atoum Documentation

Release 2.9.0

atoum Team

Apr 19, 2017

Contents

CHAPTER 1

Start with atoum

You first need to *install it*, and then try to start your *first test*. But to understand how to do it, the best is to keep in mind the philosophy of atoum.

The philosophy of atoum

You need to write a test class for each tested class. When you want to test a value, you must:

- indicate the type of this value (integer, decimal, array, String, etc.);
- indicate what you are expecting the value to be (equal to, null, containing a substring, ...).

CHAPTER 2

Installation

If you want to use atoum, simply download the latest version.

You can install atom in several ways:

- using composer;
- download the *PHAR archive* ;
- clone the *Github* repository;
- see also the *integration with your frameworks*.

Composer

Composer is a dependency management tool in PHP.

Be sure you have a working composer installation (official documentation)

Add `atoum/atoum` as a dev dependency :

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

PHAR archive

A PHAR (PHp ARchive) is created automatically on each modification of atoum.

PHAR is an archive format for PHP application.

Installation

You can download the latest stable version of atoum directly from the official website: http://downloads.atoum.org/nightly/atoum.phar

Update

To update the PHAR archive, just run the following command:

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

Note: The update process modifies the PHAR archive. But the default PHP configuration doesn't allow this. So it is mandatory to use the directive `-d phar.readonly=0`.

If a newer version is available it will be downloaded automatically and installed in the 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 !
```

If there is no newer version, atoum will stop immediately:

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

atoum doesn't require any confirmation from the user to be upgraded, because it's very easy to get back to a previous version.

List the versions contained in the archive

You can list versions in the archive by using the argument `--list-available-versions`, or `-lav`:

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

The list of versions in the archive is displayed. The currently active version is preceded by `*`.

Change the current version

To activate another version, just use the argument `--enable-version`, or `-ev`, followed by the name of the version to use:

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

Note: Modification of the current version requires the modification of the PHAR archive. The default PHP configuration doesn't allow this. So it is mandatory to use the directive `-d phar.readonly=0`.

Deleting older versions

Over time, the archive may contain multiple versions of atoum which are no longer required.

To remove them, just use the argument `--delete-version`, or `-dv` followed by the name of the version to deleted:

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

The version is then removed.

Warning: It's not possible to delete the current version.

Note: Deleting a version requires the modification of the PHAR archive. the default PHP configuration doesn't allow this. So it is mandatory to use the directive `-d phar.readonly=0`.

Github

If you want to use atoum directly from source code, you can clone or « fork » the github repository: git://github.com/atoum/atoum.git

CHAPTER 3

First Tests

You need to write a test class for each tested class.

Imagine that you want to test the traditional class `HelloWorld`, then you must create the test class `test\units\HelloWorld`.

Warning: If you are starting with atoum it is recommended to install the package ´atoum-stubs <https://packagist.org/packages/atoum/stubs>´_. This will bring autocompletion to your IDE.

Note: atoum uses namespaces. For example, to test the `Vendor\Project\HelloWorld` class, you must create the class `Vendor\Project\tests\units\HelloWorld` or `tests\units\Vendor\Project\HelloWorld`.

Here is the code of the `HelloWorld` class that we will test.

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

namespace Vendor\Project;

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

Now, here is the code of the test class that we could write.

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

```php
// The test class has is own namespace :
// The namespace of the tested class + "test\units"
namespace Vendor\Project\tests\units;

// You must include the tested class (if you don't have an autoloader)
require_once __DIR__ . '/../../HelloWorld.php';

use atoum;

/*
 * Test class for Vendor\Project\HelloWorld
 *
 * Note that they had the same name that the tested class
 * and that it derives frim the atoum class
 */
class HelloWorld extends atoum
{
    /*
     * This method is dedicated to the getHiAtoum() method
     */
    public function testGetHiAtoum ()
    {
        $this
            // creation of a new instance of the tested class
            ->given($this->newTestedInstance)

            ->then

                    // we test that the getHiAtoum method returns
                    // a string...
                    ->string($this->testedInstance->getHiAtoum())
                        // ... and that this string is the one we want,
                        // namely 'Hi atoum !'
                        ->isEqualTo('Hi atoum !')
        ;
    }
}
```

Now, launch our tests. You should see something like this:

```
$ ./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)!
```

We just test that the method `getHiAtoum`:

- returns a *string*;
- that *is equals to* `"Hi atoum !"`.

The tests passed, everything is green. Your code is solid as a rock with atoum!

Dissecting the test

It's important to understand each part of the test. Let's look at each section.

First, we use the namespace `Vendor\Project\tests\units` where `Vendor\Project` is the namespace of the class and `tests\units` the part of the namespace use by atoum to understand that we are in the test namespace. This special namespace is configurable as explained in the *appropriate section*. Then, inside the test method, we use a special syntax *given and then*. They do nothing other than making the test more readable. Finally we use a couple more simple tricks, *newTestedInstance and testedInstance* to get a new instance of the tested class.

CHAPTER 4

Running tests

Executable

atoum has an executable that allows you to run your tests from the command line.

With phar archive

If you used the phar archive it is already executable.

linux / mac

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

windows

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

With sources

If you use sources, the executable should be found in path/to/atoum/bin.

linux / mac

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

# OR #
```

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

windows

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

Examples in the rest of the documentation

In the following examples, the commands to launch tests with atoum will be written with this syntax:

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

This is exactly the command that you might use if you had *Composer* under Linux.

Files to run

For specific files

To run a specific file test, simply use the -f option or –files.

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

For a folder

To run a test in a folder, simply use the -d option or –directories.

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

You can find more useful arguments to pass to the :ref‘command line<cli-options>‘ in the relevant sections.

Filters

Once you have told to atoum *which files it must execute*, you will be able to filter what will really be executed.

By namespace

To filter on the namespace, i.e. execute only test on given namespace, you have to use the option `-ns` or `--namespaces`.

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

Note: It’s important to use double backslashes to prevent them from being interpreted by the shell.

A class or a method

To filter on a class or a method, i.e. only run tests of a class or a method, just use the option `-m` or `--methods`.

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

Note: It's important to use double backslashes to prevent them from being interpreted by the shell.

You can replace the name of the class or the method with `*` to mean `all`.

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

Using "*" instead of class name mean you can filter by method name.

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

Tags

Like many tools including Behat, atoum allows you to tag your unit tests and run only this with one or more specific tags.

To do this, we must start by defining one or more tags to one or several classes of unit tests.

This is easily done through annotations and the @tags tag:

```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()
    {
        // ...
    }
}
```

In the same way, it is also possible to tag test methods.

Note: The tags defined in a method level take precedence over those defined at the class level.

```php
<?php

namespace vendor\project\tests\units;

require_once __DIR__ . '/atoum.phar';
```

```php
use mageekguy\atoum;

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

Once the required tags are defined, just run the tests with the appropriate tags by using the option `--tags`, or `-t` in its short version:

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

Be careful, this statement only makes sense if there is one or more classes of unit testing and at least one of them has the specified tag. If not, no test will be executed.

It's possible to define several tags:

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

In the latter case, the tests that have been tagged with thisIsOneTag, either thisIsThreeTag, classes will be the only to be executed.

CHAPTER 5

How to write test cases

After you have created your *first test* and understood *how to run it<lancement-des-tests>*, you will want to write better tests. In this section you will see how to sprinkle on some syntactic sugar to help you write better test more easily.

There are several ways to write unit test with atoum, one of them is to use keywords like `given`, `if`, `and` and even `then`, `when` or `assert` so you can structure your tests to make them more readable.

`given`, `if`, `and` and `then`

You can use these keywords very intuitively:

```
<?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)
;
```

It's important to note that these keywords don't have any another purpose than presenting the test in a more readable format. They don't serve any technical purpose. The only goal is to help the reader, human or more specifically developer, to quickly understand what's happening in the test.

Thus, `given`, `if` and `and` specify the prerequisite assertions that follow the keyword `then` to pass.

However, there are no rules or grammar that dictate the order ot syntax of these keywords in atoum.

As a result, the developer should use the keywords wisely in order to make the test as readable as possible. If used incorrectly you could end up with tests like the following :

```
<?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)
;
```

For the same reason, the use of `then` is also optional.

Notice that you can write the exact same test without using any of the previous keywords:

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

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

The test will not be slower or faster to run and there is no advantage to use one notation or another, the important thing is to choose one format and stick to it. This facilitates maintenance of the tests (the problem is exactly the same as coding conventions).

when

In addition to `given`, `if`, `and` and `then`, there are also other keywords.

One of them is `when`. It has a specific feature introduced to work around the fact that it is illegal to write the following PHP code :

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

In this case the language will generate a fatal error: `Parse error: syntax error, unexpected 'unset' (T_UNSET), expecting ')'`

It is impossible to use `unset()` as an argument of a function.

To resolve this problem, the keyword `when` is able to interpret the possible anonymous function that is passed as an argument, allowing us to write the previous test in the following way:

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

Of course, if `when` doesn't receive an anonymous function as an argument, it behaves exactly as `given`, `if`, `and` and `then`, namely that it does absolutely nothing functionally speaking.

assert

Finally, there is the keyword `assert` which also has a somewhat unusual operation.

To illustrate its operation, the following test will be used :

```
<?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)
;
```

The previous test has a disadvantage in terms of maintenance, because if the developer needs to add one or more new calls to bar::doOtherThing() between the two calls already made, it will have to update the value of the argument passed to exactly(). To resolve this problem, you can reset a mock in 2 different ways :

- either by using $mock->getMockController()->resetCalls() ;
- or by using $this->resetMock($mock).

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

```
    // first way
    ->given($foo->getMockController()->resetCalls())
    ->if($bar->setValue(uniqid()))
    ->then
        ->mock($foo)
            ->call('doOtherThing')
                ->once()

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

These methods erase the memory of the controller, so it's now possible to write the next assertion like if the mock was never used.

The keyword `assert` avoids the need for explicit all to `resetCalls()` or `resetMock` and also it will erase the memory of adapters and mock's controllers defined at the time of use.

Thanks to it, it's possible to write the previous test in a simpler and more readable way, especially as it is possible to pass a string to assert that explain the role of the following assertions :

```
<?php
$this
    ->assert("Bar has no value")
        ->given($foo = new \mock\foo())
        ->and($bar = new bar($foo))
        ->if($bar->doSomething())
        ->then
            ->mock($foo)
                ->call('doOtherThing')
                    ->once()

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

Moreover the given string will be included in the messages generated by atoum if one of the assertions is not successful.

newTestedInstance & testedInstance

When performing tests, we must often create a new instance of the class and pass it through parameters. Writing helper are available for this specific case, it's `newTestedInstance` and `testedInstance`

Here's an example :

```
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)
        ;
    }
}
```

This can be simplified with a new syntax:

```
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()
        ;
    }
}
```

As seen, it's slightly simpler but especially this has two advantages:

- We do not manipulate the name of the tested class
- We do not manipulate the tested instance

Furthermore, we can easily validate that the instance is available with *isTestedInstance*, as explained in the previous example.

To pass some arguments to the constructor, it's easy through `newTestedInstance`:

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

If you want to test a static method of your class, you can retrieve the tested class with this syntax:

```
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())
       ;
    }
}
```

testedClass

Like `testedInstance`, you can use `testedClass` to write more comprehensible test. `testedClass` allows you to dynamically assert on the class being tested:

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

You can go further with the *class asseters*.

CHAPTER 6

Asserters collection

To write more explicit and less wordy tests, atoum provide several asserters who give access to specific assertions related to tested var.

As atoum different asserters are specializations of manipulated items, asserters inherits from asserters they specialize. It help keep consistency between asserters and force to use same assertion names.

This is the asserters inheritance tree:

```
-- 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
            `-- utf8String
```

Note: The general asserter/assertion syntaxe is: `$this->[asserter]($value)->[assertion];`

Note: Most of the assertions are fluent, as you will see below.

Note: At the end of this chapter you will find several *tips & tricks* related to assertion and asserter, don't forget to read it!

afterDestructionOf

It's the dedicated assertion to object destruction.

This assertion check that the given object is valid and check if `__destruct()` method is defined and then invokes it.

If `__destruct()` exists and is executed without any error or exception then the test succeeds.

```
<?php
$this
    ->afterDestructionOf($objectWithDestructor)      // succeed
    ->afterDestructionOf($objectWithoutDestructor)   // fails
;
```

array

It's the assertion dedicated to arrays.

Note: `array` is a reserved word in PHP, it hasn't been possible to create an `array` assertion. It's therefore called `phpArray` and an alias `array` was created. So, you can meet either``->phpArray()`` or `->array()`.

It's recommended to use only `->array()` in order to simplify the reading of tests.

Syntactic sugar

In order to simplify the writing of tests with arrays, some syntactic sugar is available. It allows to make various assertions directly on the keys of the tested array.

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

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

Note: This writing form is available from PHP 5.4.

child

With `child` you can assert on a subarray.

```
<?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: This is available from PHP 5.4.

contains

`contains` check that array contains some data.

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

$this
    ->array($fibonacci)
        ->contains('1')      // succeed
        ->contains(1)        // succeed, but data type...
        ->contains('2')      // ... is not checked
        ->contains(10)       // failed
;
```

Note: `contains` doesn't check recursively.

Warning:

`contains` doesn't check the data type.
If you want also to check its type, use *strictlyContains*.

containsValues

`containsValues` checks that an array contains all data from a given array.

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

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

Note: `containsValues` doesn't search recursively.

Warning:

`containsValues` doesn't test data type.
If you also want to check their types, use *strictlyContainsValues*.

hasKey

`hasKey` check that the table contains a given key.

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

$this
    ->array($fibonacci)
        ->hasKey(0)          // passes
        ->hasKey(1)          // passes
        ->hasKey('1')        // passes
        ->hasKey(10)         // failed

    ->array($atoum)
        ->hasKey('name')     // passes
        ->hasKey('price')    // fails
;
```

Note: `hasKey` doesn't search recursively.

Warning: `hasKey` doesn't test the key type.

hasKeys

`hasKeys` checks that an array contains all given keys.

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

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

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

Note: `hasKeys` doesn't search recursively.

Warning: `hasKeys` doesn't test the keys type.

hasSize

`hasSize` checks the size of an array.

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

$this
    ->array($fibonacci)
        ->hasSize(7)        // passes
        ->hasSize(10)       // fails
;
```

Note: `hasSize` is not recursive.

isEmpty

`isEmpty` checks that an array is empty.

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

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

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

isEqualTo

Hint: `isEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isEqualTo*

isIdenticalTo

Hint: `isIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isIdenticalTo*

isNotEmpty

`isNotEmpty` checks that an array is not empty.

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

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

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

isNotEqualTo

Hint: `isNotEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotEqualTo*

isNotIdenticalTo

Hint: `isNotIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotIdenticalTo*

keys

`keys` allows you to retrieve an asserter *array* containing the tested table keys.

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

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

notContains

`notContains` checks that an array doesn't contains a given data.

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

$this
    ->array($fibonacci)
        ->notContains(null)          // passes
        ->notContains(1)             // fails
        ->notContains(10)            // passes
;
```

Note: `notContains` doesn't search recursively.

Warning:

`notContains` doesn't check the data type.
If you want also to check its type, use *strictlyNotContains*.

notContainsValues

`notContainsValues` checks that an array doesn't contains any data from a given array.

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

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

Note: `notContainsValues` doesn't search recursively.

Warning:

`notContainsValues` doesn't test the data type.
If you also want to check their types, use *strictlyNotContainsValues*.

notHasKey

`notHasKey` checks that an array doesn't contains a given key.

