compile pas avec une version données de PHP, regardez dans la configuration xml pour les inclusions dépendant de la version dans la section intitulée « Série de tests »

Chapitre 8. Tester des bases de données

De nombreux exemples de tests unitaires de niveau débutant ou intermédiaire dans de nombreux langages de programmation suggèrent qu'il est parfaitement facile de tester la logique de votre application avec de simples tests. Pour les applications centrées sur une base de données, c'est loin d'être la réalité. Commencez à utiliser WordPress, TYPO3 ou Symfony avec Doctrine ou Propel, par exemple, et nous serez vite confrontés à des problèmes considérables avec PHPUnit : juste parce que la base de données est vraiment étroitement liée à ces bibliothèques.

> **Note**
>
> Assurez-vous d'avoir l'extension PHP `pdo` et les extensions spécifique à la base de données tel que `pdo_mysql` installées. Sinon, les exemples ci-dessous ne fonctionneront pas.

Vous connaissez probablement ce scénario rencontré tous les jours sur les projets, dans lequel vous voulez mettre à l'oeuvre votre savoir-faire tout neuf ou déjà aguerri en PHPUnit et où vous vous retrouvez bloqué par l'un des problèmes suivants :

1. La méthode que vous voulez tester exécute une opération JOIN plutôt vaste et utilise les données pour calculer certains résultats importants.
2. Votre logique métier exécute un mélange d'instructions SELECT, INSERT, UPDATE et DELETE.
3. Vous devez configurer les données de test dans (éventuellement beaucoup) plus de deux tables pour obtenir des données initiales raisonnables pour les méthodes que vous voulez tester.

L'extension DbUnit simplifie considérablement la configuration d'une base de données à des fins de test et vous permet de vérifier le contenu d'une base de données après avoir réalisé une suite d'opérations.

Systèmes gérés pour tester des bases de données

DbUnit gère actuellement MySQL, PostgreSQL, Oracle et SQLite. Via l'intégration de Zend Framework [http://framework.zend.com] ou de Doctrine 2 [http://www.doctrine-project.org] il est possible d'accéder à d'autres systèmes de base de données comme IBM DB2 ou Microsoft SQL Server.

Difficultés pour tester les bases de données

Il y a une bonne raison pour laquelle les exemples concernant le test unitaire n'inclut pas d'interaction avec une base de données : ces types de test sont à la fois complexes à configurer et à maintenir. Quand vous faites des tests sur votre base de données, vous devez prendre soin des variables suivantes :

- Le schéma et les tables de la base de données
- Insérer les lignes nécessaires pour le test dans ces tables
- Vérifier l'état de la base de données après que votre test a été exécuté
- Nettoyer la base de données pour chaque nouveau test

Comme de nombreuses APIs de base de données comme PDO, MySQLi ou OCI8 sont lourdes à utiliser et verbeuses à écrire, réaliser ces étapes à la main est un cauchemar absolu.

Le code de test doit être aussi court et précis que possible pour plusieurs raisons :

- Vous ne voulez pas modifier un volume considérable de code de test pour de petites modifications dans votre code de production.
- Vous voulez être capable de lire et de comprendre le code de test facilement, même des mois après l'avoir écrit.

De plus, vous devez prendre conscience que la base de données est essentiellement une variable globale pour votre code. Deux tests de votre série de tests peuvent être exécutés sur la même base de données, potentiellement en réutilisant les données plusieurs fois. Un échec dans un test peut facilement affecter le résultat des tests suivants rendant votre expérimentation de test très difficile. L'étape de nettoyage mentionnée précédemment est d'une importance majeure pour résoudre le problème posé par le fait que « la base de données est une variable globale ».

DbUnit aide à simplifier tous ces problèmes avec le test de base de données d'une manière élégante.

Là où PHPUnit ne peut pas vous aider c'est pour le fait que les tests de base de données sont très lents comparés aux tests n'en utilisant pas. Selon l'importance des interactions avec votre base de données, vos tests peuvent s'exécuter sur une durée considérable. Cependant, si vous gardez petit le volume de données utilisées pour chaque test et que vous essayez de tester le plus de code possible en utilisant des tests qui ne font pas appel à une base de données, vous pouvez facilement rester très en dessous d'une minute, même pour de grandes séries de tests.

La suite de test du projet Doctrine 2 [http://www.doctrine-project.org], par exemple, possède actuellement une suite de tests d'environ 1000 tests dont presque la moitié accède à la base de données et continue à s'exécuter en 15 secondes sur une base de données MySQL sur un ordinateur de bureau standard.

Les quatre phases d'un test de base de données

Dans son livre sur les patterns de tests xUnit, Gerard Meszaros liste les quatre phases d'un test unitaire :

1. Configurer une fixture
2. Expérimenter le système à tester
3. Vérifier les résultats
4. Nettoyer

> **Qu'est-ce qu'une fixture ?**
>
> Une fixture décrit l'état initial dans lequel se trouvent votre application et votre base de données quand vous exécutez un test.

Tester la base de données nécessite au moins d'intervenir dans setup et teardown pour nettoyer et écrire les données de fixture nécessaires dans vos tables. Cependant, l'extension de base de données possède une bonne raison de rétablir les quatre phases dans un test de base de données pour constituer le processus suivant qui est exécuté pour chacun des tests :

1. Nettoyer la base de données

Puisqu'il y a toujours un premier test qui s'exécute en faisant appel à la base de données, vous n'êtes pas sûr qu'il y ait déjà des données dans les tables. PHPUnit va exécuter un TRUNCATE sur toutes les tables que vous avez indiquées pour les remettre à l'état vide.

2. Configurer les fixtures

PHPUnit va parcourir toutes les lignes de fixture indiquées et les insérer dans leurs tables respectives.

3–5. Exécuter les tests, vérifier les résultats et nettoyer

Une fois la base de données réinitialisée et remise dans son état de départ, le test en tant que tel est exécuté par PHPUnit. Cette partie du code de test ne nécessite pas du tout de s'occuper de l'extension base de données, vous pouvez procéder et tester tout ce que vous voulez dans votre code.

Votre test peut utiliser une assertion spéciale appelée `assertDataSetsEqual()` à des fins de vérification, mais c'est totalement facultatif. Cette fonctionnalité sera expliquée dans la section « Assertions pour les bases de données ».

Configuration d'un cas de test de base de données PHPUnit

Habituellement quand vous utilisez PHPUnit, vos cas de tests devraient hériter de la classe `PHPUnit\Framework\TestCase` de la façon suivante :

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

class MyTest extends TestCase
{
    public function testCalculate()
    {
        $this->assertEquals(2, 1 + 1);
    }
}
?>
```

Si vous voulez tester du code qui fonctionne avec l'extension base de données, le setup sera un peu plus complexe et vous devrez hériter d'un cas de test abstrait différent qui nécessite que vous implémentiez deux méthodes abstraites `getConnection()` et `getDataSet()`:

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

class MyGuestbookTest extends TestCase
{
    use TestCaseTrait;

    /**
     * @return PHPUnit_Extensions_Database_DB_IDatabaseConnection
     */
    public function getConnection()
    {
        $pdo = new PDO('sqlite::memory:');
        return $this->createDefaultDBConnection($pdo, ':memory:');
    }

    /**
     * @return PHPUnit_Extensions_Database_DataSet_IDataSet
     */
    public function getDataSet()
    {
        return $this->createFlatXMLDataSet(dirname(__FILE__).'/_files/guestbook-seed.xml
    }
}
?>
```

Implémenter getConnection()

Pour permettre aux fonctionnalités de nettoyage et de chargement des fixtures de fonctionner, l'extension de base de données PHPUnit nécessite d'accéder à une connexion de base de données abstraite pour les différents fournisseurs via la bibliothèque PDO. Il est important de noter que votre application n'a pas besoin de s'appuyer sur PDO pour utiliser l'extension de base de données de PHPUnit, la connexion est principalement utilisée pour le nettoyage et la configuration de setup.

Dans l'exemple précédent, nous avons créé une connexion Sqlite en mémoire et nous l'avons passé à la méthode `createDefaultDBConnection` qui encapsule l'instance PDO et le second paramètre (le nom de la base de données) dans une couche d'abstraction très simple pour connexion aux bases de données du type `PHPUnit_Extensions_Database_DB_IDatabaseConnection`.

La section « Utiliser la connexion de base de données » explicite l'API de cette interface et comment en faire le meilleur usage.

Implémenter getDataSet()

La méthode `getDataSet()` définit à quoi doit ressembler l'état initial de la base de données avant que chaque test ne soit exécuté. L'état de la base de données est abstrait par les concepts DataSet et DataTable, tous les deux représentés par les interfaces `PHPUnit_Extensions_Database_DataSet_IDataSet` et `PHPUnit_Extensions_Database_DataSet_IDataTable`. La prochaine section décrira en détail comment ces concepts fonctionnent et quels sont les avantages à les utiliser lors des tests de base de données.

Pour l'implémentation, nous avons seulement besoin de savoir que la méthode `getDataSet()` est appelée une fois dans `setUp()` pour récupérer l'ensemble de données de la fixture et l'insérer dans la base de données. Dans l'exemple, nous utilisons une méthode fabrique `createFlatXMLDataSet($filename)` qui représente un ensemble de données à l'aide d'une représentation XML.

Qu'en est-il du schéma de base de données (DDL)?

PHPUnit suppose que le schéma de base de données avec toutes ses tables, ses triggers, séquences et vues est créé avant qu'un test soit exécuté. Cela signifie que vous, en tant que développeur, devez vous assurer que la base de données est correctement configurée avant de lancer la suite de tests.

Il y a plusieurs moyens pour satisfaire cette condition préalable au test de base de données.

1. Si vous utilisez une base de données persistante (pas Sqlite en mémoire) vous pouvez facilement configure la base de données avec des outils tels que phpMyAdmin pour MySQL et réutiliser la base de données pour chaque exécution de test.

2. Si vous utilisez des bibliothèques comme Doctrine 2 [http://www.doctrine-project.org] ou Propel [http://www.propelorm.org/] vous pouvez utiliser leurs APIs pour créer le schéma de base de données dont vous avez besoin une fois avant de lancer vos tests. Vous pouvez utiliser les possibilités apportées par l'amorce et la configuration de PHPUnit [http://www.phpunit.de/manual/current/en/textui.html] pour exécuter ce code à chaque fois que vos tests sont exécutés.

Astuce: utilisez votre propre cas de tests abstrait de base de données

En partant des exemples d'implémentation précédents, vous pouvez facilement voir que la méthode `getConnection()` est plutôt statique et peut être réutilisée dans différents cas de test de base de données. Additionnellement pour conserver de bonnes performances pour vos tests et maintenir la charge de la base de données basse vous pouvez refactoriser un peu le code pour obtenir un cas de test

abstrait générique pour votre application, qui vous permette encore d'indiquer des données de fixture différentes pour chaque cas de test :

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

abstract class MyApp_Tests_DatabaseTestCase extends TestCase
{
    use TestCaseTrait;

    // only instantiate pdo once for test clean-up/fixture load
    static private $pdo = null;

    // only instantiate PHPUnit_Extensions_Database_DB_IDatabaseConnection once per test
    private $conn = null;

    final public function getConnection()
    {
        if ($this->conn === null) {
            if (self::$pdo == null) {
                self::$pdo = new PDO('sqlite::memory:');
            }
            $this->conn = $this->createDefaultDBConnection(self::$pdo, ':memory:');
        }

        return $this->conn;
    }
}
?>
```

Mais la connexion à la base de données reste codée en dur dans la connexion PDO. PHPUnit possède une autre fonctionnalité formidable qui peut rendre ce cas de test encore plus générique. Si vous utilisez la configuration XML [appendixes.configuration.html#appendixes.configuration.php-ini-constants-variables], vous pouvez rendre la connexion à la base de données configurable pour chaque exécution de test. Créons d'abord un fichier « phpunit.xml » dans le répertoire tests/ de l'application qui ressemble à ceci :

```
<?xml version="1.0" encoding="UTF-8" ?>
<phpunit>
    <php>
        <var name="DB_DSN" value="mysql:dbname=myguestbook;host=localhost" />
        <var name="DB_USER" value="user" />
        <var name="DB_PASSWD" value="passwd" />
        <var name="DB_DBNAME" value="myguestbook" />
    </php>
</phpunit>
```

Nous pouvons maintenant modifier notre cas de test pour qu'il ressemble à ça :

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

abstract class Generic_Tests_DatabaseTestCase extends TestCase
{
    use TestCaseTrait;

    // only instantiate pdo once for test clean-up/fixture load
    static private $pdo = null;

    // only instantiate PHPUnit_Extensions_Database_DB_IDatabaseConnection once per test
    private $conn = null;
```

```
    final public function getConnection()
    {
        if ($this->conn === null) {
            if (self::$pdo == null) {
                self::$pdo = new PDO( $GLOBALS['DB_DSN'], $GLOBALS['DB_USER'], $GLOBALS[
            }
            $this->conn = $this->createDefaultDBConnection(self::$pdo, $GLOBALS['DB_DBNA
        }

        return $this->conn;
    }
}
?>
```

Nous pouvons maintenant lancer la suite de tests de la base de données en utilisant différentes configurations depuis l'interface en ligne de commandes:

```
user@desktop> phpunit --configuration developer-a.xml MyTests/
user@desktop> phpunit --configuration developer-b.xml MyTests/
```

La possibilité de lancer facilement des tests de base de données sur différentes bases de données cibles est très important si vous développez sur une machine de développement. Si plusieurs développeurs exécutent les tests de base de données sur la même connexion de base de données, vous pouvez facilement faire l'expérience d'échec de tests du fait des concurrences d'accès.

Comprendre DataSets et DataTables

Un concept centre de l'extension de base de données PHPUnit sont les DataSets et les DataTables. Vous devez comprendre ce simple concept pour maîtriser les tests de bases de données avec PHPUnit. Les DataSets et les DataTables constituent une couche d'abstraction sur les tables, les lignes et les colonnes de la base de données. Une simple API cache le contenu de la base de données sous-jacente dans une structure objet, qui peut également être implémentée par d'autres sources qui ne sont pas des bases de données.

Cette abstraction est nécessaire pour comparer le contenu constaté d'une base de données avec le contenu attendu. Les attentes peuvent être représentées dans des fichiers XML, YAML ou CSV ou des tableaux PHP par exemple. Les interfaces DataSets et DataTables permettent de comparer ces sources conceptuellement différentes en émulant un stockage en base de données relationnelle dans une approche sémantiquement similaire.

Un processus pour des assertions de base de données dans vos tests se limitera alors à trois étapes simples :

- Indiquer une ou plusieurs tables dans votre base de données via leurs noms de table (ensemble de données constatées)
- Indiquez l'ensemble de données attendu dans votre format préféré (YAML, XML, ..)
- Affirmez que les représentations des deux ensembles de données sont égaux.

Les assertions ne constituent pas le seul cas d'utilisation des DataSets et DataTables dans l'extension de base de données PHPUnit. Comme illustré dans la section précédente, ils décrivent également le contenu initial de la base de données. Vous êtes obligés de définir un ensemble de données fixture avec le cas de test Database, qui est ensuite utilisé pour :

- Supprimer toutes les lignes des tables indiquées dans le DataSet.
- Ecrire toutes les lignes dans les tables de données dans la base de données.

Implémentations disponibles

Il existe trois types différents de datasets/datatables:

- DataSets et DataTables basés sur des fichiers
- DataSets et DataTables basés sur des requêtes
- DataSets et DataTables de filtre et de combinaison

les datasets et les tables basés sur des fichiers sont généralement utilisés pour la fixture initiale et pour décrire l'état attendu d'une base de données.

DataSet en XML à plat

Le dataset le plus commun est appelé XML à plat (flat XML). C'est un format xml très simple dans lequel une balise à l'intérieur d'un noeud racine `<dataset>` représente exactement une ligne de la base de données. Les noms des balises sont ceux des tables dans lesquelles insérer les lignes et un attribut représente la colonne. Un exemple pour une simple application de livre d'or pourrait ressembler à ceci :

```
<?xml version="1.0" ?>
<dataset>
    <guestbook id="1" content="Hello buddy!" user="joe" created="2010-04-24 17:15:23" />
    <guestbook id="2" content="I like it!" user="nancy" created="2010-04-26 12:14:20" />
</dataset>
```

C'est à l'évidence facile à écrire. Ici, `<guestbook>` est le nom de la table dans laquelle les deux lignes sont insérées, chacune avec quatre colonnes « id », « content », « user » et « created » et leurs valeurs respectives.

Cependant, cette simplicité a un coût.

Avec l'exemple précédent, difficile de voir comment nous devons indiquer une table vide. Vous pouvez insérer une balise avec aucun attribut contenant le nom de la table vide. Un fichier XML à plat pour une table livre_d_or pourrait alors ressembler à ceci:

```
<?xml version="1.0" ?>
<dataset>
    <guestbook />
</dataset>
```

La gestion des valeurs NULL avec le dataset en XML à plat est fastidieuse. Une valeur NULL est différente d'une chaîne vide dans la plupart des bases de données (Oracle étant une exception), quelque chose qu'il est difficile de décrire dans le format XML à plat. Vous pouvez représenter une valeur NULL en omettant d'attribut indiquant la ligne. Si votre livre d'or autorise les entrées anonymes représentées par une valeur NULL dans la colonne utilisateur, un état hypothétique de la table guestbook pourrait r essembler à ceci:

```
<?xml version="1.0" ?>
<dataset>
    <guestbook id="1" content="Hello buddy!" user="joe" created="2010-04-24 17:15:23" />
    <guestbook id="2" content="I like it!" created="2010-04-26 12:14:20" />
</dataset>
```

Dans ce cas, la seconde entrée est postée anonymement. Cependant, ceci conduit à un sérieux problème pour la reconnaissance de la colonne. Lors des assertions d'égalité de datasets, chaque dataset doit indiquer quelle colonne une table contient. Si un attribut est NULL pour toutes les lignes de la data-table, comment l'extension de base de données sait que la colonne doit faire partie de la table ?

Le dataset en XML à plat fait maintenant une hypothèse cruciale en décrétant que les attributs de la première ligne définie pour une table définissent les colonnes de cette table. Dans l'exemple précédent, ceci signifierait que « id », « content », « user » et « created » sont les colonnes de la table guestbook. Pour la seconde ligne dans laquelle « user » n'est pas défini, un NULL sera inséré dans la base de données.

Quand la première entrée du livre d'or est supprimée du dataset, seuls « id », « content » et « created » seront des colonnes de la table guestbook, puisque « user » n'est pas indiqué.

Pour utiliser efficacement le dataset au format XML à plat quand des valeurs NULL sont pertinentes, la première ligne de chaque table ne doit contenir aucune valeur NULL, seules les lignes suivantes pouvant omettre des attributs. Ceci peut s'avérer délicat, puisque l'ordre des lignes est un élément pertinent pour les assertions de base de données.

A l'inverse, si vous n'indiquez qu'un sous-élément des colonnes de la table dans le dataset au format XML à plat, toutes les valeurs omises sont positionnées à leurs valeurs par défaut. Ceci provoquera des erreurs si l'une des valeurs omises est définie par « NOT NULL DEFAULT NULL ».

En conclusion, je ne peux que vous conseiller de n'utiliser les datasets au format XML à plat que si vous n'avez pas besoin des valeurs NULL.

Vous pouvez créer une instance de dataset au format XML à plat dans votre cas de test de base de données en appelant la méthode `createFlatXmlDataSet($filename)`:

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

class MyTestCase extends TestCase
{
    use TestCaseTrait;

    public function getDataSet()
    {
        return $this->createFlatXmlDataSet('myFlatXmlFixture.xml');
    }
}
?>
```

DataSet XML

Il existe un autre dataset XML davantage structuré, qui est un peu plus verbeux à écrire mais qui évite les problèmes de NULL du dataset au format XML à plat. Dans le noeud racine `<dataset>` vous pouvez indiquer les balises `<table>`, `<column>`, `<row>`, `<value>` et `<null />`. Un dataset équivalent à celui défini précédemment pour le livre d'or en format XML à plat ressemble à :

```
<?xml version="1.0" ?>
<dataset>
    <table name="guestbook">
        <column>id</column>
        <column>content</column>
        <column>user</column>
        <column>created</column>
        <row>
            <value>1</value>
            <value>Hello buddy!</value>
            <value>joe</value>
            <value>2010-04-24 17:15:23</value>
        </row>
        <row>
            <value>2</value>
            <value>I like it!</value>
```

```
            <null />
            <value>2010-04-26 12:14:20</value>
        </row>
    </table>
</dataset>
```

Tout `<table>` défini possède un nom et nécessite la définition de toutes les colonnes avec leurs noms. Il peut contenir zéro ou tout nombre positif d'éléments `<row>` imbriqués. Ne définir aucun élément `<row>` signifie que la table est vide. Les balises `<value>` et `<null />` doivent être indiquées dans l'ordre des éléments `<column>`précédemment donnés. La balise `<null />` signifie évidemment que la valeur est NULL.

Vous pouvez créer une instance de dataset xml dans votre cas de test de base de données en appelant la méthode `createXmlDataSet($filename)` :

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

class MyTestCase extends TestCase
{
    use TestCaseTrait;

    public function getDataSet()
    {
        return $this->createXMLDataSet('myXmlFixture.xml');
    }
}
?>
```

DataSet XML MySQL

Ce nouveau format XML est spécifique au serveur de bases de données MySQL [http://www.mysql.com]. Sa gestion a été ajoutée dans PHPUnit 3.5. Les fichiers écrits ce format peuvent être générés avec l'utilitaire `mysqldump` [http://dev.mysql.com/doc/refman/5.0/en/mysqldump.html]. Contrairement aux datasets CSV, que `mysqldump` gère également, un unique fichier de ce format XML peut contenir des données pour de multiples tables. Vous pouvez créer un fichier dans ce format en invoquant `mysqldump` de cette façon :

```
mysqldump --xml -t -u [username] --password=[password] [database] > /path/to/file.xml
```

Ce fichier peut être utilisé dans votre case de test de base de données en appelant la méthode `createMySQLXMLDataSet($filename)`:

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

class MyTestCase extends TestCase
{
    use TestCaseTrait;

    public function getDataSet()
    {
        return $this->createMySQLXMLDataSet('/path/to/file.xml');
    }
}
?>
```

DataSet YAML

Alternativement, vous pouvez utiliser un dataset YAML pour l'exemple du livre d'or:

```
guestbook:
  -
    id: 1
    content: "Hello buddy!"
    user: "joe"
    created: 2010-04-24 17:15:23
  -
    id: 2
    content: "I like it!"
    user:
    created: 2010-04-26 12:14:20
```

C'est simple, pratique ET ça règle le problème de NULL que pose le dataset équivalent au format XML à plat. Un NULL en YAML s'exprime simplement en donnant le nom de la colonne sans indiquer de valeur. Une chaîne vide est indiquée par `column1: ""`.

Le dataset YAML ne possède pas actuellement de méthode de fabrique pour le cas de tests de base de données, si bien que vous devez l'instancier manuellement :

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;
use PHPUnit\DbUnit\DataSet\YamlDataSet;

class YamlGuestbookTest extends TestCase
{
    use TestCaseTrait;

    protected function getDataSet()
    {
        return new YamlDataSet(dirname(__FILE__)."/_files/guestbook.yml");
    }
}
?>
```

DataSet CSV

Un autre dataset au format fichier est basé sur les fichiers CSV. Chaque table du dataset est représenté par un fichier CSV. Pour notre exemple de livre d'or, nous pourrions définir un fichier guestbook-table.csv:

```
id,content,user,created
1,"Hello buddy!","joe","2010-04-24 17:15:23"
2,"I like it!","nancy","2010-04-26 12:14:20"
```

Bien que ce soit très pratique à éditer avec Excel ou OpenOffice, vous ne pouvez pas indiquer de valeurs NULL avec le dataset CSV. Une colonne vide conduira à ce que la valeur vide par défaut de la base de données soit insérée dans la colonne.

