atoum Documentation

Change this: Author Name

Table of contents

CHAPTER 1

What is atoum ?

atoum is a unit testing framework, like PHPUnit or SimpleTest.

atoum distinguished itself as : * It is modern and based on the last PHP versions * It is quite simple and straightforward, with a very limited learning curve * It is intuitive : its API wants to be as close as possible as a natural language

1.1 Download & Install

For now, atoum is not tagged with a version number. If you want to use atoum, just download the last stable version. atoum aims to provide backward compatibility anyway.

atoum is distributed as a PHAR archive, an archive format dedicated to PHP, available since PHP 5.3.

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

If you want to use atoum directly from it's sources, you can clone it's git repository on github : git://github.com/mageekguy/atoum.git

1.2 A quick overview of atoum's philosophy

1.2.1 Very basic example

atoum wants you to write a test class for each class you want to test. As an example, if you want to test the famous HelloTheWorld class, you'll have to write the test\units\HelloTheWorld test class.

NOTE : atoum is, of course, namespace aware. As an example, to test the Hello\The\World class, you'll write the \Hello\The\tests\units\World class.

Here is the code of your HelloTheWorld class that we'll be using as a first example. This class will be located in PROJECT_PATH/classes/HelloTheWorld.php

```
<?php
/**
 * The class to be tested
 */
class HelloTheWorld
{
    public function getHiBob ()
    {
```

```
        return "Hi Bob !";
    }
}
```

Now, let's write our first test class. This class will be located in PROJECT_PATH/tests/HelloTheWorld.php

```
<?php
//Your test classes are in a dedicated namespace
namespace tests\units;

//You have to include your tested class
require_once __DIR__.'/../classes/HelloTheWorld.php';

//You now include atoum, using it's phar archive
require_once __DIR__.'/atoum/mageekguy.atoum.phar';

use \mageekguy\atoum;

/**
 * Test class for \HelloTheWorld
 * Test classes extends from atoum\test
 */
class HelloTheWorld extends atoum\test
{
    public function testGetHiBob ()
    {
        //new instance of the tested class
        $helloToTest = new \HelloTheWorld();

        $this->assert
                    //we expect the getHiBob method to return a string
                    ->string($helloToTest->getHiBob())
                    //and the string should be Hi Bob !
                    ->isEqualTo('Hi Bob !');
    }
}
```

Now, let's launch the tests

```
php -f ./test/HelloTheWorld.php
```

You will see something like this

```
> atoum version nightly-941-201201011548 by Frédéric Hardy
(phar:///home/documentation/projects/tests/atoum/mageekguy.atoum.phar/1)
> PHP path: /usr/bin/php5
> PHP version:
=> PHP 5.3.6-13ubuntu3.3 with Suhosin-Patch (cli) (built: Dec 13 2011 18:37:10)
=> Copyright (c) 1997-2011 The PHP Group
=> Zend Engine v2.3.0, Copyright (c) 1998-2011 Zend Technologies
=>     with Xdebug v2.1.2, Copyright (c) 2002-2011, by Derick Rethans
> tests\units\HelloTheWorld...
[S_____________________________________________________________][1/1]
=> Test duration: 0.01 second.
=> Memory usage: 0.00 Mb.
> Total test duration: 0.01 second.
> Total test memory usage: 0.00 Mb.
> Code coverage value: 100.00%
> Running duration: 0.16 second.
Success (1 test, 1/1 method, 2 assertions, 0 error, 0 exception) !
```

You're done, your code is rock solid !

1.2.2 Rule of Thumb

The basics when you’re testing things using atoum are the following : * Tell atoum what you want to work on (a variable, an object, a string, an integer, ...) * Tell atoum the state the element is expected to be in (is equal to, is null, exists, ...).

In the above example we tested that the method getHiBob * did return a string (step 1), * and that this string was equal to « Hi Bob ! » (step 2).

1.2.3 More asserters

There are of course a lot more asserters in atoum. To see the complete list, see chapter 2.

Let's see with a quick class some more asserters in atoum !

