Universidad Tecnológica de Panamá

Campus Central Dr. Víctor Levi Sasso

Facultad de Ingeniería de Sistemas Computacionales

Licenciatura en Desarrollo de Software

Integrantes del grupo:

Giovanni Navarro 8-1001-42

Viviam Ortega 8-1004-740

Grupo:

1LS242

Asignatura:

Desarrollo de Software IX

Presentación 1: Patrón cadena de Responsabilidad y Patrón cadena de comando

Profesor:

Erick Agrazal

18/04/2025

Patrón Cadena de Responsabilidad

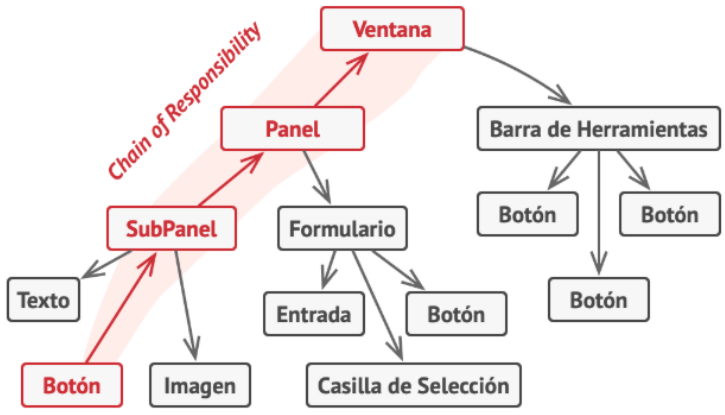
Definición

El patrón de cadena de responsabilidad es un patrón de comportamientos que permite pasar solicitudes a lo largo de cadena de manejadores. Al recibir una solicitud cada manejador determina si la procesa o la pasa al siguiente manejador de la cadena y así sucederá sucesivamente hasta que un manejador decida procesarla.

Aplicabilidad

Utilizaremos el Patrón Cadena de Responsabilidad cuando:

- El patrón Cadena de Responsabilidad es útil cuando tu programa necesita procesar distintos tipos de solicitudes de diversas maneras, especialmente cuando no se conocen de antemano los tipos exactos de solicitudes ni sus secuencias. Este patrón permite encadenar varios manejadores para que, al recibir una solicitud, cada manejador pueda decidir si puede procesarla o no, dando a todos los manejadores la oportunidad de gestionarla.
- Cuando sea esencial ejecutar varios manejadores en un orden específico. Dado que los manejadores de la cadena pueden vincularse en cualquier orden, todas las solicitudes seguirán la secuencia que se haya planificado exactamente como se diseñe.
- Cuando sea necesario que el grupo de manejadores y su orden puedan cambiar durante el tiempo de ejecución. Si proporcionas modificadores (setters) para un campo de referencia dentro de las clases manejadoras, será posible insertar, eliminar o reordenar los manejadores de manera dinámica.



Cómo implementarlo

1. Declara la interfaz manejadora y describe la firma de un método para manejar solicitudes.
2. Para eliminar código boilerplate duplicado en manejadores concretos, puede merecer la pena crear una clase manejadora abstracta base, derivada de la interfaz manejadora.
3. Una a una, crea subclases manejadoras concretas e implementa los métodos de control. Cada manejador debe tomar dos decisiones cuando recibe una solicitud:
    a. Si procesa la solicitud.
    b. Si pasa la solicitud al siguiente eslabón de la cadena.
4. El cliente puede ensamblar cadenas por su cuenta o recibir cadenas prefabricadas de otros objetos. En el último caso, debes implementar algunas clases fábrica para crear cadenas de acuerdo con los ajustes de configuración o de entorno.
5. El cliente puede activar cualquier manejador de la cadena, no solo el primero. La solicitud se pasará a lo largo de la cadena hasta que algún manejador se rehúse a pasarlo o hasta que llegue al final de la cadena.
6. Debido a la naturaleza dinámica de la cadena, el cliente debe estar listo para gestionar los siguientes escenarios:
    a. La cadena puede consistir en un único vínculo.
    b. Algunas solicitudes pueden no llegar al final de la cadena.
    c. Otras pueden llegar hasta el final de la cadena sin ser gestionadas.