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

$this
    ->array($fibonacci)
        ->notHasKey(0)           // fails
        ->notHasKey(1)           // fails
        ->notHasKey('1')         // fails
        ->notHasKey(10)          // passes

    ->array($atoum)
        ->notHasKey('name')      // fails
        ->notHasKey('price')     // passes
;
```

Note: `notHasKey` doesn't search recursively.

Warning: `notHasKey` doesn't test keys type.

notHasKeys

`notHasKeys` checks that an array doesn't contains any keys from a given array.

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

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

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

Note: `notHasKeys` doesn't search recursively.

Warning: `notHasKeys` doesn't test keys type.

size

`size` allow you to retrieve an *integer* containing the size of tested array.

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

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

strictlyContains

`strictlyContains` checks that an array contains some data (same value and same type).

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

$this
    ->array($fibonacci)
        ->strictlyContains('1')     // passes
        ->strictlyContains(1)       // fails
```

```
        ->strictlyContains('2')     // fails
        ->strictlyContains(2)       // passes
        ->strictlyContains(10)      // fails
;
```

Note: `strictlyContains` doesn't search recursively.

Warning:

`strictlyContains` test data type.
If you don't want to check the type, use *contains*.

strictlyContainsValues

`strictlyContainsValues` checks that an array contains all given data (same value and same type).

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

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

Note: `strictlyContainsValue` doesn't search recursively.

Warning:

`strictlyContainsValues` test data type.
If you don't want to check the types, use *containsValues*.

strictlyNotContains

`strictlyNotContains` check that an array doesn't contains a data (same value and same type).

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

$this
    ->array($fibonacci)
        ->strictlyNotContains(null)          // passes
        ->strictlyNotContains('1')           // fails
        ->strictlyNotContains(1)             // passes
```

```
        ->strictlyNotContains(10)           // passes
;
```

Note: `strictlyNotContains` doesn't search recursively.

Warning:

`strictlyNotContains` test data type.
If you don't want to check the type, use *contains*.

strictlyNotContainsValues

`strictlyNotContainsValues` checks that an array doesn't contains any of given data (same value and same type).

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

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

Note: `strictlyNotContainsValues` doesn't search recursively.

Warning:

`strictlyNotContainsValues` tests data type.
If you don't want to check the types, use *notContainsValues*.

values

`keys` allows you to retrieve an asserter *array* containing the tested table values.

Example:

```
<?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')
;
```

History

Version	Changes
v2.9.0	values assertion added.

boolean

This is the assertion dedicated to booleans.

If you try to test a variable that is not a boolean with this assertion, it will fail.

Note: `null` is not a boolean. Report the the PHP manual to know what is_bool considers or not to be a boolean.

isEqualTo

Hint: `isEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isEqualTo*

isFalse

`isFalse` check that the boolean is strictly equal to `false`.

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

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

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

isIdenticalTo

Hint: `isIdenticalTo` is a method inherited from `variable` asserter. For more information, refer to the documentation of *variable::isIdenticalTo*

isNotEqualTo

Hint: `isNotEqualTo` is a method inherited from `variable` asserter. For more information, refer to the documentation of *variable::isNotEqualTo*

isNotIdenticalTo

Hint: `isNotIdenticalTo` is a method inherited from `variable` asserter. For more information, refer to the documentation of *variable::isNotIdenticalTo*

isTrue

`isTrue` checks that the boolean is strictly equal to `true`.

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

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

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

castToString

It's the assertion dedicated to tests on the cast of objects to strings.

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

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

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

contains

Hint: `contains` is a method herited from `string` asserter. For more information, refer to the documentation of *string::contains*

notContains

Hint: `notContains` is a method herited from `string` asserter. For more information, refer to the documentation of *string::notContains*

hasLength

Hint: `hasLength` is a method herited from `string` asserter. For more information, refer to the documentation of *string::hasLength*

hasLengthGreaterThan

Hint: `hasLengthGreaterThan` is a method inherited from `string` asserter. For more information, refer to the documentation for *string::hasLengthGreaterThan*

hasLengthLessThan

Hint: `hasLengthLessThan` is a method inherited from `string` asserter. For more information, refer to the documentation for *string::hasLengthLessThan*

isEmpty

Hint: `isEmpty` is a method inherited from `string` asserter. For more information, refer to the documentation of *string::isEmpty*

isEqualTo

Hint: `isEqualTo` is a method inherited from `variable` asserter. For more information, refer to the documentation of *variable::isEqualTo*

isEqualToContentsOfFile

Hint: `isEqualToContentsOfFile` is a method inherited from `string` asserter. For more information, refer to the documentation of *string::isEqualToContentsOfFile*

isIdenticalTo

Hint: `isIdenticalTo` is a method inherited from `variable` asserter. For more information, refer to the documentation of *variable::isIdenticalTo*

isNotEmpty

Hint: `isNotEmpty` is a method inherited from `string` asserter. For more information, refer to the documentation of *string::isNotEmpty*

isNotEqualTo

Hint: `isNotEqualTo` is a method inherited from `variable` asserter. For more information, refer to the documentation of *variable::isNotEqualTo*

isNotIdenticalTo

Hint: `isNotIdenticalTo` is a method inherited from `variable` asserter. For more information, refer to the documentation of *variable::isNotIdenticalTo*

matches

Hint: `matches` is a method inherited from `string` asserter. For more information, refer to the documentation of *string::match*

class

It's the assertion dedicated to classes.

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

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

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

Note: The keyword `class` is a reserved word in PHP, it wasn't possible to create a `class` asserter. It's therefore called `phpClass` and an alias `class` has been created. You can meet either `->phpClass()` or `->class()`.

But it's recommended to only use `->class()`.

hasConstant

"hasConstant" checks that the class has the tested constant.

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

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

hasInterface

`hasInterface` checks that the class implements a given interface.

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

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

hasMethod

`hasMethod` checks that the class contains a given method.

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

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

hasNoParent

`hasNoParent` checks that the class doesn't inherit from any class.

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

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

Warning:

A class can implement one or more interfaces, and have no inheritance.
`hasNoParent` doesn't check interfaces, only the inherited classes.

hasParent

`hasParent` checks that the class inherits from a given class.

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

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

Warning:

A class can implement one or more interfaces, and have no inheritance.
`hasParent` doesn't check interfaces, only the inherited classes.

isAbstract

`isAbstract` checks that the class is abstract.

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

isFinal

`isFinal` checks that the class is final.

In this case, we test a non-final class (`StdClass`) :

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

In this case, the tested class is a final one

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

$this
        ->testedClass
                ->isFinal               // passes too
;
```

isSubclassOf

`isSubclassOf` checks that the tested class inherit from given class.

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

dateInterval

It's the assertion dedicated to DateInterval object.

If you try to test a value that is not a `DateInterval` (or a child class) with this assertion it will fail.

isCloneOf

Hint: `isCloneOf` is a method inherited from asserter `object`. For more information, refer to the documentation of *object::isCloneOf*

isEqualTo

`isEqualTo` checks that the duration of object `DateInterval` is equals to to the duration of another `DateInterval` object.

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

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

isGreaterThan

`isGreaterThan` checks that the duration of the object `DateInterval` is higher to the duration of the given `DateInterval` object.

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

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

isGreaterThanOrEqualTo

`isGreaterThanOrEqualTo` checks that the duration of the object `DateInterval` is higher or equals to the duration of another object `DateInterval`.

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

$this
    ->dateInterval($di)
        ->isGreaterThanOrEqualTo(   // passes
            new DateInterval('P1D')
        )
        ->isGreaterThanOrEqualTo(   // passes
            new DateInterval('P2D')
        )
        ->isGreaterThanOrEqualTo(   // fails
            new DateInterval('P3D')
```

```
        )
;
```

isIdenticalTo

Hint: `isIdenticalTo` is an inherited method from `object` asserter. For more information, refer to the documentation of *object::isIdenticalTo*

isInstanceOf

Hint: `isInstanceOf` is a method inherited from asserter `object`. For more information, refer to the documentation of *object::isInstanceOf*

isLessThan

`isLessThan` checks that the duration of the object `DateInterval` is lower than the duration of the given `DateInterval` object.

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

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

isLessThanOrEqualTo

`isLessThanOrEqualTo` checks that the duration of the object `DateInterval` is lower or equals to the duration of another object `DateInterval`.

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

$this
    ->dateInterval($di)
        ->isLessThanOrEqualTo(      // passes
            new DateInterval('P3D')
        )
        ->isLessThanOrEqualTo(      // passes
            new DateInterval('P2D')
        )
```

```
        ->isLessThanOrEqualTo(      // fails
            new DateInterval('P1D')
        )
;
```

isNotEqualTo

Hint: `isNotEqualTo` is a method inherited from `object` asserter. For more information, refer to the documentation of *object::isNotEqualTo*

isNotIdenticalTo

Hint: `isNotIdenticalTo` is an inherited method from `object` asserter. For more information, refer to the documentation of *object::isNotIdenticalTo*

isZero

`isZero` check the duration of `DateInterval` is equal to 0.

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

$this
    ->dateInterval($di1)
        ->isZero()      // passes
    ->dateInterval($di2)
        ->isZero()      // fails
;
```

dateTime

It's the assertion dedicated to DateTime object.

If you try to test a value that is not a `DateTime` (or a child class) with this assertion it will fail.

hasDate

`hasDate` checks the date part of the `DateTime` object.

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

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

```
        ->hasDate('1981', '2',  '13')   // passes
        ->hasDate(1981,   2,    13)     // passes
;
```

hasDateAndTime

`hasDateAndTime` checks date and hour part of the `DateTime` object.

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

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

hasDay

`hasDay` checks day part of the `DateTime` object.

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

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

hasHours

`hasHours` checks time part of the `DateTime` object.

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

$this
    ->dateTime($dt)
        ->hasHours('01')    // passes
        ->hasHours('1')     // passes
        ->hasHours(1)       // passes
;
```

hasMinutes

`hasMinutes` checks minutes part of the `DateTime` object.

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

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

hasMonth

`hasMonth` checks month part of the `DateTime` object.

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

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

hasSeconds

`hasSeconds` checks seconds part of the `DateTime` object.

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

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

hasTime

`hasTime` checks time part of the `DateTime` object.

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

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

hasTimezone

`hasTimezone` checks timezone part of the `DateTime` object.

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

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

hasYear

`hasYear` checks year part of the `DateTime` object.

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

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

isCloneOf

Hint: `isCloneOf` is a method inherited from asserter `object`. For more information, refer to the documentation of *object::isCloneOf*

isEqualTo

Hint: `isEqualTo` is a method inherited from `object` asserter. For more information, refer to the documentation of *object::isEqualTo*

isIdenticalTo

Hint: `isIdenticalTo` is an inherited method from `object` asserter. For more information, refer to the documentation of *object::isIdenticalTo*

isInstanceOf

Hint: `isInstanceOf` is a method inherited from asserter `object`. For more information, refer to the documentation of *object::isInstanceOf*

isNotEqualTo

Hint: `isNotEqualTo` is a method inherited from `object` asserter. For more information, refer to the documentation of *object::isNotEqualTo*

isNotIdenticalTo

Hint: `isNotIdenticalTo` is an inherited method from `object` asserter. For more information, refer to the documentation of *object::isNotIdenticalTo*

error

It's the assertion dedicated to errors.

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

Note: The syntax uses anonymous functions (also called closures) introduced in PHP 5.3. For more details, read the PHP's documentation on anonymous functions.

Warning: The error types E_ERROR, E_PARSE, E_CORE_ERROR, E_CORE_WARNING, E_COMPILE_ERROR, E_COMPILE_WARNING as well as the E_STRICT can't be managed with this function.

exists

`exists` checks that an error was raised during the execution of the previous code.

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

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

notExists

`notExists` checks that no errors was raised during the execution of the previous code.

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

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

withType

`withType` checks the type of the raised error.

```
<?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()
;
```

withAnyType

`withAnyType` does not check the type of the raised error. That's the default behaviour. So `->error()->withAnyType()->exists()` is the equivalent of `->error()->exists()`. This method allow to add semantic to your test.

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

withMessage

`withMessage` checks message content of raised error.

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

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

withAnyMessage

`withAnyMessage` does not check the error message. That's the default behaviour. So `->error()->withAnyMessage()->exists()` is the equivalent of `->error()->exists()`. This method allow to add semantic to your test.

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

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

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

withPattern

`withPattern` checks message content of raised error against a regular expression.

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

$this
    ->when(
```

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

exception

It's the assertion dedicated to exceptions.

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

Note: The syntax uses anonymous functions (also called closures) introduced in PHP 5.3. For more details, read the PHP's documentation on anonymous functions.

We can easily retrieve the last exception with `$this->exception`.

```
<?php
$this
    ->exception(
        function() use($myObject) {
            // This code throws a exception: throw new \Exception('Message', 42);
            $myObject->doOneThing('wrongParameter');
        }
    )->isIdenticalTo($this->exception) // passes
;

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

hasCode

`hasCode` checks exception code.

```
<?php
$this
    ->exception(
        function() use($myObject) {
            // This code throws a exception: throw new \Exception('Message', 42);
            $myObject->doOneThing('wrongParameter');
        }
```

```
    )
        ->hasCode(42)
;
```

hasDefaultCode

`hasDefaultCode` checks that exception code is the default value, 0.

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

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

hasMessage

`hasMessage` checks exception message.

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

hasNestedException

`hasNestedException` checks that the exception contains a reference to another exception. If the exception type is given, this will also checks the exception class.

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

```
    ->exception(
        function() use($myObject) {
            try {
                // This code throws a exception: throw new \FirstException('Message 1
↪', 42);
                $myObject->doOneThing('wrongParameter');
            }
            // ... the exception is caught...
            catch(\FirstException $e) {
                // ... and then throws encapsulated inside a second one
                throw new \SecondException('Message 2', 24, $e);
            }
        }
    )
        ->isInstanceOf('\FirstException')           // fails
        ->isInstanceOf('\SecondException')          // passes

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

isCloneOf

Hint: `isCloneOf` is a method inherited from asserter `object`. For more information, refer to the documentation of *object::isCloneOf*

isEqualTo

Hint: `isEqualTo` is a method inherited from `object` asserter. For more information, refer to the documentation of *object::isEqualTo*

isIdenticalTo

Hint: `isIdenticalTo` is an inherited method from `object` asserter. For more information, refer to the documentation of *object::isIdenticalTo*

isInstanceOf

Hint: `isInstanceOf` is a method inherited from asserter `object`. For more information, refer to the documentation of *object::isInstanceOf*

isNotEqualTo

Hint: `isNotEqualTo` is a method inherited from `object` asserter. For more information, refer to the documentation of *object::isNotEqualTo*

isNotIdenticalTo

Hint: `isNotIdenticalTo` is an inherited method from `object` asserter. For more information, refer to the documentation of *object::isNotIdenticalTo*

message

`message` allow you to get an asserter of type *string* containing the tested exception message.

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

extension

It's the asserter dedicated to PHP extension.

isLoaded

Check if the extension is loaded (installed and enabled).

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

Note: If you need to run tests only if an extension is present, you can use the *PHP annotation*.

float

It's the assertion dedicated to decimal numbers.

If you try to test a variable that is not a decimal number with this assertion, it will fail.

Note: `null` is not a decimal number. Refer to the PHP manual to know what is_float considered or not as a float.

isEqualTo

Hint: `isEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isEqualTo*

isGreaterThan

Hint: `isGreaterThan` is a method inherited from the `integer` asserter. For more information, refer to the documentation of *integer::isGreaterThan*

isGreaterThanOrEqualTo

Hint: `isGreaterThanOrEqualTo` is a method inherited from the `integer` asserter. For more information, refer to the documentation of *integer::isGreaterThanOrEqualTo*

isIdenticalTo

Hint: `isIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isIdenticalTo*

isLessThan

Hint: `isLessThan` is a method inherited from the `integer` asserter. For more informations, refer to the documentation of *integer::isLessThan*

isLessThanOrEqualTo

Hint: `isLessThanOrEqualTo` is a method inherited from the `integer` asserter. For more information, refer to the documentation of *integer::isLessThanOrEqualTo*

isNearlyEqualTo

`isNearlyEqualTo` checks that the float is approximately equal to the value received as an argument.

Indeed, in computer science, decimal numbers are managed in a way that does not allow for accurate comparisons without the use of specialized tools. Try for example to run the following command:

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

The result should be "true".

Note: For more information on this topics, read the PHP documentation on the float precision.

This method is therefore seeking to reduce this problem.

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

$this
    ->float($float)
        ->isNearlyEqualTo(0.03) // passes
        ->isEqualTo(0.03)       // fails
;
```

Note: For more information about the algorithm used, see the floating point guide.

isNotEqualTo

Hint: `isNotEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotEqualTo*

isNotIdenticalTo

Hint: `isNotIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotIdenticalTo*

isZero

Hint: `isZero` is a method inherited from the `integer` asserter. For more information, refer to the documentation of *integer::isZero*

hash

It's the assertion dedicated to tests on hashes (digital fingerprints).

contains

Hint: `contains` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::contains*

isEqualTo

Hint: `isEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isEqualTo*

isEqualToContentsOfFile

Hint: `isEqualToContentsOfFile` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::isEqualToContentsOfFile*

isIdenticalTo

Hint: `isIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isIdenticalTo*

isMd5

`isMd5` checks that the string is a `md5` format, i.r. a hexadecimal string of 32 length.

