

*Autor: Jose Luis Ortega Gila*

Tabla de Contenidos

[**Introducción** 3](#_Toc196651817)

[**Objetivos** 3](#_Toc196651818)

[**Clean Code** 4](#_Toc196651819)

[**¿Qué es la deuda técnica?** 4](#_Toc196651820)

[**¿Cómo se paga la deuda técnica?** 5](#_Toc196651821)

[**Cómo nombrar variables** 6](#_Toc196651822)

[1. Nombres poco descriptivos 6](#_Toc196651823)

[2. Nombres genéricos que no aportan contexto 6](#_Toc196651824)

[3. Uso de prefijos innecesarios 7](#_Toc196651825)

[4. Nombres con información redundante 7](#_Toc196651826)

[5. Uso de números mágicos y siglas sin significado 8](#_Toc196651827)

[6. Uso de mayúsculas incorrectas (convención PascalCase y camelCase) 10](#_Toc196651828)

[**Cómo nombrar funciones, argumentos y parámetros** 10](#_Toc196651829)

[**Consideraciones para las clases** 11](#_Toc196651830)

[1. Una clase debe tener una sola responsabilidad (Principio de Responsabilidad Única o SRP) 11](#_Toc196651831)

[2. Usar nombres claros y descriptivos para las clases 12](#_Toc196651832)

[3. Mantener las clases pequeñas y enfocadas 12](#_Toc196651833)

[4. Evitar las clases con muchos métodos públicos 12](#_Toc196651834)

[5. Mantener la cohesión de la clase 13](#_Toc196651835)

[6. Evitar la sobrecarga de getters y setters 13](#_Toc196651836)

[7. Usar excepciones adecuadamente 13](#_Toc196651837)

[**Problemática con la herencia** 13](#_Toc196651838)

[Code Smells (STUPID) 15](#_Toc196651839)

[**Singleton** 15](#_Toc196651840)

[**Tight Coupling: alto acoplamiento** 16](#_Toc196651841)

[**Untestability: código no probable** 17](#_Toc196651842)

[**Premature optimization: optimizaciones prematuras** 19](#_Toc196651843)

[**Innappropriate Intimacy** 20](#_Toc196651844)

[**Duplicate Code** 21](#_Toc196651845)

[**Principios SOLID** 21](#_Toc196651846)

[**¿Qué son los Principios SOLID?** 21](#_Toc196651847)

[**¿De dónde vienen los principios SOLID?** 21](#_Toc196651848)

[**¿Qué significa cada letra de SOLID?** 22](#_Toc196651849)

[**Single Responsibility: Responsabilidad Única** 22](#_Toc196651850)

[**Open Close Principle: Principio de Abierto y Cerrado** 26](#_Toc196651851)

[**Liskov’s Sustitution Principle: Principio de Sustitución de Liskov** 29](#_Toc196651852)

[**Interfaz Segregation: Segregación de Interfaz** 32](#_Toc196651853)

[**Dependency inversion: Inversión de dependencias** 33](#_Toc196651854)

## **Introducción**

## **Objetivos**

Este manual procura enseñar a escribir código que sea fácil de entender, por nosotros y otros desarrolladores. También pretender enseñar patrones de diseño que permitan hacer código facilmente extensible y modificable. Clean code y los principios SOLID son los principios que nos permitirán llegar a estos objetivos

Practicaremos y aprenderemos buenas prácticas para el manejo de nuestro código y reducir la deuda técnica.

## **Clean Code**

**Clean Code** (código limpio) es un concepto que se refiere a escribir código **fácil de leer, entender, mantener y extender**. Va más allá de que el código simplemente "funcione"; se trata de que esté **bien estructurado**, sea **claro para otros programadores** (o para ti mismo en el futuro) y permita cambios sin generar errores colaterales.

### **¿Qué es la deuda técnica?**

**La deuda técnica** es una metáfora que compara los problemas en el código con **una deuda financiera**.

👉 Cuando en un proyecto haces las cosas **rápido pero no del todo bien** (por falta de tiempo, presión, falta de conocimiento, etc.), introduces **errores, mala estructura o soluciones temporales** en el código.  
Eso es como pedir un préstamo: ganas tiempo ahora, pero **tendrás que pagar intereses después**.

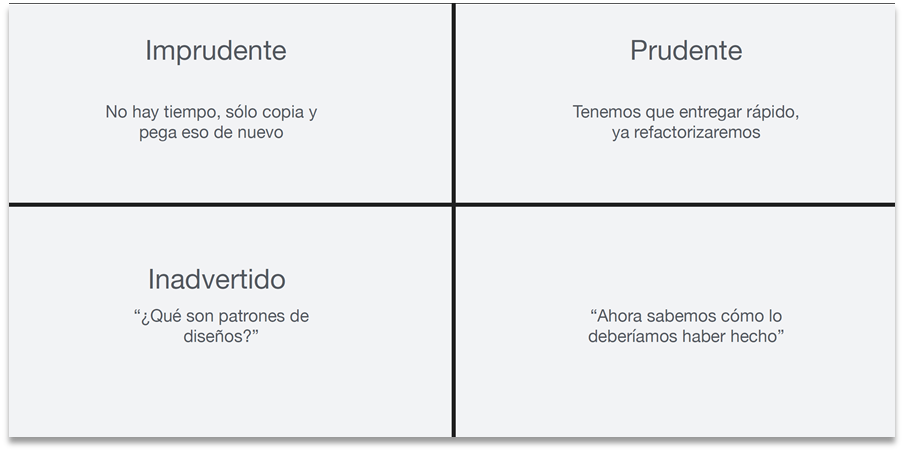
**Esos intereses** son:

* Más dificultad para entender el código.
* Más tiempo para hacer cambios.
* Mayor riesgo de errores cada vez que tocas algo.
* Más costo de mantenimiento.

**Ejemplo real sencillo**:  
Imagina que necesitas entregar una funcionalidad muy rápido. En vez de diseñarla bien, copias y pegas código por todos lados para que funcione. El proyecto se entrega a tiempo (¡bien!). Pero más adelante, cuando necesites cambiar algo, tendrás que **arreglar todos los lugares donde pegaste ese código**, lo que será **mucho más costoso y lento**.

**Conclusión**:  
👉 La deuda técnica **no es siempre mala** si la controlas y luego "la pagas" (refactorizando, mejorando el diseño).  
👉 Pero si **se acumula mucho** y no la pagas, **el proyecto se vuelve ingobernable**.

*Esquema de la deuda técnica de Martin Fowler*



* La deuda imprudente se genera cuando se quiere algo rápido o cuando, por desconocimiento, se cree que la tarea a desarrollar lleva poco tiempo y las fechas son muy ajustadas.
* La deuda imprudente e inadvertida es cuando no hay conocimiento técnico sobre patrones de diseño, código limpio…etc.
* La deuda prudente y deliberada es la que se genera cuando se tiene conocimiento sobre los principios de diseño y código limpio, pero por las prisas se deja para próximas versiones la refactorización del código.
* La deuda prudente e inadvertida se genera cuando no tenemos el conocimiento total de como va a acabar nuestro proyecto, es decir, cuando estamos programando nuestro proyecto y nos damos cuenta de que no es la mejor forma de hacerlo.