Vous pouvez créer un dataset CSV en appelant :

```
<?php
use PHPUnit\Framework\TestCase;
use PHPUnit\DbUnit\TestCaseTrait;
use PHPUnit\DbUnit\DataSet\CsvDataSet;

class CsvGuestbookTest extends TestCase
{
    use TestCaseTrait;

    protected function getDataSet()
    {
```

```
        $dataSet = new CsvDataSet();
        $dataSet->addTable('guestbook', dirname(__FILE__)."/_files/guestbook.csv");
        return $dataSet;
    }
}
?>
```

DataSet tableau

Il n'existe pas (encore) de DataSet basé sur les tableau dans l'extension base de données de PHPUnit, mais vous pouvez implémenter facilement la vôtre. Notre exemple du Livre d'or devrait ressembler à :

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

class ArrayGuestbookTest extends TestCase
{
    use TestCaseTrait;

    protected function getDataSet()
    {
        return new MyApp_DbUnit_ArrayDataSet(
            [
                'guestbook' => [
                    [
                        'id' => 1,
                        'content' => 'Hello buddy!',
                        'user' => 'joe',
                        'created' => '2010-04-24 17:15:23'
                    ],
                    [
                        'id' => 2,
                        'content' => 'I like it!',
                        'user' => null,
                        'created' => '2010-04-26 12:14:20'
                    ],
                ],
            ]
        );
    }
}
?>
```

Un DataSet PHP possède des avantages évidents sur les autres datasets utilisant des fichiers :

- Les tableaux PHP peuvent évidemment gérer les valeurs `NULL`.
- Vous n'avez pas besoin de fichiers additionnels pour les assertions et vous pouvez les renseigner directement dans les cas de test.

Pour que ce dataset ressemble aux DataSets au format XML à plat, CSV et YAML, les clefs de la première ligne spécifiée définissent les noms de colonne de la table, dans le cas précédent, ce serait « id », « content », « user » and « created ».

L'implémentation de ce DataSet tableau est simple et évidente:

```
<?php
class MyApp_DbUnit_ArrayDataSet extends PHPUnit_Extensions_Database_DataSet_AbstractData
{
    /**
     * @var array
     */
```

```
    protected $tables = [];

    /**
     * @param array $data
     */
    public function __construct(array $data)
    {
        foreach ($data AS $tableName => $rows) {
            $columns = [];
            if (isset($rows[0])) {
                $columns = array_keys($rows[0]);
            }

            $metaData = new PHPUnit_Extensions_Database_DataSet_DefaultTableMetaData($ta
            $table = new PHPUnit_Extensions_Database_DataSet_DefaultTable($metaData);

            foreach ($rows AS $row) {
                $table->addRow($row);
            }
            $this->tables[$tableName] = $table;
        }
    }

    protected function createIterator($reverse = false)
    {
        return new PHPUnit_Extensions_Database_DataSet_DefaultTableIterator($this->table
    }

    public function getTable($tableName)
    {
        if (!isset($this->tables[$tableName])) {
            throw new InvalidArgumentException("$tableName is not a table in the current
        }

        return $this->tables[$tableName];
    }
}
?>
```

Query (SQL) DataSet

Pour les assertions de base de données, vous n'avez pas seulement besoin de datasets basés sur des fichiers mais aussi de Datasets basé sur des requêtes/du SQL qui contiennent le contenu constaté de la base de données. C'est là que le DataSet Query s'illustre :

```
<?php
$ds = new PHPUnit_Extensions_Database_DataSet_QueryDataSet($this->getConnection());
$ds->addTable('guestbook');
?>
```

Ajouter une table juste par son nom est un moyen implicite de définir la table de données avec la requête suivante :

```
<?php
$ds = new PHPUnit_Extensions_Database_DataSet_QueryDataSet($this->getConnection());
$ds->addTable('guestbook', 'SELECT * FROM guestbook');
?>
```

Vous pouvez utiliser ceci en indiquant des requêtes arbitraires pour vos tables, par exemple en restreignant les lignes, les colonnes ou en ajoutant des clauses `ORDER BY`:

```
<?php
```

```
$ds = new PHPUnit_Extensions_Database_DataSet_QueryDataSet($this->getConnection());
$ds->addTable('guestbook', 'SELECT id, content FROM guestbook ORDER BY created DESC');
?>
```

La section relative aux assertions de base de données montrera plus en détails comment utiliser le Query DataSet.

Dataset (DB) de base de données

En accédant à la connexion de test, vous pouvez créer automatiquement un DataSet constitué de toutes les tables et de leur contenu de la base de données indiquée comme second paramètre de la méthode fabrique de connexion.

Vous pouvez, soit créer un dataset pour la base de données complète comme montré dans la méthode `testGuestbook()`, soit le restreindre à un ensemble de noms de tables avec une liste blanche comme montré dans la méthode `testFilteredGuestbook()`.

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

class MySqlGuestbookTest extends TestCase
{
    use TestCaseTrait;

    /**
     * @return PHPUnit_Extensions_Database_DB_IDatabaseConnection
     */
    public function getConnection()
    {
        $database = 'my_database';
        $user = 'my_user';
        $password = 'my_password';
        $pdo = new PDO('mysql:...', $user, $password);
        return $this->createDefaultDBConnection($pdo, $database);
    }

    public function testGuestbook()
    {
        $dataSet = $this->getConnection()->createDataSet();
        // ...
    }

    public function testFilteredGuestbook()
    {
        $tableNames = ['guestbook'];
        $dataSet = $this->getConnection()->createDataSet($tableNames);
        // ...
    }
}
?>
```

DataSet de remplacement

J'ai évoqué les problèmes de NULL avec les DataSet au format XML à plat et CSV, mais il y existe un contournement légèrement compliqué pour que ces deux types de datasets fonctionnent avec NULLs.

Le DataSet de remplacement est un décorateur pour un dataset existant et vous permet de remplacer des valeurs dans toute colonne du dataset par une autre valeur de remplacement. Pour que notre exemple de livre d'or fonctionne avec des valeurs NULL nous indiquons le fichier comme ceci:

```
<?xml version="1.0" ?>
```

```
<dataset>
    <guestbook id="1" content="Hello buddy!" user="joe" created="2010-04-24 17:15:23" />
    <guestbook id="2" content="I like it!" user="##NULL##" created="2010-04-26 12:14:20"
</dataset>
```

Nous encapsulons le DataSet au format XML à plat dans le DataSet de remplacement :

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

class ReplacementTest extends TestCase
{
    use TestCaseTrait;

    public function getDataSet()
    {
        $ds = $this->createFlatXmlDataSet('myFlatXmlFixture.xml');
        $rds = new PHPUnit_Extensions_Database_DataSet_ReplacementDataSet($ds);
        $rds->addFullReplacement('##NULL##', null);
        return $rds;
    }
}
?>
```

Filtre de DataSet

Si vous avez un fichier de fixture conséquent vous pouvez utiliser le filtre de DataSet pour des listes blanches ou noires des tables et des colonnes qui peuvent être contenues dans un sous-dataset. C'est particulièrement commode en combinaison avec le DataSet de base de données pour filtrer les colonnes des datasets.

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

class DataSetFilterTest extends TestCase
{
    use TestCaseTrait;

    public function testIncludeFilteredGuestbook()
    {
        $tableNames = ['guestbook'];
        $dataSet = $this->getConnection()->createDataSet();

        $filterDataSet = new PHPUnit_Extensions_Database_DataSet_DataSetFilter($dataSet)
        $filterDataSet->addIncludeTables(['guestbook']);
        $filterDataSet->setIncludeColumnsForTable('guestbook', ['id', 'content']);
        // ..
    }

    public function testExcludeFilteredGuestbook()
    {
        $tableNames = ['guestbook'];
        $dataSet = $this->getConnection()->createDataSet();

        $filterDataSet = new PHPUnit_Extensions_Database_DataSet_DataSetFilter($dataSet)
        $filterDataSet->addExcludeTables(['foo', 'bar', 'baz']); // only keep the guestb
        $filterDataSet->setExcludeColumnsForTable('guestbook', ['user', 'created']);
        // ..
    }
}
?>
```

NOTE Vous ne pouvez pas utiliser en même temps le filtrage de colonne d'inclusion et d'exclusion sur la même table, seulement sur des tables différentes. De plus, il est seulement possible d'appliquer soit une liste blanche, soit une liste noire aux tables, mais pas les deux à la fois.

DataSet composite

Le DataSet composite est très utile pour agréger plusieurs datasets déjà existants dans un unique dataset. Quand plusieurs datasets contiennent la même table, les lignes sont ajoutées dans l'ordre indiqué. Par exemple, si nous avons deux datasets *fixture1.xml* :

```
<?xml version="1.0" ?>
<dataset>
    <guestbook id="1" content="Hello buddy!" user="joe" created="2010-04-24 17:15:23" />
</dataset>
```

et *fixture2.xml*:

```
<?xml version="1.0" ?>
<dataset>
    <guestbook id="2" content="I like it!" user="##NULL##" created="2010-04-26 12:14:20"
</dataset>
```

En utiliser le DataSet composite, nous pouvons agréger les deux fichiers de fixture:

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

class CompositeTest extends TestCase
{
    use TestCaseTrait;

    public function getDataSet()
    {
        $ds1 = $this->createFlatXmlDataSet('fixture1.xml');
        $ds2 = $this->createFlatXmlDataSet('fixture2.xml');

        $compositeDs = new PHPUnit_Extensions_Database_DataSet_CompositeDataSet();
        $compositeDs->addDataSet($ds1);
        $compositeDs->addDataSet($ds2);

        return $compositeDs;
    }
}
?>
```

Attention aux clefs étrangères

Lors du SetUp de la fixture l'extension de base de données de PHPUnit insère les lignes dans la base de données dans l'ordre où elles sont indiquées dans votre fixture. Si votre schéma de base de données utilise des clefs étrangères, ceci signifie que vous devez indiquer les tables dans un ordre qui ne provoquera pas une violation de contrainte pour ces clefs étrangères.

Implementer vos propres DataSets/DataTables

Pour comprendre le fonctionnement interne des DataSets et des DataTables jetons un oeil sur l'interface d'un DataSet. Vous pouvez sauter cette partie si vous ne projetez pas d'implémenter votre propre DataSet ou DataTable.

```
<?php
```

```
interface PHPUnit_Extensions_Database_DataSet_IDataSet extends IteratorAggregate
{
    public function getTableNames();
    public function getTableMetaData($tableName);
    public function getTable($tableName);
    public function assertEquals(PHPUnit_Extensions_Database_DataSet_IDataSet $other);

    public function getReverseIterator();
}
?>
```

L'interface publique est utilisée en interne par l'assertion `assertDataSetsEqual()` du cas de test de base de données pour contrôler la qualité du dataset. De l'interface `IteratorAggregate` le IDataSet hérite la méthode `getIterator()` pour parcourir toutes les tables du dataset. La méthode additionnelle d'itérateur inverse est nécessaire pour réussir à tronquer les tables dans l'ordre inverse à celui indiqué pour satisfaire les contraintes de clés étrangères.

En fonction de l'implémentation, différentes approches sont prises pour ajouter des instances de table dans un dataset. Par exemple, les tables sont ajoutées de façon interne lors de la construction depuis le fichier source dans tous les datasets basés sur les fichiers comme `YamlDataSet`, `XmlDataSet` ou `FlatXmlDataSet`.

Une table est également représentée par l'interface suivante :

```
<?php
interface PHPUnit_Extensions_Database_DataSet_ITable
{
    public function getTableMetaData();
    public function getRowCount();
    public function getValue($row, $column);
    public function getRow($row);
    public function assertEquals(PHPUnit_Extensions_Database_DataSet_ITable $other);
}
?>
```

Mise à part la méthode `getTableMetaData()`, ça parle plutôt de soi-même. Les méthodes utilisées sont toutes nécessaires pour les différentes assertions de l'extension Base de données expliquées dans le chapitre suivant. La méthode `getTableMetaData()` doit retourner une implémentation de l'interface `PHPUnit_Extensions_Database_DataSet_ITableMetaData` qui décrit la structure de la table. Elle contient des informations sur:

- Le nom de la table
- Un tableau des noms de colonne de la table, classé par leur ordre d'apparition dans l'ensemble résultat.
- Un tableau des colonnes clefs primaires.

Cette interface possède également une assertion qui contrôle si deux instances des méta données des tables sont égales et qui sera utilisée par l'assertion d'égalité d'ensemble de données.

L'API de connexion

Il y a trois méthodes intéressantes dans l'interface de connexion qui doit être retournée par la méthode `getConnection()` du cas de test de base de données :

```
<?php
interface PHPUnit_Extensions_Database_DB_IDatabaseConnection
{
    public function createDataSet(Array $tableNames = NULL);
    public function createQueryTable($resultName, $sql);
    public function getRowCount($tableName, $whereClause = NULL);
```

```
    // ...
}
?>
```

1. La méthode `createDataSet()` crée un DataSet de base de données (DB) comme décrit dans la section relative aux implémentations de DataSet.

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

class ConnectionTest extends TestCase
{
    use TestCaseTrait;

    public function testCreateDataSet()
    {
        $tableNames = ['guestbook'];
        $dataSet = $this->getConnection()->createDataSet();
    }
}
?>
```

2. La méthode `createQueryTable()` peut être utilisée pour créer des instances d'une Query-Table, en lui passant un nom de résultat et une requête SQL. C'est une méthode pratique quand elle est associée à des assertions résultats/table comme cela sera illustré dans la prochaine section relative à l'API des assertions de base de données.

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

class ConnectionTest extends TestCase
{
    use TestCaseTrait;

    public function testCreateQueryTable()
    {
        $tableNames = ['guestbook'];
        $queryTable = $this->getConnection()->createQueryTable('guestbook', 'SELECT *
    }
}
?>
```

3. La méthode `getRowCount()` est un moyen pratique d'accéder au nombre de lignes d'une table, éventuellement filtrées par une clause where supplémentaire. Ceci peut être utilisé pour une simple assertion d'égalité :

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

class ConnectionTest extends TestCase
{
    use TestCaseTrait;

    public function testGetRowCount()
    {
        $this->assertEquals(2, $this->getConnection()->getRowCount('guestbook'));
    }
}
```

```
?>
```

API d'assertion de base de données

En tant qu'outil de test, l'extension base de données fournit certainement des assertions que vous pouvez utiliser pour vérifier l'état actuel de la base de données, des tables et du nombre de lignes des tables. Cette section décrit ces fonctionnalités en détail :

Faire une assertion sur le nombre de lignes d'une table

Il est souvent très utile de vérifier si une table contient un nombre déterminé de lignes. Vous pouvez facilement réaliser cela sans code de liaison supplémentaire en utilisant l'API de connexion. Disons que nous voulons contrôler qu'après une insertion d'une ligne dans notre livre d'or, nous n'avons plus seulement nos deux entrées initiales qui nous ont accompagnées dans tous les exemples précédents, mais aussi une troisième :

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

class GuestbookTest extends TestCase
{
    use TestCaseTrait;

    public function testAddEntry()
    {
        $this->assertEquals(2, $this->getConnection()->getRowCount('guestbook'), "Pre-Co

        $guestbook = new Guestbook();
        $guestbook->addEntry("suzy", "Hello world!");

        $this->assertEquals(3, $this->getConnection()->getRowCount('guestbook'), "Insert
    }
}
?>
```

Faire une assertion sur l'état d'une table

L'assertion précédent est utile, mais nous voudrons certainement tester le contenu présent de la table pour vérifier que toutes les valeurs ont été écrites dans les bonnes colonnes. Ceci peut être réalisé avec une assertion de table.

Pour cela, nous devons définir une instance de Query Table qui tire son contenu d'un nom de table et d'une requête SQL et le compare à un DataSet basé sur un fichier/tableau.

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

class GuestbookTest extends TestCase
{
    use TestCaseTrait;

    public function testAddEntry()
    {
        $guestbook = new Guestbook();
        $guestbook->addEntry("suzy", "Hello world!");

        $queryTable = $this->getConnection()->createQueryTable(
```

```
            'guestbook', 'SELECT * FROM guestbook'
        );
        $expectedTable = $this->createFlatXmlDataSet("expectedBook.xml")
                              ->getTable("guestbook");
        $this->assertTablesEqual($expectedTable, $queryTable);
    }
}
?>
```

Maintenant, nous devons écrire le fichier XML à plat *expectedBook.xml* pour cette assertion :

```
<?xml version="1.0" ?>
<dataset>
    <guestbook id="1" content="Hello buddy!" user="joe" created="2010-04-24 17:15:23" />
    <guestbook id="2" content="I like it!" user="nancy" created="2010-04-26 12:14:20" />
    <guestbook id="3" content="Hello world!" user="suzy" created="2010-05-01 21:47:08" /
</dataset>
```

Cette assertion ne réussira que si elle est lancée très exactement le *2010–05–01 21:47:08*. Les dates posent un problème spécial pour le test de base de données et nous pouvons contourner l'échec en omettant la colonne « created » de l'assertion.

Le fichier au format XML à plat adapté *expectedBook.xml* devra probablement ressembler à ce qui suit pour que l'assertion réussisse :

```
<?xml version="1.0" ?>
<dataset>
    <guestbook id="1" content="Hello buddy!" user="joe" />
    <guestbook id="2" content="I like it!" user="nancy" />
    <guestbook id="3" content="Hello world!" user="suzy" />
</dataset>
```

Nous devons corriger l'appel à Query Table:

```
<?php
$queryTable = $this->getConnection()->createQueryTable(
    'guestbook', 'SELECT id, content, user FROM guestbook'
);
?>
```

Faire une assertion sur le résultat d'une requête

Vous pouvez également faire une assertion sur le résultat de requêtes complexes avec l'approche Query Table, simplement en indiquant le nom d'un résultat avec une requête et en le comparant avec un ensemble de données:

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

class ComplexQueryTest extends TestCase
{
    use TestCaseTrait;

    public function testComplexQuery()
    {
        $queryTable = $this->getConnection()->createQueryTable(
            'myComplexQuery', 'SELECT complexQuery...'
        );
        $expectedTable = $this->createFlatXmlDataSet("complexQueryAssertion.xml")
                              ->getTable("myComplexQuery");
        $this->assertTablesEqual($expectedTable, $queryTable);
```

```
    }
}
?>
```

Faire une assertion sur l'état de plusieurs tables

Evidemment, vous pouvez faire une assertion sur l'état de plusieurs tables à la fois et comparer un ensemble de données obtenu par une requête avec un ensemble de données basé sur un fichier. Il y a deux façons différentes de faire des assertions de DataSet.

1. Vous pouvez utiliser le Database (DB) Dataset à partir de la connexion et le comparer au DataSet basé sur un fichier.

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

class DataSetAssertionsTest extends TestCase
{
    use TestCaseTrait;

    public function testCreateDataSetAssertion()
    {
        $dataSet = $this->getConnection()->createDataSet(['guestbook']);
        $expectedDataSet = $this->createFlatXmlDataSet('guestbook.xml');
        $this->assertDataSetsEqual($expectedDataSet, $dataSet);
    }
}
?>
```

2. Vous pouvez construire vous-même le DataSet:

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

class DataSetAssertionsTest extends TestCase
{
    use TestCaseTrait;

    public function testManualDataSetAssertion()
    {
        $dataSet = new PHPUnit_Extensions_Database_DataSet_QueryDataSet();
        $dataSet->addTable('guestbook', 'SELECT id, content, user FROM guestbook'); //
        $expectedDataSet = $this->createFlatXmlDataSet('guestbook.xml');

        $this->assertDataSetsEqual($expectedDataSet, $dataSet);
    }
}
?>
```

Foire aux questions

PHPUnit va-t'il (re-)créer le schéma de base de données pour chaque test ?

Non, PHPUnit exige que tous les objets de base de données soit disponible quand la suite démarre. La base de données, les tables, les séquences, les triggers et les vues doivent être créés avant que vous exécutiez la suite de tests.

Doctrine 2 [http://www.doctrine-project.org] ou eZ Components [http://www.ezcomponents.org] possèdent des outils puissants qui vous permettent de créer le schéma de base de données à partir de structures de données définies préalablement, cependant, ceux-ci doivent être reliés à l'extension PHPUnit pour permettre la recréation automatique de la base de données avant que la suite de tests complète ne soit exécutée.

Puisque chaque test nettoie complètement la base de données, vous n'avez même pas obligation de re-créer la base de donnée pour chaque exécution des tests. Une base de données disponible de façon permanente fonctionne parfaitement.

Suis-je obligé d'utiliser PDO dans mon application pour que l'extension de base de données fonctionne ?

Non, PDO n'est nécessaire que pour le nettoyage et la configuration de la fixture et pour les assertions. Vous pouvez utiliser n'importe laquelle des abstractions de base de données que vous voulez dans votre propre code.

Que puis-je faire quand j'obtiens une erreur « Too much Connections (Trop de connexions) » ?

Si vous ne mettez pas en cache l'instance PDO qui est créée dans la méthode `getConnection()` du cas de test le nombre de connexions à la base de données est augmenté d'une unité ou plus pour chaque test de base de données. Avec la configuration par défaut, MySql n'autorise qu'un maximum de 100 connexions concurrentes. Les autres moteurs de bases de données possèdent également des limites du nombre maximum de connexions.

La sous-section « Utilisez votre propre cas de test de base de données abstrait » illustre comment vous pouvez empêcher cette erreur de survenir en utilisant une unique instance de PDO en cache dans tous vos tests.

Comment gérer les valeurs NULL avec les DataSets au format XML à plat / CSV ?

Ne le fait pas. Pour cela, vous devez utiliser des DataSets XML ou YAML.

Chapitre 9. Doublure de test

Gerard Meszaros introduit le concept de doublure de test dans [Meszaros2007] comme ceci:

> Parfois il est parfaitement difficile de juste tester un système en cours de test (System Under Test : SUT) parce qu'il dépend d'autres composants qui ne peuvent pas être utilisés dans l'environnement de test. Ceci peut provenir du fait qu'ils ne sont pas disponibles, qu'ils ne retournent pas les résultats nécessaires pour les tests ou parce que les exécuter pourrait avoir des effets de bord indésirables. Dans d'autres cas, notre stratégie de test nécessite que nous ayons plus de contrôle ou de visibilité sur le comportement interne du SUT.
>
> Quand nous écrivons un test dans lequel nous ne pouvons pas (ou ne voulons pas) utiliser un composant réel dont on dépend (depended-on component ou DOC), nous pouvons le remplacer avec une doublure de test. La doublure de test ne se comporte pas exactement comme un vrai DOC; elle a simplement à fournir la même API que le composant réel de telle sorte que le système testé pense qu'il s'agit du vrai !
>
> —Gerard Meszaros

Les méthodes `createMock($type)` et `getMockBuilder($type)` fourni par PHPUnit peuvent être utilisées dans un test pour générer automatiquement un objet qui peut agir comme une doublure de test pour une classe originelle indiquée (interface ou non de classe). Cette doublure de test peut être utilisée dans tous les contextes où la classe originelle est attendue ou requise.

La méthode `createMock($type)` retourne immédiatement une doublure de test pour le type spécifié (interface ou classe). La création de cette doublure est effectuée en suivant par défaut les bonne pratiques (les méthodes `__construct()` et `__clone()` de la classe originale ne sont pas executées et les arguments passés à une méthode de la doublure de tests ne sont pas clonés. Si ce comportement par défaut ne correspondent pas à ce don vous avez besoin vous pouvez alors utiliser la méthode `getMockBuilder($type)` pour personnaliser la génération de doublure de test en utilisant un interface souple (fluent interface).

Par défaut, toutes les méthodes de la classe originelle sont remplacées par une implémentation fictive qui se contente de retourner `null` (sans appeler la méthode originelle). En utilisant la méthode `will($this->returnValue())` par exemple, vous pouvez configurer ces implémentations fictives pour retourner une valeur donnée quand elles sont appelées.

Limitations: méthodes final, private et static

Merci de noter que les méthodes `final`, `private` et `static` ne peuvent pas être remplacées par un bouchon (stub) ou un simulacre (mock). Elles seront ignorées par la fonction de doublure de test de PHPUnit et conserveront leur comportement initial.

Bouchons

La pratique consistant à remplacer un objet par une doublure de test qui retourne (de façon facultative) des valeurs de retour configurées est appelée *bouchonnage*. Vous pouvez utiliser un *bouchon* pour "remplacer un composant réel dont dépend le système testé de telle façon que le test possède un point de contrôle sur les entrées indirectes dans le SUT. Ceci permet au test de forcer le SUT à utiliser des chemins qu'il n'aurait pas emprunté autrement".

Exemple 9.2, « Bouchonner un appel de méthode pour retourner une valeur fixée » montre comment la méthode de bouchonnage appelle et configure des valeurs de retour. Nous utilisons d'abord la méthode `createMock()` qui est fournie par la classe `PHPUnit\Framework\TestCase` pour configurer un objet bouchon qui ressemble à un objet de `SomeClass` (Exemple 9.1, « La classe que nous voulons bouchonner »). Ensuite nous utilisons l'interface souple [http://martinfowler.com/bliki/FluentInterface.html] que PHPUnit fournit pour indiquer le comportement de ce bouchon. En

substance, cela signifie que vous n'avez pas besoin de créer plusieurs objets temporaires et les relier ensemble ensuite. Au lieu de cela, vous chaînez les appels de méthode comme montré dans l'exemple. Ceci amène à un code plus lisible et "souple".

Exemple 9.1. La classe que nous voulons bouchonner

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

class SomeClass
{
    public function doSomething()
    {
        // Do something.
    }
}
?>
```

Exemple 9.2. Bouchonner un appel de méthode pour retourner une valeur fixée

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

class StubTest extends TestCase
{
    public function testStub()
    {
        // Créer un bouchon pour la classe SomeClass.
        $stub = $this->createMock(SomeClass::class);

        // Configurer le bouchon.
        $stub->method('doSomething')
             ->willReturn('foo');

        // Appeler $stub->doSomething() va maintenant retourner
        // 'foo'.
        $this->assertEquals('foo', $stub->doSomething());
    }
}
?>
```

Limitation: Méthodes nommées "method"

L'exemple ci dessus ne fonctionne que si la classe originale ne déclare pas de méthode appelé "method".

Si la classe originale déclare une methode appelée "method" alors vous devez utiliser `$stub->expects($this->any())->method('doSomething')->willReturn('foo');`.

"Dans les coulisses", PHPUnit génère automatiquement une nouvelle classe qui implémente le comportement souhaité quand la méthode `createMock()` est utilisée.

Exemple 9.3, « L'API de construction des mocks peut-être utilisée pour configurer la doublure de test générée. » montre un exemple de comment utiliser l'interface souple du créateur de mock pour configurer la création d'une doublure de test. La configuration de cette doublure de test utilise les même bonnes pratiques utilisées par défaut par `createMock()`.

Exemple 9.3. L'API de construction des mocks peut-être utilisée pour configurer la doublure de test générée.

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

class StubTest extends TestCase
{
    public function testStub()
    {
        // Créer un bouchon pour la classe SomeClass.
        $stub = $this->getMockBuilder($originalClassName)
                     ->disableOriginalConstructor()
                     ->disableOriginalClone()
                     ->disableArgumentCloning()
                     ->disallowMockingUnknownTypes()
                     ->getMock();

        // Configurer le bouchon.
        $stub->method('doSomething')
             ->willReturn('foo');

        // Appeler $stub->doSomething() retournera désormais
        // 'foo'.
        $this->assertEquals('foo', $stub->doSomething());
    }
}
?>
```

Dans les exemples précédents, nous avons retourné des valeurs simple en utilisant `willReturn($value)`. Cette syntaxe courte est identique à `will($this->returnValue($value))`. Nous pouvons utiliser des variantes de cette syntaxte plus longue pour obtenir un comportement de bouchonnement plus complexe.

