PHPUnit Manual

Sebastian Bergmann

PHPUnit Manual

Sebastian Bergmann

Publication date Edition for PHPUnit 3.7. Updated on 2018-04-12.

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Automating Tests

Even good programmers make mistakes. The difference between a good programmer and a bad programmer is that the good programmer uses tests to detect mistakes as soon as possible. The sooner you test for a mistake the greater your chance of finding it and the less it will cost to find and fix. This explains why leaving testing until just before releasing software is so problematic. Most errors do not get caught at all, and the cost of fixing the ones you do catch is so high that you have to perform triage with the errors because you just cannot afford to fix them all.

Testing with PHPUnit is not a totally different activity from what you should already be doing. It is just a different way of doing it. The difference is between *testing*, that is, checking that your program behaves as expected, and *performing a battery of tests*, runnable code-fragments that automatically test the correctness of parts (units) of the software. These runnable code-fragments are called unit tests.

In this chapter we will go from simple `print`-based testing code to a fully automated test. Imagine that we have been asked to test PHP's built-in `array`. One bit of functionality to test is the function `count()`. For a newly created array we expect the `count()` function to return `0`. After we add an element, `count()` should return `1`. Example 1.1, “Testing array operations” shows what we want to test.

Example 1.1. Testing array operations

```
<?php
$fixture = array();
// $fixture is expected to be empty.

$fixture[] = 'element';
// $fixture is expected to contain one element.
?>
```

A really simple way to check whether we are getting the results we expect is to print the result of `count()` before and after adding the element (see Example 1.2, “Using print to test array operations”). If we get `0` and then `1`, `array` and `count()` behave as expected.

Example 1.2. Using print to test array operations

```
<?php
$fixture = array();
print count($fixture) . "\n";

$fixture[] = 'element';
print count($fixture) . "\n";
?>
```

```
0
1
```

Now, we would like to move from tests that require manual interpretation to tests that can run automatically. In Example 1.3, “Comparing expected and actual values to test array operations”, we write the comparison of the expected and actual values into the test code and print `ok` if the values are equal. If we ever see a `not ok` message, we know something is wrong.

Example 1.3. Comparing expected and actual values to test array operations

```
<?php
$fixture = array();
print count($fixture) == 0 ? "ok\n" : "not ok\n";
```

```
$fixture[] = 'element';
print count($fixture) == 1 ? "ok\n" : "not ok\n";
?>
```

```
ok
ok
```

We now factor out the comparison of expected and actual values into a function that raises an Exception when there is a discrepancy (Example 1.4, “Using an assertion function to test array operations”). This gives us two benefits: the writing of tests becomes easier and we only get output when something is wrong.

Example 1.4. Using an assertion function to test array operations

```
<?php
ini_set('error_log', '');
ini_set('display_errors', 'Off');

$fixture = array();
assertTrue(count($fixture) == 0);

$fixture[] = 'element';
assertTrue(count($fixture) == 1);

function assertTrue($condition)
{
    if (!$condition) {
        throw new Exception('Assertion failed.');
    }
}
?>
```

The test is now completely automated. Instead of just *testing* as we did with our first version, with this version we have an *automated test*.

The goal of using automated tests is to make fewer mistakes. While your code will still not be perfect, even with excellent tests, you will likely see a dramatic reduction in defects once you start automating tests. Automated tests give you justified confidence in your code. You can use this confidence to take more daring leaps in design (Refactoring), get along with your teammates better (Cross-Team Tests), improve relations with your customers, and go home every night with proof that the system is better now than it was this morning because of your efforts.

Chapter 2. PHPUnit's Goals

So far, we only have two tests for the `array` built-in and the `count()` function. When we start to test the numerous `array_*()` functions PHP offers, we will need to write a test for each of them. We could write the infrastructure for all these tests from scratch. However, it is much better to write a testing infrastructure once and then write only the unique parts of each test. PHPUnit is such an infrastructure.

A framework such as PHPUnit has to resolve a set of constraints, some of which seem always to conflict with each other. Simultaneously, tests should be:

Easy to learn to write. — If it's hard to learn how to write tests, developers will not learn to write them.

Easy to write. — If tests are not easy to write, developers will not write them.

Easy to read. — Test code should contain no extraneous overhead so that the test itself does not get lost in noise that surrounds it.

Easy to execute. — The tests should run at the touch of a button and present their results in a clear and unambiguous format.

Quick to execute. — Tests should run fast so they can be run hundreds or thousands of times a day.

Isolated. — The tests should not affect each other. If the order in which the tests are run changes, the results of the tests should not change.

Composable. — We should be able to run any number or combination of tests together. This is a corollary of isolation.

There are two main clashes between these constraints:

Easy to learn to write versus easy to write. — Tests do not generally require all the flexibility of a programming language. Many testing tools provide their own scripting language that only includes the minimum necessary features for writing tests. The resulting tests are easy to read and write because they have no noise to distract you from the content of the tests. However, learning yet another programming language and set of programming tools is inconvenient and clutters the mind.

Isolated versus quick to execute. — If you want the results of one test to have no effect on the results of another test, each test should create the full state of the world before it begins to execute and return the world to its original state when it finishes. However, setting up the world can take a long time: for example connecting to a database and initializing it to a known state using realistic data.

PHPUnit attempts to resolve these conflicts by using PHP as the testing language. Sometimes the full power of PHP is overkill for writing little straight-line tests, but by using PHP we leverage all the experience and tools programmers already have in place. Since we are trying to convince reluctant testers, lowering the barrier to writing those initial tests is particularly important.

PHPUnit errs on the side of isolation over quick execution. Isolated tests are valuable because they provide high-quality feedback. You do not get a report with a bunch of test failures, which were really caused because one test at the beginning of the suite failed and left the world messed up for the rest of the tests. This orientation towards isolated tests encourages designs with a large number of simple objects. Each object can be tested quickly in isolation. The result is better designs *and* faster tests.

PHPUnit assumes that most tests succeed and it is not worth reporting the details of successful tests. When a test fails, that fact is worth noting and reporting. The vast majority of tests should succeed and are not worth commenting on except to count the number of tests that run. This is an assumption that is really built into the reporting classes, and not into the core of PHPUnit. When the results of a test run are reported, you see how many tests were executed, but you only see details for those that failed.

Tests are expected to be fine-grained, testing one aspect of one object. Hence, the first time a test fails, execution of the test halts, and PHPUnit reports the failure. It is an art to test by running in many small tests. Fine-grained tests improve the overall design of the system.

When you test an object with PHPUnit, you do so only through the object's public interface. Testing based only on publicly visible behaviour encourages you to confront and solve difficult design problems earlier, before the results of poor design can infect large parts of the system.

Chapter 3. Installing PHPUnit

Requirements

PHPUnit 3.7 requires PHP 5.3.3; using the latest version of PHP is highly recommended.

PHPUnit requires the ctype [http://php.net/manual/en/ctype.installation.php], dom [http://php.net/manual/en/dom.setup.php], json [http://php.net/manual/en/json.installation.php], pcre [http://php.net/manual/en/pcre.installation.php], reflection [http://php.net/manual/en/reflection.installation.php], and spl [http://php.net/manual/en/spl.installation.php] extensions. These extensions are usually compiled and enabled by default. Some of them cannot even be disabled and are therefore always available.

For code coverage support, Xdebug [http://xdebug.org/] 2.1.3 is required; using the latest version of Xdebug is highly recommended. The tokenizer [http://php.net/manual/en/tokenizer.installation.php] extension is also required for the code coverage functionality to work.

The phar [http://php.net/manual/en/phar.installation.php] extension is required for using PHPUnit from a PHP Archive (PHAR). You need to configure `suhosin.executor.include.whitelist = phar` if you are using the Suhosin [http://suhosin.org/] extension and would like to use PHPUnit from a PHP Archive (PHAR).

PHP Archive (PHAR)

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

```
wget https://phar.phpunit.de/phpunit.phar
chmod +x phpunit.phar
mv phpunit.phar /usr/local/bin/phpunit
```

You can also immediately use the PHAR after you have downloaded it, of course:

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

Verifying PHPUnit PHAR Releases

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

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

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

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

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

We don't have the release manager's public key (`6372C20A`) in our local system. In order to proceed with the verification we need to retrieve the release manager's public key from a key server. One such

server is `pgp.uni-mainz.de`. The public key servers are linked together, so you should be able to connect to any key server.

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

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

```
gpg phpunit.phar.asc
gpg: Signature made Sat 19 Jul 2014 01:28:02 PM CEST using RSA key ID 6372C20A
gpg: Good signature from "Sebastian Bergmann <sb@sebastian-bergmann.de>"
gpg:                 aka "Sebastian Bergmann <sebastian@php.net>"
gpg:                 aka "Sebastian Bergmann <sebastian@thephp.cc>"
gpg:                 aka "Sebastian Bergmann <sebastian@phpunit.de>"
gpg:                 aka "Sebastian Bergmann <sebastian.bergmann@thephp.cc>"
gpg:                 aka "[jpeg image of size 40635]"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: D840 6D0D 8294 7747 2937  7831 4AA3 9408 6372 C20A
```

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

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

Composer

Simply add a dependency on `phpunit/phpunit` to your project's `composer.json` file if you use Composer [http://getcomposer.org/] to manage the dependencies of your project. Here is a minimal example of a `composer.json` file that just defines a development-time dependency on PHPUnit 3.7:

```
{
    "require-dev": {
        "phpunit/phpunit": "3.7.*"
    }
}
```

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

```
composer global require "phpunit/phpunit=3.7.*"
```

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

Note

Support for Composer and PHP Archive (PHAR) was added in PHPUnit 3.7 (and is known to be stable since PHPUnit 3.7.5). Earlier releases of PHPUnit are not available through these distribution channels.

Optional packages

The following optional packages are available:

`PHP_Invoker`

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

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

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

`DbUnit`

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

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

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

`PHPUnit_Selenium`

Selenium RC integration for PHPUnit.

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

```
"phpunit/phpunit-selenium": ">=1.2"
```

Upgrading

This section serves as a collection of minor BC issues that one might run into when upgrading from PHPUnit 3.6 to PHPUnit 3.7.

The upgrade should be rather easy and work without any issues as it was tested against all major Open Source frameworks and there was not a single problem for them. Still every project is different and if you did not get around to trying one of the release candidates and have ran into an issue this document might provide some help.

Removed deprecated OutputTestCase

The class `PHPUnit_Extensions_OutputTestCase` has been removed. PHPUnit 3.6 issued a deprecation notice when it was used. To see how output can now be tested look into the section called “Testing Output”.

Current working directory will be restored after each test case

If a test changes the current working directory (`cwd`) PHPUnit ran into issues when generating code coverage output. Now that the cwd is restored after each test case you might find that one of your tests depended on another test changing the cwd. Something that isn't desirable anyways and should be easy to fix.

Test listeners trigger one autoload call

When using custom test listeners as described in the section called “Test Listeners” PHPUnit silently ignored missing test listeners and it was quite hard to debug that issues for the user. Now one autoload call will be triggered trying to locate the class. If your autoloader produces an error when it doesn't find a test listener you might run into an issue here. Removing the

listener or making sure it's loaded in your `bootstrap.php` will solve this.

Parameters for mock objects do not get cloned anymore

Previously all object parameters where cloned when mocking. This lead to issues when testing trying to check whether the same object was passed to method or not and other problem with uncloneable objects. As a long standing feature request by many this behavior was changed. Example 10.15, “Testing that a method gets called once and with the identical object as was passed” shows where the new implementation could be useful. Example 10.16, “Create a mock object with cloning parameters enabled” shows how to switch back to previous behavior.

`addUncoveredFilesFromWhitelist` was removed in favor of `processUncoveredFilesFromWhitelist`

When generating code coverage and using `<whitelist addUncoveredFilesFromWhitelist="true">` all uncovered files got included by PHPUnit. This was an issue for people with executable code in those files. PHPUnit will now scan the file and guess what code is executable and what code is not without including it. This might lead to different code coverage reports.

To switch back to the old behavior the setting `<whitelist processUncoveredFilesFromWhitelist="true">` can be used. If you want the behavior with PHPUnit 3.6. and 3.7. it is possible to use both settings for a while.

Default value of `cacheTokens` changed to `false`

Since PHPUnit `3.7.2` we turned off the caching of tokenized files by default. When processing code coverage reports for big projects this cache consumed a lot of memory and due to the change in whitelist behavior it was problematic for folks with code bases with more than a couple of thousand classes.

If your project is smaller or you have enough memory you will get a runtime benefit by adding `cacheTokens="true"` to your `phpunit.xml` file. See the section called “PHPUnit”.

Chapter 4. Writing Tests for PHPUnit

Example 4.1, “Testing array operations with PHPUnit” shows how we can write tests using PHPUnit that exercise PHP's array operations. The example introduces the basic conventions and steps for writing tests with PHPUnit:

1. The tests for a class `Class` go into a class `ClassTest`.

2. `ClassTest` inherits (most of the time) from `PHPUnit_Framework_TestCase`.

3.

4. Inside the test methods, assertion methods such as `assertEquals()` (see the section called “Assertions”) are used to assert that an actual value matches an expected value.

Example 4.1. Testing array operations with PHPUnit

```
<?php
class StackTest extends PHPUnit_Framework_TestCase
{
    public function testPushAndPop()
    {
        $stack = array();
        $this->assertEquals(0, count($stack));

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

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

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

Test Dependencies

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

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

- A producer is a test method that yields its unit under test as return value.

- A consumer is a test method that depends on one or more producers and their return values.

Example 4.2, “Using the `@depends` annotation to express dependencies” shows how to use the `@depends` annotation to express dependencies between test methods.

Example 4.2. Using the @depends annotation to express dependencies

```
<?php
class StackTest extends PHPUnit_Framework_TestCase
{
    public function testEmpty()
    {
        $stack = array();
        $this->assertEmpty($stack);

        return $stack;
    }

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

        return $stack;
    }

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

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

To quickly localize defects, we want our attention to be focussed on relevant failing tests. This is why PHPUnit skips the execution of a test when a depended-upon test has failed. This improves defect localization by exploiting the dependencies between tests as shown in Example 4.3, “Exploiting the dependencies between tests”.

Example 4.3. Exploiting the dependencies between tests

```
<?php
class DependencyFailureTest extends PHPUnit_Framework_TestCase
{
    public function testOne()
    {
        $this->assertTrue(FALSE);
    }

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

```
}
?>
```

```
phpunit --verbose DependencyFailureTest
PHPUnit 3.7.0 by Sebastian Bergmann.

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 4.4, “Test with multiple dependencies”

Example 4.4. Test with multiple dependencies

```
<?php
class MultipleDependenciesTest extends PHPUnit_Framework_TestCase
{
    public function testProducerFirst()
    {
        $this->assertTrue(true);
        return 'first';
    }

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

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

```
?>
```

```
phpunit --verbose MultipleDependenciesTest
PHPUnit 3.7.0 by Sebastian Bergmann.

...

Time: 0 seconds, Memory: 3.25Mb

OK (3 tests, 3 assertions)
```

Data Providers

A test method can accept arbitrary arguments. These arguments are to be provided by a data provider method (`provider()` in Example 4.5, “Using a data provider that returns an array of arrays”). The data provider method to be used is specified using the `@dataProvider` annotation.

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

Example 4.5. Using a data provider that returns an array of arrays

```
<?php
class DataTest extends PHPUnit_Framework_TestCase
{
    /**
     * @dataProvider provider
     */
    public function testAdd($a, $b, $c)
    {
        $this->assertEquals($c, $a + $b);
    }

    public function provider()
    {
        return array(
          array(0, 0, 0),
          array(0, 1, 1),
          array(1, 0, 1),
          array(1, 1, 3)
        );
    }
}
?>
```

```
phpunit DataTest
PHPUnit 3.7.0 by Sebastian Bergmann.

...F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

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

/home/sb/DataTest.php:9

FAILURES!
```

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

Example 4.6. Using a data provider that returns an Iterator object

```
<?php
require 'CsvFileIterator.php';

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

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

```
phpunit DataTest
PHPUnit 3.7.0 by Sebastian Bergmann.

...F

Time: 0 seconds, Memory: 5.75Mb

There was 1 failure:

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

/home/sb/DataTest.php:11

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

Example 4.7. The CsvFileIterator class

```
<?php
class CsvFileIterator implements Iterator {
    protected $file;
    protected $key = 0;
    protected $current;

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

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

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

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

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

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

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

When a test receives input from both a `@dataProvider` method and from one or more tests it `@depends` on, the arguments from the data provider will come before the ones from depended-upon tests. The arguments from depended-upon tests will be the same for each data set. See Example 4.8, “Combination of @depends and @dataProvider in same test”

Example 4.8. Combination of @depends and @dataProvider in same test

```
<?php
class DependencyAndDataProviderComboTest extends PHPUnit_Framework_TestCase
{
    public function provider()
    {
        return array(array('provider1'), array('provider2'));
    }

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

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

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

```
phpunit --verbose DependencyAndDataProviderComboTest
```

```
PHPUnit 3.7.0 by Sebastian Bergmann.

...F

Time: 0 seconds, Memory: 3.50Mb

There was 1 failure:

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

/home/sb/DependencyAndDataProviderComboTest.php:31

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

Note

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

Note

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

Testing Exceptions

Example 4.9, “Using the @expectedException annotation” shows how to use the `@expectedException` annotation to test whether an exception is thrown inside the tested code.

Example 4.9. Using the @expectedException annotation

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

```
phpunit ExceptionTest
PHPUnit 3.7.0 by Sebastian Bergmann.
```

```
F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ExceptionTest::testException
Expected exception InvalidArgumentException


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

Additionally, you can use `@expectedExceptionMessage` and `@expectedExceptionCode` in combination with `@expectedException` to test the exception message and exception code as shown in Example 4.10, "Using the @expectedExceptionMessage and @expectedExceptionCode annotations".

Example 4.10. Using the @expectedExceptionMessage and @expectedExceptionCode annotations

```
<?php
class ExceptionTest extends PHPUnit_Framework_TestCase
{
    /**
     * @expectedException        InvalidArgumentException
     * @expectedExceptionMessage Right Message
     */
    public function testExceptionHasRightMessage()
    {
        throw new InvalidArgumentException('Some Message', 10);
    }

    /**
     * @expectedException     InvalidArgumentException
     * @expectedExceptionCode 20
     */
    public function testExceptionHasRightCode()
    {
        throw new InvalidArgumentException('Some Message', 10);
    }
}
?>
```

```
phpunit ExceptionTest
PHPUnit 3.7.0 by Sebastian Bergmann.

FF

Time: 0 seconds, Memory: 3.00Mb

There were 2 failures:

1) ExceptionTest::testExceptionHasRightMessage
Failed asserting that exception message 'Some Message' contains 'Right Message'.


2) ExceptionTest::testExceptionHasRightCode
Failed asserting that expected exception code 20 is equal to 10.
```

```
FAILURES!
Tests: 2, Assertions: 4, Failures: 2.
```

More examples of `@expectedExceptionMessage` and `@expectedExceptionCode` are shown in the section called “@expectedExceptionMessage” and the section called “@expectedExceptionCode” respectively.

Alternatively, you can use the `setExpectedException()` method to set the expected exception as shown in Example 4.11, “Expecting an exception to be raised by the tested code”.

Example 4.11. Expecting an exception to be raised by the tested code

```
<?php
class ExceptionTest extends PHPUnit_Framework_TestCase
{
    public function testException()
    {
        $this->setExpectedException('InvalidArgumentException');
    }

    public function testExceptionHasRightMessage()
    {
        $this->setExpectedException(
          'InvalidArgumentException', 'Right Message'
        );
        throw new InvalidArgumentException('Some Message', 10);
    }

    public function testExceptionHasRightCode()
    {
        $this->setExpectedException(
          'InvalidArgumentException', 'Right Message', 20
        );
        throw new InvalidArgumentException('The Right Message', 10);
    }
}?>
```

```
phpunit ExceptionTest
PHPUnit 3.7.0 by Sebastian Bergmann.

FFF

Time: 0 seconds, Memory: 3.00Mb

There were 3 failures:

1) ExceptionTest::testException
Expected exception InvalidArgumentException


2) ExceptionTest::testExceptionHasRightMessage
Failed asserting that exception message 'Some Message' contains 'Right Message'.


3) ExceptionTest::testExceptionHasRightCode
Failed asserting that expected exception code 20 is equal to 10.


FAILURES!
Tests: 3, Assertions: 6, Failures: 3.
```

Table 4.1, “Methods for testing exceptions” shows the methods provided for testing exceptions.

Table 4.1. Methods for testing exceptions

Method	Meaning
`void setExpectedException(string $exceptionName[, string $exceptionMessage = '', integer $exceptionCode = NULL])`	Set the expected `$exceptionName`, `$exceptionMessage`, and `$exceptionCode`.
`String getExpectedException()`	Return the name of the expected exception.

You can also use the approach shown in Example 4.12, “Alternative approach to testing exceptions” to test exceptions.

Example 4.12. Alternative approach to testing exceptions

```
<?php
class ExceptionTest extends PHPUnit_Framework_TestCase {
    public function testException() {
        try {
            // ... Code that is expected to raise an exception ...
        }

        catch (InvalidArgumentException $expected) {
            return;
        }

        $this->fail('An expected exception has not been raised.');
    }
}
?>
```

If the code that is expected to raise an exception in Example 4.12, “Alternative approach to testing exceptions” does not raise the expected exception, the subsequent call to `fail()` will halt the test and signal a problem with the test. If the expected exception is raised, the `catch` block will be executed, and the test will end successfully.

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 4.13, “Expecting a PHP error using @expectedException”.

Note

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

Example 4.13. Expecting a PHP error using @expectedException

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

```
    }
}
?>
```

```
phpunit -d error_reporting=2 ExpectedErrorTest
PHPUnit 3.7.0 by Sebastian Bergmann.

.

Time: 0 seconds, Memory: 5.25Mb

OK (1 test, 1 assertion)
```

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

Note

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

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

Example 4.14. Testing return values of code that uses PHP Errors

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

?>
```

```
phpunit ErrorSuppressionTest
PHPUnit 3.7.0 by Sebastian Bergmann.

.

Time: 1 seconds, Memory: 5.25Mb

OK (1 test, 1 assertion)
```

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

Testing Output

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

Example 4.15, “Testing the output of a function or method” shows how to use the `expectOutputString()` method to set the expected output. If this expected output is not generated, the test will be counted as a failure.

Example 4.15. Testing the output of a function or method

```
<?php
class OutputTest extends PHPUnit_Framework_TestCase
{
    public function testExpectFooActualFoo()
    {
        $this->expectOutputString('foo');
        print 'foo';
    }

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

```
phpunit OutputTest
PHPUnit 3.7.0 by Sebastian Bergmann.

.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 4.2, “Methods for testing output” shows the methods provided for testing output

Table 4.2. 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`.

Method	Meaning
`bool setOutputCallback(callable $callback)`	Sets up a callback that is used to, for instance, normalize the actual output.

Note

A test that emits output will fail in strict mode.

Assertions

This section lists the various assertion methods that are available.

assertArrayHasKey()

`assertArrayHasKey(mixed $key, array $array[, string $message = ''])`

Reports an error identified by `$message` if `$array` does not have the `$key`.

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

Example 4.16. Usage of assertArrayHasKey()

```
<?php
class ArrayHasKeyTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertArrayHasKey('foo', array('bar' => 'baz'));
    }
}
?>
```

```
phpunit ArrayHasKeyTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ArrayHasKeyTest::testFailure
Failed asserting that an array has the key 'foo'.

/home/sb/ArrayHasKeyTest.php:6

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

assertClassHasAttribute()

`assertClassHasAttribute(string $attributeName, string $className[, string $message = ''])`

Reports an error identified by `$message` if `$className::attributeName` does not exist.

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

Example 4.17. Usage of assertClassHasAttribute()

```
<?php
class ClassHasAttributeTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertClassHasAttribute('foo', 'stdClass');
    }
}
?>
```

```
phpunit ClassHasAttributeTest
PHPUnit 3.7.0 by Sebastian Bergmann.

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.
```

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 4.18. Usage of assertClassHasStaticAttribute()

```
<?php
class ClassHasStaticAttributeTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertClassHasStaticAttribute('foo', 'stdClass');
    }
}
?>
```

```
phpunit ClassHasStaticAttributeTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:
```

```
1) ClassHasStaticAttributeTest::testFailure
Failed asserting that class "stdClass" has static attribute "foo".

/home/sb/ClassHasStaticAttributeTest.php:6

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

assertContains()

`assertContains(mixed $needle, Iterator|array $haystack[, string $message = ''])`

Reports an error identified by `$message` if `$needle` is not an element of `$haystack`.

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

`assertAttributeContains()` and `assertAttributeNotContains()` are convenience wrappers that use a `public`, `protected`, or `private` attribute of a class or object as the haystack.

Example 4.19. Usage of assertContains()

```php
<?php
class ContainsTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertContains(4, array(1, 2, 3));
    }
}
?>
```

```
phpunit ContainsTest
PHPUnit 3.7.0 by Sebastian Bergmann.

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 4.20. Usage of assertContains()

```php
<?php
```

```
class ContainsTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertContains('baz', 'foobar');
    }
}
?>
```

```
phpunit ContainsTest
PHPUnit 3.7.0 by Sebastian Bergmann.

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 4.21. Usage of assertContains() with $ignoreCase

```
<?php
class ContainsTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertContains('foo', 'FooBar');
    }

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

```
phpunit ContainsTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F.

Time: 0 seconds, Memory: 2.75Mb

There was 1 failure:

1) ContainsTest::testFailure
Failed asserting that 'FooBar' contains "foo".

/home/sb/ContainsTest.php:6

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

assertContainsOnly()

```
assertContainsOnly(string $type, Iterator|array $haystack[, boolean
$isNativeType = NULL, string $message = ''])
```

Reports an error identified by `$message` if `$haystack` does not contain only variables of type `$type`.

`$isNativeType` is a flag used to indicate whether `$type` is a native PHP type or not.

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

`assertAttributeContainsOnly()` and `assertAttributeNotContainsOnly()` are convenience wrappers that use a `public`, `protected`, or `private` attribute of a class or object as the haystack.

Example 4.22. Usage of assertContainsOnly()

```php
<?php
class ContainsOnlyTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertContainsOnly('string', array('1', '2', 3));
    }
}
?>
```

```
phpunit ContainsOnlyTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ContainsOnlyTest::testFailure
Failed asserting that Array (
    0 => '1'
    1 => '2'
    2 => 3
) contains only values of type "string".

/home/sb/ContainsOnlyTest.php:6

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

assertContainsOnlyInstancesOf()

```
assertContainsOnlyInstancesOf(string $classname, Traversable|array
$haystack[, string $message = ''])
```

Reports an error identified by `$message` if `$haystack` does not contain only instances of class `$classname`.

Example 4.23. Usage of assertContainsOnlyInstancesOf()

```php
<?php
```

```
class ContainsOnlyInstancesOfTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertContainsOnlyInstancesOf('Foo', array(new Foo(), new Bar(), new Foo(
    }
}
?>
```

```
phpunit ContainsOnlyInstancesOfTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) ContainsOnlyInstancesOfTest::testFailure
Failed asserting that Array ([0]=> Bar Object(...)) is an instance of class "Foo".

/home/sb/ContainsOnlyInstancesOfTest.php:6

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

assertCount()

`assertCount($expectedCount, $haystack[, string $message = ''])`

Reports an error identified by `$message` if the number of elements in `$haystack` is not `$expectedCount`.

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

Example 4.24. Usage of assertCount()

```
<?php
class CountTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertCount(0, array('foo'));
    }
}
?>
```

```
phpunit CountTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) CountTest::testFailure
Failed asserting that actual size 1 matches expected size 0.

/home/sb/CountTest.php:6
```

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

assertEmpty()

`assertEmpty(mixed $actual[, string $message = ''])`

Reports an error identified by `$message` if `$actual` is not empty.

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

`assertAttributeEmpty()` and `assertAttributeNotEmpty()` are convenience wrappers that can be applied to a `public`, `protected`, or `private` attribute of a class or object.

Example 4.25. Usage of assertEmpty()

```
<?php
class EmptyTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertEmpty(array('foo'));
    }
}
?>
```

```
phpunit EmptyTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) EmptyTest::testFailure
Failed asserting that an array is empty.

/home/sb/EmptyTest.php:6

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

assertEqualXMLStructure()

`assertEqualXMLStructure(DOMElement $expectedElement, DOMElement $actualElement[, boolean $checkAttributes = FALSE, string $message = ''])`

Reports an error identified by `$message` if the XML Structure of the DOMElement in `$actualElement` is not equal to the XML structure of the DOMElement in `$expectedElement`.

Example 4.26. Usage of assertEqualXMLStructure()

```
<?php
class EqualXMLStructureTest extends PHPUnit_Framework_TestCase
{
    public function testFailureWithDifferentNodeNames()
    {
```

```
        $expected = new DOMElement('foo');
        $actual = new DOMElement('bar');

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

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

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

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

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

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

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

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

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

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

```
phpunit EqualXMLStructureTest
PHPUnit 3.7.0 by Sebastian Bergmann.

FFFF

Time: 0 seconds, Memory: 5.75Mb

There were 4 failures:

1) EqualXMLStructureTest::testFailureWithDifferentNodeNames
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'foo'
+'bar'
```

```
/home/sb/EqualXMLStructureTest.php:9

2) EqualXMLStructureTest::testFailureWithDifferentNodeAttributes
Number of attributes on node "foo" does not match
Failed asserting that 0 matches expected 1.

/home/sb/EqualXMLStructureTest.php:22

3) EqualXMLStructureTest::testFailureWithDifferentChildrenCount
Number of child nodes of "foo" differs
Failed asserting that 1 matches expected 3.

/home/sb/EqualXMLStructureTest.php:35

4) EqualXMLStructureTest::testFailureWithDifferentChildren
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'bar'
+'baz'

/home/sb/EqualXMLStructureTest.php:48

FAILURES!
Tests: 4, Assertions: 8, Failures: 4.
```

assertEquals()

`assertEquals(mixed $expected, mixed $actual[, string $message = ''])`

Reports an error identified by `$message` if the two variables `$expected` and `$actual` are not equal.

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

`assertAttributeEquals()` and `assertAttributeNotEquals()` are convenience wrappers that use a `public`, `protected`, or `private` attribute of a class or object as the actual value.

Example 4.27. Usage of assertEquals()

```
<?php
class EqualsTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertEquals(1, 0);
    }

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

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

```
phpunit EqualsTest
```

```
PHPUnit 3.7.0 by Sebastian Bergmann.

FFF

Time: 0 seconds, Memory: 5.25Mb

There were 3 failures:

1) EqualsTest::testFailure
Failed asserting that 0 matches expected 1.

/home/sb/EqualsTest.php:6

2) EqualsTest::testFailure2
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'bar'
+'baz'

/home/sb/EqualsTest.php:11

3) EqualsTest::testFailure3
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
 'foo
-bar
+bah
 baz
 '

/home/sb/EqualsTest.php:16

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

More specialized comparisons are used for specific argument types for `$expected` and `$actual`, see below.

`assertEquals(float $expected, float $actual[, string $message = '', float $delta = 0])`

Reports an error identified by `$message` if the two floats `$expected` and `$actual` are not within `$delta` of each other.

Please read "What Every Computer Scientist Should Know About Floating-Point Arithmetic [http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html]" to understand why `$delta` is necessary.

Example 4.28. Usage of assertEquals() with floats

```php
<?php
class EqualsTest extends PHPUnit_Framework_TestCase
{
    public function testSuccess()
    {
        $this->assertEquals(1.0, 1.1, '', 0.2);
    }
```

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

```
phpunit EqualsTest
PHPUnit 3.7.0 by Sebastian Bergmann.

.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 4.29. Usage of assertEquals() with DOMDocument objects

```
<?php
class EqualsTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $expected = new DOMDocument;
        $expected->loadXML('<foo><bar/></foo>');

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

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

```
phpunit EqualsTest
PHPUnit 3.7.0 by Sebastian Bergmann.

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 4.30. Usage of assertEquals() with objects

```
<?php
class EqualsTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $expected = new stdClass;
        $expected->foo = 'foo';
        $expected->bar = 'bar';

        $actual = new stdClass;
        $actual->foo = 'bar';
        $actual->baz = 'bar';

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

```
phpunit EqualsTest
PHPUnit 3.7.0 by Sebastian Bergmann.

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 4.31. Usage of assertEquals() with arrays

```
<?php
class EqualsTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertEquals(array('a', 'b', 'c'), array('a', 'c', 'd'));
    }
}
?>
```

```
phpunit EqualsTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) EqualsTest::testFailure
Failed asserting that two arrays are equal.
--- Expected
+++ Actual
@@ @@
 Array (
     0 => 'a'
-    1 => 'b'
-    2 => 'c'
+    1 => 'c'
+    2 => 'd'
 )

/home/sb/EqualsTest.php:6

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

assertFalse()

`assertFalse(bool $condition[, string $message = ''])`

Reports an error identified by `$message` if `$condition` is `TRUE`.

Example 4.32. Usage of assertFalse()

```
<?php
class FalseTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertFalse(TRUE);
    }
}
```

```
?>
```

```
phpunit FalseTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) FalseTest::testFailure
Failed asserting that true is false.

/home/sb/FalseTest.php:6

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

assertFileEquals()

`assertFileEquals(string $expected, string $actual[, string $message = ''])`

Reports an error identified by `$message` if the file specified by `$expected` does not have the same contents as the file specified by `$actual`.

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

Example 4.33. Usage of assertFileEquals()

```
<?php
class FileEqualsTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertFileEquals('/home/sb/expected', '/home/sb/actual');
    }
}
?>
```

```
phpunit FileEqualsTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) FileEqualsTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'expected
+'actual
 '

/home/sb/FileEqualsTest.php:6

FAILURES!
```

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

assertFileExists()

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

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

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

Example 4.34. Usage of assertFileExists()

```
<?php
class FileExistsTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertFileExists('/path/to/file');
    }
}
?>
```

```
phpunit FileExistsTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) FileExistsTest::testFailure
Failed asserting that file "/path/to/file" exists.

/home/sb/FileExistsTest.php:6

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

assertGreaterThan()

`assertGreaterThan(mixed $expected, mixed $actual[, string $message = ''])`

Reports an error identified by `$message` if the value of `$actual` is not greater than the value of `$expected`.

`assertAttributeGreaterThan()` is a convenience wrapper that uses a `public`, `protected`, or `private` attribute of a class or object as the actual value.

Example 4.35. Usage of assertGreaterThan()

```
<?php
class GreaterThanTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertGreaterThan(2, 1);
    }
}
```

```
?>
```

```
phpunit GreaterThanTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) GreaterThanTest::testFailure
Failed asserting that 1 is greater than 2.

/home/sb/GreaterThanTest.php:6

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

assertGreaterThanOrEqual()

`assertGreaterThanOrEqual(mixed $expected, mixed $actual[, string $message = ''])`

Reports an error identified by `$message` if the value of `$actual` is not greater than or equal to the value of `$expected`.

`assertAttributeGreaterThanOrEqual()` is a convenience wrapper that uses a `public`, `protected`, or `private` attribute of a class or object as the actual value.

Example 4.36. Usage of assertGreaterThanOrEqual()

```
<?php
class GreatThanOrEqualTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertGreaterThanOrEqual(2, 1);
    }
}
?>
```

```
phpunit GreaterThanOrEqualTest
PHPUnit 3.7.0 by Sebastian Bergmann.

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.
```

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 4.37. Usage of assertInstanceOf()

```
<?php
class InstanceOfTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertInstanceOf('RuntimeException', new Exception);
    }
}
?>
```

```
phpunit InstanceOfTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) InstanceOfTest::testFailure
Failed asserting that Exception Object (...) is an instance of class "RuntimeException".

/home/sb/InstanceOfTest.php:6

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

assertInternalType()

```
assertInternalType($expected, $actual[, $message = ''])
```

Reports an error identified by `$message` if `$actual` is not of the `$expected` type.

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

`assertAttributeInternalType()` and `assertAttributeNotInternalType()` are convenience wrappers that can be applied to a `public`, `protected`, or `private` attribute of a class or object.

Example 4.38. Usage of assertInternalType()

```
<?php
class InternalTypeTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertInternalType('string', 42);
```

```
    }
}
?>
```

```
phpunit InternalTypeTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) InternalTypeTest::testFailure
Failed asserting that 42 is of type "string".

/home/sb/InternalTypeTest.php:6

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

assertJsonFileEqualsJsonFile()

`assertJsonFileEqualsJsonFile(mixed $expectedFile, mixed $actualFile[, string $message = ''])`

Reports an error identified by `$message` if the value of `$actualFile` does not match the value of `$expectedFile`.

Example 4.39. Usage of assertJsonFileEqualsJsonFile()

```
<?php
class JsonFileEqualsJsonFileTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertJsonFileEqualsJsonFile(
          'path/to/fixture/file', 'path/to/actual/file');
    }
}
?>
```

```
phpunit JsonFileEqualsJsonFileTest
PHPUnit 3.7.0 by Sebastian Bergmann.

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", "

/home/sb/JsonFileEqualsJsonFileTest.php:5

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

assertJsonStringEqualsJsonFile()

```
assertJsonStringEqualsJsonFile(mixed $expectedFile, mixed $actualJ-
son[, string $message = ''])
```

Reports an error identified by `$message` if the value of `$actualJson` does not match the value of `$expectedFile`.

Example 4.40. Usage of assertJsonStringEqualsJsonFile()

```
<?php
class JsonStringEqualsJsonFileTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertJsonStringEqualsJsonFile(
          'path/to/fixture/file', json_encode(array("Mascot" => "ux"))
        );
    }
}
?>
```

```
phpunit JsonStringEqualsJsonFileTest
PHPUnit 3.7.0 by Sebastian Bergmann.

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.
```

assertJsonStringEqualsJsonString()

```
assertJsonStringEqualsJsonString(mixed $expectedJson, mixed $actu-
alJson[, string $message = ''])
```

Reports an error identified by `$message` if the value of `$actualJson` does not match the value of `$expectedJson`.

Example 4.41. Usage of assertJsonStringEqualsJsonString()

```
<?php
class JsonStringEqualsJsonStringTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertJsonStringEqualsJsonString(
          json_encode(array("Mascot" => "Tux")), json_encode(array("Mascott" => "ux"))
        );
    }
}
?>
```

```
phpunit JsonStringEqualsJsonStringTest
```

```
PHPUnit 3.7.0 by Sebastian Bergmann.

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.
```

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 4.42. Usage of assertLessThan()

```php
<?php
class LessThanTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertLessThan(1, 2);
    }
}
?>
```

```
phpunit LessThanTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) LessThanTest::testFailure
Failed asserting that 2 is less than 1.

/home/sb/LessThanTest.php:6

FAILURES!
```

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

assertLessThanOrEqual()

`assertLessThanOrEqual(mixed $expected, mixed $actual[, string $message = ''])`

Reports an error identified by `$message` if the value of `$actual` is not less than or equal to the value of `$expected`.

`assertAttributeLessThanOrEqual()` is a convenience wrapper that uses a `public`, `protected`, or `private` attribute of a class or object as the actual value.

Example 4.43. Usage of assertLessThanOrEqual()

```
<?php
class LessThanOrEqualTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertLessThanOrEqual(1, 2);
    }
}
?>
```

```
phpunit LessThanOrEqualTest
PHPUnit 3.7.0 by Sebastian Bergmann.

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.
```

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 4.44. Usage of assertNull()

```
<?php
class NullTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertNull('foo');
    }
```

```
}
?>
```

```
phpunit NotNullTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) NullTest::testFailure
Failed asserting that 'foo' is null.

/home/sb/NotNullTest.php:6

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

assertObjectHasAttribute()

`assertObjectHasAttribute(string $attributeName, object $object[, string $message = ''])`

Reports an error identified by `$message` if `$object->attributeName` does not exist.

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

Example 4.45. Usage of assertObjectHasAttribute()

```
<?php
class ObjectHasAttributeTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertObjectHasAttribute('foo', new stdClass);
    }
}
?>
```

```
phpunit ObjectHasAttributeTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 4.75Mb

There was 1 failure:

1) ObjectHasAttributeTest::testFailure
Failed asserting that object of class "stdClass" has attribute "foo".

/home/sb/ObjectHasAttributeTest.php:6

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

assertRegExp()

```
assertRegExp(string $pattern, string $string[, string $message = ''])
```

Reports an error identified by `$message` if `$string` does not match the regular expression `$pattern`.

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

Example 4.46. Usage of assertRegExp()

```
<?php
class RegExpTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertRegExp('/foo/', 'bar');
    }
}
?>
```

```
phpunit RegExpTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) RegExpTest::testFailure
Failed asserting that 'bar' matches PCRE pattern "/foo/".

/home/sb/RegExpTest.php:6

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

assertStringMatchesFormat()

```
assertStringMatchesFormat(string $format, string $string[, string $message = ''])
```

Reports an error identified by `$message` if the `$string` does not match the `$format` string.

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

Example 4.47. Usage of assertStringMatchesFormat()

```
<?php
class StringMatchesFormatTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertStringMatchesFormat('%i', 'foo');
    }
}
?>
```

```
phpunit StringMatchesFormatTest
```

```
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) StringMatchesFormatTest::testFailure
Failed asserting that 'foo' matches PCRE pattern "/^[+-]?\d+$/s".

/home/sb/StringMatchesFormatTest.php:6

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

The format string may contain the following placeholders:

- `%e`: Represents a directory separator, for example `/` on Linux.
- `%s`: One or more of anything (character or white space) except the end of line character.
- `%S`: Zero or more of anything (character or white space) except the end of line character.
- `%a`: One or more of anything (character or white space) including the end of line character.
- `%A`: Zero or more of anything (character or white space) including the end of line character.
- `%w`: Zero or more white space characters.
- `%i`: A signed integer value, for example `+3142`, `-3142`.
- `%d`: An unsigned integer value, for example `123456`.
- `%x`: One or more hexadecimal character. That is, characters in the range `0-9`, `a-f`, `A-F`.
- `%f`: A floating point number, for example: `3.142`, `-3.142`, `3.142E-10`, `3.142e+10`.
- `%c`: A single character of any sort.

assertStringMatchesFormatFile()

```
assertStringMatchesFormatFile(string $formatFile, string $string[,
string $message = ''])
```

Reports an error identified by `$message` if the `$string` does not match the contents of the `$formatFile`.

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

Example 4.48. Usage of assertStringMatchesFormatFile()

```
<?php
class StringMatchesFormatFileTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertStringMatchesFormatFile('/path/to/expected.txt', 'foo');
    }
```

```
}
?>
```

```
phpunit StringMatchesFormatFileTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) StringMatchesFormatFileTest::testFailure
Failed asserting that 'foo' matches PCRE pattern "/^[+-]?\d+
$/s".

/home/sb/StringMatchesFormatFileTest.php:6

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

assertSame()

`assertSame(mixed $expected, mixed $actual[, string $message = ''])`

Reports an error identified by `$message` if the two variables `$expected` and `$actual` do not have the same type and value.

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

`assertAttributeSame()` and `assertAttributeNotSame()` are convenience wrappers that use a `public`, `protected`, or `private` attribute of a class or object as the actual value.

Example 4.49. Usage of assertSame()

```
<?php
class SameTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertSame('2204', 2204);
    }
}
?>
```

```
phpunit SameTest
PHPUnit 3.7.0 by Sebastian Bergmann.

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 4.50. Usage of assertSame() with objects

```
<?php
class SameTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertSame(new stdClass, new stdClass);
    }
}
?>
```

```
phpunit SameTest
PHPUnit 3.7.0 by Sebastian Bergmann.

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.
```

assertSelectCount()

```
assertSelectCount(array $selector, integer $count, mixed $actual[,
string $message = '', boolean $isHtml = TRUE])
```

Reports an error identified by `$message` if the CSS selector `$selector` does not match `$count` elements in the DOMNode `$actual`.

`$count` can be one of the following types:

- `boolean`: Asserts for presence of elements matching the selector (`TRUE`) or absence of elements (`FALSE`).
- `integer`: Asserts the count of elements.
- `array`: Asserts that the count is in a range specified by using `<`, `>`, `<=`, and `>=` as keys.

Example 4.51. Usage of assertSelectCount()

```
<?php
class SelectCountTest extends PHPUnit_Framework_TestCase
{
    protected function setUp()
    {
        $this->xml = new DomDocument;
        $this->xml->loadXML('<foo><bar/><bar/><bar/></foo>');
```

```
    }

    public function testAbsenceFailure()
    {
        $this->assertSelectCount('foo bar', FALSE, $this->xml);
    }

    public function testPresenceFailure()
    {
        $this->assertSelectCount('foo baz', TRUE, $this->xml);
    }

    public function testExactCountFailure()
    {
        $this->assertSelectCount('foo bar', 5, $this->xml);
    }

    public function testRangeFailure()
    {
        $this->assertSelectCount('foo bar', array('>'=>6, '<'=>8), $this->xml);
    }
}
?>
```

```
phpunit SelectCountTest
PHPUnit 3.7.0 by Sebastian Bergmann.

FFFF

Time: 0 seconds, Memory: 5.50Mb

There were 4 failures:

1) SelectCountTest::testAbsenceFailure
Failed asserting that true is false.

/home/sb/SelectCountTest.php:12

2) SelectCountTest::testPresenceFailure
Failed asserting that false is true.

/home/sb/SelectCountTest.php:17

3) SelectCountTest::testExactCountFailure
Failed asserting that 3 matches expected 5.

/home/sb/SelectCountTest.php:22

4) SelectCountTest::testRangeFailure
Failed asserting that false is true.

/home/sb/SelectCountTest.php:27

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

assertSelectEquals()

`assertSelectEquals(array $selector, string $content, integer $count, mixed $actual[, string $message = '', boolean $isHtml = TRUE])`

Reports an error identified by `$message` if the CSS selector `$selector` does not match `$count` elements in the DOMNode `$actual` with the value `$content`.

`$count` can be one of the following types:

- `boolean`: Asserts for presence of elements matching the selector (`TRUE`) or absence of elements (`FALSE`).
- `integer`: Asserts the count of elements.
- `array`: Asserts that the count is in a range specified by using `<`, `>`, `<=`, and `>=` as keys.

Example 4.52. Usage of assertSelectEquals()

```
<?php
class SelectEqualsTest extends PHPUnit_Framework_TestCase
{
    protected function setUp()
    {
        $this->xml = new DomDocument;
        $this->xml->loadXML('<foo><bar>Baz</bar><bar>Baz</bar></foo>');
    }

    public function testAbsenceFailure()
    {
        $this->assertSelectEquals('foo bar', 'Baz', FALSE, $this->xml);
    }

    public function testPresenceFailure()
    {
        $this->assertSelectEquals('foo bar', 'Bat', TRUE, $this->xml);
    }

    public function testExactCountFailure()
    {
        $this->assertSelectEquals('foo bar', 'Baz', 5, $this->xml);
    }

    public function testRangeFailure()
    {
        $this->assertSelectEquals('foo bar', 'Baz', array('>'=>6, '<'=>8), $this->xml);
    }
}
?>
```

```
phpunit SelectEqualsTest
PHPUnit 3.7.0 by Sebastian Bergmann.

FFFF

Time: 0 seconds, Memory: 5.50Mb

There were 4 failures:

1) SelectEqualsTest::testAbsenceFailure
Failed asserting that true is false.

/home/sb/SelectEqualsTest.php:12

2) SelectEqualsTest::testPresenceFailure
Failed asserting that false is true.

/home/sb/SelectEqualsTest.php:17

3) SelectEqualsTest::testExactCountFailure
Failed asserting that 2 matches expected 5.
```

```
/home/sb/SelectEqualsTest.php:22

4) SelectEqualsTest::testRangeFailure
Failed asserting that false is true.

/home/sb/SelectEqualsTest.php:27

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

assertSelectRegExp()

`assertSelectRegExp(array $selector, string $pattern, integer $count, mixed $actual[, string $message = '', boolean $isHtml = TRUE])`

Reports an error identified by `$message` if the CSS selector `$selector` does not match `$count` elements in the DOMNode `$actual` with a value that matches `$pattern`.

`$count` can be one of the following types:

- `boolean`: Asserts for presence of elements matching the selector (`TRUE`) or absence of elements (`FALSE`).
- `integer`: Asserts the count of elements.
- `array`: Asserts that the count is in a range specified by using `<`, `>`, `<=`, and `>=` as keys.

Example 4.53. Usage of assertSelectRegExp()

```
<?php
class SelectRegExpTest extends PHPUnit_Framework_TestCase
{
    protected function setUp()
    {
        $this->xml = new DomDocument;
        $this->xml->loadXML('<foo><bar>Baz</bar><bar>Baz</bar></foo>');
    }

    public function testAbsenceFailure()
    {
        $this->assertSelectRegExp('foo bar', '/Ba.*/', FALSE, $this->xml);
    }

    public function testPresenceFailure()
    {
        $this->assertSelectRegExp('foo bar', '/B[oe]z]/', TRUE, $this->xml);
    }

    public function testExactCountFailure()
    {
        $this->assertSelectRegExp('foo bar', '/Ba.*/', 5, $this->xml);
    }

    public function testRangeFailure()
    {
        $this->assertSelectRegExp('foo bar', '/Ba.*/', array('>'=>6, '<'=>8), $this->xml
    }
}
?>
```

```
phpunit SelectRegExpTest
```

```
PHPUnit 3.7.0 by Sebastian Bergmann.

FFFF

Time: 0 seconds, Memory: 5.50Mb

There were 4 failures:

1) SelectRegExpTest::testAbsenceFailure
Failed asserting that true is false.

/home/sb/SelectRegExpTest.php:12

2) SelectRegExpTest::testPresenceFailure
Failed asserting that false is true.

/home/sb/SelectRegExpTest.php:17

3) SelectRegExpTest::testExactCountFailure
Failed asserting that 2 matches expected 5.

/home/sb/SelectRegExpTest.php:22

4) SelectRegExpTest::testRangeFailure
Failed asserting that false is true.

/home/sb/SelectRegExpTest.php:27

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

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 4.54. Usage of assertStringEndsWith()

```php
<?php
class StringEndsWithTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertStringEndsWith('suffix', 'foo');
    }
}
?>
```

```
phpunit StringEndsWithTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 1 second, Memory: 5.00Mb

There was 1 failure:

1) StringEndsWithTest::testFailure
```

```
Failed asserting that 'foo' ends with "suffix".

/home/sb/StringEndsWithTest.php:6

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

assertStringEqualsFile()

`assertStringEqualsFile(string $expectedFile, string $actualString[, string $message = ''])`

Reports an error identified by `$message` if the file specified by `$expectedFile` does not have `$actualString` as its contents.

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

Example 4.55. Usage of assertStringEqualsFile()

```
<?php
class StringEqualsFileTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertStringEqualsFile('/home/sb/expected', 'actual');
    }
}
?>
```

```
phpunit StringEqualsFileTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) StringEqualsFileTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'expected
-'
+'actual'

/home/sb/StringEqualsFileTest.php:6

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

assertStringStartsWith()

`assertStringStartsWith(string $prefix, string $string[, string $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 4.56. Usage of assertStringStartsWith()

```
<?php
class StringStartsWithTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertStringStartsWith('prefix', 'foo');
    }
}
?>
```

```
phpunit StringStartsWithTest
PHPUnit 3.7.0 by Sebastian Bergmann.

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.
```

assertTag()

`assertTag(array $matcher, string $actual[, string $message = '', boolean $isHtml = TRUE])`

Reports an error identified by `$message` if `$actual` is not matched by the `$matcher`.

`$matcher` is an associative array that specifies the match criteria for the assertion:

- `id`: The node with the given `id` attribute must match the corresponsing value.
- `tag`: The node type must match the corresponding value.
- `attributes`: The node's attributes must match the corresponsing values in the `$attributes` associative array.
- `content`: The text content must match the given value.
- `parent`: The node's parent must match the `$parent` associative array.
- `child`: At least one of the node's immediate children must meet the criteria described by the `$child` associative array.
- `ancestor`: At least one of the node's ancestors must meet the criteria described by the `$ancestor` associative array.
- `descendant`: At least one of the node's descendants must meet the criteria described by the `$descendant` associative array.
- `children`: Associative array for counting children of a node.
  - `count`: The number of matching children must be equal to this number.

- `less_than`: The number of matching children must be less than this number.
- `greater_than`: The number of matching children must be greater than this number.
- `only`: Another associative array consisting of the keys to use to match on the children, and only matching children will be counted.

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

Example 4.57. Usage of assertTag()

```
<?php
// Matcher that asserts that there is an element with an id="my_id".
$matcher = array('id' => 'my_id');

// Matcher that asserts that there is a "span" tag.
$matcher = array('tag' => 'span');

// Matcher that asserts that there is a "span" tag with the content
// "Hello World".
$matcher = array('tag' => 'span', 'content' => 'Hello World');

// Matcher that asserts that there is a "span" tag with content matching the
// regular expression pattern.
$matcher = array('tag' => 'span', 'content' => 'regexp:/Try P(HP|ython)/');

// Matcher that asserts that there is a "span" with an "list" class attribute.
$matcher = array(
  'tag'        => 'span',
  'attributes' => array('class' => 'list')
);

// Matcher that asserts that there is a "span" inside of a "div".
$matcher = array(
  'tag'    => 'span',
  'parent' => array('tag' => 'div')
);

// Matcher that asserts that there is a "span" somewhere inside a "table".
$matcher = array(
  'tag'      => 'span',
  'ancestor' => array('tag' => 'table')
);

// Matcher that asserts that there is a "span" with at least one "em" child.
$matcher = array(
  'tag'   => 'span',
  'child' => array('tag' => 'em')
);

// Matcher that asserts that there is a "span" containing a (possibly nested)
// "strong" tag.
$matcher = array(
  'tag'        => 'span',
  'descendant' => array('tag' => 'strong')
);

// Matcher that asserts that there is a "span" containing 5-10 "em" tags as
// immediate children.
$matcher = array(
  'tag'      => 'span',
  'children' => array(
    'less_than'    => 11,
```

```
    'greater_than' => 4,
    'only'         => array('tag' => 'em')
  )
);

// Matcher that asserts that there is a "div", with an "ul" ancestor and a "li"
// parent (with class="enum"), and containing a "span" descendant that contains
// an element with id="my_test" and the text "Hello World".
$matcher = array(
  'tag'        => 'div',
  'ancestor'   => array('tag' => 'ul'),
  'parent'     => array(
    'tag'        => 'li',
    'attributes' => array('class' => 'enum')
  ),
  'descendant' => array(
    'tag'   => 'span',
    'child' => array(
      'id'      => 'my_test',
      'content' => 'Hello World'
    )
  )
);

// Use assertTag() to apply a $matcher to a piece of $html.
$this->assertTag($matcher, $html);

// Use assertTag() to apply a $matcher to a piece of $xml.
$this->assertTag($matcher, $xml, '', FALSE);
?>
```

assertThat()

More complex assertions can be formulated using the `PHPUnit_Framework_Constraint` classes. They can be evaluated using the `assertThat()` method. Example 4.58, "Usage of assertThat()" shows how the `logicalNot()` and `equalTo()` constraints can be used to express the same assertion as `assertNotEquals()`.

```
assertThat(mixed $value, PHPUnit_Framework_Constraint $constraint[,
$message = ''])
```

Reports an error identified by `$message` if the `$value` does not match the `$constraint`.

Example 4.58. Usage of assertThat()

```
<?php
class BiscuitTest extends PHPUnit_Framework_TestCase
{
    public function testEquals()
    {
        $theBiscuit = new Biscuit('Ginger');
        $myBiscuit  = new Biscuit('Ginger');

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

Table 4.3, “Constraints” shows the available `PHPUnit_Framework_Constraint` classes.

Table 4.3. Constraints

Constraint	Meaning
`PHPUnit_ Framework_ Constraint_ Attribute attribute(PHPUnit_ Framework_ Constraint $constraint, $attributeName)`	Constraint that applies another constraint to an attribute of a class or an object.
`PHPUnit_ Framework_ Constraint_ IsAnything anything()`	Constraint that accepts any input value.
`PHPUnit_ Framework_ Constraint_ ArrayHasKey arrayHasKey(mixed $key)`	Constraint that asserts that the array it is evaluated for has a given key.
`PHPUnit_ Framework_ Constraint_ TraversableContains contains(mixed $value)`	Constraint that asserts that the `array` or object that implements the `Iterator` interface it is evaluated for contains a given value.
`PHPUnit_ Framework_ Constraint_ TraversableContainsOnly containsOnly(string $type)`	Constraint that asserts that the `array` or object that implements the `Iterator` interface it is evaluated for contains only values of a given type.
`PHPUnit_ Framework_ Constraint_ TraversableContainsOnly containsOnlyInstancesOf(string $classname)`	Constraint that asserts that the `array` or object that implements the `Iterator` interface it is evaluated for contains only instances of a given classname.
`PHPUnit_ Framework_ Constraint_ IsEqual equalTo($value, $delta = 0, $maxDepth = 10)`	Constraint that checks if one value is equal to another.
`PHPUnit_ Framework_ Constraint_ Attribute attributeEqualTo($attributeName, $value, $delta = 0, $maxDepth = 10)`	Constraint that checks if a value is equal to an attribute of a class or of an object.
`PHPUnit_ Framework_ Constraint_ FileExists fileExists()`	Constraint that checks if the file(name) that it is evaluated for exists.
`PHPUnit_ Framework_ Constraint_ GreaterThan greaterThan(mixed $value)`	Constraint that asserts that the value it is evaluated for is greater than a given value.
`PHPUnit_ Framework_ Constraint_ Or greaterThanOrEqual(mixed $value)`	Constraint that asserts that the value it is evaluated for is greater than or equal to a given value.
`PHPUnit_ Framework_ Constraint_ ClassHasAttribute classHasAttribute(string $attributeName)`	Constraint that asserts that the class it is evaluated for has a given attribute.
`PHPUnit_ Framework_ Constraint_ ClassHasStaticAttribute classHasStaticAttribute(string $attributeName)`	Constraint that asserts that the class it is evaluated for has a given static attribute.
`PHPUnit_ Framework_ Constraint_ ObjectHasAttribute hasAttribute(string $attributeName)`	Constraint that asserts that the object it is evaluated for has a given attribute.

Constraint	Meaning
`PHPUnit_ Framework_ Constraint_ IsIdentical identicalTo(mixed $value)`	Constraint that asserts that one value is identical to another.
`PHPUnit_ Framework_ Constraint_ IsFalse isFalse()`	Constraint that asserts that the value it is evaluated is `FALSE`.
`PHPUnit_ Framework_ Constraint_ IsInstanceOf isInstanceOf(string $className)`	Constraint that asserts that the object it is evaluated for is an instance of a given class.
`PHPUnit_ Framework_ Constraint_ IsNull isNull()`	Constraint that asserts that the value it is evaluated is `NULL`.
`PHPUnit_ Framework_ Constraint_ IsTrue isTrue()`	Constraint that asserts that the value it is evaluated is `TRUE`.
`PHPUnit_ Framework_ Constraint_ IsType isType(string $type)`	Constraint that asserts that the value it is evaluated for is of a specified type.
`PHPUnit_ Framework_ Constraint_ LessThan lessThan(mixed $value)`	Constraint that asserts that the value it is evaluated for is smaller than a given value.
`PHPUnit_ Framework_ Constraint_ Or lessThanOrEqual(mixed $value)`	Constraint that asserts that the value it is evaluated for is smaller than or equal to a given value.
`logicalAnd()`	Logical AND.
`logicalNot(PHPUnit_ Framework_ Constraint $constraint)`	Logical NOT.
`logicalOr()`	Logical OR.
`logicalXor()`	Logical XOR.
`PHPUnit_ Framework_ Constraint_ PCREMatch matchesRegularExpression(string $pattern)`	Constraint that asserts that the string it is evaluated for matches a regular expression.
`PHPUnit_ Framework_ Constraint_ StringContains stringContains(string $string, bool $case)`	Constraint that asserts that the string it is evaluated for contains a given string.
`PHPUnit_ Framework_ Constraint_ StringEndsWith stringEndsWith(string $suffix)`	Constraint that asserts that the string it is evaluated for ends with a given suffix.
`PHPUnit_ Framework_ Constraint_ StringStartsWith stringStartsWith(string $prefix)`	Constraint that asserts that the string it is evaluated for starts with a given prefix.

assertTrue()

```
assertTrue(bool $condition[, string $message = ''])
```

Reports an error identified by `$message` if `$condition` is `FALSE`.

Example 4.59. Usage of assertTrue()

```
<?php
class TrueTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
```

```
        $this->assertTrue(FALSE);
    }
}
?>
```

```
phpunit TrueTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.00Mb

There was 1 failure:

1) TrueTest::testFailure
Failed asserting that false is true.

/home/sb/TrueTest.php:6

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

assertXmlFileEqualsXmlFile()

`assertXmlFileEqualsXmlFile(string $expectedFile, string $actualFile[, string $message = ''])`

Reports an error identified by `$message` if the XML document in `$actualFile` is not equal to the XML document in `$expectedFile`.

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

Example 4.60. Usage of assertXmlFileEqualsXmlFile()

```
<?php
class XmlFileEqualsXmlFileTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertXmlFileEqualsXmlFile(
          '/home/sb/expected.xml', '/home/sb/actual.xml');
    }
}
?>
```

```
phpunit XmlFileEqualsXmlFileTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) XmlFileEqualsXmlFileTest::testFailure
Failed asserting that two DOM documents are equal.
--- Expected
+++ Actual
@@ @@
 <?xml version="1.0"?>
```

```
 <foo>
-  <bar/>
+  <baz/>
 </foo>

/home/sb/XmlFileEqualsXmlFileTest.php:7

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

assertXmlStringEqualsXmlFile()

`assertXmlStringEqualsXmlFile(string $expectedFile, string $actualXml[, string $message = ''])`

Reports an error identified by `$message` if the XML document in `$actualXml` is not equal to the XML document in `$expectedFile`.

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

Example 4.61. Usage of assertXmlStringEqualsXmlFile()

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

```
phpunit XmlStringEqualsXmlFileTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

1) XmlStringEqualsXmlFileTest::testFailure
Failed asserting that two DOM documents are equal.
--- Expected
+++ Actual
@@ @@
 <?xml version="1.0"?>
 <foo>
-  <bar/>
+  <baz/>
 </foo>

/home/sb/XmlStringEqualsXmlFileTest.php:7

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

assertXmlStringEqualsXmlString()

`assertXmlStringEqualsXmlString(string $expectedXml, string $actualXml[, string $message = ''])`

Reports an error identified by `$message` if the XML document in `$actualXml` is not equal to the XML document in `$expectedXml`.

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

Example 4.62. Usage of assertXmlStringEqualsXmlString()

```
<?php
class XmlStringEqualsXmlStringTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertXmlStringEqualsXmlString(
          '<foo><bar/></foo>', '<foo><baz/></foo>');
    }
}
?>
```

```
phpunit XmlStringEqualsXmlStringTest
PHPUnit 3.7.0 by Sebastian Bergmann.

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.
```

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 4.63. Error output generated when an array comparison fails

```
<?php
class ArrayDiffTest extends PHPUnit_Framework_TestCase
{
    public function testEquality() {
        $this->assertEquals(
            array(1,2,3 ,4,5,6),
```

```
            array(1,2,33,4,5,6)
        );
    }
}
?>
```

```
phpunit ArrayDiffTest
PHPUnit 3.6.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

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

/home/sb/ArrayDiffTest.php:7

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

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

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

Example 4.64. Error output when an array comparison of an long array fails

```
<?php
class LongArrayDiffTest extends PHPUnit_Framework_TestCase
{
    public function testEquality() {
        $this->assertEquals(
            array(0,0,0,0,0,0,0,0,0,0,0,1,2,3 ,4,5,6),
            array(0,0,0,0,0,0,0,0,0,0,0,0,1,2,33,4,5,6)
        );
    }
}
?>
```

```
phpunit LongArrayDiffTest
PHPUnit 3.6.0 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 5.25Mb
```

```
There was 1 failure:

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


/home/sb/LongArrayDiffTest.php:7

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

Edge cases

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

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

Example 4.65. Edge case in the diff generation when using weak comparison

```
<?php
class ArrayWeakComparisonTest extends PHPUnit_Framework_TestCase
{
    public function testEquality() {
        $this->assertEquals(
            array(1  ,2,3 ,4,5,6),
            array('1',2,33,4,5,6)
        );
    }
}
?>
```

```
phpunit ArrayWeakComparisonTest
PHPUnit 3.6.0 by Sebastian Bergmann.

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 so assertEquals considers the values as a match.

Chapter 5. 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 3.7.0 by Sebastian Bergmann.

..

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.

`S` Printed when the test has been skipped (see Chapter 9, *Incomplete and Skipped Tests*).

`I` Printed when the test is marked as being incomplete or not yet implemented (see Chapter 9, *Incomplete and Skipped Tests*).

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

Command-Line Options

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

```
phpunit --help
PHPUnit 3.7.0 by Sebastian Bergmann.

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

  --log-junit <file>        Log test execution in JUnit XML format to file.
  --log-tap <file>          Log test execution in TAP format to file.
  --log-json <file>         Log test execution in JSON format.

  --coverage-clover <file>  Generate code coverage report in Clover XML format.
  --coverage-html <dir>     Generate code coverage report in HTML format.
  --coverage-php <file>     Serialize PHP_CodeCoverage object to file.
  --coverage-text=<file>    Generate code coverage report in text format.
                            Default to writing to the standard output.
```

```
  --testdox-html <file>     Write agile documentation in HTML format to file.
  --testdox-text <file>     Write agile documentation in Text format to file.

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

  --loader <loader>         TestSuiteLoader implementation to use.
  --printer <printer>       TestSuiteListener implementation to use.
  --repeat <times>          Runs the test(s) repeatedly.

  --tap                     Report test execution progress in TAP format.
  --testdox                 Report test execution progress in TestDox format.

  --colors                  Use colors in 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-skipped         Stop execution upon first skipped test.
  --stop-on-incomplete      Stop execution upon first incomplete test.
  --strict                  Run tests in strict mode.
  -v|--verbose              Output more verbose information.
  --debug                   Display debugging information during test execution.

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

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

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

`phpunit UnitTest`
: Runs the tests that are provided by the class `UnitTest`. This class is expected to be declared in the `UnitTest.php` sourcefile.

  `UnitTest` must be either a class that inherits from `PHPUnit_Framework_TestCase` or a class that provides a `public static suite()` method which returns a `PHPUnit_Framework_Test` object, for example an instance of the `PHPUnit_Framework_TestSuite` class.

`phpunit UnitTest UnitTest.php`
: Runs the tests that are provided by the class `UnitTest`. This class is expected to be declared in the specified sourcefile.

`--log-junit`
: Generates a logfile in JUnit XML format for the tests run. See Chapter 18, *Logging* for more details.

`--log-tap`
: Generates a logfile using the Test Anything Protocol (TAP) [http://testanything.org/] format for the tests run. See Chapter 18, *Logging* for more details.

`--log-json`
: Generates a logfile using the JSON [http://www.json.org/] format. See Chapter 18, *Logging* for more details.

`--coverage-html`
: Generates a code coverage report in HTML format. See Chapter 14, *Code Coverage Analysis* for more details.

  Please note that this functionality is only available when the tokenizer and Xdebug extensions are installed.

`--coverage-clover`
: Generates a logfile in XML format with the code coverage information for the tests run. See Chapter 18, *Logging* for more details.

  Please note that this functionality is only available when the tokenizer and Xdebug extensions are installed.

`--coverage-php`
: Generates a serialized PHP_CodeCoverage object with the code coverage information.

  Please note that this functionality is only available when the tokenizer and Xdebug extensions are installed.

`--coverage-text`
: Generates a logfile or command-line output in human readable format with the code coverage information for the tests run. See Chapter 18, *Logging* for more details.

  Please note that this functionality is only available when the tokenizer and Xdebug extensions are installed.

`--testdox-html` and `--testdox-text`
: Generates agile documentation in HTML or plain text format for the tests that are run. See Chapter 15, *Other Uses for Tests* for more details.

`--filter`
: Only runs tests whose name matches the given regular expression pattern. If the pattern is not enclosed in delimiters, PHPUnit will enclose the pattern in `/` delimiters.

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

  `TestNamespace\TestCaseClass::testMethod`
  : The default test name format is the equivalent of using the `__METHOD__` magic constant inside the test method.

  `TestNamespace\TestCaseClass::testMethod with data set #0`
  : When a test has a data provider, each iteration of the data gets the current index appended to the end of the default test name.

  `TestNamespace\TestCaseClass::testMethod with data set "my named data"`
  : When a test has a data provider that uses named sets, each iteration of the data gets the current name appended to the end of the default test name. See Example 5.1, “Named data sets” for an example of named data sets.

Example 5.1 Named data sets

```
<?php
namespace TestNamespace;

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

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

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

See Example 5.2, “Filter pattern examples” for examples of valid filter patterns.

Example 5.2. Filter pattern examples

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

`--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).
`--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`.
`--printer`	Specifies the result printer to use. The printer class must extend `PHPUnit_Util_Printer` and implement the `PHPUnit_Framework_TestListener` interface.
`--repeat`	Repeatedly runs the test(s) the specified number of times.
`--tap`	Reports the test progress using the Test Anything Protocol (TAP) [http://testanything.org/]. See Chapter 18, *Logging* for more details.
`--testdox`	Reports the test progress as agile documentation. See Chapter 15, *Other Uses for Tests* for more details.
`--colors`	Use colors in output. On Windows, use ANSICON [https://github.com/adoxa/ansicon] or ConEmu [https://github.com/Maximus5/ConEmu].
`--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-skipped`	Stop execution upon first skipped test.
`--stop-on-incomplete`	Stop execution upon first incomplete test.

`--strict`	Run tests in strict mode. When strict mode is enabled:
	
	• Tests that emit output will fail.
	
	• Tests with no assertions will fail and will not generate code coverage information.
	
	• If the `PHP_Invoker` package is installed:
	
	  • A test marked as `@large` will fail if it takes longer than 60 seconds to execute. This timeout is configurable via the `timeoutForLargeTests` attribute in the XML configuration file.
	
	  • A test marked as `@medium` will fail if it takes longer than 10 seconds to execute. This timeout is configurable via the `timeoutForMediumTests` attribute in the XML configuration file.

- A test not marked as `@medium` or `@large` will fail if it takes longer than 1 second to execute. This timeout is configurable via the `timeoutForSmallTests` attribute in the XML configuration file.

`--verbose`	Output more verbose information, for instance the names of tests that were incomplete or have been skipped.
`--process-isolation`	Run each test in a separate PHP process.
`--no-globals-backup`	Do not backup and restore $GLOBALS. See the section called “Global State” for more details.
`--static-backup`	Backup and restore static attributes of user-defined classes. See the section called “Global State” for more details.
`--bootstrap`	A "bootstrap" PHP file that is run before the tests.
`--configuration`, `-c`	Read configuration from XML file. See Appendix C, *The XML Configuration File* for more details. If `phpunit.xml` or `phpunit.xml.dist` (in that order) exist in the current working directory and `--configuration` is *not* used, the configuration will be automatically read from that file.
`--no-configuration`	Ignore `phpunit.xml` and `phpunit.xml.dist` from the current working directory.
`--include-path`	Prepend PHP's `include_path` with given path(s).
`-d`	Sets the value of the given PHP configuration option.
`--debug`	Output debug information such as the name of a test when its execution starts.

Chapter 6. Fixtures

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

In Example 4.1, “Testing array operations with PHPUnit”, the fixture was simply the array that is stored in the `$stack` variable. Most of the time, though, the fixture will be more complex than a simple array, and the amount of code needed to set it up will grow accordingly. The actual content of the test gets lost in the noise of setting up the fixture. This problem gets even worse when you write several tests with similar fixtures. Without some help from the testing framework, we would have to duplicate the code that sets up the fixture for each test we write.

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

In Example 4.2, “Using the `@depends` annotation to express dependencies” we used the producer-consumer relationship between tests to share a fixture. This is not always desired or even possible. Example 6.1, “Using setUp() to create the stack fixture” shows how we can write the tests of the `StackTest` in such a way that not the fixture itself is reused but the code that creates it. First we declare the instance variable, `$stack`, that we are going to use instead of a method-local variable. Then we put the creation of the `array` fixture into the `setUp()` method. Finally, we remove the redundant code from the test methods and use the newly introduced instance variable, `$this->stack`, instead of the method-local variable `$stack` with the `assertEquals()` assertion method.

Example 6.1. Using setUp() to create the stack fixture

```
<?php
class StackTest extends PHPUnit_Framework_TestCase
{
    protected $stack;

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

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

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

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

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

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

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

Example 6.2. Example showing all template methods available

```
<?php
class TemplateMethodsTest extends PHPUnit_Framework_TestCase
{
    public static function setUpBeforeClass()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

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

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

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

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

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

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

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

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

```
phpunit TemplateMethodsTest
PHPUnit 3.7.0 by Sebastian Bergmann.

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


Time: 0 seconds, Memory: 5.25Mb

There was 1 failure:

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

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

More setUp() than tearDown()

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

Variations

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

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

Sharing Fixture

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

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

Example 6.3, “Sharing fixture between the tests of a test suite” uses the `setUpBeforeClass()` and `tearDownAfterClass()` template methods to connect to the database before the test case class' first test and to disconnect from the database after the last test of the test case, respectively.

Example 6.3. Sharing fixture between the tests of a test suite

```
<?php
class DatabaseTest extends PHPUnit_Framework_TestCase
{
    protected static $dbh;

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

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

It cannot be emphasized enough that sharing fixtures between tests reduces the value of the tests. The underlying design problem is that objects are not loosely coupled. You will achieve better results solving the underlying design problem and then writing tests using stubs (see Chapter 10, *Test Doubles*), than by creating dependencies between tests at runtime and ignoring the opportunity to improve your design.

Global State

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

In PHP, global variables work like this:

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

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

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

Note

The implementation of the backup and restore operations for global variables and static attributes of classes uses `serialize()` and `unserialize()`.

Objects of some classes that are provided by PHP itself, such as `PDO` for example, cannot be serialized and the backup operation will break when such an object is stored in the `$GLOBALS` array, for instance.

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

```
class MyTest extends PHPUnit_Framework_TestCase
{
    protected $backupGlobalsBlacklist = array('globalVariable');

    // ...
}
```

Note

Please note that setting the `$backupGlobalsBlacklist` attribute inside the `setUp()` method, for instance, has no effect.

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

```
class MyTest extends PHPUnit_Framework_TestCase
{
    protected $backupStaticAttributesBlacklist = array(
      'className' => array('attributeName')
    );

    // ...
}
```

Note

Please note that setting the `$backupStaticAttributesBlacklist` attribute inside the `setUp()` method, for instance, has no effect.

Chapter 7. Organizing Tests

One of the goals of PHPUnit (see Chapter 2, *PHPUnit's Goals*) is that tests should be composable: we want to be able to run any number or combination of tests together, for instance all tests for the whole project, or the tests for all classes of a component that is part of the project, or just the tests for a single class.

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

Composing a Test Suite Using the Filesystem

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

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

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

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

```
phpunit tests
PHPUnit 3.7.0 by Sebastian Bergmann.

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

Time: 0 seconds

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 tests/CurrencyTest
PHPUnit 3.7.0 by Sebastian Bergmann.

........

Time: 0 seconds

OK (8 tests, 8 assertions)
```

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

```
phpunit --filter testObjectCanBeConstructedForValidConstructorArgument tests
```

```
PHPUnit 3.7.0 by Sebastian Bergmann.

..

Time: 0 seconds

OK (2 test, 2 assertions)
```

Note

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

Composing a Test Suite Using XML Configuration

PHPUnit's XML configuration file (Appendix C, *The XML Configuration File*) can also be used to compose a test suite. Example 7.1, “Composing a Test Suite Using XML Configuration” shows a minimal example that will add all `*Test` classes that are found in `*Test.php` files when the `tests` directory is recursively traversed.

Example 7.1. Composing a Test Suite Using XML Configuration

```
<phpunit>
  <testsuites>
    <testsuite name="money">
      <directory>tests</directory>
    </testsuite>
  </testsuites>
</phpunit>
```

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

Example 7.2. Composing a Test Suite Using XML Configuration

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

Chapter 8. Database Testing

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

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

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

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

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

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

Supported Vendors for Database Testing

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

Difficulties in Database Testing

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

- The database schema and tables

- Inserting the rows required for the test into these tables

- Verifying the state of the database after your test has run

- Cleanup the database for each new test

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

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

- You do not want to modify considerable amount of test code for little changes in your production code.

- You want to be able to read and understand the test code easily, even months after writing it.

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

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

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

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

The four stages of a database test

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

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

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

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

1. Clean-Up Database

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

2. Set up fixture

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

3–5. Run Test, Verify outcome and Teardown

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

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

Configuration of a PHPUnit Database Test-Case

Usually when using PHPUnit your testcases would extend the `PHPUnit_Framework_TestCase` class in the following way:

```
<?php
```

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

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

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

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

Implementing getConnection()

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

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

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

Implementing getDataSet()

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

For the implementation we only need to know that the `getDataSet()` method is called once during `setUp()` to retrieve the fixture data-set and insert it into the database. In the example we are using

a factory method `createFlatXMLDataSet($filename)` that represents a data-set through an XML representation.

What about the Database Schema (DDL)?

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

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

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

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

Tip: Use your own Abstract Database TestCase

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

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

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

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

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

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

```
<?xml version="1.0" encoding="UTF-8" ?>
<phpunit>
```

```
    <php>
        <var name="DB_DSN" value="mysql:dbname=myguestbook;host=localhost" />
        <var name="DB_USER" value="user" />
        <var name="DB_PASSWD" value="passwd" />
        <var name="DB_DBNAME" value="myguestbook" />
    </php>
</phpunit>
```

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

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

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

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

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

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

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

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

Understanding DataSets and DataTables

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

This abstraction is necessary to compare the actual contents of a database against the expected contents. Expectations can be represented as XML, YAML, CSV files or PHP array for example. The DataSet and DataTable interfaces enable the comparison of these conceptually different sources, emulating relational database storage in a semantically similar approach.

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

- Specify one or more tables in your database by table name (actual dataset)
- Specify the expected dataset in your preferred format (YAML, XML, ..)

- Assert that both dataset representations equal each other.

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

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

Available Implementations

There are three different types of datasets/datatables:

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

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

Flat XML DataSet

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

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

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

However this simplicity comes at a cost.

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

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

The handling of NULL values with the flat xml dataset is tedious. A NULL value is different than an empty string value in almost any database (Oracle being an exception), something that is difficult to describe in the flat xml format. You can represent a NULL's value by omitting the attribute from the row specification. If our guestbook would allow anonymous entries represented by a NULL value in the user column, a hypothetical state of the guestbook table could look like:

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

```
    <guestbook id="2" content="I like it!" created="2010-04-26 12:14:20" />
</dataset>
```

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

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

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

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

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

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

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

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

XML DataSet

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

```
<?xml version="1.0" ?>
<dataset>
    <table name="guestbook">
        <column>id</column>
        <column>content</column>
        <column>user</column>
        <column>created</column>
        <row>
            <value>1</value>
            <value>Hello buddy!</value>
            <value>joe</value>
            <value>2010-04-24 17:15:23</value>
        </row>
        <row>
            <value>2</value>
```

```
            <value>I like it!</value>
            <null />
            <value>2010-04-26 12:14:20</value>
        </row>
    </table>
</dataset>
```

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

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

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

MySQL XML DataSet

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

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

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

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

YAML DataSet

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

```
guestbook:
  -
    id: 1
    content: "Hello buddy!"
    user: "joe"
    created: 2010-04-24 17:15:23
  -
    id: 2
```

```
    content: "I like it!"
    user:
    created: 2010-04-26 12:14:20
```

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

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

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

CSV DataSet

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

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

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

You can create a CSV DataSet by calling:

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

Array DataSet

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

```
<?php
class ArrayGuestbookTest extends PHPUnit_Extensions_Database_TestCase
{
    protected function getDataSet()
    {
```

```
        return new MyApp_DbUnit_ArrayDataSet(array(
            'guestbook' => array(
                array('id' => 1, 'content' => 'Hello buddy!', 'user' => 'joe', 'created'
                array('id' => 2, 'content' => 'I like it!',   'user' => null,  'created'
            ),
        ));
    }
}
?>
```

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

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

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

The implementation for this Array DataSet is simple and straightforward:

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

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

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

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

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

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

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

```
?>
```

Query (SQL) DataSet

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

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

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

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

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

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

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

Database (DB) Dataset

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

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

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

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

    public function testFilteredGuestbook()
```

```
    {
        $tableNames = array('guestbook');
        $dataSet = $this->getConnection()->createDataSet($tableNames);
        // ...
    }
}
?>
```

Replacement DataSet

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

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

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

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

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

DataSet Filter

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

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

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

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

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

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

Composite DataSet

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

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

and *fixture2.xml*:

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

Using the Composite DataSet we can aggregate both fixture files:

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

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

        return $compositeDs;
    }
}
?>
```

Beware of Foreign Keys

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

Implementing your own DataSets/DataTables

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

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

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

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

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

A table is also represented by the following interface:

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

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

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

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

The Connection API

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

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

```
    // ...
}
?>
```

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

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

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

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

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

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

Database Assertions API

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

Asserting the Row-Count of a Table

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

```
<?php
```

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

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

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

Asserting the State of a Table

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

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

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

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

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

```
<?xml version="1.0" ?>
<dataset>
    <guestbook id="1" content="Hello buddy!" user="joe" created="2010-04-24 17:15:23" />
    <guestbook id="2" content="I like it!" user="nancy" created="2010-04-26 12:14:20" />
    <guestbook id="3" content="Hello world!" user="suzy" created="2010-05-01 21:47:08" /
</dataset>
```

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

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

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

```
</dataset>
```

We have to fix up the Query Table call:

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

Asserting the Result of a Query

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

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

Asserting the State of Multiple Tables

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

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

```
<?php
class DataSetAssertionsTest extends PHPUnit_Extensions_Database_TestCase
{
    public function testCreateDataSetAssertion()
    {
        $dataSet = $this->getConnection()->createDataSet(array('guestbook'));
        $expectedDataSet = $this->createFlatXmlDataSet('guestbook.xml');
        $this->assertDataSetsEqual($expectedDataSet, $dataSet);
    }
}
?>
```

2. You can construct the DataSet on your own:

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

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

Frequently Asked Questions

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

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

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

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

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

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

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

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

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

How to handle NULL with Flat XML / CSV Datasets?

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

Chapter 9. Incomplete and Skipped Tests

Incomplete Tests

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

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

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

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

Example 9.1, “Marking a test as incomplete” shows a test case class, `SampleTest`, that contains one test method, `testSomething()`. By calling the convenience method `markTestIncomplete()` (which automatically raises an `PHPUnit_Framework_IncompleteTestError` exception) in the test method, we mark the test as being incomplete.

Example 9.1. Marking a test as incomplete

```
<?php
class SampleTest extends PHPUnit_Framework_TestCase
{
    public function testSomething()
    {
        // Optional: Test anything here, if you want.
        $this->assertTrue(TRUE, 'This should already work.');

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

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

```
phpunit --verbose SampleTest
PHPUnit 3.7.0 by Sebastian Bergmann.

I

Time: 0 seconds, Memory: 3.75Mb
```

```
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 9.1, “API for Incomplete Tests” shows the API for marking tests as incomplete.

Table 9.1. API for Incomplete Tests

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

Skipping Tests

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

Example 9.2, “Skipping a test” shows a test case class, `DatabaseTest`, that contains one test method, `testConnection()`. In the test case class' `setUp()` template method we check whether the MySQLi extension is available and use the `markTestSkipped()` method to skip the test if it is not.

Example 9.2. Skipping a test

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

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

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

```
phpunit --verbose DatabaseTest
PHPUnit 3.7.0 by Sebastian Bergmann.

S

Time: 0 seconds, Memory: 3.75Mb
```

```
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 9.2, “API for Skipping Tests” shows the API for skipping tests.

Table 9.2. API for Skipping Tests

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

Skipping Tests using @requires

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

Table 9.3. Possible @requires usages

Type	Possible Values	Examples	Another example
`PHP`	Any PHP version identifier	@requires PHP 5.3.3	@requires PHP 5.4-dev
`PHPUnit`	Any PHPUnit version identifier	@requires PHPUnit 3.6.3	@requires PHPUnit 3.7
`function`	Any valid parameter to function_exists [http://php.net/function_exists]	@requires function imap_open	@requires function ReflectionMethod::setAccessible
`extension`	Any extension name	@requires extension mysqli	@requires extension curl

Example 9.3. Skipping test cases using @requires

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

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

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

Chapter 10. Test Doubles

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

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

The `getMock($className)` method 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 class. This test double object can be used in every context where an object of the original class is expected.

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

Limitations

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.

Warning

Please pay attention to the fact that the parameters managing has been changed. The previous implementation clones all object parameters. It did not allow to check whether the same object was passed to method or not. Example 10.15, “Testing that a method gets called once and with the identical object as was passed” shows where the new implementation could be useful. Example 10.16, “Create a mock object with cloning parameters enabled” shows how to switch back to previous behavior.

Stubs

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

Example 10.2, “Stubbing a method call to return a fixed value” shows how to stub method calls and set up return values. We first use the `getMock()` method that is provided by the `PHPUnit_Framework_TestCase` class to set up a stub object that looks like an object of `SomeClass` (Example 10.1, “The class we want to stub”). We then use the Fluent Interface [http://martinfowler.com/bliki/FluentInterface.html] that PHPUnit provides to specify the behavior for the stub. In essence, this means that you do not need to create several temporary objects and wire them together afterwards. Instead, you chain method calls as shown in the example. This leads to more readable and "fluent" code.

Example 10.1. The class we want to stub

```
<?php
```

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

Example 10.2. Stubbing a method call to return a fixed value

```
<?php
require_once 'SomeClass.php';

class StubTest extends PHPUnit_Framework_TestCase
{
    public function testStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->getMock('SomeClass');

        // Configure the stub.
        $stub->expects($this->any())
             ->method('doSomething')
             ->will($this->returnValue('foo'));

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

"Behind the scenes", PHPUnit automatically generates a new PHP class that implements the desired behavior when the `getMock()` method is used. The generated test double class can be configured through the optional arguments of the `getMock()` method.

- By default, all methods of the given class are replaced with a test double that just returns `NULL` unless a return value is configured using `will($this->returnValue())`, for instance.
- When the second (optional) parameter is provided, only the methods whose names are in the array are replaced with a configurable test double. The behavior of the other methods is not changed. Providing `NULL` as the parameter means that no methods will be replaced.
- The third (optional) parameter may hold a parameter array that is passed to the original class' constructor (which is not replaced with a dummy implementation by default).
- The fourth (optional) parameter can be used to specify a class name for the generated test double class.
- The fifth (optional) parameter can be used to disable the call to the original class' constructor.
- The sixth (optional) parameter can be used to disable the call to the original class' clone constructor.
- The seventh (optional) parameter can be used to disable `__autoload()` during the generation of the test double class.

Alternatively, the Mock Builder API can be used to configure the generated test double class. Example 10.3, “Using the Mock Builder API can be used to configure the generated test double class” shows an example. Here's a list of the methods that can be used with the Mock Builder's fluent interface:

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

Example 10.3. Using the Mock Builder API can be used to configure the generated test double class

```
<?php
require_once 'SomeClass.php';

class StubTest extends PHPUnit_Framework_TestCase
{
    public function testStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->getMockBuilder('SomeClass')
                     ->disableOriginalConstructor()
                     ->getMock();

        // Configure the stub.
        $stub->expects($this->any())
             ->method('doSomething')
             ->will($this->returnValue('foo'));

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

Sometimes you want to return one of the arguments of a method call (unchanged) as the result of a stubbed method call. Example 10.4, “Stubbing a method call to return one of the arguments” shows how you can achieve this using `returnArgument()` instead of `returnValue()`.

Example 10.4. Stubbing a method call to return one of the arguments

```
<?php
require_once 'SomeClass.php';

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

```
        $stub = $this->getMock('SomeClass');

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

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

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

When testing a fluent interface, it is sometimes useful to have a stubbed method return a reference to the stubbed object. Example 10.5, “Stubbing a method call to return a reference to the stub object” shows how you can use `returnSelf()` to achieve this.

Example 10.5. Stubbing a method call to return a reference to the stub object

```
<?php
require_once 'SomeClass.php';

class StubTest extends PHPUnit_Framework_TestCase
{
    public function testReturnSelf()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->getMock('SomeClass');

        // Configure the stub.
        $stub->expects($this->any())
             ->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 10.6, “Stubbing a method call to return the value from a map” for an example.

Example 10.6. Stubbing a method call to return the value from a map

```
<?php
require_once 'SomeClass.php';

class StubTest extends PHPUnit_Framework_TestCase
{
    public function testReturnValueMapStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->getMock('SomeClass');

        // Create a map of arguments to return values.
```

```
        $map = array(
          array('a', 'b', 'c', 'd'),
          array('e', 'f', 'g', 'h')
        );

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

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

When the stubbed method call should return a calculated value instead of a fixed one (see `returnValue()`) or an (unchanged) argument (see `returnArgument()`), you can use `returnCallback()` to have the stubbed method return the result of a callback function or method. See Example 10.7, “Stubbing a method call to return a value from a callback” for an example.

Example 10.7. Stubbing a method call to return a value from a callback

```
<?php
require_once 'SomeClass.php';

class StubTest extends PHPUnit_Framework_TestCase
{
    public function testReturnCallbackStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->getMock('SomeClass');

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

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

A simpler alternative to setting up a callback method may be to specify a list of desired return values. You can do this with the `onConsecutiveCalls()` method. See Example 10.8, “Stubbing a method call to return a list of values in the specified order” for an example.

Example 10.8. Stubbing a method call to return a list of values in the specified order

```
<?php
require_once 'SomeClass.php';

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

```
        $stub = $this->getMock('SomeClass');

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

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

Instead of returning a value, a stubbed method can also raise an exception. Example 10.9, “Stubbing a method call to throw an exception” shows how to use `throwException()` to do this.

Example 10.9. Stubbing a method call to throw an exception

```
<?php
require_once 'SomeClass.php';

class StubTest extends PHPUnit_Framework_TestCase
{
    public function testThrowExceptionStub()
    {
        // Create a stub for the SomeClass class.
        $stub = $this->getMock('SomeClass');

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

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

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

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

Mock Objects

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

You can use a *mock object* "as an observation point that is used to verify the indirect outputs of the SUT as it is exercised. Typically, the mock object also includes the functionality of a test stub in that it

must return values to the SUT if it hasn't already failed the tests but the emphasis is on the verification of the indirect outputs. Therefore, a mock object is lot more than just a test stub plus assertions; it is used a fundamentally different way".

Limitations

Only mock objects generated within the scope of a test will be verified automatically by PHPUnit. Mock objects generated in data providers, for instance, will not be verified 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 10.10, “The Subject and Observer classes that are part of the System under Test (SUT)” shows the code for the `Subject` and `Observer` classes that are part of the System under Test (SUT).

Example 10.10. The Subject and Observer classes that are part of the System under Test (SUT)

```
<?php
class Subject
{
    protected $observers = array();
    protected $name;

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

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

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

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

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

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

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

```
    // Other methods.
}

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

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

    // Other methods.
}
?>
```

Example 10.11, “Testing that a method gets called once and with a specified argument” shows how to use a mock object to test the interaction between `Subject` and `Observer` objects.

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

Example 10.11. Testing that a method gets called once and with a specified argument

```
<?php
class SubjectTest extends PHPUnit_Framework_TestCase
{
    public function testObserversAreUpdated()
    {
        // Create a mock for the Observer class,
        // only mock the update() method.
        $observer = $this->getMock('Observer', array('update'));

        // 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 10.12. Testing that a method gets called with a number of arguments constrained in different ways

```
<?php
class SubjectTest extends PHPUnit_Framework_TestCase
{
    public function testErrorReported()
    {
        // Create a mock for the Observer class, mocking the
        // reportError() method
        $observer = $this->getMock('Observer', array('reportError'));

        $observer->expects($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 `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 10.13. More complex argument verification

```
<?php
class SubjectTest extends PHPUnit_Framework_TestCase
{
    public function testErrorReported()
    {
        // Create a mock for the Observer class, mocking the
        // reportError() method
        $observer = $this->getMock('Observer', array('reportError'));

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

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

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

Table 4.3, “Constraints” shows the constraints that can be applied to method arguments and Table 10.1, “Matchers” shows the matchers that are available to specify the number of invocations.

Table 10.1. 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.

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 10.14. Testing the concrete methods of an abstract class

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

    public abstract function abstractMethod();
}

class AbstractClassTest extends PHPUnit_Framework_TestCase
{
    public function testConcreteMethod()
    {
        $stub = $this->getMockForAbstractClass('AbstractClass');
        $stub->expects($this->any())
             ->method('abstractMethod')
             ->will($this->returnValue(TRUE));

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

```
?>
```

Example 10.15. Testing that a method gets called once and with the identical object as was passed

```
<?php
class FooTest extends PHPUnit_Framework_TestCase
{
    public function testIdenticalObjectPassed()
    {
        $expectedObject = new stdClass;

        $mock = $this->getMock('stdClass', array('foo'));
        $mock->expects($this->once())
             ->method('foo')
             ->with($this->identicalTo($expectedObject));

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

Example 10.16. Create a mock object with cloning parameters enabled

```
<?php
class FooTest extends PHPUnit_Framework_TestCase
{
    public function testIdenticalObjectPassed()
    {
        $cloneArguments = true;

        $mock = $this->getMock(
            'stdClass',
            array(),
            array(),
            '',
            FALSE,
            TRUE,
            TRUE,
            $cloneArguments
        );

        // or using the mock builder
        $mock = $this->getMockBuilder('stdClass')->enableArgumentCloning()->getMock();

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

Stubbing and Mocking Web Services

When your application interacts with a web service you want to test it without actually interacting with the web service. To make the stubbing and mocking of web services easy, the `getMockFromWsdl()` can be used just like `getMock()` (see above). The only difference is that `getMockFromWsdl()` returns a stub or mock based on a web service description in WSDL and `getMock()` returns a stub or mock based on a PHP class or interface.

Example 10.17, “Stubbing a web service” shows how `getMockFromWsdl()` can be used to stub, for example, the web service described in `GoogleSearch.wsdl`.

Example 10.17. Stubbing a web service

```
<?php
class GoogleTest extends PHPUnit_Framework_TestCase
{
    public function testSearch()
    {
        $googleSearch = $this->getMockFromWsdl(
          'GoogleSearch.wsdl', 'GoogleSearch'
        );

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

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

        $result = new stdClass;
        $result->documentFiltering = FALSE;
        $result->searchComments = '';
        $result->estimatedTotalResultsCount = 378000;
        $result->estimateIsExact = FALSE;
        $result->resultElements = array($element);
        $result->searchQuery = 'PHPUnit';
        $result->startIndex = 1;
        $result->endIndex = 1;
        $result->searchTips = '';
        $result->directoryCategories = array();
        $result->searchTime = 0.248822;

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

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

```
?>
```

Mocking the Filesystem

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

Simply add a dependency on `mikey179/vfsStream` to your project's `composer.json` file if you use Composer [http://getcomposer.org/] to manage the dependencies of your project. Here is a minimal example of a `composer.json` file that just defines a development-time dependency on PHPUnit 3.7 and vfsStream:

```
{
    "require-dev": {
        "phpunit/phpunit": "3.7.*",
        "mikey179/vfsStream": "1.*"
    }
}
```

Example 10.18, “A class that interacts with the filesystem” shows a class that interacts with the filesystem.

Example 10.18. A class that interacts with the filesystem

```
<?php
class Example
{
    protected $id;
    protected $directory;

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

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

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

Without a virtual filesystem such as vfsStream we cannot test the `setDirectory()` method in isolation from external influence (see Example 10.19, “Testing a class that interacts with the filesystem”).

Example 10.19. Testing a class that interacts with the filesystem

```
<?php
require_once 'Example.php';

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

```
        }
    }

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

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

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

The approach above has several drawbacks:

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

Example 10.20, “Mocking the filesystem in a test for a class that interacts with the filesystem” shows how vfsStream can be used to mock the filesystem in a test for a class that interacts with the filesystem.

Example 10.20. Mocking the filesystem in a test for a class that interacts with the filesystem

```
<?php
require_once 'vfsStream/vfsStream.php';
require_once 'Example.php';

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

    public function testDirectoryIsCreated()
    {
        $example = new Example('id');
        $this->assertFalse(vfsStreamWrapper::getRoot()->hasChild('id'));

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

This has several advantages:

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

Chapter 11. Testing Practices

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

During Development

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

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

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

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

During Debugging

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

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

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

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

Unit testing offers many advantages:

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

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

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

—Benjamin Smedberg

Chapter 12. Test-Driven Development

Unit Tests are a vital part of several software development practices and processes such as Test-First Programming, Extreme Programming [http://en.wikipedia.org/wiki/Extreme_Programming], and Test-Driven Development [http://en.wikipedia.org/wiki/Test-driven_development]. They also allow for Design-by-Contract [http://en.wikipedia.org/wiki/Design_by_Contract] in programming languages that do not support this methodology with language constructs.

You can use PHPUnit to write tests once you are done programming. However, the sooner a test is written after an error has been introduced, the more valuable the test is. So instead of writing tests months after the code is "complete", we can write tests days or hours or minutes after the possible introduction of a defect. Why stop there? Why not write the tests a little before the possible introduction of a defect?

Test-First Programming, which is part of Extreme Programming and Test-Driven Development, builds upon this idea and takes it to the extreme. With today's computational power, we have the opportunity to run thousands of tests thousands of times per day. We can use the feedback from all of these tests to program in small steps, each of which carries with it the assurance of a new automated test in addition to all the tests that have come before. The tests are like pitons, assuring you that, no matter what happens, once you have made progress you can only fall so far.

When you first write the test it cannot possibly run, because you are calling on objects and methods that have not been programmed yet. This might feel strange at first, but after a while you will get used to it. Think of Test-First Programming as a pragmatic approach to following the object-oriented programming principle of programming to an interface instead of programming to an implementation: while you are writing the test you are thinking about the interface of the object you are testing -- what does this object look like from the outside. When you go to make the test really work, you are thinking about pure implementation. The interface is fixed by the failing test.

> The point of Test-Driven Development [http://en.wikipedia.org/wiki/Test-driven_development] is to drive out the functionality the software actually needs, rather than what the programmer thinks it probably ought to have. The way it does this seems at first counterintuitive, if not downright silly, but it not only makes sense, it also quickly becomes a natural and elegant way to develop software.
>
> —Dan North

What follows is necessarily an abbreviated introduction to Test-Driven Development. You can explore the topic further in other books, such as *Test-Driven Development* [Beck2002] by Kent Beck or Dave Astels' *A Practical Guide to Test-Driven Development* [Astels2003].

BankAccount Example

In this section, we will look at the example of a class that represents a bank account. The contract for the `BankAccount` class not only requires methods to get and set the bank account's balance, as well as methods to deposit and withdraw money. It also specifies the following two conditions that must be ensured:

- The bank account's initial balance must be zero.
- The bank account's balance cannot become negative.

We write the tests for the `BankAccount` class before we write the code for the class itself. We use the contract conditions as the basis for the tests and name the test methods accordingly, as shown in Example 12.1, “Tests for the BankAccount class”.

Example 12.1. Tests for the BankAccount class

```
<?php
```

```
require_once 'BankAccount.php';

class BankAccountTest extends PHPUnit_Framework_TestCase
{
    protected $ba;

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

    public function testBalanceIsInitiallyZero()
    {
        $this->assertEquals(0, $this->ba->getBalance());
    }

    public function testBalanceCannotBecomeNegative()
    {
        try {
            $this->ba->withdrawMoney(1);
        }

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

            return;
        }

        $this->fail();
    }

    public function testBalanceCannotBecomeNegative2()
    {
        try {
            $this->ba->depositMoney(-1);
        }

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

            return;
        }

        $this->fail();
    }
}
?>
```

We now write the minimal amount of code needed for the first test, `testBalanceIsInitiallyZero()`, to pass. In our example this amounts to implementing the `getBalance()` method of the `BankAccount` class, as shown in Example 12.2, “Code needed for the testBalanceIsInitiallyZero() test to pass”.

Example 12.2. Code needed for the testBalanceIsInitiallyZero() test to pass

```
<?php
class BankAccount
{
    protected $balance = 0;

    public function getBalance()
    {
        return $this->balance;
```

```
    }
}
?>
```

The test for the first contract condition now passes, but the tests for the second contract condition fail because we have yet to implement the methods that these tests call.

```
phpunit BankAccountTest
PHPUnit 3.7.0 by Sebastian Bergmann.

.
Fatal error: Call to undefined method BankAccount::withdrawMoney()
```

For the tests that ensure the second contract condition to pass, we now need to implement the `withdrawMoney()`, `depositMoney()`, and `setBalance()` methods, as shown in Example 12.3, “The complete BankAccount class”. These methods are written in a such a way that they raise an `BankAccountException` when they are called with illegal values that would violate the contract conditions.

Example 12.3. The complete BankAccount class

```
<?php
class BankAccount
{
    protected $balance = 0;

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

    protected function setBalance($balance)
    {
        if ($balance >= 0) {
            $this->balance = $balance;
        } else {
            throw new BankAccountException;
        }
    }

    public function depositMoney($balance)
    {
        $this->setBalance($this->getBalance() + $balance);

        return $this->getBalance();
    }

    public function withdrawMoney($balance)
    {
        $this->setBalance($this->getBalance() - $balance);

        return $this->getBalance();
    }
}
?>
```

The tests that ensure the second contract condition now pass, too:

```
phpunit BankAccountTest
PHPUnit 3.7.0 by Sebastian Bergmann.

...
```

```
Time: 0 seconds


OK (3 tests, 3 assertions)
```

Alternatively, you can use the static assertion methods provided by the `PHPUnit_Framework_Assert` class to write the contract conditions as design-by-contract style assertions into your code, as shown in Example 12.4, “The BankAccount class with Design-by-Contract assertions”. When one of these assertions fails, an `PHPUnit_Framework_AssertionFailedError` exception will be raised. With this approach, you write less code for the contract condition checks and the tests become more readable. However, you add a runtime dependency on PHPUnit to your project.

Example 12.4. The BankAccount class with Design-by-Contract assertions

```
<?php
class BankAccount
{
    private $balance = 0;

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

    protected function setBalance($balance)
    {
        PHPUnit_Framework_Assert::assertTrue($balance >= 0);

        $this->balance = $balance;
    }

    public function depositMoney($amount)
    {
        PHPUnit_Framework_Assert::assertTrue($amount >= 0);

        $this->setBalance($this->getBalance() + $amount);

        return $this->getBalance();
    }

    public function withdrawMoney($amount)
    {
        PHPUnit_Framework_Assert::assertTrue($amount >= 0);
        PHPUnit_Framework_Assert::assertTrue($this->balance >= $amount);

        $this->setBalance($this->getBalance() - $amount);

        return $this->getBalance();
    }
}
?>
```

By writing the contract conditions into the tests, we have used Design-by-Contract to program the `BankAccount` class. We then wrote, following the Test-First Programming approach, the code needed to make the tests pass. However, we forgot to write tests that call `setBalance()`, `depositMoney()`, and `withdrawMoney()` with legal values that do not violate the contract conditions. We need a means to test our tests or at least to measure their quality. Such a means is the analysis of code-coverage information that we will discuss next.

Chapter 13. Behaviour-Driven Development

In [Astels2006], Dave Astels makes the following points:

- Extreme Programming [http://en.wikipedia.org/wiki/Extreme_Programming] originally had the rule to test everything that could possibly break.
- Now, however, the practice of testing in Extreme Programming has evolved into Test-Driven Development [http://en.wikipedia.org/wiki/Test-driven_development] (see Chapter 12, *Test-Driven Development*).
- But the tools still force developers to think in terms of tests and assertions instead of specifications.

> **So if it's not about testing, what's it about?**
>
> It's about figuring out what you are trying to do before you run off half-cocked to try to do it. You write a specification that nails down a small aspect of behaviour in a concise, unambiguous, and executable form. It's that simple. Does that mean you write tests? No. It means you write specifications of what your code will have to do. It means you specify the behaviour of your code ahead of time. But not far ahead of time. In fact, just before you write the code is best because that's when you have as much information at hand as you will up to that point. Like well done TDD, you work in tiny increments... specifying one small aspect of behaviour at a time, then implementing it.
>
> When you realize that it's all about specifying behaviour and not writing tests, your point of view shifts. Suddenly the idea of having a Test class for each of your production classes is ridiculously limiting. And the thought of testing each of your methods with its own test method (in a 1-1 relationship) will be laughable.
>
> —Dave Astels

The focus of Behaviour-Driven Development [http://en.wikipedia.org/wiki/Behavior_driven_development] is "the language and interactions used in the process of software development. Behavior-driven developers use their native language in combination with the ubiquitous language of Domain-Driven Design [http://en.wikipedia.org/wiki/Domain_driven_design] to describe the purpose and benefit of their code. This allows the developers to focus on why the code should be created, rather than the technical details, and minimizes translation between the technical language in which the code is written and the domain language spoken by the" domain experts.

> **Caution**
>
> You want to use Behat [http://behat.org/] for Behaviour-Driven Development instead of the `PHPUnit_Extensions_Story_TestCase` extension to PHPUnit that is discussed in the remainder of this chapter.

The `PHPUnit_Extensions_Story_TestCase` class adds a story framework that faciliates the definition of a Domain-Specific Language [http://en.wikipedia.org/wiki/Domain-specific_programming_language] for Behaviour-Driven Development. It can be installed like this:

```
pear install phpunit/PHPUnit_Story
```

Inside a *scenario*, `given()`, `when()`, and `then()` each represent a *step*. `and()` is the same kind as the previous step. The following methods are declared `abstract` in `PHPUnit_Extensions_Story_TestCase` and need to be implemented:

- `runGiven(&$world, $action, $arguments)`

...

- `runWhen(&$world, $action, $arguments)`

  ...

- `runThen(&$world, $action, $arguments)`

  ...

BowlingGame Example

In this section, we will look at the example of a class that calculates the score for a game of bowling. The rules for this are as follows:

- The game consists of 10 frames.
- In each frame the player has two opportunities to knock down 10 pins.
- The score for a frame is the total number of pins knocked down, plus bonuses for strikes and spares.
- A spare is when the player knocks down all 10 pins in two tries.

  The bonus for that frame is the number of pins knocked down by the next roll.

- A strike is when the player knocks down all 10 pins on the first try.

  The bonus for that frame is the value of the next two balls rolled.

Example 13.1, “Specification for the BowlingGame class” shows how the above rules can be written down as specification scenarios using `PHPUnit_Extensions_Story_TestCase`.

Example 13.1. Specification for the BowlingGame class

```
<?php
require_once 'PHPUnit/Extensions/Story/TestCase.php';
require_once 'BowlingGame.php';

class BowlingGameSpec extends PHPUnit_Extensions_Story_TestCase
{
    /**
     * @scenario
     */
    public function scoreForGutterGameIs0()
    {
        $this->given('New game')
             ->then('Score should be', 0);
    }

    /**
     * @scenario
     */
    public function scoreForAllOnesIs20()
    {
        $this->given('New game')
             ->when('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
```

```
            ->and('Player rolls', 1)
            ->and('Player rolls', 1)
            ->and('Player rolls', 1)
            ->and('Player rolls', 1)
            ->and('Player rolls', 1)
            ->and('Player rolls', 1)
            ->and('Player rolls', 1)
            ->and('Player rolls', 1)
            ->and('Player rolls', 1)
            ->and('Player rolls', 1)
            ->and('Player rolls', 1)
            ->and('Player rolls', 1)
            ->then('Score should be', 20);
    }

    /**
     * @scenario
     */
    public function scoreForOneSpareAnd3Is16()
    {
        $this->given('New game')
             ->when('Player rolls', 5)
             ->and('Player rolls', 5)
             ->and('Player rolls', 3)
             ->then('Score should be', 16);
    }

    /**
     * @scenario
     */
    public function scoreForOneStrikeAnd3And4Is24()
    {
        $this->given('New game')
             ->when('Player rolls', 10)
             ->and('Player rolls', 3)
             ->and('Player rolls', 4)
             ->then('Score should be', 24);
    }

    /**
     * @scenario
     */
    public function scoreForPerfectGameIs300()
    {
        $this->given('New game')
             ->when('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->then('Score should be', 300);
    }

    public function runGiven(&$world, $action, $arguments)
    {
        switch($action) {
            case 'New game': {
```

```
                $world['game']  = new BowlingGame;
                $world['rolls'] = 0;
            }
            break;

            default: {
                return $this->notImplemented($action);
            }
        }
    }

    public function runWhen(&$world, $action, $arguments)
    {
        switch($action) {
            case 'Player rolls': {
                $world['game']->roll($arguments[0]);
                $world['rolls']++;
            }
            break;

            default: {
                return $this->notImplemented($action);
            }
        }
    }

    public function runThen(&$world, $action, $arguments)
    {
        switch($action) {
            case 'Score should be': {
                for ($i = $world['rolls']; $i < 20; $i++) {
                    $world['game']->roll(0);
                }

                $this->assertEquals($arguments[0], $world['game']->score());
            }
            break;

            default: {
                return $this->notImplemented($action);
            }
        }
    }
}
?>
```

```
phpunit --printer PHPUnit_Extensions_Story_ResultPrinter_Text BowlingGameSpec
PHPUnit 3.7.0 by Sebastian Bergmann.

BowlingGameSpec
 [x] Score for gutter game is 0

   Given New game
    Then Score should be 0

 [x] Score for all ones is 20

   Given New game
    When Player rolls 1
     and Player rolls 1
     and Player rolls 1
     and Player rolls 1
     and Player rolls 1
     and Player rolls 1
```

```
     and Player rolls 1
     and Player rolls 1
     and Player rolls 1
     and Player rolls 1
     and Player rolls 1
     and Player rolls 1
     and Player rolls 1
     and Player rolls 1
     and Player rolls 1
     and Player rolls 1
     and Player rolls 1
     and Player rolls 1
     and Player rolls 1
    Then Score should be 20

 [x] Score for one spare and 3 is 16

   Given New game
    When Player rolls 5
     and Player rolls 5
     and Player rolls 3
    Then Score should be 16

 [x] Score for one strike and 3 and 4 is 24

   Given New game
    When Player rolls 10
     and Player rolls 3
     and Player rolls 4
    Then Score should be 24

 [x] Score for perfect game is 300

   Given New game
    When Player rolls 10
     and Player rolls 10
     and Player rolls 10
     and Player rolls 10
     and Player rolls 10
     and Player rolls 10
     and Player rolls 10
     and Player rolls 10
     and Player rolls 10
     and Player rolls 10
     and Player rolls 10
     and Player rolls 10
    Then Score should be 300

Scenarios: 5, Failed: 0, Skipped: 0, Incomplete: 0.
```

Chapter 14. Code Coverage Analysis

> The beauty of testing is found not in the effort but in the efficiency.
>
> Knowing what should be tested is beautiful, and knowing what is being tested is beautiful.
>
> —Murali Nandigama

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 helps answering questions such as:

- How do you find code that is not yet tested -- or, in other words, not yet *covered* by a test?
- How do you measure testing completeness?

An example of what code coverage statistics can mean is that if there is a method with 100 lines of code, and only 75 of these lines are actually executed when tests are being run, then the method is considered to have a code coverage of 75 percent.

PHPUnit's code coverage functionality makes use of the PHP_CodeCoverage [http://github.com/sebastianbergmann/php-code-coverage] component, which in turn leverages the statement coverage functionality provided by the Xdebug [http://www.xdebug.org/] extension for PHP.

Note

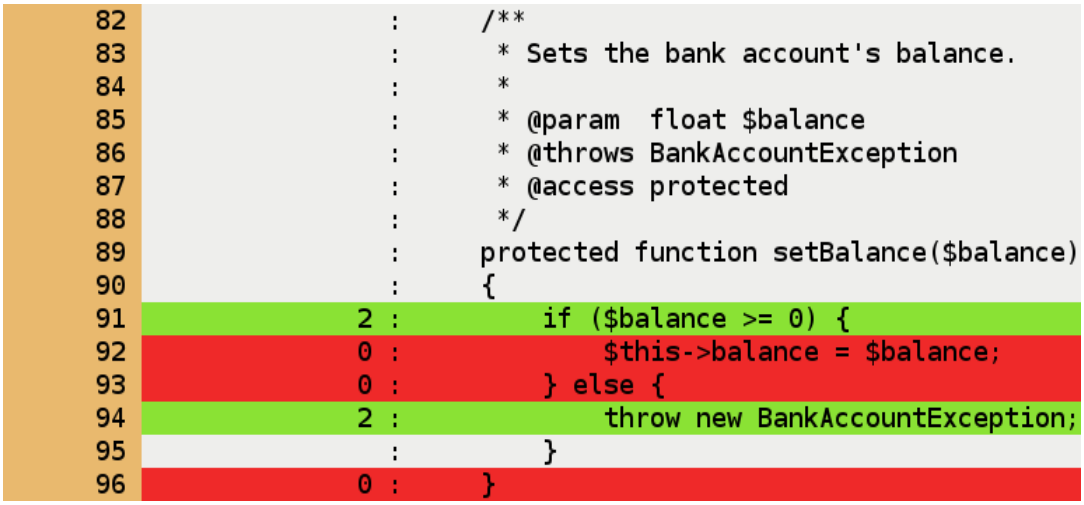
Xdebug is not distributed as part of PHPUnit. If you receive a notice while running tests that the Xdebug extension is not loaded, it means that Xdebug is either not installed or not configured properly. Before you can use the code coverage analysis features in PHPUnit, you should read the Xdebug installation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

Let us generate a code coverage report for the `BankAccount` class from Example 12.3, “The complete BankAccount class”.

```
phpunit --coverage-html ./report BankAccountTest
PHPUnit 3.7.0 by Sebastian Bergmann.

...

Time: 0 seconds

OK (3 tests, 3 assertions)

Generating report, this may take a moment.
```

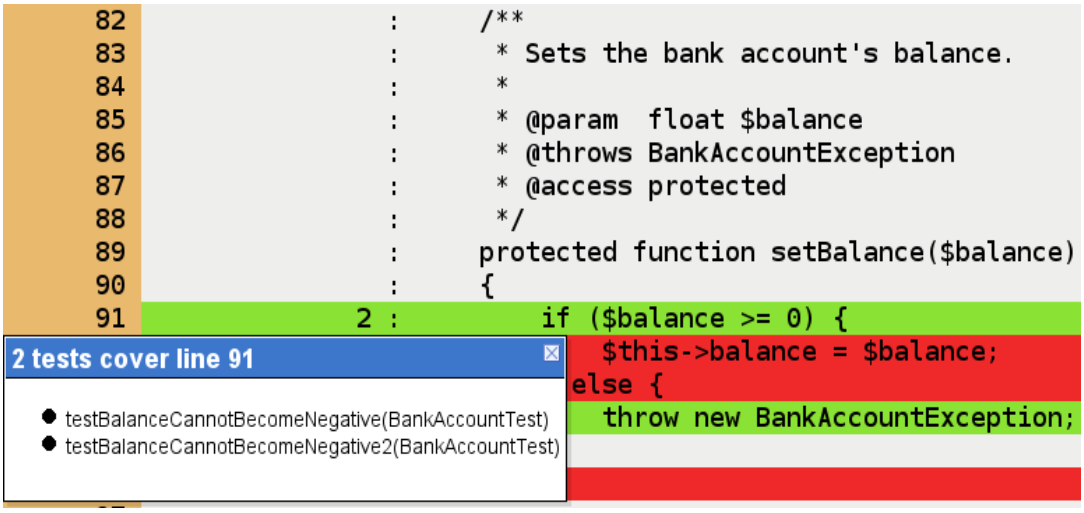
Figure 14.1, “Code Coverage for setBalance()” shows an excerpt from a Code Coverage report. Lines of code that were executed while running the tests are highlighted green, lines of code that are executable but were not executed are highlighted red, and "dead code" is highlighted grey. The number left to the actual line of code indicates how many tests cover that line.

Figure 14.1. Code Coverage for setBalance()



Clicking on the line number of a covered line will open a panel (see Figure 14.2, “Panel with information on covering tests”) that shows the test cases that cover this line.

Figure 14.2. Panel with information on covering tests



The code coverage report for our `BankAccount` example shows that we do not have any tests yet that call the `setBalance()`, `depositMoney()`, and `withdrawMoney()` methods with legal values. Example 14.1, “Test missing to achieve complete code coverage” shows a test that can be added to the `BankAccountTest` test case class to completely cover the `BankAccount` class.

Example 14.1. Test missing to achieve complete code coverage

```
<?php
require_once 'BankAccount.php';

class BankAccountTest extends PHPUnit_Framework_TestCase
{
    // ...

    public function testDepositWithdrawMoney()
    {
        $this->assertEquals(0, $this->ba->getBalance());
        $this->ba->depositMoney(1);
        $this->assertEquals(1, $this->ba->getBalance());
        $this->ba->withdrawMoney(1);
```

```
        $this->assertEquals(0, $this->ba->getBalance());
    }
}
?>
```

Figure 14.3, “Code Coverage for `setBalance()` with additional test” shows the code coverage of the `setBalance()` method with the additional test.

Figure 14.3. Code Coverage for `setBalance()` with additional test

```
     82                 :     /**
     83                 :      * Sets the bank account's balance.
     84                 :      *
     85                 :      * @param  float $balance
     86                 :      * @throws BankAccountException
     87                 :      * @access protected
     88                 :      */
     89                 :     protected function setBalance($balance)
     90                 :     {
     91               3 :         if ($balance >= 0) {
     92               1 :             $this->balance = $balance;
     93               1 :         } else {
     94               2 :             throw new BankAccountException;
     95                 :         }
     96               1 :     }
```

Specifying Covered Methods

The `@covers` annotation (see Table B.1, “Annotations for specifying which methods are covered by a test”) can be used in the test code to specify which method(s) a test method wants to test. If provided, only the code coverage information for the specified method(s) will be considered. Example 14.2, “Tests that specify which method they want to cover” shows an example.

Example 14.2. Tests that specify which method they want to cover

```
<?php
require_once 'BankAccount.php';

class BankAccountTest extends PHPUnit_Framework_TestCase
{
    protected $ba;

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

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

    /**
     * @covers BankAccount::withdrawMoney
     */
    public function testBalanceCannotBecomeNegative()
    {
```

```
        try {
            $this->ba->withdrawMoney(1);
        }

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

            return;
        }

        $this->fail();
    }

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

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

            return;
        }

        $this->fail();
    }

    /**
     * @covers BankAccount::getBalance
     * @covers BankAccount::depositMoney
     * @covers BankAccount::withdrawMoney
     */

    public function testDepositWithdrawMoney()
    {
        $this->assertEquals(0, $this->ba->getBalance());
        $this->ba->depositMoney(1);
        $this->assertEquals(1, $this->ba->getBalance());
        $this->ba->withdrawMoney(1);
        $this->assertEquals(0, $this->ba->getBalance());
    }
}
?>
```

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

Example 14.3. A test that specifies that no method should be covered

```
<?php
class GuestbookIntegrationTest extends PHPUnit_Extensions_Database_TestCase
{
    /**
     * @coversNothing
     */
    public function testAddEntry()
    {
        $guestbook = new Guestbook();
```

```
        $guestbook->addEntry("suzy", "Hello world!");

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

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 14.4, "Using the `@codeCoverageIgnore`, `@codeCoverageIgnoreStart` and `@codeCoverageIgnoreEnd` annotations".

Example 14.4. Using the `@codeCoverageIgnore`, `@codeCoverageIgnoreStart` and `@codeCoverageIgnoreEnd` annotations

```
<?php
/**
 * @codeCoverageIgnore
 */
class Foo
{
    public function bar()
    {
    }
}

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

if (FALSE) {
    // @codeCoverageIgnoreStart
    print '*';
    // @codeCoverageIgnoreEnd
}
?>
```

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

Including and Excluding Files

By default, all sourcecode files that contain at least one line of code that has been executed (and only these files) are included in the report. The sourcecode files that are included in the report can be filtered by using a blacklist or a whitelist approach.

The blacklist is pre-filled with all sourcecode files of PHPUnit itself as well as the tests. When the whitelist is empty (default), blacklisting is used. When the whitelist is not empty, whitelisting is used. Each file on the whitelist is added to the code coverage report regardless of whether or not it was executed. All lines of such a file, including those that are not executable, are counted as not executed.

When you set `processUncoveredFilesFromWhitelist="true"` in your PHPUnit configuration (see the section called “Including and Excluding Files for Code Coverage”) then these files will be included by PHP_CodeCoverage to properly calculate the number of executable lines.

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.

PHPUnit's XML configuration file (see the section called “Including and Excluding Files for Code Coverage”) can be used to control the blacklist and the whitelist. Using a whitelist is the recommended best practice to control the list of files included in the code coverage report.

Edge cases

For the most part it can safely be said that PHPUnit offers you "line based" code coverage information but due to how that information is collected there are some noteworthy edge cases.

Example 14.5.

```
<?php
// 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 15. Other Uses for Tests

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

Agile Documentation

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

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

Let us take a look at the agile documentation generated for the `BankAccount` class (from Example 12.1, “Tests for the BankAccount class”):

```
phpunit --testdox BankAccountTest
PHPUnit 3.7.0 by Sebastian Bergmann.

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

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

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

Cross-Team Tests

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

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

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

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

Chapter 16. Skeleton Generator

The PHPUnit Skeleton Generator is a tool that can generate skeleton test classes from production code classes and vice versa. It can be installed using the following command:

```
pear install phpunit/PHPUnit_SkeletonGenerator
```

Generating a Test Case Class Skeleton

When you are writing tests for existing code, you have to write the same code fragments such as

```
public function testMethod()
{
}
```

over and over again. The PHPUnit Skeleton Generator can help you by analyzing the code of the existing class and generating a skeleton test case class for it.

Example 16.1. The Calculator class

```
<?php
class Calculator
{
    public function add($a, $b)
    {
        return $a + $b;
    }
}
?>
```

The following example shows how to generate a skeleton test class for a class named `Calculator` (see Example 16.1, “The Calculator class”).

```
phpunit-skelgen --test Calculator
PHPUnit Skeleton Generator 1.0.0 by Sebastian Bergmann.

Wrote skeleton for "CalculatorTest" to "/home/sb/CalculatorTest.php".
```

For each method in the original class, there will be an incomplete test case (see Chapter 9, *Incomplete and Skipped Tests*) in the generated test case class.

Namespaced Classes and the Skeleton Generator

When you are using the skeleton generator to generate code based on a class that is declared in a namespace [http://php.net/namespace] you have to provide the qualified name of the class as well as the path to the source file it is declared in.

For instance, for a class `Calculator` that is declared in the `project` namespace you need to invoke the skeleton generator like this:

```
phpunit-skelgen --test -- "project\Calculator" Calculator.php
PHPUnit Skeleton Generator 1.0.0 by Sebastian Bergmann.

Wrote skeleton for "project\CalculatorTest" to "/home/sb/CalculatorTest.php".
```

Below is the output of running the generated test case class.

```
phpunit --bootstrap Calculator.php --verbose CalculatorTest
PHPUnit 3.7.0 by Sebastian Bergmann.

I

Time: 0 seconds, Memory: 3.50Mb

There was 1 incomplete test:

1) CalculatorTest::testAdd
This test has not been implemented yet.

/home/sb/CalculatorTest.php:38
OK, but incomplete or skipped tests!
Tests: 1, Assertions: 0, Incomplete: 1.
```

You can use `@assert` annotation in the documentation block of a method to automatically generate simple, yet meaningful tests instead of incomplete test cases. Example 16.2, “The Calculator class with @assert annotations” shows an example.

Example 16.2. The Calculator class with @assert annotations

```
<?php
class Calculator
{
    /**
     * @assert (0, 0) == 0
     * @assert (0, 1) == 1
     * @assert (1, 0) == 1
     * @assert (1, 1) == 2
     */
    public function add($a, $b)
    {
        return $a + $b;
    }
}
?>
```

Each method in the original class is checked for `@assert` annotations. These are transformed into test code such as

```
/**
     * Generated from @assert (0, 0) == 0.
     */
    public function testAdd() {
        $o = new Calculator;
        $this->assertEquals(0, $o->add(0, 0));
    }
```

Below is the output of running the generated test case class.

```
phpunit --bootstrap Calculator.php --verbose CalculatorTest
PHPUnit 3.7.0 by Sebastian Bergmann.

....

Time: 0 seconds, Memory: 3.50Mb

OK (4 tests, 4 assertions)
```

Table 16.1, “Supported variations of the @assert annotation” shows the supported variations of the `@assert` annotation and how they are transformed into test code.

Table 16.1. Supported variations of the @assert annotation

Annotation	Transformed to
`@assert (...) == X`	`assertEquals(X, method(...))`
`@assert (...) != X`	`assertNotEquals(X, method(...))`
`@assert (...) === X`	`assertSame(X, method(...))`
`@assert (...) !== X`	`assertNotSame(X, method(...))`
`@assert (...) > X`	`assertGreaterThan(X, method(...))`
`@assert (...) >= X`	`assertGreaterThanOrEqual(X, method(...))`
`@assert (...) < X`	`assertLessThan(X, method(...))`
`@assert (...) <= X`	`assertLessThanOrEqual(X, method(...))`
`@assert (...) throws X`	`@expectedException X`

Generating a Class Skeleton from a Test Case Class

When you are doing Test-Driven Development (see Chapter 12, *Test-Driven Development*) and write your tests before the code that the tests exercise, PHPUnit can help you generate class skeletons from test case classes.

Following the convention that the tests for a class `Unit` are written in a class named `UnitTest`, the test case class' source is searched for variables that reference objects of the `Unit` class and analyzing what methods are called on these objects. For example, take a look at Example 16.4, “The generated BowlingGame class skeleton” which has been generated based on the analysis of Example 16.3, “The BowlingGameTest class”.

Example 16.3. The BowlingGameTest class

```
<?php
class BowlingGameTest extends PHPUnit_Framework_TestCase
{
    protected $game;

    protected function setUp()
    {
        $this->game = new BowlingGame;
    }

    protected function rollMany($n, $pins)
    {
        for ($i = 0; $i < $n; $i++) {
            $this->game->roll($pins);
        }
    }

    public function testScoreForGutterGameIs0()
    {
        $this->rollMany(20, 0);
        $this->assertEquals(0, $this->game->score());
    }
}
?>
```

```
phpunit-skelgen --class BowlingGameTest
PHPUnit Skeleton Generator 1.0.0 by Sebastian Bergmann.

Wrote skeleton for "BowlingGame" to "./BowlingGame.php".
```

Example 16.4. The generated BowlingGame class skeleton

```
<?php
/**
 * Generated by PHPUnit_SkeletonGenerator on 2012-01-09 at 16:55:58.
 */
class BowlingGame
{
    /**
     * @todo Implement roll().
     */
    public function roll()
    {
        // Remove the following line when you implement this method.
        throw new RuntimeException('Not yet implemented.');
    }

    /**
     * @todo Implement score().
     */
    public function score()
    {
        // Remove the following line when you implement this method.
        throw new RuntimeException('Not yet implemented.');
    }
}
?>
```

Below is the output of running the test against the generated class.

```
phpunit --bootstrap BowlingGame.php BowlingGameTest
PHPUnit 3.7.0 by Sebastian Bergmann.

E

Time: 0 seconds, Memory: 3.50Mb

There was 1 error:

1) BowlingGameTest::testScoreForGutterGameIs0
RuntimeException: Not yet implemented.

/home/sb/BowlingGame.php:13
/home/sb/BowlingGameTest.php:14
/home/sb/BowlingGameTest.php:20

FAILURES!
Tests: 1, Assertions: 0, Errors: 1.
```

Chapter 17. PHPUnit and Selenium

Selenium Server

Selenium Server [http://seleniumhq.org/] is a test tool that allows you to write automated user-interface tests for web applications in any programming language against any HTTP website using any mainstream browser. It performs automated browser tasks by driving the browser's process through the operating system. Selenium tests run directly in a browser, just as real users do. These tests can be used for both *acceptance testing* (by performing higher-level tests on the integrated system instead of just testing each unit of the system independently) and *browser compatibility testing* (by testing the web application on different operating systems and browsers).

The only supported scenario of PHPUnit_Selenium is that of a Selenium 2.x server. The server can be accessed through the classic Selenium RC Api, already present in 1.x, or with the WebDriver API (partially implemented) from PHPUnit_Selenium 1.2.

The reason behind this decision is that Selenium 2 is backward compatible and Selenium RC is not maintained anymore.

Installation

First, install the Selenium Server:

1. Download a distribution archive of Selenium Server [http://seleniumhq.org/download/].
2. Unzip the distribution archive and copy `selenium-server-standalone-2.9.0.jar` (check the version suffix) to `/usr/local/bin`, for instance.
3. Start the Selenium Server server by running **`java -jar /usr/local/bin/selenium-server-standalone-2.9.0.jar`**.

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

```
"phpunit/phpunit-selenium": ">=1.2"
```

Now we can send commands to the Selenium Server using its client/server protocol.

PHPUnit_Extensions_Selenium2TestCase

The `PHPUnit_Extensions_Selenium2TestCase` test case lets you use the WebDriver API (partially implemented).

Example 17.1, “Usage example for PHPUnit_Extensions_Selenium2TestCase” shows how to test the contents of the `<title>` element of the `http://www.example.com/` website.

Example 17.1. Usage example for PHPUnit_Extensions_Selenium2TestCase

```
<?php
class WebTest extends PHPUnit_Extensions_Selenium2TestCase
{
    protected function setUp()
    {
        $this->setBrowser('firefox');
        $this->setBrowserUrl('http://www.example.com/');
    }
```

```
    public function testTitle()
    {
        $this->url('http://www.example.com/');
        $this->assertEquals('Example WWW Page', $this->title());
    }

}
?>
```

```
phpunit WebTest
PHPUnit 3.6.10 by Sebastian Bergmann.

F

Time: 28 seconds, Memory: 3.00Mb

There was 1 failure:

1) WebTest::testTitle
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'Example WWW Page'
+'IANA — Example domains'

/home/giorgio/WebTest.php:13

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

The commands of Selenium2TestCase are implemented via __call(). Please refer to the end-to-end test for PHPUnit_Extensions_Selenium2TestCase [https://github.com/sebastianbergmann/phpunit-selenium/blob/master/Tests/Selenium2TestCaseTest.php] for a list of every supported feature.

PHPUnit_Extensions_SeleniumTestCase

The `PHPUnit_Extensions_SeleniumTestCase` test case extension implements the client/server protocol to talk to Selenium Server as well as specialized assertion methods for web testing.

Example 17.2, “Usage example for PHPUnit_Extensions_SeleniumTestCase” shows how to test the contents of the `<title>` element of the `http://www.example.com/` website.

Example 17.2. Usage example for PHPUnit_Extensions_SeleniumTestCase

```
<?php
require_once 'PHPUnit/Extensions/SeleniumTestCase.php';

class WebTest extends PHPUnit_Extensions_SeleniumTestCase
{
    protected function setUp()
    {
        $this->setBrowser('*firefox');
        $this->setBrowserUrl('http://www.example.com/');
    }

    public function testTitle()
    {
        $this->open('http://www.example.com/');
        $this->assertTitle('Example WWW Page');
    }
}
```

```
?>
```

```
phpunit WebTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 9 seconds, Memory: 6.00Mb

There was 1 failure:

1) WebTest::testTitle
Current URL: http://www.iana.org/domains/example/

Failed asserting that 'IANA — Example domains' matches PCRE pattern "/Example WWW Page/"


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

Unlike with the `PHPUnit_Framework_TestCase` class, test case classes that extend `PHPUnit_Extensions_SeleniumTestCase` have to provide a `setUp()` method. This method is used to configure the Selenium Server session. See Table 17.1, “Selenium Server API: Setup” for the list of methods that are available for this.

Table 17.1. Selenium Server API: Setup

Method	Meaning
`void setBrowser(string $browser)`	Set the browser to be used by the Selenium Server server.
`void setBrowserUrl(string $browserUrl)`	Set the base URL for the tests.
`void setHost(string $host)`	Set the hostname for the connection to the Selenium Server server.
`void setPort(int $port)`	Set the port for the connection to the Selenium Server server.
`void setTimeout(int $timeout)`	Set the timeout for the connection to the Selenium Server server.
`void setSleep(int $seconds)`	Set the number of seconds the Selenium Server client should sleep between sending action commands to the Selenium Server server.

PHPUnit can optionally capture a screenshot when a Selenium test fails. To enable this, set `$captureScreenshotOnFailure`, `$screenshotPath`, and `$screenshotUrl` in your test case class as shown in Example 17.3, “Capturing a screenshot when a test fails”.

Example 17.3. Capturing a screenshot when a test fails

```
<?php
require_once 'PHPUnit/Extensions/SeleniumTestCase.php';

class WebTest extends PHPUnit_Extensions_SeleniumTestCase
{
    protected $captureScreenshotOnFailure = TRUE;
    protected $screenshotPath = '/var/www/localhost/htdocs/screenshots';
    protected $screenshotUrl = 'http://localhost/screenshots';

    protected function setUp()
```

```
    {
        $this->setBrowser('*firefox');
        $this->setBrowserUrl('http://www.example.com/');
    }

    public function testTitle()
    {
        $this->open('http://www.example.com/');
        $this->assertTitle('Example WWW Page');
    }
}
?>
```

```
phpunit WebTest
PHPUnit 3.7.0 by Sebastian Bergmann.

F

Time: 7 seconds, Memory: 6.00Mb

There was 1 failure:

1) WebTest::testTitle
Current URL: http://www.iana.org/domains/example/
Screenshot: http://localhost/screenshots/334b080f2364b5f11568ee1c7f6742c9.png

Failed asserting that 'IANA — Example domains' matches PCRE pattern "/Example WWW Page/"


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

You can run each test using a set of browsers: Instead of using `setBrowser()` to set up one browser you declare a `public static` array named `$browsers` in your test case class. Each item in this array describes one browser configuration. Each of these browsers can be hosted by different Selenium Server servers. Example 17.4, “Setting up multiple browser configurations” shows an example.

Example 17.4. Setting up multiple browser configurations

```
<?php
require_once 'PHPUnit/Extensions/SeleniumTestCase.php';

class WebTest extends PHPUnit_Extensions_SeleniumTestCase
{
    public static $browsers = array(
      array(
        'name'    => 'Firefox on Linux',
        'browser' => '*firefox',
        'host'    => 'my.linux.box',
        'port'    => 4444,
        'timeout' => 30000,
      ),
      array(
        'name'    => 'Safari on MacOS X',
        'browser' => '*safari',
        'host'    => 'my.macosx.box',
        'port'    => 4444,
        'timeout' => 30000,
      ),
      array(
        'name'    => 'Safari on Windows XP',
        'browser' => '*custom C:\Program Files\Safari\Safari.exe -url',
        'host'    => 'my.windowsxp.box',
```

```
        'port'    => 4444,
        'timeout' => 30000,
      ),
      array(
        'name'    => 'Internet Explorer on Windows XP',
        'browser' => '*iexplore',
        'host'    => 'my.windowsxp.box',
        'port'    => 4444,
        'timeout' => 30000,
      )
    );

    protected function setUp()
    {
        $this->setBrowserUrl('http://www.example.com/');
    }

    public function testTitle()
    {
        $this->open('http://www.example.com/');
        $this->assertTitle('Example Web Page');
    }
}
?>
```

`PHPUnit_Extensions_SeleniumTestCase` can collect code coverage information for tests run through Selenium:

1. Copy `PHPUnit/Extensions/SeleniumCommon/phpunit_coverage.php` into your webserver's document root directory.

2. In your webserver's `php.ini` configuration file, configure `PHPUnit/Extensions/SeleniumCommon/prepend.php` and `PHPUnit/Extensions/SeleniumCommon/append.php` as the `auto_prepend_file` and `auto_append_file`, respectively.

3. In your test case class that extends `PHPUnit_Extensions_SeleniumTestCase`, use

   ```
   protected $coverageScriptUrl = 'http://host/phpunit_coverage.php';
   ```

   to configure the URL for the `phpunit_coverage.php` script.

Table 17.2, “Assertions” lists the various assertion methods that `PHPUnit_Extensions_SeleniumTestCase` provides.

Table 17.2. Assertions

Assertion	Meaning
`void assertElementValueEquals(string $locator, string $text)`	Reports an error if the value of the element identified by `$locator` is not equal to the given `$text`.
`void assertElementValueNotEquals(string $locator, string $text)`	Reports an error if the value of the element identified by `$locator` is equal to the given `$text`.
`void assertElementValueContains(string $locator, string $text)`	Reports an error if the value of the element identified by `$locator` does not contain the given `$text`.
`void assertElementValueNotContains(string $locator, string $text)`	Reports an error if the value of the element identified by `$locator` contains the given `$text`.

Assertion	Meaning
`void assertElementContainsText(string $locator, string $text)`	Reports an error if the element identified by `$locator` does not contain the given `$text`.
`void assertElementNotContainsText(string $locator, string $text)`	Reports an error if the element identified by `$locator` contains the given `$text`.
`void assertSelectHasOption(string $selectLocator, string $option)`	Reports an error if the given option is not available.
`void assertSelectNotHasOption(string $selectLocator, string $option)`	Reports an error if the given option is available.
`void assertSelected($selectLocator, $option)`	Reports an error if the given label is not selected.
`void assertNotSelected($selectLocator, $option)`	Reports an error if the given label is selected.
`void assertIsSelected(string $selectLocator, string $value)`	Reports an error if the given value is not selected.
`void assertIsNotSelected(string $selectLocator, string $value)`	Reports an error if the given value is selected.

Table 17.3, “Template Methods” shows the template method of `PHPUnit_Extensions_SeleniumTestCase`:

Table 17.3. Template Methods

Method	Meaning
`void defaultAssertions()`	Override to perform assertions that are shared by all tests of a test case. This method is called after each command that is sent to the Selenium Server server.

Please refer to the documentation of Selenium commands [http://release.seleniumhq.org/selenium-core/1.0.1/reference.html] for a reference of the commands available and how they are used.

The commands of Selenium 1 are implemented dynamically via __call. Refer also to the API docs for PHPUnit_Extensions_SeleniumTestCase_Driver::__call() [https://github.com/sebastianbergmann/phpunit-selenium/blob/master/PHPUnit/Extensions/SeleniumTestCase/Driver.php#L410] for a list of all the supported methods on the PHP side, along with arguments and return type where available.

