PHPUnit Manual

Release 7.0

Sebastian Bergmann

Sep 16, 2020

Contents

Edition for PHPUnit 7.0. Updated on Sep 16, 2020.

Sebastian Bergmann

Contents:

CHAPTER 1

Installing PHPUnit

1.1 Requirements

PHPUnit 7.0 requires PHP 7.1; using the latest version of PHP is highly recommended.

PHPUnit requires the dom and json extensions, which are normally enabled by default.

PHPUnit also requires the pcre, reflection, and spl 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 (2.5.0 or later) and tokenizer extensions. Generating XML reports requires the xmlwriter extension.

1.2 PHP Archive (PHAR)

The easiest way to obtain PHPUnit is to download a PHP Archive (PHAR) that has all required (as well as some optional) dependencies of PHPUnit bundled in a single file.

The phar extension is required for using PHP Archives (PHAR).

If the Suhosin extension is enabled, you need to allow execution of PHARs in your `php.ini`:

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

The PHPUnit PHAR can be used immediately after download:

```
$ wget https://phar.phpunit.de/phpunit-7.0.phar
$ php phpunit-7.0.phar --version
PHPUnit x.y.z by Sebastian Bergmann and contributors.
```

It is a common practice to make the PHAR executable:

```
$ wget https://phar.phpunit.de/phpunit-7.0.phar
$ chmod +x phpunit-7.0.phar
$ ./phpunit-7.0.phar --version
```

```
PHPUnit x.y.z by Sebastian Bergmann and contributors.
```

1.2.1 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 SHA256 hashes are available for verification on 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-7.0.phar
$ wget https://phar.phpunit.de/phpunit-7.0.phar.asc
```

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

```
$ gpg phpunit-7.0.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.

```
$ curl --silent https://sebastian-bergmann.de/gpg.asc | gpg --import
gpg: key 4AA394086372C20A: 452 signatures not checked due to missing keys
gpg: /root/.gnupg/trustdb.gpg: trustdb created
gpg: key 4AA394086372C20A: public key "Sebastian Bergmann <sb@sebastian-bergmann.de>"␣
↪imported
gpg: Total number processed: 1
gpg:               imported: 1
gpg: no ultimately trusted keys found
```

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-7.0.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.

Manually verifying the authenticity and integrity of a PHPUnit PHAR using GPG is tedious. This is why PHIVE, the PHAR Installation and Verification Environment, was created. You can learn about PHIVE on its website

1.3 Composer

Add a (development-time) dependency on `phpunit/phpunit` to your project's `composer.json` file if you use Composer to manage the dependencies of your project:

```
composer require --dev phpunit/phpunit ^7.0
```

1.4 Global Installation

Please note that it is not recommended to install PHPUnit globally, as `/usr/bin/phpunit` or `/usr/local/bin/phpunit`, for instance.

Instead, PHPUnit should be managed as a project-local dependency.

Either put the PHAR of the specific PHPUnit version you need in your project's `tools` directory (which should be managed by PHIVE) or depend on the specific PHPUnit version you need in your project's `composer.json` if you use Composer.

1.5 Webserver

PHPUnit is a framework for writing as well as a commandline tool for running tests. Writing and running tests is a development-time activity. There is no reason why PHPUnit should be installed on a webserver.

If you upload PHPUnit to a webserver then your deployment process is broken. On a more general note, if your `vendor` directory is publicly accessible on your webserver then your deployment process is also broken.

Please note that if you upload PHPUnit to a webserver "bad things" may happen. You have been warned.

CHAPTER 2

Writing Tests for PHPUnit

Example 2.1 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 `assertSame()` (see *Assertions*) are used to assert that an actual value matches an expected value.

Example 2.1: Testing array operations with PHPUnit

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

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

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

Martin Fowler:

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

2.1 Test Dependencies

> *Adrian Kuhn et. al.*:
>
> 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 necessarily an encapsulated, independent entity. Often there are implicit dependencies between test methods, hidden in the implementation scenario of a test.

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 shows how to use the `@depends` annotation to express dependencies between test methods.

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

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

        return $stack;
    }

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

        return $stack;
    }

    /**
     * @depends testPush
     */
```

(continues on next page)

(continued from previous page)

```
    public function testPop(array $stack): void
    {
        $this->assertSame('foo', array_pop($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. Instead of a reference either (a) a (deep) copy via `@depends clone`, or (b) a (normal shallow) clone (based on PHP keyword `clone`) via `@depends shallowClone` are possible too.

To 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.

Example 2.3: Exploiting the dependencies between tests

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

    /**
     * @depends testOne
     */
    public function testTwo(): void
    {
    }
}
```

```
$ phpunit --verbose DependencyFailureTest
PHPUnit 7.0.0 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

Example 2.4: Test with multiple dependencies

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

        return 'first';
    }

    public function testProducerSecond(): string
    {
        $this->assertTrue(true);

        return 'second';
    }

    /**
     * @depends testProducerFirst
     * @depends testProducerSecond
     */
    public function testConsumer(string $a, string $b): void
    {
        $this->assertSame('first', $a);
        $this->assertSame('second', $b);
    }
}
```

```
$ phpunit --verbose MultipleDependenciesTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

...

Time: 0 seconds, Memory: 3.25Mb

OK (3 tests, 4 assertions)
```

2.2 Data Providers

A test method can accept arbitrary arguments. These arguments are to be provided by one or more data provider methods (`additionProvider()` in Example 2.5). 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 declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

    public function additionProvider(): array
    {
        return [
            [0, 0, 0],
            [0, 1, 1],
            [1, 0, 1],
            [1, 1, 3]
        ];
    }
}
```

```
$ phpunit DataTest
PHPUnit 7.0.0 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 is identical to 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 declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

    public function additionProvider(): array
    {
        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 7.0.0 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 is identical to 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 declare(strict_types=1);
use PHPUnit\Framework\TestCase;

require 'CsvFileIterator.php';

final class DataTest extends TestCase
{
    /**
     * @dataProvider additionProvider
     */
    public function testAdd($a, $b, $expected): void
```

(continues on next page)

(continued from previous page)

```
    {
        $this->assertSame($expected, $a + $b);
    }

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

```
$ phpunit DataTest
PHPUnit 7.0.0 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 is identical to 3.

/home/sb/DataTest.php:11

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

Example 2.8: The CsvFileIterator class

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

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

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

    public function rewind(): void
    {
        rewind($this->file);

        $this->current = fgetcsv($this->file);
        $this->key     = 0;
    }
```

(continues on next page)

(continued from previous page)

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

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

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

    public function next(): void
    {
        $this->current = fgetcsv($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

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

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

    public function testProducerFirst(): void
    {
        $this->assertTrue(true);

        return 'first';
    }

    public function testProducerSecond(): void
    {
        $this->assertTrue(true);

        return 'second';
    }

    /**
     * @depends testProducerFirst
```

(continues on next page)

(continued from previous page)

```
     * @depends testProducerSecond
     * @dataProvider provider
     */
    public function testConsumer(): void
    {
        $this->assertSame(
            ['provider1', 'first', 'second'],
            func_get_args()
        );
    }
}
```

```
$ phpunit --verbose DependencyAndDataProviderComboTest
PHPUnit 7.0.0 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 identical.
--- Expected
+++ Actual
@@ @@
Array &0 (
-    0 => 'provider1'
+    0 => 'provider2'
     1 => 'first'
     2 => 'second'
)
/home/sb/DependencyAndDataProviderComboTest.php:32

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

Example 2.10: Using multiple data providers for a single test :name: writing-tests-for-phpunit.data-providers.examples.DataTest.php

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class DataTest extends TestCase
{
    /**
     * @dataProvider additionWithNonNegativeNumbersProvider
     * @dataProvider additionWithNegativeNumbersProvider
     */
    public function testAdd($a, $b, $expected): void
    {
        $this->assertSame($expected, $a + $b);
    }
```

(continues on next page)

(continued from previous page)

```
    public function additionWithNonNegativeNumbersProvider(): array
    {
        return [
            [0, 1, 1],
            [1, 0, 1],
            [1, 1, 3]
        ];
    }

    public function additionWithNegativeNumbersProvider(): array
    {
        return [
            [-1, 1, 0],
            [-1, -1, -2],
            [1, -1, 0]
        ];
    }
}
```

```
$ phpunit DataTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

..F...                                                             6 / 6 (100
↪%)

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

1) DataTest::testAdd with data set #3 (1, 1, 3)
Failed asserting that 2 is identical to 3.

/home/sb/DataTest.php:12

FAILURES!
Tests: 6, Assertions: 6, 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.

2.3 Testing Exceptions

Example 2.11 shows how to use the `expectException()` method to test whether an exception is thrown by the code under test.

Example 2.11: Using the expectException() method

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

```
$ phpunit ExceptionTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ExceptionTest::testException
Failed asserting that exception of type "InvalidArgumentException" is thrown.

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.

Note

Note that `expectExceptionMessage()` asserts that the `$actual` message contains the `$expected` message and does not perform an exact string comparison.

2.4 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.

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 expectException()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;
use PHPUnit\Framework\Error\Error;

final class ExpectedErrorTest extends TestCase
{
    public function testFailingInclude(): void
    {
        $this->expectException(Error::class);

        include 'not_existing_file.php';
    }
}
```

```
$ phpunit -d error_reporting=2 ExpectedErrorTest
PHPUnit 7.0.0 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.

