

# Técnicas de programación avanzada

Curso 2023-2024

Tema 3.0 Calidad de código

## **Tema 1: Objetos y memoria.**

- 1.1 Características básicas del lenguaje. Primer programa. Compilación y Ejecución. IDE.
- 1.2 Sentencias de control. Secuencia, selección e iteración.
- 1.3 Abstracción. Clases, objetos, métodos y atributos.
- 1.4 Sobrecarga de métodos y encapsulamiento.

## **Tema 2. Otros conceptos fundamentales de la Programación Orientada a Objetos**

- 2.1 Herencia. Interfaces y clases abstractas. Agregación.
- 2.2 Polimorfismo.
- 2.3 Gestión de Excepciones.
- 2.4 Genericidad y plantillas.
- 2.5 Utilidades. Entrada y Salida.
- 2.6 Anotaciones.

## **Tema 3. Patrones de Diseño.**

- 3.1 Concepto de Patrones de Diseño.
- 3.2 Patrones de creación.
- 3.3 Patrones estructurales.
- 3.4 Patrones de comportamiento.

## **Tema 4. Programación de Interfaces.**

- 4.1 Interfaces Gráficas de Usuario.
- 4.2 Gestión de eventos.

## **Tema 5. Temas Avanzados.**

- 5.1 Concurrencia.
- 5.2 Inversión de Control. Definición y ejemplos. Inyección de dependencias.
- 5.3 Expresiones avanzadas del lenguaje.

## 3.0 Malas prácticas de programación - Síntomas

- **Fragilidad:** al realizar un simple cambio, se rompe el código de otras muchas partes, incluso aunque no está relacionado en modo alguno. Incrementa el coste de cambiar el código, porque se pueden generar muchos problemas inesperados.
- **Rigidez:** código difícil de cambiar, o extender. Requiere la creación de otros sistemas externos.
- **Inmovilidad:** falta de flexibilidad, es difícil separar el Sistema en componentes para reusar el código en otras partes. Lleva a la duplicidad de Código.
- **Viscosidad:** los problemas presentes en el código llevan a seguir empleando métodos malos, pues es más conveniente y fácil que aplicar buenas estructuras. Ciclo vicioso. Parches.



# 3.0 Code smells



Los "code smells" (literalmente, "olores del código") son un concepto en la ingeniería de software que se refiere a ciertas características en el código fuente que pueden indicar problemas más profundos.

No son errores en sí mismos; es decir, no son bugs que impiden que el programa funcione, sino señales de que el código puede no estar bien diseñado, lo que podría causar problemas en el futuro, especialmente en términos de mantenibilidad y escalabilidad.

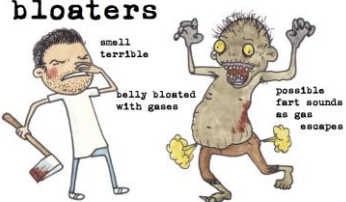


# 3.0 Code smells

Ejemplos comunes de code smells:

- **Duplicación de Código:** El mismo código aparece en más de un lugar. Esto hace que el mantenimiento sea más difícil porque cualquier cambio necesita ser replicado en múltiples lugares.
- **Clases o Métodos Largos:** Clases o métodos que intentan hacer demasiado. Esto a menudo hace que el código sea difícil de entender y mantener.
- **Código Muerto:** Código que ya no se utiliza o nunca se alcanza durante la ejecución.
- **Código Comentado:** Bloques extensos de código comentado. A menudo, esto indica una falta de confianza en la eliminación de código antiguo o una falta de claridad sobre si el código todavía es necesario.
- **God Object (Objeto Todopoderoso):** Una clase que tiene demasiadas responsabilidades y sabe demasiado o hace demasiado. Esto va en contra del principio de responsabilidad única.
- **Feature Envy (Envidia de Características):** Un método que parece estar más interesado en los métodos o propiedades de otra clase que en su propia clase.
- **Datos Primitivos Obsesivos:** Uso excesivo de primitivas en lugar de clases pequeñas para tipos de datos específicos.
- **Switch Statements Largos (o Cascadas de ifs):** Uso excesivo de declaraciones de control extensas, lo que puede hacer que el código sea difícil de seguir y mantener.
- **Acoplamiento Apretado:** Clases o componentes que están demasiado interconectados, lo que dificulta el cambio o reemplazo de una parte del sistema sin afectar a otras.
- **Demasiados Argumentos:** Métodos que requieren una gran cantidad de parámetros, lo que puede hacer que el método sea difícil de entender y usar.

