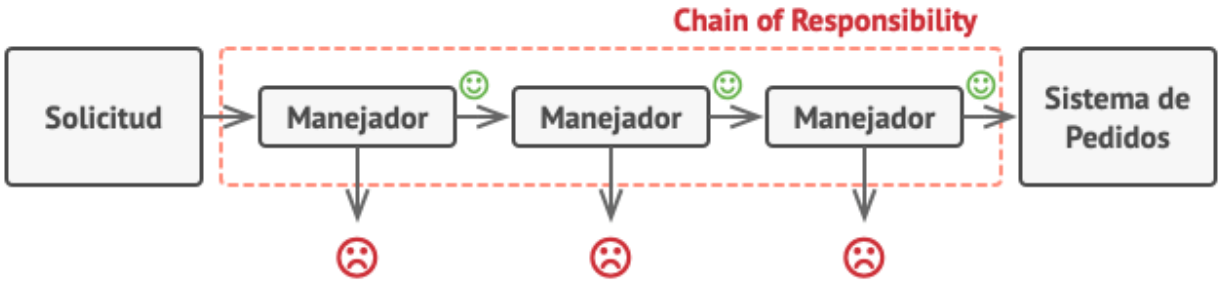
Patrones de diseño.
Silvestro Nuñez Maria Azul.

Chain of Responsability

Cadena de responsabilidad, CoR, Chain of Command

Patrón de comportamiento que 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. Sencillamente, su objetivo principal es desacoplar al emisor de un mensaje del receptor del mismo, permitiendo de esta manera que más de un objeto responda; tratan de encadenar a los receptores del mensaje que irán pasándoselo hasta que uno de ellos lo procese.



Problema:

Cuando hay comunicación entre dos objetos, normalmente estos se acoplan mediante una conexión. Pretendemos desacoplar el sistema, pero nuestro problema es que el receptor del mensaje no va a conocer directamente el origen del mismo.

Lo aplicamos:

- **Cuando el programa deba procesar distintos tipos de solicitudes de varias maneras, pero los tipos exactos de solitudes y sus secuencias no se conozcan de antemano.** Ya que permie encadenar varios manejadores y "consultar" a cada manejador si puede procesarla, así todos los manejadores tienen la oportunidad de procesar la solicitud.

- **Cuando sea fundamental ejecutar varios manejadores en un orden especifico.** Porque se pueden vincular los manejadores de la cadena en cualquier orden, todas las solicitudes recorrerán la cadena tal como se planee.

- **Cuando el grupo de manejadores y su orden deban cambiar durante el tiempo de ejecución.** Si aportas modificadores para un campo de referencia dentro de las clases manejadoras, podrás insertar, eliminar o reordenar los manejadores dinámicamente.

Solución:

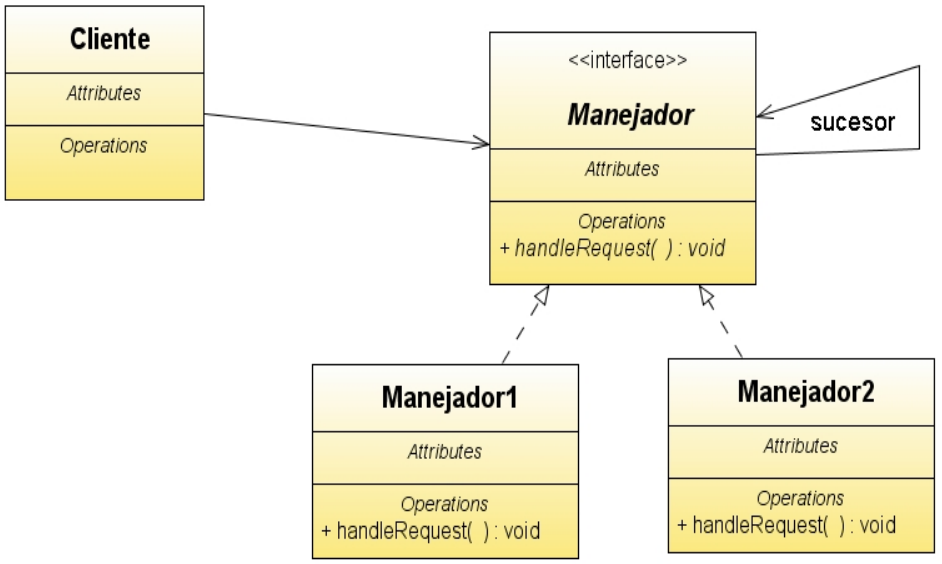
Se debe encontrar un mecanismo mediante el cual pasar mensajes a través de la cadena de objetos, para que si el que lo recibe no sabe procesarlo lo pase a otro objeto.

