**UNIVERSIDAD NACIONAL AGRARIA DE LA SELVA**

**FACULTAD DE INGENIERÍA EN INFORMÁTICA Y SISTEMAS**

**ESCUELA ACADÉMICA PROFESIONAL DE INGENIERÍA EN INFORMÁTICA Y SISTEMAS**

Diagrama

Descripción generada automáticamente

**TRABAJO ENCARGADO**

**IMPLEMENTACION DEL PATRÓN DE COMPORTAMIENTO CHAIN OF RESPONSIBILITY**

AUTORES : RIOS TELLO, AYRTON JESÚS

VALLES RODRÍGUEZ, SERGIO ALEXANDER

ASIGNATURA : DISEÑO DE SOFTWARE

DOCENTE : SOLIS BONIFACIO, HUBEL

**TINGO MARÍA - 2023**

**PERU**

**¿QUÉ ES?**

El patrón de comportamiento "Chain of Responsibility" (Cadena de Responsabilidad) es un patrón de diseño que se utiliza para lograr un desacoplamiento de objetos en un sistema. Permite a una serie de objetos llamados "manejadores" procesar una solicitud de forma secuencial hasta que uno de ellos sea capaz de manejarla.

En este patrón, cada objeto manejador tiene un enlace o referencia al siguiente manejador en la cadena. Cuando llega una solicitud, el primer manejador intenta manejarla. Si puede manejarla, lo hace y la cadena se detiene ahí. Si no puede manejarla, pasa la solicitud al siguiente manejador en la cadena y así sucesivamente, hasta que un manejador pueda manejarla o hasta que se llegue al final de la cadena sin que ningún manejador pueda manejarla.

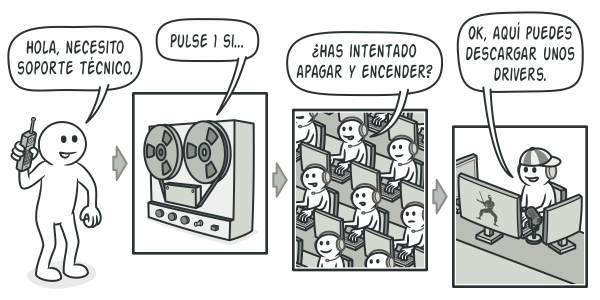
Este enfoque permite que la solicitud se propague a través de la cadena de objetos de manera transparente para el cliente que realiza la solicitud. Además, se puede agregar o modificar dinámicamente la estructura de la cadena sin afectar el cliente.

**¿CUÁL ES SU OBJETIVO?**

El objetivo principal del patrón de comportamiento Chain of Responsibility es lograr un desacoplamiento entre el emisor de una solicitud y los objetos que la manejan, permitiendo que múltiples objetos tengan la oportunidad de procesar dicha solicitud de forma secuencial.

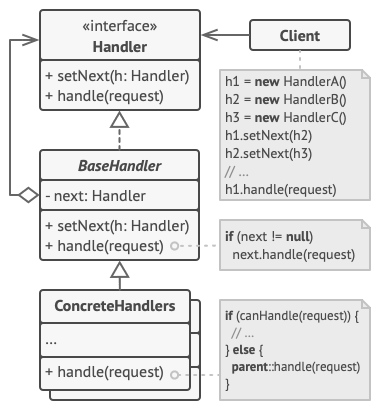
Los objetivos específicos de este patrón son:

* Desacoplamiento: El patrón Chain of Responsibility ayuda a evitar la dependencia directa entre el emisor de una solicitud y los objetos que la manejan. El emisor solo necesita conocer al primer objeto de la cadena, y los objetos subsiguientes están conectados en la cadena de forma transparente.
* Flexibilidad y extensibilidad: Debido a la naturaleza dinámica de la cadena, es posible agregar o quitar fácilmente nuevos objetos manejadores en cualquier punto de la cadena sin afectar al cliente. Esto permite una mayor flexibilidad y extensibilidad en el sistema, ya que se pueden introducir nuevos comportamientos sin modificar el código existente.
* Reutilización de código: Al utilizar el patrón Chain of Responsibility, se puede reutilizar el código de los objetos manejadores existentes para diferentes tipos de solicitudes. Cada objeto manejador se encarga de una responsabilidad específica, lo que facilita la reutilización de estos en diferentes contextos.
* Simplificación del código del cliente: El cliente que emite la solicitud no necesita conocer los detalles específicos de los objetos manejadores ni la estructura de la cadena. Esto simplifica el código del cliente y lo hace más limpio y legible.

**ANALOGÍA CON EL MUNDO REAL**

Una llamada a un servicio técnico es un gran ejemplo de como funciona el “cambio de responsabilidad”. Si la ayuda que necesitas no puede ser resuelta por un operador, este te deriva a un operador más especializado hasta poder llegar a una solución factible.

**Estructura**

1. La clase Manejadora declara la interfaz común a todos los manejadores concretos. Normalmente contiene un único método para manejar solicitudes, pero en ocasiones también puede contar con otro método para establecer el siguiente manejador de la cadena.
2. La clase Manejadora Base es opcional y es donde puedes colocar el código boilerplate (segmentos de código que suelen no alterarse) común para todas las clases manejadoras.