Caer en la deuda técnica es normal y a menudo inevitable. Pero lo que si es evitable es que aun a sabiendas de lo que se está haciendo, no nos preocupemos por pagar esa deuda técnica. Si no pagamos la deuda, nuestro proyecto crecerá y crecerá y será muy difícil el mantenimiento y la extensión del mismo, los programadores no aprenderán porque no se puede leer el código, y la tolerancia a fallos será mínima. Pagar la deuda antes de que esto suceda, evitará costos innecesarios, un conocimiento del negocio mas adecuado para los programadores, así como una reducción de incidencias y tener mas tiempo para el desarrollo.

### **¿Cómo se paga la deuda técnica?**

Para pagar la deuda técnica la palabra clave es **Refactorización.**

*La refactorización tiene como objetivo mejorar el código sin alterar su comportamiento, para que sea más entendible y tolerante a cambios*.

Esto es muy común cuando se da mantenimiento a un código que no hemos escrito.

Como hemos mencionado antes, la mala calidad del software siempre la acaba pagando alguien, el cliente, la empresa porque tiene que invertir en más recursos o el desarrollador que tiene que dedicar más tiempo y esfuerzo en programar en un sistema frágil.

Nuestro código debe ser simple y directo y debería poder leerse con la misma facilidad que un texto bien escrito.

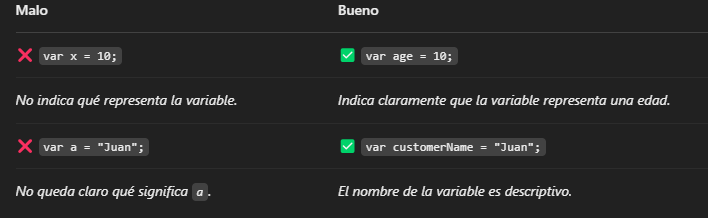
A continuación, iniciamos nuestro camino para escribir código limpio.

### **Cómo nombrar variables**

### Nombres poco descriptivos

**Un nombre poco descriptivo** es cuando eliges un nombre para una variable, función o clase que **no explica claramente qué es o qué hace**.

Son nombres que **no ayudan a entender el código** solo leyéndolo.  
Obligan a la persona que lee (¡incluso a ti mismo después de unos días!) a **deducir** o **adivinar** qué significa.

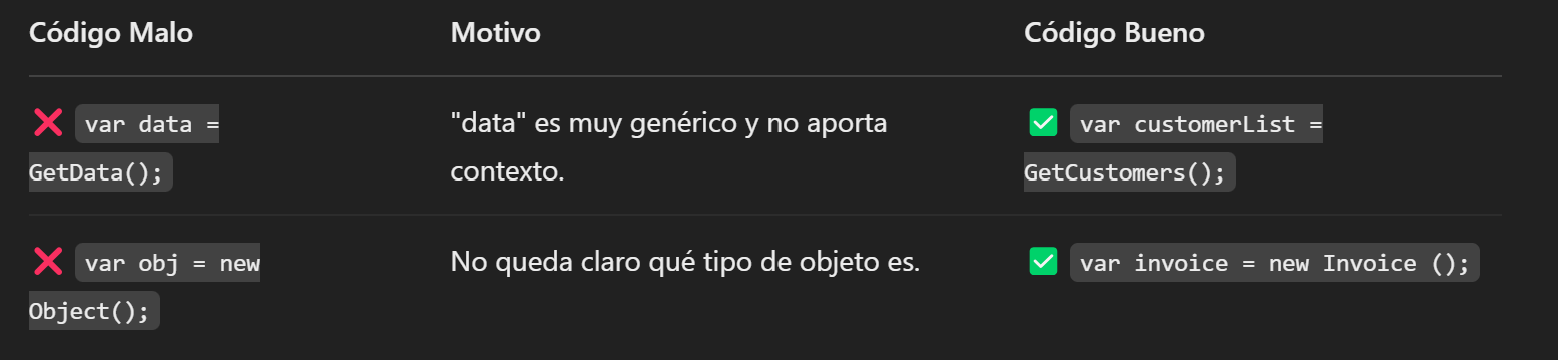


### Nombres genéricos que no aportan contexto

**Nombres genéricos que no aportan contexto** son nombres que, aunque a veces suenan "correctos", **no te dicen nada específico sobre su propósito real en el código**.  
Son **demasiado vagos**, **ambiguos** o **universales**, y **no ayudan a entender el "para qué"** de lo que nombran.

**Ejemplos típicos de nombres genéricos sin contexto:**

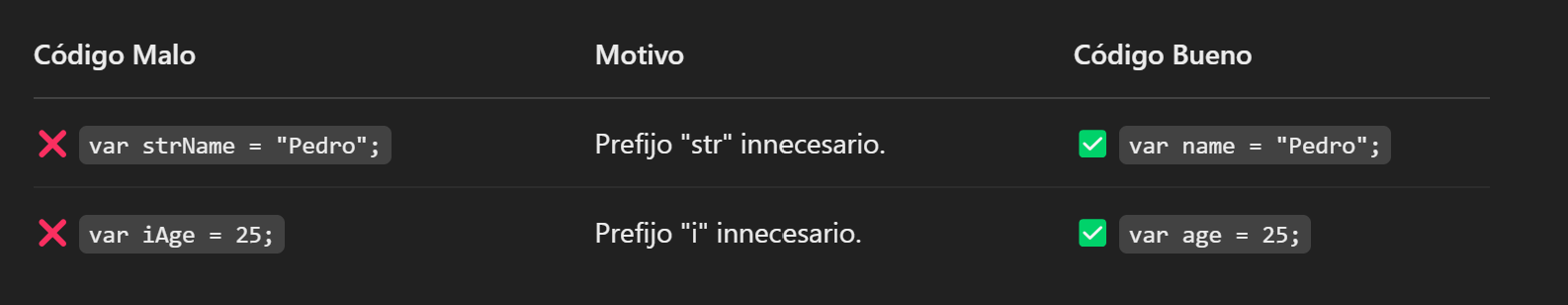
* data
* info
* object
* thing
* temp
* value



