Fundamentos de la Programación Orientada a Objetos (POO) en PHP

Sitio:	IES Chan do Monte - Aula Virtual	Impreso por:	Manuel_Dario Gonzalez_Paton
Curso:	DAW Nocturno - Desenvolvemento de Aplicacions Web Contorna Servidor	Data:	mércores, 11 de febreiro de 2026, 4:55 PM
Libro:	Fundamentos de la Programación Orientada a Objetos (POO) en PHP		

Táboa de contidos

1. Fundamentos de la Programación orientada a objetos (POO)

La POO es una metodología de programación basada en clases y objetos.

- Una **clase** es una "plantilla" o una estructura común con información estructurada que une datos (atributos) y tipo de comportamiento (métodos).
- Un **objeto** es un ejemplar concreto de una clase (esa plantilla o estructura común) con unos datos concretos. Podéis consultar una imagen con esta distinción entre clase y objeto aquí.

- **Métodos**. Son los miembros de la clase que contienen el código de la misma. Un método es como una función. Puede recibir parámetros y devolver valores.
- **Atributos o propiedades**. Almacenan información acerca del estado del objeto al que pertenecen (y por tanto, su valor puede ser distinto para cada uno de los objetos de la misma clase)

Hasta ahora, hemos usado programación estructurada (salvo para acceder a bases de datos). En la programación estructurada los datos y las funciones no están directamente relacionados, simplemente se procesan datos de entrada para obtener otros de salida.
A diferencia de la programación estructurada, en la POO se aúnan datos y comportamiento en una misma estructura: el objeto. Para manipular los datos de un objeto (sus atributos), hay que hacer uso de sus métodos (o funciones).

Las **características** principales de la POO son:

- **Herencia**. Es el proceso de crear una clase a partir de otra, heredando su comportamiento, características y permitiendo a su vez redefinirlos y/o ampliarlos en las clases heredadas.
- **Abstracción**. Hace referencia a que cada clase oculta en su interior las peculiaridades de su implementación, y presenta al exterior una serie de métodos (interfaz pública) cuyo comportamiento está bien definido. Visto desde el exterior, cada objeto es un ente abstracto que realiza un trabajo, pero la implementación no tiene por qué ser conocida.
- **Polimorfismo**. Un mismo método puede tener comportamientos distintos en función del objeto con que se utilice. Esto es habitual en la herencia.
- **Encapsulación**. En la POO se juntan en un mismo lugar los datos y el código que los manipula.

2. Programación orientada a objetos (POO)en PHP

PHP nació como un lenguaje de script y originalmente carecía de las características de la POO.

A partir de la versión 4 y sobretodo 5, se implementaron características de soporte de la POO.

Vamos a seguir la documentación de PHP online comenzando por: https://www.php.net/manual/es/language.oop5.basic.php

3. Definición de una clase

- Comienza con la palabra reservada **class**, seguido de un nombre de clase, y continuando con un par de llaves que encierran las definiciones de las propiedades y métodos pertenecientes a dicha clase.
- Los nombres de clase no son case sensitive y siguen la convención **PascalCase.** Se recomienda que se utilice una sola clase por fichero y que el fichero tenga el mismo nombre de la clase.
- La pseudovariable **$this** está disponible cuando un método es invocado dentro del contexto de un objeto.
- Para acceder a las propiedades de un objeto o a sus métodos se utiliza la sintaxis flecha ->
- Consulta la URL de PHP.net para más información y ver ejemplos: https://www.php.net/manual/es/language.oop5.basic.php

```php
<?php
class ClaseSencilla
{
    // Declaración de una propiedad
    public $var = 'un valor predeterminado';

    // Declaración de un método
    public function mostrarVar() {
        echo $this->var;
    }
}
?>
```

4. Propiedades tipadas

Las variables pertenecientes a una clase se llaman *propiedades*. Son los atributos de una clase en POO.

A partir de PHP 7.4.0, las definiciones de propiedades pueden incluir declaraciones de tipo, originando **propiedades tipadas**:

```
class Shape
{
    public int $numberOfSides;
    public string $name;

}
```

Las propiedades tipadas **deben ser inicializadas antes de acceder a ellas**, de lo contrario se produce un Error.

```
$circle = new Shape();

$circle->getName(); //causará Error
```

A partir de PHP 8.1.0, una **propiedad tipada** se puede declarar con el modificador `readonly` (de solo lectura), lo que impide la modificación de la propiedad después de la inicialización

```
class Test1 {
    public readonly string $prop;
}
```

Fuente: https://www.php.net/manual/es/language.oop5.properties.php

5. Creación de objetos

- Para crear una instancia de una clase, se suele emplear la palabra reservada **new** y a continuación el nombre de la clase.
- Se usarán paréntesis obligatoriamente si hay que pasarle parámetros (al constructor) para incializar el objeto.

```php
<?php
$instancia = new ClaseSencilla();

// Esto también se puede hacer con una variable:
$nombreClase = 'ClaseSencilla';
$instancia = new $nombreClase(); // new ClaseSencilla()
?>
```

- Un objeto se creará siempre, a menos que el objeto tenga un constructor que lance una excepción.
- Las propiedades y métodos de una clase viven en «espacios de nombres» o *categorías* diferentes, por tanto, es posible tener una propiedad y un método con el mismo nombre.
- Se accede con la flecha, por ejemplo: con **$objeto->campo** y con **$objeto->método()**

6. Visibilidad

Se pueden añadir los modificadores *public*, *protected* o *private* a una propiedad, un método o una constante.

Consulta sus significados en la URL https://www.php.net/manual/es/language.oop5.visibility.php

- **public**: accesible desde cualquier parte (visibilidad por defecto)
- **protected**: accesible desde la misma clase, mediante clases heredadas o desde la clase padre
- **private**: accesible solo desde la la clase que los definió.

7. Constantes

Las constantes pueden ser declaradas en una clase con diferentes tipos de visibilidad

- Si se hace referencia a las constantes dentro de la propia clase se utiliza la palabra clave **self** seguida del operador resolución de ámbito ::
- Si se hace referencia a las constantes fuera de la propia clase se utiliza **el nombre de la clase** seguida del operador resolución de ámbito ::

```php
<?php
/**
 * Definir MiClase
 */
class MiClase
{
    // Declarar una constante pública
    public const MY_PUBLIC = 'public';

    // Declarar una constante protegida
    protected const MY_PROTECTED = 'protected';

    // Declarar una constante privada
    private const MY_PRIVATE = 'private';

    public function foo()
    {
        echo self::MY_PUBLIC;
        echo self::MY_PROTECTED;
        echo self::MY_PRIVATE;
    }
}

$myclass = new MiClase();
MiClase::MY_PUBLIC; // Funciona
MiClase::MY_PROTECTED; // Error fatal
MiClase::MY_PRIVATE; // Error fatal
$myclass->foo(); // Funcionan Public, Protected y Private
```

8. Propiedades y métodos estáticos

- Se puede aplicar a propiedades o métodos de clases
- Son métodos o propiedades compartidos por todos los objetos de la clase (no hay una propiedad/método por cada objeto)
- Son accesibles sin la necesidad de instanciar la clase. Desde dentro de la clase con self:: y desde fuera con el nombreClase::
- Una propiedad declarada como static no puede ser accedida con un objeto de clase instanciado (aunque un método estático sí lo puede hacer).
- Si no se usa ninguna declaración de visibilidad, se tratará a las propiedades o métodos como si hubiesen sido definidos como `public`.

```php
<?php
class Foo
{
    public static $mi_static = 'foo';

    public function valorStatic() {
        return self::$mi_static;
    }
}
```

Desde fuera de la clase Foo se llamaría a la función con el operador resolución de ámbito:
Foo::$mi_static

Ejemplo #1 Ejemplo de método estático

```php
<?php
class Foo {
    public static function unMetodoEstatico() {
        // ...
    }
}

Foo::unMetodoEstatico();
```

Más información sobre static: https://www.php.net/manual/es/language.oop5.static.php

9. Constructores

- Las clases que tengan un método constructor lo invocarán en cada nuevo objeto creado, por lo que se utilizará para inicializarlo antes de ser usado.
- Se declara con function __construct, y no se debe utilizar la manera antigua de creación de constructores con el nombre de la clase.
- Para ejecutar un constructor padre desde el constructor hijo se requiere invocar a **parent::__construct().**

Consulta la URL: https://www.php.net/manual/es/language.oop5.decon.php

Constructor Promotion

A partir de PHP 8.0.0, los parámetros del constructor pueden ser *ascendidos* (*promoted* en inglés) a propiedades de la clase. Se realiza añadiendo el modificador de visibilidad antes del parámetro. El constructor admite parámetros con o sin visibilidad. Los parámetros con visibilidad serán también propiedades y los parámetros sin visibilidad serán parámetros al uso.
Serían equivalentes estas dos posibilidades:

```php
<?php
class Point {
    protected int $x;
    protected int $y;

    public function __construct(int $x, int $y = 0) {
        $this->x = $x;
        $this->y = $y;
    }
}
```

```php
<?php
class Point {
    public function __construct(protected int $x, protected int $y = 0) {
    }
}
```

10. Herencia

- Una clase puede heredar los métodos y propiedades de otra clase empleando la palabra reservada **extends** en la declaración de la clase.
- **No es posible la extensión de múltiples clases**, una clase solo puede heredar de una clase base.
- Los métodos y propiedades heredados pueden ser **sobrescritos** con la redeclaración de éstos utilizando el **mismo nombre que en la clase madre**. Sin embargo, si la clase madre definió un método como **final**, éste **no podrá ser sobrescrito.**
- Es posible acceder a los métodos sobrescritos o a las propiedades estáticas haciendo referencia a ellos con la palabra **parent::**
- Más información: https://www.php.net/manual/es/language.oop5.inheritance.php
- Para ver un ejemplo de herencia, veamos el repositorio: https://github.com/dwcs-code-2425/r26_UD4_Ejemplo_Herencia.git

```
//Fichero Animal.php
<?php
class Animal {
    protected string $nombre; //estará disponible también para las clases hijas

    public function __construct(string $nombre) {
        $this->nombre = $nombre;
    }

    public function hablar(): string {
        return "El animal hace un sonido.";
    }

    public function getNombre(): string {
        return $this->nombre;
    }
}
?>
```

```
//Fichero Perro.php
<?php
class Perro extends Animal {

    public function hablar(): string {
        return "El perro ladra.";
    }
}
?>
```

```
//Fichero Gato.php
<?php
class Gato extends Animal {

    public function hablar(): string {
        //hace uso del método hablar() de la clase madre y al resultado, le concatena un texto propio
        return parent::hablar()
            . " En concreto, el gato maúlla.";
    }
}
?>
```

¿Cómo usar estas clases?

Para que PHP reconozca las clases, tiene que saber en qué ficheros se encuentran. Se indica con las instrucciones include* o require*:

```php
//Fichero index.php
<?php
// Para que PHP reconozca las clases, es necesario incluir los archivos donde se definen.
// Si no, se produce Fatal error: Uncaught Error: Class "Perro" not found in line x

require_once "Animal.php";
require_once "Perro.php";
require_once "Gato.php";

$animal = new Animal("Elefante");
$perro = new Perro("Rex");
$gato = new Gato("Misu");

echo "<h1> Ejemplo de Herencia en PHP </h1>";
echo $animal->getNombre() . ": " . $animal->hablar();
echo "<br>";
echo $perro->getNombre() . ": " . $perro->hablar();
echo "<br>";
echo $gato->getNombre() . ": " . $gato->hablar();
?>
```

El resultado será:

Ejemplo de Herencia en PHP

Elefante: El animal hace un sonido.
Rex: El perro ladra.
Misu: El animal hace un sonido. En concreto, el gato maúlla.

11. Colecciones de objetos y relaciones entre clases

Ahora que sabemos definir una clase en PHP y utilizar herencia, vamos a ver un ejemplo de cómo definir relación entre clases, que está disponible en el mismo repositorio https://github.com/dwcs-code-2425/r26_UD4_Ejemplo_Herencia.git:

Vamos a suponer que una animal puede tener un único dueño o ningún dueño. De cada dueño necesitaremos saber algunos datos como teléfono o nombre.

