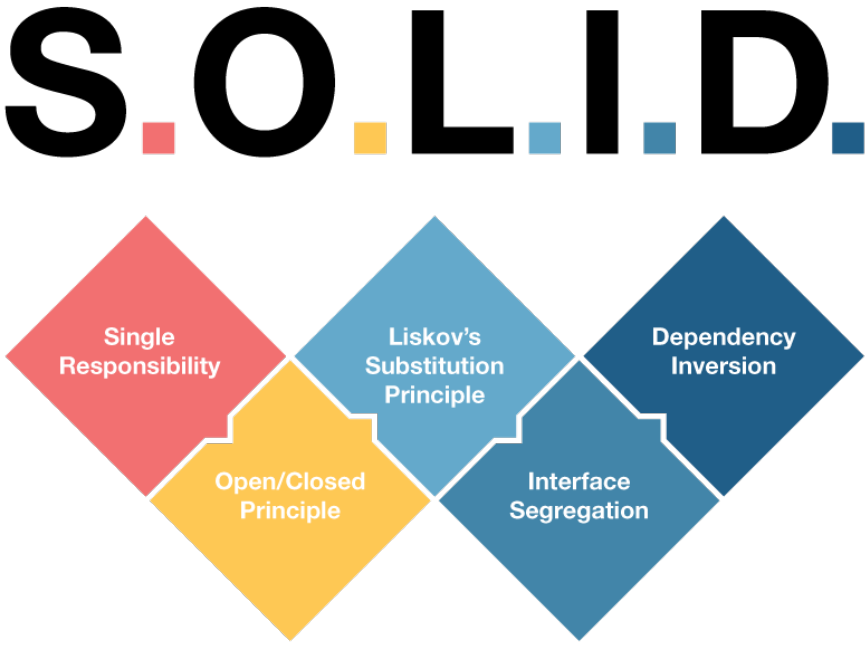
Principios SOLID con Fernando Herrera



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 elcódig, 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: *"No hay tiempo", "Sólo copia y pega eso de nuevo", "Tenemos que salir como sea"*.	Tenemos la deuda y estamos conscientes de ella, si no se paga a tiempo pagamos más intereses, los indicadores son palabras como: *"Tenemos que entregar rápido, luego refactorizamos", "Pon un mensaje de todo", "Lo vamos a tener en mente y en el futuro lo arreglamos"*
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**.

S**i no tenemos pruebas autmatizadas 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 ésto: hacer fácil a los humanos entender al código, no a la máquina.

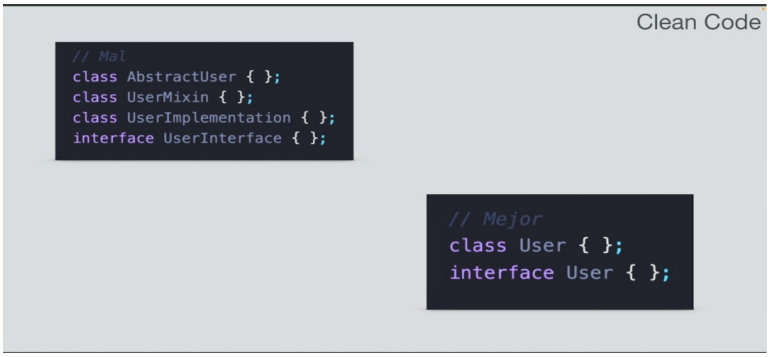
2.4 Nombres de variables

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 ésto, 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

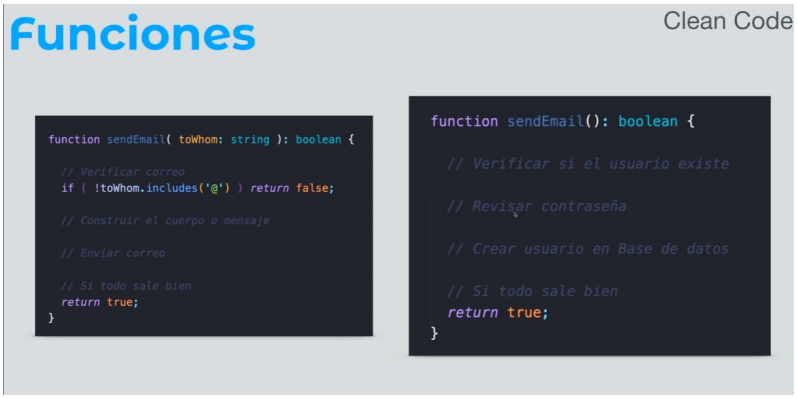
Los nombres de las funciones han de **representar acciones**. Se ha de usar el **nombre del verbo que representa la acción, y un sustantivo**.