When testing code that uses PHP built-in functions such as `fopen()` that may trigger errors 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 an exception raised by PHPUnit's error handler.

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

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class ErrorSuppressionTest extends TestCase
{
    public function testFileWriting(): void
    {
        $writer = new FileWriter;

        $this->assertFalse(@$writer->write('/is-not-writeable/file', 'stuff'));
    }
}

final class FileWriter
{
    public function write($file, $content)
    {
        $file = fopen($file, 'w');

        if ($file === false) {
            return false;
```

(continues on next page)

(continued from previous page)

```
        }

        // ...
    }
}
```

```
$ phpunit ErrorSuppressionTest
PHPUnit 7.0.0 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`.

2.5 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 feature to provide the functionality that is necessary for this.

Example 2.14 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 declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class OutputTest extends TestCase
{
    public function testExpectFooActualFoo(): void
    {
        $this->expectOutputString('foo');

        print 'foo';
    }

    public function testExpectBarActualBaz(): void
    {
        $this->expectOutputString('bar');

        print 'baz';
    }
}
```

```
$ phpunit OutputTest
PHPUnit 7.0.0 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 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.
`string getActualOutput()`	Get the actual output.

Note

A test that emits output will fail in strict mode.

2.6 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
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

```
$ phpunit ArrayDiffTest
PHPUnit 7.0.0 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 identical.
--- 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
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

```
$ phpunit LongArrayDiffTest
PHPUnit 7.0.0 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 identical.
--- Expected
+++ Actual
@@ @@
     11 => 0
     12 => 1
     13 => 2
-    14 => 3
+    14 => 33
     15 => 4
     16 => 5
     17 => 6
 )

/home/sb/LongArrayDiffTest.php:7

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

2.6.1 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 declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

```
$ phpunit ArrayWeakComparisonTest
PHPUnit 7.0.0 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 7.0.0 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 *Risky Tests*).

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

`I`

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

PHPUnit distinguishes between *failures* and *errors*. A failure is a violated PHPUnit assertion such as a failing `assertSame()` 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.

3.1 Command-Line Options

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

```
$ phpunit --help
PHPUnit 7.0.0 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.
  --whitelist <dir>           Whitelist <dir> for code coverage analysis.
  --disable-coverage-ignore   Disable annotations for ignoring code coverage.

Logging Options:

  --log-junit <file>          Log test execution in JUnit XML format to file.
  --log-teamcity <file>       Log test execution in TeamCity format to file.
  --testdox-html <file>       Write agile documentation in HTML format to␣
↪file.
  --testdox-text <file>       Write agile documentation in Text format to␣
↪file.
  --testdox-xml <file>        Write agile documentation in XML format to file.
  --reverse-list              Print defects in reverse order

Test Selection Options:

  --filter <pattern>          Filter which tests to run.
  --testsuite <name,...>      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.
  --list-suites               List available test suites.
  --test-suffix ...           Only search for test in files with specified
```

```
                              suffix(es). Default: Test.php,.phpt

Test Execution Options:

  --dont-report-useless-tests Do not report tests that do not test anything.
  --strict-coverage           Be strict about @covers annotation usage.
  --strict-global-state       Be strict about changes to global state
  --disallow-test-output      Be strict about output during tests.
  --disallow-resource-usage   Be strict about resource usage during small␣
↪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.
  --globals-backup            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-warning           Stop execution upon first warning.
  --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.
  --fail-on-warning           Treat tests with warnings as failures.
  --fail-on-risky             Treat risky tests as failures.
  -v|--verbose                Output more verbose information.
  --debug                     Display debugging information.

  --loader <loader>           TestSuiteLoader implementation to use.
  --repeat <times>            Runs the test(s) repeatedly.
  --teamcity                  Report test execution progress in TeamCity␣
↪format.
  --testdox                   Report test execution progress in TestDox␣
↪format.
  --testdox-group             Only include tests from the specified group(s).
  --testdox-exclude-group     Exclude tests from the specified group(s).
  --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).
  --no-coverage               Ignore code coverage configuration.
  --no-extensions             Do not load PHPUnit extensions.
  --include-path <path(s)>    Prepend PHP's include_path with given path(s).
```

```
  -d key[=value]                 Sets a php.ini value.
  --generate-configuration       Generate configuration file with suggested␣
↪settings.

Miscellaneous Options:

  -h|--help                      Prints this usage information.
  --version                      Prints the version and exits.
  --atleast-version <min>        Checks that version is greater than min 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 *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 *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 *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 *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 *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 *TestDox*).

`--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 for an example of named data sets.

Example 3.1: Named data sets

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

namespace TestNamespace;

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

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

`/path/to/my/test.phpt`

The test name for a PHPT test is the filesystem path.

See Example 3.2 for examples of valid filter patterns.

Example 3.2: Filter pattern examples

```
--filter 'TestNamespace\\TestCaseClass::testMethod'
--filter 'TestNamespace\\TestCaseClass'
--filter TestNamespace
--filter TestCaseClase
--filter testMethod
```

(continues on next page)

(continued from previous page)

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

See Example 3.3 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).

`--dont-report-useless-tests`

Do not report tests that do not test anything. See *Risky Tests* for details.

`--strict-coverage`

Be strict about unintentionally covered code. See *Risky Tests* for details.

`--strict-global-state`

Be strict about global state manipulation. See *Risky Tests* for details.

`--disallow-test-output`

Be strict about output during tests. See *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 *Risky Tests* for details.

`--process-isolation`

Run each test in a separate PHP process.

`--no-globals-backup`

Do not backup and restore $GLOBALS. See *Global State* for more details.

`--static-backup`

Backup and restore static attributes of user-defined classes. See *Global State* for more details.

`--colors`

Use colors in output. On Windows, use ANSICON or 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.

`--testdox`

Reports the test progress in TestDox format (see *TestDox*).

`--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 *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.

If a directory is specified and if `phpunit.xml` or `phpunit.xml.dist` (in that order) exists in this directory, 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).

3.2 TestDox

PHPUnit's TestDox functionality looks at a test class and all the test method names and converts them from camel case (or snake_case) PHP names to sentences: `testBalanceIsInitiallyZero()` (or `test_balance_is_initially_zero()` 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 7.0.0 by Sebastian Bergmann and contributors.

BankAccount
 ✓ Balance is initially zero
 ✓ 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.

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 *Testing array operations with PHPUnit*, the fixture was 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 *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 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 `assertSame()` assertion method.

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

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class StackTest extends TestCase
{
    private $stack;

    protected function setUp(): void
    {
        $this->stack = [];
```

(continues on next page)

(continued from previous page)

```
    }

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

    public function testPush(): void
    {
        array_push($this->stack, 'foo');

        $this->assertSame('foo', $this->stack[count($this->stack)-1]);
        $this->assertFalse(empty($this->stack));
    }

    public function testPop(): void
    {
        array_push($this->stack, 'foo');

        $this->assertSame('foo', array_pop($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 declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

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

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

    public function testOne(): void
    {
        fwrite(STDOUT, __METHOD__ . "\n");
```

(continues on next page)

(continued from previous page)

```
        $this->assertTrue(true);
    }

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

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

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

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

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

```
$ phpunit TemplateMethodsTest
PHPUnit 7.0.0 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.
```

4.1 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.

4.2 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.

4.3 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 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
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class DatabaseTest extends TestCase
{
    private static $dbh;

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

    public static function tearDownAfterClass(): void
    {
        self::$dbh = null;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

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 *Test Doubles*), than by creating dependencies between tests at runtime and ignoring the opportunity to improve your design.

4.4 Global State

It is hard to test code that uses singletons. 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.

Prior to version 6, by default, PHPUnit ran 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.

As of version 6, PHPUnit does not perform this backup and restore operation for global and super-global variables by default anymore. It can be activated by using the `--globals-backup` option or setting `backupGlobals="true"` in the XML configuration file.

By using the `--static-backup` option or setting `backupStaticAttributes="true"` in the XML configuration file, 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 *@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

```
final 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 *@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:

```
final 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.

5.1 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 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 need to point the PHPUnit command-line test runner to the test directory:

```
$ phpunit --bootstrap src/autoload.php tests
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

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

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.php
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

........

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 testObjectCanBeConstructedForValidConstruct
↪tests
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

..

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 *Test Dependencies*. In the next section you will see how you can make the test execution order explicit by using the XML configuration file.

5.2 Composing a Test Suite Using XML Configuration

PHPUnit's XML configuration file (*The XML Configuration File*) can also be used to compose a test suite. Example 5.1 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>
```

To run the test suite, use the the `--testsuite` option:

```
$ phpunit --bootstrap src/autoload.php --testsuite money
```

```
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

..

Time: 167 ms, Memory: 3.00Mb

OK (2 test, 2 assertions)
```

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.

6.1 Useless Tests

PHPUnit is by default strict about tests that do not test anything. This check can be disabled by using the `--dont-report-useless-tests` option on the *command line* or by setting `beStrictAboutTestsThatDoNotTestAnything="false"` in PHPUnit's *configuration file*.

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

6.2 Unintentionally Covered Code

PHPUnit can be strict about unintentionally covered code. This check can be enabled by using the `--strict-coverage` option on the *command line* or by setting `beStrictAboutCoversAnnotation="true"` in PHPUnit's *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.

