

Desarrollo Web en Entorno Servidor

5.- PHP - Librerías

IES Severo Ochoa



Índice

- Composer
 - `composer.json`
- Monolog
 - Handlers, Processors & Formatters
- PhpDocumentor / Doxygen
- PHPUnit
 - TDD
 - Aserciones
 - Proveedores de datos

Composer

- Herramienta de gestión de librerías / dependencias
 - Todo el equipo de desarrollo tiene el mismo entorno y versiones
- <https://getcomposer.org/>
 - Similar a Maven (Java) / npm (JS)
 - Escrito en PHP
 - Utiliza Packagist como repositorio



Instalación

- Instalación

<https://getcomposer.org/download/>

- Si lo has instalado mediante apt:

```
> php -r  
"copy('https://getcomposer.org/installer',  
'composer-setup.php');"  
  
> sudo php composer-setup.php --install-dir  
/usr/bin --filename composer
```

- Comprobaremos que tenemos la v2:ç

```
> composer -V
```

- Si tienes errores al ejecutar *Composer*, puede ser que necesites instalar:

```
> sudo apt install php-xml  
> sudo apt install php-mbstring
```

Primeros pasos

- Crear un repositorio

- > `composer init`

- Configuramos el nombre del paquete, descripción, autor (`nombre <email>`), tipo de paquete (`project`), etc...
 - Definimos las dependencias del proyecto (`require`) y las de desarrollo (`require-dev`) de manera interactiva.
 - En las de desarrollo se indica aquellas que no se instalarán en el entorno de producción, por ejemplo, las librerías de pruebas.
 - Tras su configuración, se crea el archivo `composer.json` y descarga las librerías en la carpeta `vendor`.

composer.json

```
{
  "name": "dwes/log",
  "description": "Pruebas con Monolog",
  "type": "project",
  "require": {
    "monolog/monolog": "^2.1"
  },
  "license": "MIT",
  "authors": [
    {
      "name": "Aitor Medrano GVA",
      "email": "medrano_ait@gva.es"
    }
  ]
}
```

Packagist

- Todos los paquetes se descargan de <https://packagist.org/>
- Podemos indicar la versión:
 - Directamente: 1.4.2
 - Con comodines: 1.*
 - A partir de: $\geq 2.0.3$
 - Sin rotura de cambios: $\wedge 1.3.2 // \geq 1.3.2 < 2.0.0$

Actualizar librerías

- Podemos definir las dependencias via el archivo `composer.json` o mediante comandos
 - > `composer require vendor/package:version`
 - Ejemplo:
 - > `composer require phpunit/phpunit --dev`
- Tras añadirlas, hemos de actualizar nuestro proyecto
 - > `composer update`
- Si creamos el archivo `composer.json` nosotros directamente, hemos de instalar las dependencias
 - > `composer install`

autoload.php

- Composer crea en `vendor/autoload.php` un archivo que hemos de incluir en todos nuestros archivos
 - En nuestro caso, solo en los archivos donde probamos las clases

Autoload de clases de aplicación

- Si queremos que composer también se encargue de cargar nuestras clases, hemos de definirlo así:

```
"autoload": {  
    "psr-4": {"Dwes\\": "app/Dwes"}  
},
```

- Posteriormente, hemos de volver a generar el autoload de *Composer*:
 - > composer dump-autoload
 - 0 > composer du

Monolog

- <https://github.com/Seldaek/monolog>
 - Librería de log en PHP que soporta diferentes niveles y salidas
 - ficheros, sockets, BBDD, Web Services, email, etc.
 - texto plano, HTML, JSON, etc.
- > composer require monolog/monolog**
- Monolog 2 requiere PHP 7.2
 - Cumple PSR-3
 - Empleada por *Laravel* y *Symfony*

Cuándo usar un log

- Seguir las acciones/movimientos de los usuarios
- Registrar las transacciones
- Rastrear los errores de usuario
- Fallos/avisos a nivel de sistema
- Interpretar y coleccionar datos para posterior investigación de patrones