Parfois vous voulez renvoyer l'un des paramètres d'un appel de méthode (non modifié) comme résultat d'un appel méthode bouchon. Exemple 9.4, « Bouchonner un appel de méthode pour renvoyer un des paramètres » montre comment vous pouvez obtenir ceci en utilisant `returnArgument()` à la place de `returnValue()`.

Exemple 9.4. Bouchonner un appel de méthode pour renvoyer un des paramètres

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

class StubTest extends TestCase
{
    public function testReturnArgumentStub()
    {
        // Créer un bouchon pour la classe SomeClass.
        $stub = $this->createMock(SomeClass::class);

        // Configurer le bouchon.
        $stub->method('doSomething')
             ->will($this->returnArgument(0));

        // $stub->doSomething('foo') retourn 'foo'
        $this->assertEquals('foo', $stub->doSomething('foo'));

        // $stub->doSomething('bar') returns 'bar'
        $this->assertEquals('bar', $stub->doSomething('bar'));
    }
}
?>
```

Quand on teste interface souple, il est parfois utile que la méthode bouchon retourne une référence à l'objet bouchon. Exemple 9.5, « Bouchonner un appel de méthode pour renvoyer une référence de l'objet bouchon. » présente comment utiliser `returnSelf()` pour accomplir cela.

Exemple 9.5. Bouchonner un appel de méthode pour renvoyer une référence de l'objet bouchon.

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

class StubTest extends TestCase
{
    public function testReturnSelf()
    {
        // Créer un bouchon pour la classe SomeClass.
        $stub = $this->createMock(SomeClass::class);

        // Configurer le bouchon.
        $stub->method('doSomething')
             ->will($this->returnSelf());

        // $stub->doSomething() retourne $stub
        $this->assertSame($stub, $stub->doSomething());
    }
}
?>
```

Parfois, une méthode bouchon doit retourner différentes valeurs selon une liste prédéfinie d'arguments. Vous pouvez utiliser `returnValueMap()` pour créer une association entre les paramètres et les valeurs de retour correspondantes. Voir Exemple 9.6, « Bouchonner un appel de méthode pour retourner la valeur à partir d'une association » pour un exemple.

Exemple 9.6. Bouchonner un appel de méthode pour retourner la valeur à partir d'une association

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

class StubTest extends TestCase
{
    public function testReturnValueMapStub()
    {
        // Créer un bouchon pour la classe SomeClass.
        $stub = $this->createMock(SomeClass::class);

        // Créer une association entre arguments et valeurs de retour
        $map = [
            ['a', 'b', 'c', 'd'],
            ['e', 'f', 'g', 'h']
        ];

        // Configurer le bouchon.
        $stub->method('doSomething')
             ->will($this->returnValueMap($map));

        // $stub->doSomething() retourne différentes valeurs selon
        // les paramètres fournis.
        $this->assertEquals('d', $stub->doSomething('a', 'b', 'c'));
        $this->assertEquals('h', $stub->doSomething('e', 'f', 'g'));
    }
}
```

```
?>
```

Quand l'appel d'une méthode bouchonné doit retourner une valeur calculée au lieu d'une valeur fixée (voir `returnValue()`) ou un paramètre (non modifié) (voir `returnArgument()`), vous pouvez utiliser `returnCallback()` pour que la méthode retourne le résultat d'une fonction ou méthode de rappel. Voir Exemple 9.7, « Bouchonner un appel de méthode pour retourner une valeur à partir d'une fonction de rappel » pour un exemple.

Exemple 9.7. Bouchonner un appel de méthode pour retourner une valeur à partir d'une fonction de rappel

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

class StubTest extends TestCase
{
    public function testReturnCallbackStub()
    {
        // Créer un bouchon pour la classe SomeClass.
        $stub = $this->createMock(SomeClass::class);

        // Configurer le bouchon.
        $stub->method('doSomething')
             ->will($this->returnCallback('str_rot13'));

        // $stub->doSomething($argument) retourne str_rot13($argument)
        $this->assertEquals('fbzrguvat', $stub->doSomething('something'));
    }
}
?>
```

Une alternative plus simple pour configurer une méthode de rappel peut consister à indiquer une liste de valeurs désirées. Vous pouvez faire ceci avec la méthode `onConsecutiveCalls()`. Voir Exemple 9.8, « Bouchonner un appel de méthode pour retourner une liste de valeurs dans l'ordre indiqué » pour un exemple.

Exemple 9.8. Bouchonner un appel de méthode pour retourner une liste de valeurs dans l'ordre indiqué

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

class StubTest extends TestCase
{
    public function testOnConsecutiveCallsStub()
    {
        // Créer un bouchon pour la classe SomeClass.
        $stub = $this->createMock(SomeClass::class);

        // Configurer le bouchon.
        $stub->method('doSomething')
             ->will($this->onConsecutiveCalls(2, 3, 5, 7));

        // $stub->doSomething() retourne une valeur différente à chaque fois
        $this->assertEquals(2, $stub->doSomething());
        $this->assertEquals(3, $stub->doSomething());
        $this->assertEquals(5, $stub->doSomething());
    }
}
?>
```

Au lieu de retourner une valeur, une méthode bouchon peut également lever une exception. Exemple 9.9, « Bouchonner un appel de méthode pour lever une exception » montre comme utiliser `throwException()` pour faire cela.

Exemple 9.9. Bouchonner un appel de méthode pour lever une exception

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

class StubTest extends TestCase
{
    public function testThrowExceptionStub()
    {
        // Créer un bouchon pour la classe SomeClass.
        $stub = $this->createMock(SomeClass::class);

        // Configurer le bouchon.
        $stub->method('doSomething')
             ->will($this->throwException(new Exception));

        // $stub->doSomething() throws Exception
        $stub->doSomething();
    }
}
?>
```

Alternativement, vous pouvez écrire le bouchon vous-même et améliorer votre conception en cours de route. Des ressources largement utilisées sont accédées via une unique façade, de telle sorte que vous pouvez facilement remplacer la ressource avec le bouchon. Par exemple, au lieu d'avoir des appels directs à la base de données éparpillés dans tout le code, vous avez un unique objet `Database`, une implémentation de l'interface `IDatabase`. Ensuite, vous pouvez créer une implémentation bouchon de `IDatabase` et l'utiliser pour vos tests. Vous pouvez même créer une option pour lancer les tests dans la base de données bouchon ou la base de données réelle, de telle sorte que vous pouvez utiliser vos tests à la fois pour tester localement pendant le développement et en intégration avec la vraie base de données.

Les fonctionnalités qui nécessitent d'être bouchonnées tendent à se regrouper dans le même objet, améliorant la cohésion. En représentant la fonctionnalité avec une unique interface cohérente, vous réduisez le couplage avec le reste du système.

Objets Mock

La pratique consistant à remplacer un objet avec une doublure de test qui vérifie des attentes, par exemple en faisant l'assertion qu'une méthode a été appelée, est appelée *mock*.

Vous pouvez utiliser un *objet mock* "comme un point d'observation qui est utilisé pour vérifier les sorties indirectes du système quand il est testé". Typiquement, le mock inclut également la fonctionnalité d'un bouchon de test, en ce sens qu'il doit retourner les valeurs du système testé s'il n'a pas déjà fait échouer les tests mais l'accent est mis sur la vérification des sorties indirectes. Ainsi, un mock est un beaucoup plus qu'un simple bouchon avec des assertions; il est utilisé d'une manière fondamentalement différente" (Gerard Meszaros).

Limitation: Vérification automatique des attentes

Seuls les objets mock générés dans le scope d'un test seront vérifiés automatiquement par PHPUnit. Les mocks générés dans les fournisseurs de données, par exemple, ou injectés dans les tests en utilisant l'annotation `@depends` ne seront pas vérifiés automatiquement par PHPUnit.

Voici un exemple: supposons que vous voulez tester que la méthode correcte, `update()` dans notre exemple, est appelée d'un objet qui observe un autre objet. Exemple 9.10, « Les classes Subject et Observer qui sont une partie du système testé » illustre le code pour les classes `Subject` et `Observer` qui sont une partie du système testé (SUT).

Exemple 9.10. Les classes Subject et Observer qui sont une partie du système testé

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

class Subject
{
    protected $observers = [];
    protected $name;

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

    public function getName()
    {
        return $this->name;
    }

    public function attach(Observer $observer)
    {
        $this->observers[] = $observer;
    }

    public function doSomething()
    {
        // Faire quelque chose.
        // ...

        // Notify les observateurs que nous faisons quelque chose
        $this->notify('something');
    }

    public function doSomethingBad()
    {
        foreach ($this->observers as $observer) {
            $observer->reportError(42, 'Something bad happened', $this);
        }
    }

    protected function notify($argument)
    {
        foreach ($this->observers as $observer) {
            $observer->update($argument);
        }
    }

    // Autres méthodes.
}

class Observer
{
    public function update($argument)
    {
        // Faire quelquechose
    }
```

```
    public function reportError($errorCode, $errorMessage, Subject $subject)
    {
        // Faire quelquechose
    }

    // Autre méthodes
}
?>
```

Exemple 9.11, « Tester qu'une méthode est appelée une fois et avec un paramètre indiqué » illustre comment utiliser un simulacre pour tester l'interaction entre les objets `Subject` et `Observer`.

Nous utilisons d'abord la méthode `getMockBuilder()` qui est fournie par la classe `PHPUnit\Framework\TestCase` pour configurer un simulacre pour `Observer`. Puisque nous donnons un tableau comme second paramètre (facultatif) pour la méthode `getMock()`, seule la méthode `update()` de la classe `Observer` est remplacée par une implémentation d'un simulacre.

Comme ce qui nous intéresse est de vérifier qu'une méthode soit appelée, et avec quels arguments, nous introduisons les méthodes `expects()` et `with()` pour spécifier comment cette interaction doit se présenter.

Exemple 9.11. Tester qu'une méthode est appelée une fois et avec un paramètre indiqué

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

class SubjectTest extends TestCase
{
    public function testObserversAreUpdated()
    {
        // Créer un simulacre pour la classe Observer,
        // ne touchant que la méthode update().
        $observer = $this->getMockBuilder(Observer::class)
                         ->setMethods(['update'])
                         ->getMock();

        // Configurer l'attente de la méthode update()
        // d'être appelée une seule fois et avec la chaîne 'something'
        // comme paramètre.
        $observer->expects($this->once())
                 ->method('update')
                 ->with($this->equalTo('something'));

        // Créer un objet Subject et y attacher l'objet
        // Observer simulé
        $subject = new Subject('My subject');
        $subject->attach($observer);

        // Appeler la méthode doSomething() sur l'objet $subject
        // que nous attendons voir appeler la méthode update() de l'objet
        // simulé Observer avec la chaîne 'something'.
        $subject->doSomething();
    }
}
?>
```

La méthode `with()` peut prendre n'importe quel nombre de paramètres, correspondant au nombre de paramètres des méthodes simulées. Vous pouvez indiquer des contraintes plus avancées qu'une simple correspondance, sur les paramètres de méthode.

Exemple 9.12. Tester qu'une méthode est appelée avec un nombre de paramètres contraints de différentes manières

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

class SubjectTest extends TestCase
{
    public function testErrorReported()
    {
        // Créer un mock pour la classe Observer, en simulant
        // la méthode reportError()
        $observer = $this->getMockBuilder(Observer::class)
                         ->setMethods(['reportError'])
                         ->getMock();

        $observer->expects($this->once())
                 ->method('reportError')
                 ->with(
                       $this->greaterThan(0),
                       $this->stringContains('Something'),
                       $this->anything()
                   );

        $subject = new Subject('My subject');
        $subject->attach($observer);

        // La méthode doSomethingBad() doit rapporter une erreur à l'observateur
        // via la méthode reportError()
        $subject->doSomethingBad();
    }
}
?>
```

La méthode `withConsecutive()` peut prendre n'importe quel nombre de tableau de paramètres, selon les appels que vous souhaitez tester. Chaque tableau est une liste de contraintes correspondant aux paramètres de la méthode mockée, comme avec `with()`.

Exemple 9.13. Tester qu'une méthode est appelée deux fois avec des arguments spécifiques.

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

class FooTest extends TestCase
{
    public function testFunctionCalledTwoTimesWithSpecificArguments()
    {
        $mock = $this->getMockBuilder(stdClass::class)
                     ->setMethods(['set'])
                     ->getMock();

        $mock->expects($this->exactly(2))
             ->method('set')
             ->withConsecutive(
                 [$this->equalTo('foo'), $this->greaterThan(0)],
                 [$this->equalTo('bar'), $this->greaterThan(0)]
             );

        $mock->set('foo', 21);
        $mock->set('bar', 48);
    }
```

```
}
?>
```

La contrainte `callback()` peut être utilisée pour une vérification plus complexe d'un argument. Cette contrainte prend comme seul paramètre une fonction de rappel PHP (callback). La fonctione de rappel PHP recevra l'argument à vérifier comme son seul paramètre et devrait renvoyer `true` si l'argument passe la vérification et `false` sinon.

Exemple 9.14. Vérification de paramètre plus complexe

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

class SubjectTest extends TestCase
{
    public function testErrorReported()
    {
        // Crée un mock pour la classe Observer, mock de la
        // méthode reportError()
        $observer = $this->getMockBuilder(Observer::class)
                         ->setMethods(['reportError'])
                         ->getMock();

        $observer->expects($this->once())
                 ->method('reportError')
                 ->with($this->greaterThan(0),
                        $this->stringContains('Something'),
                        $this->callback(function($subject){
                          return is_callable([$subject, 'getName']) &&
                                 $subject->getName() == 'My subject';
                        }));

        $subject = new Subject('My subject');
        $subject->attach($observer);

        // La méthode doSomethingBad() devrait rapporter une erreur a l'observeur
        // via la methode reportError()
        $subject->doSomethingBad();
    }
}
?>
```

Exemple 9.15. Tester qu'une méthode est appelée une seule fois avec le même objet qui a été passé

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

class FooTest extends TestCase
{
    public function testIdenticalObjectPassed()
    {
        $expectedObject = new stdClass;

        $mock = $this->getMockBuilder(stdClass::class)
                     ->setMethods(['foo'])
                     ->getMock();

        $mock->expects($this->once())
             ->method('foo')
             ->with($this->identicalTo($expectedObject));
```

```
        $mock->foo($expectedObject);
    }
}
?>
```

Exemple 9.16. Créer un OBJET mock avec les paramètres de clonnage activés

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

class FooTest extends TestCase
{
    public function testIdenticalObjectPassed()
    {
        $cloneArguments = true;

        $mock = $this->getMockBuilder(stdClass::class)
                     ->enableArgumentCloning()
                     ->getMock();

        // maintenant votre mock clone les paramètres, ainsi la contrainte identicalTo
        // échouera.
    }
}
?>
```

Tableau A.1, « Contraintes » montre les contraintes qui peuvent être appliquées aux paramètres de méthode et Tableau 9.1, « Matchers » montre les matchers qui sont disponibles pour indiquer le nombre d'invocations.

Tableau 9.1. Matchers

Matcher	Signification
`PHPUnit_ Framework_ MockObject_ Matcher_ AnyInvokedCount any()`	Retourne un matcher qui correspond quand la méthode pour laquelle il est évalué est exécutée zéro ou davantage de fois.
`PHPUnit_ Framework_ MockObject_ Matcher_ InvokedCount never()`	Retourne un matcher qui correspond quand la méthode pour laquelle il est évalué n'est jamais exécutée.
`PHPUnit_ Framework_ MockObject_ Matcher_ InvokedAtLeastOnce atLeastOnce()`	Retourne un matcher qui correspond quand la méthode pour laquelle il est évalué est exécutée au moins une fois.
`PHPUnit_ Framework_ MockObject_ Matcher_ InvokedCount once()`	Retourne un matcher qui correspond quand la méthode pour laquelle il est évalué est exécutée exactement une fois.
`PHPUnit_ Framework_ MockObject_ Matcher_ InvokedCount exactly(int $count)`	Retourne un matcher qui correspond quand la méthode pour laquelle il est évalué est exécutée exactement `$count` fois.
`PHPUnit_ Framework_ MockObject_ Matcher_ InvokedAtIndex at(int $index)`	Retourne un matcher qui correspond quand la méthode pour laquelle il est évalué est invoquée pour l'`$index` spécifié.

Note

Le paramètre `$index` du matcher `at()` fait référence à l'index, démarrant à zero, dans *toutes les invocations de la méthode* pour un objet mock. Faites preuve de prudence lors de l'utilisation de ce matcher car cela peut conduire à des tests fragiles qui seront trop étroitement liés aux détails d'implémentation spécifiques.

Comme mentionné au début, quand le comportement par défaut utilisé par la méthode `createMock()` pour générer la doublure de test ne correspond pas a vos besoins alors vous pouvez utiliser la méthode `getMockBuilder($type)` pour personnaliser la génération de la doublure de test en utilisant une interface souple. Voici une liste des méthodes fournies par le constructeur de mock:

- `setMethods(array $methods)` peut être appelé sur l'objet Mock Builder pour spécifier les méthodes qui doivent être remplacées par une une doublure de test configurable. Le comportement des autres méthodes n'est pas changé. Si vous appelez `setMethods(null)`, alors aucune méthode ne sera remplacé.
- `setConstructorArgs(array $args)` peut être appeler pour fournir un tableau de paramètres qui est passé au constructeur de la classe originale (qui n'est pas remplacé par une implémentation factice par défaut).
- `setMockClassName($name)` peut être utilisé pour spécifier un nom de classe pour la classe de la doublure de test générée.
- `disableOriginalConstructor()` peut être utilisé pour désactiver l'appel au constructeur de la classe originale.
- `disableOriginalClone()` peut être utilisé pour désactiver l'appel au constructeur de clonage de la classe originale.
- `disableAutoload()` peut être utilisé pour désactiver `__autoload()` pendant la génération de la classe de la doublure de test.

Prophecy

Prophecy [https://github.com/phpspec/prophecy] est un "framework de simulation d'objets PHP fortement arrêtée dans ses options mais tout du moins très puissant et flexible. Bien qu'il ait été initialement créé pour satisfaire les besoins de phpspec2, il est suffisamment souple pour être utilisé dans n'importe quel framework de test avec un minimum d'effort".

PHPUnit dispose d'un support intégré pour utiliser Prophecy pour créer des doublures de test. Exemple 9.17, « Tester qu'une méthode est appelée une fois et avec un paramètre indiqué » montre comment le même test montré dans Exemple 9.11, « Tester qu'une méthode est appelée une fois et avec un paramètre indiqué » peut être exprimé en utilisant la philosophie de Prophecy de prophéties et de révélations:

Exemple 9.17. Tester qu'une méthode est appelée une fois et avec un paramètre indiqué

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

class SubjectTest extends TestCase
{
    public function testObserversAreUpdated()
    {
        $subject = new Subject('My subject');

        // Crée une prophecy pour la classe Observer.
        $observer = $this->prophesize(Observer::class);

        // Configure l'attente pour que la méthode update()
        // soit appelée une seule fois avec la chaine 'something'
        // en paramètre.
        $observer->update('something')->shouldBeCalled();

        // Révèle la prophecie et attache l'objet mock
        // à $subjet
```

```
        $subject->attach($observer->reveal());

        // Appelle la méthode doSomething() sur l'objet $subject
        // dont on s'attend a ce qu'il appelle la méthode update()l'objet mocké Observer
        // avec la chaine 'something'.
        $subject->doSomething();
    }
}
?>
```

Reportez-vous a la documentation [https://github.com/phpspec/prophecy#how-to-use-it] de Prophecy pour pour plus de détails sur la création, la configuration et l'utilisation de stubs, espions, et mocks en utilisant ce framework alternatif de doublure de test.

Mocker les Traits et les classes abstraites

La méthode `getMockForTrait()` renvoie un objet mock qui utilise un Trait spécifié. Toutes les méthodes abstraites du Trait donné sont mockées. Cela permet de tester les méthodes concrètes d'un Trait.

Exemple 9.18. Tester les méthodes concrètes d'un trait

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

trait AbstractTrait
{
    public function concreteMethod()
    {
        return $this->abstractMethod();
    }

    public abstract function abstractMethod();
}

class TraitClassTest extends TestCase
{
    public function testConcreteMethod()
    {
        $mock = $this->getMockForTrait(AbstractTrait::class);

        $mock->expects($this->any())
             ->method('abstractMethod')
             ->will($this->returnValue(true));

        $this->assertTrue($mock->concreteMethod());
    }
}
?>
```

La méthode `getMockForAbstractClass()` retourne un mock pour une classe abstraite. Toutes les méthodes abstraites d'une classe mock donnée sont simulées. Ceci permet de tester les méthodes concrètes d'une classe abstraite.

Exemple 9.19. Tester les méthodes concrêtes d'une classe abstraite

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

abstract class AbstractClass
{
```

```
    public function concreteMethod()
    {
        return $this->abstractMethod();
    }

    public abstract function abstractMethod();
}

class AbstractClassTest extends TestCase
{
    public function testConcreteMethod()
    {
        $stub = $this->getMockForAbstractClass(AbstractClass::class);

        $stub->expects($this->any())
             ->method('abstractMethod')
             ->will($this->returnValue(true));

        $this->assertTrue($stub->concreteMethod());
    }
}
?>
```

Bouchon et simulacre pour Web Services

Quand votre application interagit avec un web service, vous voulez le tester sans vraiment interagir avec le web service. Pour rendre facile la création de bouchon ou de simulacre de web services, `getMockFromWsdl()` peut être utilisée de la même façon que `getMock()` (voir plus haut). La seule différence est que `getMockFromWsdl()` retourne un stub ou un basé basé sur la description en WSDL d'un web service tandis que `getMock()` retourne un bouchon ou un simulacre basé sur une classe ou une interface PHP.