First, the class to be tested, located in PROJECT_PATH/classes/BasicTypes.php.

```
<?php
class BasicTypes
{
    public function getOne (){return 1;}
    public function getTrue(){return true;}
    public function getFalse(){return false;}
    public function getHello(){return 'hello';}
    public function create(){return new BasicTypes();}
    public function getFloat(){return 1.1;}
    public function getNull(){return null;}
```

```
    public function getEmptyArray(){return array();}
    public function getArraySizeOf3(){return range(0,2,1);}
}
```

Now the test class, located in PROJECT_PATH/tests/BasicTypes.php

```
<?php //...
class BasicTypes extends atoum\test
{
    public function testBoolean ()
    {
        $bt = new \BasicTypes();
        $this->assert
                ->boolean($bt->getFalse())
                    ->isFalse()//getFalse retourne bien false
                ->boolean($bt->getTrue())
                    ->isTrue();//getTrue retourne bien true
    }

    public function testInteger ()
    {
        $bt = new \BasicTypes();
        $this->assert
                ->integer($bt->getOne())
                ->isEqualTo(1)
                ->isGreaterThan(0);
    }

    public function testString()
    {
        $bt = new \BasicTypes();
        $this->assert
                ->string($bt->getHello())
                ->isNotEmpty()
                ->isEqualTo('hello');
    }

    public function testObject ()
    {
        $bt = new \BasicTypes();
        $this->assert
                ->object($bt->create())
                ->isInstanceOf('BasicTypes')
                ->isNotIdenticalTo($bt);//Une nouvelle instance
    }
```

```
    public function testFloat()
    {
        $bt = new \BasicTypes();
        $this->assert
                ->float($bt->getFloat())
                ->isEqualTo(1.1);
    }

    public function testArray()
    {
        $bt = new \BasicTypes();
        $this->assert
                ->array($bt->getArraySizeOf3())
                    ->hasSize(3)
                    ->isNotEmpty()
                ->array($bt->getEmptyArray())
                    ->isEmpty();
    }

    public function testNull ()
    {
        $bt = new \BasicTypes();
        $this->assert
                ->variable($bt->getNull())
                ->isNull();
    }
}
```

Here you are, you saw a complet and basic example of tests using atoum.

1.2.4 Testing a Singleton

To test if your method always returns the same instance of the same object, you can ask atoum to check that the instances are identicals.

```
<?php //...
class Singleton extends atoum\test
{
    public function testGetInstance()
    {
        $this->assert
                ->object(\Singleton::getInstance())
                    ->isInstanceOf('Singleton')
                    ->isIdenticalTo(\Singleton::getInstance());
```

```
    }
}
```

1.2.5 Testing exceptions

To test exceptions atoum is using closures (introduced in PHP 5,3).

```
class ExceptionLauncher extends atoum\test
{
    public function testLaunchException ()
    {
        $exception = new \ExceptionLauncher();
        $this->assert
                ->exception(function()use($exception){
                                $exception->launchException();
                            })
                ->isInstanceOf('LaunchedException')
                ->hasMessage('Message in the exception');

    }
}
```

1.2.6 Testing errors

Again, atoum is nicely using closure to test errors (NOTICE, WARNING, ...) :

```
class RaiseError extends atoum\test
{
    public function testRaiseError ()
    {
        $error = new \RaiseError();

        $this->assert->object($error);
        $this->assert
                ->when(function()use($error){
                       $error->raise();
                })
                ->error('This is an error', E_USER_WARNING)
                   ->exists();
                //Sachant qu'il est possible de ne spécifier
                // ni message ni type attendu.
    }
}
```

1.2.7 Testing using Mocks

Mocks are of course supported by atoum ! Generating a Mock from an interface

atoum can generate a mock directly from an interface.

```
class UsingWriter extends atoum\test
{
    public function testWithMockedInterface ()
    {
        $this->mockGenerator->generate('\IWriter');
        $mockIWriter = new \mock\IWriter;

        $usingWriter = new \UsingWriter();
        //La méthode setIWriter attends un objet
        //qui implemente l'interface IWriter
        //  (setIWriter (IWriter $writer))
        $usingWriter->setIWriter($mockIWriter);

        $this->assert
                ->when(function () use($usingWriter) {
                                $usingWriter->write('hello');
                })
                ->mock($mockIWriter)
                    ->call('write')
                    ->once();
    }
}
```