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

$this
```

```
    ->hash($hash)
        ->isMd5()       // passes
    ->hash($notHash)
        ->isMd5()       // fails
;
```

isNotEqualTo

Hint: `isNotEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotEqualTo*

isNotIdenticalTo

Hint: `isNotIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotIdenticalTo*

isSha1

`isSha1` checks that the string is a `sha1` format, i.e. a hexadecimal string of 40 length.

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

$this
    ->hash($hash)
        ->isSha1()      // passes
    ->hash($notHash)
        ->isSha1()      // fails
;
```

isSha256

`isSha256` checks that the string is a `sha256` format, i.e. a hexadecimal string of 64 length.

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

$this
    ->hash($hash)
        ->isSha256()    // passes
    ->hash($notHash)
        ->isSha256()    // fails
;
```

isSha512

`isSha512` checks that the string is a `sha512` format, i.e. a hexadecimal string of 128 length.

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

$this
    ->hash($hash)
        ->isSha512()     // passes
    ->hash($notHash)
        ->isSha512()     // fails
;
```

notContains

Hint: `notContains` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::notContains*

integer

It's the assertion dedicated to integers.

If you try to test a variable that is not an integer with this assertion, it will fail.

Note: `null` isn't an integer. Refer to the PHP's manual is_int to known what's considered as an integer or not.

isEqualTo

Hint: `isEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isEqualTo*

isGreaterThan

`isGreaterThan` checks that the integer is strictly higher than given one.

```
<?php
$zero = 0;

$this
    ->integer($zero)
        ->isGreaterThan(-1)      // passes
        ->isGreaterThan('-1')    // fails because "-1"
                                 // isn't an integer
```

```
        ->isGreaterThan(0)      // fails
;
```

isGreaterThanOrEqualTo

`isGreaterThanOrEqualTo` checks that an integer is higher or equal to a given one.

```
<?php
$zero = 0;

$this
    ->integer($zero)
        ->isGreaterThanOrEqualTo(-1)     // passes
        ->isGreaterThanOrEqualTo(0)      // passes
        ->isGreaterThanOrEqualTo('-1')   // fails because "-1"
                                         // isn't an integer
;
```

isIdenticalTo

Hint: `isIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isIdenticalTo*

isLessThan

`isLessThan` checks that the integer is strictly lower than a given one.

```
<?php
$zero = 0;

$this
    ->integer($zero)
        ->isLessThan(10)     // passes
        ->isLessThan('10')   // fails because"10" isn't an integer
        ->isLessThan(0)      // fails
;
```

isLessThanOrEqualTo

`isLessThanOrEqualTo` checks that an integer is lower or equal to a given one.

```
<?php
$zero = 0;

$this
    ->integer($zero)
        ->isLessThanOrEqualTo(10)       // passes
        ->isLessThanOrEqualTo(0)        // passes
        ->isLessThanOrEqualTo('10')     // fails because "10"
```

```
                                        // isn't an integer
;
```

isNotEqualTo

Hint: `isNotEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotEqualTo*

isNotIdenticalTo

Hint: `isNotIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotIdenticalTo*

isZero

`isZero` checks that the integer is equal to 0.

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

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

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

Note: `isZero` is equivalent to `isEqualTo(0)`.

mock

It's the assertion dedicated to mocks.

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

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

Note: Refer to the documentation of *mock* for more information on how to create and manage mocks.

call

`call` let you specify which method of mock to check, it call must be followed by a call to one of the following verification method like *atLeastOnce*, *once/twice/thrice*, *exactly*, etc...

```
<?php

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

    ->if($object->methodThatCallMyMethod())  // This will call myMethod from $mock
    ->then

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

after

`after` checks if the method has been called after the one passed as parameter.

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

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

atLeastOnce

`atLeastOnce` check that the tested method (see *call*) from the mock has been called at least once.

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

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

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

before

`before` checks if the method has been called before the one passed as parameter.

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

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

exactly

`exactly` check that the tested method (see *call*) has been called a specific number of times.

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

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

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

Note: You can have a simplified version with `->{2}`.

never

`never` check that the tested method (see *call*) has never been called.

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

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

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

Note: `never` is equivalent to *exactly(0)*.

once/twice/thrice

This asserters check that the tested method (see *call*) from the tested mock has been called exactly:

- once
- twice
- 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` and `thrice` are respectively equivalent to *exactly(1)*, *exactly(2)* and *exactly(3)*.

withAnyArguments

`withAnyArguments` allow to check any argument, non-specified, when we call the tested method (see *call*) of tested mock.

This method is useful to reset the arguments of tested method, like in the following example:

```
<?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` let you specify the expected arguments that the tested method should receive when called (see *call*).

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

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

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

Warning:

`withArguments` does not check the arguments type.
If you also want to check the type, use *withIdenticalArguments*.

withIdenticalArguments

`withIdenticalArguments` let you specify the expected typed arguments that tested method should receive when called (see *call*).

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

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

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

Warning:

`withIdenticalArguments` checks the arguments type.
If you do not want to check the type, use *withArguments*.

withAtLeastArguments

`withAtLeastArguments` let you specify the minimum expected arguments that tested method should receive when called (see *call*).

```
<?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
;
```

Warning:

`withAtLeastArguments` does not check the arguments type.
If you also want to check the type, use *withAtLeastIdenticalArguments*.

withAtLeastIdenticalArguments

`withAtLeastIdenticalArguments` let you specify the minimum expected typed arguments that tested method should receive when called (see *call*).

```
<?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
;
```

Warning:

`withAtLeastIdenticalArguments` checks the arguments type.
If you do not want to check the type, use *withIdenticalArguments*.

withoutAnyArgument

`withoutAnyArgument` lets you indicate that the method should not receive any argument when called (see *call*).

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

Note: `withoutAnyArgument` is equivalent to call *withAtLeastArguments* with an empty array: `->withAtLeastArguments(array())`.

receive

It's an alias of *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;

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

wasCalled

`wasCalled` checks that at least one method of the mock has been called at least once.

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

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

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

wasNotCalled

`wasNotCalled` checks that no method of the mock has been called.

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

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

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

mysqlDateTime

It's the assertion dedicated to objects representing MySQL date and based on DateTime object.

Dates must use a format compatible with MySQL and many other DBMSS (database management system), i.e. "Y-m-d H:i:s"

Note: For more information, refer to the documentation of the date() function from the PHP manual.

If you try to test a value that's not a `DateTime` (or a child class) with this assertion it will fail.

hasDate

Hint: `hasDate` is a method inherited from the `dateTime` asserter. For more information, refer to the documentation of *dateTime::hasDate*

hasDateAndTime

Hint: `hasDateAndTime` is a method inherited from the `dateTime` asserter. For more informations, refer to the documentation of *dateTime::hasDateAndTime*

hasDay

Hint: `hasDay` is a method inherited from the `dateTime` asserter. For more information, refer to the documentation of *dateTime::hasDay*

hasHours

Hint: `hasHours` is a method inherited from the `dateTime` asserter. For more information, refer to the documentation of *dateTime::hasHours*

hasMinutes

Hint: `hasMinutes` is a method inherited from the `dateTime` asserter. For more information, refer to the documentation of *dateTime::hasMinutes*

hasMonth

Hint: `hasMonth` is a method inherited from the `dateTime` asserter. For more information, refer to the documentation of *dateTime::hasMonth*

hasSeconds

Hint: `hasSeconds` is a method inherited from the `dateTime` asserter. For more information, refer to the documentation of *dateTime::hasSeconds*

hasTime

Hint: `hasTime` is a method inherited from the `dateTime` asserter. For more information, refer to the documentation of *dateTime::hasTime*

hasTimezone

Hint: `hasTimezone` is a method inherited from the `dateTime` asserter. For more information, refer to the documentation of *dateTime::hasTimezone*

hasYear

Hint: `hasYear` is a method inherited from the `dateTime` asserter. For more information, refer to the documentation of *dateTime::hasYear*

isCloneOf

Hint: `isCloneOf` is a method inherited from asserter `object`. For more information, refer to the documentation of *object::isCloneOf*

isEqualTo

Hint: `isEqualTo` is a method inherited from `object` asserter. For more information, refer to the documentation of *object::isEqualTo*

isIdenticalTo

Hint: `isIdenticalTo` is an inherited method from `object` asserter. For more information, refer to the documentation *object::isIdenticalTo*

isInstanceOf

Hint: `isInstanceOf` is a method inherited from asserter `object`. For more information, refer to the documentation of *object::isInstanceOf*

isNotEqualTo

Hint: `isNotEqualTo` is a method inherited from `object` asserter. For more information, refer to the documentation of *object::isNotEqualTo*

isNotIdenticalTo

Hint: `isNotIdenticalTo` is an inherited method from `object` asserter. For more information, refer to the documentation of *object::isNotIdenticalTo*

object

It's the assertion dedicated to objects.

If you try to test a variable that is not an object with this assertion, it will fail.

Note: `null` isn't an object. Refer to the PHP's manual is_object to know what is considered as an object or not.

hasSize

`hasSize` checks the size of an object that implements the interface `Countable`.

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

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

isCallable

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

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

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

Note: To be identified as `callable`, your objects should be instantiated from classes that implements the magic __invoke.

Hint: `isCallable` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isCallable*

isCloneOf

`isCloneOf` checks an object is clone of a given one, that is the objects are equal but are not the same instance.

```
<?php
$object1 = new \StdClass;
$object2 = new \StdClass;
$object3 = clone($object1);
$object4 = new \StdClass;
```

```
$object4->foo = 'bar';

$this
    ->object($object1)
        ->isCloneOf($object2)    // passes
        ->isCloneOf($object3)    // passes
        ->isCloneOf($object4)    // fails
;
```

Note: For more details, read the PHP's documentation about comparing objects.

isEmpty

`isEmpty` checks the size of an object that implements the `Countable` interface is equal to 0.

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

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

Note: `isEmpty` is equivalent to `hasSize(0)`.

isEqualTo

`isEqualTo` checks that an object is equal to another. Two objects are consider equals when they have the same attributes and values, and they are instances of the same class.

Note: For more details, read the PHP's documentation about comparing objects.

Hint: `isEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isEqualTo*

isIdenticalTo

`isIdenticalTo` checks that two objects are identical. Two objects are considered identical when they refer to the same instance of the same class.

Note: For more details, read the PHP's documentation about comparing objects.

Hint: `isIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isIdenticalTo*

isInstanceOf

`isInstanceOf` checks that an object is:

- an instance of the given class,
- a subclass from the given class (abstract or not),
- an instance of class that implements a given interface.

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

$this
    ->object($object)
        ->isInstanceOf('\StdClass')     // passes
        ->isInstanceOf('\Iterator')     // fails
;


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')     // passes
        ->isInstanceOf('\FooInterface') // passes
        ->isInstanceOf('\BarClass')     // fails
        ->isInstanceOf('\StdClass')     // fails

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

Note: The name of the classes and the interfaces must be absolute, because any namespace imports are ignored.

Hint: Notice that with PHP >= 5.5 you can use the keyword `class` to get the absolute class names, for example `$this->object($foo)->isInstanceOf(FooClass::class)`.

isInstanceOfTestedClass

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

isNotCallable

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

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

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

Hint: `isNotCallable` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotCallable*

isNotEqualTo

`isEqualTo` checks that an object is not equal to another. Two objects are consider equals when they have the same attributes and values, and they are instance of the same class.

Note: For more details, read the PHP's documentation about comparing objects.

Hint: `isNotEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotEqualTo*

isNotIdenticalTo

`isIdenticalTo` checks the two objects are not identical. Two objects are considered identical when they refer to the same instance of same class.

Note: For more details, read the PHP's documentation about comparing objects.

Hint: `isNotIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotIdenticalTo*

isNotInstanceOf

`isNotInstanceOf` check that an object is not:

- an instance of the given class,
- a subclass from the given class (abstract or not),
- an instance of class that implements a given interface.

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

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

Note: As for *isInstanceOf*, the name of the classes and the interfaces must be absolute, because any namespace imports are ignored.

isNotTestedInstance

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

isTestedInstance

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

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

toString

The toString assertion casts the object to a string a returns a string asserter on the casted value.

Example:

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

output

It's the assertion dedicated to tests on outputs, so everything witch supposed to be displayed on the screen.

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

Note: The syntax uses anonymous functions (also called closures) introduced in PHP 5.3. For more details, read the PHP's documentation on anonymous functions.

contains

Hint: `contains` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::contains*

hasLength

Hint: `hasLength` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::hasLength*

hasLengthGreaterThan

Hint: `hasLengthGreaterThan` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::hasLengthGreaterThan*

hasLengthLessThan

Hint: `hasLengthLessThan` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::hasLengthLessThan*

isEmpty

Hint: `isEmpty` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::isEmpty*

isEqualTo

Hint: `isEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isEqualTo*

isEqualToContentsOfFile

Hint: `isEqualToContentsOfFile` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::isEqualToContentsOfFile*

isIdenticalTo

Hint: `isIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isIdenticalTo*

isNotEmpty

Hint: `isNotEmpty` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::isNotEmpty*

isNotEqualTo

Hint: `isNotEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotEqualTo*

isNotIdenticalTo

Hint: `isNotIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotIdenticalTo*

matches

Hint: `matches` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::match*

notContains

Hint: `notContains` is a method herited from the `string` asserter. For more information, refer to the documentation of *string::notContains*

resource

It's the assertion dedicated to the ´resources <http://php.net/language.types.resource>´_.

isOfType

This method compare the type of resource with the type of the given value provided by the argument. In the following example, we checks that the given value is a stream.

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

isStream

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

->is*() will match the type of the stream against a pattern computed from the method name: ->isFooBar() will try to match a stream with type foo bar, fooBar, foo_bar, ...

type

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

->$type is an helper providing a *string asserter* on the stream type.

sizeOf

It's the assertion dedicated to tests on the size of the arrays and objects implementing the interface `Countable`.