Using the `runSelenese($filename)` method, you can also run a Selenium test from its Selenese/HTML specification. Furthermore, using the static attribute `$seleneseDirectory`, you can automatically create test objects from a directory that contains Selenese/HTML files. The specified directory is recursively searched for `.htm` files that are expected to contain Selenese/HTML. Example 17.5, “Use a directory of Selenese/HTML files as tests” shows an example.

Example 17.5. Use a directory of Selenese/HTML files as tests

```
<?php
require_once 'PHPUnit/Extensions/SeleniumTestCase.php';
```

```
class SeleneseTests extends PHPUnit_Extensions_SeleniumTestCase
{
    public static $seleneseDirectory = '/path/to/files';
}
?>
```

From Selenium 1.1.1, an experimental feature is included allowing the user to share the session between tests. The only supported case is to share the session between all tests when a single browser is used. Call `PHPUnit_Extensions_SeleniumTestCase::shareSession(true)` in your bootstrap file to enable session sharing. The session will be reset in the case of not successul tests (failed or incomplete); it is up to the user to avoid interactions between tests by resetting cookies or logging out from the application under test (with a tearDown() method).

Chapter 18. Logging

PHPUnit can produce several types of logfiles.

Test Results (XML)

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

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

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

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

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

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

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

Test Results (TAP)

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

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

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

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

Test Results (JSON)

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

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

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

```
{"event":"suiteStart","suite":"FailureErrorTest","tests":2}
{"event":"test","suite":"FailureErrorTest",
```

```
 "test":"testFailure(FailureErrorTest)","status":"fail",
 "time":0.0082459449768066,"trace":[],
 "message":"Failed asserting that <integer:2> is equal to <integer:1>."}
{"event":"test","suite":"FailureErrorTest",
 "test":"testError(FailureErrorTest)","status":"error",
 "time":0.0083680152893066,"trace":[],"message":""}
```

Code Coverage (XML)

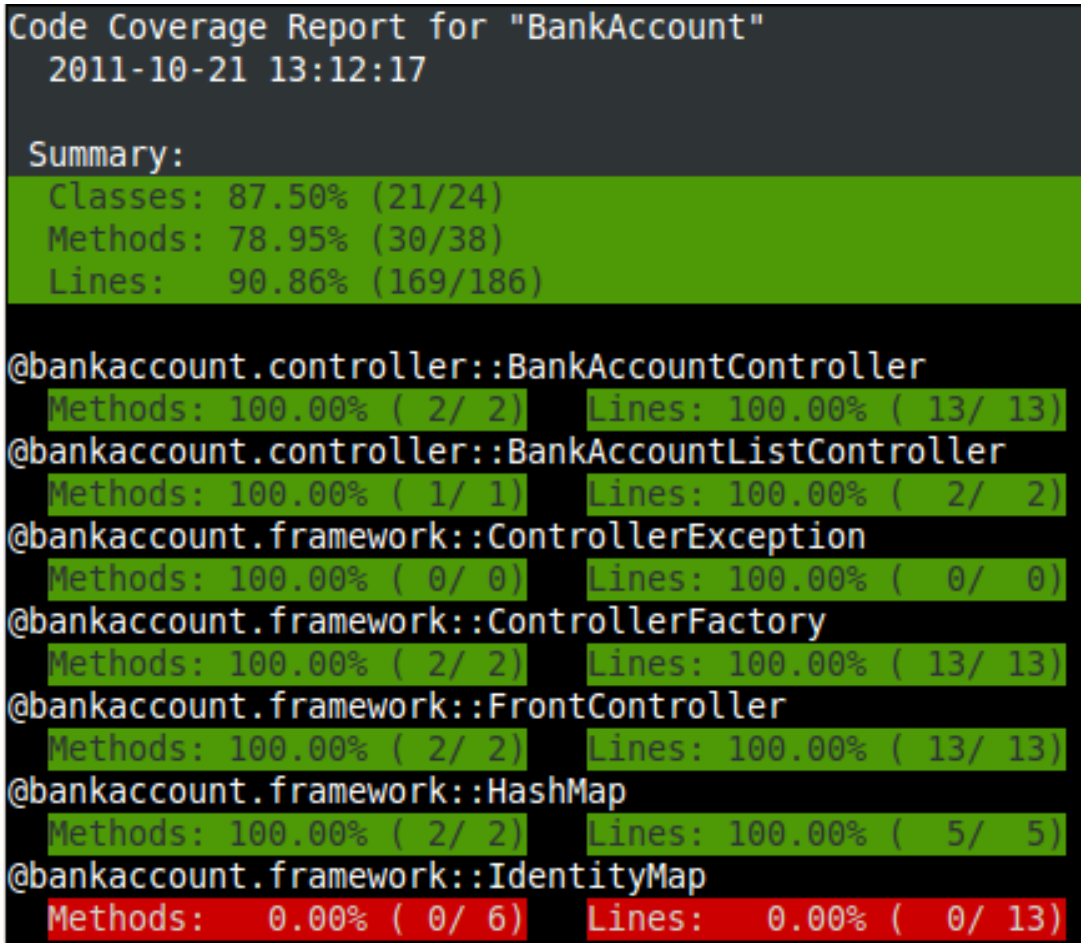
The XML format for code coverage information logging produced by PHPUnit is loosely based upon the one used by Clover [http://www.atlassian.com/software/clover/]. The following example shows the XML logfile generated for the tests in `BankAccountTest`:

```
<?xml version="1.0" encoding="UTF-8"?>
<coverage generated="1184835473" phpunit="3.6.0">
  <project name="BankAccountTest" timestamp="1184835473">
    <file name="/home/sb/BankAccount.php">
      <class name="BankAccountException">
        <metrics methods="0" coveredmethods="0" statements="0"
                 coveredstatements="0" elements="0" coveredelements="0"/>
      </class>
      <class name="BankAccount">
        <metrics methods="4" coveredmethods="4" statements="13"
                 coveredstatements="5" elements="17" coveredelements="9"/>
      </class>
      <line num="77" type="method" count="3"/>
      <line num="79" type="stmt" count="3"/>
      <line num="89" type="method" count="2"/>
      <line num="91" type="stmt" count="2"/>
      <line num="92" type="stmt" count="0"/>
      <line num="93" type="stmt" count="0"/>
      <line num="94" type="stmt" count="2"/>
      <line num="96" type="stmt" count="0"/>
      <line num="105" type="method" count="1"/>
      <line num="107" type="stmt" count="1"/>
      <line num="109" type="stmt" count="0"/>
      <line num="119" type="method" count="1"/>
      <line num="121" type="stmt" count="1"/>
      <line num="123" type="stmt" count="0"/>
      <metrics loc="126" ncloc="37" classes="2" methods="4" coveredmethods="4"
               statements="13" coveredstatements="5" elements="17"
               coveredelements="9"/>
    </file>
    <metrics files="1" loc="126" ncloc="37" classes="2" methods="4"
             coveredmethods="4" statements="13" coveredstatements="5"
             elements="17" coveredelements="9"/>
  </project>
</coverage>
```

Code Coverage (TEXT)

Human readable code coverage output for the command-line or a text file. The aim of this output format is to provide a quick coverage overview while working on a small set of classes. For bigger projects this output can be useful to get an quick overview of the projects coverage or when used with the `--filter` functionality. When used from the command-line by writing to `php://stdout` this will honor the `--colors` setting. Writing to standard out is the default option when used from the command-line. By default this will only show files that have at least one covered line. This can only be changed via the `showUncoveredFiles` xml configuration option. See the section called “Logging”.

Figure 18.1. Code Coverage output on the command-line with colors



Chapter 19. Extending PHPUnit

PHPUnit can be extended in various ways to make the writing of tests easier and customize the feedback you get from running tests. Here are common starting points to extend PHPUnit.

Subclass PHPUnit_Framework_TestCase

Write custom assertions and utility methods in an abstract subclass of `PHPUnit_Framework_TestCase` and derive your test case classes from that class. This is one of the easiest ways to extend PHPUnit.

Write custom assertions

When writing custom assertions it is the best practice to follow how PHPUnit's own assertions are implemented. As you can see in Example 19.1, "The assertTrue() and isTrue() methods of the PHPUnit_Framework_Assert class", the `assertTrue()` method is just a wrapper around the `isTrue()` and `assertThat()` methods: `isTrue()` creates a matcher object that is passed on to `assertThat()` for evaluation.

Example 19.1. The assertTrue() and isTrue() methods of the PHPUnit_Framework_Assert class

```
<?php
abstract class PHPUnit_Framework_Assert
{
    // ...

    /**
     * Asserts that a condition is true.
     *
     * @param  boolean $condition
     * @param  string  $message
     * @throws PHPUnit_Framework_AssertionFailedError
     */
    public static function assertTrue($condition, $message = '')
    {
        self::assertThat($condition, self::isTrue(), $message);
    }

    // ...

    /**
     * Returns a PHPUnit_Framework_Constraint_IsTrue matcher object.
     *
     * @return PHPUnit_Framework_Constraint_IsTrue
     * @since  Method available since Release 3.3.0
     */
    public static function isTrue()
    {
        return new PHPUnit_Framework_Constraint_IsTrue;
    }

    // ...
}?>
```

Example 19.2, "The PHPUnit_Framework_Constraint_IsTrue class" shows how `PHPUnit_Framework_Constraint_IsTrue` extends the abstract base class for matcher objects (or constraints), `PHPUnit_Framework_Constraint`.

Example 19.2. The PHPUnit_Framework_Constraint_IsTrue class

```
<?php
class PHPUnit_Framework_Constraint_IsTrue extends PHPUnit_Framework_Constraint
{
    /**
     * Evaluates the constraint for parameter $other. Returns TRUE if the
     * constraint is met, FALSE otherwise.
     *
     * @param mixed $other Value or object to evaluate.
     * @return bool
     */
    public function matches($other)
    {
        return $other === TRUE;
    }

    /**
     * Returns a string representation of the constraint.
     *
     * @return string
     */
    public function toString()
    {
        return 'is true';
    }
}?>
```

The effort of implementing the `assertTrue()` and `isTrue()` methods as well as the `PHPUnit_Framework_Constraint_IsTrue` class yields the benefit that `assertThat()` automatically takes care of evaluating the assertion and bookkeeping tasks such as counting it for statistics. Furthermore, the `isTrue()` method can be used as a matcher when configuring mock objects.

Implement PHPUnit_Framework_TestListener

Example 19.3, “A simple test listener” shows a simple implementation of the `PHPUnit_Framework_TestListener` interface.

Example 19.3. A simple test listener

```
<?php
class SimpleTestListener implements PHPUnit_Framework_TestListener
{
    public function addError(PHPUnit_Framework_Test $test, Exception $e, $time)
    {
        printf("Error while running test '%s'.\n", $test->getName());
    }

    public function addFailure(PHPUnit_Framework_Test $test, PHPUnit_Framework_Assertion
    {
        printf("Test '%s' failed.\n", $test->getName());
    }

    public function addIncompleteTest(PHPUnit_Framework_Test $test, Exception $e, $time)
    {
        printf("Test '%s' is incomplete.\n", $test->getName());
    }

    public function addSkippedTest(PHPUnit_Framework_Test $test, Exception $e, $time)
    {
        printf("Test '%s' has been skipped.\n", $test->getName());
```

```
    }

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

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

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

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

In the section called “Test Listeners” you can see how to configure PHPUnit to attach your test listener to the test execution.

Subclass PHPUnit_Extensions_TestDecorator

You can wrap test cases or test suites in a subclass of `PHPUnit_Extensions_TestDecorator` and use the Decorator design pattern to perform some actions before and after the test runs.

PHPUnit ships with one concrete test decorator: `PHPUnit_Extensions_RepeatedTest`. It is used to run a test repeatedly and only count it as a success if all iterations are successful.

Example 19.4, “The RepeatedTest Decorator” shows a cut-down version of the `PHPUnit_Extensions_RepeatedTest` test decorator that illustrates how to write your own test decorators.

Example 19.4. The RepeatedTest Decorator

```
<?php
require_once 'PHPUnit/Extensions/TestDecorator.php';

class PHPUnit_Extensions_RepeatedTest extends PHPUnit_Extensions_TestDecorator
{
    private $timesRepeat = 1;

    public function __construct(PHPUnit_Framework_Test $test, $timesRepeat = 1)
    {
        parent::__construct($test);

        if (is_integer($timesRepeat) &&
            $timesRepeat >= 0) {
            $this->timesRepeat = $timesRepeat;
        }
    }

    public function count()
```

```
    {
        return $this->timesRepeat * $this->test->count();
    }

    public function run(PHPUnit_Framework_TestResult $result = NULL)
    {
        if ($result === NULL) {
            $result = $this->createResult();
        }

        for ($i = 0; $i < $this->timesRepeat && !$result->shouldStop(); $i++) {
            $this->test->run($result);
        }

        return $result;
    }
}
?>
```

Implement PHPUnit_Framework_Test

The `PHPUnit_Framework_Test` interface is narrow and easy to implement. You can write an implementation of `PHPUnit_Framework_Test` that is simpler than `PHPUnit_Framework_TestCase` and that runs *data-driven tests*, for instance.

Example 19.5, “A data-driven test” shows a data-driven test case class that compares values from a file with Comma-Separated Values (CSV). Each line of such a file looks like `foo;bar`, where the first value is the one we expect and the second value is the actual one.

Example 19.5. A data-driven test

```
<?php
class DataDrivenTest implements PHPUnit_Framework_Test
{
    private $lines;

    public function __construct($dataFile)
    {
        $this->lines = file($dataFile);
    }

    public function count()
    {
        return 1;
    }

    public function run(PHPUnit_Framework_TestResult $result = NULL)
    {
        if ($result === NULL) {
            $result = new PHPUnit_Framework_TestResult;
        }

        foreach ($this->lines as $line) {
            $result->startTest($this);
            PHP_Timer::start();
            $stopTime = NULL;

            list($expected, $actual) = explode(';', $line);

            try {
                PHPUnit_Framework_Assert::assertEquals(
                  trim($expected), trim($actual)
```

```
                );
            }

            catch (PHPUnit_Framework_AssertionFailedError $e) {
                $stopTime = PHP_Timer::stop();
                $result->addFailure($this, $e, $stopTime);
            }

            catch (Exception $e) {
                $stopTime = PHP_Timer::stop();
                $result->addError($this, $e, $stopTime);
            }

            if ($stopTime === NULL) {
                $stopTime = PHP_Timer::stop();
            }

            $result->endTest($this, $stopTime);
        }

        return $result;
    }
}

$test = new DataDrivenTest('data_file.csv');
$result = PHPUnit_TextUI_TestRunner::run($test);
?>
```

```
PHPUnit 3.7.0 by Sebastian Bergmann.

.F

Time: 0 seconds

There was 1 failure:

1) DataDrivenTest
Failed asserting that two strings are equal.
expected string <bar>
difference      <  x>
got string      <baz>
/home/sb/DataDrivenTest.php:32
/home/sb/DataDrivenTest.php:53

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

Appendix A. Assertions

Table A.1, “Assertions” shows all the varieties of assertions.

Table A.1. Assertions

Assertion
`assertArrayHasKey($key, $array, $message = '')`
`assertArrayNotHasKey($key, $array, $message = '')`
`assertAttributeContains($needle, $haystackAttributeName, $haystackClassOrObject, $message = '', $ignoreCase = FALSE, $checkForObjectIdentity = TRUE)`
`assertAttributeContainsOnly($type, $haystackAttributeName, $haystackClassOrObject, $isNativeType = NULL, $message = '')`
`assertAttributeCount($expectedCount, $haystackAttributeName, $haystackClassOrObject, $message = '')`
`assertAttributeEmpty($haystackAttributeName, $haystackClassOrObject, $message = '')`
`assertAttributeEquals($expected, $actualAttributeName, $actualClassOrObject, $message = '', $delta = 0, $maxDepth = 10, $canonicalize = FALSE, $ignoreCase = FALSE)`
`assertAttributeGreaterThan($expected, $actualAttributeName, $actualClassOrObject, $message = '')`
`assertAttributeGreaterThanOrEqual($expected, $actualAttributeName, $actualClassOrObject, $message = '')`
`assertAttributeInstanceOf($expected, $attributeName, $classOrObject, $message = '')`
`assertAttributeInternalType($expected, $attributeName, $classOrObject, $message = '')`
`assertAttributeLessThan($expected, $actualAttributeName, $actualClassOrObject, $message = '')`
`assertAttributeLessThanOrEqual($expected, $actualAttributeName, $actualClassOrObject, $message = '')`
`assertAttributeNotContains($needle, $haystackAttributeName, $haystackClassOrObject, $message = '', $ignoreCase = FALSE, $checkForObjectIdentity = TRUE)`
`assertAttributeNotContainsOnly($type, $haystackAttributeName, $haystackClassOrObject, $isNativeType = NULL, $message = '')`
`assertAttributeNotCount($expectedCount, $haystackAttributeName, $haystackClassOrObject, $message = '')`
`assertAttributeNotEmpty($haystackAttributeName, $haystackClassOrObject, $message = '')`
`assertAttributeNotEquals($expected, $actualAttributeName, $actualClassOrObject, $message = '', $delta = 0, $maxDepth = 10, $canonicalize = FALSE, $ignoreCase = FALSE)`
`assertAttributeNotInstanceOf($expected, $attributeName, $classOrObject, $message = '')`
`assertAttributeNotInternalType($expected, $attributeName, $classOrObject, $message = '')`

Assertion
assertAttributeNotSame($expected, $actualAttributeName, $actualClassOrObject, $message = '')
assertAttributeSame($expected, $actualAttributeName, $actualClassOrObject, $message = '')
assertClassHasAttribute($attributeName, $className, $message = '')
assertClassHasStaticAttribute($attributeName, $className, $message = '')
assertClassNotHasAttribute($attributeName, $className, $message = '')
assertClassNotHasStaticAttribute($attributeName, $className, $message = '')
assertContains($needle, $haystack, $message = '', $ignoreCase = FALSE, $checkForObjectIdentity = TRUE)
assertContainsOnly($type, $haystack, $isNativeType = NULL, $message = '')
assertContainsOnlyInstancesOf($classname, $haystack, $message = '')
assertCount($expectedCount, $haystack, $message = '')
assertEmpty($actual, $message = '')
assertEqualXMLStructure(DOMElement $expectedElement, DOMElement $actualElement, $checkAttributes = FALSE, $message = '')
assertEquals($expected, $actual, $message = '', $delta = 0, $maxDepth = 10, $canonicalize = FALSE, $ignoreCase = FALSE)
assertFalse($condition, $message = '')
assertFileEquals($expected, $actual, $message = '', $canonicalize = FALSE, $ignoreCase = FALSE)
assertFileExists($filename, $message = '')
assertFileNotEquals($expected, $actual, $message = '', $canonicalize = FALSE, $ignoreCase = FALSE)
assertFileNotExists($filename, $message = '')
assertGreaterThan($expected, $actual, $message = '')
assertGreaterThanOrEqual($expected, $actual, $message = '')
assertInstanceOf($expected, $actual, $message = '')
assertInternalType($expected, $actual, $message = '')
assertJsonFileEqualsJsonFile($expectedFile, $actualFile, $message = '')
assertJsonFileNotEqualsJsonFile($expectedFile, $actualFile, $message = '')
assertJsonStringEqualsJsonFile($expectedFile, $actualJson, $message = '')
assertJsonStringEqualsJsonString($expectedJson, $actualJson, $message = '')
assertJsonStringNotEqualsJsonFile($expectedFile, $actualJson, $message = '')
assertJsonStringNotEqualsJsonString($expectedJson, $actualJson, $message = '')

Assertion
assertLessThan($expected, $actual, $message = '')
assertLessThanOrEqual($expected, $actual, $message = '')
assertNotContains($needle, $haystack, $message = '', $ignoreCase = FALSE, $checkForObjectIdentity = TRUE)
assertNotContainsOnly($type, $haystack, $isNativeType = NULL, $message = '')
assertNotCount($expectedCount, $haystack, $message = '')
assertNotEmpty($actual, $message = '')
assertNotEquals($expected, $actual, $message = '', $delta = 0, $maxDepth = 10, $canonicalize = FALSE, $ignoreCase = FALSE)
assertNotInstanceOf($expected, $actual, $message = '')
assertNotInternalType($expected, $actual, $message = '')
assertNotNull($actual, $message = '')
assertNotRegExp($pattern, $string, $message = '')
assertNotSame($expected, $actual, $message = '')
assertNotSameSize($expected, $actual, $message = '')
assertNotTag($matcher, $actual, $message = '', $isHtml = TRUE)
assertNull($actual, $message = '')
assertObjectHasAttribute($attributeName, $object, $message = '')
assertObjectNotHasAttribute($attributeName, $object, $message = '')
assertRegExp($pattern, $string, $message = '')
assertSame($expected, $actual, $message = '')
assertSameSize($expected, $actual, $message = '')
assertSelectCount($selector, $count, $actual, $message = '', $isHtml = TRUE)
assertSelectEquals($selector, $content, $count, $actual, $message = '', $isHtml = TRUE)
assertSelectRegExp($selector, $pattern, $count, $actual, $message = '', $isHtml = TRUE)
assertStringEndsNotWith($suffix, $string, $message = '')
assertStringEndsWith($suffix, $string, $message = '')
assertStringEqualsFile($expectedFile, $actualString, $message = '', $canonicalize = FALSE, $ignoreCase = FALSE)
assertStringMatchesFormat($format, $string, $message = '')
assertStringMatchesFormatFile($formatFile, $string, $message = '')
assertStringNotEqualsFile($expectedFile, $actualString, $message = '', $canonicalize = FALSE, $ignoreCase = FALSE)
assertStringNotMatchesFormat($format, $string, $message = '')
assertStringNotMatchesFormatFile($formatFile, $string, $message = '')
assertStringStartsNotWith($prefix, $string, $message = '')
assertStringStartsWith($prefix, $string, $message = '')
assertTag($matcher, $actual, $message = '', $isHtml = TRUE)

Assertion
`assertThat($value, PHPUnit_ Framework_ Constraint $constraint, $message = '')`
`assertTrue($condition, $message = '')`
`assertXmlFileEqualsXmlFile($expectedFile, $actualFile, $message = '')`
`assertXmlFileNotEqualsXmlFile($expectedFile, $actualFile, $message = '')`
`assertXmlStringEqualsXmlFile($expectedFile, $actualXml, $message = '')`
`assertXmlStringEqualsXmlString($expectedXml, $actualXml, $message = '')`
`assertXmlStringNotEqualsXmlFile($expectedFile, $actualXml, $message = '')`
`assertXmlStringNotEqualsXmlString($expectedXml, $actualXml, $message = '')`

Appendix B. Annotations

An annotation is a special form of syntactic metadata that can be added to the source code of some programming languages. While PHP has no dedicated language feature for annotating source code, the usage of tags such as `@annotation arguments` in 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.

