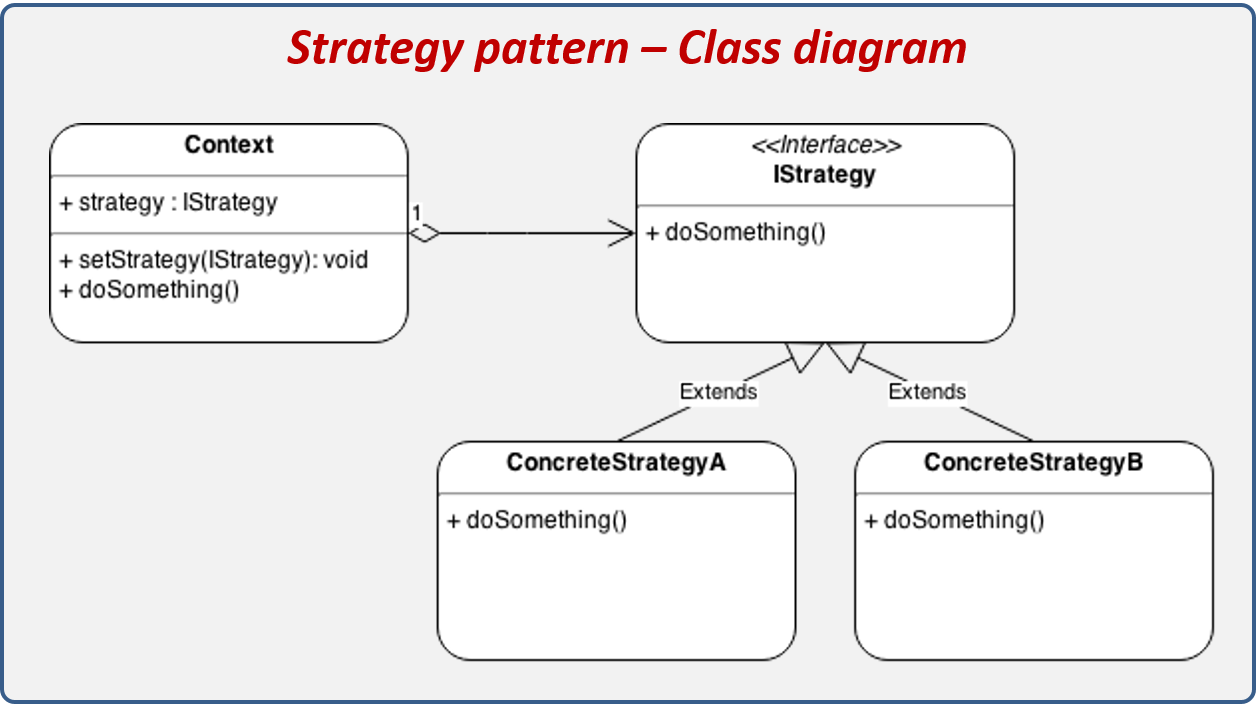
**BankStrategy**

El proyecto por analizar (https://github.com/SimonHGR/BankStrategy) se encarga de gestionar los pagos y retiros que hace una cuenta a un banco. Cuenta con clases que corresponden a la cuenta, a la interfaz (por consola) del banco y varias clases correspondientes a los distintos tipos de retiros y pagos que puede hacer el cliente.

La dificultad de la implementación radica en cómo gestionar estos distintos tipos de retiros y pagos por cada cliente de una forma que no requiera utilizar un atributo distinto para cada tipo de estrategia de pago o retiro (lo cual lo dejaría básicamente con varios atributos null, ya que solo utiliza una de cada una).

Para resolver este problema, utilizaron el patron Strategy, que consiste en “enmascarar” distintos algoritmos para seleccionar dinámicamente el mas adecuado para la situación y ejecutarlo. Este patrón se compone de 3 partes: Contexto, Estrategia y Estrategias concretas. El contexto es la clase que utiliza el algoritmo y tiene referencia a una estrategia concreta a la cual llama para ejecutar el algoritmo. La estrategia es una interfaz o clase abstracta que define la interfaz para todas las estrategias concretas. Finalmente están las estrategias concretas, que son las clases que tienen los algoritmos, que implementan la estrategia para que puedan conectarse directamente con el contexto.



El funcionamiento es básicamente tener una referencia a una estrategia concreta mediante la interfaz de estrategia, por lo que en el contexto se puede tener un único atributo para manejar estos algoritmos y solo sea necesario re-asignar el valor de este atributo para utilizar una estrategia concreta distinta. Esto aumenta la versatilidad del programa y facilita el agregar funcionalidades adicionales al modificar o agregar estrategias concretas, sin necesidad de hacer algún cambio grande en el contexto.

Este proyecto implementa el patrón strategy para gestionar las tarifas de pago y retiro para una cuenta, mediante los cuales se calcula los cobros y retiros. Estas estrategias se pueden cambiar fácilmente en tiempo de ejecución debido a que implementan dos interfaces para las estrategias de cobros y retiros, se utiliza la interfaz ApplyFeeStrategy para los cobros y GrantWithdrawalStrategy para los retiros.

Diagrama

Descripción generada automáticamente

Entonces, cuando guarda una de las estrategias concretas en sus atributos correspondientes a la interfaz y simplemente la llama para hacer las operaciones (ya sea calcular un cobro o autorizar un retiro). La estrategia utilizada es algo que se configura mediante otros parámetros internos del código, ya que depende de que tantos retiros se hagan cada cierto tiempo en esa cuenta.

Las ventajas de esta implementación es que permite configurar rápidamente las distintas estrategias para estos cálculos dependiendo de factores exteriores, también facilita el mantenimiento y adiciones al código ya que estos algoritmos se manejan en clases distintas y no se requiere un gran cambio en la clase contexto (que en este caso sería Account).

La principal desventaja es que el contexto no sabe cual es la estrategia concreta que utiliza ya que se enmascara debajo de la interfaz, lo cual puede ser un gran problema si la implementación requiere saber que tipo de cobro se le hace a la cuenta u otro tipo de situaciones donde se necesite conocer información concreta de la estrategia concreta utilizada.

Conociendo este tipo de desventaja, una solución alternativa podría ser la creación de clases dinámica y guardar el nombre de esa clase como un atributo en Account, ya con eso se podría facilitar el acceder a información acerca de cada estrategia. No obstante, esta solución podría ser mas compleja y por lo tanto, requerir mayores cambios al código de Account, además de una menor rapidez para cambiar entre distintas estrategias.