Furthermore, by setting `forceCoversAnnotation="true"` in PHPUnit's *configuration file*, a test can be marked as risky when it does not have a *@covers* annotation.

6.3 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 *command line* or by setting `beStrictAboutOutputDuringTests="true"` in PHPUnit's *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.

6.4 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 *command line* or by setting `enforceTimeLimit="true"` in PHPUnit's *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 *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 configuration *configuration file*.

A test annotated with `@small` will fail if it takes longer than 1 second to execute. This timeout is configurable via the `timeoutForSmallTests` attribute in the *configuration file*.

Note

Tests need to be explicitly annotated by either `@small`, `@medium`, or `@large` to enable run time limits.

6.5 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 *command line* or by setting `beStrictAboutChangesToGlobalState="true"` in PHPUnit's *configuration file*.

CHAPTER 7

Incomplete and Skipped Tests

7.1 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(): void
{
}
```

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 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 declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class SampleTest extends TestCase
{
    public function testSomething(): void
```

(continues on next page)

(continued from previous page)

```
    {
        // 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 7.0.0 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 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.

7.2 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 only be run if a MySQL server is available.

Example 7.2 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 declare(strict_types=1);
use PHPUnit\Framework\TestCase;
```

(continues on next page)

(continued from previous page)

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

    public function testConnection(): void
    {
        // ...
    }
}
```

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 7.0.0 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 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.

7.3 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 along with an optional operator	@requires PHP 7.1.20	@requires PHP >= 7.2
`PHPUnit`	Any PHPUnit version identifier along with an optional operator	@requires PHPUnit 7.3.1	@requires PHPUnit < 8
`OS`	A regexp matching PHP_OS	@requires OS Linux	@requires OS WIN32|WINNT
`OSFAMILY`	Any OS family	@requires OSFAMILY Solaris	@requires OSFAMILY Windows
`function`	Any valid parameter to function_exists	@requires function imap_open	@requires function ReflectionMethod::setAccessible
`extension`	Any extension name along with an optional version identifier and optional operator	@requires extension mysqli	@requires extension redis >= 2.2.0

The following operators are supported for PHP, PHPUnit, and extension version constraints: <, <=, >, >=, =, ==, !=, <>.

Example 7.3: Skipping test cases using @requires

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

/**
 * @requires extension mysqli
 */
final class DatabaseTest extends TestCase
{
    /**
     * @requires PHP >= 5.3
     */
    public function testConnection(): void
    {
        // 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 *Test Suites*

CHAPTER 8

Test Doubles

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

> *Gerard Meszaros*:
>
> 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!

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 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 except for `static` methods that will be replaced by a method throwing a `\PHPUnit\Framework\MockObject\BadMethodCallException` exception.

8.1 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 8.2 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 8.1). We then use the Fluent Interface 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 8.1: The class we want to stub

```
<?php declare(strict_types=1);
class SomeClass
{
    public function doSomething()
    {
        // Do something.
    }
}
```

Example 8.2: Stubbing a method call to return a fixed value

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class StubTest extends TestCase
{
    public function testStub(): void
    {
        // 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->assertSame('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('doSomethin` 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.

Please note that `createMock()` will automatically and recursively stub return values based on a method's return

type. Consider the example shown below:

Example 8.3: A method with a return type declaration

```
<?php declare(strict_types=1);
class C
{
    public function m(): D
    {
        // Do something.
    }
}
```

In the example shown above, the `C::m()` method has a return type declaration indicating that this method returns an object of type `D`. When a test double for `C` is created and no return value is configured for `m()` using `willReturn()` (see above), for instance, then when `m()` is invoked PHPUnit will automatically create a test double for `D` to be returned.

Similarily, if `m` had a return type declaration for a scalar type then a return value such as `0` (for `int`), `0.0` (for `float`), or `[]` (for `array`) would be generated.

Example 8.4 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 8.4: Using the Mock Builder API can be used to configure the generated test double class

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class StubTest extends TestCase
{
    public function testStub(): void
    {
        // Create a stub for the SomeClass class.
        $stub = $this->getMockBuilder(SomeClass::class)
                     ->disableOriginalConstructor()
                     ->disableOriginalClone()
                     ->disableArgumentCloning()
                     ->disallowMockingUnknownTypes()
                     ->getMock();

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

        // Calling $stub->doSomething() will now return
        // 'foo'.
        $this->assertSame('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 8.5 shows how you can achieve this using `returnArgument()` instead of `returnValue()`.

Example 8.5: Stubbing a method call to return one of the arguments

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class StubTest extends TestCase
{
    public function testReturnArgumentStub(): void
    {
        // 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->assertSame('foo', $stub->doSomething('foo'));

        // $stub->doSomething('bar') returns 'bar'
        $this->assertSame('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 8.6 shows how you can use `returnSelf()` to achieve this.

Example 8.6: Stubbing a method call to return a reference to the stub object

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class StubTest extends TestCase
{
    public function testReturnSelf(): void
    {
        // 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 8.7 for an example.

Example 8.7: Stubbing a method call to return the value from a map

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;
```

(continues on next page)

(continued from previous page)

```
final class StubTest extends TestCase
{
    public function testReturnValueMapStub(): void
    {
        // 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->assertSame('d', $stub->doSomething('a', 'b', 'c'));
        $this->assertSame('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 8.8 for an example.

Example 8.8: Stubbing a method call to return a value from a callback

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class StubTest extends TestCase
{
    public function testReturnCallbackStub(): void
    {
        // 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->assertSame('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 8.9 for an example.

Example 8.9: Stubbing a method call to return a list of values in the specified order

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class StubTest extends TestCase
{
    public function testOnConsecutiveCallsStub(): void
    {
        // 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->assertSame(2, $stub->doSomething());
        $this->assertSame(3, $stub->doSomething());
        $this->assertSame(5, $stub->doSomething());
    }
}
```

Instead of returning a value, a stubbed method can also raise an exception. Example 8.10 shows how to use `throwException()` to do this.

Example 8.10: Stubbing a method call to throw an exception

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class StubTest extends TestCase
{
    public function testThrowExceptionStub(): void
    {
        // 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 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.

8.2 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 8.11 shows the code for the `Subject` and `Observer` classes that are part of the System under Test (SUT).

Example 8.11: The Subject and Observer classes that are part of the System under Test (SUT)

```
<?php declare(strict_types=1);
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');
    }
```

(continues on next page)

(continued from previous page)

```
    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 8.12 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`. We then use `setMethods(['update'])` to configure that 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 8.12: Testing that a method gets called once and with a specified argument

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

(continues on next page)

(continued from previous page)

```
        // 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 8.13: Testing that a method gets called with a number of arguments constrained in different ways

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class SubjectTest extends TestCase
{
    public function testErrorReported(): void
    {
        // 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 8.14: Testing that a method gets called two times with specific arguments.

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class FooTest extends TestCase
{
    public function testFunctionCalledTwoTimesWithSpecificArguments(): void
    {
        $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 8.15: More complex argument verification

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class SubjectTest extends TestCase
{
    public function testErrorReported(): void
    {
        // 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');
```

(continues on next page)

(continued from previous page)

```
        $subject->attach($observer);

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

Example 8.16: Testing that a method gets called once and with the identical object as was passed

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class FooTest extends TestCase
{
    public function testIdenticalObjectPassed(): void
    {
        $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 8.17: Create a mock object with cloning parameters enabled

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

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

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

Constraints shows the constraints that can be applied to method arguments and Table 8.1 shows the matchers that are available to specify the number of invocations.

Table 8.1: 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.
- `setMethodsExcept(array $methods)` can be called on the Mock Builder object to specify the methods that will not be replaced with a configurable test double while replacing all other public methods. This works inverse to `setMethods()`.
- `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.

8.3 Prophecy

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 8.18 shows how the same test shown in Example 8.12 can be expressed using Prophecy's philosophy of prophecies and revelations:

Example 8.18: Testing that a method gets called once and with a specified argument

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class SubjectTest extends TestCase
{
    public function testObserversAreUpdated(): void
    {
        $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 for Prophecy for further details on how to create, configure, and use stubs, spies, and mocks using this alternative test double framework.

8.4 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 8.19: Testing the concrete methods of a trait

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

    public abstract function abstractMethod();
}

final class TraitClassTest extends TestCase
```

(continues on next page)

(continued from previous page)

```
{
    public function testConcreteMethod(): void
    {
        $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 8.20: Testing the concrete methods of an abstract class

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

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

    public abstract function abstractMethod();
}

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

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

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

8.5 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 create stubs and mocks of web services, the `getMockFromWsdl()` can be used 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 8.21 shows how `getMockFromWsdl()` can be used to stub, for example, the web service described in `GoogleSearch.wsdl`.

Example 8.21: Stubbing a web service

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class GoogleTest extends TestCase
{
    public function testSearch(): void
    {
        $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,
```

(continues on next page)

(continued from previous page)

```
                '',
                false,
                '',
                '',
                ''
            )
        );
    }
}
```

CHAPTER 9

Code Coverage Analysis

> *Wikipedia*:
>
> 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.

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-code-coverage component, which in turn leverages the code coverage functionality provided by the Xdebug extension for PHP or by PHPDBG.

Note

If you see a warning while running tests that no code coverage driver is available, it means that you are using the PHP CLI binary (`php`) and do not have Xdebug loaded. The Xdebug installation guide explains how Xdebug can be installed and configured. Alternatively, you may use the PHPDBG binary (`phpdbg`) instead of the PHP CLI one.

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 *The Command-Line Test Runner* for a list of command line switches that control code coverage functionality as well as *Logging* for the relevant configuration settings.

9.1 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-code-coverage 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-code-coverage 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-code-coverage.

9.2 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` *command line* option or via the configuration file (see *Whitelisting Files for Code Coverage*).

The `addUncoveredFilesFromWhitelist` and `processUncoveredFilesFromWhitelist` configuration settings are available to configure how the whitelist is used:

- `addUncoveredFilesFromWhitelist="false"` means that only whitelisted files that have at least one line of executed code are included in the code coverage report
- `addUncoveredFilesFromWhitelist="true"` (default) means that all whitelisted files are included in the code coverage report even if not a single line of code of such a file is executed
- `processUncoveredFilesFromWhitelist="false"` (default) means that a whitelisted file that has no executed lines of code will be added to the code coverage report (if `addUncoveredFilesFromWhitelist="true"` is set) but it will not be loaded by PHPUnit and it will therefore not be analysed for correct executable lines of code information

- `processUncoveredFilesFromWhitelist="true"` means that a whitelisted file that has no executed lines of code will be loaded by PHPUnit so that it can be analysed for correct executable lines of code information

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.

9.3 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 9.1.

Example 9.1: Using the `@codeCoverageIgnore`, `@codeCoverageIgnoreStart` and `@codeCoverageIgnoreEnd` annotations

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

/**
 * @codeCoverageIgnore
 */
final class Foo
{
    public function bar(): void
    {
    }
}

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

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.

9.4 Specifying Covered Code Parts

The `@covers` annotation (see the *annotation documentation*) can be used in the test code to specify which code parts a test class (or test method) wants to test. If provided, this effectively filters the code coverage report to include executed code from the referenced code parts only. Example 9.2 shows an example.

Note

If a method is specificed with the `@covers` annotation, only the referenced method will be considered as covered, but not methods called by this method. Hence, when a covered method is refactored using the *extract method* refactoring, corresponding `@covers` annotations need to be added. This is the reason it is recommended to use this annotation with class scope, not with method scope.

Example 9.2: Test class that specifies which class it wants to cover

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

/**
 * @covers \Invoice
 * @uses \Money
 */
final class InvoiceTest extends TestCase
{
    private $invoice;

    protected function setUp(): void
    {
        $this->invoice = new Invoice;
    }

    public function testAmountInitiallyIsEmpty(): void
    {
        $this->assertEquals(new Money, $this->invoice->getAmount());
    }
}
```

Example 9.3: Tests that specify which method they want to cover

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class BankAccountTest extends TestCase
{
    private $ba;

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

    /**
     * @covers \BankAccount::getBalance
     */
    public function testBalanceIsInitiallyZero(): void
```

(continues on next page)

(continued from previous page)

```
    {
        $this->assertSame(0, $this->ba->getBalance());
    }

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

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

            return;
        }

        $this->fail();
    }

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

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

            return;
        }

        $this->fail();
    }

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

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

with unit tests.

Example 9.4: A test that specifies that no method should be covered

```
<?php declare(strict_types=1);
use PHPUnit\DbUnit\TestCase

final class GuestbookIntegrationTest extends TestCase
{
    /**
     * @coversNothing
     */
    public function testAddEntry(): void
    {
        $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);
    }
}
```

9.5 Edge Cases

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

```
<?php declare(strict_types=1);
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 10

Logging

PHPUnit can produce several types of logfiles.

10.1 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. 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>
```

10.2 Code Coverage (XML)

The XML format for code coverage information logging produced by PHPUnit is loosely based upon the one used by 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"/>
```

(continues on next page)

(continued from previous page)

```
      <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>
```

10.3 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 coverage overview while working on a small set of classes. For bigger projects this output can be useful to get an 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 *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 11

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.

11.1 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.

11.2 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 11.1, the `assertTrue()` method is a wrapper around the `isTrue()` and `assertThat()` methods: `isTrue()` creates a matcher object that is passed on to `assertThat()` for evaluation.

Example 11.1: The assertTrue() and isTrue() methods of the PHPUnit\Framework\Assert class

```
<?php declare(strict_types=1);
namespace PHPUnit\Framework;

use PHPUnit\Framework\Constraint\IsTrue;

abstract class Assert
{
    // ...

    public static function assertTrue($condition, string $message = ''): void
    {
        static::assertThat($condition, static::isTrue(), $message);
```

(continues on next page)

(continued from previous page)

```
    }

    // ...

    public static function isTrue(): IsTrue
    {
        return new IsTrue;
    }

    // ...
}
```

Example 11.2 shows how `PHPUnit\Framework\Constraint\IsTrue` extends the abstract base class for matcher objects (or constraints), `PHPUnit\Framework\Constraint`.

Example 11.2: The PHPUnit\FrameworkConstraint\IsTrue class

```
<?php declare(strict_types=1);
namespace PHPUnit\Framework\Constraint;

use PHPUnit\Framework\Constraint;

final class IsTrue extends Constraint
{
    public function toString(): string
    {
        return 'is true';
    }

    protected function matches($other): bool
    {
        return $other === 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.

11.3 Implement PHPUnit\Framework\TestListener

Example 11.3 shows a simple implementation of the `PHPUnit\Framework\TestListener` interface.

Example 11.3: A simple test listener

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;
use PHPUnit\Framework\TestListener;

final class SimpleTestListener implements TestListener
{
    public function addError(PHPUnit\Framework\Test $test, \Throwable $e, float
↪$time): void
```

(continues on next page)

(continued from previous page)

```
    {
        printf("Error while running test '%s'.\n", $test->getName());
    }

    public function addWarning(PHPUnit\Framework\Test $test,␣
↪PHPUnit\Framework\Warning $e, float $time): void
    {
        printf("Warning while running test '%s'.\n", $test->getName());
    }

    public function addFailure(PHPUnit\Framework\Test $test,␣
↪PHPUnit\Framework\AssertionFailedError $e, float $time): void
    {
        printf("Test '%s' failed.\n", $test->getName());
    }

    public function addIncompleteTest(PHPUnit\Framework\Test $test, \Throwable $e,␣
↪float $time): void
    {
        printf("Test '%s' is incomplete.\n", $test->getName());
    }

    public function addRiskyTest(PHPUnit\Framework\Test $test, \Throwable $e, float
↪$time): void
    {
        printf("Test '%s' is deemed risky.\n", $test->getName());
    }

    public function addSkippedTest(PHPUnit\Framework\Test $test, \Throwable $e, float
↪$time): void
    {
        printf("Test '%s' has been skipped.\n", $test->getName());
    }

    public function startTest(PHPUnit\Framework\Test $test): void
    {
        printf("Test '%s' started.\n", $test->getName());
    }

    public function endTest(PHPUnit\Framework\Test $test, float $time): void
    {
        printf("Test '%s' ended.\n", $test->getName());
    }

    public function startTestSuite(PHPUnit\Framework\TestSuite $suite): void
    {
        printf("TestSuite '%s' started.\n", $suite->getName());
    }

    public function endTestSuite(PHPUnit\Framework\TestSuite $suite): void
    {
        printf("TestSuite '%s' ended.\n", $suite->getName());
    }
}
```

Example 11.4 shows how to use the `PHPUnit\Framework\TestListenerDefaultImplementation` trait, which lets you specify only the interface methods that are interesting for your use case, while providing empty imple-

mentations for all the others.

Example 11.4: Using test listener default implementation trait

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestListener;
use PHPUnit\Framework\TestListenerDefaultImplementation;

final class ShortTestListener implements TestListener
{
    use TestListenerDefaultImplementation;

    public function endTest(PHPUnit\Framework\Test $test, float $time): void
    {
        printf("Test '%s' ended.\n", $test->getName());
    }
}
```

In *Test Listeners* you can see how to configure PHPUnit to attach your test listener to the test execution.

CHAPTER 12

Assertions

This appendix lists the various assertion methods that are available.

12.1 Static vs. Non-Static Usage of Assertion Methods

PHPUnit's assertions are implemented in `PHPUnit\Framework\Assert`. `PHPUnit\Framework\TestCase` inherits from `PHPUnit\Framework\Assert`.

The assertion methods are declared static and can be invoked from any context using `PHPUnit\Framework\Assert::assertTrue()`, for instance, or using `$this->assertTrue()` or `self::assertTrue()`, for instance, in a class that extends `PHPUnit\Framework\TestCase`.

In fact, you can even use global function wrappers such as `assertTrue()` in any context (including classes that extend `PHPUnit\Framework\TestCase`) when you (manually) include the `src/Framework/Assert/Functions.php` sourcecode file that comes with PHPUnit.

A common question, especially from developers new to PHPUnit, is whether using `$this->assertTrue()` or `self::assertTrue()`, for instance, is "the right way" to invoke an assertion. The short answer is: there is no right way. And there is no wrong way, either. It is a matter of personal preference.