Exemple 9.20, « Bouchonner un web service » montre comment `getMockFromWsdl()` peut être utilisé pour faire un bouchon, par exemple, d'un web service décrit dans `GoogleSearch.wsdl`.

Exemple 9.20. Bouchonner un web service

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

class GoogleTest extends TestCase
{
    public function testSearch()
    {
        $googleSearch = $this->getMockFromWsdl(
          'GoogleSearch.wsdl', 'GoogleSearch'
        );

        $directoryCategory = new stdClass;
        $directoryCategory->fullViewableName = '';
        $directoryCategory->specialEncoding = '';

        $element = new stdClass;
        $element->summary = '';
        $element->URL = 'https://phpunit.de/';
        $element->snippet = '...';
        $element->title = '<b>PHPUnit</b>';
        $element->cachedSize = '11k';
        $element->relatedInformationPresent = true;
        $element->hostName = 'phpunit.de';
        $element->directoryCategory = $directoryCategory;
```

```
        $element->directoryTitle = '';

        $result = new stdClass;
        $result->documentFiltering = false;
        $result->searchComments = '';
        $result->estimatedTotalResultsCount = 3.9000;
        $result->estimateIsExact = false;
        $result->resultElements = [$element];
        $result->searchQuery = 'PHPUnit';
        $result->startIndex = 1;
        $result->endIndex = 1;
        $result->searchTips = '';
        $result->directoryCategories = [];
        $result->searchTime = 0.248822;

        $googleSearch->expects($this->any())
                     ->method('doGoogleSearch')
                     ->will($this->returnValue($result));

        /**
         * $googleSearch->doGoogleSearch() will now return a stubbed result and
         * the web service's doGoogleSearch() method will not be invoked.
         */
        $this->assertEquals(
          $result,
          $googleSearch->doGoogleSearch(
            '00000000000000000000000000000000',
            'PHPUnit',
            0,
            1,
            false,
            '',
            false,
            '',
            '',
            ''
          )
        );
    }
}
?>
```

Simuler le système de fichiers

vfsStream [https://github.com/mikey179/vfsStream] est un encapsuleur de flux [http://www.php.net/streams] pour un système de fichiers virtuel [http://en.wikipedia.org/wiki/Virtual_file_system] qui peut s'avérer utile dans des tests unitaires pour simuler le vrai système de fichiers.

Ajoutez simplement une dépendance a `mikey179/vfsStream` dans le fichier `composer.json` de votre projet si vous utilisez Composer [https://getcomposer.org/] pour gérer les dépendances de votre project. Vous trouverez ci-dessous un exemple minimal de fichier `composer.json` qui définie en dépendence de développement PHPUnit 4.6 et vfsStream:

```
{
    "require-dev": {
        "phpunit/phpunit": "~4.6",
        "mikey179/vfsStream": "~1"
    }
}
```

Exemple 9.21, « Une classe qui interagit avec le système de fichiers » montre une classe qui interagit avec le système de fichiers.

Exemple 9.21. Une classe qui interagit avec le système de fichiers

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

class Example
{
    protected $id;
    protected $directory;

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

    public function setDirectory($directory)
    {
        $this->directory = $directory . DIRECTORY_SEPARATOR . $this->id;

        if (!file_exists($this->directory)) {
            mkdir($this->directory, 0700, true);
        }
    }
}?>
```

Sans un système de fichiers virtuel tel que vfsStream, nous ne pouvons pas tester la méthode `setDirectory()` en isolation des influences extérieures (voir Exemple 9.22, « Tester une classe qui interagit avec le système de fichiers »).

Exemple 9.22. Tester une classe qui interagit avec le système de fichiers

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

class ExampleTest extends TestCase
{
    protected function setUp()
    {
        if (file_exists(dirname(__FILE__) . '/id')) {
            rmdir(dirname(__FILE__) . '/id');
        }
    }

    public function testDirectoryIsCreated()
    {
        $example = new Example('id');
        $this->assertFalse(file_exists(dirname(__FILE__) . '/id'));

        $example->setDirectory(dirname(__FILE__));
        $this->assertTrue(file_exists(dirname(__FILE__) . '/id'));
    }

    protected function tearDown()
    {
        if (file_exists(dirname(__FILE__) . '/id')) {
            rmdir(dirname(__FILE__) . '/id');
        }
    }
}
?>
```

L'approche précédente possède plusieurs inconvénients :

- Comme avec les ressources externes, il peut y a voir des problèmes intermittents avec le système de fichiers. Ceci rend les tests qui interagissent avec lui peu fiables.
- Dans les méthodes `setUp()` et `tearDown()`, nous avons à nous assurer que le répertoire n'existe pas avant et après le test.
- Si l'exécution du test s'achève avant que la méthode `tearDown()` n'ait été appelée, le répertoire va rester dans le système de fichiers.

Exemple 9.23, « Simuler le système de fichiers dans un test pour une classe qui interagit avec le système de fichiers » montre comment vfsStream peut être utilisé pour simuler le système de fichiers dans un test pour une classe qui interagit avec le système de fichiers.

Exemple 9.23. Simuler le système de fichiers dans un test pour une classe qui interagit avec le système de fichiers

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

class ExampleTest extends TestCase
{
    public function setUp()
    {
        vfsStreamWrapper::register();
        vfsStreamWrapper::setRoot(new vfsStreamDirectory('exampleDir'));
    }

    public function testDirectoryIsCreated()
    {
        $example = new Example('id');
        $this->assertFalse(vfsStreamWrapper::getRoot()->hasChild('id'));

        $example->setDirectory(vfsStream::url('exampleDir'));
        $this->assertTrue(vfsStreamWrapper::getRoot()->hasChild('id'));
    }
}
?>
```

Ceci présente plusieurs avantages :

- Le test lui-même est plus concis.
- vfsStream donne au développeur du test le plein contrôle sur la façon dont le code testé voit l'environnement du système de fichiers.
- Puisque les opérations du système de fichiers n'opèrent plus sur le système de fichiers réel, les opérations de nettoyage dans la méthode `tearDown()` ne sont plus nécessaires.

Chapitre 10. Pratiques de test

> Vous pouvez toujours écrire davantage de tests. Cependant, vous vous rendrez rapidement compte que seule une fraction des tests auxquels vous pensez est réellement utile. Ce que vous voulez, c'est écrire des tests qui échouent même si vous pensez qu'ils devraient fonctionner, ou des tests qui réussissent même si vous pensez qu'ils devraient échouer. Une autre façon d'envisager cela consiste à le faire en terme de coûts/bénéfices. Vous voulez écrire des tests qui vous paient en retour avec des informations.
>
> —Erich Gamma

Pendant le développement

Lorsque vous avez besoin de modifier la structure interne d'un logiciel sur lequel vous êtes en train de travailler pour la rendre plus facile à comprendre et moins chère à modifier sans modifier son comportement visible, une suite de tests est inestimable pour appliquer de façon sûre ces refactorings [http://martinfowler.com/bliki/DefinitionOfRefactoring.html] comme on les appelle. Sans cela, vous ne pourrez pas vous détecter que vous avez cassé le système en réalisant la restructuration.

Les conditions suivantes vous aideront à améliorer le code et la conception de votre projet, tant que vous utiliserez des tests unitaires pour vérifier que les étapes de transformation dues au refactoring préservent bien le comportement et n'introduisent pas d'erreurs :

1. Tous les tests unitaires fonctionnent correctement.
2. Le code transmet des principes de conception.
3. Le code ne contient pas de redondances.
4. Le code contient le nombre minimal de classes et de méthodes.

Quand vous avez besoin d'ajouter de nouvelles fonctionnalités au système, écrivez d'abord les tests. Puis, vous aurez terminé quand les tests s'exécuteront. Cette pratique sera discutée en détail dans le prochain chapitre.

Pendant le débogage

Quand vous recevez un rapport de bug, vous pourriez être tenté de le corriger aussi vite que possible. L'expérience prouve que cette impulsion ne vous rendra pas service ; il est probable que ce défaut en provoque un autre.

Vous pouvez maîtriser cette impulsion en suivant ce qui suit :

1. Vérifiez que vous pouvez reproduire le défaut.
2. Trouvez la démonstration la plus ciblée possible du défaut dans le code. Par exemple, si un nombre s'affiche incorrectement dans une sortie écran, trouvez l'objet qui calcule ce nombre.
3. Ecrivez un test automatisé qui échoue maintenant mais qui réussira quand le défaut sera corrigé.
4. Corrigez le défaut.

Trouver le moyen fiable le plus ciblé pour reproduire le défaut vous offre l'opportunité de vraiment en examiner la cause. Le test que vous écrivez va améliorer les chances que lorsque vous corrigerez le défaut, vous le fassiez vraiment du fait que le nouveau test réduit la probabilité de défaire cette correction lors de futures modifications du code. Tous les tests écrits avant réduisent la probabilité de causer par inadvertance un problème différent.

Le test unitaire offre de nombreux avantages :

- Tester conforte les auteurs de code et les relecteurs dans le fait que les correctifs produisent des résultats corrects..
- Créer des cas de tests est un bon moyen de conduire les développeurs à découvrir des cas limites.
- Tester fournit un bon moyen de capturer les régressions rapidement et de s'assurer qu'elles ne seront pas répétées deux fois.
- Les tests unitaires fournissent des exemples fonctionnels de la façon d'utiliser une API et peuvent aider significativement à l'effort de documentation.

Globalement, le test unitaire intégré minimise les coûts et les risques de toute modification individuelle. Il permettra au projet de mener [...] des améliorations majeures d'architecture [...] rapidement et avec confiance.

—Benjamin Smedberg

Chapitre 11. Analyse de couverture de code

> En informatique, la couverture de code est une mesure utilisée pour décrire le taux de code source testé d'un programme lorsqu'il est testé par une suite de test particulière. Un programme avec un taux de couverture de code élevée a été plus complètement testé et a une plus faible chance de contenir des bugs logiciel qu'un programme avec un taux de couverture de code faible.
>
> —Wikipedia

Dans ce chapitre, vous apprendrez tout sur la fonctionnalité de couverture de code de PHPUnit qui fournit une vision interne des parties du code de production qui sont exécutées quand les tests sont exécutés. Elle utilise le composant PHP_CodeCoverage [https://github.com/sebastianbergmann/php-code-coverage] qui tire parti de la fonctionnalité de couverture de code fournie par l'extension Xdebug [http://xdebug.org/] de PHP.

Note

Xdebug n'est pas distribué au sein de PHPUnit. Si une notice indiquant que l'extension Xdebug n'est pas chargé en lançant les tests, cela signifie que Xdebug n'est pas installé ou n'est pas configuré correctement. Avant de pouvoir utiliser les fonctionnalités de couverture de code dans PHPUnit, vous devez lire le guide d'installation de Xdebug. [http://xdebug.org/docs/install].

PHPUnit peut générer un rapport de couverture de code HTML aussi bien que des fichiers de log en XML avec les informations de couverture de code dans différents formats (Clover, Crap4J, PHPUnit). Les informations de couverture de code peuvent aussi être rapporté en text (et affiché vers STDOUT) et exporté en PHP pour un traitement ultérieur.

Reportez-vous à Chapitre 3, *Le lanceur de tests en ligne de commandes* pour obtenir la liste des options de ligne de commande qui contrôlent la fonctionnalité de couverture de code ainsi que la section intitulée « Journalisation » pour les paramètres de configuration appropriés.

Indicateurs logiciels pour la couverture de code

Différents indicateurs logiciels existent pour mesurer la couverture de code:

Couverture de ligne
: L'indicateur logiciel de *couverture de ligne* mesure si chaque ligne exécutable a été exécutée.

Couverture de fonction de méthode
: L'indicateur logiciel de *couverture de fonction de méthode* mesure si chaque fonction ou méthode à été invoquée. PHP_CodeCoverage considère qu'une fonction ou une méthode a été couverte seulement quand toutes ses lignes executables sont couvertes.

Couverture de classe et de trait
: L'indicateur logiciel de *couverture de classe et de trait* mesure si chaque méthode d'une classe ou d'un trait est couverte. PHP_CodeCoverage considère qu'une classe ou un trait est couvert seulement quand toutes ses méthodes sont couvertes.

Couverture d'opcode
: L'indicateur logiciel de *couverture d'opcode* mesure si chaque opcode d'une fonction ou d'une méthode a été executé lors de

	l'execution de la suite de test. Une ligne de code se compile habituellement en plus d'un opcode. La couverture de ligne considère une ligne de code comme couverte dès que l'un de ses opcode est executé.
Couverture de branche	L'indicateur logiciel de *couverture de branche* mesure si l'expression booléenne de chaque structure de contrôle a été évalué à `true` et à `false` pendant l'execution de la suite de test.
Couverture de chemin	L'indicateur logiciel de *couverture de chemin* mesure si chacun des chemins d'execution possible dans une fonction ou une méthode ont été suivit lors de l'execution de la suite de test. Un chemin d'execution est une séquence unique de branches depuis l'entrée de la fonction ou de la méthode jusqu'à sa sortie.
L'index Change Risk Anti-Patterns (CRAP)	L'index *Change Risk Anti-Patterns (CRAP)* est calculé en se basant sur la complexité cyclomatique et la couverture de code d'une portion de code. Du code qui n'est pas trop complexe et qui a une couverture de code adéquate aurat un index CRAP faible. L'index CRAP peut être baissé en écrivant des tests et en refactorisant le code pour diminuer sa complexité.

Note

L'indicateur logiciel de *Couverture d'Opcode*, *Couverture de branche*, et de *Couverture de chemin* ne sont pas encore supportés par PHP_CodeCoverage.

Liste blanche de fichiers

Il est requis de configurer une *liste blanche* pour dire à PHPUnit quels fichiers de code source inclure dans le raport de couverture de code. Cela peut être fait en utilisant l'option de ligne de commande `--whitelist` ou via le fichier de configuration (voir la section intitulée « Inclure des fichiers de la couverture de code »).

Optionnellement, tous les fichiers en liste blanche peut être ajouté au rapport de couverture de code en paramétrant `addUncoveredFilesFromWhitelist="true"` dans votre fichier de configuration PHPUnit (voir la section intitulée « Inclure des fichiers de la couverture de code »). Cela autorise l'inclusion de fichiers quine sont pas encore testé du tout. Si vous voulez avoir des information sur quelles lignes d'un fichier non couvert sont exécutable, par exemple, vous devez également définir `processUncoveredFilesFromWhitelist="true"` dans votre fichier de configuration PHPUnit (voir la section intitulée « Inclure des fichiers de la couverture de code »).

Note

Notez que le chargement des fichiers de code source qui est effectué lorsque `processUncoveredFilesFromWhitelist="true"` est défini peut causer des problèmes quand un fichier de code source contient du code en dehors de la portée d'une classe ou d'une fonction, par exemple.

Ignorer des blocs de code

Parfois, vous avez des blocs de code que vous ne pouvez pas tester et que vous pouvez vouloir ignorer lors de l'analyse de la couverture de code. PHPUnit vous permet de le faire en utilisant les annotations `@codeCoverageIgnore`, `@codeCoverageIgnoreStart` et `@codeCoverageIgnoreEnd` comme montré dans Exemple 11.1, « Utiliser les annotations `@codeCoverageIgnore`, `@codeCoverageIgnoreStart` et `@codeCoverageIgnoreEnd` ».

Exemple 11.1. Utiliser les annotations `@codeCoverageIgnore`, `@codeCoverageIgnoreStart` et `@codeCoverageIgnoreEnd`

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

/**
 * @codeCoverageIgnore
 */
class Foo
{
    public function bar()
    {
    }
}

class Bar
{
    /**
     * @codeCoverageIgnore
     */
    public function foo()
    {
    }
}

if (false) {
    // @codeCoverageIgnoreStart
    print '*';
    // @codeCoverageIgnoreEnd
}

exit; // @codeCoverageIgnore
?>
```

Les lignes de code ignorées (marquées comme ignorées à l'aide des annotations) sont comptées comme exécutées (si elles sont exécutables) et ne seront pas mises en évidence.

Spécifier les méthodes couvertes

L'annotation `@covers` (voir Tableau B.1, « Annotations pour indiquer quelles méthodes sont couvertes par un test ») peut être utilisée dans le code de test pour indiquer quelle(s) méthode(s) une méthode de test veut tester. Si elle est fournie, seules les informations de couverture de code pour la(les) méthode(s) indiquées seront prises en considération. Exemple 11.2, « Tests qui indiquent quelle(s) méthode(s) ils veulent couvrir » montre un exemple.

Exemple 11.2. Tests qui indiquent quelle(s) méthode(s) ils veulent couvrir

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

class BankAccountTest extends TestCase
{
    protected $ba;

    protected function setUp()
    {
        $this->ba = new BankAccount;
    }
```

```
    /**
     * @covers BankAccount::getBalance
     */
    public function testBalanceIsInitiallyZero()
    {
        $this->assertEquals(0, $this->ba->getBalance());
    }

    /**
     * @covers BankAccount::withdrawMoney
     */
    public function testBalanceCannotBecomeNegative()
    {
        try {
            $this->ba->withdrawMoney(1);
        }

        catch (BankAccountException $e) {
            $this->assertEquals(0, $this->ba->getBalance());

            return;
        }

        $this->fail();
    }

    /**
     * @covers BankAccount::depositMoney
     */
    public function testBalanceCannotBecomeNegative2()
    {
        try {
            $this->ba->depositMoney(-1);
        }

        catch (BankAccountException $e) {
            $this->assertEquals(0, $this->ba->getBalance());

            return;
        }

        $this->fail();
    }

    /**
     * @covers BankAccount::getBalance
     * @covers BankAccount::depositMoney
     * @covers BankAccount::withdrawMoney
     */
    public function testDepositWithdrawMoney()
    {
        $this->assertEquals(0, $this->ba->getBalance());
        $this->ba->depositMoney(1);
        $this->assertEquals(1, $this->ba->getBalance());
        $this->ba->withdrawMoney(1);
        $this->assertEquals(0, $this->ba->getBalance());
    }
}
?>
```

Il est également possible d'indiquer qu'un test ne doit couvrir *aucune* méthode en utilisant l'annotation `@coversNothing` (voir la section intitulée « @coversNothing »). Ceci peut être utile quand on écrit des tests d'intégration pour s'assurer que vous ne générez une couverture de code avec des tests unitaires.

Exemple 11.3. Un test qui indique qu'aucune méthode ne doit être couverte

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

class GuestbookIntegrationTest extends PHPUnit_Extensions_Database_TestCase
{
    /**
     * @coversNothing
     */
    public function testAddEntry()
    {
        $guestbook = new Guestbook();
        $guestbook->addEntry("suzy", "Hello world!");

        $queryTable = $this->getConnection()->createQueryTable(
            'guestbook', 'SELECT * FROM guestbook'
        );

        $expectedTable = $this->createFlatXmlDataSet("expectedBook.xml")
                              ->getTable("guestbook");

        $this->assertTablesEqual($expectedTable, $queryTable);
    }
}
?>
```

Cas limites

Cette section présente des cas limites remarquables qui conduisent à des informations de couverture de code prêtant à confusion.

Exemple 11.4.

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

// Because it is "line based" and not statement base coverage
// one line will always have one coverage status
if (false) this_function_call_shows_up_as_covered();

// Due to how code coverage works internally these two lines are special.
// This line will show up as non executable
if (false)
    // This line will show up as covered because it is actually the
    // coverage of the if statement in the line above that gets shown here!
    will_also_show_up_as_covered();

// To avoid this it is necessary that braces are used
if (false) {
    this_call_will_never_show_up_as_covered();
}
?>
```

Chapitre 12. Autres utilisations des tests

Une fois que vous aurez l'habitude d'écrire des tests automatisés, vous découvrirez certainement davantage d'usages pour les tests. En voici quelques exemples.

Documentation agile

Typiquement, dans un projet développé en utilisant un processus agile, tel que l'Extreme Programming, la documentation ne peut pas suivre les changements fréquents de la conception et du code du projet. l'Extreme Programming réclame la *propriété collective du code*, donc tous les développeurs ont besoin de savoir comment fonctionne l'intégralité du système. Si vous êtes suffisamment discipliné pour utiliser pour vos tests des "noms parlant" qui décrivent ce qu'une classe doit faire, vous pouvez utiliser la fonctionnalité TestDox de PHPUnit pour générer automatiquement de la documentation pour votre projet en s'appuyant sur ses tests. Cette documentation donne aux développeurs un aperçu de ce que chaque classe du projet est supposée faire.

La fonctionnalité TestDox de PHPUnit examine une classe de test et tous les noms de méthode de test pour les convertir les noms PHP au format Camel Case en phrases : `testBalanceIsInitiallyZero()` devient "Balance is initially zero". S'il existe plusieurs méthodes de test dont les noms ne diffèrent que par un suffixe constitué de un ou plusieurs chiffres, telles que `testBalanceCannotBecomeNegative()` et `testBalanceCannotBecomeNegative2()`, la phrase "Balance ne peut pas etre négative" n'apparaîtra qu'une seule fois, en supposant que tous ces tests ont réussi.

Jetons un oeil sur la documentation agile générée pour la classe `BankAccount`

```
phpunit --testdox BankAccountTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

BankAccount
 [x] Balance is initially zero
 [x] Balance cannot become negative
```

Alternativement, la documentation agile peut être générée en HTML ou au format texte et écrite dans un fichier en utilisant les paramètres `--testdox-html` et `--testdox-text`.

La documentation agile peut être utilisée pour documenter les hypothèses que vous faites sur les paquets externes que vous utilisez dans votre projet. Quand vous utilisez un paquet externe, vous vous exposez au risque que le paquet ne se comportera pas comme vous le prévoyez et que les futures versions du paquet changeront de façon subtile, ce qui cassera votre code sans que vous ne le sachiez. Vous pouvez adresser ces risques en écrivant un test à chaque fois que vous faites une hypothèse. Si votre test réussit, votre hypothèse est valide. Si vous documentez toutes vos hypothèses avec des tests, les futures livraisons du paquet externe ne poseront pas de problème : si les tests réussissent, votre système doit continuer à fonctionner.

Tests inter-équipes

Quand vous documentez des hypothèses avec des tests, vous êtes propriétaire des tests. Le fournisseur du paquet -- sur lequel vous faîtes des hypothèses -- ne connaît rien de vos tests. Si vous voulez avoir une relation plus étroite avec le fournisseur du paquet, vous pouvez utiliser les tests pour communiquer et coordonner vos activités.

Quand vous êtes d'accord pour coordonner vos activités avec le fournisseur d'un paquet, vous pouvez écrire les tests ensembles. Faites cela d'une telle façon que les tests révèlent autant d'hypothèses que possible. Les hypothèses cachées sont la mort de la coopération. Avec les tests, vous documentez

exactement ce que vous attendez du paquet fourni. Le fournisseur saura que le paquet est prêt quand tous les tests fonctionneront.

En utilisant des bouchons (voir le chapitre relatif aux "objets simulacres", précédemment dans ce livre), vous pouvez créer un découplage plus grand entre vous et le fournisseur: le travail du fournisseur est de faire fonctionner les tests avec l'implémentation réelle du paquet. Votre travail est de faire que les tests fonctionnent sur votre propre code. Jusqu'à ce que vous ayez l'implémentation réelle du paquet fourni, vous utilisez des objets bouchons. Suivant cette approche, deux équipes peuvent développer indépendamment.

Chapitre 13. Journalisation

PHPUnit peut produire plusieurs types de fichiers de logs.