### Uso de prefijos innecesarios

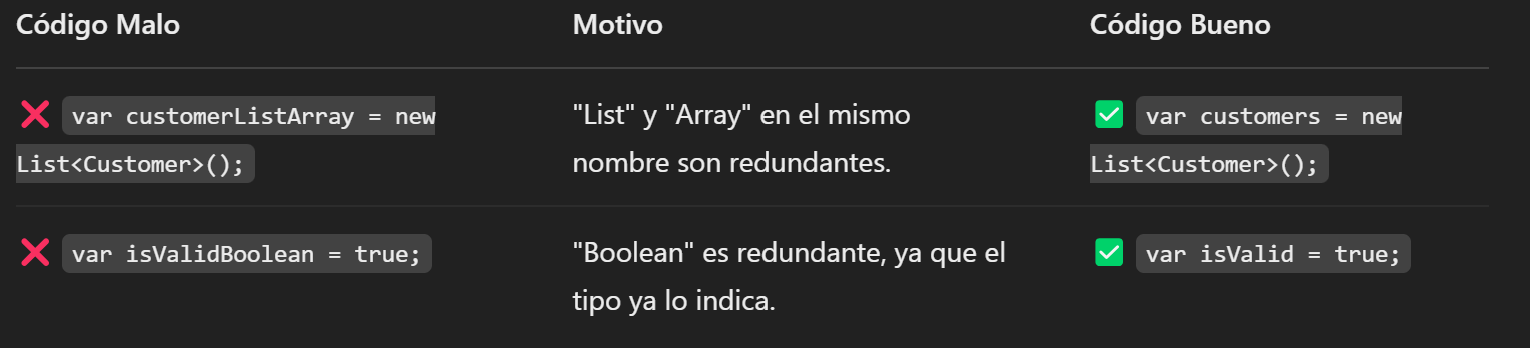
Los **prefijos innecesarios** son **palabras adicionales** al inicio de un nombre (ya sea variable, función o clase) que **no aportan valor real** y que **agregan redundancia** al código.

A menudo, estos prefijos tienen como objetivo **dar más contexto** o **hacer que el nombre suene más claro**, pero **realmente no ayudan** y pueden hacer que el código sea **más largo, menos conciso y más difícil de leer**.



### Nombres con información redundante

Son nombres que **incluyen detalles que ya son obvios o innecesarios** debido a la **estructura del código o al contexto**. Esto provoca que el código sea más **difícil de leer** y menos **conciso**.



### Uso de números mágicos y siglas sin significado

¿Qué son los números mágicos?

Los **números mágicos** son aquellos **valores numéricos literales** que se usan directamente en el código sin una explicación clara de su propósito.  
Se llaman "mágicos" porque **aparecen de la nada** y no tienen ningún contexto que explique su significado. Son **valores mágicos** en el sentido de que parece que tienen un propósito, pero no sabemos **por qué están ahí** ni **qué representan**.

¿Por qué son un problema?

* **No son legibles**: Al ver un número en el código, **no sabes inmediatamente qué representa**.
* **Difícil mantenimiento**: Si necesitas cambiar ese número en el futuro, **no sabes si has cubierto todos los casos** donde se usa, o si la lógica detrás de ese número aún es válida.
* **Reduce la claridad**: El código se vuelve **menos expresivo** y **más propenso a errores**.

public double CalculateDiscount(double price)

{

    return price \* 0.15;  // ¿Qué significa 0?15? ¿Por qué es 0?15?

}

¿Qué son las siglas sin significado?

Son **abreviaciones o siglas** que se usan en los nombres de **variables, funciones o clases** sin que sean **claras ni comprensibles** por sí solas.  
A veces, el programador que las escribió sabe lo que significan, pero alguien que lee el código (¡o incluso el propio programador después de un tiempo!) **no entenderá qué representan** sin tener que hacer suposiciones.

¿Por qué son un problema?

* **Reducción de claridad**: Las siglas pueden ser **confusas** si no son comúnmente conocidas o no están bien documentadas.
* **Dificultan la lectura**: Si el nombre es una sigla críptica, tendrás que **detenerte y adivinar su significado** en vez de entenderlo de un vistazo.
* **Aumentan la probabilidad de errores**: Si usas siglas sin significado claro, puede que alguien las use **incorrectamente** o las **confunda con otras siglas** similares.

**Ejemplos comunes de siglas sin significado**:

1. **Usar siglas sin contexto**:
   * usr para una variable que representa un usuario.
   * cnt para contar algo, pero ¿qué es lo que cuenta?
   * tmp para una variable temporal, pero ¿qué tipo de datos está almacenando?
2. **Siglas que solo tienen sentido en un contexto muy limitado**:
   * cb en vez de callback.
   * cfg en vez de configuration.
   * db en vez de database, que puede no ser claro en todos los contextos.

¿Cómo resolverlo?

La solución es **evitar siglas sin sentido** o **utilizar nombres completos y descriptivos**. Si usas siglas, asegúrate de que sean **comunes y conocidas** por todos los desarrolladores que puedan leer el código.

Ejemplo de código con **siglas sin significado**:

// Usando siglas sin claridad

public void prcUsrData(int usrId)

{

  // ¿Qué es "prc"? ¿Qué significa "usr"?

}

Ejemplo de código refactorizado:

public void processUserData(int userId)

{

    // Ahora es claro que estamos procesando los datos de un usuario

}

### Uso de mayúsculas incorrectas (convención PascalCase y camelCase)

#### 

### **Cómo nombrar funciones, argumentos y parámetros**

Sabemos que estamos desarrollando código limpio cuando **cada función hace exactamente lo que su nombre indica.**

En este código podemos observar que la función SumaYVerifica además de Sumar guarda el resultado en un fichero de texto

using System;

using System.IO;

class Program

{

    static void Main()

    {

        int resultado = SumaYVerifica(50, 60);

        Console.WriteLine($"El resultado es: {resultado}");

    }

    static int SumaYVerifica(int a, int b)

    {

        int suma = a + b;

        // Verifica si la suma es mayor a 100

        if (suma > 100)

        {

            // Guarda el resultado en un archivo si es mayor a 100

            File.WriteAllText("resultado.txt", $"Resultado mayor a 100: {suma}");

            Console.WriteLine("La suma es mayor a 100, y el resultado se ha guardado en un archivo.");

        }

        return suma;

    }

Lo correcto sería dejar que la función solo hiciera la suma y después, si procede, se guardara el resultado en fichero.

using System;

class Program

{

    static void Main()

    {

        int resultado = Suma(50, 60);

        if (resultado > 100)

        {

            // Guarda el resultado en un archivo si es mayor a 100

            File.WriteAllText("resultado.txt", $"Resultado mayor a 100: {suma}");

            Console.WriteLine("La suma es mayor a 100, y el resultado se ha guardado en…");

        }

        Console.WriteLine($"El resultado es: {resultado}");

    }

    static int Suma(int a, int b)

    {

        return a + b;

    }

}

La **simplicidad** en las funciones es fundamental, deben de ser fáciles de leer.

Las funciones deben tener un **tamaño reducido.** Cuando una función tiene muchas líneas de código significa que hace más cosas de las que debería hacer.

Si hacemos funciones de una sola línea estás no deben causar complejidad

public bool IsValidEmail(string email) => email.Contains("@") && email.Contains(".")

&& email.Length > 5 && email.IndexOf('@') < email.LastIndexOf('.');

Esta función causa complejidad. En lugar de utilizar una función flecha sería más conveniente utilizar un formato tradicional

public bool IsValidEmail(string email)

{

    if (string.IsNullOrEmpty(email)) return false;

    if (!email.Contains("@")) return false;

    if (!email.Contains(".")) return false;

    if (email.Length <= 5) return false;

    if (email.IndexOf('@') >= email.LastIndexOf('.')) return false;

    return true;

}

Dentro de la función debemos **evitar el uso del *else*** y priorizar el uso de la condicional ternaria

***condición ? expresión\_si\_verdadero : expresión\_si\_falso;***

### **Consideraciones para las clases**

### 1. Una clase debe tener una sola responsabilidad (Principio de Responsabilidad Única o SRP)

Cada clase debe tener **una única razón para cambiar**, lo que significa que debe encargarse de una **sola responsabilidad o funcionalidad**. Si una clase hace más de una cosa, es probable que se vuelva **difusa**, difícil de entender y de modificar sin afectar otras partes del sistema.

