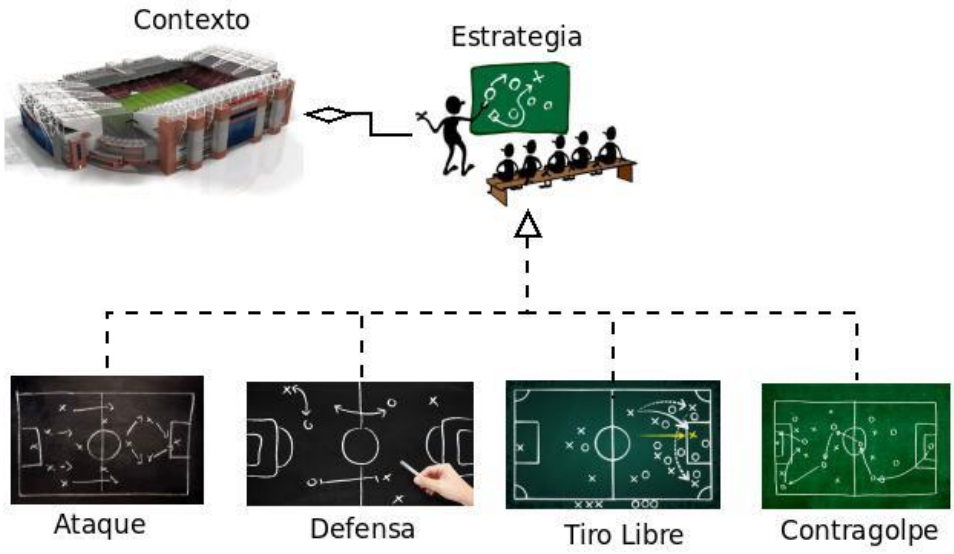
Patrón Estrategia

El patrón estrategia se base en **delegación**, este es un mecanismo, por medio del cual una clase delega en otra una determinada funcionalidad. Se aplica como sustitución a la herencia. Permite sustituir la herencia múltiple por una alternativa más conveniente, evitando los conflictos de nombres que se plantean en la herencia múltiple.

Intención

La estrategia es un patrón de diseño de comportamiento que le permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer que sus objetos sean intercambiables.



Problema

Un día decidiste crear una aplicación de navegación para viajeros ocasionales. La aplicación se centraba en un hermoso mapa que ayudaba a los usuarios a orientarse rápidamente en cualquier ciudad.

Una de las funciones más solicitadas para la aplicación fue la planificación automática de rutas. Un usuario debe poder ingresar una dirección y ver la ruta más rápida a ese destino que se muestra en el mapa.

La primera versión de la aplicación solo podía construir rutas sobre carreteras. La gente que viajaba en coche rebosaba de alegría. Pero aparentemente, no a todo el mundo le gusta conducir en sus vacaciones. Entonces, con la próxima actualización, agregó una opción para crear rutas para caminar. Inmediatamente después de eso, agregó otra opción para permitir que las personas usen el transporte público en sus rutas.

Sin embargo, eso fue solo el comienzo. Más tarde planeó agregar la creación de rutas para ciclistas. Y más adelante, otra opción para construir rutas por todos los atractivos turísticos de una ciudad.

El código del navegador se infló.

Si bien desde una perspectiva comercial la aplicación fue un éxito, la parte técnica te causó muchos dolores de cabeza. Cada vez que agregaba un nuevo algoritmo de enrutamiento, la clase principal del navegador se duplicaba en tamaño. En algún momento, la bestia se volvió demasiado difícil de mantener.

Cualquier cambio en uno de los algoritmos, ya fuera una simple corrección de errores o un ligero ajuste de la puntuación de la calle, afectó a toda la clase, lo que aumentó la posibilidad de crear un error en el código que ya funcionaba.

Además, el trabajo en equipo se volvió ineficiente. Sus compañeros de equipo, que habían sido contratados justo después del lanzamiento exitoso, se quejan de que pasan demasiado tiempo resolviendo conflictos de combinación. La implementación de una nueva función requiere que cambie la misma clase enorme, lo que entra en conflicto con el código producido por otras personas.