Niveles de log PSR-3

- **debug -100**: Información detallada con propósitos de debug. No usar en entornos de producción.
- **info - 200**: Eventos interesantes como el inicio de sesión de usuarios.
- **notice - 250**: Eventos normales pero significativos.
- **warning - 300**: Ocurrencias excepcionales que no llegan a ser error.
- **error - 400**: Errores de ejecución que permiten continuar con la ejecución de la aplicación pero que deben ser monitorizados.
- **critical - 500**: Situaciones importantes donde se generan excepciones no esperadas o no hay disponible un componente.
- **alert - 550**: Se deben tomar medidas inmediatamente.
 - Caída completa de la web, base de datos no disponible, etc...
Además, se suelen enviar mensajes por email.
- **emergency - 600**: Es el error más grave e indica que todo el sistema está inutilizable.

Mi primer log con Monolog

```
<?php
include __DIR__ . "/vendor/autoload.php";

use Monolog\Logger;
use Monolog\Handler\StreamHandler;

$log = new Logger("MiLogger");
$log->pushHandler(new StreamHandler("logs/milog.log", Logger::DEBUG));

$log->debug("Esto es un mensaje de DEBUG");
$log->info("Esto es un mensaje de INFO");
$log->warning("Esto es un mensaje de WARNING");
$log->error("Esto es un mensaje de ERROR");
$log->critical("Esto es un mensaje de CRITICAL");
$log->alert("Esto es un mensaje de ALERT");
```

Añadir información

- En todos los métodos de registro de mensajes, además del propio mensaje, le podemos pasar información como el contenido de alguna variable, usuario de la aplicación, etc..
 - Adjuntar los datos dentro de un array → *array de contexto*
 - Si queremos lo podemos hacer asociativo para facilitar la lectura del log

```
$log->warning("Producto no encontrado", [$producto]);  
$log->warning("Producto no encontrado", ["datos" =>  
$producto]);
```

Funcionamiento

- Cada instancia `Logger` tiene un nombre de canal y una pila de manejadores (*handler*).
- Cada mensaje que mandamos al log atraviesa la pila de manejadores, y cada uno decide si debe registrar la información, y si se da el caso, finalizar la propagación.
 - Por ejemplo, un `StreamHandler` en el fondo de la pila que lo escriba todo en disco, y en el tope añade un `MailHandler` que envíe un mail sólo cuando haya un error.
- Cada manejador también tiene un formateador (`Formatter`)
 - Si no se indica ninguno, se le asigna uno por defecto.

Handlers

- Cada instancia de `Logger` tendrá una pila de manejadores
- El último insertado será el primero en ejecutarse.
- Luego se “van ejecutando” conforme a la pila.
- <https://github.com/Seldaek/monolog/blob/master/doc/02-handlers-formatters-processors.md>
 - `StreamHandler(ruta, nivel)`
 - `RotatingFileHandler(ruta, maxFiles, nivel)`
 - `NativeMailerHandler(para, asunto, desde, nivel)`
 - `FirePHPHandler(nivel)`

stderr

- Si queremos que los mensajes de la aplicación salgan por el log del servidor
 - en nuestro caso el archivo `error.log` de *Apache*:

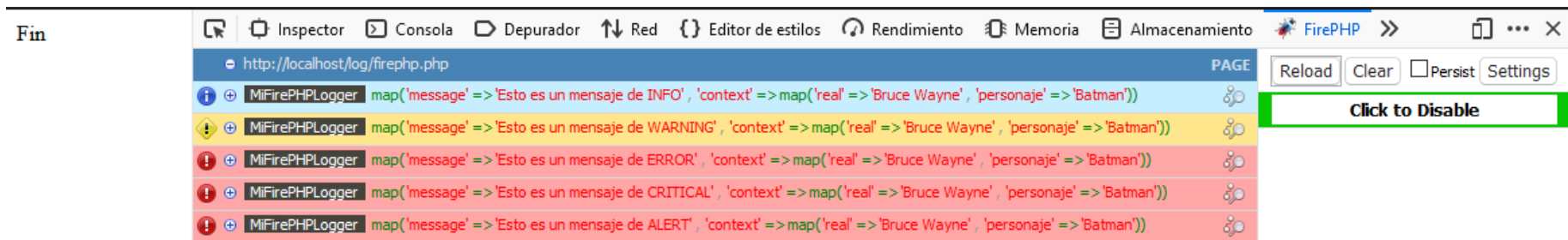
```
// error.log  
$log->pushHandler(new StreamHandler("php://stderr", Logger::DEBUG));
```