For most people it just "feels right" to use `$this->assertTrue()` because the test method is invoked on a test object. The fact that the assertion methods are declared static allows for (re)using them outside the scope of a test object. Lastly, the global function wrappers allow developers to type less characters (`assertTrue()` instead of `$this->assertTrue()` or `self::assertTrue()`).

12.2 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 12.1: Usage of assertArrayHasKey()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class ArrayHasKeyTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertArrayHasKey('foo', ['bar' => 'baz']);
    }
}
```

```
$ phpunit ArrayHasKeyTest
PHPUnit 7.0.0 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.
```

12.3 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 12.2: Usage of assertClassHasAttribute()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class ClassHasAttributeTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertClassHasAttribute('foo', stdClass::class);
    }
}
```

```
$ phpunit ClassHasAttributeTest
PHPUnit 7.0.0 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.
```

12.4 assertArraySubset()

```
assertArraySubset(array $subset, array $array[, bool $strict = false, 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 12.3: Usage of assertArraySubset()

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class ArraySubsetTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertArraySubset(['config' => ['key-a', 'key-b']], ['config' => ['key-
↪a']]);
    }
}
```

```
$ phpunit ArraySubsetTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ArraySubset::testFailure
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.
```

12.5 assertClassHasStaticAttribute()

`assertClassHasStaticAttribute(string $attributeName, string $className[, 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 12.4: Usage of assertClassHasStaticAttribute()

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class ClassHasStaticAttributeTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertClassHasStaticAttribute('foo', stdClass::class);
    }
}
```

```
$ phpunit ClassHasStaticAttributeTest
PHPUnit 7.0.0 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.
```

12.6 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 12.5: Usage of assertContains()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class ContainsTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertContains(4, [1, 2, 3]);
    }
}
```

```
$ phpunit ContainsTest
PHPUnit 7.0.0 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 12.6: Usage of assertContains()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class ContainsTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertContains('baz', 'foobar');
    }
}
```

```
$ phpunit ContainsTest
PHPUnit 7.0.0 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 12.7: Usage of assertContains() with $ignoreCase

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class ContainsTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertContains('foo', 'FooBar');
    }

    public function testOK(): void
    {
        $this->assertContains('foo', 'FooBar', '', true);
    }
}
```

```
$ phpunit ContainsTest
PHPUnit 7.0.0 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.
```

12.7 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 12.8: Usage of assertContainsOnly()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class ContainsOnlyTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertContainsOnly('string', ['1', '2', 3]);
    }
}
```

```
$ phpunit ContainsOnlyTest
PHPUnit 7.0.0 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.
```

12.8 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 12.9: Usage of assertContainsOnlyInstancesOf()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class ContainsOnlyInstancesOfTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertContainsOnlyInstancesOf(
```

(continues on next page)

(continued from previous page)

```
            Foo::class,
            [new Foo, new Bar, new Foo]
        );
    }
}
```

```
$ phpunit ContainsOnlyInstancesOfTest
PHPUnit 7.0.0 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.
```

12.9 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 12.10: Usage of assertCount()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class CountTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertCount(0, ['foo']);
    }
}
```

```
$ phpunit CountTest
PHPUnit 7.0.0 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.
```

12.10 assertDirectoryExists()

`assertDirectoryExists(string $directory[, string $message = ''])`

Reports an error identified by `$message` if the directory specified by `$directory` does not exist.

`assertDirectoryNotExists()` is the inverse of this assertion and takes the same arguments.

Example 12.11: Usage of assertDirectoryExists()

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class DirectoryExistsTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertDirectoryExists('/path/to/directory');
    }
}
```

```
$ phpunit DirectoryExistsTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) DirectoryExistsTest::testFailure
Failed asserting that directory "/path/to/directory" exists.

/home/sb/DirectoryExistsTest.php:6

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

12.11 assertDirectoryIsReadable()

`assertDirectoryIsReadable(string $directory[, string $message = ''])`

Reports an error identified by `$message` if the directory specified by `$directory` is not a directory or is not readable.

`assertDirectoryNotIsReadable()` is the inverse of this assertion and takes the same arguments.

Example 12.12: Usage of assertDirectoryIsReadable()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class DirectoryIsReadableTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertDirectoryIsReadable('/path/to/directory');
    }
}
```

```
$ phpunit DirectoryIsReadableTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) DirectoryIsReadableTest::testFailure
Failed asserting that "/path/to/directory" is readable.

/home/sb/DirectoryIsReadableTest.php:6

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

12.12 assertDirectoryIsWritable()

`assertDirectoryIsWritable(string $directory[, string $message = ''])`

Reports an error identified by `$message` if the directory specified by `$directory` is not a directory or is not writable.

`assertDirectoryNotIsWritable()` is the inverse of this assertion and takes the same arguments.

Example 12.13: Usage of assertDirectoryIsWritable()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class DirectoryIsWritableTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertDirectoryIsWritable('/path/to/directory');
    }
}
```

```
$ phpunit DirectoryIsWritableTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.
```

```
F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) DirectoryIsWritableTest::testFailure
Failed asserting that "/path/to/directory" is writable.

/home/sb/DirectoryIsWritableTest.php:6

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

12.13 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 12.14: Usage of assertEmpty()

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class EmptyTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertEmpty(['foo']);
    }
}
```

```
$ phpunit EmptyTest
PHPUnit 7.0.0 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.
```

12.14 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 12.15: Usage of assertEqualXMLStructure()

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class EqualXMLStructureTest extends TestCase
{
    public function testFailureWithDifferentNodeNames(): void
    {
        $expected = new DOMElement('foo');
        $actual = new DOMElement('bar');

        $this->assertEqualXMLStructure($expected, $actual);
    }

    public function testFailureWithDifferentNodeAttributes(): void
    {
        $expected = new DOMDocument;
        $expected->loadXML('<foo bar="true" />');

        $actual = new DOMDocument;
        $actual->loadXML('<foo/>');

        $this->assertEqualXMLStructure(
          $expected->firstChild, $actual->firstChild, true
        );
    }

    public function testFailureWithDifferentChildrenCount(): void
    {
        $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(): void
    {
        $expected = new DOMDocument;
        $expected->loadXML('<foo><bar/><bar/><bar/></foo>');

        $actual = new DOMDocument;
        $actual->loadXML('<foo><baz/><baz/><baz/></foo>');
```

(continues on next page)

(continued from previous page)

```
        $this->assertEqualXMLStructure(
          $expected->firstChild, $actual->firstChild
        );
    }
}
```

```
$ phpunit EqualXMLStructureTest
PHPUnit 7.0.0 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.
```

12.15 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 12.16: Usage of assertEquals()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class EqualsTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertEquals(1, 0);
    }

    public function testFailure2(): void
    {
        $this->assertEquals('bar', 'baz');
    }

    public function testFailure3(): void
    {
        $this->assertEquals("foo\nbar\nbaz\n", "foo\nbah\nbaz\n");
    }
}
```

```
$ phpunit EqualsTest
PHPUnit 7.0.0 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 `$message` if the absolute difference between two floats `$expected` and `$actual` is greater than `$delta`. If the absolute difference between two floats `$expected` and `$actual` is less than *or equal to* `$delta`, the assertion will pass.

Please read "What Every Computer Scientist Should Know About Floating-Point Arithmetic" to understand why `$delta` is neccessary.

Example 12.17: Usage of assertEquals() with floats

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class EqualsTest extends TestCase
{
    public function testSuccess(): void
    {
        $this->assertEquals(1.0, 1.1, '', 0.1);
    }

    public function testFailure(): void
    {
        $this->assertEquals(1.0, 1.1);
    }
}
```

```
$ phpunit EqualsTest
PHPUnit 7.0.0 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 12.18: Usage of assertEquals() with DOMDocument objects

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class EqualsTest extends TestCase
{
    public function testFailure(): void
    {
        $expected = new DOMDocument;
        $expected->loadXML('<foo><bar/></foo>');

        $actual = new DOMDocument;
        $actual->loadXML('<bar><foo/></bar>');

        $this->assertEquals($expected, $actual);
    }
}
```

```
$ phpunit EqualsTest
PHPUnit 7.0.0 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 12.19: Usage of assertEquals() with objects

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class EqualsTest extends TestCase
{
    public function testFailure(): void
    {
        $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 7.0.0 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 12.20: Usage of assertEquals() with arrays

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class EqualsTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertEquals(['a', 'b', 'c'], ['a', 'c', 'd']);
    }
}
```

```
$ phpunit EqualsTest
PHPUnit 7.0.0 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.
```

12.16 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 12.21: Usage of assertFalse()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;
```

(continues on next page)

(continued from previous page)

```
final class FalseTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertFalse(true);
    }
}
```

```
$ phpunit FalseTest
PHPUnit 7.0.0 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.
```

12.17 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 12.22: Usage of assertFileEquals()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class FileEqualsTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertFileEquals('/home/sb/expected', '/home/sb/actual');
    }
}
```

```
$ phpunit FileEqualsTest
PHPUnit 7.0.0 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.
```

12.18 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 12.23: Usage of assertFileExists()

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class FileExistsTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertFileExists('/path/to/file');
    }
}
```

