atoum Documentation

Version 3.2.0

atoum Team

déc. 20, 2018

Table des matières

CHAPITRE 1

Démarrer avec atoum

Vous devez d'abord *l'installer*, et puis essayez de démarrer votre *premier test*. Mais pour comprendre comment le faire, vous devez de garder à l'esprit de la philosophie d'atoum.

1.1 La philosophie d'atoum

Vous avez besoin d'écrire une classe de test pour chaque classe testé. Lorsque vous voulez tester une valeur, vous devez :

— indiquer le type de cette valeur (entier, décimal, tableau, chaîne de caractères, etc.) ;
— indiquer les contraintes devant s'appliquer à cette valeur (égal à, nulle, contenant quelque chose, etc.).

1.2 Extension

Atoum propose une série d'extension qui peuvent être utilisé. Pour les découvrir, il suffit d'aller sur le site dédié.

CHAPITRE 2

Installation

Si vous souhaitez utiliser atoum, il vous suffit de télécharger la dernière version.

Vous pouvez installer atoum de plusieurs manières :
- à l'aide de composer ;
- en téléchargeant l'*archive PHAR* ;
- en clonant le dépôt *Github* ;
- voir aussi *l'integration d'atoum dans votre framework*.

2.1 Composer

Composer est un outil de gestion de dépendance en PHP.

Assurez-vous que vous disposez d'une installation de composer fonctionnelle

Ajoutez `atoum/atoum` a vos dépendances de développement :

```
composer require --dev atoum/atoum
```

2.2 Archive PHAR

Une archive PHAR (PHp ARchive) est créée automatiquement à chaque modification d'atoum.

PHAR est un format d'archive applicative pour PHP.

2.2.1 Installation

Vous pouvez télécharger la dernière version stable d'atoum directement depuis le site officiel : http://downloads.atoum.org/nightly/atoum.phar

2.2.2 Mise à jour

Pour mettre à jour le PHAR, utiliser simplement la commande :

```
$ php -d phar.readonly=0 atoum.phar --update
```

Note : Le processus de mise à jour modifie l'archive PHAR. Cependant, par défaut la configuration de PHP ne l'autorise pas. Voilà pourquoi il faut utiliser la directive `-d phar.readonly=0`.

Si une version plus récente existe, elle sera alors téléchargée automatiquement et installée au sein de l'archive :

```
$ php -d phar.readonly=0 atoum.phar --update
Checking if a new version is available... Done !
Update to version 'nightly-2416-201402121146'... Done !
Enable version 'nightly-2416-201402121146'... Done !
Atoum was updated to version 'nightly-2416-201402121146' successfully !
```

S'il n'existe pas de version plus récente, atoum s'arrêtera immédiatement :

```
$ php -d phar.readonly=0 atoum.phar --update
Checking if a new version is available... Done !
There is no new version available !
```

atoum ne demande aucune confirmation de la part de l'utilisateur pour réaliser la mise à jour, car il est très facile de revenir à une version précédente.

2.2.3 Lister les versions contenues dans l'archive

Vous pouvez lister les versions disponibles dans les archives en utilisant `--list-available-versions` ou `-lav` :

```
$ php atoum.phar -lav
  nightly-941-201201011548
  nightly-1568-201210311708
* nightly-2416-201402121146
```

La liste des versions de l'archive est affichée. La version actuellement active est précédée par `*`.

2.2.4 Changer la version courante

Pour activer une autre version, il suffit d'utiliser l'argument `--enable-version`, ou `-ev` en version abrégée, suivi du nom de la version à utiliser :

```
$ php -d phar.readonly=0 atoum.phar -ev DEVELOPMENT
```

Note : La modification de la version courante nécessite la modification de l'archive PHAR. Cependant, par défaut la configuration de PHP ne l'autorise pas. Voilà pourquoi il faut utiliser la directive `-d phar.readonly=0`.

2.2.5 Suppression d'anciennes versions

Au cours du temps, l'archive peut contenir plusieurs versions d'atoum qui ne sont plus utilisées.

Pour les supprimer, il suffit d'utiliser l'argument `--delete-version`, ou `-dv` dans sa version abrégée, suivi du nom de la version à supprimer :

```
$ php -d phar.readonly=0 atoum.phar -dv nightly-941-201201011548
```

La version est alors supprimée.

Avertissement : Il n'est pas possible de supprimer la version active.

Note : La suppression d'une version nécessite la modification de l'archive PHAR. par défaut la configuration de PHP ne l'autorise pas. Voilà pourquoi il faut utiliser la directive `-d phar.readonly=0`.

2.3 Github

Si vous souhaitez utiliser atoum directement depuis ses sources, vous pouvez cloner ou « forker » le dépôt github : git://github.com/atoum/atoum.git

CHAPITRE 3

Premiers tests

Vous avez besoin d'écrire une classe de test pour chaque classe testé.

Imaginez que vous vouliez tester la traditionnelle classe `HelloWorld`, alors vous devez créer la classe de test `test\units\HelloWorld`.

Avertissement : Si vous débutez avec atoum, il est recommandé d'installer le paquet ´atoum-stubs <https://packagist.org/packages/atoum/stubs>´_. Celui-ci vous fournira de l'auto-complétion au sein de votre IDE.

Note : atoum utilise les espaces de noms. Par exemple, pour tester la classe `Vendor\Project\HelloWorld`, vous devez créer la classe `Vendor\Project\tests\units\HelloWorld`.

Voici le code de la classe `HelloWorld` que nous allons tester.

```php
<?php
# src/Vendor/Project/HelloWorld.php

namespace Vendor\Project;

class HelloWorld
{
    public function getHiAtoum ()
    {
        return 'Hi atoum !';
    }
}
```

Maintenant, voici le code de la classe de test que nous pourrions écrire.

```php
<?php
# src/Vendor/Project/tests/units/HelloWorld.php
```

(suite sur la page suivante)

(suite de la page précédente)

```php
// La classe de test a son propre namespace :
// Le namespace de la classe à tester + "tests\units"
namespace Vendor\Project\tests\units;

// Vous devez inclure la classe testée (si vous n'avez pas d'autoloader)
require_once __DIR__ . '/../../HelloWorld.php';

use atoum;

/*
 * Classe de test pour Vendor\Project\HelloWorld
 *
 * Remarquez qu'elle porte le même nom que la classe à tester
 * et qu'elle dérive de la classe atoum
 */
class HelloWorld extends atoum
{
    /*
     * Cette méthode est dédiée à la méthode getHiAtoum()
     */
    public function testGetHiAtoum ()
    {
        $this
            // création d'une nouvelle instance de la classe à tester
            ->given($this->newTestedInstance)

            ->then

                    // nous testons que la méthode getHiAtoum retourne bien
                    // une chaîne de caractère...
                    ->string($this->testedInstance->getHiAtoum())
                        // ... et que la chaîne est bien celle attendue,
                        // c'est-à-dire 'Hi atoum !'
                        ->isEqualTo('Hi atoum !')
        ;
    }
}
```

Maintenant, lançons nos tests. Vous devriez voir quelque chose comme ça :

```
$ ./vendor/bin/atoum -f src/Vendor/Project/tests/units/HelloWorld.php
> PHP path: /usr/bin/php
> PHP version:
=> PHP 5.6.3 (cli) (built: Nov 13 2014 18:31:57)
=> Copyright (c) 1997-2014 The PHP Group
=> Zend Engine v2.6.0, Copyright (c) 1998-2014 Zend Technologies
> Vendor\Project\tests\units\HelloWorld...
[S_____________________________________________________________][1/1]
=> Test duration: 0.00 second.
=> Memory usage: 0.25 Mb.
> Total test duration: 0.00 second.
> Total test memory usage: 0.25 Mb.
> Running duration: 0.04 second.
Success (1 test, 1/1 method, 0 void method, 0 skipped method, 2 assertions)!
```

Nous venons de tester que la méthode `getHiAtoum` :

— retourne une *chaîne de caractère* ;

— que *c'est égale à* `"Hi atoum !"`.

Les tests sont passés, tout est au vert. Voilà, votre code est solide comme un roc grâce à atoum !

3.1 Dissection du test

Il est important que vous compreniez chaque chose que nous utilisons dans ce test. Regardons chaque partie.

Nous utilisons l'espace de noms `Vendor\Project\tests\units` où `Vendor\Project` est l'espace de noms de la classe et `tests\units` la partie de l'espace de noms utiliser par atoum pour comprendre que nous sommes dans l'espace de nom de test. Cette espace de nom est configurable et ceci est expliqué dans la *section appropriée*. Ensuite, à l'intérieur de la méthode testée, nous utilisons une synthaxe spécial *given et then*. Ils ne font rien d'autre que rendre le test plus lisible. Finalement, nous utilisons un autre truc simple *newTestedInstance et testedInstance* pour obtenir une instance de la classe testée.

CHAPITRE 4

Lancement des tests

4.1 Exécutable

atoum dispose d'un exécutable qui vous permet de lancer vos tests en ligne de commande.

4.1.1 Avec l'archive phar

Si vous utiliser l'archive phar, elle est déjà exécutable.

linux / mac

```
$ php path/to/atoum.phar
```

windows

```
C:\> X:\Path\To\php.exe X:\Path\To\atoum.phar
```

4.1.2 Avec les sources

Si vous utilisez les sources, l'exécutable se trouve dans path/to/atoum/bin.

linux / mac

```
$ php path/to/bin/atoum

# OU #

$ ./path/to/bin/atoum
```

windows

```
C:\> X:\Path\To\php.exe X:\Path\To\bin\atoum\bin
```

4.1.3 Exemples dans le reste de la documentation

Dans les exemples suivants, les commandes pour lancer les tests avec atoum seront écrites avec la forme suivante :

```
$ ./bin/atoum
```

C'est exactement la commande que vous pourriez utiliser si vous avez *Composer* sous Linux.

4.2 Fichiers à exécuter

4.2.1 For specific files

Pour lancer les tests d'un fichier, il vous suffit d'utiliser l'option -f ou –files.

```
$ ./bin/atoum -f tests/units/MyTest.php
```

4.2.2 Pour un dossier

Pour lancer les tests d'un répertoire, il vous suffit d'utiliser l'option -d ou –directories.

```
$ ./bin/atoum -d tests/units
```

Vous trouverez d'autres arguments dans la section approprié lié à la *ligne de commande*.

4.3 Filtres

Une fois que vous avez préciser à atoum les *fichiers à exécuter*, vous pouvez filtrer ce qui sera réellement exécuter.

4.3.1 Par espace de noms

Pour filtrer sur les espace de nom, par example exécuter le test seulement sur un espace de nom, il suffit d'utiliser l'option `-ns` or `--namespaces`.

```
$ ./bin/atoum -d tests/units -ns mageekguy\\atoum\\tests\\units\\asserters
```

Note : Il est important de doubler chaque backslash pour éviter qu'ils soient interprétés par le shell.

4.3.2 Une classe ou une méthode

Pour filtrer sur une classe ou une méthode, c'est-à-dire exécuter seulement des tests d'une classe ou une méthode, il suffit d'utiliser l'option `-m` ou `--methods`.

```
$ ./bin/atoum -d tests/units -m␣
↪mageekguy\\atoum\\tests\\units\\asserters\\string::testContains
```

Note : Il est important de doubler chaque backslash pour éviter qu'ils soient interprétés par le shell.

Vous pouvez remplacer le nom de la classe ou de la méthode par `*` pour signifier `tous`.

```
$ ./bin/atoum -d tests/units -m mageekguy\\atoum\\tests\\units\\asserters\\string::*
```

En utilisant « * » au lieu d'un nom de classe signifie que vous pouvez filtrer par nom de la méthode.

```
$ ./bin/atoum -d tests/units -m *::testContains
```

4.3.3 Tags

Tout comme de nombreux outils, dont Behat, atoum vous permet de taguer vos tests unitaires et de n'exécuter que ceux ayant un ou plusieurs tags spécifiques.

Pour cela, il faut commencer par définir un ou plusieurs tags pour une ou plusieurs classes de tests unitaires.

Cela se fait très simplement grâce aux annotations et à la balise @tags :

```php
<?php

namespace vendor\project\tests\units;

require_once __DIR__ . '/atoum.phar';

use mageekguy\atoum;

/**
 * @tags thisIsOneTag thisIsTwoTag thisIsThreeTag
 */
class foo extends atoum\test
{
    public function testBar()
    {
        // ...
    }
}
```

De la même manière, il est également possible de taguer les méthodes de test.

Note : Les tags définis au niveau d'une méthode prennent le pas sur ceux définis au niveau de la classe.

```php
<?php

namespace vendor\project\tests\units;

require_once __DIR__ . '/atoum.phar';

use mageekguy\atoum;

class foo extends atoum\test
{
    /**
     * @tags thisIsOneMethodTag thisIsTwoMethodTag thisIsThreeMethodTag
     */
    public function testBar()
    {
        // ...
    }
}
```

Une fois les tags nécessaires définis, il n'y a plus qu'à exécuter les tests avec le ou les tags adéquates à l'aide de l'option `--tags`, ou `-t` dans sa version courte :

```
$ ./bin/atoum -d tests/units -t thisIsOneTag
```

Attention, cette instruction n'a de sens que s'il y a une ou plusieurs classes de tests unitaires et qu'au moins l'une d'entre elles porte le tag spécifié. Dans le cas contraire, aucun test ne sera exécuté.

Il est possible de définir plusieurs tags :

```
$ ./bin/atoum -d tests/units -t thisIsOneTag thisIsThreeTag
```

Dans ce dernier cas, les classes de tests ayant été tagués soit avec thisIsOneTag, soit avec thisIsThreeTag, seront les seules à être exécutées.

CHAPITRE 5

Comment écrire des scénarios de test

Après avoir créer votre *premier test* et compris *comment le lancer<lancement-des-tests>*, vous voulez probablement écrire de meilleurs tests. Dans cette section, vous trouverez comment ajouter du sucre syntaxique pour vous aider dans l'écriture de vos tests et ce de manière simple.

Il est possible d'écrire des tests unitaires avec atoum de plusieurs manières, et l'une d'elles est d'utiliser des mots-clefs tels que `given`, `if`, `and` ou bien encore `then`, `when` ou `assert` qui permettent de mieux organiser et de rendre plus lisibles les tests.

5.1 `given`, `if`, `and` et `then`

L'utilisation de ces mots-clefs est très intuitive :

```
<?php
$this
    ->given($computer = new computer())
    ->if($computer->prepare())
    ->and(
        $computer->setFirstOperand(2),
        $computer->setSecondOperand(2)
    )
    ->then
        ->object($computer->add())
            ->isIdenticalTo($computer)
        ->integer($computer->getResult())
            ->isEqualTo(4)
;
```

Il est important de noter que ces mots-clés n'ont pas un autre but que de donner aux tests une forme plus lisible. Il ne serve aucun but technique. Le seul but est d'aider le lecteur, les humains ou plus précisément le développeur, à comprendre ce qui se passe dans le test.

Ainsi, `given`, `if` et `and` permettent de définir les conditions préalables pour que les assertions qui suivent le mot-clef `then` passent avec succès.

Cependant, il n'y a aucune règle ou grammaire qui régissent la syntaxe de l'ordre de ces mots-clés pour atoum.

Ainsi, le développeur utilisera ces mots-clés à bon escient pour rendre les tests aussi lisibles que possible. Cependant, si elle est utilisé de façon incorrecte, vous pourriez finir avec des tests comme suit :

```
<?php
$this
    ->and($computer = new computer())
    ->if($computer->setFirstOperand(2))
    ->then
    ->given($computer->setSecondOperand(2))
        ->object($computer->add())
            ->isIdenticalTo($computer)
        ->integer($computer->getResult())
            ->isEqualTo(4)
;
```

Pour les mêmes raisons, l'utilisation de `then` est facultative.

Il est également important de noter qu'il est tout à fait possible d'écrire le même test en n'utilisant aucun mot-clef :

```
<?php
$computer = new computer();
$computer->setFirstOperand(2);
$computer->setSecondOperand(2);

$this
    ->object($computer->add())
        ->isIdenticalTo($computer)
    ->integer($computer->getResult())
        ->isEqualTo(4)
;
```

Le test ne sera pas plus lent ou plus rapide à exécuter, et il n'y a aucun avantage à utiliser une notation plutôt qu'une autre, l'important est d'en choisir une et de s'y tenir. De cette façon, cela permet de facilité la maintenance des tests (le problème est exactement le même que les conventions de codage).

5.2 when

En plus de `given`, `if`, `and` et `then`, il existe également d'autres mots-clefs.

L'un d'entre eux est `when`. Il dispose d'une fonctionnalité spécifique introduite pour contourner le fait qu'il est illégal d'écrire en PHP le code suivant :

```
<?php # ignore
$this
    ->if($array = array(uniqid()))
    ->and(unset($array[0]))
    ->then
        ->sizeOf($array)
            ->isZero()
;
```

Le langage génère en effet dans ce cas l'erreur fatale : `Parse error: syntax error, unexpected 'unset' (T_UNSET), expecting ')'`

Il est en effet impossible d'utiliser `unset()` comme argument d'une fonction.

Pour résoudre ce problème, le mot-clef `when` est capable d'interpréter l'éventuelle fonction anonyme qui lui est passée en argument, ce qui permet d'écrire le test précédent de la manière suivante :

```
<?php
$this
    ->if($array = array(uniqid()))
    ->when(
        function() use ($array) {
            unset($array[0]);
        }
    )
    ->then
      ->sizeOf($array)
        ->isZero()
;
```

Bien évidemment, si `when` ne reçoit pas de fonction anonyme en argument, il se comporte exactement comme `given`, `if`, `and` et `then`, à savoir qu'il ne fait absolument rien fonctionnellement parlant.

5.3 assert

Enfin, il existe le mot-clef `assert` qui a également un fonctionnement un peu particulier.

Pour illustrer son fonctionnement, le test suivant va être utilisé :

```
<?php
$this
    ->given($foo = new \mock\foo())
    ->and($bar = new bar($foo))
    ->if($bar->doSomething())
    ->then
        ->mock($foo)
            ->call('doOtherThing')
                ->once()

    ->if($bar->setValue(uniqid()))
    ->then
        ->mock($foo)
            ->call('doOtherThing')
                ->exactly(2)
;
```

Le test précédent présente un inconvénient en terme de maintenance, car si le développeur a besoin d'intercaler un ou plusieurs nouveaux appels à bar : :doOtherThing() entre les deux appels déjà effectués, il sera obligé de mettre à jour en conséquence la valeur de l'argument passé à exactly(). Pour remédier à ce problème, vous pouvez remettre à zéro un mock de 2 manières différentes :

— soit en utilisant $mock->getMockController()->resetCalls() ;
— soit en utilisant $this->resetMock($mock).

```
<?php
$this
    ->given($foo = new \mock\foo())
    ->and($bar = new bar($foo))
    ->if($bar->doSomething())
    ->then
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ->mock($foo)
            ->call('doOtherThing')
                ->once()

    // 1ère manière
    ->given($foo->getMockController()->resetCalls())
    ->if($bar->setValue(uniqid()))
    ->then
        ->mock($foo)
            ->call('doOtherThing')
                ->once()

    // 2ème manière
    ->given($this->resetMock($foo))
    ->if($bar->setValue(uniqid()))
    ->then
        ->mock($foo)
            ->call('doOtherThing')
                ->once()
;
```

Ces méthodes effacent la mémoire du contrôleur, il est donc possible d'écrire l'assertion suivante comme si le bouchon n'avait jamais été utilisé.

Le mot-clef `assert` permet de se passer de l'appel explicite à `resetCalls()` ou `resetMock` et de plus il provoque l'effacement de la mémoire de l'ensemble des adaptateurs et des contrôleurs de mock définis au moment de son utilisation.

Grâce à lui, il est donc possible d'écrire le test précédent d'une façon plus simple et plus lisible, d'autant qu'il est possible de passer une chaîne de caractère à assert afin d'expliquer le rôle des assertions suivantes :

```
<?php
$this
    ->assert("Bar n'a pas de valeur")
        ->given($foo = new \mock\foo())
        ->and($bar = new bar($foo))
        ->if($bar->doSomething())
        ->then
            ->mock($foo)
                ->call('doOtherThing')
                    ->once()

    ->assert('Bar a une valeur')
        ->if($bar->setValue(uniqid()))
        ->then
            ->mock($foo)
                ->call('doOtherThing')
                    ->once()
;
```

La chaîne de caractères sera de plus reprise dans les messages générés par atoum si l'une des assertions ne passe pas avec succès.

5.4 newTestedInstance & testedInstance

Lorsque l'on effectue des tests, il faut bien souvent créer une nouvelle instance de la classe et passer celle-ci dans divers paramètres. Une aide à l'écriture est disponible pour ce cas précis, il s'agit de `newTestedInstance` et de `testedInstance`

Voici un exemple :

```
namespace jubianchi\atoum\preview\tests\units;

use atoum;
use jubianchi\atoum\preview\foo as testedClass;

class foo extends atoum
{
    public function testBar()
    {
        $this
            ->if($foo = new testedClass())
            ->then
                ->object($foo->bar())->isIdenticalTo($foo)
        ;
    }
}
```

Ceci peut être simplifié avec la nouvelle syntaxe :

```
namespace jubianchi\atoum\preview\tests\units;

use atoum;

class foo extends atoum
{
    public function testBar()
    {
        $this
            ->if($this->newTestedInstance)
            ->then
                ->object($this->testedInstance->bar())
                    ->isTestedInstance()
        ;
    }
}
```

Comme on le voit, c'est légèrement plus simple, mais surtout cela présente deux avantages :

— On ne manipule pas le nom de la classe testée
— On ne manipule pas l'instance ainsi créée

Par ailleurs, on peut facilement valider que l'on a bien l'instance testée avec *isTestedInstance*, comme expliqué dans l'exemple précédent.

Pour passer des arguments au constructeur, il suffit de le faire au travers de `newTestedInstance` :

```
$this->newTestedInstance($argument1, $argument2);
```

Si vous voulez tester une méthode statique de votre classe, vous pouvez récupérer la classe testée avec cette syntaxe :

```
namespace jubianchi\atoum\preview\tests\units;

use atoum;

class foo extends atoum
{
    public function testBar()
    {
      $this
        ->if($class = $this->testedClass->getClass())
        ->then
          ->object($class::bar())
       ;
    }
 }
```

5.4.1 Accès aux constantes de la classe testée

Si vous avez besoin d'accéder aux constantes de la classe testée, vous pouvez y accéder de deux façons :

```
<?php

namespace
{
    class Foo
    {
        const A = 'a';
    }
}

namespace tests\units
{
    class Foo extends \atoum\test
    {
        public function testFoo()
        {
            $this
                ->given($this->newTestedInstance())
                ->then
                    ->string($this->getTestedClassName()::A)->isEqualTo('a')
                    ->string($this->testedInstance::A)->isEqualTo('a')
            ;
        }
    }
}
```

Avertissement : Vous avez besoin d'initialiser l'instance avec `newTestedInstance`, pour avoir accès aux constantes.

5.5 testedClass

Comme `testedInstance`, vous pouvez utiliser `testedClass` pour écrire des tests plus compréhensible. `testedClass` permet d'écrire des assertions dynamiques sur les classes testées :

```
<?php
$this
        ->testedClass
    ->hasConstant('FOO')
                ->isFinal()
;
```

Vous pouvez aller plus loin avec *les assertions de classe*.

CHAPITRE 6

Listes des asserters

Pour écrire des tests plus explicites et moins verbeux, atoum fourni plusieurs asserters qui donnent accès a des assertions spécifiques aux types testés.

atoum possède différents asserters spécialisés permettant de manipuler différents éléments, les assertes héritent d'autres qu'ils spécialisent. Ceci permettant d'aider à garder une consistance entre les différents asserters et force à utiliser les même noms d'assertion.

Voici l'arbre d'héritage des asserters :

```
-- asserter (abstract)
    |-- error
    |-- mock
    |-- stream
    |-- variable
    |   |-- array
    |   |    `-- castToArray
    |   |-- boolean
    |   |-- class
    |   |    `-- testedClass
    |   |-- integer
    |   |   |-- float
    |   |   `-- sizeOf
    |   |-- object
    |   |   |-- dateInterval
    |   |   |-- dateTime
    |   |   |   `-- mysqlDateTime
    |   |   |-- exception
    |   |   `-- iterator
    |   |       `-- generator
    |   |-- resource
    |   `-- string
    |       |-- castToString
    |       |-- hash
    |       |-- output
```

(suite sur la page suivante)

(suite de la page précédente)

```
    |       `-- utf8String
    `-- function
```

Note : La syntaxe générale des asserters/assertions est : `$this->[asserter]($value)->[assertion];`

Note : La plupart des assertions sont fluent, comme vous le verrez ci-dessous.

Note : A la fin du chapitre, vous trouverez plusieurs *trucs & astuces* relatif aux assertions et asserters, pensez à le lire !

6.1 afterDestructionOf

C'est l'assertion dédiée à la destruction des objets.

Cette assertion ne fait que prendre un objet, vérifier que la méthode `__destruct()` est bien définie puis l'appelle.

Si `__destruct()` existe bien et si son appel se passe sans erreur ni exception, alors le test passe.

```
<?php
$this
    ->afterDestructionOf($objectWithDestructor)      // passe
    ->afterDestructionOf($objectWithoutDestructor)   // échoue
;
```

6.2 array

C'est l'assertion dédiée aux tableaux.

Note : `array` étant un mot réservé en PHP, il n'a pas été possible de créer une assertion `array`. Elle s'appelle donc `phpArray` et un alias `array` a été créé. Vous pourrez donc rencontrer des `->phpArray()` ou des `->array()`.

Il est conseillé d'utiliser exclusivement `->array()` afin de simplifier la lecture des tests.

6.2.1 Sucre syntaxique

Il est à noter, qu'afin de simplifier l'écriture des tests sur les tableaux, du sucre syntaxique est disponible. Celui-ci permet d'effectuer diverses assertions directement sur les clefs du tableau testé.

```
$a = [
   'foo' => 42,
   'bar' => '1337'
];

$this
```

(suite sur la page suivante)

(suite de la page précédente)

```
   ->array($a)
      ->integer['foo']->isEqualTo(42)
      ->string['bar']->isEqualTo('1337')
;
```

Note : This writing form is available from PHP 5.4.

6.2.2 child

Avec `child` vous pouvez faire une assertion sur un sous tableau.

```
<?php
$array = array(
   'ary' => array(
      'key1' => 'abc',
      'key2' => 123,
      'key3' => array(),
   ),
);

$this
   ->array($array)
      ->child['ary'](function($child)
      {
         $child
            ->hasSize(3)
            ->hasKeys(array('key1', 'key2', 'key3'))
            ->contains(123)
            ->child['key3'](function($child)
            {
               $child->isEmpty;
            });
      });
```

Note : Cette forme d'écriture n'est disponible qu'à partir de PHP 5.4.

6.2.3 contains

`contains` vérifie qu'un tableau contient une certaine donnée.

```
<?php
$fibonacci = array('1', 2, '3', 5, '8', 13, '21');

$this
    ->array($fibonacci)
        ->contains('1')      // passe
        ->contains(1)        // passe, ne vérifie pas...
        ->contains('2')      // ... le type de la donnée
        ->contains(10)       // échoue
;
```

Note : `contains` ne fait pas de recherche récursive.

Avertissement :

`contains` ne teste pas le type de la donnée.
Si vous souhaitez vérifier également son type, utilisez *strictlyContains*.

6.2.4 containsValues

`containsValues` vérifie qu'un tableau contient toutes les données fournies dans un tableau.

```
<?php
$fibonacci = array('1', 2, '3', 5, '8', 13, '21');

$this
    ->array($array)
        ->containsValues(array(1, 2, 3))        // passe
        ->containsValues(array('5', '8', '13')) // passe
        ->containsValues(array(0, 1, 2))        // échoue
;
```

Note : `containsValues` ne fait pas de recherche récursive.

Avertissement :

`containsValues` ne teste pas le type des données.
Si vous souhaitez vérifier également leurs types, utilisez *strictlyContainsValues*.