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

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

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

isEqualTo

Hint: `isEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isEqualTo*

isGreaterThan

Hint: `isGreaterThan` is a method inherited from the `integer` asserter. For more information, refer to the documentation of *integer::isGreaterThan*

isGreaterThanOrEqualTo

Hint: `isGreaterThanOrEqualTo` is a method inherited from the `integer` asserter. For more information, refer to the documentation of *integer::isGreaterThanOrEqualTo*

isIdenticalTo

Hint: `isIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isIdenticalTo*

isLessThan

Hint: `isLessThan` is a method inherited from the `integer` asserter. For more information, refer to the documentation of *integer::isLessThan*

isLessThanOrEqualTo

Hint: `isLessThanOrEqualTo` is a method inherited from the `integer` asserter. For more information, refer to the documentation of *integer::isLessThanOrEqualTo*

isNotEqualTo

Hint: `isNotEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotEqualTo*

isNotIdenticalTo

Hint: `isNotIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotIdenticalTo*

isZero

Hint: `isZero` is a method inherited from the `integer` asserter. For more information, refer to the documentation of *integer::isZero*

stream

It's the assertion dedicated to the ´streams <http://php.net/intro.stream>´_.

It's based on atoum virtual filesystem (VFS). A new stream wrapper will be registered (starting with `atoum://`).

The mock will create a new file in the VFS and the steam path will be accessible via the `getPath` method on the stream controller (something like `atoum://mockUniqId`).

isRead

`isRead` checks if a mocked stream has been read.

```
<?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() // fails
;
```

isWritten

`isWritten` checks if a mocked stream has been written.

```
<?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() // passes
;

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

isWrited

Hint: `isWrited` is an alias to the `isWritten` method. For more information, refer to the documentation of *stream::isWritten*

string

It's the assertion dedicated to the strings.

contains

`contains` checks that a string contains another given string.

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

$this
    ->string($string)
        ->contains('ll')    // passes
        ->contains(' ')     // passes
        ->contains('php')   // fails
;
```

endWith

`endWith` checks that a string ends with another given string.

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

$this
    ->string($string)
        ->endWith('world')     // passes
        ->endWith('lo world')  // passes
        ->endWith('Hello')     // fails
        ->endWith(' ')         // fails
;
```

hasLength

`hasLength` checks the string size.

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

$this
    ->string($string)
        ->hasLength(11)     // passes
```

```
        ->hasLength(20)      // fails
;
```

hasLengthGreaterThan

`hasLengthGreaterThan` checks that the string size is greater that the given one.

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

$this
    ->string($string)
        ->hasLengthGreaterThan(10)      // passes
        ->hasLengthGreaterThan(20)      // fails
;
```

hasLengthLessThan

`hasLengthLessThan` checks that the string size is lower that the given one.

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

$this
    ->string($string)
        ->hasLengthLessThan(20)      // passes
        ->hasLengthLessThan(10)      // fails
;
```

isEmpty

`isEmpty` checks that the string is empty.

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

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

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

isEqualTo

Hint: `isEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isEqualTo*

isEqualToContentsOfFile

`isEqualToContentsOfFile` checks that the string is equal to the content of a file given by its path.

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

Note: if the file doesn't exist, the test will fails.

isIdenticalTo

Hint: `isIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isIdenticalTo*

isNotEmpty

`isNotEmpty` checks that the string is not empty.

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

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

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

isNotEqualTo

Hint: `isNotEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotEqualTo*

isNotIdenticalTo

Hint: `isNotIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotIdenticalTo*

length

`length` allows you to get an asserter of type *integer* that contains the string's size.

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

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

match

Hint: `match` is an alias of the `matches` method. For more information, refer to the documentation of *string::matches*

matches

`matches` checks that a regular expression match the tested string.

```
<?php
$phone = '0102030405';
$vdm   = "Today at 57 years, my father got a tatoot of a Unicorn on his shoulder. VDM
↪";

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

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

notContains

`notContains` checks that the tested string doesn't contains another string.

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

$this
    ->string($string)
        ->notContains('php')    // passes
        ->notContains(';')      // passes
        ->notContains('ll')     // fails
        ->notContains(' ')      // fails
;
```

notEndWith

`notEndWith` checks that the tested string doesn't ends with another string.

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

$this
    ->string($string)
        ->notEndWith('Hello')      // passes
        ->notEndWith(' ')          // passes
        ->notEndWith('world')  // fails
        ->notEndWith('lo world')        // fails
;
```

notStartWith

`notStartWith` checks that the tested string doesn't starts with another string.

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

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

startWith

`startWith` checks that the tested string starts with another string.

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

$this
    ->string($string)
        ->startWith('Hello wo') // passes
        ->startWith('He')       // passes
        ->startWith('world')    // fails
        ->startWith(' ')        // fails

;
```

utf8String

It's the asserter dedicated to UTF-8 strings.

Note: `utf8Strings` use the functions `mb_*` to manage multi-byte strings. Refer to the PHP manual for more information about mbstring extension.

contains

Hint: `contains` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::contains*

hasLength

Hint: `hasLength` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::hasLength*

hasLengthGreaterThan

Hint: `hasLengthGreaterThan` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::hasLengthGreaterThan*

hasLengthLessThan

Hint: `hasLengthLessThan` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::hasLengthLessThan*

isEmpty

Hint: `isEmpty` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::isEmpty*

isEqualTo

Hint: `isEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isEqualTo*

isEqualToContentsOfFile

Hint: `isEqualToContentsOfFile` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::isEqualToContentsOfFile*

isIdenticalTo

Hint: `isIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isIdenticalTo*

isNotEmpty

Hint: `isNotEmpty` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::isNotEmpty*

isNotEqualTo

Hint: `isNotEqualTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotEqualTo*

isNotIdenticalTo

Hint: `isNotIdenticalTo` is a method inherited from the `variable` asserter. For more information, refer to the documentation of *variable::isNotIdenticalTo*

matches

Hint: `matches` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::match*

Note: Remember to add `u` in your regular expression, in the option part. For more precision, read the PHP's documentation about the options for search in regular expression.

```
<?php
$vdm = "Today at 57 years, my father got a tatoo of a Unicorn on his shoulder. FML";
```

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

notContains

Hint: `notContains` is a method inherited from the `string` asserter. For more information, refer to the documentation of *string::notContains*

variable

It's the basic assertion of all variables. It contains the necessary tests for any type of variable.

isCallable

`isCallable` verifies that the variable can be called as a function.

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

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

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

    ->variable(array('\Vendor\Project\Foo', 'bar'))
        ->isCallable()

    ->variable('\Vendor\Project\Foo::bar')
        ->isCallable()
;
```

isEqualTo

`isEqualTo` verifies that the variable is equal to a given value.

```
<?php
$a = 'a';

$this
    ->variable($a)
        ->isEqualTo('a')    // passes
;
```

Warning:

`isEqualTo` doesn't test the type of variable.
If you also want to check the type, use *isIdenticalTo*.

isIdenticalTo

`isIdenticalTo` checks that the variable has the same value and the same type than the given data. In the case of an object, `isIdenticalTo` checks that the data is referencing the same instance.

```
<?php
$a = '1';

$this
    ->variable($a)
        ->isIdenticalTo(1)          // fails
;

$stdClass1 = new \StdClass();
$stdClass2 = new \StdClass();
$stdClass3 = $stdClass1;

$this
    ->variable($stdClass1)
        ->isIdenticalTo(stdClass3)  // passes
        ->isIdenticalTo(stdClass2)  // fails
;
```

Warning:

`isIdenticalTo` test the type of variable.
If you don't want to check its type, use *isEqualTo*.

isNotCallable

`isNotCallable` checks that the variable can't be called like a function.

```
<?php
$f = function() {
    // code
};
$int    = 1;
$string = 'nonExistingMethod';

$this
    ->variable($f)
        ->isNotCallable()   // fails

    ->variable($int)
        ->isNotCallable()   // passes

    ->variable($string)
```

```
        ->isNotCallable()    // passes

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

isNotEqualTo

`isNotEqualTo` checks that the variable does not have the same value as the given one.

```
<?php
$a       = 'a';
$aString = '1';

$this
    ->variable($a)
        ->isNotEqualTo('b')      // passes
        ->isNotEqualTo('a')      // fails

    ->variable($aString)
        ->isNotEqualTo($a)       // fails
;
```

Warning:

`isNotEqualTo` doesn't test the type of variable.
If you also want to check the type, use *isNotIdenticalTo*.

isNotIdenticalTo

`isNotIdenticalTo` checks that the variable does not have the same type nor the same value than the given one.

In the case of an object, `isNotIdenticalTo` checks that the data isn't referencing on the same instance.

```
<?php
$a = '1';

$this
    ->variable($a)
        ->isNotIdenticalTo(1)           // passes
;

$stdClass1 = new \StdClass();
$stdClass2 = new \StdClass();
$stdClass3 = $stdClass1;

$this
    ->variable($stdClass1)
        ->isNotIdenticalTo(stdClass2)   // passes
        ->isNotIdenticalTo(stdClass3)   // fails
;
```

Warning:

`isNotIdenticalTo` test the type of variable.
If you don't want to check its type, use *isNotEqualTo*.

isNull

`isNull` checks that the variable is null.

```
<?php
$emptyString = '';
$null        = null;

$this
    ->variable($emptyString)
        ->isNull()              // fails
                                // (it's empty but not null)

    ->variable($null)
        ->isNull()              // passes
;
```

isNotNull

`isNotNull` checks that the variable is not null.

```
<?php
$emptyString = '';
$null        = null;

$this
    ->variable($emptyString)
        ->isNotNull()           // passes (it's empty but not null)

    ->variable($null)
        ->isNotNull()           // fails
;
```

isNotTrue

`isNotTrue` check that the variable is strictly not equal to `true`.

```
<?php
$true  = true;
$false = false;
$this
    ->variable($true)
        ->isNotTrue()     // fails

    ->variable($false)
        ->isNotTrue()     // succeed
;
```

isNotFalse

`isNotFalse` check that the variable is strictly not equal to `false`.

```
<?php
$true  = true;
$false = false;
$this
    ->variable($false)
        ->isNotFalse()      // fails

    ->variable($true)
        ->isNotFalse()      // succeed
;
```

generator

It's the assertion dedicated to tests on generators.

The generator asserter extends the iterator asserter, so you can use any assertion from the iterator asserter.

Example:

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

In this example we create a generator that yields 3 values, and we check that the size of the generator is 3.

yields

`yields` is used to ease the test on values yielded by the generator. Each time you will call `->yields` the next value of the generator will be retrived. You will be able to use any other asserter on this value (for example `class`, `string` or `variable`).

Example:

```
<?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)
;
```

In this example we create a generator that yields 3 values : 1, 2 and 3. Then we yield each value and run an assertion on this value to check it's type and value. In the first two yields we use the variable asserter and only check the value. In the third yields call we add a check on the type of the value by using the integer asserter (any asserter could by used on this value) before checking the value.

returns

Note: This assertion will only work on PHP >= 7.0.

Since the version 7.0 of PHP, generators can return a value that can retrived via a call to the `->getReturn()` method. When you call `->returns` on the generator asserter, atoum will retrive the value from a call on the `->getReturn()` method on the asserter. Then you will be able to use any other asserter on this value just like the `yields` assertion.

Example:

```
<?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)
;
```

In this example we run some checks on all the yielded values. Then, we checks that the generator returns a integer with a value of 42 (just like a call to the yields assertion, you can use any asserter to check to returned value).

History

Version	Changes
v3.0.0	Generator asserter added.

Asserter & assertion tips

Several tips & trick are available for the assertion. Knowing them can simplify your life ;)

The first one is that all assertion are fluent. So you can chain them, just look at the previous examples.

You should also know that all assertions without parameter can be written with or without parenthesis. So `$this->integer(0)->isZero()` is the same as `$this->integer(0)->isZero`.

Alias

Sometimes you want to use something that reflect your vocabulary or your domain. atoum provide a simple mechanism, alias. Here is an example:

```
<?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)
        ;
    }
}
```

In this example, we create an alias that will create an asserter `equals` that will act exactly the same as `isEqualTo`. We could also use *beforeTestMethod* instead of the constructor. The best is to create a base class for all the test inside your project that you can extends instead of `\atoum\test`.

Custom asserter

Now that we have seen alias, we can go further by creating a custom asserter. Here is an example of an asserter for credit card.

```
<?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 $reflectionClassFactory =␣
↪null)
    {
        parent::__construct($adapter, $annotationExtractor, $asserterGenerator,
↪$assertionManager, $reflectionClassFactory);

        $this->getAsserterGenerator()->addNamespace('tests\units');
    }

    public function testFoo()
    {
        $this
            ->creditcard('4444555566660000')->isValid()
        ;
    }
}
```

So, like the alias, the best is to create a base class your test and declare your custom asserters there.

Short syntax

With *alias* you can define some intresting things. But because atoum tries to help you in the redaction of your test, we added several aliases.

- **==** is the same as the asserter *isEqualTo*
- **===** is the same as the asserter *isIdenticalTo*
- **!=** is the same as the asserter *isNotEqualTo*
- **!==** is the same as the asserter *isNotIdenticalTo*
- **<** is the same as the asserter *isLessThan*
- **<=** is the same as the asserter *isLessThanOrEqualTo*
- **>** is the same as the asserter *isGreaterThan*
- **>=** is the same as the asserter *isGreaterThanOrEqualTo*

```
<?php
namespace tests\units;

use atoum;

class stdClass extends atoum
{
    public function testFoo()
    {
        $this
            ->variable('foo')->{'=='}('foo')
            ->variable('foo')->{'foo'} // same as previous line
            ->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)
        ;
    }
}
```

CHAPTER 7

Mocking systems

Mocks are virtual class, created on the fly. They are used to isolate your test from the behaviour of the other classes. atoum has a powerful and easy-to-implement mock system allowing you to generate mocks from classes or interfaces that exist, are virtual, are abstract or an interface.

With these mocks, you can simulate behaviours by redefining the *public* methods of your classes. For the private & protected method, use the visibility extension.

> **Warning:** Most of method that configure the mock, apply only for the next mock generation!

Generate a mock

There are several ways to create a mock from an interface or a class. The simplest one is to create an object with the absolute name prefixed by `mock`:

```
<?php
// creation of a mock of the interface \Countable
$countableMock = new \mock\Countable;

// creation of a mock from the abstract class
// \Vendor\Project\AbstractClass
$vendorAppMock = new \mock\Vendor\Project\AbstractClass;

// creation of mock of the \StdClass class
$stdObject     = new \mock\StdClass;

// creation of a mock from a non-existing class
$anonymousMock = new \mock\My\Unknown\Claass;
```

Generate a mock with newMockInstance

If you prefer there is method called `newMockInstance()` that will generate a mock.

```
<?php
// creation of a mock of the interface \Countable
$countableMock = new \mock\Countable;

// is equivalent to
$this->newMockInstance('Countable');
```

Note: Like the mock generator, you can give extra parameters: `$this->newMockInstance('class name', 'mock namespace', 'mock class name', ['constructor args']);`

The mock generator

atoum relies on a specialised components to generate the mock: the `mockGenerator`. You have access to the latter in your tests in order to modify the procedure for the generation of the mocks.

By default, the mock will be generated in the "mock" namespace and behave exactly in the same way as instances of the original class (mock inherits directly from the original class).

Change the name of the class

If you wish to change the name of the class or its namespace, you must use the `mockGenerator`.

