

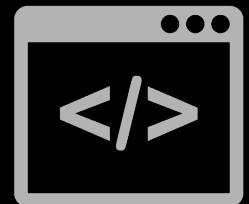
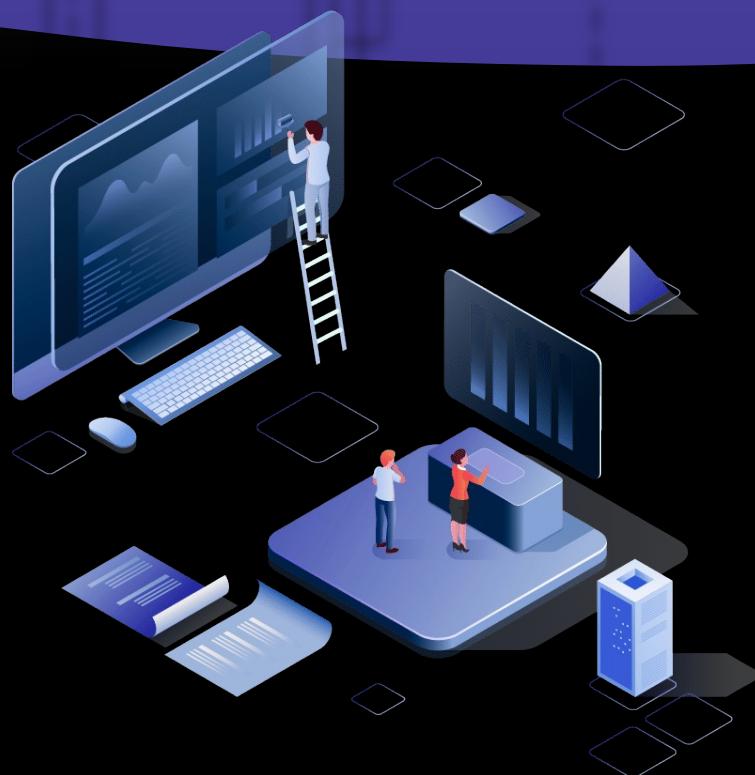
# Diseño de Sistemas



# Agenda

- Principios SOLID
- Refactoring
- Code Smells
- Entities vs Value Objects
- Biblioteca vs Framework
- Inyección de Dependencias

