



Gráfica: Tomic Riedel

# *Principios SOLID*

MSc. César Augusto López Gallego

Profesor Facultad Ingeniería en TIC – UPB

Coordinador Área de Programación, Computación y Desarrollo de  
Software

[cesar.lopezg@upb.edu.co](mailto:cesar.lopezg@upb.edu.co)

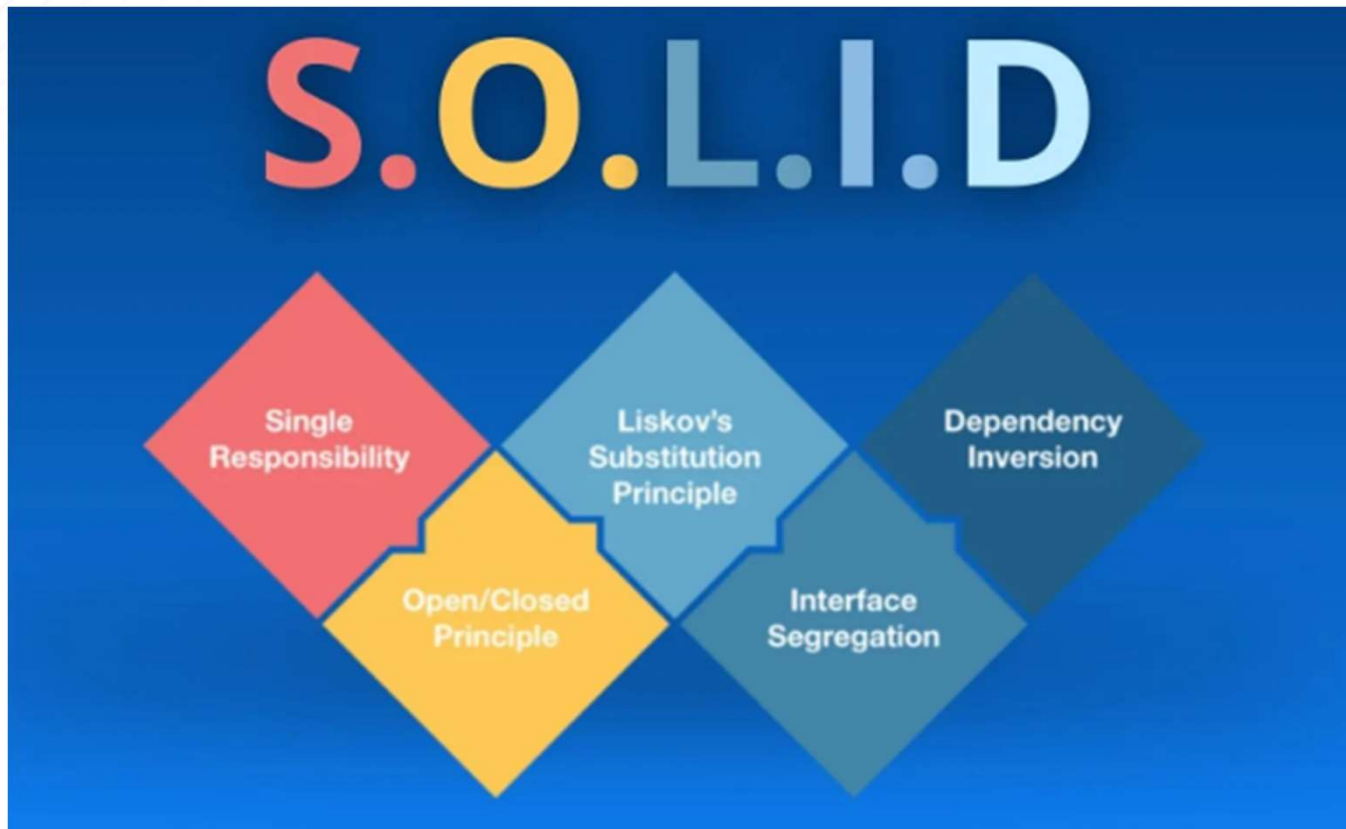


Robert C. Martin  
"Uncle Bob"

## Objetivo SOLID

Promover buenas prácticas de diseño de software, en particular en programación orientada a objetos (OOP), abordando problemas comunes que enfrentan los desarrolladores a medida que los sistemas de software crecen en tamaño y complejidad.

# 5 Principios



Gráfica: SW Hosting

S

## Principio de Responsabilidad Simple

*Una clase debe tener solo una razón (o preocupación) para cambiar.*

Debe tener solo una tarea simple y bien definida dentro del software.

¿Cómo?

Dividir las clases en unidades más pequeñas y más específicas

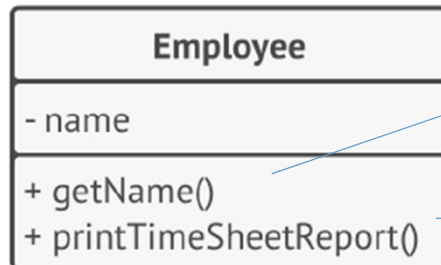
Reduce la complejidad.

Mejora la legibilidad y la comprensión

Facilita mantener y ampliar el código

# S

## Principio de Responsabilidad Simple



1ª Preocupación: la gestión de los datos del empleado cuando estos cambian. Esta es la principal función de esta clase

2ª Preocupación, Qué pasa si el formato de tiempo cambia?