Se crea una interfaz Manejador que permite tratar las peticiones en general, también algunos ManejadoresConcretos que son los que se encargan de procesar una petición concreta. El cliente que desea enviar el mensaje pasará el mismo a un Manejador concreto, que se encargará o bien de procesarlo o de transferirlo a otros objetos que pertenezcan a la cadena.

Manejador: interfaz que define las operaciones necesarias para tratar los mensajes y propagarlos si corresponde.

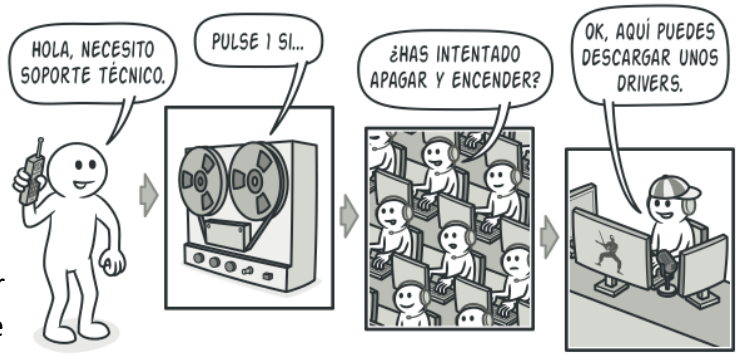
Manejadoresoncreto: implementa la interfaz Manejador. Se encarga de procesar un tipo de mensaje concreto o propagar el mensaje a otro miembro de la cadena en caso de que el mensaje no sea de dicho tipo.

Cliente: trata de enviar un mensaje a destino propagándolo mediante un Manejador conocido.



Analogía del mundo real:

Cuando se llama a atención al cliente por un problema técnico de algún producto o servicio Lo primero que oyes es la voz robótica del contestador automático. Que sugiere un numero de soluciones populares a varios problemas, si ninguna es relevante al caso. El robot te conecta con un operador humano. Si el operador humano tampoco puede ayudarte te pasan con un técnico especialista.



Implementación:

1. Declarar la interfaz manejadora y describir 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).

2. Para eliminar código boilerplate duplicado en manejadores concretos, puede merecer la pena crear una clase manejadora abstracta base, derivada de la interfaz manejadora.

   - Esta clase debe tener un campo para almacenar una referencia al siguiente manejador de la cadena. Considera hacer la clase inmutable. No obstante, si planeas modificar las cadenas durante el tiempo de ejecución, deberás definir un modificador (*setter*) para alterar el valor del campo de referencia.
   - También puedes implementar el comportamiento por defecto conveniente para el método de control, que consiste en reenviar la solicitud al siguiente objeto, a no ser que no quede ninguno. Los manejadores concretos podrán utilizar este comportamiento invocando al método padre.

3. 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.

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:
   - 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.

Pros:

- Se aplica el principio de Responsabilidad Única; elimina o reduce el acoplamiento existente entre el emisor y el receptor del mensaje. Se puede desacoplar las clases que invoquen operaciones de las que realicen operaciones.
- Se controla el orden en que se ejecutan los pasos de manera dinámica.
- Se aplica el principio de Abierto/Cerrado; permite ampliar con nuevos manejadores sin romper el código existente, mejorando así la mantenibilidad.

Contras:

- Algunas solicitudes pueden acabar sin ser gestionadas.

Patrones con los que se relaciona:

- Chain of Responsability se usa a menudo junto con Composite; cuando un componente hoja recibe una solicitud, puede pasarla a lo largo de la cadena de todos los componentes padre hasta la raíz del árbol de objetos.