PHPUnit Manual

Sebastian Bergmann

PHPUnit Manual

Sebastian Bergmann

Publication date Edition for PHPUnit 5.5. Updated on 2016-09-20.

Table of Contents

List of Tables

List of Examples

Chapter 1. Installing PHPUnit

Requirements

PHPUnit 5.5 requires PHP 5.6; using the latest version of PHP is highly recommended.

PHPUnit requires the dom [http://php.net/manual/en/dom.setup.php] and json [http://php.net/manual/en/json.installation.php] extensions, which are normally enabled by default.

PHPUnit also requires the pcre [http://php.net/manual/en/pcre.installation.php], reflection [http://php.net/manual/en/reflection.installation.php], and spl [http://php.net/manual/en/spl.installation.php] extensions. These standard extensions are enabled by default and cannot be disabled without patching PHP's build system and/or C sources.

The code coverage report feature requires the Xdebug [http://xdebug.org/] (2.2.1 or later) and tokenizer [http://php.net/manual/en/tokenizer.installation.php] extensions. Generating XML reports requires the xmlwriter [http://php.net/manual/en/xmlwriter.installation.php] extension.

PHP Archive (PHAR)

The easiest way to obtain PHPUnit is to download a PHP Archive (PHAR) [http://php.net/phar] that has all required (as well as some optional) dependencies of PHPUnit bundled in a single file.

The phar [http://php.net/manual/en/phar.installation.php] extension is required for using PHP Archives (PHAR).

The openssl [http://php.net/manual/en/openssl.installation.php] extension is required for using the `--self-update` feature of the PHAR.

If the Suhosin [http://suhosin.org/] extension is enabled, you need to allow execution of PHARs in your `php.ini`:

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

To globally install the PHAR:

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

You may also use the downloaded PHAR file directly:

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

Windows

Globally installing the PHAR involves the same procedure as manually installing Composer on Windows [https://getcomposer.org/doc/00-intro.md#installation-windows]:

1. Create a directory for PHP binaries; e.g., `C:\bin`
2. Append **`;C:\bin`** to your `PATH` environment variable (related help [http://stackoverflow.com/questions/6318156/adding-python-path-on-windows-7])

3. Download https://phar.phpunit.de/phpunit.phar and save the file as `C:\bin\phpunit.phar`
4. Open a command line (e.g., press **Windows**+**R** » type **cmd** » **ENTER**)
5. Create a wrapping batch script (results in `C:\bin\phpunit.cmd`):

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

6. Open a new command line and confirm that you can execute PHPUnit from any path:

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

For Cygwin and/or MingW32 (e.g., TortoiseGit) shell environments, you may skip step 5. above, simply save the file as `phpunit` (without `.phar` extension), and make it executable via **`chmod 775 phpunit`**.

Verifying PHPUnit PHAR Releases

All official releases of code distributed by the PHPUnit Project are signed by the release manager for the release. PGP signatures and SHA1 hashes are available for verification on phar.phpunit.de [https://phar.phpunit.de/].

The following example details how release verification works. We start by downloading `phpunit.phar` as well as its detached PGP signature `phpunit.phar.asc`:

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

We want to verify PHPUnit's PHP Archive (`phpunit.phar`) against its detached signature (`phpunit.phar.asc`):

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

We don't have the release manager's public key (`6372C20A`) in our local system. In order to proceed with the verification we need to retrieve the release manager's public key from a key server. One such server is `pgp.uni-mainz.de`. The public key servers are linked together, so you should be able to connect to any key server.

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

Now we have received a public key for an entity known as "Sebastian Bergmann <sb@sebastian-bergmann.de>". However, we have no way of verifying this key was created by the person known as Sebastian Bergmann. But, let's try to verify the release signature again.

```
gpg phpunit.phar.asc
gpg: Signature made Sat 19 Jul 2014 01:28:02 PM CEST using RSA key ID 6372C20A
gpg: Good signature from "Sebastian Bergmann <sb@sebastian-bergmann.de>"
gpg:                 aka "Sebastian Bergmann <sebastian@php.net>"
gpg:                 aka "Sebastian Bergmann <sebastian@thephp.cc>"
gpg:                 aka "Sebastian Bergmann <sebastian@phpunit.de>"
gpg:                 aka "Sebastian Bergmann <sebastian.bergmann@thephp.cc>"
```

```
gpg:                 aka "[jpeg image of size 40635]"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: D840 6D0D 8294 7747 2937  7831 4AA3 9408 6372 C20A
```

At this point, the signature is good, but we don't trust this key. A good signature means that the file has not been tampered. However, due to the nature of public key cryptography, you need to additionally verify that key `6372C20A` was created by the real Sebastian Bergmann.

Any attacker can create a public key and upload it to the public key servers. They can then create a malicious release signed by this fake key. Then, if you tried to verify the signature of this corrupt release, it would succeed because the key was not the "real" key. Therefore, you need to validate the authenticity of this key. Validating the authenticity of a public key, however, is outside the scope of this documentation.

It may be prudent to create a shell script to manage PHPUnit installation that verifies the GnuPG signature before running your test suite. For example:

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

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

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

# Verify before running
gpg --verify phpunit.phar.asc phpunit.phar
if [ $? -eq 0 ]; then
    echo
    echo -e "\033[33mBegin Unit Testing\033[0m"
    # Run the testing suite
    `$after_cmd`
    # Cleanup
    if [ "$clean" -eq 1 ]; then
        echo -e "\033[32mCleaning Up!\033[0m"
        rm -f phpunit.phar
        rm -f phpunit.phar.asc
```

```
    fi
else
    echo
    chmod -x phpunit.phar
    mv phpunit.phar /tmp/bad-phpunit.phar
    mv phpunit.phar.asc /tmp/bad-phpunit.phar.asc
    echo -e "\033[31mSignature did not match! PHPUnit has been moved to /tmp/bad-phpunit
    exit 1
fi
```

Composer

Simply add a dependency on `phpunit/phpunit` to your project's `composer.json` file if you use Composer [https://getcomposer.org/] to manage the dependencies of your project. Here is a minimal example of a `composer.json` file that just defines a development-time dependency on PHPUnit 5.5:

```
{
    "require-dev": {
        "phpunit/phpunit": "5.5.*"
    }
}
```

For a system-wide installation via Composer, you can run:

```
composer global require "phpunit/phpunit=5.5.*"
```

Make sure you have `~/.composer/vendor/bin/` in your path.

Optional packages

The following optional packages are available:

`PHP_Invoker`

A utility class for invoking callables with a timeout. This package is required to enforce test timeouts in strict mode.

This package is included in the PHAR distribution of PHPUnit. It can be installed via Composer by adding the following `"require-dev"` dependency:

```
"phpunit/php-invoker": "*"
```

`DbUnit`

DbUnit port for PHP/PHPUnit to support database interaction testing.

This package is included in the PHAR distribution of PHPUnit. It can be installed via Composer by adding the following `"require-dev"` dependency:

```
"phpunit/dbunit": ">=1.2"
```

Chapter 2. Writing Tests for PHPUnit

Example 2.1, “Testing array operations with PHPUnit” shows how we can write tests using PHPUnit that exercise PHP's array operations. The example introduces the basic conventions and steps for writing tests with PHPUnit:

1. The tests for a class `Class` go into a class `ClassTest`.
2. `ClassTest` inherits (most of the time) from `PHPUnit\Framework\TestCase`.
3. The tests are public methods that are named `test*`.

   Alternatively, you can use the `@test` annotation in a method's docblock to mark it as a test method.
4. Inside the test methods, assertion methods such as `assertEquals()` (see Appendix A, *Assertions*) are used to assert that an actual value matches an expected value.

Example 2.1. Testing array operations with PHPUnit

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

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

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

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

> Whenever you are tempted to type something into a `print` statement or a debugger expression, write it as a test instead.
>
> —Martin Fowler

Test Dependencies

> Unit Tests are primarily written as a good practice to help developers identify and fix bugs, to refactor code and to serve as documentation for a unit of software under test. To achieve these benefits, unit tests ideally should cover all the possible paths in a program. One unit test usually covers one specific path in one function or method. However a test method is not necessary an encapsulated, independent entity. Often there are implicit dependencies between test methods, hidden in the implementation scenario of a test.
>
> —Adrian Kuhn et. al.

PHPUnit supports the declaration of explicit dependencies between test methods. Such dependencies do not define the order in which the test methods are to be executed but they allow the returning of an instance of the test fixture by a producer and passing it to the dependent consumers.

- A producer is a test method that yields its unit under test as return value.
- A consumer is a test method that depends on one or more producers and their return values.

Example 2.2, “Using the `@depends` annotation to express dependencies” shows how to use the `@depends` annotation to express dependencies between test methods.

Example 2.2. Using the `@depends` annotation to express dependencies

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

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

        return $stack;
    }

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

        return $stack;
    }

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

In the example above, the first test, `testEmpty()`, creates a new array and asserts that it is empty. The test then returns the fixture as its result. The second test, `testPush()`, depends on `testEmpty()` and is passed the result of that depended-upon test as its argument. Finally, `testPop()` depends upon `testPush()`.

Note

The return value yielded by a producer is passed "as-is" to its consumers by default. This means that when a producer returns an object a reference to that object is passed to the consumers. When a copy should be used instead of a reference then `@depends clone` should be used instead of `@depends`.

To quickly localize defects, we want our attention to be focussed on relevant failing tests. This is why PHPUnit skips the execution of a test when a depended-upon test has failed. This improves defect localization by exploiting the dependencies between tests as shown in Example 2.3, “Exploiting the dependencies between tests”.

Example 2.3. Exploiting the dependencies between tests

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

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

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

```
phpunit --verbose DependencyFailureTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

FS

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

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

/home/sb/DependencyFailureTest.php:6

There was 1 skipped test:

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


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

A test may have more than one `@depends` annotation. PHPUnit does not change the order in which tests are executed, you have to ensure that the dependencies of a test can actually be met before the test is run.

A test that has more than one `@depends` annotation will get a fixture from the first producer as the first argument, a fixture from the second producer as the second argument, and so on. See Example 2.4, “Test with multiple dependencies”

Example 2.4. Test with multiple dependencies

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

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

```
        return 'first';
    }

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

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

```
phpunit --verbose MultipleDependenciesTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

...

Time: 0 seconds, Memory: 3.25Mb

OK (3 tests, 3 assertions)
```

Data Providers

A test method can accept arbitrary arguments. These arguments are to be provided by a data provider method (`additionProvider()` in Example 2.5, “Using a data provider that returns an array of arrays”). The data provider method to be used is specified using the `@dataProvider` annotation.

A data provider method must be `public` and either return an array of arrays or an object that implements the `Iterator` interface and yields an array for each iteration step. For each array that is part of the collection the test method will be called with the contents of the array as its arguments.

Example 2.5. Using a data provider that returns an array of arrays

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

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

    public function additionProvider()
    {
        return [
            [0, 0, 0],
```

```
            [0, 1, 1],
            [1, 0, 1],
            [1, 1, 3]
        ];
    }
}
?>
```

```
phpunit DataTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

...F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

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

/home/sb/DataTest.php:9

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

When using a large number of datasets it's useful to name each one with string key instead of default numeric. Output will be more verbose as it'll contain that name of a dataset that breaks a test.

Example 2.6. Using a data provider with named datasets

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

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

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

```
phpunit DataTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

...F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:
```

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

/home/sb/DataTest.php:9

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

Example 2.7. Using a data provider that returns an Iterator object

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

require 'CsvFileIterator.php';

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

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

```
phpunit DataTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

...F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

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

/home/sb/DataTest.php:11

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

Example 2.8. The CsvFileIterator class

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

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

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

```
    }

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

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

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

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

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

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

When a test receives input from both a `@dataProvider` method and from one or more tests it `@depends` on, the arguments from the data provider will come before the ones from depended-upon tests. The arguments from depended-upon tests will be the same for each data set. See Example 2.9, “Combination of @depends and @dataProvider in same test”

Example 2.9. Combination of @depends and @dataProvider in same test

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

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

    public function testProducerFirst()
    {
        $this->assertTrue(true);
        return 'first';
    }

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

    /**
     * @depends testProducerFirst
     * @depends testProducerSecond
```

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

```
phpunit --verbose DependencyAndDataProviderComboTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

...F

Time: 0 seconds, Memory: 3.50Mb

There was 1 failure:

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

/home/sb/DependencyAndDataProviderComboTest.php:31

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

Note

When a test depends on a test that uses data providers, the depending test will be executed when the test it depends upon is successful for at least one data set. The result of a test that uses data providers cannot be injected into a depending test.

Note

All data providers are executed before both the call to the `setUpBeforeClass` static method and the first call to the `setUp` method. Because of that you can't access any variables you create there from within a data provider. This is required in order for PHPUnit to be able to compute the total number of tests.

Testing Exceptions

Example 2.10, “Using the expectException() method” shows how to use the `expectException()` method to test whether an exception is thrown by the code under test.

Example 2.10. Using the expectException() method

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

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

```
phpunit ExceptionTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ExceptionTest::testException
Expected exception InvalidArgumentException

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

In addition to the `expectException()` method the `expectExceptionCode()`, `expectExceptionMessage()`, and `expectExceptionMessageRegExp()` methods exist to set up expectations for exceptions raised by the code under test.

Alternatively, you can use the `@expectedException`, `@expectedExceptionCode`, `@expectedExceptionMessage`, and `@expectedExceptionMessageRegExp` annotations to set up expectations for exceptions raised by the code under test. Example 2.11, "Using the @expectedException annotation" shows an example.

Example 2.11. Using the @expectedException annotation

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

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

```
phpunit ExceptionTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ExceptionTest::testException
Expected exception InvalidArgumentException
```

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

Testing PHP Errors

By default, PHPUnit converts PHP errors, warnings, and notices that are triggered during the execution of a test to an exception. Using these exceptions, you can, for instance, expect a test to trigger a PHP error as shown in Example 2.12, “Expecting a PHP error using @expectedException”.

Note

PHP's `error_reporting` runtime configuration can limit which errors PHPUnit will convert to exceptions. If you are having issues with this feature, be sure PHP is not configured to suppress the type of errors you're testing.

Example 2.12. Expecting a PHP error using @expectedException

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

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

```
phpunit -d error_reporting=2 ExpectedErrorTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

.

Time: 0 seconds, Memory: 5.25Mb

OK (1 test, 1 assertion)
```

`PHPUnit_Framework_Error_Notice` and `PHPUnit_Framework_Error_Warning` represent PHP notices and warnings, respectively.

Note

You should be as specific as possible when testing exceptions. Testing for classes that are too generic might lead to undesirable side-effects. Accordingly, testing for the `Exception` class with `@expectedException` or `setExpectedException()` is no longer permitted.

When testing that relies on php functions that trigger errors like `fopen` it can sometimes be useful to use error suppression while testing. This allows you to check the return values by suppressing notices that would lead to a phpunit `PHPUnit_Framework_Error_Notice`.

Example 2.13. Testing return values of code that uses PHP Errors

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

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

?>
```

```
phpunit ErrorSuppressionTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

.

Time: 1 seconds, Memory: 5.25Mb

OK (1 test, 1 assertion)
```

Without the error suppression the test would fail reporting `fopen(/is-not-writeable/file): failed to open stream: No such file or directory`.

Testing Output

Sometimes you want to assert that the execution of a method, for instance, generates an expected output (via `echo` or `print`, for example). The `PHPUnit\Framework\TestCase` class uses PHP's Output Buffering [http://www.php.net/manual/en/ref.outcontrol.php] feature to provide the functionality that is necessary for this.

Example 2.14, “Testing the output of a function or method” shows how to use the `expectOutputString()` method to set the expected output. If this expected output is not generated, the test will be counted as a failure.

Example 2.14. Testing the output of a function or method

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

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

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

```
}
?>
```

```
phpunit OutputTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

.F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

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


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

Table 2.1, “Methods for testing output” shows the methods provided for testing output

Table 2.1. Methods for testing output

Method	Meaning
`void expectOutputRegex(string $regularExpression)`	Set up the expectation that the output matches a `$regularExpression`.
`void expectOutputString(string $expectedString)`	Set up the expectation that the output is equal to an `$expectedString`.
`bool setOutputCallback(callable $callback)`	Sets up a callback that is used to, for instance, normalize the actual output.

Note

A test that emits output will fail in strict mode.

Error output

Whenever a test fails PHPUnit tries its best to provide you with as much context as possible that can help to identify the problem.

Example 2.15. Error output generated when an array comparison fails

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

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

```
?>
```

```
phpunit ArrayDiffTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

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

/home/sb/ArrayDiffTest.php:7

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

In this example only one of the array values differs and the other values are shown to provide context on where the error occurred.

When the generated output would be long to read PHPUnit will split it up and provide a few lines of context around every difference.

Example 2.16. Error output when an array comparison of an long array fails

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

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

```
phpunit LongArrayDiffTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:
```

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


/home/sb/LongArrayDiffTest.php:7

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

Edge cases

When a comparison fails PHPUnit creates textual representations of the input values and compares those. Due to that implementation a diff might show more problems than actually exist.

This only happens when using assertEquals or other 'weak' comparison functions on arrays or objects.

Example 2.17. Edge case in the diff generation when using weak comparison

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

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

```
phpunit ArrayWeakComparisonTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

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

```
     3 => 4
     4 => 5
     5 => 6
 )


/home/sb/ArrayWeakComparisonTest.php:7

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

In this example the difference in the first index between `1` and `'1'` is reported even though assertEquals considers the values as a match.

Chapter 3. The Command-Line Test Runner

The PHPUnit command-line test runner can be invoked through the `phpunit` command. The following code shows how to run tests with the PHPUnit command-line test runner:

```
phpunit ArrayTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

..

Time: 0 seconds


OK (2 tests, 2 assertions)
```

When invoked as shown above, the PHPUnit command-line test runner will look for a `ArrayTest.php` sourcefile in the current working directory, load it, and expect to find a `ArrayTest` test case class. It will then execute the tests of that class.

For each test run, the PHPUnit command-line tool prints one character to indicate progress:

`.` Printed when the test succeeds.

`F` Printed when an assertion fails while running the test method.

`E` Printed when an error occurs while running the test method.

`R` Printed when the test has been marked as risky (see Chapter 6, *Risky Tests*).

`S` Printed when the test has been skipped (see Chapter 7, *Incomplete and Skipped Tests*).

`I` Printed when the test is marked as being incomplete or not yet implemented (see Chapter 7, *Incomplete and Skipped Tests*).

PHPUnit distinguishes between *failures* and *errors*. A failure is a violated PHPUnit assertion such as a failing `assertEquals()` call. An error is an unexpected exception or a PHP error. Sometimes this distinction proves useful since errors tend to be easier to fix than failures. If you have a big list of problems, it is best to tackle the errors first and see if you have any failures left when they are all fixed.

Command-Line Options

Let's take a look at the command-line test runner's options in the following code:

```
phpunit --help
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

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

Code Coverage Options:

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

```
Logging Options:

  --log-junit <file>        Log test execution in JUnit XML format to file.
  --log-tap <file>          Log test execution in TAP format to file.
  --log-json <file>         Log test execution in JSON format.
  --testdox-html <file>     Write agile documentation in HTML format to file.
  --testdox-text <file>     Write agile documentation in Text format to file.

Test Selection Options:

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

Test Execution Options:

  --report-useless-tests    Be strict about tests that do not test anything.
  --strict-coverage         Be strict about unintentionally covered code.
  --strict-global-state     Be strict about changes to global state
  --disallow-test-output    Be strict about output during tests.
  --enforce-time-limit      Enforce time limit based on test size.
  --disallow-todo-tests     Disallow @todo-annotated tests.

  --process-isolation       Run each test in a separate PHP process.
  --no-globals-backup       Do not backup and restore $GLOBALS for each test.
  --static-backup           Backup and restore static attributes for each test.

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

  --loader <loader>         TestSuiteLoader implementation to use.
  --repeat <times>          Runs the test(s) repeatedly.
  --tap                     Report test execution progress in TAP format.
  --testdox                 Report test execution progress in TestDox format.
  --printer <printer>       TestListener implementation to use.

Configuration Options:

  --bootstrap <file>        A "bootstrap" PHP file that is run before the tests.
  -c|--configuration <file> Read configuration from XML file.
  --no-configuration        Ignore default configuration file (phpunit.xml).
  --include-path <path(s)>  Prepend PHP's include_path with given path(s).
  -d key[=value]            Sets a php.ini value.

Miscellaneous Options:

  -h|--help                 Prints this usage information.
  --version                 Prints the version and exits.
```

`phpunit UnitTest`	Runs the tests that are provided by the class `UnitTest`. This class is expected to be declared in the `UnitTest.php` sourcefile. `UnitTest` must be either a class that inherits from `PHPUnit\Framework\TestCase` or a class that provides a `public static suite()` method which returns a `PHPUnit_Framework_Test` object, for example an instance of the `PHPUnit_Framework_TestSuite` class.
`phpunit UnitTest UnitTest.php`	Runs the tests that are provided by the class `UnitTest`. This class is expected to be declared in the specified sourcefile.
`--coverage-clover`	Generates a logfile in XML format with the code coverage information for the tests run. See Chapter 13, *Logging* for more details. Please note that this functionality is only available when the tokenizer and Xdebug extensions are installed.
`--coverage-crap4j`	Generates a code coverage report in Crap4j format. See Chapter 11, *Code Coverage Analysis* for more details. Please note that this functionality is only available when the tokenizer and Xdebug extensions are installed.
`--coverage-html`	Generates a code coverage report in HTML format. See Chapter 11, *Code Coverage Analysis* for more details. Please note that this functionality is only available when the tokenizer and Xdebug extensions are installed.
`--coverage-php`	Generates a serialized PHP_CodeCoverage object with the code coverage information. Please note that this functionality is only available when the tokenizer and Xdebug extensions are installed.
`--coverage-text`	Generates a logfile or command-line output in human readable format with the code coverage information for the tests run. See Chapter 13, *Logging* for more details. Please note that this functionality is only available when the tokenizer and Xdebug extensions are installed.
`--log-junit`	Generates a logfile in JUnit XML format for the tests run. See Chapter 13, *Logging* for more details.
`--log-tap`	Generates a logfile using the Test Anything Protocol (TAP) [http://testanything.org/] format for the tests run. See Chapter 13, *Logging* for more details.
`--log-json`	Generates a logfile using the JSON [http://www.json.org/] format. See Chapter 13, *Logging* for more details.
`--testdox-html` and `--testdox-text`	Generates agile documentation in HTML or plain text format for the tests that are run. See Chapter 12, *Other Uses for Tests* for more details.
`--filter`	Only runs tests whose name matches the given regular expression pattern. If the pattern is not enclosed in delimiters, PHPUnit will enclose the pattern in `/` delimiters.

The test names to match will be in one of the following formats:

`TestNamespace\TestCaseClass::testMethod`	The default test name format is the equivalent of using the `__METHOD__` magic constant inside the test method.
`TestNamespace\TestCaseClass::testMethod with data set #0`	When a test has a data provider, each iteration of the data gets the current index appended to the end of the default test name.
`TestNamespace\TestCaseClass::testMethod with data set "my named data"`	When a test has a data provider that uses named sets, each iteration of the data gets the current name appended to the end of the default test name. See Example 3.1, “Named data sets” for an example of named data sets.

