SOLID vs CUPID

- Efrén García Valencia UO277189
- José Rodríguez Rico UO276922
- Daniel Machado Sánchez UO276257

Principios SOLID

Fueron introducidos por Robert C. Martin en su artículo académico **"Design Principles and Design Patterns"**. Estos conceptos fueron reforzados más adelante por Michael Feathers, que introdujo el acrónimo SOLID.

Básicamente son principios de diseño que buscan crear código **mantenible, entendible y flexible**. La idea fundamental que persiguen es reducir la complejidad a medida que nuestras aplicaciones aumenten en tamaño, ahorrándonos muchos problemas en el futuro.

Single Responsibility (Responsabilidad única): nos dice que una clase sólo debe tener una única responsabilidad, es decir, un único motivo para el cambio.

Open/Closed (Abierto/Cerrado): dice que las clases deberían estar abiertas para la extensión, pero cerradas para la modificación.

Liskov Substitution (Principio de sustitución de Liskov): los objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento de un programa.

Interface Segregation (Segregación de interfaces): es mejor tener muchas interfaces específicas para cada cliente que una sola interfaz de propósito general.

Dependency Inversion (Inversión de dependencias)

- Los módulos de alto nivel no deben depender de los de bajo nivel; ambos deben depender de abstracciones.
- Las abstracciones no deben depender de los detalles, sino éstos de las abstracciones.

Principios contra propiedades:

Los principios SOLID como tal pueden ser problemáticos a la hora de reemplazar cada uno por una idea útil o relevante. Esto se debe a que los principios son reglas que se pueden cumplir o no cumplir.

En cambio, las propiedades son cualidades o características del código a seguir. Estas definen un objetivo hacia el que moverse con una idea clara. A demás están relacionadas entre si, por lo que cualquier cambio para mejorar una propiedad seguramente tenga un efecto positivo en otra

Porque todos los principios SOLID están mal:

- Single responsability:

En general cualquier código puede tener varias razones para su cambio. Esto conlleva que la idea del principio de responsabilidad única es demasiado ambigua y confusa.

El principal tema que Dan North trata sobre este principio es que un código solo debe cambiar por algo en el que se pueda razonar sobre ello, es decir, por algo que "tengas en la cabeza".

- Open-closed:

La idea de este principio se ha quedado un poco anticuada. Hoy en día el código ya no es como ladrillos de edificios, difícil o arriesgado de cambiar. Por ello si necesitas hacer otra cosa o simplificar el código, la mejor solución es cambiarlo.

- Liskov Substitution principle:

El principal problema de este principio es que hay una gran cantidad de formas de combinar los subtipos y las subclases. De esta manera lo que realmente interesa es construir tipos pequeños simples y fáciles de razonar para poder componerlos en estructuras complejas.

- Interface segregation principle:

Si tienes un problema de código en el que existen muchas clases que comparten funcionalidad y quieres encapsularlas en una interfaz intermedia, lo que estás aplicando no es un principio si no un patrón de diseño. Si hubiese sido un "principio" se llamaría Stable Door Principle (principio de puerta estable). La mejor solución sería no meterse en ese lío y tener clases simples y fáciles de razonar.

- Dependecy inversión principle:

Por lo general no hay nada de malo en este principio, pero es verdad que a lo largo de estas décadas cumplir con este ha causado pérdidas grotescas de dinero. El principal problema es que la mayoría de las dependencias no necesitan inversión si no opciones. Por ello la solución es escribir código simple centrado en el uso antes que en la reutilización.

CUPID

North decía que los principios SOLID no están no están equivocados, pero son demasiado vagos para apoyarlo como desarrollador.

Por eso él desarrollo los principios CUPID cuando le pusieron el desafío de "Si pudiera dar principios para el desarrollo de software moderno en estos días, ¿cuáles elegiría?".

Lo que quería Daniel era generar una colección de propiedades centradas en la persona que desarrolla el código para que los programadores escriban código que las personas puedan entender, no solo una computadora.

Estas propiedades son:

- **Componible**: el código que es fácil de usar se puede usar en varias ocasiones. Se basa en que cuanto menos haya que aprender, menos puede ir a mal y hay menos

conflictos. Intención-revelación del nombre y del propósito: Fácil de descubrir, fácil de evaluar. dependencias mínimas

- **Filosofía de Unix**: un buen código garantiza que un programa/clase/función solo realice exactamente una tarea. Lo importante es que hace el código, no cómo cambia.
- **Predecible**: el código se comporta como se esperaba sin sorpresas. Es determinista, hace la misma cosa siempre y con unas características de funcionamiento bien entendidas. Y es observable en el lado técnico, su estado interno se puede derivar de las salidas, pero no se puede cambiar.
- **Idiomático**: Tratar con un buen código debe sentirse natural. Se utilizan los modismos del contexto (dominio, empresa, equipo) así como el ecosistema del lenguaje en el que está escrito.
- **Basado en dominio**: el lenguaje de dominio y su estructura se utilizan para un buen código. La estructura del código debe reflejar la solución y no el framework o los conceptos utilizados. En lugar de conceptos técnicos (modelos, vistas, controladores), se utilizan términos de dominio (documentos, pagos). Los límites del dominio deben tenerse en cuenta como límites del módulo y unidades de implementación.

En definitiva, CUPID no es un reemplazo de SOLID, sino un enriquecimiento.

BIBLIOGRAFÍA

https://www.baeldung.com/solid-principles

https://speakerdeck.com/tastapod/cupid-for-joyful-coding

https://dannorth.net/2021/03/16/cupid-the-back-story/

https://bbv.ch/cupid/

Conceptos de SOLID de la asignatura de Diseño del Software.