

PHPUnit Manual

Sebastian Bergmann

PHPUnit Manual

Sebastian Bergmann

Publication date Edition for PHPUnit 3.5. Updated on 2011-05-21.

Copyright © 2005, 2006, 2007, 2008, 2009, 2010, 2011 Sebastian Bergmann

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

Table of Contents

1. Automating Tests	1
2. PHPUnit's Goals	3
3. Installing PHPUnit	5
4. Writing Tests for PHPUnit	6
Test Dependencies	6
Data Providers	8
Testing Exceptions	11
Testing PHP Errors	13
Assertions	14
assertArrayHasKey()	14
assertClassHasAttribute()	15
assertClassHasStaticAttribute()	16
assertContains()	17
assertContainsOnly()	18
assertEmpty()	19
assertEqualXMLStructure()	20
assertEquals()	23
assertFalse()	29
assertFileEquals()	29
assertFileExists()	31
assertGreaterThan()	31
assertGreaterThanOrEqual()	32
assertInstanceOf()	33
assertInternalType()	34
assertLessThan()	35
assertLessThanOrEqual()	36
assertNull()	36
assertObjectHasAttribute()	37
assertRegExp()	38
assertStringMatchesFormat()	39
assertStringMatchesFormatFile()	40
assertSame()	41
assertSelectCount()	43
assertSelectEquals()	45
assertSelectRegExp()	47
assertStringEndsWith()	48
assertStringEqualsFile()	49
assertStringStartsWith()	50
assertTag()	51
assertThat()	53
assertTrue()	55
assertType()	56
assertXmlFileEqualsXmlFile()	58
assertXmlStringEqualsXmlFile()	59
assertXmlStringEqualsXmlString()	60
5. The Command-Line Test Runner	62
6. Fixtures	67
More setUp() than tearDown()	69
Variations	70
Sharing Fixture	70
Global State	70
7. Organizing Tests	73
Composing a Test Suite Using the Filesystem	73
Composing a Test Suite Using XML Configuration	74
8. TestCase Extensions	76

Testing Output	76
9. Database Testing	78
Data Sets	79
Flat XML Data Set	79
XML Data Set	81
CSV Data Set	83
Replacement Data Set	84
Operations	84
Database Testing Best Practices	84
10. Incomplete and Skipped Tests	85
Incomplete Tests	85
Skipping Tests	86
11. Test Doubles	88
Stubs	88
Mock Objects	92
Stubbing and Mocking Web Services	96
Mocking the Filesystem	97
12. Testing Practices	100
During Development	100
During Debugging	100
13. Test-Driven Development	102
BankAccount Example	102
14. Behaviour-Driven Development	107
BowlingGame Example	108
15. Code Coverage Analysis	113
Specifying Covered Methods	115
Ignoring Code Blocks	116
Including and Excluding Files	117
16. Other Uses for Tests	118
Agile Documentation	118
Cross-Team Tests	118
17. Skeleton Generator	120
Generating a Test Case Class Skeleton	120
Generating a Class Skeleton from a Test Case Class	122
18. PHPUnit and Selenium	124
Selenium RC	124
PHPUnit_Extensions_SeleniumTestCase	124
19. Logging	130
Test Results (XML)	130
Test Results (TAP)	131
Test Results (JSON)	131
Code Coverage (XML)	132
20. Extending PHPUnit	133
Subclass PHPUnit_Framework_TestCase	133
Write custom assertions	133
Implement PHPUnit_Framework_TestListener	134
Subclass PHPUnit_Extensions_TestDecorator	135
Implement PHPUnit_Framework_Test	136
A. Assertions	138
B. Annotations	141
@assert	141
@author	141
@backupGlobals	141
@backupStaticAttributes	142
@covers	142
@dataProvider	143
@depends	143
@expectedException	144

@expectedExceptionCode	144
@expectedExceptionMessage	144
@group	144
@outputBuffering	144
@runTestsInSeparateProcesses	145
@runInSeparateProcess	145
@test	145
@testdox	145
@ticket	145
C. The XML Configuration File	146
PHPUnit	146
Test Suites	146
Groups	146
Including and Excluding Files for Code Coverage	147
Logging	147
Test Listeners	148
Setting PHP INI settings, Constants and Global Variables	148
Configuring Browsers for Selenium RC	149
D. Index	150
E. Bibliography	155
F. Copyright	156

List of Figures

15.1. Code Coverage for <code>setBalance()</code>	114
15.2. Panel with information on covering tests	114
15.3. Code Coverage for <code>setBalance()</code> with additional test	115

List of Tables

4.1. Methods for testing exceptions	13
4.2. Constraints	54
8.1. OutputTestCase	77
9.1. Database Test Case Methods	78
9.2. XML Data Set Element Description	82
10.1. API for Incomplete Tests	86
10.2. API for Skipping Tests	87
11.1. Matchers	95
17.1. Supported variations of the @assert annotation	122
18.1. Selenium RC API: Setup	125
18.2. Assertions	128
18.3. Template Methods	129
A.1. Assertions	138
B.1. Annotations for specifying which methods are covered by a test	143

List of Examples

1.1. Testing array operations	1
1.2. Using print to test array operations	1
1.3. Comparing expected and actual values to test array operations	1
1.4. Using an assertion function to test array operations	2
4.1. Testing array operations with PHPUnit	6
4.2. Using the @depends annotation to express dependencies	7
4.3. Exploiting the dependencies between tests	7
4.4. Using a data provider that returns an array of arrays	8
4.5. Using a data provider that returns an Iterator object	9
4.6. The CsvFileIterator class	10
4.7. Using the @expectedException annotation	11
4.8. Expecting an exception to be raised by the tested code	12
4.9. Alternative approach to testing exceptions	13
4.10. Expecting a PHP error using @expectedException	13
4.11. Usage of assertArrayHasKey()	14
4.12. Usage of assertClassHasAttribute()	15
4.13. Usage of assertClassHasStaticAttribute()	16
4.14. Usage of assertContains()	17
4.15. Usage of assertContains()	17
4.16. Usage of assertContainsOnly()	18
4.17. Usage of assertEmpty()	19
4.18. Usage of assertEqualsXMLStructure()	20
4.19. Usage of assertEquals()	23
4.20. Usage of assertEquals() with floats	24
4.21. Usage of assertEquals() with DOMDocument objects	25
4.22. Usage of assertEquals() with objects	26
4.23. Usage of assertEquals() with arrays	28
4.24. Usage of assertFalse()	29
4.25. Usage of assertFileEquals()	30
4.26. Usage of assertFileExists()	31
4.27. Usage of assertGreaterThan()	32
4.28. Usage of assertGreaterThanOrEqual()	32
4.29. Usage of assertInstanceOf()	33
4.30. Usage of assertInternalType()	34
4.31. Usage of assertLessThan()	35
4.32. Usage of assertLessThanOrEqual()	36
4.33. Usage of assertNull()	37
4.34. Usage of assertObjectHasAttribute()	37
4.35. Usage of assertRegExp()	38
4.36. Usage of assertStringMatchesFormat()	39
4.37. Usage of assertStringMatchesFormatFile()	41
4.38. Usage of assertSame()	42
4.39. Usage of assertSame() with objects	42
4.40. Usage of assertSelectCount()	43
4.41. Usage of assertSelectEquals()	45
4.42. Usage of assertSelectRegExp()	47
4.43. Usage of assertStringEndsWith()	49
4.44. Usage of assertStringEqualsFile()	49
4.45. Usage of assertStringStartsWith()	50
4.46. Usage of assertTag()	52
4.47. Usage of assertThat()	53
4.48. Usage of assertTrue()	55
4.49. Usage of assertType()	56
4.50. Usage of assertType()	57
4.51. Usage of assertXmlFileEqualsXmlFile()	58

4.52. Usage of <code>assertXmlStringEqualsXmlFile()</code>	59
4.53. Usage of <code>assertXmlStringEqualsXmlString()</code>	61
6.1. Using <code>setUp()</code> to create the stack fixture	67
6.2. Example showing all template methods available	68
6.3. Sharing fixture between the tests of a test suite	70
7.1. Composing a Test Suite Using XML Configuration	74
7.2. Composing a Test Suite Using XML Configuration	75
8.1. Using <code>PHPUnit_Extensions_OutputTestCase</code>	76
9.1. Setting up a database test case	78
9.2. A Flat XML Data Set	79
9.3. A XML Data Set	81
9.4. The XML Data Set DTD	82
9.5. CSV Data Set Example	83
10.1. Marking a test as incomplete	85
10.2. Skipping a test	86
11.1. The class we want to stub	88
11.2. Stubbing a method call to return a fixed value	89
11.3. Using the Mock Builder API can be used to configure the generated test double class.....	90
11.4. Stubbing a method call to return one of the arguments	90
11.5. Stubbing a method call to return a value from a callback	91
11.6. Stubbing a method call to return a list of values in the specified order	91
11.7. Stubbing a method call to throw an exception	92
11.8. The Subject and Observer classes that are part of the System under Test (SUT)	93
11.9. Testing that a method gets called once and with a specified argument	94
11.10. Testing that a method gets called with a number of arguments constrained in different ways	94
11.11. Testing the concrete methods of an abstract class	95
11.12. Stubbing a web service	96
11.13. A class that interacts with the filesystem	97
11.14. Testing a class that interacts with the filesystem	98
11.15. Mocking the filesystem in a test for a class that interacts with the filesystem	98
13.1. Tests for the <code>BankAccount</code> class	102
13.2. Code needed for the <code>testBalanceIsInitiallyZero()</code> test to pass	103
13.3. The complete <code>BankAccount</code> class	104
13.4. The <code>BankAccount</code> class with Design-by-Contract assertions	105
14.1. Specification for the <code>BowlingGame</code> class	108
15.1. Test missing to achieve complete code coverage	114
15.2. Tests that specify which method they want to cover	115
15.3. Using the <code>@codeCoverageIgnore</code> , <code>@codeCoverageIgnoreStart</code> and <code>@codeCoverageIgnoreEnd</code> annotations	116
17.1. The <code>Calculator</code> class	120
17.2. The <code>Calculator</code> class with <code>@assert</code> annotations	121
17.3. The <code>BowlingGameTest</code> class	122
17.4. The generated <code>BowlingGame</code> class skeleton	123
18.1. Usage example for <code>PHPUnit_Extensions_SeleniumTestCase</code>	124
18.2. Capturing a screenshot when a test fails	126
18.3. Setting up multiple browser configurations	127
18.4. Use a directory of Selenese/HTML files as tests	129
20.1. The <code>assertTrue()</code> and <code>isTrue()</code> methods of the <code>PHPUnit_Framework_Assert</code> class	133
20.2. The <code>PHPUnit_Framework_Constraint_IsTrue</code> class	134
20.3. A simple test listener	134
20.4. The <code>RepeatedTest</code> Decorator	135
20.5. A data-driven test	136

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 his 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  
$fixture = array();  
assertTrue(count($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:

<i>Easy to learn to write.</i>	If it's hard to learn how to write tests, developers will not learn to write them.
<i>Easy to write.</i>	If tests are not easy to write, developers will not write them.
<i>Easy to read.</i>	Test code should contain no extraneous overhead so that the test itself does not get lost in noise that surrounds it.
<i>Easy to execute.</i>	The tests should run at the touch of a button and present their results in a clear and unambiguous format.
<i>Quick to execute.</i>	Tests should run fast so so they can be run hundreds or thousands of times a day.
<i>Isolated.</i>	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.
<i>Composable.</i>	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:

<i>Easy to learn to write versus easy to write.</i>	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.
<i>Isolated versus quick to execute.</i>	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

PHPUnit should be installed using the PEAR Installer [<http://pear.php.net/>]. This installer is the backbone of PEAR, which provides a distribution system for PHP packages, and is shipped with every release of PHP since version 4.3.0.

Note

PHPUnit 3.5 requires PHP 5.2.7 (or later) but PHP 5.3.3 (or later) is highly recommended.

PHP_CodeCoverage, the library that is used by PHPUnit 3.5 to collect and process code coverage information, depends on Xdebug 2.0.5 (or later) but Xdebug 2.1.0 (or later) is highly recommended.

The PEAR channel (`pear.phpunit.de`) that is used to distribute PHPUnit needs to be registered with the local PEAR environment. Furthermore, component that PHPUnit depends upon are hosted on additional PEAR channels.

```
pear channel-discover pear.phpunit.de
```

```
pear channel-discover components.ez.no
```

```
pear channel-discover pear.symfony-project.com
```

This has to be done only once. Now the PEAR Installer can be used to install packages from the PHPUnit channel:

```
pear install phpunit/PHPUnit
```

After the installation you can find the PHPUnit source files inside your local PEAR directory; the path is usually `/usr/lib/php/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 necessary an encapsulated, independent entity. Often there are implicit dependencies between test methods, hidden in the implementation scenario of a test.

—Adrian Kuhn et. al.