6.2.5 hasKey

`hasKey` vérifie qu'un tableau contient une certaine clef.

```
<?php
$fibonacci = array('1', 2, '3', 5, '8', 13, '21');
$atoum     = array(
    'name'        => 'atoum',
    'owner'       => 'mageekguy',
);

$this
    ->array($fibonacci)
        ->hasKey(0)          // passe
        ->hasKey(1)          // passe
        ->hasKey('1')        // passe
        ->hasKey(10)         // échoue

    ->array($atoum)
        ->hasKey('name')     // passe
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ->hasKey('price')    // échoue
;
```

Note : `hasKey` ne fait pas de recherche récursive.

Avertissement : `hasKey` ne teste pas le type des clefs.

6.2.6 hasKeys

`hasKeys` vérifie qu'un tableau contient toutes les clefs fournies dans un tableau.

```
<?php
$fibonacci = array('1', 2, '3', 5, '8', 13, '21');
$atoum     = array(
    'name'         => 'atoum',
    'owner'        => 'mageekguy',
);

$this
    ->array($fibonacci)
        ->hasKeys(array(0, 2, 4))           // passe
        ->hasKeys(array('0', 2))            // passe
        ->hasKeys(array('4', 0, 3))         // passe
        ->hasKeys(array(0, 3, 10))          // échoue

    ->array($atoum)
        ->hasKeys(array('name', 'owner'))   // passe
        ->hasKeys(array('name', 'price'))   // échoue
;
```

Note : `hasKeys` ne fait pas de recherche récursive.

Avertissement : `hasKey` ne teste pas le type des clefs.

6.2.7 hasSize

`hasSize` vérifie la taille d'un tableau.

```
<?php
$fibonacci = array('1', 2, '3', 5, '8', 13, '21');

$this
    ->array($fibonacci)
        ->hasSize(7)        // passe
        ->hasSize(10)       // échoue
;
```

Note : `hasSize` n'est pas récursif.

6.2.8 isEmpty

`isEmpty` vérifie qu'un tableau est vide.

```
<?php
$emptyArray    = array();
$nonEmptyArray = array(null, null);

$this
    ->array($emptyArray)
        ->isEmpty()          // passe

    ->array($nonEmptyArray)
        ->isEmpty()          // échoue
;
```

6.2.9 isEqualTo

Voir aussi :

`isEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isEqualTo*

6.2.10 isIdenticalTo

Voir aussi :

`isIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isIdenticalTo*

6.2.11 isNotEmpty

`isNotEmpty` vérifie qu'un tableau n'est pas vide.

```
<?php
$emptyArray    = array();
$nonEmptyArray = array(null, null);

$this
    ->array($emptyArray)
        ->isNotEmpty()       // échoue

    ->array($nonEmptyArray)
        ->isNotEmpty()       // passe
;
```

6.2.12 isNotEqualTo

Voir aussi :

`isNotEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotEqualTo*

6.2.13 isNotIdenticalTo

Voir aussi :

`isNotIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotIdenticalTo*

6.2.14 keys

`keys` vous permet de récupérer un asserter de type *array* contenant les clefs du tableau testé.

```
<?php
$atoum = array(
    'name'  => 'atoum',
    'owner' => 'mageekguy',
);

$this
    ->array($atoum)
        ->keys
            ->isEqualTo(
                array(
                    'name',
                    'owner',
                )
            )
;
```

6.2.15 notContains

`notContains` vérifie qu'un tableau ne contient pas une donnée.

```
<?php
$fibonacci = array('1', 2, '3', 5, '8', 13, '21');

$this
    ->array($fibonacci)
        ->notContains(null)          // passe
        ->notContains(1)             // échoue
        ->notContains(10)            // passe
;
```

Note : `notContains` ne fait pas de recherche récursive.

Avertissement :

`contains` ne teste pas le type de la donnée.
Si vous souhaitez vérifier également son type, utilisez *strictlyNotContains*.

6.2.16 notContainsValues

`notContainsValues` vérifie qu'un tableau ne contient aucune des données fournies dans un tableau.

```
<?php
$fibonacci = array('1', 2, '3', 5, '8', 13, '21');

$this
    ->array($array)
        ->notContainsValues(array(1, 4, 10))     // échoue
        ->notContainsValues(array(4, 10, 34))    // passe
        ->notContainsValues(array(1, '2', 3))    // échoue
;
```

Note : `notContainsValues` ne fait pas de recherche récursive.

Avertissement :

`notContainsValues` ne teste pas le type des données.
Si vous souhaitez vérifier également leurs types, utilisez *strictlyNotContainsValues*.

6.2.17 notHasKey

`notHasKey` vérifie qu'un tableau ne contient pas une certaine clef.

```
<?php
$fibonacci = array('1', 2, '3', 5, '8', 13, '21');
$atoum     = array(
    'name'  => 'atoum',
    'owner' => 'mageekguy',
);

$this
    ->array($fibonacci)
        ->notHasKey(0)           // échoue
        ->notHasKey(1)           // échoue
        ->notHasKey('1')         // échoue
        ->notHasKey(10)          // passe

    ->array($atoum)
        ->notHasKey('name')      // échoue
        ->notHasKey('price')     // passe
;
```

Note : `notHasKey` ne fait pas de recherche récursive.

Avertissement : `notHasKey` ne teste pas le type des clefs.

6.2.18 notHasKeys

`notHasKeys` vérifie qu'un tableau ne contient aucune des clefs fournies dans un tableau.

```
<?php
$fibonacci = array('1', 2, '3', 5, '8', 13, '21');
$atoum     = array(
    'name'         => 'atoum',
    'owner'        => 'mageekguy',
);

$this
    ->array($fibonacci)
        ->notHasKeys(array(0, 2, 4))            // échoue
        ->notHasKeys(array('0', 2))             // échoue
        ->notHasKeys(array('4', 0, 3))          // échoue
        ->notHasKeys(array(10, 11, 12))         // passe

    ->array($atoum)
        ->notHasKeys(array('name', 'owner'))    // échoue
        ->notHasKeys(array('foo', 'price'))     // passe
;
```

Note : `notHasKeys` ne fait pas de recherche récursive.

Avertissement : `notHasKey` ne teste pas le type des clefs.

6.2.19 size

`size` vous permet de récupérer un asserter de type *integer* contenant la taille du tableau testé.

```
<?php
$fibonacci = array('1', 2, '3', 5, '8', 13, '21');

$this
    ->array($fibonacci)
        ->size
            ->isGreaterThan(5)
;
```

6.2.20 strictlyContains

`strictlyContains` vérifie qu'un tableau contient une certaine donnée (même valeur et même type).

```php
<?php
$fibonacci = array('1', 2, '3', 5, '8', 13, '21');

$this
    ->array($fibonacci)
        ->strictlyContains('1')      // passe
        ->strictlyContains(1)        // échoue
        ->strictlyContains('2')      // échoue
        ->strictlyContains(2)        // passe
        ->strictlyContains(10)       // échoue
;
```

Note : `strictlyContains` ne fait pas de recherche récursive.

Avertissement :

`strictlyContains` teste le type de la donnée.
Si vous ne souhaitez pas vérifier son type, utilisez *contains*.

6.2.21 strictlyContainsValues

`strictlyContainsValues` vérifie qu'un tableau contient toutes les données fournies dans un tableau (même valeur et même type).

```php
<?php
$fibonacci = array('1', 2, '3', 5, '8', 13, '21');

$this
    ->array($array)
        ->strictlyContainsValues(array('1', 2, '3'))     // passe
        ->strictlyContainsValues(array(1, 2, 3))         // échoue
        ->strictlyContainsValues(array(5, '8', 13))      // passe
        ->strictlyContainsValues(array('5', '8', '13')) // échoue
        ->strictlyContainsValues(array(0, '1', 2))       // échoue
;
```

Note : `strictlyContainsValues` ne fait pas de recherche récursive.

Avertissement :

`strictlyContainsValues` teste le type des données.
Si vous ne souhaitez pas vérifier leurs types, utilisez *containsValues*.

6.2.22 strictlyNotContains

`strictlyNotContains` vérifie qu'un tableau ne contient pas une donnée (même valeur et même type).

```php
<?php
$fibonacci = array('1', 2, '3', 5, '8', 13, '21');

$this
    ->array($fibonacci)
        ->strictlyNotContains(null)          // passe
        ->strictlyNotContains('1')           // échoue
        ->strictlyNotContains(1)             // passe
        ->strictlyNotContains(10)            // passe
;
```

Note : `strictlyNotContains` ne fait pas de recherche récursive.

Avertissement :

`strictlyNotContains` teste le type de la donnée.
Si vous ne souhaitez pas vérifier son type, utilisez *notContains*.

6.2.23 strictlyNotContainsValues

`strictlyNotContainsValues` vérifie qu'un tableau ne contient aucune des données fournies dans un tableau (même valeur et même type).

```php
<?php
$fibonacci = array('1', 2, '3', 5, '8', 13, '21');

$this
    ->array($array)
        ->strictlyNotContainsValues(array('1', 4, 10))  // échoue
        ->strictlyNotContainsValues(array(1, 4, 10))    // passe
        ->strictlyNotContainsValues(array(4, 10, 34))   // passe
        ->strictlyNotContainsValues(array('1', 2, '3')) // échoue
        ->strictlyNotContainsValues(array(1, '2', 3))   // passe
;
```

Note : `strictlyNotContainsValues` ne fait pas de recherche récursive.

Avertissement :

`strictlyNotContainsValues` teste le type des données.
Si vous ne souhaitez pas vérifier leurs types, utilisez *notContainsValues*.

6.2.24 values

`keys` vous permet de récupérer un asserter de type *array* contenant les clefs du tableau testé.

Exemple :

```php
<?php
$this
    ->given($arr = [0 => 'foo', 2 => 'bar', 3 => 'baz'])
    ->then
      ->array($arr)->values
        ->string[0]->isEqualTo('foo')
        ->string[1]->isEqualTo('bar')
        ->string[2]->isEqualTo('baz')
;
```

Nouveau dans la version 2.9.0 : assertion values ajouté

6.3 boolean

C'est l'assertion dédiée aux booléens.

Si vous essayez de tester une variable qui n'est pas un booléen avec cette assertion, cela échouera.

Note : `null` n'est pas un booléen. Reportez-vous au manuel de PHP pour savoir ce que is_bool considère ou non comme un booléen.

6.3.1 isEqualTo

Voir aussi :

`isEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isEqualTo*

6.3.2 isFalse

`isFalse` vérifie que le booléen est strictement égal à `false`.

```php
<?php
$true  = true;
$false = false;

$this
    ->boolean($true)
        ->isFalse()      // échoue

    ->boolean($false)
        ->isFalse()      // passe
;
```

6.3.3 isIdenticalTo

Voir aussi :

`isIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isIdenticalTo*

6.3.4 isNotEqualTo

Voir aussi :

`isNotEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotEqualTo*

6.3.5 isNotIdenticalTo

Voir aussi :

`isNotIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotIdenticalTo*

6.3.6 isTrue

`isTrue` vérifie que le booléen est strictement égal à `true`.

```
<?php
$true  = true;
$false = false;

$this
    ->boolean($true)
        ->isTrue()      // passe

    ->boolean($false)
        ->isTrue()      // échoue
;
```

6.4 castToArray

C'est l'assertion dédiée aux tests sur le transtypage d'objets en tableaux.

```
<?php
class AtoumVersions {
    private $versions = ['1.0', '2.0', '2.1'];

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

$this
    ->castToArray(new AtoumVersions())
        ->contains('1.0')
;
```

Voir aussi :

L'asserter `castToArray` retourne une instance de l'asserter `array`. Vous pouvez donc utiliser toutes les assertions de l'asserter *array*

6.5 castToString

C'est l'assertion dédiée aux tests sur le transtypage d'objets en chaîne de caractères.

```
<?php
class AtoumVersion {
    private $version = '1.0';

    public function __toString() {
        return 'atoum v' . $this->version;
    }
}

$this
    ->castToString(new AtoumVersion())
        ->isEqualTo('atoum v1.0')
;
```

6.5.1 contains

Voir aussi :

`contains` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :contains*

6.5.2 notContains

Voir aussi :

`notContains` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :notContains*

6.5.3 hasLength

Voir aussi :

`hasLength` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :hasLength*

6.5.4 hasLengthGreaterThan

Voir aussi :

`hasLengthGreaterThan` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :hasLengthGreaterThan*

6.5.5 hasLengthLessThan

Voir aussi :

`hasLengthLessThan` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :hasLengthLessThan*

6.5.6 isEmpty

Voir aussi :

`isEmpty` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :isEmpty*

6.5.7 isEqualTo

Voir aussi :

`isEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isEqualTo*

6.5.8 isEqualToContentsOfFile

Voir aussi :

`isEqualToContentsOfFile` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :isEqualToContentsOfFile*

6.5.9 isIdenticalTo

Voir aussi :

`isIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isIdenticalTo*

6.5.10 isNotEmpty

Voir aussi :

`isNotEmpty` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :isNotEmpty*

6.5.11 isNotEqualTo

Voir aussi :

`isNotEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotEqualTo*

6.5.12 isNotIdenticalTo

Voir aussi :

`isNotIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotIdenticalTo*

6.5.13 matches

Voir aussi :

`matches` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :matches*

6.6 class

C'est l'assertion dédiée aux classes.

```
<?php
$object = new \StdClass;

$this
    ->class(get_class($object))

    ->class('\StdClass')
;
```

Note : Le mot-clef `class` étant réservé en PHP, il n'a pas été possible de créer une assertion `class`. Elle s'appelle donc `phpClass` et un alias `class` a été créé. Vous pourrez donc rencontrer des `->phpClass()` ou des `->class()`.

Il est conseillé d'utiliser exclusivement `->class()`.

6.6.1 hasConstant

`hasConstant` vérifie que la classe possède bien la constante testée.

```
<?php
$this
    ->class('\StdClass')
        ->hasConstant('FOO')        // échoue

    ->class('\FilesystemIterator')
        ->hasConstant('CURRENT_AS_PATHNAME')        // passe
;
```

6.6.2 hasInterface

`hasInterface` vérifie que la classe implémente une interface donnée.

```
<?php
$this
    ->class('\ArrayIterator')
        ->hasInterface('Countable')      // passe

    ->class('\StdClass')
        ->hasInterface('Countable')      // échoue
;
```

6.6.3 hasMethod

`hasMethod` vérifie que la classe contient une méthode donnée.

```
<?php
$this
    ->class('\ArrayIterator')
        ->hasMethod('count')     // passe

    ->class('\StdClass')
        ->hasMethod('count')     // échoue
;
```

6.6.4 hasNoParent

`hasNoParent` vérifie que la classe n'hérite d'aucune classe.

```
<?php
$this
    ->class('\StdClass')
        ->hasNoParent()      // passe

    ->class('\FilesystemIterator')
        ->hasNoParent()      // échoue
;
```

Avertissement :

Une classe peut implémenter une ou plusieurs interfaces et n'hériter d'aucune classe.
`hasNoParent` ne vérifie pas les interfaces, uniquement les classes héritées.

6.6.5 hasParent

`hasParent` vérifie que la classe hérite bien d'une classe.

```
<?php
$this
    ->class('\StdClass')
        ->hasParent()        // échoue

    ->class('\FilesystemIterator')
        ->hasParent()        // passe
;
```

Avertissement :

Une classe peut implémenter une ou plusieurs interfaces et n'hériter d'aucune classe.
`hasParent` ne vérifie pas les interfaces, uniquement les classes héritées.

6.6.6 isAbstract

`isAbstract` vérifie que la classe est abstraite.

```
<?php
$this
    ->class('\StdClass')
        ->isAbstract()        // échoue
;
```

6.6.7 isFinal

`isFinal` vérifie que la classe est finale.

Dans le cas suivant, nous testons une classe non final (`StdClass`) :

```
<?php
$this
        ->class('\StdClass')
                ->isFinal()             // échoue
;
```

Dans le cas suivant, la classe testée est une classe final

```
<?php
$this
        ->testedClass
                ->isFinal()             // passe
;

$this
        ->testedClass
                ->isFinal               // passe aussi
;
```

6.6.8 isSubclassOf

`isSubclassOf` vérifie que la classe hérite de la classe donnée.

```
<?php
$this
    ->class('\FilesystemIterator')
        ->isSubclassOf('\DirectoryIterator')    // passe
        ->isSubclassOf('\SplFileInfo')          // passe
        ->isSubclassOf('\StdClass')             // échoue
;
```

6.7 dateInterval

C'est l'assertion dédiée à l'objet DateInterval.

Si vous essayez de tester une variable qui n'est pas un objet `DateInterval` (ou une classe qui l'étend) avec cette assertion, cela échouera.

6.7.1 isCloneOf

Voir aussi :

`isCloneOf` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isCloneOf*

6.7.2 isEqualTo

`isEqualTo` vérifie que la durée de l'objet `DateInterval` est égale à la durée d'un autre objet `DateInterval`.

```
<?php
$di = new DateInterval('P1D');

$this
    ->dateInterval($di)
        ->isEqualTo(                // passe
            new DateInterval('P1D')
        )
        ->isEqualTo(                // échoue
            new DateInterval('P2D')
        )
;
```

6.7.3 isGreaterThan

`isGreaterThan` vérifie que la durée de l'objet `DateInterval` est supérieure à la durée d'un autre objet `DateInterval`.

```
<?php
$di = new DateInterval('P2D');

$this
    ->dateInterval($di)
        ->isGreaterThan(            // passe
            new DateInterval('P1D')
        )
        ->isGreaterThan(            // échoue
            new DateInterval('P2D')
        )
;
```

6.7.4 isGreaterThanOrEqualTo

`isGreaterThanOrEqualTo` vérifie que la durée de l'objet `DateInterval` est supérieure ou égale à la durée d'un autre objet `DateInterval`.

```
<?php
$di = new DateInterval('P2D');

$this
    ->dateInterval($di)
        ->isGreaterThanOrEqualTo(   // passe
```

(suite sur la page suivante)

(suite de la page précédente)

```
            new DateInterval('P1D')
        )
        ->isGreaterThanOrEqualTo(   // passe
            new DateInterval('P2D')
        )
        ->isGreaterThanOrEqualTo(   // échoue
            new DateInterval('P3D')
        )
;
```

6.7.5 isIdenticalTo

Voir aussi :

`isIdenticalTo` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isIdenticalTo*

6.7.6 isInstanceOf

Voir aussi :

`isInstanceOf` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isInstanceOf*

6.7.7 isLessThan

`isLessThan` vérifie que la durée de l'objet `DateInterval` est inférieure à la durée d'un autre objet `DateInterval`.

```
<?php
$di = new DateInterval('P1D');

$this
    ->dateInterval($di)
        ->isLessThan(               // passe
            new DateInterval('P2D')
        )
        ->isLessThan(               // échoue
            new DateInterval('P1D')
        )
;
```

6.7.8 isLessThanOrEqualTo

`isLessThanOrEqualTo` vérifie que la durée de l'objet `DateInterval` est inférieure ou égale à la durée d'un autre objet `DateInterval`.

```
<?php
$di = new DateInterval('P2D');

$this
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ->dateInterval($di)
        ->isLessThanOrEqualTo(      // passe
            new DateInterval('P3D')
        )
        ->isLessThanOrEqualTo(      // passe
            new DateInterval('P2D')
        )
        ->isLessThanOrEqualTo(      // échoue
            new DateInterval('P1D')
        )
;
```

6.7.9 isNotEqualTo

Voir aussi :

`isNotEqualTo` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isNotEqualTo*

6.7.10 isNotIdenticalTo

Voir aussi :

`isNotIdenticalTo` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isNotIdenticalTo*

6.7.11 isZero

`isZero` vérifie que la durée de l'objet `DateInterval` est égale à 0.

```
<?php
$di1 = new DateInterval('P0D');
$di2 = new DateInterval('P1D');

$this
    ->dateInterval($di1)
        ->isZero()      // passe
    ->dateInterval($di2)
        ->isZero()      // échoue
;
```

6.8 dateTime

C'est l'assertion dédiée à l'objet DateTime.

Si vous essayez de tester une variable qui n'est pas un objet `DateTime` (ou une classe qui l'étend) avec cette assertion, cela échouera.

6.8.1 hasDate

`hasDate` vérifie la partie date de l'objet `DateTime`.

```
<?php
$dt = new DateTime('1981-02-13');

$this
    ->dateTime($dt)
        ->hasDate('1981', '02', '13')   // passe
        ->hasDate('1981', '2',  '13')   // passe
        ->hasDate(1981,   2,    13)     // passe
;
```

6.8.2 hasDateAndTime

`hasDateAndTime` vérifie la partie date et heure de l'objet `DateTime`.

```
<?php
$dt = new DateTime('1981-02-13 01:02:03');

$this
    ->dateTime($dt)
        // passe
        ->hasDateAndTime('1981', '02', '13', '01', '02', '03')
        // passe
        ->hasDateAndTime('1981', '2',  '13', '1',  '2',  '3')
        // passe
        ->hasDateAndTime(1981,   2,    13,   1,    2,    3)
;
```

6.8.3 hasDay

`hasDay` vérifie le jour de l'objet `DateTime`.

```
<?php
$dt = new DateTime('1981-02-13');

$this
    ->dateTime($dt)
        ->hasDay(13)        // passe
;
```

6.8.4 hasHours

`hasHours` vérifie l'heure de l'objet `DateTime`.

```
<?php
$dt = new DateTime('01:02:03');

$this
    ->dateTime($dt)
        ->hasHours('01')    // passe
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ->hasHours('1')      // passe
        ->hasHours(1)        // passe
;
```

6.8.5 hasMinutes

`hasMinutes` vérifie les minutes de l'objet `DateTime`.

```
<?php
$dt = new DateTime('01:02:03');

$this
    ->dateTime($dt)
        ->hasMinutes('02')   // passe
        ->hasMinutes('2')    // passe
        ->hasMinutes(2)      // passe
;
```

6.8.6 hasMonth

`hasMonth` vérifie le mois de l'objet `DateTime`.

```
<?php
$dt = new DateTime('1981-02-13');

$this
    ->dateTime($dt)
        ->hasMonth(2)        // passe
;
```

6.8.7 hasSeconds

`hasSeconds` vérifie les secondes de l'objet `DateTime`.

```
<?php
$dt = new DateTime('01:02:03');

$this
    ->dateTime($dt)
        ->hasSeconds('03')     // passe
        ->hasSeconds('3')      // passe
        ->hasSeconds(3)        // passe
;
```

6.8.8 hasTime

`hasTime` vérifie le temps de l'objet `DateTime`.

```
<?php
$dt = new DateTime('01:02:03');

$this
    ->dateTime($dt)
        ->hasTime('01', '02', '03')      // passe
        ->hasTime('1',  '2',  '3')       // passe
        ->hasTime(1,    2,    3)         // passe
;
```

6.8.9 hasTimezone

`hasTimezone` vérifie le fuseau horaire de l'objet `DateTime`.

```
<?php
$dt = new DateTime();

$this
    ->dateTime($dt)
        ->hasTimezone('Europe/Paris')
;
```

6.8.10 hasYear

`hasYear` vérifie l'année de l'objet `DateTime`.

```
<?php
$dt = new DateTime('1981-02-13');

$this
    ->dateTime($dt)
        ->hasYear(1981)      // passe
;
```

6.8.11 isCloneOf

Voir aussi :

`isCloneOf` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isCloneOf*

6.8.12 isEqualTo

Voir aussi :

`isEqualTo` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isEqualTo*

6.8.13 isIdenticalTo

Voir aussi :

`isIdenticalTo` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isIdenticalTo*

6.8.14 isImmutable

`isImmutable` vérifie que qu'un objet `DateTime` est immuable.

```
<?php
$dt = new DateTime('1981-02-13');

$this
    ->dateTime($dt)
        ->isImmutable(1981)      // rate
;

$dt = new DateTimeImmutable('1981-02-13');

$this
    ->dateTime($dt)
        ->isImmutable(1981)      // réussit
;
```

6.8.15 isInstanceOf

Voir aussi :

`isInstanceOf` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isInstanceOf*

6.8.16 isNotEqualTo

Voir aussi :

`isNotEqualTo` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isNotEqualTo*

6.8.17 isNotIdenticalTo

Voir aussi :

`isNotIdenticalTo` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isNotIdenticalTo*

6.9 error

C'est l'assertion dédiée aux erreurs.

```
<?php
$this
    ->when(
        function() {
            trigger_error('message');
        }
    )
        ->error()
            ->exists() // or notExists
;
```

Note : La syntaxe utilise les fonctions anonymes (aussi appelées fermetures ou closures) introduites en PHP 5.3. Pour plus de précision, lisez la documentation PHP sur les fonctions anonymes.

Avertissement : Les types d'erreur E_ERROR, E_PARSE, E_CORE_ERROR, E_CORE_WARNING, E_COMPILE_ERROR, E_COMPILE_WARNING ainsi que la plupart des E_STRICT ne peuvent pas être gérés avec cette fonction.

6.9.1 exists

`exists` vérifie qu'une erreur a été levée lors de l'exécution du code précédent.

```
<?php
$this
    ->when(
        function() {
            trigger_error('message');
        }
    )
        ->error()
            ->exists()      // pass

    ->when(
        function() {
            // code without error
        }
    )
        ->error()
            ->exists()      // failed
;
```

6.9.2 notExists

`notExists` vérifie qu'aucune erreur n'a été levée lors de l'exécution du code précédent.

```
<?php
$this
    ->when(
        function() {
            trigger_error('message');
```

(suite sur la page suivante)

(suite de la page précédente)

```
        }
    )
        ->error()
            ->notExists()    // fails

    ->when(
        function() {
            // code without error
        }
    )
        ->error()
            ->notExists()    // pass
;
```

6.9.3 withType

`withType` vérifie le type de l'erreur levée.

```
<?php
$this
    ->when(
        function() {
            trigger_error('message');
        }
    )
    ->error()
        ->withType(E_USER_NOTICE)    // pass
        ->exists()

    ->when(
        function() {
            trigger_error('message');
        }
    )
    ->error()
        ->withType(E_USER_WARNING)  // failed
        ->exists()
;
```

6.9.4 withAnyType

`withAnyType` ne vérifie pas le type de l'erreur. C'est le comportement par défaut de l'asserter. Donc `->error()->withAnyType()->exists()` est l'équivalent de `->error()->exists()`. Cette méthode existe pour ajouter de la sémantique dans vos tests.

```
<?php
$this
    ->when(
        function() {
            trigger_error('message');
        }
    )
    ->error()
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ->withAnyType() // pass
        ->exists()
    ->when(
        function() {
        }
    )
    ->error()
        ->withAnyType()
        ->exists() // fails
;
```

6.9.5 withMessage

`withMessage` vérifie le contenu du message de l'erreur levée.

```
<?php
$this
    ->when(
        function() {
            trigger_error('message');
        }
    )
    ->error()
        ->withMessage('message')
        ->exists() // passes
;

$this
    ->when(
        function() {
            trigger_error('message');
        }
    )
    ->error()
        ->withMessage('MESSAGE')
        ->exists() // fails
;
```

6.9.6 withAnyMessage

`withAnyMessage` ne vérifie pas le message de l'erreur. C'est le comportement par défaut de l'asserter. Donc `->error()->withAnyMessage()->exists()` est l'équivalent de `->error()->exists()`. Cette méthode existe pour ajouter de la sémantique dans vos tests.

```
<?php
$this
    ->when(
        function() {
            trigger_error();
        }
    )
    ->error()
        ->withAnyMessage()
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ->exists() // passes
;

$this
    ->when(
        function() {
            trigger_error('message');
        }
    )
    ->error()
        ->withAnyMessage()
        ->exists() // passes
;

$this
    ->when(
        function() {
        }
    )
    ->error()
        ->withAnyMessage()
        ->exists() // fails
;
```

6.9.7 withPattern

`withPattern` vérifie le contenu du message de l'erreur soulevée par l'expression régulière.

```
<?php
$this
    ->when(
        function() {
            trigger_error('message');
        }
    )
    ->error()
        ->withPattern('/^mess.*$/')
        ->exists() // passes
;

$this
    ->when(
        function() {
            trigger_error('message');
        }
    )
    ->error()
        ->withPattern('/^mess$/')
        ->exists() // fails
;
```

6.10 exception

C'est l'assertion dédiée aux exceptions.

```
<?php
$this
    ->exception(
        function() use($myObject) {
            // ce code lève une exception: throw new \Exception;
            $myObject->doOneThing('wrongParameter');
        }
    )
;
```

Note : La syntaxe utilise les fonctions anonymes (aussi appelées fermetures ou closures) introduites en PHP 5.3. Pour plus de précision, lisez la documentation PHP sur les fonctions anonymes.