# Principios SOLID

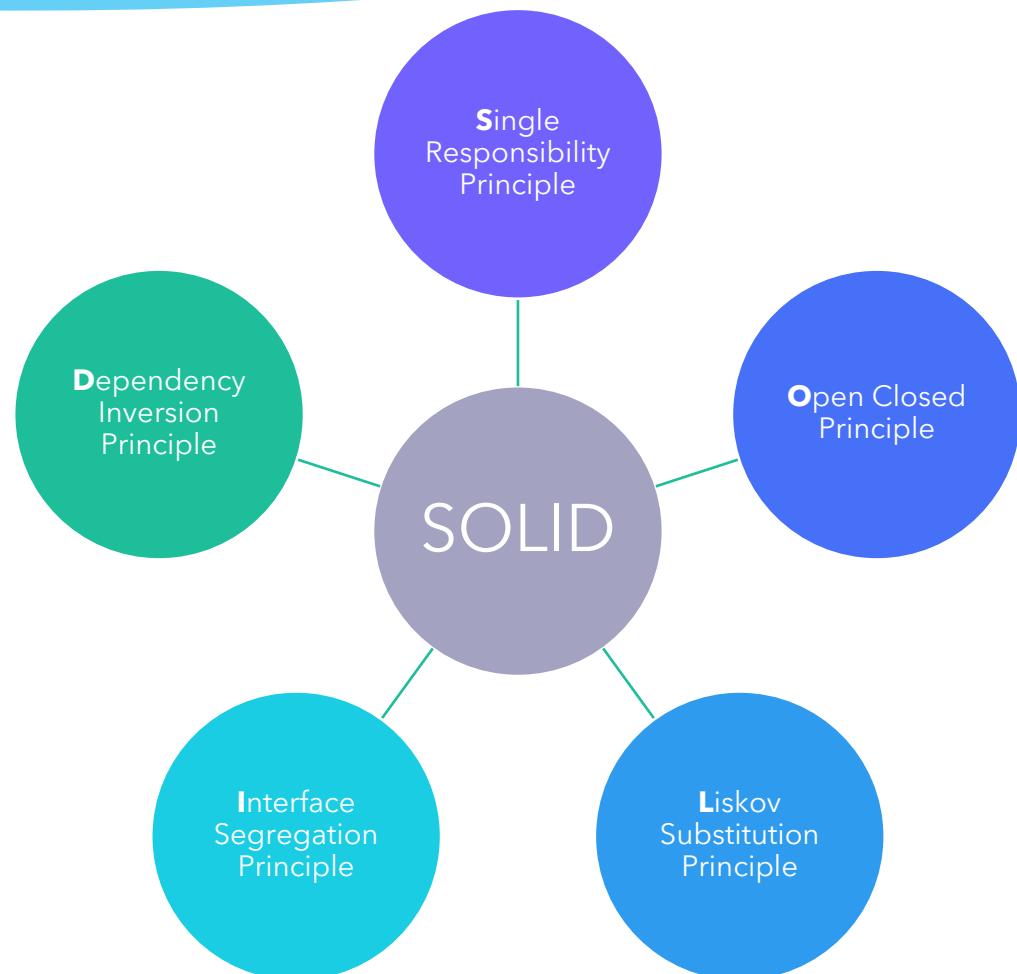


# Principios SOLID

Son principios básicos de Programación Orientada a Objetos y Diseño de Sistemas que nos ayudan a obtener mejores diseños implementando una serie de reglas o principios.

*Nos ayudan a evitar la generación de "Código Sucio"*

# Principios SOLID



# SOLID - Single Responsibility Principle

- *Cada clase debe tener responsabilidad sobre una sola parte de la funcionalidad del software -> **Clases Cohesivas***
- *Esta responsabilidad debe estar encapsulada por la clase, y todos sus servicios deben estar estrechamente alineados con esa responsabilidad.*

***Evitar la clase “Dios” propiciando la alta Cohesión***

# SOLID - Open Closed Principle

- Las entidades deben estar abiertas para la expansión, pero cerradas para su modificación.
- Se basa en la implementación de herencias y el uso de interfaces para resolver el problema.

**Se sugiere evitar la utilización excesiva de los “switches” y propiciar el polimorfismo entre objetos**

# SOLID - Liskov Substitution Principle

*"Cada clase que hereda de otra puede usarse como su superclase sin necesidad de conocer las diferencias entre las clases derivadas. Lo mismo vale para las realizaciones de Interfaces"*

***Se debería utilizar correctamente la herencia y las realizaciones de Interfaces***

# SOLID – Interface Segregation Principle

- Los clientes de un componente sólo deberían conocer de éste aquellos métodos que realmente usan y no aquellos que no necesitan usar.
- Muchas interfaces cliente específicas son mejores que una interfaz de propósito general.

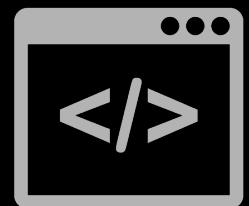
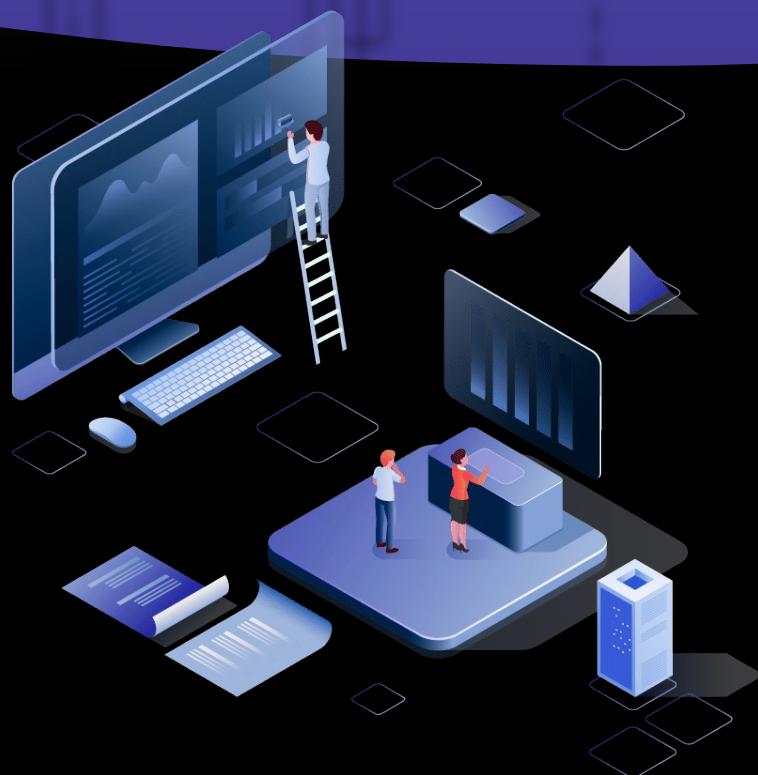
**Se debería propiciar un diseño orientado a interfaces, para mantener el acoplamiento entre clases al mínimo posible, y también evitar generar interfaces extensas (con muchos métodos)**

# SOLID – Dependency Inversion Principle

*"Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones. Es una forma de desacoplar módulos."*

***Se sugiere utilizar inyectores de dependencias***

# Refactoring



# Refactoring

*¿Qué No es refactorizar?*

- *Refactorizar no es incorporar funcionalidades nuevas*
- *Refactorizar no es optimizar*

# Refactoring

"Es **cambiar la estructura del código** para hacerlo más **simple** y **extensible**. Esto involucra muchas acciones posibles, algunas simples como cambiar el nombre de una clase o un método, o cosas más complejas como reemplazar condicionales (ifs) por objetos polimórficos o bien separar una clase en varias."

# Refactoring

El refactor busca:

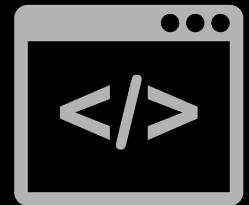
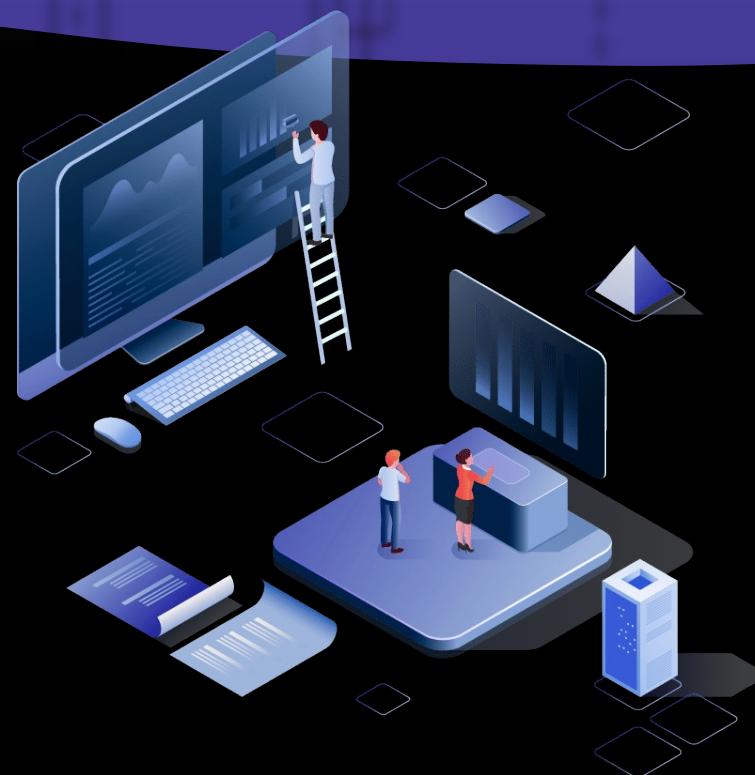
- **Métodos cortos** y con **nombres bien definidos**: que revelen exactamente el propósito para el que fueron creados
- **Métodos** y **clases** con **responsabilidades claras** y bien definidas (mayor cohesión)
- Respecto a las variables de instancia: hay que evaluar el costo-beneficio de tener en **variables valores que puedan calcularse**, como el total de deuda de un cliente, o la cantidad de hijos de un empleado

# Refactoring

El refactor busca:

- **No** tener "**god objects**" ni "**managers**", objetos que roban responsabilidades que les corresponden a otros objetos.
- Es preferible tener **objetos chicos** antes que un objeto grande con muchas responsabilidades (sobre todo si los objetos chicos pueden intercambiarse).
- **Evitar ciclos de dependencia** entre objetos si no los necesito; esto es relaciones bidireccionales innecesarias. Ej: cada cliente conoce a sus facturas, la factura ¿necesita conocer al cliente?

# Code Smells



# Code Smells

- Los Code Smells, como su traducción nos dice, son "**olores de código**".
- Los smells no necesariamente significan que esté mal lo que diseñé/implementé, sino que son señales de algo que se puede mejorar (no hay que ser terminante, muchos code smells se entrecruzan al tratar de solucionarlos).
- Nosotros debemos ser capaces de sentir estos "malos olores" en nuestro código para que, de esta forma, se pueda mejorar nuestro diseño.

Veamos algunos...

