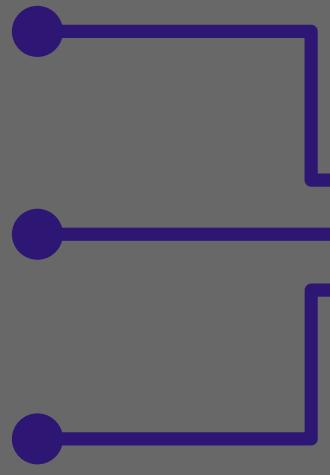


PRINCIPIOS S.O.L.I.D

AUTORÍA: GRUPO N°4
DESARROLLO DE APLICACIONES MÓVILES ANDROID TRAINEE



HISTORIA



A BRIEF OUTLINE

SOLID es el acrónimo que acuñó Michael Feathers, basándose en los principios de la programación orientada a objetos que Robert C. Martin había recopilado en el año 2000 en su paper “Design Principles and Design Patterns”.

SINGLE
RESPONSIBILITY
PRINCIPLE
(SRP)



OPEN/CLOSED
PRINCIPLE
(OCP)

LISKOV
SUBSTITUTION
PRINCIPLE
(LSP)

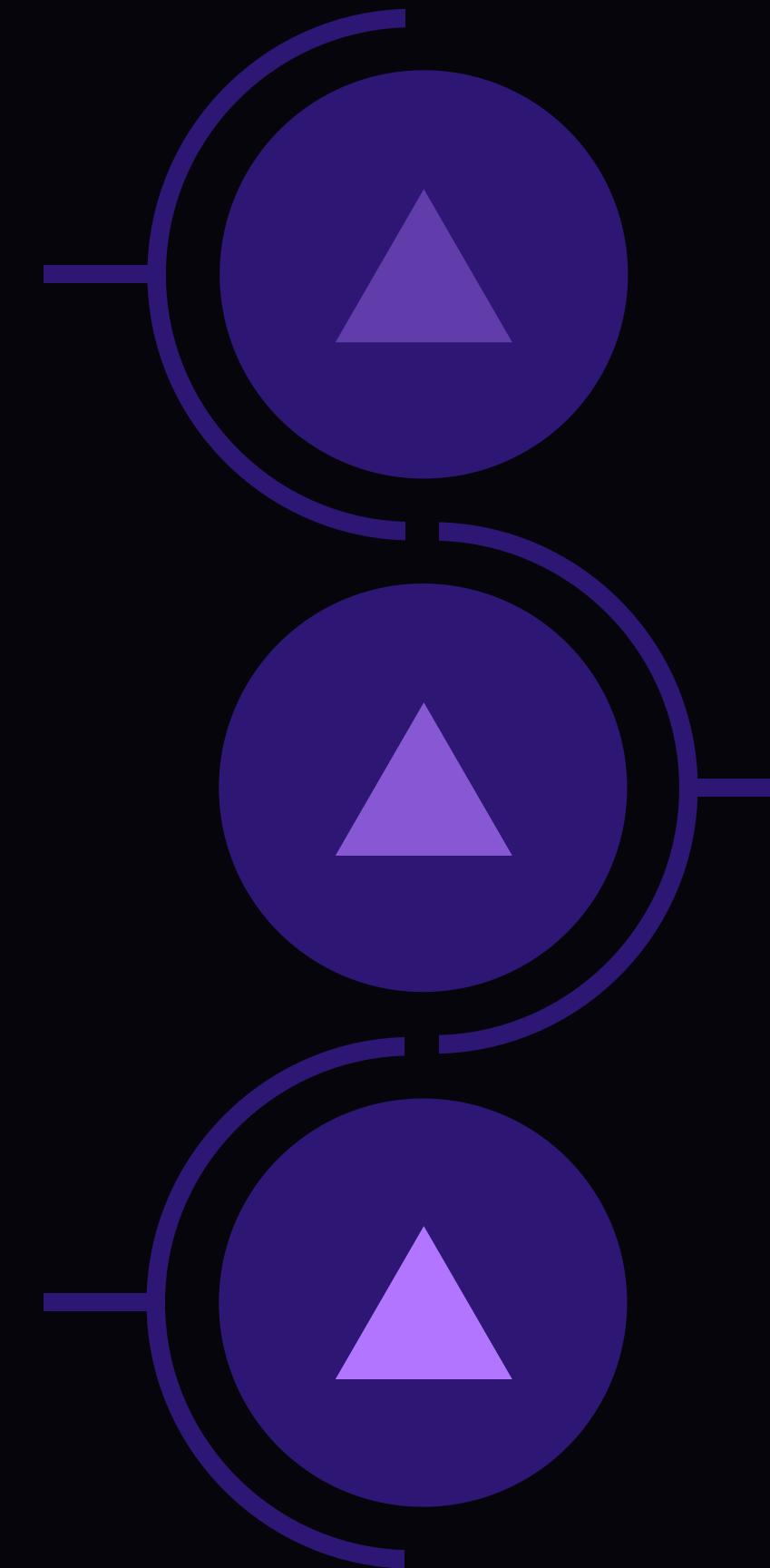


INTERFACE
SEGREGATION
PRINCIPLE
(ISP)

DEPENDENCY
INVERSION
PRINCIPLE
(DIP)



CREAR UN SOFTWARE EFICAZ: QUE CUMPLA CON SU COMETIDO Y QUE SEA ROBUSTO Y ESTABLE.