1.2.8 Generating a Mock from a class

atoum can generate a mock directly from a class definition.

```
public function testWithMockedObject ()
{
    $this->mockGenerator->generate('\Writer');
    $mockWriter = new \mock\Writer;

    $usingWriter = new \UsingWriter();
    //La méthode setWriter attends un objet
    //de type Writer (setWriter (Writer $writer))
    $usingWriter->setWriter($mockWriter);

    $this->assert
            ->when(function () use($usingWriter) {
                            $usingWriter->write('hello');
            })
            ->mock($mockWriter)
                ->call('write')
```

```
                ->once();
}
```

There is also a shorter syntax to generate mock from a class definition.

```
public function testWithMockedObject ()
{
    $mockWriter = new \mock\Writer;

    //...
}
```

atoum is able to automatically find the class definition to mock on demand so you don't have to call the mock generator.

When requesting a mock instance for a class, do not forget to specify the full class path (including namespaces).

```
namespace Package\Writers
{
    class SampleWriter implements Writer
    {
        //...
    }

}

namespace
{
    class UsingWriter
    {
        public function write(\Package\Writers\Writer $writer, $string)
        {
            $writer->write($string);
        }
    }
}
```

In this example, the class we want to mock lives in the Package\Writers namespace, so to request a mock in our test we should do :

```
namespace Package\test\units;

class UsingWriter extends atoum\test
{
```

```
    public function testWrite()
    {
        $this
            ->if($mockWriter = new \mock\Package\Writers\SampleWriter())
            ->then()
                ->when(function() use($mockWriter) {
                    $usingWriter = new \UsingWriter();
                    $usingWriter->write($mockWriter, 'Hello World!');
                })
                ->mock($mockWriter)
                    ->call('write')
                    ->withArguments('Hello World!')
                    ->once()
        ;
    }
}
```

1.2.9 Generating a Mock from scratch

atoum can also let you create and completely specify a mock object.

```
$this->mockGenerator->generate('WriterFree');
$mockWriter = new \mock\WriterFree;
$mockWriter->getMockController()->write = function($text){};

$usingWriter = new \UsingWriter();
$usingWriter->setFreeWriter($mockWriter);

$this->assert
        ->when(function () use($usingWriter) {
                        $usingWriter->write('hello');
        })
        ->mock($mockWriter)
            ->call('write')
            ->once();
```

CHAPTER 2

Asserters

2.1 variable

This is the base asserter for variables, it includes the basics assertions you may need while testing values of any type.

2.1.1 isEqualTo

isEqualTo verify that the tested variable is equal to a given value. [php] $a = 'a';

```
$this->assert
        ->variable($a)
           ->isEqualTo('a');//Will pass

isEqualTo won't verify the type of the variables, only the value itself.

$a1 = '1';

$a2 = 1;


$this->assert
        ->variable($a1)
           ->isEqualTo($a2); //Will pass
```

Note

while testing values of different types with isEqualTo, variables are compared like the == operator in PHP.

If you to verify both types and values, use the isIdenticalTo assertion instead.

2.1.2 isNotEqualTo

isNotEqualTo verify that the tested variable is different from a given value.

```
$a = 'a';

$this->assert
        ->variable($a)
           ->isNotEqualTo('a');//Will fail

$this->assert
        ->variable($a)
```

```
            ->isNotEqualTo('b');//Will pass

As isEqualTo, isNotEqualTo won't check the type of the variables.

$a1 = '1';

$this->assert
        ->variable($a1)
           ->isNotEqualTo(1);//Will fail
```

2.1.3 isIdenticalTo

isIdenticalTo verify that the tested variables are the same (types and values). In case you are testing objects, isIdenticalTo will tell if both variables represents the same instance.

```
$a1 = '1';

$this->assert
        ->variable($a1)
           ->isIdenticalTo(1); //Will fail

$stdClass    = new StdClass();
$stdClass2   = new StdClass();
$stdClassRef = $stdClass;

$this->assert
        ->variable($stdClass)
           ->isIdenticalTo(stdClass2); //Will fail

$this->assert
        ->variable($stdClass)
           ->isIdenticalTo(stdClassRef); //Will pass
```

If you don't want to consider the types of the variable, use isEqualTo.