Its `generate` method takes 3 parameters:

- the name of the interface or class to mock ;
- the new namespace, optional ;
- the new name of class, optional.

```
<?php
// creation of a mock of the interface \Countable to \MyMock\Countable
// we only change the namespace
$this->mockGenerator->generate('\Countable', '\MyMock');

// creation of a mock from the abstract class
// \Vendor\Project\AbstractClass to \MyMock\AClass
// change the namespace and class name
$this->mockGenerator->generate('\Vendor\Project\AbstractClass', '\MyMock', 'AClass');

// creation of a mock of \StdClass to \mock\OneClass
// We only changes the name of the class
$this->mockGenerator->generate('\StdClass', null, 'OneClass');

// we can now instantiate these mocks
$vendorAppMock = new \myMock\AClass;
$countableMock = new \myMock\Countable;
$stdObject     = new \mock\OneClass;
```

Note: If you use only the first argument and do not change the namespace or the name of the class, then the first solution is equivalent, easiest to read and recommended.

You can access to the code from the class generated by the mock generator by calling `$this->mockGenerator->getMockedClassCode()`, in order to debug, for example. This method takes the same arguments as the method `generate`.

```
<?php
$countableMock = new \mock\Countable;

// is equivalent to:

$this->mockGenerator->generate('\Countable');   // useless
$countableMock = new \mock\Countable;
```

Note: All what's described here with the mock generator can be apply with *newMockInstance*

Shunt calls to parent methods

shuntParentClassCalls & unShuntParentClassCalls

A mock inherits from the class from which it was generated, its methods therefore behave exactly the same way.

In some cases, it may be useful to shunt calls to parent methods so that their code is not run. The `mockGenerator` offers several methods to achieve this :

```
<?php
// The mock will not call the parent class
$this->mockGenerator->shuntParentClassCalls();

$mock = new \mock\OneClass;

// the mock will again call the parent class
$this->mockGenerator->unshuntParentClassCalls();
```

Here, all mock methods will behave as if they had no implementation however they will keep the signature of the original methods.

Note: `shuntParentClassCalls` will *only* be applied to the next generated mock. *But* if you create two mock of the same class, both will have they parent method shunted.

shunt

You can also specify the methods you want to shunt:

```
<?php
// the mock will not call the parent class for the method firstMethod......
$this->mockGenerator->shunt('firstMethod');
// ... nor for the method secondMethod
$this->mockGenerator->shunt('secondMethod');
```

```
$countableMock = new \mock\OneClass;
```

A shunted method, will have empty method body but like for `shuntParentClassCalls` the signature of the method will be the same as the mocked method.

Make an orphan method

It may be interesting to make an orphan method, that is, give him a signature and implementation empty. This can be particularly useful for generating mocks without having to instantiate all their dependencies. All the parameters of the method will also set as default value null. So it's the same a *shunted method*, but with all parameter as 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();

// We can instantiate the mock without injecting dependencies
$mock = new \mock\SecondClass();

$object = new FirstClass($mock);
```

Note: `orphanize` will *only* be applied to the next generated mock.

Modify the behaviour of a mock

Once the mock is created and instantiated, it is often useful to be able to change the behaviour of its methods. To do this, you must use its controller using one of the following methods:

- $yourMock->getMockController()->yourMethod
- $this->calling($yourMock)->yourMethod

```
<?php
$mockDbClient = new \mock\Database\Client();

$mockDbClient->getMockController()->connect = function() {};
```

```
// Equivalent to
$this->calling($mockDbClient)->connect = function() {};
```

The `mockController` allows you to redefine **only public and abstract protected methods** and puts at your disposal several methods:

```
<?php
$mockDbClient = new \mock\Database\Client();

// Redefine the method connect: it will always return true
$this->calling($mockDbClient)->connect = true;

// Redefine the method select: it will execute the given anonymous function
$this->calling($mockDbClient)->select = function() {
    return array();
};

// redefine the method query with arguments
$result = array();
$this->calling($mockDbClient)->query = function(Query $query) use($result) {
    switch($query->type) {
        case Query::SELECT:
            return $result;

        default;
            return null;
    }
};

// the method connects will throw an exception
$this->calling($mockDbClient)->connect->throw = new \Database\Client\Exception();
```

Note: The syntax uses anonymous functions (also called closures) introduced in PHP 5.3. Refer to PHP manual for more information on the subject.

As you can see, it is possible to use several methods to get the desired behaviour:

- Use a static value that will be returned by the method
- Use a short implementation thanks to anonymous functions of PHP
- Use the `throw` keyword to throw an exception

Change mock behaviour on multiple calls

You can also specify multiple values based on the order of call:

```
<?php
// default
$this->calling($mockDbClient)->count = rand(0, 10);
// equivalent to
$this->calling($mockDbClient)->count[0] = rand(0, 10);

// 1st call
$this->calling($mockDbClient)->count[1] = 13;
```

```
// 3rd call
$this->calling($mockDbClient)->count[3] = 42;
```

- The first call will return 13.
- The second will be the default behaviour, it means a random number.
- The third call will return 42.
- All subsequent calls will have the default behaviour, i.e. random numbers.

If you want several methods of the mock have the same behaviour, you can use the *methods<mock_methods>* or *methodsMatching<mock_method_matching>*.

methods

`methods` allow you, thanks to the anonymous function passed as an argument, to define what methods the behaviour must be modified:

```
<?php
// if the method has such and such name,
// we redefine its behaviour
$this
    ->calling($mock)
        ->methods(
            function($method) {
                return in_array(
                    $method,
                    array(
                        'getOneThing',
                        'getAnOtherThing'
                    )
                );
            }
        )
            ->return = uniqid()
;

// we redefines the behaviour of all methods
$this
    ->calling($mock)
        ->methods()
            ->return = null
;

// if the method begins by "get",
// we redefine its behaviour
$this
    ->calling($mock)
        ->methods(
            function($method) {
                return substr($method, 0, 3) == 'get';
            }
        )
            ->return = uniqid()
;
```

In the last example, you should instead use *methodsMatching<mock_method_matching>*.

Note: The syntax uses anonymous functions (also called closures) introduced in PHP 5.3. Refer to PHP manual for more information on the subject.

methodsMatching

`methodsMatching` allows you to set the methods where the behaviour must be modified using the regular expression passed as an argument :

```
<?php
// if the method begins by "is",
// we redefines its behaviour
$this
    ->calling($mock)
        ->methodsMatching('/^is/')
            ->return = true
;

// if the method starts by "get" (case insensitive),
// we redefines its behaviour
$this
    ->calling($mock)
        ->methodsMatching('/^get/i')
            ->throw = new \exception
;
```

Note: `methodsMatching` use preg_match and regular expressions. Refer to the PHP manual for more information on the subject.

isFluent && returnThis

Defines a fluent method, so the method return the class.

```
<?php
        $foo = new \mock\foo();
        $this->calling($foo)->bar = $foo;

        // is the same as
        $this->calling($foo)->bar->isFluent;
        // or this other one
        $this->calling($foo)->bar->returnThis;
```

doesNothing && doesSomething

Change the behaviour of the mock with `doesNothing`, the method will simply return null.

```
<?php
        class foo {
                public function bar() {
                        return 'baz';
                }
```

```
        }

        //
        // in your test
        $foo = new \mock\foo();
        $this->calling($foo)->bar = null;

        // is the same as
        $this->calling($foo)->bar->doesNothing;
        $this->variable($foo->bar())->isNull;

        // restore the behaviour
        $this->calling($foo)->bar->doesSomething;
        $this->string($foo->bar())->isEqualTo('baz');
```

Like you see in this example, if for any reason, you want to restore the behaviour of the method, use `doesSomething`.

Particular case of the constructor

To mock class constructor, you need:

- create an instance of \atoum\mock\controller class before you call the constructor of the mock ;
- set via this control the behaviour of the constructor of the mock using an anonymous function ;
- inject the controller during the instantiation of the mock in the *last* argument.

```
<?php
$controller = new \atoum\mock\controller();
$controller->__construct = function($args)
{
     // do something with the args
};

$mockDbClient = new \mock\Database\Client(DB_HOST, DB_USER, DB_PASS, $controller);
```

For simple case you can use *orphanize('__constructor')* or *shunt('__constructor')*.

Test mock

atoum lets you verify that a mock was used properly.

```
<?php
$mockDbClient = new \mock\Database\Client();
$mockDbClient->getMockController()->connect = function() {};
$mockDbClient->getMockController()->query   = array();

$bankAccount = new \Vendor\Project\Bank\Account();
$this
    // use of the mock via another object
    ->array($bankAccount->getOperations($mockDbClient))
        ->isEmpty()

    // test of the mock
```

```
    ->mock($mockDbClient)
        ->call('query')
            ->once() // check that the query method
                            // has been called only once
;
```

Note: Refer to the documentation on the *mock* for more information on testing mocks.

The mocking (mock) of native PHP functions

atoum allows to easily simulate the behaviours of native PHP functions.

```
<?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: The \ is not allowed before any functions to simulate because atoum take the resolution mechanism of PHP's namespace.

Important: For the same reason, if a native function was already called before, his mocking will be without any effect.

```
<?php

$this
   ->given($this->newTestedInstance())
   ->exception(function() { $this->testedInstance->loadConfigFile(); }) // the␣
↪function file_exists and is called before is mocking

   ->if($this->function->file_exists = true ) // the mocking can take the place of␣
↪the native function file_exists
   ->object($this->testedInstance->loadConfigFile())
      ->isTestedInstance()
;
```

Note: Check the detail about *isTestedInstance()*.

The mocking of constant

PHP constant can be declared with `defined`, but with atoum you can mock it like this:

```
<?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)
;
```

Warning, due to the nature of constant in PHP, following the *engine* you can meet some issue. Here is an example:

```
<?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())
                ->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)
            ;
        }
    }
}
```

CHAPTER 8

Execution engine

Several execution engines to run the tests (at the class or method level) are available. These are configurable via the `@engine` annotation. By default, the different tests run in parallel in PHP sub-processes, this is the `concurrent` mode.

There are currently three execution modes: :

- *inline*: tests run in the same process, this is the same behaviour as PHPUnit. Although this mode is very fast, there's no insulation of the tests.
- *isolate*: tests run sequentially in a subprocess of PHP. This form of execution is quite slow.
- *concurrent*: the default mode, the tests run in parallel, in PHP sub-processes.

Important: If you use xdebug to debug your tests (and not only for code coverage), the only execution engine available is *concurrent*.

Here's an example :

```php
<?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');
```

```
        }

        public function testBarWithoutBaz()
        {
                sleep(1);
                $this->newTestedInstance;
                $this->string($this->testedInstance->bar())
                        ->isIdenticalTo('foo');
        }
}
```

In `concurrent` mode :

```
=> 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.
```

In `inline` mode :

```
=> 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.
```

In `isolate` mode :

```
=> 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.
```

CHAPTER 9

Loop mode

When a developer is doing TDD (Test-Driven Development), it usually works as follows:

1. Start writing a test corresponding to what they want to develop,
2. run the test created,
3. write the code to pass the test,
4. then amend or complete the test and go back to step 2.

In practice, this means that they must:

- Create the code in their favourite editor,
- exit the editor and run the test in a console,
- return to their editor to write the code that enables the test to pass,
- return to the console to restart the test execution,
- return to their editor in order to amend or supplement its test

There is therefore a cycle that will be repeated until the functionality is complete.

During this cycle, the developer must repeatedly run the same terminal command to run the unit tests.

atoum offers the `loop` mode via the arguments `-l` or `--loop`, which lets the developer avoid restarting the test manually and streamline the development process.

In this mode, atoum runs the required tests.

Once the tests are complete, if the tests passed, atoum simply waits:

```
$ 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.
> 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...
```

If the developer presses the `Enter` key, atoum will reexecute the same test, without any other action from the developer.

In the case where the code doesn't pass the tests successfully, i.e. if assertions fails or if there were errors or exceptions, atoum also starts waiting :

```
$ 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...
```

If the developer presses the `Enter` key, instead of replaying the same tests again, atoum will only execute the tests that have failed, rather than replaying them all.

The developer can pops issues and replay error tests as many times as necessary simply by pressing `Enter`.

Once all failed tests pass successfully, atoum will automatically run all test suites to detect any potential regressions introduced by the corrections made by the developer.

```
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.
=> 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...
```

Of course, the `loop` mode will take only *the files with unit tests launch* by atoum.

CHAPTER 10

Debugging test cases

Sometimes tests fail and it's hard to find why.

In this case, one of the techniques available to solve the problem is to trace the behaviour of the concerned code, or directly inside the tested class using a debugger or a functions like `var_dump()` or `print_r()`, or directly inside the unit test of the class.

atoum provides some tools to help you in this process, debugging directly in unit tests.

Those tools are only available when you run atoum and enable the debug mode using the‘‘–debug‘‘ command line argument, this is to avoid unexpected debug output when running in standard mode. When the developer enables the debug mode (`--debug`), three methods can be used:

- `dump()` to dump the content of a variable ;
- `stop()` to stop a running test ;
- `executeOnFailure()` to set a closure to be executed when an assertion fails.

Those three method are accessible through the atoum fluent interface.

dump

The `dump()` method can be used as follows:

```
<?php
$this
    ->if($foo = new foo())
    ->then
        ->object($foo->setBar($bar = new bar()))
            ->isIdenticalTo($foo)
        ->dump($foo->getBar())
;
```

When the test is running, the return of the method `foo::getBar()` will be displayed through the standard output.

It's also possible to pass several arguments to `dump()`, as the following way :

```
<?php
$this
    ->if($foo = new foo())
    ->then
        ->object($foo->setBar($bar = new bar()))
            ->isIdenticalTo($foo)
        ->dump($foo->getBar(), $bar)
;
```

Important: The `dump` method is enabled only if you launch the tests with the `--debug` argument. Otherwise, this method will be totally ignored.

stop

The `stop()` method is also easy to use:

```
<?php
$this
    ->if($foo = new foo())
    ->then
        ->object($foo->setBar($bar = new bar()))
            ->isIdenticalTo($foo)
        ->stop() // the test will stop here if --debug is used
        ->object($foo->getBar())
            ->isIdenticalTo($bar)
;
```

If `--debug` is used, the last two lines will not be executed.

Important: The `stop` method is enabled only if you launch the tests with the `--debug` argument. Otherwise, this method will be totally ignored.

executeOnFailure

The method `executeOnFailure()` is very powerful and also simple to use.

Indeed it takes a closure in argument that will be executed if one of the assertions inside the test doesn't pass. It can be used as follows:

```
<?php
$this
    ->if($foo = new foo())
    ->executeOnFailure(
        function() use ($foo) {
            var_dump($foo);
        }
    )
    ->then
```

```
        ->object($foo->setBar($bar = new bar()))
            ->isIdenticalTo($foo)
        ->object($foo->getBar())
            ->isIdenticalTo($bar)
;
```

In the previous example, unlike `dump()` that systematically causing the display to standard output of the contents of the variables that are passed as argument, the anonymous function passed as an argument will cause the `foo` variable content if one of the assertions fails.

Of course, it's possible to call several times `executeOnFailure()` in the same test method to defined several closure to be executed if the test fails.

Important: The method `executeOnFailure` is enabled only if you run the tests with the argument `--debug`. Otherwise, this method will be totally ignored.

CHAPTER 11

Fine tuning atoum behaviour

The initialization methods

Here is the process, when atoum executes the test methods of a class with the default engine (`concurrent`) :

1. call of the `setUp()` method from the tested class ;
2. launch of a PHP sub-process for **each** test method ;
3. in the PHP sub-process, call of the `beforeTestMethod()` method of the test class ;
4. in the PHP sub-process, call of the test method ;
5. in the PHP sub-process, call of the `afterTestMethod()` method of the test class ;
6. once the PHP sub-process finished, call of the `tearDown()` method from the test class.

Note: For more information on the execution engine of test in atoum, you can read the section about the annotation *@engine*.

The methods `setUp()` and `tearDown()` allow respectively to initialize and clean up the test environment for all the test method of the running class.

