Tarea 2
Principios de diseño de software SOLID

Emmanuel Peto Gutiérrez

7 de septiembre de 2022

1. Principio de responsabilidad única (SRP)

Una clase debe tener una y solo una razón para cambiar, lo que significa que debe tener un solo trabajo.

Ejemplo. considere una aplicación que toma una colección de formas, entre círculos y cuadrados, y calcula la suma del área de todas las formas de la colección. Si es un cuadrado calcula ℓ^2, donde ℓ es el lado. Si es un círculo calcula πr^2, donde r es el radio.

Se puede crear una clase `AreaCalculator` que se encargue de sumar las áreas de las figuras. Dentro de la clase debe haber una función que reciba una colección de figuras, calcule la suma de las áreas y devuelva el resultado como una cadena de caracteres (String).

Ahora suponga que un usuario desea elegir que la salida pueda presentarse en formato de JSON o en formato de HTML. No se debe modificar la clase `AreaCalculator` porque su única función debe ser calcular las áreas, así que se puede crear otra clase `SumCalculatorOutputter` que se encargue de recibir una cadena de texto y darle el formato de JSON o de HTML, según lo requiera el usuario.

Conclusiones: Este principio sugiere que se debe dividir el software en partes pequeñas; es decir, en clases pequeñas y con tareas específicas. Cuando se requiera modificar el software, será más fácil para los desarrolladores modificar una clase con pocas líneas de código que un archivo con cientos de líneas. También, cuando se requiera añadir alguna funcionalidad al software se podría, por ejemplo, crear una clase nueva en vez de agregar funciones a las clases ya existentes.

2. Responsabilidad abierta-cerrada (OCR)

Simplemente significa esto: debemos escribir nuestros módulos de forma que puedan ser extendidos, sin requerir que sean modificados. Dicho de otra forma, queremos ser capaces de cambiar lo que hacen los módulos, sin cambiar el código fuente de los módulos.

Ejemplo. Suponga que se tiene la superclase `Figura`, la cual tiene el atributo `tipo`, el cual es una cadena que nos dice si la figura es un círculo, cuadrado, triángulo, etc.

Luego, suponga que se tiene una función `calculaArea(Figura f)` en alguna clase llamada `CalculoFiguras` la cual, evidentemente, calcula el área de la figura que recibe. Para saber de qué tipo de figura se trata se debe hacer un switch-case o un if-else y así hacer un cálculo diferente dependiendo si es círculo o cuadrado u otra figura.

Si se crea una clase nueva, por ejemplo `Trapecio`, se debe agregar este caso al switch de la función `calculaArea` para que soporte la operación para el trapecio. De hecho, cada vez que se agregue una figura nueva se debe modificar la clase `CalculoFiguras` para que el programa sea correcto.

En este caso es más conveniente que cada tipo de figura tenga su propio método `area()`, y así, en la función `calculaArea` simplemente devuelve el resultado de llamar el método `area` de `f`. Con esto se extiende de forma automática la función `calculaArea` sin tener que modificar la clase `CalculoFiguras`.

Conclusiones. En este principio es donde se aplican los conceptos de polimorfismo (como en el ejemplo mencionado), tipos genéricos y las plantillas (templates de C++). Este principio no parece que se pueda cumplir siempre, principalmente si no sabemos qué tipo de funcionalidades le vamos a agregar a nuestro software en el futuro.

3. Responsabilidad de sustitución de Liskov (LSR)

Las subclases deben ser sustituibles por sus clases base. Esto es, un usuario de una clase base debe continuar funcionando apropiadamente si una derivada de esa clase base se le pasa como argumento.

Ejemplo. Suponga que se tiene una clase `Elipse` y se define una función `foo(Elipse e)` (por el momento no importa lo que haga foo). Luego, se tiene una clase `Circulo` la cual es subclase de la clase `Elipse`. Entonces la función `foo` debería funcionar correctamente si se le pasa como argumento un objeto de la clase `Circulo`.

Conclusiones. Evidentemente no siempre se van a poder crear subclases que no violen este principio. Por poner un ejemplo en Java, se puede crear una clase `Cola` (como estructura de datos) que implemente la interfaz `Collection`. Cualquier clase que implemente `Collection` debe devolver un iterador (con el método `iterator()`). Suponga que se implementa una función `elimina(Collection c, int index)`, la cual elimina un elemento en la posición `index`, usando el método `remove()` del iterador. Uno esperaría que la función `elimina` funcione correctamente con una cola, sin embargo, en una cola no se puede eliminar en cualquier posición, sino sólo al final. En este caso se está violando el principio *LSR*. Para evitar este tipo de problemas se podrían hacer comprobaciones para que que sólo las subclases que cumplan con el contrato de la superclase puedan ser ejecutados por ciertas funciones, en otro caso se podría enviar un mensaje de error.

4. Principio de segregación de interfaz (ISP)

Muchas interfaces específicas de cliente son mejor que una interfaz de propósito general.

La escencia del principio es simple. Si tienes una clase que tiene varios clientes, en vez de cargar la clase con todos los métodos que el cliente necesita, crea interfaces específicas para cada cliente y herédalas en la clase. No se crea una interfaz para cada cliente, sino para cada *tipo* de cliente.

Ejemplo. Suponga que se tiene una clase `Cajero` que le permite a un cliente realizar las operaciones de consulta de saldo, depósito de efectivo y retiro de efectivo. Entonces se deberá crear una interfaz `Consulta`, una `Deposito` y otra `Retiro`, cada una definiendo los métodos adecuados, y la clase `Cajero` debe implementar estas interfaces.

Conclusiones. Este principio permite que las operaciones realizadas por un cliente no afecten a las de otro cliente. Aunque si se tienen muchos tipos de cliente se podría terminar con una clase que implementa cientos de interfaces.

5. Principio de inversión de dependencia (DIP)

Depende de abstracciones. No dependas de concretos.

Es la estrategia de depender de interfaces o funciones abstractas y clases abstractas, más que depender de funciones y clases concretas.

Ejemplo. Imagine que se desea crear la estructura de datos *pila*. Debido a que no existe una única implementación de las pilas, lo más conveniente sería declarar las funciones en una interfaz `InterPila` (por ejemplo). En dicha interfaz se colocan los nombres de las funciones que debe tener cualquier pila, los comentarios sobre las precondiciones y postcondiciones de las funciones. Cuando se requiera crear una pila, se deberá crear una clase `PilaConcreta` que implemente dicha interfaz.

Conclusiones. Este principio es conveniente porque las clases concretas cambian con frecuencia, mientras que las clases abstractas e interfaces casi no cambian. Aunque, eventualmente tendremos que depender de las clases concretas, pues en última instancia los objetos se tienen que crear de clases concretas.

Referencias

[1] Martin, R. C., *Design principles and design patterns*, www.objectmentor.com, 2000.

[2] *https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design-es*