Example 3.1 Named data sets

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

namespace TestNamespace;

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

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

`/path/to/my/test.phpt`	The test name for a PHPT test is the filesystem path.

See Example 3.2, “Filter pattern examples” for examples of valid filter patterns.

Example 3.2. Filter pattern examples

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

See Example 3.3, “Filter shortcuts” for some additional shortcuts that are available for matching data providers.

Example 3.3. Filter shortcuts

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

`--testsuite`	Only runs the test suite whose name matches the given pattern.
`--group`	Only runs tests from the specified group(s). A test can be tagged as belonging to a group using the `@group` annotation. The `@author` annotation is an alias for `@group` allowing to filter tests based on their authors.
`--exclude-group`	Exclude tests from the specified group(s). A test can be tagged as belonging to a group using the `@group` annotation.
`--list-groups`	List available test groups.
`--test-suffix`	Only search for test files with specified suffix(es).
`--report-useless-tests`	Be strict about tests that do not test anything. See Chapter 6, *Risky Tests* for details.

`--strict-coverage`	Be strict about unintentionally covered code. See Chapter 6, *Risky Tests* for details.
`--strict-global-state`	Be strict about global state manipulation. See Chapter 6, *Risky Tests* for details.
`--disallow-test-output`	Be strict about output during tests. See Chapter 6, *Risky Tests* for details.
`--disallow-todo-tests`	Does not execute tests which have the `@todo` annotation in its docblock.
`--enforce-time-limit`	Enforce time limit based on test size. See Chapter 6, *Risky Tests* for details.
`--process-isolation`	Run each test in a separate PHP process.
`--no-globals-backup`	Do not backup and restore $GLOBALS. See the section called “Global State” for more details.
`--static-backup`	Backup and restore static attributes of user-defined classes. See the section called “Global State” for more details.
`--colors`	Use colors in output. On Windows, use ANSICON [https://github.com/adoxa/ansicon] or ConEmu [https://github.com/Maximus5/ConEmu]. There are three possible values for this option: • `never`: never displays colors in the output. This is the default value when `--colors` option is not used. • `auto`: displays colors in the output unless the current terminal doesn't supports colors, or if the output is piped to a command or redirected to a file. • `always`: always displays colors in the output even when the current terminal doesn't supports colors, or when the output is piped to a command or redirected to a file. When `--colors` is used without any value, `auto` is the chosen value.
`--columns`	Defines the number of columns to use for progress output. If `max` is defined as value, the number of columns will be maximum of the current terminal.
`--stderr`	Optionally print to `STDERR` instead of `STDOUT`.
`--stop-on-error`	Stop execution upon first error.
`--stop-on-failure`	Stop execution upon first error or failure.
`--stop-on-risky`	Stop execution upon first risky test.
`--stop-on-skipped`	Stop execution upon first skipped test.
`--stop-on-incomplete`	Stop execution upon first incomplete test.
`--verbose`	Output more verbose information, for instance the names of tests that were incomplete or have been skipped.
`--debug`	Output debug information such as the name of a test when its execution starts.

`--loader`	Specifies the `PHPUnit_Runner_TestSuiteLoader` implementation to use. The standard test suite loader will look for the sourcefile in the current working directory and in each directory that is specified in PHP's `include_path` configuration directive. A class name such as `Project_Package_Class` is mapped to the source filename `Project/Package/Class.php`.
`--repeat`	Repeatedly runs the test(s) the specified number of times.
`--tap`	Reports the test progress using the Test Anything Protocol (TAP) [http://testanything.org/]. See Chapter 13, *Logging* for more details.
`--testdox`	Reports the test progress as agile documentation. See Chapter 12, *Other Uses for Tests* for more details.
`--printer`	Specifies the result printer to use. The printer class must extend `PHPUnit_Util_Printer` and implement the `PHPUnit_Framework_TestListener` interface.
`--bootstrap`	A "bootstrap" PHP file that is run before the tests.
`--configuration`, `-c`	Read configuration from XML file. See Appendix C, *The XML Configuration File* for more details. If `phpunit.xml` or `phpunit.xml.dist` (in that order) exist in the current working directory and `--configuration` is *not* used, the configuration will be automatically read from that file.
`--no-configuration`	Ignore `phpunit.xml` and `phpunit.xml.dist` from the current working directory.
`--include-path`	Prepend PHP's `include_path` with given path(s).
`-d`	Sets the value of the given PHP configuration option.

Note

Please note that as of 4.8, options can be put after the argument(s).

Chapter 4. Fixtures

One of the most time-consuming parts of writing tests is writing the code to set the world up in a known state and then return it to its original state when the test is complete. This known state is called the *fixture* of the test.

In Example 2.1, “Testing array operations with PHPUnit”, the fixture was simply the array that is stored in the `$stack` variable. Most of the time, though, the fixture will be more complex than a simple array, and the amount of code needed to set it up will grow accordingly. The actual content of the test gets lost in the noise of setting up the fixture. This problem gets even worse when you write several tests with similar fixtures. Without some help from the testing framework, we would have to duplicate the code that sets up the fixture for each test we write.

PHPUnit supports sharing the setup code. Before a test method is run, a template method called `setUp()` is invoked. `setUp()` is where you create the objects against which you will test. Once the test method has finished running, whether it succeeded or failed, another template method called `tearDown()` is invoked. `tearDown()` is where you clean up the objects against which you tested.

In Example 2.2, “Using the `@depends` annotation to express dependencies” we used the producer-consumer relationship between tests to share a fixture. This is not always desired or even possible. Example 4.1, “Using setUp() to create the stack fixture” shows how we can write the tests of the `StackTest` in such a way that not the fixture itself is reused but the code that creates it. First we declare the instance variable, `$stack`, that we are going to use instead of a method-local variable. Then we put the creation of the `array` fixture into the `setUp()` method. Finally, we remove the redundant code from the test methods and use the newly introduced instance variable, `$this->stack`, instead of the method-local variable `$stack` with the `assertEquals()` assertion method.

Example 4.1. Using setUp() to create the stack fixture

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

class StackTest extends TestCase
{
    protected $stack;

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

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

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

    public function testPop()
    {
        array_push($this->stack, 'foo');
        $this->assertEquals('foo', array_pop($this->stack));
        $this->assertTrue(empty($this->stack));
    }
}
```

```
?>
```

The `setUp()` and `tearDown()` template methods are run once for each test method (and on fresh instances) of the test case class.

In addition, the `setUpBeforeClass()` and `tearDownAfterClass()` template methods are called before the first test of the test case class is run and after the last test of the test case class is run, respectively.

The example below shows all template methods that are available in a test case class.

Example 4.2. Example showing all template methods available

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

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

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

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

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

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

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

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

    public static function tearDownAfterClass()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

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

```
        throw $e;
    }
}
?>
```

```
phpunit TemplateMethodsTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

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


Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

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

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

More setUp() than tearDown()

`setUp()` and `tearDown()` are nicely symmetrical in theory but not in practice. In practice, you only need to implement `tearDown()` if you have allocated external resources like files or sockets in `setUp()`. If your `setUp()` just creates plain PHP objects, you can generally ignore `tearDown()`. However, if you create many objects in your `setUp()`, you might want to `unset()` the variables pointing to those objects in your `tearDown()` so they can be garbage collected. The garbage collection of test case objects is not predictable.

Variations

What happens when you have two tests with slightly different setups? There are two possibilities:

- If the `setUp()` code differs only slightly, move the code that differs from the `setUp()` code to the test method.
- If you really have a different `setUp()`, you need a different test case class. Name the class after the difference in the setup.

Sharing Fixture

There are few good reasons to share fixtures between tests, but in most cases the need to share a fixture between tests stems from an unresolved design problem.

A good example of a fixture that makes sense to share across several tests is a database connection: you log into the database once and reuse the database connection instead of creating a new connection for each test. This makes your tests run faster.

Example 4.3, “Sharing fixture between the tests of a test suite” uses the `setUpBeforeClass()` and `tearDownAfterClass()` template methods to connect to the database before the test case class' first test and to disconnect from the database after the last test of the test case, respectively.

Example 4.3. Sharing fixture between the tests of a test suite

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

class DatabaseTest extends TestCase
{
    protected static $dbh;

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

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

It cannot be emphasized enough that sharing fixtures between tests reduces the value of the tests. The underlying design problem is that objects are not loosely coupled. You will achieve better results solving the underlying design problem and then writing tests using stubs (see Chapter 9, *Test Doubles*), than by creating dependencies between tests at runtime and ignoring the opportunity to improve your design.

Global State

It is hard to test code that uses singletons. [http://googletesting.blogspot.com/2008/05/tott-using-dependancy-injection-to.html] The same is true for code that uses global variables. Typically, the code you want to test is coupled strongly with a global variable and you cannot control its creation. An additional problem is the fact that one test's change to a global variable might break another test.

In PHP, global variables work like this:

- A global variable `$foo = 'bar';` is stored as `$GLOBALS['foo'] = 'bar';`.
- The `$GLOBALS` variable is a so-called *super-global* variable.
- Super-global variables are built-in variables that are always available in all scopes.
- In the scope of a function or method, you may access the global variable `$foo` by either directly accessing `$GLOBALS['foo']` or by using `global $foo;` to create a local variable with a reference to the global variable.

Besides global variables, static attributes of classes are also part of the global state.

By default, PHPUnit runs your tests in a way where changes to global and super-global variables (`$GLOBALS`, `$_ENV`, `$_POST`, `$_GET`, `$_COOKIE`, `$_SERVER`, `$_FILES`, `$_REQUEST`) do not affect other tests. Optionally, this isolation can be extended to static attributes of classes.

Note

The backup and restore operations for global variables and static class attributes use `serialize()` and `unserialize()`.

Objects of some classes (e.g., `PDO`) cannot be serialized and the backup operation will break when such an object is stored e.g. in the `$GLOBALS` array.

The `@backupGlobals` annotation that is discussed in the section called "@backupGlobals" can be used to control the backup and restore operations for global variables. Alternatively, you can provide a blacklist of global variables that are to be excluded from the backup and restore operations like this

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

    // ...
}
```

Note

Setting the `$backupGlobalsBlacklist` property inside e.g. the `setUp()` method has no effect.

The `@backupStaticAttributes` annotation discussed in the section called "@backupStaticAttributes" can be used to back up all static property values in all declared classes before each test and restore them afterwards.

It processes all classes that are declared at the time a test starts, not only the test class itself. It only applies to static class properties, not static variables within functions.

Note

The `@backupStaticAttributes` operation is executed before a test method, but only if it is enabled. If a static value was changed by a previously executed test that did not have `@backupStaticAttributes` enabled, then that value will be backed up and restored — not the originally declared default value. PHP does not record the originally declared default value of any static variable.

The same applies to static properties of classes that were newly loaded/declared within a test. They cannot be reset to their originally declared default value after the test, since that value is unknown. Whichever value is set will leak into subsequent tests.

For unit tests, it is recommended to explicitly reset the values of static properties under test in your `setUp()` code instead (and ideally also `tearDown()`, so as to not affect subsequently executed tests).

You can provide a blacklist of static attributes that are to be excluded from the backup and restore operations:

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

    // ...
}
```

Note

Setting the `$backupStaticAttributesBlacklist` property inside e.g. the `setUp()` method has no effect.

Chapter 5. Organizing Tests

One of the goals of PHPUnit is that tests should be composable: we want to be able to run any number or combination of tests together, for instance all tests for the whole project, or the tests for all classes of a component that is part of the project, or just the tests for a single class.

PHPUnit supports different ways of organizing tests and composing them into a test suite. This chapter shows the most commonly used approaches.

Composing a Test Suite Using the Filesystem

Probably the easiest way to compose a test suite is to keep all test case source files in a test directory. PHPUnit can automatically discover and run the tests by recursively traversing the test directory.

Lets take a look at the test suite of the sebastianbergmann/money [http://github.com/sebastianbergmann/money/] library. Looking at this project's directory structure, we see that the test case classes in the `tests` directory mirror the package and class structure of the System Under Test (SUT) in the `src` directory:

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

To run all tests for the library we just need to point the PHPUnit command-line test runner to the test directory:

```
phpunit --bootstrap src/autoload.php tests
PHPUnit 5.5.0 by Sebastian Bergmann.

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

Time: 636 ms, Memory: 3.50Mb

OK (33 tests, 52 assertions)
```

Note

If you point the PHPUnit command-line test runner to a directory it will look for `*Test.php` files.

To run only the tests that are declared in the `CurrencyTest` test case class in `tests/CurrencyTest.php` we can use the following command:

```
phpunit --bootstrap src/autoload.php tests/CurrencyTest
PHPUnit 5.5.0 by Sebastian Bergmann.

........

Time: 280 ms, Memory: 2.75Mb

OK (8 tests, 8 assertions)
```

For more fine-grained control of which tests to run we can use the `--filter` option:

```
phpunit --bootstrap src/autoload.php --filter testObjectCanBeConstructedForValidConstruc
PHPUnit 5.5.0 by Sebastian Bergmann.
```

```
..

Time: 167 ms, Memory: 3.00Mb

OK (2 test, 2 assertions)
```

Note

A drawback of this approach is that we have no control over the order in which the tests are run. This can lead to problems with regard to test dependencies, see the section called “Test Dependencies”. In the next section you will see how you can make the test execution order explicit by using the XML configuration file.

Composing a Test Suite Using XML Configuration

PHPUnit's XML configuration file (Appendix C, *The XML Configuration File*) can also be used to compose a test suite. Example 5.1, “Composing a Test Suite Using XML Configuration” shows a minimal `phpunit.xml` file that will add all `*Test` classes that are found in `*Test.php` files when the `tests` directory is recursively traversed.

Example 5.1. Composing a Test Suite Using XML Configuration

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

If `phpunit.xml` or `phpunit.xml.dist` (in that order) exist in the current working directory and `--configuration` is *not* used, the configuration will be automatically read from that file.

The order in which tests are executed can be made explicit:

Example 5.2. Composing a Test Suite Using XML Configuration

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

Chapter 6. Risky Tests

PHPUnit can perform the additional checks documented below while it executes the tests.

Useless Tests

PHPUnit can be strict about tests that do not test anything. This check can be enabled by using the `--report-useless-tests` option on the commandline or by setting `beStrictAboutTestsThatDoNotTestAnything="true"` in PHPUnit's XML configuration file.

A test that does not perform an assertion will be marked as risky when this check is enabled. Expectations on mock objects or annotations such as `@expectedException` count as an assertion.

Unintentionally Covered Code

PHPUnit can be strict about unintentionally covered code. This check can be enabled by using the `--strict-coverage` option on the commandline or by setting `checkForUnintentionallyCoveredCode="true"` in PHPUnit's XML configuration file.

A test that is annotated with `@covers` and executes code that is not listed using a `@covers` or `@uses` annotation will be marked as risky when this check is enabled.

Output During Test Execution

PHPUnit can be strict about output during tests. This check can be enabled by using the `--disallow-test-output` option on the commandline or by setting `beStrictAboutOutputDuringTests="true"` in PHPUnit's XML configuration file.

A test that emits output, for instance by invoking `print` in either the test code or the tested code, will be marked as risky when this check is enabled.

Test Execution Timeout

A time limit can be enforced for the execution of a test if the `PHP_Invoker` package is installed and the `pcntl` extension is available. The enforcing of this time limit can be enabled by using the `--enforce-time-limit` option on the commandline or by setting `beStrictAboutTestSize="true"` in PHPUnit's XML configuration file.

A test annotated with `@large` will fail if it takes longer than 60 seconds to execute. This timeout is configurable via the `timeoutForLargeTests` attribute in the XML configuration file.

A test annotated with `@medium` will fail if it takes longer than 10 seconds to execute. This timeout is configurable via the `timeoutForMediumTests` attribute in the XML configuration file.

A test that is not annotated with `@medium` or `@large` will be treated as if it were annotated with `@small`. A small test will fail if it takes longer than 1 second to execute. This timeout is configurable via the `timeoutForSmallTests` attribute in the XML configuration file.

Global State Manipulation

PHPUnit can be strict about tests that manipulate global state. This check can be enabled by using the `--strict-global-state` option on the commandline or by setting `beStrictAboutChangesToGlobalState="true"` in PHPUnit's XML configuration file.

Chapter 7. Incomplete and Skipped Tests

Incomplete Tests

When you are working on a new test case class, you might want to begin by writing empty test methods such as:

```
public function testSomething()
{
}
```

to keep track of the tests that you have to write. The problem with empty test methods is that they are interpreted as a success by the PHPUnit framework. This misinterpretation leads to the test reports being useless -- you cannot see whether a test is actually successful or just not yet implemented. Calling `$this->fail()` in the unimplemented test method does not help either, since then the test will be interpreted as a failure. This would be just as wrong as interpreting an unimplemented test as a success.

If we think of a successful test as a green light and a test failure as a red light, we need an additional yellow light to mark a test as being incomplete or not yet implemented. `PHPUnit_Framework_IncompleteTest` is a marker interface for marking an exception that is raised by a test method as the result of the test being incomplete or currently not implemented. `PHPUnit_Framework_IncompleteTestError` is the standard implementation of this interface.

Example 7.1, “Marking a test as incomplete” shows a test case class, `SampleTest`, that contains one test method, `testSomething()`. By calling the convenience method `markTestIncomplete()` (which automatically raises an `PHPUnit_Framework_IncompleteTestError` exception) in the test method, we mark the test as being incomplete.

Example 7.1. Marking a test as incomplete

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

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

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

An incomplete test is denoted by an `I` in the output of the PHPUnit command-line test runner, as shown in the following example:

```
phpunit --verbose SampleTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

I
```

```
Time: 0 seconds, Memory: 3.95Mb

There was 1 incomplete test:

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

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

Table 7.1, “API for Incomplete Tests” shows the API for marking tests as incomplete.

Table 7.1. API for Incomplete Tests

Method	Meaning
`void markTestIncomplete()`	Marks the current test as incomplete.
`void markTestIncomplete(string $message)`	Marks the current test as incomplete using `$message` as an explanatory message.

Skipping Tests

Not all tests can be run in every environment. Consider, for instance, a database abstraction layer that has several drivers for the different database systems it supports. The tests for the MySQL driver can of course only be run if a MySQL server is available.

Example 7.2, “Skipping a test” shows a test case class, `DatabaseTest`, that contains one test method, `testConnection()`. In the test case class' `setUp()` template method we check whether the MySQLi extension is available and use the `markTestSkipped()` method to skip the test if it is not.

Example 7.2. Skipping a test

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

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

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

A test that has been skipped is denoted by an S in the output of the PHPUnit command-line test runner, as shown in the following example:

```
phpunit --verbose DatabaseTest
```

```
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

S

Time: 0 seconds, Memory: 3.95Mb

There was 1 skipped test:

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

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

Table 7.2, “API for Skipping Tests” shows the API for skipping tests.

Table 7.2. API for Skipping Tests

Method	Meaning
`void markTestSkipped()`	Marks the current test as skipped.
`void markTestSkipped(string $message)`	Marks the current test as skipped using `$message` as an explanatory message.

Skipping Tests using @requires

In addition to the above methods it is also possible to use the `@requires` annotation to express common preconditions for a test case.

Table 7.3. Possible @requires usages

Type	Possible Values	Examples	Another example
`PHP`	Any PHP version identifier	@requires PHP 5.3.3	@requires PHP 7.1-dev
`PHPUnit`	Any PHPUnit version identifier	@requires PHPUnit 3.6.3	@requires PHPUnit 4.6
`OS`	A regexp matching PHP_OS [http://php.net/manual/en/reserved.constants.php#constant.php-os]	@requires OS Linux	@requires OS WIN32|WINNT
`function`	Any valid parameter to function_exists [http://php.net/function_exists]	@requires function imap_open	@requires function ReflectionMethod::setAccessible
`extension`	Any extension name along with an optional version identifier	@requires extension mysqli	@requires extension redis 2.2.0

Example 7.3. Skipping test cases using @requires

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

/**
 * @requires extension mysqli
```

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

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

If you are using syntax that doesn't compile with a certain PHP Version look into the xml configuration for version dependent includes in the section called “Test Suites”

Chapter 8. Database Testing

Many beginner and intermediate unit testing examples in any programming language suggest that it is perfectly easy to test your application's logic with simple tests. For database-centric applications this is far away from the reality. Start using Wordpress, TYPO3 or Symfony with Doctrine or Propel, for example, and you will easily experience considerable problems with PHPUnit: just because the database is so tightly coupled to these libraries.

> **Note**
>
> Make sure you have the PHP extension `pdo` and database specific extensions such as `pdo_mysql` installed. Otherwise the examples shown below will not work.

You probably know this scenario from your daily work and projects, where you want to put your fresh or experienced PHPUnit skills to work and get stuck by one of the following problems:

1. The method you want to test executes a rather large JOIN operation and uses the data to calculate some important results.

2. Your business logic performs a mix of SELECT, INSERT, UPDATE and DELETE statements.

3. You need to setup test data in (possibly much) more than two tables to get reasonable initial data for the methods you want to test.

The DbUnit extension considerably simplifies the setup of a database for testing purposes and allows you to verify the contents of a database after performing a series of operations.

Supported Vendors for Database Testing

DbUnit currently supports MySQL, PostgreSQL, Oracle and SQLite. Through Zend Framework [http://framework.zend.com] or Doctrine 2 [http://www.doctrine-project.org] integrations it has access to other database systems such as IBM DB2 or Microsoft SQL Server.

Difficulties in Database Testing

There is a good reason why all the examples on unit testing do not include interactions with the database: these kind of tests are both complex to setup and maintain. While testing against your database you need to take care of the following variables:

- The database schema and tables
- Inserting the rows required for the test into these tables
- Verifying the state of the database after your test has run
- Cleanup the database for each new test

Because many database APIs such as PDO, MySQLi or OCI8 are cumbersome to use and verbose in writing doing these steps manually is an absolute nightmare.

Test code should be as short and precise as possible for several reasons:

- You do not want to modify considerable amount of test code for little changes in your production code.
- You want to be able to read and understand the test code easily, even months after writing it.

Additionally you have to realize that the database is essentially a global input variable to your code. Two tests in your test suite could run against the same database, possibly reusing data multiple times. Failures in one test can easily affect the result of the following tests making your testing experience very difficult. The previously mentioned cleanup step is of major importance to solve the “database is a global input” problem.

DbUnit helps to simplify all these problems with database testing in an elegant way.

What PHPUnit cannot help you with is the fact that database tests are very slow compared to tests not using the database. Depending on how large the interactions with your database are your tests could run a considerable amount of time. However, if you keep the amount of data used for each test small and try to test as much code using non-database tests you can easily get away in under a minute even for large test suites.

The Doctrine 2 project [http://www.doctrine-project.org]'s test suite, for example, currently has a test suite of about 1000 tests where nearly half of them accesses the database and still runs in 15 seconds against a MySQL database on a standard desktop computer.

The four stages of a database test

In his book on xUnit Test Patterns Gerard Meszaros lists the four stages of a unit-test:

1. Set up fixture
2. Exercise System Under Test
3. Verify outcome
4. Teardown

> **What is a Fixture?**
>
> A fixture describes the initial state your application and database are in when you execute a test.

Testing the database requires you to hook into at least the setup and teardown to clean-up and write the required fixture data into your tables. However, the database extension has good reason to revert the four stages in a database test to resemble the following workflow that is executed for each single test:

1. Clean-Up Database

Since there is always a first test that runs against the database you do not know exactly if there is already data in the tables. PHPUnit will execute a TRUNCATE against all the tables you specified to reset their status to empty.

2. Set up fixture

PHPUnit will then iterate over all the fixture rows specified and insert them into their respective tables.

3–5. Run Test, Verify outcome and Teardown

After the database is reset and loaded with its initial state the actual test is executed by PHPUnit. This part of the test code does not require awareness of the Database Extension at all, you can go on and test whatever you like with your code.

In your test use a special assertion called `assertDataSetsEqual()` for verification purposes, however, this is entirely optional. This feature will be explained in the section “Database Assertions”.

Configuration of a PHPUnit Database Test-Case

Usually when using PHPUnit your testcases would extend the `PHPUnit\Framework\TestCase` class in the following way:

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

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

If you want to test code that works with the Database Extension the setup is a bit more complex and you have to extend a different abstract TestCase requiring you to implement two abstract methods `getConnection()` and `getDataSet()`:

```
<?php
class MyGuestbookTest extends PHPUnit_Extensions_Database_TestCase
{
    /**
     * @return PHPUnit_Extensions_Database_DB_IDatabaseConnection
     */
    public function getConnection()
    {
        $pdo = new PDO('sqlite::memory:');
        return $this->createDefaultDBConnection($pdo, ':memory:');
    }

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

Implementing getConnection()

To allow the clean-up and fixture loading functionalities to work the PHPUnit Database Extension requires access to a database connection abstracted across vendors through the PDO library. It is important to note that your application does not need to be based on PDO to use PHPUnit's database extension, the connection is merely used for the clean-up and fixture setup.

