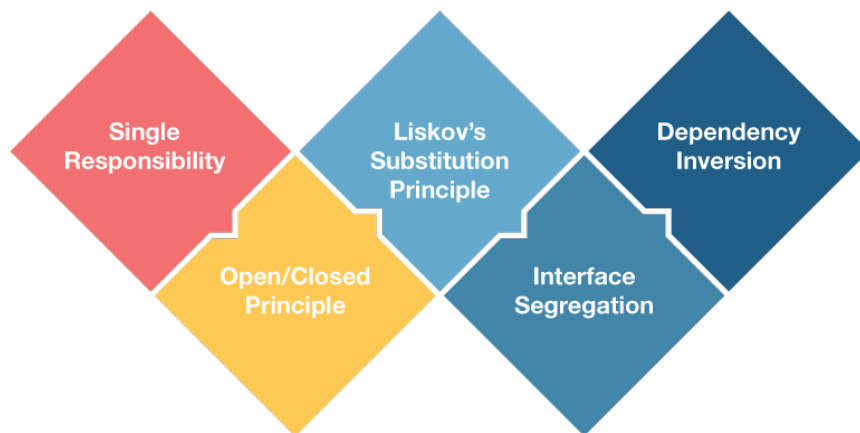


Principios SOLID con Fernando Herrera

S.O.L.I.D.



1. Introducción

Los principios SOLID aseguran la implementación de buenas prácticas para el manejo de nuestro código y reducir la deuda técnica.

Estos principios son agnósticos. Es decir, aplican a cualquier lenguaje de programación.

2. Clean Code y Deuda técnica

La deuda técnica se define como la **falta de calidad** de ciertos aspectos de nuestro código, o proceso de desarrollo del software: documentación, falta de pruebas...

La deuda técnica se paga con la **refactorización**.

2.1 ¿Qué es la deuda técnica?

La deuda técnica es La falte de calidad en el código, que genera una deuda que repercutirá en costosos futuros usualmente costos económicos

- Tiempo en realizar mantenimientos
- Tiempo en refactorizar código
- Teimpo en comprender código
- Teimpo adicional en la transferencia del código

Tipos de deuda técnica, según Martin Fowler:

Imprudente	Prudente y deliberada
El desarrollador actua de forma consciente e imprudente, lleva código de mala calidad que no es mantenible, usualmente suele decir: <i>"No hay tiempo", "Sólo copia y pega eso de nuevo", "Tenemos que salir como sea".</i>	Tenemos la deuda y estamos conscientes de ella, si no se paga a tiempo pagamos más intereses, los indicadores son palabras como: <i>"Tenemos que entregar rápido, luego refactorizamos", "Pon un mensaje de todo", "Lo vamos a tener en mente y en el futuro lo arreglamos"</i>
Imprudente e inadvertida	Prudente e inadvertida
es la más peligrosa ya que se genera por el desconociemiento o falta de experiencia, esto se genera por un desarrollador de perfil junior o peor aún, un falso senior.	En un inicio no tenemos todo el contexto del proyecto, esta dedua sale conforme avanzamos en el proyecto.

Caer en deuda técnica es normal y a menudo es inevitable.

La diferencia entre un buen desarrollador y un mal desarrollador es que el buen desarrollador se preocupa y atiende esa deuda técnica

2.2 ¿Cómo se paga la deuda técnica?

La deuda técnica se paga con **Refactorización**.

Es un proceso que tiene como objetivo mejorar el código sin alterar su comportamiento para que sea más entendible y tolerante a cambios.

Usualmente para que una **refactorización fuerte** tenga el objetivo esperado, es imprescindible contar con **pruebas automáticas**.

Si no tenemos pruebas automatizadas es común decir la frase: "Si funciona no lo toques."

La mala calidad en el software siempre la acaba pagando o asumiendo alguien.

Ya sea el cliente, el proveedor con recursos o el propio desarrollador dedicando tiempo a refactorizar o malgastando tiempo programando sobre un sistema frágil.

2.3 Clean Code

Código limpio es aquél escrito con la intención de que otra persona lo entienda.

Así mismo, podemos entender Clean Code como aquél simple y directo, que se puede leer como un libro.

La última definición dice: Programar es el arte de decirle a otro humano lo que quieres que la computadora haga.

Clean Code hace referencia a esto: hacer fácil a los humanos entender al código, no a la máquina.

2.4 Nombres de variables

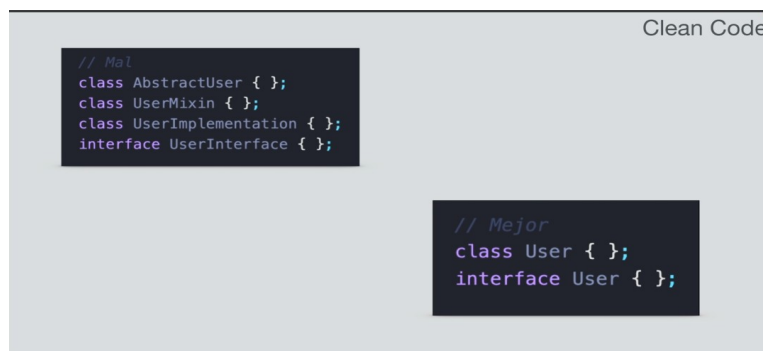
Los nombres de las variables han de ser **expresivos, pronunciables, y preferiblemente, en inglés.**

Habitualmente para **variables** se usa **camelCase**, para las **clases**, **UpperCamelCase**.

Se debe procurar no ahorrar caracteres a la hora de dar nombre a una variable, procurando **no** necesitar **comentarios** para añadir información extra para que quede claro qué propósito tiene dicha variable.



Añadido a esto, hay que procurar ahorrar información técnica a la hora de nombrar variables. Como ejemplo, a la hora de definir una clase abstracta, basta con darle un nombre común, como puede ser **Persona**, ya que el ser abstracto va implícito en el modificador de clase.



2.5 Funciones

Los nombres de las funciones han de **representar acciones**. Se ha de usar el **nombre del verbo que representa la acción, y un sustantivo**.

Los nombres han de ser descriptivos y concisos, tratando de **abstenerse de la implementación** de dicha función.

Ésto quiere decir que si tenemos una función llamada 'createUser', si no lo crea ha arrojar una excepción.

Definir correctamente el nombre de una función ayudará más adelante con el principio **'Single Responsibility'**, que es el primer principio SOLID.

Funciones

Clean Code

```
//mal
createUserIfNotExists();
updateUserIfNotEmpty();
sendEmailIfFieldsValid();
```

```
//mejor
createUser();
updateUser();
sendEmail();
```

“Sabemos que estamos desarrollando código limpio cuando cada función hace exactamente lo que su nombre indica”

Funciones

Clean Code

```
function sendEmail( toWhom: string ): boolean {
    // Verificar correo
    if ( !toWhom.includes('@') ) return false;

    // Construir el cuerpo o mensaje

    // Enviar correo

    // Si todo sale bien
    return true;
}
```

```
function sendEmail(): boolean {
    // Verificar si el usuario existe

    // Revisar contraseña

    // Crear usuario en Base de datos

    // Si todo sale bien
    return true;
}
```

A la hora de construir una función, es recomendable **limitar el número de argumentos** que recibe a un **máximo de tres**. Al superar este número, las funciones se hacen incómodas de leer y usar.

En caso de necesitar pasar una **gran cantidad de argumentos**, cabe pensar en crear una **clase o interface (typescript)** que aglomere los input, con el objetivo de simplificar la lectura y documentación de dicha función.



Adicionalmente, a la hora de crear nuestras funciones, **es recomendable**:

- Que sean **simples**.
- De tamaño reducido, siendo recomendable **no más de 20 líneas** de código.
- Funciones de una sola línea **sin causar complejidad**.
- **Evitar** uso de **'else'**.
- **Priorizar** uso de la **condición ternaria**.

2.6 Nombres según el tipo de dato

- **Arrays:** dependiendo del tipo de dato contenido, bastará con añadir una s al tipo de dato en cuestión.
Por ejemplo, para un array de **User**, se puede usar el nombre **'users'**. En caso de contener **datos específicos, atributos referentes a un objeto, nombre del tipo de dato más el atributo** en cuestión:
userNames, userEmails....
- **Booleans:** se han de usar prefijos descriptivos que den a entender inmediatamente el objetivo de la variable. Estos prefijos son **'is', 'has', 'can'**: **'isLoading', 'canRun' ...**
Es importante evitar negaciones en el nombre, ya que pueden llevar a una confusión innecesaria.

Booleanos - Booleans

Clean Code

```
//mal
const open = true;
const write = true;
const fruit = true;
const active = false;
const noValues = true;
const notEmpty = true;
```

```
//mejor
const isOpen = true;
const canWrite = true;
const hasFruit = true;
const isActive = false;
const hasValues = false;
const isEmpty = false;
```

- **Numbers:** Usar prefijos **'min', 'max', 'total', 'totalOf'** para dar el mayor valor semántico posible al nombre de la variable.

Números

Clean Code

```
//bad
const fruits = 3;
const cars = 10;
```

```
//better
const maxFruits = 5;
const minFruits = 1;
const totalFruits = 3;

const totalOfCars = 5;
```

2.7 Nombres de Clases

Se han de usar **sustantivos que identifiquen de forma rápida e inequívoca** para qué está destinada la clase.

Es por ello, importante **no** usar nombres **demasiado genéricos**, como puede ser ‘Person’, ‘Info’...

- Se ha de tener claro que el nombre de la clase es lo más importante de la misma.
De este modo, se **evita cargar la clase con funcionalidad, por ser demasiado genérica**.
- El nombre ha de estar formado por un **sustantivo o frases de sustantivos**.
- **No usar nombres genéricos**.
- Usar UpperCamelCase.

Para saber si hemos usado un **buen nombre** para una clase, podremos hacernos las **siguientes preguntas**:

- ¿Qué hace exactamente la clase?
- ¿Cómo realiza exactamente cierta tarea?
- ¿Hay algo específico sobre su ubicación?

Si algo no tiene sentido, se debe refactorizar

2.8 Principio DRY

“Si quieres ser un programador productivo, esfuérzate por escribir un código legible” - Robert C. Martín.

El principio DRY hace referencia a ‘**Don’t Repeat Yourself**’. Es decir, su misión es **evitar la duplicidad de código**.

Ventajas que obtenemos de implementar DRY en nuestro código son:

- **Simplifica las pruebas**: es más fácil probar una función, que distintas porciones de código que es básicamente lo mismo.
- Ayuda a centralizar los procesos, haciéndolos más fácil de mantener.
- Aplicar **DRY**, es sinónimo de **refactorizar**.

¿Cómo saber si estamos implementando DRY o no?

- **Si copiamos y pegamos código, es señal de que no se está implementando**.

3. Clean Code – Clases y Comentarios

Como se ha comentado anteriormente, las clases han de ser lo más específicas posible, para evitar caer en generalidades y hacer de una clase un recurso común para varias tareas distintas. Como se puede suponer, esto va contra el principio DRY de programación, siendo por tanto contraria a Clean Code.

Respecto los comentarios, si una clase / función requiere de comentarios para explicar su funcionalidad, quiere decir que el código no es lo suficientemente auto explicativo, siendo también indicativo de que no estamos cumpliendo con Clean Code: “Programar es decirle a otra persona lo que queremos que la máquina haga”

3.1 Herencia – Problemática

Derivado de la herencia de clases que heredan a su vez de otra, se puede generar un código realmente extenso, difícil de leer y mantener.

Supongamos que tenemos una clase ‘Person’ con 3 atributos: name, gender, birthdate. Luego tenemos una clase ‘User’, que hereda de ‘Person’, y añade dos atributos: ‘email’, ‘role’ y ‘lastAccess’.

A continuación, otra clase ‘UserSettings’, que hereda de ‘User’, y añade los atributos ‘workingDirectory’ y ‘lastOpenFolder’.

Como podemos comprobar, a la hora de crear un ‘userSettings’, necesitaremos pasar por constructor varios atributos, para con super, satisfacer las dependencias de las clases ‘padre’.

Esta casuística, como se puede inferir, contradice el principio de responsabilidad única, ya que por cada UserSetting, hay que crear un User y un Person.

En la medida de lo posible, es recomendable evitar heredar de otras clases, usando ‘composición’. Los extends añaden una capa extra de complejidad, que dificulta leer y comprender el código.

La composición consiste en crear una clase a partir de otras existentes para crear relaciones entre sí. Mediante composición reducimos notablemente las líneas de código y su complejidad, respetando el principio de responsabilidad única y los principios de clean code.

Para ver un ejemplo, acceder a este [repositorio](#), y buscar la transición de los archivos 06-classes-x.ts.

3.2 Estructura de clases

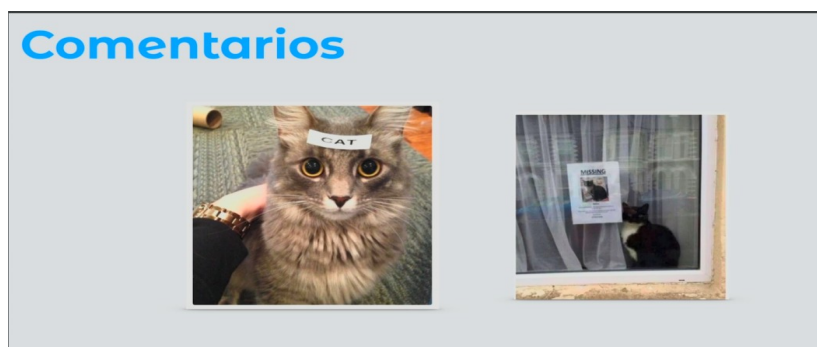
“El buen código parece estar escrito por alguien a quien le importa” – Michael Feathers.

Las consideraciones a tener en cuenta a la hora de estructurar nuestras clases, se describen en la siguiente imagen:

<pre>class HTMLElement { public static domReady: boolean = false; private _id: string; private type: string; private updatedAt: number; static createInput(id: string) { return new HTMLElement(id, 'input'); } constructor(id: string, type: string) { this._id = id; this.type = type; this.updatedAt = Date.now(); } setType(type: string) { this.type = type; this.updatedAt = Date.now(); } get id(): string { return this._id; } }</pre>	<p>Comenzar con lista de propiedades.</p> <ol style="list-style-type: none">1. Propiedades estáticas.2. Propiedades públicas de último. <p>Métodos</p> <ol style="list-style-type: none">1. Empezando por los constructores estáticos.2. Luego el constructor.3. Seguidamente métodos estáticos.4. Métodos privados después.5. Resto de métodos de instancia ordenados de mayor a menor importancia.6. Getters y Setters al final.
--	--

3.3 Comentarios en el código

El uso de los comentarios es definido en la siguiente imagen:



Como se puede entender, los comentarios son totalmente superfluos si el código está bien escrito. Es decir, buen código es aquél que se autodocumenta.

Existen excepciones, como puede ser a la hora de usar APIs de terceros, que nos pueden ahorrar implementar una solución compleja, siendo interesante **explicar el por qué, no el cómo o el qué**.

Los comentarios deben ser la excepción, no la regla.

3.4 Uniformidad en el proyecto

La uniformidad en el proyecto consiste en aplicar **soluciones similares a problemas similares**.

Una forma de añadir consistencia al código, es usar semánticas parecidas para que sea fácil de suponer o entender qué palabras clave hacen qué función. Por ejemplo:

Como se puede ver, tiene más sentido y se hace más intuitivo usar la sintaxis de la imagen de la izquierda, que la de la derecha.