FirePHPHandler

- *FirePHP* → Herramienta para hacer debug en la consola de *Firefox*
- Tras instalar la extensión en *Firefox*, habilitar las opciones y configurar el *Handler*, podemos ver los mensajes coloreados con sus datos

```
$log = new Logger("MiFirePHPLogger");
$log->pushHandler(new FirePHPHandler(Logger::INFO));

$datos = ["real" => "Bruce Wayne", "personaje" => "Batman"];
$log->debug("Esto es un mensaje de DEBUG", $datos);
$log->info("Esto es un mensaje de INFO", $datos);
$log->warning("Esto es un mensaje de WARNING", $datos);
// ...
```



The screenshot shows the Firefox browser interface with the FirePHP extension active. The top toolbar includes icons for Inspector, Consola, Depurador, Red, Editor de estilos, Rendimiento, Memoria, Almacenamiento, and FirePHP. The FirePHP extension is currently expanded, showing a list of log messages from 'MiFirePHPLogger' at the URL 'http://localhost/log/firephp.php'. The messages are color-coded by severity: INFO (blue), WARNING (yellow), ERROR (red), CRITICAL (dark red), and ALERT (dark red). Each message includes a 'message' and a 'context' array. The context array for all messages is: `map('real' => 'Bruce Wayne', 'personaje' => 'Batman')`. The messages are:

- INFO: `map('message' => 'Esto es un mensaje de INFO', 'context' => map('real' => 'Bruce Wayne', 'personaje' => 'Batman'))`
- WARNING: `map('message' => 'Esto es un mensaje de WARNING', 'context' => map('real' => 'Bruce Wayne', 'personaje' => 'Batman'))`
- ERROR: `map('message' => 'Esto es un mensaje de ERROR', 'context' => map('real' => 'Bruce Wayne', 'personaje' => 'Batman'))`
- CRITICAL: `map('message' => 'Esto es un mensaje de CRITICAL', 'context' => map('real' => 'Bruce Wayne', 'personaje' => 'Batman'))`
- ALERT: `map('message' => 'Esto es un mensaje de ALERT', 'context' => map('real' => 'Bruce Wayne', 'personaje' => 'Batman'))`

On the right side of the FirePHP panel, there are buttons for 'Reload', 'Clear', 'Persist' (unchecked), and 'Settings'. Below these buttons is a green bar with the text 'Click to Disable'.

Canales

- Se les asigna al crear el `Logger`
- En grandes aplicaciones, se crea un canal por cada subsistema: Ventas, Contabilidad, Almacén
 - No es una buena práctica usar el nombre de la clase como canal
 - Esto se hace con un Procesador

Uso dentro de clases

- Asignar una propiedad privada a `Logger`
- En el constructor de la clase, asignar el canal, manejadores y formato.

```
$this->log = new Logger("App");  
$this->log->pushHandler(new StreamHandler("log/ milog.log",  
                                          Logger::DEBUG));  
$this->log->pushHandler(new FirePHPHandler(Logger::DEBUG));
```

- Dentro de los métodos:

```
$this->log->warning("Producto no encontrado", [$producto]);
```

Procesadores

- Permiten añadir información a los mensajes.
- `pushProcessor($procesador)`

```
$log = new Logger("MiLogger");  
$log->pushHandler(new RotatingFileHandler("logs/milog.log", 0,  
                                           Logger::DEBUG));  
$log->pushProcessor(new IntrospectionProcessor());  
$log->pushHandler(new FirePHPHandler(Logger::WARNING));  
// no usa Introspection pq lo hemos apilado después, le asigno otro  
$log->pushProcessor(new WebProcessor());
```

```
[2020-11-26T13:35:31.076138+01:00] MiLogger.DEBUG: Esto es un mensaje de DEBUG []  
{"file":"C:\\xampp\\htdocs\\log\\  
procesador.php","line":12,"class":null,"function":null}  
[2020-11-26T13:35:31.078344+01:00] MiLogger.INFO: Esto es un mensaje de INFO []  
{"file":"C:\\xampp\\htdocs\\log\\  
procesador.php","line":13,"class":null,"function":null}
```



