Tarea 02: Diseño de Software SOLID

Programación Avanzada

2023-1

Valeria Jimeno Villegas - vjimenov@ciencias.unam.mx

6 de septiembre de 2022

Introducción

La primera fase del ciclo del desarrollo de software es el **diseño**, este tiene un impacto importantísimo durante todo el proceso. Si el diseño es bueno, todas las demás fases del desarrollo de software, como la codificación, el mantenimiento y el soporte, serán menos estresantes y sin complicaciones. El diseño de software ayuda a imaginar el sistema en un plano inicial, además de que puede ayudar a reducir los costos del desarrollo y mantenimiento del software. Es por eso que es necesario hacerlo de una forma organizada y así construir un código más eficiente y fácil de mantener. Para eso, en el 2000 Robert C. Martin propuso un conjunto de cinco principios, "SOLID"[1].

- S: Single Responsibility Principle
- O: Open-Closed Principle
- L: Liskov Substitution Principle
- I: Interface Segregation Principle
- D: Dependency Inversion Principle

A continuación se definen cada uno y se da un ejemplo de cada uno de ellos. Los conceptos mencionados en este trabajo están basado en: [1], [2], [3] y [4].

S. Principio de Responsabilidad Única (Single Responsibility Principle)

"A class should have one, and only one, reason to change."[1]

Este principio se refiere a que una clase debería tener una, y solo una, razón para cambiar. Para Robert C. Martin, "razón para cambiar" significa "responsabilidad". Por lo que podemos concluir que seguir este principio significa que cada clase solo debe llevar a cabo una cosa. Más simplemente, cada clase debe resolver un solo problema.

Por ejemplo, si tenemos una lista de n números: $A = [a_1, \ldots, a_n]$ y tenemos el objetivo de calcular ciertas funciones matemáticas para poder llevar a cabo un análisis estadístico (media, máximo y mínimo) por lo que si no tomáramos en cuenta este principio crearíamos una función que llevase a cabo todas las funciones. Se pudiera argumentar que resuelve un solo problema, que es el de hacer un análisis estadístico, sin embargo lo más óptimo sería programar cada función por separado y luego crear una función "principal" en donde se mande llamar cada una de las funciones.

O. Principio de Abierto/Cerrado (Open-Closed Principle)

"You should be able to extend a classes behavior, without modifying it."[1]

La idea del principio abierto/cerrado es que las clases existentes y bien probadas deberán modificarse cuando sea necesario agregar algo. Sin embargo, cambiar la clase puede generar problemas o errores. En lugar de cambiar la clase, lo ideal es extenderla. Un solo cambio en un programa puede causar una serie de cambios en cascada de módulos dependientes a este, lo que lo vuelve frágil, impredecible y en algunos casos no se podrá reutilizar. Las clases cumplen con este principio si son:

1. Abiertas para extensión, lo que significa que el comportamiento de la clase puede extenderse.
2. Cerrada por modificación, lo que significa que el código fuente está configurado y no se puede cambiar.

Siguiendo con el ejemplo del principio anterior, supongamos que convertimos la función principal en una clase llamada "operaciones" y las funciones matemáticas ahora las representamos como subclases de la clase operaciones, es decir: tenemos una clase "operaciones" y tres subclases "media", "mínimo" y "máximo". Si se quisiera ahora calcular la mediana de la lista A entonces sólo se tendría que crear otra subclase

"mediana" que herede de la clase "operaciones".

L. Principio de Sustitución de Liskov (Liskov Substitution Principle)

"Derived classes must be substitutable for their base classes."[1]

La L de SOLID alude al apellido de quien lo creó, Barbara Liskov, y dice que "las clases derivadas deben poder sustituirse por sus clases base"[4].

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.