Como se puede ver, se usa el mismo léxico para solucionar problemas similares, haciendo nuestro código intuitivo y ergonómico.

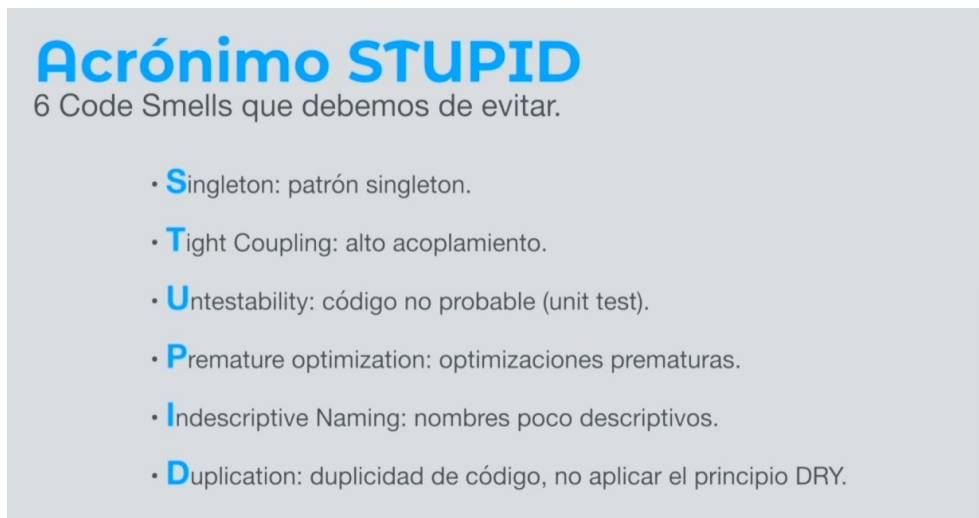
4. Acrónimo – STUPID

STUPID hace referencia a todo lo que huele mal en un código “**code smell**”. Es decir, a todo lo que nos puede hacer pensar que nuestro código **no está siguiendo los estándares Clean Code ni** sigue los principios **SOLID**.

4.1 CodeSmells – STUPID

CodeSmell **se relaciona** directamente con la **deuda técnica**. Son indicios de que nuestro código no es todo lo bueno que pudiera ser, siendo indicativos de que **nuestro código deberá ser refactorizado**.

La siguiente imagen define el significado letra del acrónimo:



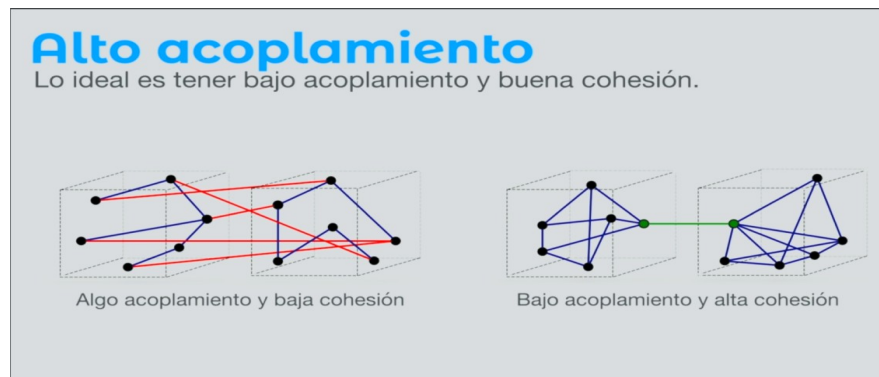
4.1.1 Patrón singleton

El patrón singleton se utiliza para crear una instancia de una clase para su posterior uso durante toda la aplicación. Esta práctica puede resultar beneficiosa en algunas cosas, pero en general es recomendada por los siguientes motivos:

- Vive en el contexto global.
- Puede ser modificado por cualquiera y en cualquier momento.
- No es rastreable.
- Difícil de testar debido a su ubicación.

4.1.2 Tight Coupling: Alta cohesión

Lo ideal es tener bajo acoplamiento y buena cohesión.



Como se puede ver en la imagen, si desarrollamos **código con alto acoplamiento**, creamos **infinidad de dependencias entre clases**, haciendo **muy difícil su mantenimiento y expansión**.

Derivadas de esta práctica, encontramos las siguientes consecuencias:

- Un cambio en un módulo, provoca **efecto dominó** de los cambios en otros módulos.
- El ensamblaje de módulos puede requerir más esfuerzo debido a la **mayor dependencia entre módulos**.
- Un módulo en particular puede ser más **difícil de reutilizar y probar** porque se deben incluir **módulos dependientes**.

“Queremos diseñar componentes que sean auto-contenidos, auto suficientes e independientes. Con un objetivo y un propósito bien definido.” - The Pragmatic Programmer.

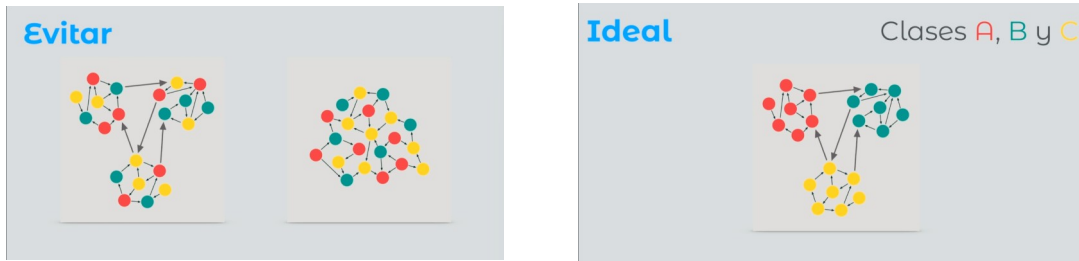
¿Qué es la cohesión?

- **Cohesión** hace referencia a **lo que hace la clase** o el módulo.
- La **baja cohesión** significaría que la clase realiza una gran variedad de acciones, no se enfoca en lo que debe de hacer (**No aplica Single Responsibility**).
- **Alta cohesión**: la clase se enfoca en lo que debería estar haciendo. **Sus métodos se relacionan con la intención de la clase**.

¿Qué es el acoplamiento?

Acoplamiento hace referencia a cuán relacionadas están dos clases entre sí.

- **Bajo acoplamiento:** cambiar algo importante en una clase no debería afectar a otra.
- **Alto acoplamiento:** se dificulta el cambio y mantenimiento de un código, dado que las clases están muy unidas. Hacer un cambio podrá requerir una renovación completa del sistema.



4.1.3 Code Smells adicionales

4.1.3.1 Código no probable

El código no probable es código difícilmente testeable.

Esto es consecuencia del alto acoplamiento, es decir, de la alta dependencia entre componentes o clases.

Debemos tener las pruebas en mente antes de la creación del código.

4.1.3.2 Optimizaciones prematura

Mantener abiertas las opciones retrasando la toma de decisiones nos permite darle mayor relevancia a lo que es más importante en una aplicación.

No debemos anticiparnos a los requisitos y desarrollar abstracciones innecesarias que puedan añadir complejidad accidental.

La complejidad accidental se define como una solución más compleja a la mínima indispensable.

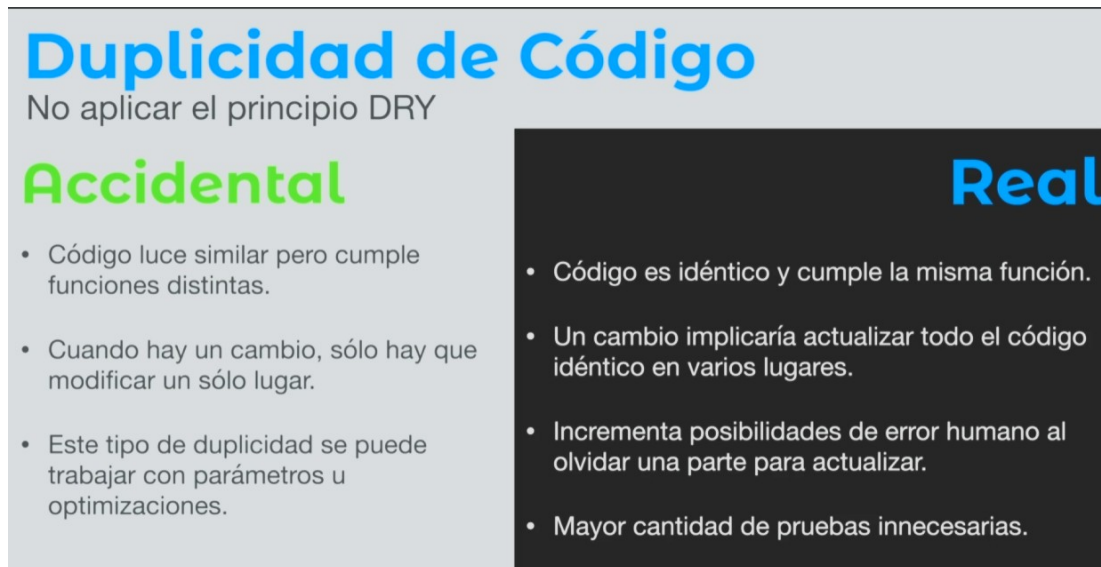
Como contraparte a la complejidad accidental, encontramos la complejidad esencial, que es la complejidad inherente al problema.

4.1.3.3 Nombres poco descriptivos

- Nombres de variables mal nombradas.
- Nombres de clases genéricas.
- Nombres de funciones mal nombradas.
- Ser muy específico o demasiado genérico.

4.1.3.4 Duplicidad de código

Existen dos tipos de duplicidad de código, la real y la no real, que son definidas en la siguiente imagen:



4.1.4 Code Smells honoríficos

4.1.4.1 Inflación de métodos o clases

Consiste en crear funciones que superan un número determinado de líneas, haciéndolos difícil de leer y mantener, y, seguramente, infringiendo el principio de responsabilidad única.

Para resolver este problema, **podemos hacer más pequeño, refactorizando en otros submétodos que hagan la tarea específica.**

Otro síntoma de mal olor son los métodos que reciben más de tres argumentos. La solución en estos casos es crear submétodos que ayuden a asegurar la legibilidad y responsabilidad única. En la siguiente imagen vemos un ejemplo gráfico de este problema:



4.1.5 Acopladores

Todos los olores contribuyen al acoplamiento excesivo entre clases o muestran lo que sucede si el acoplamiento se reemplaza por una delegación excesiva.

4.1.5.1 Feature envy

Un método accede más a los datos de otro objeto que a sus propios datos.

Suele ocurrir cuando una refactorización no es exitosa, ya que se han mezclado datos.

Cuando algún método hace referencia a una función de otro módulo, debemos pensar que la refactorización no es correcta, y debemos mover dicha función a otro lugar.

4.1.5.2 Intimidad inapropiada

Se da cuando una clase usa métodos internos de otra.

Las clases bien desarrolladas han de saber lo menos posible de otras clases.

Asegurando que este olor no existe, se facilita el mantenimiento del código, su lectura, etc.

4.1.5.3 Cadenas de mensajes

Se da cuando una función que llama a muchas funciones para ejecutar su funcionalidad.

Esto crea dependencias que dificultan el cambio de una funcionalidad a otra, por los efectos en cadena que se puedan derivar.

Para eliminar este olor, podemos tratar de eliminar pasos intermedios, y tratar de acceder a la información que necesitamos directamente.

4.1.5.4 The Middleman

Se da cuando una clase realiza una sola acción, y esta acción consiste en delegar a otra clase.

5. Principios SOLID