Patrones de comportamiento

Chain of responsability

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"
[Christopher Alexander].

Por: Brayan Estiven Aguirre Aristizábal – 20231020156
Santiago Zamudio Diaz – 20231020128
Daniel Josué Rozo Rodríguez – 20222020006

1. Introducción

Los patrones de comportamiento en el desarrollo de software tienen como propósito principal proporcionar soluciones estándar y reutilizables para problemas específicos relacionados con la interacción y la distribución de responsabilidades entre objetos. Se centran en mejorar la comunicación y la colaboración entre objetos de un sistema, asegurando que el software sea más flexible, mantenible y adaptable a cambios. Además, promueven la separación de responsabilidades claras entre componentes del sistema, mejoran la escalabilidad y facilitan la implementación de nuevas funcionalidades sin afectar otras partes del sistema. Están diseñados para mejorar la estructura interna del software y cómo sus componentes interactúan entre sí.

El patrón de Cadena de Responsabilidad es un patrón de diseño de comportamiento que permite a múltiples objetos manejar una solicitud sin conocer explícitamente quién la manejará a continuación en una cadena. Cada objeto en la cadena tiene la oportunidad de procesar la solicitud o pasarla al siguiente en la cadena. Esto promueve el desacoplamiento entre el remitente de la solicitud y los objetos que la manejan, ofreciendo flexibilidad y extensibilidad en el manejo de operaciones.

2. Chain of Responsability

Nombre del patrón:

Chain of responsability

Clasificación del patrón:

Comportamiento

Intención:

Te permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.

También conocido por:

Cadena de responsabilidad, CoR, Chain of Command

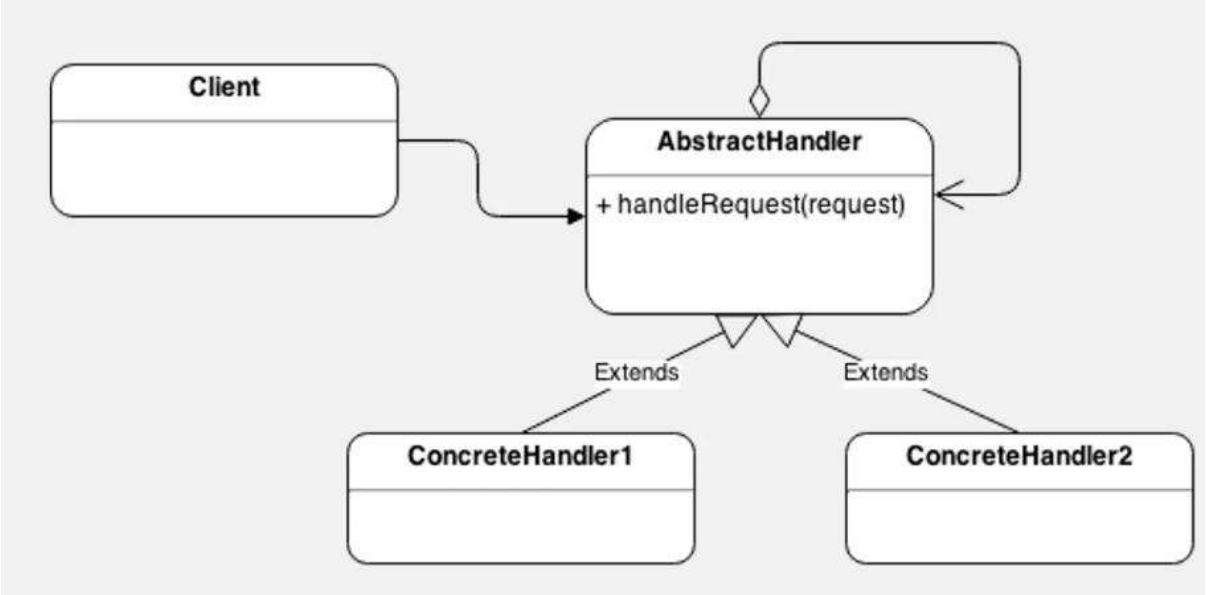
Motivación:

Permitir que múltiples objetos tengan la oportunidad de manejar una solicitud sin que el remitente de la solicitud conozca qué objeto finalmente la manejará. Este patrón desacopla el remitente de la solicitud de sus receptores, lo que proporciona flexibilidad en cómo se maneja la solicitud y permite agregar o modificar responsabilidades dinámicamente.