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 arrojarse una excepción.

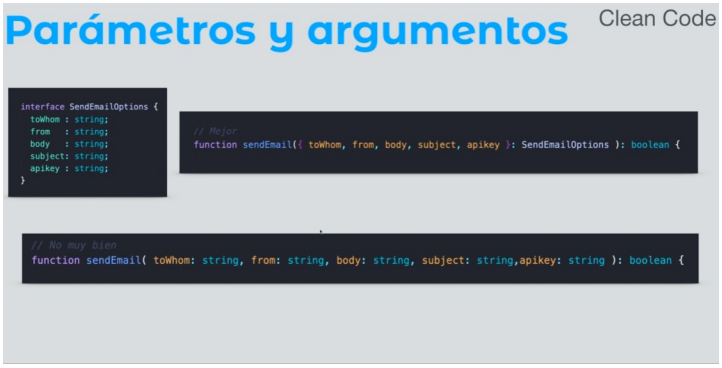
Definir correctamente el nombre de una función ayudará más adelante con el principio **‘Single Responsability’**, que es el primer principio SOLID.

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



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**:

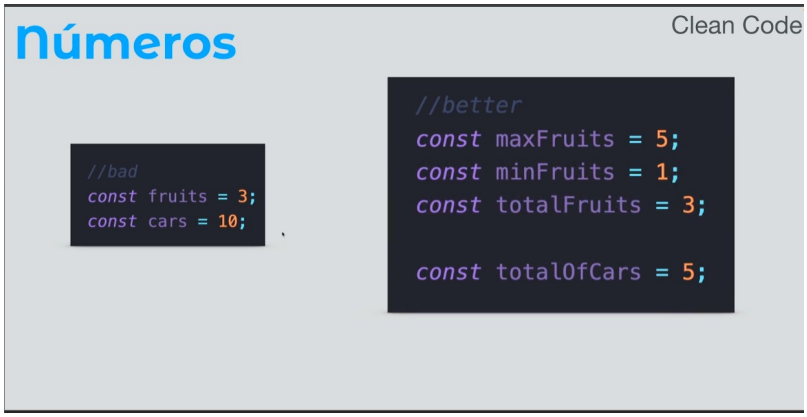
- Que sean **simples**.
- De tamaño reducido, siendo recomendable **no más de 20 líneas** de código.
- Funciones de una sola línea **sin causar complejidad**.
- **Evitar** uso de '**else**'.
- **Priorizar** uso de la **condición ternaria**.

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.

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



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 e**vita 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 s**iguientes 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 contrtaria 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.

