concise Documentation

Release 1.8.0

Elliot Chance

September 29, 2015

Contents

1 Installation **3**
- 1.1 Requirements . . . 3
- 1.2 Composer . . . 3

2 Writing Tests **5**
- 2.1 Simple Example . . . 5
- 2.2 Verify vs Assert . . . 5

3 Assertions **7**
- 3.1 Arrays . . . 7
- 3.2 Basic . . . 7
- 3.3 Booleans . . . 8
- 3.4 Date and Time . . . 8
- 3.5 Exceptions . . . 8
- 3.6 Files . . . 8
- 3.7 Numbers . . . 8
- 3.8 Objects and Classes . . . 9
- 3.9 Regular Expressions . . . 9
- 3.10 Strings . . . 9
- 3.11 Types . . . 9
- 3.12 URLs . . . 10

4 Running Tests **11**
- 4.1 The Concise CLI . . . 11
- 4.2 Continuous Integration (CI) . . . 11

5 Mocking **13**
- 5.1 Creating Mocks . . . 13
- 5.2 Verifying Mocks . . . 15
- 5.3 Exposing . . . 16
- 5.4 Stubbing . . . 17
- 5.5 Expectations . . . 18
- 5.6 Actions . . . 20
- 5.7 Properties . . . 22
- 5.8 Limitations . . . 22

6 Syntaxes **25**
- 6.1 What is a Syntax? . . . 25
- 6.2 Restricting Data Types . . . 25

6.3 Special Data Types 26

7 Modules **29**
7.1 Creating a Module 29
7.2 Loading a Module 29
7.3 Testing Modules 30

8 Extending Concise **31**
8.1 Using Concise with Other Frameworks 31

9 Changelog **33**

Concise is unit testing framework that uses plain English and minimal code. It extends and is fully compatible with existing PHPUnit projects.

Highlights include:

- 100% compatible with PHPUnit, no changes required. You may use as many features as you wish.
- Much better mocking framework with a lot less typing.
- Huge array of assertions to save on boilerplate code.
- Assert and verify are supported.

CHAPTER 1

Installation

1.1 Requirements

Concise requires PHPUnit which is included as a dependency of concise. Concise is compatable and tested against all minor releases of PHPUnit starting with with 4.0. Builds can be found at Travis CI with different `COMPOSER` values.

PHP versions 5.3 to 5.6 are supported. However, HHVM is not yet supported.

1.2 Composer

Concise is provided through Composer. The easiest way to include concise as a development dependency for your current project:

```
composer require-dev elliotchance/concise
```

Alternatively you can add it to your `composer.json` file:

```
{
    "require-dev": {
        "elliotchance/concise": "~2.0"
    }
}
```

CHAPTER 2

Writing Tests

If your familar with PHPUnit then there isn't much to explain here. You may use all the same processes and constructs as PHPUnit - the only difference is the class you extend from will be `Concise\Core\TestCase`.

2.1 Simple Example

```
class MyTest extends TestCase
{
    public function testSomeStuff()
    {
        $result = 100 + 23;
        $this->assert($result)->exactlyEquals(123);

        $a = ['foo' => 'bar'];
        $this->assertArray($a)->isAssociative;
        $this->assertArray($a)->hasKey('foo');
    }
}
```

Assertions are made from chaining words and values together. The chaining creates the syntax for the assertion:

```
// array ? has key ?
$this->assertArray($a)->hasKey('foo');
```

This syntax is used to find the assertion. If the assertion fails ti will be able to render the data and the assertion in a pretty way.

