**Patrón de diseño strategy**Presentación:

* Situación problemática
* Qué es?
* Cuándo se usa?
* Beneficios
* Estructura
* Ejemplo práctico/codificado
* Comparación con otros patrones
* Conclusión

Situación Problemática:

Imagina que estás desarrollando un sistema de navegación para una aplicación de mapas. Tu aplicación debe proporcionar direcciones para dos tipos de usuarios: conductores y peatones. Los conductores necesitan rutas que minimicen el tiempo de viaje, considerando el tráfico y las carreteras principales. Los peatones, por otro lado, necesitan rutas que minimicen la distancia y preferiblemente eviten grandes avenidas y autopistas.

Inicialmente, implementaste una única estrategia para calcular las rutas que se adapta bien a las necesidades de los conductores. Sin embargo, cuando los peatones comenzaron a usar la aplicación, encontraron que las rutas sugeridas eran a menudo poco prácticas para caminar, ya que a menudo los dirigían a través de las mismas carreteras rápidas que los conductores.

Aquí es donde surge el problema: tu aplicación necesita proporcionar rutas óptimas tanto para conductores como para peatones, pero una estrategia única no es suficiente. Cambiar la estrategia para adaptarse a las necesidades de los peatones podría afectar negativamente la experiencia de los conductores, y viceversa.

¿Qué es?:

“Strategy” es un patrón de diseño de comportamiento que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

¿Cuándo se usa? :

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.*

Beneficios:

* 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.

Estructura:

Diagrama

Descripción generada automáticamente

1. La clase Contexto mantiene una referencia a una de las estrategias concretas y se comunica con este objeto únicamente a través de la interfaz estrategia.
2. La clase Contexto mantiene una referencia a una de las estrategias concretas y se comunica con este objeto únicamente a través de la interfaz estrategia.
3. La clase Contexto mantiene una referencia a una de las estrategias concretas y se comunica con este objeto únicamente a través de la interfaz estrategia.
4. La clase Contexto mantiene una referencia a una de las estrategias concretas y se comunica con este objeto únicamente a través de la interfaz estrategia.
5. La clase Contexto mantiene una referencia a una de las estrategias concretas y se comunica con este objeto únicamente a través de la interfaz estrategia.

Ejemplo práctico/codificado:

En este ejemplo, crearé una aplicación simple de navegación que puede calcular rutas para conductores y peatones utilizando el patrón de diseño "Strategy". La aplicación calculará rutas basadas en el tiempo para conductores y en la distancia para peatones.

Primero, crearemos las clases para las estrategias:

**# Estrategia para conductores**

**class ConductorStrategy:**

**def calcular\_ruta(self, origen, destino):**

**print(f"Calculando ruta para conductor desde {origen} hasta {destino} basada en el tiempo.")**

**# Estrategia para peatones**

**class PeatonStrategy:**

**def calcular\_ruta(self, origen, destino):**

**print(f"Calculando ruta para peatón desde {origen} hasta {destino} basada en la distancia.")**

A continuación, crearemos la clase Contexto que utilizará una de las estrategias según el tipo de usuario:

**class NavegacionContexto:**

**def \_\_init\_\_(self, estrategia):**

**self.estrategia = estrategia**

**def calcular\_ruta(self, origen, destino):**

**self.estrategia.calcular\_ruta(origen, destino)**

Ahora, en el código principal, el usuario puede elegir entre ser conductor o peatón y obtener una ruta adecuada:

**# Código principal**

**if \_\_name\_\_ == "\_\_main\_\_":**

**tipo\_usuario = input("¿Eres conductor o peatón? ").lower()**

**if tipo\_usuario == "conductor":**

**estrategia = ConductorStrategy()**

**elif tipo\_usuario == "peatón":**

**estrategia = PeatonStrategy()**

**else:**

**print("Tipo de usuario no válido.")**

**exit()**

**origen = input("Ingrese el punto de origen: ")**

**destino = input("Ingrese el punto de destino: ")**

**# Creamos el contexto con la estrategia seleccionada y calculamos la ruta**

**contexto = NavegacionContexto(estrategia)**

**contexto.calcular\_ruta(origen, destino)**

Explicación:

* *Hemos creado dos clases, ConductorStrategy y PeatonStrategy, que representan las estrategias para conductores y peatones respectivamente. Estas clases tienen un método calcular\_ruta que imprime un mensaje indicando cómo se calcula la ruta según la estrategia.*
* *La clase NavegacionContexto representa el contexto en el que se utiliza una de las estrategias. Tiene un método calcular\_ruta que delega el cálculo de la ruta a la estrategia correspondiente.*
* *En el código principal, el usuario ingresa el tipo de usuario (conductor o peatón) y los puntos de origen y destino. Según el tipo de usuario seleccionado, se crea una instancia de la estrategia correspondiente y se pasa al contexto de navegación. Luego, se calcula la ruta utilizando la estrategia seleccionada.*

Comparación con otros patrones:

Bridge, State, Strategy (y, hasta cierto punto, Adapter) tienen estructuras muy similares. De hecho, todos estos patrones se basan en la composición, que consiste en delegar trabajo a otros objetos. Sin embargo, todos ellos solucionan problemas diferentes. Un patrón no es simplemente una receta para estructurar tu código de una forma específica. También puede comunicar a otros desarrolladores el problema que resuelve.

Command y Strategy pueden resultar similares porque puedes usar ambos para parametrizar un objeto con cierta acción. No obstante, tienen propósitos muy diferentes.

Decorator te permite cambiar la piel de un objeto, mientras que Strategy te permite cambiar sus entrañas.

Conclusión:

Es en situaciones como esta donde el patrón de diseño "Strategy" es esencial. Al implementar diferentes estrategias para calcular rutas (una para conductores y otra para peatones), puedes cambiar dinámicamente la estrategia según el tipo de usuario sin tener que cambiar todo el sistema de navegación.

Esto permite que la aplicación sea flexible y pueda adaptarse a diferentes escenarios, asegurando que tanto conductores como peatones obtengan las rutas más adecuadas para sus necesidades específicas.