COHESIÓN Y ACOPLAMIENTO

Guía utilizada en la clase virtual del 4 de mayo

Tanto la cohesión como el acoplamiento son conceptos importantes en la ingeniería de software y se relacionan con la calidad y la mantenibilidad del código.

La cohesión se refiere al grado en que los elementos de un módulo o componente de software están relacionados entre sí y trabajan juntos para lograr un objetivo común. Una alta cohesión significa que los elementos de un módulo están estrechamente relacionados y trabajan juntos de manera efectiva para realizar una tarea específica. En otras palabras, una alta cohesión indica que un módulo realiza una única tarea de manera efectiva y eficiente.

Por otro lado, el acoplamiento se refiere al grado en que los diferentes módulos o componentes de un sistema dependen entre sí. Un bajo acoplamiento significa que los módulos son independientes entre sí y se pueden cambiar o **reutilizar** sin afectar a otros módulos del sistema. En cambio, un alto acoplamiento significa que los módulos están fuertemente interconectados y que **los cambios en un módulo pueden afectar a otros módulos del sistema** (de alta cohesión).

En general, se considera que una buena arquitectura de software debe tener una alta cohesión y un bajo acoplamiento. Una alta cohesión asegura que los módulos o componentes de software sean fáciles de entender y mantener, mientras que un bajo acoplamiento facilita la flexibilidad y la reutilización del código.

Es importante destacar que tanto la cohesión como el acoplamiento son aspectos importantes a considerar al diseñar software. Un diseño de software bien estructurado debe buscar maximizar la cohesión y minimizar el acoplamiento, para lograr un sistema eficiente, flexible y fácil de mantener.

Es importante tener en cuenta que

- ✓ **un alto acoplamiento puede dificultar la modificación, la reutilización y el mantenimiento del código, mientras que**

- ✓ **un bajo acoplamiento puede mejorar la flexibilidad y la escalabilidad del sistema. Por lo tanto, es importante buscar un bajo acoplamiento al diseñar nuestros proyectos.**

Ejemplo de alta cohesión:

Supongamos que estamos diseñando una clase para trabajar con archivos de texto. En este caso, una buena práctica sería incluir solo métodos que sean relevantes para trabajar con archivos de texto, como por ejemplo leer, escribir y modificar archivos. Todos estos métodos estarían estrechamente relacionados con la tarea principal de la clase y, por lo tanto, tendrían una alta cohesión.

```
public class ArchivoTexto {

  public void leerArchivo(String nombreArchivo) {
    // Código para leer archivo
  }

  public void escribirArchivo(String nombreArchivo, String texto) {
    // Código para escribir en archivo
  }

  public void modificarArchivo(String nombreArchivo, String
nuevoTexto) {
    // Código para modificar archivo
  }
}
```

En este ejemplo, todos los métodos de la clase están relacionados con la tarea principal de trabajar con archivos de texto. Esto significa que la clase tiene una alta cohesión.

Ejemplo de baja cohesión:

Ahora supongamos que tenemos una clase que realiza tareas muy diversas y no relacionadas entre sí. Por ejemplo, una clase que maneja operaciones matemáticas y también tiene métodos para enviar correos electrónicos.

```
public class Utilidades {
  public void sumar(int a, int b) {
    // Código para sumar dos números
  }

  public void restar(int a, int b) {
    // Código para restar dos números
  }

  public void enviarCorreo(String destinatario, String mensaje) {
    // Código para enviar un correo electrónico
```

```
}
```

En este ejemplo, la clase `Utilidades` tiene métodos que realizan tareas muy diversas y no están relacionadas entre sí. La tarea de sumar y restar números no tiene nada que ver con el envío de correos electrónicos, lo que significa que la clase tiene una baja cohesión.

Es importante tener en cuenta que **una clase con baja cohesión puede ser difícil de entender y mantener**, ya que no está claro qué hace cada método y cómo se relacionan entre sí. Por lo tanto, es importante buscar una alta cohesión al diseñar clases y componentes de software.

