

# SESIÓN 13: PRINCIPIOS SOLID, COHESIÓN Y ACOPLAMIENTO

## Introducción

A estas alturas, el objetivo es **aprender a programar bien**.

En el mundo real, los programas:

- No los hace una sola persona
- No duran dos semanas
- Cambian constantemente

Por eso, el mayor problema del software **no es escribir código**, sino **mantenerlo con el paso del tiempo**.

Los principios **SOLID** aparecen como respuesta a este problema.

No son teoría abstracta: son el resultado de **errores reales cometidos durante años en proyectos reales**.

## El problema real del software

Imaginemos un programa muy típico:

- **Una tienda online**

Al principio el programa es pequeño:

- Pocos usuarios
- Un método de pago
- Un correo sencillo

El programador inicial suele pensar: "Voy a hacerlo rápido, luego ya lo ordenaré".

El problema es que **ese luego casi nunca llega**.

Con el tiempo:

- Se añaden más métodos de pago
- Cambian las leyes (facturación, protección de datos)
- Aparecen nuevos tipos de usuario
- Si el código está mal organizado:
- Cada cambio afecta a muchas partes
- Se introducen errores sin darse cuenta
- El proyecto se vuelve frágil

**SOLID nace para evitar que un programa funcione hoy pero sea imposible de tocar mañana.**

## Qué es SOLID

Antes de ver las letras, quédate con esta idea:

**SOLID es una forma de organizar el programa para que cambiarlo no sea una pesadilla.**

SOLID no dice cómo escribir un if o un for.

- Dice **cómo repartir las responsabilidades entre las clases**.

Cada principio SOLID responde a una pregunta distinta:

- ¿Quién se encarga de qué?
- ¿Dónde hago los cambios?
- ¿Qué pasa si mañana cambio una tecnología?

## S. Single Responsibility Principle (Responsabilidad Única)

### Qué problema intenta resolver

Cuando una clase hace muchas cosas, cualquier cambio en una de ellas puede romper las demás. Esto provoca miedo a modificar el código.

### Definición

Una clase debe tener una única responsabilidad, es decir, una única razón para cambiar.

### Explicación

No significa que una clase solo tenga un método.

Significa que **todo lo que hace está relacionado con una misma tarea**.

### Ejemplo de la vida real

En una empresa, no mezclamos:

- Contabilidad
- Atención al cliente
- Marketing

¿Por qué?

Porque cada área cambia por motivos distintos.

Si mezclamos todo, cualquier cambio afecta a todo.

## Ejemplo en la tienda online

Supongamos una clase que:

- Valida usuarios
- Guarda usuarios
- Envía correos

**Pregunta clave: ¿Por qué puede cambiar esta clase?**

- Cambia la validación
- Cambia la base de datos
- Cambia el sistema de correo

Tiene **demasiados motivos para cambiar**, por eso está mal diseñada.

## Diseño correcto

Separar en:

- Gestión de usuarios
- Persistencia de usuarios
- Envío de correos

Ahora, si cambia el correo:

- Solo cambia una clase
- El resto del sistema ni se entera

## Por qué esto es importante

Porque en proyectos reales **los cambios son constantes**. SRP reduce el impacto de cada cambio.

## O – Open / Closed Principle (Abierto / Cerrado)

### Qué problema intenta resolver

Cada vez que modificamos código que ya funciona, existe el riesgo de introducir errores nuevos.

### Definición

El software debe estar abierto a extensión, pero cerrado a modificación.

### Explicación detallada

No se trata de no tocar nunca el código, sino de **evitar modificar código estable cuando añadimos nuevas funcionalidades**.

### Ejemplo

Una ruleta eléctrica está diseñada para:

- Permitir nuevos enchufes
- Sin romper los existentes

Eso es buen diseño.

### Ejemplo en la tienda online

Imagina que cada vez que añades un método de pago:

- Tienes que modificar una clase enorme

Con el tiempo:

- Esa clase se llena de if
- Es difícil de entender
- Cualquier cambio rompe algo

### Buen diseño

Si cada método de pago es una clase independiente:

- Añadir uno nuevo no afecta a los anteriores
- El código existente permanece estable

### Por qué es clave en el mundo real

Las empresas cambian constantemente sus servicios. Un sistema que no cumple OCP **no escala**.

## L – Liskov Substitution Principle

### Qué problema intenta resolver

Muchas herencias se crean solo para reutilizar código, sin pensar en el comportamiento.

### Definición

Una clase hija debe poder sustituir a su clase padre sin alterar el funcionamiento del programa.

### Explicación

Cuando usamos herencia, estamos haciendo una promesa: "Esto se puede usar igual que su padre".

Si no se cumple, el diseño está roto.

### Ejemplo

Si un método espera un Vehículo:

- Asume que puede moverse

Si le pasas algo que hereda de Vehiculo pero no se mueve:

- El programa falla

### Por qué es importante

Frameworks y librerías confían en estas promesas.

Romperlas genera errores difíciles de detectar.

## I – Interface Segregation Principle

### Qué problema intenta resolver

Interfaces demasiado grandes obligan a implementar métodos que no se necesitan.

### Definición

Ninguna clase debería depender de métodos que no utiliza.