Aplicabilidad:

- El patrón te permite encadenar varios manejadores y, al recibir una solicitud, "preguntar" a cada manejador si puede procesarla. Así, todos los manejadores pueden procesar la solicitud.
- Ya que puedes vincular los manejadores de la cadena en cualquier orden, todas las solicitudes recorrerán la cadena exactamente como planees.
- Si aportas modificadores (setters) para un campo de referencia dentro de las clases manejadoras, podrás insertar, eliminar o reordenar los manejadores dinámicamente.

Estructura:



Participantes:

- Client: Usuario o sistema que inicia la ejecución del patrón.
- AbstractHandler: Clase base utilizada para definir la estructura de todos los ConcreteHandler. Esta clase es una agregación de sí misma, lo que le permite contener otro AbstractHandler que continuará con la cadena de ejecución.
- ConcreteHandler: Representan las implementaciones concretas de AbstractHandler las cuales se utilizarán para procesar las solicitudes.

Colaboraciones:

- El cliente solicita el procesamiento de una solicitud a una cadena de responsabilidad.
- El primer Handler intenta procesar el mensaje, sin embargo, no es capaz de procesarlo por alguna razón y envía el mensaje al siguiente handler de la cadena.
- El segundo Handler intenta procesar el mensaje sin éxito, por lo que envía el mensaje al siguiente Handler de la cadena.
- El tercer Handler también intenta procesar el mensaje sin éxito y envía el mensaje al siguiente Handler de la cadena.
- El HandlerN (Algun handler de la secuencia) por fin es capaz de procesar el mensaje exitosamente y regresa una respuesta (opcional) de tal forma que la respuesta es replicada por todos los Handlers pasados hasta llegar al Cliente.

Consecuencias:

- Puedes controlar el orden de control de solicitudes.
- Principio de responsabilidad única. Puedes desacoplar las clases que invoquen operaciones de las que realicen operaciones.

- Principio de abierto/cerrado. Puedes introducir nuevos manejadores en la aplicación sin descomponer el código cliente existente.
- Algunas solicitudes pueden acabar sin ser gestionadas

Implementación:

- Declara la interfaz manejadora y describe la firma de un método para manejar solicitudes.
- Decide cómo pasará el cliente la información de la solicitud dentro del método. La forma más flexible consiste en convertir la solicitud en un objeto y pasarlo al método de gestión como argumento.
- Una a una, crea subclases manejadoras concretas e implementa los métodos de control. Cada manejador debe tomar dos decisiones cuando recibe una solicitud:
    - Si procesa la solicitud.
    - Si pasa la solicitud al siguiente eslabón de la cadena.
- 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.
- 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.
- Debido a la naturaleza dinámica de la cadena, el cliente debe estar listo para gestionar los siguientes escenarios:
    - La cadena puede consistir en un único vínculo.
    - Algunas solicitudes pueden no llegar al final de la cadena.
    - Otras pueden llegar hasta el final de la cadena sin ser gestionadas.

