# Sección 1 y 2: Clean Code y Deuda técnica

* Deuda técnica

Es la falta de calidad, de cierto aspecto de nuestro código. Es la falta de calidad en nuestro código, o falta de calidad en algún proceso de nuestro desarrollo de software.

No siempre es código, pueden ser temas de documentación, pruebas, etc., sin embargo, se paga con una refactorización.

Da como consecuencia, en costos económicos, y en otros apuntados.

Imagen que contiene interior, cuchillo

Descripción generada automáticamente

Tipos de Deudas:

* 1. Imprudente
  2. Prudente y deliberada (si no se paga a tiempo, más intereses se pagan después)
  3. Inadvertido (desconocimiento y falta de experiencia)
  4. Prudente e inadvertida (ocurre terminando el proyecto, visualizando posibles mejoras, y preguntándose si vale la pena construir algo de nuevo)

Tabla

Descripción generada automáticamente

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

¿Cómo se paga una deuda técnica?

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

## Clean Code

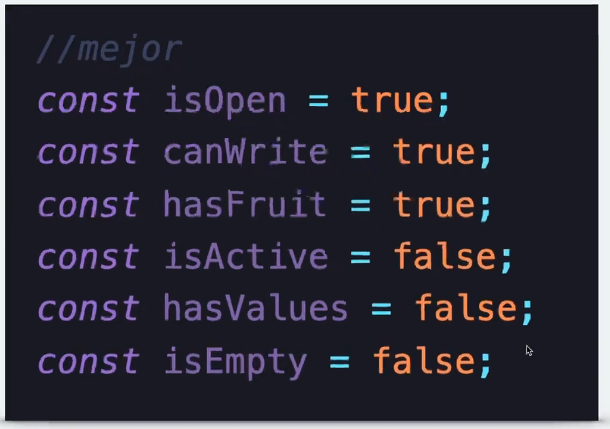
Cliente o código limpio es aquel que se ha escrito con la intención de que otra persona o tú mismo lo entienda en el futuro.

1. Definir nombres claramente en variables o constantes, sin ausencia de palabras o letras.
2. Solo ausentar en los nombres de las variables, los nombres técnicos de los lenguajes.

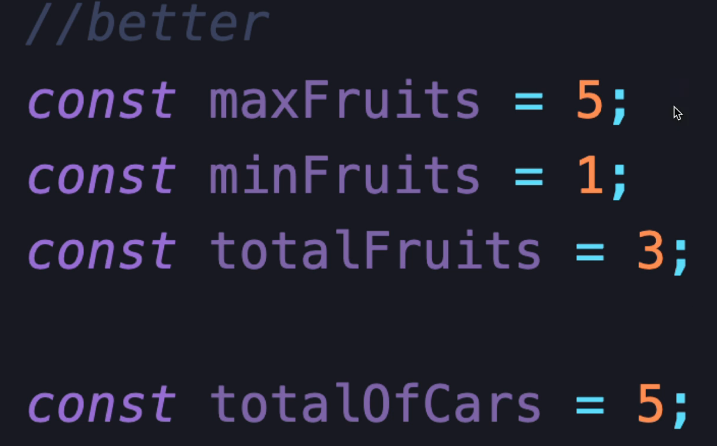
Texto

Descripción generada automáticamente

Opciones de nombres para variables booleanas



Opciones de nombres para variables numéricas



Opciones de nombres para funciones

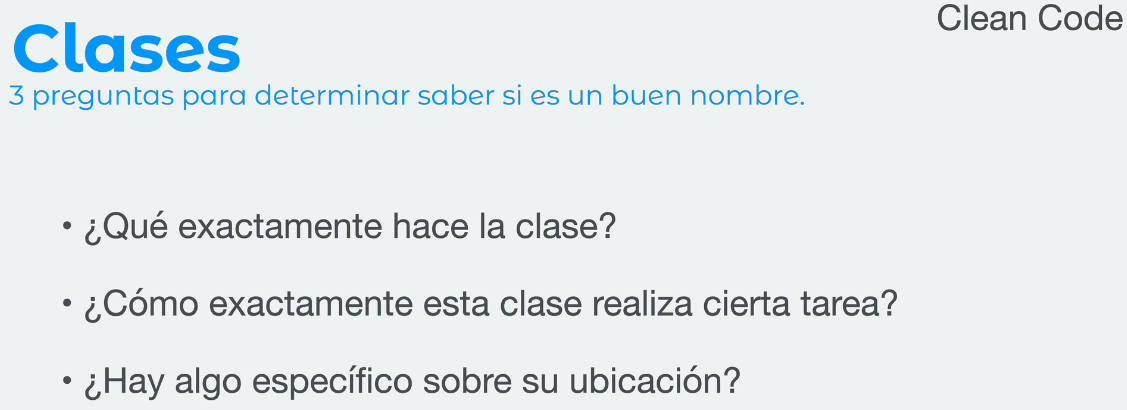
Deben ser acciones, pero describiendo lo necesario, corto y preciso.



