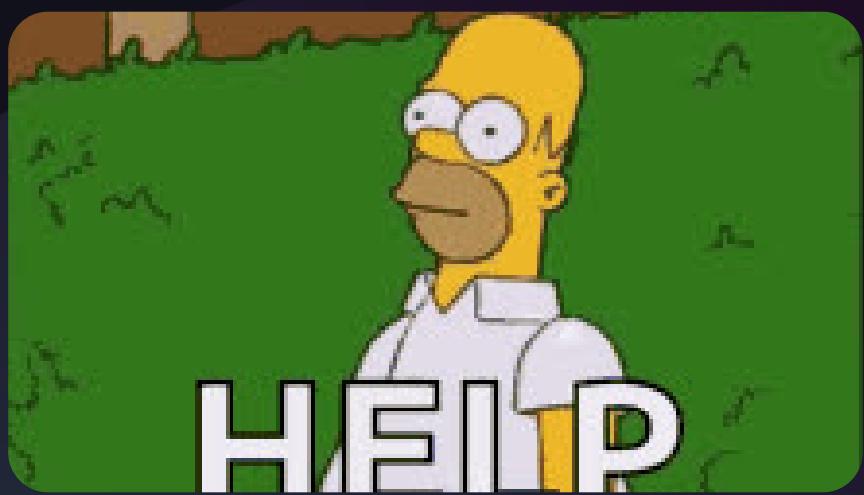


Principios SOLID

DIEGO ALEJANDRO
CIFUENTES BUITRAGO



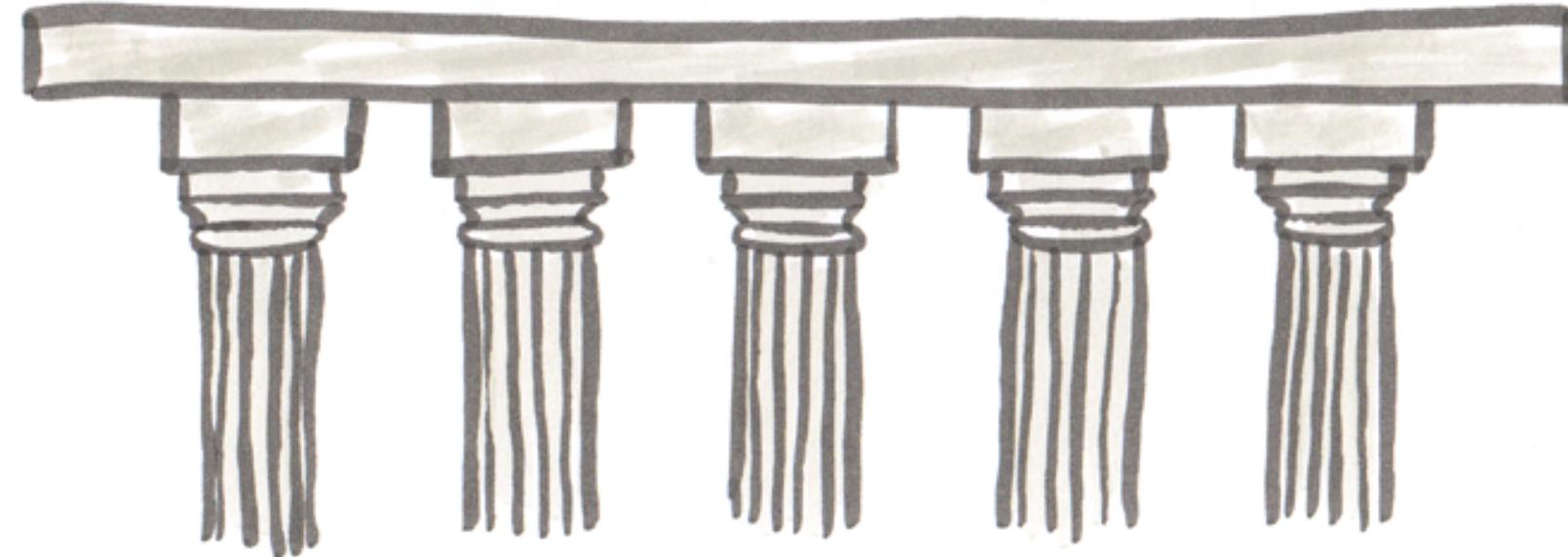
Index



LO QUE CUBRIREMOS EN ESTA SESIÓN

- QUÉ ES.
- ROBERT C. MARTIN
- PRINCIPIOS.
- LEY DE DEMETER

S O L I D



¿Qué es?

Los SOLID son cinco principios del diseño orientado a objetos. Son un conjunto de reglas y mejores prácticas a seguir al diseñar una estructura de clases



ROBERT C. MARTIN

Robert C. Martin ha sido un profesional de software desde 1970. En los últimos 35 años, ha trabajado en varias capacidades en literalmente cientos de proyectos de software. Es autor de libros "históricos" sobre programación ágil, programación extrema, UML, programación orientada a objetos y programación en C++.