Código ejemplo:

```python
from __future__ import annotations
from abc import ABC, abstractmethod
from typing import Any, Optional


class Handler(ABC):
    """
    La interfaz Handler declara un método para construir la cadena de manejadores.
    También declara un método para ejecutar una solicitud.
    """

    @abstractmethod
    def set_next(self, handler: Handler) -> Handler:
        pass

    @abstractmethod
    def handle(self, request) -> Optional[str]:
        pass
```

```python
class AbstractHandler(Handler):
    """
    El comportamiento predeterminado de encadenamiento se puede implementar dentro de una
    """

    _next_handler: Handler = None

    def set_next(self, handler: Handler) -> Handler:
        self._next_handler = handler
        # Devolver un manejador desde aquí nos permitirá enlazar los manejadores de una
        # manera conveniente como esta:
        # monkey.set_next(squirrel).set_next(dog)
        return handler

    def handle(self, request: Any) -> Optional[str]:
        if self._next_handler:
            return self._next_handler.handle(request)
        return None
```

```
"""
Todos los Concrete Handlers manejan una solicitud o la pasan al siguiente manejador e
"""


class MonkeyHandler(AbstractHandler):
    def handle(self, request: Any) -> Optional[str]:
        if request == "Plátano":
            return f"Mono: Yo comeré el {request}"
        else:
            return super().handle(request)


class SquirrelHandler(AbstractHandler):
    def handle(self, request: Any) -> Optional[str]:
        if request == "Nuez":
            return f"Ardilla: Yo comeré la {request}"
        else:
            return super().handle(request)
```

```
class DogHandler(AbstractHandler):
    def handle(self, request: Any) -> Optional[str]:
        if request == "Albóndiga":
            return f"Perro: Yo comeré la {request}"
        else:
            return super().handle(request)


def client_code(handler: Handler) -> None:
    """
    El código del cliente suele estar diseñado para trabajar con un solo manejador.
    En la mayoría de los casos, ni siquiera sabe que el manejador es parte de una cadena.
    """
```

```
    for comida in ["Nuez", "Plátano", "Taza de café"]:
        print(f"\nCliente: ¿Quién quiere un(a) {comida}?")
        resultado = handler.handle(comida)
        if resultado:
            print(f"  {resultado}")
        else:
            print(f"  {comida} quedó sin tocar.")


if __name__ == "__main__":
    mono = MonkeyHandler()
    ardilla = SquirrelHandler()
    perro = DogHandler()

    mono.set_next(ardilla).set_next(perro)

    # El cliente debería poder enviar una solicitud a cualquier manejador,
    print("Cadena: Mono > Ardilla > Perro")
    client_code(mono)
    print("\n")
```

Usos Conocidos:

Se utiliza en situaciones donde se necesita manejar solicitudes o eventos de manera flexible y desacoplada. Algunos usos comunes incluyen:

- **Manejo de solicitudes HTTP en frameworks web:** Permite que múltiples middleware o interceptores procesen solicitudes HTTP secuencialmente.
- **Eventos en GUI y sistemas de eventos:** Facilita el manejo de eventos donde cada objeto en la cadena puede decidir manejar un evento o pasarlo al siguiente objeto.
- **Filtros y validaciones:** Aplicación secuencial de filtros o validaciones a datos de entrada.
- **Logs y sistemas de registro de errores:** Permite el procesamiento y registro jerárquico de mensajes o eventos.
- **Manejo de excepciones**: Facilita el manejo y procesamiento de diferentes tipos de excepciones de manera modular.
- **Transformación de datos:** Aplicación secuencial de transformaciones a datos en múltiples pasos.

Relaciones con otros patrones:

- **Decorator:** Permite agregar responsabilidades dinámicamente a cada objeto en la cadena.
- **Command:** Puede encapsular solicitudes como objetos, facilitando su manejo dentro de la cadena.

- **Observer:** Permite notificar y manejar eventos dentro de la cadena de manera flexible y desacoplada.
- **Composite**: Estructura los objetos en la cadena en una jerarquía de árbol, facilitando el manejo de responsabilidades a diferentes niveles.
- **Strategy:** Implementa diferentes estrategias para manejar solicitudes de manera flexible, permitiendo cambiar dinámicamente cómo se procesan las solicitudes.

Referencias Bibliográficas

[1] Alonso, A. (2019). Patrones de diseño: Chain of Responsibility. Medium. Recuperado el 5 de julio de 2024, de https://adrianalonsodev.medium.com/patrones-de-dise%C3%B1o-chain-of-responsability-1460ee11c476

[2] Reactive Programming. (s.f.). Patrones de diseño: Chain of Responsibility. Recuperado el 5 de julio de 2024, de https://reactiveprogramming.io/blog/es/patrones-de-diseno/chain-of-responsability

[3] Refactoring Guru. (s.f.). Chain of Responsibility. Recuperado el 5 de julio de 2024, de https://refactoring.guru/es/design-patterns/chain-of-responsibility