Taller #5 – Felipe Lancheros 202211004

(Proyecto completo: https://github.com/edson-a-soares/gof_design_patterns

Fragmento del patrón: https://github.com/edson-a-soares/gof_design_patterns/tree/master/ChainOfResponsibility)

El proyecto en general sirve para identificar y conocer cómo funcionan los diferentes patrones de diseño basados en el libro “Design Patterns: Elements of Reusable Object-Oriented Software.” y su implementación en C++. Contiene carpetas individuales nombradas según sea el patrón. Cada carpeta se estructura mediante el código fuente:

- include: almacena los *headers* en archivos .h que contienen las definiciones de clases, funciones y otras estructuras de datos utilizadas.
- src: contienen la implementación real de las clases y funciones definidas en los archivos de encabezado

Y un archivo CMakeLists.txt hecho claramente en CMake. Este último es una herramienta de construcción y gestión de proyectos de código abierto, para configurar y generar proyectos de software multiplataforma. Este archivo contiene información sobre el proyecto, como los archivos fuente, las bibliotecas necesarias, los parámetros de compilación, las variables de entorno y otros detalles que son necesarios para construir y compilar el software correctamente. En general, no tiene mayor complejidad.

Es importante notar la separación de los archivos de encabezado y los archivos de código fuente, lo cual es una práctica común en C++ para mejorar la legibilidad y la organización del código.

Un reto en este contexto de implementación es siempre mantener una directiva del preprocesador para evitar la inclusión múltiple del archivo de encabezado, usando #infdef. Es un caso específico del lenguaje.

"Chain of Responsibility" es un patrón de diseño que permite enviar solicitudes a un objeto de manera indirecta a través de una cadena de objetos candidatos. Esta cadena de objetos se puede encadenar y pasar la solicitud de un objeto a otro hasta que un objeto sea capaz de manejarla. Este patrón es útil cuando se necesita enviar una solicitud a uno de varios objetos sin especificar explícitamente el receptor o se especifican de manera dinámica, permitiendo que un conjunto de objetos trabaje en conjunto para manejar solicitudes de manera eficiente y flexible.

El mundo del problema yace de un manejador de solicitudes de retiro de efectivo y decide si puede manejar la solicitud o si debe pasarla al siguiente banco (objeto) en la cadena de responsabilidad.

La implementación es un ejemplo que simula el funcionamiento de una estación de cajero automático. Primero, el programa muestra un menú de opciones de bancos disponibles para el usuario. Luego, se crea una instancia de la clase ATMStation con un puntero a un objeto ATMBankHandler, que se configura como una cadena de responsabilidad con cuatro tipos de bancos (Itau, HSBC, Citibank y Santander). El usuario selecciona una opción de banco y la cantidad de dinero que desea retirar. El método withdraw de la instancia ATMStation se llama con el valor ingresado y el objeto ATMBankHandler se encarga de procesar la solicitud y permitir o denegar el retiro de dinero. Finalmente, muestra un mensaje para que el usuario retire el dinero.

Es por ello que surge el reto y necesidad de implementar un destructor que libere la memoria utilizada por el siguiente objeto en la cadena de responsabilidad. Dentro de los *handler*, se encuentra su respectivo destructor.