# Código duplicado

```
public void trotar(int metros){  
    this.energía = this.energía - 10;  
    this.animo = this.animo - 3;  
    this.posición = new Poi(this.posición.getX()+metros,  
    this.posición.getY()+metros);  
}
```

```
public void caminar(int metros){  
    this.energía = this.energía - 1;  
    this.animo = this.animo -1;  
    this.posición = new Poi(this.posición.getX()+metros,  
    this.posición.getY()+metros);  
}
```

# Código duplicado

```
public void disminuirEnergia(int unidades){  
    this.energía -= unidades;  
}
```

```
public void disminuirAnimo(int unidades){  
    this.animo -= unidades;  
}
```

```
public void desplazar(int metros){  
    this.posición = new Poi(this.posición.getX()+metros,  
    this.posición.getY()+metros);  
}
```

# Código duplicado

```
public void trotar(int metros){  
    this.disminuirEnergia(10);  
    this.disminuirAnimo(3);  
    this.desplazar(metros);  
}
```

```
public void caminar(int metros){  
    this.disminuirEnergia(1);  
    this.disminuirAnimo(1);  
    this.desplazar(metros);  
}
```

# Código duplicado

- No queremos duplicar la misma idea en el código. Dos lemas que nos acompañan son:
  - Once and only once: hacer las cosas una sola vez
  - Don't repeat yourself (DRY)
- Y esto no sólo vale para el código, también aplica para el diseño en general. Dos de las formas más conocidas para evitar este smell en diseño es la composición y la herencia.
- En el ejemplo anterior, todavía se podría mejorar aun más el diseño en general y evitar la duplicación lógica con ayuda del patrón Template Method, por ejemplo.

# Métodos largos

- Un método largo podemos descomponerlo en varias partes.
- De cada parte del método debemos:
  - identificar el objetivo que cumple
  - implementar esa abstracción: ponerle un nombre representativo y encontrar qué objeto es responsable de ese objetivo.
- Los métodos generados pueden ser utilizados en otro contexto.
- Delegar el método original en varios submétodos nos permite entender mejor qué es lo que hace.

# God Class (Clase Dios)

- Las clases por lo general comienzan siendo pequeñas, pero con el tiempo se hinchan a medida que el Sistema crece. Es decir, la clase hace “de todo”.
- Son clases que poseen demasiadas responsabilidades, que están fuertemente acopladas a muchos otros objetos y que, por consecuencia, ante cualquier cambio se ven afectadas.

# God Class (Clase Dios)

Para solucionar este Smell se puede:

- **Generar nuevas clases:** si el comportamiento de la clase es grande, y cumple con varias funciones al mismo tiempo, se debe dividir la God Class en más clases que cumplan con un solo propósito. -> Alta cohesión
- **Generar subclases:** Si parte del comportamiento de la God Class se puede implementar de diferentes maneras, entonces una solución es convertir a nuestra vieja God Class en una clase Abstracta y luego generar subclases que definan el comportamiento específico necesario.
- **Generar interfaces:** Si lo que queremos es brindar una lista de operaciones con diferentes comportamientos que el cliente puede utilizar, podemos generar una/varias interface(s) y luego crear clases concretas que cumplan con dicha interfaz e implementen las diferentes funcionalidades.

# God Class (Clase Dios)

Si la God Class es responsable de una interfaz gráfica, se debe comenzar a pensar en el patrón MVC, e ir descomponiendo las responsabilidades entre las diferentes capas involucradas.

# Parámetros largos

- Puede darse cuando varios algoritmos se combinan en un solo método. O también cuando nos estamos olvidando de alguna abstracción del dominio.
- Si un objeto va a brindar un servicio a otro, lo mejor sería que la cantidad de parámetros del método en cuestión no varíe.

# Parámetros largos

Por ejemplo...

```
>> CatalogoDePelis
public void getPelicula(String nombrePeli, String autor, int anio, int versión, int
estrellas){....}
```

Este método está totalmente sujeto a esos parámetros, y cualquier cambio en alguno de ellos afectaría a todo nuestro código. Del mismo modo, al querer agregar un parámetro más, estaríamos “rompiendo” con el cliente que está utilizando este servicio.

Una posible solución:

```
public void getPelicula(BusquedaPelicula peliBuscada){....}
```

# Cadena de mensajes

Si un objeto envía un mensaje de la forma:

```
a.b().c().d();
```

Esto va en contra del consejo **Tell, don't ask** y de la **Ley de Demeter** ("*don't talk to strangers*", "*only talk to friends*"), donde un objeto sólo debería enviar mensajes:

- a sí mismo
- a objetos que conoce (como variables de instancia)
- a objetos que recibe como parámetro
- a objetos que instancia

# Herencia Rechazada

Se puede dar cuando una clase sobrescribe completamente un método base definido por su padre sin importarle lo que éste le haya dicho que tenía que hacer, o cuando una clase hereda métodos que en realidad no debería tener.

¿Por qué utilizar herencia cuando se puede obtener lo mismo con composición?