In the previous example we create an in-memory Sqlite connection and pass it to the `createDefaultDBConnection` method which wraps the PDO instance and the second parameter (the database-name) in a very simple abstraction layer for database connections of the type `PHPUnit_Extensions_Database_DB_IDatabaseConnection`.

The section “Using the Database Connection” explains the API of this interface and how you can make the best use of it.

Implementing getDataSet()

The `getDataSet()` method defines how the initial state of the database should look before each test is executed. The state of a database is abstracted through the concepts DataSet and DataTable both being represented by the interfaces `PHPUnit_Extensions_Database_DataSet_IDataSet` and `PHPUnit_Extensions_Database_DataSet_IDataTable`. The next section will describe in detail how these concepts work and what the benefits are for using them in database testing.

For the implementation we only need to know that the `getDataSet()` method is called once during `setUp()` to retrieve the fixture data-set and insert it into the database. In the example we are using a factory method `createFlatXMLDataSet($filename)` that represents a data-set through an XML representation.

What about the Database Schema (DDL)?

PHPUnit assumes that the database schema with all its tables, triggers, sequences and views is created before a test is run. This means you as developer have to make sure that the database is correctly setup before running the suite.

There are several means to achieve this pre-condition to database testing.

1. If you are using a persistent database (not Sqlite Memory) you can easily setup the database once with tools such as phpMyAdmin for MySQL and re-use the database for every test-run.

2. If you are using libraries such as Doctrine 2 [http://www.doctrine-project.org] or Propel [http://www.propelorm.org/] you can use their APIs to create the database schema you need once before you run the tests. You can utilize PHPUnit's Bootstrap and Configuration [textui.html] capabilities to execute this code whenever your tests are run.

Tip: Use your own Abstract Database TestCase

From the previous implementation example you can easily see that `getConnection()` method is pretty static and could be re-used in different database test-cases. Additionally to keep performance of your tests good and database overhead low you can refactor the code a little bit to get a generic abstract test case for your application, which still allows you to specify a different data-fixture for each test case:

```
<?php
abstract class MyApp_Tests_DatabaseTestCase extends PHPUnit_Extensions_Database_TestCase
{
    // only instantiate pdo once for test clean-up/fixture load
    static private $pdo = null;

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

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

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

This has the database connection hardcoded in the PDO connection though. PHPUnit has another awesome feature that could make this testcase even more generic. If you use the XML Configuration [appendixes.configuration.html#appendixes.configuration.php-ini-constants-variables] you could make the database connection configurable per test-run. First let's create a “phpunit.xml” file in our tests/ directory of the application that looks like:

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

We can now modify our test-case to look like:

```
<?php
abstract class Generic_Tests_DatabaseTestCase extends PHPUnit_Extensions_Database_TestCa
{
    // only instantiate pdo once for test clean-up/fixture load
    static private $pdo = null;

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

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

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

We can now run the database test suite using different configurations from the command-line interface:

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

The possibility to run the database tests against different database targets easily is very important if you are developing on the development machine. If several developers run the database tests against the same database connection you can easily experience test-failures because of race-conditions.

Understanding DataSets and DataTables

A central concept of PHPUnit's Database Extension are DataSets and DataTables. You should try to understand this simple concept to master database testing with PHPUnit. The DataSet and DataTable are an abstraction layer around your database tables, rows and columns. A simple API hides the underlying database contents in an object structure, which can also be implemented by other non-database sources.

This abstraction is necessary to compare the actual contents of a database against the expected contents. Expectations can be represented as XML, YAML, CSV files or PHP array for example. The DataSet

and DataTable interfaces enable the comparison of these conceptually different sources, emulating relational database storage in a semantically similar approach.

A workflow for database assertions in your tests then consists of three simple steps:

- Specify one or more tables in your database by table name (actual dataset)
- Specify the expected dataset in your preferred format (YAML, XML, ..)
- Assert that both dataset representations equal each other.

Assertions are not the only use-case for the DataSet and DataTable in PHPUnit's Database Extension. As shown in the previous section they also describe the initial contents of a database. You are forced to define a fixture dataset by the Database TestCase, which is then used to:

- Delete all the rows from the tables specified in the dataset.
- Write all the rows in the data-tables into the database.

Available Implementations

There are three different types of datasets/datatables:

- File-Based DataSets and DataTables
- Query-Based DataSet and DataTable
- Filter and Composition DataSets and DataTables

The file-based datasets and tables are generally used for the initial fixture and to describe the expected state of the database.

Flat XML DataSet

The most common dataset is called Flat XML. It is a very simple xml format where a tag inside the root node `<dataset>` represents exactly one row in the database. The tags name equals the table to insert the row into and an attribute represents the column. An example for a simple guestbook application could look like this:

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

This is obviously easy to write. Here `<guestbook>` is the table name where two rows are inserted into each with four columns “id”, “content”, “user” and “created” with their respective values.

However, this simplicity comes at a cost.

From the previous example it isn't obvious how you would specify an empty table. You can insert a tag with no attributes with the name of the empty table. A flat xml file for an empty guestbook table would then look like:

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

The handling of NULL values with the flat xml dataset is tedious. A NULL value is different than an empty string value in almost any database (Oracle being an exception), something that is difficult to

describe in the flat xml format. You can represent a NULL's value by omitting the attribute from the row specification. If our guestbook would allow anonymous entries represented by a NULL value in the user column, a hypothetical state of the guestbook table could look like:

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

In this case the second entry is posted anonymously. However, this leads to a serious problem with column recognition. During dataset equality assertions each dataset has to specify what columns a table holds. If an attribute is NULL for all the rows of a data-table, how would the Database Extension know that the column should be part of the table?

The flat xml dataset makes a crucial assumption now, defining that the attributes on the first defined row of a table define the columns of this table. In the previous example this would mean “id”, “content”, “user” and “created” are columns of the guestbook table. For the second row where “user” is not defined a NULL would be inserted into the database.

When the first guestbook entry is deleted from the dataset only “id”, “content” and “created” would be columns of the guestbook table, since “user” is not specified.

To use the Flat XML dataset effectively when NULL values are relevant the first row of each table must not contain any NULL value and only successive rows are allowed to omit attributes. This can be awkward, since the order of the rows is a relevant factor for database assertions.

In turn, if you specify only a subset of the table columns in the Flat XML dataset all the omitted values are set to their default values. This will lead to errors if one of the omitted columns is defined as “NOT NULL DEFAULT NULL”.

In conclusion I can only advise using the Flat XML datasets if you do not need NULL values.

You can create a flat xml dataset instance from within your Database TestCase by calling the `createFlatXmlDataSet($filename)` method:

```
<?php
class MyTestCase extends PHPUnit_Extensions_Database_TestCase
{
    public function getDataSet()
    {
        return $this->createFlatXmlDataSet('myFlatXmlFixture.xml');
    }
}
?>
```

XML DataSet

There is another more structured XML dataset, which is a bit more verbose to write but avoids the NULL problems of the Flat XML dataset. Inside the root node `<dataset>` you can specify `<table>`, `<column>`, `<row>`, `<value>` and `<null />` tags. An equivalent dataset to the previously defined Guestbook Flat XML looks like:

```
<?xml version="1.0" ?>
<dataset>
    <table name="guestbook">
        <column>id</column>
        <column>content</column>
        <column>user</column>
        <column>created</column>
        <row>
```

```
            <value>1</value>
            <value>Hello buddy!</value>
            <value>joe</value>
            <value>2010-04-24 17:15:23</value>
        </row>
        <row>
            <value>2</value>
            <value>I like it!</value>
            <null />
            <value>2010-04-26 12:14:20</value>
        </row>
    </table>
</dataset>
```

Any defined `<table>` has a name and requires a definition of all the columns with their names. It can contain zero or any positive number of nested `<row>` elements. Defining no `<row>` element means the table is empty. The `<value>` and `<null />` tags have to be specified in the order of the previously given `<column>` elements. The `<null />` tag obviously means that the value is NULL.

You can create a xml dataset instance from within your Database TestCase by calling the `createXmlDataSet($filename)` method:

```
<?php
class MyTestCase extends PHPUnit_Extensions_Database_TestCase
{
    public function getDataSet()
    {
        return $this->createXMLDataSet('myXmlFixture.xml');
    }
}
?>
```

MySQL XML DataSet

This new XML format is specific to the MySQL database server [http://www.mysql.com]. Support for it was added in PHPUnit 3.5. Files in this format can be generated using the `mysqldump` [http://dev.mysql.com/doc/refman/5.0/en/mysqldump.html] utility. Unlike CSV datasets, which `mysqldump` also supports, a single file in this XML format can contain data for multiple tables. You can create a file in this format by invoking `mysqldump` like so:

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

This file can be used in your Database TestCase by calling the `createMySQLXMLDataSet($filename)` method:

```
<?php
class MyTestCase extends PHPUnit_Extensions_Database_TestCase
{
    public function getDataSet()
    {
        return $this->createMySQLXMLDataSet('/path/to/file.xml');
    }
}
?>
```

YAML DataSet

Alternatively, you can use YAML dataset for the guestbook example:

```
guestbook:
```

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

This is simple, convient AND it solves the NULL issue that the similar Flat XML dataset has. A NULL in YAML is just the column name without no value specified. An empty string is specified as `column1: ""`.

The YAML Dataset has no factory method on the Database TestCase currently, so you have to instantiate it manually:

```
<?php
class YamlGuestbookTest extends PHPUnit_Extensions_Database_TestCase
{
    protected function getDataSet()
    {
        return new PHPUnit_Extensions_Database_DataSet_YamlDataSet(
            dirname(__FILE__)."/_files/guestbook.yml"
        );
    }
}
?>
```

CSV DataSet

Another file-based dataset is based on CSV files. Each table of the dataset is represented as a single CSV file. For our guestbook example we would define a guestbook-table.csv file:

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

While this is very convenient for editing with Excel or OpenOffice, you cannot specify NULL values with the CSV dataset. An empty column will lead to the database default empty value being inserted into the column.

You can create a CSV DataSet by calling:

```
<?php
class CsvGuestbookTest extends PHPUnit_Extensions_Database_TestCase
{
    protected function getDataSet()
    {
        $dataSet = new PHPUnit_Extensions_Database_DataSet_CsvDataSet();
        $dataSet->addTable('guestbook', dirname(__FILE__)."/_files/guestbook.csv");
        return $dataSet;
    }
}
?>
```

Array DataSet

There is no Array based DataSet in PHPUnit's Database Extension (yet), but we can implement our own easily. Our guestbook example should look like:

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

A PHP DataSet has obvious advantages over all the other file-based datasets:

- PHP Arrays can obviously handle `NULL` values.
- You won't need additional files for assertions and can specify them directly in the TestCase.

For this dataset like the Flat XML, CSV and YAML DataSets the keys of the first specified row define the table's column names, in the previous case this would be “id”, “content”, “user” and “created”.

The implementation for this Array DataSet is simple and straightforward:

```
<?php
class MyApp_DbUnit_ArrayDataSet extends PHPUnit_Extensions_Database_DataSet_AbstractData
{
    /**
     * @var array
     */
    protected $tables = [];

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

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

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

```
        }
    }

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

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

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

Query (SQL) DataSet

For database assertions you do not only need the file-based datasets but also a Query/SQL based Dataset that contains the actual contents of the database. This is where the Query DataSet shines:

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

Adding a table just by name is an implicit way to define the data-table with the following query:

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

You can make use of this by specifying arbitrary queries for your tables, for example restricting rows, column or adding `ORDER BY` clauses:

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

The section on Database Assertions will show some more details on how to make use of the Query DataSet.

Database (DB) Dataset

Accessing the Test Connection you can automatically create a DataSet that consists of all the tables with their content in the database specified as second parameter to the Connections Factory method.

You can either create a dataset for the complete database as shown in `testGuestbook()`, or restrict it to a set of specified table names with a whitelist as shown in `testFilteredGuestbook()` method.

```
<?php
class MySqlGuestbookTest extends PHPUnit_Extensions_Database_TestCase
{
    /**
```

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

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

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

Replacement DataSet

I have been talking about NULL problems with the Flat XML and CSV DataSet, but there is a slightly complicated workaround to get both types of datasets working with NULLs.

The Replacement DataSet is a decorator for an existing dataset and allows you to replace values in any column of the dataset by another replacement value. To get our guestbook example working with NULL values we specify the file like:

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

We then wrap the Flat XML DataSet into a Replacement DataSet:

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

DataSet Filter

If you have a large fixture file you can use the DataSet Filter for white- and blacklisting of tables and columns that should be contained in a sub-dataset. This is especially handy in combination with the DB DataSet to filter the columns of the datasets.

```
<?php
class DataSetFilterTest extends PHPUnit_Extensions_Database_TestCase
{
    public function testIncludeFilteredGuestbook()
    {
        $tableNames = ['guestbook'];
        $dataSet = $this->getConnection()->createDataSet();

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

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

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

NOTE You cannot use both exclude and include column filtering on the same table, only on different ones. Plus it is only possible to either white- or blacklist tables, not both of them.

Composite DataSet

The composite DataSet is very useful for aggregating several already existing datasets into a single dataset. When several datasets contain the same table the rows are appended in the specified order. For example if we have two datasets *fixture1.xml*:

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

and *fixture2.xml*:

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

Using the Composite DataSet we can aggregate both fixture files:

```
<?php
class CompositeTest extends PHPUnit_Extensions_Database_TestCase
{
    public function getDataSet()
    {
        $ds1 = $this->createFlatXmlDataSet('fixture1.xml');
        $ds2 = $this->createFlatXmlDataSet('fixture2.xml');

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

```
        return $compositeDs;
    }
}
?>
```

Beware of Foreign Keys

During Fixture SetUp PHPUnit's Database Extension inserts the rows into the database in the order they are specified in your fixture. If your database schema uses foreign keys this means you have to specify the tables in an order that does not cause foreign key constraints to fail.

Implementing your own DataSets/DataTables

To understand the internals of DataSets and DataTables, lets have a look at the interface of a DataSet. You can skip this part if you do not plan to implement your own DataSet or DataTable.

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

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

The public interface is used internally by the `assertDataSetsEqual()` assertion on the Database TestCase to check for dataset quality. From the `IteratorAggregate` interface the IDataSet inherits the `getIterator()` method to iterate over all tables of the dataset. The reverse iterator allows PHPUnit to truncate tables opposite the order they were created to satisfy foreign key constraints.

Depending on the implementation different approaches are taken to add table instances to a dataset. For example, tables are added internally during construction from the source file in all file-based datasets such as `YamlDataSet`, `XmlDataSet` or `FlatXmlDataSet`.

A table is also represented by the following interface:

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

Except the `getTableMetaData()` method it is pretty self-explainatory. The used methods are all required for the different assertions of the Database Extension that are explained in the next chapter. The `getTableMetaData()` method has to return an implementation of the `PHPUnit_Extensions_Database_DataSet_ITableMetaData` interface, which describes the structure of the table. It holds information on:

- The table name
- An array of column-names of the table, ordered by their appearance in the result-set.

- An array of the primary-key columns.

This interface also has an assertion that checks if two instances of Table Metadata equal each other, which is used by the data-set equality assertion.

The Connection API

There are three interesting methods on the Connection interface which has to be returned from the `getConnection()` method on the Database TestCase:

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

    // ...
}
?>
```

1. The `createDataSet()` method creates a Database (DB) DataSet as described in the DataSet implementations section.

```
<?php
class ConnectionTest extends PHPUnit_Extensions_Database_TestCase
{
    public function testCreateDataSet()
    {
        $tableNames = ['guestbook'];
        $dataSet = $this->getConnection()->createDataSet();
    }
}
?>
```

2. The `createQueryTable()` method can be used to create instances of a QueryTable, give them a result name and SQL query. This is a handy method when it comes to result/table assertions as will be shown in the next section on the Database Assertions API.

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

3. The `getRowCount()` method is a convienent way to access the number of rows in a table, optionally filtered by an additional where clause. This can be used with a simple equality assertion:

```
<?php
class ConnectionTest extends PHPUnit_Extensions_Database_TestCase
{
    public function testGetRowCount()
    {
        $this->assertEquals(2, $this->getConnection()->getRowCount('guestbook'));
    }
}
```

```
?>
```

Database Assertions API

For a testing tool the Database Extension surely provides some assertions that you can use to verify the current state of the database, tables and the row-count of tables. This section describes this functionality in detail:

Asserting the Row-Count of a Table

It is often helpful to check if a table contains a specific amount of rows. You can easily achieve this without additional glue code using the Connection API. Say we wanted to check that after insertion of a row into our guestbook we not only have the two initial entries that have accompanied us in all the previous examples, but a third one:

```
<?php
class GuestbookTest extends PHPUnit_Extensions_Database_TestCase
{
    public function testAddEntry()
    {
        $this->assertEquals(2, $this->getConnection()->getRowCount('guestbook'), "Pre-Co

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

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

Asserting the State of a Table

The previous assertion is helpful, but we surely want to check the actual contents of the table to verify that all the values were written into the correct columns. This can be achieved by a table assertion.

For this we would define a Query Table instance which derives its content from a table name and SQL query and compare it to a File/Array Based Data Set:

```
<?php
class GuestbookTest extends PHPUnit_Extensions_Database_TestCase
{
    public function testAddEntry()
    {
        $guestbook = new Guestbook();
        $guestbook->addEntry("suzy", "Hello world!");

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

Now we have to write the *expectedBook.xml* Flat XML file for this assertion:

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

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

This assertion would only pass on exactly one second of the universe though, on *2010–05–01 21:47:08*. Dates pose a special problem to database testing and we can circumvent the failure by omitting the "created" column from the assertion.

The adjusted *expectedBook.xml* Flat XML file would probably have to look like the following to make the assertion pass:

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

We have to fix up the Query Table call:

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

Asserting the Result of a Query

You can also assert the result of complex queries with the Query Table approach, just specify a result name with a query and compare it to a dataset:

```
<?php
class ComplexQueryTest extends PHPUnit_Extensions_Database_TestCase
{
    public function testComplexQuery()
    {
        $queryTable = $this->getConnection()->createQueryTable(
            'myComplexQuery', 'SELECT complexQuery...'
        );
        $expectedTable = $this->createFlatXmlDataSet("complexQueryAssertion.xml")
                              ->getTable("myComplexQuery");
        $this->assertTablesEqual($expectedTable, $queryTable);
    }
}
?>
```

Asserting the State of Multiple Tables

For sure you can assert the state of multiple tables at once and compare a query dataset against a file based dataset. There are two different ways for DataSet assertions.

1. You can use the Database (DB) DataSet from the Connection and compare it to a File-Based DataSet.

```
<?php
class DataSetAssertionsTest extends PHPUnit_Extensions_Database_TestCase
{
    public function testCreateDataSetAssertion()
```

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

2. You can construct the DataSet on your own:

```
<?php
class DataSetAssertionsTest extends PHPUnit_Extensions_Database_TestCase
{
    public function testManualDataSetAssertion()
    {
        $dataSet = new PHPUnit_Extensions_Database_DataSet_QueryDataSet();
        $dataSet->addTable('guestbook', 'SELECT id, content, user FROM guestbook'); //
        $expectedDataSet = $this->createFlatXmlDataSet('guestbook.xml');

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

Frequently Asked Questions

Will PHPUnit (re-)create the database schema for each test?

No, PHPUnit requires all database objects to be available when the suite is started. The Database, tables, sequences, triggers and views have to be created before you run the test suite.

Doctrine 2 [http://www.doctrine-project.org] or eZ Components [http://www.ezcomponents.org] have powerful tools that allow you to create the database schema from pre-defined datastructures. However, these have to be hooked into the PHPUnit extension to allow an automatic database re-creation before the complete test-suite is run.

Since each test completely cleans the database you are not even required to re-create the database for each test-run. A permanently available database works perfectly.

Am I required to use PDO in my application for the Database Extension to work?

No, PDO is only required for the fixture clean- and set-up and for assertions. You can use whatever database abstraction you want inside your own code.

What can I do, when I get a “Too much Connections” Error?

If you do not cache the PDO instance that is created from the TestCase `getConnection()` method the number of connections to the database is increasing by one or more with each database test. With default configuration MySql only allows 100 concurrent connections other vendors also have maximum connection limits.

The SubSection “Use your own Abstract Database TestCase” shows how you can prevent this error from happening by using a single cached PDO instance in all your tests.

How to handle NULL with Flat XML / CSV Datasets?

Do not do this. Instead, you should use either the XML or the YAML DataSets.

Chapter 9. Test Doubles

Gerard Meszaros introduces the concept of Test Doubles in [Meszaros2007] like this:

> Sometimes it is just plain hard to test the system under test (SUT) because it depends on other components that cannot be used in the test environment. This could be because they aren't available, they will not return the results needed for the test or because executing them would have undesirable side effects. In other cases, our test strategy requires us to have more control or visibility of the internal behavior of the SUT.
>
> When we are writing a test in which we cannot (or chose not to) use a real depended-on component (DOC), we can replace it with a Test Double. The Test Double doesn't have to behave exactly like the real DOC; it merely has to provide the same API as the real one so that the SUT thinks it is the real one!
>
> —Gerard Meszaros

The `createMock($type)` and `getMockBuilder($type)` methods provided by PHPUnit can be used in a test to automatically generate an object that can act as a test double for the specified original type (interface or class name). This test double object can be used in every context where an object of the original type is expected or required.

The `createMock($type)` method immediately returns a test double object for the specified type (interface or class). The creation of this test double is performed using best practice defaults (the `__construct()` and `__clone()` methods of the original class are not executed and the arguments passed to a method of the test double will not be cloned. If these defaults are not what you need then you can use the `getMockBuilder($type)` method to customize the test double generation using a fluent interface.