Résultats de test (XML)

Le fichier de log XML pour les tests produits par PHPUnit est basé sur celui qui est utilisé par la tâche JUnit de Apache Ant [http://ant.apache.org/manual/Tasks/junit.html]. L'exemple suivant montre que le fichier de log XML généré pour les tests dans `TestTableau` :

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites>
  <testsuite name="ArrayTest"
             file="/home/sb/ArrayTest.php"
             tests="2"
             assertions="2"
             failures="0"
             errors="0"
             time="0.016030">
    <testcase name="testNewArrayIsEmpty"
              class="ArrayTest"
              file="/home/sb/ArrayTest.php"
              line="6"
              assertions="1"
              time="0.008044"/>
    <testcase name="testArrayContainsAnElement"
              class="ArrayTest"
              file="/home/sb/ArrayTest.php"
              line="15"
              assertions="1"
              time="0.007986"/>
  </testsuite>
</testsuites>
```

Le fichier de log XML suivant a été généré pour deux tests, `testFailure` et `testError`, à partir d'une classe de cas de test nommée `FailureErrorTest` et montre comment les échecs et les erreurs sont signalés.

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites>
  <testsuite name="FailureErrorTest"
             file="/home/sb/FailureErrorTest.php"
             tests="2"
             assertions="1"
             failures="1"
             errors="1"
             time="0.019744">
    <testcase name="testFailure"
              class="FailureErrorTest"
              file="/home/sb/FailureErrorTest.php"
              line="6"
              assertions="1"
              time="0.011456">
      <failure type="PHPUnit_Framework_ExpectationFailedException">
testFailure(FailureErrorTest)
Failed asserting that &lt;integer:2&gt; matches expected value &lt;integer:1&gt;.

/home/sb/FailureErrorTest.php:8
</failure>
    </testcase>
```

```
    <testcase name="testError"
              class="FailureErrorTest"
              file="/home/sb/FailureErrorTest.php"
              line="11"
              assertions="0"
              time="0.008288">
      <error type="Exception">testError(FailureErrorTest)
Exception:

/home/sb/FailureErrorTest.php:13
</error>
    </testcase>
  </testsuite>
</testsuites>
```

Couverture de code (XML)

Le format XML pour journaliser les informations de couverture de code produite par PHPUnit est faiblement basé sur celui utilisé par Clover [http://www.atlassian.com/software/clover/]. L'exemple suivant montre le fichier de log XML généré pour les tests dans `BankAccountTest`:

```
<?xml version="1.0" encoding="UTF-8"?>
<coverage generated="1184835473" phpunit="3.6.0">
  <project name="BankAccountTest" timestamp="1184835473">
    <file name="/home/sb/BankAccount.php">
      <class name="BankAccountException">
        <metrics methods="0" coveredmethods="0" statements="0"
                 coveredstatements="0" elements="0" coveredelements="0"/>
      </class>
      <class name="BankAccount">
        <metrics methods="4" coveredmethods="4" statements="13"
                 coveredstatements="5" elements="17" coveredelements="9"/>
      </class>
      <line num="77" type="method" count="3"/>
      <line num="79" type="stmt" count="3"/>
      <line num="89" type="method" count="2"/>
      <line num="91" type="stmt" count="2"/>
      <line num="92" type="stmt" count="0"/>
      <line num="93" type="stmt" count="0"/>
      <line num="94" type="stmt" count="2"/>
      <line num="96" type="stmt" count="0"/>
      <line num="105" type="method" count="1"/>
      <line num="107" type="stmt" count="1"/>
      <line num="109" type="stmt" count="0"/>
      <line num="119" type="method" count="1"/>
      <line num="121" type="stmt" count="1"/>
      <line num="123" type="stmt" count="0"/>
      <metrics loc="126" ncloc="37" classes="2" methods="4" coveredmethods="4"
               statements="13" coveredstatements="5" elements="17"
               coveredelements="9"/>
    </file>
    <metrics files="1" loc="126" ncloc="37" classes="2" methods="4"
             coveredmethods="4" statements="13" coveredstatements="5"
             elements="17" coveredelements="9"/>
  </project>
</coverage>
```

Couverture de code (TEXTE)

Un affichage de la couverture de code lisible pour la ligne de commandes ou un fichier texte. Le but de ce format de sortie est de fournir un aperçu rapide de couverture en travaillant sur un petit

ensemble de classes. Pour des projets plus grand cette sortie peut être utile pour obtenir un aperçu rapide de la couverture des projets ou quand il est utilisé avec la fonctionnalité `--filter`. Quand c'est utilisé à partir de la ligne de commande en écrivant sur `php://stdout`, cela prend en compte le réglage `--colors`. Ecrire sur la sortie standard est l'option par défaut quand on utilise la ligne de commandes. Par défaut, ceci ne montrera que les fichiers qui ont au moins une ligne couverte. Ceci peut uniquement être modifié via l'option de configuration xml `showUncoveredFiles` Voir la section intitulée « Journalisation ». Par défaut, tous les fichier et leur status de couverture sont affichés dans le rapport détaillé. Ceci peut être changé via l'option de configuration xml `showOnlySummary`.

Chapitre 14. Etendre PHPUnit

PHPUnit peut être étendu de multiples façon pour rendre l'écriture des tests plus simple et personnaliser le retour que vous obtenez des tests exécutés. Voici les points de départs communs pour étendre PHPUnit.

Sous-classe PHPUnit\Framework\TestCase

Ecrivez des assertions personnalisées et des méthodes utilitaires dans une sous classe abstraite de `PHPUnit\Framework\TestCase` et faites hériter vos classes de cas de test de cette classe. C'est une des façon les plus faciles pour étendre PHPUnit.

Ecrire des assertions personnalisées

Lorsqu'on écrit des assertions personnalisées, c'est une bonne pratique de suivre la façon dont PHPUnit implémente ses propres assertions. Comme vous pouvez le voir dans Exemple 14.1, « Les méthodes assertTrue() et isTrue() de la classe PHPUnit_Framework_Assert », la méthode `assertTrue()` ne fait qu'encapsuler les méthodes `isTrue()` et `assertThat()`: `isTrue()` crée un objet matcher qui est passé à `assertThat()` pour évaluation.

Exemple 14.1. Les méthodes assertTrue() et isTrue() de la classe PHPUnit_Framework_Assert

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

abstract class PHPUnit_Framework_Assert
{
    // ...

    /**
     * Asserts that a condition is true.
     *
     * @param  boolean $condition
     * @param  string  $message
     * @throws PHPUnit_Framework_AssertionFailedError
     */
    public static function assertTrue($condition, $message = '')
    {
        self::assertThat($condition, self::isTrue(), $message);
    }

    // ...

    /**
     * Returns a PHPUnit_Framework_Constraint_IsTrue matcher object.
     *
     * @return PHPUnit_Framework_Constraint_IsTrue
     * @since  Method available since Release 3.3.0
     */
    public static function isTrue()
    {
        return new PHPUnit_Framework_Constraint_IsTrue;
    }

    // ...
}?>
```

Exemple 14.2, « La classe PHPUnit_Framework_Constraint_IsTrue » montre comment `PHPUnit_Framework_Constraint_IsTrue` étend la classe abstraite de base pour des objets matcher (ou des contraintes), `PHPUnit_Framework_Constraint`.

Exemple 14.2. La classe PHPUnit_Framework_Constraint_IsTrue

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

class PHPUnit_Framework_Constraint_IsTrue extends PHPUnit_Framework_Constraint
{
    /**
     * Evaluates the constraint for parameter $other. Returns true if the
     * constraint is met, false otherwise.
     *
     * @param mixed $other Value or object to evaluate.
     * @return bool
     */
    public function matches($other)
    {
        return $other === true;
    }

    /**
     * Returns a string representation of the constraint.
     *
     * @return string
     */
    public function toString()
    {
        return 'is true';
    }
}?>
```

L'effort d'implémentation des méthodes `assertTrue()` et `isTrue()` ainsi que la classe `PHPUnit_Framework_Constraint_IsTrue` tire bénéfice du fait que `assertThat()` prend automatiquement soin d'évaluer l'assertion et les tâches de suivi comme le décompte à des fins de statistique. Plus encore, la méthode `isTrue()` peut être utilisée comme un matcher lors de la configuration d'objets simulacres.

Implémenter PHPUnit\Framework\TestListener

Exemple 14.3, « Un simple moniteur de test » montre une implémentation simple de l'interface `PHPUnit\Framework\TestListener`.

Exemple 14.3. Un simple moniteur de test

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

class SimpleTestListener implements TestListener
{
    public function addError(PHPUnit_Framework_Test $test, Exception $e, $time)
    {
        printf("Error while running test '%s'.\n", $test->getName());
    }

    public function addFailure(PHPUnit_Framework_Test $test, PHPUnit_Framework_Assertion
```

```
    {
        printf("Test '%s' failed.\n", $test->getName());
    }

    public function addIncompleteTest(PHPUnit_Framework_Test $test, Exception $e, $time)
    {
        printf("Test '%s' is incomplete.\n", $test->getName());
    }

    public function addRiskyTest(PHPUnit_Framework_Test $test, Exception $e, $time)
    {
        printf("Test '%s' is deemed risky.\n", $test->getName());
    }

    public function addSkippedTest(PHPUnit_Framework_Test $test, Exception $e, $time)
    {
        printf("Test '%s' has been skipped.\n", $test->getName());
    }

    public function startTest(PHPUnit_Framework_Test $test)
    {
        printf("Test '%s' started.\n", $test->getName());
    }

    public function endTest(PHPUnit_Framework_Test $test, $time)
    {
        printf("Test '%s' ended.\n", $test->getName());
    }

    public function startTestSuite(PHPUnit_Framework_TestSuite $suite)
    {
        printf("TestSuite '%s' started.\n", $suite->getName());
    }

    public function endTestSuite(PHPUnit_Framework_TestSuite $suite)
    {
        printf("TestSuite '%s' ended.\n", $suite->getName());
    }
}
?>
```

Exemple 14.4, « Utiliser BaseTestListener » montre comment étendre la classe abstraite `PHPUnit_Framework_BaseTestListener`, qui permet de spécifier uniquement les méthodes d'interface qui sont intéressantes pour votre cas d'utilisation, tout en fournissant des implémentations vides p our tous les autres.

Exemple 14.4. Utiliser BaseTestListener

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

class ShortTestListener extends PHPUnit_Framework_BaseTestListener
{
    public function endTest(PHPUnit_Framework_Test $test, $time)
    {
        printf("Test '%s' ended.\n", $test->getName());
    }
}
?>
```

Dans la section intitulée « Écouteurs de tests » vous pouvez voir comment configurer PHPUnit pour brancher votre moniteur de test lors de l'exécution des tests.

Hériter de PHPUnit_Extensions_TestDecorator

Vous pouvez encapsuler des cas de test ou des séries de tests dans une sous-classe de `PHPUnit_Extensions_TestDecorator` et utiliser le Design Pattern Decorator pour réaliser certaines actions avant et après que les tests soient exécutés.

PHPUnit apporte un décorateur de test concrets : `PHPUnit_Extensions_RepeatedTest`. Il est utilisé pour exécuter de manière répétée un test et ne le comptabiliser comme succès que si toutes les itérations ont réussi.

Exemple 14.5, « Le décorateur RepeatedTest » montre une version raccourcie du décorateur de test `PHPUnit_Extensions_RepeatedTest` qui illustre comment écrire vos propres décorateurs de tests.

Exemple 14.5. Le décorateur RepeatedTest

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

require_once 'PHPUnit/Extensions/TestDecorator.php';

class PHPUnit_Extensions_RepeatedTest extends PHPUnit_Extensions_TestDecorator
{
    private $timesRepeat = 1;

    public function __construct(PHPUnit_Framework_Test $test, $timesRepeat = 1)
    {
        parent::__construct($test);

        if (is_integer($timesRepeat) &&
            $timesRepeat >= 0) {
            $this->timesRepeat = $timesRepeat;
        }
    }

    public function count()
    {
        return $this->timesRepeat * $this->test->count();
    }

    public function run(PHPUnit_Framework_TestResult $result = null)
    {
        if ($result === null) {
            $result = $this->createResult();
        }

        for ($i = 0; $i < $this->timesRepeat && !$result->shouldStop(); $i++) {
            $this->test->run($result);
        }

        return $result;
    }
}
?>
```

Implémenter PHPUnit_Framework_Test

L'interface `PHPUnit_Framework_Test` est restreinte et facile à implémenter. Vous pouvez écrire une implémentation de `PHPUnit_Framework_Test` qui est plus simple que `PHPUnit\Framework\TestCase` et qui exécute *des tests dirigés par les données*, par exemple.

Exemple 14.6, « Un test dirigé par les données » montre une classe de cas de test dirigé par les tests qui compare les valeurs d'un fichier contenant des valeurs séparées par des virgules (CSV). Chaque ligne d'un tel fichier ressemble à `foo;bar`, où la première valeur est celle que nous attendons et la seconde valeur celle constatée.

Exemple 14.6. Un test dirigé par les données

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

class DataDrivenTest implements PHPUnit_Framework_Test
{
    private $lines;

    public function __construct($dataFile)
    {
        $this->lines = file($dataFile);
    }

    public function count()
    {
        return 1;
    }

    public function run(PHPUnit_Framework_TestResult $result = null)
    {
        if ($result === null) {
            $result = new PHPUnit_Framework_TestResult;
        }

        foreach ($this->lines as $line) {
            $result->startTest($this);
            PHP_Timer::start();
            $stopTime = null;

            list($expected, $actual) = explode(';', $line);

            try {
                PHPUnit_Framework_Assert::assertEquals(
                  trim($expected), trim($actual)
                );
            }

            catch (PHPUnit_Framework_AssertionFailedError $e) {
                $stopTime = PHP_Timer::stop();
                $result->addFailure($this, $e, $stopTime);
            }

            catch (Exception $e) {
                $stopTime = PHP_Timer::stop();
                $result->addError($this, $e, $stopTime);
            }

            if ($stopTime === null) {
                $stopTime = PHP_Timer::stop();
            }
```

```
            $result->endTest($this, $stopTime);
        }

        return $result;
    }
}

$test = new DataDrivenTest('data_file.csv');
$result = PHPUnit_TextUI_TestRunner::run($test);
?>
```

```
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

.F

Time: 0 seconds

There was 1 failure:

1) DataDrivenTest
Failed asserting that two strings are equal.
expected string <bar>
difference      <  x>
got string      <baz>
/home/sb/DataDrivenTest.php:32
/home/sb/DataDrivenTest.php:53

FAILURES!
Tests: 2, Failures: 1.
```

Annexe A. Assertions

Cette annexe liste les différentes méthodes d'assertion disponibles.

De l'utilisation Statique vs. Non-Statique des méthodes d'assertion

Les assertions de PHPUnitsont implémentées dans `PHPUnit\Framework\Assert`. `PHPUnit\Framework\TestCase` hérite de `PHPUnit\Framework\Assert`.

Les méthodes d'assertion sont declarées `static` et peuvent être appelées depuis n'importe quel contexte en utilisant `PHPUnit\Framework\Assert::assertTrue()`, par exemple, ou en utilisant `$this->assertTrue()` ou `self::assertTrue()`, par exemple, dans une classe qui étends `PHPUnit\Framework\TestCase`.

En fait, vous pouvez même utiliser des fonctions globales d'encapsulation comme `assertTrue()` dans n'importe quel contexte (y compris les classes qui étendent `PHPUnit\Framework\TestCase`) quand vous incluez (manuellement) le fichier de code source `src/Framework/Assert/Functions.php` fournit avec PHPUnit.

Une question fréquente, surtout de la part des développeurs qui ne connaissent pas PHPUnit, est si utiliser `$this->assertTrue()` ou `self::assertTrue()`, par exemple, est "la bonne façon" d'appeler une assertion. La réponse courte est: il n'y a pas de bonne façon. Et il n'y a pas de mauvaise façon non plus. C'est une question de préférence personnelle.

Pour la plupart des gens, il "semble juste" d'utiliser `$this->assertTrue()` parce que la méthode de test est appelée sur un objet de test. Le fait que les méthodes d'assertion soient déclarées `static` autorise leur (ré)utilisation en dehors du scope d'un objet de test. Enfin, les fonctions globales d'encapsulation permettent aux développeurs de taper moins de caractères (`assertTrue()` au lieu de `$this->assertTrue()` ou `self::assertTrue()`).

assertArrayHasKey()

```
assertArrayHasKey(mixed $key, array $array[, string $message = ''])
```

Signale une erreur identifiée par `$message` si le tableau `$array` ne dispose pas de la clé `$key`.

`assertArrayNotHasKey()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.1. Utilisation de assertArrayHasKey()

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

class ArrayHasKeyTest extends TestCase
{
    public function testFailure()
    {
        $this->assertArrayHasKey('foo', ['bar' => 'baz']);
    }
}
?>
```

```
phpunit ArrayHasKeyTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.
```

```
F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ArrayHasKeyTest::testFailure
Failed asserting that an array has the key 'foo'.

/home/sb/ArrayHasKeyTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertClassHasAttribute()

`assertClassHasAttribute(string $attributeName, string $className[, string $message = ''])`

Signale une erreur identifiée par `$message` si `$className::attributeName` n'existe pas.

`assertClassNotHasAttribute()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.2. Utilisation de assertClassHasAttribute()

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

class ClassHasAttributeTest extends TestCase
{
    public function testFailure()
    {
        $this->assertClassHasAttribute('foo', stdClass::class);
    }
}
?>
```

```
phpunit ClassHasAttributeTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ClassHasAttributeTest::testFailure
Failed asserting that class "stdClass" has attribute "foo".

/home/sb/ClassHasAttributeTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertArraySubset()

`assertArraySubset(array $subset, array $array[, bool $strict = '', string $message = ''])`

Signale une erreur identifiée par `$message` si `$array` ne contient pas le `$subset`.

`$strict` indique de comparer l'identité des objets dans les tableaux.

Exemple A.3. Utilisation de assertArraySubset()

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

class ArraySubsetTest extends TestCase
{
    public function testFailure()
    {
        $this->assertArraySubset(['config' => ['key-a', 'key-b']], ['config' => ['key-a'
    }
}
?>
```

```
phpunit ArrayHasKeyTest
PHPUnit 4.4.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) Epilog\EpilogTest::testNoFollowOption
Failed asserting that an array has the subset Array &0 (
    'config' => Array &1 (
        0 => 'key-a'
        1 => 'key-b'
    )
).

/home/sb/ArraySubsetTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertClassHasStaticAttribute()

`assertClassHasStaticAttribute(string $attributeName, string $class-Name[, string $message = ''])`

Signale une erreur identifiée par `$message` si `$className::attributeName` n'existe pas.

`assertClassNotHasStaticAttribute()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.4. Utilisation de assertClassHasStaticAttribute()

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

class ClassHasStaticAttributeTest extends TestCase
{
    public function testFailure()
    {
        $this->assertClassHasStaticAttribute('foo', stdClass::class);
    }
```

```
}
?>
```

```
phpunit ClassHasStaticAttributeTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ClassHasStaticAttributeTest::testFailure
Failed asserting that class "stdClass" has static attribute "foo".

/home/sb/ClassHasStaticAttributeTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertContains()

`assertContains(mixed $needle, Iterator|array $haystack[, string $message = ''])`

Signale une erreur identifiée par `$message` si `$needle` n'est pas un élément de `$haystack`.

`assertNotContains()` est l'inverse de cette assertion et prend les mêmes arguments.

`assertAttributeContains()` et `assertAttributeNotContains()` sont des encapsulateurs de commodités qui utilisent un attribut `public`, `protected`, ou `private` d'une classe ou d'un objet en tant que haystack.

Exemple A.5. Utilisation de assertContains()

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

class ContainsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertContains(4, [1, 2, 3]);
    }
}
?>
```

```
phpunit ContainsTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ContainsTest::testFailure
Failed asserting that an array contains 4.

/home/sb/ContainsTest.php:6
```

```
FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

`assertContains(string $needle, string $haystack[, string $message = '', boolean $ignoreCase = false])`

Signale une erreur identifiée par `$message` si `$needle` n'est pas une sous-chaine de `$haystack`.

Si `$ignoreCase` est `true`, ce test sera sensible à la casse.

Exemple A.6. Utilisation de assertContains()

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

class ContainsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertContains('baz', 'foobar');
    }
}
?>
```

```
phpunit ContainsTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ContainsTest::testFailure
Failed asserting that 'foobar' contains "baz".

/home/sb/ContainsTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Exemple A.7. Utilisation de assertContains() with $ignoreCase

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

class ContainsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertContains('foo', 'FooBar');
    }

    public function testOK()
    {
        $this->assertContains('foo', 'FooBar', '', true);
    }
}
?>
```

```
phpunit ContainsTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.
```

```
F.

Time: 0 seconds, Memory: 2.75Mb

There was 1 failure:

1) ContainsTest::testFailure
Failed asserting that 'FooBar' contains "foo".

/home/sb/ContainsTest.php:6

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.
```

assertContainsOnly()

`assertContainsOnly(string $type, Iterator|array $haystack[, boolean $isNativeType = null, string $message = ''])`

Signale une erreur identifiée par `$message` si `$haystack` ne contient pas seulement des valeurs du type `$type`.

`$isNativeType` indique si `$type` est un type PHP natif ou non.

`assertNotContainsOnly()` est l'inverse de cette assertion et prend les mêmes arguments.

`assertAttributeContainsOnly()` et `assertAttributeNotContainsOnly()` sont des encapsulateurs de commodités qui utilisent l'attribut `public`, `protected`, ou `private` d'une classe ou d'un objet en tant que haystack.

Exemple A.8. Utilisation de assertContainsOnly()

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

class ContainsOnlyTest extends TestCase
{
    public function testFailure()
    {
        $this->assertContainsOnly('string', ['1', '2', 3]);
    }
}
?>
```

```
phpunit ContainsOnlyTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ContainsOnlyTest::testFailure
Failed asserting that Array (
    0 => '1'
    1 => '2'
    2 => 3
) contains only values of type "string".

/home/sb/ContainsOnlyTest.php:6
```

```
FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertContainsOnlyInstancesOf()

`assertContainsOnlyInstancesOf(string $classname, Traversable|array $haystack[, string $message = ''])`

Signale une erreur identifiée par `$message` si `$haystack` ne contient pas seulement des intance de la classe `$classname`.

Exemple A.9. Utilisation de assertContainsOnlyInstancesOf()

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

class ContainsOnlyInstancesOfTest extends TestCase
{
    public function testFailure()
    {
        $this->assertContainsOnlyInstancesOf(
            Foo::class,
            [new Foo, new Bar, new Foo]
        );
    }
}
?>
```

```
phpunit ContainsOnlyInstancesOfTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ContainsOnlyInstancesOfTest::testFailure
Failed asserting that Array ([0]=> Bar Object(...)) is an instance of class "Foo".

/home/sb/ContainsOnlyInstancesOfTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertCount()

`assertCount($expectedCount, $haystack[, string $message = ''])`

Signale une erreur identifiée par `$message` si le nombre d'éléments dans `$haystack` n'est pas `$expectedCount`.

`assertNotCount()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.10. Utilisation de assertCount()

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

```
class CountTest extends TestCase
{
    public function testFailure()
    {
        $this->assertCount(0, ['foo']);
    }
}
?>
```

```
phpunit CountTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) CountTest::testFailure
Failed asserting that actual size 1 matches expected size 0.

/home/sb/CountTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertDirectoryExists()

`assertDirectoryExists(string $directory[, string $message = ''])`

Signale une erreur identifiée par `$message` si le répertoire spécifié par `$directory` n'existe pas.

`assertDirectoryNotExists()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.11. Utilisation de assertDirectoryExists()

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

class DirectoryExistsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertDirectoryExists('/path/to/directory');
    }
}
?>
```

```
phpunit DirectoryExistsTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) DirectoryExistsTest::testFailure
Failed asserting that directory "/path/to/directory" exists.

/home/sb/DirectoryExistsTest.php:6
```

```
FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertDirectoryIsReadable()

`assertDirectoryIsReadable(string $directory[, string $message = ''])`

Signale une erreur identifiée par `$message` si le répertoire spécifié par `$directory` n'est pas un répertoire ou n'est pas accessible en lecture.

`assertDirectoryNotIsReadable()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.12. Utilisation de assertDirectoryIsReadable()

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

class DirectoryIsReadableTest extends TestCase
{
    public function testFailure()
    {
        $this->assertDirectoryIsReadable('/path/to/directory');
    }
}
?>
```

```
phpunit DirectoryIsReadableTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) DirectoryIsReadableTest::testFailure
Failed asserting that "/path/to/directory" is readable.

/home/sb/DirectoryIsReadableTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertDirectoryIsWritable()

`assertDirectoryIsWritable(string $directory[, string $message = ''])`

Signale une erreur identifiée par `$message` si le répertoire spécifié par `$directory` n'est pas un répertoire accessible en écriture.