Preguntas para poner un nombre correcto a una clase, debe ser un sustantivo, y no ser uno genérico.

“Si algo no tiene sentido, remuévalo o refactoriza”

Mas palabras no da a ser un mejor nombre



Diferencias entre argumentos y parámetros.



Se recomienda que sean solo funciones con 3 parámetros máximo, no más, porque están muy aglomeradas.

Recomendaciones para mejorar funciones con muchos parámetros.

Se recomienda igual ordenarlos de manera alfabética.

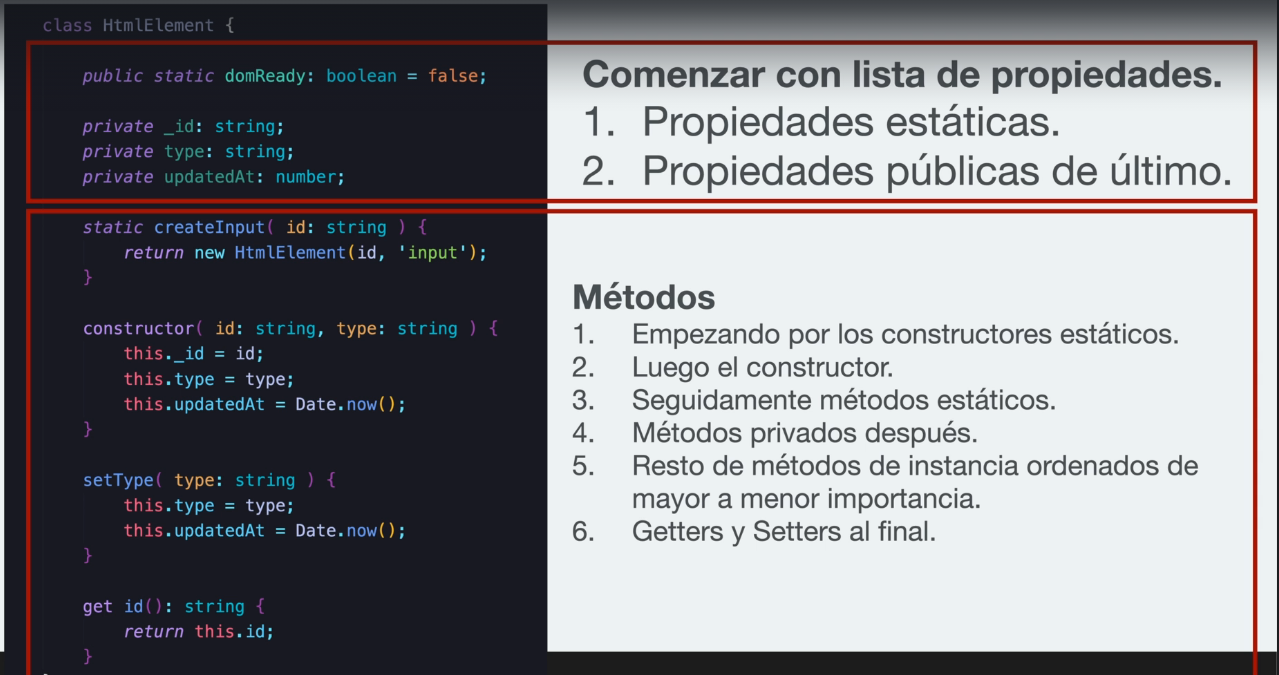


## Principio del DRY (Don’t Repeat Yourself)

* Simplemente es evitar tener duplicidad de código.
* Simplifica las pruebas.
* Ayuda a centralizar procesos.
* Aplicar el principio DRY, usualmente lleva a refactorizar.