Ejemplo de violación

public class UserManager

{

    public void CreateUser() { ... }

    public void UpdateUser() { ... }

    public void DeleteUser() { ... }

    public void SendEmail() { ... }  // ¿Debería estar aquí realmente?

}

Refactorización (separa responsabilidades)

public class UserManager

{

    public void CreateUser() { ... }

    public void UpdateUser() { ... }

    public void DeleteUser() { ... }

}

public class EmailService

{

    public void SendEmail() { ... }

}

### 2. Usar nombres claros y descriptivos para las clases

El nombre de la clase debe **reflejar claramente su responsabilidad**. Un buen nombre debe dar una idea inmediata de lo que la clase hace, para que sea fácil de identificar en el código.

public class CustomerManager { ... } // Claro y descriptivo.

Evita nombres vagos como Helper, Manager, o Service sin contexto, ya que no dicen mucho sobre lo que realmente hace la clase.

### 3. Mantener las clases pequeñas y enfocadas

Las clases deben ser **lo más pequeñas posible** y mantener un **alcance claro**. Si una clase crece demasiado, puede volverse difícil de entender y de mantener.

**Ejemplo:**

Si una clase empieza a tener demasiados métodos, es una señal de que probablemente está haciendo **demasiadas cosas** y debe dividirse en varias clases.

### 4. Evitar las clases con muchos métodos públicos

Las clases con demasiados métodos públicos suelen ser una señal de que **están tomando demasiada responsabilidad**. Cada clase debe tener una interfaz pública que sea **coherente con su propósito**. Si los métodos públicos no están alineados con esa responsabilidad, tal vez deberías repensar la clase.

#### Ejemplo:

public class OrderService

{

    public void CreateOrder() { ... }

    public void CancelOrder() { ... }

    public void CalculateDiscount() { ... }  // ¿Debería estar aquí?

    public void SendInvoice() { ... }  // ¿Debería estar aquí?

}

/ Claro y descriptivo.

En este caso, métodos como **CalculateDiscount** o **SendInvoice** podrían pertenecer a clases diferentes que se encarguen de esas funcionalidades de forma más específica.

### 5. Mantener la cohesión de la clase

Una clase debe tener **cohesión**, es decir, sus miembros (propiedades y métodos) deben estar relacionados entre sí. Si empiezas a meter funcionalidades no relacionadas dentro de una clase, pierdes la cohesión.

public class CustomerOrder

{

    public void PlaceOrder() { ... }

    public void CalculateDiscount() { ... }

    public void GenerateInvoice() { ... }

    public void SaveCustomerData() { ... }

}

### 6. Evitar la sobrecarga de getters y setters

A veces las clases se sobrecargan con **getters y setters** innecesarios. Si una propiedad tiene un valor calculado o está involucrada en una lógica más compleja, en vez de un simple getter/setter, es mejor usar métodos que expresen claramente la intención del código.

public class BankAccount

{

    private double \_balance;

    public double Balance => \_balance; // Esto es un getter simple.

    public void Deposit(double amount) { ... }  // Preferible usar métodos como estos.

}

### 7. Usar excepciones adecuadament**e**

Las clases deben ser responsables de manejar sus propios errores. Si una clase recibe parámetros o realiza una acción que puede fallar, debe manejar esas excepciones de forma adecuada o **propagar errores** hacia una capa superior.

### **Problemática con la herencia**

La herencia introduce un fuerte acoplamiento entre clases. Las clases hijas dependen estrechamente de la implementación de la clase base. Si la clase base cambia, las clases derivadas también pueden verse afectadas, lo que dificulta el mantenimiento y la extensión del código. Por ello se recomienda la composición frente a la herencia y usar interfaces para no utilizar la herencia directa. Veamos un ejemplo

class Notificador

{

    public void EnviarEmail(string mensaje)

    {

        Console.WriteLine($"Enviando Email: {mensaje}");

    }

}

class Pedido : Notificador

{

    public void ConfirmarPedido()

    {

        Console.WriteLine("Pedido confirmado.");

        EnviarEmail("Tu pedido ha sido confirmado."); // ❌ Dependencia directa

    }

}

Problema

* La clase Pedido está atada a Notificador
* No se puede cambiar fácilmente el mecanismo de notificación (cambiar email por sms).
* Si Notificador cambia su implementación, Pedido también se verá afectada.

Solución

* Bajo acoplamiento con inyección de dependencias
* En lugar de usar la herencia, usamos composición e interfaces para desacoplar las clases.

interface INotificador

{

    void Enviar(string mensaje);

}

class EmailNotificador : INotificador

{

    public void Enviar(string mensaje)

    {

  Console.WriteLine($"Enviando Email: {mensaje}");

    }

}

class Pedido

{

    private readonly INotificador \_notificador;

    public Pedido(INotificador notificador)

    {

        \_notificador = notificador;

    }

    public void ConfirmarPedido()

    {

        Console.WriteLine("Pedido confirmado.");

        \_notificador.Enviar("Tu pedido ha sido confirmado.");

    }

}

# Code Smells (STUPID)

Un Code Smell (olor a código) es una **señal de alerta** de que algo no va bien que sugiere un posible problema de el diseño o estructura. No impide que el programa funcione, pero puede ser un problema para el mantenimiento, legibilidad o la escalabilidad del código. Detectar Code Smell permite:

* Mejorar la calidad del código
* Hacer del código mas fácil de mantener

### **Singleton**

El patrón Singleton garantiza una única instancia de la clase a lo largo de toda la aplicación, y ese es su punto fuerte. Sin embargo tiene sus contras:

* + Vive en el contexto global y puede que no sepamos donde está
  + Puede ser modificado por cualquiera y en cualquier momento, por lo que es dificil de rastrear.
  + Dificil de testear debido a su ubicación

Veamos un ejemplo

public class Logger

{

    private static Logger \_instance;

    private Logger() { }

    public static Logger Instance

    {

        get

        {

            if (\_instance == null)

                \_instance = new Logger();

            return \_instance;

        }

    }

    public void Log(string message)

    {

        Console.WriteLine($"[LOG] {message}");

    }

}

class Program

{

    static void Main()

    {

  var logger = Logger.Instance;

        Logger.Instance.Log("Mensaje de prueba");

    }

}

- No se puede cambiar el comportamiento de “Logger” sin modificar la clase

- Al ser instancia única no puede testear facilmente (mocks)

- Otras clases están acopladas directamente a una implementación concreta, no a una abstracción

### **Tight Coupling: alto acoplamiento**

Lo que se busca en el desarrollo de software es que haya un *bajo acoplamiento* y una *alta cohesión*

