Become an MVB
Most Valuable Blogger

» Get Noticed
» Get Respect
» Get Published

Visit http://www.dzone.com/aboutmvb
Extensively testing a web application is the only way to make sure that the code you have written really works. Formal proofs of correctness are impossible or impractical to put together for the majority of computer programs. Therefore, actually running code in a sandbox under controlled conditions is the way to uncover bugs and drive the development of new features.

Of course, manual testing cannot be totally substituted by automated approaches, but automating a large part of an application’s tests leads to greater testing frequency, and quicker discovery of issues and regression.

Automated testing is based on the concept of test cases (classes in PHPUnit’s case) that compose a complete test suite. This suite, or a subset of it, can be run at will to check the production code correctness.

In this approach, the simplest and most effective way to write tests is to write code—simple code, but with the mandatory expressiveness of imperative languages that should give you the needed freedom.

Essentially, this is a class that extends PHPUnit_Framework_TestCase so that it can be run by the /usr/bin/phpunit script.

The methods whose names start with ‘test’ are executed, one at a time, on a different instance of this Test Case. You can execute whichever code you prefer in order to produce a set of results to confront with the ones you expect.

You can use methods on the Test Case that start with ‘assert’ to execute checks on the result of your computations that will ensure the system is behaving correctly. PHPUnit provides many simple assertion methods, but you can always define your own by adding private methods.

Executing phpunit filename.php from the command line would run the test. You can also run all the tests in a certain folder (subfolders included) simply by passing a path to the directory.

This was the simplest test we could possibly write: it will provide good practice to start with and you can augment the complexity as you gain confidence about the System Under Test (SUT).
Now that we are capable of running a test on your machine, let’s gain confidence and expand the test a bit:

```php
<?php
class ArraySumTest extends PHPUnit_Framework_TestCase
{
    public function testSumTheValuesOfAnArrayWithManyElements()
    {
        $sum = array_sum(array(1, 2, 3, 4, 5, 6));
        $this->assertEquals(21, $sum);
    }

    public function testSumTheValuesOfAnArrayWithOneValue()
    {
        $sum = array_sum(array());
        $this->assertEquals(0, $sum);
    }

    public function testSumTheValuesOfAnArrayWithFloatValues()
    {
        $sum = array_sum(array(1, 2.5));
        $this->assertEquals(3.5, $sum);
    }

    public function testSumTheValuesOfAnArrayWithManyElements()
    {
        $sum = array_sum(array(1, array(2, 3)));
        $this->assertEquals(6, $sum);
    }
}
```

When run, this test gives the following result:

```
20:00:06 [giorgio@desmond /tf/PHPUnit_refcard]
$ phpunit --colors houttuit/test.php
PHPUnit 3.5.0 by Sebastian Bergmann.

....F
Time: 0 seconds, Memory: 3.50MB
There was 1 failure:
1) ArraySumTest::testSumTheValuesOfAnArrayWithManyElements
   Failed asserting that integer:2, matches expected integer:5.
/home/giorgio/dropbox/tf/PHPUnit_refcard/houttuit/test.php:13
FAILURES!
Tests: 5, Assertions: 5, Failures: 1
```

We were asserting that the production code performed a task that it couldn’t. The test tells us it is not living up to our expectations.

Before diving into PHPUnit’s features, I want to highlight some methods that complement assertions in the flow control of PHPUnit.

**ASSERTIONS**

Assertion methods are used to check the result of your test’s execution with predefined expected values. All these methods are available on `$this`, as they are defined on `PHPUnit_Framework_TestCase`. I report here the ones that, by experience, are the most handy and widely called in my test suites.

<table>
<thead>
<tr>
<th>Method</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$this-&gt;fail($message = '')</code></td>
<td>Will instantly make the test fail, like in the case an assertion fails.</td>
</tr>
<tr>
<td><code>$this-&gt;markTestSkipped()</code></td>
<td>Tells PHPUnit that this test should be interrupted and marks it as skipped in the results.</td>
</tr>
<tr>
<td><code>$this-&gt;markTestIncomplete()</code></td>
<td>Tells PHPUnit that this test is incomplete and shouldn’t be run. It will be marked with an (i) in the results and will turn the green bar of PHPUnit into an orange one.</td>
</tr>
</tbody>
</table>

Basic assertions are the ones you always see in introductory material, and can serve many needs. They are also versatile, since corner cases can be reconduced to assertTrue() and assertEquals() in nearly any scenario.