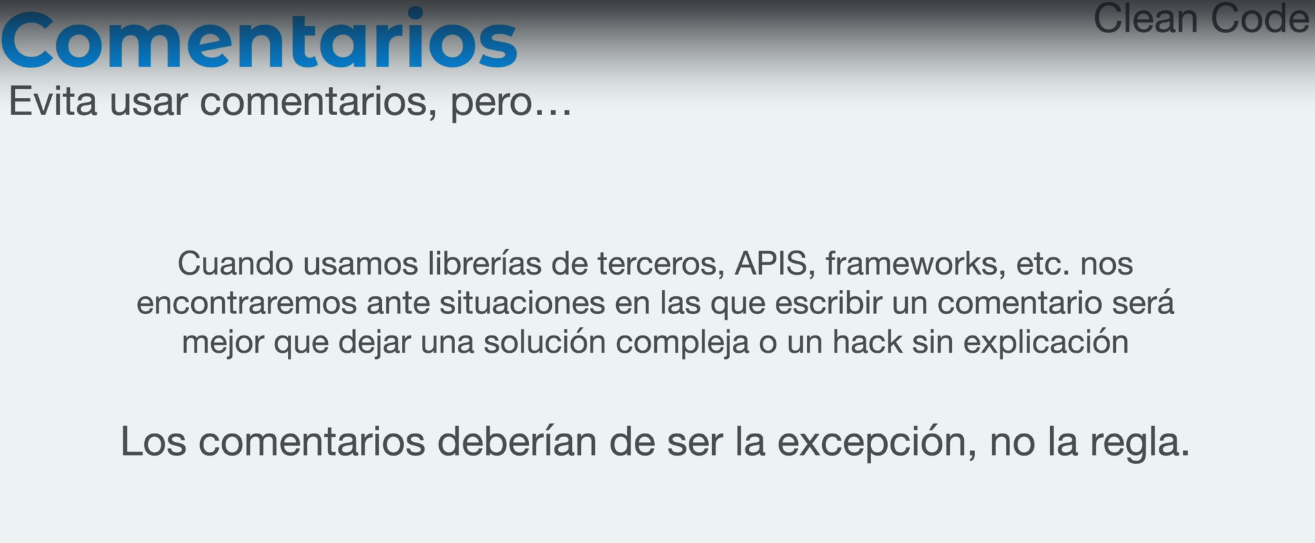
# Sección 3: Clean Code – Clases y Comentarios

Mantener una estructura de creación de clases. Se menciona una recomendación en el orden de creación que seria el siguiente en la imagen.



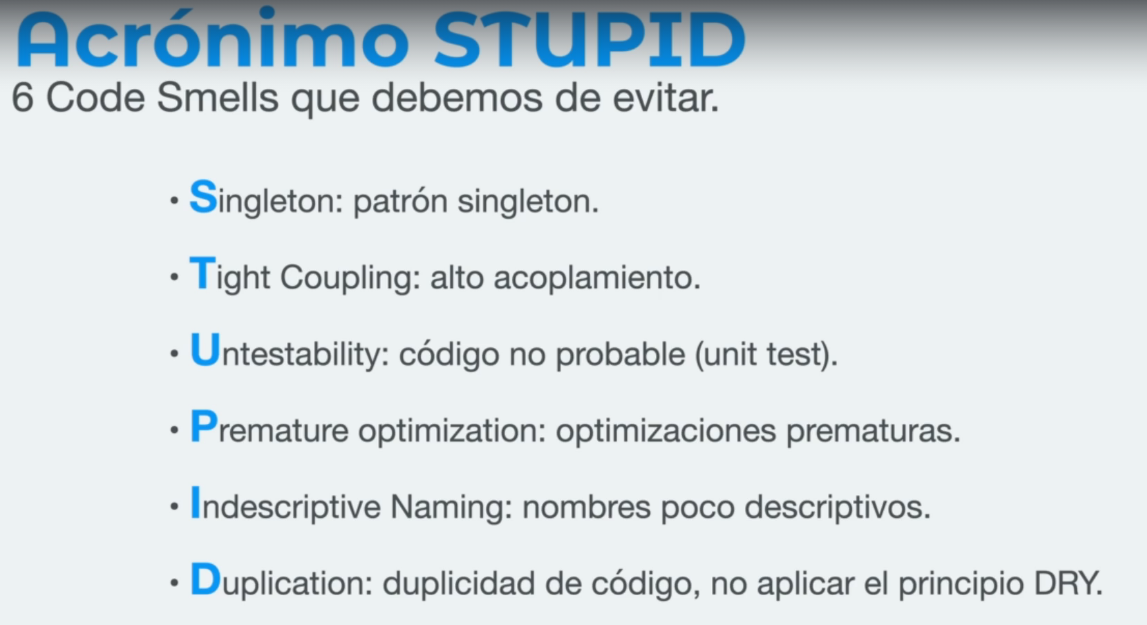
Evitar usar comentarios, solo en caso de ser necesario, por librerías, APIs, etc.

