Chain of Responsibility Pattern

Introducción y nombre

Chain of Responsibility. De comportamiento. El patrón de diseño Chain of Responsibility permite establecer una cadena de objetos receptores a través de los cuales se pasa una petición formulada por un objeto emisor.

Intención

Como se dijo anteriormente se busca establecer una cadena de receptores, donde cualquiera de ellos puede responder a la petición en función de un criterio establecido. Busca evitar un montón de if – else largos y complejos en nuestro código, pero sobre todas las cosas busca evitar que el cliente necesite conocer toda nuestra estructura jerárquica y que rol cumple cada integrante de nuestra estructura.

También conocido como

Cadena de responsabilidad.

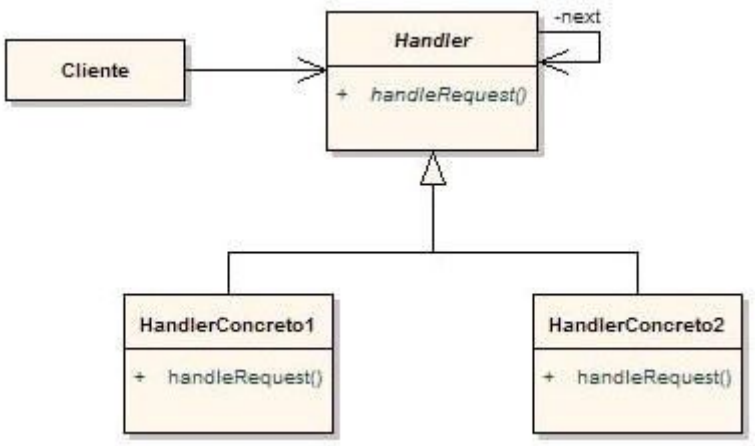
Motivación

En múltiples ocasiones, un cliente necesita que se realice una función, pero o no conoce al servidor concreto de esa función o es conveniente que no lo conozca para evitar un gran acoplamiento entre ambos.
Encadena los objetos receptores y pasa la petición a través de la cadena hasta que es procesada por algún objeto. Este evita que el cliente deba conocer toda nuestra estructura de clases, ya que cualquiera le resuelve el problema.

Solución

Se utiliza cuando:
Las peticiones emitidas por un objeto deben ser atendidas por distintos objetos receptores.
No se sabe a priori cual es el objeto que me puede resolver el problema.
Cuando un pedido debe ser manejado por varios objetos.
El conjunto de objetos que pueden tratar una petición debería ser especificado dinámicamente.

Diagrama UML



Participantes

Handler: define una interfaz para tratar las peticiones. Implementa el enlace al sucesor.

HandlerConcreto: trata las peticiones de las que es responsable. Si puede manejar la petición, lo hace, en caso contrario la reenvía a su sucesor.

Cliente: inicializa la petición.

Colaboraciones

El cliente conoce a un gestor que es el que lanza la petición a la cadena hasta que alguien la recoge.

Consecuencias

Reduce el acoplamiento.

Añade flexibilidad para asignar responsabilidades a objetos.

No se garantiza la recepción.

Implementación

Todos los objetos receptores implementarán la misma interfaz o extenderán la misma clase abstracta. En ambos casos se proveerá de un método que permita obtener el sucesor y así el paso de la petición por la cadena será lo más flexible y transparente posible.

La idea es crear un sistema que pueda servir a diversas solicitudes de manera jerárquica.

Código de muestra

Escenario: estamos realizando el software para un banco y uno de los puntos más importantes es saber quién puede aprobar un crédito. Por lo tanto el banco define las siguientes reglas de negocio:

Si el monto no supera los $ 10.000 entonces el ejecutivo de cuenta pueda aprobar el préstamo.

Si el monto esta entre los $10.000 y $50.000 entonces la persona indicada para realizar la aprobación es el líder inmediato de dicho ejecutivo.

Si el monto se encuentra entre $ 50.000 y $100.000 entonces es el Gerente quién debe realizar dicha aprobación.

Por montos superiores a los $100.000 entonces la aprobación la realizará el Director.