```
$ phpunit FileExistsTest
PHPUnit 7.0.0 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.
```

12.19 assertFileIsReadable()

`assertFileIsReadable(string $filename[, string $message = ''])`

Reports an error identified by `$message` if the file specified by `$filename` is not a file or is not readable.

`assertFileNotIsReadable()` is the inverse of this assertion and takes the same arguments.

Example 12.24: Usage of assertFileIsReadable()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class FileIsReadableTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertFileIsReadable('/path/to/file');
    }
}
```

```
$ phpunit FileIsReadableTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) FileIsReadableTest::testFailure
Failed asserting that "/path/to/file" is readable.

/home/sb/FileIsReadableTest.php:6

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

12.20 assertFileIsWritable()

`assertFileIsWritable(string $filename[, string $message = ''])`

Reports an error identified by `$message` if the file specified by `$filename` is not a file or is not writable.

`assertFileNotIsWritable()` is the inverse of this assertion and takes the same arguments.

Example 12.25: Usage of assertFileIsWritable()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class FileIsWritableTest extends TestCase
{
    public function testFailure(): void
    {
```

(continues on next page)

(continued from previous page)

```
        $this->assertFileIsWritable('/path/to/file');
    }
}
```

```
$ phpunit FileIsWritableTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) FileIsWritableTest::testFailure
Failed asserting that "/path/to/file" is writable.

/home/sb/FileIsWritableTest.php:6

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

12.21 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 12.26: Usage of assertGreaterThan()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class GreaterThanTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertGreaterThan(2, 1);
    }
}
```

```
$ phpunit GreaterThanTest
PHPUnit 7.0.0 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.
```

12.22 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 12.27: Usage of assertGreaterThanOrEqual()

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class GreatThanOrEqualTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertGreaterThanOrEqual(2, 1);
    }
}
```

```
$ phpunit GreaterThanOrEqualTest
PHPUnit 7.0.0 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.
```

12.23 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 12.28: Usage of assertInfinite()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class InfiniteTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertInfinite(1);
    }
}
```

```
$ phpunit InfiniteTest
PHPUnit 7.0.0 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.
```

12.24 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 12.29: Usage of assertInstanceOf()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class InstanceOfTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertInstanceOf(RuntimeException::class, new Exception);
    }
}
```

```
$ phpunit InstanceOfTest
PHPUnit 7.0.0 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.
```

12.25 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 12.30: Usage of assertInternalType()

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class InternalTypeTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertInternalType('string', 42);
    }
}
```

```
$ phpunit InternalTypeTest
PHPUnit 7.0.0 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.
```

12.26 assertIsReadable()

`assertIsReadable(string $filename[, string $message = ''])`

Reports an error identified by `$message` if the file or directory specified by `$filename` is not readable.

`assertNotIsReadable()` is the inverse of this assertion and takes the same arguments.

Example 12.31: Usage of assertIsReadable()

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class IsReadableTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertIsReadable('/path/to/unreadable');
    }
}
```

```
$ phpunit IsReadableTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) IsReadableTest::testFailure
Failed asserting that "/path/to/unreadable" is readable.

/home/sb/IsReadableTest.php:6

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

12.27 assertIsWritable()

`assertIsWritable(string $filename[, string $message = ''])`

Reports an error identified by `$message` if the file or directory specified by `$filename` is not writable.

`assertNotIsWritable()` is the inverse of this assertion and takes the same arguments.

Example 12.32: Usage of assertIsWritable()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class IsWritableTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertIsWritable('/path/to/unwritable');
    }
}
```

```
$ phpunit IsWritableTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) IsWritableTest::testFailure
Failed asserting that "/path/to/unwritable" is writable.

/home/sb/IsWritableTest.php:6

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

12.28 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 12.33: Usage of assertJsonFileEqualsJsonFile()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class JsonFileEqualsJsonFileTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertJsonFileEqualsJsonFile(
          'path/to/fixture/file', 'path/to/actual/file');
    }
}
```

```
$ phpunit JsonFileEqualsJsonFileTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.
```

```
F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) JsonFileEqualsJsonFile::testFailure
Failed asserting that '{"Mascot":"Tux"}' matches JSON string "["Mascott",␣
↪"Tux", "OS", "Linux"]".

/home/sb/JsonFileEqualsJsonFileTest.php:5

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

12.29 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 12.34: Usage of assertJsonStringEqualsJsonFile()

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class JsonStringEqualsJsonFileTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertJsonStringEqualsJsonFile(
            'path/to/fixture/file', json_encode(['Mascot' => 'ux'])
        );
    }
}
```

```
$ phpunit JsonStringEqualsJsonFileTest
PHPUnit 7.0.0 by Sebastian Bergmann and contributors.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) JsonStringEqualsJsonFile::testFailure
Failed asserting that '{"Mascot":"ux"}' matches JSON string "{"Mascott":"Tux"}
↪".

/home/sb/JsonStringEqualsJsonFileTest.php:5

FAILURES!
```

```
Tests: 1, Assertions: 3, Failures: 1.
```

12.30 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 12.35: Usage of assertJsonStringEqualsJsonString()

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class JsonStringEqualsJsonStringTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertJsonStringEqualsJsonString(
            json_encode(['Mascot' => 'Tux']),
            json_encode(['Mascot' => 'ux'])
        );
    }
}
```

```
$ phpunit JsonStringEqualsJsonStringTest
PHPUnit 7.0.0 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 (
 -    'Mascot' => 'Tux'
 +    'Mascot' => 'ux'
)

/home/sb/JsonStringEqualsJsonStringTest.php:5

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.
```

12.31 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 12.36: Usage of assertLessThan()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class LessThanTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertLessThan(1, 2);
    }
}
```

```
$ phpunit LessThanTest
PHPUnit 7.0.0 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.
```

12.32 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 12.37: Usage of assertLessThanOrEqual()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class LessThanOrEqualTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertLessThanOrEqual(1, 2);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

```
$ phpunit LessThanOrEqualTest
PHPUnit 7.0.0 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.
```

12.33 assertNan()

```
assertNan(mixed $variable[, string $message = ''])
```

Reports an error identified by `$message` if `$variable` is not `NAN`.

Example 12.38: Usage of assertNan()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class NanTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertNan(1);
    }
}
```

```
$ phpunit NanTest
PHPUnit 7.0.0 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.
```

12.34 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 12.39: Usage of assertNull()

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class NullTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertNull('foo');
    }
}
```

```
$ phpunit NotNullTest
PHPUnit 7.0.0 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.
```

12.35 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 12.40: Usage of assertObjectHasAttribute()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class ObjectHasAttributeTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertObjectHasAttribute('foo', new stdClass);
    }
}
```

```
$ phpunit ObjectHasAttributeTest
PHPUnit 7.0.0 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.
```

12.36 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 12.41: Usage of assertRegExp()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class RegExpTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertRegExp('/foo/', 'bar');
    }
}
```

```
$ phpunit RegExpTest
PHPUnit 7.0.0 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.
```

12.37 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 12.42: Usage of assertStringMatchesFormat()

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class StringMatchesFormatTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertStringMatchesFormat('%i', 'foo');
    }
}
```

```
$ phpunit StringMatchesFormatTest
PHPUnit 7.0.0 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.
- `%%`: A literal percent character: `%`.

12.38 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 12.43: Usage of assertStringMatchesFormatFile()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class StringMatchesFormatFileTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertStringMatchesFormatFile('/path/to/expected.txt', 'foo');
    }
}
```

```
$ phpunit StringMatchesFormatFileTest
PHPUnit 7.0.0 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.
```

12.39 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 12.44: Usage of assertSame()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class SameTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertSame('2204', 2204);
    }
}
```

```
$ phpunit SameTest
PHPUnit 7.0.0 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 12.45: Usage of assertSame() with objects

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class SameTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertSame(new stdClass, new stdClass);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

```
$ phpunit SameTest
PHPUnit 7.0.0 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.
```

12.40 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 12.46: Usage of assertStringEndsWith()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class StringEndsWithTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertStringEndsWith('suffix', 'foo');
    }
}
```

```
$ phpunit StringEndsWithTest
PHPUnit 7.0.0 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.
```

12.41 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 12.47: Usage of assertStringEqualsFile()

```php
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class StringEqualsFileTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertStringEqualsFile('/home/sb/expected', 'actual');
    }
}
```

```
$ phpunit StringEqualsFileTest
PHPUnit 7.0.0 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.
```

12.42 assertStringStartsWith()

`assertStringStartsWith(string $prefix, string $string[, string $message = ''])`

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 12.48: Usage of assertStringStartsWith()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class StringStartsWithTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertStringStartsWith('prefix', 'foo');
    }
}
```

```
$ phpunit StringStartsWithTest
PHPUnit 7.0.0 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.
```

12.43 assertThat()

More complex assertions can be formulated using the `PHPUnit\Framework\Constraint` classes. They can be evaluated using the `assertThat()` method. Example 12.49 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 12.49: Usage of assertThat()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class BiscuitTest extends TestCase
```

(continues on next page)

(continued from previous page)

```
{
    public function testEquals(): void
    {
        $theBiscuit = new Biscuit('Ginger');
        $myBiscuit  = new Biscuit('Ginger');

