**Propósito**

**Chain of Responsibility** es un patrón de diseño de comportamiento que 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..

**Problema**

Imagina que estás trabajando en un sistema de pedidos online. Quieres restringir el acceso al sistema de forma que únicamente los usuarios autenticados puedan generar pedidos. Además, los usuarios que tengan permisos administrativos deben tener pleno acceso a todos los pedidos.

Tras planificar un poco, te das cuenta de que estas comprobaciones deben realizarse secuencialmente. La aplicación puede intentar autenticar a un usuario en el sistema cuando reciba una solicitud que contenga las credenciales del usuario. Sin embargo, si esas credenciales no son correctas y la autenticación falla, no hay razón para proceder con otras comprobaciones.

Durante los meses siguientes, implementas varias de esas comprobaciones secuenciales.

* Uno de tus colegas sugiere que no es seguro pasar datos sin procesar directamente al sistema de pedidos. De modo que añades un paso adicional de validación para sanear los datos de una solicitud.
* Más tarde, alguien se da cuenta de que el sistema es vulnerable al desciframiento de contraseñas por la fuerza. Para evitarlo, añades rápidamente una comprobación que filtra las solicitudes fallidas repetidas que vengan de la misma dirección IP.
* Otra persona sugiere que podrías acelerar el sistema devolviendo los resultados en caché en solicitudes repetidas que contengan los mismos datos, de modo que añades otra comprobación que permite a la solicitud pasar por el sistema únicamente cuando no hay una respuesta adecuada en caché.

El código de las comprobaciones, que ya se veía desordenado, se vuelve más y más abotargado cada vez que añades una nueva función. En ocasiones, un cambio en una comprobación afecta a las demás. Y lo peor de todo es que, cuando intentas reutilizar las comprobaciones para proteger otros componentes del sistema, tienes que duplicar parte del código, ya que esos componentes necesitan parte de las comprobaciones, pero no todas ellas.

El sistema se vuelve muy difícil de comprender y costoso de mantener. Luchas con el código durante un tiempo hasta que un día decides refactorizarlo todo.

**Solución**

Al igual que muchos otros patrones de diseño de comportamiento, el **Chain of Responsibility**se basa en transformar comportamientos particulares en objetos autónomos llamados manejadores. En nuestro caso, cada comprobación debe ponerse dentro de su propia clase con un único método que realice la comprobación. La solicitud, junto con su información, se pasa a este método como argumento.

El patrón sugiere que vincules esos manejadores en una cadena. Cada manejador vinculado tiene un campo para almacenar una referencia al siguiente manejador de la cadena. Además de procesar una solicitud, los manejadores la pasan a lo largo de la cadena. La solicitud viaja por la cadena hasta que todos los manejadores han tenido la oportunidad de procesarla.

Y ésta es la mejor parte: un manejador puede decidir no pasar la solicitud más allá por la cadena y detener con ello el procesamiento.

En nuestro ejemplo de los sistemas de pedidos, un manejador realiza el procesamiento y después decide si pasa la solicitud al siguiente eslabón de la cadena. Asumiendo que la solicitud contiene la información correcta, todos los manejadores pueden ejecutar su comportamiento principal, ya sean comprobaciones de autenticación o almacenamiento en la memoria caché.

No obstante, hay una solución ligeramente diferente (y un poco más estandarizada) en la que, al recibir una solicitud, un manejador decide si puede procesarla. Si puede, no pasa la solicitud más allá. De modo que un único manejador procesa la solicitud o no lo hace ninguno en absoluto. Esta solución es muy habitual cuando tratamos con eventos en pilas de elementos dentro de una interfaz gráfica de usuario (GUI).

Por ejemplo, cuando un usuario hace clic en un botón, el evento se propaga por la cadena de elementos GUI que comienza en el botón, recorre sus contenedores (como formularios o paneles) y acaba en la ventana principal de la aplicación. El evento es procesado por el primer elemento de la cadena que es capaz de gestionarlo. Este ejemplo también es destacable porque muestra que siempre se puede extraer una cadena de un árbol de objetos.

Es fundamental que todas las clases manejadoras implementen la misma interfaz. Cada manejadora concreta solo debe preocuparse por la siguiente que cuente con el método ejecutar. De esta forma puedes componer cadenas durante el tiempo de ejecución, utilizando varios manejadores sin acoplar tu código a sus clases concretas.

**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 dinámicament

**Pros**

* 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.

## Contras

* Algunas solicitudes pueden acabar sin ser gestionadas.