# Herencia Rechazada

La herencia es rígida:

- Obliga a definir más métodos que los necesarios (en ciertos casos)
- En la mayoría de los lenguajes la herencia es simple, lo que se transforma en una limitante.

Possible solución: si no son lo mismo, ¿para qué forzarlo? **Utilizar composición por sobre herencia.**

# Lazy Class (Clase perezosa)

*La clase no hace nada o hace muy poco.*

- Suele darse en el caso de la extrema delegación o porque el componente fue diseñado “*por las dudas*”.
- Para evitar caer en estas clases:
  - Pensar en YAGNI (“*No lo necesitas*”) para no agregar funcionalidad hasta que sea necesario.
  - Eliminar de nuestro diseño las clases PLD (“*por las dudas*”)

# Nombres de variables cortos

```
Int a = .... ;  
...  
Double b = this.getPeriodo() + a;
```

¿Qué es A? ¿Qué es B?

El nombre de las variables debe representar su funcionalidad.

# Nombre de método muy largo

Si nos está costando escoger un nombre bien representativo para el método, y nos queda muy largo, seguramente hay más de una lógica comprendida dentro del mismo.

# Código muerto

El típico código “por las dudas” que nos marea a la hora de ver la responsabilidad que tiene asignada una clase.

>>Empresa

```
Public List<Cuenta> buscarCuentasPorPeriodo(){.....} //Por las dudas que se necesite  
Public List<Cuenta> buscarCuentasPorRangoFecha(){.....}//Por las dudas que se necesite
```

Solución: Borrar todo el code muerto y, otra vez, diseñar de modo extensible pero cerrado al cambio.

# Primitive Obsession

Suele darse al representar con ints, booleans, Strings o enumeraciones cosas que podrían ser objetos con comportamiento. Las enumeraciones nos llevan a tener sentencias condicionales en lugar de trabajar con objetos polimórficos.

Algo del estilo:

```
Public enum Estacion { VERANO, OTONIO, INVIERNO, PRIMAVERA }  
Public void actuar(){  
    switch(estacion) {  
        Case(VERANO) ..... break;  
        Case(OTONIO)... break;  
    }  
}
```

Solución: buscar abstracciones que nos permitan trabajar polimórficamente.

# Data Class – Only Accessors

- Una clase que sólo representa una estructura de datos es fácil verla porque tiene sólo getters y setters.
- Es mala práctica pensar que es bueno separar un objeto en atributos y comportamiento, ya que rompe con nuestro paradigma de objetos; donde un objeto agrupa comportamiento y atributos.

¡OJO! **Existen excepciones y son los Value Objects** (como un Mail, o un punto en un eje de coordenadas), y los objetos que modelan los parámetros que enviamos a un objeto (relacionado con *Long Parameter List*).

# Métodos fuera de lugar e Intermediario

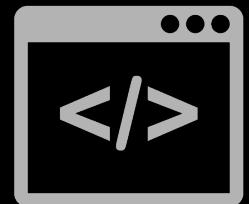
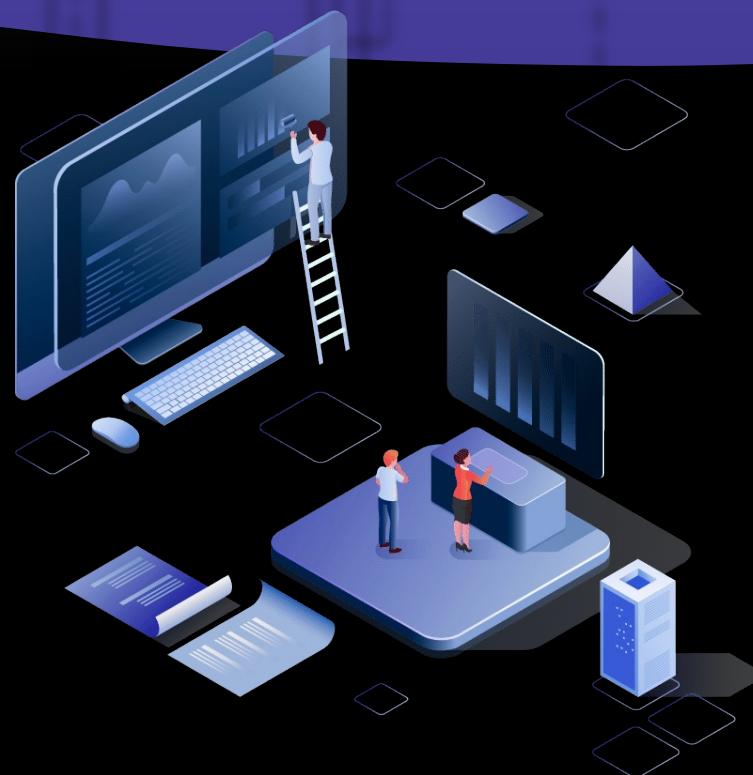
>> MiembroDeComunidad

```
public void unirseAComunidad(Comunidad comunidad){  
    comunidad.agregarMiembro(this);  
}
```

Si no accedemos a ningún atributo ni tampoco utilizamos comportamiento interno, posiblemente estamos ubicando mal el método: ¿por qué no enviar directamente el mensaje a "Comunidad"?