Se debería de comentar, “¿El por qué?”, y no el ¿qué? O, el ¿cómo?

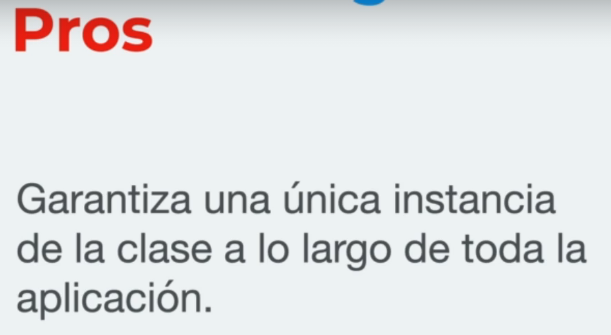
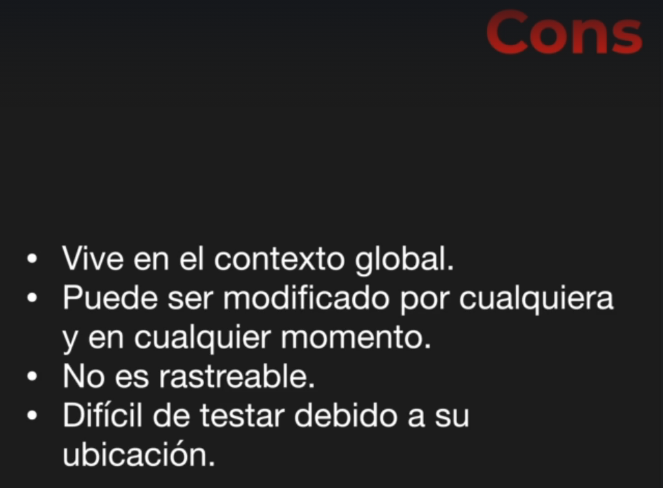