Luego de aplicar el principio, se dividen las clases en unidades más pequeñas y más específicas



Ejemplo tomado de: Dive into Design Patterns.



## S

# Principio de Responsabilidad Simple



```
public class User
{
    public string Username { get; set; }
    public string Email { get; set; }

    public void Register()
    {
        // Register user logic, e.g. save to database...

        // Send email notification
        EmailSender emailSender = new EmailSender();
        emailSender.SendEmail("Welcome to our platform!", Email);
    }
}
```

1

2

- ¿Cuáles son los cambios que preocupan a la clase?
- EmailSender es una preocupación?

```
public class EmailSender
{
    public void SendEmail(string message, string recipient)
    {
        // Email sending logic
        Console.WriteLine($"Sending email to {recipient}: {message}");
    }
}
```

```
public class User
{
    public string Username { get; set; }
    public string Email { get; set; }
}
```

```
public class UserService
{
    public void RegisterUser(User user)
    {
        // Register user logic...

        EmailSender emailSender = new EmailSender();
        emailSender.SendEmail("Welcome to our platform!", user.Email);
    }
}
```

# O

## Principio Open / Closed



Las clases deben estar abiertas para extensión pero cerradas para modificación.

Evitar que el código existente colapse cuando se implementan nuevas características.

Una clase está abierta si esta se puede extender, es decir: hacer lo que se quiera con ella: agregar nuevos métodos o campos, anular un comportamiento, etc.

La clase está cerrada si está 100% lista para ser utilizada por otras clases: su interfaz está claramente definida y no se cambiará en el futuro.

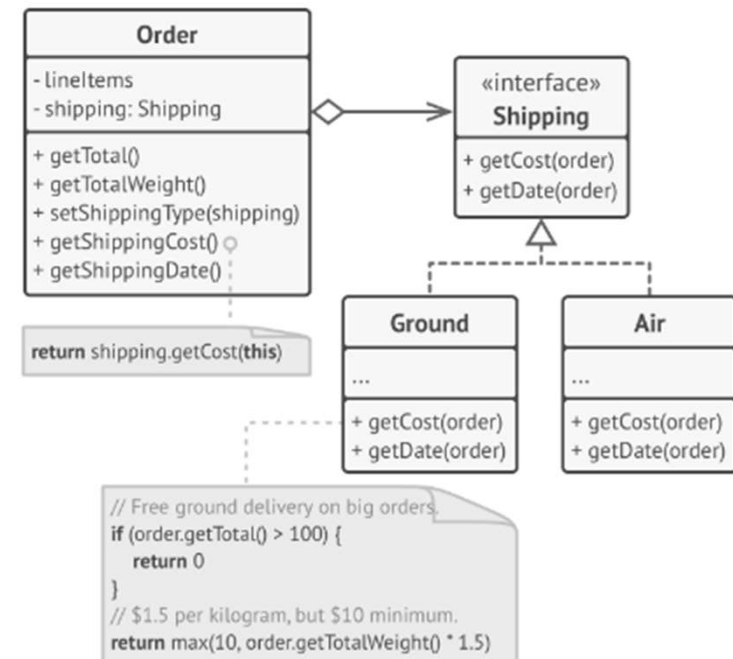
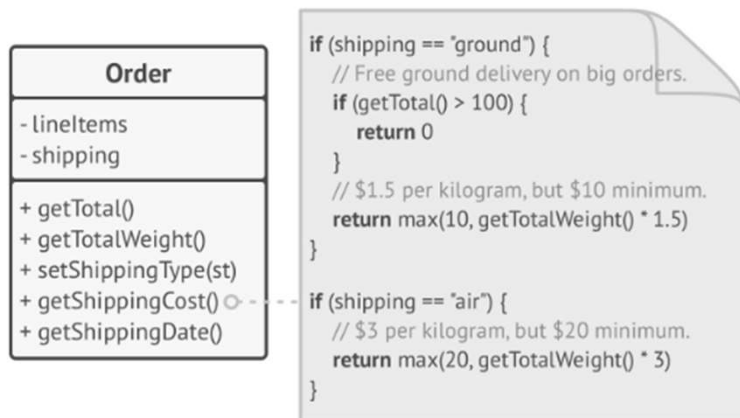
Si la clase principal tiene un error, esta es la que se debe corregir.

NO aplica la creación de una subclase para resolver el error.

*Una clase secundaria no debería ser responsable de los problemas de la clase principal.*

Fomenta el uso de la abstracción y el polimorfismo para lograr este objetivo, lo que permite que el código se extienda fácilmente a través de la herencia o la composición.

OCP invita a diseñar el software de modo que para agregar nuevas funcionalidades la vía sea agregando código nuevo. Esto contribuye a crear software fácil de mantener y con un acoplamiento



Ejemplo tomado de: Dive into Design Patterns.



