Strategy

Propósito

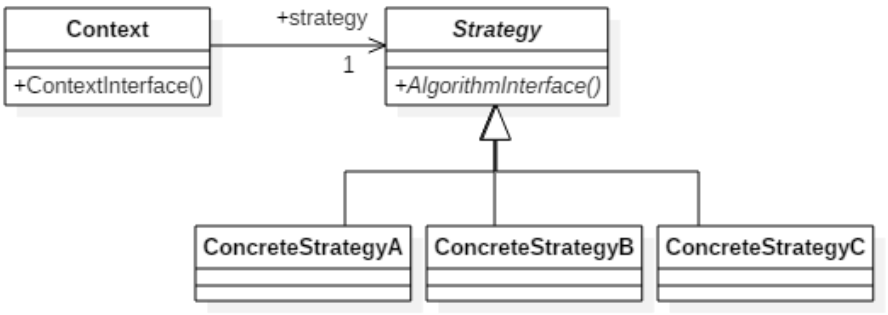
Define una familia de algoritmos, encapsula cada uno en un objeto, de modo que son intercambiables. El Strategy permite cambiar el algoritmo sin que afecte al cliente.

Aplicabilidad

Utilice el Strategy:

- Tenemos muchas clases relacionadas que difieren en su comportamiento. El Strategy permite configurar una clase con un comportamiento elegido entre varios posibles
- Se necesita manejar variantes de un algoritmo. Se puede definir, por ejemplo, diferentes versiones en cuanto a la complejidad (espacial o temporal). Se puede utilizar el Strategy cuando se implementa esas variaciones como una jerarquía de clases
- Un algoritmo utiliza datos que los clientes no deben conocer. Se puede utilizar el patrón Strategy para evitar exponer estructuras de datos complejas específicas del algoritmo
- Una clase define diferentes comportamientos que aparecen como sentencias condicionales múltiples. En vez de tener muchos condicionales, el Strategy invita a mover cada condición en su propia clase.

Estructura



Participantes

- Strategy: declara una interface común para todos los algoritmos soportados. Context utiliza esta interface para invocar el algoritmo definido en un ConcreteStrategy
- ConcreteStrategy: implementa el algoritmo utilizando la interface Strategy
- Context:
    - Se configura con un objeto ConcreteStrategy
    - Mantiene una referencia a un objeto Strategy
    - Puede definir una interface que permite al Strategy acceder a sus datos

Colaboraciones

- Strategy y Context interactúan para implementar el algoritmo elegido. Un objeto Context puede pasar al objeto Strategy los datos requeridos por el algoritmo cuando

se invoca. Otra alternativa es que el objeto Context se pase a sí mismo como argumento en las operaciones Strategy. Esto permite que el Strategy pueda llamar al Context cuando sea necesario.

- Un contexto reenvía las peticiones de su cliente a su estrategia. Normalmente los clientes crean y pasan un objeto ConcreteStrategy al contexto. Por lo tanto, los clientes interactúan únicamente con el contexto. Un cliente suele tener una familia de clases ConcreteStrategy donde elegir.

Consecuencias

El patrón Strategy tiene los siguientes beneficios y cargas:

1. Familias de algoritmos relacionados: Las jerarquías de clases Strategy definen una familia de algoritmos o comportamientos que los contextos pueden reutilizar. La herencia puede ayudar a factorizar la funcionalidad común de los algoritmos.
2. Una alternativa a la subclasificación: La herencia ofrece otra forma de soportar la variedad de algoritmos o comportamientos. Se puede subclasificar Context directamente para obtener los diferentes comportamientos. Pero esto "cablea" el comportamiento a Context. Mezcla la implementación del algoritmo con la de Context, lo que hace que Context sea más difícil de comprender, mantener y extender. Además no puedes variar el algoritmo dinámicamente. Al final terminas con muchas clases relacionadas cuya única diferencia es el algoritmo o comportamiento que emplean. Encapsular el algoritmo en una clase separada (Strategy) permite variar el algoritmo independientemente de su contexto, lo cual hace más sencillo cambiarlo, comprenderlo y extenderlo.
3. Las estrategias (Strategy) eliminan las sentencias condicionales. El patrón Strategy ofrece una alternativa a las sentencias condicionales para elegir el comportamiento deseado. Cuando se mezclan diferentes comportamientos en una clase, se hace difícil evitar las sentencias condicionales. Encapsular cada comportamiento en clases Strategy separadas elimina las sentencias condicionales.
4. Elegir entre implementaciones. El Strategy proporciona diferentes implementaciones del mismo comportamiento. El cliente puede elegir entre estrategias con diferente complejidad espacial o temporal.
5. Los clientes puede que tengan que conocer las diferentes estrategias. El patrón tiene el inconveniente cuando el cliente debe comprender antes las diferencias entre los Strategy para poder elegir el más adecuado. Esto hace que los clientes deban conocer detalles de implementación. Con lo cual hay que usar el patrón Strategy solamente cuando la variación del comportamiento sea relevante a los clientes.
6. Sobrecarga en la comunicación entre Strategy y Context. La interface Strategy es compartida por todos los ConcreteStrategy con independencia de si el algoritmo que implementan es sencillo o complejo. Con lo cual, algunos ConcreteStrategy no utilizarán toda la información que se les pasa. Los más simples no usarán ninguna. Esto significa que habrá muchas ocasiones que Context crea e inicializa parámetros que nunca serán utilizados. Si ocurre esto, entonces será necesario un acoplamiento más estrecho entre Strategy y Context.
7. Se incrementa el número de objetos: El Strategy incrementa el número de objetos de la aplicación. En ocasiones se puede reducir esta sobrecarga implementando las estrategias como objetos sin estado que comparten los contextos. Cualquier estado

residual se puede mantener en Context, que se pasa al objeto Strategy en cada petición. El patrón Flyweight describe esta solución con mayor detalle.