PERMITIR ESCALABILIDAD: QUE ACEPTE SER AMPLIADO CON NUEVAS FUNCIONALIDADES DE MANERA ÁGIL.

OBJETIVOS:

ESCRIBIR UN CÓDIGO LIMPIO Y FLEXIBLE ANTE LOS CAMBIOS: QUE SE PUEDA MODIFICAR FÁCILMENTE SEGÚN NECESIDAD, QUE SEA REUTILIZABLE Y MANTENIBLE.

CONCEPTOS CLAVES:

ACOPLAMIENTO

se refiere al **grado de interdependencia que tienen dos unidades de software entre sí**, entendiendo por unidades de software: clases, subtipos, métodos, módulos, funciones, bibliotecas, etc.

EJEMPLO: Si dos unidades de software son completamente independientes la una de la otra, decimos que están desacopladas.

COHESIÓN

La cohesión de software es el grado en que **elementos diferentes de un sistema permanecen unidos para alcanzar un mejor resultado que si trabajaran por separado**.

Se refiere a la forma en que podemos agrupar diversas unidades de software para crear una unidad mayor.



"SINGLE RESPONSIBILITY PRINCIPLE (SRP)": PRINCIPIO DE RESPONSABILIDAD ÚNICA

"A CLASS SHOULD HAVE ONE, AND ONLY ONE, REASON TO CHANGE."

El principio de Responsabilidad Única es el más importante y fundamental de SOLID, según este principio "una clase debería tener una, y solo una, razón para cambiar"

"Reúne las cosas que cambian por las mismas razones. Separa aquellas que cambian por razones diferentes"



OPEN/CLOSED PRINCIPLE (OCP): PRINCIPIO DE ABIERTO/CERRADO

“YOU SHOULD BE ABLE TO EXTEND A CLASSES BEHAVIOR, WITHOUT MODIFYING IT.”

"Deberías ser capaz de extender el comportamiento de una clase, sin modificarla". En otras palabras: las clases que usas deberían estar abiertas para poder extenderse y cerradas para modificarse.

Este principio puede ser también una paradoja. Es importante tener en cuenta el Open/Closed Principle (OCP) a la hora de desarrollar clases, librerías o frameworks

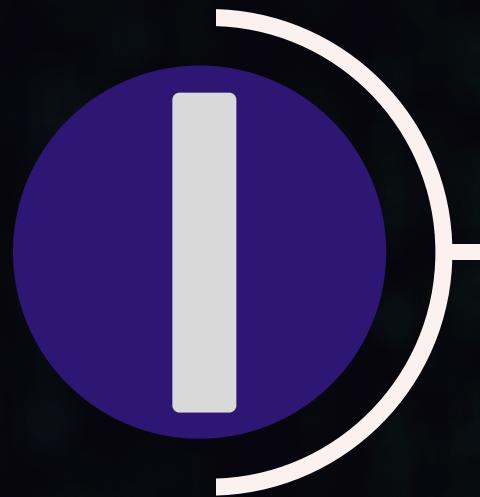


LISKOV SUBSTITUTION PRINCIPLE (LSP): PRINCIPIO DE SUSTITUCIÓN DE LISKOV

“DERIVED CLASSES MUST BE SUBSTITUTABLE FOR THEIR BASE CLASSES.”

Planteado por Barbara Liskov, dice que “las clases derivadas deben poder sustituirse por sus clases base”.

Esto significa que los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema o lo que es lo mismo: si en un programa utilizamos cierta clase, deberíamos poder usar cualquiera de sus subclases sin interferir en la funcionalidad del programa.



INTERFACE SEGREGATION PRINCIPLE (ISP): PRINCIPIO DE SEGREGACIÓN DE LA INTERFAZ

“MAKE FINE GRAINED INTERFACES THAT ARE CLIENT SPECIFIC.”

En el cuarto principio de SOLID, el tío Bob sugiere: “Haz interfaces que sean específicas para un tipo de cliente”, es decir, para una finalidad concreta.

En este sentido, según el Interface Segregation Principle (ISP), es preferible contar con muchas interfaces que definan pocos métodos que tener una interface forzada a implementar muchos métodos a los que no dará uso.



DEPENDENCY INVERSION PRINCIPLE (DIP): PRINCIPIO DE INVERSIÓN DE DEPENDENCIAS

“DEPEND ON ABSTRACTIONS, NOT ON CONCRETIONS.” (DEPENDE DE ABSTRACCIONES, NO DE CLASES CONCRETAS.)

Así, Robert C. Martin recomienda:

1. Los módulos de alto nivel no deberían depender de módulos de bajo nivel.
Ambos deberían depender de abstracciones.
2. Las abstracciones no deberían depender de los detalles. Los detalles
deberían depender de las abstracciones.