***“No comentes el código mal escrito, reescríbelo”***

# Sección 4: Acrónimo - STUPID



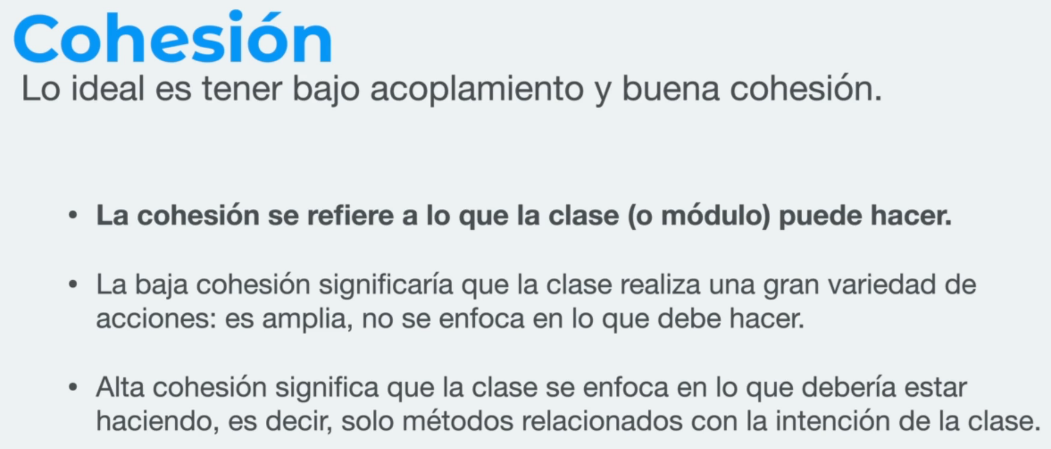
* Patrón Singleton

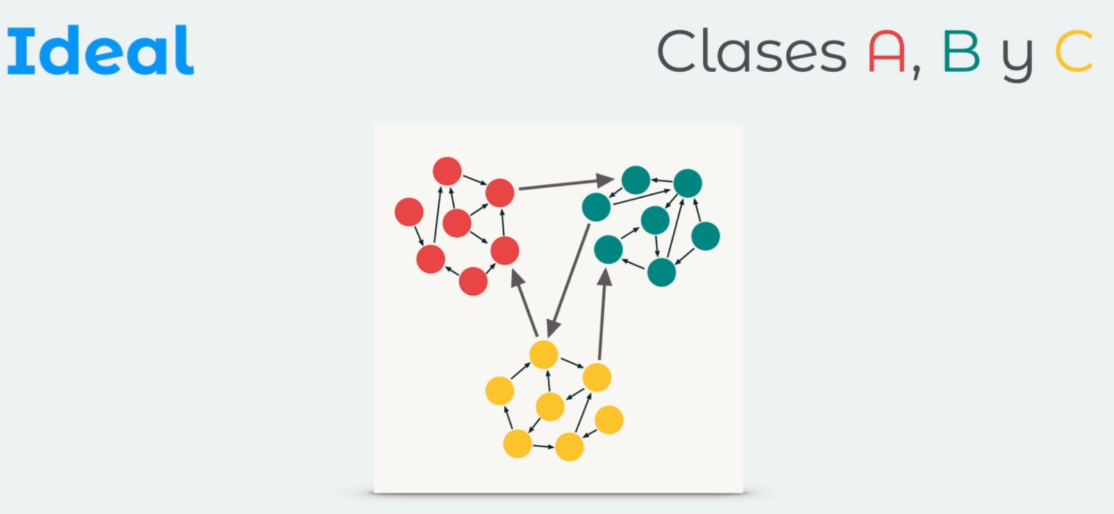
 

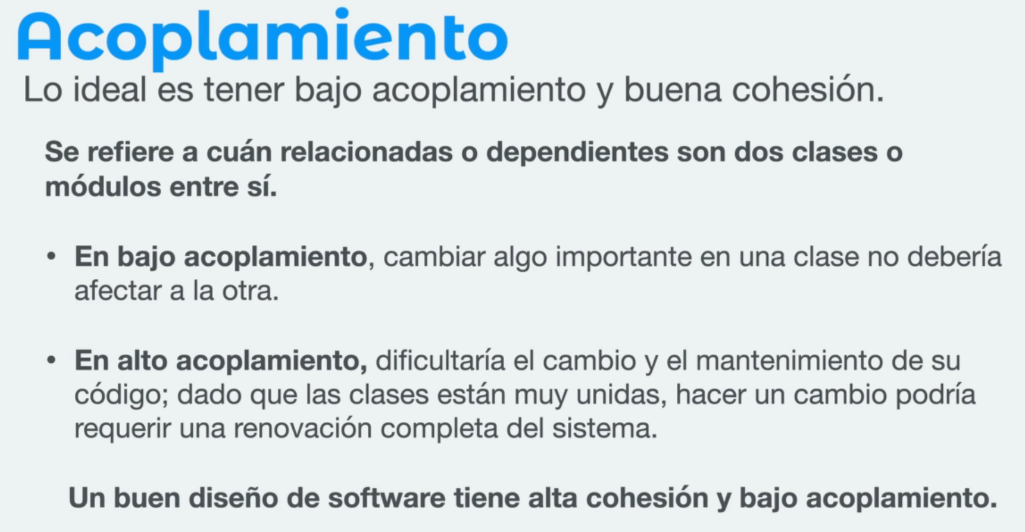
Evitar usar este patrón

* Alto acoplamiento

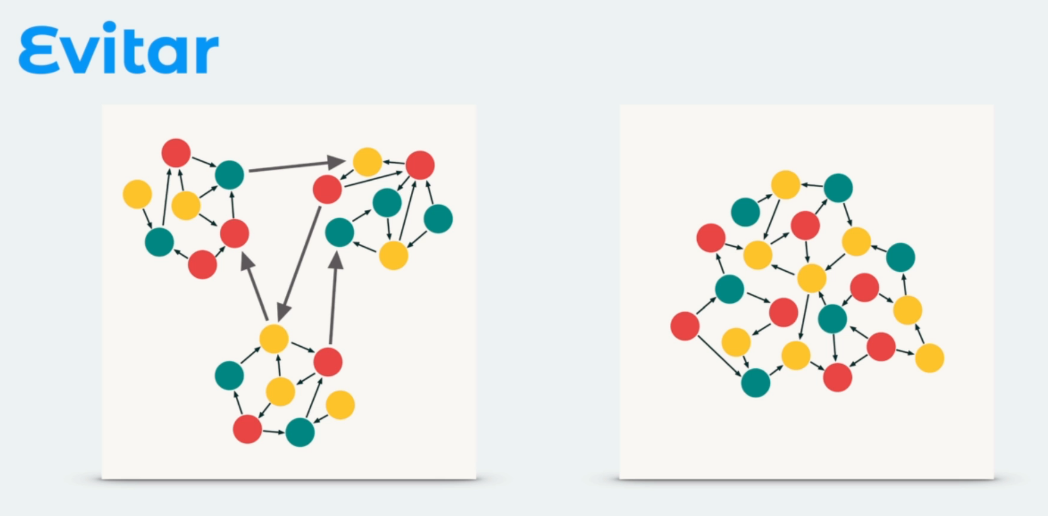
Lo ideal es tener un bajo acoplamiento y una buena cohesión.







