Guía de Buenas Prácticas - Proyecto PetBook

Документ составлен Сэром Петрухой.
02/07/2025

Introducción

La elaboración de una guía de buenas prácticas constituye un pilar fundamental en el desarrollo de cualquier proyecto de software, incluso en el ámbito académico. Establecer criterios claros y consensuados para la escritura de código, la organización de archivos, la gestión del trabajo colaborativo y la documentación permite no solo mejorar la calidad técnica del sistema, sino también fomentar la comprensión, la mantenibilidad y la eficiencia en el proceso de desarrollo.

Este documento tiene como objetivo definir lineamientos comunes que sirvan de referencia a todos los integrantes del equipo, promoviendo la coherencia, la trazabilidad y la responsabilidad compartida en cada una de las etapas del proyecto.

En un entorno donde el trabajo colectivo y la evolución constante del software son la norma, contar con una base sólida de prácticas estandarizadas resulta no solo recomendable, sino esencial.

1. Convenciones de Nomenclatura

El código debe ser legible como una carta bien escrita: clara, precisa, sin rodeos ni jeroglíficos. Una buena nomenclatura permite que incluso quien no escribió el código pueda leerlo como si lo hubiera hecho.

En programación, una nomenclatura de código se refiere a las convenciones o reglas que se siguen para nombrar elementos del código, como variables, funciones, clases, constantes, archivos, etc.

Principales tipos de nomenclatura utilizados en programación

CamelCase

- **Ejemplo:** calcularTotal, nombreUsuario
- **Uso común:** variables, funciones, métodos
- **Descripción:** La primera palabra comienza en minúscula y las siguientes con mayúscula.

PascalCase

- **Ejemplo:** CalcularTotal, NombreUsuario
- **Uso común:** nombres de clases, interfaces
- **Descripción:** Todas las palabras comienzan con mayúscula. Es similar a CamelCase, pero incluye la primera palabra en mayúscula.

snake_case

- **Ejemplo:** calcular_total, nombre_usuario
- **Uso común:** variables y funciones
- **Descripción:** Las palabras se separan mediante guiones bajos. Es una convención muy legible y común en scripts.

kebab-case

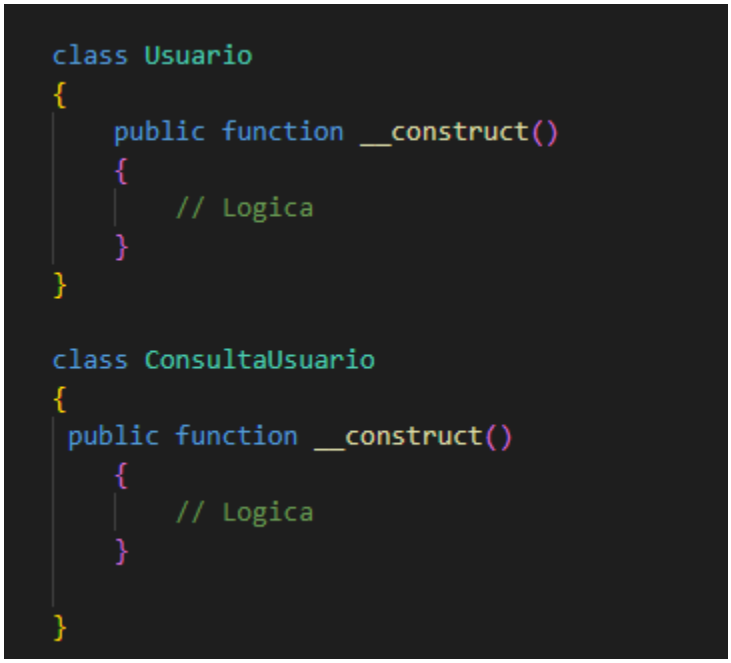
- **Ejemplo:** calcular-total, nombre-usuario
- **Uso común:** nombres de archivos, URLs, hojas de estilo (CSS)
- **Descripción:** Las palabras se separan mediante guiones. No se utiliza en la mayoría de los lenguajes de programación porque el guion se interpreta como operador de resta.