Nous pouvons même facilement récupérer la dernière exception via `$this->exception`.

```
<?php
$this
    ->exception(
        function() use($myObject) {
            // ce code lève une exception: throw new \Exception('erreur', 42);
            $myObject->doOneThing('wrongParameter');
        }
    )->isIdenticalTo($this->exception) // passes
;

$this->exception->hasCode(42); // passes
$this->exception->hasMessage('erreur'); // passes
```

Note : Avant atoum 3.0.0, si vous avez besoin d'effectuer des assertions, vous aviez besoin d'ajouter `atoum\test $test` en argument de la closure. Après la version 3.0.0, vous pouvez simplement utiliser $this à l'intérieur de la closure, afin d'effectuer des assertions.

6.10.1 hasCode

`hasCode` vérifie le code de l'exception.

```
<?php
$this
    ->exception(
        function() use($myObject) {
            // ce code lève une exception: throw new \Exception('erreur', 42);
            $myObject->doOneThing('wrongParameter');
        }
    )
        ->hasCode(42)
;
```

6.10.2 hasDefaultCode

`hasDefaultCode` vérifie que le code de l'exception est la valeur par défaut, c'est-à-dire 0.

```
<?php
$this
    ->exception(
        function() use($myObject) {
            // ce code lève une exception: throw new \Exception;
            $myObject->doOneThing('wrongParameter');
        }
    )
        ->hasDefaultCode()
;
```

Note : `hasDefaultCode` is equivalent to `hasCode(0)`.

6.10.3 hasMessage

`hasMessage` vérifie le message de l'exception.

```
<?php
$this
    ->exception(
        function() use($myObject) {
            // ce code lève une exception: throw new \Exception('Message');
            $myObject->doOneThing('wrongParameter');
        }
    )
        ->hasMessage('Message')     // passes
        ->hasMessage('message')     // fails
;
```

6.10.4 hasNestedException

`hasNestedException` vérifie que l'exception contient une référence vers l'exception précédente. Si le type d'exception est fournie, cela va aussi vérifier la classe de l'exception.

```
<?php
$this
    ->exception(
        function() use($myObject) {
            // ce code lève une exception: throw new \Exception('Message');
            $myObject->doOneThing('wrongParameter');
        }
    )
        ->hasNestedException()      // fails

    ->exception(
        function() use($myObject) {
            try {
                // ce code lève une exception: throw new \FirstException('Message 1',␣
↪42);
```

(suite sur la page suivante)

(suite de la page précédente)

```
                $myObject->doOneThing('wrongParameter');
            }
            // ... l'exception est attrapée...
            catch(\FirstException $e) {
                // ... puis relancée, encapsulée dans une seconde exception
                throw new \SecondException('Message 2', 24, $e);
            }
        }
    )
        ->isInstanceOf('\FirstException')           // fails
        ->isInstanceOf('\SecondException')          // passes

        ->hasNestedException()                      // passes
        ->hasNestedException(new \FirstException)   // passes
        ->hasNestedException(new \SecondException)  // fails
;
```

6.10.5 isCloneOf

Voir aussi :

`isCloneOf` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isCloneOf*

6.10.6 isEqualTo

Voir aussi :

`isEqualTo` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isEqualTo*

6.10.7 isIdenticalTo

Voir aussi :

`isIdenticalTo` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isIdenticalTo*

6.10.8 isInstanceOf

Voir aussi :

`isInstanceOf` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isInstanceOf*

6.10.9 isNotEqualTo

Voir aussi :

`isNotEqualTo` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isNotEqualTo*

6.10.10 isNotIdenticalTo

Voir aussi :

`isNotIdenticalTo` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isNotIdenticalTo*

6.10.11 message

`message` vous permet de récupérer un asserter de type *string* contenant le message de l'exception testée.

```
<?php
$this
    ->exception(
        function() {
            throw new \Exception('My custom message to test');
        }
    )
        ->message
            ->contains('message')
;
```

6.11 extension

C'est l'assertion dédiée aux extensions PHP.

6.11.1 isLoaded

Vérifie si l'extension est chargée (installée et activée).

```
<?php
$this
        ->extension('json')
                ->isLoaded()
        ;
```

Note : Si vous avez besoin de lancer un test uniquement si une extension est présente, vous pouvez utiliser *l'annotation PHP*.

6.12 float

C'est l'assertion dédiée aux nombres décimaux.

Si vous essayez de tester une variable qui n'est pas un nombre décimal avec cette assertion, cela échouera.

Note : `null` n'est pas un nombre décimal. Reportez-vous au manuel de PHP pour savoir ce que is_float considère ou non comme un nombre décimal.

6.12.1 isEqualTo

Voir aussi :

`isEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isEqualTo*

6.12.2 isGreaterThan

Voir aussi :

`isGreaterThan` est une méthode héritée de l'asserter `integer`. Pour plus d'informations, reportez-vous à la documentation de *integer : :isGreaterThan*

6.12.3 isGreaterThanOrEqualTo

Voir aussi :

`isGreaterThanOrEqualTo` est une méthode héritée de l'asserter `integer`. Pour plus d'informations, reportez-vous à la documentation de *integer : :isGreaterThanOrEqualTo*

6.12.4 isIdenticalTo

Voir aussi :

`isIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isIdenticalTo*

6.12.5 isLessThan

Voir aussi :

`isLessThan` est une méthode héritée de l'asserter `integer`. Pour plus d'informations, reportez-vous à la documentation de *integer : :isLessThan*

6.12.6 isLessThanOrEqualTo

Voir aussi :

`isLessThanOrEqualTo` est une méthode héritée de l'asserter `integer`. Pour plus d'informations, reportez-vous à la documentation de *integer : :isLessThanOrEqualoo*

6.12.7 isNearlyEqualTo

`isNearlyEqualTo` vérifie que le nombre décimal est approximativement égal à la valeur qu'elle reçoit en argument.

En effet, en informatique, les nombres décimaux sont gérées d'une façon qui ne permet pas d'effectuer des comparaisons précises sans recourir à des outils spécialisés. Essayez par exemple d'exécuter la commande suivante :

```
$ php -r 'var_dump(1 - 0.97 === 0.03);'
bool(false)
```

Le résultat devrait pourtant être `true`.

Note : Pour avoir plus d'informations sur ce phénomène, lisez la documentation PHP sur le type float et sa précision.

Cette méthode cherche donc à minorer ce problème.

```
<?php
$float = 1 - 0.97;

$this
    ->float($float)
        ->isNearlyEqualTo(0.03) // passe
        ->isEqualTo(0.03)       // échoue
;
```

Note : Pour avoir plus d'informations sur l'algorithme utilisé, consultez le floating point guide.

6.12.8 isNotEqualTo

Voir aussi :

`isNotEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotEqualTo*

6.12.9 isNotIdenticalTo

Voir aussi :

`isNotIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotIdenticalTo*

6.12.10 isZero

Voir aussi :

`isZero` est une méthode héritée de l'asserter `integer`. Pour plus d'informations, reportez-vous à la documentation de *integer : :isZero*

6.13 function

C'est l'asserter dédié aux *fonctions natives* qui sont mockée.

6.13.1 wasCalled

`wasCalled` vérifie que la fonction mockée a été appelée.

6.13.2 wasCalledWithArguments

`wasCalledWithArguments` permet de vérifier les arguments utilisé avec l'appel de la fonction.

6.13.3 wasCalledWithIdenticalArguments

`wasCalledWithIdenticalArguments` permet de vérifier tous les arguments utilisé avec l'appel de la fonction.

6.13.4 wasCalledWithoutAnyArgument

`wasCalledWithoutAnyArgument` vérifie que l'appel a la fonction a été effectué sans aucun argument.

6.13.5 Compter les appels

Si vous voulez vous pouvez effectuer une assertion supplémentaire en comptant le nombre d'appel à la fonction.

```
<?php
$this->function->error_log = true;

$this
   ->if($this->newTestedInstance())
   ->and($this->testedInstance->methodWithAnErrorLog($notExcepted = uniqid()))
   ->then
       ->function('error_log')
           ->wasCalledWithArguments('Value ' . $notExcepted . ' is not excepted here')
               ->once();
```

Ici, nous vérifions que notre fonction a été appelée une seule fois, avec les arguments fournis.

after

`after` vérifie que la fonction a été appelée après la méthode passée en paramètre.

Voir aussi :

`after` a le même comportement que celui existant sur l'asserter des `mock`. Pour plus d'informations, reportez-vous à la documentation de *mock : :after*

atLeastOnce

`atLeastOnce` vérifie que la fonction a été appelée au moins une fois.

Voir aussi :

`atLeastOnce` a le même comportement que celui de l'asserter des `mock`. Pour plus d'informations, reportez-vous à la documentation de *mock : :atLeastOnce*

before

`after` vérifie que la fonction a été appelée avant la méthode passée en paramètre.

Voir aussi :

`before` is the same as the one on the `mock` asserter. For more information, refer to the documentation of *mock : :before*

exactly

`exactly` check that the mocked function has been called a specific number of times.

Voir aussi :

`exactly` is the same as the one on the `mock` asserter. For more information, refer to the documentation of *mock : :exactly*

never

`never` check that the mocked function has never been called.

Voir aussi :

`never` is the same as the one on the `mock` asserter. For more information, refer to the documentation of *mock : :never*

once/twice/thrice

This asserters check that the mocked function has been called exactly :
- une fois (once)
- deux fois (twice)
- trois fois (thrice)

Voir aussi :

`once` is the same as the one on the `mock` asserter. For more information, refer to the documentation of *mock : :once/twice/thrice*

6.14 generator

Il s'agit de l'asserter dédié aux generators.

L'asserter generator hérite l'asserter `iterator`, vous pouvez donc utiliser toutes les assertions de celui-ci.

Exemple :

```
<?php
$generator = function() {
    for ($i=0; $i<3; $i++) {
        yield ($i+1);
    }
};
$this
    ->generator($generator())
        ->hasSize(3)
;
```

Dans cet exemple, nous créons un générateur qui générera 3 valeurs, et nous vérifierons que la taille de ce qui est généré est de 3.

6.14.1 yields

`yields` est utilisé pour facilité les tests sur les valeurs générée par le générateur. Chaque fois que `->yields` est appelé, la valeur suivante du générateur est récupérée. Vous pouvez ensuite utiliser tout les asserter sur cette valeur (par exemple `class`, `string` ou `variable`).

Exemple :

```
<?php
$generator = function() {
    for ($i=0; $i<3; $i++) {
        yield ($i+1);
    }
};
$this
    ->generator($generator())
        ->yields->variable->isEqualTo(1)
        ->yields->variable->isEqualTo(2)
        ->yields->integer->isEqualTo(3)
;
```

Dans cette exemple nous créer un générateur qui produit 3 valeurs : 1, 2 et 3. Ensuite nous produisons chaque valeurs et effectuons une assertion sur celle-ci pour vérifier le type et la valeur. Dans les deux premières valeurs produite, nous utilisons l'asserter `variable` et nous ne vérifions que la valeur. Avec la troisième valeur produite, nous vérifions qu'il s'agit bien d'un entier (toute asserter peut-être utiliser sur cette valeur) avant de vérifier la valeur.

6.14.2 returns

Note : Cette assertion ne fonctionne que avec PHP >= 7.0.

Depuis la version 7.0 de PHP, les générateurs peuvent retourner une valeur via un appel à la méthode `->getReturn()`. Lorsque vous appeler `->returns` sur le l'asserter generator, atoum va renvoyé la valeur via la méthode `->getReturn()``sur l'asserter. Ensuite vous pourrez utiliser n'importe quel autre asserter sur cette valeur comme avec l'assertion ``yields.

Exemple :

```
<?php
$generator = function() {
    for ($i=0; $i<3; $i++) {
        yield ($i+1);
    }
    return 42;
};
$this
    ->generator($generator())
        ->yields->variable->isEqualTo(1)
        ->yields->variable->isEqualTo(2)
        ->yields->integer->isEqualTo(3)
        ->returns->integer->isEqualTo(42)
;
```

Dans cet exemple, nous effectuons quelques vérifications sur toutes les valeurs produites. On vérifie ensuite que le générateur renvoie un entier avec une valeur de 42 (tout comme un appel à l'assertion yields, vous pouvez utiliser n'importe quel asserter pour vérifier la valeur retournée).

Nouveau dans la version 3.0.0 : Asserter generator ajouté

6.15 hash

C'est l'assertion dédiée aux tests sur les hashs (empreintes numériques).

6.15.1 contains

Voir aussi :

`contains` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :contains*

6.15.2 isEqualTo

Voir aussi :

`isEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isEqualTo*

6.15.3 isEqualToContentsOfFile

Voir aussi :

`isEqualToContentsOfFile` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :isEqualToContentsOfFile*

6.15.4 isIdenticalTo

Voir aussi :

`isIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isIdenticalTo*

6.15.5 isMd5

`isMd5` vérifie que la chaîne de caractère est du format `md5`, par exemple une chaîne de caractère d'une longueur de 32.

```
<?php
$hash    = hash('md5', 'atoum');
$notHash = 'atoum';

$this
    ->hash($hash)
        ->isMd5()        // passe
    ->hash($notHash)
        ->isMd5()        // échoue
;
```

6.15.6 isNotEqualTo

Voir aussi :

`isNotEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotEqualTo*

6.15.7 isNotIdenticalTo

Voir aussi :

`isNotIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotIdenticalTo*

6.15.8 isSha1

`isSha1` vérifie que la chaîne de caractère est du format `sha1`, par exemple une chaîne de caractère d'une longueur de 40.

```
<?php
$hash    = hash('sha1', 'atoum');
$notHash = 'atoum';

$this
    ->hash($hash)
        ->isSha1()       // passe
    ->hash($notHash)
        ->isSha1()       // échoue
;
```

6.15.9 isSha256

`isSha256` vérifie que la chaîne de caractère est du format `sha256`, par exemple une chaîne de caractère d'une longueur de 64 caractères.

```
<?php
$hash    = hash('sha256', 'atoum');
$notHash = 'atoum';

$this
    ->hash($hash)
        ->isSha256()     // passe
    ->hash($notHash)
        ->isSha256()     // échoue
;
```

6.15.10 isSha512

`isSha512` vérifie que la chaîne de caractère est du format `sha512`, c'est-à-dire, une chaîne de caractère d'une longueur de 128 caractères.

```php
<?php
$hash    = hash('sha512', 'atoum');
$notHash = 'atoum';

$this
    ->hash($hash)
        ->isSha512()    // passe
    ->hash($notHash)
        ->isSha512()    // échoue
;
```

6.15.11 notContains

Voir aussi :

`notContains` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :notContains*

6.16 integer

C'est l'assertion dédiée aux entiers.

Si vous essayez de tester une variable qui n'est pas un entier avec cette assertion, cela échouera.

Note : `null` n'est pas un entier. Reportez-vous au manuel de PHP pour savoir ce que is_int considère ou non comme un entier.

6.16.1 isEqualTo

Voir aussi :

`isEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isEqualTo*

6.16.2 isGreaterThan

`isGreaterThan` vérifie que l'entier est strictement supérieur à une certaine donnée.

```php
<?php
$zero = 0;

$this
    ->integer($zero)
        ->isGreaterThan(-1)      // passe
        ->isGreaterThan('-1')    // échoue car "-1"
                                 // n'est pas un entier
        ->isGreaterThan(0)       // échoue
;
```

6.16.3 isGreaterThanOrEqualTo

`isGreaterThanOrEqualTo` vérifie que l'entier est supérieur ou égal à une certaine donnée.

```
<?php
$zero = 0;

$this
    ->integer($zero)
        ->isGreaterThanOrEqualTo(-1)     // passe
        ->isGreaterThanOrEqualTo(0)      // passe
        ->isGreaterThanOrEqualTo('-1')   // échoue, car "-1"
                                         // n'est pas un entier
;
```

6.16.4 isIdenticalTo

Voir aussi :

`isIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isIdenticalTo*

6.16.5 isLessThan

`isLessThan` vérifie que l'entier est strictement inférieur à une certaine donnée.

```
<?php
$zero = 0;

$this
    ->integer($zero)
        ->isLessThan(10)     // passe
        ->isLessThan('10')   // échoue, car "10" n'est pas un entier
        ->isLessThan(0)      // échoue
;
```

6.16.6 isLessThanOrEqualTo

`isLessThanOrEqualTo` vérifie que l'entier est inférieur ou égal à une certaine donnée.

```
<?php
$zero = 0;

$this
    ->integer($zero)
        ->isLessThanOrEqualTo(10)        // passe
        ->isLessThanOrEqualTo(0)         // passe
        ->isLessThanOrEqualTo('10')      // échoue car "10"
                                         // n'est pas un entier
;
```

6.16.7 isNotEqualTo

Voir aussi :

`isNotEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotEqualTo*

6.16.8 isNotIdenticalTo

Voir aussi :

`isNotIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotIdenticalTo*

6.16.9 isZero

`isZero` vérifie que l'entier est égal à 0.

```
<?php
$zero    = 0;
$notZero = -1;

$this
    ->integer($zero)
        ->isZero()          // passe

    ->integer($notZero)
        ->isZero()          // échoue
;
```

Note : `isZero` est équivalent à `isEqualTo(0)`.

6.17 mock

C'est l'assertion dédiée aux mocks.

```
<?php
$mock = new \mock\MyClass;

$this
    ->mock($mock)
;
```

Note : Reportez-vous à la documentation sur *les bouchons (mock)* pour obtenir plus d'informations sur la façon de créer et gérer les bouchons.

6.17.1 call

`call` permet de spécifier une méthode du mock à tester, son appel doit être suivi d'un appel à une méthode de vérification d'appel comme *atLeastOnce*, *once/twice/thrice*, *exactly*, etc...

```
<?php

$this
    ->given($mock = new \mock\MyFirstClass)
    ->and($object = new MySecondClass($mock))

    ->if($object->methodThatCallMyMethod())  // Cela va appeler la méthode myMethod␣
↪de $mock
    ->then

    ->mock($mock)
        ->call('myMethod')
            ->once()
;
```

after

`after` vérifie que la méthode a été appelée après la méthode passée en paramètre.

```
<?php
$this
    ->when($mock = new \mock\example)
    ->if(
        $mock->test2(),
        $mock->test()
    )
    ->mock($mock)
    ->call('test')
        ->after($this->mock($mock)->call('test2')->once())
        ->once() // passe
;

$this
    ->when($mock = new \mock\example)
    ->if(
        $mock->test(),
        $mock->test2()
    )
    ->mock($mock)
    ->call('test')
        ->after($this->mock($mock)->call('test2')->once())
        ->once() // échoue
;
```

atLeastOnce

`atLeastOnce` vérifie que la méthode testée (voir *call*) du mock testé a été appelée au moins une fois.

```
<?php
$mock = new \mock\MyFirstClass;
```

(suite sur la page suivante)

(suite de la page précédente)

```
$this
    ->object(new MySecondClass($mock))

    ->mock($mock)
        ->call('myMethod')
            ->atLeastOnce()
;
```

before

`before` vérifie que la méthode a été appelée avant la méthode passée en paramètre.

```
<?php
$this
    ->when($mock = new \mock\example)
    ->if(
        $mock->test(),
        $mock->test2()
    )
    ->mock($mock)
    ->call('test')
        ->before($this->mock($mock)->call('test2')->once())
        ->once() // passe
;

$this
    ->when($mock = new \mock\example)
    ->if(
        $mock->test2(),
        $mock->test()
    )
    ->mock($mock)
    ->call('test')
        ->before($this->mock($mock)->call('test2')->once())
        ->once() // échoue
;
```

exactly

`exactly` vérifie que la méthode testée (voir *call*) du mock testé exactement un certain nombre de fois.

```
<?php
$mock = new \mock\MyFirstClass;

$this
    ->object(new MySecondClass($mock))

    ->mock($mock)
        ->call('myMethod')
            ->exactly(2)
;
```

Note : il existe une version simplifiée avec `->{2}`.

never

`never` vérifie que la méthode testée (voir *call*) du mock testé n'a jamais été appelée.

```
<?php
$mock = new \mock\MyFirstClass;

$this
    ->object(new MySecondClass($mock))

    ->mock($mock)
        ->call('myMethod')
            ->never()
;
```

Note : `never` est équivalent à *exactly(0)*.

once/twice/thrice

Ces assertions vérifient que la méthode testée (voir *call*) du mock testé a été appelée exactement :
- une fois (once)
- deux fois (twice)
- trois fois (thrice)

```
<?php
$mock = new \mock\MyFirstClass;

$this
    ->object(new MySecondClass($mock))

    ->mock($mock)
        ->call('myMethod')
            ->once()
        ->call('mySecondMethod')
            ->twice()
        ->call('myThirdMethod')
            ->thrice()
;
```

Note : `once`, `twice` et `thrice` sont respectivement équivalents à un appel à *exactly(1)*, *exactly(2)* et *exactly(3)*.

withAnyArguments

`withAnyArguments` permet de ne pas spécifier les arguments attendus lors de l'appel à la méthode testée (voir *call*) du mock testé.

Cette méthode est surtout utile pour remettre à zéro les arguments, comme dans l'exemple suivant :

```
<?php
$mock = new \mock\MyFirstClass;

$this
    ->object(new MySecondClass($mock))

    ->mock($mock)
        ->call('myMethod')
            ->withArguments('first')      ->once()
            ->withArguments('second')     ->once()
            ->withAnyArguments()->exactly(2)
;
```

withArguments

`withArguments` permet de spécifier les paramètres attendus lors de l'appel à la méthode testée (voir *call*) du mock testé.

```
<?php
$mock = new \mock\MyFirstClass;

$this
    ->object(new MySecondClass($mock))

    ->mock($mock)
        ->call('myMethod')
            ->withArguments('first', 'second')->once()
;
```

Avertissement :

`withArguments` ne teste pas le type des arguments.
Si vous souhaitez vérifier également leurs types, utilisez *withIdenticalArguments*.

withIdenticalArguments

`withIdenticalArguments` permet de spécifier les paramètres attendus lors de l'appel à la méthode testée (voir *call*) du mock testé.

```
<?php
$mock = new \mock\MyFirstClass;

$this
    ->object(new MySecondClass($mock))

    ->mock($mock)
        ->call('myMethod')
            ->withIdenticalArguments('first', 'second')->once()
;
```

Avertissement :

`withIdenticalArguments` teste le type des arguments.
Si vous ne souhaitez pas vérifier leurs types, utilisez *withArguments*.

withAtLeastArguments

`withAtLeastArguments` permet de spécifier les paramètres minimums attendus lors de l'appel à la méthode testée (voir *call*) du mock testé.

```
<?php
$this
   ->if($mock = new \mock\example)
   ->and($mock->test('a', 'b'))
   ->mock($mock)
   ->call('test')
         ->withAtLeastArguments(array('a'))->once() //passes
         ->withAtLeastArguments(array('a', 'b'))->once() //passes
         ->withAtLeastArguments(array('c'))->once() //fails
;
```

Avertissement :

`withAtLeastArguments` ne teste pas le type des arguments.
Si vous souhaitez vérifier également leurs types, utilisez *withAtLeastIdenticalArguments*.

withAtLeastIdenticalArguments

`withAtLeastIdenticalArguments` permet de spécifier les paramètres minimums attendus lors de l'appel à la méthode testée (voir *call*) du mock testé.

```
<?php
$this
    ->if($mock = new \mock\example)
    ->and($mock->test(1, 2))
    ->mock($mock)
        ->call('test')
        ->withAtLeastIdenticalArguments(array(1))->once() //passes
        ->withAtLeastIdenticalArguments(array(1, 2))->once() //passes
        ->withAtLeastIdenticalArguments(array('1'))->once() //fails
;
```

Avertissement :

`withAtLeastIdenticalArguments` teste le type des arguments.
Si vous ne souhaitez pas vérifier leurs types, utilisez *withAtLeastArguments*.

withoutAnyArgument

`withoutAnyArgument` permet de spécifier que la méthode ne doit recevoir aucun paramètre lors de son appel (voir *call*).

```
<?php
$this
    ->when($mock = new \mock\example)
    ->if($mock->test())
    ->mock($mock)
        ->call('test')
            ->withoutAnyArgument()->once() // passe
    ->if($mock->test2('argument'))
    ->mock($mock)
        ->call('test2')
            ->withoutAnyArgument()->once() // échoue
;
```

Note : `withoutAnyArgument` reviens à appeler *withAtLeastArguments* avec un tableau vide : `->withAtLeastArguments(array())`.

6.17.2 receive

Est un alias de *call*.

```
<?php
$this
    ->given(
        $connection = new mock\connection
    )
    ->if(
        $this->newTestedInstance($connection)
    )
    ->then
        ->object($this->testedInstance->noMoreValue())->isTestedInstance
        ->mock($connection)->receive('newPacket')->withArguments(new packet)->once;

   // Identique à
   $this->mock($connection)->call('newPacket')->withArguments(new packet)->once;
```

6.17.3 wasCalled

`wasCalled` vérifie qu'au moins une méthode du mock a été appelée au moins une fois.

```
<?php
$mock = new \mock\MyFirstClass;

$this
    ->object(new MySecondClass($mock))

    ->mock($mock)
        ->wasCalled()
;
```

6.17.4 wasNotCalled

`wasNotCalled` vérifie qu'aucune méthode du mock n'a été appelée.

```
<?php
$mock = new \mock\MyFirstClass;

$this
    ->object(new MySecondClass($mock))

    ->mock($mock)
        ->wasNotCalled()
;
```

6.18 mysqlDateTime

C'est l'assertion dédiée aux objets décrivant une date MySQL et basée sur l'objet DateTime.

Les dates doivent utiliser un format compatible avec MySQL et de nombreux autre SGBD (Système de gestion de base de données)), à savoir « Y-m-d H :i :s » « Y-m-d H :i :s »

Note : Reportez-vous à la documentation de la fonction date() du manuel de PHP pour plus d'information.

Si vous essayez de tester une variable qui n'est pas un objet `DateTime` (ou une classe qui l'étend) avec cette assertion, cela échouera.

6.18.1 hasDate

Voir aussi :

`hasDate` est une méthode héritée de l'asserter `dateTime`. Pour plus d'informations, reportez-vous à la documentation de *dateTime : :hasDate*

6.18.2 hasDateAndTime

Voir aussi :

`hasDateAndTime` est une méthode héritée de l'asserter `dateTime`. Pour plus d'informations, reportez-vous à la documentation de *dateTime : :hasDateAndTime*

6.18.3 hasDay

Voir aussi :

`hasDay` est une méthode héritée de l'asserter `dateTime`. Pour plus d'informations, reportez-vous à la documentation de *dateTime : :hasDay*

6.18.4 hasHours

Voir aussi :

`hasHours` est une méthode héritée de l'asserter `dateTime`. Pour plus d'informations, reportez-vous à la documentation de *dateTime : :hasHours*

6.18.5 hasMinutes

Indication : `hasMinutes` est une méthode héritée de l'asserter `dateTime`. Pour plus d'informations, reportez-vous à la documentation de *dateTime : :hasMinutes*

6.18.6 hasMonth

Voir aussi :

`hasMonth` est une méthode héritée de l'asserter `dateTime`. Pour plus d'informations, reportez-vous à la documentation de *dateTime : :hasMonth*

6.18.7 hasSeconds

Voir aussi :

`hasSeconds` est une méthode héritée de l'asserter `dateTime`. Pour plus d'informations, reportez-vous à la documentation de *dateTime : :hasSeconds*

6.18.8 hasTime

Voir aussi :

`hasTime` est une méthode héritée de l'asserter `dateTime`. Pour plus d'informations, reportez-vous à la documentation de *dateTime : :hasTime*

6.18.9 hasTimezone

Voir aussi :

`hasTimezone` est une méthode héritée de l'asserter `dateTime`. Pour plus d'informations, reportez-vous à la documentation de *dateTime : :hasTimezone*

6.18.10 hasYear

Voir aussi :

`hasYear` est une méthode héritée de l'asserter `dateTime`. Pour plus d'informations, reportez-vous à la documentation de *dateTime : :hasYear*

6.18.11 isCloneOf

Voir aussi :

`isCloneOf` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isCloneOf*

6.18.12 isEqualTo

Voir aussi :

`isEqualTo` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isEqualTo*

6.18.13 isIdenticalTo

Voir aussi :

`isIdenticalTo` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isIdenticalTo*

6.18.14 isInstanceOf

Indication : `isInstanceOf` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isInstanceOf*

6.18.15 isNotEqualTo

Voir aussi :

`isNotEqualTo` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isNotEqualTo*

6.18.16 isNotIdenticalTo

Voir aussi :

`isNotIdenticalTo` est une méthode héritée de l'asserter `object`. Pour plus d'informations, reportez-vous à la documentation de *object : :isNotIdenticalTo*

6.19 object

C'est l'assertion dédiée aux objets.