El objetivo del Dependency Inversion Principle (DIP) consiste en reducir las dependencias entre los módulos del código, es decir, alcanzar un bajo acoplamiento de las clases.

CONCLUSIONES

Los principios SOLID son eso: principios, es decir, buenas prácticas que pueden ayudar a escribir un mejor código: más limpio, mantenible y escalable.

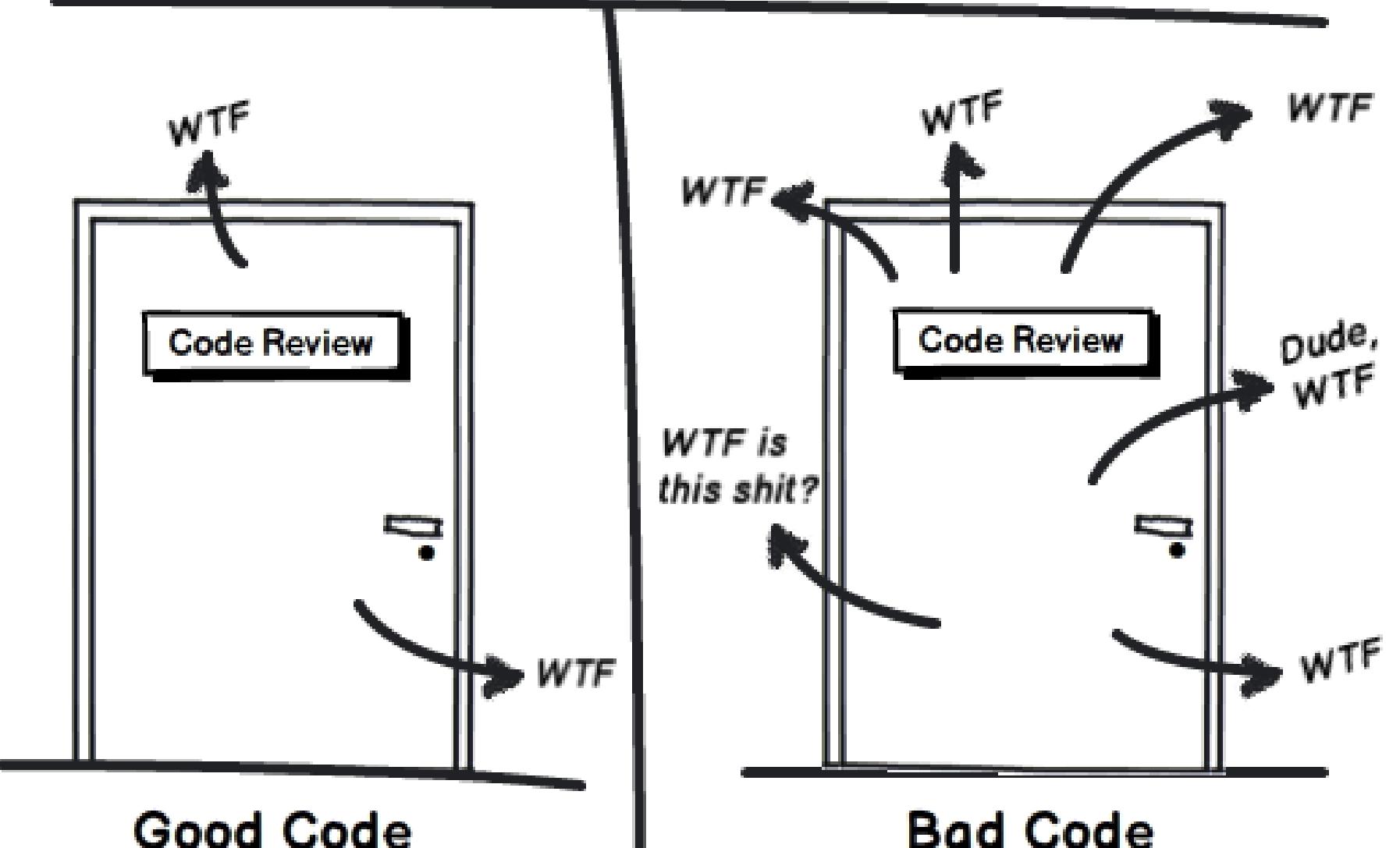
Como indica el propio Robert C. Martin en su artículo “*Getting a SOLID start*” no se trata de reglas, ni leyes, ni verdades absolutas, sino más bien soluciones de sentido común a problemas comunes. Son heurísticos, basados en la experiencia: “se ha observado que funcionan en muchos casos; pero no hay pruebas de que siempre funcionen, ni de que siempre se deban seguir.”

CONCLUSIONES

SOLID nos ayuda a categorizar lo que es un buen o mal código y es innegable que un código limpio tenderá más a salir airosa del “control de calidad de código” WTFs/Minute.

Consejo: cuando estés revisando un código, lleva la cuenta de cuántas veces por minuto sale de tu boca un WTF?

Code Quality Measurement: WTFs/Minute



<http://commadot.com>