Nota: el acrónimo SOLID fue introducido más tarde por Michael Feathers

Principios

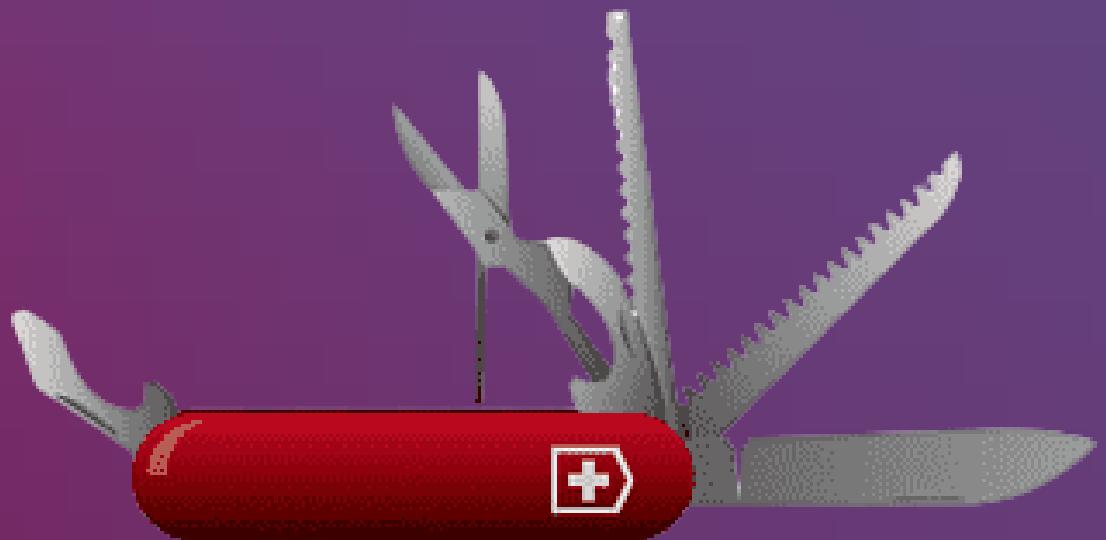
- S: (Single) Principio de responsabilidad única
- O: (Open) Principio abierto-cerrado
- L: (Liskov) Principio de sustitución de Liskov
- I: (Interface) Principio de segregación de interfaz
- D: (Dependency) Principio de inversión de dependencia



El principio de responsabilidad única establece que una clase debe hacer una cosa y, por lo tanto, debe tener una sola razón para cambiar .

Ventajas: Ayuda cuando se generen cambios sobre esta funcionalidad solo tenga que ir a un solo sitio, reutilizar código, pruebas únicas y mantenimiento de código.

S: (SINGLE) PRINCIPIO DE RESPONSABILIDAD ÚNICA



O: (Open) Principio abierto-cerrado

Los objetos o entidades deben estar abiertos por extensión, pero cerrados por modificación.

Ventajas: Facilidad de mantenimiento, reduce generar nuevos errores al extender las funcionalidades que ya estaban desarrolladas, probadas y funcionando. Al momento de implementar test unitarios nos ayuda a modificar los test existentes, si no que también se extienden. Protección de la arquitectura ante cambios de requisitos.



Bertrand Meyer 1888



- L: (Liskov) Principio de sustitución de Liskov

si en alguna parte de nuestro código estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas y que el programa siga siendo válido.

Ventajas: Código más reusable, Facilidad de entendimiento de las jerarquías de clase, Aplicando este principio se puede validar que nuestras abstracciones estan correctas.

Barbara Liskov 1888

I: (Interface) Principio de segregación de interfaz

Los objetos no deberían verse forzados a depender de interfaces que no utilizan.

Ventajas: La segregación de interfaces nos ayuda a no obligar a ninguna clase a implementar métodos que no utiliza. Esto nos evitara problemas que nos pueden llevar a errores inesperados y a dependencias no deseadas. Además nos ayuda a reutilizar código de forma más inteligente.

Robert C. Martin durante unas sesiones de consultoría en Xerox





D: (DEPENDENCY) PRINCIPIO DE INVERSIÓN DE DEPENDENCIA

- A. Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.
- B. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

VENTAJAS:

- Hacemos que el código sea más flexible y escalable así realizar los cambios sin afectar multiples lugares es mucho mas fácil.
- Se ocultan los detalles de implementación.
- Facilita la realizacion de pruebas unitarias ya que la pruebas puede injectar mockups de los servicios a utilizar dentro del constructor.

LEY DE DEMETER



La ley de Demeter es un principio básico de la programación orientada a objetos mencionada en el libro Código Limpio (Clean Code) de Robert C. Martin. Es de gran utilidad para mantener la encapsulación del código de un proyecto.

Como me dijo mi cucha XD
"No hable con desconocidos, solo con amigos"

Nospi