SCREAMING_SNAKE_CASE

- **Ejemplo:** MAX_INTENTOS, API_KEY
- **Uso común:** constantes
- **Descripción:** Es una variante de snake_case en mayúsculas. Se utiliza comúnmente para representar valores inmutables.

1.1 Clases (PHP)

- Formato: PascalCase
- Ejemplos: Usuario, ConsultarUsuario, ControladorMascota



```php
class Usuario
{
    public function __construct()
    {
        // Logica
    }
}

class ConsultaUsuario
{
 public function __construct()
    {
        // Logica
    }

}
```

1.2 Variables (PHP / JS)

- Formato: camelCase
- Ejemplos válidos: $nombreUsuario, $fechaNacimiento, let listaPublicaciones = [];

```php
<?php
$nombreUsuario;
$fechaNacimiento;
$esActivo;
?>

<script>
let nombreUsuario = $('#inputNombre').val();
const fechaActual = new Date();
</script>
```

1.3 Funciones y Métodos

- Formato: camelCase
- Prefijos comunes: obtener, listar, guardar, eliminar, validar, procesar

```php
public function guardarNombre(string $nuevoNombre): void
{
    $this->nombre = $nuevoNombre;
}

public function obtenerDatos(): array
{
    return [
        'nombre' => $this->nombre,
        'email' => $this->email
    ];
}
```

1.4 Constantes

- Formato: MAYUSCULAS_CON_GUION_BAJO

```php
<?php

define('TIEMPO_EXPIRACION_SESION', 3600);

const ESTADO_ACTIVO = 'activo';


?>
```

1.5 Tablas en Base de Datos

- Formato: PascalCase, singular
- Ejemplos: Usuario, Publicacion, Mascota

1.6 Campos de Tabla

- Formato: camelCase o snake_case (usar uno y mantenerlo)
- Ejemplos: usuarioId, fechaAlta, nombreMascota

1.7 Archivos

- PHP: NombreClase.php, funcionesValidacion.php
- JS: validacionFormulario.js, cargaDatos.js
- HTML: formulario-registro.html, vistaPublicaciones.html

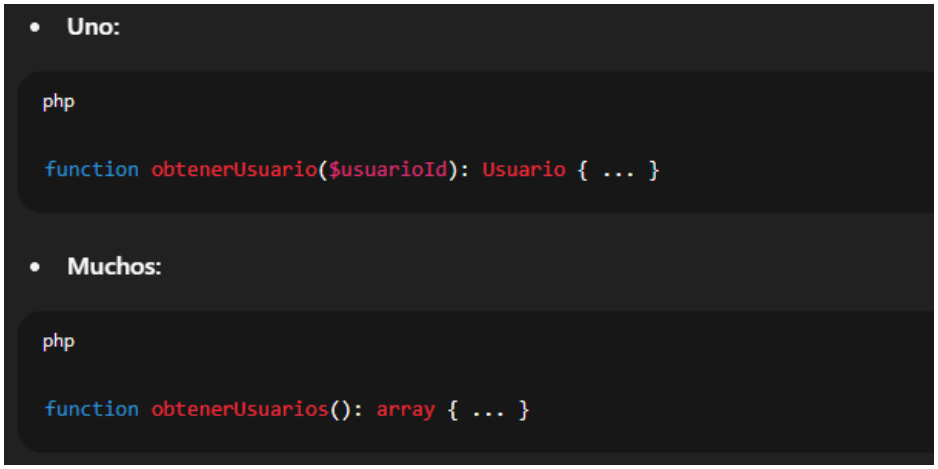
1.8 Representatividad Semántica

- El nombre debe describir el contenido o acción.
- Correcto: $publicacionExtraviadaId, $listaUsuariosActivos

```php
<?php

//Ejemplo correcto
function eliminarPublicacionUsuario($publicacionId) { ... }


//Ejemplo incorrecto
function eliminarPerdido($id) { ... } // ¿Qué se elimina? ¿De dónde?

?>
```