Un **alto acoplamiento** significa que hay dependencias entre clases lo que provoca que cuando haya un cambio en una de ellas se produzca un efecto dominó de los cambios en otros módulos. Es decir:

* + “A” tiene un atributo que se refiere a “B”
  + “A” llama a los servicios de un objeto “B”
  + “A” tiene un método que hace referencia a “B” (a través del tipo de retorno o parámetro).
  + “A” es una subclase de (o implementa) la clase “B”

// Clase concreta B

public class B

{

    public void HacerAlgo()

    {

        Console.WriteLine("B está haciendo algo");

    }

}

// Clase A fuertemente acoplada a B

public class A : B  // Herencia directa (acoplamiento fuerte)

{

    private B \_b = new B();  // Atributo concreto

    public B ObtenerB()       // Método que retorna B

    {

        return \_b;

    }

    public void Procesar(B b) // Método que recibe B

    {

        b.HacerAlgo();        // Llamada directa a métodos de B

    }

    public void Ejecutar()

    {

        \_b.HacerAlgo();       // Uso directo de B

    }

}

Lo que se pretende es diseñar clases que sean autocontenidas, autosuficientes e independientes, con un objetivo y un propósito bien definido.

Una **baja cohesión** se refiere a lo que hace la clase, que realiza una gran cantidad de acciones y no se enfoca en su propósito. Por ejemplo, una clase “Usuario” que envie emails o calcule un impuesto…etc.

En definitiva, **lo ideal** es tener un **bajo acoplamiento** y una **alta cohesión.** Las clases no tienen que ser dependientes unas de otras y deben hacer sólo lo que el propósito de su nombre indica.

### **Untestability: código no probable**

Código no testeable es aquel que no se puede probar porque:

* + Usa clases concretas y no abstracciones
  + Crea dependencias (new) dentro del código
  + Tiene lógica mezclada con entrada/salida
  + Depende de datos externos (base de datos, archivos, etc) sin abstraerlos

Código no testeable

public class ClienteService

{

    public void RegistrarCliente()

    {

        Console.WriteLine("Ingrese el nombre del cliente:");

        string nombre = Console.ReadLine(); // 💥 Entrada directa

        var repo = new ClienteRepository(); // 💥 Creación directa de dependencia

        repo.Guardar(nombre);

        Console.WriteLine("Cliente registrado."); // 💥 Salida directa

    }

}

public class ClienteRepository

{

    public void Guardar(string nombre)

    {

        // Simula guardar en base de datos

        Console.WriteLine($"Guardando '{nombre}' en la base de datos...");

    }

}

Código testeable

public interface IEntrada

{

    string Leer();

}

public interface ISalida

{

    void Escribir(string mensaje);

}

public interface IClienteRepository

{

    void Guardar(string nombre);

}

// Implementación real de entrada/salida

public class ConsolaEntrada : IEntrada

{

    public string Leer() => Console.ReadLine();

}

public class ConsolaSalida : ISalida

{

    public void Escribir(string mensaje) => Console.WriteLine(mensaje);

}

public class ClienteService

{

    private readonly IEntrada \_entrada;

    private readonly ISalida \_salida;

    private readonly IClienteRepository \_repo;

    public ClienteService(IEntrada entrada, ISalida salida, IClienteRepository repo)

    {

        \_entrada = entrada;

        \_salida = salida;

        \_repo = repo;

    }

    public void RegistrarCliente()

    {

        \_salida.Escribir("Ingrese el nombre del cliente:");

        string nombre = \_entrada.Leer();

        \_repo.Guardar(nombre);

        \_salida.Escribir("Cliente registrado.");

}

}

### 

### **Premature optimization: optimizaciones prematuras**

Es **intentar hacer que el código sea más rápido o más eficiente antes de que sea necesario**, sin saber si realmente hay un problema de rendimiento.

Esto puede traer problemas como:

* Código más **complejo y difícil de mantener**.
* **Tiempo perdido** optimizando algo que ni siquiera era lento.

Versión con optimización prematura

public class Calculadora

{

    private Dictionary<int, long> \_cache = new Dictionary<int, long>();

    public long Factorial(int n)

    {

        if (\_cache.ContainsKey(n))

            return \_cache[n];

        long resultado = 1;

        for (int i = 2; i <= n; i++)

        {

            resultado \*= i;

        }

        \_cache[n] = resultado; // 💡 Optimización con memorización

        return resultado;

    }

}

Versión sin optimización prematura

public class Calculadora

{

    public long Factorial(int n)

    {

        long resultado = 1;

        for (int i = 2; i <= n; i++)

        {

            resultado \*= i;

        }

        return resultado;

    }

}

### **Innappropiate Intimacy: Intimidad inapropiada**

Dos clases **se conocen demasiado entre sí** o **dependen fuertemente una de otra**.  
Una clase accede directamente a los detalles internos de otra (campos privados, métodos internos, etc.), lo que **rompe el principio de encapsulamiento**.

¿Por qué es un problema?

* Hace que el código sea **frágil**: si cambias una clase, puedes romper la otra.
* **Aumenta el acoplamiento**: las clases no son independientes.
* **Dificulta el mantenimiento** y la evolución del software.

Ejemplo de Inappropriate Intimacy:

public class Empleado

{

    public decimal SueldoBase { get; set; }

    public decimal Bonificacion { get; set; }

}

public class CalculadoraDeImpuestos

{

    public decimal CalcularImpuesto(Empleado empleado)

    {

        // Accede directamente a los detalles internos del empleado

        return (empleado.SueldoBase + empleado.Bonificacion) \* 0.25m;

    }

}

Aquí, CalculadoraDeImpuestos sabe demasiado sobre la estructura interna de Empleado, lo que crea una intimidad inapropiada. Si el modelo de Empleado cambia (por ejemplo, si se cambia la forma de calcular el sueldo total), también hay que modificar CalculadoraDeImpuestos.

Ejemplo sin Inappropiate Intimacy

public class Empleado

{

    public decimal SueldoBase { get; set; }

    public decimal Bonificacion { get; set; }

    public decimal ObtenerSueldoTotal()

    {

        return SueldoBase + Bonificacion;

    }

}

public class CalculadoraDeImpuestos

{

    public decimal CalcularImpuesto(Empleado empleado)

    {

        // Usa una interfaz pública, no accede directamente a los detalles internos

        return empleado.ObtenerSueldoTotal() \* 0.25m;

    }

}

Solución

Aplicar encapsulamiento. Hacer que **Empleado se responsabilice de exponer su información** de forma segura.

### **Duplicate Code (Código duplicado)**

**¿Qué significa?**  
Hay **código repetido** en varios lugares. La duplicación puede ser de lógica, estructuras de datos, validaciones, etc.

**¿Por qué es un problema?**

* Si necesitas cambiar algo, debes hacerlo en **todos los lugares** donde está duplicado.
* **Aumenta el riesgo de errores**: puedes olvidar actualizar una copia.
* **Hace que el código sea más largo** y **difícil de mantener**.

public double CalculateAreaRectangle(double width, double height)

{

    return width \* height;

}

public double CalculateArea(double width, double height)

{

    return width \* height;

}

## **Principios SOLID**

### **¿Qué son los Principios SOLID?**

**SOLID** es un conjunto de **cinco principios de diseño** para escribir **código de calidad**, que sea:

* **Fácil de entender**
* **Fácil de mantener**
* **Fácil de extender**
* **Más robusto ante cambios**

Estos principios ayudan a construir **software flexible, limpio y de bajo acoplamiento**, muy importante en proyectos medianos y grandes.