2.2 Verify vs Assert

`verify` is a stand in replacement for `assert` that does not stop the execution of the current test. This is useful when testing several values where a failure would not cause an error:

```
class MyTest extends TestCase
{
    public function testVerify()
    {
        $this->verify(1)->equals(2);
        $this->verify(2)->equals(2);
        $this->verify(3)->equals(2);
```

```
    }
}
```


1. MyTest::testVerify
2 verify failures:
1 equals 2
3 equals 2
1 assertion, 0 seconds
1 / 1 (100%)

CHAPTER 3

Assertions

3.1 Arrays

- array *array* does not have item *array* - Assert an array does not have key and value item.
- array *array* does not have key *int|string* - Assert an array does not have a key.
- array *array* does not have keys *array* -
- array *array* does not have value *mixed* - Assert an array does not have any occurrences of the given value.
- array *array* has item *array* - Assert an array has key and value item.
- array *array* has items *array* - Assert an array has all key and value items.
- array *array* has key *int|string* - Assert an array has key, returns value.
- array *array* has keys *array* - Assert an array has several keys in any order.
- array *array* has value *mixed* - Assert an array has at least one occurrence of the given value.
- array *array* has values *array* - Assert an array has several values in any order.
- array *array* is associative - Assert an array is associative.
- array *array* is empty - Assert an array is empty (no elements).
- array *array* is not associative - Assert an array is not associative.
- array *array* is not empty - Assert an array is not empty (at least one element).
- array *array* is not unique - Assert that an array only has at least one element that is repeated.
- array *array* is unique - Assert that an array only contains unique values.

3.2 Basic

- *mixed* does not equal *mixed* - Assert two value do not match with no regard to type.
- *mixed* does not exactly equal *mixed* - Assert two values are of exactly the same type and value.
- *mixed* equals *mixed* - Assert values with no regard to exact data types.
- *mixed* exactly equals *mixed* - Assert two values match data type and value.
- *mixed* is not the same as *mixed* - Assert two values are of exactly the same type and value.
- *mixed* is the same as *mixed* - Assert two values match data type and value.

3.3 Booleans

- *mixed* is false - Assert value is false.
- *mixed* is falsy - Assert a value is a false-like value.
- *mixed* is true - Assert a value is true.
- *mixed* is truthy - Assert a value is a non false-like value.

3.4 Date and Time

- date *int|string|DateTime* is after *int|string|DateTime* - A date/time is after another date/time, returns original date in the same format as provided.
- date *int|string|DateTime* is before *int|string|DateTime* - A date/time is before another date/time, returns original date in the same format as provided.

3.5 Exceptions

- closure *callable* does not throw *class* - Assert that a specific exception is not thrown.
- closure *callable* does not throw exception - Assert that no exception is thrown.
- closure *callable* throws *class* - Assert a specific exception was thrown.
- closure *callable* throws anything except *class* - Assert any exception except a specific one was thrown.
- closure *callable* throws exactly *class* - Assert a specific exception was thrown.
- closure *callable* throws exception - Assert an exception was thrown.

3.6 Files

- file *string* does not equal *string* - Compare string value with the contents of a file.
- file *string* equals *string* - Compare string value with the contents of a file.

3.7 Numbers

- *number* is between *number* and *number* - A number must be between two values (inclusive), returns value.
- *number* is greater than *number* - A number is greater than another number.
- *number* is greater than or equal to *number* - A number is greater than or equal to another number.
- *number* is less than *number* - A number is less than another number.
- *number* is less than or equal to *number* - A number is less than or equal to another number.
- *number* is not between *number* and *number* - A number must not be between two values (inclusive).
- *number* is not within *number* of *number* - Assert two values are not close to each other.
- *number* is within *number* of *number* - Assert two values are close to each other.

3.8 Objects and Classes

- *object*|*class* is an instance of *class* - Assert an objects class or subclass.
- *object*|*class* is not an instance of *class* - Assert than an object is not a class or subclass.
- object *object* does not have property *string* - Assert that an object does not have a property.
- object *object* has property *string* - Assert that an object has a property. Returns the properties value.

3.9 Regular Expressions

- string *string* does not match *regex* - Assert that a string does not match a regular expression.
- string *string* matches *regex* - Assert that a string matches a regular expression.