2.1.4 isNotIdenticalTo

isNotIdenticalTo verify that the tested variables are not the same (types or values). In case you are testing objects, isNotIdenticalTo will assert that the tested variables points to different instances.

```
$a1 = '1';

$this->assert
        ->variable($a1)
           ->isNotIdenticalTo(1); //Will Pass ($a1 is a string, does not match
an integer)
```

```
$stdClass    = new StdClass();
$stdClass2   = new StdClass();
$stdClassRef = $stdClass;

$this->assert
        ->variable($stdClass)
           ->isNotIdenticalTo(stdClass2); //Will fail, variables do not point
to the same instance, even if their values are the same

$this->assert
        ->variable($stdClass)
           ->isNotIdenticalTo(stdClassRef); //Will fail, both variables points
to the same instance of StdClass
```

If you don't want to consider the types of the variables, use isNotEqualTo.

2.1.5 isNull

isNull verify that the variable is null.

```
$a1 = '';
$this->assert
        ->variable($a1)
           ->isNull(); //Will Fail ($a1 is empty but not null)

$a2 = null;
$this->assert
        ->variable($a2)
           ->isNull(); //Will Pass
```

2.1.6 isNotNull

isNotNull verify taht the variable is not null.

```
$a1 = '';
$this->assert
        ->variable($a1)
           ->isNotNull(); //Will pass ($a1 is empty but not null)
```

2.1.7 isReferenceTo

2.2 integer

This is the asserter dedicated to integer testing. It extends the variable asserter : You can use every assertions in the variable asserter while testing integers.

If you try to test a variable that is not an integer with the integer asserter, it will raise a failure.

```
$a1 = '1';

$this->assert
        ->integer($a1) //Will fail, a1 is not an integer, event if it
represents one
```

Note

null is not considered as a valid integer. You can check PHP's is_int function to check what is considered an integer.

2.2.1 isZero

isZero verify that the tested variable is equal to 0.

```
$zero = 0;
$minusOne = -1;

$this->assert
        ->integer($zero)
           ->isZero(); // Will pass

$this->assert
        ->integer($minusOne)
           ->isZero(); // Will fail
```

2.2.2 isLessThan

isLessThan verify that the tested integer is strictly less than a given integer.

```
$zero = 0;

$this->assert
        ->integer($zero)
           ->isLessThan(10); // Will pass

$this->assert
        ->integer($zero)
           ->isLessThan('10'); // Will fail, you have to pass an actual
integer to isLessThan

$this->assert
        ->integer($zero)
           ->isLessThan(0); // Will fail, 0 is equal to 0
```

Note

values given to isLessThan must be actual integers.

2.2.3 isGreaterThan

isGreaterThan verify that the tested integer is strictly greater than a given integer.

```
$zero = 0;

$this->assert
        ->integer($zero)
           ->isGreaterThan(-1); // Will pass

$this->assert
        ->integer($zero)
           ->isGreaterThan('-1'); // Will fail, you have to pass an actual
integer to isGreaterThan

$this->assert
        ->integer($zero)
           ->isGreaterThan(0); // Will fail, 0 is equal to 0
```

Note

values given to isGreaterThan must be actual integers.

2.2.4 isLessThanOrEqualTo

isLessThanOrEqualTo verify that the tested integer is less or equal to a given integer.

```
$zero = 0;

$this->assert
        ->integer($zero)
           ->isLessThanOrEqualTo(10); // Will pass

$this->assert
        ->integer($zero)
           ->isLessThanOrEqualTo('10'); // Will fail, you have to pass an
actual integer to isLessThanOrEqualTo

$this->assert
        ->integer($zero)
           ->isLessThanOrEqualTo(0); // Will pass
```

Note

values given to isLessThanOrEqualTo must be actual integers.