<table>
<thead>
<tr>
<th>Method</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>assertEqual()</td>
<td>Fails if the two arguments are not equal (checks with the == operator).</td>
</tr>
<tr>
<td>assertSame()</td>
<td>Fails if the two arguments are not identical (checks with ===).</td>
</tr>
<tr>
<td>assertTrue()</td>
<td>Fails when this argument is not true.</td>
</tr>
<tr>
<td>assertFalse()</td>
<td>Fails when this argument is not false.</td>
</tr>
<tr>
<td>assertEmpty()</td>
<td>Fails when empty() on the parameter is false.</td>
</tr>
<tr>
<td>assertNull()</td>
<td>Checks if the variable is the special value NULL or not.</td>
</tr>
<tr>
<td>assertInstanceof()</td>
<td>Checks that an object is an instance of (using this operator) the class or interface passed.</td>
</tr>
<tr>
<td>assertInternalType()</td>
<td>Checks if a scalar is of a particular PHP-type like integer or boolean.</td>
</tr>
</tbody>
</table>

Here is a first sample of these methods in action:

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

    public function testBasicAssertionsWorkAsExpected()
    {
        $this->assertEquals(1, 1);
        $this->assertEquals(1, 1.0);
        $this->assertEquals(1, "1");
        $this->assertLessThan(1, 1.1);
        $this->assertGreaterThanOrEqual(1, 1.01);
        $this->assertGreaterThanOrEqual(1, 1.0);
        $this->assertLessThanOrEqual(1, 1.0);
        $this->assertLessThan(1, 1.01);
    }
}
```

There are more elaborate assertions which are indeed frequently called:

```php
<?php
class ArrayAndStringsAssertionsTest extends PHPUnit_Framework_TestCase
{
    public function testArrayAndStringsAssertionsWorkAsExpected()
    {
        $this->assertArrayHasKey('key', array('key' => 'value', 'anotherKey' => 'anotherValue'));
        $this->assertArrayNotHasKey('key', array('anotherKey' => 'anotherValue'));
        $this->assertContainsOnly('string', array('value', 'otherValue'));
        $this->assertEquals(42, (int) "42");
        $this->assertEquals(3.000, 3.000, 'Floats should be compared with a tolerance', 0.0001);
        $this->assertEquals(1, "1");
        $this->assertEquals(1, 1);
    }
}
```

Here is a sample test case:

```php
<?php
class ArrayAndStringsAssertionsTest extends PHPUnit_Framework_TestCase
{
    public function testArrayAndStringsAssertionsWorkAsExpected()
    {
        $this->assertArrayHasKey('key', array('key' => 'value', 'anotherKey' => 'anotherValue'));
        $this->assertArrayNotHasKey('key', array('anotherKey' => 'anotherValue'));
        $this->assertContainsOnly('string', array('value', 'otherValue'));
        $this->assertEquals(42, (int) "42");
        $this->assertEquals(3.000, 3.000, 'Floats should be compared with a tolerance', 0.0001);
        $this->assertEquals(1, "1");
        $this->assertEquals(1, 1);
    }
}
```

There are also special cases:

```php
<?php
class ArrayAndStringsAssertionsTest extends PHPUnit_Framework_TestCase
{
    public function testArrayAndStringsAssertionsWorkAsExpected()
    {
        $this->assertArrayHasKey('key', array('key' => 'value', 'anotherKey' => 'anotherValue'));
        $this->assertArrayNotHasKey('key', array('anotherKey' => 'anotherValue'));
        $this->assertContainsOnly('string', array('value', 'otherValue'));
        $this->assertEquals(42, (int) "42");
        $this->assertEquals(3.000, 3.000, 'Floats should be compared with a tolerance', 0.0001);
        $this->assertEquals(1, "1");
        $this->assertEquals(1, 1);
    }
}
```
PHPUnit: Professional Test Driven Development

Here is an example of them in action:

```php
<?php

class ExoticAssertionsTest extends PHPUnit_Framework_TestCase
{
    public function testExoticAssertionsWorkAsExpected()
    {
        $this->assertFileExists(__FILE__);  // this test won't fail, because the file exists
        $this->assertGreaterThan(23, 42);   // this test will pass
        $this->assertFileEquals(__FILE__, __FILE__); // this test will fail
    }
}
```

Custom assertions are great tools (you just have to define an `assert*()` method). However, taking advantage of the assertions already supported by the framework will not require that you reinvent the wheel, or extend your base Test Case class in all your tests.

All assertion methods take a last optional argument `$message`, which is displayed when they fail. You may enhance your tests error messages when they are not clear with this additional parameter.

**FIXTURES**

Fixtures are a common way to set up objects that are used in all test methods before each test run.

In PHPUnit, fixtures can be implemented with the `setUp()` and `tearDown()` hook methods, which can populate `$this` private fields, preparing them for usage by the single test methods.

There are also two static methods, `setUpBeforeClass()` and `tearDownAfterClass()`, that are called only once for each test case class. Because of their static modifier, they can only access static properties and are typically used for resources that are particularly heavy to set up, like database connections.

Due to this isolation property, `tearDown()` is usually not needed, so you won’t have to reset your fields between the test methods. Therefore, each test method is run in isolation and won’t interfere with the `$this` fields set in previous runs.

Here is a test case which uses fixtures implemented with `setUp()`.

```php
<?php

class FixtureTest extends PHPUnit_Framework_TestCase
{
    private $systemUnderTest;

    public function setUp()
    {
        $this->systemUnderTest = new stdClass;
    }

    public function testFixtureWorkAsExpected()
    {
        $this->assertEquals(16, $this->systemUnderTest->field);
        $this->assertInstanceof('stdClass', $this->systemUnderTest);
        $this->assertTrue(isset($this->systemUnderTest->field));
        $this->assertLessThan(16, 15);
        $this->assertGreaterThan(23, 42);
        $this->assertFileEquals(__FILE__, __FILE__);
        $this->assertFileExists(__FILE__);
    }
}
```

In case the SUT experiences a large regression, the errors presented by a test run would be much more focused, consisting only of the tests that failed first and excluding their dependent siblings.

Defining dependencies is usually only a good idea inside the same test case class. In fact, it is the only scale supported by this feature. The dependent test will be marked as `skipped` when the first one fails, otherwise it will be run normally.

Moreover, the first test can pass a computation result to its dependent siblings. This annotation tells the framework to pass the computation result to the dependent test, without the test needing to be run explicitly.

In the example above, the first test would pass the result of `$this->systemUnderTest->field` to the second test.

**ANNOTATIONS**

PHPUnit supports the use of some annotations on test case and single test methods with the goal of transforming into declarative options some common behaviors diffused in test suites. For example, running the same test multiple times with a different set of input data is an operation common enough to warrant its own annotation. The mechanism of these multiple runs isn’t duplicated throughout the test suite of our projects, instead it is kept in the framework.

PHP does not have native support for annotations, so they are embedded into the docblocks of classes and methods, much like the `@param` and `@return` annotations used for API documentation. These annotations will save you from writing boilerplate code you would otherwise end up repeating in a bunch of tests that all look alike, or inheriting from some base test case class that gets longer and longer.

**@dataProvider**

This annotation tells PHPUnit to run a single test method multiple times by passing a different data set as method parameters each time.

PHPUnit will display a different failure for each of the different runs, so that multiple tries are independent and one failing does not affect the subsequent execution of the others.

Running the same test code with a foreach() cycle would not have the same effect.

**@depends**

This annotation declares a test as dependent on a previous one, saying essentially that when a basic test on the SUT fails, it is not worthy to even try to run a more complex, related one.

In the example above, the second test will exercise `remove()` on the passed collection, and check that the element is not present anymore.

DZone, Inc. | www.dzone.com
@expectedException (with @expectedExceptionCode and @expectedExceptionMessage)

This annotation asserts that the code contained in the test methods throws an exception, and optionally checks its code and its message.

Note that any line of code can be throwing the exception, as long as it is called inside the test method. If you want a more accurate check on which statement should raise an exception, stick to try/catch blocks.

@group

Grouping aids you in running the exact amount of tests during development, and not the entire suite, which can require much more time. You will get feedback on a more frequent basis, and only from the code you select. This leaves the rest of the code base for subsequent integration (pre-commit or pre-push).

This annotation is not applicable to test methods, but only on test cases. It defines a test case as pertaining to a particular group that can be used as a filter to run all the grouped tests independently from the rest of the suite.

A test case can be in multiple groups. They can be thought of as labels rather than folders (which are an independent filter).

For examples, you may use annotations such as @group acceptance (for Selenium-based tests); @group functional (for tests which do not exercise a single unit and, hence, are a bit slower to run), and @group ticket-42 (for the tests regarding a particular bug fix).

The command used to filter tests of a particular group is *phpunit --group*, which will be treated later in the command line section. However, you can also filter groups in independently from the rest of the suite.

A test case can be in multiple groups. They can be thought of as labels rather than folders (which are an independent filter.)

@endTestInSeparateProcesses

This too is a test case annotation. It is not diffused and should be used only for special case tests. It runs tests in different executions of the *php* command, so that they do not affect each other. This mechanism can be useful when testing something related to autoloading or a similar necessary global state. For example, you may be testing that the autoloader loads the same class multiple times under different conditions, and this can only be done in different processes.

```php
/**
 * @group acceptance
 * @group someOtherGroup
 */
class AnnotationsTest extends PHPUnit_Framework_TestCase
{
    public static function dataToFeed()
    { return array(10, 12, 22); }
}
```

## Test Doubles

Test Doubles are substitutes for real objects that you don’t want to involve in your tests. By injecting Test Doubles in your System Under Test, you can effectively isolate it from the rest of the system and let it call its collaborators without null checks. You can also define what the Test Doubles expect as method parameters, or what they should return.

The taxonomy of Test Doubles comprehends:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dummies</td>
<td>Objects that exist only to satisfy type hints and runtime checks. They are passed around but no methods are called on them.</td>
</tr>
<tr>
<td>Stubs</td>
<td>Objects that return canned results when their methods are called.</td>
</tr>
<tr>
<td>Mocks</td>
<td>Stubs that can also check what parameters are passed to them.</td>
</tr>
<tr>
<td>Fakes</td>
<td>Real implementations of collaborators, but much more lightweight than the real object.</td>
</tr>
</tbody>
</table>

PHPUnit offers support for automatic generation and instantiation of Dummies, Stubs, and Mocks. It subclasses the defined class, overrides methods and eval() the resulting code. In PHPUnit, Mock is used as an umbrella term covering any Test Double category.

The getMock() or getMockBuilder() methods can be called to obtain the mock, or a Builder implementation over the mock itself. They accept an interface or a concrete class as the first argument. getMockForAbstractBase is the equivalent of getMock() for this special case.

I contributed getMockBuilder() to provide a cleaner way for complex mock instantiation, which usually requires calls with 7 arguments to getMock(). It provides a fluent interface with different methods to set the creation options one by one, prior to creating the mock.

Once you have a mock, you can perform the expects() call on it to create an expectation object. The expectation object then presents different methods which provide a fluent interface:

<table>
<thead>
<tr>
<th>Method</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>method()</td>
<td>Defines the name of the method it refers to.</td>
</tr>
<tr>
<td>with()</td>
<td>Specifies assertions to make on the parameters passed. In the simplest cases, you call it with the value you would use to call the method, in the identical order.</td>
</tr>
<tr>
<td>will()</td>
<td>Defines the behavior of the overridden method as traits such as what to return or whether to throw an exception.</td>
</tr>
</tbody>
</table>

Here is a test case which covers many instances of stubs and mocks usage:
class TestDoublesTest extends PHPUnit_Framework_TestCase {
    public function testInstantiatesAMock() {
        $dummy = $this->getMock('stdClass');
        $this->assertInstanceOf('stdClass', $dummy);
    }
    public function testInstantiatesAMockStub() {
        $stackStub = $this->getMock('SplStack');
        $stackStub->expects($this->any())->method('push')->will($this->returnValue(42));
        $this->assertContains(42, $stackStub->pop());
    }
    public function testSetsUpAMockMethodWithACallback() {
        $callback = function($argument) {
            $map = array('key' => 'value', 'otherKey' => 'otherValue');
            return $map[$argument];
        };
        $stackStub = $this->getMock('SplStack');
        // don't ask me why a Stack has getter and setters
        // I use is only because of its availability
        $stackStub->expects($this->any())->method('offsetGet')->will($this->returnValueCallback($callback));
        $this->assertContainsValues(['value', $stackStub->offsetGet('key')]);
    }
    public function testThrowsAnException() {
        $stackStub = $this->getMock('SplStack');
        $stackStub->expects($this->any())->method('push')->will($this->throwException(new InvalidArgumentException));
        $stackStub->push(42);
    }
    public function testInstantiatesAMockAndPutOtherExpectationsOnParameters() {
        $stackMock = $this->getMock('SplStack');
        $stackMock->expects($this->once())->method('offsetSet')->with(2, new stdClass);
        // I use it only because of its availability
        $stackMock = $this->getMock('SplStack');
        $stackMock->expects($this->once())->method('push')->will($this->returnValue(42));
        $stackMock->push(42);
    }
    public function testInstantiatesAMockAndPutExpectationOnMultipleParameters() {
        $stackMock = $this->getMock('SplStack');
        $stackMock->expects($this->any())->method('offsetSet')->with(42, $this->isType('int'))->with(2, $this->isType('int'))->with(42, $this->isType('int'))->willReturn(42);
        $stackMock->offsetSet(42, 42);
    }
    public function testInstantiatesAMockAndPutAnyOtherExpectationsOnParameters() {
        $stackMock = $this->getMock('SplStack');
        $stackMock->expects($this->once())->method('offsetSet')->with($this->isNull(), 'otherClass');
        $this->assertInstanceOf('stdClass', $dummy);
    }
    public function testInstantiatesAMockAndPutAnyOtherExpectationsOnParametersDifferentFromNullTo() {
        $stackMock = $this->getMock('SplStack');
        $stackMock->expects($this->once())->method('offsetSet')->with($this->isNull(), $this->identicalTo('42'));
        $stackMock->offsetSet(2, 42);
    }
    public function testInstantiatesAMockAndPutAnyOtherExpectationsOnParameters() {
        $stackMock = $this->getMock('SplStack');
        $stackMock->expects($this->once())->method('offsetSet')->with($this->isNull(), $this->identicalTo('42'));
        $stackMock->offsetSet(2, new stdClass);
    }
    public function testCallsAMockTwice() {
        $stackMock = $this->getMock('SplStack');
        $stackMock->expects($this->once())->method('push')->willReturn(42);
        $stackMock->push(23);
        $stackMock->push(42);
    }
    public function testShouldNotCallAMock() {
        $arrayIteratorMock = $this->getMockBuilder('ArrayIterator')->setMethods(['next'])->getMock();
        $arrayIteratorMock->next();
        $this->assertNull($arrayIteratorMock->current());
    }
}

COMMAND LINE USAGE

The phunit script is the main means to run PHPUnit-based test suites, even if there are ways to bypass it like Phing (http://phing.info/trac/) tasks for purposes of Continuous Integration.

Knowing how to use *phunit* effectively is therefore crucial to leverage your test suite power. These are the various options available to modify its behavior:

<table>
<thead>
<tr>
<th>Switch</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>--configuration &lt;file&gt;</td>
<td>Includes a phunit.xml file different from the one in the working directory or excludes its automatic recognition.</td>
</tr>
<tr>
<td>--no-configuration</td>
<td></td>
</tr>
<tr>
<td>--coverage-html &lt;directory&gt;</td>
<td>Generates an HTML report on code coverage in the specified folder. The xdebug extension is required for code coverage to be collected and the test execution will be much slower.</td>
</tr>
<tr>
<td>--colors</td>
<td>Specifies to use colors in the output as a nice way to visualize the test success.</td>
</tr>
<tr>
<td>--bootstrap &lt;file&gt;</td>
<td>Defines a PHP file to include before starting the test suite execution. It is included also in the configuration from some releases.</td>
</tr>
<tr>
<td>--filter</td>
<td>Lets you filter test methods that match the passed pattern. You won’t have to comment on the other tests when you want to focus on a single one.</td>
</tr>
<tr>
<td>--group</td>
<td>Filters the tests defined by the current suite and runs only the ones in the specified group.</td>
</tr>
<tr>
<td>--exclude-group</td>
<td>Runs all the tests except the ones in the chosen group.</td>
</tr>
<tr>
<td>--list-groups</td>
<td>Shows all the groups available for running independently.</td>
</tr>
<tr>
<td>--include-path</td>
<td>Sets a particular include_path ini directive for running the test.</td>
</tr>
<tr>
<td>-d key=value</td>
<td>Sets a php.ini value (temporarily)</td>
</tr>
<tr>
<td>--verbose</td>
<td>Displays a list of test case names as the execution goes along, so that you can catch errors during long run just when they happen, instead of waiting for the whole test suite to finish.</td>
</tr>
<tr>
<td>--version</td>
<td>Displays current version of PHPUnit. Handy to know, since running it without parameters starts the whole test suite. -v is not supported.</td>
</tr>
<tr>
<td>--testdox</td>
<td>Generates a “kind” of Agile documentation from the test names. If you write test names well, it documents what your class or component under test does for a living.</td>
</tr>
</tbody>
</table>

CONFIGURATION

PHPUnit supports configuration of the test suite via an XML file that can be stored under version control. Common options like bootstrap files, logging formats or filters can be saved in this file instead of being passed with each PHPUnit command. PHPUnit looks for phpunit.xml and phpunit.xml.dist, which are assumed to be present in the working directory. Usually one of them is kept in the test suite root directory.

The common practice is to store phpunit.xml.dist in the source control system and ignore phunit.xml, so that users can personalize their run if they do not have the tools for running all tests available [such as Selenium or staging http servers].
PHPUnit

It can also be the case that you do not want to run slow integration tests sometimes. With a phpunit.xml.dist present however, you will be able to run tests just after a checkout.

Root element

<phpunit> is the root element. It accepts various attributes such as “bootstrap”, which specifies a bootstrap file to execute prior to the tests, or “colors”, which prescribes the use of colors like red and green in the output.

Test suites

The <testsuites> and <testsuite> elements let you group tests from different folders. They can contain these two elements:

- <file>, which defines a single PHP file.
- <directory>, which will include all the tests inside that directory or a subdirectory. <directory> accepts a suffix attribute to filter the test case name: by default it is ‘Test’.

Filtering, logging, and more

Inside the configuration, you also have different modifiers available. You can:

- Filter tests of a certain <group> to run (or not run) them.
- Define code coverage inclusion (and exclusion) of files. The <filter> element refers to code coverage.
- Log results and statistics in various formats, with the <logging> element. It may contain various <log> entries, of various types: coverage-html, coverage-xml, json, tap, junit, testdox-html, testdox-text.
- Attach listeners with the <listeners> and <listener> elements.
- Configure a Selenium RC server to run acceptance tests, with the <selenium> and <browser> elements.

ABOUT THE AUTHOR

Giorgio Sironi is a Bachelor of Computer Engineering and works as a PHP software architect while continuing his studies in a Master Program at Politecnico di Milano. During the last six years he transitioned from his early work as a websites developer to the creation of web-based applications. He practices Test-Driven Development every day and believes in testing as a design tool. Giorgio is an international speaker at PHP conferences and you may have already heard of him as the author of the Practical PHP Testing free ebook, or the “Practical PHP Patterns” series on DZone.

RECOMMENDED BOOK

Test-Driven Development (TDD) is now an established technique for delivering better software faster. TDD is based on a simple idea: Write tests for your code before you write the code itself. However, this “simple” idea takes skill and judgment to do well. Now there’s a practical guide to TDD that takes you beyond the basic concepts. Drawing on a decade of experience building real-world systems, two TDD pioneers show how to let tests guide your development and “grow” software that is coherent, reliable, and maintainable.