Universidad Andrés Bello®

Conectar · Innovar · Liderar

FUNDAMENTOS DE PROGRAMACIÓN EN JAVA

PRINCIPIOS BÁSICOS DE DISEÑO ORIENTADO A OBJETOS

Introducción a los principios SOLID

Los principios SOLID son un conjunto de principios que ayudarán a escribir Software de calidad en cualquier lenguaje de Programación Orientada a Objetos si se encuentran aplicados correctamente, Gracias a ellos, se creará código que será más fácil de leer, testear y mantener.

Los principios en los que se basa SOLID son los siguientes:

- Principio de Responsabilidad Única.
- Principio Open/Closed.
- Principio de Sustitución de Liskov.
- Principio de Segregación de Interfaces.
- Principio de Inversión de Dependencias.

Estos principios son la base de mucho trabajo de programadores avanzados en el desarrollo deSoftware, los cuales sirven de pilar para muchas arquitecturas para proveer flexibilidad, validar partes del código de forma independiente y hacer más sencillo reestructurar un código fuente, alterando su estructura interna sin cambiar su comportamiento externo.

Los conceptos de cohesión y acoplamiento

Cohesión

La cohesión se refiere al grado en que los elementos de un módulo permanecen juntos. Por lo tanto, la cohesión mide la fuerza de la relación entre las piezas de funcionalidad dentro de un módulo dado.

Podríamos definir la cohesión como lo estrecha que es la relación entre los componentes de algo. Si hablamos de clases, una clase tendrá una cohesión alta si sus métodos están relacionados entre sí, tienen una “temática” común, trabajan con tipos similares, etc. Si pasamosa componentes de mayor tamaño, como paquetes o librerías, tendríamos una cohesión alta cuando las clases que lo forman están muy relacionadas entre sí, con un objetivo claro y focalizado.

La cohesión es un tipo de medición ordinal y se describe generalmente como "cohesión alta" o "cohesión baja". Se prefieren los módulos con una alta cohesión debido a varios rasgosdeseables del software con los que se relaciona como la robustez, la fiabilidad, la reutilización yel grado de comprensión. Por otro lado, la baja cohesión se asocia con rasgos indeseables, talescomo ser difícil de mantener, probar, volver a utilizar o incluso entender.

En la programación orientada a objetos, si los métodos que sirven a una clase tienden a ser similares en muchos aspectos, entonces se dice que la clase tiene una alta cohesión. En un sistema altamente cohesivo, la legibilidad y reusabilidad del código es mayor, mientras que la complejidad se mantiene manejable.

Se considera una cohesión alta si:

- Las funcionalidades integradas en una clase tienen mucho en común.
- Los métodos realizan un pequeño número de actividades relacionadas.

Las ventajas de la cohesión alta son:

- Reducción de módulo de complejidad (tener un menor número de operaciones).
- Aumento del sistema de mantenimiento.
- Aumento de la reutilización de módulos.

Un código muy cohesionado suele ser más fácil de entender como un todo porque tiene menos dependencias externas, tiende a ser más autocontenido. Se puede tratar como una caja negra que encapsula de forma férrea toda la lógica e información que contiene y eso hace que sea mássencillo olvidarse de su implementación interna.

Además, si se aumenta la cohesión se puede tener menos componentes, lo que ayuda a descubrir las funcionalidades del sistema y tener más claro dónde asignar cada responsabilidad.

Acoplamiento

El acoplamiento es la manera en que se relacionan varios componentes entre ellos. Si existen muchas relaciones entre los componentes, con muchas dependencias entre ellos, tendremos un grado de acoplamiento alto. Si los componentes son independientes unos de otros, el acoplamiento será bajo. Al igual que con la cohesión, se puede ver el acoplamiento a distintos niveles y existe acoplamiento entre los métodos de una misma clase, entre distintas clases o entre distintos paquetes.

Al favorecer el bajo acoplamiento lo normal es que tengamos componentes más pequeños, con responsabilidad más definida y, por tanto más fáciles de entender por separado. Estos componentes serán más reutilizables, precisamente porque al ser independientes unos de otros e incluir menos funcionalidades habrá más escenarios en los que tengan cabida.

