POLITÉCNICA

"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

MiW

Patrones de Diseño

25. *Strategy*

Luis Fernández Muñoz

https://www.linkedin.com/in/luisfernandezmunyoz

setillofm@gmail.com

INDICE

1. Problema

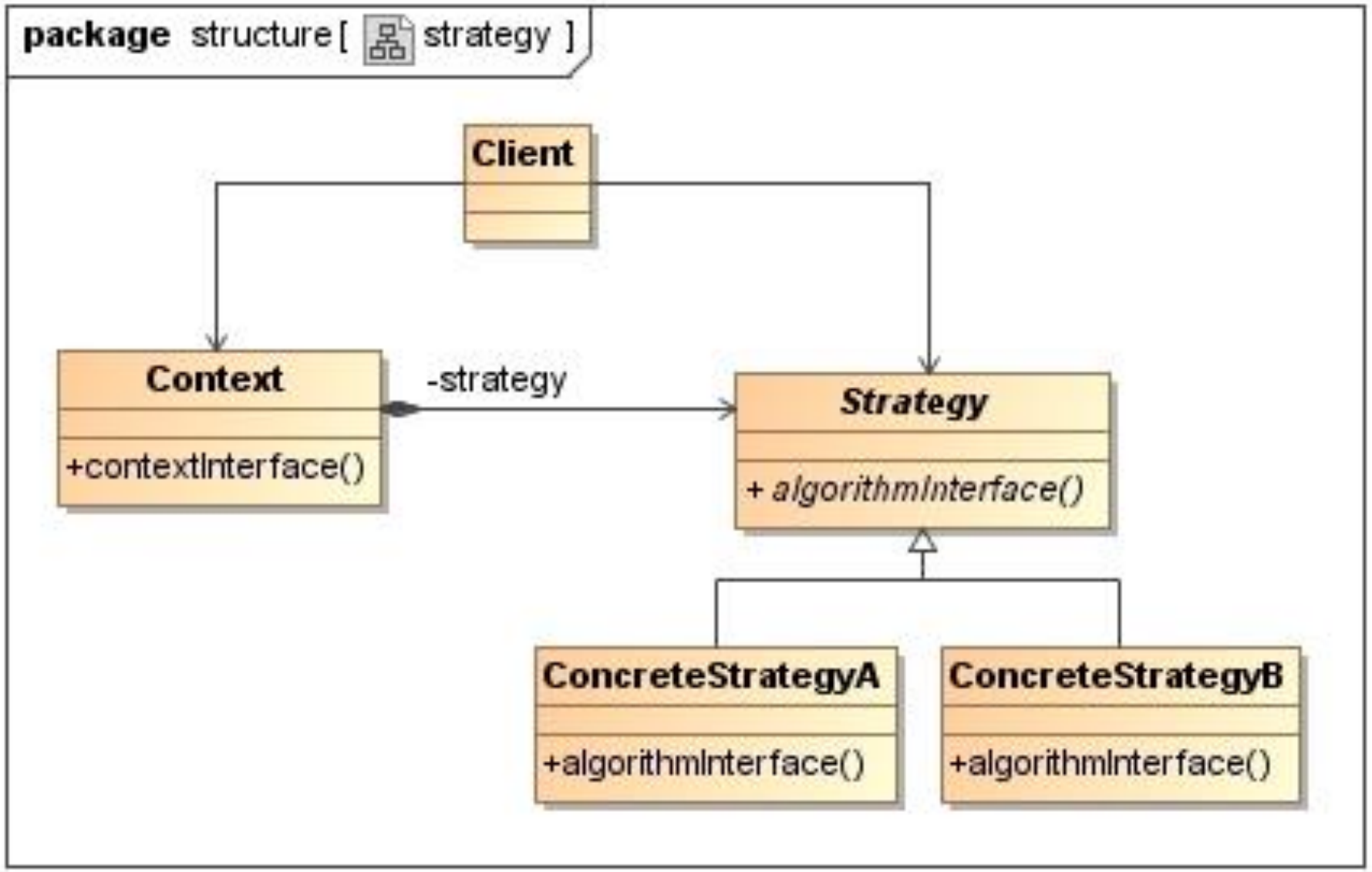
- *Cuando alguien comienza a aprender a ligar establece a fuego una técnica de acercamiento inicial (p.e. forzar casualidad, afición compartida, persona en común, te invito a algo, …) que considera que es su mejor opción (p.e. donde se siente más seguro, donde cree que cosechará más existos, …). Pero normalmente no está capacitado para improvisar con nuevas técnicas.*
- *La diferencia con los expertos en ligar es que no establecen una política rígida sino que pueden escoger entre una u otra dependiendo del contexto o incluso incorporar a lo largo de su vida nuevas técnicas que no conocía. Esto les da la flexibilidad para lograr una tasa de éxito mayor independientemente del contexto y del tiempo.*

- *Define una familia de algoritmos, encapsula cada uno y los hace intercambiables. Permite al algoritmo variar independientemente del cliente que lo use*

1. Problema

- Úsese el patrón *Strategy* cuando:
  - Muchas clases relacionadas difieren en su comportamiento. Las estrategias permiten configurar una clase con un determinado comportamiento de entre muchos posibles
  - Se necesitan distintas variantes de un algoritmo. Por ejemplo podríamos definir algoritmos que reflejasen distintas soluciones de compromiso entre tiempo y espacio. Pueden usarse estrategias cuando estas variantes se implementan como una jerarquía de clases de algoritmos
  - Un algoritmo usa datos que los clientes no deberían conocer. Úsese el patrón *Strategy* para evitar exponer estructuras de datos complejas y dependientes del algoritmo.
  - Una clase define muchos comportamientos, y éstos se representan como múltiples sentencias condicionales en sus operaciones.