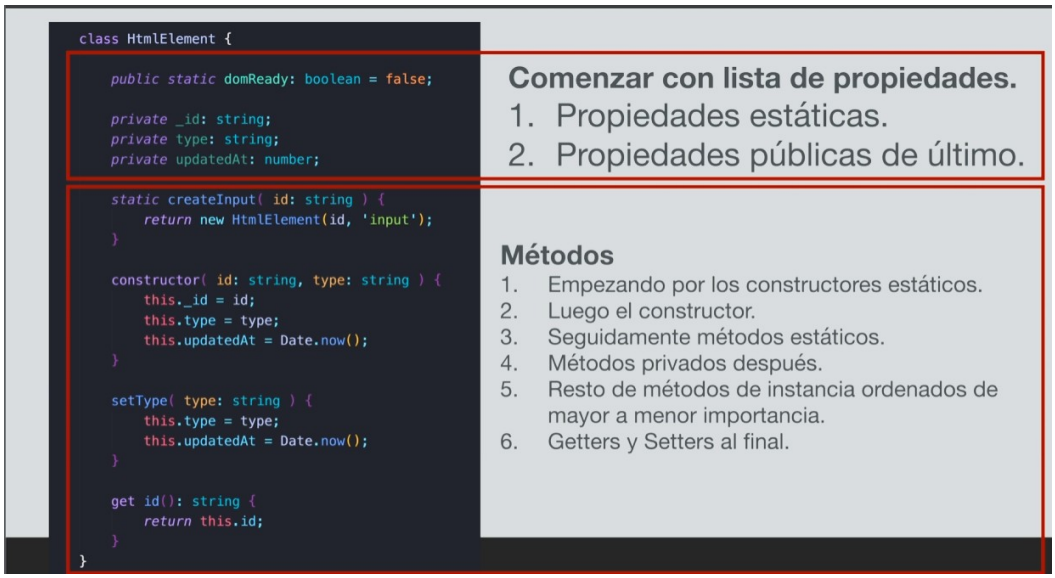
La composición consiste en crear una clase a partir de otras existentes para crear relacines entre sí.
Mediante composición reducimos notablemente las líneas de código y su complejidad, respetando el principio de responsabilidad única y los principios de clean code.

Para ver un ejemplo, acceder a este **repositorio**, y buscar la transición de los archivos 06-classes-x.ts.

3.2 Estrucutra de clases

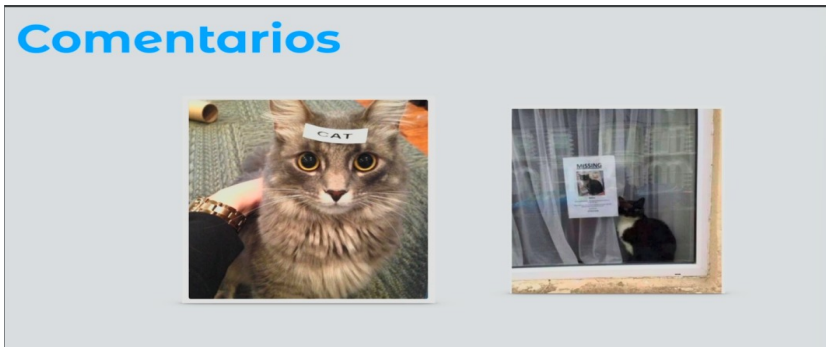
“El buen código parece estar escrito por alguien a quien le importa” – Michael Feathers.

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



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 supérfluos 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.**

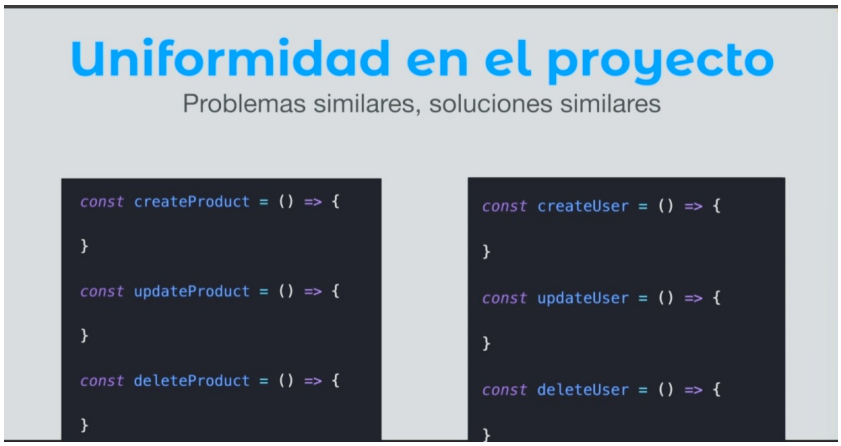
Los comentarios deben ser la excepción, no la regla.

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.