### **¿De dónde vienen los principios SOLID?**

* **Fueron popularizados por Robert C. Martin** (conocido como "**Uncle Bob**") en su libro "Agile Software Development: Principles, Patterns, and Practices".
* Después, **Michael Feathers** organizó las primeras letras de cada principio y creó el acrónimo **SOLID**.
* Nacieron como respuesta a los problemas típicos en el desarrollo de software, como:
  + Código difícil de modificar
  + Software frágil que se rompe con cualquier cambio
  + Programas imposibles de escalar sin rehacer muchas cosas

**Objetivo:** Que el código sea **fácil de cambiar sin romper** el sistema.

### **¿Qué significa cada letra de SOLID?**

| **Letra** | **Principio** | **Qué dice** |
| --- | --- | --- |
| **S** | **Single Responsibility Principle** (Responsabilidad Única) | Cada clase debe tener una sola razón para cambiar. |
| **O** | **Open/Closed Principle** (Abierto/Cerrado) | El código debe estar abierto para extensión, pero cerrado para modificación. |
| **L** | **Liskov Substitution Principle** (Sustitución de Liskov) | Los objetos derivados deben poder reemplazar a sus padres sin romper el programa. |
| **I** | **Interface Segregation Principle** (Segregación de Interfaces) | No obligues a implementar interfaces grandes; usa varias interfaces pequeñas y específicas. |
| **D** | **Dependency Inversion Principle** (Inversión de Dependencias) | Las clases deben depender de abstracciones, no de implementaciones concretas. |

### **Single Responsibility: Responsabilidad Única**

Una clase solo debe tener una responsabilidad, pero eso no significa que deba hacer una única cosa.

¿Por qué es importante?

* Si una clase tiene **muchas responsabilidades**, entonces **cualquier cambio** en una responsabilidad puede **romper o afectar** otras.
* Las clases **con una sola responsabilidad** son:
  + Más **fáciles de entender**,
  + Más **fáciles de probar**,
  + Más **fáciles de mantener**,
  + Más **resistentes a los cambios**.

A continuación, se presenta un código donde la clase ProductBloc tiene múltiples responsabilidades, violando el principio SRP

using System;

namespace ProductApp

{

    // Interfaz equivalente

    public interface IProduct

    {

        int Id { get; set; }

        string Name { get; set; }

    }

    // Clase que representa un producto

    public class Product : IProduct

    {

        public int Id { get; set; }

        public string Name { get; set; }

    }

    // Clase que maneja la lógica relacionada al producto (equivalente a ProductBloc)

    public class ProductBloc

    {

        public void LoadProduct(int id)

        {

            // Simulación de obtener producto

            Console.WriteLine($"Producto: Id={id}, Name=OLED Tv");

        }

        public void SaveProduct(Product product)

        {

            // Simulación de guardar en base de datos

            Console.WriteLine($"Guardando en base de datos: Id={product.Id}, Name={product.Name}");

        }

        public void NotifyClients()

        {

            Console.WriteLine("Enviando correo a los clientes");

        }

        public void OnAddToCart(int productId)

        {

            Console.WriteLine($"Agregando al carrito: {productId}");

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            var productBloc = new ProductBloc();

            productBloc.LoadProduct(10);

            productBloc.SaveProduct(new Product { Id = 10, Name = "OLED TV" });

            productBloc.NotifyClients();

            productBloc.OnAddToCart(10);

        }

    }

}

La clase ProductBloc está **violando el SRP** porque **mezcla muchas responsabilidades diferentes**.

Mira lo que hace ProductBloc:

* LoadProduct(int id): carga un producto -> **Responsabilidad de lectura o recuperación de datos**.
* SaveProduct(Product product): guarda un producto -> **Responsabilidad de persistencia**.
* NotifyClients(): envía correos a clientes -> **Responsabilidad de notificaciones**.
* OnAddToCart(int productId): agrega un producto al carrito ➡️ **Responsabilidad de gestión de carrito de compras**.

¿Qué problema causa esto?

* Si cambia **cómo se cargan los productos** (por ejemplo, desde una API en lugar de base de datos), tienes que tocar ProductBloc.
* Si cambia **cómo se envían notificaciones** (por ejemplo, integrar otro servicio de email), también tienes que tocar ProductBloc.
* Si cambia **cómo se maneja el carrito**, otra vez a modificar ProductBloc.

**¡Todo está amarrado y mezclado!**

Versión que si cumple con SRP

using System;

namespace ProductApp

{

    public interface IProduct

    {

        int Id { get; set; }

        string Name { get; set; }

    }

        // Clase que representa un producto

    public class Product : IProduct

    {

        public int Id { get; set; }

        public string Name { get; set; }

    }

    public class ProductService

    {

        public void LoadProduct(int id)

        {

            Console.WriteLine($"Producto: Id={id}, Name=OLED Tv");

        }

        public void SaveProduct(Product product)

        {

            Console.WriteLine($"Guardando en base de datos: Id={product.Id}, Name={product.Name}");

        }

    }

    public class EmailNotifier

    {

        public void NotifyClients()

        {

            Console.WriteLine("Enviando correo a los clientes");

        }

    }

    public class ShoppingCart

    {

        public void AddToCart(int productId)

        {

            Console.WriteLine($"Agregando al carrito: {productId}");

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            var productService = new ProductService();

            var emailNotifier = new EmailNotifier();

            var shoppingCart = new ShoppingCart();

            productService.LoadProduct(10);

            productService.SaveProduct(new Product { Id = 10, Name = "OLED TV" });

            emailNotifier.NotifyClients();

            shoppingCart.AddToCart(10);

        }

    }

}

Ahora, cada clase tiene una única responsabilidad:

| **Clase** | **Responsabilidad** |
| --- | --- |
| ProductService | Gestionar productos (cargar y guardar) |
| EmailNotifier | Enviar correos |
| ShoppingCart | Administrar el carrito de compras |

El problema con el primer código es que ProductBloc tiene varias responsabilidades distintas (datos, notificaciones, carrito).

Según el **Principio de Responsabilidad Única (SRP)**, cada clase debería tener una sola razón para cambiar.

Dividimos el código en tres clases, cada una encargándose de **un solo tipo de tarea**, logrando que el código sea **más mantenible, flexible y menos propenso a errores**.

|  |  |
| --- | --- |
|  |  |
| **ProductBloc** hace **demasiadas cosas**: acceso a datos, notificación de clientes, gestión del carrito.  **¡Mala separación de responsabilidades!** | Cada clase **tiene una única responsabilidad**.  **ProductService** = Gestiona productos.  **EmailNotifier** = Envía correos.  **ShoppingCart** = Maneja el carrito de compras.  ¡Ahora sí cumple el **Principio de Responsabilidad Única**! |

### **Open Close Principle: Principio de Abierto y Cerrado**

Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para su extensión, pero cerradas para su modificación.