By default, all methods of the original class are replaced with a dummy implementation that just returns `null` (without calling the original method). Using the `will($this->returnValue())` method, for instance, you can configure these dummy implementations to return a value when called.

Limitation: final, private, and static methods

Please note that `final`, `private` and `static` methods cannot be stubbed or mocked. They are ignored by PHPUnit's test double functionality and retain their original behavior.

Stubs

The practice of replacing an object with a test double that (optionally) returns configured return values is referred to as *stubbing*. You can use a *stub* to "replace a real component on which the SUT depends so that the test has a control point for the indirect inputs of the SUT. This allows the test to force the SUT down paths it might not otherwise execute".

Example 9.2, “Stubbing a method call to return a fixed value” shows how to stub method calls and set up return values. We first use the `createMock()` method that is provided by the `PHPUnit\Framework\TestCase` class to set up a stub object that looks like an object of `SomeClass` (Example 9.1, “The class we want to stub”). We then use the Fluent Interface [http://martinfowler.com/bliki/FluentInterface.html] that PHPUnit provides to specify the behavior for the stub. In essence, this means that you do not need to create several temporary objects and wire them together afterwards. Instead, you chain method calls as shown in the example. This leads to more readable and "fluent" code.

Example 9.1. The class we want to stub

```
<?php
```

```
use PHPUnit\Framework\TestCase;

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

Example 9.2. Stubbing a method call to return a fixed value

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

class StubTest extends TestCase
{
    public function testStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->createMock(SomeClass::class);

        // Configure the stub.
        $stub->method('doSomething')
             ->willReturn('foo');

        // Calling $stub->doSomething() will now return
        // 'foo'.
        $this->assertEquals('foo', $stub->doSomething());
    }
}
?>
```

Limitation: Methods named "method"

The example shown above only works when the original class does not declare a method named "method".

If the original class does declare a method named "method" then `$stub->expects($this->any())->method('doSomething')->willReturn('foo');` has to be used.

"Behind the scenes", PHPUnit automatically generates a new PHP class that implements the desired behavior when the `createMock()` method is used.

Example 9.3, “Using the Mock Builder API can be used to configure the generated test double class” shows an example of how to use the Mock Builder's fluent interface to configure the creation of the test double. The configuration of this test double uses the same best practice defaults used by `createMock()`.

Example 9.3. Using the Mock Builder API can be used to configure the generated test double class

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

class StubTest extends TestCase
{
    public function testStub()
```

```
    {
        // Create a stub for the SomeClass class.
        $stub = $this->getMockBuilder($originalClassName)
                     ->disableOriginalConstructor()
                     ->disableOriginalClone()
                     ->disableArgumentCloning()
                     ->disallowMockingUnknownTypes()
                     ->getMock();

        // Configure the stub.
        $stub->method('doSomething')
             ->willReturn('foo');

        // Calling $stub->doSomething() will now return
        // 'foo'.
        $this->assertEquals('foo', $stub->doSomething());
    }
}
?>
```

In the examples so far we have been returning simple values using `willReturn($value)`. This short syntax is the same as `will($this->returnValue($value))`. We can use variations on this longer syntax to achieve more complex stubbing behaviour.

Sometimes you want to return one of the arguments of a method call (unchanged) as the result of a stubbed method call. Example 9.4, “Stubbing a method call to return one of the arguments” shows how you can achieve this using `returnArgument()` instead of `returnValue()`.

Example 9.4. Stubbing a method call to return one of the arguments

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

class StubTest extends TestCase
{
    public function testReturnArgumentStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->createMock(SomeClass::class);

        // Configure the stub.
        $stub->method('doSomething')
             ->will($this->returnArgument(0));

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

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

When testing a fluent interface, it is sometimes useful to have a stubbed method return a reference to the stubbed object. Example 9.5, “Stubbing a method call to return a reference to the stub object” shows how you can use `returnSelf()` to achieve this.

Example 9.5. Stubbing a method call to return a reference to the stub object

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

```
class StubTest extends TestCase
{
    public function testReturnSelf()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->createMock(SomeClass::class);

        // Configure the stub.
        $stub->method('doSomething')
             ->will($this->returnSelf());

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

Sometimes a stubbed method should return different values depending on a predefined list of arguments. You can use `returnValueMap()` to create a map that associates arguments with corresponding return values. See Example 9.6, “Stubbing a method call to return the value from a map” for an example.

Example 9.6. Stubbing a method call to return the value from a map

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

class StubTest extends TestCase
{
    public function testReturnValueMapStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->createMock(SomeClass::class);

        // Create a map of arguments to return values.
        $map = [
            ['a', 'b', 'c', 'd'],
            ['e', 'f', 'g', 'h']
        ];

        // Configure the stub.
        $stub->method('doSomething')
             ->will($this->returnValueMap($map));

        // $stub->doSomething() returns different values depending on
        // the provided arguments.
        $this->assertEquals('d', $stub->doSomething('a', 'b', 'c'));
        $this->assertEquals('h', $stub->doSomething('e', 'f', 'g'));
    }
}
?>
```

When the stubbed method call should return a calculated value instead of a fixed one (see `returnValue()`) or an (unchanged) argument (see `returnArgument()`), you can use `returnCallback()` to have the stubbed method return the result of a callback function or method. See Example 9.7, “Stubbing a method call to return a value from a callback” for an example.

Example 9.7. Stubbing a method call to return a value from a callback

```
<?php
```

```
use PHPUnit\Framework\TestCase;

class StubTest extends TestCase
{
    public function testReturnCallbackStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->createMock(SomeClass::class);

        // Configure the stub.
        $stub->method('doSomething')
             ->will($this->returnCallback('str_rot13'));

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

A simpler alternative to setting up a callback method may be to specify a list of desired return values. You can do this with the `onConsecutiveCalls()` method. See Example 9.8, “Stubbing a method call to return a list of values in the specified order” for an example.

Example 9.8. Stubbing a method call to return a list of values in the specified order

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

class StubTest extends TestCase
{
    public function testOnConsecutiveCallsStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->createMock(SomeClass::class);

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

        // $stub->doSomething() returns a different value each time
        $this->assertEquals(2, $stub->doSomething());
        $this->assertEquals(3, $stub->doSomething());
        $this->assertEquals(5, $stub->doSomething());
    }
}
?>
```

Instead of returning a value, a stubbed method can also raise an exception. Example 9.9, “Stubbing a method call to throw an exception” shows how to use `throwException()` to do this.

Example 9.9. Stubbing a method call to throw an exception

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

class StubTest extends TestCase
{
    public function testThrowExceptionStub()
    {
        // Create a stub for the SomeClass class.
```

```
        $stub = $this->createMock(SomeClass::class);

        // Configure the stub.
        $stub->method('doSomething')
             ->will($this->throwException(new Exception));

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

Alternatively, you can write the stub yourself and improve your design along the way. Widely used resources are accessed through a single façade, so you can easily replace the resource with the stub. For example, instead of having direct database calls scattered throughout the code, you have a single `Database` object, an implementor of the `IDatabase` interface. Then, you can create a stub implementation of `IDatabase` and use it for your tests. You can even create an option for running the tests with the stub database or the real database, so you can use your tests for both local testing during development and integration testing with the real database.

Functionality that needs to be stubbed out tends to cluster in the same object, improving cohesion. By presenting the functionality with a single, coherent interface you reduce the coupling with the rest of the system.

Mock Objects

The practice of replacing an object with a test double that verifies expectations, for instance asserting that a method has been called, is referred to as *mocking*.

You can use a *mock object* "as an observation point that is used to verify the indirect outputs of the SUT as it is exercised. Typically, the mock object also includes the functionality of a test stub in that it must return values to the SUT if it hasn't already failed the tests but the emphasis is on the verification of the indirect outputs. Therefore, a mock object is a lot more than just a test stub plus assertions; it is used in a fundamentally different way" (Gerard Meszaros).

Limitation: Automatic verification of expectations

Only mock objects generated within the scope of a test will be verified automatically by PHPUnit. Mock objects generated in data providers, for instance, or injected into the test using the `@depends` annotation will not be verified automatically by PHPUnit.

Here is an example: suppose we want to test that the correct method, `update()` in our example, is called on an object that observes another object. Example 9.10, “The Subject and Observer classes that are part of the System under Test (SUT)” shows the code for the `Subject` and `Observer` classes that are part of the System under Test (SUT).

Example 9.10. The Subject and Observer classes that are part of the System under Test (SUT)

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

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

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

```
    }

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

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

    public function doSomething()
    {
        // Do something.
        // ...

        // Notify observers that we did something.
        $this->notify('something');
    }

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

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

    // Other methods.
}

class Observer
{
    public function update($argument)
    {
        // Do something.
    }

    public function reportError($errorCode, $errorMessage, Subject $subject)
    {
        // Do something
    }

    // Other methods.
}
?>
```

Example 9.11, “Testing that a method gets called once and with a specified argument” shows how to use a mock object to test the interaction between `Subject` and `Observer` objects.

We first use the `getMockBuilder()` method that is provided by the `PHPUnit\Framework\TestCase` class to set up a mock object for the `Observer`. Since we give an array as the second (optional) parameter for the `getMock()` method, only the `update()` method of the `Observer` class is replaced by a mock implementation.

Because we are interested in verifying that a method is called, and which arguments it is called with, we introduce the `expects()` and `with()` methods to specify how this interaction should look.

Example 9.11. Testing that a method gets called once and with a specified argument

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

class SubjectTest extends TestCase
{
    public function testObserversAreUpdated()
    {
        // Create a mock for the Observer class,
        // only mock the update() method.
        $observer = $this->getMockBuilder(Observer::class)
                         ->setMethods(['update'])
                         ->getMock();

        // Set up the expectation for the update() method
        // to be called only once and with the string 'something'
        // as its parameter.
        $observer->expects($this->once())
                 ->method('update')
                 ->with($this->equalTo('something'));

        // Create a Subject object and attach the mocked
        // Observer object to it.
        $subject = new Subject('My subject');
        $subject->attach($observer);

        // Call the doSomething() method on the $subject object
        // which we expect to call the mocked Observer object's
        // update() method with the string 'something'.
        $subject->doSomething();
    }
}
?>
```

The `with()` method can take any number of arguments, corresponding to the number of arguments to the method being mocked. You can specify more advanced constraints on the method's arguments than a simple match.

Example 9.12. Testing that a method gets called with a number of arguments constrained in different ways

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

class SubjectTest extends TestCase
{
    public function testErrorReported()
    {
        // Create a mock for the Observer class, mocking the
        // reportError() method
        $observer = $this->getMockBuilder(Observer::class)
                         ->setMethods(['reportError'])
                         ->getMock();

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

```
                    );

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

        // The doSomethingBad() method should report an error to the observer
        // via the reportError() method
        $subject->doSomethingBad();
    }
}
?>
```

The `withConsecutive()` method can take any number of arrays of arguments, depending on the calls you want to test against. Each array is a list of constraints corresponding to the arguments of the method being mocked, like in `with()`.

Example 9.13. Testing that a method gets called two times with specific arguments.

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

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

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

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

The `callback()` constraint can be used for more complex argument verification. This constraint takes a PHP callback as its only argument. The PHP callback will receive the argument to be verified as its only argument and should return `true` if the argument passes verification and `false` otherwise.

Example 9.14. More complex argument verification

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

class SubjectTest extends TestCase
{
    public function testErrorReported()
    {
        // Create a mock for the Observer class, mocking the
        // reportError() method
        $observer = $this->getMockBuilder(Observer::class)
                         ->setMethods(['reportError'])
                         ->getMock();
```

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

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

        // The doSomethingBad() method should report an error to the observer
        // via the reportError() method
        $subject->doSomethingBad();
    }
}
?>
```

Example 9.15. Testing that a method gets called once and with the identical object as was passed

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

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

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

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

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

Example 9.16. Create a mock object with cloning parameters enabled

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

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

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

        // now your mock clones parameters so the identicalTo constraint
        // will fail.
    }
```

```
}
?>
```

Table A.1, “Constraints” shows the constraints that can be applied to method arguments and Table 9.1, “Matchers” shows the matchers that are available to specify the number of invocations.

Table 9.1. Matchers

Matcher	Meaning
`PHPUnit_ Framework_ MockObject_ Matcher_ AnyInvokedCount any()`	Returns a matcher that matches when the method it is evaluated for is executed zero or more times.
`PHPUnit_ Framework_ MockObject_ Matcher_ InvokedCount never()`	Returns a matcher that matches when the method it is evaluated for is never executed.
`PHPUnit_ Framework_ MockObject_ Matcher_ InvokedAtLeastOnce atLeastOnce()`	Returns a matcher that matches when the method it is evaluated for is executed at least once.
`PHPUnit_ Framework_ MockObject_ Matcher_ InvokedCount once()`	Returns a matcher that matches when the method it is evaluated for is executed exactly once.
`PHPUnit_ Framework_ MockObject_ Matcher_ InvokedCount exactly(int $count)`	Returns a matcher that matches when the method it is evaluated for is executed exactly `$count` times.
`PHPUnit_ Framework_ MockObject_ Matcher_ InvokedAtIndex at(int $index)`	Returns a matcher that matches when the method it is evaluated for is invoked at the given `$index`.

Note

The `$index` parameter for the `at()` matcher refers to the index, starting at zero, in *all method invocations* for a given mock object. Exercise caution when using this matcher as it can lead to brittle tests which are too closely tied to specific implementation details.

As mentioned in the beginning, when the defaults used by the `createMock()` method to generate the test double do not match your needs then you can use the `getMockBuilder($type)` method to customize the test double generation using a fluent interface. Here is a list of methods provided by the Mock Builder:

- `setMethods(array $methods)` can be called on the Mock Builder object to specify the methods that are to be replaced with a configurable test double. The behavior of the other methods is not changed. If you call `setMethods(null)`, then no methods will be replaced.
- `setConstructorArgs(array $args)` can be called to provide a parameter array that is passed to the original class' constructor (which is not replaced with a dummy implementation by default).
- `setMockClassName($name)` can be used to specify a class name for the generated test double class.
- `disableOriginalConstructor()` can be used to disable the call to the original class' constructor.
- `disableOriginalClone()` can be used to disable the call to the original class' clone constructor.
- `disableAutoload()` can be used to disable `__autoload()` during the generation of the test double class.

Prophecy

Prophecy [https://github.com/phpspec/prophecy] is a "highly opinionated yet very powerful and flexible PHP object mocking framework. Though initially it was created to fulfil phpspec2 needs, it is flexible enough to be used inside any testing framework out there with minimal effort".

PHPUnit has built-in support for using Prophecy to create test doubles. Example 9.17, “Testing that a method gets called once and with a specified argument” shows how the same test shown in Example 9.11, “Testing that a method gets called once and with a specified argument” can be expressed using Prophecy's philosophy of prophecies and revelations:

Example 9.17. Testing that a method gets called once and with a specified argument

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

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

        // Create a prophecy for the Observer class.
        $observer = $this->prophesize(Observer::class);

        // Set up the expectation for the update() method
        // to be called only once and with the string 'something'
        // as its parameter.
        $observer->update('something')->shouldBeCalled();

        // Reveal the prophecy and attach the mock object
        // to the Subject.
        $subject->attach($observer->reveal());

        // Call the doSomething() method on the $subject object
        // which we expect to call the mocked Observer object's
        // update() method with the string 'something'.
        $subject->doSomething();
    }
}
?>
```

Please refer to the documentation [https://github.com/phpspec/prophecy#how-to-use-it] for Prophecy for further details on how to create, configure, and use stubs, spies, and mocks using this alternative test double framework.

Mocking Traits and Abstract Classes

The `getMockForTrait()` method returns a mock object that uses a specified trait. All abstract methods of the given trait are mocked. This allows for testing the concrete methods of a trait.

Example 9.18. Testing the concrete methods of a trait

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

trait AbstractTrait
{
```

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

    public abstract function abstractMethod();
}

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

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

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

The `getMockForAbstractClass()` method returns a mock object for an abstract class. All abstract methods of the given abstract class are mocked. This allows for testing the concrete methods of an abstract class.

Example 9.19. Testing the concrete methods of an abstract class

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

abstract class AbstractClass
{
    public function concreteMethod()
    {
        return $this->abstractMethod();
    }

    public abstract function abstractMethod();
}

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

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

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

Stubbing and Mocking Web Services

When your application interacts with a web service you want to test it without actually interacting with the web service. To make the stubbing and mocking of web services easy, the `getMockFromWs-`

`dl()` can be used just like `getMock()` (see above). The only difference is that `getMockFromWsdl()` returns a stub or mock based on a web service description in WSDL and `getMock()` returns a stub or mock based on a PHP class or interface.

Example 9.20, “Stubbing a web service” shows how `getMockFromWsdl()` can be used to stub, for example, the web service described in `GoogleSearch.wsdl`.

Example 9.20. Stubbing a web service

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

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

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

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

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

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

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

```
            false,
            '',
            false,
            '',
            '',
            ''
          )
        );
    }
}
?>
```

Mocking the Filesystem

vfsStream [https://github.com/mikey179/vfsStream] is a stream wrapper [http://www.php.net/streams] for a virtual filesystem [http://en.wikipedia.org/wiki/Virtual_file_system] that may be helpful in unit tests to mock the real filesystem.

Simply add a dependency on `mikey179/vfsStream` to your project's `composer.json` file if you use Composer [https://getcomposer.org/] to manage the dependencies of your project. Here is a minimal example of a `composer.json` file that just defines a development-time dependency on PHPUnit 4.6 and vfsStream:

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

Example 9.21, “A class that interacts with the filesystem” shows a class that interacts with the filesystem.

Example 9.21. A class that interacts with the filesystem

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

class Example
{
    protected $id;
    protected $directory;

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

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

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

Without a virtual filesystem such as vfsStream we cannot test the `setDirectory()` method in isolation from external influence (see Example 9.22, “Testing a class that interacts with the filesystem”).

Example 9.22. Testing a class that interacts with the filesystem

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

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

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

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

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

The approach above has several drawbacks:

- As with any external resource, there might be intermittent problems with the filesystem. This makes tests interacting with it flaky.
- In the `setUp()` and `tearDown()` methods we have to ensure that the directory does not exist before and after the test.
- When the test execution terminates before the `tearDown()` method is invoked the directory will stay in the filesystem.

Example 9.23, “Mocking the filesystem in a test for a class that interacts with the filesystem” shows how vfsStream can be used to mock the filesystem in a test for a class that interacts with the filesystem.

Example 9.23. Mocking the filesystem in a test for a class that interacts with the filesystem

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

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

    public function testDirectoryIsCreated()
    {
```

```
        $example = new Example('id');
        $this->assertFalse(vfsStreamWrapper::getRoot()->hasChild('id'));

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

This has several advantages:

- The test itself is more concise.
- vfsStream gives the test developer full control over what the filesystem environment looks like to the tested code.
- Since the filesystem operations do not operate on the real filesystem anymore, cleanup operations in a `tearDown()` method are no longer required.

Chapter 10. Testing Practices

> You can always write more tests. However, you will quickly find that only a fraction of the tests you can imagine are actually useful. What you want is to write tests that fail even though you think they should work, or tests that succeed even though you think they should fail. Another way to think of it is in cost/benefit terms. You want to write tests that will pay you back with information.
>
> —Erich Gamma

During Development

When you need to make a change to the internal structure of the software you are working on to make it easier to understand and cheaper to modify without changing its observable behavior, a test suite is invaluable in applying these so called refactorings [http://martinfowler.com/bliki/DefinitionOfRefactoring.html] safely. Otherwise, you might not notice the system breaking while you are carrying out the restructuring.

The following conditions will help you to improve the code and design of your project, while using unit tests to verify that the refactoring's transformation steps are, indeed, behavior-preserving and do not introduce errors:

1. All unit tests run correctly.
2. The code communicates its design principles.
3. The code contains no redundancies.
4. The code contains the minimal number of classes and methods.

When you need to add new functionality to the system, write the tests first. Then, you will be done developing when the test runs. This practice will be discussed in detail in the next chapter.

During Debugging

When you get a defect report, your impulse might be to fix the defect as quickly as possible. Experience shows that this impulse will not serve you well; it is likely that the fix for the defect causes another defect.

You can hold your impulse in check by doing the following:

1. Verify that you can reproduce the defect.
2. Find the smallest-scale demonstration of the defect in the code. For example, if a number appears incorrectly in an output, find the object that is computing that number.
3. Write an automated test that fails now but will succeed when the defect is fixed.
4. Fix the defect.

Finding the smallest reliable reproduction of the defect gives you the opportunity to really examine the cause of the defect. The test you write will improve the chances that when you fix the defect, you really fix it, because the new test reduces the likelihood of undoing the fix with future code changes. All the tests you wrote before reduce the likelihood of inadvertently causing a different problem.

Unit testing offers many advantages:

- Testing gives code authors and reviewers confidence that patches produce the correct results.

- Authoring testcases is a good impetus for developers to discover edge cases.
- Testing provides a good way to catch regressions quickly, and to make sure that no regression will be repeated twice.
- Unit tests provide working examples for how to use an API and can significantly aid documentation efforts.

Overall, integrated unit testing makes the cost and risk of any individual change smaller. It will allow the project to make [...] major architectural improvements [...] quickly and confidently.

—Benjamin Smedberg

Chapter 11. Code Coverage Analysis

> In computer science, code coverage is a measure used to describe the degree to which the source code of a program is tested by a particular test suite. A program with high code coverage has been more thoroughly tested and has a lower chance of containing software bugs than a program with low code coverage.
>
> —Wikipedia

In this chapter you will learn all about PHPUnit's code coverage functionality that provides an insight into what parts of the production code are executed when the tests are run. It makes use of the PHP_CodeCoverage [https://github.com/sebastianbergmann/php-code-coverage] component, which in turn leverages the code coverage functionality provided by the Xdebug [http://xdebug.org/] extension for PHP.

Note

Xdebug is not distributed as part of PHPUnit. If you receive a notice while running tests that the Xdebug extension is not loaded, it means that Xdebug is either not installed or not configured properly. Before you can use the code coverage analysis features in PHPUnit, you should read the Xdebug installation guide [http://xdebug.org/docs/install].

PHPUnit can generate an HTML-based code coverage report as well as XML-based logfiles with code coverage information in various formats (Clover, Crap4J, PHPUnit). Code coverage information can also be reported as text (and printed to STDOUT) and exported as PHP code for further processing.

Please refer to Chapter 3, *The Command-Line Test Runner* for a list of commandline switches that control code coverage functionality as well as the section called “Logging” for the relevant configuration settings.

Software Metrics for Code Coverage

Various software metrics exist to measure code coverage:

Line Coverage
: The *Line Coverage* software metric measures whether each executable line was executed.

Function and Method Coverage
: The *Function and Method Coverage* software metric measures whether each function or method has been invoked. PHP_CodeCoverage only considers a function or method as covered when all of its executable lines are covered.

Class and Trait Coverage
: The *Class and Trait Coverage* software metric measures whether each method of a class or trait is covered. PHP_CodeCoverage only considers a class or trait as covered when all of its methods are covered.

Opcode Coverage
: The *Opcode Coverage* software metric measures whether each opcode of a function or method has been executed while running the test suite. A line of code usually compiles into more than one opcode. Line Coverage regards a line of code as covered as soon as one of its opcodes is executed.

Branch Coverage
: The *Branch Coverage* software metric measures whether the boolean expression of each control structure evaluated to both `true` and `false` while running the test suite.

Path Coverage
: The *Path Coverage* software metric measures whether each of the possible execution paths in a function or method has been

followed while running the test suite. An execution path is a unique sequence of branches from the entry of the function or method to its exit.

Change Risk Anti-Patterns (CRAP) Index

The *Change Risk Anti-Patterns (CRAP) Index* is calculated based on the cyclomatic complexity and code coverage of a unit of code. Code that is not too complex and has an adequate test coverage will have a low CRAP index. The CRAP index can be lowered by writing tests and by refactoring the code to lower its complexity.

Note

The *Opcode Coverage*, *Branch Coverage*, and *Path Coverage* software metrics are not yet supported by PHP_CodeCoverage.

Whitelisting Files

It is mandatory to configure a *whitelist* for telling PHPUnit which sourcecode files to include in the code coverage report. This can either be done using the `--whitelist` commandline option or via the configuration file (see the section called “Whitelisting Files for Code Coverage”).

Optionally, all whitelisted files can be added to the code coverage report by setting `addUncoveredFilesFromWhitelist="true"` in your PHPUnit configuration (see the section called “Whitelisting Files for Code Coverage”). This allows the inclusion of files that are not tested yet at all. If you want to get information about which lines of such an uncovered file are executable, for instance, you also need to set `processUncoveredFilesFromWhitelist="true"` in your PHPUnit configuration (see the section called “Whitelisting Files for Code Coverage”).

Note

Please note that the loading of sourcecode files that is performed when `processUncoveredFilesFromWhitelist="true"` is set can cause problems when a sourcecode file contains code outside the scope of a class or function, for instance.

Ignoring Code Blocks

Sometimes you have blocks of code that you cannot test and that you may want to ignore during code coverage analysis. PHPUnit lets you do this using the `@codeCoverageIgnore`, `@codeCoverageIgnoreStart` and `@codeCoverageIgnoreEnd` annotations as shown in Example 11.1, “Using the `@codeCoverageIgnore`, `@codeCoverageIgnoreStart` and `@codeCoverageIgnoreEnd` annotations”.

Example 11.1. Using the `@codeCoverageIgnore`, `@codeCoverageIgnoreStart` and `@codeCoverageIgnoreEnd` annotations

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

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