## O

# Principio Open / Closed



```
public enum ShapeType
{
    Circle,
    Rectangle
}
```

```
public abstract class Shape
{
    public abstract double CalculateArea();
}
```

```
public class Shape
{
    public ShapeType Type { get; set; }
    public double Radius { get; set; }
    public double Length { get; set; }
    public double Width { get; set; }

    public double CalculateArea()
    {
        switch (Type)
        {
            case ShapeType.Circle:
                return Math.PI * Math.Pow(Radius, 2);
            case ShapeType.Rectangle:
                return Length * Width;
            default:
                throw new InvalidOperationException("Unsupported shape type.");
        }
    }
}
```

```
public class Circle : Shape
{
    public double Radius { get; set; }

    public override double CalculateArea()
    {
        return Math.PI * Math.Pow(Radius, 2);
    }
}
```

```
public class Rectangle : Shape
{
    public double Length { get; set; }
    public double Width { get; set; }

    public override double CalculateArea()
    {
        return Length * Width;
    }
}
```

# L

## Principio Sustitución de Liskov



Este principio nos obliga a pensar muy bien el diseño de la herencia.

Al extender una clase, se debe poder pasar objetos de la subclase en lugar de objetos de la superclase sin romper el código del cliente.

La subclase debe seguir siendo compatible con el comportamiento de la superclase.

*"Los objetos de una clase derivada deben poder sustituir a los objetos de su clase base sin alterar el comportamiento esperado del programa."*

# L

## Principio Sustitución de Liskov

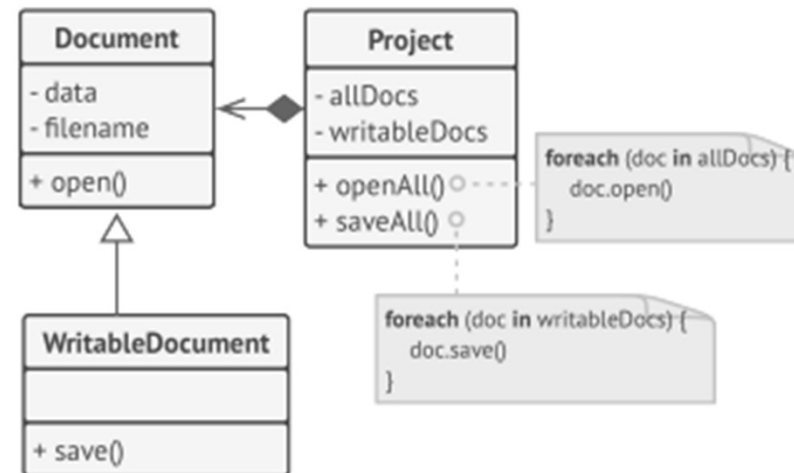
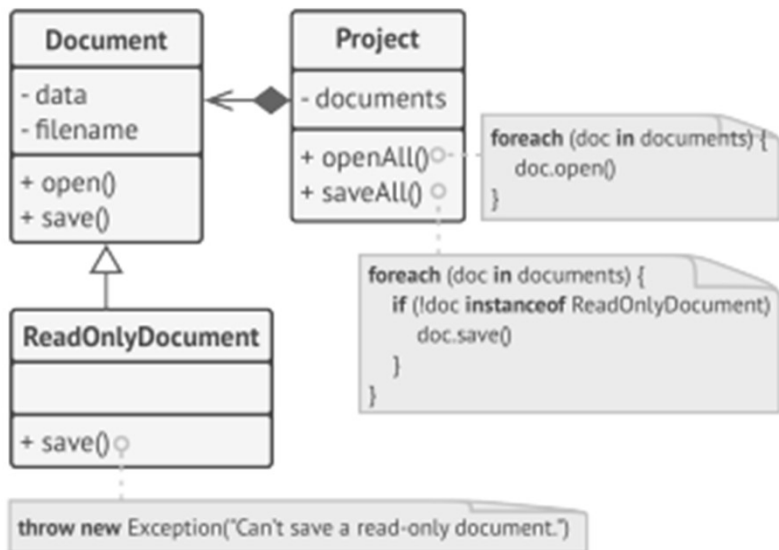


### 🔍 ¿Qué tener en cuenta al diseñar la herencia?

- ✓ **Evitar herencias incorrectas:** No forzar subclases que no cumplen el mismo contrato que la superclase.
- ✓ **Aplicar interfaces:** En lugar de heredar métodos innecesarios, definir interfaces específicas.
- ✓ **Usar composición en lugar de herencia:** Si una relación "ES UN" no es clara, quizás sea mejor que un objeto "TENGA UN" otro objeto.

## L

# Principio Sustitución de Liskov



La solución: La clase Document, es un documento readonly y la clase WritableDocument extiende Document e implementa el comportamiento save

## L

# Principio Sustitución de Liskov



```
abstract class Automovil {  
    // Método genérico para cargar energía o combustible  
    public abstract void Repostar();  
}  
  
class Electrico : Automovil {  
  
    public override void Repostar() {  
        throw new NotImplementedException("Los autos  
        eléctricos no pueden repostar gasolina.");  
    }  
}  
  
class Gasolina : Automovil {  
    public override void Repostar() {  
        Console.WriteLine("Cargando gasolina...");  
    }  
}
```

```
class Program {  
    static void Main() {  
        List<Automovil> autos = new List<Automovil> {  
            new Gasolina(),  
            new Electrico() // ⚠ Este fallará si se llama a Repostar()  
        };  
  
        foreach (var auto in autos) {  
            auto.Repostar(); // ✗ Violación del LSP: Electrico NO puede  
            ejecutar este método  
        }  
    }  
}
```