Esto quiere decir:

* Puedes **agregar nuevo comportamiento** a un sistema,
* **Sin tener que cambiar el código existente**.

¿Por qué es importante?

* Si **modificas** código que ya funciona, puedes **romper** cosas que antes estaban bien.
* **Extender** es más seguro: creas algo nuevo **sin tocar** lo que ya funciona.
* Permite que tu software sea **más mantenible**, **más estable** y **más flexible**.

Ejemplo sin cumplir el princio de abierto y cerrado

using RestSharp;

using System.Threading.Tasks;

public class TodoService

{

    private readonly RestClient \_client;

    public TodoService()

    {

        \_client = new RestClient("https://jsonplaceholder.typicode.com");

    }

    public async Task<string> GetTodoItems()

    {

        var request = new RestRequest("todos", Method.Get);

        var response = await \_client.ExecuteAsync(request);

        if (response.IsSuccessful)

        {

            return response.Content;

        }

        throw new Exception($"Error: {response.StatusCode} - {response.ErrorMessage}");

    }

}

public class PostService

{

    private readonly RestClient \_client;

    public PostService()

    {

        \_client = new RestClient("https://jsonplaceholder.typicode.com");

    }

    public async Task<string> GetPosts()

    {

        var request = new RestRequest("posts", Method.Get);

        var response = await \_client.ExecuteAsync(request);

        if (response.IsSuccessful)

        {

            return response.Content;

        }

        throw new Exception($"Error: {response.StatusCode} - {response.ErrorMessage}");

    }

}

public class PhotosService

{

    private readonly RestClient \_client;

    public PhotosService()

    {

        \_client = new RestClient("https://jsonplaceholder.typicode.com");

    }

public async Task<string> GetPhotos()

{

    var request = new RestRequest("photos", Method.Get);

    var response = await \_client.ExecuteAsync(request);

        if (response.IsSuccessful)

        {

            return response.Content;

        }

        throw new Exception($"Error: {response.StatusCode} - {response.ErrorMessage}");

    }

}

En este ejemplo se utiliza la librería “RestSharp” para consumir servicios REST. En el código tenemos tres clases (servicios) que utilizan la librería para hacer peticiones Get. Si pasado un tiempo se quisiera cambiar de proveedor de servicios REST violentariamos el principio de abierto y cerrado, porque los tres servicios tienen dependencia directa con la librería RestSharp.

Para que nuestro código cumpla con el principio de abierto y cerrado tenemos que sacar a una clase aparte la referencia a la librería Restsharp encapsulando su funcionalidad.

Ejemplo cumpliendo el principio de abierto y cerrado

using RestSharp;

using System;

using System.Threading.Tasks;

public class RestShapApiService : IApiService

{

    private readonly RestClient \_client;

    public RestShapApiService(RestClient client)

    {

        \_client = client;

    }

    public async Task<string> GetAsync(string endpoint)

    {

        var request= new Request(endpoint, Method.Get);

        var response= await \_cliente.ExecuteAsyn();

        if (response.IsSuccess)

        {

            return response.Content;

        }

throw new Exception($"Error {response.StatusCode} - {response.ErrorMessage}");

    }

}

Ahora nuestros tres servicios quedan de la siguiente manera

using RestSharp;

using System.Threading.Tasks;

public class TodoService

{

    private readonly IApiService \_apiService;

    public TodoService(IApiService apiService)

    {

        \_apiService = apiService;

    }

    public async Task<string> GetTodoItems()

    {

        return awary await \_apiService.GetAsync("Todos");

    }

}

public class PostService

{

    private readonly IApiService \_apiService;

    public PostService(IApiService apiService)

    {

        \_apiService = apiService;    }

    public async Task<string> GetPosts()

    {

         return await \_apiService.GetAsync("posts");

    }

}

public class PhotosService

{

    private readonly IApiService \_apiService;

    public PhotosService(IApiService apiService)

    {

       \_apiService = apiService;

    }

    public async Task<string> GetPhotos()

    {

        return await \_apiService.GetAsync("photos");

    }

}

Si decidiesemos cambiar de proveedor para llamar a la API solo habria que cambiar nuestro servicio creado, las clases no se verian afectadas ya que ahora no tienen dependencias directas a la liberia RestSharp.

### **Liskov’s Sustitution Principle: Principio de Sustitución de Liskov**

*Si una clase* ***hija*** *hereda de una clase* ***padre****, debería* ***poder usarse donde se espera la clase padre******sin que el programa se rompa ni cambie su comportamiento esperado***.

Cuando aplicas el **Principio de Sustitución de Liskov (LSP)**, debes pensar en estos puntos:

1.- Comportarse de forma coherente

* La **subclase** debe **mantener el mismo tipo de comportamiento general** que se espera de la clase padre.
* No puede **hacer cosas raras** o **contradecir** lo que el padre promete.

Ejemplo:

* Si Bird promete "puedo moverme", el Penguin no puede decir "me muevo solo a veces" o lanzar un error.
* Todos los pájaros deben poder **moverse** de alguna forma.

2.- No romper expectativas

* Cuando alguien usa una **clase base**, ya tiene en mente cómo debería **funcionar**.
* Si pasa una subclase, esa **expectativa debe seguir siendo cierta**.

Ejemplo:

* Si alguien crea un Bird y llama Move(), espera que **algo suceda**, no que el programa falle o que haga algo completamente distinto.

3.- No cambiar reglas

* La subclase **no debería cambiar** las **reglas de negocio** definidas por la clase base.
* Tiene que **respetar** lo que la clase base espera.

Ejemplo:

* Si Account tiene una regla de "no permitir saldo negativo", una subclase SavingsAccount no puede ignorarla y permitir saldo negativo, porque **está rompiendo la lógica esperada**.

4.- No lanzar errores inesperados

* Si la clase base **no lanza errores** en su comportamiento normal, la subclase **tampoco debería hacerlo** en condiciones normales.

Ejemplo:

* Un Bird normal **no lanza excepciones** cuando vuela.
* Si Penguin lanza NotSupportedException en Fly(), el programa puede romperse porque **no esperaba un error** allí.