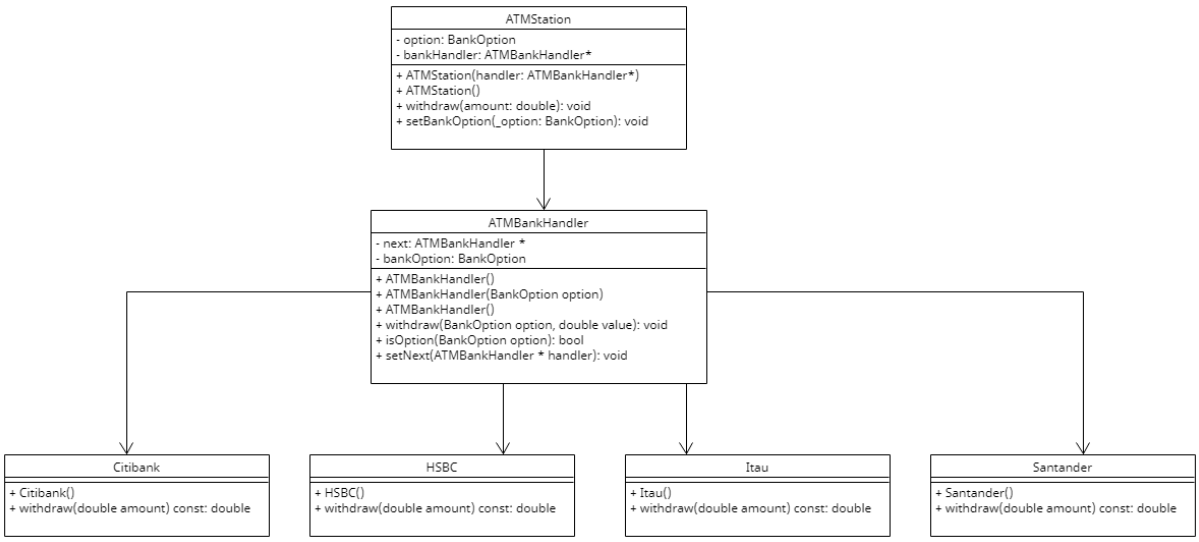
En el patrón, el *handler* es una parte crucial del diseño, puesto que maneja una solicitud específica y decide si puede manejar la solicitud o si debe pasarla al siguiente objeto en la cadena de responsabilidad. Esto proporciona una manera flexible de asignar responsabilidades a diferentes objetos y permite que los objetos trabajen juntos de manera eficiente para resolver problemas de manera dinámica. El método privado “BankOption” se encarga específicamente de este trabajo, siendo un puntero al siguiente objeto en el Chain of Responsibility. Al ser privado, se mantiene el “encapsulamiento” entre los objetos, es decir, no se conocen explícitamente entre sí.

Esta cadena va complementada por el método público y protegido withdraw. Su versión pública maneja la solicitud de retiro de dinero y la pasa al siguiente objeto en la cadena si no puede manejarla. Su versión protegida virtual y abstracta significa que cualquier clase (en este caso, banco) derivada puede implementar este método de acuerdo con su funcionalidad específica.

Para términos de coherencia, la función isOption() (dentro del *handler.cpp*) verifica si la opción de banco especificada es la misma que la opción de banco que se estableció en el constructor.

Ya el resto de las clases, son los objetos que representan a cada banco que intenta retirar dinero dentro de la cadena de valor. Misma funcionalidad con diferente nombre.

A continuación, se presenta el diagrama de clases con base al patrón de diseño:



Como se había mencionado anteriormente, el ATMStation posee un puntero al *handler*, el cual se encargará de manejar el Chain of Responsibility.

Chain of Responsibility es el patrón con mayor sentido en este proyecto porque permite separar el objeto que solicita una operación (en este caso, ATMStation) del objeto que la lleva a cabo (los objetos Itau, HSBC, Citibank y Santander).

La ventaja principal de este patrón es que se consigue una mayor flexibilidad y extensibilidad en el código, ya que se pueden agregar o eliminar bancos sin afectar al funcionamiento del resto del sistema y facilitando el mantenimiento del código. De igual forma, se evita la

necesidad de que cada banco conozca a los demás bancos o al objeto ATMStation, lo que simplifica el diseño y reduce el acoplamiento entre los objetos.

No obstante, es importante tener en cuenta las desventajas. En primer lugar, la implementación del patrón puede añadir complejidad adicional al código. En este caso, se han agregado varias clases y métodos adicionales para implementar el patrón, lo que puede aumentar el tiempo y la complejidad del desarrollo. Y, en segundo lugar, el patrón "Chain of Responsibility" puede tener limitaciones de escalabilidad, especialmente si se agregan muchas clases al patrón. Esto puede hacer que el proceso de manejo de las responsabilidades se vuelva más lento y consuma más recursos, lo que puede ser un problema en sistemas de alta carga.

Otra forma de solucionar el mundo de este problema podría ser sólo haber tenido una clase ATM por cada banco, lo que implicaría una mayor cantidad de clases y código duplicado. Sin embargo, las desventajas mencionadas para este caso empeorarían. Otra forma sería utilizar condicionales if-else para verificar el banco en el que se realiza la operación de retiro, lo que llevaría a un código más largo y difícil de mantener si se agregan nuevos bancos en el futuro. Aunque hay otras formas de solucionar los problemas que resuelve el patrón en este caso particular, el patrón de cadena de responsabilidad proporciona una solución simple que permite agregar fácilmente en el futuro sin necesidad de modificar el código existente y sin agregar una gran cantidad de clases adicionales.