SOLID VS CUPID

Andrea Delgado Alonso
Héctor Lavandeira Fernández
Laura Vigil Laruelo

01

SOLID

Principio de responsabilidad única

El objetivo es que una clase solo tenga una responsabilidad.

Ventajas:

- Simplifica la realización de pruebas
- Reduce el acoplamiento
- Mejora la mantenibilidad del proyecto.

Principio abierto - cerrado

Una clase tiene que estar abierta para la extensión, pero cerrada para la modificación.

Objetivo:

- Evitar modificar código sin querer
- Evitar errores

Principio de sustitución de Liskov

Cualquier subclase debe poder ser sustituida por su clase padre sin alterar el funcionamiento del proyecto.

Gracias a este principio se mantendrá la integridad del proyecto.

Principio de segregación de la interfaz

Este principio dice que ningun cliente debe depender de metodos que no utiliza.

Consejos:

- Crear interfaces que tengan una finalidad concreta
- Es mejor tener interfaces con pocos métodos, que una interfaz con muchos métodos que no utiliza

Principio de inversión de dependencias

Los modulos de alto nivel no deben depende de los modulos de bajo nivel. Y las abstracciones no deben depender de los detalles, si no los detalles de ellas.

Objetivo:

- Desacoplar las clases para que estas funcionen por si solas, sin depender de otras.

02

CUPID

Características comunes de las propiedades

01

PRÁCTICA

Cuando es fácil de articular, evaluar y adaptar

HUMANO

Punto de vista humano

02

03

EN CAPAS

Útil para los principiantes y capacidad de profundizar para los expertos

Propiedades

Componible

- Pequeño
- Nombre significativo
- Pocas dependencias

Filosofía Unix

- Modelo simple y consistente
- Centrada en lo que hace el código

Propiedades

Predecible

- Que el código haga lo que se espere de él
- Consistente
- Observable

Idiomático

- Código natural
- Establece restricciones y pautas

Propiedades

Basado en dominio

- Lenguaje del dominio
- Estructura del dominio
- Limites basados en dominios

03

SOLID VS CUPID

¿Porque estan mal los solid?

01

Principio responsabilidad única

- Principio vago inutilmente
- El código tiene varias razones para cambiar, y no todas están en tu cabeza

02

Principio abierto – cerrado

El principio de acumulación de cruft:

- Si los requisitos cambian, cambiamos el código
- Este principio era útil cuando el código era mas costoso de cambiar

¿Porque estan mal los solid?

03

Principio de sustitución de Liskov

Principio de advertencia de Drucker
- Necesitamos tipos pequeños y simples con los que componer estructuras complejas

05

Principio de segregación de interfaces

El principio de puerta del establo:
- Diseñar clases pequeñas basadas en roles
- Así ningún cliente depende de Métodos que no usa

¿Por qué estan mal los solid?

05

Principio de inversión de dependencias

Principio de objetivo equivocado
- Reutilizar esta sobrevalorado, hay que diseñar para usar

CONCLUSION:

Escribir código simple

Preguntas