2.2.5 isGreaterThanOrEqualTo

isGreaterThanOrEqualTo verify that the tested integer is greater or equal to a given integer.

```
$zero = 0;

$this->assert
        ->integer($zero)
           ->isGreaterThanOrEqualTo(-1); // Will pass

$this->assert
        ->integer($zero)
           ->isGreaterThanOrEqualTo('-1'); // Will fail, you have to pass an
actual integer to isGreaterThanOrEqualTo

$this->assert
        ->integer($zero)
           ->isGreaterThanOrEqualTo(0); // Will pass
```

Note

values given to isGreaterThanOrEqualTo must be actual integers.

2.3 float

This is the asserter dedicated to floating values testing.

It extends the integer asserter making it possible to test floating point values.

You can use every assertions that are present in the integer asserter. Note

Of course, while testing float values, assertions that expected integers will expect float values (isGreaterThan, isGreaterOrEqualTo, isLessThan, isLessThanOrEqualTo)

2.4 boolean

This is the asserter dedicated to boolean testing.

It extends the variable asserter.

2.4.1 isTrue

isTrue verify that the tested boolean is true (strictly equals to false).

```
$boolean = false;
$boolean2 = true;

$this->assert
        ->boolean($boolean)
           ->isTrue(); // Will fail

$this->assert
        ->boolean($boolean2)
           ->isTrue(); // Will pass
```

2.4.2 isFalse

isFalse verify that the tested boolean is false (strictly equals to false).

```
$boolean = false;
$boolean2 = true;

$this->assert
       ->boolean($boolean)
          ->isFalse(); // Will pass

$this->assert
       ->boolean($boolean2)
          ->isFalse(); // Will fail
```

2.5 string

This is the asserter dedicated to string testing.

It extends the variable asserter : You can use every assertions of the variable asserter while testing a string.

2.5.1 isEmpty

isEmpty verify that the string is empty (no characters)

```
$emptyString = '';
$nonEmptyString = ' ';

$this->assert
        ->string($emptyString)
            ->isEmpty();//Will pass
$this->assert
        ->string($nonEmptyString)
            ->isEmpty();//Will fail
```

2.5.2 isNotEmpty

isNotEmpty verify that the string is not empty (contains some characters)

```
$emptyString = '';
$nonEmptyString = ' ';

$this->assert
        ->string($emptyString)
            ->isNotEmpty();//Will fail
$this->assert
        ->string($nonEmptyString)
            ->isNotEmpty();//Will pass
```

2.5.3 match

match will try to verify that the string matches a given regular expression.

```
$polite = 'Hello the world';
$rude   = 'yeah... the world ';

$this->assert
        ->string($polite)
            ->match();//will pass

$this->assert
        ->string($rude)
            ->match();//will fail
```

2.5.4 hasLength

hasLength will verify that the string has a given length.

```
$string = 'Hello the world';

$this->assert
        ->string($string)
            ->hasLength(15);//Will pass

$this->assert
        ->string($string)
            ->hasLength(16);//Will fail
```

2.6 dateTime

This is the asserter dedicated to DateTime testing.

It extends from the variable asserter : You can use every assertions of the variable asserter while testing a DateTime object.

2.6.1 hasTimezone

2.6.2 isInYear

2.6.3 isInMonth

2.6.4 isInDay

2.6.5 hasDate

2.7 phpArray / array

This is the asserter dedicated to array testing.

It extends from the variable asserter : You can use every assertions of the variable asserter while testing arrays.

2.7.1 hasSize

hasSize will verify that the tested array has a given number of element (non recursive).

```
$array = array(1, 2, 3, 7);
$this->assert
        ->array($array)
        ->hasSize(4);//will pass

$this->assert
        ->array($array)
        ->hasSize(7);//Will fail
```

2.7.2 isEmpty

isEmpty will verify that the array is empty (does not contains any value)

```
$emptyArray = array();
$nonEmptyArray = array(null, null);

$this->assert
        ->array($emptyArray)
        ->isEmpty();//will pass

$this->assert
        ->array($nonEmptyArray)
        ->isEmpty();//will fail
```

2.7.3 isNotEmpty

isEmpty will verify that the array is not empty (contains at least one value of any kind)

```
$emptyArray = array();
$nonEmptyArray = array(null, null);

$this->assert
        ->array($emptyArray)
        ->isNotEmpty();//will fail

$this->assert
        ->array($nonEmptyArray)
        ->isNotEmpty();//will pass
```

2.7.4 contains

contains will verify that the tested array directly contains a given value (will not search for the value recursively). contains will not test the type of the value.