3.10 Strings

- string *mixed* does not end with *mixed* - Assert a string does not end with another string.
- string *mixed* does not start with *mixed* - Assert a string does not not start (begin) with another string.
- string *string* contains *string* - A string contains a substring. Returns original string.
- string *string* contains case insensitive *string* - A string contains a substring (ignoring case-sensitivity). Returns original string.
- string *string* does not contain *string* - A string does not contain a substring. Returns original string.
- string *string* does not contain case insensitive *string* - A string does not contain a substring (ignoring case-sensitivity). Returns original string.
- string *string* ends with *string* - Assert a string ends with another string.
- string *string* is empty - Assert a string is zero length.
- string *string* is not empty - Assert a string has at least one character.
- string *string* starts with *string* - Assert a string starts (begins) with another string.

3.11 Types

- *mixed* is a bool - Assert a value is true or false.
- *mixed* is a boolean - Assert a value is true or false.
- *mixed* is a number - Assert that a value is an integer or floating-point.
- *mixed* is a string - Assert value is a string.
- *mixed* is an array - Assert a value is an array.
- *mixed* is an int - Assert value is an integer type.
- *mixed* is an integer - Assert value is an integer type.
- *mixed* is an object - Assert value is an object.
- *mixed* is not a bool - Assert a value is not true or false.

- *mixed* is not a boolean - Assert a value is not true or false.
- *mixed* is not a number - Assert that a value is not an integer or floating-point.
- *mixed* is not a string - Assert a value is not a string.
- *mixed* is not an array - Assert a value is not an array.
- *mixed* is not an int - Assert a value is not an integer type.
- *mixed* is not an integer - Assert a value is not an integer type.
- *mixed* is not an object - Assert a value is not an object.
- *mixed* is not null - Assert a value is not null.
- *mixed* is not numeric - Assert value is not a number or string that represents a number.
- *mixed* is null - Assert a value is null.
- *mixed* is numeric - Assert value is a number or string that represents a number.

3.12 URLs

- url *string* has fragment *string* - URL has fragment.
- url *string* has host *string* - URL has host.
- url *string* has password *string* - URL has password.
- url *string* has path *string* - URL has path.
- url *string* has port *integer* - URL has port.
- url *string* has query *string* - URL has query.
- url *string* has scheme *string* - URL has scheme.
- url *string* has user *string* - URL has user.
- url *string* is valid - Validate URL.

CHAPTER 4

Running Tests

4.1 The Concise CLI

Concise comes with a CLI that acts as a wrapper for the original `phpunit` command. You may use all the available options of `phpunit`, however the `concise` executable offers a few more and has a much nicer result printer.

Likewise you can still run pure concise tests through the `phpunit` runner. Which is handy for existing CI systems.

4.2 Continuous Integration (CI)

The default result printer will likely not work so well with your CI and other non-interactive systems. There are several solutions for this;

1. You may continue to use the `phpunit` executable and printer which will work exactly like you expect it to.
2. There is an option for concise to use an alternate printer used for CI: `--ci`. This will hide the progress bar and only update progress line no more than once each percentage.

   The advantage of this over the traiditional `phpunit` executable is you will be able to see failures as they happen, rather than waiting till the end of the run.

CHAPTER 5

Mocking

5.1 Creating Mocks

There are three types of mocks available:

- *Normal mocks* (or simply *mocks*) define the behavior for every method you expect to interact with. If you test interacts with any method other than what you have explicitly stated an exception will be thrown.
- *Nice mocks* work like the original object where if you don't specify a given action for a method it will perform as if the mock didn't exist (pass through to the original method).
- *Partial mocks* create a mock from an already existing object. This means you can add custom rules to an object that already contains some arbitrary state.