`assertDirectoryNotIsWritable()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.13. Utilisation de assertDirectoryIsWritable()

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

```
class DirectoryIsWritableTest extends TestCase
{
    public function testFailure()
    {
        $this->assertDirectoryIsWritable('/path/to/directory');
    }
}
?>
```

```
phpunit DirectoryIsWritableTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) DirectoryIsWritableTest::testFailure
Failed asserting that "/path/to/directory" is writable.

/home/sb/DirectoryIsWritableTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertEmpty()

`assertEmpty(mixed $actual[, string $message = ''])`

Signale une erreur identifiée par `$message` si `$actual` n'est pas vide.

`assertNotEmpty()` est l'inverse de cette assertion et prend les mêmes arguments.

`assertAttributeEmpty()` et `assertAttributeNotEmpty()` sont des encapsulateurs de commodités qui peuvent être appliqués à un attribut `public`, `protected` ou `private` d'une classe ou d'un objet.

Exemple A.14. Utilisation de assertEmpty()

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

class EmptyTest extends TestCase
{
    public function testFailure()
    {
        $this->assertEmpty(['foo']);
    }
}
?>
```

```
phpunit EmptyTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) EmptyTest::testFailure
```

```
Failed asserting that an array is empty.

/home/sb/EmptyTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertEqualXMLStructure()

`assertEqualXMLStructure(DOMElement $expectedElement, DOMElement $actualElement[, boolean $checkAttributes = false, string $message = ''])`

Signale une erreur identifiée par `$message` si la structure XML du DOMElement dans `$actualElement` n'est pas égale à la structure XML du DOMElement dans `$expectedElement`.

Exemple A.15. Utilisation de assertEqualXMLStructure()

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

class EqualXMLStructureTest extends TestCase
{
    public function testFailureWithDifferentNodeNames()
    {
        $expected = new DOMElement('foo');
        $actual = new DOMElement('bar');

        $this->assertEqualXMLStructure($expected, $actual);
    }

    public function testFailureWithDifferentNodeAttributes()
    {
        $expected = new DOMDocument;
        $expected->loadXML('<foo bar="true" />');

        $actual = new DOMDocument;
        $actual->loadXML('<foo/>');

        $this->assertEqualXMLStructure(
          $expected->firstChild, $actual->firstChild, true
        );
    }

    public function testFailureWithDifferentChildrenCount()
    {
        $expected = new DOMDocument;
        $expected->loadXML('<foo><bar/><bar/><bar/></foo>');

        $actual = new DOMDocument;
        $actual->loadXML('<foo><bar/></foo>');

        $this->assertEqualXMLStructure(
          $expected->firstChild, $actual->firstChild
        );
    }

    public function testFailureWithDifferentChildren()
    {
        $expected = new DOMDocument;
        $expected->loadXML('<foo><bar/><bar/><bar/></foo>');
```

```
        $actual = new DOMDocument;
        $actual->loadXML('<foo><baz/><baz/><baz/></foo>');

        $this->assertEqualXMLStructure(
          $expected->firstChild, $actual->firstChild
        );
    }
}
?>
```

```
phpunit EqualXMLStructureTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

FFFF

Time: 0 seconds, Memory: 5.75Mb

There were 4 failures:

1) EqualXMLStructureTest::testFailureWithDifferentNodeNames
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'foo'
+'bar'

/home/sb/EqualXMLStructureTest.php:9

2) EqualXMLStructureTest::testFailureWithDifferentNodeAttributes
Number of attributes on node "foo" does not match
Failed asserting that 0 matches expected 1.

/home/sb/EqualXMLStructureTest.php:22

3) EqualXMLStructureTest::testFailureWithDifferentChildrenCount
Number of child nodes of "foo" differs
Failed asserting that 1 matches expected 3.

/home/sb/EqualXMLStructureTest.php:35

4) EqualXMLStructureTest::testFailureWithDifferentChildren
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'bar'
+'baz'

/home/sb/EqualXMLStructureTest.php:48

FAILURES!
Tests: 4, Assertions: 8, Failures: 4.
```

assertEquals()

`assertEquals(mixed $expected, mixed $actual[, string $message = ''])`

Signale une erreur identifiée par `$message` si les deux variables `$expected` et `$actual` ne sont pas égales.

`assertNotEquals()` est l'inverse de cette assertion et prend les mêmes arguments.

`assertAttributeEquals()` et `assertAttributeNotEquals()` sont des encapsulateurs de commodités qui utilisent un attribut `public`, `protected` ou `private` d'une classe ou d'un objet en tant que valeur.

Exemple A.16. Utilisation de assertEquals()

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

class EqualsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertEquals(1, 0);
    }

    public function testFailure2()
    {
        $this->assertEquals('bar', 'baz');
    }

    public function testFailure3()
    {
        $this->assertEquals("foo\nbar\nbaz\n", "foo\nbah\nbaz\n");
    }
}
?>
```

```
phpunit EqualsTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

FFF

Time: 0 seconds, Memory: 5.25Mb

There were 3 failures:

1) EqualsTest::testFailure
Failed asserting that 0 matches expected 1.

/home/sb/EqualsTest.php:6

2) EqualsTest::testFailure2
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'bar'
+'baz'

/home/sb/EqualsTest.php:11

3) EqualsTest::testFailure3
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
 'foo
-bar
+bah
 baz
 '
```

```
/home/sb/EqualsTest.php:16

FAILURES!
Tests: 3, Assertions: 3, Failures: 3.
```

Des comparaisons plus spécialisées sont utilisés pour des types d'arguments spécifiques pour `$expected` et `$actual`, voir ci-dessous.

`assertEquals(float $expected, float $actual[, string $message = '', float $delta = 0])`

Signale une erreur identifiée par `$message` si l'écart entre les deux nombres réels `$expected` et `$actual` est inférieur à `$delta`.

Lisez "What Every Computer Scientist Should Know About Floating-Point Arithmetic [http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html]" pour comprendre pourquoi `$delta` est nécessaire.

Exemple A.17. Utilisation de assertEquals() avec des nombres réels

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

class EqualsTest extends TestCase
{
    public function testSuccess()
    {
        $this->assertEquals(1.0, 1.1, '', 0.2);
    }

    public function testFailure()
    {
        $this->assertEquals(1.0, 1.1);
    }
}
?>
```

```
phpunit EqualsTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

.F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

1) EqualsTest::testFailure
Failed asserting that 1.1 matches expected 1.0.

/home/sb/EqualsTest.php:11

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.
```

`assertEquals(DOMDocument $expected, DOMDocument $actual[, string $message = ''])`

Signale une erreur identifiée par `$message` si la forme cannonique sans commentaires des documents XML représentéspar les deux objets DOMDocument `$expected` et `$actual` ne sont pas égaux.

Exemple A.18. Utilisation de assertEquals() avec des objets DOMDocument

```
<?php
```

```
use PHPUnit\Framework\TestCase;

class EqualsTest extends TestCase
{
    public function testFailure()
    {
        $expected = new DOMDocument;
        $expected->loadXML('<foo><bar/></foo>');

        $actual = new DOMDocument;
        $actual->loadXML('<bar><foo/></bar>');

        $this->assertEquals($expected, $actual);
    }
}
?>
```

```
phpunit EqualsTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) EqualsTest::testFailure
Failed asserting that two DOM documents are equal.
--- Expected
+++ Actual
@@ @@
 <?xml version="1.0"?>
-<foo>
-  <bar/>
-</foo>
+<bar>
+  <foo/>
+</bar>

/home/sb/EqualsTest.php:12

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

`assertEquals(object $expected, object $actual[, string $message = ''])`

Signale une erreur identifiée par `$message` si les deux objets `$expected` et `$actual` n'ont pas les attributs avec des valeurs égales.

Exemple A.19. Utilisation de assertEquals() avec des objets

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

class EqualsTest extends TestCase
{
    public function testFailure()
    {
        $expected = new stdClass;
        $expected->foo = 'foo';
        $expected->bar = 'bar';
```

```
        $actual = new stdClass;
        $actual->foo = 'bar';
        $actual->baz = 'bar';

        $this->assertEquals($expected, $actual);
    }
}
?>
```

```
phpunit EqualsTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) EqualsTest::testFailure
Failed asserting that two objects are equal.
--- Expected
+++ Actual
@@ @@
 stdClass Object (
-    'foo' => 'foo'
-    'bar' => 'bar'
+    'foo' => 'bar'
+    'baz' => 'bar'
 )

/home/sb/EqualsTest.php:14

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

`assertEquals(array $expected, array $actual[, string $message = ''])`

Signale une erreur identifiée par `$message` si les deux tableaux `$expected` et `$actual` ne sont pas égaux.

Exemple A.20. Utilisation de assertEquals() avec des tableaux

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

class EqualsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertEquals(['a', 'b', 'c'], ['a', 'c', 'd']);
    }
}
?>
```

```
phpunit EqualsTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:
```

```
1) EqualsTest::testFailure
Failed asserting that two arrays are equal.
--- Expected
+++ Actual
@@ @@
 Array (
     0 => 'a'
-    1 => 'b'
-    2 => 'c'
+    1 => 'c'
+    2 => 'd'
 )

/home/sb/EqualsTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertFalse()

`assertFalse(bool $condition[, string $message = ''])`

Signale une erreur identifiée par `$message` si `$condition` est `true`.

`assertNotFalse()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.21. Utilisation de assertFalse()

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

class FalseTest extends TestCase
{
    public function testFailure()
    {
        $this->assertFalse(true);
    }
}
?>
```

```
phpunit FalseTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) FalseTest::testFailure
Failed asserting that true is false.

/home/sb/FalseTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertFileEquals()

`assertFileEquals(string $expected, string $actual[, string $message = ''])`

Signale une erreur identifiée par `$message` si le fichier spécifié par `$expected` n'a pas les mêmes contenus que le fichier spécifié par `$actual`.

`assertFileNotEquals()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.22. Utilisation de assertFileEquals()

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

class FileEqualsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertFileEquals('/home/sb/expected', '/home/sb/actual');
    }
}
?>
```

```
phpunit FileEqualsTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) FileEqualsTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'expected
+'actual
 '

/home/sb/FileEqualsTest.php:6

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

assertFileExists()

`assertFileExists(string $filename[, string $message = ''])`

Signale une erreur identifiée par `$message` si le fichier spécifié par `$filename` n'existe pas.

`assertFileNotExists()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.23. Utilisation de assertFileExists()

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

class FileExistsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertFileExists('/path/to/file');
    }
}
```

```
?>
```

```
phpunit FileExistsTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) FileExistsTest::testFailure
Failed asserting that file "/path/to/file" exists.

/home/sb/FileExistsTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertFileIsReadable()

`assertFileIsReadable(string $filename[, string $message = ''])`

Signale une erreur identifiée par `$message` si le fichier spécifié par `$filename` n'est pas un fichier ou n'est pas accessible en lecture.

`assertFileNotIsReadable()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.24. Utilisation de assertFileIsReadable()

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

class FileIsReadableTest extends TestCase
{
    public function testFailure()
    {
        $this->assertFileIsReadable('/path/to/file');
    }
}
?>
```

```
phpunit FileIsReadableTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) FileIsReadableTest::testFailure
Failed asserting that "/path/to/file" is readable.

/home/sb/FileIsReadableTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertFileIsWritable()

```
assertFileIsWritable(string $filename[, string $message = ''])
```

Signale une erreur identifiée par `$message` si le fichier spécifié par `$filename` n'est pas un fichier ou n'est pas accessible en éciture.

`assertFileNotIsWritable()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.25. Utilisation de assertFileIsWritable()

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

class FileIsWritableTest extends TestCase
{
    public function testFailure()
    {
        $this->assertFileIsWritable('/path/to/file');
    }
}
?>
```

```
phpunit FileIsWritableTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) FileIsWritableTest::testFailure
Failed asserting that "/path/to/file" is writable.

/home/sb/FileIsWritableTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertGreaterThan()

```
assertGreaterThan(mixed $expected, mixed $actual[, string $message
= ''])
```

Signale une erreur identifiée par `$message` si la valeur de `$actual` n'est pas plus élevée que la valeur de `$expected`.

`assertAttributeGreaterThan()` est un encapsulateur de commodité qui utilise un attribut `public`, `protected` ou `private` d'une classe ou d'un objet en tant que valeur.

Exemple A.26. Utilisation de assertGreaterThan()

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

class GreaterThanTest extends TestCase
{
    public function testFailure()
    {
        $this->assertGreaterThan(2, 1);
    }
}
?>
```

```
phpunit GreaterThanTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) GreaterThanTest::testFailure
Failed asserting that 1 is greater than 2.

/home/sb/GreaterThanTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertGreaterThanOrEqual()

`assertGreaterThanOrEqual(mixed $expected, mixed $actual[, string $message = ''])`

Signale une erreur identifiée par `$message` si la valeur de `$actual` n'est pas plus grande ou égale à la valeur de `$expected`.

`assertAttributeGreaterThanOrEqual()` est un encapsulateur de commodité qui utilise un attribut `public`, `protected` ou `private` d'une classe ou d'un objet en tant que valeur.

Exemple A.27. Utilisation de assertGreaterThanOrEqual()

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

class GreatThanOrEqualTest extends TestCase
{
    public function testFailure()
    {
        $this->assertGreaterThanOrEqual(2, 1);
    }
}
?>
```

```
phpunit GreaterThanOrEqualTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) GreatThanOrEqualTest::testFailure
Failed asserting that 1 is equal to 2 or is greater than 2.

/home/sb/GreaterThanOrEqualTest.php:6

FAILURES!
Tests: 1, Assertions: 2, Failures: 1.
```

assertInfinite()

```
assertInfinite(mixed $variable[, string $message = ''])
```

Signale une erreur identifiée par `$message` si `$variable` n'est pas `INF`.

`assertFinite()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.28. Utilisation de assertInfinite()

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

class InfiniteTest extends TestCase
{
    public function testFailure()
    {
        $this->assertInfinite(1);
    }
}
?>
```

```
phpunit InfiniteTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) InfiniteTest::testFailure
Failed asserting that 1 is infinite.

/home/sb/InfiniteTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertInstanceOf()

```
assertInstanceOf($expected, $actual[, $message = ''])
```

Signale une erreur identifiée par `$message` si `$actual` n'est pas une instance de `$expected`.

`assertNotInstanceOf()` est l'inverse de cette assertion et prend les mêmes arguments.

`assertAttributeInstanceOf()` et `assertAttributeNotInstanceOf()` sont des encapsulateurs de commodités qui peuvent être appliqués à un attribut `public`, `protected` ou `private` d'une classe ou d'un objet.

Exemple A.29. Utilisation de assertInstanceOf()

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

class InstanceOfTest extends TestCase
{
    public function testFailure()
    {
        $this->assertInstanceOf(RuntimeException::class, new Exception);
    }
}
```

```
?>
```

```
phpunit InstanceOfTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) InstanceOfTest::testFailure
Failed asserting that Exception Object (...) is an instance of class "RuntimeException".

/home/sb/InstanceOfTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertInternalType()

`assertInternalType($expected, $actual[, $message = ''])`

Signale une erreur identifiée par `$message` si `$actual` n'est pas du type `$expected`.

`assertNotInternalType()` est l'inverse de cette assertion et prend les mêmes arguments.

`assertAttributeInternalType()` et `assertAttributeNotInternalType()` sont des encapsulateurs de commodités qui peuvent être appliqués a un attribut `public`, `protected` ou `private` d'une classe ou d'un objet.

Exemple A.30. Utilisation de assertInternalType()

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

class InternalTypeTest extends TestCase
{
    public function testFailure()
    {
        $this->assertInternalType('string', 42);
    }
}
?>
```

```
phpunit InternalTypeTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) InternalTypeTest::testFailure
Failed asserting that 42 is of type "string".

/home/sb/InternalTypeTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertIsReadable()

`assertIsReadable(string $filename[, string $message = ''])`

Signale une erreur identifiée par `$message` si le fichier ou le répertoire spécifié par `$filename` n'est pas accessible en lecture.

`assertNotIsReadable()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.31. Utilisation de assertIsReadable()

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

class IsReadableTest extends TestCase
{
    public function testFailure()
    {
        $this->assertIsReadable('/path/to/unreadable');
    }
}
?>
```

```
phpunit IsReadableTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) IsReadableTest::testFailure
Failed asserting that "/path/to/unreadable" is readable.

/home/sb/IsReadableTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertIsWritable()

`assertIsWritable(string $filename[, string $message = ''])`

Signale une erreur identifiée par `$message` si le fichier ou le répertoire spécifié par `$filename` n'est pas accessible en écriture.

`assertNotIsWritable()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.32. Utilisation de assertIsWritable()

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

class IsWritableTest extends TestCase
{
    public function testFailure()
    {
        $this->assertIsWritable('/path/to/unwritable');
    }
}
```

```
?>
```

```
phpunit IsWritableTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) IsWritableTest::testFailure
Failed asserting that "/path/to/unwritable" is writable.

/home/sb/IsWritableTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertJsonFileEqualsJsonFile()

`assertJsonFileEqualsJsonFile(mixed $expectedFile, mixed $actualFile[, string $message = ''])`

Signale une erreur identifiée par `$message` si la valeur de `$actualFile` ne correspond pas à la valeur de `$expectedFile`.

Exemple A.33. Utilisation de assertJsonFileEqualsJsonFile()

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

class JsonFileEqualsJsonFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertJsonFileEqualsJsonFile(
          'path/to/fixture/file', 'path/to/actual/file');
    }
}
?>
```

```
phpunit JsonFileEqualsJsonFileTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) JsonFileEqualsJsonFile::testFailure
Failed asserting that '{"Mascot":"Tux"}' matches JSON string "["Mascott", "Tux", "OS", "

/home/sb/JsonFileEqualsJsonFileTest.php:5

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

assertJsonStringEqualsJsonFile()

```
assertJsonStringEqualsJsonFile(mixed $expectedFile, mixed $actualJ-
son[, string $message = ''])
```

Signale une erreur identifiée par `$message` si la valeur de `$actualJson` ne correspond pas à la valeur de `$expectedFile`.

Exemple A.34. Utilisation de assertJsonStringEqualsJsonFile()

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

class JsonStringEqualsJsonFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertJsonStringEqualsJsonFile(
            'path/to/fixture/file', json_encode(['Mascot' => 'ux'])
        );
    }
}
?>
```

```
phpunit JsonStringEqualsJsonFileTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) JsonStringEqualsJsonFile::testFailure
Failed asserting that '{"Mascot":"ux"}' matches JSON string "{"Mascott":"Tux"}".

/home/sb/JsonStringEqualsJsonFileTest.php:5

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

assertJsonStringEqualsJsonString()

```
assertJsonStringEqualsJsonString(mixed $expectedJson, mixed $actual-
Json[, string $message = ''])
```

Signale une erreur identifiée par `$message` si la valeur de `$actualJson` ne correspond pas à la valeur de `$expectedJson`.

Exemple A.35. Utilisation de assertJsonStringEqualsJsonString()

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

class JsonStringEqualsJsonStringTest extends TestCase
{
    public function testFailure()
    {
        $this->assertJsonStringEqualsJsonString(
            json_encode(['Mascot' => 'Tux']),
            json_encode(['Mascot' => 'ux'])
        );
    }
```

```
}
?>
```

```
phpunit JsonStringEqualsJsonStringTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) JsonStringEqualsJsonStringTest::testFailure
Failed asserting that two objects are equal.
--- Expected
+++ Actual
@@ @@
 stdClass Object (
 -    'Mascot' => 'Tux'
 +    'Mascot' => 'ux'
)

/home/sb/JsonStringEqualsJsonStringTest.php:5

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

assertLessThan()

`assertLessThan(mixed $expected, mixed $actual[, string $message = ''])`

Signale une erreur identifiée par `$message` si la valeur de `$actual` n'est pas plus petit que `$expected`.

`assertAttributeLessThan()` est un encapsulateur de commodité qui utilise un attribut `public`, `protected` ou `private` d'une classe ou d'un objet en tant que valeur.

Exemple A.36. Utilisation de assertLessThan()

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

class LessThanTest extends TestCase
{
    public function testFailure()
    {
        $this->assertLessThan(1, 2);
    }
}
?>
```

```
phpunit LessThanTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) LessThanTest::testFailure
```

```
Failed asserting that 2 is less than 1.

/home/sb/LessThanTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertLessThanOrEqual()

`assertLessThanOrEqual(mixed $expected, mixed $actual[, string $message = ''])`

Signale une erreur identifiée par `$message` si la valeur de `$actual` n'est pas plus petite ou égale à la valeur de `$expected`.

`assertAttributeLessThanOrEqual()` est un encapsulateur de commodité qui utilise un attribut `public`, `protected` ou `private` d'une classe ou d'un objet en tant que valeur.

Exemple A.37. Utilisation de assertLessThanOrEqual()

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

class LessThanOrEqualTest extends TestCase
{
    public function testFailure()
    {
        $this->assertLessThanOrEqual(1, 2);
    }
}
?>
```

```
phpunit LessThanOrEqualTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) LessThanOrEqualTest::testFailure
Failed asserting that 2 is equal to 1 or is less than 1.

/home/sb/LessThanOrEqualTest.php:6

FAILURES!
Tests: 1, Assertions: 2, Failures: 1.
```

assertNan()

`assertNan(mixed $variable[, string $message = ''])`

Signale une erreur identifiée par `$message` si `$variable` n'est pas `NAN`.

Exemple A.38. Utilisation de assertNan()

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

```
class NanTest extends TestCase
{
    public function testFailure()
    {
        $this->assertNan(1);
    }
}
?>
```

```
phpunit NanTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) NanTest::testFailure
Failed asserting that 1 is nan.

/home/sb/NanTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertNull()

`assertNull(mixed $variable[, string $message = ''])`

Signale une erreur identifiée par `$message` si `$variable` n'est pas `null`.

`assertNotNull()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.39. Utilisation de assertNull()

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

class NullTest extends TestCase
{
    public function testFailure()
    {
        $this->assertNull('foo');
    }
}
?>
```

```
phpunit NotNullTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) NullTest::testFailure
Failed asserting that 'foo' is null.

/home/sb/NotNullTest.php:6
```

```
FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertObjectHasAttribute()

`assertObjectHasAttribute(string $attributeName, object $object[, string $message = ''])`

Signale une erreur identifiée par `$message` si `$object->attributeName` n'existe pas.

`assertObjectNotHasAttribute()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.40. Utilisation de assertObjectHasAttribute()

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

class ObjectHasAttributeTest extends TestCase
{
    public function testFailure()
    {
        $this->assertObjectHasAttribute('foo', new stdClass);
    }
}
?>
```

```
phpunit ObjectHasAttributeTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ObjectHasAttributeTest::testFailure
Failed asserting that object of class "stdClass" has attribute "foo".

/home/sb/ObjectHasAttributeTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertRegExp()

`assertRegExp(string $pattern, string $string[, string $message = ''])`

Signale une erreur identifiée par `$message` si `$string` ne correspond pas a l'expression régulière `$pattern`.

`assertNotRegExp()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.41. Utilisation de assertRegExp()

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

class RegExpTest extends TestCase
{
    public function testFailure()
```

```
    {
        $this->assertRegExp('/foo/', 'bar');
    }
}
?>
```

```
phpunit RegExpTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) RegExpTest::testFailure
Failed asserting that 'bar' matches PCRE pattern "/foo/".

/home/sb/RegExpTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertStringMatchesFormat()

`assertStringMatchesFormat(string $format, string $string[, string $message = ''])`

Signale une erreur identifiée par `$message` si `$string` ne correspond pas à la chaine `$format`.

`assertStringNotMatchesFormat()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.42. Utilisation de assertStringMatchesFormat()

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

class StringMatchesFormatTest extends TestCase
{
    public function testFailure()
    {
        $this->assertStringMatchesFormat('%i', 'foo');
    }
}
?>
```

```
phpunit StringMatchesFormatTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) StringMatchesFormatTest::testFailure
Failed asserting that 'foo' matches PCRE pattern "/^[+-]?\d+$/s".