```
    }
}

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

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

exit; // @codeCoverageIgnore
?>
```

The ignored lines of code (marked as ignored using the annotations) are counted as executed (if they are executable) and will not be highlighted.

Specifying Covered Methods

The `@covers` annotation (see Table B.1, “Annotations for specifying which methods are covered by a test”) can be used in the test code to specify which method(s) a test method wants to test. If provided, only the code coverage information for the specified method(s) will be considered. Example 11.2, “Tests that specify which method they want to cover” shows an example.

Example 11.2. Tests that specify which method they want to cover

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

class BankAccountTest extends TestCase
{
    protected $ba;

    protected function setUp()
    {
        $this->ba = new BankAccount;
    }

    /**
     * @covers BankAccount::getBalance
     */
    public function testBalanceIsInitiallyZero()
    {
        $this->assertEquals(0, $this->ba->getBalance());
    }

    /**
     * @covers BankAccount::withdrawMoney
     */
    public function testBalanceCannotBecomeNegative()
    {
        try {
            $this->ba->withdrawMoney(1);
        }
```

```
        catch (BankAccountException $e) {
            $this->assertEquals(0, $this->ba->getBalance());

            return;
        }

        $this->fail();
    }

    /**
     * @covers BankAccount::depositMoney
     */
    public function testBalanceCannotBecomeNegative2()
    {
        try {
            $this->ba->depositMoney(-1);
        }

        catch (BankAccountException $e) {
            $this->assertEquals(0, $this->ba->getBalance());

            return;
        }

        $this->fail();
    }

    /**
     * @covers BankAccount::getBalance
     * @covers BankAccount::depositMoney
     * @covers BankAccount::withdrawMoney
     */
    public function testDepositWithdrawMoney()
    {
        $this->assertEquals(0, $this->ba->getBalance());
        $this->ba->depositMoney(1);
        $this->assertEquals(1, $this->ba->getBalance());
        $this->ba->withdrawMoney(1);
        $this->assertEquals(0, $this->ba->getBalance());
    }
}
?>
```

It is also possible to specify that a test should not cover *any* method by using the `@coversNothing` annotation (see the section called “@coversNothing”). This can be helpful when writing integration tests to make sure you only generate code coverage with unit tests.

Example 11.3. A test that specifies that no method should be covered

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

class GuestbookIntegrationTest extends PHPUnit_Extensions_Database_TestCase
{
    /**
     * @coversNothing
     */
    public function testAddEntry()
    {
        $guestbook = new Guestbook();
        $guestbook->addEntry("suzy", "Hello world!");
```

```
        $queryTable = $this->getConnection()->createQueryTable(
            'guestbook', 'SELECT * FROM guestbook'
        );

        $expectedTable = $this->createFlatXmlDataSet("expectedBook.xml")
                              ->getTable("guestbook");

        $this->assertTablesEqual($expectedTable, $queryTable);
    }
}
?>
```

Edge Cases

This section shows noteworthy edge cases that lead to confusing code coverage information.

Example 11.4.

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

// Because it is "line based" and not statement base coverage
// one line will always have one coverage status
if (false) this_function_call_shows_up_as_covered();

// Due to how code coverage works internally these two lines are special.
// This line will show up as non executable
if (false)
    // This line will show up as covered because it is actually the
    // coverage of the if statement in the line above that gets shown here!
    will_also_show_up_as_covered();

// To avoid this it is necessary that braces are used
if (false) {
    this_call_will_never_show_up_as_covered();
}
?>
```

Chapter 12. Other Uses for Tests

Once you get used to writing automated tests, you will likely discover more uses for tests. Here are some examples.

Agile Documentation

Typically, in a project that is developed using an agile process, such as Extreme Programming, the documentation cannot keep up with the frequent changes to the project's design and code. Extreme Programming demands *collective code ownership*, so all developers need to know how the entire system works. If you are disciplined enough to consequently use "speaking names" for your tests that describe what a class should do, you can use PHPUnit's TestDox functionality to generate automated documentation for your project based on its tests. This documentation gives developers an overview of what each class of the project is supposed to do.

PHPUnit's TestDox functionality looks at a test class and all the test method names and converts them from camel case PHP names to sentences: `testBalanceIsInitiallyZero()` becomes "Balance is initially zero". If there are several test methods whose names only differ in a suffix of one or more digits, such as `testBalanceCannotBecomeNegative()` and `testBalanceCannotBecomeNegative2()`, the sentence "Balance cannot become negative" will appear only once, assuming that all of these tests succeed.

Let us take a look at the agile documentation generated for a `BankAccount` class:

```
phpunit --testdox BankAccountTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

BankAccount
 [x] Balance is initially zero
 [x] Balance cannot become negative
```

Alternatively, the agile documentation can be generated in HTML or plain text format and written to a file using the `--testdox-html` and `--testdox-text` arguments.

Agile Documentation can be used to document the assumptions you make about the external packages that you use in your project. When you use an external package, you are exposed to the risks that the package will not behave as you expect, and that future versions of the package will change in subtle ways that will break your code, without you knowing it. You can address these risks by writing a test every time you make an assumption. If your test succeeds, your assumption is valid. If you document all your assumptions with tests, future releases of the external package will be no cause for concern: if the tests succeed, your system should continue working.

Cross-Team Tests

When you document assumptions with tests, you own the tests. The supplier of the package -- who you make assumptions about -- knows nothing about your tests. If you want to have a closer relationship with the supplier of a package, you can use the tests to communicate and coordinate your activities.

When you agree on coordinating your activities with the supplier of a package, you can write the tests together. Do this in such a way that the tests reveal as many assumptions as possible. Hidden assumptions are the death of cooperation. With the tests, you document exactly what you expect from the supplied package. The supplier will know the package is complete when all the tests run.

By using stubs (see the chapter on "Mock Objects", earlier in this book), you can further decouple yourself from the supplier: The job of the supplier is to make the tests run with the real implementation of the package. Your job is to make the tests run for your own code. Until such time as you have the

real implementation of the supplied package, you use stub objects. Following this approach, the two teams can develop independently.

Chapter 13. Logging

PHPUnit can produce several types of logfiles.

Test Results (XML)

The XML logfile for test results produced by PHPUnit is based upon the one used by the JUnit task for Apache Ant [http://ant.apache.org/manual/Tasks/junit.html]. The following example shows the XML logfile generated for the tests in `ArrayTest`:

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites>
  <testsuite name="ArrayTest"
             file="/home/sb/ArrayTest.php"
             tests="2"
             assertions="2"
             failures="0"
             errors="0"
             time="0.016030">
    <testcase name="testNewArrayIsEmpty"
              class="ArrayTest"
              file="/home/sb/ArrayTest.php"
              line="6"
              assertions="1"
              time="0.008044"/>
    <testcase name="testArrayContainsAnElement"
              class="ArrayTest"
              file="/home/sb/ArrayTest.php"
              line="15"
              assertions="1"
              time="0.007986"/>
  </testsuite>
</testsuites>
```

The following XML logfile was generated for two tests, `testFailure` and `testError`, of a test case class named `FailureErrorTest` and shows how failures and errors are denoted.

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuites>
  <testsuite name="FailureErrorTest"
             file="/home/sb/FailureErrorTest.php"
             tests="2"
             assertions="1"
             failures="1"
             errors="1"
             time="0.019744">
    <testcase name="testFailure"
              class="FailureErrorTest"
              file="/home/sb/FailureErrorTest.php"
              line="6"
              assertions="1"
              time="0.011456">
      <failure type="PHPUnit_Framework_ExpectationFailedException">
testFailure(FailureErrorTest)
Failed asserting that &lt;integer:2&gt; matches expected value &lt;integer:1&gt;.

/home/sb/FailureErrorTest.php:8
</failure>
    </testcase>
    <testcase name="testError"
              class="FailureErrorTest"
```

```
              file="/home/sb/FailureErrorTest.php"
              line="11"
              assertions="0"
              time="0.008288">
      <error type="Exception">testError(FailureErrorTest)
Exception:

/home/sb/FailureErrorTest.php:13
</error>
    </testcase>
  </testsuite>
</testsuites>
```

Test Results (TAP)

The Test Anything Protocol (TAP) [http://testanything.org/] is Perl's simple text-based interface between testing modules. The following example shows the TAP logfile generated for the tests in `ArrayTest`:

```
TAP version 13
ok 1 - testNewArrayIsEmpty(ArrayTest)
ok 2 - testArrayContainsAnElement(ArrayTest)
1..2
```

The following TAP logfile was generated for two tests, `testFailure` and `testError`, of a test case class named `FailureErrorTest` and shows how failures and errors are denoted.

```
TAP version 13
not ok 1 - Failure: testFailure(FailureErrorTest)
  ---
  message: 'Failed asserting that <integer:2> matches expected value <integer:1>.'
  severity: fail
  data:
    got: 2
    expected: 1
  ...
not ok 2 - Error: testError(FailureErrorTest)
1..2
```

Test Results (JSON)

The JavaScript Object Notation (JSON) [http://www.json.org/] is a lightweight data-interchange format. The following example shows the JSON messages generated for the tests in `ArrayTest`:

```
{"event":"suiteStart","suite":"ArrayTest","tests":2}
{"event":"test","suite":"ArrayTest",
 "test":"testNewArrayIsEmpty(ArrayTest)","status":"pass",
 "time":0.000460147858,"trace":[],"message":""}
{"event":"test","suite":"ArrayTest",
 "test":"testArrayContainsAnElement(ArrayTest)","status":"pass",
 "time":0.000422954559,"trace":[],"message":""}
```

The following JSON messages were generated for two tests, `testFailure` and `testError`, of a test case class named `FailureErrorTest` and show how failures and errors are denoted.

```
{"event":"suiteStart","suite":"FailureErrorTest","tests":2}
{"event":"test","suite":"FailureErrorTest",
 "test":"testFailure(FailureErrorTest)","status":"fail",
 "time":0.0082459449768066,"trace":[],
 "message":"Failed asserting that <integer:2> is equal to <integer:1>."}
```

```
{"event":"test","suite":"FailureErrorTest",
 "test":"testError(FailureErrorTest)","status":"error",
 "time":0.0083680152893066,"trace":[],"message":""}
```

Code Coverage (XML)

The XML format for code coverage information logging produced by PHPUnit is loosely based upon the one used by Clover [http://www.atlassian.com/software/clover/]. The following example shows the XML logfile generated for the tests in `BankAccountTest`:

```
<?xml version="1.0" encoding="UTF-8"?>
<coverage generated="1184835473" phpunit="3.6.0">
  <project name="BankAccountTest" timestamp="1184835473">
    <file name="/home/sb/BankAccount.php">
      <class name="BankAccountException">
        <metrics methods="0" coveredmethods="0" statements="0"
                 coveredstatements="0" elements="0" coveredelements="0"/>
      </class>
      <class name="BankAccount">
        <metrics methods="4" coveredmethods="4" statements="13"
                 coveredstatements="5" elements="17" coveredelements="9"/>
      </class>
      <line num="77" type="method" count="3"/>
      <line num="79" type="stmt" count="3"/>
      <line num="89" type="method" count="2"/>
      <line num="91" type="stmt" count="2"/>
      <line num="92" type="stmt" count="0"/>
      <line num="93" type="stmt" count="0"/>
      <line num="94" type="stmt" count="2"/>
      <line num="96" type="stmt" count="0"/>
      <line num="105" type="method" count="1"/>
      <line num="107" type="stmt" count="1"/>
      <line num="109" type="stmt" count="0"/>
      <line num="119" type="method" count="1"/>
      <line num="121" type="stmt" count="1"/>
      <line num="123" type="stmt" count="0"/>
      <metrics loc="126" ncloc="37" classes="2" methods="4" coveredmethods="4"
               statements="13" coveredstatements="5" elements="17"
               coveredelements="9"/>
    </file>
    <metrics files="1" loc="126" ncloc="37" classes="2" methods="4"
             coveredmethods="4" statements="13" coveredstatements="5"
             elements="17" coveredelements="9"/>
  </project>
</coverage>
```

Code Coverage (TEXT)

Human readable code coverage output for the command-line or a text file. The aim of this output format is to provide a quick coverage overview while working on a small set of classes. For bigger projects this output can be useful to get an quick overview of the projects coverage or when used with the `--filter` functionality. When used from the command-line by writing to `php://stdout` this will honor the `--colors` setting. Writing to standard out is the default option when used from the command-line. By default this will only show files that have at least one covered line. This can only be changed via the `showUncoveredFiles` xml configuration option. See the section called “Logging”. By default all files and their coverage status are shown in the detailed report. This can be changed via the `showOnlySummary` xml configuration option.

Chapter 14. Extending PHPUnit

PHPUnit can be extended in various ways to make the writing of tests easier and customize the feedback you get from running tests. Here are common starting points to extend PHPUnit.

Subclass PHPUnit\Framework\TestCase

Write custom assertions and utility methods in an abstract subclass of `PHPUnit\Framework\TestCase` and derive your test case classes from that class. This is one of the easiest ways to extend PHPUnit.

Write custom assertions

When writing custom assertions it is the best practice to follow how PHPUnit's own assertions are implemented. As you can see in Example 14.1, “The assertTrue() and isTrue() methods of the PHPUnit_Framework_Assert class”, the `assertTrue()` method is just a wrapper around the `isTrue()` and `assertThat()` methods: `isTrue()` creates a matcher object that is passed on to `assertThat()` for evaluation.

Example 14.1. The assertTrue() and isTrue() methods of the PHPUnit_Framework_Assert class

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

abstract class PHPUnit_Framework_Assert
{
    // ...

    /**
     * Asserts that a condition is true.
     *
     * @param  boolean $condition
     * @param  string  $message
     * @throws PHPUnit_Framework_AssertionFailedError
     */
    public static function assertTrue($condition, $message = '')
    {
        self::assertThat($condition, self::isTrue(), $message);
    }

    // ...

    /**
     * Returns a PHPUnit_Framework_Constraint_IsTrue matcher object.
     *
     * @return PHPUnit_Framework_Constraint_IsTrue
     * @since  Method available since Release 3.3.0
     */
    public static function isTrue()
    {
        return new PHPUnit_Framework_Constraint_IsTrue;
    }

    // ...
}?>
```

Example 14.2, “The PHPUnit_Framework_Constraint_IsTrue class” shows how `PHPUnit_Framework_Constraint_IsTrue` extends the abstract base class for matcher objects (or constraints), `PHPUnit_Framework_Constraint`.

Example 14.2. The PHPUnit_Framework_Constraint_IsTrue class

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

class PHPUnit_Framework_Constraint_IsTrue extends PHPUnit_Framework_Constraint
{
    /**
     * Evaluates the constraint for parameter $other. Returns true if the
     * constraint is met, false otherwise.
     *
     * @param mixed $other Value or object to evaluate.
     * @return bool
     */
    public function matches($other)
    {
        return $other === true;
    }

    /**
     * Returns a string representation of the constraint.
     *
     * @return string
     */
    public function toString()
    {
        return 'is true';
    }
}?>
```

The effort of implementing the `assertTrue()` and `isTrue()` methods as well as the `PHPUnit_Framework_Constraint_IsTrue` class yields the benefit that `assertThat()` automatically takes care of evaluating the assertion and bookkeeping tasks such as counting it for statistics. Furthermore, the `isTrue()` method can be used as a matcher when configuring mock objects.

Implement PHPUnit_Framework_TestListener

Example 14.3, “A simple test listener” shows a simple implementation of the `PHPUnit_Framework_TestListener` interface.

Example 14.3. A simple test listener

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

class SimpleTestListener implements PHPUnit_Framework_TestListener
{
    public function addError(PHPUnit_Framework_Test $test, Exception $e, $time)
    {
        printf("Error while running test '%s'.\n", $test->getName());
    }

    public function addFailure(PHPUnit_Framework_Test $test, PHPUnit_Framework_Assertion
    {
        printf("Test '%s' failed.\n", $test->getName());
    }
```

```
    public function addIncompleteTest(PHPUnit_Framework_Test $test, Exception $e, $time)
    {
        printf("Test '%s' is incomplete.\n", $test->getName());
    }

    public function addRiskyTest(PHPUnit_Framework_Test $test, Exception $e, $time)
    {
        printf("Test '%s' is deemed risky.\n", $test->getName());
    }

    public function addSkippedTest(PHPUnit_Framework_Test $test, Exception $e, $time)
    {
        printf("Test '%s' has been skipped.\n", $test->getName());
    }

    public function startTest(PHPUnit_Framework_Test $test)
    {
        printf("Test '%s' started.\n", $test->getName());
    }

    public function endTest(PHPUnit_Framework_Test $test, $time)
    {
        printf("Test '%s' ended.\n", $test->getName());
    }

    public function startTestSuite(PHPUnit_Framework_TestSuite $suite)
    {
        printf("TestSuite '%s' started.\n", $suite->getName());
    }

    public function endTestSuite(PHPUnit_Framework_TestSuite $suite)
    {
        printf("TestSuite '%s' ended.\n", $suite->getName());
    }
}
?>
```

Example 14.4, “Using base test listener” shows how to subclass the `PHPUnit_Framework_BaseTestListener` abstract class, which lets you specify only the interface methods that are interesting for your use case, while providing empty implementations for all the others.

Example 14.4. Using base test listener

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

class ShortTestListener extends PHPUnit_Framework_BaseTestListener
{
    public function endTest(PHPUnit_Framework_Test $test, $time)
    {
        printf("Test '%s' ended.\n", $test->getName());
    }
}
?>
```

In the section called “Test Listeners” you can see how to configure PHPUnit to attach your test listener to the test execution.

Subclass PHPUnit_Extensions_TestDecorator

You can wrap test cases or test suites in a subclass of `PHPUnit_Extensions_TestDecorator` and use the Decorator design pattern to perform some actions before and after the test runs.

PHPUnit ships with one concrete test decorator: `PHPUnit_Extensions_RepeatedTest`. It is used to run a test repeatedly and only count it as a success if all iterations are successful.

Example 14.5, "The RepeatedTest Decorator" shows a cut-down version of the `PHPUnit_Extensions_RepeatedTest` test decorator that illustrates how to write your own test decorators.

Example 14.5. The RepeatedTest Decorator

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

require_once 'PHPUnit/Extensions/TestDecorator.php';

class PHPUnit_Extensions_RepeatedTest extends PHPUnit_Extensions_TestDecorator
{
    private $timesRepeat = 1;

    public function __construct(PHPUnit_Framework_Test $test, $timesRepeat = 1)
    {
        parent::__construct($test);

        if (is_integer($timesRepeat) &&
            $timesRepeat >= 0) {
            $this->timesRepeat = $timesRepeat;
        }
    }

    public function count()
    {
        return $this->timesRepeat * $this->test->count();
    }

    public function run(PHPUnit_Framework_TestResult $result = null)
    {
        if ($result === null) {
            $result = $this->createResult();
        }

        for ($i = 0; $i < $this->timesRepeat && !$result->shouldStop(); $i++) {
            $this->test->run($result);
        }

        return $result;
    }
}
?>
```

Implement PHPUnit_Framework_Test

The `PHPUnit_Framework_Test` interface is narrow and easy to implement. You can write an implementation of `PHPUnit_Framework_Test` that is simpler than `PHPUnit\Framework\TestCase` and that runs *data-driven tests*, for instance.

Example 14.6, “A data-driven test” shows a data-driven test case class that compares values from a file with Comma-Separated Values (CSV). Each line of such a file looks like `foo;bar`, where the first value is the one we expect and the second value is the actual one.

Example 14.6. A data-driven test

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

class DataDrivenTest implements PHPUnit_Framework_Test
{
    private $lines;

    public function __construct($dataFile)
    {
        $this->lines = file($dataFile);
    }

    public function count()
    {
        return 1;
    }

    public function run(PHPUnit_Framework_TestResult $result = null)
    {
        if ($result === null) {
            $result = new PHPUnit_Framework_TestResult;
        }

        foreach ($this->lines as $line) {
            $result->startTest($this);
            PHP_Timer::start();
            $stopTime = null;

            list($expected, $actual) = explode(';', $line);

            try {
                PHPUnit_Framework_Assert::assertEquals(
                  trim($expected), trim($actual)
                );
            }

            catch (PHPUnit_Framework_AssertionFailedError $e) {
                $stopTime = PHP_Timer::stop();
                $result->addFailure($this, $e, $stopTime);
            }

            catch (Exception $e) {
                $stopTime = PHP_Timer::stop();
                $result->addError($this, $e, $stopTime);
            }

            if ($stopTime === null) {
                $stopTime = PHP_Timer::stop();
            }

            $result->endTest($this, $stopTime);
        }

        return $result;
    }
}
```

```
$test = new DataDrivenTest('data_file.csv');
$result = PHPUnit_TextUI_TestRunner::run($test);
?>
```

```
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

.F

Time: 0 seconds

There was 1 failure:

1) DataDrivenTest
Failed asserting that two strings are equal.
expected string <bar>
difference      <  x>
got string      <baz>
/home/sb/DataDrivenTest.php:32
/home/sb/DataDrivenTest.php:53

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

Appendix A. Assertions

This appendix lists the various assertion methods that are available.

assertArrayHasKey()

```
assertArrayHasKey(mixed $key, array $array[, string $message = ''])
```

Reports an error identified by `$message` if `$array` does not have the `$key`.

`assertArrayNotHasKey()` is the inverse of this assertion and takes the same arguments.

Example A.1. Usage of assertArrayHasKey()

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

class ArrayHasKeyTest extends TestCase
{
    public function testFailure()
    {
        $this->assertArrayHasKey('foo', ['bar' => 'baz']);
    }
}
?>
```

```
phpunit ArrayHasKeyTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ArrayHasKeyTest::testFailure
Failed asserting that an array has the key 'foo'.

/home/sb/ArrayHasKeyTest.php:6

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

assertClassHasAttribute()

```
assertClassHasAttribute(string $attributeName, string $className[,
string $message = ''])
```

Reports an error identified by `$message` if `$className::attributeName` does not exist.

`assertClassNotHasAttribute()` is the inverse of this assertion and takes the same arguments.

Example A.2. Usage of assertClassHasAttribute()

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

```
class ClassHasAttributeTest extends TestCase
{
    public function testFailure()
    {
        $this->assertClassHasAttribute('foo', stdClass::class);
    }
}
?>
```

```
phpunit ClassHasAttributeTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ClassHasAttributeTest::testFailure
Failed asserting that class "stdClass" has attribute "foo".