Ventajas

- La función principal del patrón de cadena de responsabilidad es: combinación dinámica, desacoplamiento de solicitante y receptor.
- El solicitante y el receptor están débilmente acoplados: el solicitante no necesita conocer al receptor ni cómo tratar con él. Cada objeto de responsabilidad solo es responsable de su propio ámbito de responsabilidad, y el resto se entrega a los sucesores. Los componentes están completamente desacoplados.
- Combinación dinámica de responsabilidades: el modelo de cadena de responsabilidad dispersará las funciones en objetos de responsabilidad separados y luego las combinará dinámicamente para formar una cadena cuando se use, de modo que los objetos de responsabilidad se puedan asignar de manera flexible y las responsabilidades se puedan agregar y cambiar de manera flexible.

Desventajas

- No hay garantía de que se procesen todas las solicitudes. Dado que una solicitud no tiene un destinatario claro, no hay garantía de que se procese y es posible que la solicitud no se procese hasta que llegue al final de la cadena.
- En comparación con una cadena de responsabilidad más larga, el procesamiento de solicitudes puede involucrar múltiples objetos de procesamiento y el rendimiento del sistema se verá afectado en cierta medida.
- La racionalidad del establecimiento de la cadena de responsabilidad debe ser garantizada por el cliente, lo que aumenta la complejidad del cliente, y puede provocar errores en el sistema por la configuración incorrecta de la cadena de responsabilidad, como las llamadas de bucle.

Ejemplo en Código

Este ejemplo muestra cómo los manejadores trabajan en equipo para procesar solicitudes de manera flexible y dinámica, asegurándose de que cada solicitud sea atendida por el manejador adecuado, si existe uno.

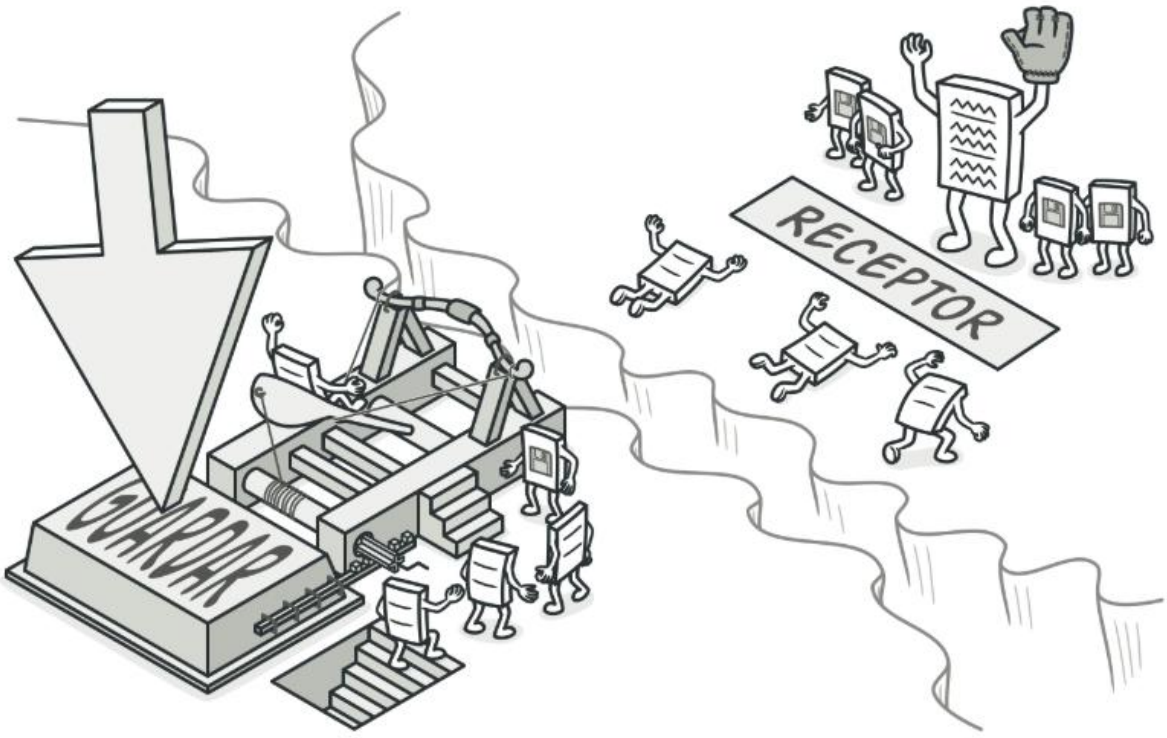
```
// Interfaz manejadora
interface Manejador {
    void setSiguiente(Manejador manejador);
    void manejarSolicitud(String solicitud);
}

// Clase base abstracta
abstract class ManejadorBase implements Manejador {
    protected Manejador siguiente;
    @Override
    public void setSiguiente(Manejador manejador) {
        this.siguiente = manejador;
    }

    @Override
    public void manejarSolicitud(String solicitud) {
        if (siguiente != null) {
            siguiente.manejarSolicitud(solicitud);
        }
    }
}

// Manejadores concretos
class ManejadorA extends ManejadorBase {
    @Override
    public void manejarSolicitud(String solicitud) {
        if (solicitud.equals("A")) {
            System.out.println("ManejadorA procesó la solicitud.");
        } else {
            super.manejarSolicitud(solicitud);
        }
    }
}
```

```java
class ManejadorB extends ManejadorBase {
    @Override
    public void manejarSolicitud(String solicitud) {
        if (solicitud.equals("B")) {
            System.out.println("ManejadorB procesó la solicitud.");
        } else {
            super.manejarSolicitud(solicitud);
        }
    }
}

// Cliente
public class CadenaDeResponsabilidad {
    public static void main(String[] args) {
        Manejador manejadorA = new ManejadorA();
        Manejador manejadorB = new ManejadorB();

        manejadorA.setSiguiente(manejadorB);
        manejadorA.manejarSolicitud("A");
        manejadorA.manejarSolicitud("B");
        manejadorA.manejarSolicitud("C");
    }
}
```