Normalmente, esta clase define un campo para almacenar una referencia al siguiente manejador. Los clientes pueden crear una cadena pasando un manejador al constructor o modificador (setter) del manejador previo. La clase también puede implementar el comportamiento de gestión por defecto: puede pasar la ejecución al siguiente manejador después de comprobar su existencia.

1. Los Manejadores Concretos contienen el código para procesar las solicitudes. Al recibir una solicitud, cada manejador debe decidir si procesarla y, además, si la pasa a lo largo de la cadena.

Habitualmente los manejadores son autónomos e inmutables, y aceptan toda la información necesaria únicamente a través del constructor.

1. El Cliente puede componer cadenas una sola vez o componerlas dinámicamente, dependiendo de la lógica de la aplicación. Observa que se puede enviar una solicitud a cualquier manejador de la cadena; no tiene por qué ser al primero.

**Aplicabilidad**

* Utiliza el patrón Chain of Responsibility cuando tu programa deba procesar distintos tipos de solicitudes de varias maneras, pero los tipos exactos de solicitudes y sus secuencias no se conozcan de antemano.
* El patrón te permite encadenar varios manejadores y, al recibir una solicitud, “preguntar” a cada manejador si puede procesarla. De esta forma todos los manejadores tienen la oportunidad de procesar la solicitud.
* Utiliza el patrón cuando sea fundamental ejecutar varios manejadores en un orden específico.
* Ya que puedes vincular los manejadores de la cadena en cualquier orden, todas las solicitudes recorrerán la cadena exactamente como planees.
* Utiliza el patrón Chain of Responsibility cuando el grupo de manejadores y su orden deban cambiar durante el tiempo de ejecución.
* Si aportas modificadores (*setters*) para un campo de referencia dentro de las clases manejadoras, podrás insertar, eliminar o reordenar los manejadores dinámicamente.

Ejemplo Practico:

Primero comenzamos creando la clase request:

class Request:

    def \_\_init\_\_(self, amount):

        self.amount = amount

esta clase contiene un atributo amount que contiene el monto de la solicitud.

Luego, creamos la clase abstracta Handler, que es la interfaz para los manejadores de solicitud. Tiene un atributo next\_handler para almacenar una referencia al siguiente manejador en la cadena y un método abstracto handle\_request() para manejar la solicitud.

class Handler(ABC):

    def \_\_init\_\_(self):

        self.next\_handler = None

    def set\_next\_handler(self, handler):

        self.next\_handler = handler

    @abstractmethod

    def handle\_request(self, request):

        pass

Después, implementamos tres clases de manejadores concretos: LowAmountHandler, MediumAmountHandler y HighAmountHandler. Cada uno de ellos hereda de Handler y sobrescribe el método handle\_request() para realizar la lógica de manejo de la solicitud correspondiente al nivel de monto.

Primer manejador de solicitud: maneja solicitudes con un monto menor o igual a $100

class LowAmountHandler(Handler):

    def handle\_request(self, request):

        if request.amount <= 100:

            print("La solicitud de valor", request.amount," ha sido aprobada por el manejador de monto bajo.")

        elif self.next\_handler:

            self.next\_handler.handle\_request(request)

Segundo manejador de solicitud: maneja solicitudes con un monto entre $101 y $500

class MediumAmountHandler(Handler):

    def handle\_request(self, request):

        if 101 <= request.amount <= 500:

            print("La solicitud de valor", request.amount," ha sido aprobada por el manejador de monto medio.")

        elif self.next\_handler:

            self.next\_handler.handle\_request(request)

Tercer manejador de solicitud: maneja solicitudes con un monto mayor a $500

class HighAmountHandler(Handler):

    def handle\_request(self, request):

        if request.amount > 500:

            print("La solicitud de valor", request.amount," ha sido aprobada por el manejador de monto alto.")

        elif self.next\_handler:

            self.next\_handler.handle\_request(request)

Cada manejador verifica si puede manejar la solicitud en función del monto. Si puede hacerlo, imprime un mensaje indicando que la solicitud ha sido aprobada. Si no puede manejarla, pasa la solicitud al siguiente manejador en la cadena llamando al método handle\_request() del siguiente manejador.

Creación de la cadena de responsabilidad

low\_amount\_handler = LowAmountHandler()

medium\_amount\_handler = MediumAmountHandler()

high\_amount\_handler = HighAmountHandler()

low\_amount\_handler.set\_next\_handler(medium\_amount\_handler)

medium\_amount\_handler.set\_next\_handler(high\_amount\_handler)

Luego instanciamos la clase Request dándoles sus respectivos valores:

request1 = Request(50)

low\_amount\_handler.handle\_request(request1)

request2 = Request(200)

low\_amount\_handler.handle\_request(request2)

request3 = Request(1000)

low\_amount\_handler.handle\_request(request3)

Como se puede apreciar en el ejemplo solo se hace llamado al manejador de monto bajo.

Texto

Descripción generada automáticamente

Sin embargo, cuando se ejecuta el código nos podemos dar cuenta de que solo la primera solicitud fue trabajada por el manejador de monto bajo mientras que los otros dos con los que no podía trabajar se los paso a los manejadores que si lo pudieran hacer.