@author

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

@backupGlobals

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

```
/**
 * @backupGlobals disabled
 */
class MyTest extends PHPUnit_Framework_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:

```
/**
 * @backupGlobals disabled
 */
class MyTest extends PHPUnit_Framework_TestCase
{
    /**
     * @backupGlobals enabled
     */
    public function testThatInteractsWithGlobalVariables()
    {
        // ...
    }
}
```

@backupStaticAttributes

The backup and restore operations for static attributes of classes can be completely disabled for all tests of a test case class like this

```
/**
 * @backupStaticAttributes disabled
 */
class MyTest extends PHPUnit_Framework_TestCase
{
    // ...
}
```

The `@backupStaticAttributes` annotation can also be used on the test method level. This allows for a fine-grained configuration of the backup and restore operations:

```
/**
 * @backupStaticAttributes disabled
 */
class MyTest extends PHPUnit_Framework_TestCase
{
    /**
     * @backupStaticAttributes enabled
     */
    public function testThatInteractsWithStaticAttributes()
    {
        // ...
    }
}
```

@codeCoverageIgnore*

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

For usage see the section called “Ignoring Code Blocks”.

@covers

The `@covers` annotation can be used in the test code to specify which method(s) a test method wants to test:

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

If provided, only the code coverage information for the specified method(s) will be considered.

Table B.1, “Annotations for specifying which methods are covered by a test” shows the syntax of the `@covers` annotation.

Table B.1. Annotations for specifying which methods are covered by a test

Annotation	Description
`@covers ClassName::methodName`	Specifies that the annotated test method covers the specified method.
`@covers ClassName`	Specifies that the annotated test method covers all methods of a given class.

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

@coversDefaultClass

The `@coversDefaultClass` annotation can be used to specify a default namespace or class name so long names don't need to be repeated for every `@covers` annotation. See Example B.1, “Using @coversDefaultClass to shorten annotations”.

Example B.1. Using @coversDefaultClass to shorten annotations

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

@coversNothing

The `@coversNothing` annotation can be used in the test code to specify that no code coverage information will be recorded for the annotated test case.

This can be used for integration testing. See Example 14.3, “A test that specifies that no method should be covered” for an example.

The annotation can be used on the class and the method level and will override any `@covers` tags.

@dataProvider

A test method can accept arbitrary arguments. These arguments are to be provided by a data provider method (`provider()` in Example 4.5, “Using a data provider that returns an array of arrays”). The data provider method to be used is specified using the `@dataProvider` annotation.

See the section called “Data Providers” for more details.

@depends

PHPUnit supports the declaration of explicit dependencies between test methods. Such dependencies do not define the order in which the test methods are to be executed but they allow the returning of an instance of the test fixture by a producer and passing it to the dependent consumers. Example 4.2, “Using the `@depends` annotation to express dependencies” shows how to use the `@depends` annotation to express dependencies between test methods.

See the section called “Test Dependencies” for more details.

@expectedException

Example 4.9, “Using the @expectedException annotation” shows how to use the `@expectedException` annotation to test whether an exception is thrown inside the tested code.

See the section called “Testing Exceptions” for more details.

@expectedExceptionCode

The `@expectedExceptionCode` annotation, in conjunction with the `@expectedException` allows making assertions on the error code of a thrown exception thus being able to narrow down a specific exception.

```
class MyTest extends PHPUnit_Framework_TestCase
{
    /**
     * @expectedException     MyException
     * @expectedExceptionCode 20
     */
    public function testExceptionHasErrorcode20()
    {
        throw new MyException('Some Message', 20);
    }
}
```

To ease testing and reduce duplication a shortcut can be used to specify a class constant as an `@expectedExceptionCode` using the "`@expectedExceptionCode ClassName::CONST`" syntax.

```
class MyTest extends PHPUnit_Framework_TestCase
{
    /**
      * @expectedException     MyException
      * @expectedExceptionCode MyClass::ERRORCODE
      */
    public function testExceptionHasErrorcode20()
    {
      throw new MyException('Some Message', 20);
    }
}
class MyClass
{
    const ERRORCODE = 20;
}
```

@expectedExceptionMessage

The `@expectedExceptionMessage` annotation works similar to `@expectedException-Code` as it lets you make an assertion on the error message of an exception.

```
class MyTest extends PHPUnit_Framework_TestCase
{
    /**
     * @expectedException        MyException
     * @expectedExceptionMessage Some Message
     */
    public function testExceptionHasRightMessage()
    {
        throw new MyException('Some Message', 20);
    }
}
```

The expected message can be a substring of the exception Message. This can be useful to only assert that a certain name or parameter that was passed in shows up in the exception and not fixate the whole exception message in the test.

```
class MyTest extends PHPUnit_Framework_TestCase
{
     /**
      * @expectedException        MyException
      * @expectedExceptionMessage broken
      */
     public function testExceptionHasRightMessage()
     {
         $param = "broken";
         throw new MyException('Invalid parameter "'.$param.'".', 20);
     }
}
```

To ease testing and reduce duplication a shortcut can be used to specify a class constant as an `@expectedExceptionMessage` using the "`@expectedExceptionMessage ClassName::CONST`" syntax. A sample can be found in the section called “@expectedException-Code”.

@group

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

```
class MyTest extends PHPUnit_Framework_TestCase
```

```
{
    /**
     * @group specification
     */
    public function testSomething()
    {
    }

    /**
     * @group regresssion
     * @group bug2204
     */
    public function testSomethingElse()
    {
    }
}
```

Tests can be selected for execution based on groups using the `--group` and `--exclude-group` options of the command-line test runner or using the respective directives of the XML configuration file.

@large

The `@large` annotation is an alias for `@group large`.

If the `PHP_Invoker` package is installed and strict mode is enabled, a large test will fail if it takes longer than 60 seconds to execute. This timeout is configurable via the `timeoutForLargeTests` attribute in the XML configuration file.

@medium

The `@medium` annotation is an alias for `@group medium`. A medium test must not depend on a test marked as `@large`.

If the `PHP_Invoker` package is installed and strict mode is enabled, a medium test will fail if it takes longer than 10 seconds to execute. This timeout is configurable via the `timeoutForMediumTests` attribute in the XML configuration file.

@preserveGlobalState

When a test is run in a separate process, PHPUnit will attempt to preserve the global state from the parent process by serializing all globals in the parent process and unserializing them in the child process. This can cause problems if the parent process contains globals that are not serializable. To fix this, you can prevent PHPUnit from preserving global state with the `@preserveGlobalState` annotation.

```
class MyTest extends PHPUnit_Framework_TestCase
{
    /**
     * @runInSeparateProcess
     * @preserveGlobalState disabled
     */
    public function testInSeparateProcess()
    {
        // ...
    }
}
```

@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 Table 9.3, “Possible @requires usages”

@runTestsInSeparateProcesses

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

```
/**
 * @runTestsInSeparateProcesses
 */
class MyTest extends PHPUnit_Framework_TestCase
{
    // ...
}
```

Note: By default, PHPUnit will attempt to preserve the global state from the parent process by serializing all globals in the parent process and unserializing them in the child process. This can cause problems if the parent process contains globals that are not serializable. See the section called “@preserveGlobalState” for information on how to fix this.

@runInSeparateProcess

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

```
class MyTest extends PHPUnit_Framework_TestCase
{
    /**
     * @runInSeparateProcess
     */
    public function testInSeparateProcess()
    {
        // ...
    }
}
```

Note: By default, PHPUnit will attempt to preserve the global state from the parent process by serializing all globals in the parent process and unserializing them in the child process. This can cause problems if the parent process contains globals that are not serializable. See the section called “@preserveGlobalState” for information on how to fix this.

@small

The `@small` annotation is an alias for `@group small`. A small test must not depend on a test marked as `@medium` or `@large`.

If the `PHP_Invoker` package is installed and strict mode is enabled, a small test will fail if it takes longer than 1 second to execute. This timeout is configurable via the `timeoutForSmallTests` attribute in the XML configuration file.

Note

By default, all tests are considered to be small if they are not marked as `@medium` or `@large`. Please note, however, that `--group` and the related options will only consider a test to be in the `small` group if it is explicitly marked with the appropriate annotation.

@test

As an alternative to prefixing your test method names with `test`, you can use the `@test` annotation in a method's DocBlock to mark it as a test method.

```
/**
 * @test
 */
public function initialBalanceShouldBe0()
{
    $this->assertEquals(0, $this->ba->getBalance());
}
```

@testdox

@ticket

Appendix C. The XML Configuration File

PHPUnit

The attributes of the `<phpunit>` element can be used to configure PHPUnit's core functionality.

```
<phpunit
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="https://schema.phpunit.de/3.7/phpunit.xsd"
         backupGlobals="true"
         backupStaticAttributes="false"
         <!--bootstrap="/path/to/bootstrap.php"-->
         cacheTokens="false"
         colors="false"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         forceCoversAnnotation="false"
         mapTestClassNameToCoveredClassName="false"
         printerClass="PHPUnit_TextUI_ResultPrinter"
         <!--printerFile="/path/to/ResultPrinter.php"-->
         processIsolation="false"
         stopOnError="false"
         stopOnFailure="false"
         stopOnIncomplete="false"
         stopOnSkipped="false"
         testSuiteLoaderClass="PHPUnit_Runner_StandardTestSuiteLoader"
         <!--testSuiteLoaderFile="/path/to/StandardTestSuiteLoader.php"-->
         timeoutForSmallTests="1"
         timeoutForMediumTests="10"
         timeoutForLargeTests="60"
         strict="false"
         verbose="false">
  <!-- ... -->
</phpunit>
```

The XML configuration above corresponds to the default behaviour of the TextUI test runner documented in the section called “Command-Line Options”.

Additional options that are not available as command-line options are:

`convertErrorsToExceptions`

By default, PHPUnit will install an error handler that converts the following errors to exceptions:

- `E_WARNING`
- `E_NOTICE`
- `E_USER_ERROR`
- `E_USER_WARNING`
- `E_USER_NOTICE`
- `E_STRICT`
- `E_RECOVERABLE_ERROR`

	• E_DEPRECATED • E_USER_DEPRECATED Set convertErrorsToExceptions to false to disable this feature.
convertNoticesToExceptions	When set to false, the error handler installed by convertErrorsToExceptions will not convert E_NOTICE, E_USER_NOTICE, or E_STRICT errors to exceptions.
convertWarningsToExceptions	When set to false, the error handler installed by convertErrorsToExceptions will not convert E_WARNING or E_USER_WARNING errors to exceptions.
forceCoversAnnotation	Code Coverage will only be recorded for tests that use the @covers annotation documented in the section called "@covers".
timeoutForLargeTests	If the PHP_Invoker package is installed and strict mode is enabled, this attribute sets the timeout for all tests marked as @large. If a test does not complete within this configured timeout, it will fail.
timeoutForMediumTests	If the PHP_Invoker package is installed and strict mode is enabled, this attribute sets the timeout for all tests marked as @medium. If a test does not complete within this configured timeout, it will fail.
timeoutForSmallTests	If the PHP_Invoker package is installed and strict mode is enabled, this attribute sets the timeout for all tests not marked as @medium or @large. If a test does not complete within this configured timeout, it will fail.

Test Suites

The <testsuites> element and its one or more <testsuite> children can be used to compose a test suite out of test suites and test cases.

```
<testsuites>
  <testsuite name="My Test Suite">
    <directory>/path/to/*Test.php files</directory>
    <file>/path/to/MyTest.php</file>
    <exclude>/path/to/exclude</exclude>
  </testsuite>
</testsuites>
```

Using the phpVersion and phpVersionOperator attributes, a required PHP version can be specified. The example below will only add the /path/to/*Test.php files and /path/to/MyTest.php file if the PHP version is at least 5.3.0.

```
<testsuites>
    <testsuite name="My Test Suite">
      <directory suffix="Test.php" phpVersion="5.3.0" phpVersionOperator=">=">/path/to/f
      <file phpVersion="5.3.0" phpVersionOperator=">=">/path/to/MyTest.php</file>
    </testsuite>
  </testsuites>
```

The phpVersionOperator attribute is optional and defaults to >=.

Groups

The <groups> element and its <include>, <exclude>, and <group> children can be used to select groups of tests marked with the @group annotation (documented in the section called "@group") that should (not) be run.

```
<groups>
  <include>
    <group>name</group>
  </include>
  <exclude>
    <group>name</group>
  </exclude>
</groups>
```

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

- `--group name`
- `--exclude-group name`

Including and Excluding Files for Code Coverage

The <filter> element and its children can be used to configure the blacklist and whitelist for the code coverage reporting.

```
<filter>
  <blacklist>
    <directory suffix=".php">/path/to/files</directory>
    <file>/path/to/file</file>
    <exclude>
      <directory suffix=".php">/path/to/files</directory>
      <file>/path/to/file</file>
    </exclude>
  </blacklist>
  <whitelist processUncoveredFilesFromWhitelist="true">
    <directory suffix=".php">/path/to/files</directory>
    <file>/path/to/file</file>
    <exclude>
      <directory suffix=".php">/path/to/files</directory>
      <file>/path/to/file</file>
    </exclude>
  </whitelist>
</filter>
```

Logging

The <logging> element and its <log> children can be used to configure the logging of the test execution.

```
<logging>
  <log type="coverage-html" target="/tmp/report" charset="UTF-8"
       highlight="false" lowUpperBound="35" highLowerBound="70"/>
  <log type="coverage-clover" target="/tmp/coverage.xml"/>
  <log type="coverage-php" target="/tmp/coverage.serialized"/>
  <log type="coverage-text" target="php://stdout" showUncoveredFiles="false"/>
```

```
  <log type="json" target="/tmp/logfile.json"/>
  <log type="tap" target="/tmp/logfile.tap"/>
  <log type="junit" target="/tmp/logfile.xml" logIncompleteSkipped="false"/>
  <log type="testdox-html" target="/tmp/testdox.html"/>
  <log type="testdox-text" target="/tmp/testdox.txt"/>
</logging>
```

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

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

The `charset`, `highlight`, `lowUpperBound`, `highLowerBound`, `logIncompleteSkipped` and `showUncoveredFiles` attributes have no equivalent TextUI test runner option.

- `charset`: Character set to be used for the generated HTML pages
- `highlight`: When set to `true`, the code in your coverage reports is syntax highlighted.
- `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.

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(
  array('Sebastian'),
  22,
  'April',
  19.78,
  NULL,
  new stdClass
);
```

Setting PHP INI settings, Constants and Global Variables

The `<php>` element and its children can be used to configure PHP settings, constants, and global variables. It can also be used to prepend the `include_path`.

```
<php>
  <includePath>.</includePath>
  <ini name="foo" value="bar"/>
  <const name="foo" value="bar"/>
  <var name="foo" value="bar"/>
  <env name="foo" value="bar"/>
  <post name="foo" value="bar"/>
  <get name="foo" value="bar"/>
  <cookie name="foo" value="bar"/>
  <server name="foo" value="bar"/>
  <files name="foo" value="bar"/>
  <request name="foo" value="bar"/>
</php>
```

The XML configuration above corresponds to the following PHP code:

```
ini_set('foo', 'bar');
define('foo', 'bar');
$GLOBALS['foo'] = 'bar';
$_ENV['foo'] = 'bar';
$_POST['foo'] = 'bar';
$_GET['foo'] = 'bar';
$_COOKIE['foo'] = 'bar';
$_SERVER['foo'] = 'bar';
$_FILES['foo'] = 'bar';
$_REQUEST['foo'] = 'bar';
```

Configuring Browsers for Selenium RC

The `<selenium>` element and its `<browser>` children can be used to configure a list of Selenium RC servers.

```
<selenium>
  <browser name="Firefox on Linux"
           browser="*firefox /usr/lib/firefox/firefox-bin"
```

```
           host="my.linux.box"
           port="4444"
           timeout="30000"/>
</selenium>
```

The XML configuration above corresponds to the following PHP code:

```
class WebTest extends PHPUnit_Extensions_SeleniumTestCase
{
    public static $browsers = array(
      array(
        'name'    => 'Firefox on Linux',
        'browser' => '*firefox /usr/lib/firefox/firefox-bin',
        'host'    => 'my.linux.box',
        'port'    => 4444,
        'timeout' => 30000
      )
    );

    // ...
}
```

Appendix D. Index

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

O

P

R

S

T

U

W

X

Appendix E. Bibliography

[Astels2003] *Test Driven Development*. David Astels. Copyright © 2003. Prentice Hall. ISBN 0131016490.

[Astels2006] *A New Look at Test-Driven Development*. David Astels. Copyright © 2006. http://blog.daveastels.com/files/BDD_Intro.pdf.

[Beck2002] *Test Driven Development by Example*. Kent Beck. Copyright © 2002. Addison-Wesley. ISBN 0-321-14653-0.

[Meszaros2007] *xUnit Test Patterns: Refactoring Test Code*. Gerard Meszaros. Copyright © 2007. Addison-Wesley. ISBN 978-0131495050.

Appendix F. Copyright

```
THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF
SUCH TERMS AND CONDITIONS.

1. Definitions

   a. "Adaptation" means a work based upon the Work, or upon the
      Work and other pre-existing works, such as a translation,
      adaptation, derivative work, arrangement of music or other
      alterations of a literary or artistic work, or phonogram or
      performance and includes cinematographic adaptations or any
      other form in which the Work may be recast, transformed, or
      adapted including in any form recognizably derived from the
      original, except that a work that constitutes a Collection
      will not be considered an Adaptation for the purpose of this
      License. For the avoidance of doubt, where the Work is a
      musical work, performance or phonogram, the synchronization of
      the Work in timed-relation with a moving image ("synching")
      will be considered an Adaptation for the purpose of this
      License.

   b. "Collection" means a collection of literary or artistic works,
      such as encyclopedias and anthologies, or performances,
      phonograms or broadcasts, or other works or subject matter
      other than works listed in Section 1(f) below, which, by
      reason of the selection and arrangement of their contents,
      constitute intellectual creations, in which the Work is
      included in its entirety in unmodified form along with one or
      more other contributions, each constituting separate and
      independent works in themselves, which together are assembled
      into a collective whole. A work that constitutes a Collection
      will not be considered an Adaptation (as defined above) for
      the purposes of this License.

   c. "Distribute" means to make available to the public the
      original and copies of the Work or Adaptation, as appropriate,
      through sale or other transfer of ownership.

   d. "Licensor" means the individual, individuals, entity or
      entities that offer(s) the Work under the terms of this License.

   e. "Original Author" means, in the case of a literary or artistic
      work, the individual, individuals, entity or entities who
      created the Work or if no individual or entity can be
      identified, the publisher; and in addition (i) in the case of
      a performance the actors, singers, musicians, dancers, and
      other persons who act, sing, deliver, declaim, play in,
      interpret or otherwise perform literary or artistic works or
      expressions of folklore; (ii) in the case of a phonogram the
      producer being the person or legal entity who first fixes the
      sounds of a performance or other sounds; and, (iii) in the
      case of broadcasts, the organization that transmits the
      broadcast.

   f. "Work" means the literary and/or artistic work offered under
      the terms of this License including without limitation any
      production in the literary, scientific and artistic domain,
      whatever may be the mode or form of its expression including
      digital form, such as a book, pamphlet and other writing; a
      lecture, address, sermon or other work of the same nature; a
      dramatic or dramatico-musical work; a choreographic work or
      entertainment in dumb show; a musical composition with or
      without words; a cinematographic work to which are assimilated
      works expressed by a process analogous to cinematography; a
      work of drawing, painting, architecture, sculpture, engraving
```

or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

g. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

h. "Publicly Perform" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

i. "Reproduce" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

b. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

c. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

d. to Distribute and Publicly Perform Adaptations.

e. For the avoidance of doubt:

```
      i. Non-waivable Compulsory License Schemes. In those
         jurisdictions in which the right to collect royalties
         through any statutory or compulsory licensing scheme cannot
         be waived, the Licensor reserves the exclusive right to
         collect such royalties for any exercise by You of the
         rights granted under this License;

      ii. Waivable Compulsory License Schemes. In those
          jurisdictions in which the right to collect royalties
          through any statutory or compulsory licensing scheme can
          be waived, the Licensor waives the exclusive right to
          collect such royalties for any exercise by You of the
          rights granted under this License; and,

      iii. Voluntary License Schemes. The Licensor waives the right
           to collect royalties, whether individually or, in the
           event that the Licensor is a member of a collecting
           society that administers voluntary licensing schemes, via
           that society, from any exercise by You of the rights
           granted under this License.

The above rights may be exercised in all media and formats whether
now known or hereafter devised. The above rights include the right
to make such modifications as are technically necessary to exercise
the rights in other media and formats. Subject to Section 8(f), all
rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly
   made subject to and limited by the following restrictions:

   a. You may Distribute or Publicly Perform the Work only under the
      terms of this License. You must include a copy of, or the
      Uniform Resource Identifier (URI) for, this License with every
      copy of the Work You Distribute or Publicly Perform. You may
      not offer or impose any terms on the Work that restrict the
      terms of this License or the ability of the recipient of the
      Work to exercise the rights granted to that recipient under
      the terms of the License. You may not sublicense the Work. You
      must keep intact all notices that refer to this License and to
      the disclaimer of warranties with every copy of the Work You
      Distribute or Publicly Perform. When You Distribute or
      Publicly Perform the Work, You may not impose any effective
      technological measures on the Work that restrict the ability
      of a recipient of the Work from You to exercise the rights
      granted to that recipient under the terms of the License. This
      Section 4(a) applies to the Work as incorporated in a
      Collection, but this does not require the Collection apart
      from the Work itself to be made subject to the terms of this
      License. If You create a Collection, upon notice from any
      Licensor You must, to the extent practicable, remove from the
      Collection any credit as required by Section 4(b), as
      requested. If You create an Adaptation, upon notice from any
      Licensor You must, to the extent practicable, remove from the
      Adaptation any credit as required by Section 4(b), as requested.

   b. If You Distribute, or Publicly Perform the Work or any
      Adaptations or Collections, You must, unless a request has
      been made pursuant to Section 4(a), keep intact all copyright
      notices for the Work and provide, reasonable to the medium or
      means You are utilizing: (i) the name of the Original Author
      (or pseudonym, if applicable) if supplied, and/or if the
      Original Author and/or Licensor designate another party or
      parties (e.g., a sponsor institute, publishing entity,
```

journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv), consistent with Section 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4 (b) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

c. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

   a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

   b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

   a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

   b. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

   c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

   d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

   e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

   f. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be

```
      enforced according to the corresponding provisions of the
      implementation of those treaty provisions in the applicable
      national law. If the standard suite of rights granted under
      applicable copyright law includes additional rights not
      granted under this License, such additional rights are deemed
      to be included in the License; this License is not intended to
      restrict the license of any rights under applicable law.

Creative Commons is not a party to this License, and makes no
warranty whatsoever in connection with the Work. Creative Commons
will not be liable to You or any party on any legal theory for any
damages whatsoever, including without limitation any general,
special, incidental or consequential damages arising in connection
to this license. Notwithstanding the foregoing two (2) sentences,
if Creative Commons has expressly identified itself as the Licensor
hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the
Work is licensed under the CCPL, Creative Commons does not authorize
the use by either party of the trademark "Creative Commons" or any
related trademark or logo of Creative Commons without the prior
written consent of Creative Commons. Any permitted use will be in
compliance with Creative Commons' then-current trademark usage
guidelines, as may be published on its website or otherwise made
available upon request from time to time. For the avoidance of
doubt, this trademark restriction does not form part of this
License.

Creative Commons may be contacted at http://creativecommons.org/.

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