# 3.0 Code smells

Family	Name	Description of the family
<b>bloaters</b> 	Long method, Large class, Primitive obsession, long parameter list, data clumps	Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually, these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).
<b>Object-Orientation Abusers</b>	Alternative Classes with Different Interfaces, Refused Bequest, Switch Statements, Temporary Field	All these smells are incomplete or incorrect application of object-oriented programming principles.
<b>Change preventers</b>	Divergent Change, Parallel Inheritance Hierarchies, Shotgun Surgery	These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too. Program development becomes much more complicated and expensive as a result.
<b>Dispensables</b>	Comments, Duplicate Code, Data Class, Dead Code, Lazy Class, Speculative Generality	A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.
<b>Couplers</b>	Feature Envy, Inappropriate Intimacy, Incomplete Library Class, Message Chains, Middle Man	All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

## 3.0 Buenas prácticas de programación

- **Abstracción:** La abstracción es un proceso de interpretación y diseño que implica reconocer y enfocarse en las características importantes de una situación u objeto, y filtrar o ignorar todas las particularidades no esenciales. Dejar a un lado los detalles de un objeto y definir las características específicas de éste, aquellas que lo distinguen de los demás tipos de objetos. Hay que centrarse en lo que es y lo que hace un objeto, antes de decidir cómo debería ser implementado.
- **Encapsulación:** Es la propiedad que permite asegurar que la información de un objeto está oculta del mundo exterior. Consiste en agrupar en una Clase las características (atributos) con un acceso privado y los comportamientos (métodos) con un acceso público → Acceder o modificar los miembros de una clase a través de sus métodos.
- **Modularidad:** La modularidad es la propiedad que permite dividir una aplicación en partes más pequeñas, cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.
- **Herencia/Generalización:** permite la reusabilidad de Código extendiendo clases y añadiendo nuevos métodos o atributos. Evita la repetición de código similar (redundancia).

## 3.0 Principios SOLID

Estos principios guían a los desarrolladores en la creación de sistemas de software que sean fáciles de mantener, comprender y extender.

**S - Principio de Responsabilidad Única (Single Responsibility Principle):** Este principio establece que una clase debería tener solo una razón para cambiar. Significa que una clase debe tener solo un trabajo o responsabilidad. Al adherirse a este principio, se minimiza el impacto de los cambios, ya que cada cambio afecta solo a la clase o módulo que tiene la responsabilidad correspondiente.

**O - Principio de Abierto/Cerrado (Open/Closed Principle):** Según este principio, las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para extensión, pero cerradas para modificaciones. Esto significa que deberías ser capaz de agregar nuevas funcionalidades sin cambiar el código existente. Esto se logra a menudo mediante el uso de interfaces o clases abstractas, permitiendo que el comportamiento de las clases se extienda sin necesidad de modificarlas.



## 3.0 Principios SOLID

**L - Principio de Sustitución de Liskov (Liskov Substitution Principle):** Este principio establece que los objetos de una superclase deberían ser capaces de ser reemplazados por objetos de sus subclases sin afectar la correctitud del programa. En términos prácticos, significa que las subclases no deben alterar el comportamiento esperado de la superclase y deben adherirse a sus contratos de interfaz.