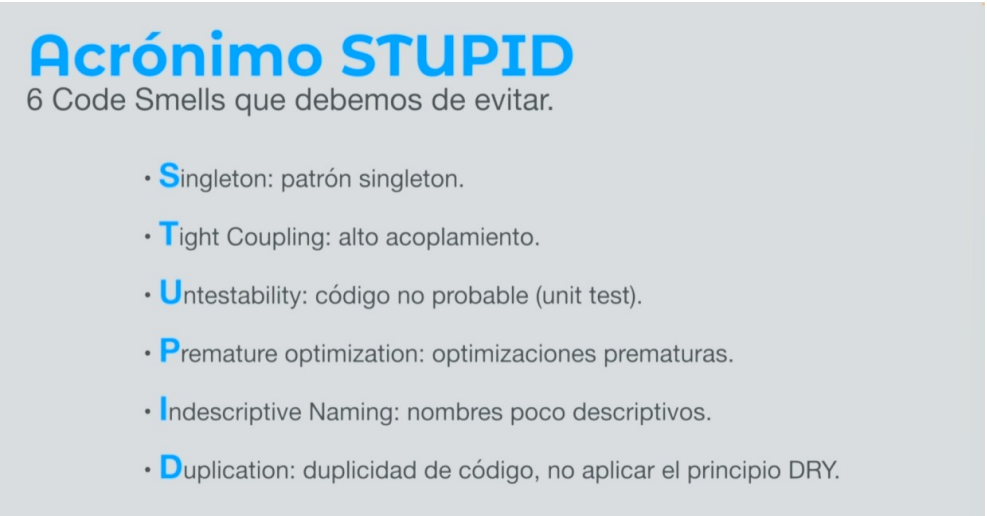
4. Acrónimo – STUPID

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:



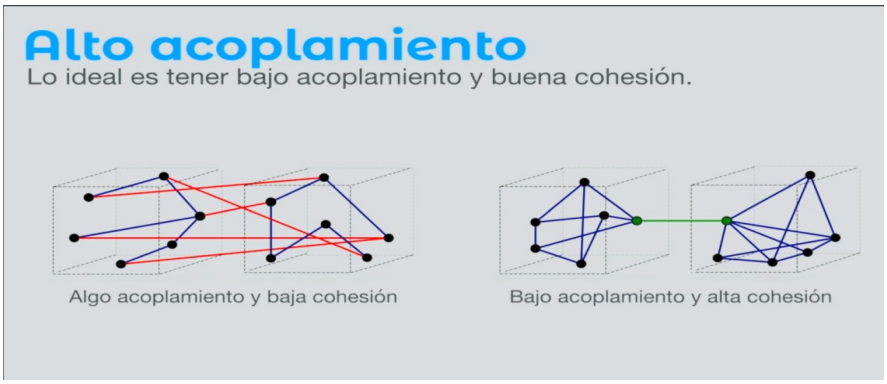
4.1.1 Patrón singleton

El patrón singleton se utilizar para crear una instancia de una clase para su posterior uso durante toda la aplicación. Esta práctica puede resultar beneficiosa en algunos 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 ubicacón.

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 pueede 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.

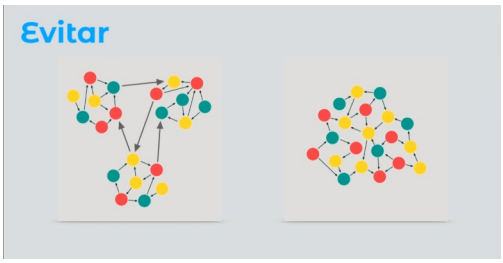
¿Qué es la cohesión?

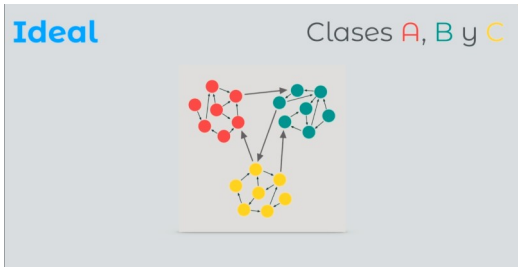
- **Cohesión** hace referencia a **lo que hace la clase** o el módulo.
- La **baja cohesión** significaría que la clase realiza una gran variedad de acciones, no se enfoca en lo que debe de hacer (**No aplica Single Responsability**).
- **Alta cohesión**: la clase se enfoca en lo que debería estar haciendo.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

¿Qué es el acopliamiento?

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.