Este Smell nos da lugar a hablar de la asignación de posibles "Casos de Uso" en métodos de objetos de negocio (*en las próximas Slides lo trataremos*).

# Entities vs Value Objects



# Entity vs Value Object

Tanto las entidades como los value objects representan abstracciones para el dominio.

La diferencia radica en que **las entidades son**, generalmente, **abstracciones principales y/o relevantes para el dominio**, con sentido de existencia propio (o anidadas a otras entidades); mientras que **los value objects** generalmente representan **abstracciones "accesorias"** que solamente guardan datos. Estos datos no tienen sentido propio e indiscutible, sino que solamente tienen sentido si se los contextualiza con alguna entidad.

Por ejemplo, podríamos tener una clase Alumno con un atributo "teléfono", el cual podría estar siendo representado con una instancia de la clase "Teléfono". Esta clase podría tener como atributos el número, el código de área y el tipo (Móvil, Fijo, etc.)

# Entity vs Value Object – Tipos de igualdad

## Igualdad de referencia (identidad)

- Dos objetos son iguales si son exactamente el mismo, esto es, si están referenciando a la misma porción de memoria.

## Igualdad estructural (equivalencia)

- Dos objetos son iguales si todos los valores que tienen sus atributos, en un momento determinado, coinciden.

## Igualdad de identificadores

- Implica que una clase tiene un atributo de identificación, natural o ficticio.
- Dos objetos son iguales si tienen exactamente el mismo valor para su identificador.

# Entity vs Value Object – Vida útil

## Entidades

- Viven "**a largo plazo**"
- Pueden tener un historial
- Tienen sentido de existencia propio.

## Value Objects

- Son **desechables**, es decir que su vida útil es "muy corta".
- No tienen sentido de existencia propio, sino que están atados a entidades.

# Entity vs Value Object – Inmutabilidad

## Entidades

- Generalmente son **mutables**, es decir que sus atributos pueden cambiar sus valores a lo largo del tiempo.

## Value Objects

- Son **inmutables**, es decir que sus atributos no pueden cambiar de valores.
- Si se necesita cambiar el valor de algún atributo, se desecha la instancia y se crea un value object nuevo.

# Entity vs Value Object

## ***¿Entidad o Value Object?***

Muchas veces sucede que tomamos la decisión de modelar alguna abstracción como una clase por distintos motivos, como:

- Pueden existir muchos casos, que no sabemos cuántos son.
- La frecuencia al cambio puede ser “no baja”.
- Se requiere agregar, cambiar o eliminar algunos casos en momento de ejecución.
- Se requiere tener consistencia de datos.

Estas decisiones a veces pueden desencadenar en la creación de una clase que no necesariamente tiene comportamiento propio más que simples setters y getters de sus atributos.

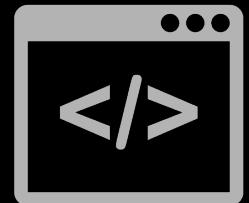
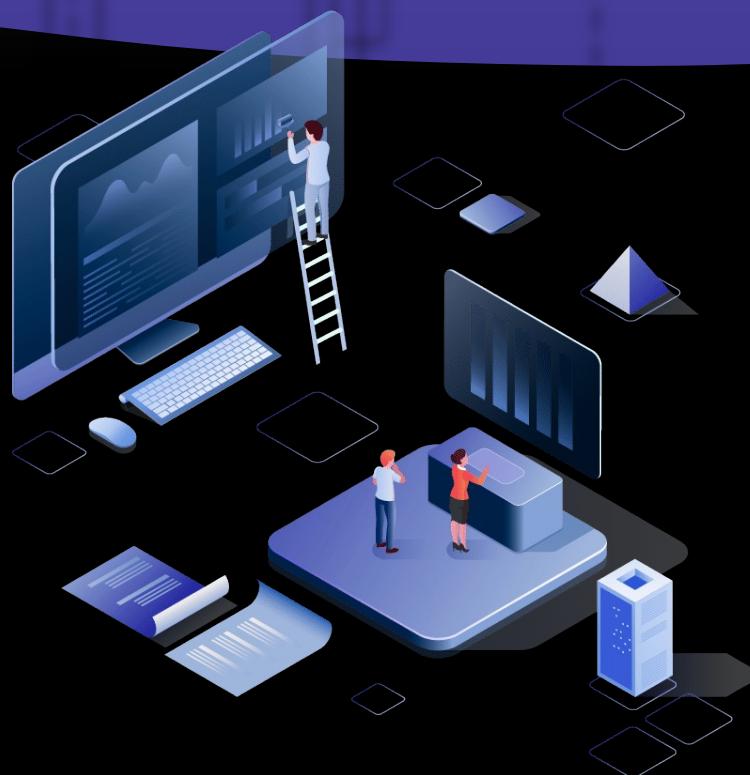
# Entity vs Value Object