Si vous essayez de tester une variable qui n'est pas un objet avec cette assertion, cela échouera.

Note : `null` n'est pas un objet. Reportez-vous au manuel de PHP pour savoir ce que is_object considère ou non comme un objet.

6.19.1 hasSize

`hasSize` vérifie la taille d'un objet qui implémente l'interface `Countable`.

```
<?php
$countableObject = new GlobIterator('*');

$this
    ->object($countableObject)
        ->hasSize(3)
;
```

6.19.2 isCallable

```
<?php
class foo
{
    public function __invoke()
    {
        // code
    }
}

$this
    ->object(new foo)
        ->isCallable()  // passe

    ->object(new StdClass)
        ->isCallable()  // échoue
;
```

Note : Pour être identifiés comme `callable`, vos objets devront être instanciés à partir de classes qui implémentent la méthode magique __invoke.

Voir aussi :

`isCallable` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isCallable*

6.19.3 isCloneOf

`isCloneOf` vérifie qu'un objet est le clone d'un objet donné, c'est-à-dire que les objets sont égaux, mais ne pointent pas vers la même instance.

```
<?php
$object1 = new \StdClass;
$object2 = new \StdClass;
$object3 = clone($object1);
$object4 = new \StdClass;
$object4->foo = 'bar';

$this
    ->object($object1)
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ->isCloneOf($object2)    // passe
        ->isCloneOf($object3)    // passe
        ->isCloneOf($object4)    // échoue
;
```

Note : Pour plus de précision, lisez la documentation PHP sur la comparaison d'objet.

6.19.4 isEmpty

`isEmpty` vérifie que la taille d'un objet implémentant l'interface `Countable` est égale à 0.

```
<?php
$countableObject = new GlobIterator('atoum.php');

$this
    ->object($countableObject)
        ->isEmpty()
;
```

Note : `isEmpty` est équivalent à `hasSize(0)`.

6.19.5 isEqualTo

`isEqualTo` vérifie qu'un objet est égal à un autre. Deux objets sont considérés égaux lorsqu'ils ont les mêmes attributs et valeurs, et qu'ils sont des instances de la même classe.

Note : Pour plus de précision, lisez la documentation PHP sur la comparaison d'objet.

Voir aussi :

`isEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isEqualTo*

6.19.6 isIdenticalTo

`isIdenticalTo` vérifie que deux objets sont identiques. Deux objets sont considérés identiques lorsqu'ils font référence à la même instance de la même classe.

Note : Pour plus de précision, lisez la documentation PHP sur la comparaison d'objet.

Voir aussi :

`isIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isIdenticalTo*

6.19.7 isInstanceOf

`isInstanceOf` vérifie qu'un objet est :
- une instance de la classe donnée,
- une sous-classe de la classe donnée (abstraite ou non),
- une instance d'une classe qui implémente l'interface donnée.

```
<?php
$object = new \StdClass();

$this
    ->object($object)
        ->isInstanceOf('\StdClass')     // passe
        ->isInstanceOf('\Iterator')     // échoue
;


interface FooInterface
{
    public function foo();
}

class FooClass implements FooInterface
{
    public function foo()
    {
        echo "foo";
    }
}

class BarClass extends FooClass
{
}

$foo = new FooClass;
$bar = new BarClass;

$this
    ->object($foo)
        ->isInstanceOf('\FooClass')     // passe
        ->isInstanceOf('\FooInterface') // passe
        ->isInstanceOf('\BarClass')     // échoue
        ->isInstanceOf('\StdClass')     // échoue

    ->object($bar)
        ->isInstanceOf('\FooClass')     // passe
        ->isInstanceOf('\FooInterface') // passe
        ->isInstanceOf('\BarClass')     // passe
        ->isInstanceOf('\StdClass')     // échoue
;
```

Note : Les noms des classes et des interfaces doivent être absolus, car les éventuelles importations d'espace de nommage ne sont pas prises en compte.

Indication : Notez qu'avec PHP >= `5.5` vous pouvez utiliser le mot-clé `class` pour obtenir les noms de classe

absolus, par exemple `$this->object($foo)->isInstanceOf(FooClass::class)`.

6.19.8 isInstanceOfTestedClass

```
<?php
$this->newTestedInstance;
$object = new TestedClass();
$this->object($this->testedInstance)->isInstanceOfTestedClass;
$this->object($object)->isInstanceOfTestedClass;
```

6.19.9 isNotCallable

```
<?php
class foo
{
    public function __invoke()
    {
        // code
    }
}

$this
    ->variable(new foo)
        ->isNotCallable()    // échoue

    ->variable(new StdClass)
        ->isNotCallable()    // passe
;
```

Voir aussi :

`isNotCallable` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotCallable*

6.19.10 isNotEqualTo

`isEqualTo` vérifie qu'un objet n'est pas égal à un autre. Deux objets sont considérés égaux lorsqu'ils ont les mêmes attributs et valeurs, et qu'ils sont des instances de la même classe.

Note : Pour plus de précision, lisez la documentation PHP sur la comparaison d'objet.

Voir aussi :

`isNotEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotEqualTo*

6.19.11 isNotIdenticalTo

`isIdenticalTo` vérifie que deux objets ne sont pas identiques. Deux objets sont considérés identiques lorsqu'ils font référence à la même instance de la même classe.

Note : Pour plus de précision, lisez la documentation PHP sur la comparaison d'objet.

Voir aussi :

`isNotIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotIdenticalTo*

6.19.12 isNotInstanceOf

`isNotInstanceOf` vérifie qu'un objet n'est pas :
- une instance de la classe donnée,
- une sous-classe de la classe donnée (abstraite ou non),
- une instance d'une classe qui implémente l'interface donnée.

```
<?php
$object = new \StdClass();

$this
    ->object($object)
        ->isNotInstanceOf('\StdClass')      // échoue
        ->isNotInstanceOf('\Iterator')      // échoue
;
```

Note : Tout comme *isInstanceOf*, le nom de la classe ou de l'interface doivent être absolus car les imports de namespace seront ignorés.

6.19.13 isNotTestedInstance

```
<?php
$this->newTestedInstance;
$this->object($this->testedInstance)->isNotTestedInstance; // fail
```

6.19.14 isTestedInstance

```
<?php
$this->newTestedInstance;
$this->object($this->testedInstance)->isTestedInstance;

$object = new TestedClass();
$this->object($object)->isTestedInstance; // échec
```

6.19.15 toString

L'assertion toString caste un objet en string et retourne un asserter *string* sur cette valeur.

Exemple :

```
<?php
$this
  ->object(
    new class {
      public function __toString()
      {
        return 'foo';
      }
    }
  )
    ->isIdenticalTo('foo') // rate
    ->toString
      ->isIdenticalTo('foo') // passe
;
```

6.20 output

C'est l'assertion dédiée aux tests sur les sorties, c'est-à-dire tout ce qui est censé être affiché à l'écran.

```
<?php
$this
    ->output(
        function() {
            echo 'Hello world';
        }
    )
;
```

Note : La syntaxe utilise les fonctions anonymes (aussi appelées fermetures ou closures) introduites en PHP 5.3. Pour plus de précision, lisez la documentation PHP sur les fonctions anonymes.

6.20.1 contains

Voir aussi :

`contains` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :contains*

6.20.2 hasLength

Voir aussi :

`hasLength` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :hasLength*

6.20.3 hasLengthGreaterThan

Voir aussi :

`hasLengthGreaterThan` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :hasLengthGreaterThan*

6.20.4 hasLengthLessThan

Voir aussi :

`hasLengthLessThan` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :hasLengthLessThan*

6.20.5 isEmpty

Voir aussi :

`isEmpty` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :isEmpty*

6.20.6 isEqualTo

Voir aussi :

`isEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isEqualTo*

6.20.7 isEqualToContentsOfFile

Voir aussi :

`isEqualToContentsOfFile` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :isEqualToContentsOfFile*

6.20.8 isIdenticalTo

Voir aussi :

`isIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isIdenticalTo*

6.20.9 isNotEmpty

Voir aussi :

`isNotEmpty` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :isNotEmpty*

6.20.10 isNotEqualTo

Voir aussi :

`isNotEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotEqualTo*

6.20.11 isNotIdenticalTo

Voir aussi :

`isNotIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotIdenticalTo*

6.20.12 matches

Voir aussi :

`matches` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :matches*

6.20.13 notContains

Voir aussi :

`notContains` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :notContains*

6.21 resource

C'est l'assertion dédié au ´ressource <http://php.net/language.types.resource>´_.

6.21.1 isOfType

Cette méthode permet de comparer le type de ressource avec le type de la valeur fournie en paramètre. Dans l'exemple suivant, on vérifie que le paramètre est un stream.

```
$this
    ->resource($variable)
        ->isOfType('stream')
;
```

6.21.2 isStream

```
$this
    ->resource($variable)
        ->isStream()
;
```

->is*() va faire correspondre le type du flux au nom d'une méthode : ->isFooBar() va essayer de trouver un flux correspondant à foo bar, fooBar, foo_bar...

6.21.3 type

```
$this
    ->resource($variable)
        ->type
                        ->isEqualTo('stream')
                        ->matches('/foo.*bar/')
;
```

->$type est un helper fournissant *l'assertion des string* sur le type de stream.

6.22 sizeOf

C'est l'assertion dédiée aux tests sur la taille des tableaux et des objets implémentant l'interface `Countable`.

```
<?php
$array           = array(1, 2, 3);
$countableObject = new GlobIterator('*');

$this
    ->sizeOf($array)
        ->isEqualTo(3)

    ->sizeOf($countableObject)
        ->isGreaterThan(0)
;
```

6.22.1 isEqualTo

Voir aussi :

`isEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isEqualTo*

6.22.2 isGreaterThan

Voir aussi :

`isGreaterThan` est une méthode héritée de l'asserter `integer`. Pour plus d'informations, reportez-vous à la documentation de *integer : :isGreaterThan*

6.22.3 isGreaterThanOrEqualTo

Voir aussi :

`isGreaterThanOrEqualTo` est une méthode héritée de l'asserter `integer`. Pour plus d'informations, reportez-vous à la documentation de *integer : :isGreaterThanOrEqualTo*

6.22.4 isIdenticalTo

Voir aussi :

`isIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isIdenticalTo*

6.22.5 isLessThan

Voir aussi :

`isLessThan` est une méthode héritée de l'asserter `integer`. Pour plus d'informations, reportez-vous à la documentation de *integer : :isLessThan*

6.22.6 isLessThanOrEqualTo

Voir aussi :

`isLessThanOrEqualTo` est une méthode héritée de l'asserter `integer`. Pour plus d'informations, reportez-vous à la documentation de *integer : :isLessThanOrEqualTo*

6.22.7 isNotEqualTo

Voir aussi :

`isNotEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotEqualTo*

6.22.8 isNotIdenticalTo

Voir aussi :

`isNotIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotIdenticalTo*

6.22.9 isZero

Voir aussi :

`isZero` est une méthode héritée de l'asserter `integer`. Pour plus d'informations, reportez-vous à la documentation de *integer : :isZero*

6.23 stream

C'est l'assertion dédié au ´streams <http://php.net/intro.stream>´_.

Elle est basée sur le système de fichier virtuel d'atoum (VFS). Un nouveau stream wrapper sera enregistré (qui commence par `atoum://`).

Le bouchon va créer un nouveau fichier dans le VFS et le chemin du flux sera accessible en appellant la méthode `getPath` sur le controlleur de flux (par exemple `atoum://mockUniqId`).

6.23.1 isRead

`isRead` vérifie si le flux bouchoné à bien été lu.

```php
<?php
$this
    ->given(
        $streamController = \atoum\mock\stream::get(),
        $streamController->file_get_contents = 'myFakeContent'
    )
    ->if(file_get_contents($streamController->getPath()))
    ->stream($streamController)
        ->isRead() // passe
;

$this
    ->given(
        $streamController = \atoum\mock\stream::get(),
        $streamController->file_get_contents = 'myFakeContent'
    )
    ->if() //we do nothing
    ->stream($streamController)
        ->isRead() // échoue
;
```

6.23.2 isWritten

`isWritten` vérifie si le flux bouchoné à bien été écrit.

```php
<?php
$this
    ->given(
        $streamController = \atoum\mock\stream::get(),
        $streamController->file_put_contents = strlen($content = 'myTestContent')
    )
    ->if(file_put_contents($streamController->getPath(), $content))
    ->stream($streamController)
        ->isWritten() // passe
;

$this
   ->given(
     $streamController = \atoum\mock\stream::get(),
     $streamController->file_put_contents = strlen($content = 'myTestContent')
   )
   ->if() // we do nothing
   ->stream($streamController)
      ->isWritten() // échoue
;
```

6.23.3 isWrited

Indication : `isWrited` est un alias de la méthode `isWritten`. Pour plus d'informations, reportez-vous à la documentation de *stream : :isWritten*

6.24 string

C'est l'assertion dédiée aux chaînes de caractères.

6.24.1 contains

`contains` vérifie qu'une chaîne de caractère contient une autre chaîne de caractère donnée.

```
<?php
$string = 'Hello world';

$this
    ->string($string)
        ->contains('ll')     // passe
        ->contains(' ')      // passe
        ->contains('php')    // échoue
;
```

6.24.2 endWith

`endWith` vérifie qu'une chaîne de caractère se termine par une autre chaîne de caractère donnée.

```
<?php
$string = 'Hello world';

$this
    ->string($string)
        ->endWith('world')      // passe
        ->endWith('lo world')   // passe
        ->endWith('Hello')      // échoue
        ->endWith(' ')          // échoue
;
```

6.24.3 hasLength

`hasLength` vérifie la taille de la chaîne de caractère.

```
<?php
$string = 'Hello world';

$this
    ->string($string)
        ->hasLength(11)      // passe
        ->hasLength(20)      // échoue
;
```

6.24.4 hasLengthGreaterThan

`hasLengthGreaterThan` vérifie que la taille d'une chaîne de caractères est plus grande qu'une valeur donnée.

```
<?php
$string = 'Hello world';

$this
    ->string($string)
        ->hasLengthGreaterThan(10)      // passe
        ->hasLengthGreaterThan(20)      // échoue
;
```

6.24.5 hasLengthLessThan

`hasLengthLessThan` vérifie que la taille d'une chaîne de caractères est plus petite qu'une valeur donnée.

```
<?php
$string = 'Hello world';

$this
    ->string($string)
        ->hasLengthLessThan(20)      // passe
        ->hasLengthLessThan(10)      // échoue
;
```

6.24.6 isEmpty

`isEmpty` vérifie qu'une chaîne de caractères est vide.

```
<?php
$emptyString    = '';
$nonEmptyString = 'atoum';

$this
    ->string($emptyString)
        ->isEmpty()             // passe

    ->string($nonEmptyString)
        ->isEmpty()             // échoue
;
```

6.24.7 isEqualTo

Voir aussi :

`isEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isEqualTo*

6.24.8 isEqualToContentsOfFile

`isEqualToContentsOfFile` vérifie qu'une chaîne de caractère est égale au contenu d'un fichier donné par son chemin.

```
<?php
$this
    ->string($string)
        ->isEqualToContentsOfFile('/path/to/file')
;
```

Note : si le fichier n'existe pas, le test échoue.

6.24.9 isIdenticalTo

Voir aussi :

`isIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isIdenticalTo*

6.24.10 isNotEmpty

`isNotEmpty` vérifie qu'une chaîne de caractères n'est pas vide.

```
<?php
$emptyString    = '';
$nonEmptyString = 'atoum';

$this
    ->string($emptyString)
        ->isNotEmpty()          // échoue

    ->string($nonEmptyString)
        ->isNotEmpty()          // passe
;
```

6.24.11 isNotEqualTo

Voir aussi :

`isNotEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotEqualTo*

6.24.12 isNotIdenticalTo

Voir aussi :

`isNotIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotIdenticalTo*

6.24.13 length

`length` vous permet de récupérer un asserter de type *integer* contenant la taille de la chaîne de caractères testée.

```
<?php
$string = 'atoum';

$this
    ->string($string)
        ->length
            ->isGreaterThanOrEqualTo(5)
;
```

6.24.14 match

Indication : `match` est un alias de la méthode `matches`. Pour plus d'informations, reportez-vous à la documentation de *string : :matches*

6.24.15 matches

`matches` vérifie qu'une expression régulière correspond à la chaîne de caractères.

```
<?php
$phone = '0102030405';
$vdm = "Aujourd'hui, à 57 ans, mon père s'est fait tatouer une licorne sur l'épaule.␣
↪VDM";

$this
    ->string($phone)
        ->matches('#^0[1-9]\d{8}$#')

    ->string($vdm)
        ->matches("#^Aujourd'hui.*VDM$#")
;
```

6.24.16 notContains

`notContains` vérifie qu'une chaîne de caractère ne contient pas une autre chaîne de caractère donnée.

```
<?php
$string = 'Hello world';

$this
    ->string($string)
        ->notContains('php')    // passe
        ->notContains(';')      // passe
        ->notContains('ll')     // échoue
        ->notContains(' ')      // échoue
;
```

6.24.17 notEndWith

`notEndWith` vérifie qu'une chaîne de caractère ne se termine pas par une autre chaîne de caractère donnée.

```
<?php
$string = 'Hello world';

$this
    ->string($string)
        ->notEndWith('Hello')      // passe
        ->notEndWith(' ')          // passe
        ->notEndWith('world')      // échoue
        ->notEndWith('lo world')   // échoue
;
```

6.24.18 notMatches

`notMatches` vérifie qu'une expression régulière ne correspond pas à la chaîne de caractères.

```
<?php
$phone = '0102030405';
$vdm = "Aujourd'hui, à 57 ans, mon père s'est fait tatouer une licorne sur l'épaule.␣
↪VDM";

$this
    ->string($phone)
        ->notMatches('#azerty#') // passe
        ->notMatches('#^0[1-9]\d{8}$#') // échoue

    ->string($vdm)
        ->notMatches("#^Hier.*VDM$#") // passe
        ->notMatches("#^Aujourd'hui.*VDM$#") // échoue
;
```

6.24.19 notStartWith

`notStartWith` vérifie qu'une chaîne de caractère ne commence pas par une autre chaîne de caractère donnée.

```
<?php
$string = 'Hello world';

$this
    ->string($string)
        ->notStartWith('world')      // passe
        ->notStartWith(' ')          // passe
        ->notStartWith('Hello wo')   // échoue
        ->notStartWith('He')         // échoue
;
```

6.24.20 startWith

`startWith` vérifie qu'une chaîne de caractère commence par une autre chaîne de caractère donnée.

```
<?php
$string = 'Hello world';

```

(suite sur la page suivante)

(suite de la page précédente)

```
$this
    ->string($string)
        ->startWith('Hello wo') // passe
        ->startWith('He')       // passe
        ->startWith('world')    // échoue
        ->startWith(' ')        // échoue

;
```

Nouveau dans la version 3.3.0 : Ajout de l'assertion *notMatches*

6.25 utf8String

C'est l'assertion dédiée aux chaînes de caractères UTF-8.

Note : `utf8Strings` utilise les fonctions `mb_*` pour gérer les chaînes multioctets. Reportez-vous au manuel de PHP pour avoir plus d'information sur l'extension mbstring.

6.25.1 contains

Voir aussi :

`contains` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :contains*

6.25.2 hasLength

Voir aussi :

`hasLength` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :hasLength*

6.25.3 hasLengthGreaterThan

Voir aussi :

`hasLengthGreaterThan` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :hasLengthGreaterThan*

6.25.4 hasLengthLessThan

Voir aussi :

`hasLengthLessThan` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :hasLengthLessThan*

6.25.5 isEmpty

Voir aussi :

`isEmpty` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :isEmpty*

6.25.6 isEqualTo

Voir aussi :

`isEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isEqualTo*

6.25.7 isEqualToContentsOfFile

Voir aussi :

`isEqualToContentsOfFile` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :isEqualToContentsOfFile*

6.25.8 isIdenticalTo

Voir aussi :

`isIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isIdenticalTo*

6.25.9 isNotEmpty

Voir aussi :

`isNotEmpty` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :isNotEmpty*

6.25.10 isNotEqualTo

Voir aussi :

`isNotEqualTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotEqualTo*

6.25.11 isNotIdenticalTo

Voir aussi :

`isNotIdenticalTo` est une méthode héritée de l'asserter `variable`. Pour plus d'informations, reportez-vous à la documentation de *variable : :isNotIdenticalTo*

6.25.12 matches

Indication : `matches` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :matches*

Note : Pensez à bien ajouter `u` comme option de recherche dans votre expression régulière. Pour plus de précision, lisez la documentation PHP sur les options de recherche des expressions régulières.

```
<?php
$vdm = "Aujourd'hui, à 57 ans, mon père s'est fait tatouer une licorne sur l'épaule.␣
↪FML";

$this
    ->utf8String($vdm)
        ->matches("#^Aujourd'hui.*VDM$#u")
;
```

6.25.13 notContains

Voir aussi :

`notContains` est une méthode héritée de l'asserter `string`. Pour plus d'informations, reportez-vous à la documentation de *string : :notContains*

6.26 variable

C'est l'assertion de base de toutes les variables. Elle contient les tests nécessaires à n'importe quel type de variable.

6.26.1 isCallable

`isCallable` vérifie que la variable peut être appelée comme fonction.

```
<?php
$f = function() {
    // code
};

$this
    ->variable($f)
        ->isCallable()  // passe

    ->variable('\Vendor\Project\foobar')
        ->isCallable()

    ->variable(array('\Vendor\Project\Foo', 'bar'))
        ->isCallable()

    ->variable('\Vendor\Project\Foo::bar')
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ->isCallable()
;
```

6.26.2 isEqualTo

`isEqualTo` vérifie que la variable est égale à une certaine donnée.

```
<?php
$a = 'a';

$this
    ->variable($a)
        ->isEqualTo('a')     // passe
;
```

Avertissement :

`isEqualTo` ne teste pas le type de la variable.
Si vous souhaitez vérifier également son type, utilisez *isIdenticalTo*.

6.26.3 isIdenticalTo

`isIdenticalTo` vérifie que la variable a la même valeur et le même type qu'une certaine donnée. Dans le cas d'objets, `isIdenticalTo` vérifie que les données pointent sur la même instance.

```
<?php
$a = '1';

$this
    ->variable($a)
        ->isIdenticalTo(1)          // échoue
;

$stdClass1 = new \StdClass();
$stdClass2 = new \StdClass();
$stdClass3 = $stdClass1;

$this
    ->variable($stdClass1)
        ->isIdenticalTo(stdClass3)  // passe
        ->isIdenticalTo(stdClass2)  // échoue
;
```

Avertissement :

`isIdenticalTo` teste le type de la variable.
Si vous ne souhaitez pas vérifier son type, utilisez *isEqualTo*.

6.26.4 isNotCallable

`isNotCallable` vérifie que la variable ne peut pas être appelée comme fonction.

```
<?php
$f = function() {
    // code
};
$int    = 1;
$string = 'nonExistingMethod';

$this
    ->variable($f)
        ->isNotCallable()    // échoue

    ->variable($int)
        ->isNotCallable()    // passe

    ->variable($string)
        ->isNotCallable()    // passe

    ->variable(new StdClass)
        ->isNotCallable()    // passe
;
```

6.26.5 isNotEqualTo

`isNotEqualTo` vérifie que la variable n'a pas la même valeur qu'une certaine donnée.

```
<?php
$a       = 'a';
$aString = '1';

$this
    ->variable($a)
        ->isNotEqualTo('b')      // passe
        ->isNotEqualTo('a')      // échoue

    ->variable($aString)
        ->isNotEqualTo($a)       // échoue
;
```

Avertissement :

`isNotEqualTo` ne teste pas le type de la variable.
Si vous souhaitez vérifier également son type, utilisez *isNotIdenticalTo*.

6.26.6 isNotIdenticalTo

`isNotIdenticalTo` vérifie que la variable n'a ni le même type ni la même valeur qu'une certaine donnée.

Dans le cas d'objets, `isNotIdenticalTo` vérifie que les données ne pointent pas sur la même instance.

```php
<?php
$a = '1';

$this
    ->variable($a)
        ->isNotIdenticalTo(1)           // passe
;

$stdClass1 = new \StdClass();
$stdClass2 = new \StdClass();
$stdClass3 = $stdClass1;

$this
    ->variable($stdClass1)
        ->isNotIdenticalTo(stdClass2)   // passe
        ->isNotIdenticalTo(stdClass3)   // échoue
;
```

Avertissement :

`isNotIdenticalTo` teste le type de la variable.
Si vous ne souhaitez pas vérifier son type, utilisez *isNotEqualTo*.

6.26.7 isNull

`isNull` vérifie que la variable est nulle.

```php
<?php
$emptyString = '';
$null        = null;

$this
    ->variable($emptyString)
        ->isNull()              // échoue
                                // (c'est vide, mais pas null)

    ->variable($null)
        ->isNull()              // passe
;
```

6.26.8 isNotNull

`isNotNull` vérifie que la variable n'est pas nulle.

```php
<?php
$emptyString = '';
$null        = null;

$this
    ->variable($emptyString)
        ->isNotNull()           // passe (c'est vide, mais pas null)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ->variable($null)
        ->isNotNull()           // échoue
;
```

6.26.9 isNotTrue

`isNotTrue` vérifie que la variable n'est strictement pas égale à `true`.

```
<?php
$true  = true;
$false = false;
$this
    ->variable($true)
        ->isNotTrue()      // échoue

    ->variable($false)
        ->isNotTrue()      // passe
;
```

6.26.10 isNotFalse

`isNotFalse` vérifie que la variable n'est strictement pas égale à `false`.

```
<?php
$true  = true;
$false = false;
$this
    ->variable($false)
        ->isNotFalse()     // échoue

    ->variable($true)
        ->isNotFalse()     // passe
;
```

6.27 Asserter & assertion trucs et astuces

Plusieurs trucs et astuces sont disponibles pour les assertions. Les connaître peuvent simplifier votre vie ;)

Le premier est que chaque assertion est fluent (chaînable). Donc vous pouvez les enchaîner, il suffit de regarder les exemples précédents.

Vous devez également savoir que toutes les assertions sans paramètres peuvent être écrites avec ou sans parenthèses. Donc `$this->integer(0)->isZero()` est la même chose que `$this->integer(0)->isZero`.

6.27.1 Alias

Parfois, vous voulez utiliser quelque chose qui reflètent votre vocabulaire ou votre domaine. atoum fournis un mécanisme simple, les alias. En voici un exemple :

```php
<?php
namespace tests\units;

use mageekguy\atoum;

class stdClass extends atoum\test
{
    public function __construct(adapter $adapter = null, annotations\extractor
↪$annotationExtractor = null, asserter\generator $asserterGenerator = null,␣
↪test\assertion\manager $assertionManager = null, \closure $reflectionClassFactory =␣
↪null)
    {
        parent::__construct($adapter, $annotationExtractor, $asserterGenerator,
↪$assertionManager, $reflectionClassFactory);

        $this
            ->from('string')->use('isEqualTo')->as('equals')
        ;
    }

    public function testFoo()
    {
        $this
            ->string($u = uniqid())->equals($u)
        ;
    }
}
```

Dans cet exemple, nous créons un alias pour faire un nouvel asserter `equal```qui agira de la même manière que ``isEqualTo`. Vous pouvez utiliser *beforeTestMethod* à la place du constructeur. Afin de partager ces alias entre les différents tests, le mieux est de créer une classe de base pour vos tests à l'intérieur de votre projet que vous pourrez étendre à la place `\atoum\test`.