N**o debemos anticiparnos a los requisitos y desarrollar abstracciones innecesarias que puedan añadir complejidad accidental.**

La c**omplejidad accidenta**l se define como una **solución más compleja a la mínima indispensable.**

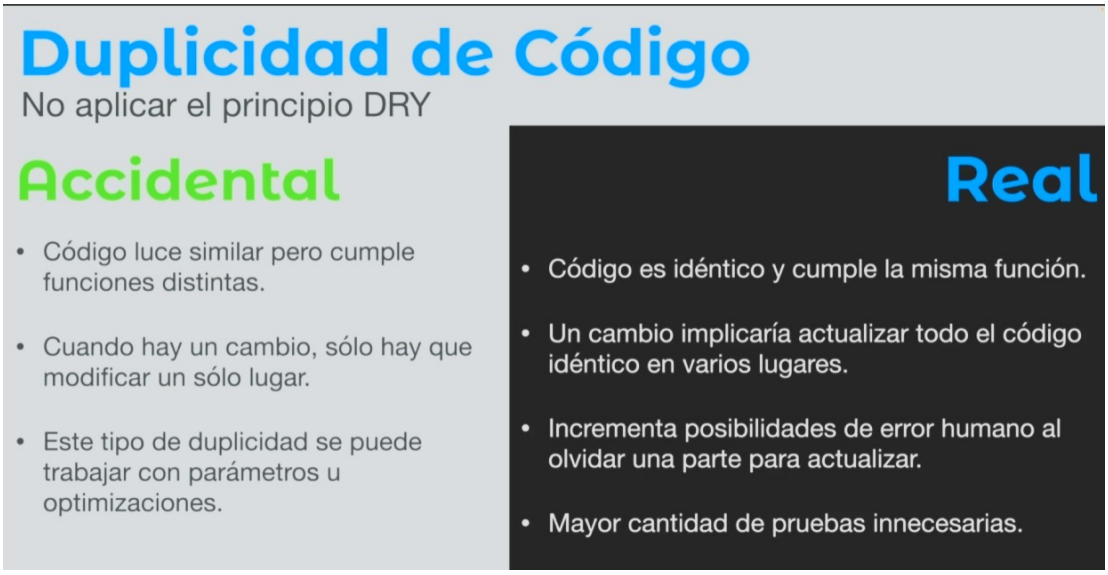
Como contraparte a la complejidad accidental, encontramos la **complejidad esencial**, que **es** la **complejidad inherente al problema**.

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 definidias en la siguiente imagen:



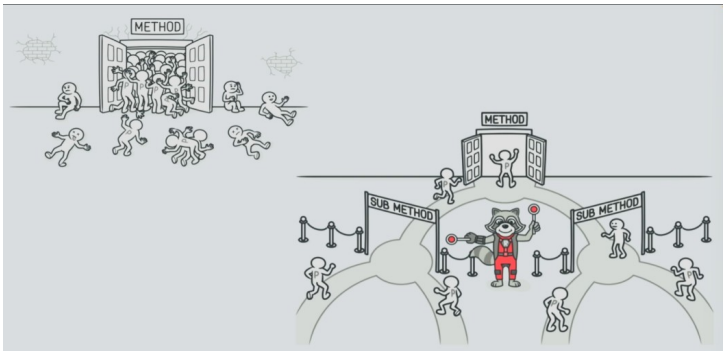
4.1.4 Code Smells honoríficos

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, inflingiendo el principio de responsabilidad única.

Para resolver este problema, p**odemos 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 l oque sucede si el acoplamiento se reemplaza por una delegación excesiva.

4.1.5.1 Feature envy

Un método accese 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

Antes de empezar con esta sección, es importante **diferenciar entre principios y reglas.**

Las reglas son normas a seguir, ya que de no hacerlo, algo dejará de funcionar.

Los principios son recomendaciones cuyo propósito es hacer que escribamos mejor código.
A pesar de ser recomendaciones, **el hecho de no seguir estos principios provoca que nuestro código aumente la deuda técnica**, llevando a un mayor costo tanto en tiempo como económico para su producción.

Los principios SOLID nos indican cómo organizar nuestras funciones y estructuras de datos en componentes y cómo dichos componentes deben estar interconectados.