Un riesgo asociado a un diseño en el que todos los componentes están muy desacoplados unos de otros son que al tender a ser componentes más pequeños y reutilizables, se acaban re utilizando en muchos contextos, por lo que aquello que inicialmente era una ventaja porque nospermitía desacoplar aspectos dentro de una funcionalidad, acaba por acoplar unas funcionalidades con otras a través del uso de componentes comunes.

El bajo acoplamiento permite:

- Minimiza el riesgo de tener que cambiar múltiples unidades de Software cuando se debealterar una.
- Mejorar la mantenibilidad de las unidades de Software.
- Evitar que un defecto en una unidad puede propagarse a otras, haciendo incluso másdifícil de detectar dónde está el problema.
- Aumentar la reutilización de las unidades de Software.

Principio de Responsabilidad Única

El principio de Responsabilidad Única señala que un objeto debe realizar una única cosa, es muy habitual que esto no se cumpla, y se acabe teniendo clases que tienen varias responsabilidadeslógicas a la vez.

Para detectar si se está violando este principio basta con detectar situaciones en las que una clase podría dividirse en varias. Por ejemplo, si una clase hace muchas cosas, lo más probablees que tenga muchos métodos públicos, y que tengan poco que ver entre ellos.

Otra forma de detectar el incumplimiento de este principio es por el número de imports, ya que si se necesitan importar demasiadas clases para hacer un trabajo, es posible que se esté realizando trabajo de más. También ayuda fijarse a qué paquetes pertenecen esos imports. Si se logra apreciar que pueden ser agrupados con facilidad, puede que esté avisando de que se están haciendo cosas muy diferentes.

Cada vez que se escribe una nueva funcionalidad y esa clase se ve afectada es porque estáinvolucrada en demasiadas cosas.

El Principio de Responsabilidad Única es una herramienta indispensable para proteger el códigofrente a cambios, ya que implica que sólo debería haber un motivo por el que modificar una clase.

Principio de Abierto-Cerrado

Este principio hace referencia a que una entidad de Software debería estar abierta a extensión pero cerrada a modificación, esto significa que se debe extender el comportamiento de las clasessin necesidad de modificar su código. Esto ayuda a seguir añadiendo funcionalidades sin afectarel código existente. Nuevas funcionalidades implicarán añadir nuevas clases y métodos, pero engeneral no debería suponer modificar lo que ya ha sido escrito.

Una de las formas más sencillas para detectarlo es darse cuenta de qué clases se modifican a menudo. Si cada vez que hay un nuevo requisito o una modificación de los existentes, las mismasclases se ven afectadas, podemos empezar a entender que estamos violando este principio.

Un ejemplo de aplicación del principio Abierto/Cerrado en código Java sería lo siguiente:

Se cuenta con una clase (Vehiculo) con un método que se encarga de imprimir un vehículo porpantalla. Por supuesto, cada vehículo tiene su propia forma de ser pintado:

```
public class Vehiculo{
    public TipoVehiculo getTipo() {
    }
}
```

Básicamente es una clase que especifica su tipo mediante un enumerado. Podemos tener por ejemplo un enum con un par de tipos:

```
public  enum  TipoVehiculo  {
    AUTO,
    MOTO
}
```

El método de la clase que se encarga de pintar los vehículos es el siguiente:

```
public  void  Imprmir(Vehiculo  veh)  {
    switch (veh.getTipo()) {
        case AUTO:
            ImprimirAuto(veh); break;
        case MOTO:
            Imprimir(veh);
            break;
    }
}
```

Mientras no se necesiten dibujar más tipos de vehículos ni se vea que este switch se repiteen varias partes del código, no se debe modificar. Incluso el hecho de que cambie la forma de dibujar un auto o una moto estaría encapsulado en sus propios métodos y no afectaría al resto del código. Sin embargo, se puede llegar al punto en el que se necesite dibujar uno o más tiposde vehículos, lo que implicaría crear un nuevo enumerado, un nuevo case y un nuevo método para implementar el dibujado del vehículo. En este caso sería buena idea aplicar el principio Abierto/Cerrado.