Para ello crearemos la clase Persona:

```php
<?php
class Persona {
    private string $nombre;
    private string $telefono;

    public function __construct(string $nombre, string $telefono) {
        $this->nombre = $nombre;
        $this->telefono = $telefono;
    }

    public function getNombre(): string {
        return $this->nombre;
    }

    public function getTelefono(): string {
        return $this->telefono;
    }

    public function verInformacion(): string {
        return $this->nombre . " (" . $this->telefono . ")";
    }
}
?>
```

Modificaremos la clase **Animal** para que tenga un nuevo atributo $dueno de tipo Persona nullable:

```php
<?php
class Animal {
    protected string $nombre;
    protected ?Persona $dueno = null;

    public function __construct(string $nombre) {
        $this->$nombre = $nombre;
    }

    public function setDueno(Persona $dueno): void {
        $this->$dueno = $dueno;
    }

    public function getDueno(): ?Persona {
        return $this->dueno;
    }

    public function hablar(): string {
        return "El animal hace un sonido.";
    }

    public function getNombre(): string {
        return $this->nombre;
    }

    public function verInformacion(): string {
        $info = $this->nombre;
        if ($this->dueno !== null) {
            $info .= " (Dueño: " . $this->dueno->verInformacion() . ")";
        }
        else {
             $info .= " (Sin dueño)";
        }
        return $info;
    }
}
?>
```

Colecciones de objetos

En las relaciones 1:N o M:N entre clases, en PHP se crearán atributos de tipo array.

Por ejemplo, en el caso de una clínica veterinaria, tendrá una serie de pacientes de tipo Animal (1:N). La colección de pacientes es un array. En PHP no podemos indicar el tipo del array, pero cuando insertemos objetos en el array serán de tipo Animal o de sus subclases.

```php
<?php
class Clinica {

    private string $nombre;
    private array $animales = []; // array de objetos Animal

    public function __construct(string $nombre) {
        $this->nombre = $nombre;
    }

    // Añadir un animal
    public function agregarAnimal(Animal $animal): void {
        $this->animales[] = $animal;
    }

    // Mostrar información de todos los animales
    public function listarAnimales(): void {
        echo "<h3>Animales en {$this->nombre}</h3>";
        foreach ($this->animales as $animal) {
            echo $animal->verInformacion() . "<br>";
        }
    }

    // Contar cuántos animales hay de cada tipo
    public function contarPorTipo(): void {
        $conteo = [];
        foreach ($this->animales as $animal) {
            // get_class devuelve el nombre de la clase del objeto con el que se hizo new
            $tipo = get_class($animal);
            if (!isset($conteo[$tipo])) {
                $conteo[$tipo] = 0;
            }
            $conteo[$tipo]++;
        }

        echo "<h4>Conteo por tipo:</h4>";
        foreach ($conteo as $tipo => $cantidad) {
            echo $tipo . ": " . $cantidad . "<br>";
        }
    }
}
?>
```

12. Comparación de objetos en PHP

- `==` **compara objetos por valor**: dos objetos son iguales si son de la **misma clase** y **tienen las mismas propiedades con los mismos valores**, aunque sean instancias distintas.
- `===` **compara por identidad**: solo es verdadero si ambas variables apuntan **al mismo objeto en memoria**.

```
$afro18 = new Baile("AFRO", 18);

$afro8 = new Baile("AFRO", 8);




var_dump($afro18 == $afro8);

echo "<br/>";

var_dump($afro18 === $afro8);

echo "<br/>";

$afro18_bis = $afro18;

var_dump($afro18 === $afro18_bis);

echo "<br/>";

$afro18_alias= &$afro18;

var_dump($afro18 === $afro18_alias);

echo "<br/>";
```

Devolverá:

```
bool(false)
bool(false)
bool(true)
bool(true)
```

`in_array()` por defecto compara con ==. Si quisiéramos que comparase con ===, habría que pasarle un tercer parámetro true.

```
in_array($baile, $this->bailes, true);
```

13. Clases abstractas

- Una **clase abstracta** representa una **idea general que sirve como base** para otras clases.
- **No se puede instanciar** directamente un objeto de esa clase: `new ClaseAbstracta(); //` ❌ . Sin embargo, una clase abstracta **puede tener constructor**, que, aunque no se puede llamar directamente, podrá será utilizado por las clases hijas que sí se instancien.
- Una clase **debe ser abstracta** si contiene **al menos un método abstracto**.
- Una clase abstracta de be utilizar la palabra clave ***abstract*** en su declaración
- Un **método abstracto**:
  - Utiliza la palabra ***abstract*** en su firma
  - Solo define la **firma** del método
  - **No tiene implementación**
- Cuando una clase hereda de una clase abstracta:
  - Está obligada a **implementar todos los métodos abstractos** heredados
  - Utiliza la palabra clave **extends** (como en la herencia al uso)
- Las **firmas de los métodos** deben coincidir:
  - Mismo nombre
  - Mismos tipos
  - Mismo número de argumentos obligatorios
  - Se permiten argumentos opcionales adicionales

Ejemplo

En una biblioteca existen distintos tipos de recursos, como libros y revistas.
Todos los recursos comparten información común, pero no tiene sentido crear un recurso genérico, **solo recursos concretos.**

Clase abstracta Recurso

```php
abstract class Recurso {
    protected string $titulo;

    public function __construct(string $titulo) {
        $this->titulo = $titulo;
    }

    abstract public function getTipo(): string;
}
```

Clase concreta: Libro

```php
class Libro extends Recurso {
    private string $isbn;

    public function __construct(
        string $titulo,
        string $isbn
    ) {
        parent::__construct($titulo);
        $this->isbn = $isbn;
    }

    public function getTipo(): string {
        return "Libro";
    }
}
```

Clase concreta: Revista

```php
class Revista extends Recurso {
    private int $numero;

    public function __construct(
        string $titulo,
        int $numero
    ) {
        parent::__construct($titulo);
        $this->numero = $numero;
    }

    public function getTipo(): string {
        return "Revista";
    }
}
```

Podemos ver ejemplos en la documentación: https://www.php.net/manual/es/language.oop5.abstract.php

14. Interfaces

Interfaces en PHP

Una **interfaz** define un **contrato** que una clase debe cumplir. Indica *qué métodos* debe implementar una clase, pero **no aporta la implementación concreta.**

- Las interfaces definen **qué métodos deben ser implementados**.
- No contienen código de los métodos, solo su firma.
- **Todos los métodos son públicos**.
- Se definen con la palabra clave `interface`.
- Las clases las implementan con `implements`.
- Una clase puede implementar **varias interfaces**.
- Las interfaces pueden extender otras interfaces.
- Pueden contener **constantes**.

Ejemplo: Exportación de recursos

En nuestro proyecto de **Recursos** (Libro, Revista, Vídeo), queremos que los objetos puedan **exportarse**, pero sin obligarlos a atarse a un formato concreto (JSON, texto, XML, etc.).

Para ello definimos una interfaz con un único método:

```
interface Exportable {
    public function exportar(): string;
}
```

La interfaz no dice *cómo* se exporta, solo que el objeto *puede exportarse*.

Implementación

Podeos obligar a que la clase abstracta `Recurso` implemente la interfaz `Exportable` y podemos darle una implementación por defecto coincidente con la cadena que devuelve el método `getDescription()`.

```
    abstract class Recurso implements Exportable{

    // constructor, getters y setters




    public function exportar(): string {
        return $this->getDescripcion();
    }
}
```

Cada hija podrá **sobrescribir** el método exportar() como mejor le convenga:

```
  class Video extends Recurso
{
    //constructor, getters y setters

    public function exportar(): string
    {
         return json_encode([
            "tipo" => $this->getTipo(),
            "titulo" => $this->titulo
        ]);
    }

}
```

Para probar el resultado, si llamamos al mismo método exportar, podremos tener 2 comportamientos diferentes debido al **polimorfismo**:

```
echo $video->exportar();
echo $revista->exportar();
  //El resultado será:

{"tipo":"Video","titulo":"Curso de PHP"}
Tipo: Revista, Título: Tech Monthly, Número: 42
```

Documentación oficial: Interfaces en PHP

15. Uso de interfaces mejorado

Patrón Strategy aplicado correctamente: exportación de recursos

En un diseño inicial, cada clase (*Libro, Revista, Video*) implementaba directamente un método `exportar()`. Aunque funcional, este enfoque presenta limitaciones importantes desde el punto de vista del diseño orientado a objetos.

Diseño inicial (limitaciones)

- Cada clase queda **acoplada** a un formato de exportación
- No es posible cambiar el formato sin modificar la clase
- Se mezcla lógica de dominio con lógica de exportación

Mejora con el Patrón Strategy

El Patrón Strategy propone separar el **objeto** del **algoritmo que realiza una acción**, delegando ese comportamiento en un objeto intercambiable.

Interfaz de la estrategia

```
interface Exportador {
    public function exportar(Recurso $recurso): string;
}
```

Clase Recurso (Contexto)

La clase `Recurso` mantiene una referencia a un objeto que implementa la interfaz `Exportador`, que puede cambiarse dinámicamente con un setter().

```
abstract class Recurso {

    protected string $titulo;
    protected Exportador $exportador;

    public function __construct(string $titulo) {
        $this->titulo = $titulo;
    }

    public function setExportador(Exportador $exportador): void {
        $this->exportador = $exportador;
    }

    public function exportar(): string {
        return $this->exportador->exportar($this);
    }

    abstract public function getTipo(): string;
    abstract public function getDescripcion(): string;
}
```

Estrategias concretas

```
class ExportadorTexto implements Exportador {
    public function exportar(Recurso $recurso): string {
        return $recurso->getDescripcion();
    }
}
```

```
class ExportadorJSON implements Exportador {
    public function exportar(Recurso $recurso): string {
        return json_encode([
            "tipo" => $recurso->getTipo(),
            "titulo" => $recurso->getTitulo()
        ]);
    }
}
```

Uso del patrón

```
$video = new Video("POO en PHP", 120);

$video->setExportador(new ExportadorTexto());
echo $video->exportar();

$video->setExportador(new ExportadorJSON());
echo $video->exportar();
```

¿Por qué este diseño es mejor?

Diseño inicial	**Patrón Strategy**
Formato fijo por clase	Formato intercambiable
Alto acoplamiento	Bajo acoplamiento
Difícil de extender a otros formatos	Abierto a extensión
Modificaciones invasivas en las clases del dominio (Recurso e hijas)	Cambios localizados (en las clases que implementan Exportable)

💡 Con Strategy, las clases de dominio permanecen estables mientras los comportamientos pueden variar libremente.

El diagrama del Patrón Strategy:

Fuente:

https://danielggarcia.wordpress.com/2014/05/12/patrones-de-comportamiento-iv-patron-strategy/

16. Clases abstractas vs Interfaces

Clases abstractas vs Interfaces en PHP

En Programación Orientada a Objetos, las **clases abstractas** y las **interfaces** se utilizan para definir comportamientos comunes, pero cumplen **roles distintos** en el diseño del software.

Clases abstractas

Una **clase abstracta** representa un concepto incompleto que sirve como base para otras clases. No se puede instanciar directamente.

- Pueden tener **atributos**
- Pueden tener **métodos implementados**
- Pueden tener **métodos abstractos**
- Una clase solo puede **heredar de una** clase abstracta

```
abstract class Recurso {

    protected string $titulo;

    public function __construct(string $titulo) {
        $this->titulo = $titulo;
    }

    public function getTitulo(): string {
        return $this->titulo;
    }

    abstract public function getTipo(): string;
    abstract public function getDescripcion(): string;
}
```

📌 Una clase abstracta define **qué es** un objeto y **qué comportamiento común tiene**.

Interfaces

Una **interfaz** define un **contrato** que una clase se compromete a cumplir. No representa un objeto, sino una *capacidad* o *comportamiento*.

- Solo contiene **métodos sin implementación**
- Todos los métodos son **públicos**
- No tiene atributos (podría tener constantes)
- Una clase puede **implementar varias interfaces**

```
interface Exportador {
    public function exportar(Recurso $recurso): string;
}
```

📌 Una interfaz define **qué se puede hacer**, no cómo se hace.

Diferencia conceptual clave

Clases abstractas	Interfaces
Representan un tipo de objeto	Representan un comportamiento
Pueden tener estado (atributos)	No tienen estado (aunque sí pueden tener constantes)
Herencia simple	Implementación múltiple
Relación “ES UN”	Relación “PUEDE HACER”

Ejemplo combinado

En nuestro proyecto:

- `Libro`, `Revista`, `Video` **son** Recursos
- Un objeto **usa** un `Exportador` para exportarse

> 💡 Cuando dudas entre interfaz o clase abstracta, pregúntate:
> **¿estoy definiendo qué es el objeto o qué puede hacer?**

17. Traits

Traits en PHP

Los **Traits** permiten reutilizar código entre varias clases sin herencia múltiple. A veces se dice que simulan esa herencia múltiple que en PHP no es posible (No es posible extender de 2 clases simultáneamente).

