TESTING IS NOT DIFFICULT!

Practical Php Testing

by Giorgio Sironi

Don't let your php be eaten alive

Invisible to the eye blog
http://giorgiosironi.blogspot.com

Table of Contents

Introduction

This practical testing book is aimed to php developers and features the articles from the Practical php testing series of my blog, plus new content only available in this book:

- **preface** to explain why we should care about testing php applications.
- **bonus chapter on TDD theory and a case study.**
- **code samples**, some of whom were originally kept on pastebin.com; this php code complements many chapters. Code is highlighted with a special background.
- Sets of **TDD exercises** at the end of each chapter. Test-Driven Development is a practice where tests are written before the production code. This is not a book specifically on TDD, but these exercises will help you grasp the fundamentals of this methodology and its advantages: testable code, effective test suite and the good design that ensues. The TDD theory (chapter 10) closes the circle.
- A **glossary** that substitutes external links to wiki and other posts, to not interrupt your reading with terms lookup.

About the author

My name is Giorgio Sironi and I am an undergraduate 29+ student at the Politecnico di Milano in Italy (the equivalent of a 3.9 GPA in United States). I attend the faculty of Ingegneria Informatica (Computer Engineering). I've worked and still work in the web development field and I'm the creator of Ossigeno Cms and NakedPhp, that I use to create management application (e-school, business) and websites.

You probably have downloaded this book from my blog, Invisible to the eye:

http://giorgiosironi.blogspot.com

If not, no problem! What is important is that you have access to the information you need to grow as a developer or engineer. Come visit the blog for more discussions on testing and object-oriented architecture.

Contacts

You are free to contact me on my blog, or in other places on the web:

piccoloprincipeazzurro at gmail dot com

http://twitter.com/giorgiosironi

Donations

If you find helpful my work and you want to support it someway, I accept donations:

http://doiop.com/giorgiosironipaypal

Or, just in case the redirected url above does not work:

https://www.paypal.com/cgi-bin/webscr?cmd=_donations&business=WF7MNWBMZE9KQ&lc=GB&item_name=Giorgio%20Sironi&item_number=invisibletotheeye&currency_code=EUR&bn=PP%2dDonationsBF%3abtn_donate_LG%2egif%3aNonHosted

Errata

Practical Php Testing 1st edition (December 2, 2009).

Errata web page: http://giorgiosironi.blogspot.com/2009/12/practical-php-testing-errata.html

Unless a further edition may be publicated, refer to the linked web page for corrections.

Disclaimer

This book has been written for educational and informational purposes and the author made every reasonable attempt to achieve accuracy of the content. The author assumes no responsibility for errors or omissions and is not liable for incorrect or no longer up-to-date information may cause to your work.

Copyright?

Acknowledgements

The ebook cover is a parody of an illustration from The Little Prince by Antoine de Saint-Exupéry, where an elephant is eaten by a snake in its entirety. The original elephant is replaced by a elePHPant image from a photography, courtesy of Raphael Dohms (http://twitter.com/rdohms).

All other images in this book are create from scratch or provenient from Wikimedia Commons.

This book is dedicated to Kiki, who tolerates me when I get in the *zone* and dive into programming without caring about the outside world.

Preface: why testing?

I don't like books prefaces that take me days to read before arriving to the real content, so I will be very brief here.

- How many times in the last month have you seen a **broken screen in the browser**?
- How many times did you have to **debug in the browser**, by looking at the output, inserting debug statements and breaking redirects?
- How many times did you perform manual testing, by loading a staging version of your application and **tried out different workflows in the browser**?

If the answer to these questions is more than *very few*, chances are that you should give automated testing a chance. Testing can be handled professionally, by writing a test suite that you can run at the push of a button and from the command line. If well-written, this suite will show you a list of localized errors and where to go to fix them. It will eliminate most of the debugging from your day.

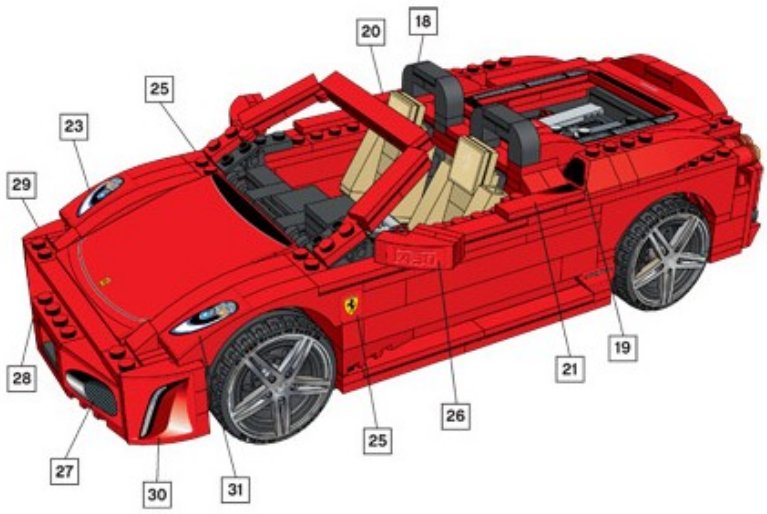
How testing does so? **Unit testing** is the testing flavor I prefer and which gives much advantages. Since Car examples are very popular, I would say that it is comparable to testing a car by dismantling it and running workbench tests on every single piece, with the difference that the pieces do not wear out and that the process takes from some milliseconds to a minute (since you can sometimes decide that you are only interested in exercising the engine or the doors). There are other testing paradigms, such as integration testing, but they are less effective in locating problems and bugs than class-focused test cases (since the typical unit under test is a single class), and in dictating an Api and an architecture.

Unit testing means freedom of development, without the fear of breaking something and not noticing it: running a full test on the components affected by code modifications takes only *seconds*. It also results in well-designed code, because cohesion and decoupling are testability requirements. Code is a children game: it can be a miniature car which you can't alter, or it can be a set of Lego bricks which you can always construct new buildings and starships with. I'm sure the majority of this book readers were the children that liked disassembling games.

Now that we are adults we do not play with Lego bricks anymore, but with classes. The difference resides in if we want to play with decoupled and testable classes or with an untestable mess.

Very often I heard people saying “this class is too difficult to test”. My response is: refactor the design to aid testability. Testing is not a tedious non-functional specification which forces the developer to add otherwise unuseful code: it is the driving force behind good architecture. As you will discover in TDD exercises, if we write code to satisfy a test suite, and then throw away the test suite, we are still left with a better product than with the result of cowboy coding. More focused classes, small interfaces and good Api are the by-products of unit testing.

Testing is not a code base detail in this book's weltanschauung: it is the starting point.



Chapter 1: PHPUnit usage

What is *unit testing* and why a php programmer should adopt it? It may seem simple, but testing is the only way to ensure your work is completed and you will not called in the middle of the night by a client whose website is going nuts. The need for quality is particularly present in the php environment, where it is very simple to deploy an interpreted script, but **it is also very simple to break something**: a missing semicolon in a common file can halt the entire application.
Unit testing is the process of writing tests which exercise the basic functionality of the smallest cohesive unit in php: a class. You can also write tests for procedures and functions, but unit testing works at its best with cohesive and decouple classes. Thus, object-oriented programming is a requirement; this process is contrapposed to functional and integration testing, which build medium and large graphs of objects when run. Unit testing istances one, or very few, units at the time, and this implies that unit tests are tipically easy to run in every environment and do not burden the system with heavy operations.

When the time comes for unit and functional testing, there's only one leader in the php world: PHPUnit. PHPUnit is the xUnit instance for the average and top-class php programmer; born as a port of JUnit, has quickly filled the gap with the original Java tool thanks to the potential of a dynamic language such as php. It even surpassed the features and the scope of JUnit providing a simple mocking framework (whose utility will be discovered during this journey in testing).

The most common and simplest way to install PHPUnit is as Pear package. On your development box, you need only the php binary and the *pear* executable.

```
sudo pear channel-discover pear.phpunit.de
sudo pear install phpunit/PHPUnit
```

Using a root shell (or a administrator on if you develop on other operating systems) instead of *sudo* is totally equivalent. Administrator permissions are usually mandatory to install Pear packages.

These commands tell *pear* to add the channel of PHPUnit developers, and to install the *PHPUnit* package from the now-available *phpunit* channel.

The chosen release is the latest stable package; at the time of this writing, the 3.4 version.
If the installation is successful, you now have a *phpunit* executable available from the command line. This is where you will run tests; if you use an IDE, probably there is a menu for running tests that will show you the command line output (and you should also install phpunit from the IDE interface to make it discover the new tool).

In this book, I prefer to use the command-line interface to not add other levels of indirection which could get in the way of learning. Everything you can do with an IDE like Netbeans or Eclipse, you can surely do in the command-line environment.

Before exploring the endless possibilities of testing, let's write our first one: the simplest test that could possibly work. I saved this php code in MyTest.php:

```
class MyTest extends PHPUnit_Framework_TestCase
{
    public function testComparesNumbers()
    {
        $this->assertTrue(1 == 1);
    }
}
```

What is a test? And a test case? A test case is constituted by a method by a class which extends *PHPUnit_Framework_TestCase*, which as its name tells is an abstract test case class provided by the PHPUnit framework. When developing a object-oriented application, you may want to start with **one test case per every class you want to test** (and if you're going the TDD way every class will be tested), thus there will be a 1:1 correspondence between classes and test cases. For the moment, we don't want to go too fast and we simply write a class that tests php common functionality.
Every test is a method which by convention starts with the keyword 'test'. Also for convention, the method name should tell what operation the system under test is capable, in this test .
Every method will be run independently in an isolated environment, and will make some assertion on what should happen. assertTrue() is one of the many assert*() method inherited from the abstract test case, which declares the test failed if an argument different from *true* is passed to it. The test as it is written now should pass. In fact, we can simply run it and find out:

```
[giorgio@Marty:~]$ phpunit MyTest.php
PHPUnit 3.4.0 by Sebastian Bergmann.

.

Time: 1 second

OK (1 test, 1 assertion)
```

Instant feedback is one of the pillars of TDD and of unit testing in general: the code in tests should instance your classes and exercising their functionality to ensure they don't blow up and respect the specifications. With the phpunit script, it's very simple and fast to run a test case or a group of them after you have made a change to your class and make sure you haven't break an existing feature.
The result of phpunit run is easily interpretable: a dot (.) for every test method which passed (without failed assertions), and a statistic of the number of tests and assertions encountered.
Let's try to **make it fail**, changing *1==1* to *1==0*:

```
[giorgio@Marty:~]$ phpunit MyTest.php
PHPUnit 3.4.0 by Sebastian Bergmann.

F

Time: 0 seconds

There was 1 failure:

1) MyTest::testComparesNumbers
Failed asserting that  is true.

/media/sdb1/giorgio/MyTest.php:6

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

For every failed test, you get an F instead of the point in the summary. Other letters can be encountered, for instance the E if the test caused no assertion to fail but it raised an exception or a php error. The only passed tests are the one which present a dot: you should strive for a long list of dots that fill more than one row.
You also get **a description of the assertion which has failed and in what test case, method and line it resides**. Since you can run many test cases as a single command, this is very useful to localize defects and regression.
This time the test has failed because it is bad written: zero is not equal to

one and php is right in giving out false. But assertTrue() does not know this and in the next chapter we'll write some tests which works upon userland code and it is in fact useful to detect if production classes are still satisfying all their requirements.

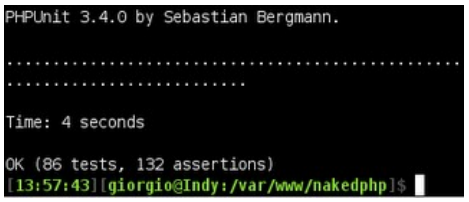
TDD exercises

1.1 Suppose you have to code a class that calculates the factorial of an integer N, which is the product of all integers from 1 to N. Write a failing test for it (do not code the class for now! Only the test. Suppose that the class and its methods already exists just like you want them to be.)

1.2 Add more test methods, which try different input numbers: 1, 4, 20. Verify that different factorials are calculated correctly (again, just the tests and no production code in this phase).

1.3 Write the production code needed to make all tests pass.

Chapter 2: write clever tests



In the previous chapter, we have discovered the syntax and the infrastructure needed to run a test with phpunit. Now we are going to show a practical example using a test case - production code couple of classes.
What we are going to test is the Spl class **ArrayIterator**; for the readers that do not know this class, it is a simple Iterator implementation which abstracts away a foreach on the elements of an array.
Of course it would be very useful to write the tests before the production code class, but this is not the time to talk about TDD and its advantages: let's simply write a few tests to ensure the implementation works as we desire. This is also **a common way to study the components of an object-oriented system**: read and understand its unit test and write more of them to verify our expectations about the production classes behavior are fulfilled.
Let's start with the simplest test case: an empty array.

```
class ArrayIteratorTest extends PHPUnit_Framework_TestCase
{
    public function testEmptyArrayIsNotIteratedOver()
    {
        $iterator = new ArrayIterator(array());
        foreach ($iterator as $element) {
            $this->fail();
        }
    }
}
```

The test case class is named ArrayIteratorTest, following the convention of using a 1:1 mapping from production classes to test ones. The test method simply creates a new instance of the system under test, setting up the situation to have it iterate over the empty array. If the execution path enter the foreach, the test fails, as the call to fail() is equivalent to assertTrue(false).
The next step is to cover other possible situations:

```
    public function testIteratesOverOneElementArrays()
    {
        $iterator = new ArrayIterator(array('foo'));
        $i = 0;
        foreach ($iterator as $element) {
```

```
            $this->assertEquals('foo', $element);
            $i++;
        }
        $this->assertEquals(1, $i);
    }
```

This test ensures that one-element numeric arrays are iterated correctly. The first assertion states that every element which is passed as the foreach argument is the element in the array, while the second that the foreach is executed only one time. You have probably guessed that assertEquals() confronts its two arguments with the == operator and fails if the result is *false*.
When it is not too computational expensive, we should strive to have the few possible assertions per method; so we can separate the test method testIteratesOverOneElementArrays() in two distinct ones:

```
    public function testIteratesOverOneElementArraysUsingValues()
    {
        $iterator = new ArrayIterator(array('foo'));
        foreach ($iterator as $element) {
            $this->assertEquals('foo', $element);
        }
    }

    public function testIteratesOneTimeOverOneElementArrays()
    {
        $iterator = new ArrayIterator(array('foo'));
        $i = 0;
        foreach ($iterator as $element) {
            $i++;
        }
        $this->assertEquals(1, $i);
    }
```

Now the two test methods are nearly independent and can fail independently to provide information on two different broken behaviors: not using the array values and iterating more than one time on an element. This is a very simple case, but try to think of this example of a methodology to **identify responsibilities of a production class**: the test names should describe what features the class provides at a good level of specification (and they are really used for this purpose in Agile documentation). This is what we are doing by **adopting descriptive test names and using a single assertion per test** where it is possible: broken up the role of the class in tiny pieces which together give the full picture of the unit requirements.
We can go further and test also the use of ArrayIterator on associative

arrays:

```
    public function testIteratesOverAssociativeArrays()
    {
        $iterator = new ArrayIterator(array('foo' => 'bar'));
        $i = 0;
        foreach ($iterator as $key => $element) {
            $i++;
            $this->assertEquals('foo', $key);
            $this->assertEquals('bar', $element);
        }
        $this->assertEquals(1, $i);
    }
```

As an exercise you can try to refine this method two three independent ones, for instance creating the first of them with a name such as testIteratesOverAssociativeArraysUsingArrayKeysAsForeachKeys(). Don't worry about long method names as long as they are long to strengthen the specification, but only when the code can be refactored to smaller test methods. Even then, finding descriptive test names is the most difficult part of the process.
We can go on and add other test methods, and Spl has many.

Whenever a bug is found which you can impute to the class under test, you should **add a test method which exposes the bug**, and thus fails; then you can fix the class to make the test pass. This methodology helps to not reintroduce the bug in subsequent changes to the class, as a regression test is in place. It also defines more and more the behavior of a class by adding a method at the time.
The TDD methodology not only forces to add test methods to expose bug, but also to define new features. Implementing a user story is done by first writing a fail test which exposes the problem (the feature is not present at the time in the class) and then by implementing it.

I hope you're liking this journey in testing and you're considering to test extensively your code if you currently are not using phpunit or similar tools. In the next chapter, we will make a panoramic the assertion methods which phpunit provides to simplify the tester work. Remember that, in software unit testing, developer and tester coincide, or at least are at one next to the other, in the case of pair programming.

TDD exercises

2.1 What does happen to your class from 1.1 when you try to calculate

the factorial of 0? (it is assumed by definition equal to 1.) Add a failing test case.

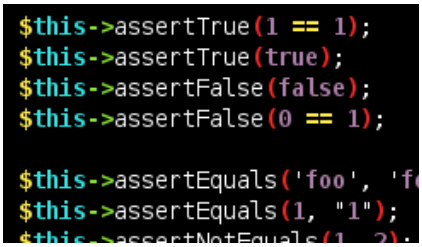
2.2 Modify the production class to make all the tests pass.

Code sample

```
<?php

class ArrayIteratorTest extends PHPUnit_Framework_TestCase
{
    public function testEmptyArrayIsNotIteratedOver()
    {
        $iterator = new ArrayIterator(array());
        foreach ($iterator as $element) {
            $this->fail();
        }
    }

/*
 * separate this method in two distinct ones express better the
intents of
 * this test case
    public function testIteratesOverOneElementArrays()
    {
        $iterator = new ArrayIterator(array('foo'));
        $i = 0;
        foreach ($iterator as $element) {
            $this->assertEquals('foo', $element);
            $i++;
        }
        $this->assertEquals(1, $i);
    }
*/

    public function testIteratesOverOneElementArraysUsingValues()
    {
        $iterator = new ArrayIterator(array('foo'));
        foreach ($iterator as $element) {
            $this->assertEquals('foo', $element);
        }
    }

    public function testIteratesOneTimeOverOneElementArrays()
    {
        $iterator = new ArrayIterator(array('foo'));
        $i = 0;
        foreach ($iterator as $element) {
            $i++;
        }
        $this->assertEquals(1, $i);
    }

    public function testIteratesOverAssociativeArrays()
    {
        $iterator = new ArrayIterator(array('foo' => 'bar'));
        $i = 0;
```

```
        foreach ($iterator as $key => $element) {
            $i++;
            $this->assertEquals('foo', $key);
            $this->assertEquals('bar', $element);
        }
        $this->assertEquals(1, $i);
    }
}
```

Chapter 3: assertions



Assertions are declarations that must hold true for a test to be declared successful: a test pass when it does not execute assertions or the one called are all verified correctly. Assertions are the final goal of a test, the place where you confront the expected and precalculated values of your variables with the ones that come from the system under test.
Assertions are implemented with methods and you have to **make sure they are actually executed**: thus, an if() construct inside a test is considered an anti-pattern as test methods should follow only one possible execution path where they find the assertions defined by the programmer.
There is also a assert() construct in php, used for enable checks on variables in production code. The assertions used in tests are a little different as they are real code (and not code passed in a string) and they do not clutter the production code but constitute a valuable part of test cases.
In phpunit there are some convenience methods which help to write expressive code and do different kind of assertions. These methods are available with a public signature on the test case class which is extended by default.

The **first assertion which fails causes an exception to be raised** and captured by phpunit runner. This means that if you are using an exception per test you are safe, but if you are writing test methods which contain multiple assertions, beware that the first failure will prevent the subsequent assertions from being executed. Only the assert*() calls which strictly depends on the previous ones to make sense should be placed in the same method as them.
Here is a list of the most common assertions available in phpunit: since the documentation is very good on this features I'm not going to go into the details. Most important and widely used methods are evidenced in bold.

- **assertTrue($argument)** takes a boolean as a mandatory parameter and make the test fail if $argument is not *true*. You must pass to it a result from a method which returns booleans, such as a comparison operator result. **assertFalse($argument)** presents the inverse of this method behavior, failing if $argument is different

from *false*.

- **assertEquals($expected, $actual)** takes two arguments and confront them with the == operator, declaring the test failed if they do not match. The canned result should be put in the $expected argument, while the result obtained from the system under test in the $actual one: they will be shown in this order if the test fails, along with a comparison of the arguments dumps when applicable. assertNotEquals() is this method's opposite.
- **assertSame($expected, $actual)** does the identical job of assertEquals(), but comparing the arguments with the === operator, which checks also the equality of variable types along with their values.
- **assertContains($needle, $haystack)** searches $needle in $haystack, which can be an array or an Iterator implementation. assertNotContains() can also be very handy.
- assertArrayHasKey($key, $array) evals if $key is in $array. It is used for both numeric and associative ones.
- assertContainsOnly($type, $haystack) fails if $haystack contains element whose type differs from $type. $type is one of the possible result from gettype().
- **assertType($type, $variable)** fails if $variable is not a $type. $type is specified as in assertContainsOnly(), or with PHPUnit types constants.
- assertNotNull($variable) fails if $variable is the special value *null*.
- assertLessThan(), assertGreaterThan(), assertGreatherThanOrEqual(), assertLessThanOrEquals() perform verifications on numbers and their names are probably self explanatory. They all take two arguments.
- assertStringsStartsWith($prefix, $string) and assertStringsEndsWith($suffix, $string) are also self explanatory and section a string for you, avoiding the need for substr() magic in a test.

Remember that you can still make up nearly any assertion by calling a verification method and pass the result to assertTrue(). Moreover, nearly everyone of this methods support a supplemental string parameter named $message, which will be shown in the case of a failing test caused by the assertion; if you're making up a complex method for a custom assertion you may want to provide $message to assertTrue() to provide information in case the production code regress. Obviously, the custom assertion

methods should be tested too.

I think you will start soon to use the more expressive assertions for what you are testing for: test methods should be short and easily understandable, and assertion methods which abstract away the verification burden are very beneficial. In the next parts, we'll dig into ways to reuse test code and in the annotations which phpunit recognizes to drive our test execution, such as *@dataProvider* and *@depends*.

TDD exercises

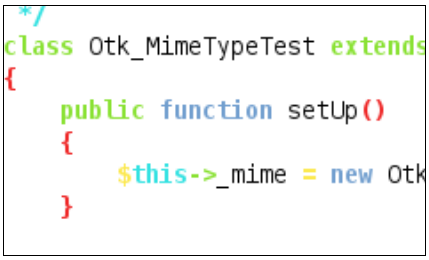
3.1 Write tests for a class which takes a single argument in the constructor and gives it back when a getter is called (*assertSame()* or *assertEquals()*?), then write the production class.

3.2 Write tests for the sort() php function, for simplicity with integer arrays as data.

Code sample

```
<?php

class AssertionsTest extends PHPUnit_Framework_TestCase
{
    public function
testAssertionsMethodsWorksWithCorrectArguments()
    {
        $this->assertTrue(1 == 1);
        $this->assertTrue(true);
        $this->assertFalse(false);
        $this->assertFalse(0 == 1);

        $this->assertEquals('foo', 'foo');
        $this->assertEquals(1, "1");
        $this->assertNotEquals(1, 2);
        $this->assertSame(1, 1);
        $this->assertNotSame(1, "1");

        $this->assertContains('foo', array('foo', 'otherValue'));
        $this->assertContains('foo', new
ArrayIterator(array('foo', 'otherValue')));
        $this->assertNotContains('foo', array('otherValue'));
        $this->assertContainsOnly('string', array('foo',
'otherValue'));

        $this->assertArrayHasKey(0, array('foo'));
        $this->assertArrayHasKey('foo', array('foo' => 'bar',
'otherValue'));
    }
}
```

Chapter 4: fixtures



In the previous parts, we have explored how to install phpunit and how to write tests which exercise our production code. Also we have learned to use the assertion methods to check the actual results: now we are ready to improve the test code from a refactoring point of view, and to take advantage of phpunit features.
While writing more and more test methods, you can notice that you follow a common pattern, commonly known as *arrange-act-assert*; this is the main motif of state based testing.
The first operation in a test is usually to set up the system under test, being it an object or a complex object graph; then the methods of the object are called during the act part and some assertions (hopefully not more than one) are done on the results returned from these calls. In some cases, when you have allocated external resources like a fake database, a final cleaning up phase is needed.
What you will actually discover is that often **part of the arrange phase and the final cleanup code are shared between test methods**: for example in case you are testing a single class, the instantiation of an object is a simple operation you can extract from the test methods. To support this extraction, phpunit (and all xUnit frameworks) provide the setUp() and tearDown() template methods.
These methods are executed respectively before and after *every* test method: default implementations are provided in PHPUnit_Framework_TestCase with an empty body. You can override this empty methods when useful, to have arrange/cleanup code to be shared between tests in the same test case and prepare a known state before every run. This known state is called a *fixture*.
Your test case class can go from this:

```
<?php
class ArrayIteratorTest extends PHPUnit_Framework_TestCase
{
    public function testSomething()
    {
        $iterator = new ArrayIterator(array('a', 'b', 'c'));
        // act, assert...
    }

    public function testOtherFeature()
```

```
    {
        $iterator = new ArrayIterator(array('a', 'b', 'c'));
        // act, assert...
    }
}
```

to this:

```
<?php
class ArrayIteratorwithFixtureTest extends
PHPUnit_Framework_TestCase
{
    private $_iterator;

    public function setUp()
    {
        $this->_iterator = new ArrayIterator(array('a', 'b',
'c'));
    }

    public function testSomething()
    {
        // act on $this->_iterator, assert...
    }

    public function testOtherFeature()
    {
        // act on $this->_iterator, assert...
    }
}
```

Observe that, since an object of this class will be created to run the test, you can conserve every variable you want as a private member, and then have a reference to it available in the test method. setUp() usage provides a cleaner and *dont-repeat-yourself* solution, and saves many lines of code when many test methods are needed.

Here is some know-how on using fixtures:

- **usually the tearDown() method should not be provided** since the fixture is an object graph and will be garbage-collected after all the tests are executed, or overwritten by the next setUp() call. Thus, the empty body provided by default is often enough.
- the fixture methods are executed for every test, so **the test methods have the same state as a starting point**. When more than one fixture is requested, the common practice is to break down the test case, preparing more than one test case class for the system under test; these classes represents different scenarios and

together constitutes the overall test suite for this system.

- **sharing a fixture between test cases can be a smell for a bad design**, since they are not insulated enough and classes know too much of each other. This cannot be done with setUp() methods however, but there are suite-level setup available in phpunit if you must share a fixture. However, keep in mind that you probably can refactor your classes to improve the maintainability of the application and of its test suite.
- **setUpBeforeClass() and tearDownAfterClass()** are two hooks (static methods) which are executed before a test case methods are considered and after the overall process is finished. They are the equivalent of setUp() and tearDown(), but at the test case level instead of the test method one.
- finally, **assertPreConditions() and assertPostConditions()** are two methods executed before and after the a test method. They differ from setUp() and tearDown() since they are executed only if the test did not already fail and they can halt the execution with a failing assertion. setUp() and tearDown() should never throw exceptions and they are executed anyway, unconcerned by the current test outcome.

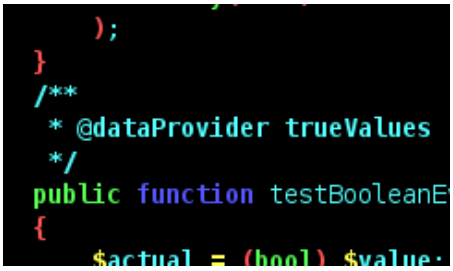
This is all you must know on test fixtures to start experimenting with them. I hope your test code will be much more well written after introducing setUp().
In the next chapter, we'll explore the annotations that can influence phpunit test runner, like @depends and @dataProvider.

TDD exercises

4.1 Write a class Sorter that wraps the sort() native function and returns an ordered integer array without touching the original. Start with some tests before writing a single line of production code.

4.2 How many *new* do you have in the test case? Refactor till you have only one object creation.

Chapter 5: annotations



Now that we have learned much about writing tests (with or without fixtures) and using assertions, we can improve our tests further by exploiting phpunit features. This awesome testing tool provides support for several annotations which can add behavior to your test case class **without making you write boilerplate code**. Annotations are a standard way to add metadata to code entities, such as classes or methods, and are composed by a @ tag followed by zero, one or more arguments. While the parsing implementation is left to the tool which will use them, their aspect is consistent: phpDocumentor also collects @param and @return annotations to describe an Api.
Remember that annotations must be placed in docblock comments as in the php engine there is no native support for them: phpunit extracts them from the comment block using reflection.

While writing an Api or also a simple class, the corner cases and incorrect inputs have to be taken into consideration. The standard way to manage errors and bad situations in an oop application is to use exceptions. But how to **test that a method raises an exception when needed**? Of course the normal behavior is tested with real data that returns a real result. For the exceptional behavior, we can start with this test method:

```
    public function testMethodRaiseException()
    {
        try {
            $iterator = new ArrayIterator(42);
            $this->fail();
        } catch (InvalidArgumentException $e) {
        }
    }
```

The purpose of this code is to raise an exception by passing invalid data to the constructor of ArrayIterator, which requires an array. If the exception is raised accordingly, it bubbles up to the end of the try block and it is catched correctly, making the test pass. If the exception is not thrown, the call to fail() declares the test failed.
However, this paradigm will be repeated very often everytime you need to test an exception and so it can be abstracted away. Also, this code does

not convey the intent of testing an exception since it is cluttered with details like an empty catch block and calls to fail().
Phpunit already abstracts away this code providing an annotation, **@expectedException**, which has to be put in the method docblock:

```
/**
 * @expectedException InvalidArgumentException
 */
public function testMethodRaiseExceptionAgain()
{
    $iterator = new ArrayIterator(42);
}
```

This code is much more clear than the constructs we used earlier. The only code present in the method is the one required to throw the exception, while the intent is described in the method name and in its annotations.

Another common repetition is **testing a method with different kind of inputs**, while executing always the same code. This is commonly resolved with a loop:

```
public function testBooleanEvaluationInALoop()
{
    $values = array(1, '1', 'on', true);
    foreach ($values as $value) {
        $actual = (bool) $value;
        $this->assertTrue($actual);
    }
}
```

But phpunit can do the loop for you, taking advantage of the @dataProvider annotation:

```
public static function trueValues()
{
    return array(
        array(1),
        array('1'),
        array('on'),
        array(true)
    );
}
/**
 * @dataProvider trueValues
 */
public function testBooleanEvaluation($value)
{
    $actual = (bool) $value;
    $this->assertTrue($actual);
}
```

This annotation should be followed by the name of a static method *in* the test case which returns an array of data sets to be passed to the test method. Phpunit will iterate over this array and using each one of its elements (which will be an array containing the arguments) to run the test method, telling you which data set was in use in case of a test failure. Of course you can put anything in the data sets: input for the SUT or expected result, or both.
The code becomes a bit longer, but the expressivity of defining the concept of different data sets in a standard way are worth considering.

The last common situation we will look at today is **test dependency**. Again, we are talking about dependency beneath the same test case since interdependencies between unit tests are a small of high coupling and should be raise suspects about your classes design.
It happens often that some test methods are more specific than the first you wrote and they will obviously fail if the formers do. The classic example is the add()/remove() tests on a container: to make sure remove() works you have to use add() for the *arrange* part of the test method. Phpunit solve this common **problem of logic and temporal precedence** (I won't present a workaround like in the other cases since it was not possible to solve this issue before phpunit 3.4 introduced @depends):

```
    public function testArrayAdditionWorks()
    {
        $array = array();
        $array[0] = 'foo';
        $this->assertTrue(isset($array[0]));
        return $array;
    }

    /**
     * @depends testArrayAdditionWorks
     */
    public function testArrayRemovalWorks($fixture)
    {
        unset($fixture[0]);
        $this->assertFalse(isset($fixture[0]));
    }
```

Not only testArrayAdditionWorks() is executed before testArrayRemovalWorks(), but since it returns something, this result is passed as an argument to the dependent method. If the former test fails, however, the dependent ones are marked as skipped as they will fail anyway by definition. They will clutter the output too, while it is clear that the functionality that needs repairment is the array addition.

I hope this standard phpunit annotations can help you enjoy writing tests for your php classes, leaving you the exciting work and taking off the boring one. In the next parts, we'll look at refactoring for test code before taking a journey with stubs and mocks.

TDD exercises

5.1 Refactor the tests from 4.2 using the @dataProvider annotation.

5.2 Write tests for a Collection class which stores objects, and has the methods add(), contains() and remove(). Where you should put @depends annotations?

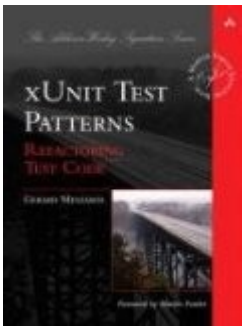
5.2 Implement the Collection class.

Code sample

```php
<?php

class AnnotationsTest extends PHPUnit_Framework_TestCase
{
    public function testMethodRaiseException()
    {
        try {
            $iterator = new ArrayIterator(42);
            $this->fail();
        } catch (InvalidArgumentException $e) {
        }
    }

    /**
     * @expectedException InvalidArgumentException
     */
    public function testMethodRaiseExceptionAgain()
    {
        $iterator = new ArrayIterator(42);
    }

    public function testBooleanEvaluationInALoop()
    {
        $values = array(1, '1', 'on', true);
        foreach ($values as $value) {
            $actual = (bool) $value;
            $this->assertTrue($actual);
        }
    }

    public static function trueValues()
    {
        return array(
            array(1),
            array('1'),
            array('on'),
            array(true)
        );
    }
    /**
     * @dataProvider trueValues
     */
    public function testBooleanEvaluation($value)
    {
        $actual = (bool) $value;
        $this->assertTrue($actual);
    }

    public function testArrayAdditionWorks()
    {
```

```
        $array = array();
        $array[0] = 'foo';
        $this->assertTrue(isset($array[0]));
        return $array;
    }

    /**
     * @depends testArrayAdditionWorks
     */
    public function testArrayRemovalWorks($fixture)
    {
        unset($fixture[0]);
        $this->assertFalse(isset($fixture[0]));
    }
}
```

Chapter 6: refactoring and patterns



In the xUnit world, tests are code. While there are testing tools which treats tests as data, phpunit and companions recognize classes and objects: this means that they are first class citizens and there should be no distinction in importance between production and test code.

Why it is important to refactor production code? To **improve maintainability** and ensure that changes which break the system appear less often. The same can be said for the tests: a suite that embrace change and is maintainable will make the developers actually use it, from the start to the long run. While the focus is usually on production code refactoring, today we will talk about test refactoring and the patterns where you should head to.
The worst thing that can happen is having an outdated test suite which fails because it is not maintained with production code: it will quickly lose credibility and thus it will be run sparingly, and then forgotten.
One of the best methodologies to improve production code maintainability is to test it: the more easy to test is a class, the more decoupled and maintainable it becomes. You will often find yourself refactoring a production class to simplify testing, for instance making Demeter happy, resulting in the application to have a simpler design.
Following our duality of production and test code, sometimes test methods and test cases grow and present a lot of repetition. What can be done to avoid these problems and maintain an agile (with the lowercase *a*) test suite is to **refactor test code towards some patterns**, some of them you already started to grasp in the previous chapters of this book. Test code has often a low complexity compared to production code: it runs in an isolated environment, with nearly no state to maintain, with very decoupled classes (the test cases) and the help of a framework. Thus, it's tempting to use a lot of copy&paste to write tests, but knowing a bunch of patterns can flatten even tests little complexity and help you avoid duplication. As all patterns, they have been catalogued and given a standard name.

- **Standard Fixture** and **Fresh Fixture** reuse the code which builds fixtures for the tests (and not the fixture itself). This pattern can be implemented with phpunit setUp() method.
- **Shared Fixture** reuse the same object graph for a set of tests:

obviously it should have no state or a way to reach a particular state for testing purposes. This pattern can be implemented with phpunit setUpBeforeClass() method.

- **Four Phase Test** is the classical motif of a test method: arrange, act, assert and the optional teardown.
- **Test Runner** and **Test Suite Object** are pattern which phpunit implements for you. You can then specifiy metadata to alter the building of a test suite or execution options, or specifical annotations which the runner supports.
- **State Verification** is the simplest way of using phpunit and it's what we have done until now, writing assertions on explicit results of the system under test. **Behavior Verification** is based on making assertiong on indirect results, like method calls on collaborators and will be treated in the next chapters; **Mock** and **Stub** are patterns used in Behavior Verification, and phpunit provides support for their dynamic creation.
- **Table Truncation Teardown** and **Transaction RollBack TearDown** are standard patterns for testing components which interact with a database.
- **Literal, Derived and Generated Value** are patterns to provide fake data to the system under test. They all have their place in unit testing, depending on the unit purpose.

If you are interested in learning more about patterns you should check out the book xUnit Test Patterns: Refactoring Test Code and its website (http://xunitpatterns.com/index.html), which is a very complete guide to probably every single testing construct that has been explored in the xUnit frameworks so far. On the website you can find description and usage examples of all the patterns described here and of other specific ones. Moreover, remember that test code is still code and the **basic refactorings** like Extract Method, Extract Superclass, Introduce Explaining Variable etc. **are valid also in the testing land**. Simply refactor some boilerplate code in private methods of a test case can save you the boring job of updating duplicated blocks.

As a side note, remember that when refactoring production code you have the safety net of the test suite, that will tell you when you have just broke something. No one tests the tests, however, and so you may want to **temporarily break the behavior under test before refactoring a method or a test case**. Simply altering the return statements of

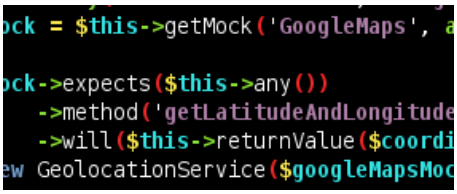
production methods can make the test fail so you can control that it continue to fail after the refactoring. When writing the original test, the TDD methodology crafts the method even before the production code exists and this is one of the main reason why the test is solid; a test is valid if it's able to find problems in the production code: that is, failing when it should fail.

I hope this book is becoming interesting as now you have learned your tests have the same importance of the production code. They can even be more important: if I have to choose between throwing away the production code and its documentation, and losing a good test suite, I will definitely throw away the first. It can be rewritten using the tests, while writing a complete test suite of an application without any tests is an harder task. In the next parts, we'll enter the world of Test Doubles and of behavior verification, taking advantage of Mocks, Stubs, Fakes and similar patterns.

TDD exercises

6.1 Refactor the tests you have written in the previous parts to eliminate duplication, introducing creation methods and ensuring the four/three phases of tests are identifiable.
6.2 Use the Pdo Sqlite driver to implement Table Truncation TearDown for a small test database (one table will suffice). You should use the tearDown() method to empty the table.

Chapter 7: stubs



The name *Unit testing* explains the most special and powerful characteristic of this type of testing: the granularity of verifications is taken to the class size, which is the smallest cohesive unit we can find in an object-oriented application.
Despite these good intentions, **tests often cross the borders of the single class** because the system under test is constituted by more than one object: the main one which methods are called on (to verify the results returned with assertions) and its collaborators.
This is expected as no object is an island: there are very few classes which work alone. While we can skip the middle man on the upper side by calling methods directly on the system under test, there's no trivial way to wire the internal method calls to predefined result.
However, it's very important to find a way for insulating a class from its collaborators: while a small percentage of functional tests (which exercise an object graph instead of a single unit) are acceptable and required in a test suite, the main concern of a test-infected developer should be to have the most **fine-grained test suite** of the country:

- **tests are very coupled to the production code**, in particular to the classes which they use directly. It's a good idea to limit this coupling to the unit under test as much as possible;
- the **possible interactions with a single object are a few**, but integrating collaborators raises the number of situations that can happen very quickly;
- the **collaborators could be very slow** (relying on a database or querying a web service) or not even worth testing because it is not our responsibility;
- finally, **if a collaborator breaks, not only its tests will fail** but also the tests of the classes which use it. The defect could then be hard to locate.

In short, every collaborator introduced in the test arrangement is a step towards integration testing rather than unit testing. However, don't take this advice as a prohibition: **many classes (like Entities/newables, but also infrastructure ones like an ArrayIterator) are too much of an hassle to substitute** with the techniques described later in this post. You

should definitely instantiate User and Group in tests of service classes which acts on them.

The verb *substitute* is appropriate: the only way to keep out collaborators from a unit test is to replace them with other objects. These objects are commonly known as Test Doubles and they are subclasses or alternative implementations respectively of the class or interface required by the SUT. This property allows them to be considered *instanceof* the original collaborator class/interface and to be kept as a field reference or to be passed as a method parameter, which are the only ways I can think of to recognize a collaborator.
Dependency Injection **plays a key role in many unit tests**: in the case of a field reference on a class, there is the need to replace this reference with the Test Double one, to hijack the method calls on the collaborator itself. Since field references are usually private, it is difficult to access them (requiring reflection) and violating a class encapsulation to simplify testing does not seem a good idea.
Thus, **the natural way to provide Test Doubles for collaborators is Dependency Injection**, being it implemented via the constructor or via setters. Instead of instantiating the production class, simply produce an object of its custom-made subclass and use it during the construction of the SUT. My SUTs usually have optional constructor parameters to allow passing in a null when the collaborator is not needed at all.

While entering the Test Doubles world, the average programmer hears many terms which describes substitutes of an object's collaborators. In crescent order of complexity, they are:

- **Dummies**: object which do not provide any interaction, but are placeholders used to satisfy compile-time dependencies and to avoid breaking the null checks: no method is called on them but they are likely to be required parameters of a SUT method or part of its constructor signature. To avoid the creation of dummies I prefer to keep all constructor parameters optional, trusting that the Factory which creates the object passes regular collaborators instead of null values.
- **Stubs**: objects where some methods have been overridden to return canned results. They may have more than one precalculated result available, depending on the method parameters combination, but these variables are known values once you reach the *act* part of the test method.

- **Mock objects**: also known as Test Spies, mock objects are used in a testing style different from what we have worked on since the first chapter of this book (behavior based testing). They will be the main argument of the next episode.
- **Fakes**: objects which have a working implementation, but much simpler than the real collaborator one. An in-memory ArrayObject which substitutes a database result Iterator is an example of a Fake.

You generally don't need to write a Dummy object since there is no interaction with it: the real collaborator can be used instead if its constructor is thin. A Fake is a running implementation so if an already existing class cannot work, there's usually no other choice than write a real subclass and reusing it in all the tests that require the collaborator.
For Stubs and Mocks the situation is different: there are plenty of frameworks for nearly every language which provide help in generating them, which take care of evaluating the subclass code and instantiating an object. **Phpunit incorporates a small mocking framework**, accessible through the method getMock() on the test case class. Remember that while the method is named getMock(), both Stubs and Mocks can be created via this Api. In this chapter we'll focus again on state verification and we'll use a Stub to improve the granularity of a test.

We are going to give **a meaningful example of unit testing using a Stub**. In this example, a GeolocationService takes a User object and fills its latitude and longitude fields using the location specified.
GeolocationService requires an instance of a fictional GoogleMaps object to work, and since we all love Dependency Injection it is passed in its constructor.
Note that GoogleMaps can also be an interface or a base abstract class: there is no technical difference. Moreover, if it was a less important collaborator it can even be passed as a locate() parameter.
This is the test case:

```
class GeolocationServiceTest extends PHPUnit_Framework_TestCase
{
    public function testProvidesLatitudeAndLongitudeForAnUser()
    {
        $coordinates = array('latitude' => '42N', 'longitude' =>
'12E');
        $googleMapsMock = $this->getMock('GoogleMaps',
array('getLatitudeAndLongitude'));
        $googleMapsMock->expects($this->any())
                       ->method('getLatitudeAndLongitude')
                       ->will($this->returnValue($coordinates));
```

```
        $service = new GeolocationService($googleMapsMock);
        $user = new User;
        $user->location = 'Rome';
        $service->locate($user);
        $this->assertEquals('42N', $user->latitude);
        $this->assertEquals('12E', $user->longitude);
    }
}
```

Note that I have created a Stub only for the external service and not for the User class. The former is external, slow, and unpredictable, but the latter is simple, with little or none internal behavior, and there's a small chance it will break. *Nothing behaves like a String more than a String*, as Misko Hevery says. The test method now focuses on exercising the locate() method and not also the GoogleMaps class.

Finally, let's take a look at the **getMock() Api:**

```
object getMock($originalClassName, [array $methods, [array
$arguments,

[string $mockClassName, [boolean $callOriginalConstructor,

[boolean $callOriginalClone, [boolean $callAutoload]]]]])
```

$originalClassName is the name of the class you want to create a Stub/Mock for. $methods is a numeric indexed array containing the names of the methods you want to subclass; if left empty, every method will be substituted. $arguments are arguments to pass to the constructor of the original class (almost never used), while $mockClassName is a custom name for the subclass created.
The last three arguments are boolean values used to determine if leaving the original constructor or clone method, or to allow autoloading of $originalClassName. They default to true. Often you want to set $callOriginalConstructor to false if its signature requires other collaborators to be passed in. All arguments but the first one are optional.
The mock produced, along with the original class methods, also has the expects() one available. For now, simply calling it with the argument $this->any() will do the job. This method returns a PHPUnit_Framework_MockObject_Builder_InvocationMocker instance; in short, an internal object that you can call method() and will() on to decide the method name to replace and its predefined behavior. The simplest possible behavior is $this->returnValue(...), but also $this->returnArgument($argumentNumber) is available, along with $this-

>returnCallback($callbackName); refer to the phpunit documentation for supplemental informations.

I hope this introduction to Stub objects has helped you grasping the essence of Unit testing in php. Feel free to ask clarifications in the comments. In the next chapter of this book, we will explore the possibilities of Mock objects and behavior based testing.

TDD exercises

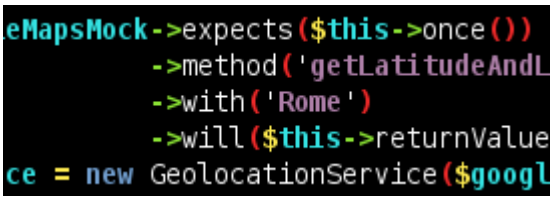
7.1 Write an infinite FibonacciIterator that returns the Fibonacci series and test it. The Fibonacci series is 0, 1, 1, 2, 3, 5, 8... every term is the sum of the two previous ones.
7.2 Write a EvenIterator which takes a FibonacciIterator an iterates only on the even-indexed values (returning 0, 1, 3, 8, 21...).
7.3 Write tests for the EvenIterator class, stubbing out the FibonacciIterator using an ArrayIterator in substitution, which is provided by the Spl (otherwise it will never terminate!)
7.4 Write a class that uses HTTP_Client Pear package to check a list links and find out which are broken (404 error). Start with testing it without instancing the real HTTP_Client class by stubbing it.

Code sample

```
<?php

/**
 * For brevity I use public fields instead of getters and setters.
 */
class User
{
    public $location;
    public $latitude;
    public $longitude;
}

class GoogleMaps
{
    public function getLatitudeAndLongitude($location)
    {
        // some obscure network code to contact maps.google.com
        // ...
        return array('latitude' => $someValue, 'longitude' =>
$someOtherValue);
    }
}

class GeolocationService
{
    private $_maps;

    public function __construct(GoogleMaps $maps)
    {
        $this->_maps = $maps;
    }

    public function locate(User $user)
    {
        $location = $user->location;
        if ($location === null) {
            $location = 'Milan';
        }
        $coordinates = $this->_maps-
>getLatitudeAndLongitude($location);
        $user->latitude = $coordinates['latitude'];
        $user->longitude = $coordinates['longitude'];
    }
}

class GeolocationServiceTest extends PHPUnit_Framework_TestCase
{
    public function testProvidesLatitudeAndLongitudeForAnUser()
    {
        $coordinates = array('latitude' => '42N', 'longitude' =>
```

```
'12E');
        $googleMapsMock = $this->getMock('GoogleMaps',
array('getLatitudeAndLongitude'));
        $googleMapsMock->expects($this->any())
                       ->method('getLatitudeAndLongitude')
                       ->will($this->returnValue($coordinates));
        $service = new GeolocationService($googleMapsMock);
        $user = new User;
        $user->location = 'Rome';
        $service->locate($user);
        $this->assertEquals('42N', $user->latitude);
        $this->assertEquals('12E', $user->longitude);
    }
}
```

Chapter 8: mocks



In the previous chapter of this book, we have listed the various types of Test Doubles along with the ones that phpunit can easily generate: Stubs and Mocks. The latter are utilized in a different kind of testing than the one presented so far: behavior verification.

The behavior verification testing style differs from the state verification one in the subjects of the assertion methods. While state verification specifies explicit assertion methods to be called upon a test result, behavior verification is focused on checking the actions the *system under test* undertakes. **These actions comprehend which methods it calls**, and how many times it does so; but also **the parameters it passes** to these methods and their order.
The standard interaction with collaborators in object-oriented systems consists of method calls. This kind of testing prescribes to place assertions directly in the overridden methods of Test Doubles, or at the end of every test, to verify that the SUT behavior conforms to specifical rules. These Test Doubles, which can run assertions on their methods parameters, are called Mock Objects (or simply Mocks). The contraposition here is with Stubs, which extend the capabilities of a state based testing but do not make any assumption on method calls or parameters.
Note that the assertions on parameters are placed inside the generated methods, while assertions on method calls are executed by phpunit after the test has run. This means that in a pure behavior verification test you won't find any assert*() calls, which perform state verification.

Now we are going to rewrite the unit test of the previous chapter taking advantage of phpunit mocks generation, but with a mixed approach which contains also explicit assertions. The test was about verifying that the GeolocationService class made use of a GoogleMaps collaborator to find out the latitude and longitude of an User object, and the key characteristic was insulation of the test from the GoogleMaps real implementation with a Test Double. You can find the Stub example at the end of the previous chapter.

```
class GeolocationServiceWithMocksTest extends
```

```
PHPUnit_Framework_TestCase
{
    public function testProvidesLatitudeAndLongitudeForAnUser()
    {
        $coordinates = array('latitude' => '42N', 'longitude' =>
'12E');
        $googleMapsMock = $this->getMock('GoogleMaps',
array('getLatitudeAndLongitude'));
        $googleMapsMock->expects($this->once())
                       ->method('getLatitudeAndLongitude')
                       ->with('Rome')
                       ->will($this->returnValue($coordinates));
        $service = new GeolocationService($googleMapsMock);
        $user = new User;
        $user->location = 'Rome';
        $service->locate($user);
        $this->assertEquals('42N', $user->latitude);
        $this->assertEquals('12E', $user->longitude);
    }
}
```

The test is actually very similar to the Stub one, but there are some differences:

- the expect() method of the mock returns an expectation object with a fluent interface we can work with. However, this time **a matcher is passed which specifies how many times the mocked method should be called**. In the Stub example, the matcher used is $this->any(), that does not run any assertions on the number of calls at the end of the test. Other available matchers are $this->never() and $this->exactly($number). The power of the matchers used in xUnit frameworks is they augment the test's code readability, making it similar to plain English.
- On the expectation object, along with will() and method(), we are also **calling with() to specify the parameter we want to check** as passed to getLatitudeAndLongitude(). If we wanted to check more parameters as exact values, we would pass an array to with() containing the actual list. However, we can make also weak assertions by using constraints objects, like $this->attributeEqualTo($name, $value) or $this->isInstanceOf($className), or maybe $this->anything() if no assertion has to be made on a particular parameter.
- There is no formal definition that says Mocks can't return canned results, as this is often mandatory for the code flow and to complete the test successfully. Though, if you TDD the system under test

using mocks without predefined results, it's likely that you will produce a class with a different programming style which works with those tests, and uses mocks very effectively.

- Whenever you write a with() call or a matcher in expects(), **be aware you are building a Mock** and not a Stub.

You can find the complete, running test case at the end of the chapter. I tried to include complete examples in this book to show the practical side of testing instead of tips which are great in theory, but fail to apply in a real situation.

After this example of behavior verification, which makes use of the most advanced phpunit features, we are ready to explore the code coverage features in the next chapter.

TDD exercises

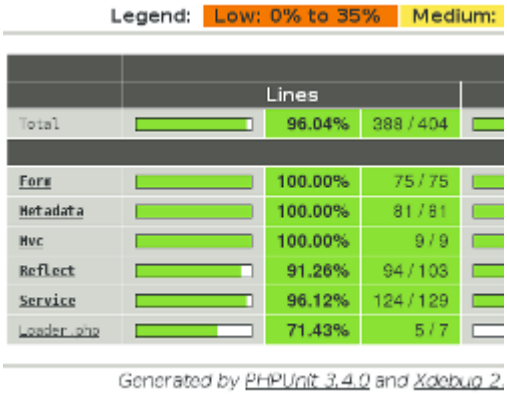
8.1 Write tests for a class PermissionReader that has a __toString() method which produces a human readable string, containing the permissions of a SplFileInfo which is passed to it in the constructor. No actual file should be used, only mocks (you can use actual files to learn about Spl api but they should not be present in the final production code and unit tests).

Code sample

```
<?php

/**
 * For brevity I use public fields instead of getters and setters.
 */
class User
{
    public $location;
    public $latitude;
    public $longitude;
}

class GoogleMaps
{
    public function getLatitudeAndLongitude($location)
    {
        // some obscure network code to contact maps.google.com
        // ...
        return array('latitude' => $someValue, 'longitude' =>
$someOtherValue);
    }
}

class GeolocationService
{
    private $_maps;

    public function __construct(GoogleMaps $maps)
    {
        $this->_maps = $maps;
    }

    public function locate(User $user)
    {
        $location = $user->location;
        if ($location === null) {
            $location = 'Milan';
        }
        $coordinates = $this->_maps-
>getLatitudeAndLongitude($location);
        $user->latitude = $coordinates['latitude'];
        $user->longitude = $coordinates['longitude'];
    }
}

class GeolocationServiceWithMocksTest extends
PHPUnit_Framework_TestCase
{
    public function testProvidesLatitudeAndLongitudeForAnUser()
    {
```

```
        $coordinates = array('latitude' => '42N', 'longitude' =>
'12E');
        $googleMapsMock = $this->getMock('GoogleMaps',
array('getLatitudeAndLongitude'));
        $googleMapsMock->expects($this->once())
                       ->method('getLatitudeAndLongitude')
                       ->with('Rome')
                       ->will($this->returnValue($coordinates));
        $service = new GeolocationService($googleMapsMock);
        $user = new User;
        $user->location = 'Rome';
        $service->locate($user);
        $this->assertEquals('42N', $user->latitude);
        $this->assertEquals('12E', $user->longitude);
    }
}
```

Chapter 9: command line options



Optimizing a test suite by adding test methods and test cases can be useful to improve the quality of your application code. Yet, every optimization starts with a profiling phase, that tells you where there is a need for test cases and where there is already a good *coverage*.
Code coverage is defined as the ratio of lines of code exercised by the unit tests to the overall number of lines; the same ratio can be calculated using code blocks as the unit of measure.
Phpunit provides code coverage reports generation via command line switches: one of them is **--coverage-html $directory** which places a human-readable html report in the $directory specified. There are other formats available, such as Xml, created for the purpose of interpreting a report with a third party application.
Please note that phpunit code coverage features require the xdebug extension to work.

Another useful switch is the **--configuration $file** one. $file should be an xml configuration file that tells phpunit what files have to be considered as containing test cases. This is very handy to compose a suite and can substitute the famous and hard to mantain AllTests.php files.
Here is a simple example for a configuration file:

```
<phpunit>
    <testsuite name="Ossigeno Test Suite">
        <directory suffix="Test.php">tests/</directory>
        <directory
suffix="Test.php">application/modules</directory>
    </testsuite>
</phpunit>
```

Running phpunit with this switch, instead of specifying a particular file, will force the runner to consider all php files which name ends in "Test.php" in the directories tests/ and application/modules/. While running a single test case gives as output a line of dots, running all these tests in sequence will result in multiple lines and in a list of all failed tests (although you can require the list of skipped and incomplete tests by using the **--verbose**

switch) in the overall list generated according to the configuration.

Along with the --configuration option, I strongly suggest to use the **--bootstrap $phpScript** directive. Your test cases probably need a global bootstrap phase for autoloading and setting up the include_path or other options. In some old versions of phpunit, you had to include a require_once() call at the start of each test script to make sure it was executed before the test. Now you can simply tell phpunit to run a file of your choice before starting with the test phase.

Running an entire suite is a good practice to discover if your changes or refactorings have broken some functionalities. However, it's an overkill if you have to do it very often, like in a short feedback cycle for TDD: supposing you have more than one test case for your SUT, it can be useful to select all those tests and leaving out the rest of the suite.
This is the case when the **@group** annotation is handy. You can mark with the **@group $name** annotation the docblock of test case classes, and also add multiple lines if you feel the contained tests can be useful in more than one scenario. Then the **--group $group** command line switch excludes test cases which do not belong to $group from being run.

So we can finally give an example of running a test suite:

```
phpunit --bootstrap tests/TestHelper.php
--configuration=tests/configuration.xml
```

for instance we can restrict the selected tests to the NakedPhp_Form package ones:

```
phpunit --bootstrap tests/TestHelper.php
--configuration=tests/configuration.xml --group=NakedPhp_Form
```

or requiring a code coverage report to see where we need to add test code:

```
phpunit --bootstrap tests/TestHelper.php
--configuration=tests/configuration.xml --coverage-html directory/
```

I hope these tips will be useful to you for utilizing phpunit at its best. It is a very well-crafted tool that you can take advantage of for TDD purposes, and also for functional and integration tests. Although the name suggests unit testing as a goal, you should certainly include in your test suite some functional tests, which exercise a feature provided of more than one

object, and integration tests, which covers the wiring of your object and verify that your application works on an end-to-end basis.

*Bonus tip: using **--no-globals-backup** and **--no-static-backup** can speed up your tests execution by avoiding unuseful isolation of tests. If your application has no global state they will work correctly anyway.*

TDD exercises

9.1 Run all your tests and generate an html coverage report. If you have TDDed your classes, the coverage should be close to 100%.

Chapter 10: The TDD theory

Test-Driven Development is a very practical approach about testing and design, and in this chapter I will describe the fundamentals of TDD and the benefits it gives to an application.

TDD is a test-first approach, and prescribes to **start writing tests even before a line of production code is written**. The application design is done a bit at the time: every test that is added specifies a part of the internal or external Api. If you have done some exercises along the way to this chapter, you are already familiar with this concept.

This does not mean that no design should be done upfront: TDDers prefer yo say that L*ittle Design Up Front* should substitute *Big Design Up Front*. The reason for doing less design is taken to embrace iterative methods like TDD, where after every iteration the developer gets to know more about the domain and the system he is building. I am not talking about Agile iterations (some weeks): TDD is a lower-level practice which influences how you write code. TDD iterations are usually 5-minutes long, and every longer iteration may suggest you should break it up in more pieces (which corresponds to fine-grained tests).

Note that a big advantage of TDD is the testability of the code produced: since you write tests before any production code, the resulting classes will be forced to be testable: you decide the Api. Moreover, writing tests first ensures that they will be written at all, and not skipped at the end of the day to write other code.

But the biggest advantage of design crafted via TDD is that since classes are forced to be testable, they are hence maintainable and less entangled. **A testable design is a decoupled one** and we gain all the advantages of testability.

TDD Phases

The basic TDD cycle (around five minutes) consists in three phases. Red and green are the typical colorization of XUnit results.

RED

You write a failing test. This test should be exercising one feature of your SUT, which does not exist yet. The more fine-grained the test is, the

shorter the cycle will become and there will be less chances to breaking up working code or wondering on how to make the tests pass.

Note that **the test should fail**, to ensure that it actually tests the SUT. Since the SUT or the particular SUT's functionality does not exist yet, it is obvious that a correct test will fail.

Sometimes you craft a passing test to improve coverage or that accidentally passes since your code already implements the functionality. There are different reactions in this case:

- if the SUT already implements correctly the functionality you intend to test, leave the test in place and restart the cycle. Probably you will add this type of tests to gain confidence before major refactorings.
- if the SUT does not implement the full functionality, try to change the input to catch the SUT when it is failing. Maybe it does return a correct result for this data by accident.

GREEN

When you have a shiny new failing test, you should now open the SUT's class file(s) and start writing production code. You are allowed to write as much production code as it is needed to make the test pass.

You're **not allowed to write more code**. If I can remove a line of code from your SUT and still get all tests green, this means this line is not necessary (so remove it yourself before I do). This design choice is done to prevent bloat and, more importantly, to not allow untested code to leak into the SUT.

REFACTOR (sometimes orange)

After having committed nearly every design crime to get a green result, like using a buch of GOTOs with a look-up table, it's time to clean up and refactor the mess.

Obviusly I'm joking and refactoring is usually not necessary. Though, **after a certain number of cycles the new code really adds up and you may want take the chance to reorganize** the code base before it becomes a mine field.

The key here is that you must start refactoring after a green phase and finish with the tests still green. If you reach a new and better codebase state, you should pass only trough green states, never breaking anything.

The reason of this rule-of-thumb is that once you start breaking the code it becomes difficult to going back. The best solution is to maintain a working application all the time.

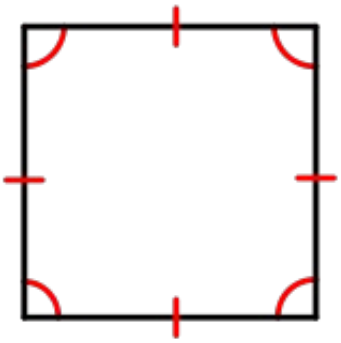
You can also refactor tests: in this case you should work during a red phase, even by temporarily breaking the SUT, to test that the tests really fail when they should do (no pun intended… to watch the watchmen).

More often, you will refactor tests and production code at the same time to better assign responsibilities. Try to keep a working system for as long as you can, or you can find yourself in a situation where you cannot get a green state anymore and you are forced to revert to the initial version.

After [potential] refactoring, go back to red and start writing another test.

Now you are an expert on TDD theory and you have a reference guide here. The next step is **practicing, practicing and practicing**. It is said that to become an expert TDDer you should write at least 1500 tests, so you'd better off starting now. :)

Chapter 11: Testing sqrt()



As a final case study, I want to introduce a very simple SUT: the php *sqrt()* function, which calculates the square root mathematical function on its argument.

For the readers that are now experiencing school reminescences, the square root functions for real numbers is defined as the number that once multiplicated for itself gives the argument of the function. Thus, the square root of 9 is 3, while the function is not defined for negative numbers.
More *complex* extensions of the square root functions are not commonly used in web development and we will stick to this definition, which corresponds to php standard implementation.

Note that I chose a simple case, without external dependencies. Though, this case study is still **generic** as dependencies are always mocked or stubbed out from a SUT, and tests concentrate on its functionalities.

So how can we extensively test the sqrt() function? The first obvious answer would be specifying every positive number as its input, along with the square root found with a pocket calculator. This approach is not feasible as integers are represented with 32 bit in php and, leaving out one bit for sign, there are about 2^31 values to try in the function, which are **too many** test cases for an exhaustive test. If we include also the 64-bit floating point numbers in test cases, the number of cases grows even more, but we will limit this discussion to integer inputs and outputs to exclude floating point precision problems.

Remember that object-oriented systems maintain also state, which means other internal parameters of an object can change and influence the results. Thus exhaustive testing is never used.

Before discussing what possible input values we should consider, let me introduce two tests design methods:

- **black box testing**: deriving test cases from the external interface of the SUT.
- **white box testing**: deriving test cases from the internal structure of the SUT.

While we should strive for decoupling between production classes, the bond between unit tests and their SUTs is forced to be strong. The more we move to an higher view (functional and integration tests), and the more we abstract away to simplify maintenance. Integration and acceptance tests are usually black-box tests which do not care about how we produce a value (how the square root function is calculated or an html page is generated), but only about the final result.

Unit tests instead are focused on ensuring that a unit of functionality works well, and are written to drive design of the SUT.

When we read code coverage reports, we tune unit tests to cover more code paths (but if you do TDD code coverage is always very high). If-else branches, while conditions and earlier returns are exercised by different white-box tests.

If we had the implementation of sqrt() in userland code, we should code different tests that exercise different code branches. Better, we should write different tests and extend the code just enough to make these tests pass.

There is an equivalence relation between input values that stimulate the SUT in the same way (run the same code). We should include in white-box tests only some values from each equivalence class, usually including the boundaries of every range if we are dealing with numeric inputs.

Though, we are limited to black box testing in this case since sqrt() is implemented in C, which does not fall under the umbrella of our code coverage tools.

Note that if we TDD the class, we are doing white box testing and every new test method or input value we add **fail by definition** (red phase). This means that the new code that is added is exercised only by this input and state combinations and was not covered by previous tests.

Then the test passes (green), and the cycle repeats with new white-box tests that do not overlap. So TDD as predicted is a great method for writing unit tests. In the orange phase, we refactor code and it can happen that different tests cover the same code after refactoring. In this case, we can indeed extend refactoring to the tests themselves, but test abundancy is rarely a problem in real world (test performance is) and a long test suite is a safety net for future refactoring, as long as it runs sufficiently fast.

My black box test case for sqrt() is at the end of the chapter, while a white box test case would be the final product of the exercise.

TDD Exercises

11.1 TDD a Calculator class, which has only the method Calculator::sqrt(). Implement the class one test at the time.

Remember that:

- you should add a *failing* test and then improving production code to handle the new test case.

- you can add a test only if the tests are green

- you can add production code only if the tests are red, and if I removed part of your addition the test should return red (ensuring you are not writing unused code)

- you **cannot use the sqrt() function** in this exercise

- you can use a guess-and-check method: to find the square root of 36, try 1, try 2, try 3... until you get to 6. Round the result to the nearest integer.

Code sample

```
<?php

class SqrtBlackTest extends PHPUnit_Framework_TestCase
{
    public function inputNumbers()
    {
        return array(
            array(0, 0),
            array(1, 1),
            array(4, 2),
            array(9, 3),
            array(-1, NAN),
            array(-2, NAN),
            array(1000000, 1000),
        );
    }

    /**
     * @dataProvider inputNumbers
     */
    public function testSquareRootIsCalculated($input, $output)
    {
        $this->assertEquals($output, sqrt($input));
    }
}
```

Glossary

- **Api**: interface that a software program implements in order to allow other software to interact with it. In object-oriented programming, an Api is composed of interfaces and final classes which the client program depends on and that are provided by the original program, which maybe a library or a framework like Zend Framework or PHPUnit.
- **constraint object**: representation of a specification about objects. For instance, a IsEqual constraint object in PHPUnit can be used to check other object are equal to the one passed at construction. In DDD this type of object is called a Specification pattern and it is the personification (or objectification) of an selection criteria: the more you think in objects, the more reusable the resulting code will be.
- **Law of Demeter**: Only talking to immediate object references, often stated as *using only one dot or → in every line of code*. The purpose of this law (actually a suggestion) is to protect a class from changes in far collaborators, by accessing only its immediate friends from its code.
- **DDD aka Domain-Driven Design**: a style of development that focus in domain knowledge, where the core of an application consists in the classes and objects that represents the domain and everything else depends on them.
- **dependency injection**: the process of supplying an external dependency to a software component, instead of forcing it to look up for collaborators. This is in opposition to service locator solutions and hardcoded wiring: DI takes away the wiring responsibility from the class.
- **entity**: a class which purpose is maintain state, with few or no references to external services. User and Post are common examples of entity classes in web applications. For this entities to be testable without using a database, they should be Plain Old Php Objects that do not extend anything.
- **fluent interface**: method chaining that provides the ability to write code like Object.DoSomething().DoSomethingElse().DoAnotherThing(); every method returns the object itself ($this).

- **Miško Hevery**: Agile coach at Google, guru of TDD and design for testability.
- **newable**: class with little behavior and lot of state, such as String or an entity. Not prone to dependency injection since it may be serialized and created at any time.
- **phpDocumentor**: auto-documentation tool for the php language. The tool generate Api documentation containing method signature by parsing standard comment blocks in the php code.
- **PHPUnit**: php instance of the xUnit family of testing frameworks. PHPUnit is the standard for testing every kind of php code and it is used extensively throughout this book.
- **refactoring**: changing a computer program's internal structure without modifying its external functional behavior. Refactoring *improves the design of existing code* and it is often compared to cleaning the dishes.
- **reflection**: process by which a computer program can observe and modify its own structure and behavior, e.g. obtaining the list of a class's methods.
- **require_once()**: statement used in php to include other source files. In php there is no one-time compilation and classes can be included on the fly by requiring other php scripts, a job that is usually performed by an autoloader.
- **Spl**: Standard Php Library, the main object-oriented component of php. It is included by default in every php 5 installation but lacks many functionalities.
- **SUT**: system under test. In unit tests, indicates a class, while in functional and integration tests an object graph.
- **Test doubles**: replacement used in unit tests to isolate the SUT from collaborators. **Stubs** and **mocks** are explained in chapter 7 and 8 and are the mainly used instances of Test doubles. Another type is the Fake object, which is a running implementation of an interface that is particularly useful in testing for its simplicity (e.g. in-memory database instances).
- **TDD**: development practice where first the developer writes a failing automated test case, then produces code to pass that test and finally refactors the new code to acceptable standards. TDD is

explained in chapter 10.

- **weltanschauung**: German term for world view, ultimate global vision of a person or organization. By extension, it means philosophy of life.
- **xdebug**: PHP extension for powerful debugging and testing. This extension is fundamental to gain introspection on executed code and it is used by phpunit 3 to generate code coverage reports.
- **xUnit**: the set of testing frameworks that includes JUnit, PHPUnit, SUnit, Nunit... This frameworks share a very similar Api (test cases, setUp, tearDown(), assert*(), ...)