## L

# Principio Sustitución de Liskov



```
abstract class Automovil {  
    public abstract void Conducir();  
}
```

```
// Interfaz para autos que usan gasolina  
interface ICombustible {  
    void Repostar();  
}
```

```
// Interfaz para autos eléctricos  
interface IElectrico {  
    void CargarBateria();  
}
```

```
class Gasolina : Automovil, ICombustible {  
    public override void Conducir() {  
        Console.WriteLine("Conduciendo un auto de gasolina...");  
    }  
  
    public void Repostar() {  
        Console.WriteLine("Cargando gasolina...");  
    }  
}  
  
class Electrico : Automovil, IElectrico {  
    public override void Conducir() {  
        Console.WriteLine("Conduciendo un auto eléctrico...");  
    }  
  
    public void CargarBateria() {  
        Console.WriteLine("Cargando batería...");  
    }  
}
```

## L

# Principio Sustitución de Liskov



```
class Program {
    static void Main() {
        List<Automovil> autos = new List<Automovil> {
            new Gasolina(),
            new Electrico()
        };

        foreach (var auto in autos) {
            auto.Conducir(); // ☒ Ambos pueden conducir sin
            problemas
        }

        // Lista de autos a gasolina
        List<ICombustible> autosGasolina = new List<ICombustible>
        {
            new Gasolina()
        };
    }
}
```

```
foreach (var auto in autosGasolina) {
    auto.Repostar(); // ☒ Solo los de gasolina pueden
    repostar
}

// Lista de autos eléctricos
List<IElectrico> autosElectricos = new List<IElectrico> {
    new Electrico()
};

foreach (var auto in autosElectricos) {
    auto.CargarBateria(); // ☒ Solo los eléctricos pueden
    cargar batería
}
}
```

# L

## Principio Sustitución de Liskov



- **Error común:** Heredar métodos que no tienen sentido en algunas subclases.
- **Solución:** Usar interfaces y dividir correctamente las responsabilidades.
- **Resultado:** Código más limpio, flexible y fácil de mantener.

# L

## Principio Sustitución de Liskov



Requisitos formales para las subclases, y específicamente para sus métodos.

1. Los tipos de parámetros en un método de una subclase deben coincidir o ser más abstractos que los tipos de parámetros en el método de la superclase.
2. El tipo de retorno en un método de una subclase debe coincidir o ser un subtipo del tipo de retorno en el método de la superclase.
3. Un método en una subclase no debe generar tipos de excepciones que no se espera que genere el método base
4. Una subclase no debe reforzar las condiciones previas
5. Una subclase no debe debilitar las condiciones posteriores
6. Las invariantes de una superclase deben conservarse
7. Una subclase no debe cambiar los valores de los campos privados de la superclase.

## L

# Principio Sustitución de Liskov



- ✓ 1. Los tipos de parámetros en un método de una subclase deben coincidir o ser más abstractos

📌 **Regla:**

Una subclase no debe restringir los tipos de parámetros. Puede aceptar un tipo más general, pero no más específico.

```
class Taller {  
    public virtual void Reparar(Vehiculo vehiculo) {  
        Console.WriteLine("Reparando vehículo...");  
    }  
}  
  
class TallerEspecializado : Taller {  
    public override void Reparar(Auto auto) { // ✗ Error: más específico que Vehiculo  
        Console.WriteLine("Reparando auto...");  
    }  
}
```

```
class Taller {  
    public virtual void Reparar(Vehiculo vehiculo) {  
        Console.WriteLine("Reparando vehículo...");  
    }  
}  
  
class TallerEspecializado : Taller {  
    public override void Reparar(Vehiculo vehiculo) { // ✓ Tipo igual o más abstracto  
        Console.WriteLine("Reparando vehículo en taller especializado...");  
    }  
}
```



## L

# Principio Sustitución de Liskov



✓ 2. El tipo de retorno en un método de una subclase debe coincidir o ser un subtipo

📌 Regla:

El método en la subclase puede devolver un tipo más específico que la superclase.

```
class Fabrica {  
    public virtual Vehiculo CrearVehiculo() {  
        return new Vehiculo();  
    }  
}  
  
class FabricaAutos : Fabrica {  
    public override string CrearVehiculo() { // ✗ Error: cambio de tipo de retorno  
        return "Auto creado";  
    }  
}
```

```
class Fabrica {  
    public virtual Vehiculo CrearVehiculo() {  
        return new Vehiculo();  
    }  
}  
  
class FabricaAutos : Fabrica {  
    public override Auto CrearVehiculo() { // ✓ En este caso Auto:Vehiculo. Tipo de retorno más específico.  
        return new Auto();  
    }  
}
```

## L

# Principio Sustitución de Liskov



## ✓ 3. Un método en una subclase no debe generar tipos de excepciones inesperadas

### 📌 Regla:

Si la superclase no lanza excepciones, la subclase tampoco debería lanzar excepciones inesperadas.

```
class Vehiculo {  
    public virtual void Arrancar() {  
        Console.WriteLine("Vehículo arrancando...");  
    }  
}  
  
class Auto : Vehiculo {  
    public override void Arrancar() {  
        throw new InvalidOperationException("Fallo en el arranque"); // ❌ Error inesperado  
    }  
}
```

```
class Vehiculo {  
    public virtual void Arrancar() {  
        Console.WriteLine("Vehículo arrancando...");  
    }  
}  
  
class Auto : Vehiculo {  
    public override void Arrancar() {  
        Console.WriteLine("Auto arrancando...");  
    }  
}
```