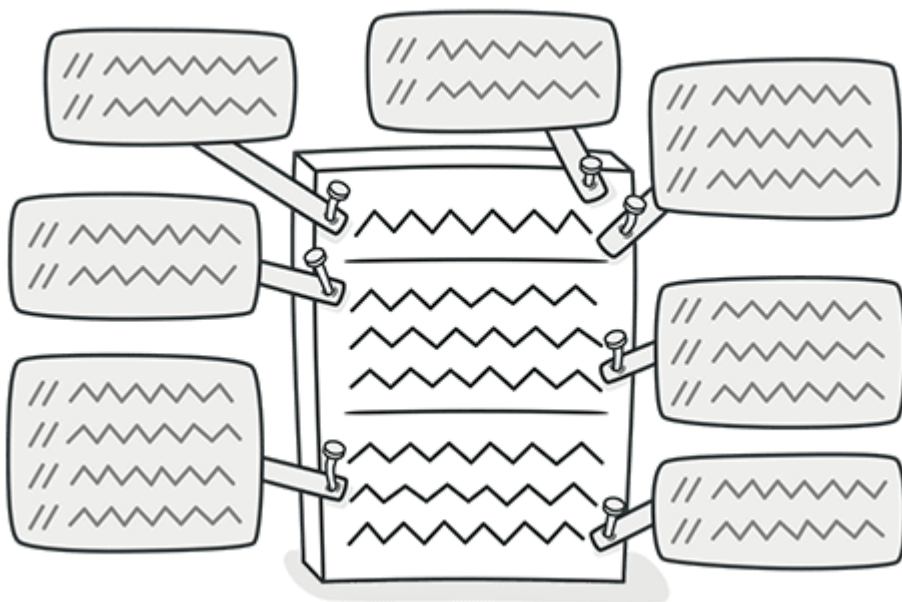
**I - Principio de Segregación de Interfaces (Interface Segregation Principle):** Este principio aboga por utilizar múltiples interfaces específicas en lugar de una única interfaz genérica. Las clases no deben verse obligadas a implementar interfaces que no utilizan. Al separar las interfaces, se evita que las clases dependan de métodos que no usan, lo que aumenta la cohesión del sistema.

**D - Principio de Inversión de Dependencias (Dependency Inversion Principle):** Este principio establece que las clases de alto nivel no deben depender de clases de bajo nivel, sino que ambas deben depender de abstracciones. Además, las abstracciones no deben depender de detalles, sino que los detalles deben depender de abstracciones. Esto conduce a un menor acoplamiento entre las clases y, por lo tanto, a un sistema más modular y fácil de actualizar o modificar.

# 3.0 Cosas a evitar y recomendaciones

to DO – Keep it simple and easy	NOT to do
<ul style="list-style-type: none"> <li>Lanzar programa desde una clase creada específicamente para ello</li> <li>Llamar a métodos desde dentro de los switch</li> <li>Separar acciones en métodos</li> <li>Usar polimorfismo en vez de condicionales en las clases</li> <li>Usar comentarios cuando sea necesario para explicar por qué se implementa así el código o similar</li> <li>Si usamos Código de otra persona, debemos dar atribuciones y referenciarlo.</li> <li>Seguir las convenciones para nombres de clases, paquetes, módulos, métodos, variables, constantes... (mayúsculas/minúsculas, camelCase, snake_case)</li> </ul>	<ul style="list-style-type: none"> <li>Tener todo mi programa en una misma clase</li> <li>Poner dentro de los switch muchas líneas de código</li> <li>Tener métodos que realizan diferentes funciones</li> <li>Heredar de una clase y repetir el mismo código en las clases hijas</li> <li>Abusar de comentarios y emplearlos como sustitutos de funciones/métodos</li> <li>Usar variables con nombres genéricos, letras,</li> <li>Mezclar estilos. Check: <a href="https://en.wikipedia.org/wiki/Naming_convention_(programming)">https://en.wikipedia.org/wiki/Naming_convention_(programming)</a></li> </ul>

## 3.0 Comentarios: no abusar



- No deben sustituir a una funcionalidad. A veces se pueden sustituir por un método
- Si el código es claro y el nombre de las variables y funciones está bien elegido, no son necesarios
- Si el Código cambia, hay que actualizar los comentarios relacionados. Un exceso de ellos aumenta el tiempo requerido para esta tarea

**@annotations & JavaDocs NO SON COMENTARIOS!**  
**USAR SIEMPRE**

# Técnicas de Programación Avanzada

```
exit(); //Gracias!
```