Una solución a esto mediante herencia y polimorfismo, es sustituir ese enumerado por clasesreales, y que cada clase sepa cómo pintarse:

```
public abstract class Vehiculo {
    public abstract void Imprimir();
}
```

```
public class Auto extends Vehiculo
{
@Override public void Imprimir() {
      }
  }
```

```
public class Moto extends Vehiculo
{
       @Override public void Imprimir() {
        }
  }
```

El método anterior quedaría reducido a esto:

```
public void Imprimir(Vehiculo veh) {
    veh.Imprimir();
}
```

Siguiendo estas modificaciones, añadir nuevos vehículos es tan sencillo como crear la clasecorrespondiente que extienda de Vehiculo:

```
public class Camion extends Vehiculo {
  @Override public void Imprimir() {
   }
}
```

Este principio es casi imposible de llevar a cabo en un 100% y puede hacer que sea ilegible e incluso más difícil de mantener. Este principio, al igual que el resto de las reglas SOLID se deben aplicar cuando corresponda y sin obsesionarse con cumplirlas en cada punto del desarrollo. En la mayoría de los casos es más sencillo limitarse a usarlas cuando surjan necesidades reales.

Principio de Sustitución de Liskov

El principio de sustitución de Liskov menciona que si en alguna parte del código se está usandouna clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas yque el programa siga siendo válido. Esto obliga a asegurarse de que cuando se extiende una clase no se está alterando el comportamiento de la clase padre.

En lenguaje más formal: si S es un subtipo de T, entonces los objetos de tipo T en un programade computadora pueden ser sustituidos por objetos de tipo S (es decir, los objetos de tipo S pueden sustituir objetos de tipo T), sin alterar ninguna de las propiedades deseables de ese programa (la corrección, la tarea que realiza, etc.).

Algunos beneficios del principio de Sustitución de Liskov son los siguientes:

- El código es más reutilizable.
- Facilidad de entendimiento de las jerarquías de clase.
- Validar que las abstracciones están correctas.

Es muy común basar los diseños en las propiedades de las clases del objeto que representan elmundo real, lo que puede llevar a cometer errores, sin embargo, si se basan en las conductas delos objetos, es posible evitar estos errores.

Principio de Segregación de Interfaces

El principio de segregación de interfaces viene a decir que ninguna clase debería depender de métodos que no usa. Por lo tanto, cuando se crean interfaces que definan comportamientos, esimportante asegurarse de que todas las clases que implementen esas interfaces vayan a necesitar y ser capaces de agregar comportamientos a todos los métodos. En caso contrario, esmejor tener varias interfaces más pequeñas.

Las interfaces ayudan a desacoplar módulos entre sí. Esto debido a que si se tiene una interfazque explica el comportamiento que el módulo espera para comunicarse con otros módulos, siempre se podrá crear una clase que lo implemente de modo que cumpla las condiciones. El módulo que describe la interfaz no tiene que saber nada sobre el código y, sin embargo, se podrá trabajar con él sin problemas.

La problemática surge cuando esas interfaces intentan definir más cosas de las debidas, por ejemplo, que las clases hijas no usen muchos de esos métodos, y habrá que entregarles una implementación. Es muy habitual lanzar una excepción o simplemente no hacer nada al respecto.

Si se lanza una excepción, es más que probable que el módulo que define esa interfaz use el método en algún momento, y esto hará fallar el programa. El resto de implementaciones que sepuedan dar generan efectos secundarios no esperados, y a los que sólo se podrá responder conociendo el código fuente del módulo en cuestión.

Si al implementar una interfaz se logra apreciar que uno o varios de los métodos no tienen sentido y pareciera hacer falta dejarlos vacíos o lanzar excepciones, es muy probable que se esté violando este principio. Si la interfaz forma parte del código, se debe dividir en varias interfaces que definan comportamientos más específicos.

El principio de segregación de interfaces ayuda a no obligar a ninguna clase a implementarmétodos que no utiliza. Esto evitará problemas que puedan llevar a errores inesperados y adependencias no deseadas

Anexos y Referencias:

1. **Principios-solid**
   https://enmilocalfunciona.io/principios-solid/

2. **¿Qué son los Principios SOLID?**
   https://devexperto.com/principios-solid/