Patrón Cadena de Comando

Definición

El patrón cadena de comando es un patrón de diseño de comportamiento que convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar.



Aplicabilidad

Esta cadena se utiliza de las siguientes formas:

- El patrón resulta útil cuando necesitas parametrizar objetos con operaciones, ya que convierte una llamada a un método específico en un objeto autónomo. Esta transformación permite realizar diversas acciones, como pasar comandos como argumentos, almacenarlos en otros objetos o modificar comandos vinculados durante el tiempo de ejecución.

- Cuando necesites poner operaciones en cola, programar su ejecución o ejecutarlas de forma remota. Los comandos, al igual que cualquier otro objeto, pueden serializarse, lo que significa que pueden convertirse en cadenas para almacenarse en archivos o bases de datos. Posteriormente, estas cadenas pueden restaurarse como los objetos de comando originales, permitiendo retrasar o programar su ejecución. Además, este patrón facilita la posibilidad de poner comandos en cola, registrarlos o incluso enviarlos a través de la red.

- Cuando necesites implementar operaciones reversibles, ya que es una de las formas más populares de manejar deshacer/rehacer. Para revertir operaciones, es necesario implementar un historial de las operaciones realizadas, que consiste en una pila con los objetos de comando ejecutados y copias de seguridad del estado de la aplicación. Sin embargo, este enfoque tiene dos desventajas: guardar el estado de una aplicación puede ser complicado debido a la privacidad de algunos datos, aunque este problema puede mitigarse con el patrón Memento; además, las copias de seguridad pueden consumir mucha memoria RAM. Como alternativa, en lugar de restaurar el estado pasado, el comando puede realizar la operación inversa, aunque esta solución puede ser difícil o incluso imposible de implementar en algunos casos.

Cómo implementarlo

1. Declara la interfaz de comando con un único método de ejecución.
2. Empieza extrayendo solicitudes y poniéndolas dentro de clases concretas de comando que implementen la interfaz de comando. Cada clase debe contar con un grupo de campos para almacenar los argumentos de las solicitudes junto con referencias al objeto receptor. Todos estos valores deben inicializarse a través del constructor del comando.
3. Identifica clases que actúen como *emisoras*. Añade los campos para almacenar comandos dentro de estas clases. Las emisoras deberán comunicarse con sus comandos tan solo a través de la interfaz de comando. Normalmente las emisoras no crean objetos de comando por su cuenta, sino que los obtienen del código cliente.
4. Cambia las emisoras de forma que ejecuten el comando en lugar de enviar directamente una solicitud al receptor.
5. El cliente debe inicializar objetos en el siguiente orden:
    a. Crear receptores.
    b. Crear comandos y asociarlos con receptores si es necesario.
    c. Crear emisores y asociarlos con comandos específicos.

Ventajas

- Facilita la implementación de operaciones deshacer/rehacer.

- Promueve la extensibilidad y la flexibilidad del código.
- Puede implementar la ejecución diferida de operaciones (colas y registros de comandos).

Desventajas

- El código puede volverse más complicado ya que está introduciendo una capa completamente nueva entre remitentes y receptores.