Para este caso se ha decidido realizar un patrón Chain of Responsibility. Se decide crear una interface llamada IAprobador que debe implmentar toda clase que pertenezca a nuestra cadena de responsabilidades.

[code]
```
public interface IAprobador {
   public void setNext(IAprobador aprobador);
   public IAprobador getNext();
   public void solicitudPrestamo(int monto);
}
```

Ahora veamos las clases que son aprobadoras:

```
public class EjecutivoDeCuenta implements IAprobador {

   private IAprobador next;

   @Override
   public IAprobador getNext() {
      return next;
   }

   @Override
   public void solicitudPrestamo(int monto) {
      if (monto <= 10000) {
         System.out.println("Lo manejo yo, el ejecutivo de cuentas");
      } else {
         next.solicitudPrestamo(monto);
      }
   }

   @Override
   public void setNext(IAprobador aprobador) {
      next = aprobador;
   }
}
public class LiderTeamEjecutivo implements IAprobador {

   private IAprobador next;

   @Override
   public IAprobador getNext() {
      return next;
   }

   @Override
   public void solicitudPrestamo(int monto) {
      if (monto > 10000 && monto <= 50000) {
         System.out.println("Lo manejo yo, el lider");
      } else {
         next.solicitudPrestamo(monto);
      }
   }

   @Override
   public void setNext(IAprobador aprobador) {
      next = aprobador;
   }
}

public class Gerente implements IAprobador {

   private IAprobador next;
```

```
    @Override
    public IAprobador getNext() {
        return next;
    }

    @Override
    public void solicitudPrestamo(int monto) {
        if (monto > 50000 && monto <= 100000) {
            System.out.println("Lo manejo yo, el gerente");
        } else {
            next.solicitudPrestamo(monto);
        }
    }

    @Override
    public void setNext(IAprobador aprobador) {
        next = aprobador;
    }
}

public class Director implements IAprobador {

    private IAprobador next;

    @Override
    public IAprobador getNext() {
        return next;
    }

    @Override
    public void solicitudPrestamo(int monto) {
        if (monto >= 100000) {
            System.out.println("Lo manejo yo, el director");
        }
    }

    @Override
    public void setNext(IAprobador aprobador) {
        next = aprobador;
    }
}
```

Y, por último, el banco, que a fin de cuentas es quién decide las reglas del negocio.

```
public class Banco implements IAprobador {

    private IAprobador next;

    @Override
    public IAprobador getNext() {
        return next;
    }

    @Override
    public void solicitudPrestamo(int monto) {
        EjecutivoDeCuenta ejecutivo = new EjecutivoDeCuenta();
        this.setNext(ejecutivo);

        LiderTeamEjecutivo lider = new LiderTeamEjecutivo();
        ejecutivo.setNext(lider);

        Gerente gerente = new Gerente();
        lider.setNext(gerente);

        Director director = new Director();
        gerente.setNext(director);
```

```
        next.solicitudPrestamo(monto);
    }

    @Override
    public void setNext(IAprobador aprobador) {
        next = aprobador;
    }
}
```

Veamos como es su funcionamiento:

```
public class Main {
    public static void main(String[] args) {
        Banco banco = new Banco();
        banco.solicitudPrestamo(56000);
    }
}
```

Y el resultado por consola es: Lo manejo yo, el gerente

[/code]

Cuando utilizarlo

La motivación detrás de este patrón es crear un sistema que pueda servir a diversas solicitudes de manera jerárquica. En otras palabras, si un objeto que es parte de un sistema no sabe cómo responder a una solicitud, la pasa a lo largo del árbol de objetos. Como el nombre lo implica, cada objeto de dicho árbol puede tomar la responsabilidad y atender la solicitud.

Un ejemplo típico podría ser el lanzar un trabajo de impresión. El cliente no sabe siquiera qué impresoras están instaladas en el sistema, símplemente lanza el trabajo a la cadena de objetos que representan a las impresoras. Cada uno de ellos lo deja pasar, hasta que alguno, finalmente lo ejecuta.

Hay un desacoplamiento evidente entre el objeto que lanza el trabajo (el cliente) y el que lo realiza (impresora).