En este ejemplo, creamos un Trait **Logger** que registra acciones de los recursos en el log de PHP usando `error_log`.

- Se declaran como una **clase**, con la palabra `trait` en lugar de `class`
- No es posible instanciar un Trait en sí mismo.
- Los Traits pueden contener atributos de instancia y también estáticos, constantes, **métodos abstractos, estáticos y concretos**
- Una clase puede hacer uso de los atributos y métodos de un Trait a través de la palabra clave `use`.
- Una clase puede usar múltiples Traits, si los declara separados por comas: `use Trait1, Trait2;` .
- Evitan duplicación de código que puede ser reutilizado por otras clases
- Para llamar a sus métodos o acceder a sus atributos se usa la sintaxis `->` o `::`, según corresponda, como si fueran métodos o atributos de la propia clase.
- Es un añadido a la herencia tradicional, que permite el uso de métodos de clase sin necesidad de herencia.

Ejemplo: trait Logger

```
trait Logger {
    private string $nivelLog = 'INFO';

    public function log(string $mensaje, ?string $nivel = null): void {
        $nivel = $nivel ?? $this->nivelLog;
      //escribe en un fichero app.log en el directorio actual
        error_log("[$nivel] " . date('Y-m-d H:i:s') . " - " . $mensaje . "\n", 3, __DIR__ . '/app.log');
    }
}
```

> Este Trait permite que cualquier clase lo use para registrar mensajes sin repetir código. Permite usar cualquier nivel de log:

Nivel	Descripción
DEBUG	Información detallada para depuración, normalmente solo en desarrollo
INFO	Mensajes informativos sobre la ejecución normal
NOTICE	Algo inusual pero que no es un error
WARNING / WARN	Advertencias sobre situaciones que podrían causar problemas
ERROR	Errores que afectan la ejecución de una parte del programa
CRITICAL / FATAL	Fallos graves que pueden detener la aplicación
ALERT / EMERGENCY	Errores críticos que requieren atención inmediata

Uso en la clase Recurso (y en sus hijas)

```
abstract class Recurso {
    use Logger;

    protected string $titulo;

    public function __construct(string $titulo) {
        $this->titulo = $titulo;
        $this->log("Se ha creado el recurso: " . $titulo);
    }

    abstract public function getDescripcion(): string;
}

class Video extends Recurso {
    private int $duracion;

    public function __construct(string $titulo, int $duracion) {
        parent::__construct($titulo);
        $this->duracion = $duracion;
        $this->log("Video creado: {$titulo}, duración {$duracion} min");
    }

   //otros métodos
}

class Libro extends Recurso {
    private string $isbn;

    public function __construct(string $titulo, string $isbn) {
        parent::__construct($titulo);
        $this->isbn= $isbn;
        $this->log("Libro creado: {$titulo}, ISBN: {$isbn}");
    }


}
```

Uso práctico

```
$video = new Video("POO en PHP", 120);
$libro = new Libro("Aprendiendo PHP", "123456789...");

// Esto escribirá mensajes en el log app.log
```

Ventajas

- Permite compartir un **método concreto entre varias clases** (no solo dentro de una jerarquía de herencia).
- Evita duplicar código en cada subclase (o clase fuera de la jerarquía de herencia).
- Mantiene las clases centradas en su lógica de negocio

Desventajas

- **Confusión en la procedencia de métodos:** Si se usan muchos Traits, es difícil saber qué viene de dónde => Difícil de mantener
- **Conflictos de nombres:** Dos Traits pueden definir métodos iguales. Se resuelven con `insteadof` o `as`, pero complica el código. No lo veremos en el módulo, pero está presente en la documentación oficial:https://www.php.net/manual/es/language.oop5.traits.php

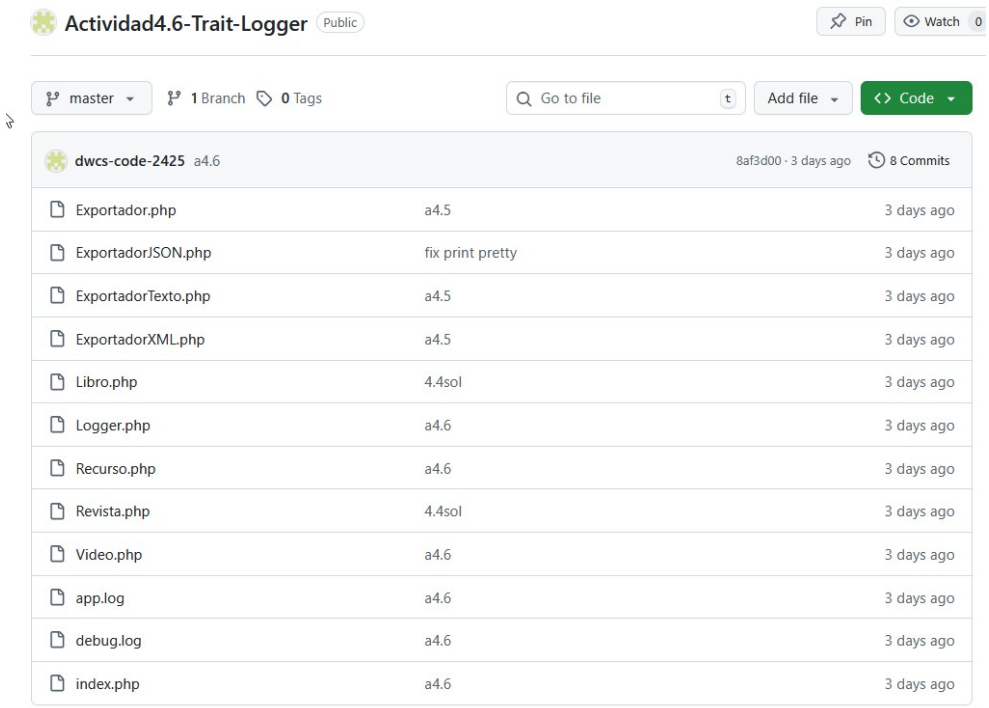
Se recomienda no abusar del uso de Traits. **Solo para comportamientos/atributos concretos y reutilizables**, que no dependan fuertemente del contexto de la clase.

> 💡 En este caso, usamos un Trait en lugar de una interfaz porque necesitamos **código concreto reutilizable**, no solo un contrato. Una interfaz define **qué se puede hacer**; un Trait define **cómo se hace**.

18. Namespaces

POO en PHP: Organizando nuestras clases con Namespaces

Hasta ahora hemos trabajado con una jerarquía de clases como **Recurso** (abstracta), de la que heredan **Libro**, **Revista** y **Vídeo**. También usamos la interfaz **Exportador** con estrategias (*Strategy Pattern*) como **ExportadorJSON**, **ExportadorXML** y **ExportadorTexto**, y un **trait Logger**.



Pero nuestras clases empiezan a crecer y nos encontramos con un problema: supongamos que **queremos crear otra clase llamada Recurso** (para representar hardware, por ejemplo, un array de recursos recomendados para reproducir un vídeo). ¿Cómo llamamos a ambas "Recurso" sin conflictos? Aquí nos pueden ayudar los **namespaces: tanto a organizar nuestro código conceptualmente, como a evitar conflictos de nombres.**

1. ¿Qué es un namespace?

Un **namespace** es como un *apellido* para las clases, interfaces, traits, funciones y constantes. Permite tener nombres duplicados en contextos diferentes sin conflictos porque el namespace se une al nombre de la clase para crear su **nombre completamente cualificado.**

2. Cómo declarar namespaces

```php
<?php
// Definimos un namespace para recursos bibliográficos
namespace App\Model\Biblioteca;

abstract class Recurso {
    use Logger;
    protected $titulo;
    protected Exportador $exportador;
    public function __construct($titulo) {
        $this->titulo = $titulo;
        $this->log("Se ha creado el recurso: " . $titulo, "DEBUG", "debug.log");

    }
}

// Definimos otra namespace para recursos de hardware

namespace App\Model\Infraestructura;
//Recurso hardware
class Recurso{
    private string $descripcion;

    public function __construct(string $descripcion) {
        $this->descripcion = $descripcion;
    }
}
```

Si intentamos crear un nuevo archivo **Recurso.php** dentro de la misma ubicación que la clase abstracta, veremos que el sistema de ficheros no nos lo permite. Esto nos sirve también para aplicar una convención: La jerarquía de niveles de namespaces se corresponde con una jerarquía de carpetas.

En nuestros ejemplos vamos a usar los siguientes **namespaces**:

- **App\Model\Biblioteca** → para la clase `Recurso` bibliográfica y sus subclases
- **App\Model\Infraestructura** → para la clase `Recurso` de hardware.
- **App\Service** → para la interfaz `Exportador` y sus implementaciones
- **App\Service\Traits**→ para el trait `Logger`

Correspondencia entre subniveles de namespace con jerarquía de carpetas

En PHP, los namespaces son **una forma lógica de organizar el código**, y no tienen que corresponder estrictamente con la estructura de carpetas. En nuestro caso:

- El prefijo `App` funciona como un *namespace raíz* que indica que todas las clases pertenecen a nuestra aplicación.
- No hace falta crear una carpeta `App` si no queremos; es más un concepto de organización que un requisito físico.
- Los subniveles como `Model\Biblioteca` o `Model\Infraestructura` sí suelen tener carpetas correspondientes (`src/Model/Biblioteca`, `src/Model/Infraestructura`) para mantener el código ordenado.
- Aprovechando la reorganización de código, **crearemos una carpeta log** para almacenar ahí los ficheros de log de la aplicación. Aquí **no se aplican los namespaces** porque no son clases ni código PHP.

Nuevo esquema de carpetas

En proyectos profesionales, solemos organizar nuestro código separando **Model** y **Service** dentro de un namespace raíz, por ejemplo `App`. Esto facilita la lectura, mantenimiento y escalabilidad del proyecto.

```
Proyecto/
├─ index.php
├─ log/                    <-- archivos de log
├─ src/                    <-- carpeta raíz del código
│  ├─ Model/               <-- contiene la lógica de datos
│  │  ├─ Biblioteca/
│  │  │  └─ Recurso.php    <-- clase App\Model\Biblioteca\Recurso y sus subclases omitidas aquí por ahorrar espacio
│  │  └─ Infraestructura/
│  │     └─ Recurso.php    <-- clase App\Model\Infraestructura\Recurso
│  ├─ Service/             <-- contiene la lógica de negocio
│  │  └─ Exportador.php    <-- interfaz App\Service\Exportador y las clases que la implementan, omitidas aquí para ahorrar
espacio
│  └─ Traits
│        └─ Logger.php      <-- trait App\Service\Traits\Logger
```

Correspondencia entre carpetas y namespaces

- `App\Model\Biblioteca\Recurso` → corresponde a `src/Model/Biblioteca/Recurso.php`
- `App\Model\Infraestructura\Recurso` → corresponde a `src/Model/Infraestructura/Recurso.php`
- `App\Service\ExportadorService` → corresponde a `src/Service/ExportadorService.php`
- `App\LoggerTrait` → corresponde a `src/LoggerTrait.php`

Carpetas Model y Service

- **Model:** Representa datos o entidades del dominio (por ejemplo, libros, vídeos, hardware). Contiene propiedades y lógica mínima de manipulación de datos.
- **Service:** Contiene la *lógica de negocio* que opera sobre los modelos, como exportar datos, validar reglas o coordinar varias entidades.

Separar **Model** y **Service** ayuda a:

- Mantener el código limpio y organizado.
- Facilitar las pruebas unitarias y el mantenimiento.
- Seguir buenas prácticas y patrones de diseño como MVC

Utilización de “App” como namespace lógico raíz

- Permite diferenciar nuestro código de librerías externas (por ejemplo, `Monolog\Logger` o `Symfony\Component`) que se usarán más adelante
- Veremos que facilita la autocarga cuando usemos Composer mapeando `App\` a `src/`
- Nos da flexibilidad: podemos reorganizar las carpetas sin romper los namespaces.
- Si fuésemos a crear código distribuible como una librería para uso de terceros en lugar de App, se usaría el nombre de la empresa o de la librería, por ejemplo, la librería de MongoDB -> usábamos `new MongoDB\Client();`

Namespaces vs Carpetas en PHP

Es importante no confundir **namespaces** con las carpetas físicas en las que se guarda el código. Algunas claves para entenderlo:

- **Namespace:** Es un *contexto lógico* para organizar clases, interfaces, traits y funciones. Por ejemplo:

  ```
  namespace App\Model\Biblioteca;

  class Recurso { ... }
  ```

  Aquí `App\Model\Biblioteca` es solo un nombre de espacio, no tiene que existir como carpetas físicas obligatoriamente, aunque sí es convención en los subniveles.