        $this->assertThat(
          $theBiscuit,
          $this->logicalNot(
            $this->equalTo($myBiscuit)
          )
        );
    }
}
```

Table 12.1 shows the available `PHPUnit\Framework\Constraint` classes.

Constraint
`PHPUnit\Framework\Constraint\Attribute attribute(PHPUnit\Framework\Constraint $constraint`
`PHPUnit\Framework\Constraint\IsAnything anything()`
`PHPUnit\Framework\Constraint\ArrayHasKey arrayHasKey(mixed $key)`
`PHPUnit\Framework\Constraint\TraversableContains contains(mixed $value)`
`PHPUnit\Framework\Constraint\TraversableContainsOnly containsOnly(string $type)`
`PHPUnit\Framework\Constraint\TraversableContainsOnly containsOnlyInstancesOf(string $clas`
`PHPUnit\Framework\Constraint\IsEqual equalTo($value, $delta = 0, $maxDepth = 10)`
`PHPUnit\Framework\Constraint\Attribute attributeEqualTo($attributeName, $value, $delta =`
`PHPUnit\Framework\Constraint\DirectoryExists directoryExists()`
`PHPUnit\Framework\Constraint\FileExists fileExists()`
`PHPUnit\Framework\Constraint\IsReadable isReadable()`
`PHPUnit\Framework\Constraint\IsWritable isWritable()`
`PHPUnit\Framework\Constraint\GreaterThan greaterThan(mixed $value)`
`PHPUnit\Framework\Constraint\Or greaterThanOrEqual(mixed $value)`
`PHPUnit\Framework\Constraint\ClassHasAttribute classHasAttribute(string $attributeName)`
`PHPUnit\Framework\Constraint\ClassHasStaticAttribute classHasStaticAttribute(string $attr`
`PHPUnit\Framework\Constraint\ObjectHasAttribute objectHasAttribute(string $attributeName)`
`PHPUnit\Framework\Constraint\IsIdentical identicalTo(mixed $value)`
`PHPUnit\Framework\Constraint\IsFalse isFalse()`
`PHPUnit\Framework\Constraint\IsInstanceOf isInstanceOf(string $className)`
`PHPUnit\Framework\Constraint\IsNull isNull()`
`PHPUnit\Framework\Constraint\IsTrue isTrue()`
`PHPUnit\Framework\Constraint\IsType isType(string $type)`
`PHPUnit\Framework\Constraint\LessThan lessThan(mixed $value)`
`PHPUnit\Framework\Constraint\Or lessThanOrEqual(mixed $value)`
`logicalAnd()`
`logicalNot(PHPUnit\Framework\Constraint $constraint)`
`logicalOr()`
`logicalXor()`
`PHPUnit\Framework\Constraint\PCREMatch matchesRegularExpression(string $pattern)`
`PHPUnit\Framework\Constraint\StringContains stringContains(string $string, bool $case)`
`PHPUnit\Framework\Constraint\StringEndsWith stringEndsWith(string $suffix)`

Table 1

Constraint
PHPUnit\Framework\Constraint\StringStartsWith stringStartsWith(string $prefix)

12.44 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 12.50: Usage of assertTrue()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class TrueTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertTrue(false);
    }
}
```

```
$ phpunit TrueTest
PHPUnit 7.0.0 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.
```

12.45 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 12.51: Usage of assertXmlFileEqualsXmlFile()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class XmlFileEqualsXmlFileTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertXmlFileEqualsXmlFile(
          '/home/sb/expected.xml', '/home/sb/actual.xml');
    }
}
```

```
$ phpunit XmlFileEqualsXmlFileTest
PHPUnit 7.0.0 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.
```

12.46 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 12.52: Usage of assertXmlStringEqualsXmlFile()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;
```

(continues on next page)

(continued from previous page)

```
final class XmlStringEqualsXmlFileTest extends TestCase
{
    public function testFailure(): void
    {
        $this->assertXmlStringEqualsXmlFile(
          '/home/sb/expected.xml', '<foo><baz/></foo>');
    }
}
```

```
$ phpunit XmlStringEqualsXmlFileTest
PHPUnit 7.0.0 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.
```

12.47 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 12.53: Usage of assertXmlStringEqualsXmlString()

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class XmlStringEqualsXmlStringTest extends TestCase
{
    public function testFailure(): void
```

(continues on next page)

(continued from previous page)

```
    {
        $this->assertXmlStringEqualsXmlString(
          '<foo><bar/></foo>', '<foo><baz/></foo>');
    }
}
```

```
$ phpunit XmlStringEqualsXmlStringTest
PHPUnit 7.0.0 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.
```

CHAPTER 13

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 a 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.

13.1 @author

The `@author` annotation is an alias for the `@group` annotation (see *@group*) and allows to filter tests based on their authors.

13.2 @after

The `@after` annotation can be used to specify methods that should be called after each test method in a test case class.

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class MyTest extends TestCase
```

(continues on next page)

(continued from previous page)

```
{
    /**
     * @after
     */
    public function tearDownSomeFixtures(): void
    {
        // ...
    }

    /**
     * @after
     */
    public function tearDownSomeOtherFixtures(): void
    {
        // ...
    }
}
```

13.3 @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.

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class MyTest extends TestCase
{
    /**
     * @afterClass
     */
    public static function tearDownSomeSharedFixtures(): void
    {
        // ...
    }

    /**
     * @afterClass
     */
    public static function tearDownSomeOtherSharedFixtures(): void
    {
        // ...
    }
}
```

13.4 @backupGlobals

The backup and restore operations for global variables can be completely disabled for all tests of a test case class like this

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

/**
 * @backupGlobals disabled
 */
final 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:

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

/**
 * @backupGlobals disabled
 */
final class MyTest extends TestCase
{
    /**
     * @backupGlobals enabled
     */
    public function testThatInteractsWithGlobalVariables(): void
    {
        // ...
    }
}
```

13.5 @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:

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

/**
 * @backupStaticAttributes enabled
 */
final class MyTest extends TestCase
{
    /**
     * @backupStaticAttributes disabled
     */
    public function testThatInteractsWithStaticAttributes(): void
    {
        // ...
    }
}
```

Note

`@backupStaticAttributes` is limited by PHP internals and may cause unintended static values to persist and leak into subsequent tests in some circumstances.

See *Global State* for details.

13.6 @before

The `@before` annotation can be used to specify methods that should be called before each test method in a test case class.

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class MyTest extends TestCase
{
    /**
     * @before
     */
    public function setupSomeFixtures(): void
    {
        // ...
    }

    /**
     * @before
     */
    public function setupSomeOtherFixtures(): void
    {
        // ...
    }
}
```

13.7 @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.

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class MyTest extends TestCase
{
    /**
     * @beforeClass
     */
    public static function setUpSomeSharedFixtures(): void
    {
        // ...
    }

    /**
     * @beforeClass
```

(continues on next page)

(continued from previous page)

```
     */
    public static function setUpSomeOtherSharedFixtures(): void
    {
        // ...
    }
}
```

13.8 @codeCoverageIgnore*

The `@codeCoverageIgnore`, `@codeCoverageIgnoreStart` and `@codeCoverageIgnoreEnd` annotations can be used to exclude lines of code from the coverage analysis.

For usage see *Ignoring Code Blocks*.

13.9 @covers

The `@covers` annotation can be used in the test code to specify which parts of the code it is supposed to test:

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

If provided, this effectively filters the code coverage report to include executed code from the referenced code parts only. This will make sure that code is only marked as covered if there are dedicated tests for it, but not if it used indirectly by the tests for a different class, thus avoiding false positives for code coverage.

This annotation can be added to the docblock of the test class or the individual test methods. The recommended way is to add the annotation to the docblock of the test class, not to the docblock of the test methods.

When the `forceCoversAnnotation` configuration option in the *configuration file* is set to `true`, every test method needs to have an associated `@covers` annotation (either on the test class or the individual test method).

Table 13.1 shows the syntax of the `@covers` annotation. The section *Specifying Covered Code Parts* provides longer examples for using the annotation.

Pleae note that this annotation requires a fully-qualified class name (FQCN). To make this more obvious to the reader, it is recommended to use a leading backslash (even if this not required for the annotation to work correctly).

Table 13.1: Annotations for specifying which methods are covered by a test

Annotation	Description
`@covers ClassName::methodName` (not recommended)	Specifies that the annotated test method covers the specified method.
`@covers ClassName` (recommended)	Specifies that the annotated test method covers all methods of a given class.
`@covers ClassName<extended>` (not recommended)	Specifies that the annotated test method covers all methods of a given class and its parent class(es).
`@covers ClassName::<public>` (not recommended)	Specifies that the annotated test method covers all public methods of a given class.
`@covers ClassName::<protected>` (not recommended)	Specifies that the annotated test method covers all protected methods of a given class.
`@covers ClassName::<private>` (not recommended)	Specifies that the annotated test method covers all private methods of a given class.
`@covers ClassName::<!public>` (not recommended)	Specifies that the annotated test method covers all methods of a given class that are not public.
`@covers ClassName::<!protected>` (not recommended)	Specifies that the annotated test method covers all methods of a given class that are not protected.
`@covers ClassName::<!private>` (not recommended)	Specifies that the annotated test method covers all methods of a given class that are not private.
`@covers ::functionName` (recommended)	Specifies that the annotated test method covers the specified global function.

13.10 @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 13.1.

Pleae note that this annotation requires a fully-qualified class name (FQCN). To make this more obvious to the reader, it is recommended to use a leading backslash (even if this not required for the annotation to work correctly).

Example 13.1: Using @coversDefaultClass to shorten annotations

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

/**
 * @coversDefaultClass \Foo\CoveredClass
 */
final class CoversDefaultClassTest extends TestCase
{
    /**
     * @covers ::publicMethod
     */
    public function testSomething(): void
    {
        $o = new Foo\CoveredClass;
        $o->publicMethod();
    }
}
?>
```

13.11 @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 *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.

13.12 @dataProvider

A test method can accept arbitrary arguments. These arguments are to be provided by one or more data provider methods (`provider()` in *Using a data provider that returns an array of arrays*). The data provider method to be used is specified using the `@dataProvider` annotation.

See *Data Providers* for more details.

13.13 @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. *Using the @depends annotation to express dependencies* shows how to use the `@depends` annotation to express dependencies between test methods.

See *Test Dependencies* for more details.

13.14 @doesNotPerformAssertions

Prevents a test that performs no assertions from being considered risky.

13.15 @group

A test can be tagged as belonging to one or more groups using the `@group` annotation like this

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class MyTest extends TestCase
{
    /**
     * @group specification
     */
    public function testSomething(): void
    {
    }

    /**
     * @group regression
     * @group bug2204
```

(continues on next page)

(continued from previous page)

```
     */
    public function testSomethingElse(): void
    {
    }
}
```

The `@group` annotation can also be provided for the test class. It is then "inherited" to all test methods of that test class.

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.

13.16 @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.

13.17 @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.

13.18 @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.

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class MyTest extends TestCase
{
    /**
     * @runInSeparateProcess
     * @preserveGlobalState disabled
     */
    public function testInSeparateProcess(): void
    {
        // ...
    }
}
```

13.19 @requires

The `@requires` annotation can be used to 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 *Possible @requires usages*

13.20 @runTestsInSeparateProcesses

Indicates that all tests in a test class should be run in a separate PHP process.

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

/**
 * @runTestsInSeparateProcesses
 */
final 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 *@preserveGlobalState* for information on how to fix this.

13.21 @runInSeparateProcess

Indicates that a test should be run in a separate PHP process.

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

final class MyTest extends TestCase
{
    /**
     * @runInSeparateProcess
     */
    public function testInSeparateProcess(): void
    {
        // ...
    }
}
```

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 *@preserveGlobalState* for information on how to fix this.

13.22 @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.

13.23 @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(): void
{
    $this->assertSame(0, $this->ba->getBalance());
}
```

13.24 @testdox

Specifies an alternative description used when generating the agile documentation sentences.

The `@testdox` annotation can be applied to both test classes and test methods.

```
<?php declare(strict_types=1);
use PHPUnit\Framework\TestCase;

/**
 * @testdox A bank account
 */
final class BankAccountTest extends TestCase
{
    /**
     * @testdox has an initial balance of zero
     */
    public function balanceIsInitiallyZero(): void
    {
        $this->assertSame(0, $this->ba->getBalance());
    }
}
```

Note

Prior to PHPUnit 7.0 (due to a bug in the annotation parsing), using the `@testdox` annotation also activated the behaviour of the `@test` annotation.

13.25 @testWith

Instead of implementing a method for use with `@dataProvider`, you can define a data set using the `@testWith` annotation.

A data set consists of one or many elements. To define a data set with multiple elements, define each element in a separate line. Each element of the data set must be an array defined in JSON.

See *Data Providers* to learn more about passing a set of data to a test.

```
/**
 * @testWith ["test", 4]
 *           ["longer-string", 13]
 */
public function testStringLength(string $input, int $expectedLength): void
{
    $this->assertSame($expectedLength, strlen($input));
}
```

An object representation in JSON will be converted into an associative array.

```
/**
 * @testWith [{"day": "monday", "conditions": "sunny"}, ["day", "conditions"]]
 */
public function testArrayKeys(array $array, array $keys): void
{
    $this->assertSame($keys, array_keys($array));
}
```

13.26 @ticket

The `@ticket` annotation is an alias for the `@group` annotation (see *@group*) and allows to filter tests based on their ticket ID.

13.27 @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
 * @uses   \Money
 */
public function testMoneyCanBeDepositedInAccount(): void
{
```

(continues on next page)

(continued from previous page)

```
    // ...
}
```

Example 9.2 shows another example.

In addition to being helpful for persons reading the code, this annotation is useful in strict coverage mode where unintentionally covered code will cause a test to fail. See *Unintentionally Covered Code* for more information regarding strict coverage mode.

Please note that this annotation requires a fully-qualified class name (FQCN). To make this more obvious to the reader, it is recommended to use a leading backslash (even if this is not required for the annotation to work correctly).

CHAPTER 14

The XML Configuration File

14.1 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="https://schema.phpunit.de/7.0/phpunit.
↪xsd"
         backupGlobals="true"
         backupStaticAttributes="false"
         <!--bootstrap="/path/to/bootstrap.php"-->
         cacheTokens="false"
         colors="false"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         forceCoversAnnotation="false"
         printerClass="PHPUnitTextUIResultPrinter"
         <!--printerFile="/path/to/ResultPrinter.php"-->
         processIsolation="false"
         stopOnError="false"
         stopOnFailure="false"
         stopOnIncomplete="false"
         stopOnSkipped="false"
         stopOnRisky="false"
         testSuiteLoaderClass="PHPUnitRunnerStandardTestSuiteLoader"
         <!--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 *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`

A test will be marked as risky (see *Unintentionally Covered Code*) when it does not have a *@covers* annotation.

`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.

14.2 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/
↪files</directory>
    <file phpVersion="5.3.0" phpVersionOperator=">=">/path/to/MyTest.php</file>
  </testsuite>
</testsuites>
```

The `phpVersionOperator` attribute is optional and defaults to >=.

14.3 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 *@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`

14.4 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>
```

(continues on next page)

(continued from previous page)

```
    </exclude>
  </whitelist>
</filter>
```

14.5 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/html" lowUpperBound="35"
       highLowerBound="70"/>
  <log type="coverage-xml" target="tmp/report/xml"/>
  <log type="coverage-clover" target="/tmp/coverage.clover.xml"/>
  <log type="coverage-crap4j" target="/tmp/coverage.crap4j.xml"/>
  <log type="coverage-php" target="/tmp/coverage.serialized.php"/>
  <log type="coverage-text" target="php://stdout" showUncoveredFiles="false"/>
  <log type="junit" target="/tmp/logfile.xml"/>
  <log type="teamcity" target="/tmp/teamcity.txt"/>
  <log type="testdox-html" target="/tmp/testdox.html"/>
  <log type="testdox-text" target="/tmp/testdox.txt"/>
  <log type="testdox-xml" target="/tmp/testdox.xml"/>
</logging>
```

The XML configuration above corresponds to invoking the TextUI test runner with the following options:

- `--coverage-html /tmp/report/html`
- `--coverage-xml /tmp/report/xml`
- `--coverage-clover /tmp/coverage.clover.xml`
- `--coverage-crap4j /tmp/coverage.crap4j.xml`
- `--coverage-php /tmp/coverage.serialized.php`
- `--coverage-text`
- `> /tmp/logfile.txt`
- `--log-junit /tmp/logfile.xml`
- `--log-teamcity /tmp/teamcity.txt`
- `--testdox-html /tmp/testdox.html`
- `--testdox-text /tmp/testdox.txt`
- `--testdox-xml /tmp/testdox.xml`

The `lowUpperBound`, `highLowerBound`, 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.

14.6 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
);
```

14.7 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';
```

By default, environment variables are not overwritten if they exist already. To force overwriting existing variables, use the `force` attribute:

```
<php>
  <env name="foo" value="bar" force="true"/>
</php>
```

CHAPTER 15

Bibliography

[Meszaros2007] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*.

CHAPTER 16

Copyright

```
Copyright (c) 2005-2020 Sebastian Bergmann.

This work is licensed under the Creative Commons Attribution 3.0
Unported License.

A summary of the license is given below, followed by the full legal
text.

---------------------------------------------------------------------

You are free:

    * to Share - to copy, distribute and transmit the work
    * to Remix - to adapt the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by
the author or licensor (but not in any way that suggests that they
endorse you or your use of the work).

    * For any reuse or distribution, you must make clear to others
      the license terms of this work. The best way to do this is with
      a link to this web page.

    * Any of the above conditions can be waived if you get
      permission from the copyright holder.

    * Nothing in this license impairs or restricts the author's moral
      rights.

Your fair dealing and other rights are in no way affected by the
above.
```

(continues on next page)

(continued from previous page)

```
This is a human-readable summary of the Legal Code (the full
license) below.

=====================================================================

Creative Commons Legal Code
Attribution 3.0 Unported

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE
LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN
ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS
INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO
WARRANTIES REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS
LIABILITY FOR DAMAGES RESULTING FROM ITS USE.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS
CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS
PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE
WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW
IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND
AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS
LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU
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
```

(continues on next page)

(continued from previous page)

```
   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
```

(continues on next page)

(continued from previous page)

```
      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
```

(continues on next page)

(continued from previous page)

```
            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
```

(continues on next page)

(continued from previous page)

```
      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
```

(continues on next page)

(continued from previous page)

```
   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
```

(continues on next page)

(continued from previous page)

```
      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/.

======================================================================
```