If you want to test both the type and the value, you will use strictlyContains.

```
$arrayWithNull = array(null);
$arrayWithEmptyString = array('', 1);
$arrayWithArrayWithNull = array(array(null));
$arrayWithString1 = array('1', 2, 3);

$this->assert
        ->array($arrayWithNull)
            ->contains(null);//will pass
        ->array($arrayWithEmptyString)
            ->contains(null)//will pass (null == '')
        ->array($arrayWithArrayWithNull)
            ->contains(null);//will fail, does not search recursively
        ->array($arrayWithString1)
            ->contains(1);//will pass, does not match the type
```

2.7.5 notContains

notContains will verify that the tested array does not contains a given value (will not search for the value recursively). notContains will not test the type of the value.

If you want to test both the type and the value, you will use strictlyNotContains.

```
$arrayWithNull = array(null);
$arrayWithEmptyString = array('', 1);
$arrayWithArrayWithNull = array(array(null));
$arrayWithString1 = array('1', 2, 3);

$this->assert
        ->array($arrayWithNull)
```

```
            ->notContains(null);//will fail
        ->array($arrayWithEmptyString)
            ->notContains(null)//will fail (null == '')
        ->array($arrayWithArrayWithNull)
            ->notContains(null);//will pass, does not search recursively
        ->array($arrayWithString1)
            ->notContains(1);//will fail, 1 == '1'
```

2.7.6 containsValues

containsValues will verify that the tested array does contains some values (given in an array) containsValues will not test the type of the values to look for.

If you want to test both the types and the values, you will use strictlyContainsValues.

```
$arrayWithString1And2And3 = array('1', 2, 3);

$this->assert
        ->array($arrayWithString1And2And3)
            ->containsValues(array(1, 2, 3))//will pass
            ->containsValues(array('1', '2', '3'))//will pass
            ->containsValues(array('1, 2, 3));//will pass
```

2.7.7 notContainsValues

notContainsValues will verify that the tested array does not contains any value of a given array notContainsValues will not test the type of the values to look for.

If you want to test both the types and the values, you will use strictlyNotContainsValues.

```
$arrayWithString1And2And3 = array('1', 2, 3);

$this->assert
        ->array($arrayWithString1And2And3)
            ->notContainsValues(array(1, 4, 5))//will faill as '1' is in the
tested array
            ->notContainsValues(array(4, 6, '2'))//will fail as 2 is in the
tested array
            ->notContainsValues(array('1', 2, 3))//will fail as all the values
are in the tested array
            ->notContainsValues(array(4, 5, 6));//will pass as none of the
values are in the tested array
```

2.7.8 strictlyContainsValues

strictlyContainsValues will verify that the tested array contains all the values of a given array stricltyContainsValues will test the type of the values to look for.

If you do not want to test both the types and the values, you will use containsValues.

```
$arrayWithString1And2And3 = array('1', 2, 3);

$this->assert
        ->array($arrayWithString1And2And3)
            ->notContainsValues(array(1, 2, 3))//will faill as '1' is in the
tested array, not 1
            ->notContainsValues(array('3', '2', '1'))//will fail as '3' and
'2' are not in the tested array, but 2 and 3 are
            ->notContainsValues(array(2, '1', 3));//will pass as all the
values are in the tested array
```

2.7.9 strictlyNotContainsValues

strictlyNotContainsValues will verify that the tested array does not contains any value of a given array strictlyNotContainsValues will test the type of the values to look for.

