PROMETEO

PROGRAMACIÓN

Principios SOLID

¿Qué es SOLID?

SOLID es un conjunto de 5 principios de diseño orientado a objetos que nos ayudan a:

- Crear código más mantenible
- Reducir acoplamiento
- Mejorar la escalabilidad
- Facilitar pruebas
- Preparar el proyecto para crecer (por ejemplo, pasar de consola a API)

Fue formulado por:

- Robert C. Martin
  (también conocido como *Uncle Bob*)

¿Qúe significa SOLID?

- **S** → Single Responsibility Principle
- **O** → Open/Closed Principle
- **L** → Liskov Substitution Principle
- **I** → Interface Segregation Principle
- **D** → Dependency Inversion Principle

S: Single Responsibility Principle

Definición

★ Una clase debe tener una única razón para cambiar.

También formulado como:

★ Una clase debe tener una sola responsabilidad.

¿Qué significa realmente?

Responsabilidad ≠ Método
Responsabilidad = Motivo de cambio

Ejemplo correcto en tu MVC

- `LibroService` → Lógica de negocio
- `LibroRepository` → Persistencia
- `LibroController` → Comunicación con usuario
- `AppLibro` → Arranque y menú

O: Open/Closed Principle

Definición

Las clases deben estar abiertas a extensión, pero cerradas a modificación.

¿Qué significa realmente “abierta” y “cerrada”?

- **Abierta a extensión**

  El comportamiento del sistema puede ampliarse.

  Se pueden añadir nuevas funcionalidades.

- **Cerrada a modificación**

  El código existente no debe modificarse para introducir esos nuevos comportamientos.

  No se toca lo que ya funciona.

O: Open/Closed Principle

La idea central del OCP

El principio busca evitar algo muy común:

> Cada vez que se añade una nueva funcionalidad, se modifica código ya existente.

Eso provoca:

- Riesgo de romper comportamientos previos.
- Introducción de errores.
- Incremento del acoplamiento.
- Dependencia excesiva de estructuras rígidas.

L: Liskov Substitution Principle

Definición

> Los objetos de una subclase deben poder sustituir a los de su clase base sin alterar el funcionamiento del programa.

Propuesto por:

- Barbara Liskov

Traducción práctica

Si una clase implementa una interfaz, debe cumplir el contrato completamente.

L: Liskov Substitution Principle

¿Qué significa realmente “sustituir”?

Si un sistema está diseñado para trabajar con un tipo base:

Cualquier implementación derivada debe:

- Mantener el comportamiento esperado.
- Respetar las reglas establecidas.
- No introducir efectos inesperados.

LSP no trata solo de herencia, trata sobre contratos, comportamiento, expectativas y sustituibilidad semántica. NO es solo una cuestión de compilación, es una cuestión de coherencia lógica.

I: Interface Segregation Principle

Definición

Ninguna clase debe estar obligada a implementar métodos que no usa.

Interfaz incorrecta:

```
public interface LibroService {
    void guardar();
    void imprimir();
    void exportarPDF();
    void conectarAPI();
}
```

No todas las implementaciones necesitan todo eso. Las interfaces tienen que ser pequeñas y específicas

I: Interface Segregation Principle

¿Qué significa realmente?

Cuando una abstracción (normalmente una interfaz) define demasiadas operaciones:

- Obliga a las implementaciones a depender de cosas que no necesitan.
- Genera acoplamiento innecesario.
- Introduce fragilidad en el sistema.

ISP propone:

> Interfaces pequeñas, específicas y cohesionadas.

El problema de las interfaces “grandes” es que suelen agrupar responsabilidades distintas, mezclar comportamientos y obligar a implementar métodos irrelevantes. En general, lo que se conoce como “**Fat Interface”.**

D: Dependency Inversion Principle

Definición

> Los módulos de alto nivel no deben depender de módulos de bajo nivel.
> Ambos deben depender de abstracciones.

Normalmente, en un diseño tradicional:

Clase A → Clase B → Clase C

La clase superior depende directamente de la inferior.

Con DIP ocurre algo diferente. Se invierte la dirección de la dependencia:

Clase A → Interfaz Clase B → Interfaz

Las clases concretas dependen de algo abstracto. Por eso se llama inversión de dependencias.

D: Dependency Inversion Principle

¿Qué son módulos de alto y bajo nivel?

🔹 **Módulos de alto nivel**

- Contienen la lógica principal del sistema.
- Representan reglas de negocio.
- Coordinan el comportamiento general.

Son los más importantes.

🔹 **Módulos de bajo nivel**

- Se encargan de detalles técnicos.
- Acceso a base de datos.
- Lectura de archivos.
- Envío de correos.
- Conexión con APIs externas.

Son implementaciones concretas.

Beneficios globales de SOLID

- Código más limpio
- Menos acoplamiento
- Más reutilizable
- Más profesional
- Fácil de testear
- Fácil de convertir en API

SOLID no es teoría avanzada.

Es:

> Escribir código hoy pensando en el yo del futuro.


PROMETEO