## L

# Principio Sustitución de Liskov



## ✓ 4. Una subclase no debe reforzar las condiciones previas

### 📌 Regla:

Las condiciones previas en la subclase deben ser **iguales o más débiles** que en la superclase.

```
class Vehiculo {  
    public virtual void CargarCombustible(int litros) {  
        Console.WriteLine($"Cargando {litros} litros de combustible...");  
    }  
}  
  
class Auto : Vehiculo {  
    public override void CargarCombustible(int litros) {  
        if (litros < 10) {  
            throw new ArgumentException("Se deben cargar al menos 10 litros"); // ✗ Más restrictivo  
        }  
        Console.WriteLine($"Cargando {litros} litros en el auto...");  
    }  
}
```

```
class Auto : Vehiculo {  
    public override void CargarCombustible(int litros) {  
        Console.WriteLine($"Cargando {litros} litros en el auto...");  
    }  
}
```

## L

# Principio Sustitución de Liskov



## ✓ 5. Una subclase no debe debilitar las condiciones posteriores

### 📌 Regla:

Si un método en la superclase garantiza un estado después de ejecutarse, la subclase no puede romper esa garantía.

```
class Vehiculo {  
    public virtual int ObtenerVelocidadMaxima() {  
        return 200;  
    }  
}  
  
class Bicicleta : Vehiculo {  
    public override int ObtenerVelocidadMaxima() {  
        return -10; // ✗ No tiene sentido una velocidad negativa  
    }  
}
```

```
class Bicicleta : Vehiculo {  
    public override int ObtenerVelocidadMaxima() {  
        return 50; // ✓ Mantiene la condición esperada (velocidad positiva)  
    }  
}
```

## L

# Principio Sustitución de Liskov



## ✓ 6. Las invariantes de una superclase deben conservarse

### 📌 Regla:

Las reglas que definen un objeto deben seguir siendo válidas en sus subclases.

```
class Vehiculo {  
    protected int velocidad;  
  
    public Vehiculo() {  
        velocidad = 0; // 🚗 Siempre inicia en 0  
    }  
}  
  
class Auto : Vehiculo {  
    public Auto() {  
        velocidad = -100; // ❌ No tiene sentido que un auto inicie con velocidad negativa  
    }  
}
```

```
class Auto : Vehiculo {  
    public Auto() {  
        velocidad = 0; // ✅ Mantiene la invariante de la superclase  
    }  
}
```



## L

# Principio Sustitución de Liskov



✓ 7. Una subclase no debe cambiar los valores de los campos privados de la superclase

📌 Regla:

Los atributos privados de la superclase **no deben modificarse directamente** en la subclase.


```
class Vehiculo {  
    private int velocidad = 100; // 🚗 Configuración interna  
}  
  
class Auto : Vehiculo {  
    public Auto() {  
        this.velocidad = 200; // ❌ No se puede modificar un campo privado  
    }  
}
```

```
class Vehiculo {  
    protected int velocidad = 100;  
}  
  
class Auto : Vehiculo {  
    public Auto() {  
        this.velocidad = 150; // ✅ Se usa un campo "protected" en lugar de "private"  
    }  
}
```

# L

## Principio Sustitución de Liskov



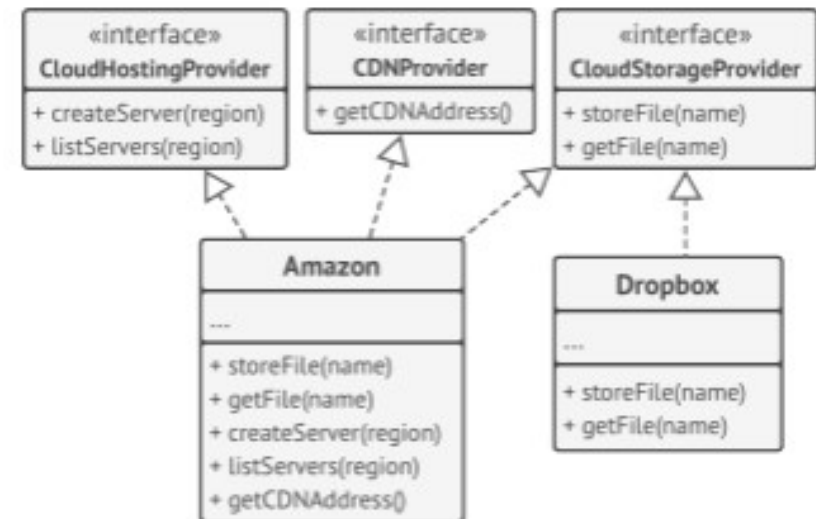
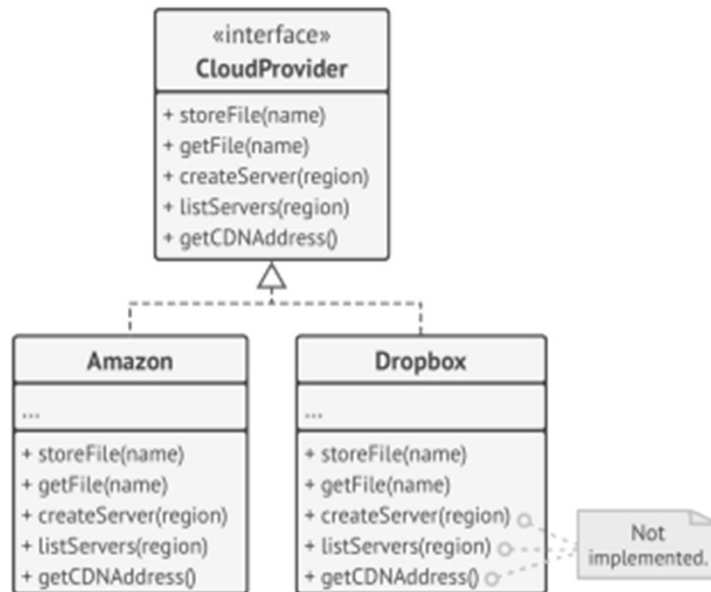
- ☒ Ventajas de Aplicar el Principio de Sustitución de Liskov
  - Código más robusto y mantenible 
  - Evita errores por comportamientos inesperados de subclases.
  - Mayor reutilización de código ☒ Diseñar correctamente la jerarquía permite extender funcionalidades sin problemas.
  - Facilita la escalabilidad ☒ Se pueden agregar nuevas subclases sin romper el código existente.