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

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

Example 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 3.5.13 by Sebastian Bergmann.
```

```
FS
```

```
Time: 0 seconds, Memory: 4.00Mb
```

```
There was 1 failure:
```

```
1) DependencyFailureTest::testOne
Failed asserting that <boolean: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.phpunit --verbose DependencyFailureTest
```

```
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
FS
```

```
Time: 0 seconds, Memory: 4.00Mb
```

```
There was 1 failure:
```

```
1) DependencyFailureTest::testOne
Failed asserting that <boolean: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.

Data Providers

A test method can accept arbitrary arguments. These arguments are to be provided by a data provider method (`provider()` in Example 4.4, “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 4.4. 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 3.5.13 by Sebastian Bergmann.

...F

Time: 0 seconds, Memory: 4.00Mb

There was 1 failure:

1) testAdd(DataTest) with data (1, 1, 3)
Failed asserting that <integer:2> matches expected value <integer:3>.
/home/sb/DataTest.php:21

FAILURES!

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

phpunit DataTest

PHPUnit 3.5.13 by Sebastian Bergmann.

...F

Time: 0 seconds, Memory: 4.00Mb

There was 1 failure:

1) testAdd(DataTest) with data (1, 1, 3)
Failed asserting that <integer:2> matches expected value <integer:3>.
/home/sb/DataTest.php:21

FAILURES!

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

Example 4.5. 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 3.5.13 by Sebastian Bergmann.

...F

Time: 0 seconds, Memory: 4.00Mb

There was 1 failure:

1) DataTest::testAdd with data set #3 ('1', '1', '3')
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-3
+2

/home/sb/DataTest.php:11

FAILURES!
Tests: 4, Assertions: 4, Failures: 1.
phpunit DataTest
PHPUnit 3.5.13 by Sebastian Bergmann.

...F

Time: 0 seconds, Memory: 4.00Mb

There was 1 failure:

1) DataTest::testAdd with data set #3 ('1', '1', '3')
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-3
+2

/home/sb/DataTest.php:11

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

```

Example 4.6. 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++;
}
}
?>

```

Note

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.

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.

Testing Exceptions

Example 4.7, “Using the `@expectedException` annotation” shows how to use the `@expectedException` annotation to test whether an exception is thrown inside the tested code.

Example 4.7. Using the `@expectedException` annotation

```

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

```

```
?>
```

```
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) testException(ExceptionTest)
Expected exception InvalidArgumentException

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit ExceptionTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) testException(ExceptionTest)
Expected exception InvalidArgumentException

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

Alternatively, you can use the `setExpectedException()` method to set the expected exception as shown in Example 4.8, “Expecting an exception to be raised by the tested code”.

Example 4.8. Expecting an exception to be raised by the tested code

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

```
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) testException(ExceptionTest)
Expected exception InvalidArgumentException

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit ExceptionTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F
```

```

Time: 0 seconds

There was 1 failure:

1) testException(ExceptionTest)
Expected exception InvalidArgumentException

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

```

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

Table 4.1. Methods for testing exceptions

Method	Meaning
<code>void setExpectedException(string \$exceptionName)</code>	Set the name of the expected exception to <code>\$exceptionName</code> .
<code>String getExpectedException()</code>	Return the name of the expected exception.

You can also use the approach shown in Example 4.9, “Alternative approach to testing exceptions” to test exceptions.

Example 4.9. 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 4.9, “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 4.10, “Expecting a PHP error using `@expectedException`”.

Example 4.10. 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 3.5.13 by Sebastian Bergmann.

.

Time: 0 seconds

OK (1 test, 1 assertion)phpunit ExpectedErrorTest
PHPUnit 3.5.13 by Sebastian Bergmann.

.

Time: 0 seconds

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 the `Exception` class, for instance, is too general in most cases and may have undesirable side-effects.

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 4.11. Usage of `assertArrayHasKey()`

```

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

```

```

PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

```

```

1) ArrayHasKeyTest::testFailure
Failed asserting that an array has the key <string:foo>.
/home/sb/ArrayHasKeyTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit ArrayHasKeyTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) ArrayHasKeyTest::testFailure
Failed asserting that an array has the key <string: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 4.12. Usage of assertClassHasAttribute()

```

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

```

```

PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

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.phpunit ClassHasAttributeTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

```

```

Time: 0 seconds

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
$class_name[, string $message = ''])

```

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

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

Example 4.13. Usage of `assertClassHasStaticAttribute()`

```

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

```

```

PHPUnit 3.5.13 by Sebastian Bergmann.

```

```
F
```

```
Time: 0 seconds
```

```
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.phpunit ClassHasStaticAttributeTest
PHPUnit 3.5.13 by Sebastian Bergmann.

```

```
F
```

```
Time: 0 seconds
```

```
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 4.14. Usage of `assertContains()`

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

```
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) ContainsTest::testFailure
Failed asserting that an array contains <integer:4>.
/home/sb/ContainsTest.php:6
```

```
FAILURES!
```

```
Tests: 1, Assertions: 1, Failures: 1.phpunit ContainsTest
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) ContainsTest::testFailure
Failed asserting that an array contains <integer:4>.
/home/sb/ContainsTest.php:6
```

```
FAILURES!
```

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

```
assertContains(string $needle, string $haystack[, string $message = ''])
```

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

Example 4.15. Usage of `assertContains()`

```
<?php
```

```

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

```

```

PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) ContainsTest::testFailure
Failed asserting that <string:foobar> contains "baz".
/home/sb/ContainsTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit ContainsTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) ContainsTest::testFailure
Failed asserting that <string:foobar> contains "baz".
/home/sb/ContainsTest.php:6

FAILURES!
Tests: 1, Assertions: 1, 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 actual value.

Example 4.16. Usage of `assertContainsOnly()`

```

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

```

```

        $this->assertContainsOnly('string', array('1', '2', 3));
    }
}
?>

```

```
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
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.phpunit ContainsOnlyTest
```

```
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
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.
```

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 4.17. Usage of `assertEmpty()`

```
<?php
```

```

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

```

PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

```

1) EmptyTest::testFailure
Failed asserting that
Array
(
    [0] => foo
)
is empty.

```

/home/sb/EmptyTest.php:6

FAILURES!

Tests: 1, Assertions: 1, Failures: 1.phpunit EmptyTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

```

1) EmptyTest::testFailure
Failed asserting that
Array
(
    [0] => foo
)
is empty.

```

/home/sb/EmptyTest.php:6

FAILURES!

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

assertEqualXMLStructure()

```

assertEqualXMLStructure(DOMNode $expectedNode, DOMNode $actualNode[,
boolean $checkAttributes = FALSE, string $message = ''])

```

Reports an error identified by \$message if the XML Structure of the DOMNode in \$actualNode is not equal to the XML structure of the DOMNode in \$expectedNode.

Example 4.18. 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, $actual, 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, $actual);
    }

    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, $actual);
    }
}
?>

```

PHPUnit 3.5.13 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 <integer:0> matches expected <integer:1>.

/home/sb/EqualXMLStructureTest.php:20

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

/home/sb/EqualXMLStructureTest.php:31

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

/home/sb/EqualXMLStructureTest.php:42

FAILURES!
Tests: 4, Assertions: 15, Failures: 4.
phpunit EqualXMLStructureTest
PHPUnit 3.5.13 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 <integer:0> matches expected <integer:1>.

/home/sb/EqualXMLStructureTest.php:20

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

/home/sb/EqualXMLStructureTest.php:31

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

/home/sb/EqualXMLStructureTest.php:42
```

```
FAILURES!
Tests: 4, Assertions: 15, 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 4.19. 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 3.5.13 by Sebastian Bergmann.
```

```
FFF
```

```
Time: 0 seconds
```

```
There were 3 failures:
```

```
1) EqualsTest::testFailure
Failed asserting that <integer:0> matches expected value <integer:1>.
/home/sb/EqualsTest.php:11

2) EqualsTest::testFailure2
Failed asserting that two strings are equal.
expected string <bar>
difference      < x>
got string      <baz>
/home/sb/EqualsTest.php:16

3) EqualsTest::testFailure3
Failed asserting that two strings are equal.
--- Expected
```

```

+++ Actual
@@ -1,3 +1,3 @@
  foo
-bar
+bah
  baz

/home/sb/EqualsTest.php:21

FAILURES!
Tests: 3, Assertions: 3, Failures: 3.phpunit EqualsTest
PHPUnit 3.5.13 by Sebastian Bergmann.

FFF

Time: 0 seconds

There were 3 failures:

1) EqualsTest::testFailure
Failed asserting that <integer:0> matches expected value <integer:1>.
/home/sb/EqualsTest.php:11

2) EqualsTest::testFailure2
Failed asserting that two strings are equal.
expected string <bar>
difference      < x>
got string      <baz>
/home/sb/EqualsTest.php:16

3) EqualsTest::testFailure3
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ -1,3 +1,3 @@
  foo
-bar
+bah
  baz

/home/sb/EqualsTest.php:21

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 about comparing floating-point numbers [http://en.wikipedia.org/wiki/IEEE_754#Comparing_floating-point_numbers] to understand why `$delta` is necessary.

Example 4.20. Usage of `assertEquals()` with floats

```

<?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 3.5.13 by Sebastian Bergmann.

.F

Time: 0 seconds

There was 1 failure:

1) EqualsTest::testFailure
Failed asserting that <double:1.1> matches expected value <double:1>.
/home/sb/EqualsTest.php:11

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.phpunit EqualsTest
PHPUnit 3.5.13 by Sebastian Bergmann.

.F

Time: 0 seconds

There was 1 failure:

1) EqualsTest::testFailure
Failed asserting that <double:1.1> matches expected value <double:1>.
/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 4.21. 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 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) EqualsTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ -1,4 +1,4 @@
 <?xml version="1.0"?>
-<foo>
- <bar/>
-</foo>
+<bar>
+ <foo/>
+</bar>

/home/sb/EqualsTest.php:12

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit EqualsTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) EqualsTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ -1,4 +1,4 @@
 <?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 4.22. 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 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

```
1) EqualsTest::testFailure
Failed asserting that two objects are equal.
--- Expected
+++ Actual
@@ -1,5 +1,5 @@
    stdClass Object
    (
-     [foo] => foo
-     [bar] => bar
+     [foo] => bar
+     [baz] => bar
    )
```

/home/sb/EqualsTest.php:14

FAILURES!

Tests: 1, Assertions: 1, Failures: 1.phpunit EqualsTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

```
1) EqualsTest::testFailure
Failed asserting that two objects are equal.
--- Expected
+++ Actual
@@ -1,5 +1,5 @@
    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 4.23. 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 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) EqualsTest::testFailure  
Failed asserting that two arrays are equal.  
--- Expected  
+++ Actual  
@@ -1,6 +1,6 @@  
  Array  
  (  
      [0] => a  
-   [1] => b  
-   [2] => c  
+   [1] => c  
+   [2] => d  
  )
```

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

```
FAILURES!  
Tests: 1, Assertions: 1, Failures: 1.phpunit EqualsTest  
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) EqualsTest::testFailure  
Failed asserting that two arrays are equal.  
--- Expected  
+++ Actual  
@@ -1,6 +1,6 @@  
  Array  
  (  
      [0] => a  
-   [1] => b  
-   [2] => c  
+   [1] => c  
+   [2] => d  
  )
```

```

    [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 4.24. Usage of assertFalse()

```

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

```

```

PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) FalseTest::testFailure
Failed asserting that <boolean:true> is false.
/home/sb/FalseTest.php:11

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit FalseTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) FalseTest::testFailure
Failed asserting that <boolean:true> is false.
/home/sb/FalseTest.php:11

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 4.25. Usage of assertFileEquals()

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

```
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) FileEqualsTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ -1,2 +1,2 @@
-expected
+actual
```

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

```
FAILURES!
```

```
Tests: 1, Assertions: 3, Failures: 1.phpunit FileEqualsTest
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) FileEqualsTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ -1,2 +1,2 @@
-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 4.26. Usage of `assertFileExists()`

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

```
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

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.phpunit FileExistsTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

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 4.27. Usage of assertGreaterThan()

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

```
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) GreaterThanTest::testFailure
Failed asserting that <integer:1> is greater than <integer:2>.
/home/sb/GreaterThanTest.php:6
```

```
FAILURES!
```

```
Tests: 1, Assertions: 1, Failures: 1.phpunit GreaterThanTest
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) GreaterThanTest::testFailure
Failed asserting that <integer:1> is greater than <integer: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 4.28. Usage of assertGreaterThanOrEqual()

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

```
}
?>
```

```
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) GreatThanOrEqualTest::testFailure
Failed asserting that <integer:1> is equal to <integer:2> or is greater than <integer:2>
/home/sb/GreaterThanOrEqualTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit GreaterThanOrEqualTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) GreatThanOrEqualTest::testFailure
Failed asserting that <integer:1> is equal to <integer:2> or is greater than <integer:2>
/home/sb/GreaterThanOrEqualTest.php:6

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

assertInstanceOf()

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

Reports an error identified by \$message if \$actual is not an instance of \$expected.

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

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

Example 4.29. Usage of assertInstanceOf()

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

```
PHPUnit 3.5.13 by Sebastian Bergmann.

F
```

```

Time: 0 seconds

There was 1 failure:

1) InstanceOfTest::testFailure
Failed asserting that <Exception> is an instance of class "RuntimeException".
/home/sb/InstanceOfTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit InstanceOfTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) InstanceOfTest::testFailure
Failed asserting that <Exception> 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 4.30. Usage of `assertInternalType()`

```

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

```

```

PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) InternalTypeTest::testFailure
Failed asserting that <integer:42> is of type "string".
/home/sb/InstanceOfTest.php:6

FAILURES!

```

```

Tests: 1, Assertions: 1, Failures: 1.phpunit InternalTypeTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) InternalTypeTest::testFailure
Failed asserting that <integer:42> is of type "string".
/home/sb/InstanceOfTest.php:6

FAILURES!
Tests: 1, Assertions: 1, 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 4.31. Usage of assertLessThan()

```

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

```

PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) LessThanTest::testFailure
Failed asserting that <integer:2> is less than <integer:1>.
/home/sb/LessThanTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit LessThanTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:
    
```

```

1) LessThanTest::testFailure
Failed asserting that <integer:2> is less than <integer: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 4.32. Usage of assertLessThanOrEqual()

```

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

```

```

PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) LessThanOrEqualTest::testFailure
Failed asserting that <integer:2> is equal to <integer:1> or is less than <integer:1>.
/home/sb/LessThanOrEqualTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit LessThanOrEqualTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) LessThanOrEqualTest::testFailure
Failed asserting that <integer:2> is equal to <integer:1> or is less than <integer:1>.
/home/sb/LessThanOrEqualTest.php:6

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

```

assertNull()

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

Reports an error identified by \$message if \$variable is not NULL.

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

Example 4.33. Usage of assertNull()

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

```
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) NullTest::testFailure
Failed asserting that <string:foo> is null.
/home/sb/NullTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit NotNullTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) NullTest::testFailure
Failed asserting that <string:foo> is null.
/home/sb/NullTest.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 4.34. Usage of assertObjectHasAttribute()

```
<?php
```

```

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

```

```
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
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.phpunit ObjectHasAttributeTest
PHPUnit 3.5.13 by Sebastian Bergmann.

```

```
F
```

```
Time: 0 seconds
```

```
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 4.35. Usage of assertRegExp()

```

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

```

```
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```

F

Time: 0 seconds

There was 1 failure:

1) RegExpTest::testFailure
Failed asserting that <string:bar> matches PCRE pattern "/foo/".
/home/sb/RegExpTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit RegExpTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) RegExpTest::testFailure
Failed asserting that <string: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 4.36. Usage of `assertStringMatchesFormat()`

```

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

```

```

PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) StringMatchesFormatTest::testFailure
Failed asserting that two strings are equal.
--- Expected

```

```

+++ Actual
@@ @@
-%i
+foo

/home/sb/StringMatchesFormatTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit StringMatchesFormatTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) StringMatchesFormatTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-%i
+foo

/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 4.37. Usage of assertStringMatchesFormatFile()

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

```
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) StringMatchesFormatFileTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-%i
+foo
```

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

```
FAILURES!
```

```
Tests: 1, Assertions: 1, Failures: 1.phpunit StringMatchesFormatFileTest
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) StringMatchesFormatFileTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-%i
+foo
```

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

```
FAILURES!
```

```
Tests: 1, Assertions: 1, 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 4.38. Usage of `assertSame()`

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

```
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) SameTest::testFailure
<integer:2204> does not match expected type "string".
/home/sb/SameTest.php:6
```

```
FAILURES!
```

```
Tests: 1, Assertions: 1, Failures: 1.phpunit SameTest
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) SameTest::testFailure
<integer:2204> does not match expected type "string".
/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 4.39. Usage of `assertSame()` with objects

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

```

PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

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.phpunit SameTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

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 4.40. 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 3.5.13 by Sebastian Bergmann.

FFFF

Time: 0 seconds, Memory: 5.75Mb

There were 4 failures:

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

/home/sb/SelectCountTest.php:12

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

/home/sb/SelectCountTest.php:17

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

/home/sb/SelectCountTest.php:22

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

/home/sb/SelectCountTest.php:27

FAILURES!

Tests: 4, Assertions: 4, Failures: 4.

phpunit SelectCountTest

PHPUnit 3.5.13 by Sebastian Bergmann.

FFFF

Time: 0 seconds, Memory: 5.75Mb

There were 4 failures:

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

/home/sb/SelectCountTest.php:12

```

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

/home/sb/SelectCountTest.php:17

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

/home/sb/SelectCountTest.php:22

4) SelectCountTest::testRangeFailure
Failed asserting that <boolean: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 4.41. 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 3.5.13 by Sebastian Bergmann.

FFFF

Time: 0 seconds, Memory: 5.75Mb

There were 4 failures:

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

/home/sb/SelectEqualsTest.php:12

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

/home/sb/SelectEqualsTest.php:17

3) SelectEqualsTest::testExactCountFailure
Failed asserting that <integer:2> matches expected <integer:5>.

/home/sb/SelectEqualsTest.php:22

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

/home/sb/SelectEqualsTest.php:27

FAILURES!

Tests: 4, Assertions: 4, Failures: 4.

phpunit SelectEqualsTest

PHPUnit 3.5.13 by Sebastian Bergmann.

FFFF

Time: 0 seconds, Memory: 5.75Mb

There were 4 failures:

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

/home/sb/SelectEqualsTest.php:12

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

/home/sb/SelectEqualsTest.php:17

3) SelectEqualsTest::testExactCountFailure
Failed asserting that <integer:2> matches expected <integer:5>.

/home/sb/SelectEqualsTest.php:22

4) SelectEqualsTest::testRangeFailure
Failed asserting that <boolean: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 4.42. 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 3.5.13 by Sebastian Bergmann.
```

```
FFFF
```

```
Time: 0 seconds, Memory: 5.75Mb
```

```
There were 4 failures:

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

/home/sb/SelectRegExpTest.php:12

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

/home/sb/SelectRegExpTest.php:17

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

/home/sb/SelectRegExpTest.php:22

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

/home/sb/SelectRegExpTest.php:27

FAILURES!
Tests: 4, Assertions: 4, Failures: 4.
phpunit SelectRegExpTest
PHPUnit 3.5.13 by Sebastian Bergmann.

FFFF

Time: 0 seconds, Memory: 5.75Mb

There were 4 failures:

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

/home/sb/SelectRegExpTest.php:12

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

/home/sb/SelectRegExpTest.php:17

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

/home/sb/SelectRegExpTest.php:22

4) SelectRegExpTest::testRangeFailure
Failed asserting that <boolean: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`.

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

Example 4.43. Usage of `assertStringEndsWith()`

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

```
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) StringEndsWithTest::testFailure
Failed asserting that <string:foo> ends with "suffix".
/home/sb/StringEndsWithTest.php:6
```

```
FAILURES!
```

```
Tests: 1, Assertions: 1, Failures: 1.phpunit StringEndsWithTest
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) StringEndsWithTest::testFailure
Failed asserting that <string: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 4.44. Usage of `assertStringEqualsFile()`

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

```
}
}
?>
```

```
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) StringEqualsFileTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ -1,2 +1 @@
-expected
-
+actual
\ No newline at end of file

/home/sb/StringEqualsFileTest.php:6

FAILURES!
Tests: 1, Assertions: 2, Failures: 1.phpunit StringEqualsFileTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) StringEqualsFileTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ -1,2 +1 @@
-expected
-
+actual
\ No newline at end of file

/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 4.45. Usage of assertStringStartsWith()

```
<?php
```

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

```
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) StringStartsWithTest::testFailure
Failed asserting that <string:foo> starts with "prefix".
/home/sb/StringStartsWithTest.php:6
```

```
FAILURES!
```

```
Tests: 1, Assertions: 1, Failures: 1.phpunit StringStartsWithTest
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) StringStartsWithTest::testFailure
Failed asserting that <string: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 corresponding value.
- `tags`: The node type must match the corresponding value.
- `attributes`: The node's attributes must match the corresponding 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 4.46. 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' => '/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 4.47, “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 4.47. 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 4.2, “Constraints” shows the available PHPUnit_Framework_Constraint classes.

Table 4.2. 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_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_PCERMatch 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 4.48. Usage of assertTrue()

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

```

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

```

```

PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) TrueTest::testFailure
Failed asserting that <boolean:false> is true.
/home/sb/TrueTest.php:11

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit TrueTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) TrueTest::testFailure
Failed asserting that <boolean:false> is true.
/home/sb/TrueTest.php:11

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

```

assertType ()

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

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

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

`assertAttributeType ()` and `assertAttributeNotType ()` 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 `assertType()`

```

<?php
class TypeTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertType('Exception', new stdClass);
    }
}
?>

```

```

PHPUnit 3.5.13 by Sebastian Bergmann.

```

```
F

Time: 0 seconds

There was 1 failure:

3) TypeTest::testFailure
Failed asserting that <stdClass> is an instance of class "Exception".
/home/sb/TypeTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit TypeTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

3) TypeTest::testFailure
Failed asserting that <stdClass> is an instance of class "Exception".
/home/sb/TypeTest.php:6

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

Alternatively, \$expected can be one of these constants to indicate an internal type:

- PHPUnit_Framework_Constraint_IsType::TYPE_ARRAY("array")
- PHPUnit_Framework_Constraint_IsType::TYPE_BOOL("bool")
- PHPUnit_Framework_Constraint_IsType::TYPE_FLOAT("float")
- PHPUnit_Framework_Constraint_IsType::TYPE_INT("int")
- PHPUnit_Framework_Constraint_IsType::TYPE_NULL("null")
- PHPUnit_Framework_Constraint_IsType::TYPE_NUMERIC("numeric")
- PHPUnit_Framework_Constraint_IsType::TYPE_OBJECT("object")
- PHPUnit_Framework_Constraint_IsType::TYPE_RESOURCE("resource")
- PHPUnit_Framework_Constraint_IsType::TYPE_STRING("string")

Example 4.50. Usage of assertType()

```
<?php
class TypeTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertType(PHPUnit_Framework_Constraint_IsType::TYPE_STRING, 2204);
    }
}
?>
```

PHPUnit 3.5.13 by Sebastian Bergmann.

```

F

Time: 0 seconds

There was 1 failure:

1) TypeTest::testFailure
Failed asserting that <integer:2204> is of type "string".
/home/sb/TypeTest.php:6

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit TypeTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) TypeTest::testFailure
Failed asserting that <integer:2204> is of type "string".
/home/sb/TypeTest.php:6

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

```

Note

`assertType()` will be removed in PHPUnit 3.6 and should no longer be used. `assertInternalType` (see the section called “`assertInternalType()`”) should be used for asserting internal types such as `integer` or `string` whereas `assertInstanceOf` (see the section called “`assertInstanceOf()`”) should be used for asserting that an object is an instance of a specified class or interface.

`assertXmlFileEqualsXmlFile()`

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

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 4.51. Usage of `assertXmlFileEqualsXmlFile()`

```

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

```

PHPUnit 3.5.13 by Sebastian Bergmann.

```

F

Time: 0 seconds

There was 1 failure:

1) XmlFileEqualsXmlFileTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ -1,4 +1,4 @@
 <?xml version="1.0"?>
 <foo>
- <bar/>
+ <baz/>
 </foo>

/home/sb/XmlFileEqualsXmlFileTest.php:7

FAILURES!
Tests: 1, Assertions: 3, Failures: 1.phpunit XmlFileEqualsXmlFileTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) XmlFileEqualsXmlFileTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ -1,4 +1,4 @@
 <?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 4.52. Usage of `assertXmlStringEqualsXmlFile()`

```
<?php
class XmlStringEqualsXmlFileTest extends PHPUnit_Framework_TestCase
{
```

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

PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

```
1) XmlStringEqualsXmlFileTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ -1,4 +1,4 @@
 <?xml version="1.0"?>
 <foo>
- <bar/>
+ <baz/>
 </foo>
```

/home/sb/XmlStringEqualsXmlFileTest.php:7

FAILURES!

Tests: 1, Assertions: 2, Failures: 1.phpunit XmlStringEqualsXmlFileTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

```
1) XmlStringEqualsXmlFileTest::testFailure
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ -1,4 +1,4 @@
 <?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 4.53. Usage of `assertXmlStringEqualsXmlString()`

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

```
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) XmlStringEqualsXmlStringTest::testFailure
Failed asserting that two strings are equal.
```

```
--- Expected
```

```
+++ Actual
```

```
@@ -1,4 +1,4 @@
```

```
<?xml version="1.0"?>
<foo>
- <bar/>
+ <baz/>
</foo>
```

```
/home/sb/XmlStringEqualsXmlStringTest.php:7
```

```
FAILURES!
```

```
Tests: 1, Assertions: 1, Failures: 1.phpunit XmlStringEqualsXmlStringTest
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 0 seconds
```

```
There was 1 failure:
```

```
1) XmlStringEqualsXmlStringTest::testFailure
Failed asserting that two strings are equal.
```

```
--- Expected
```

```
+++ Actual
```

```
@@ -1,4 +1,4 @@
```

```
<?xml version="1.0"?>
<foo>
- <bar/>
+ <baz/>
</foo>
```

```
/home/sb/XmlStringEqualsXmlStringTest.php:7
```

```
FAILURES!
```

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

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 3.5.13 by Sebastian Bergmann.  
  
..  
  
Time: 0 seconds  
  
OK (2 tests, 2 assertions)phpunit ArrayTest  
PHPUnit 3.5.13 by Sebastian Bergmann.  
  
..  
  
Time: 0 seconds  
  
OK (2 tests, 2 assertions)
```

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 10, *Incomplete and Skipped Tests*).
- I Printed when the test is marked as being incomplete or not yet implemented (see Chapter 10, *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.

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

```
PHPUnit 3.5.13 by Sebastian Bergmann.  
  
Usage: phpunit [switches] UnitTest [UnitTest.php]  
       phpunit [switches] <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-dbus                    Log test execution to DBUS.  
--log-json <file>           Log test execution in JSON format.  
  
--coverage-html <dir>        Generate code coverage report in HTML format.  
--coverage-clover <file>     Write code coverage data in Clover XML format.  
  
--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.
```

```

--group ...                Only runs tests from the specified group(s).
--exclude-group ...        Exclude tests from the specified group(s).
--list-groups              List available test groups.

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

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

--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                   Mark a test as incomplete if no assertions are made.
--verbose                  Output more verbose information.
--wait                     Waits for a keystroke after each test.

--skeleton-class           Generate Unit class for UnitTest in UnitTest.php.
--skeleton-test            Generate UnitTest class for Unit in Unit.php.

--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.
--syntax-check             Try to check source files for syntax errors.

--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.

--help                     Prints this usage information.
--version                  Prints the version and exits.

--debug                    Output debugging information.phpunit --help
PHPUnit 3.5.13 by Sebastian Bergmann.

Usage: phpunit [switches] UnitTest [UnitTest.php]
       phpunit [switches] <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-dbus                  Log test execution to DBUS.
--log-json <file>         Log test execution in JSON format.

--coverage-html <dir>      Generate code coverage report in HTML format.
--coverage-clover <file>  Write code coverage data in Clover XML format.

--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.
--group ...                Only runs tests from the specified group(s).
--exclude-group ...        Exclude tests from the specified group(s).
--list-groups              List available test groups.

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

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

```

```

--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               Mark a test as incomplete if no assertions are made.
--verbose               Output more verbose information.
--wait                 Waits for a keystroke after each test.

--skeleton-class        Generate Unit class for UnitTest in UnitTest.php.
--skeleton-test         Generate UnitTest class for Unit in Unit.php.

--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.
--syntax-check         Try to check source files for syntax errors.

--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.

--help                 Prints this usage information.
--version              Prints the version and exits.

--debug                Output debugging information.

```

`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 an `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 19, *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 19, *Logging* for more details.

`--log-dbus` Log test execution using DBUS [<http://www.freedesktop.org/wiki/Software/dbus>].

`--log-json` Generates a logfile using the JSON [<http://www.json.org/>] format. See Chapter 19, *Logging* for more details.

`--coverage-html` Generates a code coverage report in HTML format. See Chapter 15, *Code Coverage Analysis* for more details.

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

<code>--coverage-clover</code>	Generates a logfile in XML format with the code coverage information for the tests run. See Chapter 19, <i>Logging</i> for more details. Please note that this functionality is only available when the tokenizer and Xdebug extensions are installed.
<code>--testdox-html</code> and <code>--testdox-text</code>	Generates agile documentation in HTML or plain text format for the tests that are run. See Chapter 16, <i>Other Uses for Tests</i> for more details.
<code>--filter</code>	Only runs tests whose name matches the given pattern. The pattern can be either the name of a single test or a regular expression [http://www.php.net/pcre] that matches multiple test names.
<code>--group</code>	Only runs tests from the specified group(s). A test can be tagged as belonging to a group using the <code>@group</code> annotation. The <code>@author</code> annotation is an alias for <code>@group</code> allowing to filter tests based on their authors.
<code>--exclude-group</code>	Exclude tests from the specified group(s). A test can be tagged as belonging to a group using the <code>@group</code> annotation.
<code>--list-groups</code>	List available test groups.
<code>--loader</code>	Specifies the <code>PHPUnit_Runner_TestSuiteLoader</code> 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 <code>include_path</code> configuration directive. Following the PEAR Naming Conventions, a class name such as <code>Project_Package_Class</code> is mapped to the sourcefile name <code>Project/Package/Class.php</code> .
<code>--repeat</code>	Repeatedly runs the test(s) the specified number of times.
<code>--tap</code>	Reports the test progress using the Test Anything Protocol (TAP) [http://testanything.org/]. See Chapter 19, <i>Logging</i> for more details.
<code>--testdox</code>	Reports the test progress as agile documentation. See Chapter 16, <i>Other Uses for Tests</i> for more details.
<code>--colors</code>	Use colors in output.
<code>--stderr</code>	Optionally print to <code>STDERR</code> instead of <code>STDOUT</code> .
<code>--stop-on-error</code>	Stop execution upon first error.
<code>--stop-on-failure</code>	Stop execution upon first error or failure.
<code>--stop-on-skipped</code>	Stop execution upon first skipped test.
<code>--stop-on-incomplete</code>	Stop execution upon first incomplete test.
<code>--strict</code>	Mark a test as incomplete if no assertions are made.
<code>--verbose</code>	Output more verbose information, for instance the names of tests that were incomplete or have been skipped.

<code>--wait</code>	Waits for a keystroke after each test. This is useful if you are running the tests in a window that stays open only as long as the test runner is active.
<code>--skeleton-class</code>	Generates a skeleton class <code>Unit</code> (in <code>Unit.php</code>) from a test case class <code>UnitTest</code> (in <code>UnitTest.php</code>). See Chapter 17, <i>Skeleton Generator</i> for more details.
<code>--skeleton-test</code>	Generates a skeleton test case class <code>UnitTest</code> (in <code>UnitTest.php</code>) for a class <code>Unit</code> (in <code>Unit.php</code>). See Chapter 17, <i>Skeleton Generator</i> for more details.
<code>--process-isolation</code>	Run each test in a separate PHP process.
<code>--no-globals-backup</code>	Do not backup and restore <code>\$GLOBALS</code> . See the section called “Global State” for more details.
<code>--static-backup</code>	Backup and restore static attributes of user-defined classes. See the section called “Global State” for more details.
<code>--syntax-check</code>	Enables the syntax check of test source files.
<code>--bootstrap</code>	A "bootstrap" PHP file that is run before the tests.
<code>--configuration, -c</code>	Read configuration from XML file. See Appendix C, <i>The XML Configuration File</i> for more details. If <code>phpunit.xml</code> or <code>phpunit.xml.dist</code> (in that order) exist in the current working directory and <code>--configuration</code> is <i>not</i> used, the configuration will be automatically read from that file.
<code>--no-configuration</code>	Ignore <code>phpunit.xml</code> and <code>phpunit.xml.dist</code> from the current working directory.
<code>--include-path</code>	Prepend PHP's <code>include_path</code> with given path(s).
<code>-d</code>	Sets the value of the given PHP configuration option.
<code>--debug</code>	Output debug information such as the name of a test when its execution starts.

Note

When the tested code contains PHP syntax errors, the TextUI test runner might exit without printing error information. The standard test suite loader can optionally check the test suite sourcefile for PHP syntax errors, but not sourcefiles included by the test suite sourcefile.

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 `$fixture` 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 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()
    {
        print __METHOD__ . "\n";
    }

    protected function setUp()
    {
        print __METHOD__ . "\n";
    }

    protected function assertPreConditions()
    {
        print __METHOD__ . "\n";
    }

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

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

    protected function assertPostConditions()
    {
        print __METHOD__ . "\n";
    }

    protected function tearDown()
    {
        print __METHOD__ . "\n";
    }

    public static function tearDownAfterClass()
    {
        print __METHOD__ . "\n";
    }

    protected function onNotSuccessfulTest(Exception $e)
    {
        print __METHOD__ . "\n";
        throw $e;
    }
}
```

```
}  
?>
```

```
PHPUnit 3.5.13 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  
  
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.phpunit TemplateMethodsTest  
PHPUnit 3.5.13 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  
  
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 11, *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-dependency-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 static attributes of classes requires PHP 5.3 (or greater).

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 Object_Freezer [<http://github.com/sebastianbergmann/php-object-freezer/>] 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 Object directory:

```
Object
|-- Freezer
|   |-- HashGenerator
|   |   |-- NonRecursiveSHA1.php
|   |   |-- HashGenerator.php
|   |   |-- IdGenerator
|   |   |   |-- UUID.php
|   |   |   |-- IdGenerator.php
|   |   |-- LazyProxy.php
|   |   |-- Storage
|   |   |   |-- CouchDB.php
|   |   |-- Storage.php
|   |   |-- Util.php
|   |-- Freezer.php
Tests
|-- Freezer
|   |-- HashGenerator
|   |   |-- NonRecursiveSHA1Test.php
|   |   |-- IdGenerator
|   |   |   |-- UUIDTest.php
|   |   |-- Storage
|   |   |   |-- CouchDB
|   |   |   |   |-- WithLazyLoadTest.php
|   |   |   |   |-- WithoutLazyLoadTest.php
|   |   |-- StorageTest.php
|   |   |-- UtilTest.php
|   |-- FreezerTest.php
```

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

```
PHPUnit 3.5.13 by Sebastian Bergmann.

..... 60 / 75
.....

Time: 0 seconds

OK (75 tests, 164 assertions)phpunit Tests
PHPUnit 3.5.13 by Sebastian Bergmann.

..... 60 / 75
.....

Time: 0 seconds

OK (75 tests, 164 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 `Object_FreezerTest` test case class in `Tests/FreezerTest.php` we can use the following command:

```
PHPUnit 3.5.13 by Sebastian Bergmann.
.....

Time: 0 seconds

OK (28 tests, 60 assertions)phpunit Tests/FreezerTest
PHPUnit 3.5.13 by Sebastian Bergmann.
.....

Time: 0 seconds

OK (28 tests, 60 assertions)
```

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

```
PHPUnit 3.5.13 by Sebastian Bergmann.
.

Time: 0 seconds

OK (1 test, 2 assertions)phpunit --filter testFreezingAnObjectWorks Tests
PHPUnit 3.5.13 by Sebastian Bergmann.
.

Time: 0 seconds

OK (1 test, 2 assertions)
```

Note

A drawback of this approach is that we have no control over the order in which the test 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` is recursively traversed.

Example 7.1. Composing a Test Suite Using XML Configuration

```
<phpunit>
```

```
<testsuites>
  <testsuite name="Object_Freezer">
    <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="Object_Freezer">
      <file>Tests/Freezer/HashGenerator/NonRecursiveSHA1Test.php</file>
      <file>Tests/Freezer/IdGenerator/UUIDTest.php</file>
      <file>Tests/Freezer/UtilTest.php</file>
      <file>Tests/FreezerTest.php</file>
      <file>Tests/Freezer/StorageTest.php</file>
      <file>Tests/Freezer/Storage/CouchDB/WithLazyLoadTest.php</file>
      <file>Tests/Freezer/Storage/CouchDB/WithoutLazyLoadTest.php</file>
    </testsuite>
  </testsuites>
</phpunit>
```

Chapter 8. TestCase Extensions

PHPUnit provides extensions to the standard base-class for test classes, `PHPUnit_Framework_TestCase`.

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_Extensions_OutputTestCase` 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 8.1, “Using `PHPUnit_Extensions_OutputTestCase`” shows how to subclass `PHPUnit_Extensions_OutputTestCase` and use its `expectOutputString()` method to set the expected output. If this expected output is not generated, the test will be counted as a failure.

Example 8.1. Using `PHPUnit_Extensions_OutputTestCase`

```
<?php
require_once 'PHPUnit/Extensions/OutputTestCase.php';

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

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

```
PHPUnit 3.5.13 by Sebastian Bergmann.

.F

Time: 0 seconds

There was 1 failure:

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

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.phpunit OutputTest
PHPUnit 3.5.13 by Sebastian Bergmann.

.F
```

```

Time: 0 seconds

There was 1 failure:

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

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

Table 8.1, “OutputTestCase” shows the methods provided by PHPUnit_Extensions_OutputTestCase.

Table 8.1. OutputTestCase

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

There are two other extensions to PHPUnit_Framework_TestCase, PHPUnit_Extensions_Database_TestCase and PHPUnit_Extensions_SeleniumTestCase, that are covered in Chapter 9, *Database Testing* and Chapter 18, *PHPUnit and Selenium*, respectively.

Chapter 9. Database Testing

While creating tests for your software you may come across database code that needs to be unit tested. The database extension has been created to provide an easy way to place your database in a known state, execute your database-affecting code, and ensure that the expected data is found in the database.

The quickest way to create a new Database Unit Test is to extend the `PHPUnit_Extensions_Database_TestCase` class. This class provides the functionality to create a database connection, seed your database with data, and after executing a test comparing the contents of your database with a data set that can be built in a variety of formats. In Example 9.1, “Setting up a database test case” you can see examples of `getConnection()` and `getDataSet()` implementations.

Example 9.1. Setting up a database test case

```
<?php
require_once 'PHPUnit/Extensions/Database/TestCase.php';

class DatabaseTest extends PHPUnit_Extensions_Database_TestCase
{
    protected function getConnection()
    {
        $pdo = new PDO('mysql:host=localhost;dbname=testdb', 'root', '');
        return $this->createDefaultDBConnection($pdo, 'testdb');
    }

    protected function getDataSet()
    {
        return $this->createFlatXMLDataSet(dirname(__FILE__).'/_files/bank-account-seed.xml');
    }
}
?>
```

The `getConnection()` method must return an implementation of the `PHPUnit_Extensions_Database_DB_IDatabaseConnection` interface. The `createDefaultDBConnection()` method can be used to return a database connection. It accepts a PDO object as the first parameter and the name of the schema you are testing against as the second parameter.

The `getDataSet()` method must return an implementation of the `PHPUnit_Extensions_Database_DataSet_IDataSet` interface. There are currently three different data sets available in PHPUnit. These data sets are discussed in the section called “Data Sets”

Table 9.1. Database Test Case Methods

Method	Meaning
<code>PHPUnit_Extensions_Database_DB_IDatabaseConnection</code> <code>getConnection()</code>	Implement to return the database connection that will be checked for expected data sets and tables.
<code>PHPUnit_Extensions_Database_DataSet_IDataSet</code> <code>getDataSet()</code>	Implement to return the data set that will be used in in database set up and tear down operations.
<code>PHPUnit_Extensions_Database_Operation_DatabaseOperation</code> <code>getSetUpOperation()</code>	Override to return a specific operation that should be performed on the test database at the beginning of each test. The various operations are detailed in the section called “Operations”.
<code>PHPUnit_Extensions_Database_Operation_DatabaseOperation</code> <code>getTearDownOperation()</code>	Override to return a specific operation that should be performed on the test database at the

Method	Meaning
	end of each test. The various operations are detailed in the section called “Operations”.
PHPUnit_Extensions_Database_DB_DefaultDatabaseConnection createDefaultDBConnection(PDO \$connection, string \$schema)	Return a database connection wrapper around the \$connection PDO object. The database schema being tested against should be specified by \$schema. The result of this method can be returned from getConnection().
PHPUnit_Extensions_Database_DataSet_FlatXmlDataSet createFlatXMLDataSet(string \$xmlFile)	Returns a flat XML data set that is created from the XML file located at the absolute path specified in \$xmlFile. More details about flat XML files can be found in the section called “Flat XML Data Set”. The result of this method can be returned from getDataSet().
PHPUnit_Extensions_Database_DataSet_XmlDataSet createXMLDataSet(string \$xmlFile)	Returns a XML data set that is created from the XML file located at the absolute path specified in \$xmlFile. More details about XML files can be found in the section called “XML Data Set”. The result of this method can be returned from getDataSet().
void assertTablesEqual(PHPUnit_Extensions_Database_DataSet_ITable \$expected, PHPUnit_Extensions_Database_DataSet_ITable \$actual)	Reports an error if the contents of the \$expected table do not match the contents in the \$actual table.
void assertDataSetsEqual(PHPUnit_Extensions_Database_DataSet_IDataSet \$expected, PHPUnit_Extensions_Database_DataSet_IDataSet \$actual)	Reports an error if the contents of the \$expected data set do not match the contents in the \$actual data set.

Data Sets

Data sets are the basic building blocks for both your database fixture as well as the assertions you may make at the end of your test. When returning a data set as a fixture from the `PHPUnit_Extensions_Database_TestCase::getDataSet()` method, the default implementation in PHPUnit will automatically truncate all tables specified and then insert the data from your data set in the order specified by the data set. For your convenience there are several different types of data sets that can be used at your convenience.

Flat XML Data Set

The flat XML data set is a very simple XML format that uses a single XML element for each row in your data set. An example of a flat XML data set is shown in Example 9.2, “A Flat XML Data Set”.

Example 9.2. A Flat XML Data Set

```
<?xml version="1.0" encoding="UTF-8" ?>
<dataset>
  <post
    post_id="1"
    title="My First Post"
    date_created="2008-12-01 12:30:29"
    contents="This is my first post" rating="5"
```

```

/>
<post
  post_id="2"
  title="My Second Post"
  date_created="2008-12-04 15:35:25"
  contents="This is my second post"
/>
<post
  post_id="3"
  title="My Third Post"
  date_created="2008-12-09 03:17:05"
  contents="This is my third post"
  rating="3"
/>

<post_comment
  post_comment_id="2"
  post_id="2"
  author="Frank"
  content="That is because this is simply an example."
  url="http://phpun.it/"
/>
<post_comment
  post_comment_id="1"
  post_id="2"
  author="Tom"
  content="While the topic seemed interesting the content was lacking."
/>

<current_visitors />
</dataset>

```

As you can see the formatting is extremely simple. Each of the elements within the root `<dataset>` element represents a row in the test database with the exception of the last `<current_visitors />` element (this will be discussed shortly.) The name of the element is the equivalent of a table name in your database. The name of each attribute is the equivalent of a column name in your databases. The value of each attribute is the equivalent of the value of that column in that row.

This format, while simple, does have some special considerations. The first of these is how you deal with empty tables. With the default operation of `CLEAN_INSERT` you can specify that you want to truncate a table and not insert any values by listing that table as an element without any attributes. This can be seen in Example 9.2, “A Flat XML Data Set” with the `<current_visitors />` element. The most common reason you would want to ensure an empty table as a part of your fixture is when your test should be inserting data to that table. The less data your database has, the quicker your tests will run. So if I were testing my simple blogging software's ability to add comments to a post, I would likely change Example 9.2, “A Flat XML Data Set” to specify `post_comment` as an empty table. When dealing with assertions it is often useful to ensure that a table is not being unexpectedly written to, which could also be accomplished in FlatXML using the empty table format.

The second consideration is how `NULL` values are defined. The nature of the flat XML format only allows you to explicitly specify strings for column values. Of course your database will convert a string representation of a number or date into the appropriate data type. However, there are no string representations of `NULL`. You can imply a `NULL` value by leaving a column out of your element's attribute list. This will cause a `NULL` value to be inserted into the database for that column. This leads me right into my next consideration that makes implicit `NULL` values somewhat difficult to deal with.

The third consideration is how columns are defined. The column list for a given table is determined by the attributes in the first element for any given table. In Example 9.2, “A Flat XML Data Set” the `post` table would be considered to have the columns `post_id`, `title`, `date_created`, `contents` and `rating`. If the first `<post>` were removed then the `post` would no longer be considered to have the `rating` column. This means that the first element of a given name defines the structure of that table. In the simplest of examples, this means that your first defined row must have a value for

every column that you expect to have values for in the rest of rows for that table. If an element further into your data set specifies a column that was not specified in the first element then that value will be ignored. You can see how this influenced the order of elements in my dataset in Example 9.2, “A Flat XML Data Set”. I had to specify the second `<post_comment>` element first due to the non-NULL value in the `url` column.

There is a way to work around the inability to explicitly set NULL in a flat XML data using the Replacement data set type. This will be discussed further in the section called “Replacement Data Set”.

XML Data Set

While the flat XML data set is simple it also proves to be limiting. A more powerful xml alternative is the XML data set. It is a more structured xml format that allows you to be much more explicit with your data set. An example of the XML data set equivalent to the previous flat XML example is shown in Example 9.3, “A XML Data Set”.

Example 9.3. A XML Data Set

```
<?xml version="1.0" encoding="UTF-8" ?>
<dataset>
  <table name="post">
    <column>post_id</column>
    <column>title</column>
    <column>date_created</column>
    <column>contents</column>
    <column>rating</column>
    <row>
      <value>1</value>
      <value>My First Post</value>
      <value>2008-12-01 12:30:29</value>
      <value>This is my first post</value>
      <value>5</value>
    </row>
    <row>
      <value>2</value>
      <value>My Second Post</value>
      <value>2008-12-04 15:35:25</value>
      <value>This is my second post</value>
      <null />
    </row>
    <row>
      <value>3</value>
      <value>My Third Post</value>
      <value>2008-12-09 03:17:05</value>
      <value>This is my third post</value>
      <value>3</value>
    </row>
  </table>
  <table name="post_comment">
    <column>post_comment_id</column>
    <column>post_id</column>
    <column>author</column>
    <column>content</column>
    <column>url</column>
    <row>
      <value>1</value>
      <value>2</value>
      <value>Tom</value>
      <value>While the topic seemed interesting the content was lacking.</value>
      <null />
    </row>
  </table>
</dataset>
```

```

    <value>2</value>
    <value>2</value>
    <value>Frank</value>
    <value>That is because this is simply an example.</value>
    <value>http://phpun.it</value>
  </row>
</table>
<table name="current_visitors">
  <column>current_visitors_id</column>
  <column>ip</column>
</table>
</dataset>

```

The formatting is more verbose than that of the Flat XML data set but in some ways much easier to understand. The root element is again the `<dataset>` element. Then directly under that element you will have one or more `<table>` elements. Each `<table>` element must have a name attribute with a value equivalent to the name of the table being represented. The `<table>` element will then include one or more `<column>` elements containing the name of a column in that table. The `<table>` element will also include any number (including zero) of `<row>` elements. The `<row>` element will be what ultimately specifies the data to store/remove/update in the database or to check the current database against. The `<row>` element must have the same number of children as there are `<column>` elements in that `<table>` element. The order of your child elements is also determined by the order of your `<column>` elements for that table. There are two different elements that can be used as children of the `<row>` element. You may use the `<value>` element to specify the value of that column in much the same way you can use attributes in the flat XML data set. The content of the `<value>` element will be considered the content of that column for that row. You may also use the `<null>` element to explicitly indicate that the column is to be given a NULL value. The `<null>` element does not contain any attributes or children.

You can use the DTD in Example 9.4, “The XML Data Set DTD” to validate your XML data set files. A reference of the valid elements in an XML data set can be found in Table 9.2, “XML Data Set Element Description”

Example 9.4. The XML Data Set DTD

```

<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT dataset (table+) | ANY>
<!ELEMENT table (column*, row*)>
<!ATTLIST table
  name CDATA #REQUIRED
>
<!ELEMENT column (#PCDATA)>
<!ELEMENT row (value | null | none)*>
<!ELEMENT value (#PCDATA)>
<!ELEMENT null EMPTY>

```

Table 9.2. XML Data Set Element Description

Element	Purpose	Contents	Attributes
<code><dataset></code>	The root element of the xml data set file.	One or more <code><table></code> elements.	None
<code><table></code>	Defines a table in the dataset.	One or more <code><column></code> elements and zero or more <code><row></code> elements.	name - The name of the table.
<code><column></code>	Defines a column in the current table.	A text node containing the name of the column.	None

Element	Purpose	Contents	Attributes
<row>	Defines a row in the table.	One or more <value> or <null> elements.	None
<value>	Sets the value of the column in the same position as this value.	A text node containing the value of the corresponding column.	None
<null>	Sets the value of the column in the same position of this value to NULL.	None	None

CSV Data Set

While XML data sets provide a convenient way to structure your data, in many cases they can be time consuming to hand edit as there are not very many XML editors that provide an easy way to edit tabular data via xml. In these cases you may find the CSV data set to be much more useful. The CSV data set is very simple to understand. Each CSV file represents a table. The first row in the CSV file must contain the column names and all subsequent rows will contain the data for those columns.

To construct a CSV data set you must instantiate the `PHPUnit_Extensions_Database_DataSet_CsvDataSet` class. The constructor takes three parameters: `$delimiter`, `$enclosure` and `$escape`. These parameters allow you to specify the exact formatting of rows within your CSV file. So this of course means it doesn't have to really be a CSV file. You can also provide a tab delimited file. The default constructor will specify a comma delimited file with fields enclosed by a double quote. If there is a double quote within a value it will be escaped by an additional double quote. This is as close to an accepted standard for CSV as there is.

Once your CSV data set class is instantiated you can use the method `addTable()` to add CSV files as tables to your data set. The `addTable` method takes two parameters. The first is `$tableName` and contains the name of the table you are adding. The second is `$csvFile` and contains the path to the CSV you will be using to set the data for that table. You can call `addTable()` for each table you would like to add to your data set.

In Example 9.5, "CSV Data Set Example" you can see an example of how three CSV files can be combined into the database fixture for a database test case.

Example 9.5. CSV Data Set Example

```

--- fixture/post.csv ---
post_id,title,date_created,contents,rating
1,My First Post,2008-12-01 12:30:29,This is my first post,5
2,My Second Post,2008-12-04 15:35:25,This is my second post,
3,My Third Post,2008-12-09 03:17:05,This is my third post,3

--- fixture/post_comment.csv ---
post_comment_id,post_id,author,content,url
1,2,Tom,While the topic seemed interesting the content was lacking.,
2,2,Frank,That is because this is simply an example.,http://phpunit.it

--- fixture/current_visitors.csv ---
current_visitors_id,ip

--- DatabaseTest.php ---
<?php
require_once 'PHPUnit/Extensions/Database/TestCase.php';
require_once 'PHPUnit/Extensions/Database/DataSet/CsvDataSet.php';

```

```
class DatabaseTest extends PHPUnit_Extensions_Database_TestCase
{
    protected function getConnection()
    {
        $pdo = new PDO('mysql:host=localhost;dbname=testdb', 'root', '');
        return $this->createDefaultDBConnection($pdo, 'testdb');
    }

    protected function getDataSet()
    {
        $dataSet = new PHPUnit_Extensions_Database_DataSet_CsvDataSet();
        $dataSet->addTable('post', 'fixture/post.csv');
        $dataSet->addTable('post_comment', 'fixture/post_comment.csv');
        $dataSet->addTable('current_visitors', 'fixture/current_visitors.csv');
        return $dataSet;
    }
}
?>
```

Unfortunately, while the CSV dataset is appealing from the aspect of edit-ability, it has the same problems with NULL values as the flat XML dataset. There is no native way to explicitly specify a null value. If you do not specify a value (as I have done with some of the fields above) then the value will actually be set to that data type's equivalent of an empty string. Which in most cases will not be what you want. I will address this shortcoming of both the CSV and the flat XML data sets shortly.

Replacement Data Set

...

Operations

...

Database Testing Best Practices

...

Chapter 10. 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 10.1, “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 10.1. 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 3.5.13 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.phpunit --verbose SampleTest
PHPUnit 3.5.13 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 10.1, “API for Incomplete Tests” shows the API for marking tests as incomplete.

Table 10.1. API for Incomplete Tests

Method	Meaning
<code>void markTestIncomplete()</code>	Marks the current test as incomplete.
<code>void markTestIncomplete(string \$message)</code>	Marks the current test as incomplete using <code>\$message</code> 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 10.2, “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 10.2. 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 3.5.13 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.phpunit --verbose DatabaseTest
PHPUnit 3.5.13 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 10.2, “API for Skipping Tests” shows the API for skipping tests.

Table 10.2. API for Skipping Tests

Method	Meaning
<code>void markTestSkipped()</code>	Marks the current test as skipped.
<code>void markTestSkipped(string \$message)</code>	Marks the current test as skipped using <code>\$message</code> as an explanatory message.

Chapter 11. 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.

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 11.2, "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 11.1, "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 11.1. The class we want to stub

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

```
?>
```

Example 11.2. 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.
- 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 11.3, "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.
- `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).

- `getMockClassName($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 11.3. 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 11.4, “Stubbing a method call to return one of the arguments” shows how you can achieve this using `returnArgument()` instead of `returnValue()`.

Example 11.4. 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 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 11.5, “Stubbing a method call to return a value from a callback” for an example.

Example 11.5. 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 11.6, “Stubbing a method call to return a list of values in the specified order” for an example.

Example 11.6. 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 11.7, “Stubbing a method call to throw an exception” shows how to use `throwException()` to do this.

Example 11.7. 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".

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 11.8, “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 11.8. The Subject and Observer classes that are part of the System under Test (SUT)

```
<?php
class Subject
{
    protected $observers = array();

    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 11.9, “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 11.9. 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;
        $subject->attach($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 parameters to the method being mocked. You can specify more advanced constraints on the method argument than a simple match.

Example 11.10. 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;
        $subject->attach($observer);

        // The doSomethingBad() method should report an error to the observer
        // via the reportError() method
    }
}
```

```

        $subject->doSomethingBad();
    }
}
?>

```

Table 4.2, “Constraints” shows the constraints that can be applied to method arguments and Table 11.1, “Matchers” shows the matchers that are available to specify the number of invocations.

Table 11.1. 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.

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 11.11. 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());
    }
}
?>

```

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 `getMockFromWsd1()` can be used just like `getMock()` (see above). The only difference is that `getMockFromWsd1()` 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 11.12, “Stubbing a web service” shows how `getMockFromWsd1()` can be used to stub, for example, the web service described in `GoogleSearch.wsdl`.

Example 11.12. Stubbing a web service

```
<?php
class GoogleTest extends PHPUnit_Framework_TestCase
{
    public function testSearch()
    {
        $googleSearch = $this->getMockFromWsd1(
            '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` [<http://code.google.com/p/bovigo/wiki/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.

To install `vfsStream`, the PEAR channel (`pear.php-tools.net`) that is used for its distribution needs to be registered with the local PEAR environment:

```
pear channel-discover pear.php-tools.net
```

This has to be done only once. Now the PEAR Installer can be used to install `vfsStream`:

```
pear install pat/vfsStream-alpha
```

Example 11.13, “A class that interacts with the filesystem” shows a class that interacts with the filesystem.

Example 11.13. 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 11.14, “Testing a class that interacts with the filesystem”).

Example 11.14. 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 11.15, “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 11.15. 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 12. 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 13. 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 13.1, "Tests for the `BankAccount` class".

Example 13.1. 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 13.2, “Code needed for the `testBalanceIsInitiallyZero()` test to pass”.

Example 13.2. 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 3.5.13 by Sebastian Bergmann.

```

```

.
Fatal error: Call to undefined method BankAccount::withdrawMoney()phpunit BankAccountTes
PHPUnit 3.5.13 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 13.3, “The complete `BankAccount` class”. These methods are written in a such a way that they raise a `BankAccountException` when they are called with illegal values that would violate the contract conditions.

Example 13.3. 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 3.5.13 by Sebastian Bergmann.  
  
...  
Time: 0 seconds  
  
OK (3 tests, 3 assertions)phpunit BankAccountTest  
PHPUnit 3.5.13 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 13.4, “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 13.4. 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 14. 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 13, *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.

The `PHPUnit_Extensions_Story_TestCase` class adds a story framework that facilitates the definition of a Domain-Specific Language [http://en.wikipedia.org/wiki/Domain-specific_programming_language] for Behaviour-Driven Development. 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)`
- ...


```

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 3.5.13 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
    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
      and Player rolls 10
    Then Score should be 300
```

```
Scenarios: 5, Failed: 0, Skipped: 0, Incomplete: 0.phpunit --story BowlingGameSpec
PHPUnit 3.5.13 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
    Then Score should be 20
```

```
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
    and Player rolls 10
  Then Score should be 300
```

Scenarios: 5, Failed: 0, Skipped: 0, Incomplete: 0.

Chapter 15. 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.

Let us generate a code coverage report for the `BankAccount` class from Example 13.3, “The complete `BankAccount` class”.

```
PHPUnit 3.5.13 by Sebastian Bergmann.  
  
...  
  
Time: 0 seconds  
  
OK (3 tests, 3 assertions)  
  
Generating report, this may take a moment.phpunit --coverage-html ./report BankAccountTe  
PHPUnit 3.5.13 by Sebastian Bergmann.  
  
...  
  
Time: 0 seconds  
  
OK (3 tests, 3 assertions)  
  
Generating report, this may take a moment.
```

Figure 15.1, “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 15.1. Code Coverage for setBalance()

```

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      2 :          if ($balance >= 0) {
92      0 :              $this->balance = $balance;
93      0 :          } else {
94      2 :              throw new BankAccountException;
95      :          }
96      0 :      }

```

Clicking on the line number of a covered line will open a panel (see Figure 15.2, “Panel with information on covering tests”) that shows the test cases that cover this line.

Figure 15.2. Panel with information on covering tests

```

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      2 :          if ($balance >= 0) {

```

2 tests cover line 91

- testBalanceCannotBecomeNegative(BankAccountTest)
- testBalanceCannotBecomeNegative2(BankAccountTest)

```

          $this->balance = $balance;
        else {
          throw new BankAccountException;
        }
      }

```

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 15.1, “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 15.1. 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 15.3, “Code Coverage for `setBalance()` with additional test” shows the code coverage of the `setBalance()` method with the additional test.

Figure 15.3. 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 15.2, “Tests that specify which method they want to cover” shows an example.

Example 15.2. 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());
}
}
?>

```

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 15.3, “Using the `@codeCoverageIgnore`, `@codeCoverageIgnoreStart` and `@codeCoverageIgnoreEnd` annotations”.

Example 15.3. 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 lines of code that are marked as to be 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. When whitelisting is used, each file on the whitelist is optionally added to the code coverage report regardless of whether or not it was executed.

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.

Alternatively, you can configure the sourcecode files that are included in the report using the `PHP_CodeCoverage_Filter` API.

Chapter 16. 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 13.1, "Tests for the `BankAccount` class"):

```
PHPUnit 3.5.13 by Sebastian Bergmann.  
  
BankAccount  
[x] Balance is initially zero  
[x] Balance cannot become negativephpunit --testdox BankAccountTest  
PHPUnit 3.5.13 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 17. Skeleton Generator

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. PHPUnit can help you by analyzing the code of the existing class and generating a skeleton test case class for it.

Example 17.1. 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 17.1, “The Calculator class”).

```
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
Wrote test class skeleton for Calculator to CalculatorTest.php.phpunit --skeleton-test C  
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
Wrote test class skeleton for Calculator to CalculatorTest.php.
```

For each method in the original class, there will be an incomplete test case (see Chapter 10, *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 Bar that is declared in the Foo namespace you need to invoke the skeleton generator like this:

```
phpunit --skeleton-test "Foo\Bar" Bar.php
```

Below is the output of running the generated test case class.

```
PHPUnit 3.5.13 by Sebastian Bergmann.
```

```
I
```

```
Time: 0 seconds, Memory: 4.00Mb
```

```

There was 1 incomplete test:

1) CalculatorTest::testAdd
This test has not been implemented yet.

/home/sb/CalculatorTest.php:41
OK, but incomplete or skipped tests!
Tests: 1, Assertions: 0, Incomplete: 1.phpunit --verbose CalculatorTest
PHPUnit 3.5.13 by Sebastian Bergmann.

I

Time: 0 seconds, Memory: 4.00Mb

There was 1 incomplete test:

1) CalculatorTest::testAdd
This test has not been implemented yet.

/home/sb/CalculatorTest.php:41
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 17.2, “The Calculator class with `@assert` annotations” shows an example.

Example 17.2. 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 3.5.13 by Sebastian Bergmann.

```

```

.....

Time: 0 seconds

OK (4 tests, 4 assertions)phpunit CalculatorTest
PHPUnit 3.5.13 by Sebastian Bergmann.

.....

Time: 0 seconds

OK (4 tests, 4 assertions)

```

Table 17.1, “Supported variations of the @assert annotation” shows the supported variations of the @assert annotation and how they are transformed into test code.

Table 17.1. 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 13, *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 17.4, “The generated `BowlingGame` class skeleton” which has been generated based on the analysis of Example 17.3, “The `BowlingGameTest` class”.

Example 17.3. 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());
    }
}
?>
```

Example 17.4. The generated BowlingGame class skeleton

```
<?php
/**
 * Generated by PHPUnit on 2008-03-10 at 17:18:33.
 */
class BowlingGame
{
    /**
     * @todo Implement roll().
     */
    public function roll()
    {
        // Remove the following line when you implement this method.
        throw new RuntimeException('Not yet implemented.');
```

Chapter 18. PHPUnit and Selenium

Selenium RC

Selenium RC [<http://seleniumhq.org/projects/remote-control/>] 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 uses Selenium Core [<http://seleniumhq.org/>], a library that performs automated browser tasks using JavaScript. 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).

Let us take a look at how Selenium RC is installed:

1. Download a distribution archive of Selenium RC [<http://seleniumhq.org/projects/remote-control/>].
2. Unzip the distribution archive and copy `server/selenium-server.jar` to `/usr/local/bin`, for instance.
3. Start the Selenium RC server by running `java -jar /usr/local/bin/selenium-server.jar`.

Now we can send commands to the Selenium RC server using its client/server protocol.

PHPUnit_Extensions_SeleniumTestCase

The `PHPUnit_Extensions_SeleniumTestCase` test case extension implements the client/server protocol to talk to Selenium RC as well as specialized assertion methods for web testing.

Example 18.1, “Usage example for `PHPUnit_Extensions_SeleniumTestCase`” shows how to test the contents of the `<title>` element of the `http://www.example.com/` website.

Example 18.1. 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 3.5.13 by Sebastian Bergmann.
```

```
F
```

```
Time: 5 seconds
```

```

There was 1 failure:

1) testTitle(WebTest)
Current URL: http://www.example.com/

Failed asserting that two strings are equal.
expected string <Example WWW Page>
difference      <      xx>
got string      <Example Web Page>
/home/sb/WebTest.php:30

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit WebTest
PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 5 seconds

There was 1 failure:

1) testTitle(WebTest)
Current URL: http://www.example.com/

Failed asserting that two strings are equal.
expected string <Example WWW Page>
difference      <      xx>
got string      <Example Web Page>
/home/sb/WebTest.php:30

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 RC session. See Table 18.1, “Selenium RC API: Setup” for the list of methods that are available for this.

Table 18.1. Selenium RC API: Setup

Method	Meaning
<code>void setBrowser(string \$browser)</code>	Set the browser to be used by the Selenium RC server.
<code>void setBrowserUrl(string \$browserUrl)</code>	Set the base URL for the tests.
<code>void setHost(string \$host)</code>	Set the hostname for the connection to the Selenium RC server.
<code>void setPort(int \$port)</code>	Set the port for the connection to the Selenium RC server.
<code>void setTimeout(int \$timeout)</code>	Set the timeout for the connection to the Selenium RC server.
<code>void setSleep(int \$seconds)</code>	Set the number of seconds the Selenium RC client should sleep between sending action commands to the Selenium RC 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 18.2, “Capturing a screenshot when a test fails”.

Example 18.2. 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 3.5.13 by Sebastian Bergmann.

F

Time: 5 seconds

There was 1 failure:

1) testTitle(WebTest)

Current URL: http://www.example.com/

Screenshot: http://localhost/screenshots/6c8e1e890fff864bd56db436cd0f309e.png

Failed asserting that two strings are equal.

expected string <Example WWW Page>

difference < xx>

got string <Example Web Page>

/home/sb/WebTest.php:30

FAILURES!

Tests: 1, Assertions: 1, Failures: 1.phpunit WebTest

PHPUnit 3.5.13 by Sebastian Bergmann.

F

Time: 5 seconds

There was 1 failure:

1) testTitle(WebTest)

Current URL: http://www.example.com/

Screenshot: http://localhost/screenshots/6c8e1e890fff864bd56db436cd0f309e.png

Failed asserting that two strings are equal.

expected string <Example WWW Page>

difference < xx>

got string <Example Web Page>

/home/sb/WebTest.php:30

```
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 RC servers. Example 18.3, “Setting up multiple browser configurations” shows an example.

Example 18.3. 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/SeleniumTestCase/phpunit_coverage.php into your webserver's document root directory.
2. In your webserver's php.ini configuration file, configure PHPUnit/Extensions/SeleniumTestCase/prepend.php and PHPUnit/Extensions/SeleniumTestCase/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 18.2, “Assertions” lists the various assertion methods that PHPUnit_Extensions_SeleniumTestCase provides.

Table 18.2. 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.
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 18.3, “Template Methods” shows the template method of `PHPUnit_Extensions_SeleniumTestCase`:

Table 18.3. Template Methods

Method	Meaning
<code>void defaultAssertions()</code>	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 RC server.

Please refer to the documentation of Selenium commands [http://seleniumhq.org/docs/04_selenese_commands.html] for a reference of the commands available and how they are used.

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 18.4, “Use a directory of Selenese/HTML files as tests” shows an example.

Example 18.4. 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';
}
?>
```

Chapter 19. 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/OptionalTasks/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 integer:2 matches expected value integer:1.

/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="3.5.0">
  <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>
```

Chapter 20. 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 20.1, “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 20.1. 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 20.2, “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 20.2. 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 $other Value or object to evaluate.
     * @return bool
     */
    public function evaluate($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 20.3, “A simple test listener” shows a simple implementation of the `PHPUnit_Framework_TestListener` interface.

Example 20.3. 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 two concrete test decorators: `PHPUnit_Extensions_RepeatedTest` and `PHPUnit_Extensions_TestSetup`. The former is used to run a test repeatedly and only count it as a success if all iterations are successful. The latter was discussed in Chapter 6, *Fixtures*.

Example 20.4, “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 20.4. 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 20.5, “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 20.5. 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();

            list($expected, $actual) = explode(';', $line);

            try {
                PHPUnit_Framework_Assert::assertEquals(

```

```

        trim($expected), trim($actual)
    );
}

catch (PHPUnit_Framework_AssertionFailedError $e) {
    $result->addFailure($this, $e, PHP_Timer::stop());
}

catch (Exception $e) {
    $result->addError($this, $e, PHP_Timer::stop());
}

$result->endTest($this, PHP_Timer::stop());
}

return $result;
}
}

$test = new DataDrivenTest('data_file.csv');
$result = PHPUnit_TextUI_TestRunner::run($test);
?>

```

PHPUnit 3.5.13 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
<code>assertArrayHasKey(\$key, array \$array, \$message = '')</code>
<code>assertArrayNotHasKey(\$key, array \$array, \$message = '')</code>
<code>assertAttributeContains(\$needle, \$haystackAttributeName, \$haystackClassOrObject, \$message = '', \$ignoreCase = FALSE)</code>
<code>assertAttributeContainsOnly(\$type, \$haystackAttributeName, \$haystackClassOrObject, \$isNativeType = NULL, \$message = '')</code>
<code>assertAttributeEmpty(\$haystackAttributeName, \$haystackClassOrObject, \$message = '')</code>
<code>assertAttributeEquals(\$expected, \$actualAttributeName, \$actualClassOrObject, \$message = '', \$delta = 0, \$maxDepth = 10, \$canonicalize = FALSE, \$ignoreCase = FALSE)</code>
<code>assertAttributeGreaterThan(\$expected, \$actualAttributeName, \$actualClassOrObject, \$message = '')</code>
<code>assertAttributeGreaterThanOrEqual(\$expected, \$actualAttributeName, \$actualClassOrObject, \$message = '')</code>
<code>assertAttributeInstanceOf(\$expected, \$attributeName, \$classOrObject, \$message = '')</code>
<code>assertAttributeInternalType(\$expected, \$attributeName, \$classOrObject, \$message = '')</code>
<code>assertAttributeLessThan(\$expected, \$actualAttributeName, \$actualClassOrObject, \$message = '')</code>
<code>assertAttributeLessThanOrEqual(\$expected, \$actualAttributeName, \$actualClassOrObject, \$message = '')</code>
<code>assertAttributeNotContains(\$needle, \$haystackAttributeName, \$haystackClassOrObject, \$message = '', \$ignoreCase = FALSE)</code>
<code>assertAttributeNotContainsOnly(\$type, \$haystackAttributeName, \$haystackClassOrObject, \$isNativeType = NULL, \$message = '')</code>
<code>assertAttributeNotEmpty(\$haystackAttributeName, \$haystackClassOrObject, \$message = '')</code>
<code>assertAttributeNotEquals(\$expected, \$actualAttributeName, \$actualClassOrObject, \$message = '', \$delta = 0, \$maxDepth = 10, \$canonicalize = FALSE, \$ignoreCase = FALSE)</code>
<code>assertAttributeNotInstanceOf(\$expected, \$attributeName, \$classOrObject, \$message = '')</code>
<code>assertAttributeNotInternalType(\$expected, \$attributeName, \$classOrObject, \$message = '')</code>
<code>assertAttributeNotSame(\$expected, \$actualAttributeName, \$actualClassOrObject, \$message = '')</code>
<code>assertAttributeNotType(\$expected, \$attributeName, \$classOrObject, \$message = '')</code>
<code>assertAttributeSame(\$expected, \$actualAttributeName, \$actualClassOrObject, \$message = '')</code>

Assertion
<code>assertAttributeType(\$expected, \$attributeName, \$classOrObject, \$message = '')</code>
<code>assertClassHasAttribute(\$attributeName, \$className, \$message = '')</code>
<code>assertClassHasStaticAttribute(\$attributeName, \$className, \$message = '')</code>
<code>assertClassNotHasAttribute(\$attributeName, \$className, \$message = '')</code>
<code>assertClassNotHasStaticAttribute(\$attributeName, \$className, \$message = '')</code>
<code>assertContains(\$needle, \$haystack, \$message = '', \$ignoreCase = FALSE)</code>
<code>assertContainsOnly(\$type, \$haystack, \$isNativeType = NULL, \$message = '')</code>
<code>assertEmpty(\$actual, \$message = '')</code>
<code>assertEqualXMLStructure(DOMNode \$expectedNode, DOMNode \$actualNode, \$checkAttributes = FALSE, \$message = '')</code>
<code>assertEquals(\$expected, \$actual, \$message = '', \$delta = 0, \$maxDepth = 10, \$canonicalize = FALSE, \$ignoreCase = FALSE)</code>
<code>assertFalse(\$condition, \$message = '')</code>
<code>assertFileEquals(\$expected, \$actual, \$message = '', \$canonicalize = FALSE, \$ignoreCase = FALSE)</code>
<code>assertFileExists(\$filename, \$message = '')</code>
<code>assertFileNotEquals(\$expected, \$actual, \$message = '', \$canonicalize = FALSE, \$ignoreCase = FALSE)</code>
<code>assertFileNotExists(\$filename, \$message = '')</code>
<code>assertGreaterThan(\$expected, \$actual, \$message = '')</code>
<code>assertGreaterThanOrEqual(\$expected, \$actual, \$message = '')</code>
<code>assertInstanceOf(\$expected, \$actual, \$message = '')</code>
<code>assertInternalType(\$expected, \$actual, \$message = '')</code>
<code>assertLessThan(\$expected, \$actual, \$message = '')</code>
<code>assertLessThanOrEqual(\$expected, \$actual, \$message = '')</code>
<code>assertNotContains(\$needle, \$haystack, \$message = '', \$ignoreCase = FALSE)</code>
<code>assertNotContainsOnly(\$type, \$haystack, \$isNativeType = NULL, \$message = '')</code>
<code>assertNotEmpty(\$actual, \$message = '')</code>
<code>assertNotEquals(\$expected, \$actual, \$message = '', \$delta = 0, \$maxDepth = 10, \$canonicalize = FALSE, \$ignoreCase = FALSE)</code>
<code>assertNotInstanceOf(\$expected, \$actual, \$message = '')</code>
<code>assertNotInternalType(\$expected, \$actual, \$message = '')</code>
<code>assertNotNull(\$actual, \$message = '')</code>
<code>assertNotRegExp(\$pattern, \$string, \$message = '')</code>
<code>assertNotSame(\$expected, \$actual, \$message = '')</code>
<code>assertNotTag(\$matcher, \$actual, \$message = '', \$isHtml = TRUE)</code>

Assertion
<code>assertNotType(\$expected, \$actual, \$message = '')</code>
<code>assertNull(\$actual, \$message = '')</code>
<code>assertObjectHasAttribute(\$attributeName, \$object, \$message = '')</code>
<code>assertObjectNotHasAttribute(\$attributeName, \$object, \$message = '')</code>
<code>assertRegExp(\$pattern, \$string, \$message = '')</code>
<code>assertSame(\$expected, \$actual, \$message = '')</code>
<code>assertSelectCount(\$selector, \$count, \$actual, \$message = '', \$isHtml = TRUE)</code>
<code>assertSelectEquals(\$selector, \$content, \$count, \$actual, \$message = '', \$isHtml = TRUE)</code>
<code>assertSelectRegExp(\$selector, \$pattern, \$count, \$actual, \$message = '', \$isHtml = TRUE)</code>
<code>assertStringEndsNotWith(\$suffix, \$string, \$message = '')</code>
<code>assertStringEndsWith(\$suffix, \$string, \$message = '')</code>
<code>assertStringEqualsFile(\$expectedFile, \$actualString, \$message = '', \$canonicalize = FALSE, \$ignoreCase = FALSE)</code>
<code>assertStringMatchesFormat(\$format, \$string, \$message = '')</code>
<code>assertStringMatchesFormatFile(\$formatFile, \$string, \$message = '')</code>
<code>assertStringNotEqualsFile(\$expectedFile, \$actualString, \$message = '', \$canonicalize = FALSE, \$ignoreCase = FALSE)</code>
<code>assertStringNotMatchesFormat(\$format, \$string, \$message = '')</code>
<code>assertStringNotMatchesFormatFile(\$formatFile, \$string, \$message = '')</code>
<code>assertStringStartsNotWith(\$prefix, \$string, \$message = '')</code>
<code>assertStringStartsWith(\$prefix, \$string, \$message = '')</code>
<code>assertTag(\$matcher, \$actual, \$message = '', \$isHtml = TRUE)</code>
<code>assertThat(\$value, PHPUnit_Framework_Constraint \$constraint, \$message = '')</code>
<code>assertTrue(\$condition, \$message = '')</code>
<code>assertType(\$expected, \$actual, \$message = '')</code>
<code>assertXmlFileEqualsXmlFile(\$expectedFile, \$actualFile, \$message = '')</code>
<code>assertXmlFileNotEqualsXmlFile(\$expectedFile, \$actualFile, \$message = '')</code>
<code>assertXmlStringEqualsXmlFile(\$expectedFile, \$actualXml, \$message = '')</code>
<code>assertXmlStringEqualsXmlString(\$expectedXml, \$actualXml, \$message = '')</code>
<code>assertXmlStringNotEqualsXmlFile(\$expectedFile, \$actualXml, \$message = '')</code>
<code>assertXmlStringNotEqualsXmlString(\$expectedXml, \$actualXml, \$message = '')</code>

Appendix B. Annotations

An annotation is a special form of syntactic metadata that can be added to the source code of some programming languages. While PHP has no dedicated language feature for annotating source code, the usage of tags such as `@annotation` arguments in documentation block has been established in the PHP community to annotate source code. In PHP documentation blocks are reflective: they can be accessed through the Reflection API's `getDocComment()` method on the function, class, method, and attribute level. Applications such as PHPUnit use this information at runtime to configure their behaviour.

This appendix shows all the varieties of annotations supported by PHPUnit.

@assert

You can use the `@assert` annotation in the documentation block of a method to automatically generate simple, yet meaningful tests instead of incomplete test cases when using the Skeleton Generator (see Chapter 17, *Skeleton Generator*):

```
/**
 * @assert (0, 0) == 0
 */
public function add($a, $b)
{
    return $a + $b;
}
```

These annotations 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));
}
```

@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()
    {
        // ...
    }
}

```

@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
<code>@covers ClassName::methodName</code>	Specifies that the annotated test method covers the specified method.
<code>@covers ClassName</code>	Specifies that the annotated test method covers all methods of a given class.
<code>@covers ClassName<extended></code>	Specifies that the annotated test method covers all methods of a given class and its parent class(es) and interface(s).
<code>@covers ClassName::<public></code>	Specifies that the annotated test method covers all public methods of a given class.
<code>@covers ClassName::<protected></code>	Specifies that the annotated test method covers all protected methods of a given class.
<code>@covers ClassName::<private></code>	Specifies that the annotated test method covers all private methods of a given class.
<code>@covers ClassName::<!public></code>	Specifies that the annotated test method covers all methods of a given class that are not public.
<code>@covers ClassName::<!protected></code>	Specifies that the annotated test method covers all methods of a given class that are not protected.
<code>@covers ClassName::<!private></code>	Specifies that the annotated test method covers all methods of a given class that are not private.

@dataProvider

A test method can accept arbitrary arguments. These arguments are to be provided by a data provider method (`provider()` in Example 4.4, “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 4.7, “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

...

@expectedExceptionMessage

...

@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 switches of the command-line test runner or using the respective directives of the XML configuration file.

@outputBuffering

The @outputBuffering annotation can be used to control PHP's output buffering [<http://www.php.net/manual/en/intro.outcontrol.php>] like this

```
/**
 * @outputBuffering enabled
 */
class MyTest extends PHPUnit_Framework_TestCase
{
    // ...
}
```

The @outputBuffering annotation can also be used on the test method level. This allows for fine-grained control over the output buffering:

```
/**
 * @outputBuffering disabled
 */
class MyTest extends PHPUnit_Framework_TestCase
{
    /**
     * @outputBuffering enabled
     */
    public function testThatPrintsSomething()
    {
        // ...
    }
}
```

@runTestsInSeparateProcesses

@runInSeparateProcess

@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 backupGlobals="true"
  backupStaticAttributes="false"
  <!--bootstrap="/path/to/bootstrap.php"-->
  colors="false"
  convertErrorsToExceptions="true"
  convertNoticesToExceptions="true"
  convertWarningsToExceptions="true"
  forceCoversAnnotation="false"
  mapTestClassNameToCoveredClassName="false"
  processIsolation="false"
  stopOnError="false"
  stopOnFailure="false"
  stopOnIncomplete="false"
  stopOnSkipped="false"
  syntaxCheck="false"
  testSuiteLoaderClass="PHPUnit_Runner_StandardTestSuiteLoader"
  <!--testSuiteLoaderFile="/path/to/StandardTestSuiteLoader.php"-->
  strict="false"
  verbose="false">
  <!-- ... -->
</phpunit>
```

The XML configuration above corresponds to the default behaviour of the TextUI test runner.

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>
  </testsuite>
</testsuites>
```

Groups

The `<groups>` element and its `<include>`, `<exclude>`, and `<group>` children can be used to select groups of tests from a suite of tests 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 switches:

- --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>
    <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"
    yui="true" highlight="false"
    lowUpperBound="35" highLowerBound="70"/>
  <log type="coverage-clover" target="/tmp/coverage.xml"/>
  <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 switches:

- --coverage-html /tmp/report
- --coverage-clover /tmp/coverage.xml
- --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`, `yui`, `highlight`, `lowUpperBound`, `highLowerBound`, and `logIncompleteSkipped` attributes have no equivalent TextUI test runner switch.

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

\$backupGlobalsBlacklist, 71
\$backupStaticAttributesBlacklist, 71
@assert, 121, 141
@author, , 141
@backupGlobals, 71, 141, 141
@backupStaticAttributes, 71, 142, 142
@codeCoverageIgnore, 116
@codeCoverageIgnoreEnd, 116
@codeCoverageIgnoreStart, 116
@covers, 115, 142
@dataProvider, 8, 11, 11, 143
@depends, 6, 11, 11, 143
@expectedException, 11, 144
@expectedExceptionCode, 144
@expectedExceptionMessage, 144
@group, , , 144
@outputBuffering, 144, 144
@runInSeparateProcess, 145
@runTestsInSeparateProcesses, 145
@test, , 145
@testdox, 145
@ticket, 145

A

Agile Documentation, , , 118
Annotation, 6, , 6, 8, 11, 11, 11, , , , 115, 116, 121, 141
anything(),
arrayHasKey(),
assertArrayHasKey(), 14,
assertArrayNotHasKey(), 14,
assertAttributeContains(), 17,
assertAttributeContainsOnly(), 18,
assertAttributeEmpty(), 19,
assertAttributeEquals(), 23,
assertAttributeGreaterThan(), 31,
assertAttributeGreaterThanOrEqual(), 32,
assertAttributeInstanceOf(), 33,
assertAttributeInternalType(), 34,
assertAttributeLessThan(), 35,
assertAttributeLessThanOrEqual(), 36,
assertAttributeNotContains(), 17,
assertAttributeNotContainsOnly(), 18,
assertAttributeNotEmpty(), 19,
assertAttributeNotEquals(), 23,
assertAttributeNotInstanceOf(), 33,
assertAttributeNotInternalType(), 34,
assertAttributeNotSame(), 41,
assertAttributeNotType(), 56,
assertAttributeSame(), 41,
assertAttributeType(), 56,

assertClassHasAttribute(), 15,
assertClassHasStaticAttribute(), 16,
assertClassNotHasAttribute(), 15,
assertClassNotHasStaticAttribute(), 16,
assertContains(), 17,
assertContainsOnly(), 18,
assertEmpty(), 19,
assertEquals(), 23,
assertEqualXMLStructure(), 20,
assertFalse(), 29,
assertFileEquals(), 29,
assertFileExists(), 31,
assertFileNotEquals(), 30,
assertFileNotExists(), 31,
assertGreaterThan(), 31,
assertGreaterThanOrEqual(), 32,
assertInstanceOf(), 33,
assertInternalType(), 34,
Assertion, 141
Assertions, 2
assertLessThan(), 35,
assertLessThanOrEqual(), 36,
assertNotContains(), 17,
assertNotContainsOnly(), 18,
assertNotEmpty(), 19,
assertNotEquals(), 23,
assertNotInstanceOf(), 33,
assertNotInternalType(), 34,
assertNotNull(), 36,
assertNotRegExp(), 38,
assertNotSame(), 41,
assertNotTag(), 51,
assertNotType(), 56,
assertNull(), 36,
assertObjectHasAttribute(), 37,
assertObjectNotHasAttribute(), 37,
assertPostConditions(), 68
assertPreConditions(), 68
assertRegExp(), 38,
assertSame(), 41,
assertSelectCount(), 43,
assertSelectEquals(), 45,
assertSelectRegExp(), 47,
assertStringEndsNotWith(), 48,
assertStringEndsWith(), 48,
assertStringEqualsFile(), 49,
assertStringMatchesFormat(), 39,
assertStringMatchesFormatFile(), 40,
assertStringNotEqualsFile(), 49,
assertStringNotMatchesFormat(), 39,
assertStringNotMatchesFormatFile(), 40,
assertStringStartsNotWith(), 50,
assertStringStartsWith(), 50,
assertTag(), 51,
assertThat(), 53,
assertTrue(), 55,
assertType(), 56,
assertXmlFileEqualsXmlFile(), 58,

assertXmlFileNotEqualsXmlFile(), 58,
assertXmlStringEqualsXmlFile(), 59,
assertXmlStringEqualsXmlString(), 60,
assertXmlStringNotEqualsXmlFile(), 59,
assertXmlStringNotEqualsXmlString(), 60,
attribute(),
attributeEqualTo(),
Automated Documentation, 118
Automated Test, 2

B

Behaviour-Driven Development, 107
Blacklist, 117, 147

C

classHasAttribute(),
classHasStaticAttribute(),
Code Coverage, , , 113, 117, 142, 147
Configuration, ,
Constant, 148
contains(),

D

Data-Driven Tests, 136
Database, 78
DbUnit, 78
DBUS,
Defect Localization, 7
Depended-On Component, 88
Design-by-Contract, 102
Documenting Assumptions, 118
Domain-Driven Design, 107

E

equalTo(),
Error, 62
Error Handler, 13
expects(), 89, 90, 90, 91, 91, 92
Extreme Programming, 102, 107, 118

F

Failure, 62
fileExists(),
Fixture, 67
Fluent Interface, 88

G

getMock(), 89, 90, 91, 91, 92
getMockBuilder(), 90
getMockForAbstractClass(), 95
getMockFromWsd(), 96
Global Variable, 71, 148
greaterThan(),
greaterThanOrEqual(),

H

hasAttribute(),

I

identicalTo(),
include_path,
Incomplete Test, 85, 120
isFalse(),
isInstanceOf(),
isNull(),
isTrue(),
isType(),

J

JSON,

L

lessThan(),
lessThanOrEqual(),
Logfile, ,
Logging, 130, 147
logicalAnd(),
logicalNot(),
logicalOr(),
logicalXor(),

M

matchesRegularExpression(),
method(), 89, 90, 90, 91, 91, 92
Mock Object, 92, 93

O

onConsecutiveCalls(), 91
onNotSuccessfulTest(), 68

P

PHP Error, 13
PHP Notice, 13
PHP Warning, 13
php.ini, 148
PHPUnit_Extensions_OutputTestCase, 76
PHPUnit_Extensions_RepeatedTest, 135
PHPUnit_Extensions_SeleniumTestCase, 124
PHPUnit_Extensions_Story_TestCase, 107
PHPUnit_Extensions_TestDecorator, 135
PHPUnit_Extensions_TestSetup, 135
PHPUnit_Framework_Assert, 105
PHPUnit_Framework_Error, 13
PHPUnit_Framework_Error_Notice, 14
PHPUnit_Framework_Error_Warning, 14
PHPUnit_Framework_IncompleteTest, 85
PHPUnit_Framework_IncompleteTestError, 85
PHPUnit_Framework_Test, 136
PHPUnit_Framework_TestCase, 6, 133
PHPUnit_Framework_TestListener, 134, 148
PHPUnit_Runner_TestSuiteLoader,
PHP_CodeCoverage_Filter, 117
Process Isolation,

R

Refactoring, 100
Report,
returnArgument(), 90
returnCallback(), 91
returnValue(), 89, 90

S

Selenium RC, 124, 149
setUp(), 67, 68, 68
setUpBeforeClass, 70
setUpBeforeClass(), 68, 68
Skeleton Generator, , , 120
stringContains(),
stringEndsWith(),
stringStartsWith(),
Stub, 88
Stubs, 119
Syntax Check, 66
System Under Test, 88

T

tearDown(), 67, 68, 68
tearDownAfterClass, 70
tearDownAfterClass(), 68, 68
Template Method, 67, 68, 68, 68
Test Dependencies, 6
Test Double, 88
Test Groups, , , , 146
Test Isolation, , , , 71
Test Listener, 148
Test Suite, 73, 146
Test-Driven Development, 102, 107
Test-First Programming, 102
TestDox, 118, 145
throwException(), 92

U

Unit Test, 1, 102

W

Whitelist, 117, 147
will(), 89, 90, 90, 91, 91, 92

X

Xdebug, 113
XML Configuration, 74

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

Copyright (c) 2005-2009 Sebastian Bergmann.

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

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

You are free:

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

Under the following conditions:

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

- * For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.
- * Any of the above conditions can be waived if you get permission from the copyright holder.
- * Nothing in this license impairs or restricts the author's moral rights.

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

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

=====

Creative Commons Legal Code
Attribution 3.0 Unported

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

License

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

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

THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

- a. "Adaptation" means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.
- b. "Collection" means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for 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, MERCHANTABILITY, 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/>.

=====