## ***¿Entidad o Value Object?***

Lo importante es entender que:

- Una entidad puede que no tenga comportamiento y eso no la convierte en un value object pues tiene sentido de existencia propio y es algo “muy relevante” para nuestro dominio.
- Un value object es un objeto inmutable, desecharable, accesorio de algo principal, que no tiene sentido de existencia propio sino que debe ser contextualizado.
- Obviamente una entidad puede tener comportamiento, es decir, lógica propia del negocio; y eso no le quita su condición de “entidad”.

# Biblioteca vs Framework



# Bibliotecas

*"La biblioteca se desarrolla, se compila y se carga de forma separada al código de nuestra aplicación, pero las funcionalidades que define son utilizadas indistintamente por nuestro código como si de parte de él se tratase."*

*"Las bibliotecas resuelven un problema de reutilización de lógica asociada a abstracciones, representada e implementada a través de código."*

*"Las bibliotecas definen funcionalidades y estructuras concretas que pueden ser utilizadas o no por el desarrollador en un proyecto que las incluye, de la forma que el desarrollador considere oportuna. "*

# Bibliotecas

*"El control del flujo del programa es manejado por el desarrollador que eventualmente instancia estructuras de la biblioteca o hace llamadas a operaciones que ésta define. Es decir, cada llamada hace algún trabajo y retorna el control al usuario que la llamó."*

***Es importante destacar que a las bibliotecas las llamamos nosotros, cuando nosotros las necesitamos, no ellas a nosotros.***

# Frameworks

*"Los frameworks definen una estructura, una forma de trabajar siguiendo determinados lineamientos."*

*"(...) no solo se definen funciones y componentes que pueden ser reutilizados, sino que se definen piezas de software reutilizables que permiten ser extendidas para cubrir las necesidades particulares de cada proyecto pero a su vez definen una forma, un marco de trabajo común para todos ellos."*

# Frameworks

*"Los frameworks definen código que representa las abstracciones y la lógica surgidas de una estructura de trabajo. Los frameworks obligan al desarrollador a trabajar de una forma específica, siguiendo los lineamientos que definen.*

*Los frameworks condicionan fuertemente el diseño de un componente de software. Suelen manejar ellos el flujo de ejecución del programa e invertir el control, siendo ellos quienes deciden cuándo llamar al código del usuario. (El desarrollador jamás define el método main, sino que es el framework el que lo define y eventualmente llama al código del usuario)"*

# Frameworks - Tipos

Según la rigidez de un framework se lo puede clasificar como Dogmático o No dogmático.

## Dogmáticos

- "Los frameworks dogmáticos son aquellos que opinan acerca de la "manera correcta" de gestionar cualquier tarea en particular. Ofrecen soporte para el desarrollo rápido en un dominio en particular (resolver problemas de un tipo en particular) porque la manera correcta de hacer cualquier cosa está generalmente bien comprendida y bien documentada."

## No dogmáticos

- "*Los frameworks no dogmáticos, en contraposición, tienen muchas menos restricciones sobre el modo mejor de unir componentes para alcanzar un objetivo, o incluso qué componentes deberían usarse. Hacen más fácil para los desarrolladores usar las herramientas más adecuadas para completar una tarea en particular, si bien al coste de que se necesita encontrar esos componentes por uno mismo.*"