The methods `beforeTestMethod()` and `afterTestMethod()` allows respectively to initialize and clean up the execution environment of the individual tests for all test method of the class. In contrast of `setUp()` and `tearDown()`, they are executed in the same subprocess.

It's also the reason why the methods `beforeTestMethod()` and `afterTestMethod()` accept as argument the name of the test method executed, in order to adjust the treatment accordingly.

```
<?php
namespace vendor\project\tests\units;

use
    mageekguy\atoum,
    vendor\project
```

```
;

require __DIR__ . '/atoum.phar';

class bankAccount extends atoum
{
    public function setUp()
    {
        // Executed *before all* test methods.
        // global initialization.
    }

    public function beforeTestMethod($method)
    {
        // Executed *before each* test method.

        switch ($method)
        {
            case 'testGetOwner':
                // Initialization for testGetOwner().
            break;

            case 'testGetOperations':
                // Initialization for testGetOperations().
            break;
        }
    }

    public function testGetOwner()
    {
        // ...
    }

    public function testGetOperations()
    {
        // ...
    }

    public function afterTestMethod($method)
    {
        // Executed *after each* test method.

        switch ($method)
        {
            case 'testGetOwner':
                // Cleaning for testGetOwner().
            break;

            case 'testGetOperations':
                // Cleaning for testGetOperations().
            break;
        }
    }

    public function tearDown()
    {
        // Executed after all the test methods.
        // Overall cleaning.
```

```
    }
}
```

By default, the `setUp()`, `beforeTestMethod()`, `afterTestMethod()` and `tearDown()` methods does absolutely nothing.

It is therefore the responsibility of the developer to overload when needed in the test classes concerned.

CHAPTER 12

Configuration & bootstraping

When atoum start, several steps will be involved, some of them can be influenced by some special files.

We can have a simplified view of theses special files with:

1. Load the *autoloader file*
2. Load the *configuration file*
3. Load the *bootstrap file*

Note: All of theses files are optional!

Note: You can use atoum `--init` to generate theses files.

Autoloader file

The autoloader file is the way you will define how the class to test will be found by atoum.

The default name of the file is `.autoloader.atoum.php`, atoum will load it automatically if this file is located in the current directory. You can define it through the cli with `--autoloader-file` or `-af` (*see the cli options*).

If the autoloader file doesn't exist, atoum will try to load `vendor/autoload.php` from composer. So most of the time, you will have nothing to do ;).

Configuration file

The configuration file is the way you can configure how atoum works.

The default name of the file is `.atoum.php`, atoum will load it automatically if this file is located in the current directory. You can define it through the cli with `-c`.

If you have in one of the parent directory a `.atoum.php` it will also be loaded. So you can have a default configuration to have the loop or debug mode activated by default.

Code coverage

By default, if PHP has the extension Xdebug, atoum indicates in command line report, the rate of tests code coverage. Some change in the behaviour of coverage can be adapt through the *cli options*.

If the coverage rate is 100%, atoum merely indicated. But otherwise, it displays the overall coverage and that of each method of the class tested in the form of a percentage.

```
$ 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%
=> 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) !
```

However, it is possible to get a more accurate representation of the rate of code coverage by tests, in HTML report.

To get it, simply rely on models of configuration files included in atoum.

Phar users

If you use the PHAR archive, it must retrieve them by using the following command:

```
php atoum.phar -er /path/to/destination/directory
```

Once the extraction is done, you should have in the "directory/path/to/destination/directory" a directory called "resources/configurations/runner".

Composer users

If you are using atoum with a github repository clone *Github* or with composer *Composer*, the models can be found in `/path/to/atoum/resources/configurations/runner`

Custom coverage reports

In this directory, there is, among other interesting things, a template of configuration file for atoum named `coverage.php.dist` that you need to copy to the location of your choice. Rename the `coverage.php`.

After copying the file, just have to change it with the editor of your choice to define the directory where the HTML files will be generated and the URL from which the report should be accessible.

For example:

```
$coverageField = new atoum\report\fields\runner\coverage\html(
    'Code coverage of my project',
    '/path/to/destination/directory'
);

$coverageField->setRootUrl('http://url/of/web/site');
```

Note: It is also possible to change the title of the report using the first argument to the constructor of the class `mageekguy\atoum\report\fields\runner\coverage\html`.

Once this is done, you just have to use the configuration file (or include it in your configuration file) when running the tests, as follows:

```
$ ./bin/atoum -c path/to/coverage.php -d tests/units
```

Once the tests run, atoum generate the code coverage report in HTML format in the directory that you set earlier, and it will be readable using the browser of your choice.

Note: The calculation of code coverage by tests as well as the generation of the corresponding report may slow significantly the performance of the tests. Then it can be interesting, not to systematically use the corresponding configuration file, or disable them temporarily using the -ncc argument.

Using standard reports

atoum come with a lot of standard reports: tap, xunit, html, cli, phing, vim, ... There is also some *fun reports* too. You will find the most important of them here.

Note: If you want to go further, there is an *extension* dedicated to the reports called `reports-extension`.

Report configuration

Branch and path coverage

You can enable the coverage of branch and path inside the configuration with `enableBranchAndPathCoverage`. This will improve the value of the code coverage by not only checking the method in the code called, but also that each branch is called. To make it simple, if you have an `if` the coverage report will change if you check the else.

```
$script->enableBranchAndPathCoverage();
```

```
=> Class Foo\Bar: Line: 31.46%
# with branch and path coverage
=> Class Foo\Bar: Line: 31.46% Path: 1.50% Branch: 26.06%
```

Disabling coverage for a class

If you want to exclude some class from coverage, you can use `$script->noCodeCoverageForClasses(\myClass::class)`

HTML report

By default atoum provide a basic html report. For advanced html report, you should use the reports-extension.

```
<?php
$report = $script->addDefaultReport();
$coverageField = new atoum\report\fields\runner\coverage\html('Your Project Name', __
↪DIR__ . '/reports');
// Please replace in next line http://url/of/web/site by the root url of your code␣
↪coverage web site.
$coverageField->setRootUrl('http://url/of/web/site');
$report->addField($coverageField);
```

CLI report

The CLI report is the report you have when you launch the test. there is several options available

- hideClassesCoverageDetails: Will disable the coverage of the class.
- hideMethodsCoverageDetails: Will disable the coverage of the methods.

```
<?php
$script->addDefaultReport() // in default reports there is the cli report
    ->hideClassesCoverageDetails()
    ->hideMethodsCoverageDetails();
```

Displaying the logo of atoum

```
<?php
$report = $script->addDefaultReport();

// This will add the atoum logo before each run.
$report->addField(new atoum\report\fields\runner\atoum\logo());

// This will add a green or red logo after each run depending on its status.
$report->addField(new atoum\report\fields\runner\result\logo());
```

Treemap report

```
<?php
$report = $script->addDefaultReport();
```

```
$coverageHtmlField = new atoum\report\fields\runner\coverage\html('Your Project Name',
↪ __DIR__ . '/reports');
// Please replace in next line http://url/of/web/site by the root url of your code␣
↪coverage web site.
$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);
```

Notifications

atoum is able to warn you when the tests are run using several notification system: *Growl*, *Mac OS X Notification Center*, *Libnotify*.

Growl

This feature requires the presence of the executable `growlnotify`. To check if it is available, use the following command:

```
$ which growlnotify
```

You will have the path to the executable or the message `growlnotify not found` if it is not installed.

Then just add the following code to your configuration file:

```
<?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')
;

$report = $script->AddDefaultReport();
$report->addField($notifier, array(atoum\runner::runStop));
```

Mac OS X Notification Center

This feature uses the `terminal-notifier` utility. To check if it is available, use the following command:

```
$ which terminal-notifier
```

You will have the path to the executable or the message `terminal-notifier not found` if it is not installed.

Note: Visit the project's Github page to get more information on `terminal-notifier`.

Then just add the following code to your configuration file:

```
<?php
$notifier = new \mageekguy\atoum\report\fields\runner\result\notifier\terminal();

$report = $script->AddDefaultReport();
$report->addField($notifier, array(atoum\runner::runStop));
```

On OS X, you can define a command to be executed when the user clicks on the 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))
;
```

The example above shows how to automatically open the code coverage report when the user clicks on the notification.

Libnotify

This feature requires the presence of the executable `notify-send`. To check if it is available, use the following command:

```
$ which notify-send
```

You will have the path to the executable or the message `notify-send not found` if it is not installed.

Then just add the following code to your configuration file:

```
<?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));
```

Configuration of the test

A lot of possibility to configure how atoum will find and execute the test is available. You can use the arguments in the cli or the configuration file. Because, a simple code will explain a lot more than a long text, just read this:

```php
<?php
$testGenerator = new atoum\test\generator();

// your unit test's directory. (-d)
$testGenerator->setTestClassesDirectory(__DIR__ . '/test/units');

// your unit test's namespace.
$testGenerator->setTestClassNamespace('your\project\namespace\tests\units');

// your unit test's runner.
$testGenerator->setRunnerPath('path/to/your/tests/units/runner.php');

$script->getRunner()->setTestGenerator($testGenerator);
```

You can also define the directory of your test with `$runner->addTestsFromDirectory(path)`. atoum will load all the class that can be tested from this directory like you can do with *-d* argument in cli.

```php
<?php
$runner->addTestsFromDirectory(__DIR__ . '/test/units');
```

Bootstrap file

atoum allows the definition of a `bootstrap` file, which will be run before each test method and which therefore allows to initialize the test execution environment.

The default name of the file is `.bootstrap.atoum.php`, atoum will load it automatically if this file is located in the current directory. You can define it through the cli with `-bf` or `--bootstrap-file`.

This makes possible to define, for example, the reading of a configuration file or perform any other operation necessary for the proper execution of the tests.

The definition of this bootstrap file can be done in two different ways, either in command line, or via a configuration file.

```
$ ./bin/atoum -bf path/to/bootstrap/file
```

Note: A bootstrap file is not a *configuration file* file and therefore does not have the same opportunities.

In a configuration file, atoum is configurable via the `$runner` variable, which is not defined in a `bootstrap` file.

Moreover, they are not included at the same time, since the configurations file is included by atoum before the tests run but after tests launch, while the `bootstrap`, if it's define, is the first file included by atoum itself. Finally, the `bootstrap` file can allow to not have to systematically include the `scripts/runner.php` file or atoum PHAR archive in test classes. However, in this case, it will not be possible to directly execute a test file directly from the PHP executable in command line. To do this, simply include in the `bootstrap` the file `scripts/runner.php` or PHAR archive of atoum and systematically execute tests by command line via `scripts/runner.php` or PHAR archive.

Therefore, the `bootstrap` file must at least contain this:

```php
<?php

// if the PHAR archive is used:
require_once path/to/atoum.phar;
```

```
// or if sources is used and you want to load the $runner
// require_once path/atoum/scripts/runner.php
```

Having fun with atoum

Report

Tests reports can be decorated to be more pleasant or fun to read. To do this in the *configuration file* of atoum, add the following code

```
<?php
// Default configuration file is .atoum.php
// ...

$stdout = new \mageekguy\atoum\writers\std\out;
$report = new \mageekguy\atoum\reports\realtime\nyancat;
$script->addReport($report->addWriter($stdout));
```

You should also try `\mageekguy\atoum\reports\realtime\santa` reports ;)

CHAPTER 13

Annotations

In this section we simply list all annotation that you can use with atoum.

Class's annotation

- *@dataProvider*
- *@extensions*
- @hasNotVoidMethods
- @hasVoidMethods
- @ignore
- @maxChildrenNumber
- @methodPrefix
- *@namespace*
- *@php*
- *@tags*

Method's annotation

- *@dataProvider*
- *@engine*
- *@extensions*
- @ignore
- @isNotVoid

- @isVoid
- *@php*
- *@tags*

Data providers

To help you to effectively test your classes, atoum provide you some data providers.

A data provider is a method in class test which generate arguments for test method, arguments that will be used by the method to validate assertions.

If a test method `testFoo` takes arguments and no annotation on a data provider is set, atoum will automatically seek the protected `testFooDataProvider` method.

However, you can manually set the method name of the data provider through the `@dataProvider` annotation applied to the relevant test method, as follows :

```
<?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)
        ;
    }

    // ...
}
```

Obviously, do not forget to define, at the level of the test method, arguments that correspond to those that will be returned by the data provider. If not, atoum will generate an error when running the tests.

The data provider method is a single protected method that returns an array or an iterator containing data :

```
<?php
class calculator extends atoum
{
    // ...

    // Provides data for testSum().
    protected function sumDataProvider()
    {
        return array(
            array( 1, 1),
            array( 1, 2),
            array(-1, 1),
            array(-1, 2),
        );
    }
}
```

When running the tests, atoum will call test method `testSum()` with the arguments `(1, 1)`, `(1, 2)`, `(-1, 1)` and `(-1, 2)` returned by the method `sumDataProvider()`.

> **Warning:** The insulation of the tests will not be used in this context, which means that each successive call to the method `testSum()` will be realized in the same PHP process.

Data provider as closure

You can also use a closure to define a data provider instead of an annotation. In your method *beforeTestMethod*, you can use this example:

```
<?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),
           );
         });
      }
   }
}
```

Data provider injected in test method

There is also, an injection of mock in the test method parameters. So take a simple example:

```
<?php
class cachingIterator extends atoum
{
    public function test__construct()
    {
        $this
            ->given($iterator = new \mock\iterator())
            ->then
                ->object($this->newTestedInstance($iterator))
        ;
    }
}
```

You can write this instead:

```
<?php

class cachingIterator extends atoum
{
```

```
    public function test__construct(\iterator $iterator)
    {
        $this
            ->object($this->newTestedInstance($iterator))
        ;
    }
}
```

In this case, no need for data provider. But, if you want to customize the mock you will require it or use *beforeTestMethod*.

```
<?php

class cachingIterator extends atoum
{
    public function test__construct(\iterator $iterator)
    {
        $this
            ->object($this->newTestedInstance($iterator))
        ;
    }

    public function beforeTestMethod($method)
    {
        // orphanize the controller for the next mock generated, here $iterator
        $this->mockGenerator->orphanize('__construct');
    }
}
```

PHP Extensions

Some of your tests may require one or more PHP extension(s). Telling atoum that a test requires an extension is easily done through annotations and the tag `@extensions`. After the tag `@extensions`, just add one or more extension names, separated by a space.

```
<?php

namespace vendor\project\tests\units;

class foo extends \atoum
{
    /**
     * @extensions intl
     */
    public function testBar()
    {
        // ...
    }
}
```

The test will only be executed if the extension is present. If not the test will be skipped and this message will be displayed.

```
vendor\project\tests\units\foo::testBar(): PHP extension 'intl' is not loaded
```

Note: By default the tests will pass when a test is skipped. But you can use the *–fail-if-skipped-methods* command line option to make the test fail when an extension is not present.

PHP Version

Some of your tests may require a specific version of PHP to work (for example, the test may only work on PHP 7). Telling atoum that the test requires a version of PHP is done through annotations and the tag `@php`.

By default, without providing any operator, the tests will only be executed if the PHP version is greater or equal to the version in the tag:

```
class testedClassname extends atoum\test
{
   /**
    * @php 7.0
    */
   public function testMethod()
   {
      // test content
   }
}
```

With this example the test will only be executed if the PHP version is greater of equal to PHP 7.0. If not the test will be skipped and this message will be displayed:

```
vendor\project\tests\units\foo::testBar(): PHP version 5.5.9-1ubuntu4.5 is not >= to␣
↪7.0
```

Note: By default the tests will pass when a test is skipped. But you can use the *–fail-if-skipped-methods* command line option to make the test fail when an extension is not present.

Internally, atoum uses PHP's version_compare function to do the comparison.

You're not limited to the greater equal operator. You can pass any version_compare operator.

For example:

```
class testedClassname extends atoum\test
{
   /**
    * @php < 5.4
    */
   public function testMethod()
   {
      // test content
   }
}
```

Will skip the test if the PHP version is greater or equal to 5.4

```
vendor\project\tests\units\foo::testBar(): PHP version 5.5.9-1ubuntu4.5 is not < to 5.
↪4
```

You can also pass multiple conditions, with multiple `@php` annotations. For example:

```
class testedClassname extends atoum\test
{
   /**
    * @php >= 5.4
    * @php <= 7.0
    */
   public function testMethod()
   {
      // test content
   }
}
```

CHAPTER 14

Command line options

Most options exist in two forms, a short form of 1 to 6 characters and a longer more explicit form. The two different forms do exactly the same thing and can be used interchangeably.

Some options accept multiple values:

```
$ ./bin/atoum -f tests/units/MyFirstTest.php tests/units/MySecondTest.php
```

Note: If you use an option multiple times, only the last one will be used.