### Explicación

Una interfaz representa un contrato.

Si el contrato es enorme, muchas clases se ven obligadas a cumplir cosas que no les corresponden.

### Ejemplo

Un mando a distancia con 50 botones:

- La mayoría no se usan
- Complica su uso

### Buen diseño

Interfaces pequeñas y específicas:

- Cada clase implementa solo lo necesario

### Por qué importa

- Reduce dependencias y hace el sistema más flexible.

## D – Dependency Inversion Principle (el más importante)

### Qué problema intenta resolver

Cuando una clase depende directamente de otra concreta, cualquier cambio en esa dependencia afecta a todo el sistema.

### Definición clara

Los módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones.

### Explicación detallada

No debemos programar pensando en una tecnología concreta, sino en **qué hace esa tecnología**.

### Ejemplo razonado

Si todo tu sistema depende de MySQL:

- Cambiar de base de datos implica reescribir código

Si depende de una interfaz:

- Cambiar la implementación no rompe nada

### Por qué es clave

Las tecnologías cambian, el negocio no debería romperse por ello.

## Cohesión

La cohesión indica **lo bien relacionadas que están las responsabilidades dentro de una clase**.

Una clase con alta cohesión:

- Tiene un objetivo claro
- Todo lo que hace está relacionado

Una clase con baja cohesión:

- Hace muchas cosas distintas
- Es difícil de entender

La alta cohesión hace el código más fácil de mantener.

## Acoplamiento

El acoplamiento indica **cuánto afecta un cambio en una clase al resto del sistema**.

Alto acoplamiento:

- Cambios pequeños: muchos errores
- Bajo acoplamiento: Cambios localizados

El buen diseño busca siempre **bajo acoplamiento**.

## Relación entre SOLID, cohesión y acoplamiento

Los principios SOLID existen para:

- Aumentar la cohesión
- Reducir el acoplamiento

Por eso se consideran la base del código limpio.

## Caso real profesional: Shopify

Shopify gestiona millones de transacciones diarias.

Si su código estuviera mal diseñado:

- No podrían añadir funcionalidades
- Cada cambio sería un riesgo enorme

Aplicando SOLID:

- Separan responsabilidades
- Aíslan dependencias
- Escalan sin romper el sistema

## Resumen

Programar bien no es escribir más código, sino escribir código que pueda cambiar sin romperse.

**SOLID no es una moda: es supervivencia del software.**

SOLID no son cinco reglas independientes. Es una **forma de pensar el diseño del software**.

Todos los principios apuntan a lo mismo:

- Evitar clases gigantes
- Evitar dependencias rígidas
- Evitar miedo a cambiar el código

## SOLID visto como un todo

Vamos a unirlo todo:

- **S (Responsabilidad única)**: evita clases que hacen de todo
  - mejora la cohesión
- **O (Abierto/Cerrado)**: evita tocar código estable
  - reduce errores
- **L (Liskov)**: evita herencias incorrectas
  - garantiza comportamiento coherente
- **I (Segregación)**: evita interfaces enormes
  - reduce dependencias innecesarias
- **D (Inversión)**: evita depender de tecnologías concretas
  - reduce acoplamiento

Cuando aplicas SOLID correctamente:

- Las clases son pequeñas
- Los cambios están localizados
- El sistema es flexible

## Relación final entre SOLID, cohesión y acoplamiento

Ahora podemos entender bien estos dos conceptos:

### Cohesión

- Una clase con **una sola responsabilidad**
  - alta cohesión
- Clases con responsabilidades mezcladas
  - baja cohesión

**SOLID** (especialmente la S) **aumenta la cohesión**.

### **Acoplamiento**

- Dependencias directas y rígidas
  - alto acoplamiento
- Dependencias a través de interfaces
  - bajo acoplamiento

**SOLID** (especialmente la D y la I) **reduce el acoplamiento**.

## **Ejemplo completo integrado**

### **Diseño incorrecto (sin SOLID)**

Una sola clase:

- Gestiona usuarios
- Cobra pedidos
- Accede a base de datos
- Envía correos

Problemas:

- Difícil de entender
- Cada cambio afecta a todo
- Alto acoplamiento
- Baja cohesión

### **Diseño correcto (aplicando SOLID)**

Clases separadas:

- Usuario
- Pedido
- Pago
- Email

Cada clase:

- Tiene una responsabilidad
- Depende de interfaces
- Puede cambiar sin romper las demás

Resultado:

- Código mantenable
- Código profesional

## Errores típicos

1. **Clases demasiado grandes**
  - Porque se piensa solo en que funcione
2. **Muchos if/else según el tipo**
  - Porque no se piensa en extensión
3. **Herencias mal usadas**
  - Porque se hereda para reutilizar código
4. **Dependencia directa de tecnologías**
  - Porque no se piensa en el cambio futuro

Estos errores son normales, SOLID existe para evitarlos.

## Qué NO es SOLID

- No es un framework
- No es una librería

- No es algo que se aplique siempre al 100%

SOLID es una **guía**.

Programar bien no es escribir código rápido.

Programar bien es escribir código que otros puedan entender y modificar sin miedo.

SOLID no se memoriza, se entiende y se practica.