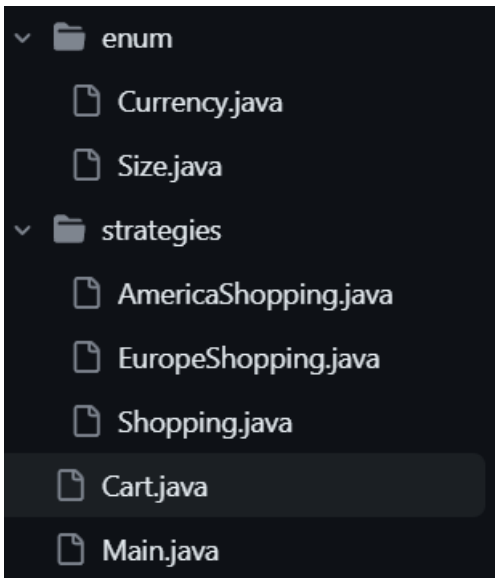
TALLER 5

Patrón: Strategy

El link del repositorio es: https://github.com/android-code/design-patterns/tree/master Más exactamente, el patron está en el siguiente link: https://github.com/android-code/design-patterns/tree/master/strategy/example

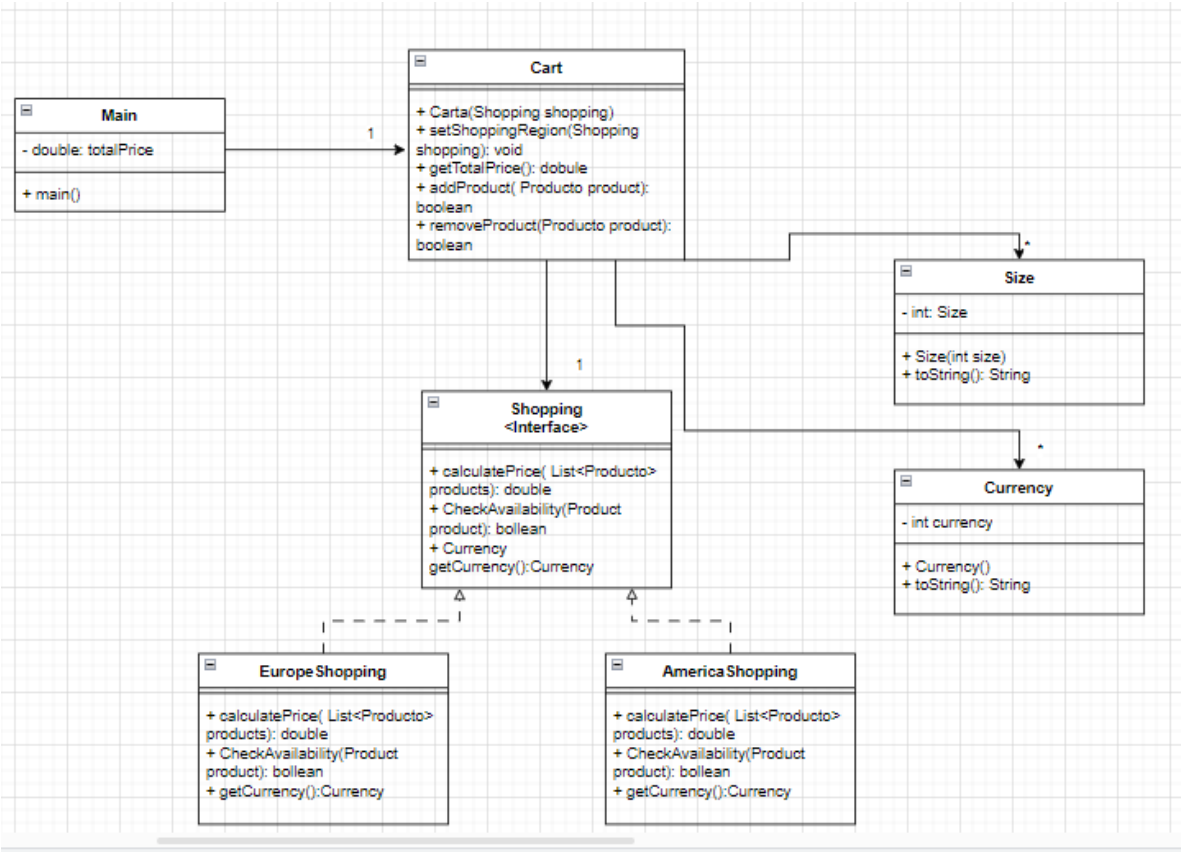
- El proyecto en código Java presentado tiene el propósito de ejemplificar gran parte de los patrones. En este caso, al estar analizando el patrón de Strategy, el ejemplo guarda productos y calcula el valor de un conjunto dependiendo de la zona geográfica en la cual se pidan los productos.

  Al ser un ejemplo no presenta grandes problemas, sin embargo, para una aplicación en la que se calcule el precio total de los productos dependiendo de cientos de países, la complejidad del problema a solucionar es mayor. En gran parte porque varios métodos para saber su disponibilidad y precio total.

- 

  El proyecto usa una estructura MVC, en el que la vista es la clase de Main al dar la opción al usuario para interactuar con el sistema y le permite añadir productos. El controller es la clase de Cart, la cual aplica los métodos de las demás clases. Dentro de las carpetas de enum se presentan los parámetros por defecto para clasificar parámetros de los productos. Por ultimo se tiene las clases de la carpeta Strategies, la cual, presenta la interfaz Shopping y sus subclases donde se implementará Strategies al incluir los métodos dependiendo de la zona geográfica.

- El patrón de Strategy es un patrón que permite tener un grupo de métodos o algoritmos que cambian a partir de uno o varios parámetros y encapsularlos en clases separadas para modificar sus métodos u objetos individualmente. Lo que implica que la clase originalmente encargada de ejecutar estos métodos delega el trabajo al parámetro y objeto vinculado con las subclases, en lugar de ejecutarlo por su cuenta. De esta forma mejora el acoplamiento al dividir las tareas y controlar las modificaciones de los métodos u objetos en las subclases de manera individual. También, sustituye la herencia por composición. El patrón se implementa con una interfaz que tiene un contrato con sus subclases, las cuales ejecutan métodos dependiendo del parámetro asociado a su subclase. Este patrón suele utilizarse para mejorar el uso de variantes de un algoritmo dentro de un objeto y poder cambiar de un algoritmo a otro durante el tiempo de ejecución.
- En el proyecto se aplica el patrón de Strategies al hacer la interfaz de Shopping y utilizar como subclases AmericanShopping y EuropeShopping. En este caso, los métodos que identifican la disponibilidad y los precios totales de los productos cambian por cada zona geográfica, por lo que, los algoritmos tienen variantes que se localizan en sus clases. Esto se puede ver en el siguiente diagrama:



- Es una buena idea utilizar este patrón para la aplicación, ya que, el objetivo es calcular el precio total de varios productos, sin embargo, los métodos en cada país son diferente, por lo que, Strategy permite separar cada una de las formas de calcularlo por país. Esto implica que la aplicación tiene control de cada método individual sin

generar acoplamiento en una sola clase que calcule el precio con gran cantidad de condicionales.

- Se puede utilizar otros métodos para emular Strategy:
    - Utilizar gran cantidad de condicionales en una clase, pero genera acoplamiento, por lo que soluciona los problemas, pero no el objetivo de usar Strategy
    - Utilizar herencia que simule el trabajo de la interfaz con subclases. En este caso sí se emularía los objetivos de Strategy al no generar acoplamiento.