SOLID vs CUPID

Andrea María Delgado Alonso – UO271355

Héctor Lavandeira Fernández – UO277303

Laura Vigil Laruelo – UO271432

SOLID

SOLID es un acrónimo que representa una serie de principios de programación y diseño de software aplicables a la programación orientada a objetos, cuyos objetivos principales son:

1. Crear software de calidad que funcione y a la vez sea robusto
2. Escribir código limpio y flexible ante cambios
3. Permitir la escabilidad

Propiedades SOLID:

1. Principio de responsabilidad única → según este principio, una clase o un método debe hacer solamente una cosa, es decir, debe tener una tarea especifica y acotada. De esta manera, la clase o método será más fácil de mantener. Con esto se consigue simplificar la tarea de realizar pruebas de testing, se disminuye el acoplamiento y mejora la organización de clases y paquetes del proyecto.

2. Principio de abierto y cerrado → formulado por Bertrand Meyer en 1988, el segundo principio dice que las clases deberían estar abiertas para poder extenderse, y cerradas para modificarse. El objetivo de esto es evitar que el código se modifique sin intención y cause errores.

3. Principio de sustitución de Liskov → este principio, cuyo nombre viene de Barbara Liskov, quien lo creó, dice que cualquier subclase debería poder ser sustituida por la clase padre sin alterar el correcto funcionamiento del sistema. Gracias a este principio se mantendrá la integridad del proyecto.

4. Principio de segregación de la interfaz → según este principio, ningún cliente debería depender de los métodos que no utiliza. Por tanto, deben crearse interfaces específicas para un tipo de cliente, que tengan una finalidad concreta. Es mejor tener muchas interfaces con pocos métodos, que tener una interfaz forzada con muchos métodos a los que no dará uso.

5. Principio de inversión de la dependencia → este principio dice que los módulos de alto nivel no deberían depender de los módulos de bajo nivel. También dice que las abstracciones no deberían depender de los detalles, sino que son los detalles los que deberían depender de las abstracciones. El objetivo es desacoplar las clases para que estas puedan funcionar por sí solas, sin depender de otras.

CUPID

CUPID es una alternativa a SOLID, sustituye los principios por propiedades, es decir, características del código interrelacionadas entre sí que permiten definir un objetivo, el código realizado permite acercarse a este.

¿Qué características comunes deben de tener estas propiedades?

Práctica: una propiedad es practica cuando es fácil de:

1. Articular, es decir, sencilla
2. Evaluar
3. Adaptar

Humano: las propiedades deben de poder ser comprendidas desde el punto de vista humana, no el código en sí.

En capas: los principiantes deben de encontrar estas propiedades útiles para caminar hacia un objetivo, pero además permiten a los más experto tener más cabida para profundizar en el software

Propiedades CUPID

1. Componible → algo componible es:
    - Pequeño: da la ventaja de cuando una API es pequeña existen errores
    - Nombre que revele las intenciones: así nuestro código es más sencillo de encontrar, además si sabemos qué hace un componente podemos evaluar rápidamente si es algo que nos interesa o no
    - Pocas dependencias: así se reduce la incompatibilidad de versiones
2. Filosofía Unix
    - Modelo simple y consistente que realice una sola cosa
    - Esto es distinto a responsabilidad única, esta filosofía se centra en que hace el código, no como va a cambiar
3. Predecible
    - El código debe comportarse tal y como se refleja en este
    - Sea consistente es decir siempre se comporte de la misma manera.
    - Observable, pero de una manera más técnica, podemos inferir su estado interno a partir de sus salidas.
4. Idiomático
    - El código debe de verse como cualquier otro código, que sea natural para las personas.
    - Establece restricciones y pautas para fomentar la coherencia.
5. Basado en el dominio
    - Usar el lenguaje del dominio
    - Usar la estructura del dominio, la organización del proyecto debe girar en torno a una solución no al framework (controladores, modelo, view...).
    - Límites basados en dominios → se quiere que la estructura sea la que informe como hago el código como lo despliego.

SOLID vs CUPID

Los principios SOLID a menudo se presentan como el núcleo de una buena práctica de diseño de código. Sin embargo, cada uno de S, O, L, I y D no significa necesariamente lo que los programadores esperan que signifique o se les enseñe.

Porque los SOLID están mal

EL principio de responsabilidad única dice que el código tiene que hacer una sola cosa y solo debe tener una razón para el cambio. Este principio SOLID lo catalogaron como el "principio vago inútilmente", ya que cualquier código tiene varias razones para cambiar que probablemente no pasen ni por tu cabeza. Por lo que se dijo que hay que escribir código simple que encaje en tu cabeza, es decir, que puedas razonar acerca de ello.

El principio abierto-cerrado dice que nuestro código tiene que estar abierto para extensión, cerrado para modificación. Pero el principio de acumulación de cruft dice que cuando los requisitos cambian, el código existente ahora es incorrecto, así que lo reemplazamos con un código que funcione, ya que ahora el código es maleable. Antes el principio abierto-cerrado podía tener mas sentido, cuando el código era costoso de cambiar.

El principio de sustitución de Liskov, dice que la sustitución con un subtipo conserva todas las propiedades deseables del tipo original. El principio de advertencia de Drucker nos dice que no hay nada tan inútil como hacer con gran eficiencia algo que no debería hacerse en absoluto. Lo que queremos son tipos pequeños y simples con los que podemos componer cualquier estructura mas compleja.

El principio de segregación de interfaz dice que cualquier interfaz pequeña es mejor que un objeto grande. El principio de puerta del establo dice que prácticamente todo es mejor que un objeto grande. Hay que diseñar clases pequeñas basadas en roles y ningún cliente depende de métodos que no utiliza

El principio de inversión de dependencia dice que los módulos de alto nivel no deberían depender de los módulos de nivel inferior. El principio de objetivo equivocado dice que reutilizar está sobrevalorado, hay que diseñar para usar.

En resumen, hay que escribir código sencillo, ya que el código simple es fácil de razonar y se puede hacer fácilmente varias cosas relacionadas.

Bibliografía

https://gustavopeiretti.com/principios-solid-con-ejemplos/
https://profile.es/blog/principios-solid-desarrollo-software-calidad/
https://desarrollowp.com/blog/tutoriales/principios-solid-de-la-programacion-orientada-a-objetos/#:~:text=SOLID%20es%20un%20acr%C3%B3nimo%20acu%C3%B1ado,con%20los%20patrones%20de%20dise%C3%B1o.
CUPID—for joyful coding - Dan North & Associates Ltd
SOLID - Wikipedia, la enciclopedia libre
https://www.youtube.com/watch?v=2QahGarHpXQ
https://speakerdeck.com/tastapod/why-every-element-of-solid-is-wrong
https://dannorth.net/2021/03/16/cupid-the-back-story/