/home/sb/StringMatchesFormatTest.php:6

FAILURES!
```

```
Tests: 1, Assertions: 1, Failures: 1.
```

Le format de chaine peut contenir les arguments suivants:

- `%e`: Représente un séparateur de répertoire, par exemple `/` sur Linux.
- `%s`: Un ou plusieurs de n'importe quoi (caractère ou espace) sauf le caractère de fin de ligne.
- `%S`: Zéro ou pusieurs de n'importe quoi (caractère ou espace) sauf le caractère de fin de ligne.
- `%a`: Un ou plusieurs de n'importe quoi (caractère ou espace) incluant le caratère de fin de ligne.
- `%A`: Zéro ou pusieurs de n'importe quoi (caractère ou espace) incluant le caratère de fin de ligne.
- `%w`: Zéro ou plusieurs caratères espace.
- `%i`: Une valeure entière signée, par exemple `+3142`, `-3142`.
- `%d`: Une valeure entière non signée, par exemple `123456`.
- `%x`: Un ou plusieurs caractères hexadecimaux. Ce sont les caractères compris entre `0-9`, `a-f`, `A-F`.
- `%f`: Un nombre à virgule flottante, par exemple: `3.142`, `-3.142`, `3.142E-10`, `3.142e+10`.
- `%c`: Un caractère unique quel qu'il soit.

assertStringMatchesFormatFile()

```
assertStringMatchesFormatFile(string $formatFile, string $string[,
string $message = ''])
```

Signale une erreur identifiée par `$message` si `$string` ne correspond pas au contenu du `$formatFile`.

`assertStringNotMatchesFormatFile()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.43. Utilisation de assertStringMatchesFormatFile()

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

class StringMatchesFormatFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertStringMatchesFormatFile('/path/to/expected.txt', 'foo');
    }
}
?>
```

```
phpunit StringMatchesFormatFileTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) StringMatchesFormatFileTest::testFailure
Failed asserting that 'foo' matches PCRE pattern "/^[+-]?\d+
$/s".
```

```
/home/sb/StringMatchesFormatFileTest.php:6

FAILURES!
Tests: 1, Assertions: 2, Failures: 1.
```

assertSame()

`assertSame(mixed $expected, mixed $actual[, string $message = ''])`

Signale une erreur identifiée par `$message` si les deux variables `$expected` et `$actual` ne sont pas du même type et n'ont pas la même valeur.

`assertNotSame()` est l'inverse de cette assertion et prend les mêmes arguments.

`assertAttributeSame()` et `assertAttributeNotSame()` sont des encapsulateurs de commodités qui utilisent un attribut `public`, `protected` ou `private` d'une classe ou d'un objet en tant que valeur.

Exemple A.44. Utilisation de assertSame()

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

class SameTest extends TestCase
{
    public function testFailure()
    {
        $this->assertSame('2204', 2204);
    }
}
?>
```

```
phpunit SameTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) SameTest::testFailure
Failed asserting that 2204 is identical to '2204'.

/home/sb/SameTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

`assertSame(object $expected, object $actual[, string $message = ''])`

Signale une erreur identifiée par `$message` si les deux variables `$expected` et `$actual` ne référencent pas le même objet.

Exemple A.45. Utilisation de assertSame() with objects

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

class SameTest extends TestCase
{
```

```
    public function testFailure()
    {
        $this->assertSame(new stdClass, new stdClass);
    }
}
?>
```

```
phpunit SameTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) SameTest::testFailure
Failed asserting that two variables reference the same object.

/home/sb/SameTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertStringEndsWith()

`assertStringEndsWith(string $suffix, string $string[, string $message = ''])`

Signale une erreur identifiée par `$message` si `$string` ne termine pas par `$suffix`.

`assertStringEndsNotWith()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.46. Utilisation de assertStringEndsWith()

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

class StringEndsWithTest extends TestCase
{
    public function testFailure()
    {
        $this->assertStringEndsWith('suffix', 'foo');
    }
}
?>
```

```
phpunit StringEndsWithTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 1 second, Memory: 5.00Mb

There was 1 failure:

1) StringEndsWithTest::testFailure
Failed asserting that 'foo' ends with "suffix".

/home/sb/StringEndsWithTest.php:6

FAILURES!
```

```
Tests: 1, Assertions: 1, Failures: 1.
```

assertStringEqualsFile()

`assertStringEqualsFile(string $expectedFile, string $actualString[, string $message = ''])`

Signale une erreur identifiée par `$message` le fichier spécifié par `$expectedFile` n'a pas `$actualString` comme contenu.

`assertStringNotEqualsFile()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.47. Utilisation de assertStringEqualsFile()

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

class StringEqualsFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertStringEqualsFile('/home/sb/expected', 'actual');
    }
}
?>
```

```
phpunit StringEqualsFileTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) StringEqualsFileTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'expected
-'
+'actual'

/home/sb/StringEqualsFileTest.php:6

FAILURES!
Tests: 1, Assertions: 2, Failures: 1.
```

assertStringStartsWith()

`assertStringStartsWith(string $prefix, string $string[, string $message = ''])`

Signale une erreur identifiée par `$message` si `$string` ne commence pas par `$prefix`.

`assertStringStartsNotWith()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.48. Utilisation de assertStringStartsWith()

```
<?php
```

```
use PHPUnit\Framework\TestCase;

class StringStartsWithTest extends TestCase
{
    public function testFailure()
    {
        $this->assertStringStartsWith('prefix', 'foo');
    }
}
?>
```

```
phpunit StringStartsWithTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) StringStartsWithTest::testFailure
Failed asserting that 'foo' starts with "prefix".

/home/sb/StringStartsWithTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertThat()

Des assertions plus complexes peuvent être formulées en utilisant les classes `PHPUnit_Framework_Constraint`. Elles peuvent être evaluées avec la méthode `assertThat()`. Exemple A.49, « Utilisation de assertThat() » montre comment les contraintes `logicalNot()` et `equalTo()` peuvent être utilisées pour exprimer la même assertion que `assertNotEquals()`.

```
assertThat(mixed $value, PHPUnit_Framework_Constraint $constraint[,
$message = ''])
```

Signale une erreur identifiée par `$message`si `$value` ne correspond pas à `$constraint`.

Exemple A.49. Utilisation de assertThat()

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

class BiscuitTest extends TestCase
{
    public function testEquals()
    {
        $theBiscuit = new Biscuit('Ginger');
        $myBiscuit  = new Biscuit('Ginger');

        $this->assertThat(
          $theBiscuit,
          $this->logicalNot(
            $this->equalTo($myBiscuit)
          )
        );
    }
}
```

```
?>
```

Tableau A.1, « Contraintes » montre les classes `PHPUnit_Framework_Constraint` disponibles.

Tableau A.1. Contraintes

Contrainte	Signification
`PHPUnit_ Framework_ Constraint_ Attribute attribute(PHPUnit_ Framework_ Constraint $constraint, $attributeName)`	Contrainte qui applique une autre contrainte à un attribut d'une classe ou d'un objet.
`PHPUnit_ Framework_ Constraint_ IsAnything anything()`	Contrainte qui accepte toute valeur d'entrée.
`PHPUnit_ Framework_ Constraint_ ArrayHasKey arrayHasKey(mixed $key)`	Ccontrainte qui valide que le tableau évalué a une clé donnée.
`PHPUnit_ Framework_ Constraint_ TraversableContains contains(mixed $value)`	Contrainte qui valide que le `tableau` ou l'objet qui implémente l'interface `Iterator` évalué contient une valeur donnée.
`PHPUnit_ Framework_ Constraint_ TraversableContainsOnly containsOnly(string $type)`	Contrainte qui valide que le `tableau` ou l'objet qui implémente l'interface `Iterator` évalué ne contient que des valeurs d'un type donné.
`PHPUnit_ Framework_ Constraint_ TraversableContainsOnly containsOnlyInstancesOf(string $classname)`	Contrainte qui valide que le `tableau` ou l'objet qui implémente l'interface `Iterator` évalué ne contient que des instance d'une classe donnée.
`PHPUnit_ Framework_ Constraint_ IsEqual equalTo($value, $delta = 0, $maxDepth = 10)`	Contrainte qui vérifie si une valeur est égale à une autre.
`PHPUnit_ Framework_ Constraint_ Attribute attributeEqualTo($attributeName, $value, $delta = 0, $maxDepth = 10)`	Contrainte qui vérifie si une valeur est égale à un attribut d'une classe ou d'un objet.
`PHPUnit_ Framework_ Constraint_ DirectoryExists directo- ryExists()`	Contrainte qui vérifie si le répertoire évalué existe.
`PHPUnit_ Framework_ Constraint_ FileExists fileExists()`	Contrainte qui vérifie si le fichier(name) évalué existe.
`PHPUnit_ Framework_ Constraint_ IsReadable isReadable()`	Contrainte qui vérifie si le fichier(name) évalué est accessible en écriture.
`PHPUnit_ Framework_ Constraint_ IsWritable isWritable()`	Contrainte qui vérifie si le fichier(name) évalué est accessible en écriture.
`PHPUnit_ Framework_ Constraint_ GreaterThan greaterThan(mixed $value)`	Contrainte qui valide que la valeur évaluée est supérieure à une valeur donnée.
`PHPUnit_ Framework_ Constraint_ Or greaterThanOrEqual(mixed $va- lue)`	Contrainte qui valide que la valeur évaluée est supérieure ou égale à une valeur donnée.
`PHPUnit_ Framework_ Constraint_ ClassHasAttribute`	La contrainte qui valide que la classe évaluée possède un attribut donné.

Contrainte	Signification
`classHasAttribute(string $attributeName)`	
`PHPUnit_ Framework_ Constraint_ ClassHasStaticAttribute classHasStaticAttribute(string $attributeName)`	La contrainte qui valide que la classe évaluée possède un attribut statique donné.
`PHPUnit_ Framework_ Constraint_ ObjectHasAttribute hasAttribute(string $attributeName)`	La contrainte qui valide que l'objet évalué possède un attribut donné.
`PHPUnit_ Framework_ Constraint_ IsIdentical identicalTo(mixed $value)`	Contrainte qui valide qu'une valeur est identique à une autre.
`PHPUnit_ Framework_ Constraint_ IsFalse isFalse()`	Contrainte qui valide qu'une valeur évaluée est `false`.
`PHPUnit_ Framework_ Constraint_ IsInstanceOf isInstanceOf(string $className)`	La contrainte qui VALIDE que l'objet évalué est une instance d'une classe donnée.
`PHPUnit_ Framework_ Constraint_ IsNull isNull()`	Contrainte qui valide qu'une valeur évaluée est `null`.
`PHPUnit_ Framework_ Constraint_ IsTrue isTrue()`	Contrainte qui valide qu'une valeur évaluée est `true`.
`PHPUnit_ Framework_ Constraint_ IsType isType(string $type)`	Contrainte qui valide que la valeur évaluée est d'un type spécifié.
`PHPUnit_ Framework_ Constraint_ LessThan lessThan(mixed $value)`	Constraint that asserts that the value it is evaluated for is smaller than a given value.
`PHPUnit_ Framework_ Constraint_ Or lessThanOrEqual(mixed $value)`	Contrainte qui valide que la valeur évaluée est inférieure ou égale à une valeur donnée.
`logicalAnd()`	ET logique (AND).
`logicalNot(PHPUnit_ Framework_ Constraint $constraint)`	NON logique (NOT).
`logicalOr()`	OU logique (OR).
`logicalXor()`	OU exclusif logique (XOR).
`PHPUnit_ Framework_ Constraint_ PCREMatch matchesRegularExpression(string $pattern)`	Contrainte qui valide que la chaine évaluée correspond à une expression régulière.
`PHPUnit_ Framework_ Constraint_ StringContains stringContains(string $string, bool $case)`	Contrainte qui valide que la chaine évaluée contient une chaine donnée.
`PHPUnit_ Framework_ Constraint_ StringEndsWith stringEndsWith(string $suffix)`	Contrainte qui valide que la chaine évaluée termine par un suffix donné.
`PHPUnit_ Framework_ Constraint_ StringStartsWith stringStartsWith(string $prefix)`	Contrainte qui valide que la chaine évaluée commence par un préfix donné.

assertTrue()

```
assertTrue(bool $condition[, string $message = ''])
```

Signale une erreur identifiée par `$message` si `$condition` est `false`.

`assertNotTrue()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.50. Utilisation de assertTrue()

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

class TrueTest extends TestCase
{
    public function testFailure()
    {
        $this->assertTrue(false);
    }
}
?>
```

```
phpunit TrueTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) TrueTest::testFailure
Failed asserting that false is true.

/home/sb/TrueTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

assertXmlFileEqualsXmlFile()

```
assertXmlFileEqualsXmlFile(string $expectedFile, string $actual-
File[, string $message = ''])
```

Signale une erreur identifiée par `$message` si le document XML dans `$actualFile` n'est pas égal au document XML dans `$expectedFile`.

`assertXmlFileNotEqualsXmlFile()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.51. Utilisation de assertXmlFileEqualsXmlFile()

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

class XmlFileEqualsXmlFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertXmlFileEqualsXmlFile(
          '/home/sb/expected.xml', '/home/sb/actual.xml');
    }
}
```

```
?>
```

```
phpunit XmlFileEqualsXmlFileTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) XmlFileEqualsXmlFileTest::testFailure
Failed asserting that two DOM documents are equal.
--- Expected
+++ Actual
@@ @@
 <?xml version="1.0"?>
 <foo>
-  <bar/>
+  <baz/>
 </foo>

/home/sb/XmlFileEqualsXmlFileTest.php:7

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

assertXmlStringEqualsXmlFile()

`assertXmlStringEqualsXmlFile(string $expectedFile, string $actualXml[, string $message = ''])`

Signale une erreur identifiée par `$message` si le document XML dans `$actualXml` n'est pas égal au document XML dans `$expectedFile`.

`assertXmlStringNotEqualsXmlFile()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.52. Utilisation de assertXmlStringEqualsXmlFile()

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

class XmlStringEqualsXmlFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertXmlStringEqualsXmlFile(
          '/home/sb/expected.xml', '<foo><baz/></foo>');
    }
}
?>
```

```
phpunit XmlStringEqualsXmlFileTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:
```

```
1) XmlStringEqualsXmlFileTest::testFailure
Failed asserting that two DOM documents are equal.
--- Expected
+++ Actual
@@ @@
 <?xml version="1.0"?>
 <foo>
-  <bar/>
+  <baz/>
 </foo>

/home/sb/XmlStringEqualsXmlFileTest.php:7

FAILURES!
Tests: 1, Assertions: 2, Failures: 1.
```

assertXmlStringEqualsXmlString()

`assertXmlStringEqualsXmlString(string $expectedXml, string $actualXml[, string $message = ''])`

Signale une erreur identifiée par `$message` si le document XML dans `$actualXml` n'est pas égal au document XML dans `$expectedXml`.

`assertXmlStringNotEqualsXmlString()` est l'inverse de cette assertion et prend les mêmes arguments.

Exemple A.53. Utilisation de assertXmlStringEqualsXmlString()

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

class XmlStringEqualsXmlStringTest extends TestCase
{
    public function testFailure()
    {
        $this->assertXmlStringEqualsXmlString(
          '<foo><bar/></foo>', '<foo><baz/></foo>');
    }
}
?>
```

```
phpunit XmlStringEqualsXmlStringTest
PHPUnit 6.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) XmlStringEqualsXmlStringTest::testFailure
Failed asserting that two DOM documents are equal.
--- Expected
+++ Actual
@@ @@
 <?xml version="1.0"?>
 <foo>
-  <bar/>
+  <baz/>
 </foo>
```

```
/home/sb/XmlStringEqualsXmlStringTest.php:7

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Annexe B. Annotations

Une annotation est une forme spéciale de métadonnée syntaxique qui peut être ajoutée au code source de certains langages de programmation. Bien que PHP n'ait pas de fonctionnalité dédiée à l'annotation du code source, l'utilisation d'étiquettes telles que `@annotation arguments` dans les blocs de documentation s'est établi dans la communauté PHP pour annoter le code source. En PHP, les blocs de documentation sont réflexifs : ils peuvent être accédés via la méthode de l'API de réflexivité `getDocComment()` au niveau des fonctions, classes, méthodes et attributs. Des applications telles que PHPUnit utilisent ces informations durant l'exécution pour adapter leur comportement.

Note

Un "doc comment" en PHP doit commencé par `/**` et se terminer avec `*/`. Les annotations se trouvant des des commentaires d'un autre style seront ignorées.

Cette annexe montre toutes les sortes d'annotations gérées par PHPUnit.

@author

L'annotation `@author` est un alias pour l'annotation `@group` (voir la section intitulée « @group ») et permet de filtrer des tests selon leurs auteurs.

@after

L'annotation `@after` peut être utilisé pour spécifier des méthodes devant être appelées après chaque méthode de test dans une classe de cas de tests.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @after
     */
    public function tearDownSomeFixtures()
    {
        // ...
    }

    /**
     * @after
     */
    public function tearDownSomeOtherFixtures()
    {
        // ...
    }
}
```

@afterClass

L'annotation `@afterClass` peut-être utilisées pour spécifier des méthodes statiques devant être appelées après chaque méthode de test dans une classe de test pour être exécuté afin de nettoyer des fixtures partagées.

```
use PHPUnit\Framework\TestCase;
```

```
class MyTest extends TestCase
{
    /**
     * @afterClass
     */
    public static function tearDownSomeSharedFixtures()
    {
        // ...
    }

    /**
     * @afterClass
     */
    public static function tearDownSomeOtherSharedFixtures()
    {
        // ...
    }
}
```

@backupGlobals

Les opérations de sauvegarde et de restauration des variables globales peuvent être complètement désactivées pour tous les tests d'une classe de cas de test comme ceci :

```
use PHPUnit\Framework\TestCase;

/**
 * @backupGlobals disabled
 */
class MyTest extends TestCase
{
    // ...
}
```

L'annotation `@backupGlobals` peut également être utilisée au niveau d'une méthode. Cela permet une configuration fine des des opérations de sauvegarde et de restauration :

```
use PHPUnit\Framework\TestCase;

/**
 * @backupGlobals disabled
 */
class MyTest extends TestCase
{
    /**
     * @backupGlobals enabled
     */
    public function testThatInteractsWithGlobalVariables()
    {
        // ...
    }
}
```

@backupStaticAttributes

L'annotation `@backupStaticAttributes` peut être utilisée pour enregistrer tous les attributs statiques dans toutes les classes déclarées avant chaque test et les restaurer après. Elle peut être utilisé au niveau de la classe ou au niveau de la méthode :

```
use PHPUnit\Framework\TestCase;
```

```
/**
 * @backupStaticAttributes enabled
 */
class MyTest extends TestCase
{
    /**
     * @backupStaticAttributes disabled
     */
    public function testThatInteractsWithStaticAttributes()
    {
        // ...
    }
}
```

Note

`@backupStaticAttributes` est limitée par le fonctionnement interne de PHP et peut entrainer la persistance inattendue de valeurs statique et fuiter dans les tests suivants tests dans certaines circonstances.

Voir la section intitulée « Etat global » pour les détails.

@before

L'annotation `@before` peut être utilisée pour spécifier des méthodes devant être appelées avant chaque méthode de test dans une classe de cas de test.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @before
     */
    public function setupSomeFixtures()
    {
        // ...
    }

    /**
     * @before
     */
    public function setupSomeOtherFixtures()
    {
        // ...
    }
}
```

@beforeClass

L'annotation `@beforeClass` peut être utilisée pour spécifier des méthodes statiques qui doivent être appellées avant chaque méthodes de test dans une classe de test qui sont exécutés pour paramétrer des fixtures partagées.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
```

```
    /**
     * @beforeClass
     */
    public static function setUpSomeSharedFixtures()
    {
        // ...
    }

    /**
     * @beforeClass
     */
    public static function setUpSomeOtherSharedFixtures()
    {
        // ...
    }
}
```

@codeCoverageIgnore*

Les annotations `@codeCoverageIgnore`, `@codeCoverageIgnoreStart` et `@codeCoverageIgnoreEnd` peuvent être utilisées pour exclure des lignes de code de l'analyse de couverture.

Pour la manière de les utiliser, voir la section intitulée « Ignorer des blocs de code ».

@covers

L'annotation `@covers` peut être utilisée dans le code de test pour indique quelle(s) méthode(s) une méthode de test veut tester:

```
/**
 * @covers BankAccount::getBalance
 */
public function testBalanceIsInitiallyZero()
{
    $this->assertEquals(0, $this->ba->getBalance());
}
```

Si elle est fournie, seule l'information de couverture de code pour la(les) méthode(s) sera prise en considération.

Tableau B.1, « Annotations pour indiquer quelles méthodes sont couvertes par un test » montre la syntaxe de l'annotation `@covers`.

Tableau B.1. Annotations pour indiquer quelles méthodes sont couvertes par un test

Annotation	Description
`@covers ClassName::methodName`	Indique que la méthode de test annotée couvre la méthode indiquée.
`@covers ClassName`	Indique que la méthode de test annotée couvre toutes les méthodes d'une classe donnée.
`@covers ClassName<extended>`	Indique que la méthode de test annotée couvre toutes les méthodes d'une classe donnée

Annotation	Description
	ainsi que les classe(s) et interface(s) parentes.
@covers ClassName::<public>	Indique que la méthode de test annotée couvre toutes les méthodes publiques d'une classe donnée.
@covers ClassName::<protected>	Indique que la méthode de test annotée couvre toutes les méthodes protected d'une classe donnée.
@covers ClassName::<private>	Indique que la méthode de test annotée couvre toutes les méthodes privées d'une classe donnée.
@covers ClassName::<!public>	Indique que la méthode de test annotée couvre toutes les méthodes d'une classe donnée qui ne sont pas publiques.
@covers ClassName::<!protected>	Indique que la méthode de test annotée couvre toutes les méthodes d'une classe donnée qui ne sont pas protected.
@covers ClassName::<!private>	Indique que la méthode de test annotée couvre toutes les méthodes d'une classe donnée qui ne sont pas privées.
@covers ::functionName	Indique que la méthode de test annotée couvre la méthode globale spécifiée.

@coversDefaultClass

L'annotation `@coversDefaultClass` peut être utilisée pour spécifier un espace de nom ou un nom de classe par défaut. Ainsi, les noms long n'ont pas besoin d'être répétés pour chaque annotation `@covers`. Voir Exemple B.1, « Utiliser @coversDefaultClass pour simplifier les annotations ».

Exemple B.1. Utiliser @coversDefaultClass pour simplifier les annotations

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

/**
 * @coversDefaultClass \Foo\CoveredClass
 */
class CoversDefaultClassTest extends TestCase
{
    /**
     * @covers ::publicMethod
     */
    public function testSomething()
    {
        $o = new Foo\CoveredClass;
        $o->publicMethod();
    }
}
```

```
?>
```

@coversNothing

L'annotation `@coversNothing` peut être utilisée dans le code de test pour indiquer qu'aucune information de couverture de code ne sera enregistrée pour le cas de test annoté.

Ceci peut être utilisé pour les tests d'intégration. Voir Exemple 11.3, « Un test qui indique qu'aucune méthode ne doit être couverte » pour un exemple.

L'annotation peut être utilisée au niveau de la classe et de la méthode et sera surchargée par toute étiquette `@covers`.

@dataProvider

Une méthode de test peut accepter des paramètres arbitraires. Ces paramètres peuvent être fournis pas une méthode fournisseuse de données ((`provider()` dans Exemple 2.5, « Utiliser un fournisseur de données qui renvoie un tableau de tableaux »). La méthode fournisseur de données peut être indiquée en utilisant l'annotation `@dataProvider`.

Voir la section intitulée « Fournisseur de données » pour plus de détails.

@depends

PHPUnit gère la déclaration des dépendances explicites entre les méthodes de test. De telles dépendances ne définissent pas l'ordre dans lequel les méthodes de test doivent être exécutées mais elles permettent de retourner l'instance d'une fixture de test par un producteur et de la passer aux consommateurs dépendants. Exemple 2.2, « Utiliser l'annotation `@depends` pour exprimer des dépendances » montre comment utiliser l'annotation `@depends` pour exprimer des dépendances entre méthodes de test.

Voir la section intitulée « Dépendances des tests » pour plus de détails.

@expectedException

Exemple 2.10, « Utiliser la méthode expectException() » montre comment utiliser l'annotation `@expectedException` pour tester si une exception est levée dans le code testé.

Voir la section intitulée « Tester des exceptions » pour plus de détails.

@expectedExceptionCode

L'annotation `@expectedExceptionCode`, en conjonction avec `@expectedException` permet de faire des assertions sur le code d'erreur d'une exception levée ce qui permet de cibler une exception particulière.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @expectedException     MyException
     * @expectedExceptionCode 20
     */
    public function testExceptionHasErrorcode20()
    {
        throw new MyException('Some Message', 20);
```

```
    }
}
```

Pour faciliter les tests et réduire la duplication, un raccourci peut être utilisé pour indiquer une constante de classe comme un `@expectedExceptionCode` en utilisant la syntaxe "`@expectedExceptionCode ClassName::CONST`".