If you do not want to test both the types and the values, you will use notContainsValues.

```
$arrayWithString1And2And3 = array('1', 2, 3);

$this->assert
        ->array($arrayWithString1And2And3)
            ->notContainsValues(array(1, 4, 5))//will pass as none of the
values are in the tested array (1 !== '1')
            ->notContainsValues(array(4, 6, '2'))//will pass as none of the
values are in the tested array (2 !== '2')
            ->notContainsValues(array('1', 2, 3))//will fail as all of the
values are in the tested array
            ->notContainsValues(array(4, 5, 6));//will pass as none of the
values are in the tested array
```

2.7.10 strictlyContains

contains will verify that the tested array directly contains a given value (will not search for the value recursively). contains will test the type of the value.

If you do not want to test both the type and the value, you will use contains.

```
$arrayWithNull = array(null);
$arrayWithEmptyString = array('', 1);
$arrayWithArrayWithNull = array(array(null));
$arrayWithString1 = array('1', 2, 3);

$this->assert
        ->array($arrayWithNull)
```

```
            ->strictlyContains(null)//will pass
        ->array($arrayWithEmptyString)
            ->strictlyContains(null)//will fail (null !== '')
        ->array($arrayWithArrayWithNull)
            ->strictlyContains(null)//will fail, does not search recursively
        ->array($arrayWithString1)
            ->strictlyContains(1)//will fail, 1 !== '1'
        ->array($arrayWithString1)
            ->strictlyContains('1');//Will pass
```

2.7.11 strictlyNotContains

strictlyNotContains will verify that the tested array does not contains a given value (will not search for the value recursively). strictlyNotContains will test the type of the value.

If you do not want to test both the type and the value, you will use notContains.

```
$arrayWithNull = array(null);
$arrayWithEmptyString = array('', 1);
$arrayWithArrayWithNull = array(array(null));
$arrayWithString1 = array('1', 2, 3);

$this->assert
        ->array($arrayWithNull)
            ->strictlyNotContains(null)//will fail
        ->array($arrayWithEmptyString)
            ->strictlyNotContains(null)//will pass (null !== '')
        ->array($arrayWithArrayWithNull)
            ->strictlyNotContains(null)//will pass, does not search recursively
        ->array($arrayWithString1)
            ->strictlyNotContains(1);//will pass, 1 !== '1'
```

2.7.12 hasKey

hasKey will verify that the given array has a given key

```
$array  = array(2, 4, 6);
$array2 = array("2"=>1, "3"=>2, "4"=>3);

$this->assert
        ->array($array1)
            ->hasKey(1)//will pass
            ->hasKey(2)//will pass
            ->hasKey('1')//will pass, keys are "casted", and $array[1] do
exists
            ->hasKey(5);//will fail
$this->assert
```

```
        ->array($array2)
            ->hasKey(2)//will pass
            ->hasKey("3")//will pass
            ->hasKey(0);//will fail
```

2.7.13 notHasKey

notHasKey will verify that the given array does not have a given key

```
$array  = array(2, 4, 6);
$array2 = array("2"=>1, "3"=>2, "3"=>3);

$this->assert
        ->array($array1)
            ->notHasKey(1)//will fail
            ->notHasKey(2)//will fail
            ->notHasKey('1')//will fail, keys are "casted", and $array[1] do
exists
            ->notHasKey(5);//will pass
$this->assert
        ->array($array2)
            ->notHasKey(2)//will fail
            ->notHasKey("3")//will fail
            ->notHasKey(0);//will pass
```

2.7.14 hasKeys

hasKeys will verify that the tested array contains all the given keyx (given as an array)

```
$array  = array(2, 4, 6);
$array2 = array("2"=>1, "3"=>2, "4"=>3);

$this->assert
        ->array($array1)
            ->hasKeys(array(1, 2))//will pass
            ->hasKeys(array('0', 2))//will pass
            ->hasKeys(array("2", 0))//will pass
            ->hasKeys(array(0, 3))//will fail, $array[3] does not exists

$this->assert
        ->array($array2)
            ->hasKeys(array(2, "3"))//will pass
            ->hasKeys(array("3", 4));//will pass
```