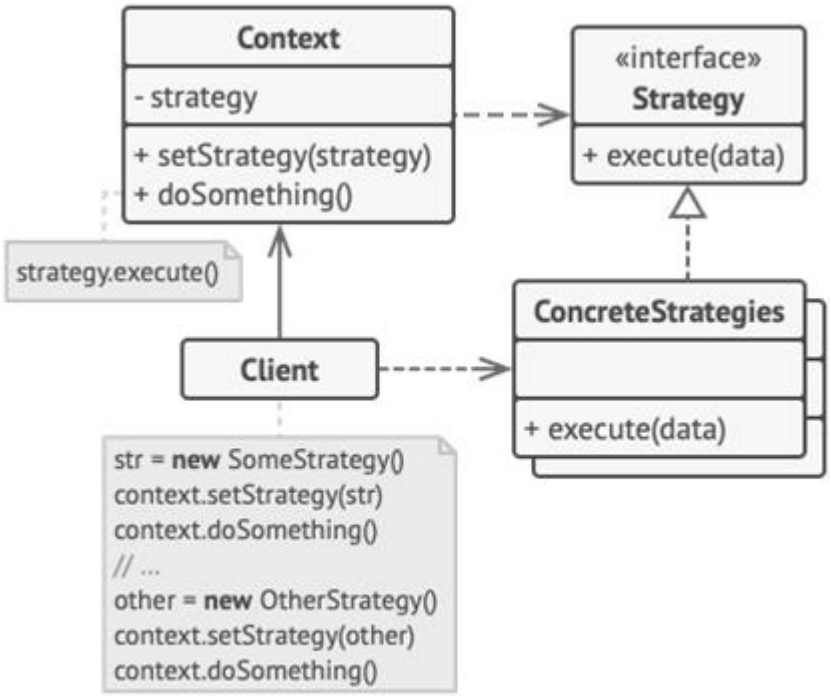
Solución

El patrón de estrategia sugiere que se tome una clase que haga algo específico de muchas maneras diferentes y extraiga todos estos algoritmos en clases separadas llamadas *estrategias* .

La clase original, llamada ***contexto*** , 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 sí solo.

El **contexto** no es responsable de seleccionar un algoritmo apropiado para el trabajo. En cambio, el cliente pasa la estrategia deseada al contexto. De hecho, el contexto no sabe mucho sobre estrategias. Funciona con todas las estrategias a través de la misma interfaz genérica, que solo expone un único método para activar el algoritmo encapsulado dentro de la estrategia seleccionada.

De esta manera el contexto se independiza de estrategias concretas, por lo que puedes agregar nuevos algoritmos o modificar los existentes sin cambiar el código del contexto u otras estrategias.



1. El **contexto** mantiene una referencia a una de las estrategias concretas y se comunica con este objeto solo a través de la interfaz de estrategia.

2. La interfaz de **Estrategia** es común a todas las estrategias concretas. Declara un método que utiliza el contexto para ejecutar una estrategia.

3. Las **estrategias** concretas implementan diferentes variaciones de un algoritmo que utiliza el contexto.

4. El **contexto** llama al método de ejecución en el objeto de **estrategia** vinculado cada vez que necesita ejecutar el algoritmo. El contexto no sabe con qué tipo de estrategia trabaja o cómo se ejecuta el algoritmo.

5. El **Cliente** crea un objeto de **estrategia** específico y lo pasa al contexto. El **contexto** expone un setter que permite a los clientes reemplazar la estrategia asociada con el contexto en tiempo de ejecución.

Utilice el patrón de estrategia cuando desee utilizar diferentes variantes de un algoritmo dentro de un objeto y poder cambiar de un algoritmo a otro durante el tiempo de ejecución.
El patrón de estrategia le permite alterar indirectamente el comportamiento del objeto en tiempo de ejecución al asociarlo con diferentes sub-objetos que pueden realizar subtareas específicas de diferentes maneras.

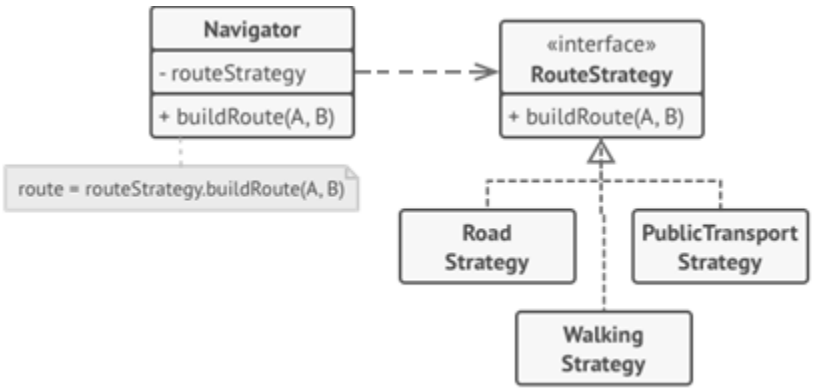
Use la estrategia cuando tenga muchas clases similares que solo difieren en la forma en que ejecutan algún comportamiento.
El patrón de estrategia le permite extraer el comportamiento variable en una jerarquía de clases separada y combinar las clases originales en una sola, reduciendo así el código duplicado.