```
# Only test MySecondTest.php
$ ./bin/atoum -f MyFirstTest.php -f MySecondTest.php

# Only test MyThirdTest.php and MyFourthTest.php
$ ./bin/atoum -f MyFirstTest.php MySecondTest.php -f MyThirdTest.php MyFourthTest.php
```

Configuration & bootstrap

-af <file> / –autoloader-file <file>

This option lets you specify the path to the *autoloader file*.

```
$ ./bin/atoum -af /path/to/autloader.php
$ ./bin/atoum --autoloader-file /path/to/autoloader.php
```

-bf <file> / –bootstrap-file <file>

This option lets you specify the path to the *bootstrap file*.

```
$ ./bin/atoum -bf /path/to/bootstrap.php
$ ./bin/atoum --bootstrap-file /path/to/bootstrap.php
```

-c <file> / –configuration <file>

This option lets you specify the path to the *configuration file* used for running the tests.

```
$ ./bin/atoum -c config/atoum.php
$ ./bin/atoum --configuration tests/units/conf/coverage.php
```

Filtering

-d <directories> / –directories <directories>

This option lets you specify the tests directory(ies) to run.

```
$ ./bin/atoum -d tests/units/db/
$ ./bin/atoum --directories tests/units/db/ tests/units/entities/
```

-f <files> / –files <files>

This option lets you specify the test files to run.

```
$ ./bin/atoum -f tests/units/db/mysql.php
$ ./bin/atoum --files tests/units/db/mysql.php tests/units/db/pgsql.php
```

-g <pattern> / –glob <pattern>

This option lets you specify the test files to launch based on a pattern.

```
$ ./bin/atoum -g ???
$ ./bin/atoum --glob ???
```

-m <class::method> / –methods <class::methods>

This option lets you filter the classes and methods to launch.

```
# launch only the method testMyMethod of the class␣
↪vendor\\project\\test\\units\\myClass
$ ./bin/atoum -m vendor\\project\\test\\units\\myClass::testMyMethod
$ ./bin/atoum --methods vendor\\project\\test\\units\\myClass::testMyMethod

# launch all the test methods in class vendor\\project\\test\\units\\myClass
$ ./bin/atoum -m vendor\\project\\test\\units\\myClass::*
$ ./bin/atoum --methods vendor\\project\\test\\units\\myClass::*

# launch only methods named testMyMethod from all test classes
```

```
$ ./bin/atoum -m *::testMyMethod
$ ./bin/atoum --methods *::testMyMethod
```

Note: Refer to the section on filters by *A class or a method* for more information.

-ns <namespaces> / –namespaces <namespaces>

This option lets you filter the classes and methods tested, based on namespaces.

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

Note: Refer to the section on filters *By namespace* for more information.

-t <tags> / –tags <tags>

This option lets you filter the classes and methods to launch based on tags.

```
$ ./bin/atoum -t OneTag
$ ./bin/atoum --tags OneTag TwoTag
```

Note: Refer to the section on filters by *Tags* for more information.

–test-all

This option lets you run the tests in directories defined in the configuration file through `$script->addTestAllDirectory('path/to/directory')`.

```
$ ./bin/atoum --test-all
```

–test-it

This option lets you launch atoum own unit tests to check that it runs smoothly on your server.

```
$ ./bin/atoum --test-it
```

-tfe <extensions> / –test-file-extensions <extensions>

This option lets you specify the extensions of test files to run.

```
$ ./bin/atoum -tfe phpt
$ ./bin/atoum --test-file-extensions phpt php5t
```

Debug & loop

–debug

This option allows you to enable debug mode

```
$ ./bin/atoum --debug
```

Note: Refer to the section on the *Debugging test cases* for more information.

-l / –loop

This option allows you to activate the loop mode of atoum.

```
$ ./bin/atoum -l
$ ./bin/atoum --loop
```

Note: Refer to the section on the *Loop mode* for more information.

Coverage & reports

-drt <string> / –default-report-title <string>

This option lets you specify atoum reports default title.

```
$ ./bin/atoum -drt Title
$ ./bin/atoum --default-report-title "My Title"
```

Note: If the title contains spaces, you must surround it with quotes.

-ft / –force-terminal

This option lets you force the output to the terminal.

```
$ ./bin/atoum -ft
$ ./bin/atoum --force-terminal
```

-sf <file> / –score-file <file>

This option lets you specify the path to the output file created by atoum.

```
$ ./bin/atoum -sf /path/to/atoum.score
$ ./bin/atoum --score-file /path/to/atoum.score
```

-ncc / –no-code-coverage

This option lets you disable the generation of the code coverage report.

```
$ ./bin/atoum -ncc
$ ./bin/atoum --no-code-coverage
```

-nccfc <classes> / –no-code-coverage-for-classes <classes>

This option lets you disable the generation of the code coverage report for one or more class.

```
$ ./bin/atoum -nccfc vendor\\project\\db\\mysql
$ ./bin/atoum --no-code-coverage-for-classes vendor\\project\\db\\mysql␣
↪vendor\\project\\db\\pgsql
```

Note: It's important to double each backslash to avoid them being interpreted by the shell.

-nccfns <namespaces> / –no-code-coverage-for-namespaces <namespaces>

This option lets you disable the generation of the code coverage report for one or more namespaces.

```
$ ./bin/atoum -nccfns vendor\\outside\\lib
$ ./bin/atoum --no-code-coverage-for-namespaces vendor\\outside\\lib1␣
↪vendor\\outside\\lib2
```

Note: It's important to double each backslash to avoid them being interpreted by the shell.

-nccid <directories> / –no-code-coverage-in-directories <directories>

This option lets you disable the generation of the code coverage report for one or more directories.

```
$ ./bin/atoum -nccid /path/to/exclude
$ ./bin/atoum --no-code-coverage-in-directories /path/to/exclude/1 /path/to/exclude/2
```

-ulr / –use-light-report

This option lets you reduce the output generated by atoum.

```
$ ./bin/atoum -ulr
$ ./bin/atoum --use-light-report

[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][  59/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 118/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 177/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 236/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 295/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 354/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 413/1141]
```

```
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 472/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 531/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 590/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 649/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 708/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 767/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 826/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 885/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][ 944/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][1003/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][1062/1141]
[SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS>][1121/1141]
[SSSSSSSSSSSSSSSSSS_____________________________________________________][1141/1141]
Success (154 tests, 1141/1141 methods, 0 void method, 0 skipped method, 16875␣
↪assertions) !
```

Failure & success

-fivm, –fail-if-void-methods

This option makes the test suite fail if there is at least one void test method.

```
$ ./bin/atoum -fivm
$ ./bin/atoum --fail-if-void-methods
```

-fism, –fail-if-skipped-methods

This option makes the test suite fail if there is at least one skipped test method

```
$ ./bin/atoum -fism
$ ./bin/atoum --fail-if-skipped-methods
```

Other arguments

-mcn <integer> / –max-children-number <integer>

This option lets you set the maximum number of processes launched to run the tests.

```
$ ./bin/atoum -mcn 5
$ ./bin/atoum --max-children-number 3
```

-p <file> / –php <file>

This option lets you specify the path to the php executable used to run your tests.

```
$ ./bin/atoum -p /usr/bin/php5
$ ./bin/atoum --php /usr/bin/php5
```

By default, the value is search amongst the following values (in order):

- PHP_BINARY constant
- PHP_PEAR_PHP_BIN environment variable
- PHPBIN environment variable
- constant PHP_BINDIR + '/php'

-h / –help

This option lets you display a list of available options.

```
$ ./bin/atoum -h
$ ./bin/atoum --help
```

-v / –version

This option lets you display the current version of atoum.

```
$ ./bin/atoum -v
$ ./bin/atoum --version

atoum version DEVELOPMENT by Frédéric Hardy (/path/to/atoum)
```

CHAPTER 15

Cookbook

Change the default namespace

At the execution beginning of a test class, atoum computes the name of the tested class. To do this, by default, it replaces in the class name the following regular expression `#(?:^|\\\)tests?\\\units?\\#i` by char `\`.

Thus, if the test class name is `vendor\project\tests\units\foo`, it will deduct in that the tested class named is `vendor\project\foo`. However, it may be necessary that the namespace of the test classes may not match this regular expression, and in this case, atoum then stops with the following error message:

```
> 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
--------------------------------------------------------------------------------
↪--------------------------------------------------------------------------------
↪----------------------------------
```

We must therefore change the regular expression we used, this is possible in several ways. The easiest way is to applied the annotations `@namespace` to the test class in the following way :

```
<?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()
   {
```

```
        /* ... */
    }
}
```

This method is quick and simple to implement, but it has the disadvantage of having to be repeated in each test class, which is not so maintainable if there is some change in their namespace. The alternative is to call the `atoum\test::setTestNamespace()` method in the constructor of the test class, in this way:

```
<?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()
    {
        /* ... */
    }
}
```

The `atoum\test:setTestNamespace()` method indeed accepts a single argument which must be the regular expression matches the namespace of your test class. And to not have to repeat the call to this method in each test class, just do it once and for all in an abstract class in the following manner:

```
<?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);
    }
}
```

Thus, you will only have to do derive your unit test classes from this abstract class:

```php
<?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()
   {
      /* ... */
   }
}
```

In case of unit tests namespace change, it is therefore necessary to change only the abstract class.

Moreover, it's not mandatory to use a regular expression, either at the level of the `@namespace` annotation or the method `atoum\test::setTestNamespace()` a simple string can also works.

Indeed atoum by default use a regular expression so that the user can use a wide range of namespaces without the need to configure it at this level. This therefore allows it to accept for example, without any special configuration the following namespaces:

- `test\unit\`
- `Test\Unit\`
- `tests\units\`
- `Tests\Units\`
- `TEST\UNIT\`

However, in general, the namespace used to test classes is fixed, and it's not necessary to use a regular expression if the default isn't suitable. In our case, it could be replaced with the string `my\tests`, for example through the `@namespace` annotation :

```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()
   {
      /* ... */
   }
}
```

Test of a singleton

To test a method that always returns the same instance of an object, checks that two calls to the tested method are the same.

```
<?php
$this
    ->object(\Singleton::getInstance())
        ->isInstanceOf('Singleton')
        ->isIdenticalTo(\Singleton::getInstance())
;
```

Hook git

A good practice, when using a version control system, is to never add a non-functional code in repository, in order to retrieve a version clean and usable code at any time and any place the history of the deposit.

This implies, among other things, that the unit tests must pass in their entirety before the files created or modified are added to the repository, and as a result, the developer is supposed to run the unit tests before pushed its code to the repository.

However, in fact, it is very easy for the developer to omit this step and your repository may therefore contain, more or less imminent, code which does not respect the constraints imposed by unit tests.

Fortunately, version control software in general and in particular Git has a mechanism, known as the pre-commit hook name to automatically perform tasks when adding code in a repository.

The installation of a pre-commit hook is very simple and takes place in two stages.

Step 1: Creation of the script to run

When adding code to a repository, Git looks for the file `.git/hook/pre-commit` in the root of the repository and executes it if it exists and that it has the necessary rights.

To set up the hook, you must therefore create the `.git/hook/pre-commit` file and add the following code:

The code below assumes that your unit tests are in files with the extension `.php` and directories path contains `/Tests/Units`. If this is not the case, you will need to modify the script depending on your context.

Note: In the above example, the test files must include atoum for the hook works.

The tests are run very quickly with atoum, all unit tests can be run before each commit with a hook like this :

```
#!/bin/sh
./bin/atoum -d tests/
```

Step 2: Add execution rights

To be usable by Git, the file `.git/hook/pre-commit` must be executable by using the following command, executed in command line from the directory of your deposit:

```
$ chmod u+x `.git/hook/pre-commit`
```

From this moment on, unit tests contained in the directories with the path contains `/ Tests/Units` will be launched automatically when you perform the command `git commit`, if files with the extension `.php` have been changed.

And if unfortunately a test does not pass, the files will not be added to the repository. You must then make the necessary adjustments, use the command `git add` on modified files and use again `git commit`.

Use in behat

The *asserters* from atoum are very easy to use outside your traditional unit tests. Just import the class *mageekguy-atoumasserter* without forgetting to load the required classes (atoum provides an autoload class available in *classes/autoloader.php*). The following example illustrates this usage of asserter from atoumin your Behat *steps*.

Installation

Simply install atoum and Behat in your project via pear, git clone, zip... Here is an example with dependency manager *Composer* :

```
"require-dev": {
        "behat/behat": "2.4@stable",
        "atoum/atoum": "~2.5",
}
```

It is obviously mandatory to update your composer dependencies with the command :

```
$ php composer.phar update
```

Configuration

As mentioned in the introduction, just import the asserter classes from atoum and ensure that they are loaded. For Behat, configuration of asserters are done inside the class *FeatureContext.php* (located by default in your directory */root-of-project/features/bootstrap/*).

```
<?php

use Behat\Behat\Context\ClosuredContextInterface,
    Behat\Behat\Context\TranslatedContextInterface,
    Behat\Behat\Context\BehatContext,
    Behat\Behat\Exception\PendingException,
    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();
    }
}
```

Usage

After these 2 particular trivial steps, your *steps* can be extended with the atoum asserters :

```
<?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);
    }
}
```

Once again, this is only an example specific to Behat but it remains valid for all needs of using the asserters of atoum outside the initial context.

Use with continous integration tools (CI)

Use inside Jenkins (or Hudson)

It's very simple to the results of atoum to Jenkins (or Hudson) as xUnit results.

Step1: Add a xUnit report to the configuration of atoum

Like other coverage report, you can use specific *report* from the configuration.

If you don't have a configuration file

If you don't have a configuration file for atoum yet, we recommend that you extract the directory resource of atoum in that one of your choice by using the following command :

- If you are using the Phar archive of atoum :

```
$ php atoum.phar --extractRessourcesTo /tmp/atoum-src
$ cp /tmp/atoum-src/resources/configurations/runner/xunit.php.dist /my/project/atoum.
↪php
```

- If you are using the sources of atoum :

```
$ cp /path/to/atoum/resources/configurations/runner/xunit.php.dist /my/project/.atoum.
↪php
```

- You can also directly copy the files from the Github repository

There is one last step, edit this file to set the path to the xUnit report where atoum will generate it. This file is ready to use, with him, you will keep the default report and gain a xUnit report for each launch of tests.

If you already have a configuration file

If you already have a configuration file, simply add the following lines:

```
<?php

//...

/*
 * Xunit report
 */
$xunit = new atoum\reports\asynchronous\xunit();
$runner->addReport($xunit);

/*
 * Xunit writer
 */
$writer = new atoum\writers\file('/path/to/the/report/atoum.xunit.xml');
$xunit->addWriter($writer);
```

Step 2: Test the configuration

To test this configuration, simply run atoum specifying the configuration file you want to use :

```
$ ./bin/atoum -d /path/to/the/unit/tests -c /path/to/the/configuration.php
```

Note: If you named your configuration file `.atoum.php`, it will be load automatically. The `-c` parameter is optional in this case. To let atoum load automatically the `.atoum.php` file, you will need to run test from the folder where this file resides or one of his childs.

At the end of the tests, you will have the xUnit report inside the folder specified in the configuration.

Step 3: Launching tests via Jenkins (or Hudson)

There are several possibilities depending on how you build your project :

- If you use a script, simply add the previous command.
- If you use a utility tool like phing or ant, simply add an exec task like :

```
<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>
```

Notice the addition of `-p /path/to/php` that permit to atoum to know the path to the php binary to use to run the unit tests.

Step 4: Publish the report with Jenkins (or Hudson)

Simply enable the publication of report with JUnit or xUnit format of the plugin you are using, specifying the path to the file generated by atoum.

Use with Travis-CI

It's simple to use atoum with a tool like Travis-CI. Indeed, all the steps are described in the official documentation : * Create your .travis.yml in your project; * Add it the next two lines:

```
before_script: wget http://downloads.atoum.org/nightly/atoum.phar
script: php atoum.phar
```

Here is an example file *.travis.yml* where the unit tests in the *tests* folder will be run.

```
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/
```

Use with Phing

atoum test suite can be easily ran inside your phing configuration using the integrated *phing/AtoumTask.php* task. A valid build example can be found in the resources/phing/build.xml file.

You must register the custom task using the taskdef native phing task :

```
<taskdef name="atoum" classpath="vendor/atoum/atoum/resources/phing" classname=
↪"AtoumTask"/>
```

Then you can use it inside one of your buildfile target:

```
<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>
```

The paths given in these examples have been taken from a standard composer installation. All the possible parameters are defined below, you can change values or omit some to rely on defaults. There is three kind of parameters:

atoum configurations

- *bootstrap*: Bootstrap file to be included before executing each test method
  - default: `.bootstrap.atoum.php`
- *atoumpharpath*: If atoum is used as phar, path to the phar file
- *atoumautoloaderpath*: Autoloader file before executing each test method
  - default: `.autoloader.atoum.php`
- *phppath*: Path to `php` executable
- *maxchildren*: Maximum number of sub-processus which will be run simultaneously

Flags

- *codecoverage*: Enable code coverage (only possible if XDebug in installed)
  - default: `false`
- *showcodecoverage*: Display code coverage report
  - default: `true`
- *showduration*: Display test execution duration
  - default: `true`
- *showmemory*: Display consumend memory
  - default: `true`
- *showmissingcodecoverage*: Display missing code coverage
  - default: `true`
- *showprogress*: Display test execution progress bar
  - default: `true`
- *branchandpathcoverage*: Enable branch and path coverage
  - default: `false`
- telemetry: Enable telemetry report (*atoum/reports-extension* must be installed)
  - default: `false`

Reports

- *codecoveragexunitpath*: Path to xunit report file
- *codecoveragecloverpath*: Path to clover report file
- *Code Coverage Basic*
  - *codecoveragereportpath*: Path to HTML report
  - *codecoveragereporturl*: URL to HTML report
- *Code Coverage Tree Map*:
  - *codecoveragetreemappath*: Path to tree map
  - *codecoveragetreemapurl*: URL to tree map
- Code Coverage Advanced
  - *codecoveragereportextensionpath*: Path to HTML report
  - *codecodecoveragereportextensionurl*: URL to HTML report
- Telemetry
  - *telemetryprojectname*: Name of telemetry report to be sent

Use with frameworks

Use with ezPublish

Step 1: Installation of atoum in eZ Publish

The eZ Publish framework have already a directory dedicated to tests, logically named tests. It's in this directory that should be placed the *PHAR archive* of atoum. The unit test files using atoum will be placed in a subdirectory *tests/atoum* so they don't conflict with the existing.

Step 2: Creating the class of the base tests

A class based on atoum must extend the class `\mageekguy\atoum\test`. However, this one doesn't take into account of *eZ Publish* specificities. It's therefore mandatory to define a base test class, derived from `\mageekguy\atoum\test`, which will take into account these specificities and will derive all of the classes of unit tests. To do this, just defined the following class in the file `tests\atoum\test.php`:

```
<?php

namespace ezp;

use mageekguy\atoum;

require_once __DIR__ . '/atoum.phar';

// Autoloading : eZ
require 'autoload.php';

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
{
}

?>
```

Step 3: Creating a test class

By default, atoum asks that unit tests classes are in a namespace containing *test(s)unit(s)*, in order to deduce the name of the tested class. For example, the namespace *nameofprojet* will be used in the following. For simplicity, it's further advisable to model the test tree on the tested classes tree, in order to quickly locate the class of a tested class, and vice versa.

```
<?php

namespace nameofproject\tests\units;

require_once '../test.php';

use ezp;

class cache extends ezp\test
{
   public function testClass()
   {
           $this->assert->hasMethod('__construct');
   }
}
```

Step 4: Running the unit tests

Once a test class created, simply execute this command-line to start the test from the root of the project:

```
# php tests/atoum/atoum.phar -d tests/atoum/units
```

Thanks to Jérémy Poulain for this tutorial.

Use with Symfony 2

If you want to use atoum within your Symfony projects, you can install the Bundle AtoumBundle.

If you want to install and configure atoum manually, here's how to do it.

Step 1: installation of atoum

If you use Symfony 2.0, *download the PHAR* and place it in the vendor directory which is at the root of your project.

If you use Symfony 2.1+, *add atoum in your composer.json*.

Step 2: create the test class

Imagine that we wanted to test this 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: For more information about creating Entity in Symfony 2, refer to the official documentation

Create the directory Tests/Units in your Bundle (for example src/Acme/DemoBundle/Tests/Units). It's in this directory that will be stored all tests of this Bundle.

Create a Test.php file that will serve as a base for all new tests in this Bundle.

```
<?php
// src/Acme/DemoBundle/Tests/Units/Test.php
namespace Acme\DemoBundle\Tests\Units;
```

```
// It includes the class loader and active it
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;

// For Symfony 2.0 only !
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: The inclusion of atoum's PHAR archive is only necessary for Symfony 2.0. Remove this line if you use Symfony 2.1+.

Note: By default, atoum uses namespace tests/units for testing. However Symfony 2 and its class loader require capitalization at the beginning of the names. For this reason, we change tests namespace through the method: setTestNamespace('TestsUnits').

Step 3: write a test

In the Tests/Units directory, simply recreate the tree of the classes that you want to test (for example src/Acme/DemoBundle/Tests/Units/Entity/Car.php).

Create our test file:

```
<?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')
        ;
    }
}
```

Step 4: launch tests

If you use Symfony 2.0:

```
# Launch tests of one file
$ php vendor/atoum.phar -f src/Acme/DemoBundle/Tests/Units/Entity/Car.php

# Launch all tests of the Bundle
$ php vendor/atoum.phar -d src/Acme/DemoBundle/Tests/Units
```

If you use Symfony 2.1+:

```
# Launch tests of one file
$ ./bin/atoum -f src/Acme/DemoBundle/Tests/Units/Entity/Car.php

# Launch all tests of the Bundle
$ ./bin/atoum -d src/Acme/DemoBundle/Tests/Units
```

Note: You can get more information on the *test launch* in the chapter which is dedicated to.

In any case, this is what you should get:

```
> 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) !
```

Use with symfony 1.4

If you want to use atoum inside your Symfony 1.4 project, you can install the plugins sfAtoumPlugin. It's available on this address: https://github.com/atoum/sfAtoumPlugin.

Installation

There are several ways to install this plugin in your project:

- installation via composer
- installation via git submodules

Using composer

Add this lines inside the composer.json file:

```
"require"     : {
  "atoum/sfAtoumPlugin": "*"
},
```

After a `php composer.phar update` the plugin should be in the plugin folder and atoum in the `vendor` folder.

Then, in your ProjectConfiguration file, you have to activate the plugin and define the atoum path.

```
<?php
sfConfig::set('sf_atoum_path', dirname(__FILE__) . '/../vendor/atoum/atoum');

if (sfConfig::get('sf_environment') != 'prod')
{
  $this->enablePlugins('sfAtoumPlugin');
}
```

Using a git submodule

First, install atoum as a submodule:

```
$ git submodule add git://github.com/atoum/atoum.git lib/vendor/atoum
```

Then install sfAtoumPlugin as a git submodule:

```
$ git submodule add git://github.com/atoum/sfAtoumPlugin.git plugins/sfAtoumPlugin
```

Finally, enable the plugin in in your ProjectConfiguration file:

```
<?php
if (sfConfig::get('sf_environment') != 'prod')
{
  $this->enablePlugins('sfAtoumPlugin');
}
```

Write tests

Tests must include the bootstrap file from the plugin:

```
<?php
require_once __DIR__ . '/../../../../plugins/sfAtoumPlugin/bootstrap/unit.php';
```

Launch tests

The symfony command `atoum:test` is available. The tests can then be launched in this way:

```
$ ./symfony atoum:test
```

All the arguments of atoum are available.

It's therefore, for example, possible to give a configuration file like this :

```
php symfony atoum:test -c config/atoum/hudson.php
```

Symfony 1 plugin

To use atoum within a symfony project 1, a plug-in exists and is available at the following address: https://github.com/atoum/sfAtoumPlugin.

The instructions for installation and use are the cookbook *Use with symfony 1.4* as well as on the github page.

Symfony 2 bundle

To use atoum inside a Symfony 2 project, the bundle AtoumBundle is available.

The instructions for installation and use are the cookbook *Use with Symfony 2* as well as on the github page.

Zend Framework 2 component

If you want to use atoum within a Zend Framework 2 project, a component exists and is available at the following address.

The instructions for installation and usage are available on this page.

CHAPTER 16

Extensions

atoum has an extension mechanism for adding functionality that is not intended to be in the core of atoum. Here is the list of existing extensions and the link to the documentation of each.

- bdd-extension :helps you write tests using behavior-driven development,
- visibility-extension : lets you override the visibility of methods so that you can test protected and private methods,
- json-schema-extension : enables you to valididate a JSON return with a JSON-Schema,
- ruler-extension : add an extra parameter to the atoum CLI command to launch only some tests based on complex rules.
- blackfire-extension : write your Blackfire test suites with atoum.

CHAPTER 17

Integration of atoum in your IDE

Sublime Text 2

A plug-in for SublimeText 2 allows the execution of unit tests by atoum and the visualization of the results without leaving the editor.

The information necessary for its installation and its configuration are available on the blog's author.

VIM

atoum comes with a plug-in for VIM.

It lets you run tests and obtain a report in a separate window without leaving VIM.

It's possible to navigate through errors, or even to go to the line in the editor with an assertion from failed test with a simple combination of keystrokes.

Installation of the plug-in atoum for VIM

You will find the file corresponding to the plug-in, named `atoum.vmb`, in the directory named `resources/vim`.

If you are using the PHAR archive, you must extract the file with the following command:

```
$ php atoum.phar --extractResourcesTo path/to/a/directory
```

Once the extraction is performed, the file `atoum.vmb` corresponding to the plug-in for VIM will be in the directory `path/to/a/directory/resources/vim`.

Once you have the `atoum.vmb` file, you need to edit it with VIM:

```
$ vim path/to/atoum.vmb
```

Now install the plug-in by using the command in VIM:

```
:source %
```

Use of the atoum plug-in for VIM

To use the plug-in, atoum must be installed and you must be editing a file containing a class of unit tests based on atoum.

Once configured, the following command will launch the execution of tests:

```
:Atoum
```

The tests will be executed, and once finished, a report based on the configuration of atoum file located in the directory `ftplugin/php/atoum.vim` in your `.vim` directory is generated in a new window.

Of course, you are free to bind this command to any combination of keystrokes of your choice, for example adding the following line in your `.vimrc` file:

```
nnoremap *.php <F12> :Atoum<CR>
```

`F12` in normal mode will call the command `:Atoum`.

Configuration file management of atoum

You can specify another configuration file for atoum by adding the following line to your `.vimrc` file:

```
call atoum#defineConfiguration('/path/to/project/directory', '/path/to/atoum/
↪configuration/file', '.php')
```

Indeed the function `atoum#defineConfiguration` lets you configure the file to use, based on the directory where the unit test files are located. It take three arguments:

- a path to the directory containing the unit tests;
- a path to the configuration file of atoum to be used;
- the extension's file of relevant unit tests.

You can access help about the plug-in in VIM with the following command:

```
:help atoum
```

Coverage reports inside vim

You can configure a specific *report* for the coverage in vim. In your atoum configuration file, set:

... code-block:: php

    <?php use mageekguyatoum; $vimReport = new atoumreportsasynchronousvim(); $vimReport->addWriter($stdOutWriter); $runner->addReport($vimReport);

PhpStorm

atoum comes with an official plug-in for PHPStorm. It really helps you in your day-to-day development. The main functionality are:

- Go to the test class from the tested class (shortcut : alt+shift+K)
- Go to the tested class from the test class (shortcut : alt+shift+K)
- Execute tests inside PhpStorm (shortcut : alt+shift+M)
- Easily identify test files by a custom icon

Installation

It's easy to install, simply follow theses steps:

- Open PhpStorm
- Go to *File -> Settings*, then click on *Plugins*
- Click on Browse repositories
- Search for *atoum* in the list, then click on the install button
- Restart PhpStorm

If you need more information check the repository of the plugins.

Atom

atoum comes with an official package for atom. It helps you in several tasks :

- A panel with all tests
- Run all the tests, a directory or the current one

Installation

It's easy to install, simply follow the official documentation or theses steps:

- Open atom
- Go to *Settings*, then click on *Install*
- Search for *atoum* in the list, then click on the install button

If you need more information check the repository of the package.

Automatically open failed tests

atoum is able to automatically open files from failed tests at the end of there execution. Several editors are currently supported:

- *macvim* (Mac OS X)
- *gvim* (Unix)

- *PhpStorm* (Mac OS X/Unix)
- *gedit* (Unix)

To use this feature, you need to change the *configuration file*:

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);
```

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);
```

PhpStorm

If you are under Mac OS X, use the following configuration:

```
<?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
    // If PhpStorm is installed in /Applications
```

```
    ->addField(new macos\phpstorm())

    // If you have installed PhpStorm
    // in another directory than /Applications
    // ->addField(
    //     new macos\phpstorm(
    //         '/path/to/PhpStorm.app/Contents/MacOS/webide'
    //     )
    // )
;

$runner->addReport($cliReport);
```

Under Unix environment, use the following configuration:

```
<?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('/path/to/PhpStorm/bin/phpstorm.sh')
    )
;

$runner->addReport($cliReport);
```

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);
```

CHAPTER 18

Contribute

How to contribute

Important: We need help writing this section!

Coding convention

The source code of atoum follows some conventions. If you wish to contribute to this project, your code must follow the same rules:

- Indentation must be done with tabs,
- Namespaces, classes, members, methods, and constants follow the `lowerCamelCase` convention,
- Code must be tested.

The example below makes no sense but shows in more in detail the way in which the code is written:

```php
<?php

namespace mageekguy\atoum\coding;

use
    mageekguy\atoum,
    type\hinting
;

class standards
{
    const standardsConst = 'standardsConst';
    const secondStandardsConst = 'secondStandardsConst';
```

```
    public $public;
    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);
        };
```

```
    }
}
```

Also here is an example of an unit test:

```
<?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')
        ;
    }
}
```

CHAPTER 19

Licences

First there is the documentation of this documentation. It's under CC by-nc-sa 4.0 .

Then you have the licence of the main project. It's under the BSD-3-Clause .