PATRÓN DE DISEÑO

Ronald Diaz - 202111309
Felipe Pineda - 202112562

Patrón de diseño escogido: **Strategy**
Repositorio de patrón de diseño: **https://github.com/java9s/strategy-pattern-example.git**

STRATEGY

Consiste en un patrón de diseño de comportamiento que permite definir una familia de algoritmos, colocando cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

EJECUTABILIDAD

Utiliza el patrón Strategy cuando quieras utiliza distintas variantes de un algoritmo dentro de un objeto y poder cambiar de un algoritmo a otro durante el tiempo de ejecución.

El patrón Strategy te permite alterar indirectamente el comportamiento del objeto durante el tiempo de ejecución asociándolo con distintos subobjetos que pueden realizar subtareas específicas de distintas maneras.

Utiliza el patrón Strategy cuando tengas muchas clases similares que sólo se diferencien en la forma en que ejecutan cierto comportamiento.

El patrón Strategy te permite extraer el comportamiento variante para ponerlo en una jerarquía de clases separada y combinar las clases originales en una, reduciendo con ello el código duplicado.

Utiliza el patrón para aislar la lógica de negocio de una clase, de los detalles de implementación de algoritmos que pueden no ser tan importantes en el contexto de esa lógica.

El patrón Strategy te permite aislar el código, los datos internos y las dependencias de varios algoritmos, del resto del código. Los diversos clientes obtienen una interfaz simple para ejecutar los algoritmos y cambiarlos durante el tiempo de ejecución.

Utiliza el patrón cuando tu clase tenga un enorme operador condicional que cambie entre distintas variantes del mismo algoritmo.

El patrón Strategy te permite suprimir dicho condicional extrayendo todos los algoritmos para ponerlos en clases separadas, las cuales implementan la misma interfaz. El objeto original delega la ejecución a uno de esos objetos, en lugar de implementar todas las variantes del algoritmo.

REPOSITORIO IMPLEMENTADO:

En este repositorio lo que buscaban era encontrar a través de un navegador la ruta más fácil, para llegar a tu destino donde a través del patrón Strategy tomes una clase que hace algo específico de muchas formas diferentes y extraigas todos esos algoritmos para colocarlos en clases separadas llamadas *estrategias*.

La clase original, llamada *navigation*, debe tener un campo para almacenar una referencia a una de las estrategias. El contexto delega el trabajo a un objeto de estrategia vinculado en lugar de ejecutarlo por su cuenta. La clase navigation no es responsable de seleccionar un algoritmo adecuado para la tarea. En lugar de eso, el cliente pasa la estrategia deseada a la clase contexto. De hecho, la clase navigation no sabe mucho acerca de las estrategias. Funciona con todas las estrategias a través de la misma interfaz genérica, que sólo expone un único método para disparar el algoritmo encapsulado dentro de la estrategia seleccionada.

De esta forma, el contexto se vuelve independiente de las estrategias concretas, así que puedes añadir nuevos algoritmos o modificar los existentes sin cambiar el código de la clase contexto o de otras estrategias.

¿CÓMO IMPLEMENTARLO?

1. En la clase navigation, identifica un algoritmo que tienda a sufrir cambios frecuentes. También puede ser un enorme condicional que seleccione y ejecute una variante del mismo algoritmo durante el tiempo de ejecución.
2. Declara la interfaz estrategia común a todas las variantes del algoritmo.
3. Uno a uno, extrae todos los algoritmos y ponlos en sus propias clases. Todas deben implementar la misma interfaz estrategia.
4. En la clase Navigator, añade un campo para almacenar una referencia a un objeto de estrategia. Proporciona un modificador *set* para sustituir valores de ese campo. La clase contexto debe trabajar con el objeto de estrategia únicamente a través de la interfaz estrategia, además se puede definir una interfaz que permita a la estrategia acceder a sus datos.
5. Los clientes de la clase navigation deben asociarla con una estrategia adecuada que coincida con la forma en la que esperan que la clase contexto realice su trabajo principal.

PROS Y CONTRAS

PROS:

- Puedes intercambiar algoritmos usados dentro de un objeto durante el tiempo de ejecución.
- Puedes aislar los detalles de implementación de un algoritmo del código que lo utiliza.
- Puedes sustituir la herencia por composición.
- *Principio de abierto/cerrado*. Puedes introducir nuevas estrategias sin tener que cambiar el contexto.

CONTRAS:

- Si sólo tienes un par de algoritmos que raramente cambian, no hay una razón real para complicar el programa en exceso con nuevas clases e interfaces que vengan con el patrón.
- Los clientes deben conocer las diferencias entre estrategias para poder seleccionar la adecuada.
- Muchos lenguajes de programación modernos tienen un soporte de tipo funcional que permite implementar distintas versiones de un algoritmo dentro de un grupo de funciones anónimas. Entonces puedes utilizar estas funciones exactamente como habrías utilizado los objetos de estrategia, pero sin saturar tu código con clases e interfaces adicionales.

DIAGRAMA:

Un diagrama parecido a codigo seria este: