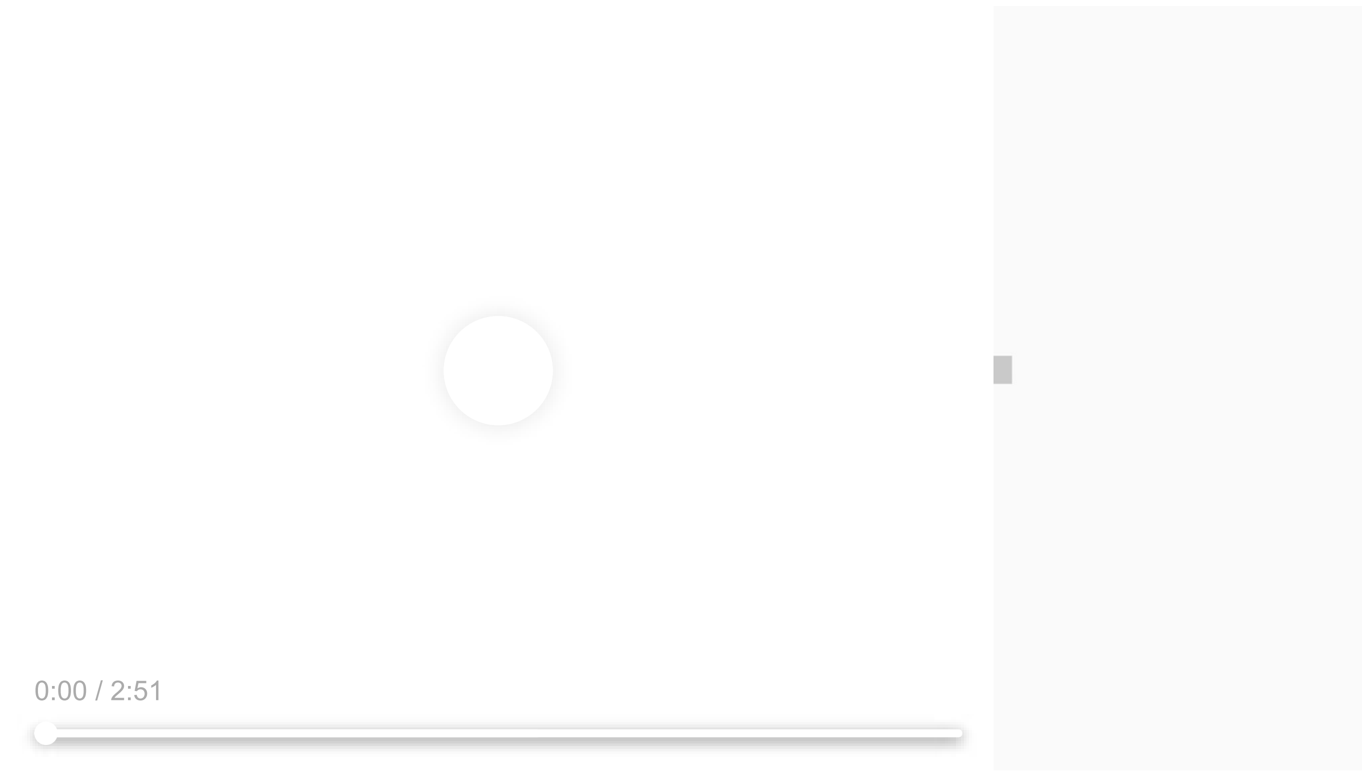
PHPUnit | Day 1 | Pre-Work

PHPUnit

DAY 1

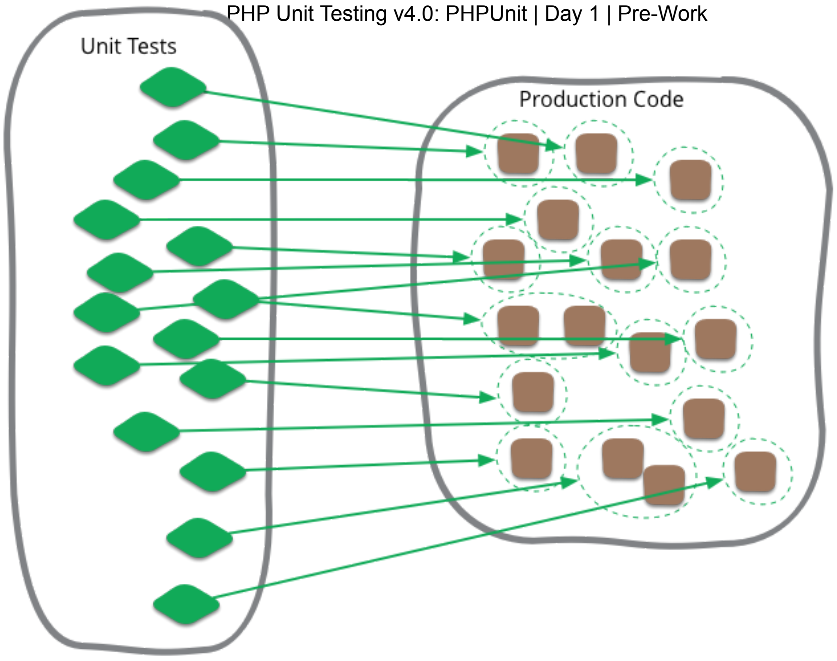
Table of contents:

What is Unit Testing



To view this video please enable JavaScript, and consider upgrading to a newer web browser

Unit testing is a software testing method by which individual units of source code are tested to determine whether code is fit for use.
Unit Testing is **testing isolated units of code** to ensure correctness: the function (or a method) should perform as expected, which we usually check by comparing the generated output with the expected output.



This also means that we organize our code into small units that can be independently run and operated, so we can perform individual & isolated tests on functions/methods.

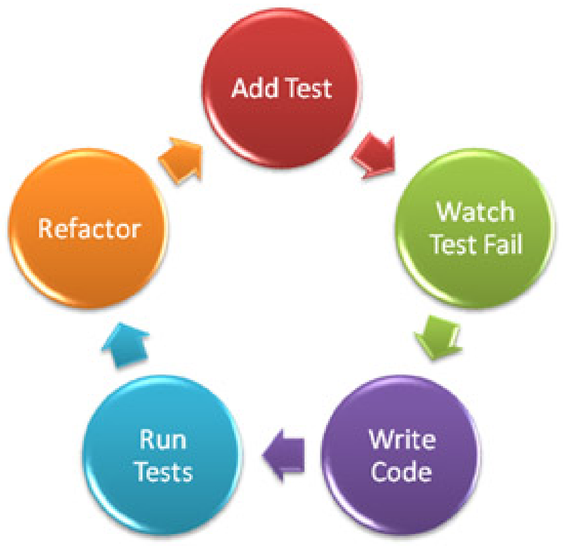
Benefits of Unit Testing

- Ensures code works correctly now and in the future
- Adds additional documentation
- Reduces the chances of bugs or issues
- Improves the ability to refactor (already existent code)
- Helps write better designed code
- Focuses just on a small parts of the software system - isolated mode
- Considered to be significantly faster than other kinds of tests

TDD (Test Driven Development)

Test Driven Development is the way to develop better software with less bugs and more time for building new features. The generally accepted definition means that we **write the tests first**, and then write the code that should be tested. Accordingly, the steps in writing code according to TDD are:

1. Write test first
2. Run test (ensure that the test fails, since there is no existing code)
3. Write featured code (that should be tested)
4. Run test again
5. Repeat until the test is passed



Automatic vs. Manual Testing

Unit testing makes it easy to never forget a special case and add new special cases if needed. Automated unit testing allow us to run our tests quickly and in an automated fashion. You should write them isolated from all of your code that actually performs something.
Now, covering special cases is always a tricky business. Every special case that you have to test requires a bunch of work to make everything set-up.
However, once you get your tests automated, you will be way faster in checking your code compared to any kind of testing that rely on manual input. Your testing team will get the opportunity to focus on testing the changes throughout the application, as well as on the overall application operations.
Be aware: unit testing is not magic that will make your software development easy, but it will help you to write better software with less defects in less amount of time. Therefore, unit testing is considered a good and useful practice that you should incorporate in your coding standards.

Other types of Testing

There are also other types of testing at all levels of an application:

- **Integration testing** - deals with the integration of the software
- **Functional testing** - ensures the function of the application
- **Behavior testing** - Takes a series of steps to ensure some behavior in the application is correct
- **Acceptance testing** - Typically made by stakeholders to ensure it meets whatever requirements were designed for it

Install PHPUnit

There are several different ways how PHPUnit can be installed. Whatever install path you choose, you are basically adding the PHPUnit library (as a dependency) to your project files.

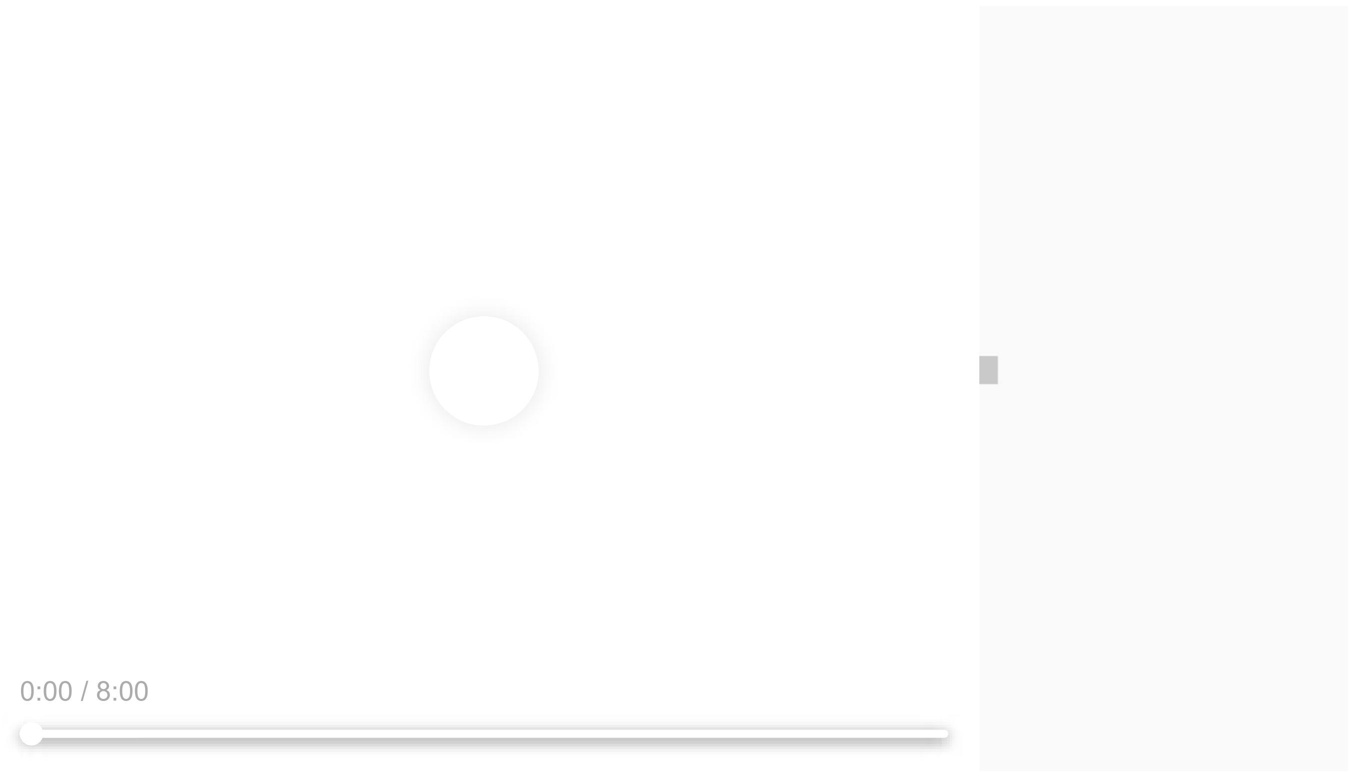
First, check the recent version of PHP you have installed with XAMPP. Open a console over **XAMPP (Windows)**, or open a console in your Apache root folder and check the status with

```
php --version
```

Note: PHPUnit 6.+ requires PHP 7; using the latest version of PHP is highly recommended.

There are several different ways how you can install and use PHPUnit. In this course, will use the more complex one, the **Composer** tool. Composer is a dependency manager for PHP applications, very similar in function to **npm** that we have already used for Angular. Composer is configured based on JSON dependency description file.

Composer.phar installation and usage



To view this video please enable JavaScript, and consider upgrading to a newer web browser

Windows

For **Windows**, you use the installation from https://getcomposer.org/
Currently, you need to go to “Getting Started” section:
https://getcomposer.org/doc/00-intro.md#installation-windows and follow the Installer instructions after downloading EXE file. Note that a restart of your console/operating system may be needed in the process.

Linux

For **Linux** Ubuntu based distros, do not install Composer through your distro (**do not use** something like **sudo apt-get install composer**), otherwise, you will end up with an old version of Composer.
You should open https://getcomposer.org/download/ and follow the instructions for downloading & local install (important note: due to different composer versions, your SHA384 may differ - please copy/paste commands in console directly from browser: **do NOT use the code below, given ONLY as illustration**:

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.p
php -r "if (hash_file('SHA384', 'composer-setup.php') ===
'544e09ee996cdf60ece3804abc52599c22b1f40f4323403c44d44fdfdd586475ca
{ echo 'Installer verified'; } else { echo 'Installer corrupt'; unl
echo PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

This script (that you can execute line by line in your console) downloads the latest **composer.phar** version to your current folder.

If you would prefer to use composer.phar globally, copy it to a folder in your PATH, for instance:

```
sudo mv composer.phar /usr/local/bin/composer
```

MacOS

Downloading directly from the composer website
Copy and paste that link – https://getcomposer.org/composer.phar – to your browser. It’s always the latest version of Composer.
After getting it, open your terminal to test it. You need just run that command (depending on the actual download path):

```
php ~/Downloads/composer.phar --version
```

If you would like to use Composer without the need to write a path every time , do the following:

```
cp ~/Downloads/composer.phar
/usr/local/bin/composer
sudo chmod +x /usr/local/bin/composer
```

Afterwards, Composer should be accessible in the console regardless of your current path.

More information about installing Composer on a Mac can be found here:

https://pilsniak.com/install-composer-mac-os/

Composer installation check

Check the Composer installation over the console with

```
composer --version
```

Now, you are ready to use Composer in your PHP projects.

Addendum: how Composer works

Note: following chapter demonstrates Composer in action and is not directly connected to PHPUnit tests. You can skip it during the first run.
Create a **phpunit** folder, preferably under your Apache root folder (something like **/var/www/html/** or **c:\xampp\htdocs**, depending on your installation), and save the following file as **composer.json**:

```
{
"name": "johndoe/phpunit",
"authors": [
     {
         "name": "John Doe",
         "email": "john.doe@your-mail-address.com"
     }
 ],

"require": {},

"require-dev": {
     "phpunit/phpunit": "^6.2"
 },

"autoload": {
      "psr-4": {
          "TDD\\": "src/"
      }
},

"autoload-dev": {
      "psr-4": {
          "TDD\\Test\\": "tests/"
      }
}


}
```

This is a basic **composer.json** file.

Note: you can also create your own composer file with:

```
$ composer init
```

in the console.

Now, we are going to add some development requirements to our project. With

```
$ composer require --dev phpunit/phpunit ^6.2
```

you will add a dependency to your composer.json that requires phpunit 6.2 and higher, so your composer.json will look like:

```
{
  "name": "johndoe/phpunit",
  "authors": [
      {
          "name": "John Doe",
          "email": "john.doe@your-mail-address.com"
      }
  ],
  "require": {},
  "require-dev": {
      "phpunit/phpunit": "6.2"
  },
```

The previous command is also triggering **composer install** , which will download several files into the **/vendor** folder of your **phpunit** folder. This can take some time, depending on your composer buffer and internet connection.
Now, run the command in console:

```
vendor/bin/phpunit --version // On MacOS and Linux
vendor\bin\phpunit --version // On Windows
```

This way you can verify that PHPUnit is installed successfully on your **composertest** folder.
Note: more information about the **composer.json** data model can be found under: https://getcomposer.org/doc/04-schema.md

First Unit Test

0:00 / 4:46

To view this video please enable JavaScript, and consider upgrading to a newer web browser

Usually, when a large project is developed, we will divide our sources into several folders:

- **phpunit** folder under your Apache root folder
- **phpunit/src** folder for our executable sources, and
- **phpunit/tests** folder for our test

Create **composer.json** in the **phpunit** folder:

```
{
"name": "johndoe/phpunit",
"authors": [
     {
         "name": "John Doe",
         "email": "john.doe@your-mail-address.com"
     }
 ],

"require": {},

"require-dev": {
     "phpunit/phpunit": "^6.2"
 },

"autoload": {
      "psr-4": {
          "TDD\\": "src/"
      }
},

"autoload-dev": {
      "psr-4": {
          "TDD\\Test\\": "tests/"
      }
}


}
```

Within the src directory create a file called **Receipt.php**:

```
<?php
namespace TDD;

class Receipt {
  public function total(array $items = []) {
  // takes array of items, by default an empty array
  return array_sum($items);
  // array_sum is built in function which sum up
  // an array passed as an argument.
  // We could of course build our own sum solution,
  // but here we want to test "right" against "wrong" sums
  }

  public function tax($amount, $tax) {
          return ($amount * $tax);
       }

}
```

