**Diapositiva 2**

La Programación Orientada a Objetos (POO) se popularizó en los años 80 y 90 por una combinación de factores técnicos, culturales y económicos que la hicieron muy atractiva para resolver los problemas crecientes del desarrollo de software. Aquí te explico las principales razones:

* 1 - La programación estructurada mostraba limitación en los sistemas complejos
* 2 - Éxito de lenguajes como **C++** (años 80) que fue una evolución de C que incorporó orientación a objetos. Los desarrolladores hicieron la transición sin abandonar sus bases
* 3 - Durante los años 90, empresas como Microsoft, IBM y Sun Microsystems adoptaron e impulsaron la orientación a objetos. Microsoft promovió el uso de C++ en sus productos, y Sun lanzó **Java** en 1995, un lenguaje orientado a objetos diseñado para internet y multiplataforma.
* 4 - La POO introdujo conceptos como **herencia**, **polimorfismo** y **encapsulamiento**, que favorecen la reutilización del código y facilitan el mantenimiento a largo plazo. Estos conceptos respondían a necesidades reales en la industria.

**Diapositiva 3**

Los **principios SOLID** nacieron como una respuesta a los problemas que los programadores empezaban a sufrir a medida que los sistemas de software se volvían **más grandes, más complejos y más difíciles de mantener** ya que los programas **se volvían frágiles, rígidos y difíciles de cambiar:**

* **Frágiles**: Cambiar una parte del código rompía otras partes inesperadamente.
* **Rígidos**: Era muy difícil hacer cambios porque todo estaba demasiado acoplado.
* **Inmutables**: Costaba mucho extender o evolucionar el sistema sin modificar grandes partes del código.

**Diapositiva 4**

Para solucionar estos problemas, **Robert C. Martin** (más conocido como "Uncle Bob") estudió qué prácticas ayudaban a crear **software más limpio, flexible y mantenible**, y en los años 90 agrupó cinco principios fundamentales, que juntos forman el acrónimo **SOLID**:

* **S**: *Single Responsibility Principle* (Principio de Responsabilidad Única)
* **O**: *Open/Closed Principle* (Principio de Abierto/Cerrado)
* **L**: *Liskov Substitution Principle* (Principio de Sustitución de Liskov)
* **I**: *Interface Segregation Principle* (Principio de Segregación de Interfaces)
* **D**: *Dependency Inversion Principle* (Principio de Inversión de Dependencias)

**Diapositiva 5**

**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.

**Diapositiva 6**

### **¿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.

**Diapositiva 7**

*Esquema de la deuda técnica de Martin Fowler*

Tabla

El contenido generado por IA puede ser incorrecto.

* 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 cómo 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 más adecuado para los programadores, así como una reducción de incidencias y tener más tiempo para el desarrollo.

**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**.

**Diapositiva 8**

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.

**Diapositiva 9**

### **Cómo nombrar variables**

**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.

**Diapositiva 10**

**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

**Diapositiva 11**

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**.

**Diapositiva 12**

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**.

**Diapositiva 13**

¿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**.

**Diapositiva 14**

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.

**Diapositiva 15**

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 configuración.

- db en vez de database, que puede no ser claro en todos los contextos.

**Diapositiva 16**

¿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.

**Diapositiva 17**

Uso de mayúsculas incorrectas

**Diapositiva 18**

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

**Diapositiva 19**

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

**Diapositiva 20 y 21**

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

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

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

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

**Diapositiva 22 Y 23**

### 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.

Refactorización (separa responsabilidades)

**Diapositiva 24**

### 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.

### Usar excepciones adecuadamente

### 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.

**Diapositiva 25**

### 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.

}

**Diapositiva 26**

### **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

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.

**Diapositiva 27 Y 28**

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

**Diapositiva 29**

Un Code Smell (olor a código) es una señal de alerta de que algo no va bien. 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 más fácil de mantener

**Diapositiva 30**

*El patrón Singleton garantiza una única instancia de la clase a lo largo de toda la aplicación*

***Problema***

* Vive en el contexto global y puede que no sepamos done está
* Puede ser modificado por cualquiera y en cualquier momento. Difícil de rastrear.
* No se puede cambiar el comportamiento de “Logger” sin modificar la clase
* Al ser instancia única no puede testear fácilmente
* Otras clases están acopladas directamente a una implementación concreta, no a una abstracción

**Diapositiva 31**

**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”

**Diapositiva 32**

Ejemplo de alto acoplamiento

**Diapositiva 33**

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.

**Diapositiva 34**

### **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

**Diapositiva 35**

Ejemplo de código no testeable

**Diapositiva 36**

Ejemplo de código testeable

**Diapositiva 37**

### **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.

**Diapositiva 38**

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.

**Diapositiva 39**

Ejemplo de intimidad inapropiada

**Diapositiva 40**

Ejemplo de Inappropriate Intimacy:

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

Solución

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

**Diapositiva 41**

### **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**.