# I ISP: Principio de Segregación de Interfaces



Una interfaz no debe obligar a una clase a implementar métodos que no usa.

Es mejor tener **varias interfaces pequeñas y específicas** en lugar de una **única interfaz grande y genérica** que fuerce a las clases a implementar métodos innecesarios.



# I ISP: Principio de Segregación de Interfaces



## Ventajas:

- ✓ **Evita clases sobrecargadas:** No se obliga a una clase a implementar métodos que no necesita.
- ✓ **Mejora la flexibilidad:** Se pueden cambiar o agregar funcionalidades sin afectar clases que no las usan.
- ✓ **Facilita la mantenibilidad:** El código es más claro y fácil de entender.

## ✂ ¿Qué se debe tener en cuenta al diseñar para cumplir ISP?

- [1] **Evitar interfaces muy generales:** Si una interfaz tiene demasiados métodos, puede estar mal diseñada.
- [2] **Agrupar métodos relacionados:** Crear interfaces específicas con métodos afines.
- [3] **Aplicar el principio "Solo lo que necesito":** Cada clase debe implementar solo las interfaces que usa.
- [4] **Usar interfaces pequeñas y reutilizables:** Esto mejora la flexibilidad y el mantenimiento.

# D

## DIP: Principio de Inversión de Dependencia

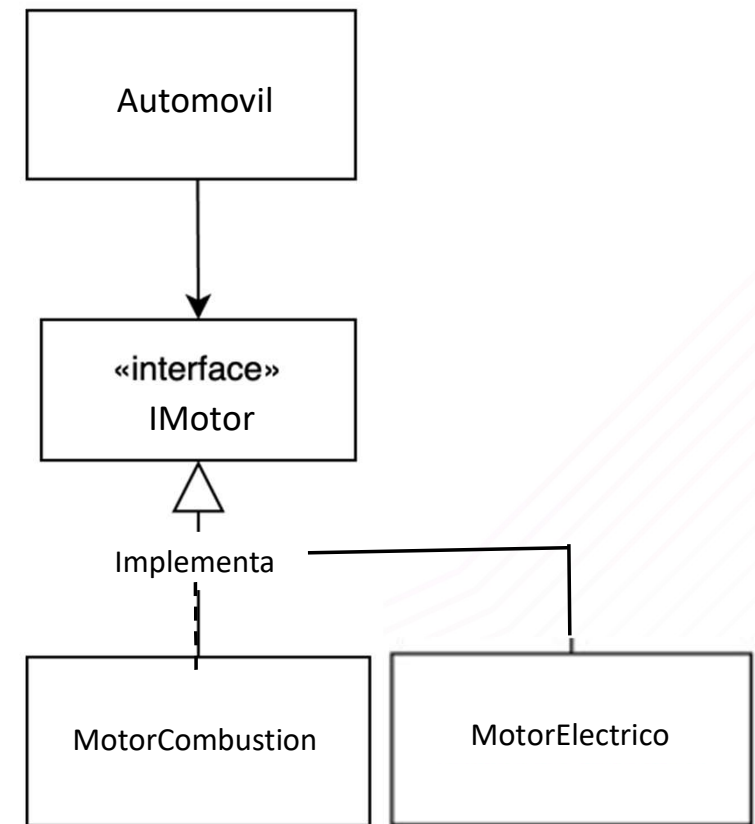


- Las clases de alto nivel (tienen implementada la lógica del negocio) no deberían depender de las clases de bajo nivel (tienen implementaciones específicas). Ambas deberían depender de abstracciones.
- Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de abstracciones (interfaces o clases abstractas).
- Cuando se cumple lo anterior, se puede decir que hay una inversión de dependencia
- Este principio promueve un código más flexible mediante:
  - El desacoplamiento entre módulos (clases)
  - El uso de interfaces o clases abstractas para definir dependencias



# Inyección de Dependencias en el Constructor

- Es la forma más común de Inversión de Dependencia(DI)
- Patrón de diseño en el que las dependencias de una clase se proporcionan desde el exterior en lugar de ser creadas dentro de la clase.
- Consiste en proporcionar las dependencias a través del constructor de la clase.
- Ventajas:
- Reducción del acoplamiento --> La clase no crea sus dependencias directamente, sino que las recibe en su constructor.