Every method and class that we create needs to have a corresponding test class (an ideal test case).
Within the phpunit directory, create a new directory called **tests**. Inside it create a new file called **ReceiptTest.php:**

```
<?php
namespace TDD\Test;
      // declare(strict_types=1);

use PHPUnit\Framework\TestCase; // imports PHPUnit core
class called TestCase
use TDD\Receipt; // imports Receipt class
// Create Test Class

/*
* @covers TDD\Receipt
*/
class ReceiptTest extends TestCase {

  public function testTotal() {

      $Receipt = new Receipt(); // dummy object for testing


      $input = [0, 2, 5, 8]; // put the inputs in its own
array
      $output = $Receipt->total($input); // store the sum of
the array within a $output variable
      $this->assertEquals( // Built-in assertEquals() method
for defining the test. It accepts three parameters
              14, // 1st param is expected value
              $output, // 2nd param is the $output variable
              'When summing the total should equal 15' //
3rd param is a message displayed in case of failure
              );

  }


}
```

Run in your console under in folder phpunit:

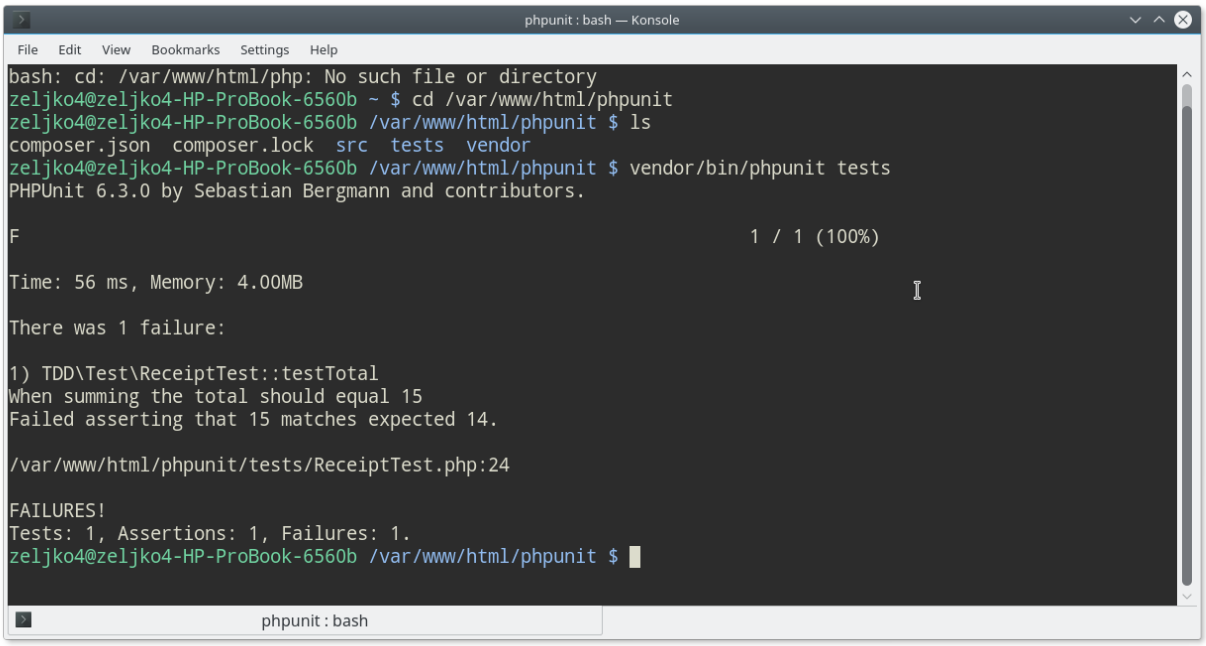
```
composer install
```

Note: to make **composer.json** change(s) actuable, you need to run **composer upgrade** and/or **composer install** after a change.

Finally, run following command in your console:

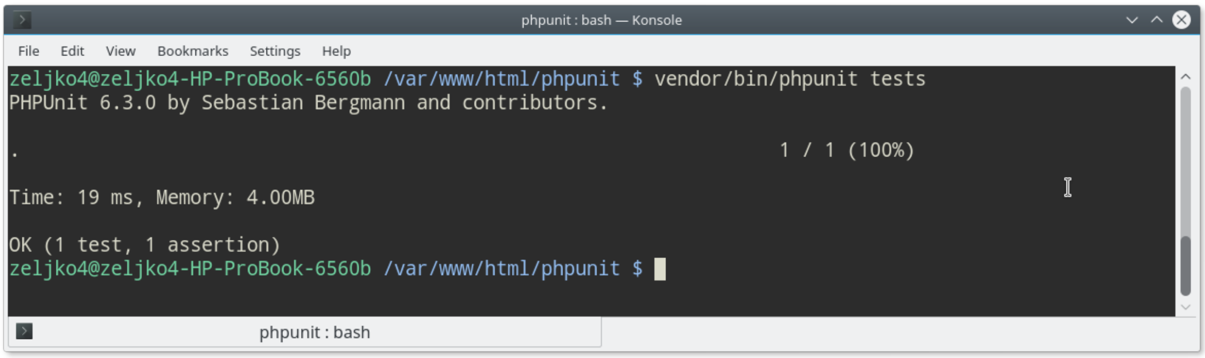
```
vendor/bin/phpunit tests // on Mac
vendor\bin\phpunit tests // on Windows
```

Our test fails because the expected value is 14 instead of 15 (0 + 2 + 5 + 8)! PHP Unit immediately reports this error as follows:



```
bash: cd: /var/www/html/php: No such file or directory
zeljko4@zeljko4-HP-ProBook-6560b ~ $ cd /var/www/html/phpunit
zeljko4@zeljko4-HP-ProBook-6560b /var/www/html/phpunit $ ls
composer.json  composer.lock  src  tests  vendor
zeljko4@zeljko4-HP-ProBook-6560b /var/www/html/phpunit $ vendor/bin/phpunit tests
PHPUnit 6.3.0 by Sebastian Bergmann and contributors.

F                                                                  1 / 1 (100%)

Time: 56 ms, Memory: 4.00MB

There was 1 failure:

1) TDD\Test\ReceiptTest::testTotal
When summing the total should equal 15
Failed asserting that 15 matches expected 14.

/var/www/html/phpunit/tests/ReceiptTest.php:24

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
zeljko4@zeljko4-HP-ProBook-6560b /var/www/html/phpunit $
```

Go back to **ReceiptTest.php** file and change (in the function testTotal()) the first parameter of **$this**->assertEquals from **14** to **15**, then save the file. Rerun the test in the same way as in the previous step (vendor/bin/phpunit tests or vendor\bin\phpunit tests). Our test should pass now:



```
zeljko4@zeljko4-HP-ProBook-6560b /var/www/html/phpunit $ vendor/bin/phpunit tests
PHPUnit 6.3.0 by Sebastian Bergmann and contributors.

.                                                                  1 / 1 (100%)

Time: 19 ms, Memory: 4.00MB

OK (1 test, 1 assertion)
zeljko4@zeljko4-HP-ProBook-6560b /var/www/html/phpunit $
```

We are done with our first unit test.

Note: pay attention to one small detail: **composer.json** has instructed our phpunit that some classes should be **autoloaded** (from folder “src”, in this case). This loader is buried in the vendor/autoload.php automatic call (through PHPUnit). Autoloading of php classes is principally a tricky subject and you should approach in composer.json very carefully.

PHPUnit closures and Arrange-Act-Assert Pattern

One of the most time-consuming parts of writing tests is writing the code to set the world up in a known state (for instance, all of your objects created from a database table) and then returning it to its original state when the test is complete. This known state is called the **fixture** of the test.

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

Open your **ReceiptTest.php** file. First we will need the two methods, **setup()** and **tearDown()** to create and destroy our test environment.

Refactor your existing code as follows in order to meet the rules of the pattern:

```
<?php
namespace TDD\Test;
      // declare(strict_types=1);

use PHPUnit\Framework\TestCase; // imports PHPUnit core
class called TestCase
use TDD\Receipt; // imports Receipt class
// Create Test Class

/*
* @covers TDD\Receipt
*/
class ReceiptTest extends TestCase {

  public function setUp() {
      $this->Receipt = new Receipt();// create new instance
of the Receipt class
  }
  public function tearDown() {
      unset($this->Receipt); // unset the instance to ensure
that PHP doesn't carry any over from one test to the next
  }

  public function testTotal() {

      $input = [0, 2, 5, 8]; // put the inputs in its own
array
      $output = $this->Receipt->total($input); // store the
sum of the array within a $output variable
      $this->assertEquals( // Built-in assertEquals() method
for defining the test. It accepts three parameters
              14, // 1st param is expected value
              $output, // 2nd param is the $output variable
              'When summing the total should equal 15' //
3rd param is a message displayed in case of failure
              );

  }

}
```

The benefit: the refactoring of the code does not change the functionality of the test, instead it improves it: our test is now more isolated and ensures the instance of Receipt object will be removed properly after our tests.
Save the file and run the test again, to ensure that everything still work as expected.

A quote from Wikipedia:
Software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

In testing theory, this software design pattern is called **Arrange-Act-Assert Pattern**. Arrange-Act-Assert pattern is commonly used when creating automated tests. In short:

- **Arrange** - all necessary preconditions and inputs for our test case
- **Act** - act on the object or method we are testing
- **Assertion** - expected results have occurred

This basic pattern of Arrange-Act-Assert is at the heart of all tests we will write.
General principles that you should try to achieve in your unit test:

- Your test should happen in isolation (of other code and environment) as much as possible
- Focus on a single method that does only one thing
- Tests should be easy to write, a hard test generally means re-writing your implementation

Add more tests: Improved TDD Example

Let’s create a new scenario of using Unit Testing. Make sure that you are developing the following example in your set-up environment from the previous section.