Evitar lo de la derecha, lo **ideal**, es lo de la izquierda.



* Codigo no probable (Codigo difícilmente testeable)
* Codigo con alto acoplamiento.
* Codigo con muchas dependencias no inyectadas.
* Dependencias en el contexto global (tipo singleton)

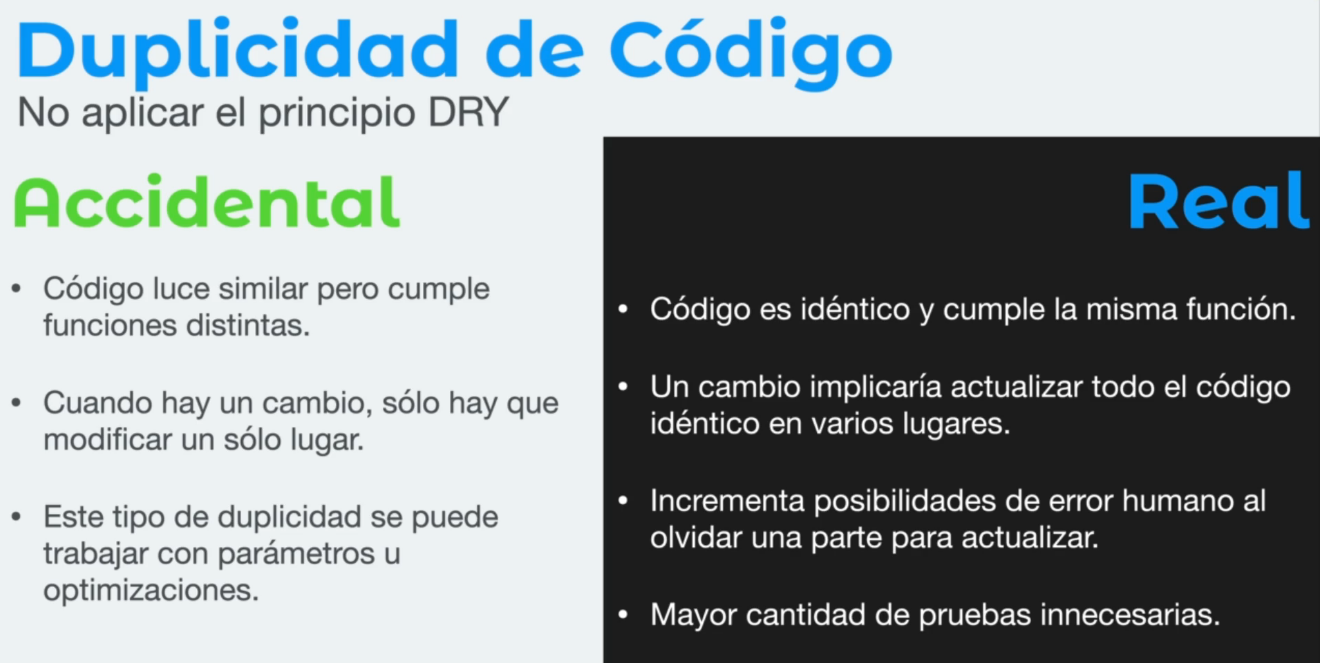
Debemos de tener en mente las pruebas desde la creación del código.