2.7.15 notHasKeys

notHasKeys will verify that the tested array does not contains any of the given keys (given as an array of keys)

```
$array  = array(2, 4, 6);
$array2 = array("2"=>1, "3"=>2, "4"=>3);

$this->assert
        ->array($array1)
            ->notHasKeys(array(1, 2))//will fail, all the keys exists in the
tested array
            ->notHasKeys(array('0', 3))//will fail, $array['0'] exists
            ->notHasKeys(array("4", 5))//will pass, none of the keys exists in
the tested array
            ->notHasKeys(array(3, 'two'))//will pass, none of the keys exists
in the tested array

$this->assert
        ->array($array2)
            ->notHasKeys(array(2, "3"))//will pass
            ->notHasKeys(array("3", 4));//will pass
```

2.8 sizeOf

This asserter is dedicated to test the length of an array.

It extends from the integer asserter : You can use every assertions of the integer asserter while testing the size of an array.

2.9 object

This is the asserter dedicated to object testing.

It extends from the variable asserter : You can use every assertions of the variable asserter while testing an object.

2.9.1 isInstanceOf

isInstanceOf will tell if the tested object is an instance of a given interface and or a sub-class of a given type.

```
$stdClass = new stdClass();
$this->assert
        ->object($stdClass)
            ->isInstanceOf('\StdClass')//Will pass
            ->isInstanceOf('\Iterator');//Will fail


interface SomeInterface
{
    public function doTest();
```

```
}

class SomeClass implements SomeInterface
{
    public function doTest ()
    {
        echo "testing atoum is the best thing ever.";
    }
}

class SomeChildClass extends SomeClass
{

}

$someClass = new SomeClass();
$someClone = clone($someClass);
$someChildClass = new SomeChildClass();

$this->assert
        ->object($someClass)
            ->isInstanceOf('\SomeClass')//will pass
            ->isInstanceOf('\SomeInterface')//will pass
            ->isInstanceOf('\SomeChildClass');//will fail

$this->assert
        ->object($someClone)
            ->isInstanceOf('\SomeClass')//will pass
            ->isInstanceOf('\SomeInterface')//will pass
            ->isInstanceOf('\SomeChildClass');//will fail

$this->assert
        ->object($someChildClass)
            ->isInstanceOf('\SomeClass')//will pass, inheritance
            ->isInstanceOf('\SomeInterface')//will pass, inheritance of
interfaces
            ->isInstanceOf('\SomeChildClass');//will pass
```

2.9.2 hasSize

hasSize will check the size of an object. This assertion have sense mainly if your object implements the Countable interface.

2.9.3 isEmpty

2.10 phpClass / class

This is the asserter dedicated to class definition testing.

2.10.1 hasParent

2.10.2 hasNoParent

2.10.3 isSubclassOf

2.10.4 hasInterface

2.10.5 isAbstract

2.10.6 hasMethod

2.11 testedClass

This is the asserter dedicated to the tested class definition testing.

It extends the phpClass (class) asserter : You can use every assertions of the phpClass asserter while testing the tested class.

2.12 hash

This is the asserter dedicated to the validation of hashing function results.

It extends the string asserters : You can use every asertions of the string asserter while testing a hash.

2.12.1 isSha1

isSha1 verify that the given hash *could be* the result of a sha1 hash.

2.12.2 isSha256

isSha256 verify that the given hash *could be* the result of a sha256 hash.

2.12.3 isSha512

isSha256 verify that the given hash *could be* the result of a sha512 hash.

2.12.4 isMd5

md5 verify that the given hash *could be* the result of a md5 hash.

2.13 error

This asserter is dedicated to error testing.

2.13.1 exists

2.13.2 notExists

2.13.3 withType

2.13.4 withAnyType

2.13.5 withMessage

2.13.6 withAnyMessage

2.13.7 withPattern

2.14 exception

This is the asserter dedicated to exception testing.

It extends from the object asserter : You can use every assertions of the object asserter while testing exceptions.

atoum takes part of closures to test exceptions.

```
$this->assert
        ->exception(function () {
            //this code will raise an exception
            throw new Exception('This is an exception');
        })
```

2.14.1 hasDefaultCode

2.14.2 hasCode

2.14.3 hasMessage

2.14.4 hasNestedException

2.15 mock

This is the asserter dedicated to test your code using mock objects.