Ejemplo de Código

Este ejemplo nos muestra las acciones de encender y apagar una luz en un objeto de comando.

```
// Interfaz Command
interface Command {
    void execute();
}


// Receptor
class Luz {
    void encender() {
        System.out.println("La luz está encendida.");
    }

    void apagar() {
        System.out.println("La luz está apagada.");
    }
}

// Comandos concretos
class EncenderLuzCommand implements Command {
    private Luz luz;
    EncenderLuzCommand(Luz luz) {
        this.luz = luz;
    }

    @Override
    public void execute() {
```

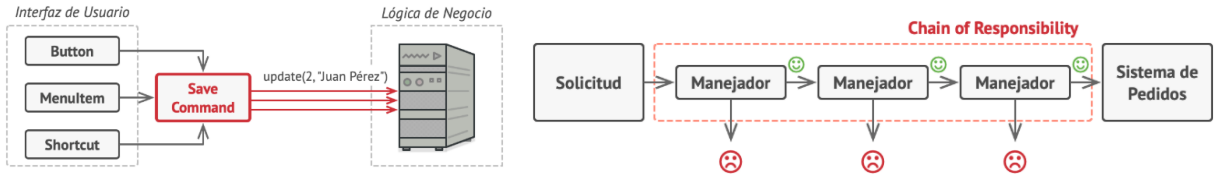
```
        luz.encender();
    }
}

class ApagarLuzCommand implements Command {
    private Luz luz;
    ApagarLuzCommand(Luz luz) {
        this.luz = luz;
    }

    @Override
    public void execute() {
        luz.apagar();
    }
}

// Invocador
class Interruptor {
    private Command command;
    void setCommand(Command command) {
        this.command = command;
    }

    void presionarBoton() {
        command.execute();
    }
}

// Cliente
public class PatronCommandEjemplo {
    public static void main(String[] args) {
        Luz luz = new Luz();
        Command encender = new EncenderLuzCommand(luz);
        Command apagar = new ApagarLuzCommand(luz);
        Interruptor interruptor = new Interruptor();

        interruptor.setCommand(encender);
        interruptor.presionarBoton();
        interruptor.setCommand(apagar);
```

```
        interruptor.presionarBoton();
    }
}
```

Diferencias entre cadena de responsabilidad y cadena de comando

La cadena de responsabilidad pasa una solicitud secuencialmente a lo largo de una cadena dinámica de receptores potenciales hasta que uno de ellos la gestiona. En comparación con la cadena de comando esta establece conexiones unidireccionales entre emisores y receptores.

Los manejadores del patrón de cadena de responsabilidad se pueden implementar como Comandos, Sin embargo, hay otra solución en la que la propia solicitud es un objeto Comando.



Conclusiones

Giovanni Navarro: El patrón Cadena de Responsabilidad y el patrón Comando ayudan a organizar mejor el código y hacerlo más flexible. La Cadena de Responsabilidad permite que varios objetos intenten manejar una solicitud sin depender entre ellos, mientras que el Comando encapsula acciones como objetos, facilitando su gestión, reutilización y posible deshacer.

Viviam Ortega: La Cadena de Responsabilidad permite distribuir las responsabilidades entre varios objetos, y el Comando facilita ejecutar acciones de forma controlada y flexible. Ambos mejoran la mantenibilidad del código al separar claramente quién solicita una acción y quién la ejecuta.

Bibliografía

Chain of Responsability. (s/f). Reactiveprogramming.io. Recuperado el 18 de abril de 2025, de https://reactiveprogramming.io/blog/es/patrones-de-diseno/chain-of-responsability

Chain of Responsibility. (s/f). Refactoring.guru. Recuperado el 18 de abril de 2025, de https://refactoring.guru/es/design-patterns/chain-of-responsibility

Chain of responsibility. (s/f). Patrones de Diseño Java. Recuperado el 18 de abril de 2025, de https://java-design-patterns.com/es/patterns/chain-of-responsibility/

(S/f). Devexpert.io. Recuperado el 18 de abril de 2025, de https://devexpert.io/chain-of-responsibility/

Patrón de cadena de responsabilidad (patrón de cadena de responsabilidad) - programador clic. (s. f.). https://programmerclick.com/article/13471013292/

Patrón Command | Diseño de software. (s. f.). https://sivanahamer.github.io/software-design/docs/patterns/comportamiento/command/

Command. (s. f.). https://refactoring.guru/es/design-patterns/command