Ejemplo de alto acoplamiento:

Supongamos que estamos diseñando una aplicación que tiene una clase **`Calculadora`** que realiza operaciones matemáticas y una clase **`BaseDatos`** que maneja la conexión y el acceso a una base de datos. Si la clase **`Calculadora`** necesita acceder a la base de datos para realizar algunas operaciones, podría hacerlo directamente llamando a los métodos de la clase **`BaseDatos`**. Esto creará un alto acoplamiento entre las dos clases, lo que significa que cualquier cambio en la clase **`BaseDatos`** podría afectar la clase **`Calculadora`**.

```
public class Calculadora {
  private BaseDatos baseDatos;

  public Calculadora() {
    baseDatos = new BaseDatos();
  }

  public int sumar(int a, int b) {
    // Realizar la suma
    int resultado = a + b;

    // Guardar el resultado en la base de datos
    baseDatos.guardar(resultado);

    return resultado;
  }
}

public class BaseDatos {
  public void guardar(int resultado) {
    // Código para guardar el resultado en la base de datos
  }
}
```

En este ejemplo, la clase `Calculadora` está acoplada directamente a la clase `BaseDatos`, lo que significa que cualquier cambio en la clase `BaseDatos` podría afectar la clase `Calculadora`. Esto crea un alto acoplamiento entre las dos clases.

Ejemplo de bajo acoplamiento:

Supongamos que tenemos dos clases: **`Calculadora`** e **`Impresora`**. La clase **`Calculadora`** realiza cálculos matemáticos y la clase **`Impresora`** imprime los resultados de estos cálculos. Una forma de hacer esto es que la clase **`Calculadora`** tenga una referencia a la clase **`Impresora`** y le pase el resultado para que lo imprima. Este enfoque puede generar un acoplamiento fuerte entre ambas clases, ya que la clase **`Calculadora`** depende directamente de la clase **`Impresora`**.

Un enfoque con bajo acoplamiento sería separar la funcionalidad de imprimir en una clase separada, por ejemplo, una clase **`ResultadoImpresora`**. La clase **`Calculadora`** podría simplemente devolver el resultado de sus cálculos y la clase **`ResultadoImpresora`** se encargaría de imprimir el resultado. De esta forma, la clase **`Calculadora`** no tiene que conocer la implementación de la clase **`Impresora`**, reduciendo así el acoplamiento entre ambas clases.

Por ejemplo, este puede ser el código;

```
public class Calculadora {

    public int sumar(int a, int b) {
        return a + b;
    }

    public int restar(int a, int b) {
        return a - b;
    }

    // Método que devuelve un objeto ResultadoImpresora
    public ResultadoImpresora imprimir(int resultado) {
        return new ResultadoImpresora(resultado);
    }
}


public class ResultadoImpresora {
         private int resultado;

         public ResultadoImpresora(int resultado) {
             this.resultado = resultado;
         }

         public void imprimir() {
```

```
                System.out.println("El resultado es: " +
resultado);
            }
}
```

Ahora supongamos que tenemos una aplicación que tiene una clase `Cliente` que interactúa con una clase `Factura`. Si la clase `Cliente` necesita interactuar con la clase `Factura`, podría hacerlo a través de una **interfaz** `FacturaInt`, que define los métodos que la clase `Cliente` necesita para interactuar con la clase `Factura`. La clase `Factura` implementaría la interfaz `FacturaInt`, pero la clase `Cliente` no tendría que conocer los detalles de implementación de la clase `Factura`. Esto crea un bajo acoplamiento entre las dos clases.

```
public interface FacturaInt {
  void crearFactura(Factura factura);
  Factura obtenerFactura(int id);
}


public class Cliente {
  private FacturaInt facturaInt;

  public Cliente(FacturaInt facturaInt) {
    this.facturaInt = facturaInt;
  }

  public void pagarFactura(int idFactura) {
    Factura factura = facturaInt.obtenerFactura(idFactura);
    // Código para realizar el pago de la factura
  }
}

public class Factura implements FacturaInt {
  public void crearFactura(Factura factura) {
    // Código para crear una factura
  }

  public Factura obtenerFactura(int id) {
    // Código para obtener una factura por ID
    return null;
   }
}
```

En este ejemplo, la clase `Cliente` interactúa con la clase `Factura` a través de la interfaz `FacturaInt`, lo que significa que la clase `Cliente` no tiene que conocer los detalles de implementación de la clase `Factura`. Esto crea un bajo acoplamiento entre las dos clases.