Ejemplo de violación del principio de Liskov’s

namespace LiskovViolation

{

    public class Tesla

    {

        private int numberOfSeats;

        public Tesla(int seats) => numberOfSeats = seats;

        public int GetNumberOfTeslaSeats() => numberOfSeats;

    }

    public class Audi

    {

        private int numberOfSeats;

        public Audi(int seats) => numberOfSeats = seats;

        public int GetNumberOfAudiSeats() => numberOfSeats;

    }

    public class Toyota

    {

        private int numberOfSeats;

        public Toyota(int seats) => numberOfSeats = seats;

        public int GetNumberOfToyotaSeats() => numberOfSeats;

    }

    public class Honda

    {

        private int numberOfSeats;

        public Honda(int seats) => numberOfSeats = seats;

        public int GetNumberOfHondaSeats() => numberOfSeats;

    }

}

class Program

{

    static void Main(string[] args)

    {

        var cars = new List<object>

        {

            new Tesla(7), new Audi(2),  new Toyota(5),  new Honda(5)

        };

        PrintCarSeats(cars);

    }

    static void PrintCarSeats(List<object> cars)

    {

        foreach (var car in cars)

        {

            if (car is Tesla tesla)

            {

                Console.WriteLine($"Tesla: {tesla.GetNumberOfTeslaSeats()}");   continue;

            }

            if (car is Audi audi)

            {

                Console.WriteLine($"Audi: {audi.GetNumberOfAudiSeats()}");    continue;

            }

            if (car is Toyota toyota)

            {

                Console.WriteLine($"Toyota: {toyota.GetNumberOfToyotaSeats()}");    continue;

            }

            if (car is Honda honda)

            {

                Console.WriteLine($"Honda: {honda.GetNumberOfHondaSeats()}");  continue;

            }

        }

    }

}

Tenemos varias clases que representan marcas de vehículos. Todas ellas tienen un método que indica el número de asientos que tiene cada marca. La clase que arranca el programa tiene un array de objetos que almacena un objeto de cada marca de vehículos indicando el número de asientos. Posteriormente llama a un metodo que recorre el array de objetos imprimiendo el número de asientos de cada marca.

Este código viola el Principio de Sustitución de Liskov’s porque el método “PrintCarsSeats” debe conocer los detalles de cada clase para funcionar. Además viola el Principio de Abierto y Cerrado porque si se añade una nueva marca, hay que modificar el método.

Ejemplo que cumple el Principio de Sustitución de Liskov’s

namespace Liskov

{

    public abstract class Vehicle

    {

        public int NumberOfSeats { get; }

        protected Vehicle(int numberOfSeats)

        {

            NumberOfSeats = numberOfSeats;

        }

        public int GetNumberOfSeats()

        {

            return NumberOfSeats;

        }

    }

    // Clases derivadas

  public class Tesla : Vehicle

    {

  public Tesla(int seats) : base(seats) { }

    }

    public class Audi : Vehicle

    {

        public Audi(int seats) : base(seats) { }

    }

    public class Toyota : Vehicle

    {

        public Toyota(int seats) : base(seats) { }

    }

    public class Honda : Vehicle

    {

        public Honda(int seats) : base(seats) { }

    }

    class Program

    {

        static void Main()

        {

            var cars = new List<Vehicle>      {

                new Tesla(7),

                new Audi(2),

                new Toyota(5),

                new Honda(5)

            };

            PrintCarSeats(cars);

        }

        static void PrintCarSeats(List<Vehicle> cars)

        {

            foreach (var car in cars)  { Console.WriteLine($"{car.GetType().Name}, {car.GetNumberOfSeats()}"); }

        }

    }

}

Creamos una clase abstracta denominada vehículo que va a tener un método “GetNumberOfSeats” que va a devolver el número de asientos de cualquier marca. Para ello derivamos de la clase “Vehículo” todas las clases que representan las marcas. Estas clases ya no tienen el metodo específico que indica el número de asientos, no hace falta, porque lo heredan de “Vehicle”. Además cumple con el principio de Abierto y Cerrado ya que puedo añadir mas marcas sin tener que modificar el metodo que imprime las marcas

### **Interfaz Segregation: Segregación de Interfaz**

Los clientes no deberían verse obligados a depender de interfaces que no utilizan

¿Qué significa?

En lugar de tener una sola interfaz grande que obligue a las clases a implementar muchos métodos (algunos innecesarios) es mejor tener interfaces más pequeñas y específicas que agrupen comportamientos relacionados.

¿Por qué es importante?

* + Hace que el sistema sea más flexible y mantenible.
  + Favorece el bajo acoplamiento y la alta cohesión.

Ejemplo que viola el principio

public interface ITrabajador

{

    void Trabajar();

    void Comer();

}

public class Empleado : ITrabajador

{

    public void Trabajar()

    {

        Console.WriteLine("El empleado está trabajando.");

    }

    public void Comer()

    {

        Console.WriteLine("El empleado está comiendo.");

    }

}

public class Robot : ITrabajador

{

    public void Trabajar()

    {

        Console.WriteLine("El robot está trabajando.");

    }

    public void Comer()

    {

        throw new NotImplementedException();

    }

}

Tanto la clase Empleado como la clase Robot implementan la interfaz ITrabajador.

Esto viola el principio porque obliga tanto al empleado como al robot a trabajar y a comer, sin embargo un robot no come. El robot no deberia implementar ese método.

Código que no viola el principio

public interface ITrabajador

{

    void Trabajar();

}

public interface IComedor

{

    void Comer();

}

public class Empleado : ITrabajador, IComedor

{

    public void Trabajar()

    {

        Console.WriteLine("El empleado está trabajando.");

    }

    public void Comer()

    {

        Console.WriteLine("El empleado está comiendo.");

    }

}

public class Robot : ITrabajador

{

    public void Trabajar()

    {

        Console.WriteLine("El robot está trabajando.");

    }

}

Creamos una interface para el comportamiento de comer y ese comportamiento no se lo asignos al Robot, mientras que el trabajador sigue implementando el comportamiento de trabajar y comer.

### **Dependency inversion: Inversión de dependencias**

Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.  
Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.

¿Qué significa?

* + Los módulos de alto nivel (como la lógica de negocio) no deberían conocer los detalles de los **módulos de bajo nivel** (como una clase que accede a la base de datos).
  + En lugar de eso, ambos deben comunicarse a través de **interfaces o abstracciones**
  + Así, puedes cambiar la implementación sin tocar el módulo de alto nivel, lo cual favorece la flexibilidad y la reutilización.

Ejemplo de código que no cumple con el principio

public class MySqlRepository

{

    public void Save(string data)

    {

        Console.WriteLine($"Guardando '{data}' en MySQL");

    }

}

public class BusinessService

{

    private MySqlRepository \_repository = new MySqlRepository();

    public void Process()

    {

        // Lógica de negocio

        \_repository.Save("Datos importantes");

    }

}

En este código podemos observar que la Clase BusinessService tiene una fuerte dependencia con la clase MySqlRepository, ya que crea un instancia de la misma para guardar datos en el repositorio.

El problema está en que si el día de mañana yo decido cambiar de MySql a otro proveedor de base de datos, mi clase de negocio se ve afectada. Si esto se repite en un proyecto grande, es un problema.

Ejemplo de código que si aplica el principio

public interface IRepository

{

    void Save(string data);

}

public class MySqlRepository : IRepository

{

    public void Save(string data)

    {

        Console.WriteLine($"Guardando '{data}' en MySQL");

    }

}

public class BusinessService

{

    private readonly IRepository \_repository;

    public BusinessService(IRepository repository)

    {

        \_repository = repository;

    }

    public void Process()

    {

        \_repository.Save("Datos importantes");

    }

}

Se crea una abstracción con la interface IRepositorio que será implementado por la clase MySqlRepository, y en la clase de negocio BusinessService le pasamos por parámetro la interfaz (inyección de dependencias).

Esto permite que si se cambia de base de datos, mi capa de negocio no se vea afectada.