Formateadores

- Se asocian a los manejadores con `setFormatter`
- <https://github.com/Seldaek/monolog/blob/master/doc/02-handlers-formatters-processors.md>
 - `LineFormatter`
 - `HtmlFormatter`
 - `JsonFormatter`

```
$log = new Logger("MiLogger");  
$rfh = new RotatingFileHandler("logs/milog.log", Logger::DEBUG);  
$rfh->setFormatter(new JsonFormatter());  
$log->pushHandler($rfh);
```

```
{"message":"Esto es un mensaje de DEBUG","context":  
{}, "level":100, "level_name":"DEBUG", "channel":"MiLogger", "datetime":"2020-11-27T15:36:52.747211+01:00", "extra":{}}  
{"message":"Esto es un mensaje de INFO","context":  
{}, "level":200, "level_name":"INFO", "channel":"MiLogger", "datetime":"2020-11-27T15:36:52.747538+01:00", "extra":{}}
```

Factoría Monolog

- Para evitar crear un `Logger` en cada clase, centramos su creación en una única clase.

```
use Monolog\Logger;
use Monolog\Handler\StreamHandler;

class LogFactory {

    public static function getLogger(string $canal = "miApp") : Logger {
        $log = new Logger($canal);
        $log->pushHandler(new StreamHandler("logs/
miApp.log", Logger::DEBUG));

        return $log;
    }
}
```


Factoría PSR

- En vez de devolver un `Logger` de `Monolog`, es mejor devolver el interfaz `Psr\Log\LoggerInterface`

```
use Monolog\Handler\StreamHandler;
use Monolog\Logger;
use Psr\Log\LoggerInterface;

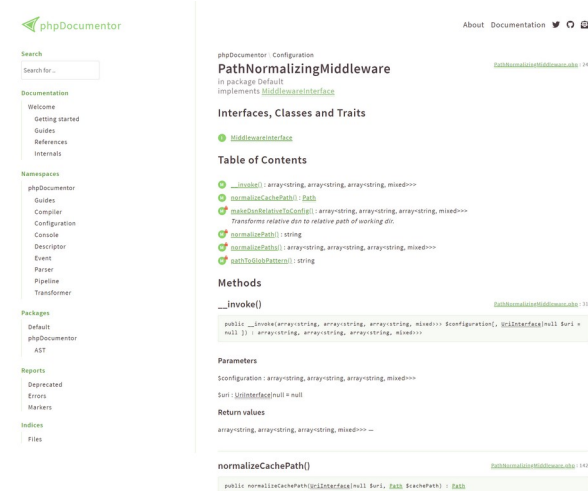
class LogFactory {

    public static function getLogger(string $canal = "miApp") : LoggerInterface {
        $log = new Logger($canal);
        $log->pushHandler(new StreamHandler("log/miApp.log", Logger::DEBUG));

        return $log;
    }
}
```

phpDocumentor

- <https://www.phpdoc.org/>
- Herramienta que facilita la documentación del código PHP
- Crea un sitio web con el API de la aplicación.
- Se basa en el uso de anotaciones sobre los *docblocks*.



Instalación y uso

- > sudo apt install plantuml
 - > wget <https://phpdoc.org/phpDocumentor.phar>
 - > chmod +x phpDocumentor.phar
 - > mv phpDocumentor.phar /usr/local/bin/**phpdoc**
 - > phpdoc --version
 - > phpdoc -d  ./app -t  docs/api
- Carpeta fuente Carpeta destino

DocBlock

```
<?php
/**
 * *Sumario*, una sola línea
 *
 * *Descripción* que puede utilizar varias líneas
 * y que ofrece detalles del elemento o referencias
 * para ampliar la información
 *
 * @param string $miArgumento con una *descripción* del arg
umento
 * que puede usar varias líneas.
 *
 * @return void
 */
function miFuncion($miArgumento)
{
}
```

<https://docs.phpdoc.org/3.0/guide/guides/docblocks.html#inside-docblocks>

Documentando el código

- En todos los elementos, además del sumario y/o descripción, pondremos:
 - En las clases:
 - `@author` nombre <email>
 - `@package` ruta del namespace
 - En las propiedades:
 - `@var` tipo descripción
 - En los métodos:
 - `@param` tipo \$nombre descripción
 - `@throws` `ClassNotFoundException` descripción
 - `@return` tipo descripción

Ejemplos I

```
/**
 * Clase que representa un cliente
 *
 * El cliente se encarga de almacenar los soporte que tiene alquilado,
 * de manera que podemos alquilar y devolver productos mediante las operaciones
 * homónimas.
 *
 * @package Dwes\Videoclub\Model
 * @author Aitor Medrano <medrano_ait@gva.es>
 */
class Cliente {

    public $nombre;
    private $numero;

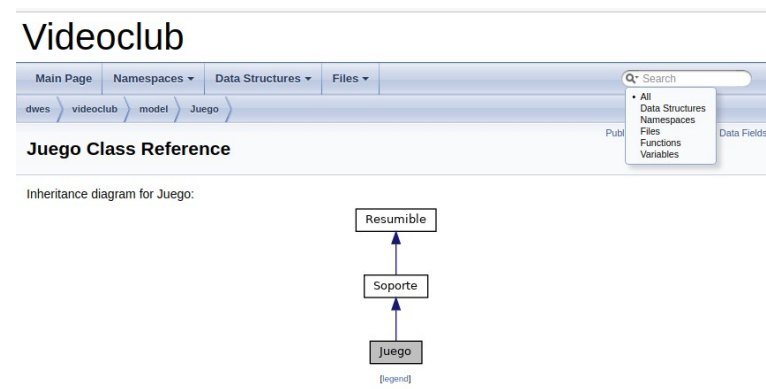
    /**
     * Colección de soportes alquilados
     * @var array<Soporte>
     */
    private $soportesAlquilados;
```

Ejemplos II

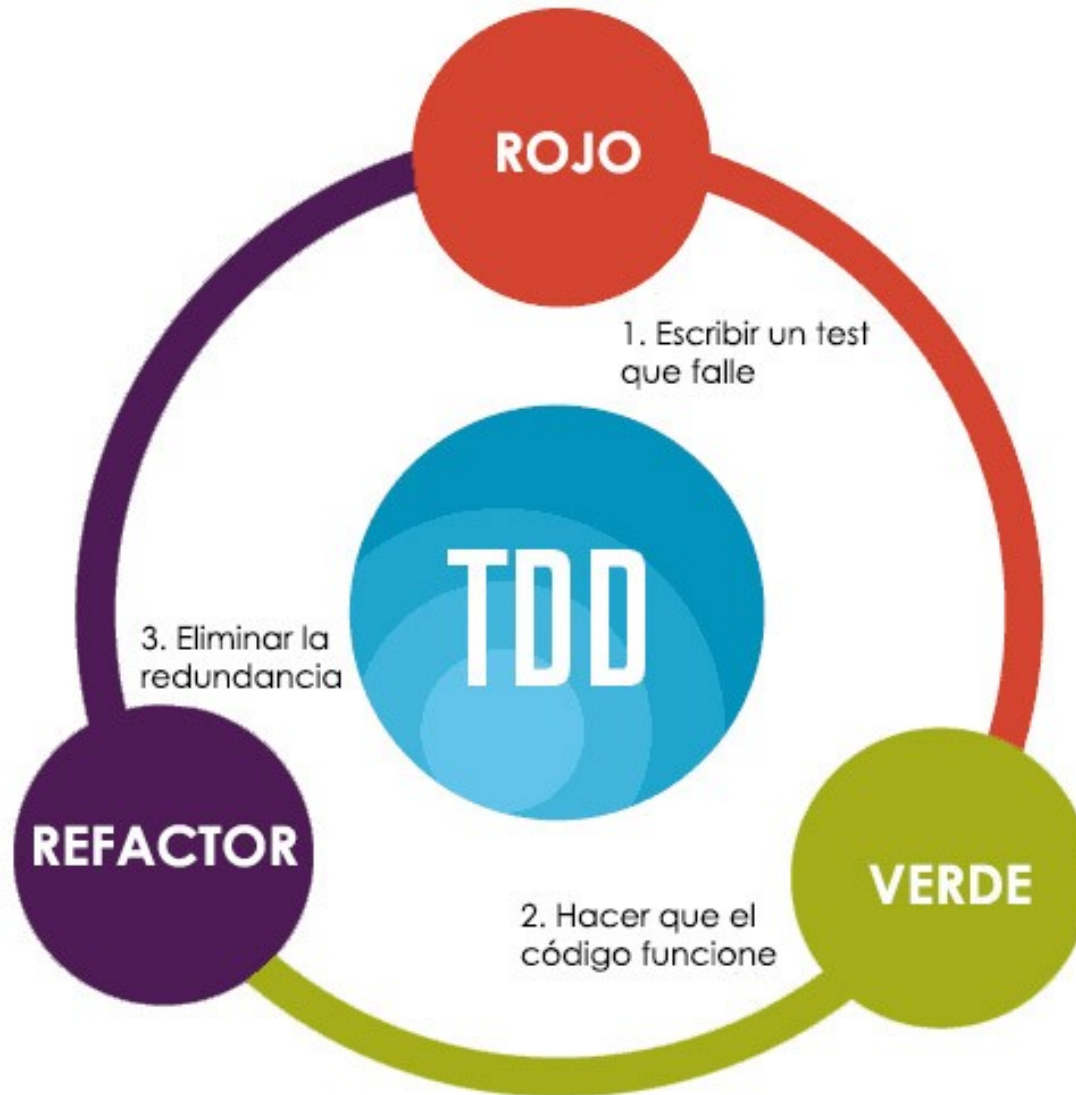
```
/**
 * Comprueba si el soporte recibido ya lo tiene alquilado el cliente
 * @param Soporte $soporte Soporte a comprobar
 * @return bool true si lo tiene alquilado
 */
public function tieneAlquilado(Soporte $soporte) : bool {
    // ...
}
```

Doxygen

- <https://www.doxygen.nl/index.html>
- Herramienta de documentación multilenguaje
- `> apt install doxygen`
- `> apt install doxygen-gui`
- `> doxywizard`



TDD



PHPUnit

- Herramienta de automatización de pruebas unitarias
- <https://phpunit.de/>
 - Basada en Junit
 - PHPUnit 9 requiere PHP 7.3
- <https://phpunit.readthedocs.io/es/latest/index.html>



PHPUnit

Hola Prueba

```
<?php
use PHPUnit\Framework\TestCase;

class PilaTest extends TestCase
{
    public function testPushAndPop()
    {
        $pila = [];
        $this->assertSame(0, count($pila));

        array_push($pila, 'batman');
        $this->
>assertSame('batman', $pila[count($pila)-1]);
        $this->assertSame(1, count($pila));

        $this->assertSame('batman', array_pop($pila));
        $this->assertSame(0, count($pila));
    }
}
```

tests/PilaTest.php

Ejecutando la prueba

- > `./vendor/bin/phpunit tests/PilaTest.php`
- > `./vendor/bin/phpunit tests`
- > `./vendor/bin/phpunit --testdox tests`
- > `./vendor/bin/phpunit --testdox --colors tests`

```
PS C:\Users\Aitor\Dropbox\2021\DWS\05 PHP Librerias\prueba> ./vendor/bin/phpunit tests
PHPUnit 9.4.3 by Sebastian Bergmann and contributors.

.                                                                1 / 1 (100%)

Time: 00:00.286, Memory: 4.00 MB

OK (1 test, 5 assertions)
PS C:\Users\Aitor\Dropbox\2021\DWS\05 PHP Librerias\prueba> ./vendor/bin/phpunit --testdox tests
PHPUnit 9.4.3 by Sebastian Bergmann and contributors.

Pila
 ✓ Push and pop

Time: 00:00.041, Memory: 6.00 MB

OK (1 test, 5 assertions)
```

PHPUnit y Composer

- Colocamos todas las pruebas en una carpeta `tests` en el raíz.
- En `composer.json`, añadiremos:

```
"require-dev": {  
    "phpunit/  
phpunit": "^7"  
},  
"scripts": {  
    "test": "phpunit --testdox --  
colors tests"  
}
```

- Ejecutamos las pruebas con:
- `> composer test`

Diseñando una prueba

- La clase debe heredar de **TestCase**
- El nombre de la clase debe acabar en **Test**
- Una prueba implica un método de prueba (público) por cada funcionalidad a probar
 - Normalmente se asocia a un método de clase
 - Debe nombrarse **testXXXXXXXX**
 - Es muy importante que el nombre sea muy claro (*camelCase*)
- Debemos preparar varias aserciones para toda la casuística: rangos de valores, tipos de datos, excepciones, etc...

Aserciones (*predicados*) I

- El predicado siempre es verdadero en ese lugar.
- `assertTrue` / `assertFalse` : Comprueba que la condición dada sea evaluada como `true` / `false`
- `assertEquals` / `assertSame`: Comprueba que dos variables sean iguales
- `assertNotEquals` / `assertNotSame`: Comprueba que dos variables NO sean iguales
 - `Same` → comprueba los tipos. Si no coinciden los tipos y los valores, la aserción fallará
 - `Equals` → sin comprobación estricta

Aserciones II

- `assertArrayHasKey` / `assertArrayNotHasKey`: Comprueba que un array posea un key determinado / o NO lo posea
- `assertArraySubset`: Comprueba que un array posea otro array como subset del mismo
- `assertAttributeContains` / `assertAttributeNotContains` : Comprueba que un atributo de una clase contenga una variable determinada / o NO contenga una variable determinada
- `assertAttributeEquals`: Comprueba que un atributo de una clase sea igual a una variable determinada.

Comprando la salida output

- `expectOutputString(salidaEsperada)`
- `expectOutputRegex(expresionRegularEsperada)`
- Primero se informa del resultado a esperar.
- Después se invoca al método que realiza el `echo/print`

Ejemplo aserciones

```
use PHPUnit\Framework\TestCase;

class CintaVideoTest extends TestCase {
    public function testConstructor()
    {
        $cinta = new CintaVideo("Los cazafantasmas", 23, 3.5, 107);

        $this->assertSame( $cinta->getNumero(), 23);
    }

    public function testMuestraResumen()
    {
        $cinta = new CintaVideo("Los cazafantasmas", 23, 3.5, 107);
        $resultado = "<br>Película en VHS:";
        $resultado .= "<br>Los cazafantasmas<br>3.5 (IVA no inclui
do) ";
        $resultado .= "<br>Duración: 107 minutos";

        $this->expectOutputString($resultado);
        $cinta->muestraResumen();
    }
}
```

Proveedores de datos

- Se declaran en el *docblock*
`@dataProvider nombreMetodo`
- Método público que devuelve un array de arrays, donde cada elemento es un caso de prueba
- La clase de prueba recibe como parámetros los datos a probar y el resultado de la prueba
 - En su *docblock* indica el nombre del *dataProvider*

Ejemplo Provider

```
/**
 * @dataProvider cintasProvider
 */
public function testMuestraResumenConProvider($titulo, $id, $precio,
$duracion, $esperado)
{
    $cinta = new CintaVideo($titulo, $id, $precio, $duracion);
    $this->expectOutputString($esperado);
    $cinta->muestraResumen();
}

public function cintasProvider() {
    return [
        "cazafantasmas" => ["Los cazafantasmas", 23, 3.5, 107, "<br>P
elícula en VHS:<br>Los cazafantasmas<br>3.5 (IVA no incluido)<br>Dura
ción: 107 minutos"],
        "superman" => ["Superman", 24, 3, 188, "<br>Película en VHS:<
br>Superman<br>3 (IVA no incluido)<br>Duración: 188 minutos"],
    ];
}
```

Probando excepciones

- Las pruebas han de cubrir todos los casos posibles.
- Poder hacer pruebas que esperen que se lance una excepción (y que el mensaje contenga cierta información).

```
expectException (Excepcion::class)
```

```
expectExceptionCode (codigoExcepcion)
```

```
expectExceptionMessage (mensaje)
```

- Primero se pone la expectativa, y luego se provoca que se lance la excepción

Ejemplo excepción

```
public function testAlquilarCupoLleno() {  
    $soporte1 = new CintaVideo("Los cazafantasmas", 23, 3.5, 107);  
    $soporte2 = new Juego("The Last of Us Part II", 26, 49.99, "PS4", 1  
, 1);  
    $soporte3 = new Dvd("Origen", 24, 15, "es,en,fr", "16:9");  
    $soporte4 = new Dvd("El Imperio Contraataca", 4, 3, "es,en", "16:9")  
;  
  
    $cliente1 = new Cliente("Bruce Wayne", 23);  
    $cliente1->alquilar($soporte1);  
    $cliente1->alquilar($soporte2);  
    $cliente1->alquilar($soporte3);  
  
    $this->expectException(CupoSuperadoException::class);  
    $cliente1->alquilar($soporte4);  
}
```

Cobertura de código













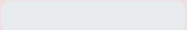
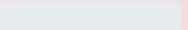
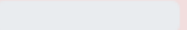







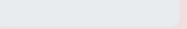




- Cantidad de código que las pruebas cubren.
- Recomendado entre el 95 y el 100%.
- CRAP: Análisis y Predicciones sobre el Riesgo en Cambios
 - cantidad de esfuerzo, dolor y tiempo requerido para mantener una porción de código
 - Debe mantenerse ≤ 5
- Añadimos en *Composer* un nuevo script:

```
"coverage": "phpunit --coverage-html coverage --coverage-filter  
app tests"
```

Informe de cobertura de pruebas

- Y posteriormente ejecutamos
> `composer coverage`

C:\xampp\htdocs\videoclub4\app\Dwes\Videoclub / Model / (Dashboard)

| | Code Coverage | | | | | | | | |
|--|---|---------|----------|--|---------|--------|---|---------|-------|
| | Lines | | | Functions and Methods | | | Classes and Traits | | |
| Total |  | 13.01% | 16 / 123 |  | 21.43% | 6 / 28 |  | 16.67% | 1 / 6 |
|  CintaVideo.php |  | 100.00% | 7 / 7 |  | 100.00% | 2 / 2 |  | 100.00% | 1 / 1 |
|  Cliente.php |  | 0.00% | 0 / 53 |  | 0.00% | 0 / 7 |  | 0.00% | 0 / 1 |
|  Dvd.php |  | 0.00% | 0 / 7 |  | 0.00% | 0 / 2 |  | 0.00% | 0 / 1 |
|  Juego.php |  | 0.00% | 0 / 14 |  | 0.00% | 0 / 3 |  | 0.00% | 0 / 1 |
|  Soporte.php |  | 90.00% | 9 / 10 |  | 80.00% | 4 / 5 |  | 0.00% | 0 / 1 |
|  Videoclub.php |  | 0.00% | 0 / 32 |  | 0.00% | 0 / 9 |  | 0.00% | 0 / 1 |

Es necesario tener instalado xdebug → `apt install php-xdebug`

Más PHPUnit

- Dependencia entre casos de prueba con el atributo `@depends`
- Completamente configurable mediante el archivo `phpunit.xml`:
 - <https://phpunit.readthedocs.io/es/latest/configuration.html>
- Preparando las pruebas con `setUpBeforeClass()` y `tearDownAfterClass()`
- Objetos y pruebas *Mock* (dobles) con `createMock()`

¿Alguna pregunta?