* Optimizaciones prematuras

Hay que encontrar el equilibrio.



* Nombres poco descriptivos
* Nombres de variables mal nombradas.
* Nombres de clases genéricas.
* Nombres de funciones mal nombradas.
* Ser mu especifico o demasiado genérico.
* Duplicidad de Codigo



Otros Code Smells

* Inflación
* Lista larga de parámetros
* Obsesión primitiva

Acopladores

* Feature Envy
* Intimidad inapropiada -> clases usan datos de otras clases
* Cadena de mensajes -> Evitar usar nieto, tataranieto, etc.
* The middle man

Refactoring.Guru

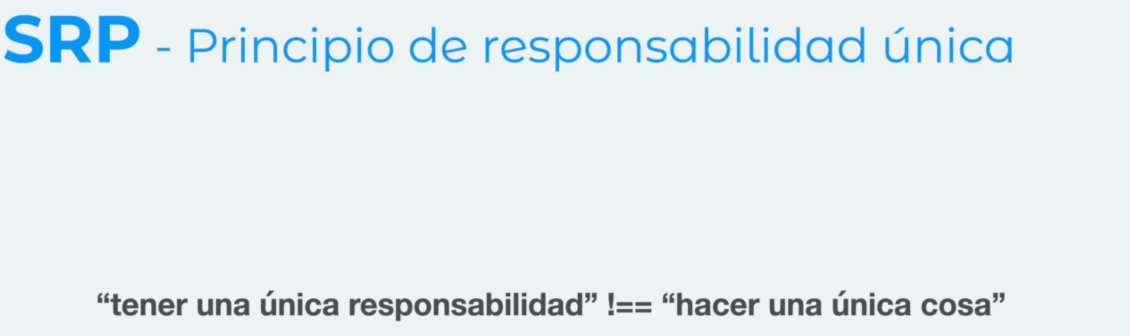
# Sección 5: Principios SOLID

Los **5 principios S.O.L.I.D.** de diseño de software son:

* 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)

Y en esta sección hablaremos sobre cada uno de ellos.

Son principios, como recomendaciones para hacer mejor las cosas, no reglas, que se deben cumplir sí o sí.

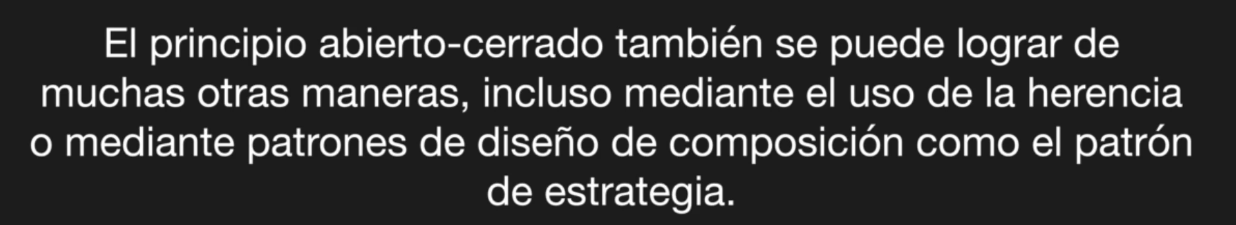


Detectar Violaciones del Solid Clean



Principio de abierto y cerrado

Nuestras entidades, tienen que estar abiertas a la extensión, pero fuera de la disminución