- **Carpeta física:** Es la ubicación real de los archivos en el sistema. Por ejemplo:

  ```
  src/Model/Biblioteca/Recurso.php
  ```

  Aunque la carpeta suele reflejar la estructura de namespaces (por claridad y compatibilidad con el estándar PSR-4), no es un requisito.

- **Regla práctica:** Los namespaces definen cómo PHP resuelve las clases, mientras que las carpetas ayudan a organizar los archivos en el proyecto. Con autoloaders o autocarga, podemos mapear un namespace a cualquier carpeta, incluso si los nombres no coinciden exactamente.

Conceptualmente, los **namespaces** pueden considerarse como “apellidos” que distinguen clases con el mismo nombre, mientras que las carpetas son simplemente “cajas” donde guardas esos archivos. Pueden coincidir, pero no siempre es obligatorio.

19. Namespaces (continuación)

Namespaces en PHP: Referencias y la constante mágica __NAMESPACE__

En PHP, para acceder a cualquier clase, función o constante **globales**, se puede utilizar un **nombre completamente cualificado** con una barra invertida inicial:

- `\strlen()` la función strlen() de PHP
- `\Exception` la clase genérica Exception
- `\PHP_INT_MAX` constante que representa el entero más grande soportado en PHP.

1. Formas de referirse a una clase, interfaz, trait, ... en cuanto a namespace se refiere

Se puede hacer referencia a una clase de tres maneras:

a) Nombre no cualificado

Es el nombre de clase sin prefijo. Se interpreta dentro del **namespace actual**. Por ejemplo:

```
namespace App\Model\Biblioteca;

$libro = new Recurso(); // Se interpreta como App\Model\Biblioteca\Recurso
```

b) Nombre cualificado

Es un nombre de clase con un prefijo relativo al namespace actual:

```
namespace App\Model\Biblioteca;

$exportador = new Exportador\ExportadorJSON();
// Se interpreta como App\Model\Biblioteca\Exportador\ExportadorJSON y no se reconocerá
```

c) Nombre completamente cualificado

Comienza con una barra invertida y no depende del namespace actual:

```
$recursoHardware = new \App\Model\Infraestructura\Recurso(); // Se interpreta siempre como App\Model\Infraestructura\Recurso
```

2. Constante mágica __NAMESPACE__

El valor de `__NAMESPACE__` es una cadena con el nombre del namespace actual. Ejemplos:

```
namespace App\Model\Biblioteca;

echo __NAMESPACE__; // Salida: "App\Model\Biblioteca"
```

```
namespace App\Model\Biblioteca;
namespace App\Model\Infraestructura;

echo __NAMESPACE__; // Salida: "App\Model\Infraestructura"
```

```
namespace App\Model\Biblioteca;

echo \__NAMESPACE__; // Desde el espacio global devuelve ""
```

2. Cómo importar o usar namespaces

- Para importar un namespace se usa `use` **fuera** del componente que importa el espacio de nombres (la clase o interfaz o trait o constante...) (no confundir con **use** de un **Trait** que va **dentro** de la clase).
- Se debe bajar a nivel del elemento a importar, es decir, hay que usar `use App\Model\Biblioteca\Recurso` y no basta con usar `use App\Model\Biblioteca` si lo que se pretende es usar la clase Recurso.
- Se pueden importar varios elementos a la vez con un namespace común con la sintaxis: `use App\Model\Biblioteca\{Recurso, Libro, Revista}`
- También se pueden agrupar varios namespaces separados por comas, con la misma sentencia use: `use App\Model\Biblioteca\Recurso,`

`App\Model\Infraestructura\Recurso;`

- Cada **namespace** crea un "contexto" propio. Una clase llamada `Recurso` en `App\Model\Biblioteca` no choca con la clase `Recurso` en `App\Model\Infraestructura`.
- Aunque PHP lo permite, no se considera buena práctica declarar varios namespaces dentro de un mismo fichero.
- Para evitar colisiones dentro de un fichero, por ejemplo, cuando necesitamos 2 clases con el mismo nombre en el mismo archivo, podemos u**sar un alias** con `as`:

```php
<?php

   namespace App\Model\Biblioteca;
   use App\Model\Infraestructura\Recurso as RecursoInfra;

class Video extends Recurso
{
    private int $duracion; // Duración en minutos
    //Recursos de infraestructura
    private array $recursos = [];

    public function __construct(
        string $titulo,
        int $duracion
    ) {
        parent::__construct($titulo);
        $this->duracion = $duracion;
        $this->log("Video creado: {$titulo}, duración {$duracion} min");
    }

   public function agregarRecurso(RecursoInfra $recurso): void
    {
        $this->recursos[] = $recurso;
        $this->log("Recurso agregado al video '{$this->titulo}': {$recurso->getDescripcion()}");
    }
// Faltan getters y setters
}
```

Otra posibilidad es **usar el namespace completamente cualificado** en lugar del nombre de la clase allí donde se haga referencia a ella (Ojo a la barra invertida \ inicial que indica que es el namespace global):

```php
<?php

   namespace App\Model\Biblioteca;
   //Ya no se añade el use

class Video extends Recurso
{
    private int $duracion; // Duración en minutos
    //Recursos de infraestructura
    private array $recursos = [];

    public function __construct(
        string $titulo,
        int $duracion
    ) {
        parent::__construct($titulo);
        $this->duracion = $duracion;
        $this->log("Video creado: {$titulo}, duración {$duracion} min");
    }

 public function agregarRecurso(\App\Model\Infraestructura\Recurso $recurso): void
    {
        $this->recursos[] = $recurso;
        $this->log("Recurso agregado al video '{$this->titulo}': {$recurso->getDescripcion()}");
    }
```

20. Autocarga de clases o autoload

Autocarga de clases en PHP

A medida que el número de clases crece y se organizan en distintos directorios, incluir manualmente todos los archivos mediante `include`, `require` o sus versiones `_once` se vuelve poco práctico.

¿Qué es la autocarga de clases?

La **autocarga de clases** permite que PHP cargue automáticamente el archivo de una clase cuando se intenta crear un objeto de dicha clase y esta todavía no existe en el código.

Cuando PHP no encuentra una clase, su nombre se pasa como parámetro a una o varias **funciones de autocarga**, que se encargarán de buscar el archivo correspondiente en los directorios que se hayan definido.

La función `spl_autoload_register`

Para registrar funciones de autocarga se utiliza la función `spl_autoload_register`, que añade la función indicada a una **cola de funciones**.

Esta cola sigue un comportamiento **FIFO (First In, First Out)**, es decir, las funciones se ejecutan en el mismo orden en el que fueron registradas, de forma similar a una cola en una carnicería.

La **Standard PHP Library (SPL)** será la encargada de evaluar esta cola. Cada función registrada recibirá como parámetro el nombre de la clase que se intenta cargar. Si la clase pertenece a un namespace, el namespace formará parte de la clase.

Si todas las clases estuviesen en el mismo directorio y no utilizasen namespaces, esta función podría funcionar:

```
spl_autoload_register(function ($nombre_clase) {
    include $nombre_clase . '.php';
});
```

En este ejemplo básico, PHP intentará incluir un archivo con el mismo nombre que la clase seguido de la extensión `.php`.

Autocarga en nuestro proyecto de biblioteca

En la propuesta de solución de la actividad, las clases se han organizado en distintos subdirectorios, por ejemplo:

- `src/Model/Biblioteca` (Recurso, Libro, Revista, Video)
- `src/Service`(Exportador y sus implementaciones)

Para gestionar esta estructura se ha creado un archivo **autoload.php**. En él se utiliza la función `spl_autoload_register` que recibe una función anónima que recorre un array de directorios posibles.

▸ **Ver código del fichero autoload.php**

```
// Namespace raíz del proyecto
const APP_NAMESPACE_BASE = 'App\\';
// Directorio base del proyecto
const APP_DIRECTORIO_BASE = __DIR__ . DIRECTORY_SEPARATOR . 'src' . DIRECTORY_SEPARATOR;

spl_autoload_register(function (string $nombreClase) {

    // Normalizamos el nombre en el caso de que empiece con \
    $nombreClase = ltrim($nombreClase, '\\');

    // Si la clase NO empieza por App\, no es de nuestra aplicación
    if (strpos($nombreClase, APP_NAMESPACE_BASE) !== 0) {
        return;
    }

    // Quitamos "App\" del inicio del namespace
    $rutaRelativa = substr($nombreClase, strlen(APP_NAMESPACE_BASE));

    // Sustituimos \ por DIRECTORY_SEPARATOR para construir la ruta
    $rutaRelativa = str_replace('\\', DIRECTORY_SEPARATOR, $rutaRelativa);

    // Archivo final
    $archivo = APP_DIRECTORIO_BASE . $rutaRelativa . '.php';

    // Si existe, lo cargamos
    if (file_exists($archivo)) {
        require_once $archivo;
    }
});
```

Explicación: Esta función de autocarga se ejecuta cada vez que PHP intenta usar una clase que no ha sido cargada. Verifica que la clase pertenezca al namespace `App\`, construye la ruta del archivo correspondiente y, si existe, lo incluye automáticamente con `require_once`.

Esta función se ejecuta automáticamente cada vez que PHP intenta utilizar una clase que todavía no ha sido cargada.

En el código propocionado del fichero `autoload.php`, se define como constante el **namespace raíz** de la aplicación (`App\`) y el **directorio base** donde se encuentra el código fuente (`src/`). De este modo, el autoload solo se encarga de cargar las clases que pertenecen a nuestra aplicación.

Cuando se recibe el nombre de una clase, primero se normaliza eliminando una posible barra inversa inicial (`\`), ya que esta no forma parte del nombre real del namespace.

A continuación, se comprueba que la clase pertenezca al namespace `App\`. Si no es así, la función finaliza y no intenta cargar ningún archivo, evitando búsquedas innecesarias o errores.

Si la clase pertenece a la aplicación, se elimina el namespace base y se transforma el resto del namespace en una ruta de directorios, sustituyendo los separadores de namespace (`\`) por el separador de directorios del sistema operativo. Podemos hacerlo porque nuestros niveles de namespace coinciden con la jerarquía de ficheros.

Finalmente, se construye la ruta completa del archivo PHP correspondiente y, si este existe el fichero, se carga mediante `require_once`. Si el archivo no se encuentra, la clase no podrá ser instanciada y PHP generará un error indicando que la clase no existe.

De este modo, al crear un objeto *Libro*, *Revista* o *Video*, PHP cargará automáticamente la clase correspondiente. Ocurrirá así con todos los elementos definidos en su propio fichero: interfaces, clases abstractas, implementaciones, etc.

Importante:
Si no se encuentra el archivo de una clase, no se podrá crear el objeto y se producirá un error fatal del tipo:
`Fatal error: Uncaught Error: Class "X" not found`

¿Cómo se usa?

Para que la función de autocarga funcione**, solo hay que incluir el archivo `autoload.php`** al inicio de la aplicación, normalmente en el archivo raíz (`index.php`) o en cualquier punto de entrada principal del proyecto.

Por ejemplo, en `index.php` pondríamos:

```
// Cargamos el autoload de la aplicación
require_once __DIR__ . '/autoload.php';

// Ahora podemos usar cualquier clase del namespace App sin hacer include manual
use App\Model\Biblioteca\Recurso;
use App\Service\Exportador;
// Continúa con el flujo normal de index.php
$libro  =new Libro("Aprendiendo PHP", "978-3-16-148410-0");
//...
```

Documentación y ejemplos

Documentación oficial sobre la autocarga de clases en PHP:
https://www.php.net/manual/es/language.oop5.autoload.php

21. Tipos enum o enumeraciones

Enums en PHP (PHP 8.1+)

Un **enum** es un tipo especial que permite definir un **conjunto cerrado de valores posibles.** Se usa cuando una variable solo puede tomar ciertos valores válidos.

¿Por qué usar enum?

- Evita valores inválidos
- Mejora la legibilidad del código
- Facilita el mantenimiento
- Reemplaza strings "mágicos"

Ejemplo: Estado de un recurso

Se definen en su propio fichero `EstadoRecurso.php`

```
namespace App\Model\Biblioteca\Enum;

enum EstadoRecurso: string {
    case DISPONIBLE = 'disponible';
    case PRESTADO = 'prestado';
    case RESERVADO = 'reservado';
}
```

Uso en una clase

```
use App\Model\Biblioteca\Enum\EstadoRecurso;

class Recurso {
    private EstadoRecurso $estado;

   // otras propiedades, constructor, getters y setters

    public function isDisponible(): bool {
        return $this->estado === EstadoRecurso::DISPONIBLE;
    }
}
```

Con **enum**, el estado del recurso siempre será válido y controlado desde el propio lenguaje.

22. Excepciones personalizadas

Manejo de excepciones en PHP

En PHP, las **excepciones** permiten controlar errores que podrían interrumpir la aplicación, como intentar prestar un recurso que no está disponible o un usuario inexistente.

Ejemplo en nuestra Biblioteca

```
    if (!$recurso->isDisponible()) {
        throw new \Exception("Recurso no disponible");
    }
```

Al lanzar una excepción con `throw`, el flujo del método se detiene y se puede capturar desde otro bloque con:

```
try {
    $prestamo = $servicio->prestar("Juan", 123);
} catch (\Exception $e) {
    echo "No se pudo realizar el préstamo: " . $e->getMessage();
}
```

Esto permite que la aplicación maneje errores de forma controlada, mostrando un mensaje al usuario o registrándolo en un log, sin detener todo el programa.

Excepciones personalizadas

En el caso de que, dependiendo del tipo de excepción, se requiera un tratamiento diferente, por ejemplo:

- Para **mostrar mensajes diferentes al usuario** según el tipo de error
- Para **reaccionar de forma distinta** (ej: enviar email si un recurso no está disponible)
- Si el sistema crece y se empieza a tener **muchos tipos de errores distintos** en los servicios

```
namespace App\Exception;

use Exception;

class RecursoNoDisponibleException extends Exception
{
    public function __construct($mensaje = "El recurso no está disponible para préstamo.", $codigo = 0)
    {
        parent::__construct($mensaje, $codigo);
    }
}
```

Cómo lanzar la excepción personalizada

```
 use App\Exception\RecursoNoDisponibleException;

if (!$recurso->isDisponible()) {
    throw new RecursoNoDisponibleException();
}
```

Cómo capturarla

El orden influye. Siempre se debe capturar la excepción más específica antes que la más general.

```
try {
    $prestamo = $biblioteca->prestar("Juan", 123);
} catch (RecursoNoDisponibleException $e) {
    echo "No se pudo prestar el recurso: " . $e->getMessage();
} catch (\Exception $e) {
    echo "Otro error: " . $e->getMessage();
}
```

23. Tipo DateTimeImmutable

Trabajando con `DateTimeImmutable` en PHP

DateTimeImmutable es una clase para manejar fechas y horas en PHP sin permitir modificaciones al objeto original.

1. Crear la fecha actual

```
$ahora = new DateTimeImmutable(); //crea un objeto con la fecha y hora actuales
echo $ahora->format('d-m-Y H:i:s'); //crea una cadena con el formato indicado
```

2. Crear una fecha y hora específica

```
$fecha = new DateTimeImmutable('2026-01-30 14:30:00');
echo $fecha->format('d-m-Y H:i:s');
```

3. Formatear a string

Usamos `format('d-m-Y H:i:s')` para mostrar la fecha en formato **día-mes-año hora:minutos:segundos**.

4. Uso como atributo en una clase

```
class Prestamo {

    private DateTimeImmutable $fechaPrestamo;
    private ?DateTimeImmutable $fechaDevolucion=null;
}
```

Referencia oficial de PHP:
https://www.php.net/manual/es/datetime.format.php
Allí encontrarás todos los patrones y su significado (`d` = día, `m` = mes, `Y` = año completo, `H` = hora 24h, `i` = minutos, `s` = segundos).

24. Servicios

Lógica de negocio en servicios

En aplicaciones web, es buena práctica separar la **lógica de negocio** de las entidades y del acceso a datos. Para esto usamos **servicios**, que son clases encargadas de ejecutar operaciones relacionadas con casos de uso.

1. Qué hacen los servicios

- Contienen la lógica de negocio: validaciones, reglas, coordinación entre entidades.
- No almacenan datos por sí mismos (excepto referencias a repositorios o entidades). Los repositorios veremos que son elementos que interactúan con las bases de datos para realizar las operaciones CRUD y las entidades son las clases que tendrán su correspondencia en la base de datos.
- Permiten que las entidades (como `Usuario` o `Recurso`) sean más limpias y agnósticas. (Dedicadas únicamente a representar datos y no tanto a la lógica de la aplicación).

2. Métodos por caso de uso

Cada método de un servicio normalmente corresponde a un **caso de uso** (funcionalidad) de la aplicación:

```
class PrestamoService {
    public function prestarRecurso(string $emailUsuario, int $idRecurso) { ... }
    public function devolverRecurso(int $idPrestamo) { ... }
    public function renovarPrestamo(int $idPrestamo) { ... }
}
```

3. Beneficios

- Entidades más simples y reutilizables.
- Lógica concentrada y fácil de probar unitariamente.
- Facilita mantenimiento y evolución de la aplicación.
- Los controladores web solo llaman al servicio y manejan la presentación y respuesta.

Resumen práctico:
- Servicio = lógica de negocio centralizada
- Método = caso de uso concreto
- Entidades = simples, sin reglas de negocio complejas

25. Composer

1.Utilización de Composer

Sabemos de la unidad 2 y 3 que Composer es un **gestor de paquetes para PHP**. Nos permite:

- Instalar y actualizar librerías externas fácilmente (como PHPUnit).
- Gestionar dependencias de forma ordenada.
- Pero también, ayuda a **estandarizar la carga automática de clases (autoload)** con **PSR-4.**

PSR-4

PSR-4 (PHP Standard Recommendation) es una **recomendación oficial de PHP** que define cómo cargar clases automáticamente según su **namespace**. Se basa en 2 aspectos principalmente

-

1. Namespace raíz → carpeta base

En `composer.json` se define la relación entre el *namespace raíz* y la carpeta donde se encuentran las clases:

```
"autoload": {
    "psr-4": {
        "App\\": "src/"
    }
}
```

Esto indica que **todas las clases que empiecen por `App\` se encuentran en la carpeta `src/`**.

2. Cada subnivel del namespace → subcarpeta

Cada `\` del namespace se traduce en una subcarpeta dentro de la carpeta base. Por ejemplo:

```
namespace App\Model\Biblioteca;

class Libro {}
```

Se traduce a la siguiente estructura de carpetas:

```
src/
└── Model/
    └── Biblioteca/
        └── Libro.php
```

Así, al usar Composer, basta con:

```

require_once __DIR__.DIRECTORY_SEPARATOR."vendor".DIRECTORY_SEPARATOR."autoload.php";

use App\Model\Biblioteca\Libro;

$libro = new Libro();
```

Composer ya sabe dónde buscar el archivo `Libro.php` sin necesidad de `require` manual ni del autoload.php que vimos en la sección 20.

Resumen

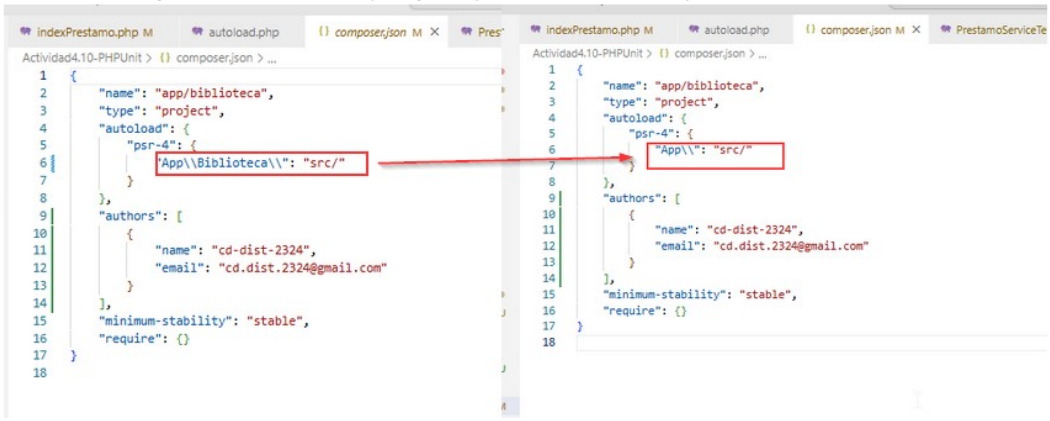
- **Namespace raíz → carpeta base**: indica dónde empiezan a vivir todas las clases.
- **Subniveles → subcarpetas**: cada nivel del namespace corresponde a una carpeta.
- Gracias a PSR-4, las clases se cargan automáticamente y el proyecto queda organizado y predecible.

Ventajas principales

- Ya no necesitamos un `autoload.php` manual.
- Todas las librerías y nuestras clases se cargan automáticamente.
- Fácil integración con frameworks (Symfony, Laravel, etc.)

2. Pasos para integrar Composer en el proyecto

1. **Inicializar Composer en un terminal (dentro de la carpeta raíz del proyecto):** `composer init`
2. **Respuestas al menú:**
   - Package name: `app/biblioteca`
   - Description: opcional (tecla ENTER)
   - Author: opcional (tecla ENTER)
   - Minimum stability: `stable`
   - Package Type: `project`
   - License: (tecla *ENTER) o MIT*
   - Dependencies interactivas: `no`
   - Dependencies interactivas de dev: `no`
   - PSR-4 autoload mapping: En principio, ENTER y luego lo ajustamos en composer.json : `App\\ → src/`



3. **Actualizar autoload** en la terminal, en la raíz del proyecto: `composer dump-autoload`
4. **Cambiar autoload en el código:** en `index.php` o puntos de entrada, sustituir:

```
require_once __DIR__.DIRECTORY_SEPARATOR."autoload.php";
```

por:

```
require_once __DIR__.DIRECTORY_SEPARATOR."vendor".DIRECTORY_SEPARATOR."autoload.php";
```

Probaremos a ejecutar el index.php y comprobar si se ejecuta correctamente.

26. PHPUnit

1. Introducción a PHPUnit

PHPUnit es una librería que permite **probar automáticamente código PHP**. Nos ayuda a:

- Verificar que los servicios funcionan correctamente.
- Detectar errores sin necesidad de ejecutar toda la aplicación.
- Diseñar mejor las clases siguiendo buenas prácticas (TDD, separación de responsabilidades).

2. Instalación de PHPUnit

1. Instalaremos la librería de PHPUnit con Composer: `composer require --dev phpunit/phpunit`
2. Crear carpeta `tests/` en la misma ubicación que `src/`. Dentro de la carpeta `tests/` crearemos una subcarpeta `Service/`. Dentro de la carpeta `tests/` se suele replicar la misma estructura de carpetas que en `src/`
3. Crear un archivo `phpunit.xml` en la carpeta raíz del proyecto para configurar el arranque y los conjuntos de tests:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit colors="true" bootstrap="vendor/autoload.php">
  <testsuites>
    <testsuite name="Biblioteca">
      <directory>tests</directory>
    </testsuite>
  </testsuites>
</phpunit>
```

- **Con bootstrap,** le indicamos a PHPUnit **qué archivo cargar primero**, normalmente el autoload de Composer. Esto permite que todas las clases de `src/` y `tests/` se puedan usar **sin hacer require manuales**.
- Permite agrupar tests en **testsuites (conjuntos de tests).** En el ejemplo, se le dice a PHPUnit: **"busca todos los tests en la carpeta `tests/` ".**
- Con colors="true", veremos el resultado de color verde o rojo para indicar el éxito o fracaso del test (más visual)

Sin `phpunit.xml` tendríamos que escribir en la terminal: `vendor/bin/phpunit --bootstrap vendor/autoload.php tests/`

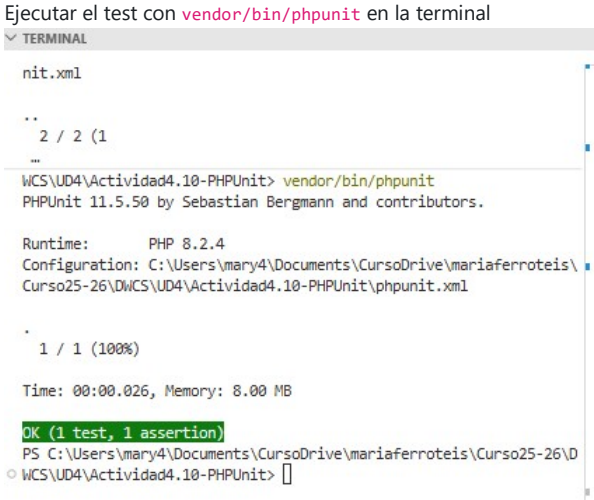
En cambio, con phpunit.xml, escribimos en la terminal simplemente: `vendor/bin/phpunit`

3. Actividad práctica 2

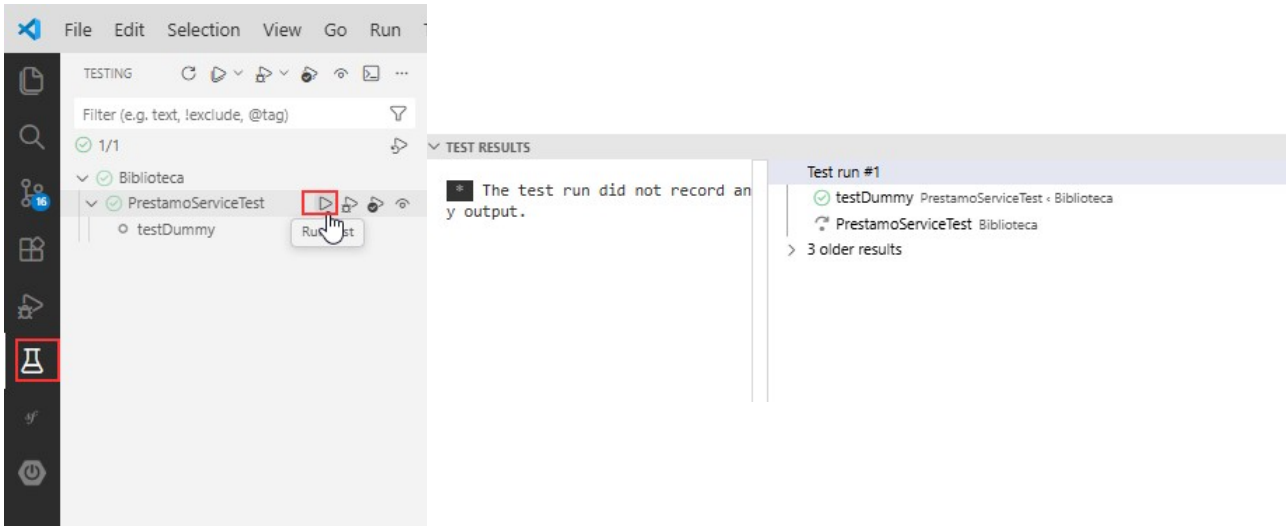
- Crear un test sencillo que siempre pase, solo para comprobar que la librería PHPUnit está bien instalada. Las clases de test Se suelen llamar igual que la clase que estamos probando terminada **con el sufijo *Test*** y extienden de `PHPUnit\Framework\TestCase` :

```
use PHPUnit\Framework\TestCase;

class PrestamoServiceTest extends TestCase {
  public function testDummy() {
    $this->assertTrue(true);
  }
}
```

- Ejecutar el test con `vendor/bin/phpunit` en la terminal



- O ejecutar el test desde la interfaz gráfica desde VS Code.



- Observar resultado OK o verde

26.1. Tests unitarios

Tipos de tests

Tipo de test	Qué prueba	Ejemplo en nuestra aplicación
Unitario	Prueba una unidad pequeña de código de forma aislada. Rápido, repetible y sin depender de recursos externos.	Método `prestar()` en `PrestamoServiceTest` usando usuarios y recursos en memoria.
Integración	Comprueba que varias partes del sistema funcionan juntas. Incluye dependencias reales como base de datos o repositorios.	Repositorio de Usuarios accediendo a una base de datos de pruebas para verificar consultas.

Existen también otros tipos de tests, como los **funcionales** o **end-to-end**, que prueban la aplicación completa desde la perspectiva del usuario, así como tests de rendimiento o de seguridad. Estos no se profundizan en esta asignatura, pero es importante saber que existen.

Tests unitarios

Un **test unitario** es una prueba automática que verifica que una **unidad pequeña de código** (normalmente un método o una clase) funciona correctamente de forma **aislada**, para unos datos concretos.

Los tests unitarios se caracterizan por ser **automáticos, repetibles y rápidos**, lo que permite ejecutarlos de forma frecuente durante el desarrollo.

El objetivo de un test unitario es comprobar el **comportamiento** del código, no su implementación interna, y permitir detectar errores de forma rápida y fiable.

Aplicación al `PrestamoServiceTest`

En esta práctica, el método `prestar(string $nombreUsuario, int $idRecurso)` del servicio **PrestamoService** es una buena candidata para ser probada mediante tests unitarios, ya que contiene lógica de negocio clara y reglas bien definidas.

El test unitario debe comprobar que este método:

- Lanza una excepción si el usuario no existe.
- Lanza una excepción si el recurso no existe.
- Lanza una excepción si el recurso no está disponible.
- Lanza una excepción si el usuario ha alcanzado el máximo de préstamos.
- Devuelve un objeto `Prestamo` válido cuando se cumplen todas las condiciones.

Cada uno de estos casos se prueba en un método independiente dentro de `PrestamoServiceTest`, asegurando que el comportamiento del servicio es el esperado.

Por qué este test es unitario

El test de `PrestamoService` es un **test unitario** porque:

- Se prueba una única unidad de código: el método `prestar()`.

- No se accede a bases de datos ni a servicios externos.
- Los usuarios y recursos se crean en memoria y están totalmente controlados.
- Si el test falla, se puede identificar fácilmente qué regla de negocio no se cumple.

Gracias a estos tests, cualquier cambio en la lógica de préstamos puede validarse de forma automática, aumentando la fiabilidad y mantenibilidad del código.

26.2. Uso de PHPUnit

PHPUnit: Cómo escribir tests

1. Convenciones básicas

- Las clases de test suelen llamarse igual que la clase que prueban, **terminando en** `Test`. Ejemplo: `PrestamoServiceTest`.
- Cada clase de test **hereda de** `PHPUnit\Framework\TestCase`.
- Para organizar el código de tests, vamos a crearlos dentro del `namespace App\Tests\*`. Ejemplo: `namespace App\Tests\Service;`
- También modificaremos el archivo composer.json para añadir el mapeado del `namespace App\Tests\` a la carpeta `tests/`

```
    "autoload": {
        "psr-4": {
            "App\\": "src/"
        }
    },
     "autoload-dev": {
        "psr-4": {
            "App\\Tests\\": "tests/"
        }
    },
```

- Después de cambiar los mapeados ejecutaremos `composer dump-autoload`
- Los métodos de test suelen empezar con `test` y tener un nombre descriptivo de la casuística que prueban.

2. Métodos assert

PHPUnit y la clase `PHPUnit\Framework\TestCase` proporcionan muchos métodos para verificar condiciones:

- `$this->assertTrue($condicion, $message="")`. Para condiciones booleanas, por ejemplo: `$this->assertTrue($libro->isDisponible());`
- `$this->assertFalse($condicion, $message="")`
- `$this->assertEquals($esperado, $real, $message="")`. Para comparar valores u objetos iguales. El mensaje de error es más expresivo que en los dos casos anteriores: *Failed asserting that 4 matches expected 1.*
- `$this->assertCount($numero, $array, $message="")`. Útil para comprobar el tamaño de un array.
- `$this->assertInstanceOf(Clase::class, $objeto, $message="")`. Comprueba que $objeto sea de la clase Clase::class, por ejemplo:

```
$this->assertInstanceOf(Recurso::class, $libro);
```

- Y muchos más (`assertEmpty`, `assertContains`, etc.)

> NOTA: ::class es un operador que devuelve el nombre completo de una clase como string.

Podéis consultar más métodos en la documentación oficial: https://docs.phpunit.de/en/11.5/assertions.html

En todos ellos, `$message` es un mensaje personalizado que aparece si el test falla. Es opcional y si no se pone nada, PHPUnit muestra un mensaje genérico, por ejemplo: *Failed asserting tha 2 matches expected 1*

3. Comprobar que se lanza una excepción

Si queremos probar que un método lanza una excepción:

```
namespace App\Tests\Service;
use PHPUnit\Framework\TestCase;

class PrestamoServiceTest extends TestCase {

    public function testUsuarioNoEncontradoLanzaExcepcion() {
        $this->expectException(\Exception::class);
        $this->expectExceptionMessage("Usuario no encontrado");

        $service = new \App\Service\PrestamoService();
        $service->prestar("inexistente", 1);
    }
}
```

Con `expectException` y `expectExceptionMessage` podemos verificar tanto el tipo de excepción como el mensaje.

4. Métodos de inicialización y limpieza

PHPUnit tiene métodos especiales para preparar el entorno:

- `setUpBeforeClass(): void` → se ejecuta **una vez antes de todos los tests**
- `setUp(): void` → se ejecuta **antes de cada test**
- `tearDown(): void` → se ejecuta **después de cada test** para limpiar.
- `tearDownAfterClass(): void` → se ejecuta **una vez después de todos los tests**.

Método	Ejecuta antes de	Atributos	Independencia	Cuándo usar
protected setUp()	Cada test	Instancia (`$this->`)	Cada test empieza limpio	Para objetos que cambian entre tests y garantizar independencia
public static setUpBeforeClass()	Una vez antes de todos los tests	Estático (`self::$`)	Todos los tests comparten el mismo estado	Para objetos pesados que no cambian y optimizar la ejecución (una conexión a BD)

Las mismas consideraciones de visibilidad y static se aplicarían para los métodos tearDown, salvo en que se ejecutan después de cada test o después de todos los tests.

5. Ejemplo resumido de test para `PrestamoService`

```php
<?php
namespace App\Tests\Service;

use App\Model\Biblioteca\Enum\EstadoRecurso;
use App\Model\Biblioteca\Libro;
use PHPUnit\Framework\TestCase;
use App\Service\PrestamoService;
use App\Model\Biblioteca\Usuario;

class PrestamoServiceTest extends TestCase
{

    private PrestamoService $service;

    protected function setUp(): void {
        $this->service = new PrestamoService();
        $usuario = new Usuario("juan", "juan@example.com");
        $libro = new Libro("Libro de prueba", "Autor de prueba");

        $this->service->registrarUsuario($usuario);
        $this->service->registrarRecurso($libro);
    }

    public function testPrestarRecursoDisponible(): void {
        $prestamo = $this->service->prestar("juan@example.com", 1);
        $this->assertEquals(EstadoRecurso::PRESTADO, $prestamo->getRecurso()->getEstado());
    }

    public function testUsuarioNoEncontradoLanzaExcepcion(): void {
        $this->expectException(\Exception::class);
        $this->expectExceptionMessage("Usuario no encontrado");

        $this->service->prestar("inexistente", 1);
    }

}
```

Observa cómo:

- La clase termina en `Test` y hereda de `TestCase`.
- Se usa `setUp()` para preparar datos antes de cada test.
- Se comprueban resultados con métodos `assert*` y excepciones con `expectException`.

26.3. Consejos para escribir tests

1 Casos básicos a probar

- **Flujo normal (happy path)**: Usuario registrado y recurso disponible → préstamo correcto.
  - Estado del recurso = `PRESTADO`
  - El préstamo se añade al usuario
- **Errores esperados**:
  - Usuario no registrado → lanza excepción "Usuario no encontrado"
  - Recurso no registrado → lanza excepción "Recurso no encontrado"
  - Recurso no disponible → lanza excepción "Recurso no disponible"
  - Usuario con máximo de préstamos → lanza excepción "El usuario ha alcanzado el máximo de préstamos"
- **Devolución de préstamo**:
  - Recurso vuelve a `DISPONIBLE`
  - Préstamo eliminado del usuario

2 Cobertura y caminos críticos

Tipos de cobertura a considerar:

- **Métodos:** todos los métodos públicos deberían tener al menos un test.
- **Condiciones:** cada `if / else` → un test por camino.
- **Combinaciones:** escenarios con varias condiciones (usuario registrado pero recurso no disponible, etc.).

Camino crítico: partes del sistema donde un fallo tendría mayor impacto. Ejemplos:

- Prestar un recurso no disponible
- Usuario con máximo de préstamos
- Usuario o recurso no registrados

> Mensaje clave: Un buen test cubre todos los escenarios de uso normal, los errores esperados y los caminos críticos de la lógica.

3.1 Cómo ver la cobertura en HTML

1. Configura Xdebug en php.ini con `xdebug.mode=coverage`.
2. Añade en `phpunit.xml` la etiqueta <source> con el filtro de código .php para indicar dónde está el código fuente a analizar :

```
...
</testsuites>
  <source>
    <include>
      <directory suffix=".php">src</directory>
    </include>
  </source>
...
</phpunit>
```

3. Ejecuta PHPUnit la generación de un informe de cobertura en formato HTML dentro del directorio *coverage*

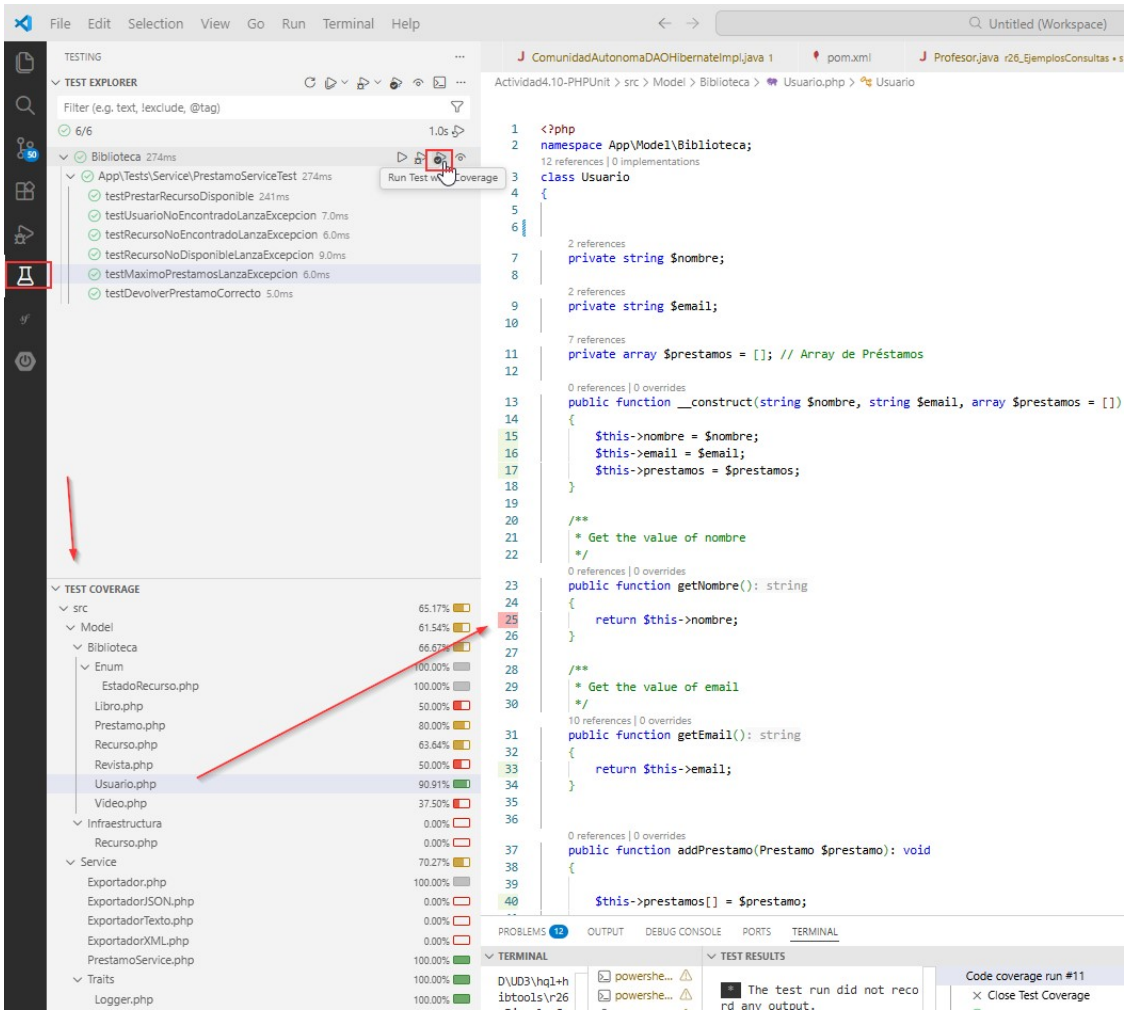
```
vendor/bin/phpunit --coverage-html coverage
```

4. Abre el informe en navegador:

```
coverage/index.html
```

3.2 Cómo ver la cobertura en VS Code

En la sección de Testing, tenéis también la opción de *Run Test with Cover.* Ejecutará los tests y os indicará en el panel Test Coverage el porcentaje del código probado. Si hacéis clic sobre cada fichero, veréis un indicador en rojo/verde para indicar las líneas que se han probado y las que no.



4 Tips prácticos para escribir tests

- Piensa en **historias de usuario**: caso feliz y casos de error.
- **Un test por comportamiento:** no mezclar varias comprobaciones en un mismo test.
- Usa `setUp()` para preparar el escenario (usuarios y recursos). *No hacer asserts aquí.*
- Prueba errores explícitamente con `expectException()`.
- Nombres descriptivos de test: `testPrestarRecursoDisponible`, `testUsuarioNoEncontradoLanzaExcepcion`.
- Prioriza lógica importante: estados del recurso, límites de préstamos, excepciones.

5 Tabla resumen de cobertura

Escenario	Qué comprobar
Flujo normal	Préstamo creado, estado recurso = `PRESTADO`, préstamo añadido al usuario
Usuario no encontrado	Lanza excepción "Usuario no encontrado"
Recurso no encontrado	Lanza excepción "Recurso no encontrado"
Recurso no disponible	Lanza excepción "Recurso no disponible"
Usuario con máximo de préstamos	Lanza excepción "El usuario ha alcanzado el máximo de préstamos"
Devolución	Estado recurso = `DISPONIBLE`, préstamo eliminado del usuario

26.4. Foco en tests útiles y no en 100% de cobertura

Tests y cobertura

Antes de avanzar, recordemos cómo escribir tests que aportan valor:

- **Útil:** comprueba *comportamiento observable* y *lógica de negocio*.
- **Independiente:** no depende de otros tests ni del estado global.
- **Descriptivo:** nombre claro, un test por comportamiento.

Ejemplo de test útil:

```
public function testPrestarRecursoDisponible(): void
{
    $usuario = $this->service->getUsuarioByEmail("juan@example.com");
    $prestamosAntesDePrestar = $usuario->getPrestamos();

    $prestamo = $this->service->prestar("juan@example.com", 1);

    $this->assertEquals(EstadoRecurso::PRESTADO, $prestamo->getRecurso()->getEstado());
    $this->assertCount(count($prestamosAntesDePrestar)+1, $prestamo->getUsuario()->getPrestamos());
}
```

Ejemplo de test inútil (no aporta valor):

```
public function testIdDelLibro(): void
{
    $libro = new Libro("Libro de prueba", "Autor");
    $this->assertEquals(1, $libro->getId());
}
```

❌ Solo prueba cómo está implementado el ID, no la lógica de negocio. Puede fallar si otros tests crean libros antes.

💡 Cobertura 100% no significa tests buenos. Prioriza caminos críticos, happy path y errores esperados.

26.5. Ejemplo de tests unitarios de Repository con BD en memoria

1 Tests de Repositorios con SQLite en memoria

Para hacer tests de los repositorios sin tocar la base de datos real, es más cómodo y rápido usar **SQLite en memoria. (**También es posible utilizar una base de datos de pruebas que replique el esquema de la de producción)

2 Ventajas de SQLite en memoria

- Muy rápida, no requiere servidor.
- Cada test empieza con BD limpia.
- Evita contaminar la base de datos real.

3 Pasos para crear un test con SQLite.

0. Asegúrate de que la extensión PDO para SQLite está habilitada en php.ini:

```
943 ;extension=pdo_firebird
944 extension=pdo_mysql
945 ;extension=pdo_oci
946 ;extension=pdo_odbc
947 ;extension=pdo_pgsql
948 extension=pdo_sqlite
949 ;extension=pgsql
950 ;extension=shmop
```

1. Crear la conexión PDO a SQLite en memoria :

```
$pdo = new PDO('sqlite::memory:');
```

2. Crear las tablas necesarias para el test (lo encapsularemos en un método `crearEsquema()`):

```
$pdo->exec("
    CREATE TABLE usuarios (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        nombre TEXT NOT NULL,
        email TEXT NOT NULL
    );
");
```

3. Crear el Repository pasando la conexión PDO:

```
$repository = new UsuarioRepository($pdo);
```

4. Escribir tests que usen el Repository para guardar y recuperar datos

4 Ejemplo completo de test de UsuarioRepository

```php
namespace App\Tests\Repository;
use PDO;
use PHPUnit\Framework\TestCase;
use App\Model\Biblioteca\Usuario;
use App\Repository\UsuarioRepository;


class UsuarioRepositoryTest extends TestCase
{
    private PDO $pdo;
    private UsuarioRepository $repository;

    protected function setUp(): void
    {
        $this->pdo = new PDO('sqlite::memory:');
         //Configuramos la conexión para que lance PDOExcepction en caso de error (no es necesario para versione PHP recientes)
        $this->pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

       //Creamos la/s tablas necesarias
        $this->crearEsquema();
        $this->repository = new UsuarioRepository($this->pdo);
    }

    private function crearEsquema(): void
    {
        $this->pdo->exec("
            CREATE TABLE usuarios (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                nombre TEXT NOT NULL,
                email TEXT NOT NULL
            )
        ");
    }

    public function testCrearYBuscarUsuario(): void
    {
        $usuario = new Usuario("Juan", "juan@example.com");

        $usuario = $this->repository->create($usuario);

        $this->assertNotNull($usuario);
        $this->assertNotNull($usuario->getId());

        $usuarioBD = $this->repository->findByEmail("juan@example.com");

        $this->assertNotNull($usuarioBD);
        $this->assertEquals("Juan", $usuarioBD->getNombre());
        $this->assertEquals("juan@example.com", $usuarioBD->getEmail());
        $this->assertEquals($usuario->getId(), $usuarioBD->getId());
    }
}
```

27. Patrón Repository

Patrón Repository **(Repositorio)** es un patrón de diseño de acceso a datos . Actúa como un **mediador** entre la lógica de negocio (servicios) y la capa de datos, abstrayendo cómo se almacenan, buscan y recuperan los objetos, facilitando el mantenimiento y las pruebas unitarias al centralizar las operaciones CRUD. En definitiva, el **Repository** es el componente que se encarga de interactuar con el sistema de persistencia, es decir, *guardar y recuperar objetos* de un almacenamiento (BD, memoria, fichero...).

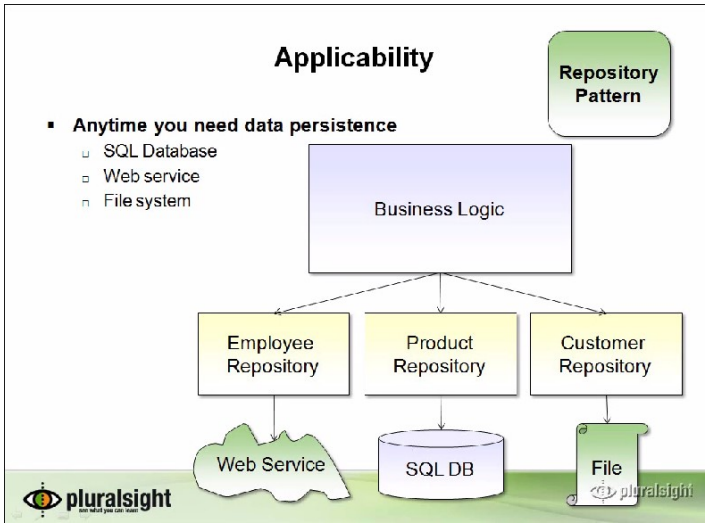
Beneficios:

- Separa *Service* de acceso a datos
- Permite *tests independientes* usando repositorios en memoria o SQLite
- Permite interactuar con la capa de persistencia como si fuera una colección de objetos (con métodos create, read, delete, update y findByCondiciones, etc).

Hasta ahora, nuestra Biblioteca almacena datos en memoria, pero si quisiéramos almacenarlos en una base de datos relacional, como MySQL, lo ideal sería crear componentes Repository.

El repositorio puede trabajar con cualquier sistema de persistencia. Lo ideal es crear una interfaz y que haya una implementación dependiente del tipo de persistencia elegida.

En la imagen, se observa cómo la Capa de Negocio (los servicios) usan los repositorios para obtener/guardar datos. La capa de servicios no sabe dónde se guardan o de dónde se recuperan porque obtienen los datos de los Repositorios. Son estos últimos quienes sí saben del sistema de persistencia o de la fuente de datos, que puede ser una base de datos, un fichero o un servicio web.



Ejemplo de Repository abstracto:

Las operaciones comunes como:

- SELECT * FROM tabla;
- SELECT * FROM tabla WHERE claveprimaria=?

podrían incluirse en una clase abstracta AbstractRepository.

```
namespace App\Repository;
use PDO;

abstract class AbstractRepository
{
    protected PDO $pdo;
    protected string $table;
    protected string $primaryKey = 'id';

    public function __construct(PDO $pdo)
    {
        $this->pdo = $pdo;
    }

    public function findAll(): array
    {
        $stmt = $this->pdo->query("SELECT * FROM {$this->table}");
        return $stmt->fetchAll(PDO::FETCH_ASSOC);
    }

    public function find(int $id): ?array
    {
        $stmt = $this->pdo->prepare("SELECT * FROM {$this->table} WHERE {$this->primaryKey} = ?");
        $stmt->execute([$id]);
        return $stmt->fetch(PDO::FETCH_ASSOC) ?: null;
    }

   public abstract function create(object $object): ?object ;  //La creación y la actualización son más dependientes del nº de
columnas de la tabla y menos sencillo de generalizar.
   // Podría haber otro método abstracto para update

}
```

Para cada entidad (o clase persistente) habrá un repositorio concreto. Por ejemplo: Para la clase Usuario, habrá un UsuarioRepository.

Ejemplo de un Repository concreto:

```php
namespace App\Repository;
use PDO;

class UsuarioRepository extends AbstractRepository
{
    protected string $table = 'usuarios';

    public function findByEmail(string $email): ?Usuario
    {
        $stmt = $this->pdo->prepare("SELECT * FROM  {$this->table} WHERE email = ?");
        $stmt->execute([$email]);
        $datos = $stmt->fetch(PDO::FETCH_ASSOC) ?: null;
        if ($datos) {
            $usuario = new Usuario($datos['nombre'], $datos['email']);
            $usuario->setId($datos['id']);
            return $usuario;
        }
        else {
            return null;
        }
    }


    public function create($usuario): ?object
    {
        $stmt = $this->pdo->prepare(
            "INSERT INTO  {$this->table}(nombre, email) VALUES (?, ?)"
        );
        $stmt->execute([
            $usuario->getNombre(),
            $usuario->getEmail()
        ]);

        //Habría que añadir a Usuario el atributo id nullable y su getter y setter para que esto funcione
        $id = $this->pdo->lastInsertId();
        if($id === false) {
            return null;
        }
        $usuario->setId($id);
        return $usuario;

    }
}
```

Podemos usar este Repository con:

- PDO + SQLite en memoria (para tests rápidos)
- PDO + MariaDB/MySQL (base de datos de pruebas MariaDB/MySQL)

27.1. PrestamoServiceConRepo

Asociación `PrestamoServiceConRepo` y `UsuarioRepository`

Hasta ahora hemos visto un PrestamoService con sus datos guardados en arrays, en memoria y cómo hacer sus tests.

Después hemos creado unos repositorios y hemos probado los repositorios que interactuaban con una base de datos en memoria con SQLite.

Ahora vamos a añadir al servicio una dependencia de UsuarioRepository a través de un nuevo atributo.

PrestamoServiceConRepo

Para evitar confundirnos, vamos a crear una copia de PrestamoService en la misma ubicación (conservaremos el fichero PrestamoService) y la vamos a llamar PrestamoServiceConRepo que tendrá una dependencia de UsuarioRepository.

Cambiaremos el uso del array $usuarios por el uso de UsuarioRepository:

1- Creamos un atributo privado UsuarioRepository $usuarioRepository;

2- Lo añadimos también en el constructor del Servicio

3- Modificamos el método prestar para que haga uso de $this->usuarioRepository

4- Modificamos el método registrarUsuario para que haga uso de $this->usuarioRepository

```php
namespace App\Service;

use App\Model\Biblioteca\Enum\EstadoRecurso;
use App\Model\Biblioteca\Prestamo;
use App\Model\Biblioteca\Usuario;
use App\Model\Biblioteca\Recurso;
use App\Repository\UsuarioRepository;
use App\Service\Traits\Logger;

  class PrestamoServiceConRepo
{

    //Creamos una propiedad del tipo repository
    private UsuarioRepository $usuarioRepository;

   //Comentamos el array de $usuarios que estábamos utilizando:
   // private array $usuarios;

    //Establecemos en el constructor la propiedad
    public function __construct(UsuarioRepository $usuarioRepository)
    {
        $this->usuarioRepository = $usuarioRepository;
    }


    public function prestar(string $nombreUsuario, int $idRecurso): Prestamo
    {

       // if (!isset($this->usuarios[$nombreUsuario])) {
       $usuario = $this->usuarioRepository->findByEmail($nombreUsuario);

        if ($usuario===null){
            $this->log("Usuario no encontrado: " . $nombreUsuario, "ERROR", "app.log");
            throw new \Exception("Usuario no encontrado");
        }

        if (!isset($this->recursos[$idRecurso])) {
            $this->log("Recurso no encontrado: " . $idRecurso, "ERROR", "app.log");
            throw new \Exception("Recurso no encontrado");
        }
        //Si llegamos aquí, es que el usuario y el recurso existen
        //$usuario = $this->usuarios[$nombreUsuario];
        $recurso = $this->recursos[$idRecurso];

        if (!$recurso->isDisponible()) {
            $this->log("Recurso no disponible: " . $idRecurso, "ERROR", "app.log");
            throw new \Exception("Recurso no disponible");
        }
        if (count($usuario->getPrestamos()) >= self::MAX_PRESTAMOS) {
            $this->log("Usuario ha alcanzado el máximo de préstamos: " . $nombreUsuario, "ERROR", "app.log");
            throw new \Exception("El usuario ha alcanzado el máximo de préstamos");
        }
        //Realizar el préstamo

        $prestamo = new Prestamo($recurso, $usuario);
        $recurso->setEstado(EstadoRecurso::PRESTADO);
        $usuario->addPrestamo($prestamo);
        return $prestamo;
    }

     public function registrarUsuario(Usuario $usuario): ?Usuario
    {
       //$this->usuarios[$usuario->getEmail()] = $usuario;
       return $this->usuarioRepository->create($usuario);
    }


   public function getUsuarioByEmail(string $email):?Usuario{
        // if(isset($this->usuarios[$email])){
        //     return $this->usuarios[$email];
        // }
        // return null;
        return $this->usuarioRepository->findByEmail($email);
```

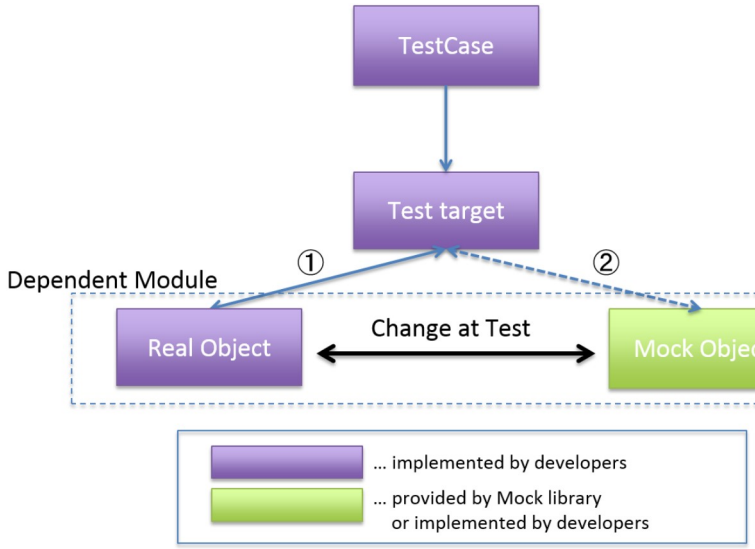
```
    }
    // resto de la lógica sigue igual

}
```

27.2. Pruebas con Mocks y PHPUnit

¿Cómo probar un servicio que depende un repositorio?

Ahora nuestro Servicio PrestamoServiceConRepo tiene una dependencia (un atributo) de tipo UsuarioRepository y tendremos que cambiar nuestros tests.

El nuevo test debe enfocarse en probar la funcionalidad del servicio (Test Target), no en la del repositorio. Como el nuevo servicio tiene una dependencia de UsuarioRepository(Real Object en la imagen), vamos a cambiarlo por un Mock Object en el nuevo test.



¿Qué es un Mock?

Un ***mock*** es un objeto simulado utilizado en **pruebas unitarias** que **imita el comportamiento de un objeto real,** permitiendo aislar la lógica a probar y comprobar interacciones sin depender de recursos externos. Un servicio va a tener una dependencia de un repositorio. Si queremos probar el servicio, crearemos un objeto *Mock* que imite el comportamiento del Repositorio y así podremos probar la lógica del Servicio.

¿Cómo trabajar con Mocks en PHPUnit?

PHPUnit permite crear **mocks** gracias a la clase base `PHPUnit\Framework\TestCase`. Cuando una clase de test extiende `TestCase`, obtiene métodos para crear dobles de prueba (mocks).

1 `createMock()`

`createMock()` es un método que proporciona `TestCase`. Sirve para crear un objeto falso que simula una clase o interfaz real.

```
$usuarioRepositoryMock = $this->createMock(UsuarioRepository::class);
```

Lo que hace PHPUnit internamente es:

- Crea una clase anónima que implementa `UsuarioRepository`
- No ejecuta el código real
- Permite definir qué deben devolver sus métodos

> "Este objeto parece un UsuarioRepository, pero no lo es"

2 `method()`

`method()` se usa para indicar **qué método del mock** queremos configurar.

```
$usuarioRepositoryMock
    ->method('findByEmail');
```

Esto significa:

> "Cuando alguien llame al método `findByEmail()` de este mock..."

⚠ Importante:

- El nombre del método se pasa como **string**
- Ese método **debe existir** en la clase mockeada

---

3 `willReturn()`

`willReturn()` define **qué valor devolverá el método** configurado con `method()`.

```
$usuarioRepositoryMock
    ->method('findByEmail')
    ->willReturn($usuario);
```

Se puede leer como:

> "Cada vez que se llame a `findByEmail()`, devuelve este usuario"

---

4 Otros métodos importantes en mocks

4.1 `willThrowException()`

Hace que el método del mock lance una excepción.

```
$repoMock
    ->method('findByEmail')
    ->willThrowException(new Exception("Error de base de datos"));
```

Ideal para simular fallos externos.

4.2 `with()`

Permite indicar **con qué argumentos** debe ser llamado el método. `with()` y `willReturn()` **pueden intercambiarse**, aunque lo más legible es poner `with()` primero y luego `willReturn()`.

```
$repoMock
    ->method('findByEmail')
    ->with('juan@example.com')
    ->willReturn($usuario);
```

Si se llama al método con otro argumento, el test fallará.

4.3 `expects()`

Verifica cuántas veces debe llamarse un método. `expects()` **debe ir antes de** `method()`:

```
$repoMock
    ->expects($this->once())
    ->method('findByEmail')
    ->willReturn($usuario);
```

Otros ejemplos:

- `$this->never()` : Útil cuando queremos asegurarnos de que un método no se llama (si hay errores, no queremos que se llame al método create() y se guarde en la base de datos).
- `$this->exactly(2)`: Útil en bucles: Cuando queremos asegurarnos de que se llama a create 2 veces para que se guarden exactamente 2 objetos.

Diferencia entre Mock y Stub en PHPUnit

En PHPUnit, **mock** y **stub** son *roles conceptuales*, no tipos de objetos distintos. Se crean usando el mismo método: `$this->createMock()`.

1. Stub

Un **stub** sirve para *proporcionar datos controlados* al test. No verifica si se llamó al método, solo importa lo que devuelve.

```
$repo = $this->createMock(UserRepository::class);

// Stub: devuelve un usuario sin importar cuántas veces se llama
$repo->method('find')
     ->willReturn(new Usuario('Ana', 'ana@edu.com'));
```

En este caso, `find()` siempre devuelve un **Usuario**, pero el test no falla si no se llama.

---

2. Mock

Un **mock** sirve para *verificar interacciones y comportamiento*. Comprueba si un método se llama, cuántas veces y con qué parámetros.

```
$repo = $this->createMock(UserRepository::class);

// Mock: espera que se llame exactamente 1 vez con un Usuario
$repo->expects($this->once())
     ->method('save')
// asegura que se llama con un argumento que es de la clase Usuario
     ->with($this->isInstanceOf(Usuario::class));
```

Si `save()` no se llama, o se llama más de una vez, el test falla.

Nota importante

- En PHPUnit **ambos se crean igual** con `createMock()`.
- La diferencia está en *cómo configurar el objeto*: `method()->willReturn()` = stub, `expects()` = mock.
- Un mismo objeto puede ser stub y mock al mismo tiempo, según el método utilizado.