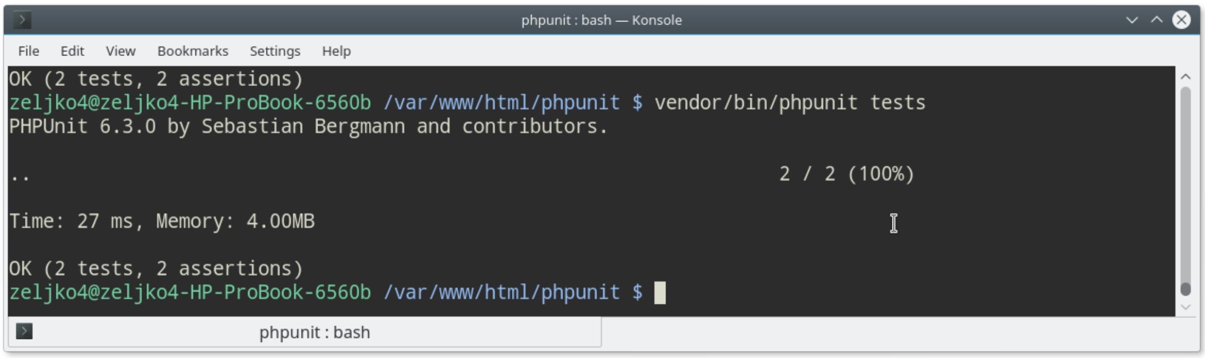
First let’s define our tax method inside our developing application (**Receipt.php**):

```php
<?php
namespace TDD;

class Receipt {
   public function total(array $items = []) {
  // takes array of items, by default an empty array
    return array_sum($items);
  // array_sum is built in function which sum up
  // an array passed as an argument
  }

   public function tax($amount, $tax) {
          return ($amount * $tax);
  }

}
```

Now let’s create the test method called **testTax()** which will test, whether the **tax()** method in the Application works as expected **(ReceiptTest.php)**:

```
<?php
namespace TDD\Test;

  use PHPUnit\Framework\TestCase; // imports PHPUnit core
class called TestCase
  use TDD\Receipt; // imports Receipt class
  /*
   * @covers TDD\Receipt
   */
  class ReceiptTest extends TestCase {

      public function setUp() {
          $this->Receipt = new Receipt();// create new
instance of the Receipt class
      }
      public function tearDown() {
          unset($this->Receipt); // unset the instance to
ensure that PHP doesn't carry any over from one test to the
next
      }

      public function testTotal() {

          $input = [0, 2, 5, 8]; // put the inputs in its
own array
          $output = $this->Receipt->total($input); // store
the sum of the array within a $output variable
          $this->assertEquals( // Built-in assertEquals()
method for defining the test. It accepts three parameters
                  15, // 1st param is expected value
                  $output, // 2nd param is the $output
variable
                  'When summing the total should equal
15' // 3rd param is a message displayed in case of failure
                  );

      }

       public function testTax() {
              $inputAmount = 5;
              $inputTax = 0.25;
              $output = $this->Receipt->tax($inputAmount,
$inputTax);
              $this->assertEquals(1.25, $output,
                  'Wrong tax calculation, expected: 1.25'
                  );
      }

}
```

Let’s run our test. We should get a message that the test is passed(OK):



```
OK (2 tests, 2 assertions)
zeljko4@zeljko4-HP-ProBook-6560b /var/www/html/phpunit $ vendor/bin/phpunit tests
PHPUnit 6.3.0 by Sebastian Bergmann and contributors.

..                                                                2 / 2 (100%)

Time: 27 ms, Memory: 4.00MB

OK (2 tests, 2 assertions)
zeljko4@zeljko4-HP-ProBook-6560b /var/www/html/phpunit $
```

Tests organization & *Test.php files

Probably the easiest way to organize your tests is to keep all test case source files in a **tests** directory. PHPUnit can automatically discover and run the tests by recursively traversing the test directory.
In the figure below we see that the test case classes in the tests directory mirror the package and class structure of the System Under Test in the src directory:

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

To run all tests for the library we just need to point the PHPUnit command-line test runner to the **tests** directory. After that, PHPUnit will look automatically for all ***Test.php** files.

Filter Tests

Our testing went fine so far and our code is not breaking. This isn’t too bad, since we have only 2 tests so far. But imagine what happens when we have an entire application with hundreds of files and corresponding tests.
One of the goals of PHPUnit is that tests should be composable: we want to be able to run any number or combination of tests together, for instance all tests for the whole project or the tests for all classes of a component that is part of the project or just the tests for a single class.
Luckily, the developers of PHP Unit offer us some powerful **filtering capabilities** for our tests. The simplest one we’ve already used, filtering by directory or by file. That is the test parameter we keep passing it in our PHPUnit command line interface, like this:

```
vendor\bin\phpunit tests  // Windows
vendor/bin/phpunit tests  // MacOS or Linux
```

Filter by file

Filter down to just a single file

```
vendor\bin\phpunit tests\ReceiptTest.php   //Windows
vendor/bin/phpunit tests/ReceiptTest.php   // MacOS/Linux
```

Filter by String

```
vendor\bin\phpunit tests --filter=testTax  //Windows
vendor/bin/phpunit tests --filter=testTax // MacOS/Linux
```

The string after filter will be a regular expression match against classes, methods and namespaces. In this case we pass the method name **testTax**. When we execute this, we can see that we have only one test executed, our **testTax()** method.

We can do something similar for filtering by Class and method name, like this:

```
vendor\bin\phpunit tests --filter=ReceiptTest::testTax //or
vendor/bin/phpunit tests --filter=ReceiptTest::testTax
```

Testing using XML document

A drawback of the previous approach is that we have no control over the order in which the tests are run. This can lead to problems in regard to test dependencies. In this section you will see how you can make the test execution order explicit by using the XML configuration file.
PHPUnit.xml file provides test groups to test specific groups of test it also allows us to no longer specify the directory to search for the test. Instead XML accesses the basic configuration file for us.
Within the phpunit folder create a new file called **phpunit.xml** and paste the following XML code inside of it:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit
  colors="true"
  processIsolation="false"
  stopOnFailure="false"
  syntaxCheck="false"
  >
  <php>
      <ini name="memory_limit" value="-1"/>
      <ini name="apc.enable_cli" value="1"/>
  </php>
  <!-- Add any additional test suites you want to run here -
->
  <testsuites>
      <testsuite name="app">
          <directory>./tests</directory>
      </testsuite>
      <testsuite name="receipt">
          <directory>./tests</directory>
          <exclude>./tests/ReceiptItems.php</exclude>
      </testsuite>
  </testsuites>
  <filter>
      <!--
      <blacklist>
          <directory
suffix=".php">/path/to/files</directory>
          <file>/path/to/file</file>
          <exclude>
              <directory
suffix=".php">/path/to/files</directory>
              <file>/path/to/file</file>
          </exclude>
      </blacklist>
      -->
      <whitelist processUncoveredFilesFromWhitelist="true">
          <directory suffix=".php">./src</directory>
          <!--
          <file>/path/to/file</file>
          <exclude>
              <directory
suffix=".php">/path/to/files</directory>
              <file>/path/to/file</file>
          </exclude>
          -->
      </whitelist>
  </filter>
  <!--
  <logging>
      <log
          type="coverage-html"
          target="./tmp/coverage/html/"
```

```
            charset="UTF-8"
            highlight="true"
            lowUpperBound="60"
            highLowerBound="90"
        />
        <log
            type="coverage-clover"
            target="./tmp/coverage/clover.xml"
        />
    </logging>
    -->
</phpunit>
```

Open your Terminal/CMD and run the following command:

```
vendor\bin\phpunit
```

(note: slashes instead backslashes for MacOS / Linux / GitHub console.)

You will notice that you don't have to define the directory of our tests, in order for it to be executed. It also provides us with colorful output (not all Terminal have this feature), that makes easier to see when the test fails (red color) or passes (green color).
At the top of the XML file we've defined some settings such as terminal colors. On line 14 we defined <testsuites> tag, by providing a name and identifying the directory to which they belong.
We have created two testsuites. The first one with the name "app" runs all of the files in our directory tests. The second one called "receipt" runs all the files in our directory excluding the file ReceiptItems.php (currently not created).

Let's run the first testsuite:

```
vendor/bin/phpunit --testsuite=app
```

We can even pass a filter like this:

```
vendor/bin/phpunit --testsuite=app --filter=textTax
```

Note: **phpunit.xml** can have much simpler structure. In another example, we can explicitly setup the test execution order:

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

More information about PHPUnit configuration file can be found here:
https://phpunit.de/manual/current/en/appendixes.configuration.html

Last modified: Thursday, 21 June 2018, 10:01 AM