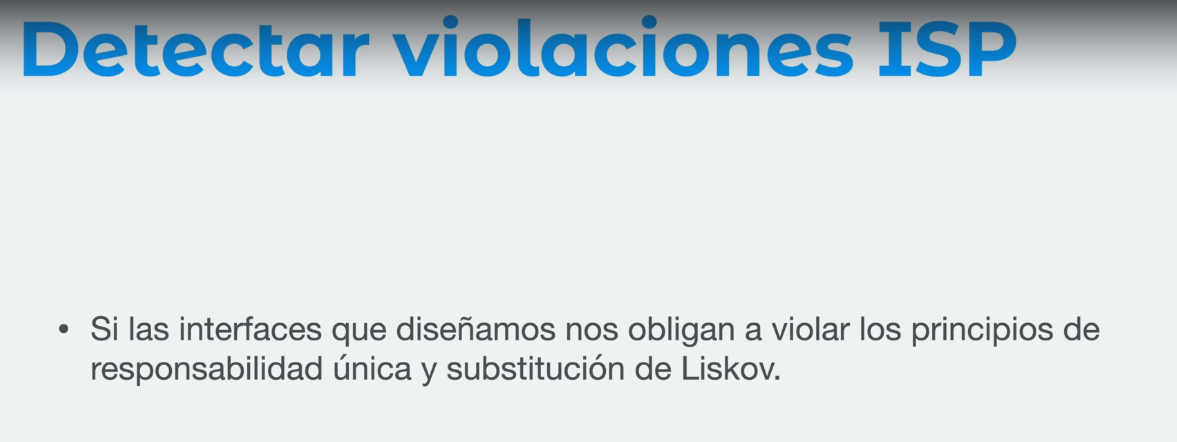


Clases Abstractas

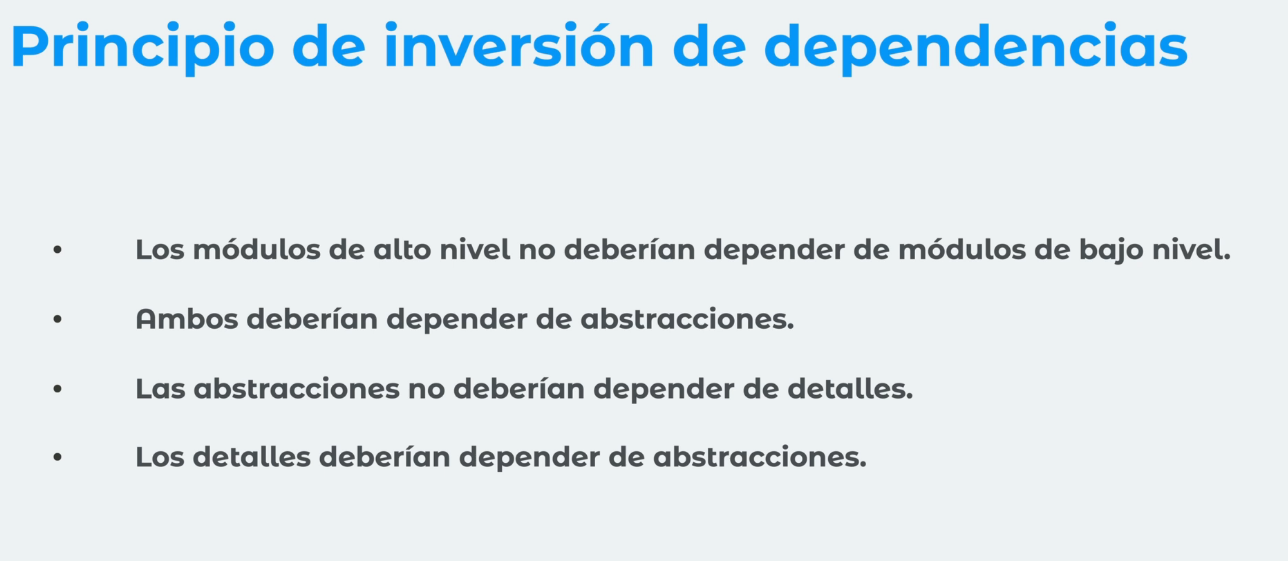
* Sirven para trabajar con herencia

Principio de segregación de la interfaz

* Este principio establece que los clientes no deberían verse forzados, a depender de interfaces, que no usan.



Principio de inversión de dependencias



* Este principio, tiene como base, el basarnos en abstracciones.