El acrónimo SOLID hace referencia a:

- S – Single Responsibility Principle (SRP)
- O – Open/Closed Principle (OCP)
- L – Liskov Substitution Principle (LSP)
- I – Interface Segregation Principle (ISP)
- D – Dependency Inversion Principle (DIP)

5.1 Responsabilidad Única (Single Responsability)

“Nunca debería haber más de un motivo por el cual cambiar una clase o módulo” – Robert C. Martin.

Tener más de una responsabilidad, hace que nuestro código sea menos flexible y tolerante al cambio.

Esto no quiere decir que nuestras clases deban tener un sólo método, sino que n**uestras clases solo deberían cambiar por un motivo.** Es decir, que todas las funciones estén relacionadas con un proceso común, y coherente.

Por ejemplo, no tendría sentido que una clase de persistencia, se encargue a su vez de gestionar items en un carro.

La clase de persistencia debería encargarse únicamente gestionar la persistencia de un objeto, y nada más.

De esta forma, solo se deberá cambiar dicha clase si los datos de la persistencia cambian.

5.1.1 Detectar incumplimiento de SRP

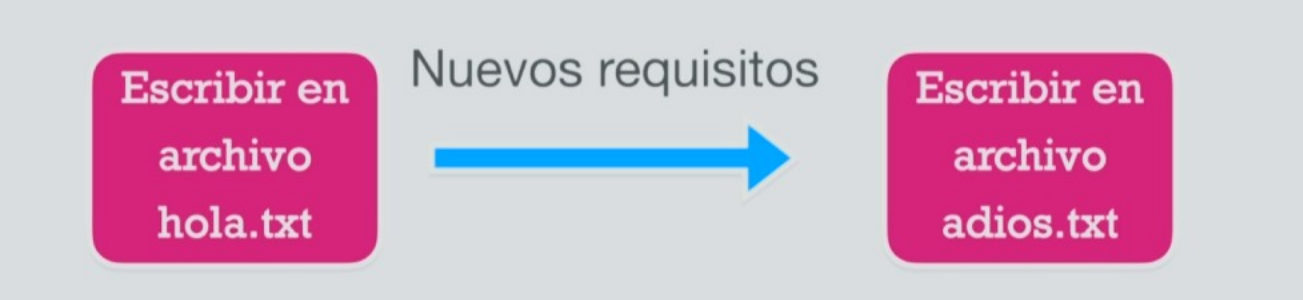
- Nombres de clases y módulos demasiado genéricos.
- Cambios en el código suelen afectar la clase o módulo.
- La clase involucra múltiples capas.
- Número elevado de importaciones.
- Cantidad elevada de métodos públicos.

5.2 Principio de abierto y cerrado (Open and Close)

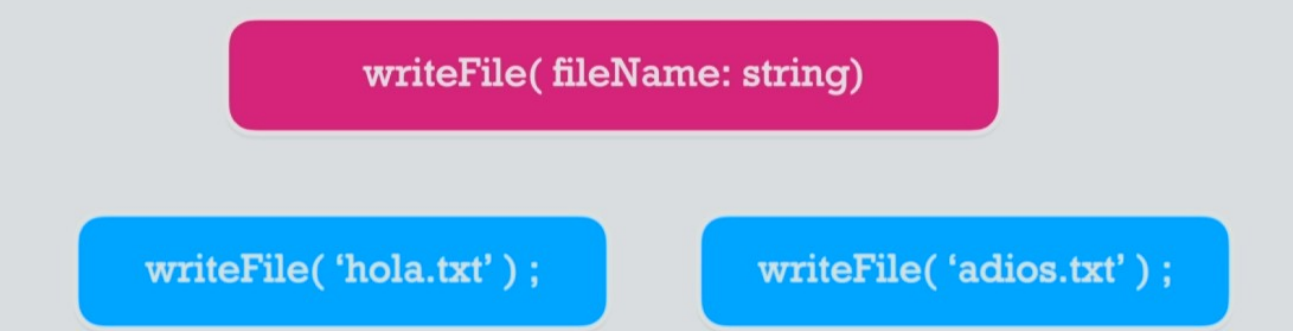
Aunque este principio depende del contexto, establece que las entidades de software deben estar **abiertas para la extensión, pero cerradas para la modificación.**

La siguiente imagen describe esta definición de forma más visual:

Mal:



Bien:



En definitiva, si debemos modificar un comportamiento de una clase, y abrir la clase para implementar ese cambio, no estamos cumpliendo con el principio de Open-Close.