# Biblioteca vs Framework

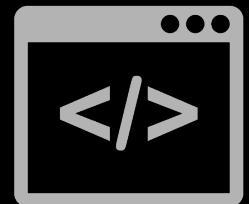
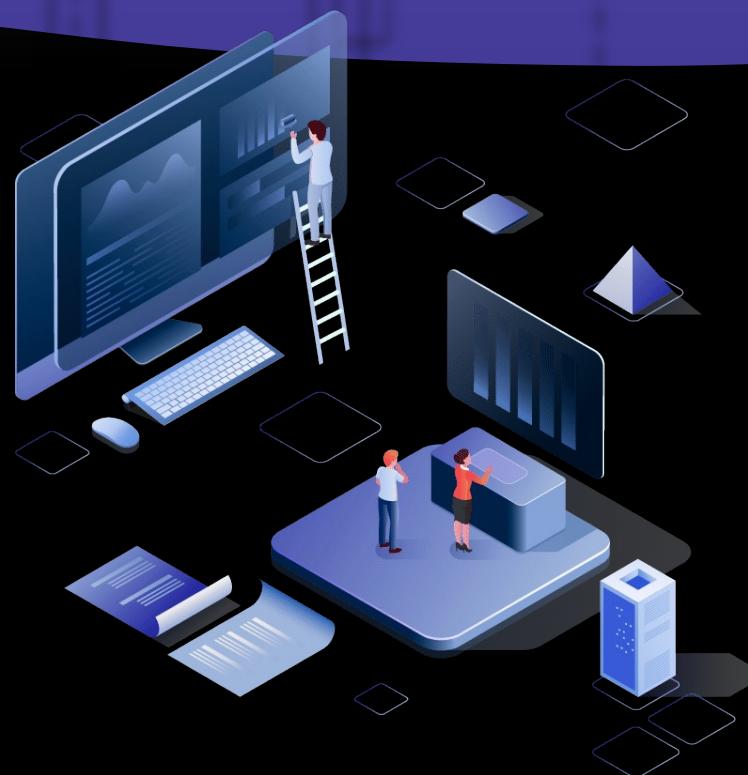
## Biblioteca

- Es responsabilidad del desarrollador decidir cómo y cuándo utilizar los componentes
- Las decisiones de diseño que se toman suelen tener bajo impacto en el diseño del código que la utiliza
- Utilizan **control directo**: El usuario llama funciones de la biblioteca e instancia las estructuras que la biblioteca pueda definir.

## Framework

- Este define la forma en la que se estructurará el código y el desarrollador se ve obligado a seguir esa forma
- Las decisiones de diseño que se toman pueden condicionar fuertemente el diseño del código cliente
- Utilizan **control inverso**: Es el framework el que llama funciones abstractas que el usuario define en concreto.

# Inyección de Dependencias



# Inyección de Dependencias

Partamos del siguiente ejemplo:

```
>> Incidente
```

```
public void enviarAvisos(){
    this.interesados.forEach(i -> whatsAppSender.enviar(i, "Ha ocurrido un incidente"));
}
```

Suponiendo que existe un objeto “whatsAppSender”, ¿cómo llegamos a conocerlo? Algunas opciones:

- Tener una única instancia de su clase accesible globalmente.
- Obtener ese objeto a través de otro que lo provea.
- Que a la clase Incidente le llegue por parámetro whatsAppSender

# Inyección de Dependencias - Singleton

>> Incidente

```
public void enviarAvisos(){
    this.interesados.forEach(i -> WhatsAppSender.getInstance().enviar(i, "Ha ocurrido un
incidente"));
}
```

>> WhatsAppSender

```
private static WhatsAppSender instance = null;

public static WhatsAppSender getInstance() {
    if(instance == null)
        instance = new WhatsAppSender(); // más toda la configuración
    return instance;
}
```

# Inyección de Dependencias – Service Locator

>> Incidente

```
public void enviarAvisos(){
    this.interesados.forEach(i -> ServiceLocator.get("whatsAppSender").enviar(i, "Ha ocurrido
un incidente"));
}
```

# Inyección de Dependencias - Inyectando la dependencia

>> Incidente

```
public Incidente(WhatsAppSender whatsAppSender){...}

public void enviarAvisos(){
    this.interesados.forEach(i -> this.whatsAppSender.enviar(i, "Ha ocurrido un incidente"));
}
```

# Inyección de Dependencias

Suponiendo que **A**, instancia de la clase 1, necesita a **B**, instancia de la clase 2, para poder realizar la tarea **X**...

## Singleton

- **A solicita a clase 2 una instancia, la cual devuelve siempre la instancia B.**
- “B” es un objeto global, única instancia para toda la ejecución.
- Fuerte acoplamiento entre Clase 1 y Clase 2.
- Difícil de testear pues es complicado mockear a B

## Service Locator

- **A solicita al Service Locator alguien que sea capaz de realizar la tarea X y éste le devuelve la instancia B o algún otro objeto que cumpla con la misma interface.**
- El Service Locator es un objeto global que permite generar distintas configuraciones.
- Permite el mockeo de objetos

## Inyección de Dependencias

- **A recibe como parámetro, en su constructor, a B o algún otro objeto que cumpla con la misma interface.**
- A no solicita a nadie la instancia B (o algún otro similar), sino que “le llega desde afuera”.
- Es más testeable pues permite el mockeo de objetos.
- Se puede combinar con las anteriores.

# Gracias

