**STRATEGY PATTERN**

**LOS PRINCIPIOS DE DISEÑO UTILIZADOS EN ESTE PATRÓN SON:**

* Identifica aspectos que cambien, encapsúlalos y sepáralos de lo que se mantiene igual.
* Programa para una interfaz no para una implementación.
* Favorece la composición sobre la herencia.

Esto nos permitirá encapsular los cambios, facilitando el mantenimiento y la expiación del sistema.

**Los posibles casos de uso de este patrón serian.**

* Cuando nuestros objetos deben estar preparados para afrontar diversos escenarios sin cambiar las interacciones de estos con el cliente.
* Si estás pensando usar la herencia para poder agregar nuevos comportamientos a tus objetos.
* Si dentro de tu clase haces uso intensivo de las condicionales if, else, swicth, case (eso quiere decir que tu clase tiene asignado muchos comportamientos y/o responsabilidades)

**OTRO EJEMPLO:**

Imagina que eres el director técnico de un equipo de fútbol, y tienes un juego muy importante el próximo domingo, ¿cómo plantearías el juego?

Como eres un buen director técnico, tienes que prever los posibles escenarios y tener una estrategia preparada para cada escenario.

De esa manera la formación y/o comportamiento de tus jugadores deberían cambiar dependiendo de los diversos factores que se presenten en el desarrollo juego, tales como: cuando atacas, cuando defiendes, expulsan a un jugador, vas ganando, vas perdiendo, juegan de visita, juegan de local, el rival es débil, el rival es fuerte, etc.

**BENEFICIOS:**

* Puede intercambiar algoritmos utilizados dentro de un objeto en tiempo de ejecución.
* Puede aislar los detalles de implementación de un algoritmo del código que lo usa.
* Puede reemplazar la herencia con la composición.
* Aplica el principio abierto / cerrado.
* Puede introducir nuevas estrategias sin tener que cambiar el contexto.

**DESVENTAJAS:**

* Si solo tiene un par de algoritmos y rara vez cambian, no hay una razón real para complicar demasiado el programa con nuevas clases e interfaces que vienen junto con el patrón.
* Los clientes deben ser conscientes de las diferencias entre las estrategias para poder seleccionar una adecuada.
* Muchos lenguajes de programación modernos tienen soporte de tipo funcional que le permite implementar diferentes versiones de un algoritmo dentro de un conjunto de funciones anónimas.
* Entonces podría usar estas funciones exactamente como lo habría hecho con los objetos de estrategia, pero sin inflar su código con clases e interfaces adicionales.