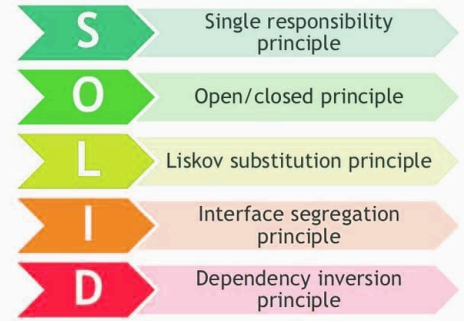
Revisión de Principios SOLID

En esta clase realizaremos una revisión profunda de los principios SOLID, con especial énfasis en la detección de violaciones comunes y en la refactorización de código aplicando buenas prácticas.

1. Teoría: Principios SOLID



1.1. Principio de Responsabilidad Única (SRP)

El Principio de Responsabilidad Única establece que una clase debe tener una única razón para cambiar. Esto implica que debe estar enfocada en una sola tarea o responsabilidad. Una violación típica de este principio ocurre cuando una clase combina lógica de negocio, persistencia de datos y manejo de interfaz. Esto dificulta el mantenimiento, la reutilización y la escalabilidad.

Ejemplo de violación: una clase 'Reporte' que además de generar datos se encarga de guardarlos en disco y de mostrarlos en pantalla.

1.2. Principio Abierto/Cerrado (OCP)

El Principio Abierto/Cerrado establece que las entidades de software deben estar abiertas para extensión pero cerradas para modificación. En la práctica, esto significa que debemos poder agregar nuevas funcionalidades sin tener que modificar el código existente. Se logra mediante abstracciones (interfaces, clases abstractas) y la aplicación de polimorfismo.

Una violación típica de OCP ocurre cuando necesitamos modificar una clase existente cada vez que surge un nuevo requerimiento. Por ejemplo, una clase 'CalculadoraImpuestos' que requiere ser editada para soportar cada nuevo tipo de impuesto.

1.3. Principio de Sustitución de Liskov (LSP)

El Principio de Sustitución de Liskov establece que las clases hijas deben poder sustituir a sus clases padres sin alterar el comportamiento esperado del sistema. Esto implica que las subclases no deben violar las expectativas establecidas por la clase base, evitando efectos colaterales o comportamientos inesperados.

Una violación común ocurre cuando una subclase sobreescribe un método de tal forma que rompe la lógica prevista en la clase padre. Por ejemplo, una clase 'Ave' con un método 'volar()', y una subclase 'Pingüino' que implementa 'volar()' arrojando una excepción.

2. Ejercitación: Principios SOLID

2.1. Principio de Responsabilidad Única (SRP)

Una tienda online procesa pedidos y la clase `OrderProcessor` terminó acumulando demasiadas tareas. Queremos **una única razón de cambio por clase**.

Click acá para ir al código

Consignas de refactorización:

- Extraer **servicios** con interfaces: `IOrderValidator`, `IOrderRepository`, `INotificationService`, `ILogger`.
- `OrderProcessor` debe **coordinar** (orquestar) y no hacer el trabajo de cada servicio.
- Mantener la **misma funcionalidad observable** (persistir, loguear, notificar).

Criterios de aceptación

- Puedo cambiar la estrategia de persistencia (archivo → memoria) **sin tocar** `OrderProcessor`.
- Puedo cambiar email por notificación "mock" **sin tocar** `OrderProcessor`.
- `OrderProcessor` queda con **una sola razón de cambio**: la orquestación del proceso.

2.2. Principio Abierto/Cerrado (OCP)

Hay diferentes categorías de productos con reglas de precio. Cada nuevo tipo obliga a editar la misma clase.

Click acá para ir al código

Consignas de refactorización

- Introducir **abstracción**: IPricingRule con bool IsMatch(Product) y decimal Compute(Product).
- PriceCalculator pasa a **componer** una lista de reglas (IEnumerable<IPricingRule>) y aplicar la que corresponda (o un pipeline si combinan).
- **Agregar un tipo nuevo** (p. ej., Seasonal) **sin modificar** código existente: solo añadir una nueva regla.

Criterios de aceptación

- Eliminar el switch/if…else por **polimorfismo**.
- Soportar **múltiples reglas** o una regla por tipo, según diseño elegido.
- La **incorporación de nuevos tipos** no modifica PriceCalculator (solo registra reglas).

(Opcional: agregar un test donde, al introducir Seasonal, no se toca PriceCalculator.)

3.3. Principio de Sustitución de Liskov (LSP)

`Square` hereda de `Rectangle` y sobreescribe setters para mantener lados iguales.
Algunos clientes de `Rectangle` esperan poder setear ancho y alto de forma independiente.
El resultado: comportamiento sorprendente.

Click acá para ir al código

Demostración del problema

```
var rect = new SolidWorkshop.LSP.Rectangle();
Console.WriteLine(SolidWorkshop.LSP.GeometryClient.MakeItWide(rect)); // 20 (ok)

var square = new SolidWorkshop.LSP.Square();
Console.WriteLine(SolidWorkshop.LSP.GeometryClient.MakeItWide(square));
// 20 esperado, pero da 100 (10x10) o 4 (2x2) según el orden → rompe LSP
```

Consignas de refactorización (posibles caminos)

- **Opción A (modelar correctamente):** no heredar Square de Rectangle. Crear abstracción IShape con Area() y modelar Rectangle y Square como **tipos hermanos**, cada uno con su contrato consistente.
- **Opción B (inmutabilidad/constructores):** volver **inmutables** las dimensiones (solo por constructor) y eliminar setters que rompen el contrato esperado del cliente.
- **Opción C (separar responsabilidades de mutación):** si se requieren mutaciones, definir métodos con **intención explícita** (p. ej., Resize(width, height) vs ResizeToSquare(side)) en tipos diferentes.

Criterios de aceptación

- El cliente GeometryClient no debe cambiar; **intercambiar** un Rectangle por un Square (o por otra forma) **no altera** las expectativas del contrato.
- No hay efectos laterales sorpresivos al setear propiedades.

- El diseño final **respeta sustitución**: cualquier IShape puede ser consumida por clientes que esperan una forma.