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
      * @expectedException     MyException
      * @expectedExceptionCode MyClass::ERRORCODE
      */
    public function testExceptionHasErrorcode20()
    {
      throw new MyException('Some Message', 20);
    }
}
class MyClass
{
    const ERRORCODE = 20;
}
```

@expectedExceptionMessage

L'annotation `@expectedExceptionMessage` fonctionne de manière similaire à `@expectedExceptionCode` et vous permet de faire une assertion sur le message d'erreur d'une exception.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @expectedException        MyException
     * @expectedExceptionMessage Some Message
     */
    public function testExceptionHasRightMessage()
    {
        throw new MyException('Some Message', 20);
    }
}
```

Le message attendu peut être une partie d'une chaîne d'un message d'exception. Ceci peut être utile pour faire une assertion sur le fait qu'un nom ou un paramètre qui est passé s'affiche dans une exception sans fixer la totalité du message d'exception dans le test.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
     /**
      * @expectedException        MyException
      * @expectedExceptionMessage broken
      */
     public function testExceptionHasRightMessage()
     {
         $param = "broken";
         throw new MyException('Invalid parameter "'.$param.'".', 20);
     }
```

```
}
```

Pour faciliter les tests et réduire la duplication, un raccourci peut être utilisé pour indiquer une constante de classe comme un `@expectedExceptionCode` en utilisant la syntaxe "`@expectedExceptionCode ClassName::CONST`". Un exemple peut être trouvé dans la section intitulée « @expectedExceptionCode ».

@expectedExceptionMessageRegExp

Le message d'exception attendu peut aussi être spécifié par une expression régulière en utilisant l'annotation `@expectedExceptionMessageRegExp`. C'est utile pour des situations où une sous-chaine n'est pas adaptée pour correspondre au message donné.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
     /**
      * @expectedException              MyException
      * @expectedExceptionMessageRegExp /Argument \d+ can not be an? \w+/
      */
     public function testExceptionHasRightMessage()
     {
         throw new MyException('Argument 2 can not be an integer');
     }
}
```

@group

Un test peut être marqué comme appartement à un ou plusieurs groupes en utilisant l'annotation `@group` comme ceci

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @group specification
     */
    public function testSomething()
    {
    }

    /**
     * @group regresssion
     * @group bug2204
     */
    public function testSomethingElse()
    {
    }
}
```

Des tests peuvent être sélectionnés pour l'exécution en se basant sur les groupes en utilisant les options `--group` et `--exclude-group` du lanceur de test en ligne de commandes ou en utilisant les directives respectives du fichier de configuration XML.

@large

L'annotation `@large` est un alias pour `@group large`.

Si le paquet `PHP_Invoker` est installé et que le mode strict est activé, un test "large" échouera s'il prend plus de 60 secondes pour s'exécuter. Ce délai est configurable via l'attribut `timeoutForLargeTests` dans le fichier de configuration XML.

@medium

L'annotation `@medium` est un alias pour `@group medium`. Un test "medium" ne doit pas dépendre d'un test marqué comme `@large`.

Si le paquet `PHP_Invoker` est installé et que le mode strict est activé, un test "medium" échouera s'il prend plus de 10 secondes pour s'exécuter. Ce délai est configurable via l'attribut `timeoutForMediumTests` dans le fichier de configuration XML.

@preserveGlobalState

Quand un test est exécuté dans un processus séparé, PHPUnit va tenter de conserver l'état global du processus parent en sérialisant toutes les globales dans le processus parent et en les désérialisant dans le processus enfant. Cela peut poser des problèmes si le processus parent contient des globales qui ne sont pas sérialisable. Pour corriger cela, vous pouvez empêcher PHPUnit de conserver l'état global avec l'annotation `@preserveGlobalState`.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @runInSeparateProcess
     * @preserveGlobalState disabled
     */
    public function testInSeparateProcess()
    {
        // ...
    }
}
```

@requires

L'annotation `@requires` peut être utilisée pour sauter des tests lorsque des pré-requis communs, comme la version de PHP ou des extensions installées, ne sont pas fournis.

Une liste complète des possibilités et des exemples peuvent être trouvés à Tableau 7.3, « Usages possibles de @requires »

@runTestsInSeparateProcesses

Indique que tous les tests d'une classe de tests doivent être executés dans un processus PHP séparé.

```
use PHPUnit\Framework\TestCase;

/**
 * @runTestsInSeparateProcesses
 */
class MyTest extends TestCase
{
    // ...
}
```

Note: Par défaut, PHPUnit va essayer de conserver l'état global depuis le processus parent en sérialisant toutes les globales dans le processus parent et en les désérialisant dans le processus enfant. Cela peut poser des problèmes si le processus parent contient des globales qui ne sont pas sérialisable. Voir la section intitulée « @preserveGlobalState » pour plus d'information sur comment le corriger.

@runInSeparateProcess

Indique qu'un test doit être executé dans un processus PHP séparé.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @runInSeparateProcess
     */
    public function testInSeparateProcess()
    {
        // ...
    }
}
```

Note: Par défaut, PHPUnit va essayer de conserver l'état global depuis le processus parent en sérialisant toutes les globales dans le processus parent et en les désérialisant dans le processus enfant. Cela peut poser des problèmes si le processus parent contient des globales qui ne sont pas sérialisable. Voir la section intitulée « @preserveGlobalState » pour plus d'information sur comment le corriger.

@small

L'annotation `@small` est un alias pour `@group small`. Un test "small" ne doit pas dépendre d'un test marqué comme `@medium` ou `@large`.

Si le paquet `PHP_Invoker` est installé et que le mode strict est activé, un test "small" va échoué s'il prend plus d'1 seconde pour s'executer. Ce délai est configurable via l'attrubut `timeoutForSmallTests` dans le fichier de configuration XML.

Note

Les tests doivent être explicitement annotés par soit `@small`, `@medium`, ou `@large` pour activer les temps limites d'execution.

@test

Comme alternative à préfixer vos noms de méthodes de test avec `test`, vous pouvez utiliser l'annotation `@test` dans le bloc de documentation d'une méthode pour la marquer comme méthode de test.

```
/**
 * @test
 */
public function initialBalanceShouldBe0()
{
    $this->assertEquals(0, $this->ba->getBalance());
}
```

@testdox

@ticket

@uses

L'annotation `@uses` spécifies du code qui sera exécuté par un test, mais qui n'et pas destiné à être couvert par le test. Un bon exemple est un objet-valeur qui est nécessaire pour tester une partie du code.

```
/**
 * @covers BankAccount::deposit
 * @uses   Money
 */
public function testMoneyCanBeDepositedInAccount()
{
    // ...
}
```

Cette annotation est spéciali This annotation is notamment utile en mode de couverture stricte où du code involontairement couvert va faire échouer un test. Voir la section intitulée « Code non-intentionnellement couvert » fpour plus d'informations sur le mode de couverture stricte.

Annexe C. Le fichier de configuration XML

PHPUnit

Les attributs d'un élément `<phpunit>` peuvent être utilisés pour configurer les fonctionnalités du coeur de PHPUnit.

```
<phpunit
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="https://schema.phpunit.de/6.3/phpunit.xsd"
         backupGlobals="true"
         backupStaticAttributes="false"
         <!--bootstrap="/path/to/bootstrap.php"-->
         cacheTokens="false"
         colors="false"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         forceCoversAnnotation="false"
         mapTestClassNameToCoveredClassName="false"
         printerClass="PHPUnit_TextUI_ResultPrinter"
         <!--printerFile="/path/to/ResultPrinter.php"-->
         processIsolation="false"
         stopOnError="false"
         stopOnFailure="false"
         stopOnIncomplete="false"
         stopOnSkipped="false"
         stopOnRisky="false"
         testSuiteLoaderClass="PHPUnit_Runner_StandardTestSuiteLoader"
         <!--testSuiteLoaderFile="/path/to/StandardTestSuiteLoader.php"-->
         timeoutForSmallTests="1"
         timeoutForMediumTests="10"
         timeoutForLargeTests="60"
         verbose="false">
  <!-- ... -->
</phpunit>
```

Le fichier de configuration XML ci-dessus correspond au comportement par défaut du lanceur de tests TextUI documenté dans la section intitulée « Options de la ligne de commandes ».

Des options supplémentaires qui ne sont pas disponibles en tant qu'option de ligne de commandes sont :

`convertErrorsToExceptions`

Par défaut, PHPUnit va installer un gestionnaire d'erreur qui converti les erreurs suivantes en exceptions :

- `E_WARNING`
- `E_NOTICE`
- `E_USER_ERROR`
- `E_USER_WARNING`
- `E_USER_NOTICE`
- `E_STRICT`
- `E_RECOVERABLE_ERROR`

	• E_DEPRECATED • E_USER_DEPRECATED Mettre convertErrorsToExceptions à false désactivera cette fonctionnalité.
convertNoticesToExceptions	Si défini à false, le gestionnaire d'erreurs installé par convertErrorsToExceptions ne convertira pas les erreurs E_NOTICE, E_USER_NOTICE, ou E_STRICT en exceptions.
convertWarningsToExceptions	Si défini à false, le gestionnaire d'erreurs installé par convertErrorsToExceptions ne convertira pas les erreurs E_WARNING ou E_USER_WARNING en exceptions.
forceCoversAnnotation	La couverture de code ne sera enregistrée que pour les tests qui utilisent l'annotation @covers documentée dans la section intitulée « @covers ».
timeoutForLargeTests	Si les limites de temps basées sur la taille du test sont appliquées alors cet attribut défini le délai pour tous les tests marqués comme @large. Si un test ne se termine pas dans le délai configuré, il échouera.
timeoutForMediumTests	Si les limites de temps basées sur la taille du test sont appliquées alors cet attribut défini le délai pour tous les tests marqués comme @medium. Si un test ne se termine pas dans le délai configuré, il échouera.
timeoutForSmallTests	Si les limites de temps basées sur la taille du test sont appliquées alors cet attribut défini le délai pour tous les tests qui ne sont pas marqués comme @medium or @large. Si un test ne se termine pas dans le délai configuré, il échouera.

Série de tests

L'élément <testsuites> et son ou ses enfants <testsuite> peuvent être utilisés pour composer une série de tests à partir des séries de test et des cas de test.

```
<testsuites>
  <testsuite name="My Test Suite">
    <directory>/path/to/*Test.php files</directory>
    <file>/path/to/MyTest.php</file>
    <exclude>/path/to/exclude</exclude>
  </testsuite>
</testsuites>
```

En utilisant les attributs phpVersion et phpVersionOperator, une version requise de PHP peut être indiquée. L'exemple ci-dessous ne va ajouter que les fichiers /path/to/*Test.php et /path/to/MyTest.php si la version de PHP est au moins 5.3.0.

```
<testsuites>
    <testsuite name="My Test Suite">
      <directory suffix="Test.php" phpVersion="5.3.0" phpVersionOperator=">=">/path/to/f
      <file phpVersion="5.3.0" phpVersionOperator=">=">/path/to/MyTest.php</file>
    </testsuite>
  </testsuites>
```

L'attribut `phpVersionOperator` est facultatif et vaut par défaut `>=`.

Groupes

L'élément `<groups>` et ses enfants `<include>`, `<exclude>` et `<group>` peuvent être utilisés pour choisir des groupes de tests marqués avec l'annotation `@group` (documenté dans la section intitulée « @group ») qui doivent (ou ne doivent pas) être exécutés.

```
<groups>
  <include>
    <group>name</group>
  </include>
  <exclude>
    <group>name</group>
  </exclude>
</groups>
```

La configuration XML ci-dessus revient à appeler le lanceur de test TextUI avec les options suivantes:

- `--group name`
- `--exclude-group name`

Inclure des fichiers de la couverture de code

L'élément `<filter>` et ses enfants peuvent être utilisés pour configurer les listes blanches pour les rapports de couverture de code.

```
<filter>
  <whitelist processUncoveredFilesFromWhitelist="true">
    <directory suffix=".php">/path/to/files</directory>
    <file>/path/to/file</file>
    <exclude>
      <directory suffix=".php">/path/to/files</directory>
      <file>/path/to/file</file>
    </exclude>
  </whitelist>
</filter>
```

Journalisation

L'élément `<logging>` et ses enfants `<log>` peuvent être utilisés pour configurer la journalisation de l'exécution des tests.

```
<logging>
  <log type="coverage-html" target="/tmp/report" lowUpperBound="35"
       highLowerBound="70"/>
  <log type="coverage-clover" target="/tmp/coverage.xml"/>
  <log type="coverage-php" target="/tmp/coverage.serialized"/>
  <log type="coverage-text" target="php://stdout" showUncoveredFiles="false"/>
  <log type="junit" target="/tmp/logfile.xml" logIncompleteSkipped="false"/>
  <log type="testdox-html" target="/tmp/testdox.html"/>
  <log type="testdox-text" target="/tmp/testdox.txt"/>
</logging>
```

La configuration XML ci-dessus revient à invoquer le lanceur de tests TextUI avec les options suivantes :

- `--coverage-html /tmp/report`

- `--coverage-clover /tmp/coverage.xml`
- `--coverage-php /tmp/coverage.serialized`
- `--coverage-text`
- `> /tmp/logfile.txt`
- `--log-junit /tmp/logfile.xml`
- `--testdox-html /tmp/testdox.html`
- `--testdox-text /tmp/testdox.txt`

Les attributs `lowUpperBound`, `highLowerBound`, `logIncompleteSkipped` et `showUncoveredFiles` n'ont pas d'options équivalentes pour le lanceur de tests TextUI.

- `lowUpperBound`: pourcentage de couverture maximum considérée comme étant faible.
- `highLowerBound`: pourcentage de couverture minimum considérée comme étant forte.
- `showUncoveredFiles`: Montre tous les fichiers en liste blanche dans la sortie `--coverage-text` et pas seulement ceux possédant des informations de couverture.
- `showOnlySummary`: Montre seulement le résumé dans la sortie `--coverage-text`.

Écouteurs de tests

L'élément `<listeners>` et ses enfants `<listener>` peuvent être utilisés pour brancher des écouteurs de tests additionnels lors de l'exécution des tests.

```
<listeners>
  <listener class="MyListener" file="/optional/path/to/MyListener.php">
    <arguments>
      <array>
        <element key="0">
          <string>Sebastian</string>
        </element>
      </array>
      <integer>22</integer>
      <string>April</string>
      <double>19.78</double>
      <null/>
      <object class="stdClass"/>
    </arguments>
  </listener>
</listeners>
```

La configuration XML ci-dessus revient à brancher l'objet `$listener` (voir ci-dessous) à l'exécution des tests :

```
$listener = new MyListener(
    ['Sebastian'],
    22,
    'April',
    19.78,
    null,
    new stdClass
```

```
);
```

Configurer les réglages de PHP INI, les constantes et les variables globales

L'élément `<php>` et ses enfants peuvent être utilisés pour configurer les réglages PHP, les constantes et les variables globales. Il peut également être utilisé pour préfixer l'`include_path`.

```
<php>
  <includePath>.</includePath>
  <ini name="foo" value="bar"/>
  <const name="foo" value="bar"/>
  <var name="foo" value="bar"/>
  <env name="foo" value="bar"/>
  <post name="foo" value="bar"/>
  <get name="foo" value="bar"/>
  <cookie name="foo" value="bar"/>
  <server name="foo" value="bar"/>
  <files name="foo" value="bar"/>
  <request name="foo" value="bar"/>
</php>
```

La configuration XML ci-dessus correspond au code PHP suivant :

```
ini_set('foo', 'bar');
define('foo', 'bar');
$GLOBALS['foo'] = 'bar';
$_ENV['foo'] = 'bar';
$_POST['foo'] = 'bar';
$_GET['foo'] = 'bar';
$_COOKIE['foo'] = 'bar';
$_SERVER['foo'] = 'bar';
$_FILES['foo'] = 'bar';
$_REQUEST['foo'] = 'bar';
```

Annexe D. Index

Index

Symboles

A

B

C

D

E

F

G

H

I

J

L

M

O

P

R

S

T

V

W

X

Annexe E. Bibliographie

[Astels2003] *Test Driven Development*. David Astels. Copyright © 2003. Prentice Hall. ISBN 0131016490.

[Beck2002] *Test Driven Development by Example*. Kent Beck. Copyright © 2002. Addison-Wesley. ISBN 0-321-14653-0.

[Meszaros2007] *xUnit Test Patterns: Refactoring Test Code*. Gerard Meszaros. Copyright © 2007. Addison-Wesley. ISBN 978-0131495050.

Annexe F. Copyright

```
Copyright (c) 2005-2017 Sebastian Bergmann.

Cette oeuvre est soumise à la licence Creative Commons Attribution 3.0
non transposée.

Un résumé de la licence est donné ci-dessous, suivi de la version
intégrale.

---------------------------------------------------------------------

Vous êtes libre de :

    * partager - reproduire, distribuer et communiquer l'oeuvre
    * adapter - adapter l'oeuvre

Selon les conditions suivantes:

Attribution. Vous devez attribuer l'oeuvre de la manière indiquée par
l'auteur de l'oeuvre ou le titulaire des droits (mais pas d'une manière qui suggérerait
soutiennent ou approuvent votre utilisation de l'oeuvre).

    * A chaque réutilisation ou distribution de cette oeuvre, vous devez faire apparaîtr
    la licence selon laquelle elle est mise à disposition. La meilleure manière de l'ind
    est un lien vers cette page web.

    * N'importe laquelle des conditions ci-dessus peut être waived
      si vous avez l'autorisation du titulaire de droits.

    * Rien dans cette licence ne contrevient ou ne restreint les
      droits moraux de l'auteur.

Votre droit à l'utilisation équitable et vos autres droits ne sont en aucune manière
affectés par ce qui précède.

Ceci est le résumé explicatif "lisible par les humains" du Code Juridique (la version in
la licence) ci-dessous.

=====================================================================

Creative Commons Legal Code
Attribution 3.0 Unported

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE
LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN
ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS
INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO
WARRANTIES REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS
LIABILITY FOR DAMAGES RESULTING FROM ITS USE.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS
CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS
PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE
WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW
IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND
AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS
LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU
```

```
THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF
SUCH TERMS AND CONDITIONS.

1. Definitions

   a. "Adaptation" means a work based upon the Work, or upon the
      Work and other pre-existing works, such as a translation,
      adaptation, derivative work, arrangement of music or other
      alterations of a literary or artistic work, or phonogram or
      performance and includes cinematographic adaptations or any
      other form in which the Work may be recast, transformed, or
      adapted including in any form recognizably derived from the
      original, except that a work that constitutes a Collection
      will not be considered an Adaptation for the purpose of this
      License. For the avoidance of doubt, where the Work is a
      musical work, performance or phonogram, the synchronization of
      the Work in timed-relation with a moving image ("synching")
      will be considered an Adaptation for the purpose of this
      License.

   b. "Collection" means a collection of literary or artistic works,
      such as encyclopedias and anthologies, or performances,
      phonograms or broadcasts, or other works or subject matter
      other than works listed in Section 1(f) below, which, by
      reason of the selection and arrangement of their contents,
      constitute intellectual creations, in which the Work is
      included in its entirety in unmodified form along with one or
      more other contributions, each constituting separate and
      independent works in themselves, which together are assembled
      into a collective whole. A work that constitutes a Collection
      will not be considered an Adaptation (as defined above) for
      the purposes of this License.

   c. "Distribute" means to make available to the public the
      original and copies of the Work or Adaptation, as appropriate,
      through sale or other transfer of ownership.

   d. "Licensor" means the individual, individuals, entity or
      entities that offer(s) the Work under the terms of this License.

   e. "Original Author" means, in the case of a literary or artistic
      work, the individual, individuals, entity or entities who
      created the Work or if no individual or entity can be
      identified, the publisher; and in addition (i) in the case of
      a performance the actors, singers, musicians, dancers, and
      other persons who act, sing, deliver, declaim, play in,
      interpret or otherwise perform literary or artistic works or
      expressions of folklore; (ii) in the case of a phonogram the
      producer being the person or legal entity who first fixes the
      sounds of a performance or other sounds; and, (iii) in the
      case of broadcasts, the organization that transmits the
      broadcast.

   f. "Work" means the literary and/or artistic work offered under
      the terms of this License including without limitation any
      production in the literary, scientific and artistic domain,
      whatever may be the mode or form of its expression including
      digital form, such as a book, pamphlet and other writing; a
      lecture, address, sermon or other work of the same nature; a
      dramatic or dramatico-musical work; a choreographic work or
      entertainment in dumb show; a musical composition with or
      without words; a cinematographic work to which are assimilated
      works expressed by a process analogous to cinematography; a
      work of drawing, painting, architecture, sculpture, engraving
```

or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

g. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

h. "Publicly Perform" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

i. "Reproduce" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

b. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

c. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

d. to Distribute and Publicly Perform Adaptations.

e. For the avoidance of doubt:

```
      i. Non-waivable Compulsory License Schemes. In those
         jurisdictions in which the right to collect royalties
         through any statutory or compulsory licensing scheme cannot
         be waived, the Licensor reserves the exclusive right to
         collect such royalties for any exercise by You of the
         rights granted under this License;

      ii. Waivable Compulsory License Schemes. In those
          jurisdictions in which the right to collect royalties
          through any statutory or compulsory licensing scheme can
          be waived, the Licensor waives the exclusive right to
          collect such royalties for any exercise by You of the
          rights granted under this License; and,

      iii. Voluntary License Schemes. The Licensor waives the right
           to collect royalties, whether individually or, in the
           event that the Licensor is a member of a collecting
           society that administers voluntary licensing schemes, via
           that society, from any exercise by You of the rights
           granted under this License.

The above rights may be exercised in all media and formats whether
now known or hereafter devised. The above rights include the right
to make such modifications as are technically necessary to exercise
the rights in other media and formats. Subject to Section 8(f), all
rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly
   made subject to and limited by the following restrictions:

   a. You may Distribute or Publicly Perform the Work only under the
      terms of this License. You must include a copy of, or the
      Uniform Resource Identifier (URI) for, this License with every
      copy of the Work You Distribute or Publicly Perform. You may
      not offer or impose any terms on the Work that restrict the
      terms of this License or the ability of the recipient of the
      Work to exercise the rights granted to that recipient under
      the terms of the License. You may not sublicense the Work. You
      must keep intact all notices that refer to this License and to
      the disclaimer of warranties with every copy of the Work You
      Distribute or Publicly Perform. When You Distribute or
      Publicly Perform the Work, You may not impose any effective
      technological measures on the Work that restrict the ability
      of a recipient of the Work from You to exercise the rights
      granted to that recipient under the terms of the License. This
      Section 4(a) applies to the Work as incorporated in a
      Collection, but this does not require the Collection apart
      from the Work itself to be made subject to the terms of this
      License. If You create a Collection, upon notice from any
      Licensor You must, to the extent practicable, remove from the
      Collection any credit as required by Section 4(b), as
      requested. If You create an Adaptation, upon notice from any
      Licensor You must, to the extent practicable, remove from the
      Adaptation any credit as required by Section 4(b), as requested.

   b. If You Distribute, or Publicly Perform the Work or any
      Adaptations or Collections, You must, unless a request has
      been made pursuant to Section 4(a), keep intact all copyright
      notices for the Work and provide, reasonable to the medium or
      means You are utilizing: (i) the name of the Original Author
      (or pseudonym, if applicable) if supplied, and/or if the
      Original Author and/or Licensor designate another party or
      parties (e.g., a sponsor institute, publishing entity,
```

journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv), consistent with Section 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4 (b) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

c. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

   a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

   b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

   a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

   b. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

   c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

   d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

   e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

   f. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be

```
      enforced according to the corresponding provisions of the
      implementation of those treaty provisions in the applicable
      national law. If the standard suite of rights granted under
      applicable copyright law includes additional rights not
      granted under this License, such additional rights are deemed
      to be included in the License; this License is not intended to
      restrict the license of any rights under applicable law.

Creative Commons is not a party to this License, and makes no
warranty whatsoever in connection with the Work. Creative Commons
will not be liable to You or any party on any legal theory for any
damages whatsoever, including without limitation any general,
special, incidental or consequential damages arising in connection
to this license. Notwithstanding the foregoing two (2) sentences,
if Creative Commons has expressly identified itself as the Licensor
hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the
Work is licensed under the CCPL, Creative Commons does not authorize
the use by either party of the trademark "Creative Commons" or any
related trademark or logo of Creative Commons without the prior
written consent of Creative Commons. Any permitted use will be in
compliance with Creative Commons' then-current trademark usage
guidelines, as may be published on its website or otherwise made
available upon request from time to time. For the avoidance of
doubt, this trademark restriction does not form part of this
License.

Creative Commons may be contacted at http://creativecommons.org/.

======================================================================
```