/home/sb/ClassHasAttributeTest.php:6

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

assertArraySubset()

`assertArraySubset(array $subset, array $array[, bool $strict = '', string $message = ''])`

Reports an error identified by `$message` if `$array` does not contains the `$subset`.

`$strict` is a flag used to compare the identity of objects within arrays.

Example A.3. Usage of assertArraySubset()

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

class ArraySubsetTest extends TestCase
{
    public function testFailure()
    {
        $this->assertArraySubset(['config' => ['key-a', 'key-b']], ['config' => ['key-a'
    }
}
?>
```

```
phpunit ArrayHasKeyTest
PHPUnit 4.4.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) Epilog\EpilogTest::testNoFollowOption
Failed asserting that an array has the subset Array &0 (
```

```
    'config' => Array &1 (
        0 => 'key-a'
        1 => 'key-b'
    )
).

/home/sb/ArraySubsetTest.php:6

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

assertClassHasStaticAttribute()

`assertClassHasStaticAttribute(string $attributeName, string $class-Name[, string $message = ''])`

Reports an error identified by `$message` if `$className::attributeName` does not exist.

`assertClassNotHasStaticAttribute()` is the inverse of this assertion and takes the same arguments.

Example A.4. Usage of assertClassHasStaticAttribute()

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

class ClassHasStaticAttributeTest extends TestCase
{
    public function testFailure()
    {
        $this->assertClassHasStaticAttribute('foo', stdClass::class);
    }
}
?>
```

```
phpunit ClassHasStaticAttributeTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ClassHasStaticAttributeTest::testFailure
Failed asserting that class "stdClass" has static attribute "foo".

/home/sb/ClassHasStaticAttributeTest.php:6

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

assertContains()

`assertContains(mixed $needle, Iterator|array $haystack[, string $message = ''])`

Reports an error identified by `$message` if `$needle` is not an element of `$haystack`.

`assertNotContains()` is the inverse of this assertion and takes the same arguments.

`assertAttributeContains()` and `assertAttributeNotContains()` are convenience wrappers that use a `public`, `protected`, or `private` attribute of a class or object as the haystack.

Example A.5. Usage of assertContains()

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

class ContainsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertContains(4, [1, 2, 3]);
    }
}
?>
```

```
phpunit ContainsTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ContainsTest::testFailure
Failed asserting that an array contains 4.

/home/sb/ContainsTest.php:6

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

`assertContains(string $needle, string $haystack[, string $message = '', boolean $ignoreCase = false])`

Reports an error identified by `$message` if `$needle` is not a substring of `$haystack`.

If `$ignoreCase` is `true`, the test will be case insensitive.

Example A.6. Usage of assertContains()

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

class ContainsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertContains('baz', 'foobar');
    }
}
?>
```

```
phpunit ContainsTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb
```

```
There was 1 failure:

1) ContainsTest::testFailure
Failed asserting that 'foobar' contains "baz".

/home/sb/ContainsTest.php:6

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

Example A.7. Usage of assertContains() with $ignoreCase

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

class ContainsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertContains('foo', 'FooBar');
    }

    public function testOK()
    {
        $this->assertContains('foo', 'FooBar', '', true);
    }
}
?>
```

```
phpunit ContainsTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F.

Time: 0 seconds, Memory: 2.75Mb

There was 1 failure:

1) ContainsTest::testFailure
Failed asserting that 'FooBar' contains "foo".

/home/sb/ContainsTest.php:6

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

assertContainsOnly()

`assertContainsOnly(string $type, Iterator|array $haystack[, boolean $isNativeType = null, string $message = ''])`

Reports an error identified by `$message` if `$haystack` does not contain only variables of type `$type`.

`$isNativeType` is a flag used to indicate whether `$type` is a native PHP type or not.

`assertNotContainsOnly()` is the inverse of this assertion and takes the same arguments.

`assertAttributeContainsOnly()` and `assertAttributeNotContainsOnly()` are convenience wrappers that use a `public`, `protected`, or `private` attribute of a class or object as the haystack.

Example A.8. Usage of assertContainsOnly()

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

class ContainsOnlyTest extends TestCase
{
    public function testFailure()
    {
        $this->assertContainsOnly('string', ['1', '2', 3]);
    }
}
?>
```

```
phpunit ContainsOnlyTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ContainsOnlyTest::testFailure
Failed asserting that Array (
    0 => '1'
    1 => '2'
    2 => 3
) contains only values of type "string".

/home/sb/ContainsOnlyTest.php:6

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

assertContainsOnlyInstancesOf()

`assertContainsOnlyInstancesOf(string $classname, Traversable|array $haystack[, string $message = ''])`

Reports an error identified by `$message` if `$haystack` does not contain only instances of class `$classname`.

Example A.9. Usage of assertContainsOnlyInstancesOf()

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

class ContainsOnlyInstancesOfTest extends TestCase
{
    public function testFailure()
    {
        $this->assertContainsOnlyInstancesOf(
            Foo::class,
            [new Foo, new Bar, new Foo]
        );
    }
}
?>
```

```
phpunit ContainsOnlyInstancesOfTest
```

```
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ContainsOnlyInstancesOfTest::testFailure
Failed asserting that Array ([0]=> Bar Object(...)) is an instance of class "Foo".

/home/sb/ContainsOnlyInstancesOfTest.php:6

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

assertCount()

`assertCount($expectedCount, $haystack[, string $message = ''])`

Reports an error identified by `$message` if the number of elements in `$haystack` is not `$expectedCount`.

`assertNotCount()` is the inverse of this assertion and takes the same arguments.

Example A.10. Usage of assertCount()

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

class CountTest extends TestCase
{
    public function testFailure()
    {
        $this->assertCount(0, ['foo']);
    }
}
?>
```

```
phpunit CountTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) CountTest::testFailure
Failed asserting that actual size 1 matches expected size 0.

/home/sb/CountTest.php:6

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

assertEmpty()

`assertEmpty(mixed $actual[, string $message = ''])`

Reports an error identified by `$message` if `$actual` is not empty.

`assertNotEmpty()` is the inverse of this assertion and takes the same arguments.

`assertAttributeEmpty()` and `assertAttributeNotEmpty()` are convenience wrappers that can be applied to a `public`, `protected`, or `private` attribute of a class or object.

Example A.11. Usage of assertEmpty()

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

class EmptyTest extends TestCase
{
    public function testFailure()
    {
        $this->assertEmpty(['foo']);
    }
}
?>
```

```
phpunit EmptyTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) EmptyTest::testFailure
Failed asserting that an array is empty.

/home/sb/EmptyTest.php:6

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

assertEqualXMLStructure()

`assertEqualXMLStructure(DOMElement $expectedElement, DOMElement $actualElement[, boolean $checkAttributes = false, string $message = ''])`

Reports an error identified by `$message` if the XML Structure of the DOMElement in `$actualElement` is not equal to the XML structure of the DOMElement in `$expectedElement`.

Example A.12. Usage of assertEqualXMLStructure()

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

class EqualXMLStructureTest extends TestCase
{
    public function testFailureWithDifferentNodeNames()
    {
        $expected = new DOMElement('foo');
        $actual = new DOMElement('bar');

        $this->assertEqualXMLStructure($expected, $actual);
    }

    public function testFailureWithDifferentNodeAttributes()
```

```
    {
        $expected = new DOMDocument;
        $expected->loadXML('<foo bar="true" />');

        $actual = new DOMDocument;
        $actual->loadXML('<foo/>');

        $this->assertEqualXMLStructure(
          $expected->firstChild, $actual->firstChild, true
        );
    }

    public function testFailureWithDifferentChildrenCount()
    {
        $expected = new DOMDocument;
        $expected->loadXML('<foo><bar/><bar/><bar/></foo>');

        $actual = new DOMDocument;
        $actual->loadXML('<foo><bar/></foo>');

        $this->assertEqualXMLStructure(
          $expected->firstChild, $actual->firstChild
        );
    }

    public function testFailureWithDifferentChildren()
    {
        $expected = new DOMDocument;
        $expected->loadXML('<foo><bar/><bar/><bar/></foo>');

        $actual = new DOMDocument;
        $actual->loadXML('<foo><baz/><baz/><baz/></foo>');

        $this->assertEqualXMLStructure(
          $expected->firstChild, $actual->firstChild
        );
    }
}
?>
```

```
phpunit EqualXMLStructureTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

FFFF

Time: 0 seconds, Memory: 5.75Mb

There were 4 failures:

1) EqualXMLStructureTest::testFailureWithDifferentNodeNames
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'foo'
+'bar'

/home/sb/EqualXMLStructureTest.php:9

2) EqualXMLStructureTest::testFailureWithDifferentNodeAttributes
Number of attributes on node "foo" does not match
Failed asserting that 0 matches expected 1.

/home/sb/EqualXMLStructureTest.php:22
```

```
3) EqualXMLStructureTest::testFailureWithDifferentChildrenCount
Number of child nodes of "foo" differs
Failed asserting that 1 matches expected 3.

/home/sb/EqualXMLStructureTest.php:35

4) EqualXMLStructureTest::testFailureWithDifferentChildren
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'bar'
+'baz'

/home/sb/EqualXMLStructureTest.php:48

FAILURES!
Tests: 4, Assertions: 8, Failures: 4.
```

assertEquals()

`assertEquals(mixed $expected, mixed $actual[, string $message = ''])`

Reports an error identified by `$message` if the two variables `$expected` and `$actual` are not equal.

`assertNotEquals()` is the inverse of this assertion and takes the same arguments.

`assertAttributeEquals()` and `assertAttributeNotEquals()` are convenience wrappers that use a `public`, `protected`, or `private` attribute of a class or object as the actual value.

Example A.13. Usage of assertEquals()

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

class EqualsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertEquals(1, 0);
    }

    public function testFailure2()
    {
        $this->assertEquals('bar', 'baz');
    }

    public function testFailure3()
    {
        $this->assertEquals("foo\nbar\nbaz\n", "foo\nbah\nbaz\n");
    }
}
?>
```

```
phpunit EqualsTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

FFF

Time: 0 seconds, Memory: 5.25Mb
```

```
There were 3 failures:

1) EqualsTest::testFailure
Failed asserting that 0 matches expected 1.

/home/sb/EqualsTest.php:6

2) EqualsTest::testFailure2
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'bar'
+'baz'

/home/sb/EqualsTest.php:11

3) EqualsTest::testFailure3
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
 'foo
-bar
+bah
 baz
 '

/home/sb/EqualsTest.php:16

FAILURES!
Tests: 3, Assertions: 3, Failures: 3.
```

More specialized comparisons are used for specific argument types for `$expected` and `$actual`, see below.

`assertEquals(float $expected, float $actual[, string $message = '', float $delta = 0])`

Reports an error identified by `$message` if the two floats `$expected` and `$actual` are not within `$delta` of each other.

Please read "What Every Computer Scientist Should Know About Floating-Point Arithmetic [http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html]" to understand why `$delta` is necessary.

Example A.14. Usage of assertEquals() with floats

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

class EqualsTest extends TestCase
{
    public function testSuccess()
    {
        $this->assertEquals(1.0, 1.1, '', 0.2);
    }

    public function testFailure()
    {
        $this->assertEquals(1.0, 1.1);
    }
```

```
}
?>
```

```
phpunit EqualsTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

.F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

1) EqualsTest::testFailure
Failed asserting that 1.1 matches expected 1.0.

/home/sb/EqualsTest.php:11

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

`assertEquals(DOMDocument $expected, DOMDocument $actual[, string $message = ''])`

Reports an error identified by `$message` if the uncommented canonical form of the XML documents represented by the two DOMDocument objects `$expected` and `$actual` are not equal.

Example A.15. Usage of assertEquals() with DOMDocument objects

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

class EqualsTest extends TestCase
{
    public function testFailure()
    {
        $expected = new DOMDocument;
        $expected->loadXML('<foo><bar/></foo>');

        $actual = new DOMDocument;
        $actual->loadXML('<bar><foo/></bar>');

        $this->assertEquals($expected, $actual);
    }
}
?>
```

```
phpunit EqualsTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) EqualsTest::testFailure
Failed asserting that two DOM documents are equal.
--- Expected
+++ Actual
@@ @@
 <?xml version="1.0"?>
-<foo>
-  <bar/>
```

```
-</foo>
+<bar>
+  <foo/>
+</bar>

/home/sb/EqualsTest.php:12

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

`assertEquals(object $expected, object $actual[, string $message = ''])`

Reports an error identified by `$message` if the two objects `$expected` and `$actual` do not have equal attribute values.

Example A.16. Usage of assertEquals() with objects

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

class EqualsTest extends TestCase
{
    public function testFailure()
    {
        $expected = new stdClass;
        $expected->foo = 'foo';
        $expected->bar = 'bar';

        $actual = new stdClass;
        $actual->foo = 'bar';
        $actual->baz = 'bar';

        $this->assertEquals($expected, $actual);
    }
}
?>
```

```
phpunit EqualsTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) EqualsTest::testFailure
Failed asserting that two objects are equal.
--- Expected
+++ Actual
@@ @@
 stdClass Object (
-    'foo' => 'foo'
-    'bar' => 'bar'
+    'foo' => 'bar'
+    'baz' => 'bar'
 )

/home/sb/EqualsTest.php:14

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

```
assertEquals(array $expected, array $actual[, string $message = ''])
```

Reports an error identified by `$message` if the two arrays `$expected` and `$actual` are not equal.

Example A.17. Usage of assertEquals() with arrays

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

class EqualsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertEquals(['a', 'b', 'c'], ['a', 'c', 'd']);
    }
}
?>
```

```
phpunit EqualsTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) EqualsTest::testFailure
Failed asserting that two arrays are equal.
--- Expected
+++ Actual
@@ @@
 Array (
     0 => 'a'
-    1 => 'b'
-    2 => 'c'
+    1 => 'c'
+    2 => 'd'
 )

/home/sb/EqualsTest.php:6

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

assertFalse()

```
assertFalse(bool $condition[, string $message = ''])
```

Reports an error identified by `$message` if `$condition` is `true`.

`assertNotFalse()` is the inverse of this assertion and takes the same arguments.

Example A.18. Usage of assertFalse()

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

class FalseTest extends TestCase
{
    public function testFailure()
    {
```

```
        $this->assertFalse(true);
    }
}
?>
```

```
phpunit FalseTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) FalseTest::testFailure
Failed asserting that true is false.

/home/sb/FalseTest.php:6

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

assertFileEquals()

`assertFileEquals(string $expected, string $actual[, string $message = ''])`

Reports an error identified by `$message` if the file specified by `$expected` does not have the same contents as the file specified by `$actual`.

`assertFileNotEquals()` is the inverse of this assertion and takes the same arguments.

Example A.19. Usage of assertFileEquals()

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

class FileEqualsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertFileEquals('/home/sb/expected', '/home/sb/actual');
    }
}
?>
```

```
phpunit FileEqualsTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) FileEqualsTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'expected
+'actual
```

```
'

/home/sb/FileEqualsTest.php:6

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

assertFileExists()

`assertFileExists(string $filename[, string $message = ''])`

Reports an error identified by `$message` if the file specified by `$filename` does not exist.

`assertFileNotExists()` is the inverse of this assertion and takes the same arguments.

Example A.20. Usage of assertFileExists()

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

class FileExistsTest extends TestCase
{
    public function testFailure()
    {
        $this->assertFileExists('/path/to/file');
    }
}
?>
```

```
phpunit FileExistsTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) FileExistsTest::testFailure
Failed asserting that file "/path/to/file" exists.

/home/sb/FileExistsTest.php:6

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

assertGreaterThan()

`assertGreaterThan(mixed $expected, mixed $actual[, string $message = ''])`

Reports an error identified by `$message` if the value of `$actual` is not greater than the value of `$expected`.

`assertAttributeGreaterThan()` is a convenience wrapper that uses a `public`, `protected`, or `private` attribute of a class or object as the actual value.

Example A.21. Usage of assertGreaterThan()

```
<?php
```

```
use PHPUnit\Framework\TestCase;

class GreaterThanTest extends TestCase
{
    public function testFailure()
    {
        $this->assertGreaterThan(2, 1);
    }
}
?>
```

```
phpunit GreaterThanTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) GreaterThanTest::testFailure
Failed asserting that 1 is greater than 2.

/home/sb/GreaterThanTest.php:6

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

assertGreaterThanOrEqual()

`assertGreaterThanOrEqual(mixed $expected, mixed $actual[, string $message = ''])`

Reports an error identified by `$message` if the value of `$actual` is not greater than or equal to the value of `$expected`.

`assertAttributeGreaterThanOrEqual()` is a convenience wrapper that uses a `public`, `protected`, or `private` attribute of a class or object as the actual value.

Example A.22. Usage of assertGreaterThanOrEqual()

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

class GreatThanOrEqualTest extends TestCase
{
    public function testFailure()
    {
        $this->assertGreaterThanOrEqual(2, 1);
    }
}
?>
```

```
phpunit GreaterThanOrEqualTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:
```

```
1) GreatThanOrEqualTest::testFailure
Failed asserting that 1 is equal to 2 or is greater than 2.

/home/sb/GreaterThanOrEqualTest.php:6

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

assertInfinite()

`assertInfinite(mixed $variable[, string $message = ''])`

Reports an error identified by `$message` if `$variable` is not `INF`.

`assertFinite()` is the inverse of this assertion and takes the same arguments.

Example A.23. Usage of assertInfinite()

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

class InfiniteTest extends TestCase
{
    public function testFailure()
    {
        $this->assertInfinite(1);
    }
}
?>
```

```
phpunit InfiniteTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) InfiniteTest::testFailure
Failed asserting that 1 is infinite.

/home/sb/InfiniteTest.php:6

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

assertInstanceOf()

`assertInstanceOf($expected, $actual[, $message = ''])`

Reports an error identified by `$message` if `$actual` is not an instance of `$expected`.

`assertNotInstanceOf()` is the inverse of this assertion and takes the same arguments.

`assertAttributeInstanceOf()` and `assertAttributeNotInstanceOf()` are convenience wrappers that can be applied to a `public`, `protected`, or `private` attribute of a class or object.

Example A.24. Usage of assertInstanceOf()

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

class InstanceOfTest extends TestCase
{
    public function testFailure()
    {
        $this->assertInstanceOf(RuntimeException::class, new Exception);
    }
}
?>
```

```
phpunit InstanceOfTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) InstanceOfTest::testFailure
Failed asserting that Exception Object (...) is an instance of class "RuntimeException".

/home/sb/InstanceOfTest.php:6

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

assertInternalType()

`assertInternalType($expected, $actual[, $message = ''])`

Reports an error identified by `$message` if `$actual` is not of the `$expected` type.

`assertNotInternalType()` is the inverse of this assertion and takes the same arguments.

`assertAttributeInternalType()` and `assertAttributeNotInternalType()` are convenience wrappers that can be applied to a `public`, `protected`, or `private` attribute of a class or object.

Example A.25. Usage of assertInternalType()

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

class InternalTypeTest extends TestCase
{
    public function testFailure()
    {
        $this->assertInternalType('string', 42);
    }
}
?>
```

```
phpunit InternalTypeTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F
```

```
Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) InternalTypeTest::testFailure
Failed asserting that 42 is of type "string".

/home/sb/InternalTypeTest.php:6

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

assertJsonFileEqualsJsonFile()

`assertJsonFileEqualsJsonFile(mixed $expectedFile, mixed $actualFile[, string $message = ''])`

Reports an error identified by `$message` if the value of `$actualFile` does not match the value of `$expectedFile`.

Example A.26. Usage of assertJsonFileEqualsJsonFile()

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

class JsonFileEqualsJsonFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertJsonFileEqualsJsonFile(
          'path/to/fixture/file', 'path/to/actual/file');
    }
}
?>
```

```
phpunit JsonFileEqualsJsonFileTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) JsonFileEqualsJsonFile::testFailure
Failed asserting that '{"Mascott":"Tux"}' matches JSON string "["Mascott", "Tux", "OS",

/home/sb/JsonFileEqualsJsonFileTest.php:5

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

assertJsonStringEqualsJsonFile()

`assertJsonStringEqualsJsonFile(mixed $expectedFile, mixed $actualJson[, string $message = ''])`

Reports an error identified by `$message` if the value of `$actualJson` does not match the value of `$expectedFile`.

Example A.27. Usage of assertJsonStringEqualsJsonFile()

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

class JsonStringEqualsJsonFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertJsonStringEqualsJsonFile(
            'path/to/fixture/file', json_encode(['Mascott' => 'ux'])
        );
    }
}
?>
```

```
phpunit JsonStringEqualsJsonFileTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) JsonStringEqualsJsonFile::testFailure
Failed asserting that '{"Mascott":"ux"}' matches JSON string "{"Mascott":"Tux"}".

/home/sb/JsonStringEqualsJsonFileTest.php:5

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

assertJsonStringEqualsJsonString()

`assertJsonStringEqualsJsonString(mixed $expectedJson, mixed $actualJson[, string $message = ''])`

Reports an error identified by `$message` if the value of `$actualJson` does not match the value of `$expectedJson`.

Example A.28. Usage of assertJsonStringEqualsJsonString()

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

class JsonStringEqualsJsonStringTest extends TestCase
{
    public function testFailure()
    {
        $this->assertJsonStringEqualsJsonString(
            json_encode(['Mascott' => 'Tux']),
            json_encode(['Mascott' => 'ux'])
        );
    }
}
?>
```

```
phpunit JsonStringEqualsJsonStringTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.
```

```
F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) JsonStringEqualsJsonStringTest::testFailure
Failed asserting that two objects are equal.
--- Expected
+++ Actual
@@ @@
 stdClass Object (
 -    'Mascott' => 'Tux'
 +    'Mascott' => 'ux'
)

/home/sb/JsonStringEqualsJsonStringTest.php:5

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

assertLessThan()

`assertLessThan(mixed $expected, mixed $actual[, string $message = ''])`

Reports an error identified by `$message` if the value of `$actual` is not less than the value of `$expected`.

`assertAttributeLessThan()` is a convenience wrapper that uses a `public`, `protected`, or `private` attribute of a class or object as the actual value.

Example A.29. Usage of assertLessThan()

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

class LessThanTest extends TestCase
{
    public function testFailure()
    {
        $this->assertLessThan(1, 2);
    }
}
?>
```

```
phpunit LessThanTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) LessThanTest::testFailure
Failed asserting that 2 is less than 1.

/home/sb/LessThanTest.php:6
```

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

assertLessThanOrEqual()

`assertLessThanOrEqual(mixed $expected, mixed $actual[, string $message = ''])`

Reports an error identified by `$message` if the value of `$actual` is not less than or equal to the value of `$expected`.

`assertAttributeLessThanOrEqual()` is a convenience wrapper that uses a `public`, `protected`, or `private` attribute of a class or object as the actual value.

Example A.30. Usage of assertLessThanOrEqual()

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

class LessThanOrEqualTest extends TestCase
{
    public function testFailure()
    {
        $this->assertLessThanOrEqual(1, 2);
    }
}
?>
```

```
phpunit LessThanOrEqualTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) LessThanOrEqualTest::testFailure
Failed asserting that 2 is equal to 1 or is less than 1.

/home/sb/LessThanOrEqualTest.php:6

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

assertNan()

`assertNan(mixed $variable[, string $message = ''])`

Reports an error identified by `$message` if `$variable` is not `NAN`.

Example A.31. Usage of assertNan()

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

class NanTest extends TestCase
{
    public function testFailure()
```

```
    {
        $this->assertNan(1);
    }
}
?>
```

```
phpunit NanTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) NanTest::testFailure
Failed asserting that 1 is nan.

/home/sb/NanTest.php:6

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

assertNull()

`assertNull(mixed $variable[, string $message = ''])`

Reports an error identified by `$message` if `$variable` is not `null`.

`assertNotNull()` is the inverse of this assertion and takes the same arguments.

Example A.32. Usage of assertNull()

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

class NullTest extends TestCase
{
    public function testFailure()
    {
        $this->assertNull('foo');
    }
}
?>
```

```
phpunit NotNullTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) NullTest::testFailure
Failed asserting that 'foo' is null.

/home/sb/NotNullTest.php:6

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

assertObjectHasAttribute()

`assertObjectHasAttribute(string $attributeName, object $object[, string $message = ''])`

Reports an error identified by `$message` if `$object->attributeName` does not exist.

`assertObjectNotHasAttribute()` is the inverse of this assertion and takes the same arguments.

Example A.33. Usage of assertObjectHasAttribute()

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

class ObjectHasAttributeTest extends TestCase
{
    public function testFailure()
    {
        $this->assertObjectHasAttribute('foo', new stdClass);
    }
}
?>
```

```
phpunit ObjectHasAttributeTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ObjectHasAttributeTest::testFailure
Failed asserting that object of class "stdClass" has attribute "foo".

/home/sb/ObjectHasAttributeTest.php:6

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

assertRegExp()

`assertRegExp(string $pattern, string $string[, string $message = ''])`

Reports an error identified by `$message` if `$string` does not match the regular expression `$pattern`.

`assertNotRegExp()` is the inverse of this assertion and takes the same arguments.

Example A.34. Usage of assertRegExp()

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

class RegExpTest extends TestCase
{
    public function testFailure()
    {
        $this->assertRegExp('/foo/', 'bar');
```

```
    }
}
?>
```

```
phpunit RegExpTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) RegExpTest::testFailure
Failed asserting that 'bar' matches PCRE pattern "/foo/".

/home/sb/RegExpTest.php:6

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

assertStringMatchesFormat()

`assertStringMatchesFormat(string $format, string $string[, string $message = ''])`

Reports an error identified by `$message` if the `$string` does not match the `$format` string.

`assertStringNotMatchesFormat()` is the inverse of this assertion and takes the same arguments.

Example A.35. Usage of assertStringMatchesFormat()

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

class StringMatchesFormatTest extends TestCase
{
    public function testFailure()
    {
        $this->assertStringMatchesFormat('%i', 'foo');
    }
}
?>
```

```
phpunit StringMatchesFormatTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) StringMatchesFormatTest::testFailure
Failed asserting that 'foo' matches PCRE pattern "/^[+-]?\d+$/s".

/home/sb/StringMatchesFormatTest.php:6

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

The format string may contain the following placeholders:

- `%e`: Represents a directory separator, for example `/` on Linux.
- `%s`: One or more of anything (character or white space) except the end of line character.
- `%S`: Zero or more of anything (character or white space) except the end of line character.
- `%a`: One or more of anything (character or white space) including the end of line character.
- `%A`: Zero or more of anything (character or white space) including the end of line character.
- `%w`: Zero or more white space characters.
- `%i`: A signed integer value, for example `+3142`, `-3142`.
- `%d`: An unsigned integer value, for example `123456`.
- `%x`: One or more hexadecimal character. That is, characters in the range `0-9`, `a-f`, `A-F`.
- `%f`: A floating point number, for example: `3.142`, `-3.142`, `3.142E-10`, `3.142e+10`.
- `%c`: A single character of any sort.

assertStringMatchesFormatFile()

```
assertStringMatchesFormatFile(string $formatFile, string $string[,
string $message = ''])
```

Reports an error identified by `$message` if the `$string` does not match the contents of the `$formatFile`.

`assertStringNotMatchesFormatFile()` is the inverse of this assertion and takes the same arguments.

Example A.36. Usage of assertStringMatchesFormatFile()

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

class StringMatchesFormatFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertStringMatchesFormatFile('/path/to/expected.txt', 'foo');
    }
}
?>
```

```
phpunit StringMatchesFormatFileTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) StringMatchesFormatFileTest::testFailure
Failed asserting that 'foo' matches PCRE pattern "/^[+-]?\d+
$/s".

/home/sb/StringMatchesFormatFileTest.php:6
```

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

assertSame()

`assertSame(mixed $expected, mixed $actual[, string $message = ''])`

Reports an error identified by `$message` if the two variables `$expected` and `$actual` do not have the same type and value.

`assertNotSame()` is the inverse of this assertion and takes the same arguments.

`assertAttributeSame()` and `assertAttributeNotSame()` are convenience wrappers that use a `public`, `protected`, or `private` attribute of a class or object as the actual value.

Example A.37. Usage of assertSame()

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

class SameTest extends TestCase
{
    public function testFailure()
    {
        $this->assertSame('2204', 2204);
    }
}
?>
```

```
phpunit SameTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) SameTest::testFailure
Failed asserting that 2204 is identical to '2204'.

/home/sb/SameTest.php:6

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

`assertSame(object $expected, object $actual[, string $message = ''])`

Reports an error identified by `$message` if the two variables `$expected` and `$actual` do not reference the same object.

Example A.38. Usage of assertSame() with objects

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

class SameTest extends TestCase
{
    public function testFailure()
    {
```

```
        $this->assertSame(new stdClass, new stdClass);
    }
}
?>
```

```
phpunit SameTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) SameTest::testFailure
Failed asserting that two variables reference the same object.

/home/sb/SameTest.php:6

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

assertStringEndsWith()

`assertStringEndsWith(string $suffix, string $string[, string $message = ''])`

Reports an error identified by `$message` if the `$string` does not end with `$suffix`.

`assertStringEndsNotWith()` is the inverse of this assertion and takes the same arguments.

Example A.39. Usage of assertStringEndsWith()

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

class StringEndsWithTest extends TestCase
{
    public function testFailure()
    {
        $this->assertStringEndsWith('suffix', 'foo');
    }
}
?>
```

```
phpunit StringEndsWithTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 1 second, Memory: 5.00Mb

There was 1 failure:

1) StringEndsWithTest::testFailure
Failed asserting that 'foo' ends with "suffix".

/home/sb/StringEndsWithTest.php:6

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

assertStringEqualsFile()

```
assertStringEqualsFile(string $expectedFile, string $actualString[,
string $message = ''])
```

Reports an error identified by `$message` if the file specified by `$expectedFile` does not have `$actualString` as its contents.

`assertStringNotEqualsFile()` is the inverse of this assertion and takes the same arguments.

Example A.40. Usage of assertStringEqualsFile()

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

class StringEqualsFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertStringEqualsFile('/home/sb/expected', 'actual');
    }
}
?>
```

```
phpunit StringEqualsFileTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) StringEqualsFileTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'expected
-'
+'actual'

/home/sb/StringEqualsFileTest.php:6

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

assertStringStartsWith()

```
assertStringStartsWith(string $prefix, string $string[, string $mes-
sage = ''])
```

Reports an error identified by `$message` if the `$string` does not start with `$prefix`.

`assertStringStartsNotWith()` is the inverse of this assertion and takes the same arguments.

Example A.41. Usage of assertStringStartsWith()

```
<?php
```

```
use PHPUnit\Framework\TestCase;

class StringStartsWithTest extends TestCase
{
    public function testFailure()
    {
        $this->assertStringStartsWith('prefix', 'foo');
    }
}
?>
```

```
phpunit StringStartsWithTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) StringStartsWithTest::testFailure
Failed asserting that 'foo' starts with "prefix".

/home/sb/StringStartsWithTest.php:6

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

assertThat()

More complex assertions can be formulated using the `PHPUnit_Framework_Constraint` classes. They can be evaluated using the `assertThat()` method. Example A.42, "Usage of assertThat()" shows how the `logicalNot()` and `equalTo()` constraints can be used to express the same assertion as `assertNotEquals()`.

```
assertThat(mixed $value, PHPUnit_Framework_Constraint $constraint[,
$message = ''])
```

Reports an error identified by `$message` if the `$value` does not match the `$constraint`.

Example A.42. Usage of assertThat()

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

class BiscuitTest extends TestCase
{
    public function testEquals()
    {
        $theBiscuit = new Biscuit('Ginger');
        $myBiscuit  = new Biscuit('Ginger');

        $this->assertThat(
          $theBiscuit,
          $this->logicalNot(
            $this->equalTo($myBiscuit)
          )
        );
    }
}
?>
```

Table A.1, “Constraints” shows the available `PHPUnit_Framework_Constraint` classes.

Table A.1. Constraints

Constraint	Meaning
`PHPUnit_ Framework_ Constraint_ Attribute attribute(PHPUnit_ Framework_ Constraint $constraint, $attributeName)`	Constraint that applies another constraint to an attribute of a class or an object.
`PHPUnit_ Framework_ Constraint_ IsAnything anything()`	Constraint that accepts any input value.
`PHPUnit_ Framework_ Constraint_ ArrayHasKey arrayHasKey(mixed $key)`	Constraint that asserts that the array it is evaluated for has a given key.
`PHPUnit_ Framework_ Constraint_ TraversableContains contains(mixed $value)`	Constraint that asserts that the `array` or object that implements the `Iterator` interface it is evaluated for contains a given value.
`PHPUnit_ Framework_ Constraint_ TraversableContainsOnly containsOnly(string $type)`	Constraint that asserts that the `array` or object that implements the `Iterator` interface it is evaluated for contains only values of a given type.
`PHPUnit_ Framework_ Constraint_ TraversableContainsOnly containsOnlyInstancesOf(string $classname)`	Constraint that asserts that the `array` or object that implements the `Iterator` interface it is evaluated for contains only instances of a given classname.
`PHPUnit_ Framework_ Constraint_ IsEqual equalTo($value, $delta = 0, $maxDepth = 10)`	Constraint that checks if one value is equal to another.
`PHPUnit_ Framework_ Constraint_ Attribute attributeEqualTo($attributeName, $value, $delta = 0, $maxDepth = 10)`	Constraint that checks if a value is equal to an attribute of a class or of an object.
`PHPUnit_ Framework_ Constraint_ FileExists fileExists()`	Constraint that checks if the file(name) that it is evaluated for exists.
`PHPUnit_ Framework_ Constraint_ GreaterThan greaterThan(mixed $value)`	Constraint that asserts that the value it is evaluated for is greater than a given value.
`PHPUnit_ Framework_ Constraint_ Or greaterThanOrEqual(mixed $value)`	Constraint that asserts that the value it is evaluated for is greater than or equal to a given value.
`PHPUnit_ Framework_ Constraint_ ClassHasAttribute classHasAttribute(string $attributeName)`	Constraint that asserts that the class it is evaluated for has a given attribute.
`PHPUnit_ Framework_ Constraint_ ClassHasStaticAttribute classHasStaticAttribute(string $attributeName)`	Constraint that asserts that the class it is evaluated for has a given static attribute.
`PHPUnit_ Framework_ Constraint_ ObjectHasAttribute hasAttribute(string $attributeName)`	Constraint that asserts that the object it is evaluated for has a given attribute.

Constraint	Meaning
`PHPUnit_ Framework_ Constraint_ IsIdentical identicalTo(mixed $value)`	Constraint that asserts that one value is identical to another.
`PHPUnit_ Framework_ Constraint_ IsFalse isFalse()`	Constraint that asserts that the value it is evaluated is `false`.
`PHPUnit_ Framework_ Constraint_ IsInstanceOf isInstanceOf(string $className)`	Constraint that asserts that the object it is evaluated for is an instance of a given class.
`PHPUnit_ Framework_ Constraint_ IsNull isNull()`	Constraint that asserts that the value it is evaluated is `null`.
`PHPUnit_ Framework_ Constraint_ IsTrue isTrue()`	Constraint that asserts that the value it is evaluated is `true`.
`PHPUnit_ Framework_ Constraint_ IsType isType(string $type)`	Constraint that asserts that the value it is evaluated for is of a specified type.
`PHPUnit_ Framework_ Constraint_ LessThan lessThan(mixed $value)`	Constraint that asserts that the value it is evaluated for is smaller than a given value.
`PHPUnit_ Framework_ Constraint_ Or lessThanOrEqual(mixed $value)`	Constraint that asserts that the value it is evaluated for is smaller than or equal to a given value.
`logicalAnd()`	Logical AND.
`logicalNot(PHPUnit_ Framework_ Constraint $constraint)`	Logical NOT.
`logicalOr()`	Logical OR.
`logicalXor()`	Logical XOR.
`PHPUnit_ Framework_ Constraint_ PCREMatch matchesRegularExpression(string $pattern)`	Constraint that asserts that the string it is evaluated for matches a regular expression.
`PHPUnit_ Framework_ Constraint_ StringContains stringContains(string $string, bool $case)`	Constraint that asserts that the string it is evaluated for contains a given string.
`PHPUnit_ Framework_ Constraint_ StringEndsWith stringEndsWith(string $suffix)`	Constraint that asserts that the string it is evaluated for ends with a given suffix.
`PHPUnit_ Framework_ Constraint_ StringStartsWith stringStartsWith(string $prefix)`	Constraint that asserts that the string it is evaluated for starts with a given prefix.

assertTrue()

`assertTrue(bool $condition[, string $message = ''])`

Reports an error identified by `$message` if `$condition` is `false`.

`assertNotTrue()` is the inverse of this assertion and takes the same arguments.

Example A.43. Usage of assertTrue()

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

```
class TrueTest extends TestCase
{
    public function testFailure()
    {
        $this->assertTrue(false);
    }
}
?>
```

```
phpunit TrueTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) TrueTest::testFailure
Failed asserting that false is true.

/home/sb/TrueTest.php:6

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

assertXmlFileEqualsXmlFile()

`assertXmlFileEqualsXmlFile(string $expectedFile, string $actualFile[, string $message = ''])`

Reports an error identified by `$message` if the XML document in `$actualFile` is not equal to the XML document in `$expectedFile`.

`assertXmlFileNotEqualsXmlFile()` is the inverse of this assertion and takes the same arguments.

Example A.44. Usage of assertXmlFileEqualsXmlFile()

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

class XmlFileEqualsXmlFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertXmlFileEqualsXmlFile(
          '/home/sb/expected.xml', '/home/sb/actual.xml');
    }
}
?>
```

```
phpunit XmlFileEqualsXmlFileTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:
```

```
1) XmlFileEqualsXmlFileTest::testFailure
Failed asserting that two DOM documents are equal.
--- Expected
+++ Actual
@@ @@
 <?xml version="1.0"?>
 <foo>
-  <bar/>
+  <baz/>
 </foo>

/home/sb/XmlFileEqualsXmlFileTest.php:7

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

assertXmlStringEqualsXmlFile()

`assertXmlStringEqualsXmlFile(string $expectedFile, string $actualXml[, string $message = ''])`

Reports an error identified by `$message` if the XML document in `$actualXml` is not equal to the XML document in `$expectedFile`.

`assertXmlStringNotEqualsXmlFile()` is the inverse of this assertion and takes the same arguments.

Example A.45. Usage of assertXmlStringEqualsXmlFile()

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

class XmlStringEqualsXmlFileTest extends TestCase
{
    public function testFailure()
    {
        $this->assertXmlStringEqualsXmlFile(
          '/home/sb/expected.xml', '<foo><baz/></foo>');
    }
}
?>
```

```
phpunit XmlStringEqualsXmlFileTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) XmlStringEqualsXmlFileTest::testFailure
Failed asserting that two DOM documents are equal.
--- Expected
+++ Actual
@@ @@
 <?xml version="1.0"?>
 <foo>
-  <bar/>
+  <baz/>
 </foo>
```

```
/home/sb/XmlStringEqualsXmlFileTest.php:7

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

assertXmlStringEqualsXmlString()

`assertXmlStringEqualsXmlString(string $expectedXml, string $actualXml[, string $message = ''])`

Reports an error identified by `$message` if the XML document in `$actualXml` is not equal to the XML document in `$expectedXml`.

`assertXmlStringNotEqualsXmlString()` is the inverse of this assertion and takes the same arguments.

Example A.46. Usage of assertXmlStringEqualsXmlString()

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

class XmlStringEqualsXmlStringTest extends TestCase
{
    public function testFailure()
    {
        $this->assertXmlStringEqualsXmlString(
          '<foo><bar/></foo>', '<foo><baz/></foo>');
    }
}
?>
```

```
phpunit XmlStringEqualsXmlStringTest
PHPUnit 5.5.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) XmlStringEqualsXmlStringTest::testFailure
Failed asserting that two DOM documents are equal.
--- Expected
+++ Actual
@@ @@
 <?xml version="1.0"?>
 <foo>
-  <bar/>
+  <baz/>
 </foo>

/home/sb/XmlStringEqualsXmlStringTest.php:7

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

Appendix B. Annotations

An annotation is a special form of syntactic metadata that can be added to the source code of some programming languages. While PHP has no dedicated language feature for annotating source code, the usage of tags such as `@annotation arguments` in documentation block has been established in the PHP community to annotate source code. In PHP documentation blocks are reflective: they can be accessed through the Reflection API's `getDocComment()` method on the function, class, method, and attribute level. Applications such as PHPUnit use this information at runtime to configure their behaviour.

Note

A doc comment in PHP must start with `/**` and end with `*/`. Annotations in any other style of comment will be ignored.

This appendix shows all the varieties of annotations supported by PHPUnit.

@author

The `@author` annotation is an alias for the `@group` annotation (see the section called “@group”) and allows to filter tests based on their authors.

@after

The `@after` annotation can be used to specify methods that should be called after each test method in a test case class.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @after
     */
    public function tearDownSomeFixtures()
    {
        // ...
    }

    /**
     * @after
     */
    public function tearDownSomeOtherFixtures()
    {
        // ...
    }
}
```

@afterClass

The `@afterClass` annotation can be used to specify static methods that should be called after all test methods in a test class have been run to clean up shared fixtures.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
```

```
    /**
     * @afterClass
     */
    public static function tearDownSomeSharedFixtures()
    {
        // ...
    }

    /**
     * @afterClass
     */
    public static function tearDownSomeOtherSharedFixtures()
    {
        // ...
    }
}
```

@backupGlobals

The backup and restore operations for global variables can be completely disabled for all tests of a test case class like this

```
use PHPUnit\Framework\TestCase;

/**
 * @backupGlobals disabled
 */
class MyTest extends TestCase
{
    // ...
}
```

The `@backupGlobals` annotation can also be used on the test method level. This allows for a fine-grained configuration of the backup and restore operations:

```
use PHPUnit\Framework\TestCase;

/**
 * @backupGlobals disabled
 */
class MyTest extends TestCase
{
    /**
     * @backupGlobals enabled
     */
    public function testThatInteractsWithGlobalVariables()
    {
        // ...
    }
}
```

@backupStaticAttributes

The `@backupStaticAttributes` annotation can be used to back up all static property values in all declared classes before each test and restore them afterwards. It may be used at the test case class or test method level:

```
use PHPUnit\Framework\TestCase;

/**
```

```
 * @backupStaticAttributes enabled
 */
class MyTest extends TestCase
{
    /**
     * @backupStaticAttributes disabled
     */
    public function testThatInteractsWithStaticAttributes()
    {
        // ...
    }
}
```

Note

`@backupStaticAttributes` is limited by PHP internals and may cause unintended static values to persist and leak into subsequent tests in some circumstances.

See the section called “Global State” for details.

@before

The `@before` annotation can be used to specify methods that should be called before each test method in a test case class.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @before
     */
    public function setupSomeFixtures()
    {
        // ...
    }

    /**
     * @before
     */
    public function setupSomeOtherFixtures()
    {
        // ...
    }
}
```

@beforeClass

The `@beforeClass` annotation can be used to specify static methods that should be called before any test methods in a test class are run to set up shared fixtures.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @beforeClass
     */
    public static function setUpSomeSharedFixtures()
    {
```

```
        // ...
    }

    /**
     * @beforeClass
     */
    public static function setUpSomeOtherSharedFixtures()
    {
        // ...
    }
}
```

@codeCoverageIgnore*

The `@codeCoverageIgnore`, `@codeCoverageIgnoreStart` and `@codeCoverageIgnoreEnd` annotations can be used to exclude lines of code from the coverage analysis.

For usage see the section called “Ignoring Code Blocks”.

@covers

The `@covers` annotation can be used in the test code to specify which method(s) a test method wants to test:

```
/**
 * @covers BankAccount::getBalance
 */
public function testBalanceIsInitiallyZero()
{
    $this->assertEquals(0, $this->ba->getBalance());
}
```

If provided, only the code coverage information for the specified method(s) will be considered.

Table B.1, “Annotations for specifying which methods are covered by a test” shows the syntax of the `@covers` annotation.

Table B.1. Annotations for specifying which methods are covered by a test

Annotation	Description
`@covers ClassName::methodName`	Specifies that the annotated test method covers the specified method.
`@covers ClassName`	Specifies that the annotated test method covers all methods of a given class.
`@covers ClassName<extended>`	Specifies that the annotated test method covers all methods of a given class and its parent class(es) and interface(s).
`@covers ClassName::<public>`	Specifies that the annotated test method covers all public methods of a given class.
`@covers ClassName::<protected>`	Specifies that the annotated test method covers all protected methods of a given class.

Annotation	Description
`@covers ClassName::<private>`	Specifies that the annotated test method covers all private methods of a given class.
`@covers ClassName::<!public>`	Specifies that the annotated test method covers all methods of a given class that are not public.
`@covers ClassName::<!protected>`	Specifies that the annotated test method covers all methods of a given class that are not protected.
`@covers ClassName::<!private>`	Specifies that the annotated test method covers all methods of a given class that are not private.
`@covers ::functionName`	Specifies that the annotated test method covers the specified global function.