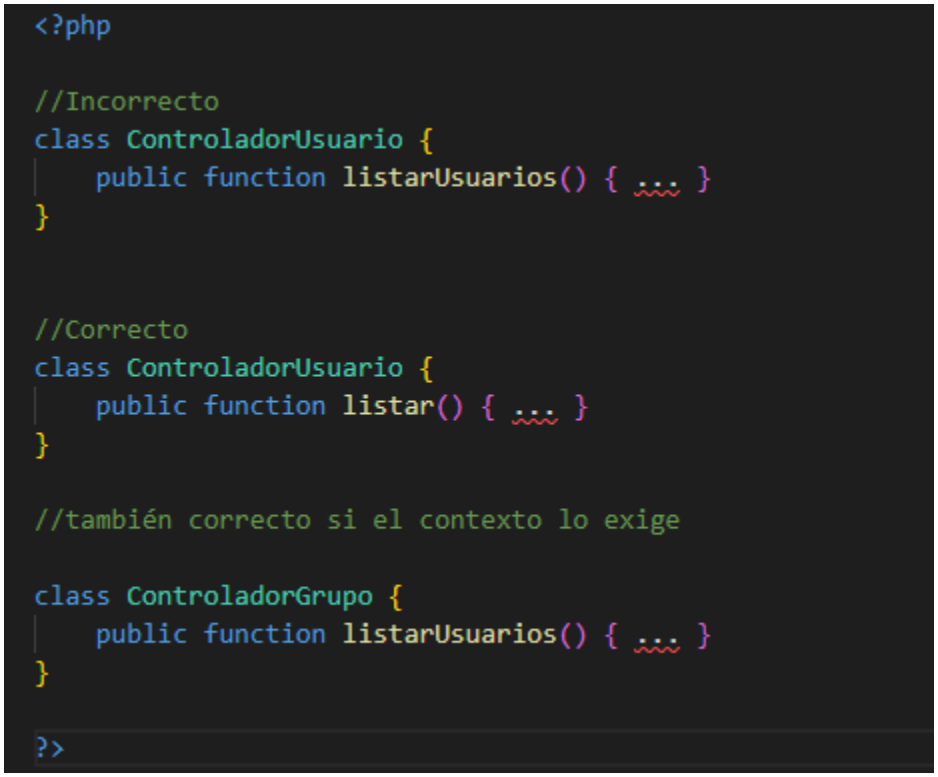
1.9 Pluralidad Semántica

- Usar nombres en plural si manejan múltiples entidades: obtenerUsuarios()
- En singular, si representan uno solo: obtenerUsuario($usuarioId)



1.10 Redundancia de nombres

- Evitar repetir el nombre de la entidad dentro de sus propios métodos.
- Incorrecto: Usuario::listarUsuarios();
- Correcto: Usuario::listar(); Grupo::listarUsuarios();



1.11 Encapsulamiento de lógica compleja

- Extraer fragmentos complejos en funciones auxiliares con nombres claros y específicos.
- Ejemplo: generarEstadisticaMensual() → obtenerDatosDelMes() → calcularTotales() → mostrarResumen()

```php
<?php

// Sin encapsulamiento
class Factura
{
    public function calcularTotal(float $subtotal): float
    {
        $impuesto = $subtotal * 0.21;
        $descuento = $subtotal > 1000 ? $subtotal * 0.1 : 0;
        return $subtotal + $impuesto - $descuento;
    }
}

//Con encapsulamiento de lógica
class Factura
{
    public function calcularTotal(float $subtotal): float
    {
        return $subtotal + $this->calcularImpuesto($subtotal) - $this->calcularDescuento($subtotal);
    }

    private function calcularImpuesto(float $monto): float
    {
        return $monto * 0.21;
    }

    private function calcularDescuento(float $monto): float
    {
        return $monto > 1000 ? $monto * 0.1 : 0;
    }
}



?>
```

1.12 Alineamiento visual

- Alinear las asignaciones para facilitar el escaneo.

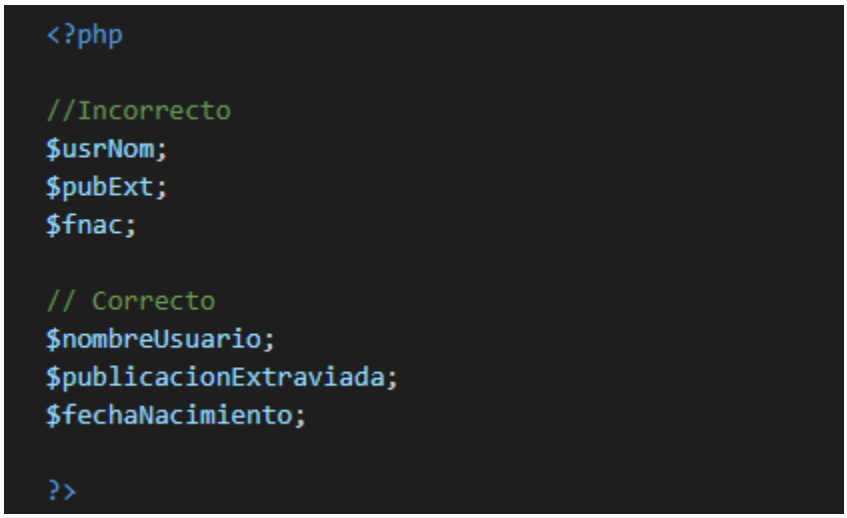
```php
<?php

//Incorrecto
$nombre = 'Ana';
$apellido = 'Pérez';
$correo = 'ana@mail.com';

// Correcto

$nombre    = 'Ana';
$apellido  = 'Pérez';
$correo    = 'ana@mail.com';

?>
<html>

    <input type="text" id="nombre"    name="nombre">
    <input type="text" id="apellido" name="apellido">

</html>
```

```php
// ❌ Desordenado
$datos = [
    "nombre" => "Ana",
    "apellidoLargo" => "Fernández",
    "edad" => 30
];

// ✅ Alineado
$datos = [
    "nombre"         => "Ana",
    "apellidoLargo"  => "Fernández",
    "edad"           => 30
];
```

1.13 Evitar abreviaturas

El uso de abreviaturas reduce la claridad del código, dificulta su comprensión por parte de otros desarrolladores y puede generar ambigüedad. Emplear nombres completos y descriptivos favorece la legibilidad, facilita el mantenimiento y mejora la colaboración en entornos de desarrollo profesional.

- Evitar abreviaturas salvo que sean estándar (URL, ID).
- Incorrecto: $usrNom; Correcto: $nombreUsuario

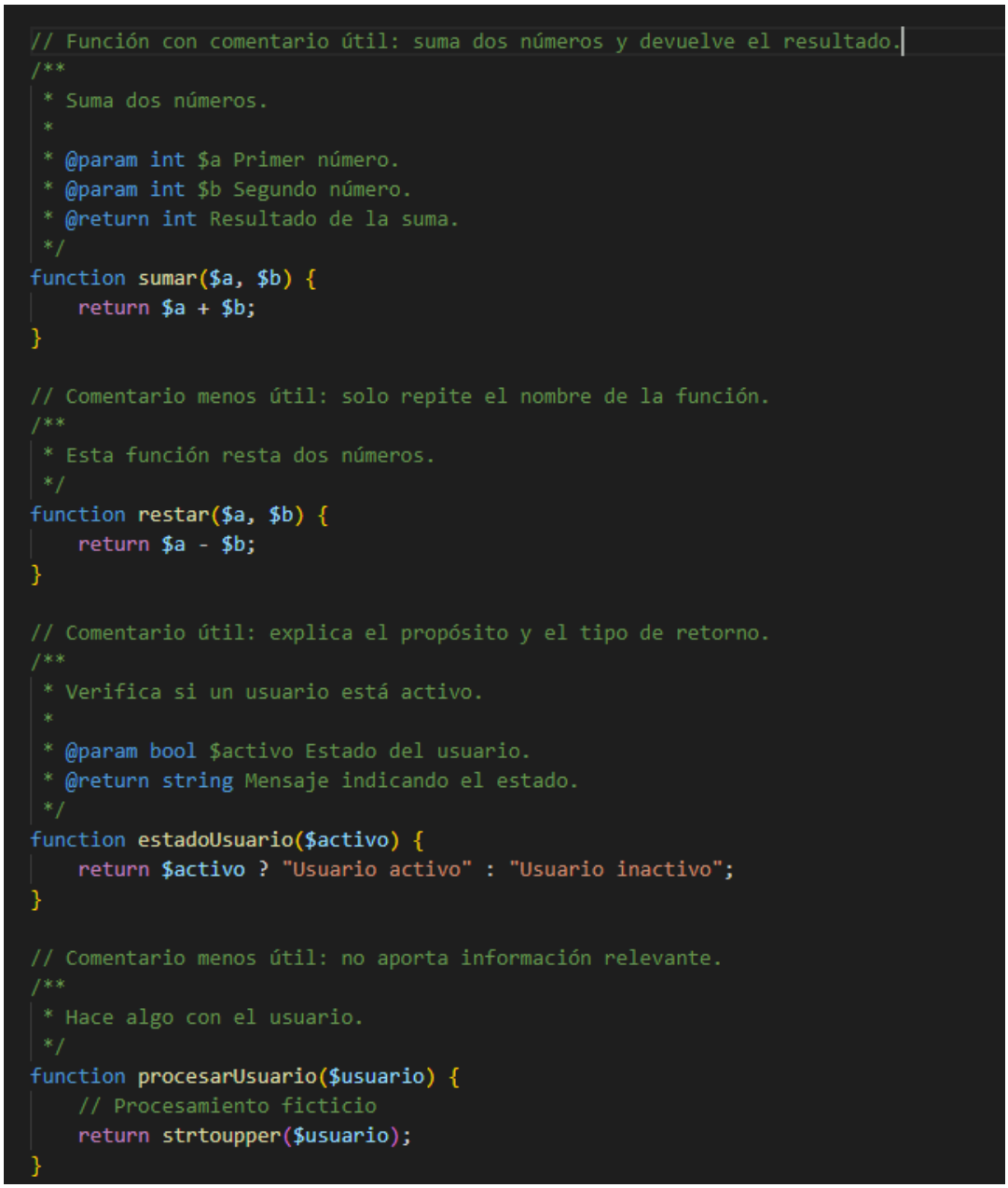


```php
<?php

//Incorrecto
$usrNom;
$pubExt;
$fnac;

// Correcto
$nombreUsuario;
$publicacionExtraviada;
$fechaNacimiento;

?>
```

1.14 Comentarios útiles y justificados

Comentar código de manera eficiente implica escribir anotaciones que realmente aporten valor y claridad al lector del código.

- No comentar obviedades.
- Comentar algoritmos, excepciones y decisiones no evidentes.
- Usar estilo de documentación si es posible.



```php
// Función con comentario útil: suma dos números y devuelve el resultado.
/**
 * Suma dos números.
 *
 * @param int $a Primer número.
 * @param int $b Segundo número.
 * @return int Resultado de la suma.
 */
function sumar($a, $b) {
    return $a + $b;
}

// Comentario menos útil: solo repite el nombre de la función.
/**
 * Esta función resta dos números.
 */
function restar($a, $b) {
    return $a - $b;
}

// Comentario útil: explica el propósito y el tipo de retorno.
/**
 * Verifica si un usuario está activo.
 *
 * @param bool $activo Estado del usuario.
 * @return string Mensaje indicando el estado.
 */
function estadoUsuario($activo) {
    return $activo ? "Usuario activo" : "Usuario inactivo";
}

// Comentario menos útil: no aporta información relevante.
/**
 * Hace algo con el usuario.
 */
function procesarUsuario($usuario) {
    // Procesamiento ficticio
    return strtoupper($usuario);
}
```

2. Estructura del Proyecto

Una estructura clara del proyecto permite que cualquier integrante del equipo pueda ubicarse rápidamente dentro del código, encontrar los archivos correctos y respetar la separación de responsabilidades.

2.1 Estructura Sugerida (PHP + HTML + jQuery)

```
/PetBook
│
├── /app
│   ├── /Controllers     # Lógica del servidor (PHP)
│   ├── /Models          # Clases que representan entidades y acceden a la BD
│   ├── /Views           # Vistas HTML principales
│   └── /Helpers         # Funciones utilitarias y validaciones
│
├── /public
│   ├── /css             # Hojas de estilo
│   ├── /js              # Scripts JS/jQuery
│   ├── /img             # Imágenes estáticas
│   └── index.php        # Punto de entrada
│
├── /database
│   ├── /migrations      # Scripts para crear/modificar la BD
│   └── conexion.php     # Conexión a base de datos
│
├── /routes              # Archivo o carpeta de enrutamiento (si aplica)
│
├── /tests               # Pruebas manuales o automáticas
│
├── README.md            # Guía del proyecto
└── .gitignore           # Ignorar archivos temporales/cache
```

2.2 Separación de Capas

- Modelo: Clases que representan entidades y acceso a datos (Ej: UsuarioModel.php)
- Vista: Archivos HTML que muestran los datos (Ej: registro.html)
- Controlador: Coordina entrada del usuario, vista y modelo (Ej: UsuarioController.php)

2.3 Organización de scripts

- Dividir scripts JS por funcionalidad: validacionesFormulario.js, ajaxUsuarios.js
- Separar CSS general de específico: main.css, formulario.css

- Evitar mezclar lógica PHP con HTML salvo en includes parciales (header.php, footer.php)

2.4 Rutas y Enrutamiento

- Definir rutas en archivos separados si se usa routing (ej: routes.php)
- Mantener claridad entre URLs y controladores que responden

2.5 Archivos de configuración

- Usar archivos separados para configuración (config.php, .env)
- Nunca incluir claves sensibles directamente en el repositorio

2.6 Guías de carpetas compartidas

- Todos deben conocer y respetar la estructura
- Tener un README o guía rápida del proyecto dentro del repositorio

2.7 Orden dentro de los archivos

- En PHP: propiedades → constructor → métodos públicos → privados
- En JS: variables → funciones → listeners → inicializadores
- En HTML: usar etiquetas semánticas (<header>, <main>, <section>, <footer>)

3. Estilo de Código

Más allá de que el código funcione, debe poder leerse, mantenerse y comprenderse fácilmente. Un buen estilo reduce errores, facilita el trabajo en equipo y eleva la calidad del proyecto.

3.1 Sangrías y Tabulaciones

- Usar 4 espacios por nivel de indentación.
- Evitar mezclar tabulaciones con espacios.
- Los bloques siempre deben estar correctamente indentados.

```php
//Correcto
if ($usuario->esActivo()) {
    enviarNotificacion($usuario);
}

//Incorrecto
if ($usuario->esActivo())
enviarNotificacion($usuario);
```

3.2 Longitud de líneas

- Limitar a máximo 100 caracteres por línea (ideal: 80).
- Si una línea se extiende demasiado, dividirla de forma legible.

3.3 Uso de llaves {}

- Siempre usar llaves, incluso si el bloque tiene una sola línea.
- Aumenta la claridad y previene errores futuros.

```php
//Correcto
if ($esAdmin) {
    mostrarPanelAdmin();
}

// Incorrecto
if ($esAdmin) mostrarPanelAdmin(); // ¡peligroso!
```

3.4 Espaciado y saltos de línea

- Dejar una línea en blanco entre bloques lógicos.
- Separar operadores y argumentos con espacios.

```php
$precioConDescuento = $producto->precio - $producto->descuento;

if ($precioConDescuento < 100) {
    mostrarOfertaEspecial();
}
```

3.5 Uso de espacios en funciones y estructuras

- Correcto: if ($x == 3) { ... }
- Incorrecto: if($x==3){ ... }

```php
//Correcto
if( $x==3){}

//Incorrecto
if ($x == 3) {}
```

3.6 Encabezado de archivo

- Incluir encabezado informativo opcional con autor, fecha y propósito del archivo.

3.8 Código muerto o comentado

- Eliminar cualquier bloque de código comentado que no será reutilizado.

3.9 Estandarización del idioma

- Todo el código debe estar en un solo idioma (preferentemente español neutro).
- Correcto: $usuarioId, $publicacionActiva
- Incorrecto: $userId, $activaPublication