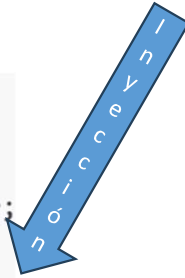


# Inyección de Dependencias en el Constructor

```
public class Automovil
{
    private readonly IMotor _motor;

    public Automovil(IMotor motor)
    {
        _motor = motor;
    }

    public void Arrancar()
    {
        _motor.Encender();
    }
}
```



```
public interface IMotor
{
    void Encender();
}
```

```
public class MotorCombustion : IMotor
{
    public void Encender()
    {
        Console.WriteLine("Motor de combustión encendido.");
    }
}
```

```
public class MotorElectrico : IMotor
{
    public void Encender()
    {
        Console.WriteLine("Motor eléctrico encendido.");
    }
}
```

```
IMotor motorCombustion = new MotorCombustion();
Automovil auto1 = new Automovil(motorCombustion);
auto1.Arrancar(); // Salida: Motor de combustión encendido.
```

```
IMotor motorElectrico = new MotorElectrico();
Automovil auto2 = new Automovil(motorElectrico);
auto2.Arrancar(); // Salida: Motor eléctrico encendido.
```

# D

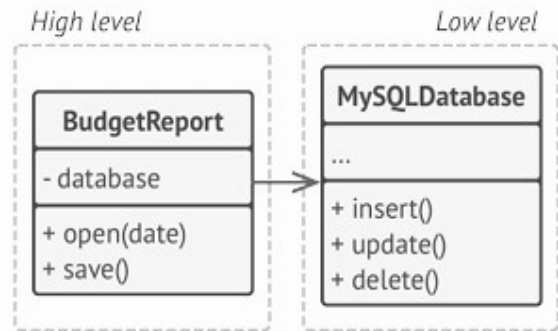
## DIP: Principio de Inversión de Dependencia



Recomendaciones para aplicar este principio:

1. Es necesario describir las interfaces para las operaciones de bajo nivel en las que se basan las clases de alto nivel, preferiblemente en términos comerciales.
  2. Diseñe de manera que las clases de alto nivel dependan de esas interfaces, en lugar de clases concretas de bajo nivel.
  3. Una vez que las clases de bajo nivel implementan estas interfaces, se vuelven dependientes del nivel de lógica empresarial, lo que invierte la dirección de la dependencia original.
- (DIP va de la mano con el principio Open/Closed)

# D DIP: Principio de Inversión de Dependencia

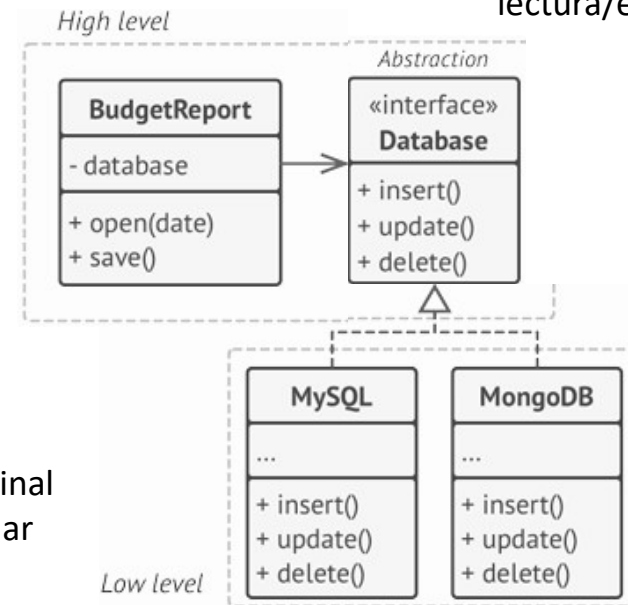


La clase BudgetReport utiliza la clase MySQLDatabase de bajo nivel para leer y conservar sus datos.

Un cambio en MySQLDatabase, podría alterar el funcionamiento de BudgetReport

La clase BudgetReport utiliza la interfaz, en lugar de conectarse con las clases de bajo nivel

Se crea una interfaz de alto nivel que describa las operaciones de lectura/escritura



La clase de bajo nivel original se puede cambiar o ampliar implementando la nueva Linterfaz de lectura/escritura declarada por la lógica de negocio

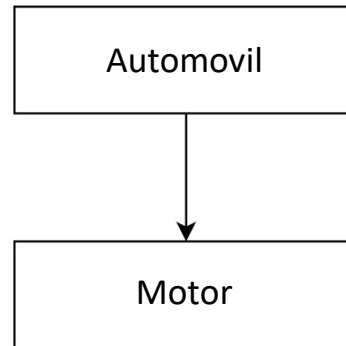
# D DIP: Principio de Inversión de Dependencia

```
public class Automovil
{
    private Motor _motor;

    public Automovil()
    {
        _motor = new Motor();
    }

    public void Arrancar()
    {
        _motor.Encender();
    }
}
```

Clase de alto nivel



🚨 VIOLA DIP: Hay una Dependencia directa de una implementación concreta

```
public class Motor
{
    public void Encender()
    {
        Console.WriteLine("Motor encendido.");
    }
}
```

Clase de bajo nivel



- La clase Automovil está **directamente acoplada** a la implementación concreta de Motor.
- Si se quiere cambiar Motor por MotorElectrico, se tendría que modificar Automovil, lo que rompe el principio de **abierto/cerrado (OCP)**.
- No se puede realizar pruebas unitarias de Automovil fácilmente porque depende de una implementación específica de Motor.



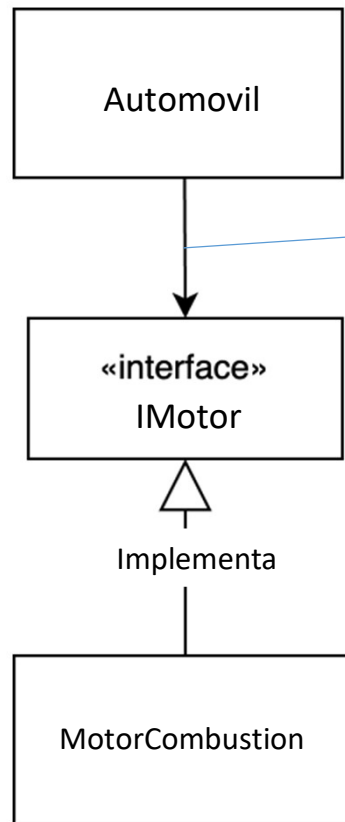
## D

## DIP: Principio de Inversión de Dependencia



```
public interface IMotor
{
    void Encender();
}
```

```
public class MotorCombustion : IMotor
{
    public void Encender()
    {
        Console.WriteLine("Motor de combustión encendido.");
    }
}
```



Se inyecta la dependencia en el constructor

```
public class Automovil
{
    private readonly IMotor _motor

    public Automovil(IMotor motor)
    {
        _motor = motor;
    }

    public void Arrancar()
    {
        _motor.Encender();
    }
}
```

☒ Beneficios:

- **Desacoplamiento:** Automovil no depende de ninguna implementación concreta de IMotor.

- **Facilidad de cambio:** Se puede cambiar MotorCombustion por MotorElectrico sin modificar Automovil.



# D

## DIP: Principio de Inversión de Dependencia



### 💧 Errores Comunes al Violar DIP

#### 1. Acoplamiento fuerte entre clases

- Cuando una clase de alto nivel crea instancias de clases de bajo nivel (new directamente dentro de la clase).

#### 2. Uso de implementaciones en lugar de abstracciones

- Si un método espera un MotorCombustion en lugar de un IMotor, el código no es extensible.

#### 3. Abuso de contenedores de inyección de dependencia

- Aunque la inyección de dependencias es útil, si se abusa creando demasiadas interfaces innecesarias, el código se vuelve complejo sin beneficio real.

## Ejemplo: Aplicación de SOLID al proyecto del Concesionario



1. **SRP (Responsabilidad Única):**
  - Se separaron validaciones en la clase `Validaciones`.
  - Su única responsabilidad es hacer las validaciones.
2. **OCP (Abierto/Cerrado):**
  - `Automovil` es más extensible sin modificarla directamente.
  - Si cambia la validación de algún atributo, ya no se tiene que cambiar la implementación dentro de `Automovil`
3. **LSP (Sustitución de Liskov):**
  - Se revisaron herencias para evitar conflictos en el comportamiento.
4. **ISP (Segregación de Interfaces):**
  - Se mantuvo `IMantenimiento`, pero podría segmentarse aún más si es necesario.

## Ejemplo: Aplicación de SOLID al proyecto del Concesionario



### 5. DIP (Inversión de Dependencias):

- Automovil accedía a atributos definidos en Concesionario, como `valor_minimo_nuevo`, lo que generaba **acoplamiento innecesario**.
- Para romper el acoplamiento entre Automovil y Validacion, se hace lo siguiente: Se diseña una arquitectura donde Automovil no dependa directamente de Validaciones, sino que reciba una instancia de un servicio de validación a través de inyección de dependencias (DIP).

# Bibliografía



- Shvets Alexander. Dive into Design Patterns. Refactoring.guru. 2019.
- Perera Srinath. Software Architecture and Decision-Making\_ Leveraging Leadership, Technology, and Product Management to Build Great Products. Addison-Wesley. 2024
- Pacheco, Diego\_Sgro, Sam\_ - Principles of Software Architecture Modernization \_ Delivering engineering excellence with the art of fixing microservices, mono. BPB Publicatio. 2024
- Gabriel Baptista, Francesco Abbruzzese - Software Architecture with C\_ 12 and .NET 8 - Fourth Edition\_ Build enterprise applications using microservices, DevOps, EF Core. 2024
- Oliver Goldman.Effective Software Architecture\_ Building Better Software Faster (2024, Addison-Wesley Professional)
- John Gilbert - Software Architecture Patterns for Serverless Systems\_ Architecting for innovation with event-driven microservices (2024)

# Cibergrafía



- Freecodecamp
- Openwebinars.net
- YouTube: Software Architecture Monday
- refactoring.guru
- Github
- Coursera "Software Architecture". University of Alberta