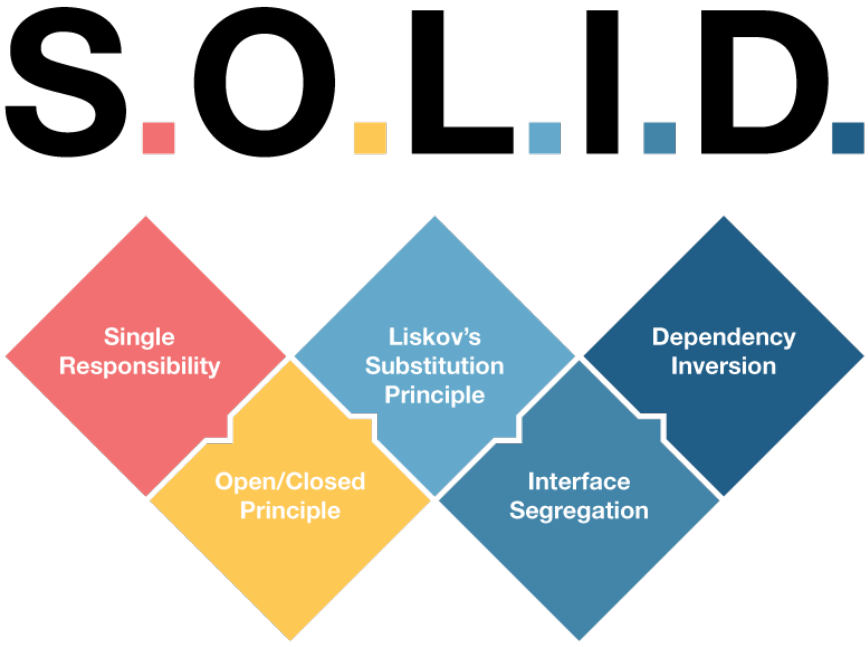
Principios SOLID con Fernando Herrera



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.

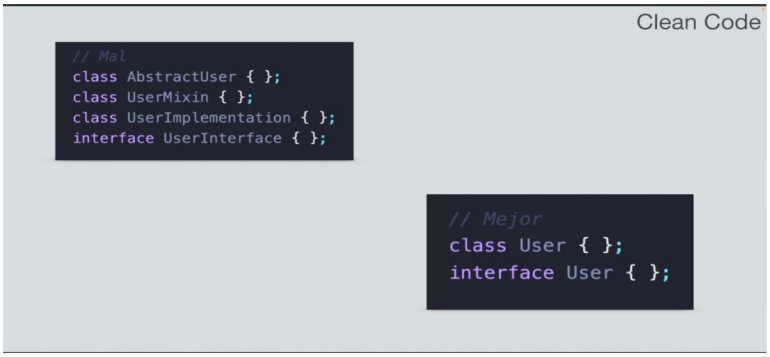
2.4 Nombres de variables

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

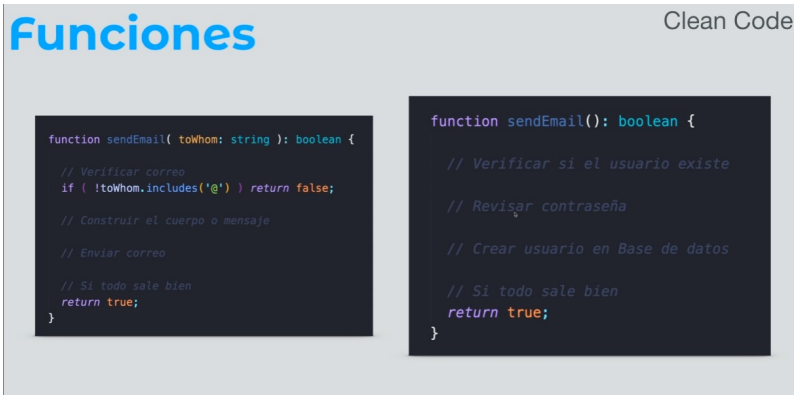
Los nombres de las funciones han de **representar acciones**. Se ha de usar el **nombre del verbo que representa la acción, y un sustantivo**.

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.

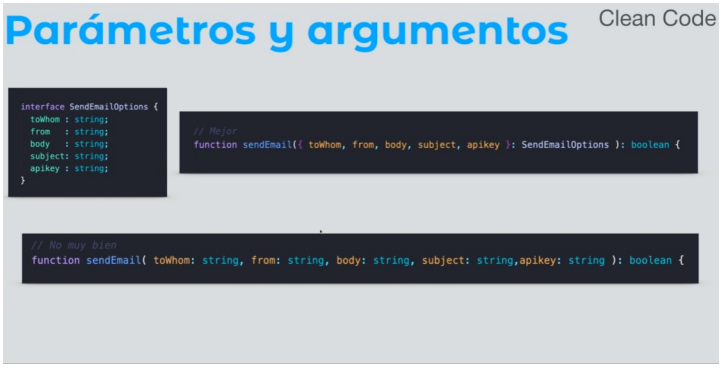
Definir correctamente el nombre de una función ayudará más adelante con el principio **‘Single Responsability’**, que es el primer principio SOLID.

“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**:

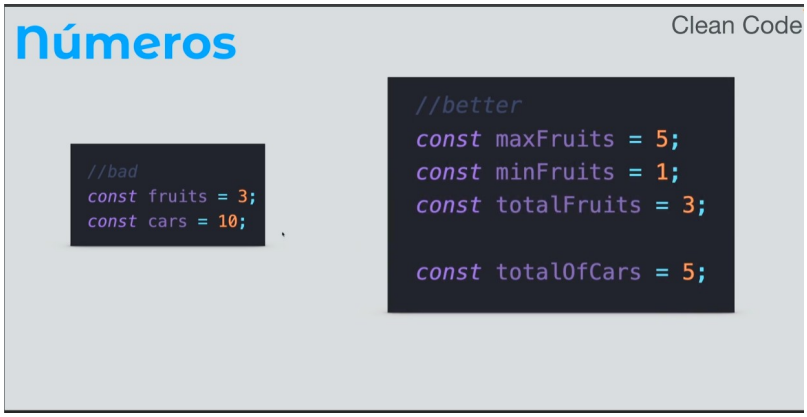
- Que sean **simples**.
- De tamaño reducido, siendo recomendable **no más de 20 líneas** de código.
- Funciones de una sola línea **sin causar complejidad**.
- **Evitar** uso de '**else**'.
- **Priorizar** uso de la **condición ternaria**.

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