6.27.2 Asserter personnalisé

Maintenant que nous avons vu alias, nous pouvons aller plus loin en créant un asserter personnalisé. Voici un exemple d'un asserter pour carte de crédit.

```php
<?php
namespace tests\units;
use mageekguy\atoum;

class creditcard extends atoum\asserters\string
{
    public function isValid($failMessage = null)
    {
        return $this->match('/(?:\d{4}){4}/', $failMessage ?: $this->_('%s is not a␣
↪valid credit card number', $this));
    }
}

class stdClass extends atoum\test
{
    public function __construct(adapter $adapter = null, annotations\extractor
↪$annotationExtractor = null, asserter\generator $asserterGenerator = null,␣
↪test\assertion\manager $assertionManager = null, \closure $reflectionClassFactory =␣
↪null)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {
        parent::__construct($adapter, $annotationExtractor, $asserterGenerator,
↪$assertionManager, $reflectionClassFactory);

        $this->getAsserterGenerator()->addNamespace('tests\units');
    }

    public function testFoo()
    {
        $this
            ->creditcard('4444555566660000')->isValid()
        ;
    }
}
```

Tout comme pour un alias, il est conseillé de créer une classe de base pour vos tests et déclarer l'asserter personnalisé à cet endroit.

6.27.3 Syntaxe courte

Avec un *alias* vous pouvez définir plusieurs choses intéressantes. Afin de vous aider dans la rédaction de vos tests, plusieurs alias sont disponibles nativement.

- **==** est la même chose que l'asserter *isEqualTo*
- **===** est la même chose que l'asserter *isIdenticalTo*
- **!=** est la même chose que l'asserter *isNotEqualTo*
- **!==** est la même chose que l'asserter *isNotIdenticalTo*
- **<** est équivalent à *isLessThan*
- **<=** est la même chose que l'asserter *isLessThanOrEqualTo*
- **>** est la même chose que l'asserter *isGreaterThan*
- **>=** est la même chose que l'asserter *isGreaterThanOrEqualTo*

```
<?php
namespace tests\units;

use atoum;

class stdClass extends atoum
{
    public function testFoo()
    {
        $this
            ->variable('foo')->{'=='}('foo')
            ->variable('foo')->{'foo'} // équivalent à la ligne précédente
            ->variable('foo')->{'!='}('bar')

            ->object($this->newInstance)->{'=='}($this->newInstance)
            ->object($this->newInstance)->{'!='}(new \exception)
            ->object($this->newTestedInstance)->{'==='}($this->testedInstance)
            ->object($this->newTestedInstance)->{'!=='}($this->newTestedInstance)

            ->integer(rand(0, 10))->{'<'}(11)
            ->integer(rand(0, 10))->{'<='}(10)
            ->integer(rand(0, 10))->{'>'}(-1)
            ->integer(rand(0, 10))->{'>='}(0)
```

(suite sur la page suivante)

(suite de la page précédente)

```
            ;
        }
}
```

CHAPITRE 7

Système de mocks

Les mocks(bouchons) sont des classes virtuels créer à la volée. Ils sont utilisé pour isolé les tests du comportements des autres classes. atoum a un système de mock simple et puissant, permettant de générer des mocks depuis une classe ou une interface qui existe ou est virtuel, ou encore est abstraite.

Grâce à ces bouchons, vous pourrez simuler des comportements en redéfinissant les méthodes *publiques* de vos classes. Pour les méthodes private et protected, vous pouvez utiliser l'extension de visibilité.

Avertissement : La plupart de méthode qui configurent le mock, s'appliquent uniquement pour la prochaine génération de ceux-ci !

7.1 Générer un bouchon

Il y a plusieurs manières de créer un bouchon à partir d'une interface ou d'une classe. Le plus simple est de créer un objet avec le nom absolu préfixé de `mock` :

```
<?php
// création d'un bouchon de l'interface \Countable
$countableMock = new \mock\Countable;

// création d'un bouchon de la classe abstraite
// \Vendor\Project\AbstractClass
$vendorAppMock = new \mock\Vendor\Project\AbstractClass;

// creation of mock of the \StdClass class
$stdObject     = new \mock\StdClass;

// création d'un bouchon à partir d'une classe inexistante
$anonymousMock = new \mock\My\Unknown\Claass;
```

7.1.1 Générer un mock avec newMockInstance

Si vous préférez il existe une méthode `newMockInstance()` qui permet la génération d'un mock.

```
<?php
// création d'un bouchon de l'interface \Countable
$countableMock = new \mock\Countable;

// est équivalent à
$this->newMockInstance('Countable');
```

Note : Comme le générateur de mock, vous pouvez fournir des paramètres en plus : `$this->newMockInstance('class name', 'mock namespace', 'mock class name', ['constructor args']);`

7.2 Le générateur de bouchon

atoum s'appuie sur un composant spécialisé pour générer les bouchons : le `mockGenerator`. Vous avez accès à ce dernier dans vos tests afin de modifier la procédure de génération des mocks.

Par défaut, le mock sera généré dans le namespace « mock » et fonctionnera exactement de la même manière que l'instance de la classe originale (le mock hérite directement de la classe d'origine).

7.2.1 Changer le nom de la classe

Si vous désirez changer le nom de la classe ou son espace de nom, vous devez utiliser le `mockGenerator`.

La méthode `generate` prend trois paramètres :

- le nom de l'interface ou de la classe à bouchonner ;
- le nouvel espace de nom, optionnel ;
- le nouveau nom de la classe, optionnel.

```
<?php
// création d'un bouchon de l'interface \Countable vers \MyMock\Countable
// on ne change que l'espace de nom
$this->mockGenerator->generate('\Countable', '\MyMock');

// création d'un bouchon de la classe abstraite
// \Vendor\Project\AbstractClass to \MyMock\AClass
// on change l'espace de nom et le nom de la classe
$this->mockGenerator->generate('\Vendor\Project\AbstractClass', '\MyMock', 'AClass');

// création d'un bouchon de la classe \StdClass vers \mock\OneClass
// on ne change que le nom de la classe
$this->mockGenerator->generate('\StdClass', null, 'OneClass');

// on peut maintenant instancier ces mocks
$vendorAppMock = new \myMock\AClass;
$countableMock = new \myMock\Countable;
$stdObject     = new \mock\OneClass;
```

Note : Si vous n'utilisez que le premier argument et ne changez pas le namespace ou le nom de la classe, alors la première solution est équivalente, plus simple à lire et recommandée.

Vous pouvez accéder au code généré pour la classe par le générateur de mock en appelant `$this->mockGenerator->getMockedClassCode()`, pour débuguer par exemple. Cette méthode prend les mêmes arguments que la méthode `generate`.

```
<?php
$countableMock = new \mock\Countable;

// est équivalent à:

$this->mockGenerator->generate('\Countable');   // inutile
$countableMock = new \mock\Countable;
```

Note : Tout ce qui est décrit ici avec le générateur de mock peut être utilisé avec *newMockInstance*

7.2.2 Shunter les appels aux méthodes parentes

shuntParentClassCalls & unShuntParentClassCalls

Un bouchon hérite directement de la classe à partir de laquelle il a été généré, ses méthodes se comportent donc exactement de la même manière.

Dans certains cas, il peut être utile de shunter les appels aux méthodes parentes afin que leur code ne soit plus exécuté. Le `mockGenerator` met à votre disposition plusieurs méthodes pour y parvenir :

```
<?php
// le bouchon ne fera pas appel à la classe parente
$this->mockGenerator->shuntParentClassCalls();

$mock = new \mock\OneClass;

// le bouchon fera à nouveau appel à la classe parente
$this->mockGenerator->unshuntParentClassCalls();
```

Ici, toutes les méthodes du bouchon se comporteront comme si elles n'avaient pas d'implémentation par contre elles conserveront la signature des méthodes originales.

Note : `shuntParentClassCalls` va *seulement* être appliqué à la prochaine génération de mock. *Mais* si vous créer deux mock de la même classe, les deux auront leurs méthodes parente shunté.

shunt

Vous pouvez également préciser les méthodes que vous souhaitez shunter :

```
<?php
// le bouchon ne fera pas appel à la classe parente pour la méthode firstMethod......
$this->mockGenerator->shunt('firstMethod');
```

(suite sur la page suivante)

(suite de la page précédente)

```
// ... ni pour la méthode secondMethod
$this->mockGenerator->shunt('secondMethod');

$countableMock = new \mock\OneClass;
```

Une méthode shuntée aura un corps de méthode vide mais comme pour `shuntParentClassCalls` la signature de la méthode sera la même que celle bouchonée.

7.2.3 Rendre une méthode orpheline

Il peut parfois être intéressant de rendre une méthode orpheline, c'est-à-dire, lui donner une signature et une implémentation vide. Cela peut être particulièrement utile pour générer des bouchons sans avoir à instancier toutes leurs dépendances. Tous les paramètres de la méthode seront également définis avec comme valeur par défaut null. C'est donc la même chose que *shunter une méthode* mais avec tout les paramètres a null.

```
<?php
class FirstClass {
    protected $dep;

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

class SecondClass {
    protected $deps;

    public function __construct(ThirdClass $a, FourthClass $b) {
        $this->deps = array($a, $b);
    }
}

$this->mockGenerator->orphanize('__construct');
$this->mockGenerator->shuntParentClassCalls();

// Nous pouvons instancier le bouchon sans injecter ses dépendances
$mock = new \mock\SecondClass();

$object = new FirstClass($mock);
```

Note : `orphanize` va *seulement* être appliqué à la prochaine génération de mock.

7.3 Modifier le comportement d'un bouchon

Une fois le bouchon créé et instancié, il est souvent utile de pouvoir modifier le comportement de ses méthodes. Pour cela, il faut passer par son contrôleur en utilisant l'une des méthodes suivantes :

- $yourMock->getMockController()->yourMethod
- $this->calling($yourMock)->yourMethod

```php
<?php
$mockDbClient = new \mock\Database\Client();

$mockDbClient->getMockController()->connect = function() {};
// Equivalent to
$this->calling($mockDbClient)->connect = function() {};
```

Le `mockController` vous permet de redéfinir **uniquement les méthodes publiques et abstraites protégées** et met à votre disposition plusieurs méthodes :

```php
<?php
$mockDbClient = new \mock\Database\Client();

// redéfinit la méthode connect : elle retournera toujours true
$this->calling($mockDbClient)->connect = true;

// redéfinit la méthode select : elle exécutera la fonction anonyme passée
$this->calling($mockDbClient)->select = function() {
    return array();
};

// redéfinit la méthode query avec des arguments
$result = array();
$this->calling($mockDbClient)->query = function(Query $query) use($result) {
    switch($query->type) {
        case Query::SELECT:
            return $result;

        default;
            return null;
    }
};

// la méthode connect lèvera une exception
$this->calling($mockDbClient)->connect->throw = new \Database\Client\Exception();
```

Note : La syntaxe utilise les fonctions anonymes (aussi appelées fermetures ou closures) introduites en PHP 5.3. Reportez-vous au manuel de PHP pour avoir plus d'informations sur le sujet.

Comme vous pouvez le voir, il est possible d'utiliser plusieurs méthodes afin d'obtenir le comportement souhaité :

— Utiliser une valeur statique qui sera retournée par la méthode
— Utiliser une implémentation courte grâce aux fonctions anonymes de PHP
— Utiliser le mot-clef `throw` pour lever une exception

7.3.1 Changement de comportement du mock sur plusieurs appels

Vous pouvez également spécifier plusieurs valeurs en fonction de l'ordre d'appel :

```php
<?php
// default
$this->calling($mockDbClient)->count = rand(0, 10);
// équivalent à
$this->calling($mockDbClient)->count[0] = rand(0, 10);
```

(suite sur la page suivante)

(suite de la page précédente)

```
// 1er appel
$this->calling($mockDbClient)->count[1] = 13;

// 3ème appel
$this->calling($mockDbClient)->count[3] = 42;
```

— Le premier appel retournera 13.
— Le second aura le comportement par défaut, c'est-à-dire un nombre aléatoire.
— Le troisième appel retournera 42.
— Tous les appels suivants auront le comportement par défaut, c'est à dire des nombres aléatoires.

Si vous souhaitez que plusieurs méthodes du bouchon aient le même comportement, vous pouvez utiliser les méthodes *methods* ou *methodsMatching*.

7.3.2 methods

`methods` vous permet, grâce à la fonction anonyme passée en argument, de définir pour quelles méthodes le comportement doit être modifié :

```
<?php
// si la méthode a tel ou tel nom,
// on redéfinit son comportement
$this
    ->calling($mock)
        ->methods(
            function($method) {
                return in_array(
                    $method,
                    array(
                        'getOneThing',
                        'getAnOtherThing'
                    )
                );
            }
        )
            ->return = uniqid()
;

// on redéfinit le comportement de toutes les méthodes
$this
    ->calling($mock)
        ->methods()
            ->return = null
;

// si la méthode commence par "get",
// on redéfinit son comportement
$this
    ->calling($mock)
        ->methods(
            function($method) {
                return substr($method, 0, 3) == 'get';
            }
        )
            ->return = uniqid()
;
```

Dans le cas du dernier exemple, vous devriez plutôt utiliser *methodsMatching*.

Note : La syntaxe utilise les fonctions anonymes (aussi appelées fermetures ou closures) introduites en PHP 5.3. Reportez-vous au manuel de PHP pour avoir plus d'informations sur le sujet.

7.3.3 methodsMatching

`methodsMatching` vous permet de définir les méthodes où le comportement doit être modifié grâce à l'expression rationnelle passée en argument :

```
<?php
// si la méthode commence par "is",
// on redéfinit son comportement
$this
    ->calling($mock)
        ->methodsMatching('/^is/')
            ->return = true
;

// si la méthode commence par "get" (insensible à la casse),
// on redéfinit son comportement
$this
    ->calling($mock)
        ->methodsMatching('/^get/i')
            ->throw = new \exception
;
```

Note : `methodsMatching` utilise preg_match et les expressions rationnelles. Reportez-vous au manuel de PHP pour avoir plus d'informations sur le sujet.

7.3.4 isFluent && returnThis

Défini une méthode fluent (chaînable), ainsi la méthode appelée retourne l'instance de la classe.

```
<?php
        $foo = new \mock\foo();
        $this->calling($foo)->bar = $foo;

        // est identique à
        $this->calling($foo)->bar->isFluent;
        // ou a celui-ci
        $this->calling($foo)->bar->returnThis;
```

7.3.5 doesNothing && doesSomething

Changer le comportement du mock avec `doesNothing`, la méthode retournera simple null.

```
<?php
        class foo {
```

(suite sur la page suivante)

(suite de la page précédente)

```
                public function bar() {
                        return 'baz';
                }
        }

        //
        // in your test
        $foo = new \mock\foo();
        $this->calling($foo)->bar = null;

        // est identique à
        $this->calling($foo)->bar->doesNothing;
        $this->variable($foo->bar())->isNull;

        // restaure le comportement
        $this->calling($foo)->bar->doesSomething;
        $this->string($foo->bar())->isEqualTo('baz');
```

Comme on le voix dans l'exemple, si pour une raison quelconque, vous souhaitez rétablir le comportement de la méthode, utilisez `doesSomething`.

7.3.6 Cas particulier du constructeur

Pour mocker le constructeur de la classe, vous avez besoin de :

- créer une instance de la classe atoummockcontroller avant d'appeler le constructeur du bouchon ;
- définir via ce contrôleur le comportement du constructeur du bouchon à l'aide d'une fonction anonyme ;
- injecter le contrôleur lors de l'instanciation du bouchon en *dernier* argument.

```
<?php
$controller = new \atoum\mock\controller();
$controller->__construct = function($args)
{
     // faire quelque chose avec les arguments
};

$mockDbClient = new \mock\Database\Client(DB_HOST, DB_USER, DB_PASS, $controller);
```

Pour les cas simple, vous pouvez utiliser *orphanize("__constructor")* ou *shunt("__constructor")*.

7.4 Tester un bouchon

atoum vous permet de vérifier qu'un bouchon a été utilisé correctement.

```
<?php
$mockDbClient = new \mock\Database\Client();
$mockDbClient->getMockController()->connect = function() {};
$mockDbClient->getMockController()->query   = array();

$bankAccount = new \Vendor\Project\Bank\Account();
$this
    // utilisation du bouchon via un autre objet
    ->array($bankAccount->getOperations($mockDbClient))
        ->isEmpty()
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // test du bouchon
    ->mock($mockDbClient)
        ->call('query')
            ->once() // vérifie que la méthode query
                            // n'a été appelé qu'une seule fois
;
```

Note : Reportez-vous à la documentation sur l'assertion *mock* pour obtenir plus d'informations sur les tests des bouchons.

7.5 Le bouchonnage (mock) des fonctions natives de PHP

atoum permet de très facilement simuler le comportement des fonctions natives de PHP.

```
<?php

$this
   ->assert('the file exist')
      ->given($this->newTestedInstance())
      ->if($this->function->file_exists = true)
      ->then
      ->object($this->testedInstance->loadConfigFile())
         ->isTestedInstance()
         ->function('file_exists')->wasCalled()->once()

   ->assert('le fichier does not exist')
      ->given($this->newTestedInstance())
      ->if($this->function->file_exists = false )
      ->then
      ->exception(function() { $this->testedInstance->loadConfigFile(); })
;
```

Important : On ne peut pas mettre de \ devant les fonctions à simuler, car atoum s'appuie sur le mécanisme de résolution des espaces de nom de PHP.

Important : Pour la même raison, si une fonction native a déjà été appelée, son bouchonnage sera sans effet.

```
<?php

$this
   ->given($this->newTestedInstance())
   ->exception(function() { $this->testedInstance->loadConfigFile(); }) //la fonction␣
↪file_exists est appelée avant son bouchonnage

   ->if($this->function->file_exists = true ) // le bouchonnage ne pourra pas prendre␣
↪la place de la fonction native file_exists
   ->object($this->testedInstance->loadConfigFile())
```

(suite sur la page suivante)

(suite de la page précédente)

```
        ->isTestedInstance()
;
```

Note : Plus d'information via *isTestedInstance()*.

7.6 Les bouchons de constantes

Les constantes PHP peuvent être déclarées avec `defined`, cependant avec atoum vous pouvez les bouchonner de cette manière :

```
<?php
$this->constant->PHP_VERSION_ID = '606060'; // troll \o/

$this
    ->given($this->newTestedInstance())
    ->then
        ->variable($this->testedInstance->hello())->isEqualTo(PHP_VERSION_ID)
    ->if($this->constant->PHP_VERSION_ID = uniqid())
    ->then
        ->variable($this->testedInstance->hello())->isEqualTo(PHP_VERSION_ID)
;
```

Attention, due à la nature des constantes en PHP, suivant *l'engine* utilisé vous pouvez rencontrer différents problèmes. En voici un exemple :

```
<?php

namespace foo {
    class foo {
        public function hello()
        {
            return PHP_VERSION_ID;
        }
    }
}

namespace tests\units\foo {
    use atoum;

    /**
     * @engine inline
     */
    class foo extends atoum
    {
        public function testFoo()
        {
            $this
                ->given($this->newTestedInstance())
                ->then
                    ->variable($this->testedInstance->hello())->isEqualTo(PHP_VERSION_
↪ID)
                ->if($this->constant->PHP_VERSION_ID = uniqid())
```

(suite sur la page suivante)

(suite de la page précédente)

```
                ->then
                    ->variable($this->testedInstance->hello())->isEqualTo(PHP_VERSION_
↪ID)
            ;
        }

        public function testBar()
        {
            $this
                ->given($this->newTestedInstance())
                ->if($this->constant->PHP_VERSION_ID = $mockVersionId = uniqid()) //␣
↪inline engine will fail here
                ->then
                    ->variable($this->testedInstance->hello())->isEqualTo(
↪$mockVersionId)
                ->if($this->constant->PHP_VERSION_ID = $mockVersionId = uniqid()) //␣
↪isolate/concurrent engines will fail here
                ->then
                    ->variable($this->testedInstance->hello())->isEqualTo(
↪$mockVersionId)
            ;
        }
    }
}
```

CHAPITRE 8

Les moteurs d'exécution

Plusieurs moteurs d'exécution des tests (au niveau de la classe ou des méthodes) existent. Ceci est configuré via l'annotation `@engine`. Par défaut, les différents tests s'exécutent en parallèle, dans des sous-processus PHP, c'est le mode `concurrent`.

Il existe actuellement trois moteurs d'exécution :

- *inline* : les tests s'exécutent dans le même processus, cela revient au même comportement que PHPUnit. Même si ce mode est très rapide, il n'y a pas d'isolation des tests.
- *isolate* : les tests s'exécutent de manière séquentielle dans un sous-processus PHP. Ce mode d'exécution est assez lent.
- *concurrent* : le mode par défaut, les tests s'exécutent en parallèle, dans des sous-processus PHP.

Important : Si vous utilisez xdebug pour déboguer vos tests (et pas seulement pour la couverture de code), le seul moteur d'exécution disponible est *concurrent*.

Voici un exemple :

```
<?php

/**
 * @engine concurrent
 */
class Foo extends \atoum
{
      public function testBarWithBaz()
      {
              sleep(1);
              $this->newTestedInstance;
              $baz = new \Baz();
              $this->object($this->testedInstance->setBaz($baz))
                      ->isIdenticalTo($this->testedInstance);

              $this->string($this->testedInstance->bar())
                      ->isIdenticalTo('baz');
```

(suite sur la page suivante)

(suite de la page précédente)

```
        }

        public function testBarWithoutBaz()
        {
                sleep(1);
                $this->newTestedInstance;
                $this->string($this->testedInstance->bar())
                        ->isIdenticalTo('foo');
        }
}
```

En mode `concurent` :

```
=> Test duration: 2.01 seconds.
=> Memory usage: 0.50 Mb.
> Total test duration: 2.01 seconds.
> Total test memory usage: 0.50 Mb.
> Running duration: 1.08 seconds.
```

En mode `inline` :

```
=> Test duration: 2.01 seconds.
=> Memory usage: 0.25 Mb.
> Total test duration: 2.01 seconds.
> Total test memory usage: 0.25 Mb.
> Running duration: 2.01 seconds.
```

En mode `isolate` :

```
=> Test duration: 2.00 seconds.
=> Memory usage: 0.50 Mb.
> Total test duration: 2.00 seconds.
> Total test memory usage: 0.50 Mb.
> Running duration: 2.10 seconds.
```

CHAPITRE 9

Mode répétition

Lorsqu'un développeur fait du développement piloté par les tests, il travaille généralement de la manière suivante :

1. il commence par créer le test correspondant à ce qu'il veut développer ;
2. il exécute ensuite le test qu'il vient de créer ;
3. il écrit le code permettant au test de passer avec succès ;
4. il modifie ou complète son test et repars à l'étape 2.

Concrètement, cela signifie qu'il doit :

— créer son code dans son éditeur favori ;
— quitter son éditeur puis exécuter son test dans une console ;
— revenir à son éditeur pour écrire le code permettant au test de passer avec succès ;
— revenir à la console afin de relancer l'exécution de son test ;
— revenir à son éditeur afin de modifier ou compléter son test ;

Il y a donc un cycle qui se répète jusqu'à ce que la fonctionnalité soit terminée.

Au cours de ce cycle, le développeur doit à plusieurs reprises exécuter la même commande pour exécuter les tests unitaires.

atoum propose le mode `loop` disponible via les arguments `-l` ou `--loop`, qui permet au développeur de ne pas avoir à relancer manuellement les tests et permet donc de fluidifier le processus de développement.

Dans ce mode, atoum exécute les tests demandés.

Une fois les tests terminés, si les tests ont été passés avec succès, atoum se met simplement en attente :

```
$ php tests/units/classes/adapter.php -l
> PHP path: /usr/bin/php
> PHP version:
=> PHP 5.6.3 (cli) (built: Nov 13 2014 18:31:57)
=> Copyright (c) 1997-2014 The PHP Group
=> Zend Engine v2.6.0, Copyright (c) 1998-2014 Zend Technologies
> mageekguy\atoum\tests\units\adapter...
[SS_____________________________________________________________][2/2]
=> Test duration: 0.00 second.
=> Memory usage: 0.50 Mb.
```

(suite sur la page suivante)

(suite de la page précédente)

```
> Total test duration: 0.00 second.
> Total test memory usage: 0.50 Mb.
> Running duration: 0.05 second.
Success (1 test, 2/2 methods, 0 void method, 0 skipped method, 4 assertions)!
Press <Enter> to reexecute, press any other key and <Enter> to stop...
```

Si le développeur presse la touche `Enter`, atoum réexécutera à nouveau les mêmes tests, sans aucune autre action de la part du développeur.

Dans le cas où le code ne passe pas les tests avec succès, c'est-à-dire si les assertions échouent ou s'il y a des erreurs ou des exceptions, atoum se remet en mode d'attente :

```
$ php tests/units/classes/adapter.php -l> PHP path: /usr/bin/php
> PHP version:
=> PHP 5.6.3 (cli) (built: Nov 13 2014 18:31:57)
=> Copyright (c) 1997-2014 The PHP Group
=> Zend Engine v2.6.0, Copyright (c) 1998-2014 Zend Technologies
> mageekguy\atoum\tests\units\adapter...
[FS_____________________________________________________________][2/2]
=> Test duration: 0.00 second.
=> Memory usage: 0.25 Mb.
> Total test duration: 0.00 second.
> Total test memory usage: 0.25 Mb.
> Running duration: 0.05 second.
Failure (1 test, 2/2 methods, 0 void method, 0 skipped method, 0 uncompleted method,␣
↪1 failure, 0 error, 0 exception)!
> There is 1 failure:
=> mageekguy\atoum\tests\units\adapter::test__call():
In file /media/data/dev/atoum-documentation/tests/vendor/atoum/atoum/tests/units/
↪classes/adapter.php on line 16, mageekguy\atoum\asserters\string() failed: strings␣
↪are not equal
-Expected
+Actual
@@ -1 +1 @@
-string(32) "1305beaa8f3f2f932f508d4af7f89094"
+string(32) "d905c0b86bf89f9a57d4da6101f93648"
Press <Enter> to reexecute, press any other key and <Enter> to stop...
```

Si le développeur presse la touche `Enter`, au lieu de rejouer les mêmes tests, atoum n'exécutera que les tests en échec, au lieu de rejouer l'ensemble.

Le développeur pourra alors dépiler les problèmes et rejouer les tests en erreur autant de fois que nécessaire simplement en appuyant sur `Enter`.

De plus, une fois que tous les tests en échec passeront à nouveau avec succès, atoum exécutera automatiquement la totalité de la suite de tests afin de détecter les éventuelles régressions introduites par la ou les corrections effectuées par le développeur.

```
Press <Enter> to reexecute, press any other key and <Enter> to stop...
> PHP path: /usr/bin/php
> PHP version:
=> PHP 5.6.3 (cli) (built: Nov 13 2014 18:31:57)
=> Copyright (c) 1997-2014 The PHP Group
=> Zend Engine v2.6.0, Copyright (c) 1998-2014 Zend Technologies
> mageekguy\atoum\tests\units\adapter...
[S_____________________________________________________________][1/1]
=> Test duration: 0.00 second.
```

(suite sur la page suivante)

(suite de la page précédente)

```
=> Memory usage: 0.25 Mb.
> Total test duration: 0.00 second.
> Total test memory usage: 0.25 Mb.
> Running duration: 0.05 second.
Success (1 test, 1/1 method, 0 void method, 0 skipped method, 2 assertions)!
> PHP path: /usr/bin/php
> PHP version:
=> PHP 5.6.3 (cli) (built: Nov 13 2014 18:31:57)
=> Copyright (c) 1997-2014 The PHP Group
=> Zend Engine v2.6.0, Copyright (c) 1998-2014 Zend Technologies
> mageekguy\atoum\tests\units\adapter...
[SS_____________________________________________________________][2/2]
=> Test duration: 0.00 second.
=> Memory usage: 0.50 Mb.
> Total test duration: 0.00 second.
> Total test memory usage: 0.50 Mb.
> Running duration: 0.05 second.
Success (1 test, 2/2 methods, 0 void method, 0 skipped method, 4 assertions)!
Press <Enter> to reexecute, press any other key and <Enter> to stop...
```

Bien évidemment, le mode `loop` ne prend en compte que *le ou les fichiers de tests unitaires lancés* par atoum.

CHAPITRE 10

Débogage des scénarios de test

Parfois, un test ne passe pas et il est difficile d'en découvrir la raison.

Dans ce cas, l'une des techniques possibles pour remédier au problème est de tracer le comportement du code concerné, soit directement au cœur de la classe testée à l'aide d'un débogueur ou de fonctions du type de `var_dump()` ou `print_r()`, soit au niveau du test unitaire.

atoum fourni un certain nombre d'outils pour faciliter la tâche de débogage directement dans les tests unitaires.

Ces outils ne sont cependant actifs que lorsqu'atoum est exécuté à l'aide de l'argument `--debug`, afin que l'exécution des tests unitaires ne soit pas perturbée par les instructions relatives au débogage hors de ce contexte. Lorsque l'argument `--debug` est utilisé, trois méthodes peuvent être activée :

- `dump()` qui permet de connaître le contenu d'une variable ;
- `stop()` qui permet d'arrêter l'exécution d'un test ;
- `executeOnFailure()` qui permet de définir une fonction anonyme qui ne sera exécutée qu'en cas d'échec d'une assertion.

Ces trois méthodes s'intègrent parfaitement dans l'interface fluide qui caractérise atoum.

10.1 dump

La méthode `dump()` peut s'utiliser de la manière suivante :

```
<?php
$this
    ->if($foo = new foo())
    ->then
        ->object($foo->setBar($bar = new bar()))
            ->isIdenticalTo($foo)
        ->dump($foo->getBar())
;
```

Lors de l'exécution du test, le retour de la méthode `foo::getBar()` sera affiché sur la sortie standard.

Il est également possible de passer plusieurs arguments à `dump()`, de la manière suivante :

```
<?php
$this
    ->if($foo = new foo())
    ->then
        ->object($foo->setBar($bar = new bar()))
            ->isIdenticalTo($foo)
        ->dump($foo->getBar(), $bar)
;
```

Important : La méthode `dump` n'est activée que si vous lancez les tests avec l'argument `--debug`. Dans le cas contraire, cette méthode sera totalement ignorée.

10.2 stop

L'utilisation de la méthode `stop()` est également très simple :

```
<?php
$this
    ->if($foo = new foo())
    ->then
        ->object($foo->setBar($bar = new bar()))
            ->isIdenticalTo($foo)
        ->stop() // le test s'arrêtera ici si --debug est utilisé
        ->object($foo->getBar())
            ->isIdenticalTo($bar)
;
```

Si `--debug` est utilisé, les 2 dernières lignes ne seront pas exécutées.

Important : La méthode `stop` n'est activée que si vous lancez les tests avec l'argument `--debug`. Dans le cas contraire, cette méthode sera totalement ignorée.

10.3 executeOnFailure

La méthode `executeOnFailure()` est très puissante et tout aussi simple à utiliser.

Elle prend en effet en argument une fonction anonyme qui sera exécutée si et seulement si l'une des assertions composant le test n'est pas vérifiée. Elle s'utilise de la manière suivante :

```
<?php
$this
    ->if($foo = new foo())
    ->executeOnFailure(
        function() use ($foo) {
            var_dump($foo);
        }
    )
    ->then
        ->object($foo->setBar($bar = new bar()))
```

(suite sur la page suivante)

(suite de la page précédente)

```
            ->isIdenticalTo($foo)
        ->object($foo->getBar())
            ->isIdenticalTo($bar)
;
```

Dans l'exemple précédent, contrairement à `dump()` qui provoque systématiquement l'affichage sur la sortie standard le contenu des variables qui lui sont passées en argument, la fonction anonyme passée en argument ne provoquera l'affichage du contenu de la variable `foo` que si l'une des assertions suivantes est en échec.

Bien évidemment, il est possible de faire appel plusieurs fois à `executeOnFailure()` dans une même méthode de test pour définir plusieurs fonctions anonymes différentes devant être exécutées en cas d'échec du test.

Important : La méthode `executeOnFailure` n'est activée que si vous lancez les tests avec l'argument `--debug`. Dans le cas contraire, cette méthode sera totalement ignorée.

CHAPITRE 11

Ajustement du comportement d'atoum

11.1 Les méthodes d'initialisation

Voici le processus, lorsque atoum exécute les méthodes de test d'une classe avec le moteur par défaut (`concurrent`) :

1. appel de la méthode `setUp()` de la classe de test ;
2. lancement d'un sous-processus PHP pour exécuter **chaque méthode** de test ;
3. dans le sous-processus PHP, appel de la méthode `beforeTestMethod()` de la classe de test ;
4. dans le sous-processus PHP, appel de la méthode de test ;
5. dans le sous-processus PHP, appel de la méthode `afterTestMethod()` de la classe de test ;
6. une fois le sous-processus PHP terminé, appel de la méthode `tearDown()` de la classe de test.

Note : Pour plus d'informations sur les moteurs d'exécution des tests d'atoum, vous pouvez lire le paragraphe sur l'annotation *@engine*.

Les méthodes `setUp()` et `tearDown()` permettent donc respectivement d'initialiser et de nettoyer l'environnement de test pour l'ensemble des méthodes de test de la classe exécutée.

Les méthodes `beforeTestMethod()` et `afterTestMethod()` permet respectivement d'initialiser et de nettoyer l'environnement d'exécution des tests individuels pour toutes les méthodes de test de la classe. À l'opposé `setUp()` et `tearDown()`, sont exécutées dans le sous-processus lui-même.

C'est d'ailleurs la raison pour laquelle les méthodes `beforeTestMethod()` et `afterTestMethod()` acceptent comme argument le nom de la méthode de test exécutée, afin de pouvoir ajuster les traitements en conséquence.

```
<?php
namespace vendor\project\tests\units;

use
    mageekguy\atoum,
    vendor\project
;
```

(suite sur la page suivante)

(suite de la page précédente)

```
require __DIR__ . '/atoum.phar';

class bankAccount extends atoum
{
    public function setUp()
    {
        // Exécutée *avant l'ensemble* des méthodes de test.
        // Initialisation globale.
    }

    public function beforeTestMethod($method)
    {
        // Exécutée *avant chaque* méthode de test.

        switch ($method)
        {
            case 'testGetOwner':
                // Initialisation pour testGetOwner().
            break;

            case 'testGetOperations':
                // Initialisation pour testGetOperations().
            break;
        }
    }

    public function testGetOwner()
    {
        // ...
    }

    public function testGetOperations()
    {
        // ...
    }

    public function afterTestMethod($method)
    {
        // Exécutée *après chaque* méthode de test.

        switch ($method)
        {
            case 'testGetOwner':
                // Nettoyage pour testGetOwner().
            break;

            case 'testGetOperations':
                // Nettoyage pour testGetOperations().
            break;
        }
    }

    public function tearDown()
    {
        // Exécutée après l'ensemble des méthodes de test.
        // Nettoyage global.
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

Par défaut, les méthodes `setUp()`, `beforeTestMethod()`, `afterTestMethod()` et `tearDown()` ne font absolument rien.

Il est donc de la responsabilité du programmeur de les surcharger lorsque c'est nécessaire dans les classes de test concerné.

CHAPITRE 12

Configuration & bootstraping

Plusieurs étapes vont se succéder au lancement d'atoum, certaines d'entre elles peuvent être influencées par des fichiers spéciaux.

On peut avoir une vue simplifiée de ces fichiers spéciaux et *optionnelle* en :

1. Chargement de l'"*autoloader*
2. Chargement du *fichier de configuration*
3. Chargement du *fichier de bootstrap*

Note : Vous pouvez utiliser atoum *–init* pour générer ces fichiers.

12.1 L'autoloader

Le fichier d'autoload (autoloader) est ce que vous allez utiliser pour définir comment atoum va trouver la classe à tester.

Le nom du fichier par défaut est `.autoloader.atoum.php`. atoum le chargera automatiquement s'il se trouve dans le dossier courant. Vous pouvez le définir dans la ligne de commande avec `--autoloader-file` ou `-af` (*voir les options de la ligne de commande*). L'objectif du fichier d'autoloader est de permettre de charger les classes nécessaire pour exécuter les tests. Vous pouvez trouver plus d'informations sur l'auto-chargement des classes dans le manuel php.

Si l'autoloader n'existe pas, atoum essayera de charger le fichier `vendor/autoload.php` de composer. Vous n'aurez donc rien à faire dans la majorité des cas. ;).

12.2 Fichier de configuration

Le fichier de configuration vous permet de configurer comment atoum fonctionne. Si vous nommez votre fichier de configuration `.atoum.php`, atoum le chargera automatiquement si ce fichier se trouve dans le répertoire courant. Le paramètre `-c` est donc optionnel dans ce cas.

Note : La configuration de atoum supporte l'héritage de fichier. Si vous avez un fichier `.atoum.php` dans le répertoire parent, il sera également chargé. Vous pouvez ainsi avoir un fichier de configuration par défaut afin d'avoir le mode loop ou debug activé par défaut.

12.2.1 Exemple existant

atoum fourni un fichier d'exemple basique. Suivant le type d'installation de atoum, il y a plusieurs façons de les voir :

Depuis une installation PHAR

Si vous utlisez l'archive PHAR, il faut les extraire en utilisant la commande suivante :

```
php atoum.phar -er /path/to/destination/directory
```

Une fois l'extraction effectuée, vous devriez avoir dans le répertoire /path/to/destination/directory un répertoire nommé resources/configurations/runner.

Depuis une installation composer

Dans le cas où vous utilisez atoum en ayant cloné le dépôt *Github* ou l'ayant installé via *Composer*, les modèles se trouvent dans `/path/to/atoum/resources/configurations/runner`

12.2.2 Couverture du code

Par défaut, si PHP dispose de l'extension Xdebug, atoum indique en ligne de commande le taux de couverture du code par les tests venant d'être exécutés. Certains comportements de la couverture de code peuvent être adaptés via les *options de l'interface en ligne de commande*.

Si le taux de couverture est de 100%, atoum se contente de l'indiquer. Mais dans le cas contraire, il affiche le taux de couverture globale ainsi que celui de chaque méthode de la classe testée sous la forme d'un pourcentage.

```
$ php tests/units/classes/template.php
> atoum version DEVELOPMENT by Frederic Hardy (/Users/fch/Atoum)
> PHP path: /usr/local/bin/php
> PHP version:
=> PHP 5.3.8 (cli) (built: Sep 21 2011 23:14:37)
=> Copyright (c) 1997-2011 The PHP Group
=> Zend Engine v2.3.0, Copyright (c) 1998-2011 Zend Technologies
=>     with Xdebug v2.1.1, Copyright (c) 2002-2011, by Derick Rethans
> mageekguy\atoum\tests\units\template...
[SSSSSSSSSSSSSSSSSSSSSSSSSS__________________________________][27/27]
=> Test duration: 15.63 seconds.
=> Memory usage: 8.25 Mb.
> Total test duration: 15.63 seconds.
> Total test memory usage: 8.25 Mb.
> Code coverage value: 92.52%
=> Class mageekguy\atoum\template: 91.14%
==> mageekguy\atoum\template::setWith(): 80.00%
==> mageekguy\atoum\template::resetChildrenData(): 25.00%
==> mageekguy\atoum\template::addToParent(): 0.00%
==> mageekguy\atoum\template::unsetAttribute(): 0.00%
```

(suite sur la page suivante)

(suite de la page précédente)

```
=> Class mageekguy\atoum\template\data: 96.43%
==> mageekguy\atoum\template\data::__toString(): 0.00%
> Running duration: 2.36 seconds.
Success (1 test, 27 methods, 485 assertions, 0 error, 0 exception) !
```

Il est cependant possible d'obtenir une représentation plus précise du taux de couverture du code par les tests, sous la forme d'un rapport au format HTML. Cela peut être trouver dans l'extension report.

Rapport de couverture personnalisée

Dans ce répertoire, il y a, entre autre chose intéressante, un modèle de fichier de configuration pour atoum nommé `coverage.php.dist` qu'il vous faudra copier à l'emplacement de votre choix. Renommez le `coverage.php`.

Une fois le fichier copié, il n'y a plus qu'à le modifier à l'aide de l'éditeur de votre choix afin de définir le répertoire dans lequel les fichiers HTML devront être générés ainsi que l'URL à partir de laquelle le rapport devra être accessible.

Par exemple :

```
$coverageField = new atoum\report\fields\runner\coverage\html(
    'Code coverage de mon projet',
    '/path/to/destination/directory'
);

$coverageField->setRootUrl('http://url/of/web/site');
```

Note : Il est également possible de modifier le titre du rapport à l'aide du premier argument du constructeur de la classe `mageekguy\atoum\report\fields\runner\coverage\html`.

Une fois ceci en place, vous avez simplement a utiliser le fichier de configuration (ou l'inclure dans le fichier de configuration) lorsque vous lancer les tests, comme ceci :

```
$ ./bin/atoum -c path/to/coverage.php -d tests/units
```

Une fois les tests exécutés, atoum génèrera alors le rapport de couverture du code au format HTML dans le répertoire que vous aurez défini précédemment, et il sera lisible à l'aide du navigateur de votre choix.

Note : Le calcul du taux de couverture du code par les tests ainsi que la génération du rapport correspondant peuvent ralentir de manière notable l'exécution des tests. Il peut être alors intéressant de ne pas utiliser systématiquement le fichier de configuration correspondant, ou bien de les désactiver temporairement à l'aide de l'argument -ncc.

12.2.3 Utilisation de rapports standards

atoum est fourni avec de nombreux rapports standards : tap, xunit, html, cli, phing, vim, ... Il y a aussi quelques *rapports funs*. Vous trouverez les plus importants ici.

Note : Si vous souhaitez aller plus loin, il y a une extension dédiée aux rapports appelée `reports-extension`.

Configuration de rapports

Couverture des branches et chemins

Dans le fichier de configuration, vous pouvez activer la couverture des branches et chemins à l'aide de l'option `enableBranchAndPathCoverage`. Cette action améliorera la qualité de la couverture du code car elle ne se limitera pas à vérifier qu'une fonction est appelée, mais également que chaque branche l'est également. Pour faire simple, si vous avez un `if`, le rapport changera si vous cherchez le `else`. Vous pouvez aussi l'activer via la ligne commande avec *l'option –epbc*.

```
$script->enableBranchAndPathCoverage();
```

```
=> Class Foo\Bar: Line: 31.46%
# avec la couverture des branches et chemins
=> Class Foo\Bar: Line: 31.46% Path: 1.50% Branch: 26.06%
```

Désactiver la couverture pour une classe

Si vous souhaitez exclure certaines classes de la couverture de code, vous pouvez utiliser `$script->noCodeCoverageForClasses(\myClass::class)`.

Rapport HTML

Par défaut, atoum fournit un rapport HTML basique. Pour un rapport plus avancé, vous pouvez utiliser reports-extension.

```
<?php
$report = $script->addDefaultReport();
$coverageField = new atoum\report\fields\runner\coverage\html('Your Project Name', __
↪DIR__ . '/reports');
// Remplacez cette url par l'url racine de votre site de couverture de code.
$coverageField->setRootUrl('http://url/of/web/site');
$report->addField($coverageField);
```

Rapport CLI

Le rapport CLI est celui qui s'affiche quand vous lancez le test. Ce rapport a quelques options de configuration disponibles

- hideClassesCoverageDetails : Désactive la couverture d'une classe.
- hideMethodsCoverageDetails : Désactive la couverture d'une méthode.

```
<?php
$script->addDefaultReport() // les rapports par défaut incluent celui-ci
    ->hideClassesCoverageDetails()
    ->hideMethodsCoverageDetails();
```

Afficher le logo d'atoum

```
<?php
$report = $script->addDefaultReport();

// Cette ligne ajoute le logo d'atoum à chaque exécution
$report->addField(new atoum\report\fields\runner\atoum\logo());

// Celle-ci va ajouter un logo vert ou rouge après chaque exécution en fonction du
↪status de cette dernière
$report->addField(new atoum\report\fields\runner\result\logo());
```

Rapport Treemap

```
<?php
$report = $script->addDefaultReport();

$coverageHtmlField = new atoum\report\fields\runner\coverage\html('Your Project Name',
↪ __DIR__ . '/reports');
// Remplacez cette url par l'url racine de votre site de couverture de code.
$coverageHtmlField->setRootUrl('http://url/of/web/site');
$report->addField($coverageField);

$coverageTreemapField = new atoum\report\fields\runner\coverage\treemap('Your project
↪name', __DIR__ . '/reports');
$coverageTreemapField
   ->setTreemapUrl('http://url/of/treemap')
   ->setHtmlReportBaseUrl($coverageHtmlField->getRootUrl());

$report->addField($coverageTreemapField);
```

12.2.4 Notifications

atoum est capable de vous prévenir lorsque les tests sont exécutés en utilisant plusieurs systèmes de notification : *Growl*, *Mac OS X Notification Center*, *Libnotify*.

Growl

Cette fonctionnalité nécessite la présence de l'exécutable `growlnotify`. Pour vérifier s'il est disponible, utilisez la commande suivante :

```
$ which growlnotify
```

Vous aurez alors le chemin de l'exécutable ou alors le message `growlnotify not found` s'il n'est pas installé.

Il suffit ensuite d'ajouter le code suivant à votre fichier de configuration :

```
<?php
$images = '/path/to/atoum/resources/images/logo';

$notifier = new \mageekguy\atoum\report\fields\runner\result\notifier\image\growl();
$notifier
    ->setSuccessImage($images . DIRECTORY_SEPARATOR . 'success.png')
    ->setFailureImage($images . DIRECTORY_SEPARATOR . 'failure.png')
;
```

(suite sur la page suivante)

(suite de la page précédente)

```
$report = $script->AddDefaultReport();
$report->addField($notifier, array(atoum\runner::runStop));
```

Mac OS X Notification Center

Cette fonctionnalité nécessite la présence de l'exécutable `terminal-notifier`. Pour vérifier s'il est disponible, utilisez la commande suivante :

```
$ which terminal-notifier
```

Vous aurez alors le chemin de l'exécutable ou alors le message `terminal-notifier not found` s'il n'est pas installé.

Note : Rendez-vous sur la page Github du projet pour obtenir plus d'information sur l'installation de `terminal-notifier`.

Il suffit ensuite d'ajouter le code suivant à votre fichier de configuration :

```
<?php
$notifier = new \mageekguy\atoum\report\fields\runner\result\notifier\terminal();

$report = $script->AddDefaultReport();
$report->addField($notifier, array(atoum\runner::runStop));
```

Sous OS X, vous avez la possibilité de définir une commande qui sera exécutée lorsque l'utilisateur cliquera sur la notification.

```
<?php
$coverage = new atoum\report\fields\runner\coverage\html(
    'Code coverage',
    $path = sys_get_temp_dir() . '/coverage_' . time()
);
$coverage->setRootUrl('file://' . $path);

$notifier = new \mageekguy\atoum\report\fields\runner\result\notifier\terminal();
$notifier->setCallbackCommand('open file://' . $path . '/index.html');

$report = $script->AddDefaultReport();
$report
    ->addField($coverage, array(atoum\runner::runStop))
    ->addField($notifier, array(atoum\runner::runStop))
;
```

L'exemple ci-dessus montre comment ouvrir le rapport de couverture du code lorsque l'utilisateur clique sur la notification.

Libnotify

Cette fonctionnalité nécessite la présence de l'exécutable `notify-send`. Pour vérifier s'il est disponible, utilisez la commande suivante :

```
$ which notify-send
```

Vous aurez alors le chemin de l'exécutable ou alors le message `notify-send not found` s'il n'est pas installé.

Il suffit ensuite d'ajouter le code suivant à votre fichier de configuration :

```
<?php
$images = '/path/to/atoum/resources/images/logo';

$notifier = new␣
↪\mageekguy\atoum\report\fields\runner\result\notifier\image\libnotify();
$notifier
    ->setSuccessImage($images . DIRECTORY_SEPARATOR . 'success.png')
    ->setFailureImage($images . DIRECTORY_SEPARATOR . 'failure.png')
;

$report = $script->AddDefaultReport();
$report->addField($notifier, array(atoum\runner::runStop));
```

12.2.5 Configuration du test

De nombreuses possibilités sont disponibles pour configurer comment atoum va exécuter le test. Vous pouvez utiliser les arguments en ligne de commande ou le fichier de configuration. Un code simple valant une longue explication, l'exemple suivant devrait être explicite :

```
<?php
$testGenerator = new atoum\test\generator();

// répertoire contenant le test unitaire. (-d)
$testGenerator->setTestClassesDirectory(__DIR__ . '/test/units');

// le namespace du test unitaire.
$testGenerator->setTestClassNamespace('your\project\namespace\tests\units');

// le runner de votre test unitaire.
$testGenerator->setRunnerPath('path/to/your/tests/units/runner.php');

$script->getRunner()->setTestGenerator($testGenerator);
// ou
$runner->setTestGenerator($testGenerator);
```

Vous pouvez également définir le répertoire du test avec `$runner->addTestsFromDirectory(path)`. atoum chargera toutes les classes qui puissent être testées présentes dans ce dossier tout comme vous pouvez faire avec l'argument en ligne de commande *-d*.

```
<?php
$runner->addTestsFromDirectory(__DIR__ . '/test/units');
```

12.3 Fichier de bootstrap

atoum autorise la définition d'un fichier de `bootstrap` qui sera exécuté avant chaque méthode de test et qui permet donc d'initialiser l'environnement d'exécution des tests.

Le nom par défaut du fichier est `.bootstrap.atoum.php`, atoum chargera le fichier automatiquement, si celui-ci est situé dans le répertoire courant. Vous pouvez le définir en cli avec `-bf` ou `--bootstrap-file`.

Il devient ainsi possible de définir, par exemple, de lire un fichier de configuration ou de réaliser toute autre opération nécessaire à la bonne exécution des tests.

La définition de ce fichier de bootstrap peut se faire de deux façons différentes, soit en ligne de commande, soit via un fichier de configuration.

```
$ ./bin/atoum -bf path/to/bootstrap/file
```

Note : Le fichier de bootstrap n'est pas un *fichier de configuration* et , n'as pas les même possibilités.

Dans un fichier de configuration, atoum est configurable via la variable `$runner`, qui n'est pas définie dans un fichier de `bootstrap`.

De plus, ils ne sont pas inclus au même moment, puisque le fichier de configuration est inclus par atoum avant le début de l'exécution des tests mais après le lancement des tests, alors que le fichier de `bootstrap`, s'il est défini, est le tout premier fichier inclus par atoum proprement dit. Enfin, le fichier de `bootstrap` peut permettre de ne pas avoir à inclure systématiquement le fichier `scripts/runner.php` ou l'archive PHAR de atoum dans les classes de test. Cependant, dans ce cas, il ne sera plus possible d'exécuter directement un fichier de test directement via l'exécutable PHP en ligne de commande. Pour cela, il suffit d'inclure dans le fichier de `bootstrap` le fichier `scripts/runner.php` ou l'archive PHAR d'atoum et d'exécuter systématiquement les tests en ligne de commande via `scripts/runner.php` ou l'archive PHAR.

Le fichier de `bootstrap` doit donc au minimum contenir ceci :

```php
<?php

// si l'archive PHAR est utilisée :
require_once path/to/atoum.phar;

// ou si vous voulez charger le $runner
// require_once path/atoum/scripts/runner.php
```

12.4 Amusons-nous avec atoum

12.4.1 Rapport

Les rapports de tests peuvent être décorés afin d'être plus agréables ou sympa à lire. Pour cela, dans le *fichier de configuration* d'atoum, ajoutez le code suivant

```php
<?php
// Le fichier de configuration par défaut est .atoum.php
// ...

$stdout = new \mageekguy\atoum\writers\std\out;
$report = new \mageekguy\atoum\reports\realtime\nyancat;
$script->addReport($report->addWriter($stdout));
```

Vous pouvez aussi essayer `\mageekguy\atoum\reports\realtime\santa` comme rapport ;)

CHAPITRE 13

Annotations

Dans cette section nous listons toutes les annotations utilisables avec atoum.

13.1 Annotation de classe

- *@dataProvider*
- *@extensions*
- @hasNotVoidMethods
- @hasVoidMethods
- @ignore
- @maxChildrenNumber
- @methodPrefix
- *@namespace*
- *@php*
- *@tags*

13.2 Annotation des méthodes

- *@dataProvider*
- *@engine*
- *@extensions*
- @ignore
- @isNotVoid
- @isVoid
- *@php*
- *@tags*

13.3 Data providers

Afin de permettre de tester efficacement vos classes, atoum fourni des data provider (fournisseur de données).

Un data provider est une méthode spécifique d'une classe de test chargée de générer des arguments pour une méthode de test, arguments qui seront utilisés par ladite méthode pour valider des assertions.

Si méthode de test `testFoo` prend des arguments, mais qu'aucune annotation précisant le data provider n'est présente, atoum va automatiquement rechercher une méthode `testFooDataProvider`.

Vous pouvez également définir manuellement le nom de la méthode du data provider grâce à l'annotation `@dataProvider` à ajouter à la méthode de test :

```
<?php
class calculator extends atoum
{
    /**
     * @dataProvider sumDataProvider
     */
    public function testSum($a, $b)
    {
        $this
            ->if($calculator = new project\calculator())
            ->then
                ->integer($calculator->sum($a, $b))->isEqualTo($a + $b)
        ;
    }

    // ...
}
```

Bien évidemment, il faut penser a définir les arguments de la méthode de test qui vont recevoir les données retournées par le data provider. Dans le cas contraire, atoum va retourner des erreurs lors de l'exécution des tests.

Un data provider est une méthode protégée qui retourne un tableau ou un itérateur qui contient de simples valeurs :

```
<?php
class calculator extends atoum
{
    // ...

    // Fourni des données pour testSum().
    protected function sumDataProvider()
    {
        return array(
            array( 1, 1),
            array( 1, 2),
            array(-1, 1),
            array(-1, 2),
        );
    }
}
```

Lors de l'exécution des test, atoum va appeler la méthode `testSum()` avec les arguments `(1, 1)`, `(1, 2)`, `(-1, 1)` et `(-1, 2)`, tels que retournés par la méthode `sumDataProvider()`.

Avertissement : L'isolation des tests ne sera pas utilisée dans ce cas d'utilisation, ce qui signifie que les appels successifs à la méthode `testSum()` seront exécutés dans le même processus PHP.

13.3.1 Data provider en tant que closure

Vous pouvez également utiliser une closure pour définir un data provider au lieu d'une annotation. Dans votre méthode *beforeTestMethod*, vous pouvez utiliser l'exemple suivant pour définir une closure :

```
<?php
class calculator extends atoum
{
   // ...
   public function beforeTestMethod($method)
   {
      if ($method == 'testSum')
      {
         $this->setDataProvider($method, function() {
           return array(
               array( 1, 1),
               array( 1, 2),
               array(-1, 1),
               array(-1, 2),
           );
         });
      }
   }
}
```

13.3.2 Data provider injecté dans les méthode de test

Il y a aussi, une injection de bouchon dans les paramètres de la méthode de test. Prenons un exemple simple :

```
<?php
class cachingIterator extends atoum
{
    public function test__construct()
    {
        $this
            ->given($iterator = new \mock\iterator())
            ->then
                ->object($this->newTestedInstance($iterator))
        ;
    }
}
```

Vous pouvez l'écrire ainsi :

```
<?php

class cachingIterator extends atoum
{
    public function test__construct(\iterator $iterator)
    {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        $this
            ->object($this->newTestedInstance($iterator))
        ;
    }
}
```

Dans ce cas, pas besoin de data provider. Cependant, si vous désirez changer le comportement de vos bouchons, cela requiert l'utilisation de *beforeTestMethod*.

```
<?php

class cachingIterator extends atoum
{
    public function test__construct(\iterator $iterator)
    {
        $this
            ->object($this->newTestedInstance($iterator))
        ;
    }

    public function beforeTestMethod($method)
    {
        // rend le controlleur orphelin pour le prochain mock généré, ici $iterator
        $this->mockGenerator->orphanize('__construct');
    }
}
```

13.4 PHP Extensions

Certains des tests peuvent requérir une ou plusieurs extensions PHP. atoum permet de définir cela directement à travers une annotation `@extensions`. Après l'annotation `@extensions`, ajouter simplement le nom d'une ou plusieurs extensions, séparés par une virgule.

```
<?php

namespace vendor\project\tests\units;

class foo extends \atoum
{
    /**
     * @extensions intl
     */
    public function testBar()
    {
        // ...
    }
}
```

Le test ne sera exécuté que si l'extension intl est présente. Dans le cas contraire, le test sera passé et le message suivant sera affiché.

```
vendor\project\tests\units\foo::testBar(): PHP extension 'intl' is not loaded
```

Note : Par défaut, le test est validé lorsqu'il a été passé. Mais vous pouvez utiliser *–fail-if-skipped-methods* l'option de la ligne de commande afin de faire échouer les tests passés.

13.5 PHP Version

Certains de vos tests peuvent requérir une version spécifique de PHP pour fonctionner (par exemple, pour certains tests ne fonctionnant qu'avec PHP 7). Dire à atoum qu'un test requiert une version spécifique de PHP s'effectue au travers de l'annotation `@php`.

Par défaut, sans fournir d'opérateur, le test ne sera exécuté que si la version de PHP est supérieure ou égale à la version du tag :

```
class testedClassname extends atoum\test
{
   /**
    * @php 7.0
    */
   public function testMethod()
   {
      // contenu du test
   }
}
```

Dans cette exemple, le test ne sera exécuté que si la version de PHP est supérieure ou égale à PHP 7.0. Dans le cas contraire, le test sera passé et le message suivant sera affiché :

```
vendor\project\tests\units\foo::testBar(): PHP version 5.5.9-1ubuntu4.5 is not >= to␣
↪7.0
```

Note : Par défaut, le test est considéré valide lorsqu'il est passé. Mais vous pouvez utiliser *–fail-if-skipped-methods* l'option de la ligne de commande afin de faire échouer les tests passés.

En interne, atoum utilise le comparateur de version de PHP pour effectuer la comparaison.

Vous n'êtes pas limité à l'opérateur égal ou supérieur. Vous pouvez passer tout les opérateurs acceptés par version_compare.

Par exemple :

```
class testedClassname extends atoum\test
{
   /**
    * @php < 5.4
    */
   public function testMethod()
   {
      // contenu du test
   }
}
```

Va passer le test si la version de PHP est supérieure ou égale à PHP 5.4

```
vendor\project\tests\units\foo::testBar(): PHP version 5.5.9-1ubuntu4.5 is not < to 5.
↪4
```

Vous pouvez aussi utiliser de multiples conditions, avec l'annotation `@php`. Par exemple :

```
class testedClassname extends atoum\test
{
   /**
    * @php >= 5.4
    * @php <= 7.0
    */
   public function testMethod()
   {
      // contenu du test
   }
}
```

CHAPITRE 14

Option de la ligne de commande

La plupart des options existent sous deux formes, une courte de 1 à 6 caractères et une longue, plus explicative. Les deux formes différentes font exactement la même chose et peuvent être utilisés indifféremment.

Certaines options acceptent plusieurs valeurs :

```
$ ./bin/atoum -f tests/units/MyFirstTest.php tests/units/MySecondTest.php
```

Note : Si vous utilisez une option plusieurs fois, seul la dernière servira.

```
# Ne test que MySecondTest.php
$ ./bin/atoum -f MyFirstTest.php -f MySecondTest.php

# Ne test que MyThirdTest.php et MyFourthTest.php
$ ./bin/atoum -f MyFirstTest.php MySecondTest.php -f MyThirdTest.php MyFourthTest.php
```

14.1 Configuration & bootstrap

14.1.1 -af <file> / –autoloader-file <file>

Cette option vous permet de spécifier le chemin du *fichier d'autoloader*.

```
$ ./bin/atoum -af /path/to/autloader.php
$ ./bin/atoum --autoloader-file /path/to/autoloader.php
```

14.1.2 -bf <file> / –bootstrap-file <file>

Cette option vous permet de spécifier le chemin du *fichier de bootstrap*.

```
$ ./bin/atoum -bf /path/to/bootstrap.php
$ ./bin/atoum --bootstrap-file /path/to/bootstrap.php
```

14.1.3 -c <file> / –configuration <file>

Cette option vous permet de spécifier le chemin vers le *fichier de configuration* à utiliser pour lancer les tests.

```
$ ./bin/atoum -c config/atoum.php
$ ./bin/atoum --configuration tests/units/conf/coverage.php
```

14.1.4 -xc, –xdebug-config

Cette option vous permet de spécifié la variable *XDEBUG_CONFIG*. Ceci peux aussi être configurer avec *$runner->setXdebugConfig()*.

14.2 Filtrage

14.2.1 -d <directories> / –directories <directories>

Cette option vous permet de spécifier le répertoire des test à exécuter. Vous pouvez aussi le *configurer*.

```
$ ./bin/atoum -d tests/units/db/
$ ./bin/atoum --directories tests/units/db/ tests/units/entities/
```

14.2.2 -f <files> / –files <files>

Cette option vous permet de spécifier le ou les fichiers de tests à lancer.

```
$ ./bin/atoum -f tests/units/db/mysql.php
$ ./bin/atoum --files tests/units/db/mysql.php tests/units/db/pgsql.php
```

14.2.3 -g <pattern> / –glob <pattern>

Cette option vous permet de spécifier les fichiers de tests à lancer en fonction d'un schéma.

```
$ ./bin/atoum -g ???
$ ./bin/atoum --glob ???
```

14.2.4 -m <class : :method> / –methods <class : :methods>

Cette option vous permet de filtrer les classes et les méthodes à lancer.

```
# lance uniquement la méthode testMyMethod de la classe␣
↪vendor\\project\\test\\units\\myClass
$ ./bin/atoum -m vendor\\project\\test\\units\\myClass::testMyMethod
$ ./bin/atoum --methods vendor\\project\\test\\units\\myClass::testMyMethod
```

(suite sur la page suivante)

(suite de la page précédente)

```
# lance toutes les méthodes de test de la classe vendor\\project\\test\\units\\myClass
$ ./bin/atoum -m vendor\\project\\test\\units\\myClass::*
$ ./bin/atoum --methods vendor\\project\\test\\units\\myClass::*

# lance uniquement les méthodes testMyMethod de toutes les classes de test
$ ./bin/atoum -m *::testMyMethod
$ ./bin/atoum --methods *::testMyMethod
```

Note : Reportez-vous à la section sur les filtres par *Une classe ou une méthode* pour avoir plus d'informations.

14.2.5 -ns <namespaces> / –namespaces <namespaces>

Cette option vous permet de filtrer les classes et les méthodes en fonction des espaces de noms.

```
$ ./bin/atoum -ns mageekguy\\atoum\\tests\\units\\asserters
$ ./bin/atoum --namespaces mageekguy\\atoum\\tests\\units\\asserters
```

Note : Reportez-vous à la section sur les filtres *Par espace de noms* pour avoir plus d'informations.

14.2.6 -t <tags> / –tags <tags>

Cette option vous permet de filtrer les classes et les méthodes à lancer en fonction des tags.

```
$ ./bin/atoum -t OneTag
$ ./bin/atoum --tags OneTag TwoTag
```

Note : Reportez-vous à la section sur les filtres par *Tags* pour avoir plus d'informations.

14.2.7 –test-all

Cette option vous permet de lancer les tests se trouvant dans les répertoires définis dans le fichier de configuration via `$script->addTestAllDirectory('path/to/directory')`.

```
$ ./bin/atoum --test-all
```

14.2.8 –test-it

Cette option vous permet de lancer les tests unitaires d'atoum pour vérifier qu'il fonctionne parfaitement sur votre serveur. Vous pouvez aussi le configurer avec `$script->testIt();`.

```
$ ./bin/atoum --test-it
```

14.2.9 -tfe <extensions> / –test-file-extensions <extensions>

Cette option vous permet de spécifier le ou les extensions des fichiers de tests à lancer.

```
$ ./bin/atoum -tfe phpt
$ ./bin/atoum --test-file-extensions phpt php5t
```

14.3 Débugage & boucle

14.3.1 –debug

Cette option vous permet d'activer le mode debug

```
$ ./bin/atoum --debug
```

Note : Reportez-vous à la section sur le *Débogage des scénarios de test* pour avoir plus d'informations.

14.3.2 -l / –loop

Cette option vous permet d'activer le mode loop d'atoum.

```
$ ./bin/atoum -l
$ ./bin/atoum --loop
```

Note : Reportez-vous à la section sur le *Mode répétition* pour avoir plus d'informations.

14.3.3 –disable-loop-mode

Cette option vous permet de désactivé le mode loop. Ceci permet d'écraser un mode loop activé via le fichier de configuration.

14.3.4 +verbose / ++verbose

Cette option active le mode verbose de atoum.

```
$ ./bin/atoum ++verbose
```

14.4 Couverture & rapports

14.4.1 -drt <string> / –default-report-title <string>

Cette option permet de spécifier le titre par défaut du rapport d'atoum.

```
$ ./bin/atoum -drt Title
$ ./bin/atoum --default-report-title "My Title"
```

Note : Si le titre comporte des espaces, il faut obligatoirement l'entourer de guillemets.

14.4.2 -ebpc, –enable-branch-and-path-coverage

Cette option active la couverture sur les branches et chemin. Vous pouvez aussi le faire *au travers de la configuration*.

```
$ ./bin/atoum -ebpc
$ ./bin/atoum --enable-branch-and-path-coverage
```

14.4.3 -ft / –force-terminal

Cette option vous permet de forcer la sortie vers le terminal.

```
$ ./bin/atoum -ft
$ ./bin/atoum --force-terminal
```

14.4.4 -sf <file> / –score-file <file>

Cette option vous permet de spécifier le chemin vers le fichier des résultats créé par atoum.

```
$ ./bin/atoum -sf /path/to/atoum.score
$ ./bin/atoum --score-file /path/to/atoum.score
```

14.4.5 -ncc / –no-code-coverage

Cette option vous permet de désactiver la génération du rapport de la couverture de code.

```
$ ./bin/atoum -ncc
$ ./bin/atoum --no-code-coverage
```

14.4.6 -nccfc <classes> / –no-code-coverage-for-classes <classes>

Cette option vous permet de désactiver la génération du rapport de couverture de code pour un ou plusieurs classes.

```
$ ./bin/atoum -nccfc vendor\\project\\db\\mysql
$ ./bin/atoum --no-code-coverage-for-classes vendor\\project\\db\\mysql␣
↪vendor\\project\\db\\pgsql
```

Note : Il est important de doubler chaque backslash pour éviter qu'ils soient interprétés par le shell.

14.4.7 -nccfns <namespaces> / –no-code-coverage-for-namespaces <namespaces>

Cette option vous permet de désactiver la génération du rapport de couverture de code pour un ou plusieurs namespaces.

```
$ ./bin/atoum -nccfns vendor\\outside\\lib
$ ./bin/atoum --no-code-coverage-for-namespaces vendor\\outside\\lib1␣
↪vendor\\outside\\lib2
```

Note : Il est important de doubler chaque backslash pour éviter qu'ils soient interprétés par le shell.

14.4.8 -nccid <directories> / –no-code-coverage-in-directories <directories>

Cette option vous permet de désactiver la génération du rapport de couverture de code pour un ou plusieurs répertoires.

```
$ ./bin/atoum -nccid /path/to/exclude
$ ./bin/atoum --no-code-coverage-in-directories /path/to/exclude/1 /path/to/exclude/2
```

14.4.9 -nccfm <method> / –no-code-coverage-for-methods <method>

Cette option vous permet de désactiver la génération du rapport de couverture de code pour une ou plusieurs méthodes.

```
$ ./bin/atoum -nccfm foo\\test\\units\\myClass::testMyMethod␣
↪foo\\test\\units\\myClassToo::testMyMethod
$ ./bin/atoum --no-code-coverage-for-methods foo\\test\\units\\myClass::testMyMethod␣
↪foo\\test\\units\\myClassToo::testMyMethod
```

14.4.10 -udr / –use-dot-report

Cette option vous permet d'afficher uniquement la progression de l'exécution avec des points.

```
$ bin/atoum -udr
$ bin/atoum --use-dot-report
.............................................................. [60/65]
.....                                                          [65/65]
Success (1 test, 65/65 methods, 0 void method, 0 skipped method, 872 assertions)!
```

14.4.11 -ulr / –use-light-report

Cette option vous permet d'alléger la sortie généré par atoum.

```
$ ./bin/atoum -ulr
$ ./bin/atoum --use-light-report

[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][  59/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 118/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 177/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 236/1141]
```

(suite sur la page suivante)

(suite de la page précédente)

```
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 295/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 354/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 413/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 472/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 531/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 590/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 649/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 708/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 767/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 826/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 885/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 944/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][1003/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][1062/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][1121/1141]
[SSSSSSSSSSSSSSSSSS_____________________________________________][1141/1141]
Success (154 tests, 1141/1141 methods, 0 void method, 0 skipped method, 16875␣
↪assertions) !
```

14.4.12 -utr / –use-tap-report

Cette option créer un rapport de type tap

```
$ ./bin/atoum -utr
$ ./bin/atoum --use-tap-report
```

14.5 Échec & succès

14.5.1 -fivm / –fail-if-void-methods

Cette option va faire échouer la suite de tests s'il y a au moins une méthode vide.

```
$ ./bin/atoum -fivm
$ ./bin/atoum --fail-if-void-methods
```

14.5.2 -fism / –fail-if-skipped-methods

Cette option va faire échouer la suite de tests s'il y a au moins une méthode ignorée

```
$ ./bin/atoum -fism
$ ./bin/atoum --fail-if-skipped-methods
```

14.6 Autres arguments

14.6.1 -mcn <integer> / –max-children-number <integer>

Cette option vous permet de définir le nombre maximum de processus lancés pour exécuter les tests.

```
$ ./bin/atoum -mcn 5
$ ./bin/atoum --max-children-number 3
```

14.6.2 -p <file> / –php <file>

Cette option vous permet de spécifier le chemin de l'exécutable php à utiliser pour lancer vos tests.

```
$ ./bin/atoum -p /usr/bin/php5
$ ./bin/atoum --php /usr/bin/php5
```

Par défaut, la valeur est recherchée parmi les valeurs suivantes (dans l'ordre) :
- constante PHP_BINARY
- variable d'environnement PHP_PEAR_PHP_BIN
- variable d'environnement PHPBIN
- constante PHP_BINDIR + "/php"

14.6.3 -h / –help

Cette option vous permet d'afficher la liste des options disponibles.

```
$ ./bin/atoum -h
$ ./bin/atoum --help
```

14.6.4 –init <directory>

Cette commande initialise quelques fichiers de configuration.

```
$ ./bin/atoum --init path/to/configuration/directory
```

14.6.5 -v / –version

Cette option vous permet d'afficher la version courante d'atoum.

```
$ ./bin/atoum -v
$ ./bin/atoum --version

atoum version DEVELOPMENT by Frédéric Hardy (/path/to/atoum)
```

Nouveau dans la version 3.3.0 : Ajout du dot report

CHAPITRE 15

Cookbook

15.1 Changer l'espace de nom par défaut

Au début de l'exécution d'une classe de test, atoum calcule le nom de la classe testée. Pour cela, par défaut, il remplace dans le nom de la classe de test l'expression régulière `#(?:^|\\)tests?\\units?\\#i` par le caractère `\`.

Ainsi, si la classe de test porte le nom `vendor\project\tests\units\foo`, il en déduira que la classe testée porte le nom `vendor\project\foo`. Cependant, il peut être nécessaire que l'espace de nom des classes de test ne corresponde pas à cette expression régulière, et dans ce cas, atoum s'arrête alors avec le message d'erreur suivant :

```
> exception 'mageekguy\atoum\exceptions\runtime' with message 'Test class
↪'project\vendor\my\tests\foo' is not in a namespace which match pattern '#(?:^
↪|\\)ests?\\unit?s\#i'' in /path/to/unit/tests/foo.php
--------------------------------------------------------------------------------
↪--------------------------------------------------------------------------------
↪----------------------------------
```

Il faut donc modifier l'expression régulière utilisée, ceci est possible de plusieurs manières. Le plus simple est de faire appel à l'annotation `@namespace` appliquée à la classe de test, de la manière suivante :

```php
<?php

namespace vendor\project\my\tests;

require_once __DIR__ . '/atoum.phar';

use mageekguy\atoum;

/**
 * @namespace \my\tests
 */
abstract class aClass extends atoum
{
   public function testBar()
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {
        /* ... */
    }
}
```

Cette méthode est simple et rapide à mettre en œuvre, mais elle présente l'inconvénient de devoir être répétée dans chaque classe de test, ce qui peut compliquer leur maintenance en cas de modification de leur espace de nom. L'alternative consiste à faire appel à la méthode `atoum\test::setTestNamespace()` dans le constructeur de la classe, de cette manière :

```
<?php

namespace vendor\project\my\tests;

require_once __DIR__ . '/atoum.phar';

use mageekguy\atoum;

abstract class aClass extends atoum
{
    public function __construct(score $score = null, locale $locale = null, adapter
↪$adapter = null)
    {
        $this->setTestNamespace('\\my\\tests');

        parent::__construct($score, $locale, $adapter);
    }

    public function testBar()
    {
        /* ... */
    }
}
```

La méthode `atoum\test::setTestNamespace()` accepte en effet un unique argument qui doit être l'expression régulière correspondant à l'espace de nom de votre classe de test. Et pour ne pas avoir à répéter l'appel à cette méthode dans chaque classe de test, il suffit de le faire une bonne fois pour toutes dans une classe abstraite de la manière suivante :

```
<?php

namespace vendor\project\my\tests;

require_once __DIR__ . '/atoum.phar';

use mageekguy\atoum;

abstract class Test extends atoum
{
    public function __construct(score $score = null, locale $locale = null, adapter
↪$adapter = null)
    {
         $this->setTestNamespace('\\my\\tests');

        parent::__construct($score, $locale, $adapter);
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
}
```

Ainsi, vous n'aurez plus qu'à faire dériver vos classes de tests unitaires de cette classe abstraite :

```
<?php

namespace vendor\project\my\tests\modules;

require_once __DIR__ . '/atoum.phar';

use mageekguy\atoum;
use vendor\project\my\tests;

class aModule extends tests\Test
{
   public function testDoSomething()
   {
      /* ... */
   }
}
```

En cas de modification de l'espace de nommage réservé aux tests unitaires, il ne sera donc nécessaire de ne modifier que la classe abstraite.

De plus, il n'est pas obligatoire d'utiliser une expression régulière, que ce soit au niveau de l'annotation `@namespace` ou de la méthode `atoum\test::setTestNamespace()`, et une simple chaîne de caractères peut également fonctionner.

En effet, atoum fait appel par défaut à une expression régulière afin que son utilisateur puisse utiliser par défaut un large panel d'espaces de nom sans avoir besoin de le configurer à ce niveau. Cela lui permet donc d'accepter par exemple sans configuration particulière les espaces de nomsuivants :

- `test\unit\`
- `Test\Unit\`
- `tests\units\`
- `Tests\Units\`
- `TEST\UNIT\`

Cependant, en règle général, l'espace de nom utilisé pour les classes de test est fixe et il n'est donc pas nécessaire de recourir à une expression régulière si celle par défaut ne convient pas. Dans notre cas, elle pourrait être remplacée par la chaîne de caractères `my\tests`, par exemple grâce à l'annotation `@namespace` :

```
<?php

namespace vendor\project\my\tests;

require_once __DIR__ . '/atoum.phar';

use mageekguy\atoum;

/**
 * @namespace \my\tests\
 */
abstract class aClass extends atoum
{
   public function testBar()
   {
      /* ... */
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

15.2 Test d'un singleton

Pour tester si une méthode retourne bien systématiquement la même instance d'un objet, vérifiez que deux appels successifs à la méthode testée sont bien identiques.

```
<?php
$this
    ->object(\Singleton::getInstance())
        ->isInstanceOf('Singleton')
        ->isIdenticalTo(\Singleton::getInstance())
;
```

15.3 Hook git

Une bonne pratique, lorsqu'on utilise un logiciel de gestion de versions, est de ne jamais ajouter à un dépôt du code non fonctionnel, afin de pouvoir récupérer une version propre et utilisable du code à tout moment et à n'importe quel endroit de l'historique du dépôt.

Cela implique donc, entre autres, que les tests unitaires doivent passer dans leur intégralité avant que les fichiers créés ou modifiés soient ajoutés au dépôt et, en conséquence, le développeur est censé exécuter les tests unitaires avant d'intégrer son code dans le dépôt.

Cependant, dans les faits, il est très facile pour le développeur d'omettre cette étape, et votre dépôt peut donc contenir à plus ou moins brève échéance du code ne respectant pas les contraintes imposées par les tests unitaires.

Heureusement, les logiciels de gestion de versions en général et Git en particulier disposent d'un mécanisme, connu sous le nom de hook de pré-commit permettant d'exécuter automatiquement des tâches lors de l'ajout de code dans un dépôt.

L'installation d'un hook de pré-commit est très simple et se déroule en deux étapes.

15.3.1 Étape 1 : Création du script à exécuter

Lors de l'ajout de code à un dépôt, Git recherche le fichier `.git/hook/pre-commit` à la racine du dépôt et l'exécute s'il existe et qu'il dispose des droits nécessaires.

Pour mettre en place le hook, il vous faut donc créer le fichier `.git/hook/pre-commit` et y ajouter le code suivant :

Le code ci-dessous suppose que vos tests unitaires sont dans des fichiers ayant l'extension `.php` et dans des répertoires dont le chemin contient `/Tests/Units/`. Si ce n'est pas votre cas, vous devrez modifier le script suivant votre contexte.

Note : Dans l'exemple ci-dessus, les fichiers de test doivent inclure atoum pour que le hook fonctionne.

Les tests étant exécutés très rapidement avec atoum, on peut donc lancer l'ensemble des tests unitaires avant chaque commit avec un hook comme celui-ci :

```
#!/bin/sh
./bin/atoum -d tests/
```

15.3.2 Étape 2 : Ajout des droits d'exécution

Pour être utilisable par Git, le fichier `.git/hook/pre-commit` doit être rendu exécutable à l'aide de la commande suivante, exécutée en ligne de commande à partir du répertoire de votre dépôt :

```
$ chmod u+x `.git/hook/pre-commit`
```

À partir de cet instant, les tests unitaires contenus dans les répertoires dont le chemin contient `/Tests/Units/` seront lancés automatiquement lorsque vous effectuerez la commande `git commit`, si des fichiers ayant l'extension `.php` ont été modifiés.

Et si d'aventure un test ne passe pas, les fichiers ne seront pas ajoutés au dépôt. Il vous faudra alors effectuer les corrections nécessaires, utiliser la commande `git add` sur les fichiers modifiés et utiliser à nouveau `git commit`.

15.4 Utilisation dans behat

Les *asserters* d'atoum sont très facilement utilisables hors de vos tests unitaires classiques. Il vous suffit d'importer la classe *mageekguyatoumasserter* en n'oubliant pas d'assurer le chargement des classes nécessaires (atoum fournit une classe d'autoload disponible dans *classes/autoloader.php*). L'exemple suivant illustre cette utilisation des asserters atoum à l'intérieur de vos *steps* Behat.

15.4.1 Installation

Installez simplement atoum et Behat dans votre projet via pear, git clone, zip... Voici un exemple avec le gestionnaire de dépendances *Composer* :

```
"require-dev": {
        "behat/behat": "2.4@stable",
        "atoum/atoum": "~2.5"
}
```

Il est évidemment nécessaire de remettre à jour vos dépendances composer en lançant la commande :

```
$ php composer.phar update
```

15.4.2 Configuration

Comme mentionné en introduction, il suffit d'importer la classe d'asserter et d'assurer le chargement des classes d'atoum. Pour Behat, la configuration des asserters s'effectue dans votre classe *FeatureContext.php* (située par défaut dans le répertoire */RACINE DE VOTRE PROJET/features/bootstrap/*).

```
<?php

use Behat\Behat\Context\ClosuredContextInterface,
    Behat\Behat\Context\TranslatedContextInterface,
    Behat\Behat\Context\BehatContext,
    Behat\Behat\Exception\PendingException,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    Behat\Behat\Context\Step;
use Behat\Gherkin\Node\PyStringNode,
    Behat\Gherkin\Node\TableNode;

use atoum\asserter; // <- atoum asserter

require_once __DIR__ . '/../../vendor/atoum/atoum/classes/autoloader.php'; // <-
↪autoload

class FeatureContext extends BehatContext
{
    private $assert;

    public function __construct(array $parameters)
    {
        $this->assert = new asserter\generator();
    }
}
```

15.4.3 Utilisation

Après ces 2 étapes particulièrement triviales, vos *steps* peuvent s'enrichir des asserters atoum :

```
<?php

// ...

class FeatureContext extends BehatContext
{//...

    /**
     * @Then /^I should get a good response using my favorite "([^"]*)"$/
     */
    public function goodResponse($contentType)
    {
        $this->assert
            ->integer($response->getStatusCode())
                ->isIdenticalTo(200)
            ->string($response->getHeader('Content-Type'))
                ->isIdenticalTo($contentType);
    }
}
```

Encore une fois, ceci n'est qu'un exemple spécifique à Behat mais il reste valable pour tous les besoins d'utilisation des asserters d'atoum hors contexte initial.

15.5 Utilisation dans des outils d'intégration continue (CI)

15.5.1 Utilisation dans Jenkins (ou Hudson)

Il est très simple d'intégrer les résultats de tests atoum à Jenkins (ou Hudson) en tant que résultats xUnit.

Étape 1 : Ajout d'un rapport xUnit à la configuration atoum

Comme pour les autres rapports de couverture, vous pouvez définir des *rapports spécifiques* dans la configuration.

Si vous n'avez pas de fichier de configuration

Si vous ne disposez pas encore d'un fichier de configuration pour atoum, nous vous recommandons d'extraire le répertoire ressource d'atoum dans celui de votre choix à l'aide de la commande suivante :

— Si vous utilisez l'archive Phar d'atoum :

```
$ php atoum.phar --extractRessourcesTo /tmp/atoum-src
$ cp /tmp/atoum-src/resources/configurations/runner/xunit.php.dist /mon/projet/.atoum.
↪php
```

— Si vous utilisez les sources d'atoum :

```
$ cp /chemin/vers/atoum/resources/configurations/runner/xunit.php.dist /mon/projet/.
↪atoum.php
```

— Vous pouvez également copier le fichier directement depuis le dépôt Github

Il ne vous reste plus qu'à éditer ce fichier pour choisir l'emplacement où atoum génèrera le rapport xUnit. Ce fichier est prêt à l'emploi, avec lui, vous conserverez le rapport par défaut d'atoum et vous obtiendrez un rapport xUnit à la suite de chaque lancement des tests.

Si vous avez déjà un fichier de configuration

Si vous disposez déjà d'un fichier de configuration, il vous suffit d'y ajouter les lignes suivantes :

```
<?php

//...

/*
 * Xunit report
 */
$xunit = new atoum\reports\asynchronous\xunit();
$runner->addReport($xunit);

/*
 * Xunit writer
 */
$writer = new atoum\writers\file('/chemin/vers/le/rapport/atoum.xunit.xml');
$xunit->addWriter($writer);
```

Étape 2 : Tester la configuration

Pour tester cette configuration, il suffit de lancer atoum en lui précisant le fichier de configuration que vous souhaitez utiliser :

```
$ ./bin/atoum -d /chemin/vers/les/tests/units -c /chemin/vers/la/configuration.php
```

Note : Si vous avez nommé votre fichier de configuration `.atoum.php`, atoum le chargera automatiquement. Le paramètre `-c` est donc optionnel dans ce cas. Pour qu'atoum charge automatiquement ce fichier, vous devrez lancer les tests à partir du dossier où se trouve le fichier `.atoum.php` ou d'un de ses enfants.

À la fin de l'exécution des tests, vous devriez voir le rapport xUnit dans le répertoire indiqué dans le fichier de configuration.

Étape 3 : Lancement des tests via Jenkins (ou Hudson)

Il existe pour cela plusieurs possibilités selon la façon dont vous construisez votre projet :
- Si vous utilisez un script, il vous suffit d'y ajouter la commande précédente.
- Si vous passez par un utilitaire tel que phing ou ant, il suffit d'ajouter une tâche exec :

```
<target name="unitTests">
  <exec executable="/usr/bin/php" failonerror="yes" failifexecutionfails="yes">
    <arg line="/path/to/atoum.phar -p /path/to/php -d /path/to/test/folder -c /path/
↪to/atoumConfig.php" />
  </exec>
</target>
```

Vous noterez l'ajout du paramètre `-p /chemin/vers/php` qui permet d'indiquer à atoum le chemin vers le binaire PHP qu'il doit utiliser pour exécuter les tests unitaires.

Étape 4 : Publier le rapport avec Jenkins (ou Hudson)

Il suffit tout simplement d'activer la publication des rapports au format JUnit ou xUnit, en fonction du plug-in que vous utilisez, en lui indiquant le chemin d'accès au fichier généré par atoum.

15.5.2 Utilisation avec Travis-ci

Il est assez simple d'utiliser atoum dans l'outil qu'est Travis-CI. En effet, l'ensemble des étapes est indiqué dans la documentation de travis : * Créer votre fichier .travis.yml dans votre projet ; * Ajoutez-y les deux lignes suivantes :

```
before_script: wget http://downloads.atoum.org/nightly/atoum.phar
script: php atoum.phar
```

Voici un exemple de fichier *.travis.yml* dont les tests présents dans le dossier *tests* seront exécuter.

```
language: php
php:
  - 5.4
  - 5.5
  - 5.6

before_script: wget http://downloads.atoum.org/nightly/atoum.phar
script: php atoum.phar -d tests/
```

15.6 Utilisation avec Phing

La suite de tests de atoum peut facilement être exécutée au sein de votre configuration phing via l'intégration de la tâche *phing/AtoumTask.php*. Un exemple valide peut être trouvé dans le fichier resources/phing/build.xml.

Vous devez néanmoins enregistrer votre tâche personnalisée en utilisant taskdef , une tâche native de phing :

```
<taskdef name="atoum" classpath="vendor/atoum/atoum/resources/phing" classname=
↪"AtoumTask"/>
```

Ensuite vous pouvez l'utiliser à l'intérieur de l'une de vos étapes du fichier de build :

```
<target name="test">
  <atoum
    atoumautoloaderpath="vendor/atoum/atoum/classes/autoloader.php"
    phppath="/usr/bin/php"
    codecoverage="true"
    codecoveragereportpath="reports/html/"
    showcodecoverage="true"
    showmissingcodecoverage="true"
    maxchildren="5"
  >
    <fileset dir="tests/units/">
      <include name="**/*.php"/>
    </fileset>
  </atoum>
</target>
```

Les chemins donnés dans cet exemple a été pris à partir d'une installation standard via composer. Tous les paramètres possibles sont définis ci-dessous, vous pouvez modifier les valeurs ou en omettre certains et hériter des valeurs par défaut. Il y a trois types de paramètres :

15.6.1 Configuration d'atoum

- *bootstrap* : fichier de bootstrap à inclure, exécuté avant chaque méthode de test
  - default : `.bootstrap.atoum.php`
- *atoumpharpath* : si atoum est utilisé au travers d'un phar, chemin vers celui-ci
- *atoumautoloaderpath* : fichier d'autoloader, le fichier est exécuté avant chaque méthode de test
  - default : `.autoloader.atoum.php`
- *phppath* : chemin vers l'exécutable `php`
- *maxchildren* : nombre maximum de sous-process qui peuvent tourner simultanément

15.6.2 Flags

- *codecoverage* : active la couverture de code(uniquement si XDebug est disponible)
  - default : `false`
- *showcodecoverage* : montre le rapport de couverture de code
  - default : `true`
- *showduration* : montre la durée de l'exécution des tests
  - default : `true`
- *showmemory* : affiche la consommation mémoire
  - default : `true`
- *showmissingcodecoverage* : montre la couverture de code manquante
  - default : `true`

- *showprogress* : affiche la barre de progression de l'exécution des tests
  - default : `true`
- *branchandpathcoverage* : active la couverture de code sur les chemins et branches
  - default : `false`
- telemetry : active le rapport telemetry (l'extension *atoum/reports-extension* doit être installée)
  - default : `false`

15.6.3 Rapports

- *codecoveragexunitpath* : chemin vers le rapport xunit
- *codecoveragecloverpath* : chemin vers le rapport clover
- *Couverture de code basic*
  - *codecoveragereportpath* : chemin vers le rapport html
  - *codecoveragereporturl* : url dans le rapport HTML
- *Couverture de code treemap* :
  - *codecoveragetreemappath* : chemin vers le rapport treemap
  - *codecoveragetreemapurl* : url pour le treemap
- Couverture de code avancée
  - *codecoveragereportextensionpath* : chemin vers le rapport html
  - *codecodecoveragereportextensionurl* : url du rapport HTML
- Telemetry
  - *telemetryprojectname* : nom du projet a envoyer à telemetry

15.7 Utilisation avec des frameworks

15.7.1 Utilisation avec ez Publish

Étape 1 : Installation d'atoum au sein d'eZ Publish

Le framework eZ Publish possède déjà un répertoire dédié aux tests, nommé logiquement tests. C'est donc dans ce répertoire que devra être placé l'"*archive PHAR* d'atoum. Les fichiers de tests unitaires utilisant atoum seront quant à eux placés dans un sous-répertoire *tests/atoum* afin qu'ils ne soient pas en conflit avec l'existant.

Étape 2 : Création de la classe de test de base

Une classe de test basée sur atoum doit étendre la classe `\mageekguy\atoum\test`. Toutefois, celle-ci ne tient pas compte des spécifications de *eZ Publish*. . Il est donc nécessaire de définir une classe de test de base, dérivée de `\mageekguy\atoum\test`, qui prendra en compte ces spécifités et donc dérivera l'ensemble des classes de tests unitaires. Pour cela, il suffit de définir la classe suivante dans le fichier `tests\atoum\test.php` :

```
<?php

namespace ezp;

use mageekguy\atoum;

require_once __DIR__ . '/atoum.phar';

// Autoloading : eZ
require 'autoload.php';
```

(suite sur la page suivante)

(suite de la page précédente)

```
if ( !ini_get( "date.timezone" ) )
{
        date_default_timezone_set( "UTC" );
}

require_once( 'kernel/common/i18n.php' );

\eZContentLanguage::setCronjobMode();

/**
 * @abstract
 */
abstract class test extends atoum\test
{
}

?>
```

Étape 3 : Création d'une classe de test

Par défaut, atoum demande à ce que les classes de tests unitaires soient dans un espace de noms contenant *test(s)unit(s)*, afin de pouvoir déduire le nom de la classe testée. À titre d'exemple, l'espace de noms *nomprojet* sera utilisé dans ce qui suit. Pour plus de simplicité, il est de plus conseillé de calquer l'arborescence des classes de test sur celle des classes testées, afin de pouvoir localiser rapidement la classe de test d'une classe, et inversement.

```
<?php

namespace nomdeprojet\tests\units;

require_once '../test.php';

use ezp;

class cache extends ezp\test
{
   public function testClass()
   {
           $this->assert->hasMethod('__construct');
   }
}
```

Étapes 4 : Exécution des tests unitaires

Une fois une classe de test créée, il suffit d'exécuter en ligne de commande l'instruction ci-dessous pour lancer le test, en se plaçant à la racine du projet :

```
# php tests/atoum/atoum.phar -d tests/atoum/units
```

Merci Jérémy Poulain pour ce tutoriel.

15.7.2 Utilisation avec Symfony 2

Si vous souhaitez utiliser atoum au sein de vos projets Symfony, vous pouvez installer le Bundle AtoumBundle.

Si vous souhaitez installer et configurer atoum manuellement, voici comment faire.

Étape 1 : installation d'atoum

Si vous utilisez Symfony 2.0, *téléchargez l'archive PHAR* et placez-la dans le répertoire vendor qui est à la racine de votre projet.

Si vous utilisez Symfony 2.1+, *ajoutez atoum dans votre fichier composer.json*.

Étape 2 : création de la classe de test

Imaginons que nous voulions tester cet Entity :

```
<?php
// src/Acme/DemoBundle/Entity/Car.php
namespace Acme\DemoBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Acme\DemoBundle\Entity\Car
 * @ORM\Table(name="car")
 * @ORM\Entity(repositoryClass="Acme\DemoBundle\Entity\CarRepository")
 */
class Car
{
    /**
     * @var integer $id
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string $name
     * @ORM\Column(name="name", type="string", length=255)
     */
    private $name;

    /**
     * @var integer $max_speed
     * @ORM\Column(name="max_speed", type="integer")
     */

    private $max_speed;
}
```

Note : Pour plus d'informations sur la création d'Entity dans Symfony 2, référez vous au manuel officiel

Créez le répertoire Tests/Units dans votre Bundle (par exemple src/Acme/DemoBundle/Tests/Units). C'est dans ce répertoire que seront stockés tous les tests de ce Bundle.

Créez un fichier Test.php qui servira de base à tous les futurs tests de ce Bundle.

```
<?php
// src/Acme/DemoBundle/Tests/Units/Test.php
namespace Acme\DemoBundle\Tests\Units;

// On inclus et active le class loader
require_once __DIR__ . '/../../../../../vendor/symfony/symfony/src/Symfony/Component/
↪ClassLoader/UniversalClassLoader.php';

$loader = new \Symfony\Component\ClassLoader\UniversalClassLoader();

$loader->registerNamespaces(
    array(
        'Symfony'         => __DIR__ . '/../../../../../vendor/symfony/src',
        'Acme\DemoBundle' => __DIR__ . '/../../../../../src'
    )
);

$loader->register();

use mageekguy\atoum;

// Pour Symfony 2.0 uniquement !
require_once __DIR__ . '/../../../../../vendor/atoum.phar';

abstract class Test extends atoum
{
    public function __construct(
        adapter $adapter = null,
        annotations\extractor $annotationExtractor = null,
        asserter\generator $asserterGenerator = null,
        test\assertion\manager $assertionManager = null,
        \closure $reflectionClassFactory = null
    )
    {
        $this->setTestNamespace('Tests\Units');
        parent::__construct(
            $adapter,
            $annotationExtractor,
            $asserterGenerator,
            $assertionManager,
            $reflectionClassFactory
        );
    }
}
```

Note : L'inclusion de l'archive PHAR d'atoum n'est nécessaire que pour Symfony 2.0. Supprimez cette ligne dans le cas où vous utilisez Symfony 2.1+.

Note : Par défaut, atoum utilise le namespace tests/units pour les tests. Or Symfony 2 et son class loader exige des majuscules au début des noms. Pour cette raison, nous changeons le namespace des tests grâce à la méthode

setTestNamespace("TestsUnits").

Étape 3 : écriture d'un test

Dans le répertoire Tests/Units, il vous suffit de recréer l'arborescence des classes que vous souhaitez tester (par exemple src/Acme/DemoBundle/Tests/Units/Entity/Car.php).

Créons notre fichier de test :

```
<?php
// src/Acme/DemoBundle/Tests/Units/Entity/Car.php
namespace Acme\DemoBundle\Tests\Units\Entity;

require_once __DIR__ . '/../Test.php';

use Acme\DemoBundle\Tests\Units\Test;

class Car extends Test
{
    public function testGetName()
    {
        $this
            ->if($car = new \Acme\DemoBundle\Entity\Car())
            ->and($car->setName('Batmobile'))
                ->string($car->getName())
                    ->isEqualTo('Batmobile')
                    ->isNotEqualTo('De Lorean')
        ;
    }
}
```

Étape 4 : lancement des tests

Si vous utilisez Symfony 2.0 :

```
# Lancement des tests d'un fichier
$ php vendor/atoum.phar -f src/Acme/DemoBundle/Tests/Units/Entity/Car.php

# Lancement de tous les tests du Bundle
$ php vendor/atoum.phar -d src/Acme/DemoBundle/Tests/Units
```

Si vous utilisez Symfony 2.1+ :

```
# Lancement des tests d'un fichier
$ ./bin/atoum -f src/Acme/DemoBundle/Tests/Units/Entity/Car.php

# Lancement de tous les tests du Bundle
$ ./bin/atoum -d src/Acme/DemoBundle/Tests/Units
```

Note : Vous pouvez obtenir plus d'informations sur le *lancement des tests* dans le chapitre qui y est consacré.

Dans tous les cas, voilà ce que vous devriez obtenir :

```
> PHP path: /usr/bin/php
> PHP version:
> PHP 5.3.15 with Suhosin-Patch (cli) (built: Aug 24 2012 17:45:44)
======================================================================
> Copyright (c) 1997-2012 The PHP Group
=====================================
> Zend Engine v2.3.0, Copyright (c) 1998-2012 Zend Technologies
=============================================================
>     with Xdebug v2.1.3, Copyright (c) 2002-2012, by Derick Rethans
=================================================================
> Acme\DemoBundle\Tests\Units\Entity\Car...
[S_____________________________________________________________][1/1]
> Test duration: 0.01 second.
============================
> Memory usage: 0.50 Mb.
========================
> Total test duration: 0.01 second.
> Total test memory usage: 0.50 Mb.
> Code coverage value: 42.86%
> Class Acme\DemoBundle\Entity\Car: 42.86%
========================================
> Acme\DemoBundle\Entity\Car::getId(): 0.00%
------------------------------------------
> Acme\DemoBundle\Entity\Car::setMaxSpeed(): 0.00%
------------------------------------------------
> Acme\DemoBundle\Entity\Car::getMaxSpeed(): 0.00%
------------------------------------------------
> Running duration: 0.24 second.
Success (1 test, 1/1 method, 0 skipped method, 4 assertions) !
```

15.7.3 Utilisation avec symfony 1.4

Si vous souhaitez utiliser atoum au sein de vos projets Symfony 1.4, vous pouvez installer le plugin sfAtoumPlugin. Celui-ci est disponible à l'adresse suivante : https://github.com/atoum/sfAtoumPlugin.

Installation

Il existe plusieurs méthodes d'installation du plugin dans votre projet :

- installation via composer
- installation via des submodules git

En utilisant composer

Ajouter ceci dans le composer.json :

```
"require"     : {
  "atoum/sfAtoumPlugin": "*"
},
```

Après avoir effectué un `php composer.phar update`, le plugin devrait se trouver dans le dossier plugins et atoum dans un dossier `vendor`.

Il faut ensuite activer le plugin dans le ProjectConfiguration et indiquer le chemin d'atoum.

```
<?php
sfConfig::set('sf_atoum_path', dirname(__FILE__) . '/../vendor/atoum/atoum');

if (sfConfig::get('sf_environment') != 'prod')
{
  $this->enablePlugins('sfAtoumPlugin');
}
```

En utilisant des submodules git

Il faut tout d'abord ajouter atoum en tant que submodule :

```
$ git submodule add git://github.com/atoum/atoum.git lib/vendor/atoum
```

Puis ensuite ajouter le sfAtoumPlugin en tant que submodule :

```
$ git submodule add git://github.com/atoum/sfAtoumPlugin.git plugins/sfAtoumPlugin
```

Enfin, il faut activer le plugin dans le fichier ProjectConfiguration :

```
<?php
if (sfConfig::get('sf_environment') != 'prod')
{
  $this->enablePlugins('sfAtoumPlugin');
}
```

Ecrire les tests

Les tests doivent inclure le fichier de bootstrap se trouvant dans le plugin :

```
<?php
require_once __DIR__ . '/../../../../plugins/sfAtoumPlugin/bootstrap/unit.php';
```

Lancer les tests

La commande symfony `atoum:test` est disponible. Les tests peuvent alors se lancer de cette façon :

```
$ ./symfony atoum:test
```

Toutes les paramètres d'atoum sont disponibles.

Il est donc, par exemple, possible de passer un fichier de configuration comme ceci :

```
php symfony atoum:test -c config/atoum/hudson.php
```

15.7.4 Plugin symfony 1

Pour utiliser atoum au sein d'un projet symfony 1, un plug-in existe et est disponible à l'adresse suivante : https://github.com/atoum/sfAtoumPlugin.

Toutes les instructions pour son installation et son utilisation se trouvent dans le cookbook *Utilisation avec symfony 1.4* ainsi que sur la page github.

15.7.5 Bundle Symfony 2

Pour utiliser atoum au sein d'un projet Symfony 2, le bundle AtoumBundle est disponible.

Toutes les instructions pour son installation et son utilisation se trouvent dans le cookbook *Utilisation avec Symfony 2* ainsi que sur la page github.

15.7.6 Composant Zend Framework 2

Si vous souhaitez utiliser atoum au sein d'un projet Zend Framework 2, un composant existe et est disponible à l'adresse suivante.

Toutes les instructions pour son installation et son utilisation sont disponibles sur cette page.

CHAPITRE 16

Intégration d'atoum dans votre IDE

16.1 Sublime Text 2

Un plug-in pour SublimeText 2 permet l'exécution des tests unitaires par atoum et la visualisation du résultat sans quitter l'éditeur.

Les informations nécessaires à son installation et à sa configuration sont disponibles sur le blog de son auteur.

16.2 VIM

atoum est livré avec un plug-in facilitant son utilisation dans l'éditeur VIM.

Il permet d'exécuter les tests sans quitter VIM et d'obtenir le rapport correspondant dans une fenêtre de l'éditeur.

Il est alors possible de naviguer parmi les éventuelles erreurs, voire de se rendre à la ligne correspondant à une assertion ne passant pas à l'aide d'une simple combinaison de touches.

16.2.1 Installation du plug-in atoum pour VIM

Vous trouverez le fichier correspondant au plug-in, nommé `atoum.vmb`, dans le répertoire `resources/vim`.

Si vous utilisez l'archive PHAR, il faut extraire le fichier à l'aide de la commande suivante :

```
$ php atoum.phar --extractResourcesTo path/to/a/directory
```

Une fois l'extraction réalisée, le fichier `atoum.vmb` correspondant au plug-in pour VIM sera dans le répertoire `path/to/a/directory/resources/vim`.

Une fois en possession du fichier `atoum.vmb`, il faut l'éditer à l'aide de VIM :

```
$ vim path/to/atoum.vmb
```

Il n'y a plus ensuite qu'à demander à VIM l'installation du plug-in à l'aide de la commande :

```
:source %
```

16.2.2 Utilisation du plug-in atoum pour VIM

Pour utiliser le plug-in, atoum doit évidemment être installé et vous devez être en train d'éditer un fichier contenant une classe de tests unitaires basée sur atoum.

Une fois dans cette configuration, la commande suivante lancera l'exécution des tests :

```
:Atoum
```

Les tests sont alors exécutés, et une fois qu'ils sont terminés, un rapport basé sur le fichier de configuration d'atoum qui se trouve dans le répertoire `ftplugin/php/atoum.vim` de votre répertoire `.vim` est généré dans une nouvelle fenêtre.

Évidemment, vous êtes libre de lier cette commande à la combinaison de touches de votre choix, en ajoutant par exemple la ligne suivante dans votre fichier `.vimrc` :

```
nnoremap *.php <F12> :Atoum<CR>
```

L'utilisation de la touche `F12` de votre clavier en mode normal appellera alors la commande `:Atoum`.

16.2.3 Gestion des fichiers de configuration d'atoum

Vous pouvez indiquer un autre fichier de configuration pour atoum en ajoutant la ligne suivante à votre fichier `.vimrc` :

```
call atoum#defineConfiguration('/path/to/project/directory', '/path/to/atoum/
↪configuration/file', '.php')
```

La fonction `atoum#defineConfiguration` permet en effet de définir le fichier de configuration à utiliser en fonction du répertoire où se trouve le fichier de tests unitaires. Elle accepte pour cela trois arguments :

- un chemin d'accès vers le répertoire contenant les tests unitaires ;
- un chemin d'accès vers le fichier de configuration d'atoum devant être utilisé ;
- l'extension des fichiers de tests unitaires concernés.

Pour plus de détails sur l'utilisation du plug-in, une aide est disponible dans VIM à l'aide de la commande suivante :

```
:help atoum
```

16.2.4 Rapports de couverture pour vim

Vous pouvez configurer un rapport *spécifique* pour la couverture au sein de vim. Dans votre fichier de configuration atoum, définissez :

... code-block : : php

< ?php use mageekguyatoum ; $vimReport = new atoumreportsasynchronousvim() ; $vimReport->addWriter($stdOutWriter) ; $runner->addReport($vimReport) ;

16.3 PhpStorm

atoum possède avec un plug-in officiel pour PHPStorm. Il vous aide, au quotidien, dans votre développement. Les principales fonctionnalités sont :

- Accès à la classe de test depuis la classe testée (raccourci : alt+shift+K)
- Accès à la classe testée depuis la de test (raccourcis : alt+shift+K)
- Execute tests inside PhpStorm (shortcut : alt+shift+M)
- Identification facile des fichiers de test via une icône spécifique

16.3.1 Installation

C'est simple à installer, pour cela il suffit de suivre les étapes suivantes :

- Ouvrir PHPStorm
- Aller dans *Fichier -> Paramètres*, cliquer sur *Plugins*
- Cliquer sur parcourir le répertoire
- Chercher *atoum* dans la liste, cliquer sur le bouton installation
- Redémarrer PHPStorm

Si vous avez besoin de plus d'information, il suffit de lire le repository du plugin.

16.4 Atom

atoum possède un plug-in officiel pour atom. Celui-ci vous aide dans plusieurs tâches :

- Un panneau avec tous les tests
- Exécuter tous les tests, dans un répertoire ou dans le répertoire courant

16.4.1 Installation

Il est simple d'installation, il suffit de suivre les étapes d'installation officiel ou les étapes suivantes :

- Ouvrir atom
- Aller dans *Paramètres*, cliquer sur *Installation*
- Chercher *atoum* dans la liste, cliquer sur le bouton installation

Si vous avez besoin de plus d'information, il suffit de lire le repository du package.

16.5 Ouvrir automatiquement les tests en échec

atoum est capable d'ouvrir automatiquement les fichiers des tests en échec à la fin de l'exécution. Plusieurs éditeurs sont actuellement supportés :

- *macvim* (Mac OS X)
- *gvim* (Unix)
- *PhpStorm* (Mac OS X/Unix)
- *gedit* (Unix)

Pour utiliser cette fonctionnalité, vous devrez modifier le *fichier de configuration* d'atoum :

16.5.1 macvim

```
<?php
use
    mageekguy\atoum,
    mageekguy\atoum\report\fields\runner\failures\execute\macos
;

$stdOutWriter = new atoum\writers\std\out();
$cliReport = new atoum\reports\realtime\cli();
$cliReport->addWriter($stdOutWriter);

$cliReport->addField(new macos\macvim());

$runner->addReport($cliReport);
```

16.5.2 gvim

```
<?php
use
    mageekguy\atoum,
    mageekguy\atoum\report\fields\runner\failures\execute\unix
;

$stdOutWriter = new atoum\writers\std\out();
$cliReport = new atoum\reports\realtime\cli();
$cliReport->addWriter($stdOutWriter);

$cliReport->addField(new unix\gvim());

$runner->addReport($cliReport);
```

16.5.3 PhpStorm

Si vous travaillez sous Mac OS X, utilisez la configuration suivante :

```
<?php
use
    mageekguy\atoum,
    mageekguy\atoum\report\fields\runner\failures\execute\macos
;

$stdOutWriter = new atoum\writers\std\out();
$cliReport = new atoum\reports\realtime\cli();
$cliReport->addWriter($stdOutWriter);

$cliReport
    // Si PhpStorm est installé dans /Applications
    ->addField(new macos\phpstorm())

    // Si vous avez installé PhpStorm
    // dans un dossier différent de /Applications
    // ->addField(
    //     new macos\phpstorm(
    //         '/path/to/PhpStorm.app/Contents/MacOS/webide'
    //     )
```

(suite sur la page suivante)

(suite de la page précédente)

```
    // )
;

$runner->addReport($cliReport);
```

Dans un environnement Unix, utilisez la configuration suivante :

```
<?php
use
    mageekguy\atoum,
    mageekguy\atoum\report\fields\runner\failures\execute\unix
;

$stdOutWriter = new atoum\writers\std\out();
$cliReport = new atoum\reports\realtime\cli();
$cliReport->addWriter($stdOutWriter);

$cliReport
    ->addField(
        new unix\phpstorm('/chemin/vers/PhpStorm/bin/phpstorm.sh')
    )
;

$runner->addReport($cliReport);
```

16.5.4 gedit

```
<?php
use
    mageekguy\atoum,
    mageekguy\atoum\report\fields\runner\failures\execute\unix
;

$stdOutWriter = new atoum\writers\std\out();
$cliReport = new atoum\reports\realtime\cli();
$cliReport->addWriter($stdOutWriter);

$cliReport->addField(new unix\gedit());

$runner->addReport($cliReport);
```

CHAPITRE 17

Questions Fréquentes

17.1 Si vous avec une erreur inconnue, vérifier si vous utiliser un error_log ?

Si vous utilisez *error_log*, vous rencontrerez une erreur « Error UNKNOWN in » de atoum. Pour éviter cela, utiliser *un mock d'une fonction native* de *error_log*

```
<?php
namespace Foo
{
        class TestErrorLog
        {
            public function runErrorLog()
            {
                error_log('message');
                return true;
            }
        }
}

namespace Foo\test\unit
{
        class TestErrorLog extends \atoum
        {
            public function testRunErrorLog()
            {
                        $this->function->error_log = true;
                        $this->newTestedInstance;
                        $this->boolean($this->testedInstance->runErrorLog())->isTrue;
                        $this->function('error_log')->wasCalled()->once();
            }
        }
}
```

17.2 atoum s'est-il toujours appelé atoum ?

Non, au début, atoum était nommé ogo. Lorsque vous écrivez PHP sur un clavier azerty, puis que vous décalé d'une touche vers la gauche, vous écrivez ogo.

17.3 Quelle est la licence de atoum ?

atoum est distribué sous la licence BSD-3-Clause. Regarder le fichier de LICENSE embarqué pour plus de détails.

17.4 Que est la feuille de route ?

Le plus simple est de regarder les tags de milestone sur github.

CHAPITRE 18

Participer

18.1 Comment participer

Important : We need help to write this section !

18.2 Convention de codage

Le code source d'atoum respecte certaines conventions. Si vous souhaitez contribuer au projet, votre code devra respecter ces mêmes règles :

— L'indentation est faite avec le caractère de tabulation,
— Les noms des espaces de noms, classes, membres, méthodes et constantes sont en `lowerCamelCase`,
— Le code doit être testé.

L'exemple ci-dessous n'a aucun sens, mais il permet de présenter plus en détail la manière dont le code est écrit :

```
<?php

namespace mageekguy\atoum\coding;

use
    mageekguy\atoum,
    type\hinting
;

class standards
{
    const standardsConst = 'standardsConst';
    const secondStandardsConst = 'secondStandardsConst';

    public $public;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    protected $protected;
    private $private = array();

    public function publicFunction($parameter, hinting\claass $optional = null)
    {
        $this->public = trim((string) $parameter);
        $this->protected = $optional ?: new hinting\claass();

        if (($variable = $this->protectedFunction()) === null)
        {
            throw new atoum\exception();
        }

        $flag = 0;
        switch ($variable)
        {
            case self::standardsConst:
                $flag = 1;
                break;

            case self::standardsConst:
                $flag = 2;
                break;

            default:
                return null;
        }

        if ($flag < 2)
        {
            return false;
        }
        else
        {
            return true;
        }
    }

    protected function protectedFunction()
    {
        try
        {
            return $this->protected->get();
        }
        catch (atoum\exception $exception)
        {
            throw new atoum\exception\runtime();
        }
    }

    private function privateFunction()
    {
        $array = $this->private;

        return function(array $param) use ($array) {
            return array_merge($param, $array);
        };
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
}
```

Voici également un exemple de test unitaire :

```
<?php

namespace tests\units\mageekguy\atoum\coding;

use
    mageekguy\atoum,
    mageekguy\atoum\coding\standards as testedClass
;

class standards extends atoum\test
{
    public function testPublicFunction()
    {
        $this
            ->if($object = new testedClass())
            ->then
                ->boolean($object->publicFunction(testedClass::standardsConst))->
↪isFalse()
                ->boolean($object->publicFunction(testedClass::secondStandardsConst))-
↪>isTrue()
            ->if($mock = new \mock\type\hinting\claass())
            ->and($this->calling($mock)->get = null)
            ->and($object = new testedClass())
            ->then
                ->exception(function() use ($object) {
                            $object->publicFunction(uniqid());
                        }
                    )
                        ->IsInstanceOf('\\mageekguy\\atoum\\exception')
        ;
    }
}
```

CHAPITRE 19

Licences

Premièrement, il y a la licence de cette documentation. Celle-ci est CC by-nc-sa 4.0.

Ensuite, vous avez la licence du projet lui-même. Celle-ci est‘BSD-3-Clause <https://github.com/atoum/atoum/blob/master/COPYING>‘_.