Por ejemplo, si tenemos una clase "Rectángulo" la cual tiene dos atributos, "alto" y "ancho" y un método "área", y por otro lado tenemos una una subclase que hereda de "Rectángulo", llamada "Cuadrado", en donde esta asume que el "ancho" y "alto" es el mismo. Si un objeto "Cuadrado" se usa en un contexto en el que se espera un "Rectángulo", puede ocurrir un comportamiento inesperado porque las dimensiones de un "Cuadrado" no pueden (o más bien no deben) modificarse de forma independiente.

I. Principio de Segregación de la Interfaz (Interface Segregation Principle)

"Make fine grained interfaces that are client specific."[1]

La idea general del principio de segregación de interfaces es que es preferible contar con muchas interfaces específicas para cada cliente para así no obligar a esto a implementar interfaces que no utilizan, que es lo que pasaría si se tuviese una gran interfaz de propósito general.

Por ejemplo si tuviéramos una clase "Mamíferos", la cual está compuesta por dos métodos: "Mamíferos.caminar()" y "Mamíferos.nadar()". Y por otro lado tenemos dos subclases: "Humano" y "Ballena". "Humano" tiene ambos métodos, sin embargo "Ballena" sólo tiene el método ".nadar()". De lo anterior que se pudieran hacer que cada subclase pudiera heredar solo lo que necesita, en este caso se pudieran hacer dos clases nuevas: "Walker" y "Swimmer", además de eliminar la clase "Mamíferos"

y dejar las clases “Humano” y “Ballena”, las cuales heredarán los métodos de las clases “Walker” y “Swimmer”, y sólo “Swimmer”, respectivamente.

D. Principio de Inversión de Dependencias (Dependency Inversion Principle)

“Depend on abstractions, not on concretions.”[1]

En este principio, Robert C. Martin recomienda[2] :

Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones. 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.

Por ejemplo[1], si vamos a una tienda local a comprar algo y decidimos pagar con tarjeta de débito. Entonces, cuando le damos la tarjeta al empleado para realizar el pago, el empleado no se molesta en verificar qué tipo de tarjeta le dimos. No importa el tipo de tarjeta de crédito o débito que tengamos para pagar; el empleado simplemente la deslizará. Entonces, en este ejemplo, se puede notar que tanto nosotros como el empleado dependemos de la abstracción de la tarjeta de crédito y no nos interesan los detalles de la tarjeta. Esto es lo que es un principio de inversión de dependencia.

Conclusiones

A lo largo de mi experiencia y de mis estudios no me había topado con estos principios del diseño de software. Creo que una de las bases más importantes de la programación es el diseño de los programas y si no tenemos en cuenta estos principios pudiera ser que los programas que hagamos pudieran enfrentar a ciertos problemas ya sea en el proceso de producción o de mantenimientos. Además de que un programa, en el caso de POO, que tenga un conjunto de clases bien diseñadas y escritas puede minimizar el tiempo que se le dedique al código así como minimizar el número de *bugs* que se puedan generar. Creo que será interesante comenzar a ponerlos en practica y ver que tanto nos cuesta el cambio de pensamiento.

[1]Ejemplo sacado de [5].

Referencias

[1] R. C. Martin, "Design principles and design patterns," *Object Mentor*, vol. 1, no. 34, p. 597, 2000.

[2] H. Singh and S. I. Hassan, "Effect of solid design principles on quality of software: An empirical assessment," *International Journal of Scientific & Engineering Research*, vol. 6, no. 4, 2015.

[3] J. C. Martínez, C. Henao, F. Henao, and E. Zapata, "Utilización de arquitecturas limpias para trabajo con buenas prácticas en la construcción de aplicaciones java." *Revista Innovación Digital y Desarrollo Sostenible-IDS*, vol. 1, no. 2, pp. 133–140, 2021.

[4] "Solid: los 5 principios que te ayudarán a desarrollar software de calidad," https://profile.es/blog/principios-solid-desarrollo-software-calidad/, accessed: 2020-09-03.

[5] "All you need to know about solid principles in java," https://www.edureka.co/blog/solid-principles-in-java/, accessed: 2020-09-03.