@coversDefaultClass

The `@coversDefaultClass` annotation can be used to specify a default namespace or class name. That way long names don't need to be repeated for every `@covers` annotation. See Example B.1, “Using @coversDefaultClass to shorten annotations”.

Example B.1. Using @coversDefaultClass to shorten annotations

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

/**
 * @coversDefaultClass \Foo\CoveredClass
 */
class CoversDefaultClassTest extends TestCase
{
    /**
     * @covers ::publicMethod
     */
    public function testSomething()
    {
        $o = new Foo\CoveredClass;
        $o->publicMethod();
    }
}
?>
```

@coversNothing

The `@coversNothing` annotation can be used in the test code to specify that no code coverage information will be recorded for the annotated test case.

This can be used for integration testing. See Example 11.3, “A test that specifies that no method should be covered” for an example.

The annotation can be used on the class and the method level and will override any `@covers` tags.

@dataProvider

A test method can accept arbitrary arguments. These arguments are to be provided by a data provider method (`provider()` in Example 2.5, “Using a data provider that returns an array of arrays”). The data provider method to be used is specified using the `@dataProvider` annotation.

See the section called “Data Providers” for more details.

@depends

PHPUnit supports the declaration of explicit dependencies between test methods. Such dependencies do not define the order in which the test methods are to be executed but they allow the returning of an instance of the test fixture by a producer and passing it to the dependent consumers. Example 2.2, “Using the `@depends` annotation to express dependencies” shows how to use the `@depends` annotation to express dependencies between test methods.

See the section called “Test Dependencies” for more details.

@expectedException

Example 2.10, “Using the expectException() method” shows how to use the `@expectedException` annotation to test whether an exception is thrown inside the tested code.

See the section called “Testing Exceptions” for more details.

@expectedExceptionCode

The `@expectedExceptionCode` annotation, in conjunction with the `@expectedException` allows making assertions on the error code of a thrown exception thus being able to narrow down a specific exception.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @expectedException     MyException
     * @expectedExceptionCode 20
     */
    public function testExceptionHasErrorcode20()
    {
        throw new MyException('Some Message', 20);
    }
}
```

To ease testing and reduce duplication a shortcut can be used to specify a class constant as an `@expectedExceptionCode` using the "`@expectedExceptionCode ClassName::CONST`" syntax.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
      * @expectedException     MyException
      * @expectedExceptionCode MyClass::ERRORCODE
      */
    public function testExceptionHasErrorcode20()
    {
```

```
      throw new MyException('Some Message', 20);
    }
}
class MyClass
{
    const ERRORCODE = 20;
}
```

@expectedExceptionMessage

The `@expectedExceptionMessage` annotation works similar to `@expectedExceptionCode` as it lets you make an assertion on the error message of an exception.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @expectedException        MyException
     * @expectedExceptionMessage Some Message
     */
    public function testExceptionHasRightMessage()
    {
        throw new MyException('Some Message', 20);
    }
}
```

The expected message can be a substring of the exception Message. This can be useful to only assert that a certain name or parameter that was passed in shows up in the exception and not fixate the whole exception message in the test.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
     /**
      * @expectedException        MyException
      * @expectedExceptionMessage broken
      */
     public function testExceptionHasRightMessage()
     {
         $param = "broken";
         throw new MyException('Invalid parameter "'.$param.'".', 20);
     }
}
```

To ease testing and reduce duplication a shortcut can be used to specify a class constant as an `@expectedExceptionMessage` using the "`@expectedExceptionMessage ClassName::CONST`" syntax. A sample can be found in the section called “@expectedExceptionCode”.

@expectedExceptionMessageRegExp

The expected message can also be specified as a regular expression using the `@expectedExceptionMessageRegExp` annotation. This is helpful for situations where a substring is not adequate for matching a given message.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
```

```
     /**
      * @expectedException             MyException
      * @expectedExceptionMessageRegExp /Argument \d+ can not be an? \w+/
      */
     public function testExceptionHasRightMessage()
     {
         throw new MyException('Argument 2 can not be an integer');
     }
}
```

@group

A test can be tagged as belonging to one or more groups using the `@group` annotation like this

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @group specification
     */
    public function testSomething()
    {
    }

    /**
     * @group regresssion
     * @group bug2204
     */
    public function testSomethingElse()
    {
    }
}
```

Tests can be selected for execution based on groups using the `--group` and `--exclude-group` options of the command-line test runner or using the respective directives of the XML configuration file.

@large

The `@large` annotation is an alias for `@group large`.

If the `PHP_Invoker` package is installed and strict mode is enabled, a large test will fail if it takes longer than 60 seconds to execute. This timeout is configurable via the `timeoutForLargeTests` attribute in the XML configuration file.

@medium

The `@medium` annotation is an alias for `@group medium`. A medium test must not depend on a test marked as `@large`.

If the `PHP_Invoker` package is installed and strict mode is enabled, a medium test will fail if it takes longer than 10 seconds to execute. This timeout is configurable via the `timeoutForMediumTests` attribute in the XML configuration file.

@preserveGlobalState

When a test is run in a separate process, PHPUnit will attempt to preserve the global state from the parent process by serializing all globals in the parent process and unserializing them in the child

process. This can cause problems if the parent process contains globals that are not serializable. To fix this, you can prevent PHPUnit from preserving global state with the `@preserveGlobalState` annotation.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @runInSeparateProcess
     * @preserveGlobalState disabled
     */
    public function testInSeparateProcess()
    {
        // ...
    }
}
```

@requires

The `@requires` annotation can be used skip tests when common preconditions, like the PHP Version or installed extensions, are not met.

A complete list of possibilities and examples can be found at Table 7.3, “Possible @requires usages”

@runTestsInSeparateProcesses

Indicates that all tests in a test class should be run in a separate PHP process.

```
use PHPUnit\Framework\TestCase;

/**
 * @runTestsInSeparateProcesses
 */
class MyTest extends TestCase
{
    // ...
}
```

Note: By default, PHPUnit will attempt to preserve the global state from the parent process by serializing all globals in the parent process and unserializing them in the child process. This can cause problems if the parent process contains globals that are not serializable. See the section called “@preserveGlobalState” for information on how to fix this.

@runInSeparateProcess

Indicates that a test should be run in a separate PHP process.

```
use PHPUnit\Framework\TestCase;

class MyTest extends TestCase
{
    /**
     * @runInSeparateProcess
     */
    public function testInSeparateProcess()
    {
        // ...
    }
```

```
}
```

Note: By default, PHPUnit will attempt to preserve the global state from the parent process by serializing all globals in the parent process and unserializing them in the child process. This can cause problems if the parent process contains globals that are not serializable. See the section called “@preserveGlobalState” for information on how to fix this.

@small

The `@small` annotation is an alias for `@group small`. A small test must not depend on a test marked as `@medium` or `@large`.

If the `PHP_Invoker` package is installed and strict mode is enabled, a small test will fail if it takes longer than 1 second to execute. This timeout is configurable via the `timeoutForSmallTests` attribute in the XML configuration file.

Note

Tests need to be explicitly annotated by either `@small`, `@medium`, or `@large` to enable run time limits.

@test

As an alternative to prefixing your test method names with `test`, you can use the `@test` annotation in a method's DocBlock to mark it as a test method.

```
/**
 * @test
 */
public function initialBalanceShouldBe0()
{
    $this->assertEquals(0, $this->ba->getBalance());
}
```

@testdox

@ticket

@uses

The `@uses` annotation specifies code which will be executed by a test, but is not intended to be covered by the test. A good example is a value object which is necessary for testing a unit of code.

```
/**
 * @covers BankAccount::deposit
 * @uses   Money
 */
public function testMoneyCanBeDepositedInAccount()
{
```

```
    // ...
}
```

This annotation is especially useful in strict coverage mode where unintentionally covered code will cause a test to fail. See the section called “Unintentionally Covered Code” for more information regarding strict coverage mode.

Appendix C. The XML Configuration File

PHPUnit

The attributes of the `<phpunit>` element can be used to configure PHPUnit's core functionality.

```
<phpunit
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://schema.phpunit.de/4.5/phpunit.xsd"
         backupGlobals="true"
         backupStaticAttributes="false"
         <!--bootstrap="/path/to/bootstrap.php"-->
         cacheTokens="false"
         colors="false"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         forceCoversAnnotation="false"
         mapTestClassNameToCoveredClassName="false"
         printerClass="PHPUnit_TextUI_ResultPrinter"
         <!--printerFile="/path/to/ResultPrinter.php"-->
         processIsolation="false"
         stopOnError="false"
         stopOnFailure="false"
         stopOnIncomplete="false"
         stopOnSkipped="false"
         stopOnRisky="false"
         testSuiteLoaderClass="PHPUnit_Runner_StandardTestSuiteLoader"
         <!--testSuiteLoaderFile="/path/to/StandardTestSuiteLoader.php"-->
         timeoutForSmallTests="1"
         timeoutForMediumTests="10"
         timeoutForLargeTests="60"
         verbose="false">
  <!-- ... -->
</phpunit>
```

The XML configuration above corresponds to the default behaviour of the TextUI test runner documented in the section called “Command-Line Options”.

Additional options that are not available as command-line options are:

`convertErrorsToExceptions`

By default, PHPUnit will install an error handler that converts the following errors to exceptions:

- `E_WARNING`
- `E_NOTICE`
- `E_USER_ERROR`
- `E_USER_WARNING`
- `E_USER_NOTICE`
- `E_STRICT`
- `E_RECOVERABLE_ERROR`

	• E_DEPRECATED • E_USER_DEPRECATED Set convertErrorsToExceptions to false to disable this feature.
convertNoticesToExceptions	When set to false, the error handler installed by convertErrorsToExceptions will not convert E_NOTICE, E_USER_NOTICE, or E_STRICT errors to exceptions.
convertWarningsToExceptions	When set to false, the error handler installed by convertErrorsToExceptions will not convert E_WARNING or E_USER_WARNING errors to exceptions.
forceCoversAnnotation	Code Coverage will only be recorded for tests that use the @covers annotation documented in the section called "@covers".
timeoutForLargeTests	If time limits based on test size are enforced then this attribute sets the timeout for all tests marked as @large. If a test does not complete within its configured timeout, it will fail.
timeoutForMediumTests	If time limits based on test size are enforced then this attribute sets the timeout for all tests marked as @medium. If a test does not complete within its configured timeout, it will fail.
timeoutForSmallTests	If time limits based on test size are enforced then this attribute sets the timeout for all tests not marked as @medium or @large. If a test does not complete within its configured timeout, it will fail.

Test Suites

The <testsuites> element and its one or more <testsuite> children can be used to compose a test suite out of test suites and test cases.

```
<testsuites>
  <testsuite name="My Test Suite">
    <directory>/path/to/*Test.php files</directory>
    <file>/path/to/MyTest.php</file>
    <exclude>/path/to/exclude</exclude>
  </testsuite>
</testsuites>
```

Using the phpVersion and phpVersionOperator attributes, a required PHP version can be specified. The example below will only add the /path/to/*Test.php files and /path/to/MyTest.php file if the PHP version is at least 5.3.0.

```
<testsuites>
    <testsuite name="My Test Suite">
      <directory suffix="Test.php" phpVersion="5.3.0" phpVersionOperator=">=">/path/to/f
      <file phpVersion="5.3.0" phpVersionOperator=">=">/path/to/MyTest.php</file>
    </testsuite>
  </testsuites>
```

The phpVersionOperator attribute is optional and defaults to >=.

Groups

The `<groups>` element and its `<include>`, `<exclude>`, and `<group>` children can be used to select groups of tests marked with the `@group` annotation (documented in the section called "@group") that should (not) be run.

```
<groups>
  <include>
    <group>name</group>
  </include>
  <exclude>
    <group>name</group>
  </exclude>
</groups>
```

The XML configuration above corresponds to invoking the TextUI test runner with the following options:

- `--group name`
- `--exclude-group name`

Whitelisting Files for Code Coverage

The `<filter>` element and its children can be used to configure the whitelist for the code coverage reporting.

```
<filter>
  <whitelist processUncoveredFilesFromWhitelist="true">
    <directory suffix=".php">/path/to/files</directory>
    <file>/path/to/file</file>
    <exclude>
      <directory suffix=".php">/path/to/files</directory>
      <file>/path/to/file</file>
    </exclude>
  </whitelist>
</filter>
```

Logging

The `<logging>` element and its `<log>` children can be used to configure the logging of the test execution.

```
<logging>
  <log type="coverage-html" target="/tmp/report" lowUpperBound="35"
       highLowerBound="70"/>
  <log type="coverage-clover" target="/tmp/coverage.xml"/>
  <log type="coverage-php" target="/tmp/coverage.serialized"/>
  <log type="coverage-text" target="php://stdout" showUncoveredFiles="false"/>
  <log type="json" target="/tmp/logfile.json"/>
  <log type="tap" target="/tmp/logfile.tap"/>
  <log type="junit" target="/tmp/logfile.xml" logIncompleteSkipped="false"/>
  <log type="testdox-html" target="/tmp/testdox.html"/>
  <log type="testdox-text" target="/tmp/testdox.txt"/>
</logging>
```

The XML configuration above corresponds to invoking the TextUI test runner with the following options:

- `--coverage-html /tmp/report`
- `--coverage-clover /tmp/coverage.xml`
- `--coverage-php /tmp/coverage.serialized`
- `--coverage-text`
- `--log-json /tmp/logfile.json`
- `> /tmp/logfile.txt`
- `--log-tap /tmp/logfile.tap`
- `--log-junit /tmp/logfile.xml`
- `--testdox-html /tmp/testdox.html`
- `--testdox-text /tmp/testdox.txt`

The `lowUpperBound`, `highLowerBound`, `logIncompleteSkipped` and `showUncoveredFiles` attributes have no equivalent TextUI test runner option.

- `lowUpperBound`: Maximum coverage percentage to be considered "lowly" covered.
- `highLowerBound`: Minimum coverage percentage to be considered "highly" covered.
- `showUncoveredFiles`: Show all whitelisted files in `--coverage-text` output not just the ones with coverage information.
- `showOnlySummary`: Show only the summary in `--coverage-text` output.

Test Listeners

The `<listeners>` element and its `<listener>` children can be used to attach additional test listeners to the test execution.

```
<listeners>
  <listener class="MyListener" file="/optional/path/to/MyListener.php">
    <arguments>
      <array>
        <element key="0">
          <string>Sebastian</string>
        </element>
      </array>
      <integer>22</integer>
      <string>April</string>
      <double>19.78</double>
      <null/>
      <object class="stdClass"/>
    </arguments>
  </listener>
</listeners>
```

The XML configuration above corresponds to attaching the `$listener` object (see below) to the test execution:

```
$listener = new MyListener(
    ['Sebastian'],
    22,
    'April',
    19.78,
```

```
    null,
    new stdClass
);
```

Setting PHP INI settings, Constants and Global Variables

The `<php>` element and its children can be used to configure PHP settings, constants, and global variables. It can also be used to prepend the `include_path`.

```
<php>
  <includePath>.</includePath>
  <ini name="foo" value="bar"/>
  <const name="foo" value="bar"/>
  <var name="foo" value="bar"/>
  <env name="foo" value="bar"/>
  <post name="foo" value="bar"/>
  <get name="foo" value="bar"/>
  <cookie name="foo" value="bar"/>
  <server name="foo" value="bar"/>
  <files name="foo" value="bar"/>
  <request name="foo" value="bar"/>
</php>
```

The XML configuration above corresponds to the following PHP code:

```
ini_set('foo', 'bar');
define('foo', 'bar');
$GLOBALS['foo'] = 'bar';
$_ENV['foo'] = 'bar';
$_POST['foo'] = 'bar';
$_GET['foo'] = 'bar';
$_COOKIE['foo'] = 'bar';
$_SERVER['foo'] = 'bar';
$_FILES['foo'] = 'bar';
$_REQUEST['foo'] = 'bar';
```

Appendix D. Index

Index

Symbols

A

C

D

E

F

G

H

I

J

L

M

O

P

R

S

T

W

X

Appendix E. Bibliography

[Astels2003] *Test Driven Development*. David Astels. Copyright © 2003. Prentice Hall. ISBN 0131016490.

[Beck2002] *Test Driven Development by Example*. Kent Beck. Copyright © 2002. Addison-Wesley. ISBN 0-321-14653-0.

[Meszaros2007] *xUnit Test Patterns: Refactoring Test Code*. Gerard Meszaros. Copyright © 2007. Addison-Wesley. ISBN 978-0131495050.

Appendix F. Copyright

THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF
SUCH TERMS AND CONDITIONS.

1. Definitions

   a. "Adaptation" means a work based upon the Work, or upon the
      Work and other pre-existing works, such as a translation,
      adaptation, derivative work, arrangement of music or other
      alterations of a literary or artistic work, or phonogram or
      performance and includes cinematographic adaptations or any
      other form in which the Work may be recast, transformed, or
      adapted including in any form recognizably derived from the
      original, except that a work that constitutes a Collection
      will not be considered an Adaptation for the purpose of this
      License. For the avoidance of doubt, where the Work is a
      musical work, performance or phonogram, the synchronization of
      the Work in timed-relation with a moving image ("synching")
      will be considered an Adaptation for the purpose of this
      License.

   b. "Collection" means a collection of literary or artistic works,
      such as encyclopedias and anthologies, or performances,
      phonograms or broadcasts, or other works or subject matter
      other than works listed in Section 1(f) below, which, by
      reason of the selection and arrangement of their contents,
      constitute intellectual creations, in which the Work is
      included in its entirety in unmodified form along with one or
      more other contributions, each constituting separate and
      independent works in themselves, which together are assembled
      into a collective whole. A work that constitutes a Collection
      will not be considered an Adaptation (as defined above) for
      the purposes of this License.

   c. "Distribute" means to make available to the public the
      original and copies of the Work or Adaptation, as appropriate,
      through sale or other transfer of ownership.

   d. "Licensor" means the individual, individuals, entity or
      entities that offer(s) the Work under the terms of this License.

   e. "Original Author" means, in the case of a literary or artistic
      work, the individual, individuals, entity or entities who
      created the Work or if no individual or entity can be
      identified, the publisher; and in addition (i) in the case of
      a performance the actors, singers, musicians, dancers, and
      other persons who act, sing, deliver, declaim, play in,
      interpret or otherwise perform literary or artistic works or
      expressions of folklore; (ii) in the case of a phonogram the
      producer being the person or legal entity who first fixes the
      sounds of a performance or other sounds; and, (iii) in the
      case of broadcasts, the organization that transmits the
      broadcast.

   f. "Work" means the literary and/or artistic work offered under
      the terms of this License including without limitation any
      production in the literary, scientific and artistic domain,
      whatever may be the mode or form of its expression including
      digital form, such as a book, pamphlet and other writing; a
      lecture, address, sermon or other work of the same nature; a
      dramatic or dramatico-musical work; a choreographic work or
      entertainment in dumb show; a musical composition with or
      without words; a cinematographic work to which are assimilated
      works expressed by a process analogous to cinematography; a
      work of drawing, painting, architecture, sculpture, engraving

or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

g. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

h. "Publicly Perform" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

i. "Reproduce" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

b. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

c. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

d. to Distribute and Publicly Perform Adaptations.

e. For the avoidance of doubt:

```
      i. Non-waivable Compulsory License Schemes. In those
         jurisdictions in which the right to collect royalties
         through any statutory or compulsory licensing scheme cannot
         be waived, the Licensor reserves the exclusive right to
         collect such royalties for any exercise by You of the
         rights granted under this License;

      ii. Waivable Compulsory License Schemes. In those
          jurisdictions in which the right to collect royalties
          through any statutory or compulsory licensing scheme can
          be waived, the Licensor waives the exclusive right to
          collect such royalties for any exercise by You of the
          rights granted under this License; and,

      iii. Voluntary License Schemes. The Licensor waives the right
           to collect royalties, whether individually or, in the
           event that the Licensor is a member of a collecting
           society that administers voluntary licensing schemes, via
           that society, from any exercise by You of the rights
           granted under this License.

The above rights may be exercised in all media and formats whether
now known or hereafter devised. The above rights include the right
to make such modifications as are technically necessary to exercise
the rights in other media and formats. Subject to Section 8(f), all
rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly
   made subject to and limited by the following restrictions:

   a. You may Distribute or Publicly Perform the Work only under the
      terms of this License. You must include a copy of, or the
      Uniform Resource Identifier (URI) for, this License with every
      copy of the Work You Distribute or Publicly Perform. You may
      not offer or impose any terms on the Work that restrict the
      terms of this License or the ability of the recipient of the
      Work to exercise the rights granted to that recipient under
      the terms of the License. You may not sublicense the Work. You
      must keep intact all notices that refer to this License and to
      the disclaimer of warranties with every copy of the Work You
      Distribute or Publicly Perform. When You Distribute or
      Publicly Perform the Work, You may not impose any effective
      technological measures on the Work that restrict the ability
      of a recipient of the Work from You to exercise the rights
      granted to that recipient under the terms of the License. This
      Section 4(a) applies to the Work as incorporated in a
      Collection, but this does not require the Collection apart
      from the Work itself to be made subject to the terms of this
      License. If You create a Collection, upon notice from any
      Licensor You must, to the extent practicable, remove from the
      Collection any credit as required by Section 4(b), as
      requested. If You create an Adaptation, upon notice from any
      Licensor You must, to the extent practicable, remove from the
      Adaptation any credit as required by Section 4(b), as requested.

   b. If You Distribute, or Publicly Perform the Work or any
      Adaptations or Collections, You must, unless a request has
      been made pursuant to Section 4(a), keep intact all copyright
      notices for the Work and provide, reasonable to the medium or
      means You are utilizing: (i) the name of the Original Author
      (or pseudonym, if applicable) if supplied, and/or if the
      Original Author and/or Licensor designate another party or
      parties (e.g., a sponsor institute, publishing entity,
```

journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv), consistent with Section 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4 (b) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

c. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

   a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

   b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

   a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

   b. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

   c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

   d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

   e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

   f. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be

```
      enforced according to the corresponding provisions of the
      implementation of those treaty provisions in the applicable
      national law. If the standard suite of rights granted under
      applicable copyright law includes additional rights not
      granted under this License, such additional rights are deemed
      to be included in the License; this License is not intended to
      restrict the license of any rights under applicable law.

Creative Commons is not a party to this License, and makes no
warranty whatsoever in connection with the Work. Creative Commons
will not be liable to You or any party on any legal theory for any
damages whatsoever, including without limitation any general,
special, incidental or consequential damages arising in connection
to this license. Notwithstanding the foregoing two (2) sentences,
if Creative Commons has expressly identified itself as the Licensor
hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the
Work is licensed under the CCPL, Creative Commons does not authorize
the use by either party of the trademark "Creative Commons" or any
related trademark or logo of Creative Commons without the prior
written consent of Creative Commons. Any permitted use will be in
compliance with Creative Commons' then-current trademark usage
guidelines, as may be published on its website or otherwise made
available upon request from time to time. For the avoidance of
doubt, this trademark restriction does not form part of this
License.

Creative Commons may be contacted at http://creativecommons.org/.

======================================================================
```