**Propósito**

**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.

**Problema**

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

Una de las funciones más solicitadas para la aplicación era la planificación automática de rutas. Un usuario debía poder introducir una dirección y ver la ruta más rápida a ese destino mostrado en el mapa.

La primera versión de la aplicación sólo podía generar las rutas sobre carreteras. Las personas que viajaban en coche estaban locas de alegría. Pero, aparentemente, no a todo el mundo le gusta conducir durante sus vacaciones. De modo que, en la siguiente actualización, añadiste una opción para crear rutas a pie. Después, añadiste otra opción para permitir a las personas utilizar el transporte público en sus rutas.

Sin embargo, esto era sólo el principio. Más tarde planeaste añadir la generación de rutas para ciclistas, y más tarde, otra opción para trazar rutas por todas las atracciones turísticas de una ciudad.

Aunque desde una perspectiva comercial la aplicación era un éxito, la parte técnica provocaba muchos dolores de cabeza. Cada vez que añadías un nuevo algoritmo de enrutamiento, la clase principal del navegador doblaba su tamaño. En cierto momento, la bestia se volvió demasiado difícil de mantener.

Cualquier cambio en alguno de los algoritmos, ya fuera un sencillo arreglo de un error o un ligero ajuste de la representación de la calle, afectaba a toda la clase, aumentando las probabilidades de crear un error en un código ya funcional.

Además, el trabajo en equipo se volvió ineficiente. Tus compañeros, contratados tras el exitoso lanzamiento, se quejaban de que dedicaban demasiado tiempo a resolver conflictos de integración. Implementar una nueva función te exige cambiar la misma clase enorme, entrando en conflicto con el código producido por otras personas.

**Solución**

El patrón Strategy sugiere que tomes esa 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 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 su cuenta.

La clase contexto 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 contexto 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.

En nuestra aplicación de navegación, cada algoritmo de enrutamiento puede extraerse y ponerse en su propia clase con un único método crearRuta. El método acepta un origen y un destino y devuelve una colección de puntos de control de la ruta.

Incluso contando con los mismos argumentos, cada clase de enrutamiento puede crear una ruta diferente. A la clase navegadora principal no le importa qué algoritmo se selecciona ya que su labor principal es representar un grupo de puntos de control en el mapa. La clase tiene un método para cambiar la estrategia activa de enrutamiento, de modo que sus clientes, como los botones en la interfaz de usuario, pueden sustituir el comportamiento seleccionado de enrutamiento por otro.

## Analogía

**Aplicabilidad**

 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.

**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 te 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.