5.1.1 Normal Mocks

```
$this->mock('\My\Class')
     ->expect('myMethod')->once()->andReturn(123)
     ->stub('myOtherMethod')->andThrow(new \Exception('Uh-oh!'));
     ->get();
```

The class you are mocking must exist. Either already loaded or able to be loaded through the class loader(s). This is not a limitation because concise does this for safety. If you want to create a mock but you do not need to inherit from another class then you can leave the class name out and a mock will be created from a `\stdClass`:

```
$this->mock()
     ->expect('myMethod')->andReturn(123)
     ->get();
```

This type of mock does not invoke the constructor since plain mocks are supposed to be totally hollow and the constructor could potentially setup an unexpected state or call methods that would not have actions associated with them.

5.1.2 Nice Mocks

Nice mocks work like the original object where if you don't specify a given action for a method it will perform as if the mock didn't exist (pass through to the original method).

```
$this->niceMock('\My\Class')
     ->...
     ->get();
```

5.1.3 Partial Mocks

Partial mocks create a mock from an already existing object. This means you can add custom rules to an object that already contains some arbitrary state.

```
$calculator = new Calculator();
$calculator->setMemory(10);

$mock = $this->partialMock($calculator)
             ->get();

$mock->addToMemory(20);
$mock->getMemory(); // 30
```

5.1.4 Constructors

If you are mocking a class with a constructor you can provide the constructor arguments as a second parameter:

```
class MyClass
{
    public function __construct($number, $string) {}
}

$this->mock('\My\Class', array(123, 'foobar'))
     ->...
```

Or you can disable the original constructor:

```
$this->mock('\My\Class')
     ->disableConstructor()
     ->...
```

Note: Constructors are always run by default, even in normal mocks (which have all methods stubbed off). The reason for this is even in a normal mock you may want the constructor to set up the state of the object, whilst leaving you with the ability to turn this off with `disableConstructor()`.

5.1.5 Programmatically Building Mocks

You would have noticed that all mock definitions end with `get()` which compiles the rules into the actual mock for use. If you try to use the object before then you will be talking to the `MockBuilder` instance.

This allows you to generate mocks programmatically:

```
public function createMockForCalc($expectsAdd = false)
{
    $mock = $this->mock('\My\Calculator');
    if ($expectsAdd) {
        $mock->expects('add');
    }
    else {
        $mock->stub('add');
    }
    $mock->andReturn(8);
    return $mock->get();
}
```

Conversely, you may use `get()` multiple times to generate different classes with the same rules:

```
$mockTemplate = $this->mock()
                     ->stub(['add' => 8]);
$mock1 = $mockTemplate->get();
$mock2 = $mockTemplate->get();

echo get_class($mock1) . " " . get_class($mock2); // stdClass_abd1240f stdClass_4432eba7
```

5.1.6 Changing the Class Name and Namespace of a Mock

The name of your class will be generated automatically to be unique, however if you want to name your class something specific you can specify this:

```
$mock = $this->mock('\My\Calculator')
             ->setCustomClassName('Calc')
             ->get();
echo get_class($mock);

// My\Calc
```

If the class name you specify does not contain a namespace then it will be placed into the same namespace as the original class you are mocking. However, you can change the namespace completely by specifying the fully-qualified class:

```
$mock = $this->mock('\My\Calculator')
             ->setCustomClassName('Secret\Location\Calc')
             ->get();
echo get_class($mock);

// Secret\Location\Calc
```

Or even move the class into the global namespace by preceding the class name with a backslash:

```
$mock = $this->mock('\My\Calculator')
             ->setCustomClassName('\Calculator')
             ->get();
echo get_class($mock);

// Calculator
```

5.2 Verifying Mocks

All mocks are automatically asserted (checking that all the requirements have been fulfilled) at the end of each test case.

5.2.1 Manually Verifying Mocks

Sometimes you may want or need to verify them before the end of the test. For example:

```
public function testMock()
{
    $mock = $this->mock('MyClass')
                 ->expect('myMethod')
                 ->get();
```

```
    // ... do some stuff
    $this->assertMock($mock);
}
```

In the example above the mock will be asserted on the spot and cause the same failure if any requirements are no fulfilled, however it does some other things to the mock:

- A mock can only be asserted once, that means that since we are validating it here it will *not* be validated again when the test ends, and;
- Validating a mock more than once (calling `assertMock()`) more than once on the same mock will yield an error.

5.3 Exposing

5.3.1 Methods

Exposing a method will simply make its visibility `public` this does not interfere with any actions behavior of the method:

```
class MyClass
{
    protected function foo()
    {
        return 'bar';
    }
}
```

```
$mock = $this->niceMock('MyClass')
              ->expose('foo')
              ->get();
$mock->foo();
```

If you need to expose several methods there is also a variety of ways this can be done:

```
$mock = $this->niceMock('MyClass')
              ->expose('foo')
              ->expose('bar')
              ->get();
```

```
$mock = $this->niceMock('MyClass')
              ->expose(['foo', 'bar'])
              ->get();
```

```
$mock = $this->niceMock('MyClass')
              ->expose('foo', 'bar')
              ->get();
```

Some caveats:

- The method you are exposing must exist, but it doesn't have to be `protected`. Exposing a `public` method is allowed but would have no effect.
- You cannot expose a `private` method. If you try you will get an exception.

5.3.2 All Methods

In some cases you may want to expose all the non-public methods in a mock. This is generally unwise because your testing code should ideally only use the public API provided by the objects and services that you are testing.

```
$mock = $this->niceMock('MyClass')
             ->exposeAll()
             ->get();

$mock->secretMethod();
```

`exposeAll()` will actually retrieve the methods available on the object and promote any method that is not `final` or `private` to a `public` visibility. See Mocking Final Classes and Methods for more information.

5.3.3 Properties

In some case you may also want to get or set properties on an object that do not have a public visibility.

```
class MyClass
{
    protected $value = 'foo';
}
```

```
public function testValueIsFoo()
{
    $myClass = new MyClass();
    $this->assert($this->getProperty($myClass, 'value'))->equals('foo');
}
```

The above will work for all visibilities of a property.

Likewise you can use the `setProperty` method provided by `Concise\Core\TestCase`:

```
public function testValueIsBar()
{
    $myClass = new MyClass();
    $this->setProperty($myClass, 'value', 'bar');
    $this->assert($this->getProperty($myClass, 'value'))->equals('bar');
}
```

5.4 Stubbing

Stubbing is the act of changing the return value or associated action of a method when it is invoked (the basic principle of a mock). You are not specifing any expectation so the stubbed method may be called zero or more times:

```
$calculatorMock = $this->mock('\Calculator')
                       ->stub('add')->andReturn(8)
                       ->get();

$calculatorMock->add(); // returns 8
```

If you only want to stub a method to return a value then you can use the array version to specify one or more stubs:

```
$calculatorMock = $this->mock('\Calculator')
                       ->stub(['add' => 8])
                       ->get();
```

Concise allows for all the same rules with `static` methods with exactly the same syntax.

5.4.1 Setting the Same Actions on Multiple Methods

You can set actions on multiple methods at the same time by specifying them in the same clause like:

```
$calculatorMock = $this->mock('\Calculator')
                       ->stub('add', 'subtract')->andReturn(0)
                       ->get();
```

In the above example both `add` and `subtract` will return `0` when called. It is a shorter way of writing:

```
$calculatorMock = $this->mock('\Calculator')
                       ->stub('add')->andReturn(0)
                       ->stub('subtract')->andReturn(0)
                       ->get();

// or

$calculatorMock = $this->mock('\Calculator')
                       ->stub(['add' => 0, 'subtract' => 0])
                       ->get();
```

5.5 Expectations

Expectations require some criteria to be fulfilled during the test. This may be that a method is called a specified amount of times:

```
$calculatorMock = $this->mock('\Calculator')
                       ->expect('add')->once()->andReturn(8)
                       ->get();
```

The number of exptected times may be one of:

- `never()` - fail if this method is called.
- `once()` - must be exactly once.
- `twice()` - must be called exactly twice.
- `times(int)` - exact number of times.

All method expectations must have an action except in the case of `never()`.

Stubs and expectation share the same commonality when it comes to actions when the method is called.

For convenience there is also an `expects` method that performs exactly the same way.

5.5.1 Setting the Same Expectations on Multiple Methods

Like stubbing, you can set requirements on multiple methods at the same time by specifying them in the same clause like:

```
$calculatorMock = $this->mock('\Calculator')
                       ->expects('add', 'subtract')
                       ->get();
```

In the above example both `add` and `subtract` will be expected to be called, it is a shorter way of writing:

```
$calculatorMock = $this->mock('\Calculator')
                       ->expects('add')
                       ->expects('subtract')
                       ->get();
```

Likewise, an action or requirement will be applied to all of the methods in the clause like:

```
$calculatorMock = $this->mock('\Calculator')
                       ->expects('add', 'subtract')->twice()->andReturn(0)
                       ->get();
```

`add` and `subtract` will be each have to be called twice and will return `0`.

5.5.2 Expecting Arguments

Stubs and expectations may have an additional `with()` clause:

```
$calculatorMock = $this->mock('\Calculator')
                       ->stub('add')->with(3, 5)->andReturn(8)
                       ->get();

$calculatorMock->add(3, 5); // returns 8
```

You may specify more than one `with()` condition to handle different scenarios:

```
$calculatorMock = $this->mock('\Calculator')
                       ->stub('add')->with(3, 5)->andReturn(8)
                                    ->with(2, 7)->andReturn(9)
                       ->get();
```

When you are using `with()` you cannot specify the number of expected calls for a method, but rather you must specify the number of times for each `with()` condition:

```
$calculatorMock = $this->mock('\Calculator')
                       ->expects('add')->with(3, 5)->twice()
                                       ->with(2, 7)
                       ->get();
```

In the example above `add(3, 5)` must be invoked twice *and* `add(2, 7)` must be invoked once (the `expects` clause will default to once).

5.5.3 Ignoring Parameter Values

Sometimes you only need to restrict some of the incoming paramter values, in this case there is a `ANYTHING` constant provided by `Concise\Core\TestCase`:

```
$calculatorMock = $this->mock('\Calculator')
                       ->expects('add')->with(3, self::ANYTHING)
                       ->get();
```

5.6 Actions

5.6.1 andDo(callback)

```
$this->mock()
     ->stub('myMethod')->andDo(function() {
         echo "myMethod() was called.";
     }))
     ->get();
```

This can also be used as a way to handle state that might be too complicated for the mocking engine:

```
$calledOddTimes = false;
$this->mock()
     ->stub('myMethod')->andDo(function() use (&$calledOddTimes) {
         $calledOddTimes = !$calledOddTimes;
     }))
     ->get();
$this->assert($calledOddTimes)->isTrue;
```

`andDo` will pass through arguments:

```
$mock = $this->mock('MyClass')
             ->expect('foo')->andDo(function ($a, $b) {
                 echo $a + $b;
             })
             ->get();

$mock->foo(3, 5);

// prints:
// 8
```

5.6.2 andReturn(value)

Return `value` where `value` can be of any type.

You may also provide more than one argument to specify multiple resturn values:

```
$mock = $this->mock()
             ->stub('myMethod')->andReturn('foo', 123)
             ->get();
$mock->myMethod(); // 'foo'
$mock->myMethod(); // 123
```

When using multiple return values the method can not be called more times than you have return values for - otherwise an exception is thrown.

5.6.3 andReturnCallback(callback)

Return the value returned by a callback function.

```
$mock = $this->mock()
             ->stub('myMethod')->andReturnCallback(function () {
                return 'foo';
```

```
            })
            ->get();
$mock->myMethod(); // 'foo'
```

The return value is evaluated when the invocation is made, so you can return different values for each invocation.

An optional `Concise\Mock\InvocationInterface` is passed through as the first and only argument to gain insight about the invocation:

```
$mock = $this->mock()
            ->stub('myMethod')->andReturnCallback(
                function (InvocationInterface $invoke) {
                    return $invoke->getInvokeCount();
                }
            )
            ->get();
$mock->myMethod(); // 1
$mock->myMethod(); // 2
```

You can also access the invocation arguments:

```
$mock = $this->mock()
            ->stub('myMethod')->andReturnCallback(
                function (InvocationInterface $invoke) {
                    return $invoke->getArgument(1);
                }
            )
            ->get();
$mock->myMethod('foo', 'bar'); // bar
```

5.6.4 andReturnProperty(propertyName)

To return the value of a property (of any visibility) when a method is invoked you can use `andReturnProperty()`:

```
class MyClass
{
    protected $hidden = 'foo';

    public function myMethod()
    {
        return 'bar';
    }
}

$mock = $this->mock()
            ->stub('myMethod')->andReturnProperty('hidden')
            ->get();
$mock->myMethod(); // foo
```

5.6.5 andReturnSelf()

Return the mock instance (`return $this`). This is useful when you are mocking classes that using the chaining principle with methods.

5.6.6 andThrowException(exception)

Throw the `exception` when the method is called.

```
$this->mock()
     ->stub('myMethod')->andThrow(new \Exception('Uh-oh!'))
     ->get();
```

5.7 Properties

5.7.1 Setting a Single Property

You can set a properties when creating a mock using `setProperty`:

```
$mock = $this->niceMock('MyClass')
              ->setProperty('foo', 'bar')
              ->get();

$this->foo; // bar
```

5.7.2 Setting Multiple Properties

Setting multiple properties can be done with `setProperties`:

```
$mock = $this->niceMock('MyClass')
              ->setProperties([
                  'foo' => 'bar',
                  'bar' => 'baz',
              ])
              ->get();

$this->bar; // baz
```

When using `setProperties` it will *add on* the provided properties, not replace any previously set ones.

5.7.3 Other Information

Note: The property or properties are set after all other aspects of the mock have been setup. This means properties that may be set as part of a partial mock will be overridden by the properties provided.

This feature was introduced in Multiverse (Release v1.7.0).

5.8 Limitations

5.8.1 Traits

A `trait` cannot be mocked: Issue #66

5.8.2 Final Classes and Methods

Classes that are `final` will not be available to mock - an exception will be thrown if this is attempted.

This also applies to `final` methods.

CHAPTER 6

Syntaxes

6.1 What is a Syntax?

A *syntax* explains how an assertion gets translated for a matcher. A simple example is:

```
? equals ?
```

Which can be used in your test:

```
$this->assert(123)->equals(456);
```

So the syntax `? equals ?` matches with two data items; `123` and `456`.

Syntaxes can contain multiple words in a row:

```
? is greater than ?
```

Which can be used the same way as:

```
$this->assert(123)->isGreaterThan(456);
```

Syntaxes can contain as many data points as you need:

```
? is within ? of ?
```

```
$this->assert(1)->isWithin(0.2)->of(0.9);
```

And finally the syntax can start with a data element or not:

```
date ? is after ?
```

```
$this->assertDate($foo)->isAfter(time());
```

6.2 Restricting Data Types

In a lot of cases it only makes sense for an assertion to work with known data types, for example:

```
? starts with ?
```

Here we are talking about strings. If someone were to put through a type that doesn't make sense or cannot be computed like:

```
$this->assert(new stdClass())->startsWith(true);
```

We would no doubt get some error, or at the very least the assertion would return an unreliable result.

There are two ways to mitigate this:

1. Do the type checking yourself in the matcher class by checking each data element for a sane type.
2. Use the syntax to specify the allowed types. This is must easer:

```
    ?:string starts with ?:string
```

Now concise will do the type checking for us. If we get some bad types it will throw an exception explaining the error and never need to the call the actual matcher. It also means that your matcher class can guarantee that the data elements taken in are both strings.

More complex requirements can be specified by separating with a comma:

```
?:int,float is greater than ?:int,float
```

Or, the reverse logic can be used to blacklist types (instead of whitelist) them:

```
?:!object is scalar
```

Will accept any type that is *not* an `object`.

6.3 Special Data Types

Due to PHP's relaxed typing we want to be sure we don't potentially run into this problem:

```
$this->assert('123')->isGreaterThan(1.23);
```

This will fail because `'123'` is a string, but it can also be treated as a number. So concise provides some special types that do value checking as well:

```
?:number is greater than ?:number
```

We can now safely use number-like values:

```
$this->assert('123')->isGreaterThan(1.23); // numbers
$this->assert('foo')->isGreaterThan(1.23); // 'foo' is not a number
```

See the table below for all the supported types:

Type	Example values
`int`	`123`
`integer`	
`float`	`1.23`
`double`	
`string`	`"abc"`
`array`	`array()`
`resource`	`fopen('.', 'r')`
`object`	`new \stdClass()`
`callable`	`function () { }`
`regex`	`"/foo/"`
`class`	`"Concise\Core\TestCase"`
`number`	`123`, `1.23`, `"12.3"`
`bool`	`true`

Separately from the type names in the table you may also specify specific classes:

```
?:DateTime is a date
?:\DateTime is a date
```

Subclasses are allowed.

CHAPTER 7

Modules

Modules contain assertions. If you need to create your own assertions you will likely want to create one or more modules.

7.1 Creating a Module

Each module is a class that extends `Concise\Module\AbstractModule` and contains methods that are annotated with the syntaxes they will match on, for example:

```
class UrlModule extends \Concise\Module\AbstractModule
{
    public function getName()
    {
        return "URLs";
    }

    /**
     * Validate URL.
     *
     * @syntax url ?:string is valid
     */
    public function urlIsValid()
    {
        $this->failIf(
            filter_var($this->data[0], FILTER_VALIDATE_URL) === false
        );
    }
}
```

Methods in a module can have zero or more `@syntax` annotations. You may use `fail()` or `failIf(bool)` to throw failures. Any value returned will be returned as-is to be used in nested assertions.

7.2 Loading a Module

Modules can be loaded through the `ModuleManager` like:

```
ModuleManager::getInstance()->loadModule(new MyModule());
```

Some things to note:

- It is safe to load the same module multiple times. Internally modules are identified by their class name so loading the same module will be ignored.
- Once modules are loaded into the `ModuleManager` they remain there for the entire run. If you had a bootstrap file for your test suite it would be a good idea to load your modules here, otherwise putting them in the appropriate test cases is fine too.

7.3 Testing Modules

Use the `Concise\Module\AbstractModuleTestCase` when testing modules:

```
class MyModuleTest extends AbstractMatcherTestCase
{
    public function setUp()
    {
        parent::setUp();
        $this->module = new MyModule();
    }

    public function testIntegerIsAnInteger()
    {
        $this->assert(123)->isAnInteger;
    }

    /**
     * @expectedException @expectedException \Concise\Core\DidNotMatchException
     */
    public function testFloatIsNotAnInteger()
    {
        $this->assert(123.0)->isAnInteger;
    }
}
```

CHAPTER 8

Extending Concise

8.1 Using Concise with Other Frameworks

Concise is designed to work perfectly over the top of PHPUnit. But it can also be used by any other framework by simply instantiating and managing the `Concise\Core\TestCase` yourself:

```
use \Concise\Core\TestCase;

class MyTinyTestSuite
{
    protected $testCase;

    public function __construct()
    {
        $this->testCase = new TestCase();
    }

    public function checkSomething()
    {
        $this->testCase->setUp();
        $this->testCase->assert(3 + 5)->equals(8);
        $this->testCase->tearDown();
    }
}
```

Since Concise implicitly expects `setUp()` and `tearDown()` methods to be called at appropriate times but does not enforce this behaviour - if you use it differently then it may do unexpected things.

CHAPTER 9

Changelog

The changelog is maintained via Github releases.