Utilice el patrón para aislar la lógica empresarial de una clase de los detalles de implementación de los algoritmos que pueden no ser tan importantes en el contexto de esa lógica.
El patrón de estrategia le permite aislar el código, los datos internos y las dependencias de varios algoritmos del resto del código. Varios clientes obtienen una interfaz simple para ejecutar los algoritmos y cambiarlos en tiempo de ejecución.

Use el patrón cuando su clase tenga una declaración condicional masiva que cambie entre diferentes variantes del mismo algoritmo.
El patrón de estrategia le permite eliminar dicho condicional al extraer todos los algoritmos en clases separadas, todas las cuales implementan la misma interfaz. El objeto original delega la ejecución a uno de estos objetos, en lugar de implementar todas las variantes del algoritmo.

Cómo implementar

1. En la clase de **contexto**, identifique un algoritmo que sea propenso a cambios frecuentes. También puede ser un condicional masivo que selecciona y ejecuta una variante del mismo algoritmo en tiempo de ejecución.
2. Declarar la interfaz de estrategia común a todas las variantes del algoritmo.
3. Uno por uno, extraiga todos los algoritmos en sus propias clases. Todos deberían implementar la interfaz de estrategia.
4. En la clase de contexto, agregue un campo para almacenar una referencia a un objeto de estrategia. Proporcione un setter para reemplazar los valores de ese campo. El contexto debe funcionar con el objeto de estrategia solo a través de la interfaz de estrategia. El contexto puede definir una interfaz que permita a la estrategia acceder a sus datos.
5. Los clientes del contexto deben asociarlo con una estrategia adecuada que coincida con la forma en que esperan que el contexto realice su trabajo principal.



En nuestra aplicación de navegación, cada algoritmo de enrutamiento se puede extraer a su propia clase. El método acepta un origen y un destino y devuelve una colección de los puntos de control de la ruta.

Aunque con los mismos argumentos, cada clase de enrutamiento puede crear una ruta diferente, a la clase de navegador principal realmente no le importa qué algoritmo se selecciona, ya que su trabajo principal es representar un conjunto de puntos de control en el mapa. La clase tiene un método para cambiar la estrategia de enrutamiento activa,

por lo que sus clientes, como los botones en la interfaz de usuario, pueden reemplazar el comportamiento de enrutamiento seleccionado actualmente por otro.

Relaciones con otros patrones

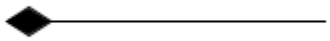
- Bridge , State , Strategy (y hasta cierto punto Adapter) tienen estructuras muy similares. De hecho, todos estos patrones se basan en la composición, que es delegar trabajo a otros objetos. Sin embargo, todos resuelven problemas diferentes. Un patrón no es solo una receta para estructurar su código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

Estrategia versus Puente

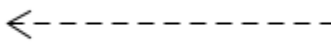
El diagrama de clases UML para el **patrón Estrategia** es muy similar que el diagrama del **patrón Puente**. Sin embargo, estos dos patrones de diseño no son iguales en su intención. Mientras que el patrón Strategy está pensado para el comportamiento, el patrón Bridge está pensado para la estructura.

Cuando comparamos el **patron Strategy** con el **patron Puente**, generalmente el **patrón Puente** busca que una clase desacople una parte que le pertenece a ella misma, acomodando dicha "parte" en otra clase. En cambio el fin del **patrón Strategy** es diferente, este busca hacer que otra(s) clase(s) implemente deferentes funcionalidades las cuales posiblemente vayan a cambiar o extenderse con el tiempo.

El **Patrón Puente** es un **patrón estructural** (¿CÓMO SE CONSTRUYE UN COMPONENTE DE SOFTWARE?), aquí la relación suele ser de **composición**



El **Patrón Estrategia** es un **patrón comportamental** (¿CÓMO QUIERES EJECUTAR UN COMPORTAMIENTO EN EL SOFTWARE?), aquí la relación suele ser de **asociación.**



Se utiliza el **patrón de estrategia** cuando se abstrae el comportamiento que podría ser proporcionado por una fuente externa y se utiliza el **patrón de puente** cuando los que extrae es en realidad parte vital y estructural de la clase. El código real será muy similar - sólo estás aplicando los patrones por razones ligeramente diferentes.

Recursos y Fuentes
https://refactoring.guru/design-patterns/strategy
https://www.youtube.com/watch?v=VQ8V0ym2JSo&t=84s