Para promover el cumplimiento de este principio, podemos implementar en la medida de lo posible el desacople de dependencias entre clases.

Para ver un ejemplo sobre éste principio, acceder a este repositorio y ver los archivos '02-open-close-x.ts'. En este ejemplo se puede comprobar como se ha eliminado la dependencia, y el cambio en la forma de acceder a la información solo se implementará en el archivo '02-open-close-c.ts'.

5.2.1 Detectar incumplimiento de Open-Close

- Cambios en otras clases normalmente afectan a nuestra clase o módulo.
- Cuando una clase o módulo afecta a muchas capas (Presentación, almacenamiento, etc.).

Cuando exista la duda entre optimizar el código o hacerlo más fácil de leer, siempre hay que optar por escribir un código que sea más legible para otra persona.

5.3 Principio de sustitución de Liskov

“Las funciones que utilicen punteros o referencias a clases base deben ser capaces de usar objetos de clases derivadas sin saberlo”. - Robert C. Martin.

Recibe este nombre debido a que fue la doctora Bárbara Liskov, que recibió el premio Turing por contribuciones a los fundamentos prácticos y teóricos del lenguaje de progamación y el diseño de sistemas, especialmente relacionados con la abstracción de datos, la tolerancia a fallas y computación distribuida.

En resumidas cuentas, el principio viene a decir:

“Siendo U un subtipo de T, cualquier instancia de T debería poder ser sustituida por cualquier instancia de U sin alterar las propiedades del sistema.”

Para ver un ejemplo sobre éste principio, acceder a este repositorio y ver los archivos ‘03-liskov-x.ts’

5.4 Segregación de Interfaz (Interface segregation)

“Los clientes no deberían estar obligados a depender de interfaces que no utilicen” – Robert C. Martin.

De existir un cambio en una interfaz implementada en una clase, **podría ocurrir que nuestro código dejase de funcionar, o se produjese un efecto dominó para que el cambio no afectase a las clases que las implementan.**

Para ver un ejemplo sobre éste principio, acceder a este repositorio y ver los archivos ‘04-segregation.ts’

5.4.1 Detectar incumplimiento de Segregación de interfaz.

- Si las interfaces que diseñamos obligan a incumplir los principios de responsabilidad única y sustitución de Liskov.

5.5 Principio de 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 concreciones. Los detalles deben depender de abstracciones.” - Robert C. Martin

Esto quiere decir que:

- Los módulos de alto nivel no deberían depender de módulos de bajo nivel.
- Ambos deberían depender de abstracciones.
- Las abstracciones no deberían depender de detalles.
- Los detalles deberían depender de abstracciones.

Los componentes más importantes son aquellos que se centran en resolver el problema subyacente al negocio, es decir, la capa de dominio.

Los m**enos importantes son los que están próximos a la infraestructura, es decir, aquellos relacionadoss con la UI, la persistencia, la comunicación con APIs externas, etc**.

Es importante tener desacopladas las distintas capas de lo relativo al dominio de la aplicación. Cambiar de un sistema de base de datos a otro, por ejemplo, no debería implicar que las demás capas deban cambiar.

Cuando se habla **Depender de abstracciones,** se habla de **interfaces o clases abstractas.**

Uno de los motivos más importantes por el cual las reglas de negocio o **capa de dominio deben depender de estas** y no de concreciones, es que **aumenta su tolerancia al cambio.**

C**ada cambio en un componente abstracto implica un cambio en su implementación.**

Por el contrario, **los cambios en implmentaciones concreta**s, la mayoría de veces, n**o requieren cambios en las interfaces que implementa.**

En programación, **dependencia** hace referencia a que un **módulo necesita de otro para poder realizar su trabajo.**

En algún punto, nuestro programa requerirá muchos módulos, siendo este el punto en el que se deba usar inyección de dependencias. Sirva la siguiente imagen como ejemplo: