PHPUnit Manual

Sebastian Bergmann

PHPUnit Manual

Sebastian Bergmann

Publication date Edition for PHPUnit 3.4. Updated on 2010-04-22.

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 10
    - Testing PHP Errors 12
5. The Command-Line Test Runner 13
6. Fixtures 18
    - More setUp() than tearDown() 20
    - Variations 21
    - Sharing Fixture 21
    - Global State 22
7. Organizing Tests 24
    - Composing a Test Suite Using the Filesystem 24
    - Composing a Test Suite Using XML Configuration 25
    - Using the TestSuite Class 26
8. TestCase Extensions 29
    - Testing Output 29
9. Database Testing 31
    - Data Sets 32
        - Flat XML Data Set 32
        - XML Data Set 34
        - CSV Data Set 36
        - Replacement Data Set 37
    - Operations 37
    - Database Testing Best Practices 37
10. Incomplete and Skipped Tests 38
    - Incomplete Tests 38
    - Skipping Tests 39
11. Test Doubles 41
    - Stubs 41
    - Mock Objects 44
    - Stubbing and Mocking Web Services 46
    - Mocking the Filesystem 48
12. Testing Practices 51
    - During Development 51
    - During Debugging 51
13. Test-Driven Development 53
    - BankAccount Example 53
14. Behaviour-Driven Development 58
    - BowlingGame Example 59
15. Code Coverage Analysis 64
    - Specifying Covered Methods 66
    - Ignoring Code Blocks 67
    - Including and Excluding Files 68
16. Other Uses for Tests 69
    - Agile Documentation 69
    - Cross-Team Tests 69
17. Skeleton Generator 71
    - Generating a Test Case Class Skeleton 71
    - Generating a Class Skeleton from a Test Case Class 73
18. PHPUnit and Selenium 75
    - Selenium RC 75

PHPUnit_Extensions_SeleniumTestCase ... 75
19. Logging ... 81
Test Results (XML) ... 81
Test Results (TAP) ... 82
Test Results (JSON) ... 82
Code Coverage (XML) ... 83
20. Build Automation ... 84
Apache Ant ... 84
Apache Maven ... 85
Phing ... 85
21. Continuous Integration ... 87
Atlassian Bamboo ... 87
CruiseControl ... 88
phpUnderControl ... 90
22. PHPUnit API ... 91
Overview ... 91
PHPUnit_Framework_Assert ... 91
`assertArrayHasKey()` ... 93
`assertClassHasAttribute()` ... 94
`assertClassHasStaticAttribute()` ... 94
`assertContains()` ... 95
`assertContainsOnly()` ... 97
`assertEqualXMLStructure()` ... 98
`assertEquals()` ... 98
`assertFalse()` ... 104
`assertFileEquals()` ... 105
`assertFileExists()` ... 106
`assertGreaterThan()` ... 107
`assertGreaterThanOrEqual()` ... 108
`assertLessThan()` ... 109
`assertLessThanOrEqual()` ... 110
`assertNull()` ... 110
`assertObjectHasAttribute()` ... 111
`assertRegExp()` ... 112
`assertSame()` ... 113
`assertSelectCount()` ... 114
`assertSelectEquals()` ... 115
`assertSelectRegExp()` ... 115
`assertStringEndsWith()` ... 115
`assertStringEqualsFile()` ... 116
`assertStringStartsWith()` ... 117
`assertTag()` ... 118
`assertThat()` ... 120
`assertTrue()` ... 122
`assertType()` ... 123
`assertXmlFileEqualsXmlFile()` ... 124
`assertXmlStringEqualsXmlFile()` ... 126
`assertXmlStringEqualsXmlString()` ... 127
PHPUnit_Framework_Test ... 129
PHPUnit_Framework_TestCase ... 129
PHPUnit_Framework_TestSuite ... 131
PHPUnit_Framework_TestResult ... 132
Package Structure ... 134
23. Extending PHPUnit ... 135
Subclass PHPUnit_Framework_TestCase ... 135
Assert Classes ... 135
Subclass PHPUnit_Extensions_TestDecorator ... 135
Implement PHPUnit_Framework_Test ... 136

Subclass PHPUnit_Framework_TestResult ... 137
Implement PHPUnit_Framework_TestListener ... 137
New Test Runner ... 139
A. Assertions ... 140
B. Annotations ... 143
`@assert` ... 143
`@backupGlobals` ... 143
`@backupStaticAttributes` ... 144
`@covers` ... 144
`@dataProvider` ... 145
`@depends` ... 145
`@expectedException` ... 146
`@group` ... 146
`@outputBuffering` ... 146
`@runTestsInSeparateProcesses` ... 147
`@runInSeparateProcess` ... 147
`@test` ... 147
`@testdox` ... 147
`@ticket` ... 147
C. The XML Configuration File ... 148
PHPUnit ... 148
Test Suites ... 148
Groups ... 149
Including and Excluding Files for Code Coverage ... 149
Logging ... 150
Test Listeners ... 150
Setting PHP INI settings, Constants and Global Variables ... 151
Configuring Browsers for Selenium RC ... 151
D. Index ... 153
E. Bibliography ... 158
F. Copyright ... 159

List of Figures

15.1. Code Coverage for `setBalance()` 65
15.2. Panel with information on covering tests 65
15.3. Code Coverage for `setBalance()` with additional test 66
22.1. The five basic classes and interfaces in PHPUnit 91

List of Tables

4.1. Methods for testing exceptions 11
8.1. OutputTestCase 30
9.1. Database Test Case Methods 31
9.2. XML Data Set Element Description 35
10.1. API for Incomplete Tests 39
10.2. API for Skipping Tests 40
11.1. Matchers 46
17.1. Supported variations of the @assert annotation 73
18.1. Selenium RC API: Setup 76
18.2. Assertions 79
18.3. Template Methods 80
22.1. Constraints 121
22.2. Bottleneck Methods 128
22.3. Marking a test as being incomplete or skipped 128
22.4. Accessing non-public attributes 128
22.5. Implementor Methods 129
22.6. TestCase 129
22.7. Template Methods 130
22.8. Creating named or unnamed instances 131
22.9. Adding and retrieving tests 131
22.10. Template Methods 132
22.11. TestResult 133
22.12. TestResult and TestListener 133
22.13. TestListener Callbacks 133
A.1. Assertions 140
B.1. Annotations for specifying which methods are covered by a test 145

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 data providers 8
4.5. Using the @expectedException annotation 10
4.6. Expecting an exception to be raised by the tested code 10
4.7. Alternative approach to testing exceptions 11
4.8. Expecting a PHP error using @expectedException 12
6.1. Using setUp() to create the stack fixture 18
6.2. Example showing all template methods available 19
6.3. Sharing fixture between the tests of a test suite 21
7.1. Composing a Test Suite Using XML Configuration 25
7.2. Composing a Test Suite Using XML Configuration 26
7.3. The top-level AllTests class 26
7.4. The Package_AllTests class 26
7.5. The MySuite class 27
8.1. Using PHPUnit_Extensions_OutputTestCase 29
9.1. Setting up a database test case 31
9.2. A Flat XML Data Set 32
9.3. A XML Data Set 34
9.4. The XML Data Set DTD 35
9.5. CSV Data Set Example 36
10.1. Marking a test as incomplete 38
10.2. Skipping a test 39
11.1. The class we want to stub 41
11.2. Stubbing a method call to return a fixed value 42
11.3. Stubbing a method call to return one of the arguments 42
11.4. Stubbing a method call to return a value from a callback 43
11.5. Stubbing a method call to throw an exception 43
11.6. The Subject and Observer classes that are part of the System under Test (SUT) 44
11.7. Testing that a methods gets called once and with a specified parameter 45
11.8. Testing the concrete methods of an abstract class 46
11.9. Stubbing a web service 47
11.10. A class that interacts with the filesystem 48
11.11. Testing a class that interacts with the filesystem 48
11.12. Mocking the filesystem in a test for a class that interacts with the filesystem 49
13.1. Tests for the BankAccount class 53
13.2. Code needed for the testBalanceIsInitiallyZero() test to pass 54
13.3. The complete BankAccount class 55
13.4. The BankAccount class with Design-by-Contract assertions 56
14.1. Specification for the BowlingGame class 59
15.1. Test missing to achieve complete code coverage 65
15.2. Tests that specify which method they want to cover 66
15.3. Using the `@codeCoverageIgnoreStart` and `@codeCoverageIgnoreEnd` annotations 67
17.1. The Calculator class 71
17.2. The Calculator class with @assert annotations 72
17.3. The BowlingGameTest class 73
17.4. The generated BowlingGame class skeleton 74
18.1. Usage example for PHPUnit_Extensions_SeleniumTestCase 75
18.2. Capturing a screenshot when a test fails 77

18.3. Setting up multiple browser configurations 78
18.4. Use a directory of Selenese/HTML files as tests 80
20.1. Apache Ant build.xml file that invokes PHPUnit 84
20.2. Phing build.xml file that invokes PHPUnit 85
21.1. bamboo-init.properties 87
21.2. build.xml 87
21.3. projects/Money/build.xml 89
21.4. config.xml 89
22.1. Design-by-Contract Style Assertions 91
22.2. Using assertions with messages 92
22.3. Usage of assertArrayHasKey() 93
22.4. Usage of assertClassHasAttribute() 94
22.5. Usage of assertClassHasStaticAttribute() 95
22.6. Usage of assertContains() 95
22.7. Usage of assertContains() 96
22.8. Usage of assertContainsOnly() 97
22.9. Usage of assertEqualXMLStructure() 98
22.10. Usage of assertEquals() 98
22.11. Usage of assertEquals() with floats 100
22.12. Usage of assertEquals() with DOMDocument objects 101
22.13. Usage of assertEquals() with objects 102
22.14. Usage of assertEquals() with arrays 103
22.15. Usage of assertFalse() 104
22.16. Usage of assertFileEquals() 105
22.17. Usage of assertFileExists() 106
22.18. Usage of assertGreaterThan() 107
22.19. Usage of assertGreaterThanOrEqual() 108
22.20. Usage of assertLessThan() 109
22.21. Usage of assertLessThanOrEqual() 110
22.22. Usage of assertNull() 111
22.23. Usage of assertObjectHasAttribute() 111
22.24. Usage of assertRegExp() 112
22.25. Usage of assertSame() 113
22.26. Usage of assertSame() with objects 114
22.27. Usage of assertSelectCount() 115
22.28. Usage of assertSelectEquals() 115
22.29. Usage of assertSelectRegExp() 115
22.30. Usage of assertStringEndsWith() 115
22.31. Usage of assertStringEqualsFile() 116
22.32. Usage of assertStringStartsWith() 117
22.33. Usage of assertTag() 119
22.34. Usage of assertThat() 120
22.35. Usage of assertTrue() 122
22.36. Usage of assertType() 123
22.37. Usage of assertXmlFileEqualsXmlFile() 125
22.38. Usage of assertXmlStringEqualsXmlFile() 126
22.39. Usage of assertXmlStringEqualsXmlString() 127
22.40. Creating and running a test suite 132
23.1. The RepeatedTest Decorator 135
23.2. A data-driven test 136
23.3. A simple test listener 137
23.4. Running and observing a test suite 139

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

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

Easy to learn to write.	If it's hard to learn how to write tests, developers will not learn to write them.
Easy to write.	If tests are not easy to write, developers will not write them.
Easy to read.	Test code should contain no extraneous overhead so that the test itself does not get lost in noise that surrounds it.
Easy to execute.	The tests should run at the touch of a button and present their results in a clear and unambiguous format.
Quick to execute.	Tests should run fast so so they can be run hundreds or thousands of times a day.
Isolated.	The tests should not affect each other. If the order in which the tests are run changes, the results of the tests should not change.
Composable.	We should be able to run any number or combination of tests together. This is a corollary of isolation.

There are two main clashes between these constraints:

Easy to learn to write versus easy to write.	Tests do not generally require all the flexibility of a programming language. Many testing tools provide their own scripting language that only includes the minimum necessary features for writing tests. The resulting tests are easy to read and write because they have no noise to distract you from the content of the tests. However, learning yet another programming language and set of programming tools is inconvenient and clutters the mind.
Isolated versus quick to execute.	If you want the results of one test to have no effect on the results of another test, each test should create the full state of the world before it begins to execute and return the world to its original state when it finishes. However, setting up the world can take a long time: for example connecting to a database and initializing it to a known state using realistic data.

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

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

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

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

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

Chapter 3. Installing PHPUnit

PHPUnit 3.4 requires PHP 5.1.4 (or later) but PHP 5.3.2 (or later) is highly recommended. It 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.

The PEAR channel (`pear.phpunit.de`) that is used to distribute PHPUnit needs to be registered with the local PEAR environment. Furthermore, a component that PHPUnit depends upon is hosted on the Symfony Components PEAR channel (`pear.symfony-project.com`).

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

```
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`.

Although using the PEAR Installer is the only supported way to install PHPUnit, you can install PHPUnit manually. For manual installation, do the following:

1. Download a release archive from `http://pear.phpunit.de/get/` and extract it to a directory that is listed in the `include_path` of your `php.ini` configuration file.

2. Prepare the `phpunit` script:

   a. Rename the `phpunit.php` script to `phpunit`.

   b. Replace the `@php_bin@` string in it with the path to your PHP command-line interpreter (usually `/usr/bin/php`).

   c. Copy it to a directory that is in your path and make it executable (`chmod +x phpunit`).

3. Prepare the `PHPUnit/Util/PHP.php` script:

   a. Replace the `@php_bin@` string in it with the path to your PHP command-line interpreter (usually `/usr/bin/php`).

Chapter 4. Writing Tests for PHPUnit

Example 4.1, "Testing array operations with PHPUnit" shows how we can write tests using PHPUnit that exercice 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 "PHPUnit_Framework_Assert") are used to assert that an actual value matches an expected value.

Example 4.1. Testing array operations with PHPUnit

```php
<?php
require_once 'PHPUnit/Framework.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
<?php
class StackTest extends PHPUnit_Framework_TestCase
{
    public function testEmpty()
    {
        $stack = array();
        $this->assertTrue(empty($stack));

        return $stack;
    }

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

        return $stack;
    }

    /**
     * @depends testPush
     */
    public function testPop(array $stack)
    {
        $this->assertEquals('foo', array_pop($stack));
        $this->assertTrue(empty($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 upons `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
<?php
class DependencyFailureTest extends PHPUnit_Framework_TestCase
{
    public function testOne()
    {
        $this->assertTrue(FALSE);
    }

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

```
?>
```

```
PHPUnit 3.4.2 by Sebastian Bergmann.

DependencyFailureTest
FS

Time: 0 seconds

There was 1 failure:

1) testOne(DependencyFailureTest)
Failed asserting that <boolean:false> is true.
/home/sb/DependencyFailureTest.php:6

There was 1 skipped test:

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

FAILURES!
Tests: 2, Assertions: 1, Failures: 1, Skipped: 1.phpunit --verbose DependencyFailureTest
PHPUnit 3.4.2 by Sebastian Bergmann.

DependencyFailureTest
FS

Time: 0 seconds

There was 1 failure:

1) testOne(DependencyFailureTest)
Failed asserting that <boolean:false> is true.
/home/sb/DependencyFailureTest.php:6

There was 1 skipped test:

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

FAILURES!
Tests: 2, 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 data providers”). 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 data providers

```
<?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 3.4.2 by Sebastian Bergmann.

...F

Time: 0 seconds

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 3.4.2 by Sebastian Bergmann.

...F

Time: 0 seconds

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.
```

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 4.5, "Using the @expectedException annotation" shows how to use the `@expectedException` annotation to test whether an exception is thrown inside the tested code.

Example 4.5. Using the @expectedException annotation

```
<?php
require_once 'PHPUnit/Framework.php';

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

```

PHPUnit 3.4.2 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 3.4.2 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 4.6, "Expecting an exception to be raised by the tested code".

Example 4.6. Expecting an exception to be raised by the tested code

```
<?php
```

```
require_once 'PHPUnit/Framework.php';

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

```

PHPUnit 3.4.2 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 3.4.2 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
`void setExpectedException(string $exceptionName)`	Set the name of the expected exception to `$exceptionName`.
`String getExpectedException()`	Return the name of the expected exception.

You can also use the approach shown in Example 4.7, “Alternative approach to testing exceptions” to test exceptions.

Example 4.7. Alternative approach to testing exceptions

```
<?php
require_once 'PHPUnit/Framework.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 4.7, “Alternative approach to testing exceptions” does not raise the expected exception, the subsequent call to `fail()` (see Table 22.2, “Bottleneck Methods”) 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 4.8, “Expecting a PHP error using @expectedException”.

Example 4.8. 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 3.4.2 by Sebastian Bergmann.

.

Time: 0 seconds

OK (1 test, 1 assertion)phpunit ExpectedErrorTest
PHPUnit 3.4.2 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.

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 3.4.2 by Sebastian Bergmann.

..

Time: 0 seconds


OK (2 tests, 2 assertions)phpunit ArrayTest
PHPUnit 3.4.2 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 3.4.2 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-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.
  --coverage-source <dir>  Write code coverage / source data in XML format.

  --story-html <file>      Write Story/BDD results in HTML format to file.
  --story-text <file>      Write Story/BDD results in Text format to file.

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

  --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.

  --story                  Report test execution progress in Story/BDD format.
  --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-failure        Stop execution upon first error or failure.
  --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.
  --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.phpunit --help
PHPUnit 3.4.2 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-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.
  --coverage-source <dir>  Write code coverage / source data in XML format.

  --story-html <file>      Write Story/BDD results in HTML format to file.
  --story-text <file>      Write Story/BDD results in Text format to file.

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

  --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.

  --story                  Report test execution progress in Story/BDD format.
  --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-failure        Stop execution upon first error or failure.
  --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.
  --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.
```

`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-json`	Generates a logfile using the JSON [http://www.json.org/] format. See Chapter 19, *Logging* for more details.
`--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.
`--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.
`--coverage-clover`	Generates a logfile in XML format with the code coverage information for the tests run. See Chapter 19, *Logging* for more details. Please note that this functionality is only available when the tokenizer and Xdebug extensions are installed.
`--coverage-source`	Generates one XML file per covered PHP source file to a given directory. Each <line> element holds a line of PHP sourcecode that is annotated with code coverage information.

	Please note that this functionality is only available when the tokenizer and Xdebug extensions are installed.
`--story-html` and `--story-text`	Generates reports in HTML or plain text format for the Behaviour-Driven Development scenarios that are run. See Chapter 14, *Behaviour-Driven Development* for more details.
`--testdox-html` and `--testdox-text`	Generates agile documentation in HTML or plain text format for the tests that are run. See Chapter 16, *Other Uses for Tests* for more details.
`--filter`	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.
`--group`	Only runs tests from the specified group(s). A test can be tagged as belonging to a group using the `@group` annotation.
`--exclude-group`	Exclude tests from the specified group(s). A test can be tagged as belonging to a group using the `@group` annotation.
`--list-groups`	List available test groups.
`--loader`	Specifies the `PHPUnit_Runner_TestSuiteLoader` implementation to use.
	The standard test suite loader will look for the sourcefile in the current working directory and in each directory that is specified in PHP's `include_path` configuration directive. Following the PEAR Naming Conventions, a class name such as `Project_Package_Class` is mapped to the sourcefile name `Project/Package/Class.php`.
`--repeat`	Repeatedly runs the test(s) the specified number of times.
`--story`	Reports the test progress in a format that fits Behaviour-Driven Development. See Chapter 14, *Behaviour-Driven Development* for more details.
`--tap`	Reports the test progress using the Test Anything Protocol (TAP) [http://testanything.org/]. See Chapter 19, *Logging* for more details.
`--testdox`	Reports the test progress as agile documentation. See Chapter 16, *Other Uses for Tests* for more details.
`--colors`	Use colors in output.
`--syntax-check`	Enables the syntax check of test source files.
`--stderr`	Optionally print to `STDERR` instead of `STDOUT`.
`--stop-on-failure`	Stop execution upon first error or failure.
`--verbose`	Output more verbose information, for instance the names of tests that were incomplete or have been skipped.
`--wait`	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.

`--skeleton-class`	Generates a skeleton class `Unit` (in `Unit.php`) from a test case class `UnitTest` (in `UnitTest.php`). See Chapter 17, *Skeleton Generator* for more details.
`--skeleton-test`	Generates a skeleton test case class `UnitTest` (in `UnitTest.php`) for a class `Unit` (in `Unit.php`). See Chapter 17, *Skeleton Generator* for more details.
`--process-isolation`	Run each test in a separate PHP process.
`--no-globals-backup`	Do not backup and restore $GLOBALS. See the section called “Global State” for more details.
`--static-backup`	Backup and restore static attributes of user-defined classes. See the section called “Global State” for more details.
`--bootstrap`	A "bootstrap" PHP file that is run before the tests.
`--configuration`	Read configuration from XML file. See Appendix C, *The XML Configuration File* for more details. If `phpunit.xml` or `phpunit.xml.dist` (in that order) exist in the current working directory and `--configuration` is *not* used, the configuration will be automatically read from that file.
`--no-configuration`	Ignore `phpunit.xml` and `phpunit.xml.dist` from the current working directory.
`--include-path`	Prepend PHP's `include_path` with given path(s).
`-d`	Sets the value of the given PHP configuration option.

Note

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
<?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
<?php
require_once 'PHPUnit/Framework.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 3.4.2 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 3.4.2 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 `setUp()` and `tearDown()` template methods of the `PHPUnit_Framework_TestSuite` class (see the section called “Using the TestSuite Class”) to connect to the database before the test suite's first test and to disconnect from the database after the last test of the test suite, respectively. The `$sharedFixture` attribute of an `PHPUnit_Framework_TestSuite` object is available in the object's aggregated `PHPUnit_Framework_TestSuite` and `PHPUnit_Framework_TestCase` objects.

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

```php
<?php
require_once 'PHPUnit/Framework.php';

class DatabaseTestSuite extends PHPUnit_Framework_TestSuite
{
    protected function setUp()
    {
        $this->sharedFixture = new PDO(
          'mysql:host=wopr;dbname=test',
          'root',
          ''
        );
    }

    protected function tearDown()
    {
        $this->sharedFixture = 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-dependancy-injection-to.html] The same is true for code that uses global variables. Typically, the code you want to test is coupled strongly with a global variable and you cannot control its creation. An additional problem is the fact that one test's change to a global variable might break another test.

In PHP, global variables work like this:

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

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

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

Note

The implementation of the backup and restore operations for 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                              Tests
|-- Freezer                         |-- Freezer
|   |-- HashGenerator               |   |-- HashGenerator
|   |   `-- NonRecursiveSHA1.php    |   |   `-- NonRecursiveSHA1Test.php
|   |-- HashGenerator.php           |   |
|   |-- IdGenerator                 |   |-- IdGenerator
|   |   `-- UUID.php                |   |   `-- UUIDTest.php
|   |-- IdGenerator.php             |   |
|   |-- LazyProxy.php               |   |
|   |-- Storage                     |   |-- Storage
|   |   `-- CouchDB.php             |   |   `-- CouchDB
|   |                               |   |       |-- WithLazyLoadTest.php
|   |                               |   |       `-- WithoutLazyLoadTest.php
|   |-- Storage.php                 |   |-- StorageTest.php
|   `-- Util.php                    |   `-- UtilTest.php
`-- Freezer.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 3.4.2 by Sebastian Bergmann.

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

Time: 0 seconds

OK (75 tests, 164 assertions)phpunit Tests
PHPUnit 3.4.2 by Sebastian Bergmann.

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

Time: 0 seconds

OK (75 tests, 164 assertions)
```

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 3.4.2 by Sebastian Bergmann.

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

Time: 0 seconds

OK (28 tests, 60 assertions)phpunit Tests/FreezerTest
PHPUnit 3.4.2 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 3.4.2 by Sebastian Bergmann.

.

Time: 0 seconds

OK (1 test, 2 assertions)phpunit --filter testFreezingAnObjectWorks Tests
PHPUnit 3.4.2 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”.

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>
```

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”.

Alternatively, we can make the order in which tests are executed 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>
```

Using the TestSuite Class

The `PHPUnit_Framework_TestSuite` class of the PHPUnit framework allows us to organize tests into a hierarchy of test suite objects.

Example 7.3, “The top-level AllTests class” shows the top-level `AllTests` class for a project that has a package named `Package`.

Example 7.3. The top-level AllTests class

```
<?php
require_once 'PHPUnit/Framework.php';

require_once 'Package/AllTests.php';
// ...

class AllTests
{
    public static function suite()
    {
        $suite = new PHPUnit_Framework_TestSuite('Project');

        $suite->addTest(Package_AllTests::suite());
        // ...

        return $suite;
    }
}
?>
```

The top-level `AllTests` class aggregates the package-level `Package_AllTests` class that in turn aggregates the test case classes for the classes of said package.

Example 7.4. The Package_AllTests class

```
<?php
require_once 'PHPUnit/Framework.php';
require_once 'Framework/ClassTest.php';
// ...

class Package_AllTests
{
    public static function suite()
    {
        $suite = new PHPUnit_Framework_TestSuite('Package');

        $suite->addTestSuite('Package_ClassTest');
        // ...

        return $suite;
    }
}
?>
```

The `Package_ClassTest` class is a normal test case class that extends the `PHPUnit_Framework_TestCase` base class.

- Executing `phpunit AllTests` in the `Tests` directory will run all tests.
- Executing `phpunit AllTests` in the `Tests/Package` directory will run only the tests for the `Package_*` classes.
- Executing `phpunit ClassTest` in the `Tests/Package` directory will run only the tests for the `Package_Class` class (which are declared in the `Package_ClassTest` class).
- Executing `phpunit --filter testSomething ClassTest` in the `Tests/Package` directory will run only the test named `testSomething` from the `Package_ClassTest` class.

The `PHPUnit_Framework_TestSuite` class offers two template methods, `setUp()` and `tearDown()`, that are called before the first test of the test suite and after the last test of the test suite, respectively.

Example 7.5. The MySuite class

```
<?php
require_once 'MyTest.php';

class MySuite extends PHPUnit_Framework_TestSuite
{
    public static function suite()
    {
        return new MySuite('MyTest');
    }

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

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

The `MyTest` test case class that is added to the test suite `MySuite` in Example 7.5, “The MySuite class” has two test methods, `testOne()` and `testTwo()` as well as the `setUp()` and `tearDown()` methods. The output shows in which order these eight methods are called:

```
MySuite::setUp()
MyTest::setUp()
MyTest::testOne()
MyTest::tearDown()
MyTest::setUp()
MyTest::testTwo()
MyTest::tearDown()
MySuite::tearDown()
```

Variables stored in `$this->sharedFixture` by the `setUp()` method of the `PHPUnit_Framework_TestSuite` class are available as `$this->sharedFixture` in all the test that are aggregated by the test suite object (see the section called “Sharing Fixture”).

Note

A `TestSuite`'s `setUp()` and `tearDown()` methods will be called even if no test of the test suite is run because it is, for instance, filtered.

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
<?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 3.4.2 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 3.4.2 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-effecting 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.
    }
}
?>
```

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
`PHPUnit_Extensions_Database_DB_IDatabaseConnection getConnection()`	Implement to return the database connection that will be checked for expected data sets and tables.
`PHPUnit_Extensions_Database_DataSet_IDataSet getDataSet()`	Implement to return the data set that will be used in in database set up and tear down operations.
`PHPUnit_Extensions_Database_Operation_DatabaseOperation getSetUpOperation()`	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”.
`PHPUnit_Extensions_Database_Operation_DatabaseOperation getTearDownOperation()`	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 where 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>
    <row>
```

```
      <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
`<dataset>`	The root element of the xml data set file.	One or more `<table>` elements.	None
`<table>`	Defines a table in the dataset.	One or more `<column>` elements and zero or more `<row>` elements.	`name` - The name of the table.
`<column>`	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://phpun.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
<?php
require_once 'PHPUnit/Framework.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 3.4.2 by Sebastian Bergmann.

SampleTest
```

```
I


Time: 0 seconds

There was 1 incomplete test:

1) testSomething(SampleTest)
This test has not been implemented yet.
/home/sb/SampleTest.php:14

OK, but incomplete or skipped tests!
Tests: 1, Assertions: 0, Incomplete: 1.phpunit --verbose SampleTest
PHPUnit 3.4.2 by Sebastian Bergmann.

SampleTest
I

Time: 0 seconds

There was 1 incomplete test:

1) testSomething(SampleTest)
This test has not been implemented yet.
/home/sb/SampleTest.php:14

OK, but incomplete or skipped tests!
Tests: 1, Assertions: 0, 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**
`void markTestIncomplete()`	Marks the current test as incomplete.
`void markTestIncomplete(string $message)`	Marks the current test as incomplete using `$message` as an explanatory message.

Skipping Tests

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

Example 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
<?php
require_once 'PHPUnit/Framework.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 3.4.2 by Sebastian Bergmann.

DatabaseTest
S


Time: 0 seconds

There was 1 skipped test:

1) testConnection(DatabaseTest)
The MySQLi extension is not available.
/home/sb/DatabaseTest.php:11

OK, but incomplete or skipped tests!
Tests: 1, Assertions: 0, Skipped: 1.phpunit --verbose DatabaseTest
PHPUnit 3.4.2 by Sebastian Bergmann.

DatabaseTest
S

Time: 0 seconds

There was 1 skipped test:

1) testConnection(DatabaseTest)
The MySQLi extension is not available.
/home/sb/DatabaseTest.php:11

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
`void markTestSkipped()`	Marks the current test as skipped.
`void markTestSkipped(string $message)`	Marks the current test as skipped using `$message` 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 refered 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 (see Table 22.6, “TestCase”) 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
<?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.

Sometimes you want to return one of the arguments of a method call (unchanged) as the result of a stubbed method call. Example 11.3, “Stubbing a method call to return one of the arguments” shows how you can achieve this using `returnArgument()` instead of `returnValue()`.

Example 11.3. 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 11.4, “Stubbing a method call to return a value from a callback” for an example.

Example 11.4. 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'));
    }
}
?>
```

Instead of returning a value, a stubbed method can also raise an exception. Example 11.5, “Stubbing a method call to throw an exception” shows how to use `throwException()` to do this.

Example 11.5. 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 refered 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 11.6, "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 11.6. 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');
    }

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

    // Other methods.
}

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

    // Other methods.
}
?>
```

Example 11.7, “Testing that a methods gets called once and with a specified parameter” 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 (see Table 22.6, “TestCase”) 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 11.7. Testing that a methods gets called once and with a specified parameter

```
<?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();
    }
}
?>
```

Table 22.1, “Constraints” shows the constraints and Table 11.1, “Matchers” shows the matchers that are available.

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 11.8. Testing the concrete methods of an abstract class

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

Example 11.9, “Stubbing a web service” shows how `getMockFromWsdl()` can be used to stub, for example, the web service described in `GoogleSearch.wsdl`.

Example 11.9. Stubbing a web service

```
<?php
class GoogleTest extends PHPUnit_Framework_TestCase
{
    public function testSearch()
    {
        $googleSearch = $this->getMockFromWsdl(
          'GoogleSearch.wsdl', 'GoogleSearch'
        );

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

        $element = new StdClass;
        $element->summary = '';
        $element->URL = 'http://www.phpunit.de/';
        $element->snippet = '...';
        $element->title = '<b>PHPUnit</b>';
        $element->cachedSize = '11k';
        $element->relatedInformationPresent = TRUE;
        $element->hostName = 'www.phpunit.de';
        $element->directoryCategory = $directoryCategory;
        $element->directoryTitle = '';

        $result = new StdClass;
        $result->documentFiltering = FALSE;
        $result->searchComments = '';
        $result->estimatedTotalResultsCount = 378000;
        $result->estimateIsExact = FALSE;
        $result->resultElements = array($element);
        $result->searchQuery = 'PHPUnit';
        $result->startIndex = 1;
        $result->endIndex = 1;
        $result->searchTips = '';
        $result->directoryCategories = array();
        $result->searchTime = 0.248822;

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

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

Mocking the Filesystem

vfsStream [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 11.10, “A class that interacts with the filesystem” shows a class that interacts with the filesystem.

Example 11.10. 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 11.11, “Testing a class that interacts with the filesystem”).

Example 11.11. 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 11.12, "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 11.12. 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 'PHPUnit/Framework.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 3.4.2 by Sebastian Bergmann.

.
Fatal error: Call to undefined method BankAccount::withdrawMoney()phpunit BankAccountTes
PHPUnit 3.4.2 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 an `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 3.4.2 by Sebastian Bergmann.

...

Time: 0 seconds


OK (3 tests, 3 assertions)phpunit BankAccountTest
PHPUnit 3.4.2 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
<?php
require_once 'PHPUnit/Framework.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 faciliates 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)`

  ...

BowlingGame Example

In this section, we will look at the example of a class that calculates the score for a game of bowling. The rules for this are as follows:

- The game consists of 10 frames.
- In each frame the player has two opportunities to knock down 10 pins.
- The score for a frame is the total number of pins knocked down, plus bonuses for strikes and spares.
- A spare is when the player knocks down all 10 pins in two tries.

  The bonus for that frame is the number of pins knocked down by the next roll.
- A strike is when the player knocks down all 10 pins on his first try.

  The bonus for that frame is the value of the next two balls rolled.

Example 14.1, "Specification for the BowlingGame class" shows how the above rules can be written down as specification scenarios using `PHPUnit_Extensions_Story_TestCase`.

Example 14.1. Specification for the BowlingGame class

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

class BowlingGameSpec extends PHPUnit_Extensions_Story_TestCase
{
    /**
     * @scenario
     */
    public function scoreForGutterGameIs0()
    {
        $this->given('New game')
             ->then('Score should be', 0);
    }

    /**
     * @scenario
     */
    public function scoreForAllOnesIs20()
    {
        $this->given('New game')
             ->when('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
             ->and('Player rolls', 1)
```

```
                 ->and('Player rolls', 1)
                 ->and('Player rolls', 1)
                 ->then('Score should be', 20);
    }

    /**
     * @scenario
     */
    public function scoreForOneSpareAnd3Is16()
    {
        $this->given('New game')
             ->when('Player rolls', 5)
             ->and('Player rolls', 5)
             ->and('Player rolls', 3)
             ->then('Score should be', 16);
    }

    /**
     * @scenario
     */
    public function scoreForOneStrikeAnd3And4Is24()
    {
        $this->given('New game')
             ->when('Player rolls', 10)
             ->and('Player rolls', 3)
             ->and('Player rolls', 4)
             ->then('Score should be', 24);
    }

    /**
     * @scenario
     */
    public function scoreForPerfectGameIs300()
    {
        $this->given('New game')
             ->when('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->and('Player rolls', 10)
             ->then('Score should be', 300);
    }

    public function runGiven(&$world, $action, $arguments)
    {
        switch($action) {
            case 'New game': {
                $world['game']  = new BowlingGame;
                $world['rolls'] = 0;
            }
            break;

            default: {
                return $this->notImplemented($action);
            }
        }
    }
    public function runWhen(&$world, $action, $arguments)
    {
        switch($action) {
            case 'Player rolls': {
                $world['game']->roll($arguments[0]);
                $world['rolls']++;
            }
            break;

            default: {
                return $this->notImplemented($action);
            }
        }
    }

    public function runThen(&$world, $action, $arguments)
    {
        switch($action) {
            case 'Score should be': {
                for ($i = $world['rolls']; $i < 20; $i++) {
                    $world['game']->roll(0);
                }

                $this->assertEquals($arguments[0], $world['game']->score());
            }
            break;

            default: {
                return $this->notImplemented($action);
            }
        }
    }
}
?>
```

```

PHPUnit 3.4.2 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
    Then Score should be 300

Scenarios: 5, Failed: 0, Skipped: 0, Incomplete: 0.phpunit --story BowlingGameSpec
PHPUnit 3.4.2 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
    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 effiency.

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.

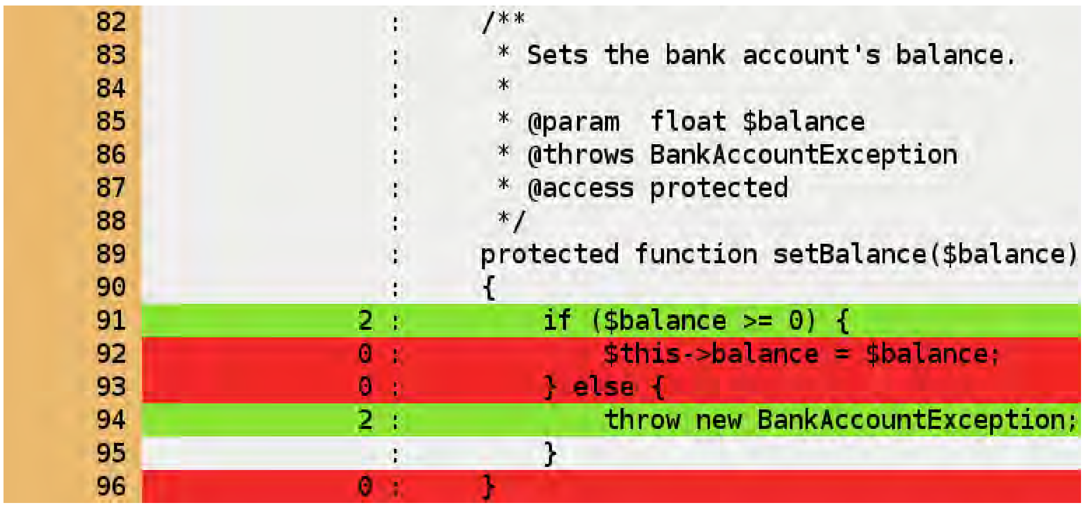
PHPUnit's code coverage functionality makes use of the Xdebug [http://www.xdebug.org/] extension for PHP.

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

```
PHPUnit 3.4.2 by Sebastian Bergmann.

...

Time: 0 seconds

OK (3 tests, 3 assertions)

Generating report, this may take a moment.phpunit --coverage-html ./report BankAccountTe
PHPUnit 3.4.2 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.

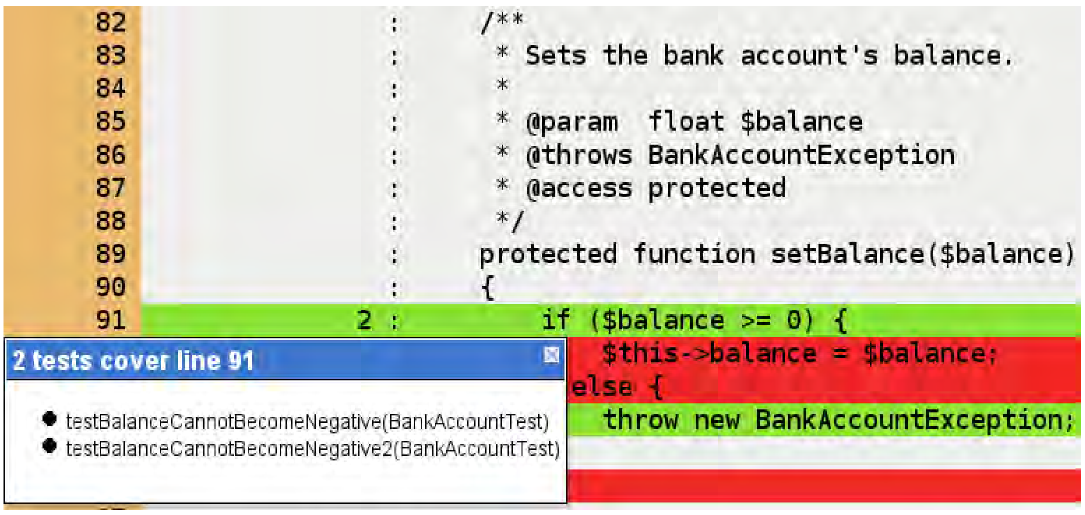
Figure 15.1. Code Coverage for `setBalance()`



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

2 tests cover line 91

- testBalanceCannotBecomeNegative(BankAccountTest)
- testBalanceCannotBecomeNegative2(BankAccountTest)

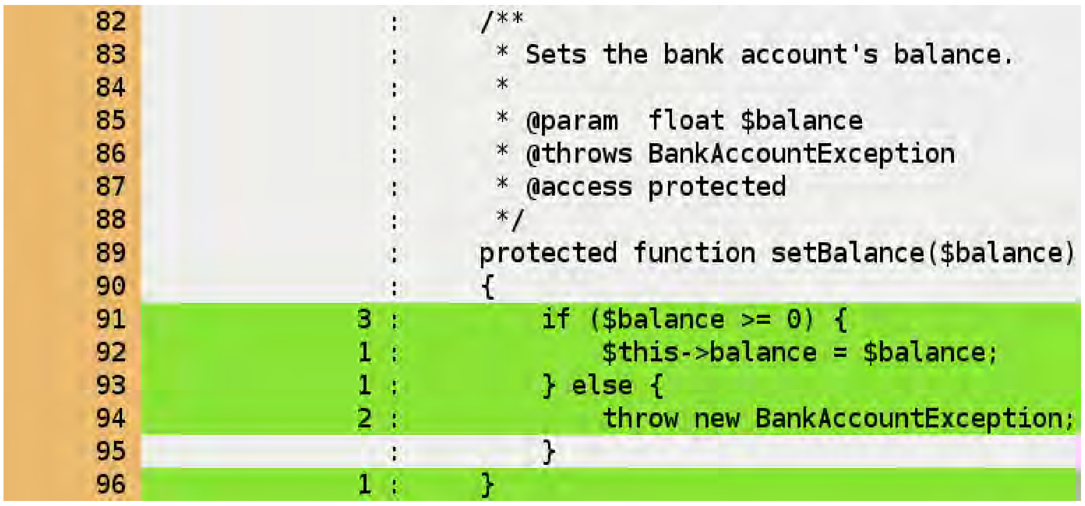
The code coverage report for our `BankAccount` example shows that we do not have any tests yet that call the `setBalance()`, `depositMoney()`, and `withdrawMoney()` methods with legal values. Example 15.1, "Test missing to achieve complete code coverage" shows a test that can be added to the `BankAccountTest` test case class to completely cover the `BankAccount` class.

Example 15.1. Test missing to achieve complete code coverage

```php
<?php
require_once 'PHPUnit/Framework.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 'PHPUnit/Framework.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 `@codeCoverageIgnoreStart` and `@codeCoverageIgnoreEnd` annotations as shown in Example 15.3, “Using the `@codeCoverageIgnoreStart` and `@codeCoverageIgnoreEnd` annotations”.

Example 15.3. Using the `@codeCoverageIgnoreStart` and `@codeCoverageIgnoreEnd` annotations

```
<?php
```

```
class Sample
{
    // ...

    public function doSomething()
    {
        if (0) {
            // @codeCoverageIgnoreStart
            $this->doSomethingElse();
            // @codeCoverageIgnoreEnd
        }
    }

    // ...
}
?>
```

The lines of code between the `@codeCoverageIgnoreStart` and `@codeCoverageIgnoreEnd` 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 `PHPUnit_Util_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 3.4.2 by Sebastian Bergmann.

BankAccount
 [x] Balance is initially zero
 [x] Balance cannot become negativephpunit --testdox BankAccountTest
PHPUnit 3.4.2 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 3.4.2 by Sebastian Bergmann.

Wrote test class skeleton for Calculator to CalculatorTest.php.phpunit --skeleton Calcul
PHPUnit 3.4.2 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 3.4.2 by Sebastian Bergmann.

CalculatorTest
I
```

```
Time: 0 seconds

There was 1 incomplete test:

1) testAdd(CalculatorTest)
This test has not been implemented yet.
/home/sb/CalculatorTest.php:54

OK, but incomplete or skipped tests!
Tests: 1, Assertions: 0, Incomplete: 1.phpunit --verbose CalculatorTest
PHPUnit 3.4.2 by Sebastian Bergmann.

CalculatorTest
I

Time: 0 seconds

There was 1 incomplete test:

1) testAdd(CalculatorTest)
This test has not been implemented yet.
/home/sb/CalculatorTest.php:54

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 3.4.2 by Sebastian Bergmann.

....

Time: 0 seconds


OK (4 tests, 4 assertions)phpunit CalculatorTest
PHPUnit 3.4.2 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
<?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.');
    }

    /**
     * @todo Implement score().
     */
    public function score()
    {
        // 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 3.4.2 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 3.4.2 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
`void setBrowser(string $browser)`	Set the browser to be used by the Selenium RC server.
`void setBrowserUrl(string $browserUrl)`	Set the base URL for the tests.
`void setHost(string $host)`	Set the hostname for the connection to the Selenium RC server.
`void setPort(int $port)`	Set the port for the connection to the Selenium RC server.
`void setTimeout(int $timeout)`	Set the timeout for the connection to the Selenium RC server.
`void setSleep(int $seconds)`	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
<?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 3.4.2 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 3.4.2 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
<?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
`void defaultAssertions()`	Override to perform assertions that are shared by all tests of a test case. This method is called after each command that is sent to the Selenium 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
<?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="3.4.2">
  <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. Build Automation

This chapter provides an overview of build automation tools that are commonly used with PHPUnit.

Apache Ant

Apache Ant [http://ant.apache.org/] is a Java-based build tool. In theory, it is kind of like *make*, without *make*'s wrinkles. Build files for Apache Ant are XML-based, calling out a target tree where various tasks get executed.

Example 20.1, "Apache Ant build.xml file that invokes PHPUnit" shows an Apache Ant build.xml file that invokes PHPUnit using the built-in `<exec>` task. The build process is aborted if a test fails (`failonerror="true"`).

Example 20.1. Apache Ant build.xml file that invokes PHPUnit

```
<project name="Money" default="build">
 <target name="clean">
  <delete dir="${basedir}/build"/>
 </target>

 <target name="prepare">
  <mkdir dir="${basedir}/build/logs"/>
 </target>

 <target name="phpunit">
  <exec dir="${basedir}" executable="phpunit" failonerror="true">
   <arg line="--log-xml ${basedir}/build/logs/phpunit.xml MoneyTest" />
  </exec>
 </target>

 <target name="build" depends="clean,prepare,phpunit"/>
</project>
```

```

Buildfile: build.xml

clean:

prepare:
    [mkdir] Created dir: /home/sb/Money/build/logs

phpunit:
     [exec] PHPUnit 3.4.2 by Sebastian Bergmann.
     [exec]
     [exec] ......................
     [exec]
     [exec] Time: 0 seconds
     [exec]
     [exec] OK (22 tests, 34 assertions)

build:

BUILD SUCCESSFUL
Total time: 0 secondsant
Buildfile: build.xml

clean:

prepare:
    [mkdir] Created dir: /home/sb/Money/build/logs

phpunit:
     [exec] PHPUnit 3.4.2 by Sebastian Bergmann.
     [exec]
     [exec] ......................
     [exec]
     [exec] Time: 0 seconds
     [exec]
     [exec] OK (22 tests, 34 assertions)

build:

BUILD SUCCESSFUL
Total time: 0 seconds
```

The XML logfile for test results produced by PHPUnit (see the section called “Test Results (XML)”) is based upon the one used by the `<junit>` [http://ant.apache.org/manual/OptionalTasks/junit.html] task that comes with Apache Ant.

Apache Maven

Apache Maven [http://maven.apache.org/] is a software project management and comprehension tool. Based on the concept of a Project Object Model (POM), Apache Maven can manage a project's build, reporting and documentation from a central place of information. Maven for PHP [http://www.php-maven.org/] uses the power of Maven for building, testing, and documenting PHP projects.

Phing

Phing [http://www.phing.info/] (PHing Is Not GNU make) is a project build system based on Apache Ant. You can do anything with it that you could do with a traditional build system such as GNU make, and its use of simple XML build files and extensible PHP "task" classes make it an easy-to-use and highly flexible build framework. Features include file transformations (e.g. token replacement, XSLT transformation, Smarty template transformations), file system operations, interactive build support, SQL execution, CVS operations, tools for creating PEAR packages, and much more.

Example 20.2, “Phing build.xml file that invokes PHPUnit” shows a Phing build.xml file that invokes PHPUnit using the built-in `<phpunit>` task. The build process is aborted if a test fails (`haltonfailure="true"`).

Example 20.2. Phing build.xml file that invokes PHPUnit

```
<project name="Money" default="build">
 <target name="clean">
  <delete dir="build"/>
 </target>

 <target name="prepare">
  <mkdir dir="build/logs"/>
 </target>

 <target name="phpunit">
  <phpunit printsummary="true" haltonfailure="true">
    <formatter todir="build/logs" type="xml"/>
    <batchtest>
      <fileset dir=".">
        <include name="*Test.php"/>
      </fileset>
    </batchtest>
  </phpunit>
 </target>

 <target name="build" depends="clean,prepare,phpunit"/>
</project>
```

```

Buildfile: /home/sb/Money/build.xml

Money > clean:


Money > prepare:

    [mkdir] Created dir: /home/sb/Money/build/logs

Money > phpunit:

  [phpunit] Test: MoneyTest, Run: 22, Failures: 0, Errors: 0,
            Incomplete: 0, Skipped: 0, Time elapsed: 0.06887 s

Money > build:


BUILD FINISHED

Total time: 0.2632 secondsphing
Buildfile: /home/sb/Money/build.xml

Money > clean:

Money > prepare:

    [mkdir] Created dir: /home/sb/Money/build/logs

Money > phpunit:

  [phpunit] Test: MoneyTest, Run: 22, Failures: 0, Errors: 0,
            Incomplete: 0, Skipped: 0, Time elapsed: 0.06887 s

Money > build:

BUILD FINISHED

Total time: 0.2632 seconds
```

Chapter 21. Continuous Integration

This chapter provides an overview of Continuous Integration summarizing the technique and its application with PHPUnit.

> Continuous Integration [http://www.martinfowler.com/articles/continuousIntegration.html] is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily, leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.
>
> —Martin Fowler

Continuous Integration demands a fully automated and reproducible build, including testing, that runs many times a day. This allows each developer to integrate daily thus reducing integration problems. While this can be achieved by setting up a cronjob [http://en.wikipedia.org/wiki/Crontab] that makes a fresh checkout from the project's source code repository [http://en.wikipedia.org/wiki/Source_code_repository] at regular intervals, runs the tests, and publishes the results, a more comfortable solution may be desired.

Atlassian Bamboo

Atlassian Bamboo [http://www.atlassian.com/software/bamboo/] is a Continuous Integration (CI) server that assists software development teams by providing automated building and testing of software source-code status, updates on successful/failed builds, and reporting tools for statistical analysis.

The following example assumes that the Bamboo distribution archive has been unpacked into `/usr/local/Bamboo`.

1.
   ```
   cd /usr/local/Bamboo
   ```

2. Edit the `webapp/WEB-INF/classes/bamboo-init.properties` file.

3. Optionally install the `bamboo-checkstyle` plugin.

Example 21.1. bamboo-init.properties

```
bamboo.home=/usr/local/Bamboo
```

4.
   ```
   ./bamboo.sh start
   ```

5. Open `http://localhost:8085/` in your webbrowser.

6. Follow the guided installation instructions.

7. Configure Apache Ant as a Builder in the Administration panel.

Bamboo is now configured and we can set up a project plan. First we need a project, though. For the purpose of this example lets assume we have a copy of the `Money` sample that ships with PHPUnit in a Subversion repository (`file:///var/svn/money`). Together with the `*.php` files we also have the following Apache Ant build script (`build.xml`) in the repository.

Example 21.2. build.xml

```
<project name="Money" default="build">
 <target name="clean">
```

```
  <delete dir="${basedir}/build"/>
 </target>

 <target name="prepare">
  <mkdir dir="${basedir}/build"/>
  <mkdir dir="${basedir}/build/logs"/>
 </target>

 <target name="phpcs">
  <exec dir="${basedir}"
        executable="phpcs"
        output="${basedir}/build/logs/checkstyle.xml"
        failonerror="false">
   <arg line="--report=checkstyle ." />
  </exec>
 </target>

 <target name="phpunit">
  <exec dir="${basedir}" executable="phpunit" failonerror="true">
   <arg line="--log-xml         ${basedir}/build/logs/phpunit.xml
              --coverage-clover ${basedir}/build/logs/clover.xml
              --coverage-html   ${basedir}/build/coverage
              MoneyTest" />
  </exec>
 </target>

 <target name="build" depends="clean,prepare,phpcs,phpunit"/>
</project>
```

Now that we have a project, we can create a plan for it in Bamboo.

8. Open `http://localhost:8080/` in your webbrowser.

9. Follow the guided "Create a Plan" instructions.

10. In step 3 of "Create a Plan", check the "The build will produce test results" and "Clover output will be produced" options and provide the paths to the XML files produced by PHPUnit.

    If you installed the `bamboo-checkstyle` plugin also check the "CheckStyle output will be produced" option and provide the path of the XML file produced by PHP_CodeSniffer [http://pear.php.net/package/PHP_CodeSniffer/].

11. In step 5 of "Create a Plan", set up an artifact for the HTML files (`*.*`, `build/coverage`) that PHPUnit produces.

CruiseControl

CruiseControl [http://cruisecontrol.sourceforge.net/] is a framework for continuous build processes and includes, but is not limited to, plugins for email notification, Apache Ant [http://ant.apache.org/], and various source control tools. A web interface is provided to view the details of the current and previous builds.

The following example assumes that CruiseControl has been installed into `/usr/local/cruisecontrol`.

1.
```
cd /usr/local/cruisecontrol
```

2.
```
mkdir -p projects/Money/build/logs
```

3.
```
cd projects/Money
```

4.
```
svn co file:///var/svn/money source
```

5. Edit the `build.xml` file.

Example 21.3. projects/Money/build.xml

```
<project name="Money" default="build" basedir=".">
 <target name="checkout">
  <exec dir="${basedir}/source/" executable="svn">
   <arg line="up"/>
  </exec>
 </target>

 <target name="test">
  <exec dir="${basedir}/source" executable="phpunit" failonerror="true">
   <arg line="--log-xml ${basedir}/build/logs/phpunit.xml MoneyTest"/>
  </exec>
 </target>

 <target name="build" depends="checkout,test"/>
</project>
```

6.
```
cd /usr/local/cruisecontrol
```

7. Edit the `config.xml` file.

Example 21.4. config.xml

```
<cruisecontrol>
  <project name="Money" buildafterfailed="false">
    <plugin
    name="svnbootstrapper"
    classname="net.sourceforge.cruisecontrol.bootstrappers.SVNBootstrapper"/>
    <plugin
    name="svn"
    classname="net.sourceforge.cruisecontrol.sourcecontrols.SVN"/>

    <listeners>
      <currentbuildstatuslistener file="logs/${project.name}/status.txt"/>
    </listeners>

    <bootstrappers>
      <svnbootstrapper localWorkingCopy="projects/${project.name}/source/"/>
    </bootstrappers>

    <modificationset>
      <svn localWorkingCopy="projects/${project.name}/source/"/>
    </modificationset>

    <schedule interval="300">
      <ant
      anthome="apache-ant-1.7.0"
      buildfile="projects/${project.name}/build.xml"/>
    </schedule>

    <log dir="logs/${project.name}">
      <merge dir="projects/${project.name}/build/logs/"/>
    </log>

    <publishers>
      <currentbuildstatuspublisher
      file="logs/${project.name}/buildstatus.txt"/>
      <email
      mailhost="localhost"
      buildresultsurl="http://cruise.example.com/buildresults/${project.name}"
      skipusers="true"
      spamwhilebroken="true"
      returnaddress="project@example.com">
        <failure address="dev@lists.example.com" reportWhenFixed="true"/>
      </email>
    </publishers>
  </project>
</cruisecontrol>
```

Now we can (re)start the CruiseControl server.

8.
```
./cruisecontrol.sh
```

9. Open `http://localhost:8080/` in your webbrowser.

phpUnderControl

phpUnderControl [http://www.phpundercontrol.org/] is an extension for CruiseControl that integrates several PHP development tools, such as PHPUnit for testing, PHP_CodeSniffer [http://pear.php.net/package/PHP_CodeSniffer] for static code analysis [http://en.wikipedia.org/wiki/Static_code_analysis], and PHPDocumentor [http://www.phpdoc.org/] for API documentation generation [http://en.wikipedia.org/wiki/Documentation_generator]. It comes with a powerful command-line tool that can, among other things, automatically create CruiseControl's XML configuration files for your project. The following example assumes that CruiseControl has been installed into `/usr/local/cruisecontrol`.

1.
```
pear install --alldeps phpunit/phpUnderControl
```

2.
```
phpuc install /usr/local/cruisecontrol
```

3.
```
phpuc project --version-control svn
              --version-control-url file:///var/svn/money
              --test-case MoneyTest
              --test-file MoneyTest.php
              --test-dir .
              --project-name Money
              /usr/local/cruisecontrol
```

The above command creates the project directory and the project's `build.xml` configuration file, performs the initial checkout from the source repository, and adds the new project to the global `config.xml` configuration file. Now we can (re)start the CruiseControl server.

4.
```
cd /usr/local/cruisecontrol
```

5.
```
./cruisecontrol.sh
```

6. Open `http://localhost:8080/` in your webbrowser.

Chapter 22. PHPUnit API

For most uses, PHPUnit has a simple API: subclass `PHPUnit_Framework_TestCase` for your test cases and call `assertTrue()` or `assertEquals()`. However, for those of you who would like to look deeper into PHPUnit, here are all of its published methods and classes.

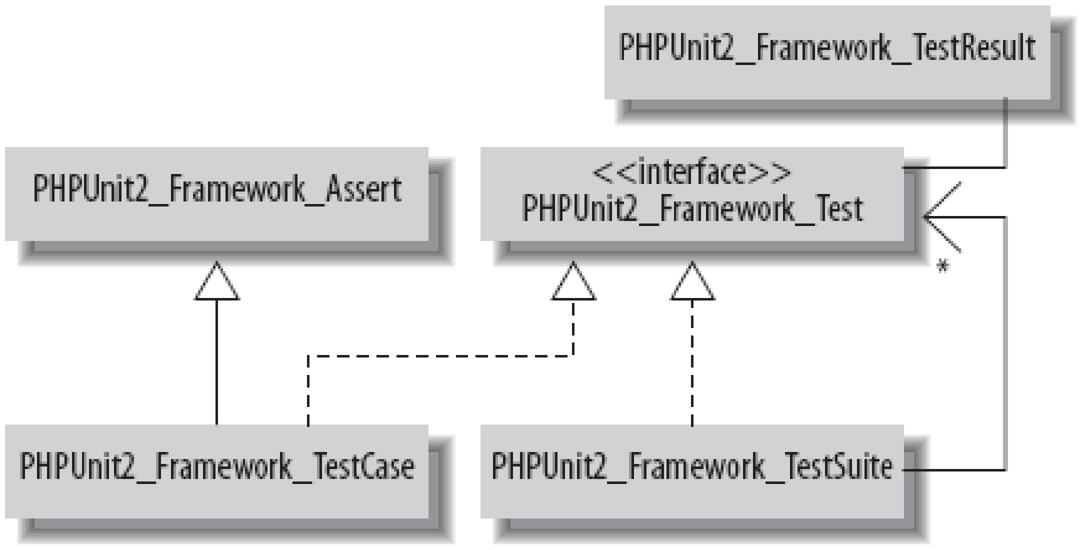
Overview

Most of the time, you will encounter five classes or interfaces when you are using PHPUnit:

`PHPUnit_Framework_Assert`	A collection of static methods for checking actual values against expected values.
`PHPUnit_Framework_Test`	The interface of all objects that act like tests.
`PHPUnit_Framework_TestCase`	A single test.
`PHPUnit_Framework_TestSuite`	A collection of tests.
`PHPUnit_Framework_TestResult`	A summary of the results of running one or more tests.

Figure 22.1, "The five basic classes and interfaces in PHPUnit" shows the relationship of the five basic classes and interfaces in PHPUnit: `PHPUnit_Framework_Assert`, `PHPUnit_Framework_Test`, `PHPUnit_Framework_TestCase`, `PHPUnit_Framework_TestSuite`, and `PHPUnit_Framework_TestResult`.

Figure 22.1. The five basic classes and interfaces in PHPUnit


PHPUnit2_Framework_TestResult
PHPUnit2_Framework_Assert
<<interface>>
PHPUnit2_Framework_Test
*
PHPUnit2_Framework_TestCase
PHPUnit2_Framework_TestSuite

PHPUnit_Framework_Assert

Most test cases written for PHPUnit are derived indirectly from the class `PHPUnit_Framework_Assert`, which contains methods for automatically checking values and reporting discrepancies. The methods are declared static, so you can write design-by-contract style assertions in your methods and have them reported through PHPUnit (Example 22.1, "Design-by-Contract Style Assertions").

Example 22.1. Design-by-Contract Style Assertions

```
<?php
```

```
require_once 'PHPUnit/Framework.php';

class Sample
{
    public function aSampleMethod($object)
    {
        PHPUnit_Framework_Assert::assertNotNull($object);
    }
}

$sample = new Sample;
$sample->aSampleMethod(NULL);
?>
```

```
Fatal error: Uncaught exception 'PHPUnit_Framework_ExpectationFailedException'
with message 'Failed asserting that <null> is not identical to <null>.'
```

Most of the time, though, you'll be checking the assertions inside of tests.

There are two variants of each of the assertion methods: one takes a message to be displayed with the error as a parameter, and one does not. The optional message is typically displayed when a failure is displayed, which can make debugging easier.

Example 22.2. Using assertions with messages

```
<?php
require_once 'PHPUnit/Framework.php';

class MessageTest extends PHPUnit_Framework_TestCase
{
    public function testMessage()
    {
        $this->assertTrue(FALSE, 'This is a custom message.');
    }
}
?>
```

The following example shows the output you get when you run the `testMessage()` test from Example 22.2, “Using assertions with messages”, using assertions with messages:

```


PHPUnit 3.4.4 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) testMessage(MessageTest)
This is a custom message.
Failed asserting that <boolean:false> is true.
/home/sb/MessageTest.php:8

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.phpunit MessageTest
PHPUnit 3.4.4 by Sebastian Bergmann.

F

Time: 0 seconds
```

```
There was 1 failure:

1) testMessage(MessageTest)
This is a custom message.
Failed asserting that <boolean:false> is true.
/home/sb/MessageTest.php:8

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

Below are all the varieties of assertions.

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 22.3. Usage of assertArrayHasKey()

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

```

PHPUnit 3.4.4 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 3.4.4 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 22.4. Usage of assertClassHasAttribute()

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

```

PHPUnit 3.4.4 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 3.4.4 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 $className[, string $message = ''])`

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

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

Example 22.5. Usage of assertClassHasStaticAttribute()

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

```

PHPUnit 3.4.4 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 3.4.4 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 22.6. Usage of assertContains()

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

```

PHPUnit 3.4.4 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 3.4.4 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 22.7. Usage of assertContains()

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

```

PHPUnit 3.4.4 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 3.4.4 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 22.8. Usage of assertContainsOnly()

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

```

PHPUnit 3.4.4 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 3.4.4 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.
```

assertEqualXMLStructure()

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

XXX

Example 22.9. Usage of assertEqualXMLStructure()

```php
<?php
?>
```

```

phpunit EqualXMLStructureTest
```

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 22.10. Usage of assertEquals()

```php
<?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 3.4.4 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 3.4.4 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 neccessary.

Example 22.11. Usage of assertEquals() with floats

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

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

```

PHPUnit 3.4.4 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 3.4.4 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 `$message` if the XML documents represented by the two DOMDocument objects `$expected` and `$actual` are not equal.

Example 22.12. 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 3.4.4 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 3.4.4 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 22.13. 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 3.4.4 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 3.4.4 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 22.14. Usage of assertEquals() with arrays

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

```
PHPUnit 3.4.4 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 3.4.4 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.
```

assertFalse()

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

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

Example 22.15. Usage of assertFalse()

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

```


PHPUnit 3.4.4 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 3.4.4 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 22.16. Usage of assertFileEquals()

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

```


PHPUnit 3.4.4 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 3.4.4 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 22.17. Usage of assertFileExists()

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

```

PHPUnit 3.4.4 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 3.4.4 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 22.18. Usage of assertGreaterThan()

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

```

PHPUnit 3.4.4 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 3.4.4 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 22.19. Usage of assertGreaterThanOrEqual()

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

```


PHPUnit 3.4.4 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 3.4.4 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.
```

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 22.20. Usage of assertLessThan()

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

```
PHPUnit 3.4.4 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 3.4.4 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 22.21. Usage of assertLessThanOrEqual()

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

```

PHPUnit 3.4.4 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 3.4.4 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 22.22. Usage of assertNull()

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

```


PHPUnit 3.4.4 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 3.4.4 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 22.23. Usage of assertObjectHasAttribute()

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

```

PHPUnit 3.4.4 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 3.4.4 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 22.24. Usage of assertRegExp()

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

```

PHPUnit 3.4.4 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 3.4.4 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.
```

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 22.25. Usage of assertSame()

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

```
PHPUnit 3.4.4 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 3.4.4 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 22.26. Usage of assertSame() with objects

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

```

PHPUnit 3.4.4 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 3.4.4 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])`

XXX

Example 22.27. Usage of assertSelectCount()

```
<?php
?>
```

```

phpunit SelectCountTest
```

assertSelectEquals()

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

XXX

Example 22.28. Usage of assertSelectEquals()

```
<?php
?>
```

```

phpunit SelectEqualsTest
```

assertSelectRegExp()

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

XXX

Example 22.29. Usage of assertSelectRegExp()

```
<?php
?>
```

```

phpunit SelectRegExpTest
```

assertStringEndsWith()

`assertStringEndsWith(string $suffix, string $string[, string $message = ''])`

Reports an error identified by `$message` if the `$string` does not end with `$suffix`.

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

Example 22.30. Usage of assertStringEndsWith()

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

```
PHPUnit 3.4.4 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 3.4.4 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 22.31. Usage of assertStringEqualsFile()

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

```
PHPUnit 3.4.4 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 3.4.4 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 22.32. Usage of assertStringStartsWith()

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

```
PHPUnit 3.4.4 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 3.4.4 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 corresponsing value.
- `tags`: The node type must match the corresponding value.
- `attributes`: The node's attributes must match the corresponsing values in the `$attributes` associative array.
- `content`: The text content must match the given value.
- `parent`: The node's parent must match the `$parent` associative array.
- `child`: At least one of the node's immediate children must meet the criteria described by the `$child` associative array.
- `ancestor`: At least one of the node's ancestors must meet the criteria described by the `$ancestor` associative array.
- `descendant`: At least one of the node's descendants must meet the criteria described by the `$descendant` associative array.
- `children`: Associative array for counting children of a node.
  - `count`: The number of matching children must be equal to this number.
  - `less_than`: The number of matching children must be less than this number.

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

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

Example 22.33. 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 22.34, “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 22.34. 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 22.1, “Constraints” shows the available `PHPUnit_Framework_Constraint` classes.

Table 22.1. Constraints

Constraint	Meaning
`PHPUnit_Framework_Constraint_Attribute attribute(PHPUnit_Framework_Constraint $constraint, $attributeName)`	Constraint that applies another constraint to an attribute of a class or an object.
`PHPUnit_Framework_Constraint_IsAnything anything()`	Constraint that accepts any input value.
`PHPUnit_Framework_Constraint_ArrayHasKey arrayHasKey(mixed $key)`	Constraint that asserts that the array it is evaluated for has a given key.
`PHPUnit_Framework_Constraint_TraversableContains contains(mixed $value)`	Constraint that asserts that the `array` or object that implements the `Iterator` interface it is evaluated for contains a given value.
`PHPUnit_Framework_Constraint_IsEqual equalTo($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.
`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`.

Constraint	**Meaning**
`PHPUnit_Framework_Constraint_IsTrue isTrue()`	Constraint that asserts that the value it is evaluated is `TRUE`.
`PHPUnit_Framework_Constraint_IsType isType(string $type)`	Constraint that asserts that the value it is evaluated for is of a specified type.
`PHPUnit_Framework_Constraint_LessThan lessThan(mixed $value)`	Constraint that asserts that the value it is evaluated for is smaller than a given value.
`PHPUnit_Framework_Constraint_Or lessThanOrEqual(mixed $value)`	Constraint that asserts that the value it is evaluated for is smaller than or equal to a given value.
`logicalAnd()`	Logical AND.
`logicalNot(PHPUnit_Framework_Constraint $constraint)`	Logical NOT.
`logicalOr()`	Logical OR.
`logicalXor()`	Logical XOR.
`PHPUnit_Framework_Constraint_PCREMatch matchesRegularExpression(string $pattern)`	Constraint that asserts that the string it is evaluated for matches a regular expression.
`PHPUnit_Framework_Constraint_StringContains stringContains(string $string, bool $case)`	Constraint that asserts that the string it is evaluated for contains a given string.
`PHPUnit_Framework_Constraint_StringEndsWith stringEndsWith(string $suffix)`	Constraint that asserts that the string it is evaluated for ends with a given suffix.
`PHPUnit_Framework_Constraint_StringStartsWith stringStartsWith(string $prefix)`	Constraint that asserts that the string it is evaluated for starts with a given prefix.

assertTrue()

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

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

Example 22.35. Usage of assertTrue()

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

```

PHPUnit 3.4.4 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 3.4.4 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`.

`$expected` can be one of these constants:

- `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"`)

`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 22.36. Usage of assertType()

```
<?php
class TypeTest extends PHPUnit_Framework_TestCase
{
    public function testFailure()
    {
        $this->assertType(PHPUnit_Framework_Constraint_IsType::TYPE_STRING, 2204);
    }
    public function testFailure2()
    {
        $this->assertType('string', 2204);
    }

    public function testFailure3()
    {
        $this->assertType('Exception', new stdClass);
    }
}
?>
```

```
PHPUnit 3.4.4 by Sebastian Bergmann.

FFF

Time: 0 seconds

There were 3 failures:

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

2) TypeTest::testFailure2
Failed asserting that <integer:2204> is of type "string".
/home/sb/TypeTest.php:11

3) TypeTest::testFailure3
Failed asserting that <stdClass> is an instance of class "Exception".
/home/sb/TypeTest.php:16

FAILURES!
Tests: 3, Assertions: 3, Failures: 3.phpunit TypeTest
PHPUnit 3.4.4 by Sebastian Bergmann.

FFF

Time: 0 seconds

There were 3 failures:

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

2) TypeTest::testFailure2
Failed asserting that <integer:2204> is of type "string".
/home/sb/TypeTest.php:11

3) TypeTest::testFailure3
Failed asserting that <stdClass> is an instance of class "Exception".
/home/sb/TypeTest.php:16

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

assertXmlFileEqualsXmlFile()

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

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

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

Example 22.37. Usage of assertXmlFileEqualsXmlFile()

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

```

PHPUnit 3.4.4 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 3.4.4 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 22.38. Usage of assertXmlStringEqualsXmlFile()

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

```

PHPUnit 3.4.4 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 3.4.4 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 22.39. Usage of assertXmlStringEqualsXmlString()

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

```

PHPUnit 3.4.4 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 3.4.4 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.
```

You may find that you need other assertions than these to compare objects specific to your project. Create your own `Assert` class to contain these assertions to simplify your tests.

Failing assertions all call a single bottleneck method, `fail(string $message)`, which throws an `PHPUnit_Framework_AssertionFailedError`. There is also a variant which takes no parameters. Call `fail()` explicitly when your test encounters an error. The test for an expected exception is an example. Table 22.2, “Bottleneck Methods” lists the bottlenext methods in PHPUnit.

Table 22.2. Bottleneck Methods

Method	Meaning
`void fail()`	Reports an error.
`void fail(string $message)`	Reports an error identified by `$message`.

`markTestIncomplete()` and `markTestSkipped()` are convenience methods for marking a test as being incomplete or skipped.

Table 22.3. Marking a test as being incomplete or skipped

Method	Meaning
`void markTestIncomplete(string $message)`	Marks the current test as being incomplete, `$message` is optional.
`void markTestSkipped(string $message)`	Marks the current test as being skipped, `$message` is optional.

Although unit tests are about testing the public interface of a class, you may sometimes want to test the values of non-public attributes. The `readAttribute()` method enables you to do this and returns the value of a given (static) attribute from a given class or object.

Table 22.4. Accessing non-public attributes

Method	Meaning
`Mixed readAttribute($classOrObject, $attributeName)`	Returns the value of a given (static) attribute (`$attributeName`) of a class or of an object. This also works for attributes that are declared `protected` or `private`.

PHPUnit_Framework_Test

`PHPUnit_Framework_Test` is the generic interface used by all objects that can act as tests. Implementors may represent one or more tests. The two methods are shown in Table 22.5, "Implementor Methods".

Table 22.5. Implementor Methods

Method	Meaning
`int count()`	Return the number of tests.
`void run(PHPUnit_Framework_TestResult $result)`	Run the tests and report the results on `$result`.

`PHPUnit_Framework_TestCase` and `PHPUnit_Framework_TestSuite` are the two most prominent implementors of `PHPUnit_Framework_Test`. You can implement `PHPUnit_Framework_Test` yourself. The interface is kept small intentionally so it will be easy to implement.

PHPUnit_Framework_TestCase

Your test case classes will inherit from `PHPUnit_Framework_TestCase`. Most of the time, you will run tests from automatically created test suites. In this case, each of your tests should be represented by a method named `test*` (by convention).

`PHPUnit_Framework_TestCase` implements `PHPUnit_Framework_Test::count()` so that it always returns `1`. The implementation of `PHPUnit_Framework_Test::run(PHPUnit_Framework_TestResult $result)` in this class runs `setUp()`, runs the test method, and then runs `tearDown()`, reporting any exceptions to the `PHPUnit_Framework_TestResult`.

Table 22.6, "TestCase" shows the methods provided by `PHPUnit_Framework_TestCase`.

Table 22.6. TestCase

Method	Meaning
`__construct()`	Creates a test case.
`__construct(string $name)`	Creates a named test case. Names are used to print the test case and often as the name of the test method to be run by reflection.
`string getName()`	Return the name of the test case.
`void setName($name)`	Set the name of the test case.
`PHPUnit_Framework_TestResult run(PHPUnit_Framework_TestResult $result)`	Convenience method to run the test case and report it in `$result`.
`void runTest()`	Override with a testing method if you do not want the testing method to be invoked by reflection.
`object getMock($originalClassName, [array $methods, [array $arguments, [string $mockClassName, [boolean $callOriginal-`	Returns a mock object (see Chapter 11, *Test Doubles*) for the class specified by `$originalClassName`. By default, all methods of the given class are mocked. When the second (optional) parameter is provided, only the methods

Method	Meaning
`Constructor, [boolean $callOriginalClone, [boolean $callAutoload]]]]]])`	whose names are in the array are mocked. The third (optional) parameter may hold a parameter array that is passed to the mock object's constructor. The fourth (optional) parameter can be used to specify a class name for the mock object. The fifth (optional) parameter can be used to disable the call to the original object's `__construct()` method. The sixth (optional) parameter can be used to disable the call to the original object's `__clone()` method. The seventh (optional) parameter can be used to disable `__autoload()` during mock object creation.
`object getMockForAbstractClass($originalClassName, [array $arguments, [string $mockClassName, [boolean $callOriginalConstructor, [boolean $callOriginalClone, [boolean $callAutoload]]]]])`	Returns a mock object (see Chapter 11, *Test Doubles*) for the abstract class specified by `$originalClassName`. All abstract methods of the given abstract class are mocked. This allows for testing the concrete methods of an abstract class.
`object getMockFromWsdl($wsdlFile, [string $originalClassName, [string $mockClassName, [array $methods, [boolean $callOriginalConstructor)`	Returns a mock object (see Chapter 11, *Test Doubles*) for the SOAP web services described in `$wsdlFile`.
`void iniSet(string $varName, mixed $newValue)`	This method is a wrapper for the `ini_set()` [http://www.php.net/ini_set] function that automatically resets the modified `php.ini` setting to its original value after the test is run.
`void setLocale(integer $category, string $locale, ...)`	This method is a wrapper for the `setlocale()` [http://www.php.net/setlocale] function that automatically resets the locale to its original value after the test is run.

There are two template methods -- `setUp()` and `tearDown()` -- you can override to create and dispose of the objects against which you are going to test. Table 22.7, “Template Methods” shows these methods. The template methods `assertPreConditions()` and `assertPostConditions()` can be used to define assertions that should be performed by all tests of a test case class.

Table 22.7. Template Methods

Method	Meaning
`void setUp()`	Override to set up the fixture, for example create an object graph.
`void assertPreConditions()`	Override to perform assertions shared by all tests of a test case class. This method is called before the execution of a test starts and after setUp() is called.
`void assertPostConditions()`	Override to perform assertions shared by all tests of a test case class. This method is called before the execution of a test ends and before tearDown() is called.
`void tearDown()`	Override to tear down the fixture, for example clean up an object graph.

PHPUnit_Framework_TestSuite

A `PHPUnit_Framework_TestSuite` is a composite of `PHPUnit_Framework_Test`s. At its simplest, it contains a bunch of test cases, all of which are run when the suite is run. Since it is a composite, however, a suite can contain suites which can contain suites and so on, making it easy to combine tests from various sources and run them together.

In addition to the `PHPUnit_Framework_Test` methods -- `run(PHPUnit_Framework_TestResult $result)` and `count()` -- `PHPUnit_Framework_TestSuite` provides methods to create named or unnamed instances. Table 22.8, "Creating named or unnamed instances" shows the methods to create `PHPUnit_Framework_TestSuite` instances.

Table 22.8. Creating named or unnamed instances

Method	Meaning
`__construct()`	Return an empty test suite.
`__construct(string $theClass)`	Return a test suite containing an instance of the class named `$theClass` for each method in the class named `test*`. If no class of name `$theClass` exists an empty test suite named `$theClass` is returned.
`__construct(string $theClass, string $name)`	Return a test suite named `$name` containing an instance of the class named `$theClass` for each method in the class named `test*`.
`__construct(ReflectionClass $theClass)`	Return a test suite containing an instance of the class represented by `$theClass` for each method in the class named `test*`.
`__construct(ReflectionClass $theClass, $name)`	Return a test suite named `$name` containing an instance of the class represented by `$theClass` for each method in the class named `test*`.
`string getName()`	Return the name of the test suite.
`void setName(string $name)`	Set the name of the test suite.
`void markTestSuiteSkipped(string $message)`	Marks the current test suite as being skipped, `$message` is optional.

`PHPUnit_Framework_TestSuite` also provides methods for adding and retrieving `PHPUnit_Framework_Test`s, as shown in Table 22.9, "Adding and retrieving tests".

Table 22.9. Adding and retrieving tests

Method	Meaning
`void addTestSuite(PHPUnit_Framework_TestSuite $suite)`	Add another test suite to the test suite.
`void addTestSuite(string $theClass)`	Add a test suite containing an instance of the class named `$theClass` for each method in the class named `test*` to the test suite.
`void addTestSuite(ReflectionClass $theClass)`	Add a test suite containing an instance of the class represented by `$theClass` for each method in the class named `test*` to the test suite.

Method	Meaning
`void addTest(PHPUnit_Framework_Test $test)`	Add `$test` to the suite.
`void addTestFile(string $filename)`	Add the tests that are defined in the class(es) of a given sourcefile to the suite.
`void addTestFiles(array $filenames)`	Add the tests that are defined in the classes of the given sourcefiles to the suite.
`int testCount()`	Return the number of tests directly (not recursively) in this suite.
`PHPUnit_Framework_Test[] tests()`	Return the tests directly in this suite.
`PHPUnit_Framework_Test testAt(int $index)`	Return the test at the `$index`.

Example 22.40, “Creating and running a test suite” shows how to create and run a test suite.

Example 22.40. Creating and running a test suite

```php
<?php
require_once 'PHPUnit/Framework.php';

require_once 'ArrayTest.php';

// Create a test suite that contains the tests
// from the ArrayTest class.
$suite = new PHPUnit_Framework_TestSuite('ArrayTest');

// Run the tests.
$suite->run();
?>
```

Chapter 7, *Organizing Tests* shows how to use the `PHPUnit_Framework_TestSuite` class to organize test suites by hierarchically composing test cases.

The `PHPUnit_Framework_TestSuite` class provides two template methods -- `setUp()` and `tearDown()` -- that are called before and after the tests of a test suite are run, respectively.

Table 22.10. Template Methods

Method	Meaning
`void setUp()`	Called before the first test of the test suite is run.
`void tearDown()`	Called after the last test of the test suite has been run.

PHPUnit_Framework_TestResult

While you are running all these tests, you need somewhere to store all the results: how many tests ran, which failed, and how long they took. `PHPUnit_Framework_TestResult` collects these results. A single `PHPUnit_Framework_TestResult` is passed around the whole tree of tests; when a test runs or fails, the fact is noted in the `PHPUnit_Framework_TestResult`. At the end of the run, `PHPUnit_Framework_TestResult` contains a summary of all the tests.

`PHPUnit_Framework_TestResult` is also a subject than can be observed by other objects wanting to report test progress. For example, a graphical test runner might observe the `PHPUnit_Framework_TestResult` and update a progress bar every time a test starts.

Table 22.11, “TestResult” summarizes the API of `PHPUnit_Framework_TestResult`.

Table 22.11. TestResult

Method	Meaning
`void addError(PHPUnit_Framework_Test $test, Exception $e)`	Record that running `$test` caused `$e` to be thrown unexpectedly.
`void addFailure(PHPUnit_Framework_Test $test, PHPUnit_Framework_AssertionFailedError $e)`	Record that running `$test` caused `$e` to be thrown unexpectedly.
`PHPUnit_Framework_TestFailure[] errors()`	Return the errors recorded.
`PHPUnit_Framework_TestFailure[] failures()`	Return the failures recorded.
`PHPUnit_Framework_TestFailure[] notImplemented()`	Return the incomplete test cases recorded.
`int errorCount()`	Return the number of errors.
`int failureCount()`	Return the number of failures.
`int notImplementedCount()`	Return the number of incomplete test cases.
`int count()`	Return the total number of test cases run.
`boolean wasSuccessful()`	Return whether or not all tests ran successfully.
`boolean allCompletlyImplemented()`	Return whether or not all tests were completely implemented.
`void collectCodeCoverageInformation(bool $flag)`	Enables or disables the collection of Code Coverage information.
`array getCodeCoverageInformation()`	Return the code coverage information collected.

If you want to register as an observer of a `PHPUnit_Framework_TestResult`, you need to implement `PHPUnit_Framework_TestListener`. To register, call `addListener()`, as shown in Table 22.12, “TestResult and TestListener”.

Table 22.12. TestResult and TestListener

Method	Meaning
`void addListener(PHPUnit_Framework_TestListener $listener)`	Register `$listener` to receive updates as results are recorded in the test result.
`void removeListener(PHPUnit_Framework_TestListener $listener)`	Unregister `$listener` from receiving updates.

Table 22.13, “TestListener Callbacks” shows the methods that test listeners implement; also see Example 23.3, “A simple test listener”.

Table 22.13. TestListener Callbacks

Method	Meaning
`void addError(PHPUnit_Framework_Test $test, Exception $e)`	`$test` has thrown `$e`.
`void addFailure(PHPUnit_Framework_Test $test, PHPUnit_Framework_AssertionFailedError $e)`	`$test` has failed an assertion, throwing a kind of `PHPUnit_Framework_AssertionFailedError`.

Method	**Meaning**
`void addIncompleteTest(PHPUnit_Framework_Test $test, Exception $e)`	`$test` is an incomplete test.
`void addSkippedTest(PHPUnit_Framework_Test $test, Exception $e)`	`$test` is a test that has been skipped.
`void startTestSuite(PHPUnit_Framework_TestSuite $suite)`	`$suite` is about to be run.
`void endTestSuite(PHPUnit_Framework_TestSuite $suite)`	`$suite` has finished running.
`void startTest(PHPUnit_Framework_Test $test)`	`$test` is about to be run.
`void endTest(PHPUnit_Framework_Test $test)`	`$test` has finished running.

Package Structure

Many of the classes mentioned so far in this book come from `PHPUnit/Framework`. Here are all the packages in PHPUnit:

- `PHPUnit/Framework`

  The basic classes in PHPUnit.

- `PHPUnit/Extensions`

  Extensions to the PHPUnit framework.

- `PHPUnit/Runner`

  Abstract support for running tests.

- `PHPUnit/TextUI`

  The text-based test runner.

- `PHPUnit/Util`

  Utility classes used by the other packages.

Chapter 23. 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 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.

Assert Classes

Write your own class with assertions special to your purpose.

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 23.1, "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 23.1. The RepeatedTest Decorator

```php
<?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 23.2, “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 23.2. A data-driven test

```
<?php
require_once 'PHPUnit/Framework.php';
require_once 'PHPUnit/Util/Timer.php';
require_once 'PHPUnit/TextUI/TestRunner.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);
            PHPUnit_Util_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, PHPUnit_Util_Timer::stop());
            }

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

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

        return $result;
    }
}

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

```
PHPUnit 3.4.2 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.
```

Subclass PHPUnit_Framework_TestResult

By passing a special-purpose `PHPUnit_Framework_TestResult` object to the `run()` method, you can change the way tests are run and what result data gets collected.

Implement PHPUnit_Framework_TestListener

You do not necessarily need to write a whole subclass of `PHPUnit_Framework_TestResult` in order to customize it. Most of the time, it will suffice to implement a new `PHPUnit_Framework_TestListener` (see Table 22.13, “TestListener Callbacks”) and attach it to the `PHPUnit_Framework_TestResult` object, before running the tests.

Example 23.3, “A simple test listener” shows a simple implementation of the `PHPUnit_Framework_TestListener` interface.

Example 23.3. A simple test listener

```
<?php
require_once 'PHPUnit/Framework.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_AssertionFailedError $e,
             $time)
  {
    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()
    );
  }
}
?>
```

Example 23.4, “Running and observing a test suite” shows how to run and observe a test suite.

Example 23.4. Running and observing a test suite

```
<?php
require_once 'PHPUnit/Framework.php';

require_once 'ArrayTest.php';
require_once 'SimpleTestListener.php';

// Create a test suite that contains the tests
// from the ArrayTest class.
$suite = new PHPUnit_Framework_TestSuite('ArrayTest');

// Create a test result and attach a SimpleTestListener
// object as an observer to it.
$result = new PHPUnit_Framework_TestResult;
$result->addListener(new SimpleTestListener);

// Run the tests.
$suite->run($result);
?>
```

```
TestSuite 'ArrayTest' started.
Test 'testNewArrayIsEmpty' started.
Test 'testNewArrayIsEmpty' ended.
Test 'testArrayContainsAnElement' started.
Test 'testArrayContainsAnElement' ended.
TestSuite 'ArrayTest' ended.
```

New Test Runner

If you need different feedback from the test execution, write your own test runner, interactive or not. The abstract `PHPUnit_Runner_BaseTestRunner` class, which the `PHPUnit_TextUI_TestRunner` class (the PHPUnit command-line test runner) inherits from, can be a starting point for this.

Appendix A. Assertions

Table A.1, “Assertions” shows all the varieties of assertions.

Table A.1. Assertions

Assertion
assertArrayHasKey($key, array $array, $message = '')
assertArrayNotHasKey($key, array $array, $message = '')
assertAttributeContains($needle, $haystackAttributeName, $haystackClassOrObject, $message = '', $ignoreCase = FALSE)
assertAttributeContainsOnly($type, $haystackAttributeName, $haystackClassOrObject, $isNativeType = NULL, $message = '')
assertAttributeEquals($expected, $actualAttributeName, $actual-ClassOrObject, $message = '', $delta = 0, $maxDepth = 10, $canoni-calizeEol = FALSE, $ignoreCase = FALSE)
assertAttributeGreaterThan($expected, $actualAttributeName, $actu-alClassOrObject, $message = '')
assertAttributeGreaterThanOrEqual($expected, $actualAttributeName, $actualClassOrObject, $message = '')
assertAttributeLessThan($expected, $actualAttributeName, $actual-ClassOrObject, $message = '')
assertAttributeLessThanOrEqual($expected, $actualAttributeName, $actualClassOrObject, $message = '')
assertAttributeNotContains($needle, $haystackAttributeName, $haystackClassOrObject, $message = '', $ignoreCase = FALSE)
assertAttributeNotContainsOnly($type, $haystackAttributeName, $haystackClassOrObject, $isNativeType = NULL, $message = '')
assertAttributeNotEquals($expected, $actualAttributeName, $actual-ClassOrObject, $message = '', $delta = 0, $maxDepth = 10, $canoni-calizeEol = FALSE, $ignoreCase = FALSE)
assertAttributeNotSame($expected, $actualAttributeName, $actual-ClassOrObject, $message = '')
assertAttributeNotType($expected, $attributeName, $classOrObject, $message = '')
assertAttributeSame($expected, $actualAttributeName, $actualClas-sOrObject, $message = '')
assertAttributeType($expected, $attributeName, $classOrObject, $message = '')
assertClassHasAttribute($attributeName, $className, $message = '')
assertClassHasStaticAttribute($attributeName, $className, $message = '')
assertClassNotHasAttribute($attributeName, $className, $message = '')
assertClassNotHasStaticAttribute($attributeName, $className, $mes-sage = '')
assertContains($needle, $haystack, $message = '', $ignoreCase = FALSE)

Assertion
assertContainsOnly($type, $haystack, $isNativeType = NULL, $message = '')
assertEqualXMLStructure(DOMNode $expectedNode, DOMNode $actualNode, $checkAttributes = FALSE, $message = '')
assertEquals($expected, $actual, $message = '', $delta = 0, $maxDepth = 10, $canonicalizeEol = FALSE, $ignoreCase = FALSE)
assertFalse($condition, $message = '')
assertFileEquals($expected, $actual, $message = '', $canonicalizeEol = FALSE, $ignoreCase = FALSE)
assertFileExists($filename, $message = '')
assertFileNotEquals($expected, $actual, $message = '', $canonicalizeEol = FALSE, $ignoreCase = FALSE)
assertFileNotExists($filename, $message = '')
assertGreaterThan($expected, $actual, $message = '')
assertGreaterThanOrEqual($expected, $actual, $message = '')
assertLessThan($expected, $actual, $message = '')
assertLessThanOrEqual($expected, $actual, $message = '')
assertNotContains($needle, $haystack, $message = '', $ignoreCase = FALSE)
assertNotContainsOnly($type, $haystack, $isNativeType = NULL, $message = '')
assertNotEquals($expected, $actual, $message = '', $delta = 0, $maxDepth = 10, $canonicalizeEol = FALSE, $ignoreCase = FALSE)
assertNotNull($actual, $message = '')
assertNotRegExp($pattern, $string, $message = '')
assertNotSame($expected, $actual, $message = '')
assertNotTag($matcher, $actual, $message = '', $isHtml = TRUE)
assertNotType($expected, $actual, $message = '')
assertNull($actual, $message = '')
assertObjectHasAttribute($attributeName, $object, $message = '')
assertObjectNotHasAttribute($attributeName, $object, $message = '')
assertRegExp($pattern, $string, $message = '')
assertSame($expected, $actual, $message = '')
assertSelectCount($selector, $count, $actual, $message = '', $isHtml = TRUE)
assertSelectEquals($selector, $content, $count, $actual, $message = '', $isHtml = TRUE)
assertSelectRegExp($selector, $pattern, $count, $actual, $message = '', $isHtml = TRUE)
assertStringEndsNotWith($suffix, $string, $message = '')
assertStringEndsWith($suffix, $string, $message = '')
assertStringEqualsFile($expectedFile, $actualString, $message = '', $canonicalizeEol = FALSE, $ignoreCase = FALSE)

Assertion
assertStringNotEqualsFile($expectedFile, $actualString, $message = '', $canonicalizeEol = FALSE, $ignoreCase = FALSE)
assertStringStartsNotWith($prefix, $string, $message = '')
assertStringStartsWith($prefix, $string, $message = '')
assertTag($matcher, $actual, $message = '', $isHtml = TRUE)
assertThat($value, PHPUnit_Framework_Constraint $constraint, $mes-sage = '')
assertTrue($condition, $message = '')
assertType($expected, $actual, $message = '')
assertXmlFileEqualsXmlFile($expectedFile, $actualFile, $message = '')
assertXmlFileNotEqualsXmlFile($expectedFile, $actualFile, $message = '')
assertXmlStringEqualsXmlFile($expectedFile, $actualXml, $message = '')
assertXmlStringEqualsXmlString($expectedXml, $actualXml, $message = '')
assertXmlStringNotEqualsXmlFile($expectedFile, $actualXml, $mes-sage = '')
assertXmlStringNotEqualsXmlString($expectedXml, $actualXml, $mes-sage = '')

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));
}
```

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

@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 data providers”). 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 4.5, “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.

@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="false"
         backupStaticAttributes="true"
         bootstrap="/path/to/bootstrap.php"
         colors="false"
         convertErrorsToExceptions="true"
         convertNoticesToExceptions="true"
         convertWarningsToExceptions="true"
         processIsolation="true"
         stopOnFailure="true"
         syntaxCheck="false"
         testSuiteLoaderClass="PHPUnit_Runner_StandardTestSuiteLoader">
  <!-- ... -->
</phpunit>
```

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

- `--no-globals-backup`
- `--static-backup`
- `--bootstrap /path/to/bootstrap.php`
- `--colors`
- `--process-isolation`
- `--stop-on-failure`
- `--no-syntax-check`
- `--loader PHPUnit_Runner_StandardTestSuiteLoader`

The `convertErrorsToExceptions`, `convertNoticesToExceptions`, and `convertWarningsToExceptions` attributes have no equivalent TextUI test runner switch.

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>
```

The XML configuration above corresponds to using the `PHPUnit_Util_Filter` class as follows:

```
PHPUnit_Util_Filter::addDirectoryToFilter(
  '/path/to/files', '.php'
);

PHPUnit_Util_Filter::addFileToFilter('/path/to/file');

PHPUnit_Util_Filter::removeDirectoryFromFilter(
  '/path/to/files', '.php'
);

PHPUnit_Util_Filter::removeFileFromFilter('/path/to/file');
```

```
PHPUnit_Util_Filter::addDirectoryToWhitelist(
  '/path/to/files', '.php'
);

PHPUnit_Util_Filter::addFileToWhitelist('/path/to/file');

PHPUnit_Util_Filter::removeDirectoryFromWhitelist(
  '/path/to/files', '.php'
);

PHPUnit_Util_Filter::removeFileFromWhitelist('/path/to/file');
```

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-xml" 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-xml /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.

```
<php>
  <ini name="foo" value="bar"/>
  <const name="foo" value="bar"/>
  <var name="foo" value="bar"/>
</php>
```

The XML configuration above corresponds to the following PHP code:

```
ini_set('foo', 'bar');
declare('foo', 'bar');
$GLOBALS['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, 22
$backupStaticAttributesBlacklist, 22
@assert, 72, 143
@backupGlobals, 22, 143, 143
@backupStaticAttributes, 22, 144, 144
@codeCoverageIgnoreEnd, 67
@codeCoverageIgnoreStart, 67
@covers, 66, 144
@dataProvider, 8, 9, 10, 145
@depends, 6, 9, 10, 145
@expectedException, 10, 146
@group, , , , 146
@outputBuffering, 146, 146
@runInSeparateProcess, 147
@runTestsInSeparateProcesses, 147
@test, , 147
@testdox, 147
@ticket, 147

A

addTest(),
addTestFile(),
addTestFiles(),
addTestSuite(),
Agile Documentation, , , 69
Annotation, 6, , 6, 8, 9, 10, 10, , , , 66, 67, 72, 143
anything(),
Apache Ant, 84, 85, 88
Apache Maven, 85
arrayHasKey(),
assertArrayHasKey(), 93,
assertArrayNotHasKey(), 93,
assertAttributeContains(), 95,
assertAttributeContainsOnly(), 97,
assertAttributeEquals(), 98,
assertAttributeGreaterThan(), 107,
assertAttributeGreaterThanOrEqual(), 108,
assertAttributeLessThan(), 109,
assertAttributeLessThanOrEqual(), 110,
assertAttributeNotContains(), 95,
assertAttributeNotContainsOnly(), 97,
assertAttributeNotEquals(), 98,
assertAttributeNotSame(), 113,
assertAttributeNotType(), 123,
assertAttributeSame(), 113,
assertAttributeType(), 123,
assertClassHasAttribute(), 94,
assertClassHasStaticAttribute(), 94,
assertClassNotHasAttribute(), 94,
assertClassNotHasStaticAttribute(), 94,

assertContains(), 95,
assertContainsOnly(), 97,
assertEquals(), 98,
assertEqualXMLStructure(), 98,
assertFalse(), 104,
assertFileEquals(), 105,
assertFileExists(), 106,
assertFileNotEquals(), 105,
assertFileNotExists(), 106,
assertGreaterThan(), 107,
assertGreaterThanOrEqual(), 108,
Assertion, 143
Assertions, 2, 91, 135
assertLessThan(), 109,
assertLessThanOrEqual(), 110,
assertNotContains(), 95,
assertNotContainsOnly(), 97,
assertNotEquals(), 98,
assertNotNull(), 110,
assertNotRegExp(), 112,
assertNotSame(), 113,
assertNotTag(), 118,
assertNotType(), 123,
assertNull(), 110,
assertObjectHasAttribute(), 111,
assertObjectNotHasAttribute(), 111,
assertPostConditions(), 19,
assertPreConditions(), 19,
assertRegExp(), 112,
assertSame(), 113,
assertSelectCount(), 114,
assertSelectEquals(), 115,
assertSelectRegExp(), 115,
assertStringEndsNotWith(), 115,
assertStringEndsWith(), 115,
assertStringEqualsFile(), 116,
assertStringNotEqualsFile(), 116,
assertStringStartsNotWith(), 117,
assertStringStartsWith(), 117,
assertTag(), 118,
assertThat(), 120,
assertTrue(), 122,
assertType(), 123,
assertXmlFileEqualsXmlFile(), 124,
assertXmlFileNotEqualsXmlFile(), 124,
assertXmlStringEqualsXmlFile(), 126,
assertXmlStringEqualsXmlString(), 127,
assertXmlStringNotEqualsXmlFile(), 126,
assertXmlStringNotEqualsXmlString(), 127,
Atlassian Bamboo, 87
attribute(),
attributeEqualTo(),
Automated Documentation, 69
Automated Test, 2

B

Behaviour-Driven Development, , , 58
Blacklist, 68, 149

C

classHasAttribute(),
classHasStaticAttribute(),
Code Coverage, , , , 64, 68, 144, 149
Configuration, ,
Constant, 151
contains(),
CruiseControl, 88

D

Data-Driven Tests, 136
Database, 31
DbUnit, 31
Defect Localization, 7
Depended-On Component, 41
Design-by-Contract, 53, 91
Documenting Assumptions, 69
Domain-Driven Design, 58

E

equalTo(),
Error, 13
Error Handler, 12
expects(), 42, 42, 43, 43
Extreme Programming, 53, 58, 69

F

fail(),
Failure, 13
fileExists(),
Fixture, 18
Fluent Interface, 41

G

getMock(), 42, 42, 43, 43
getMockForAbstractClass(), 46
getMockFromWsdl(), 46
Global Variable, 22, 151
greaterThan(),
greaterThanOrEqual(),

H

hasAttribute(),

I

identicalTo(),
include_path,
Incomplete Test, 38, 71, 128
isFalse(),
isInstanceOf(),
isNull(),
isTrue(),
isType(),

J

JSON,

L

lessThan(),
lessThanOrEqual(),
Logfile, ,
Logging, 81, 150
logicalAnd(),
logicalNot(),
logicalOr(),
logicalXor(),

M

markTestIncomplete(),
markTestSkipped(),
matchesRegularExpression(),
method(), 42, 42, 43, 43
Mock Object, 44, 45

O

Observer Pattern, 133
onNotSuccessfulTest(), 19

P

Phing, 85
PHP Error, 12
PHP Notice, 12
PHP Warning, 12
php.ini, 151
phpUnderControl, 90
PHPUnit_Extensions_OutputTestCase, 29
PHPUnit_Extensions_RepeatedTest, 135
PHPUnit_Extensions_SeleniumTestCase, 75
PHPUnit_Extensions_Story_TestCase, 58
PHPUnit_Extensions_TestDecorator, 135
PHPUnit_Extensions_TestSetup, 135
PHPUnit_Framework_Assert, 56, 91, 135
PHPUnit_Framework_Error, 12
PHPUnit_Framework_Error_Notice, 12
PHPUnit_Framework_Error_Warning, 12
PHPUnit_Framework_IncompleteTest, 38
PHPUnit_Framework_IncompleteTestError, 38
PHPUnit_Framework_Test, 129, 136
PHPUnit_Framework_TestCase, 6, 129, 135
PHPUnit_Framework_TestListener, 133, 137, 150
PHPUnit_Framework_TestResult, 132, 137, 137, 139
PHPUnit_Framework_TestSuite, 26, 131, 139
PHPUnit_Runner_TestSuiteLoader,
PHPUnit_Util_Filter, 68, 149
Process Isolation,

R

readAttribute(),
Refactoring, 51
Report,
returnArgument(), 42
returnCallback(), 43
returnValue(), 42

S

Selenium RC, 75, 151
setUp(), 18, 19, 19, 27, ,
setUpBeforeClass(), 19, 19
Skeleton Generator, , , 71
Skipped Test, 128
stringContains(),
stringEndsWith(),
stringStartsWith(),
Stub, 41
Stubs, 70
Syntax Check, 17
System Under Test, 41

T

tearDown(), 18, 19, 19, 27, ,
tearDownAfterClass(), 19, 19
Template Method, 18, 19, 19, 19, 27
Test Dependencies, 6
Test Double, 41
Test Groups, , , , 149
Test Isolation, , , , 22
Test Listener, 150
Test Suite, 24, 148
Test-Driven Development, 53, 58
Test-First Programming, 53
TestDox, 69, 147
throwException(), 43

U

Unit Test, 1, 53

W

Whitelist, 68, 149
will(), 42, 42, 43, 43

X

Xdebug, 64
XML Configuration, 25

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