2. Solución


package structure [strategy]
Client
Context
+contextInterface()
-strategy
Strategy
+ algorithmInterface()
ConcreteStrategyA
+algorithmInterface()
ConcreteStrategyB
+algorithmInterface()

2. Solución

- Las interfaces Estrategia y Contexto deben permitir a una EstrategiaConcreta acceder de manera eficiente a cualquier dato que ésta necesite del contexto, y viceversa.
  - Un enfoque es hacer que Contexto pase los datos como parámetros a las operaciones de Estrategia – en otras palabras, lleva los datos a la estrategia-. Esto mantiene a Estrategia y Contexto desacoplados. Por otro lado, Contexto podría pasar datos a la Estrategia que ésta no necesita
  - Otra técnica consiste en que un contexto se pase a sí mismo como argumento, y que la estrategia pida los datos explícitamente al contexto. Como alternativa, la estrategia puede guardar una referencia a su contexto, eliminando así la necesidad de pasar nada. Pero ahora Contexto debe definir una interfaz más elaborada para sus datos, lo que acopla más estrechamente a Estrategia y Contexto

2. Solución

- Pueden usarse las plantillas para configurar una clase con una estrategia. Esta técnica solo puede aplicar si se puede seleccionar la Estrategia en tiempo de compilación y no hay que cambiarla en tiempo de ejecución. En este caso, la clase Contexto se define en una clase plantilla que tiene como parámetro una clase Estrategia.
- La clase Contexto puede simplificarse en caso de que tenga sentido no tener un objeto Estrategia. Contexto comprueba si tiene un objeto Estrategia antes de acceder a él. En caso de que exista, lo usa normalmente. Si no hay una estrategia, Contexto realiza el comportamiento predeterminado. La ventaja de este enfoque es que los clientes no tienen que tratar con los objetos Estrategia a menos que no les sirva el comportamiento predeterminado

3. Consecuencias

- Las jerarquías de clases Estrategia definen una familia de algoritmos o comportamientos para ser reutilizados por los contextos. La herencia puede ayudar a sacar factor común de la funcionalidad de estos algoritmos.
- La herencia ofrece otra forma de permitir una variedad de algoritmos o comportamientos. Se puede heredar de Contexto para proporcionar diferentes comportamientos. Pero esto liga el comportamiento al Contexto, mezclando la implementación del algoritmo con la del Contexto, lo que hace que éste sea más difícil de comprender, mantener y extender. Y no se puede modificar el algoritmo dinámicamente. Acabaremos teniendo muchas clases relacionadas cuya única diferencia es el algoritmo o comportamiento que utilizan. Encapsular el algoritmo en clases Estrategia separadas nos permite variar el algoritmos independiente de su contexto, haciéndolos más fácil de cambiar, comprender y extender.

3. Consecuencias

- Ofrece una alternativa a las sentencias condicionales para seleccionar el comportamiento deseado. Cuando se juntan muchos comportamientos en una clase es difícil no usar sentencias condicionales para seleccionar el comportamiento correcto.
- Las estrategias pueden proporcionar distintas implementaciones del mismo comportamiento. El cliente puede elegir entre estrategias con diferentes soluciones de compromiso entre tiempo y espacio
- Tiene el inconveniente potencial de que un cliente debe comprender cómo difieren las Estrategias antes de seleccionar la adecuada. Los clientes pueden estar expuestos a cuestiones de implantación. Por tanto, debería usarse solo cuando la variación de comportamiento sea relevante a los clientes

3. Consecuencias

- La interfaz de Estrategia es compartida por todas las clases EstrategiaConcreta, ya sea el algoritmo que implementa trivial o complejo. Por tanto, es probable que algunos objetos EstrategiaConcreta no usen toda la información que reciben a través de dicha interfaz; las estrategias concretas simples pueden incluso no utilizar nada de dicha información. Eso siginifica que habrá veces en las que el contexto crea e inicializa parámetros que nunca se usan. Si esto puede ser un problema, necesitaremos un acoplamiento más fuerte entre Estrategia y Contexto.
- Las estrategias aumentan el número de objetos de una aplicación. A veces se puede reducir este coste implementando las estrategias como objetos sin estado que puedan ser compartidos por el contexto. El contexto mantiene cualquier estado residual, pasándolo en cada petición al objeto Estrategia. Las estrategias compartidas no deberían mantener el estado entre invocaciones

4. Relaciones

- *Flyweight* para crear las distintas estrategias

5. Comparativa

- ***Strategy** vs **Decorator.***
  - Ver *Decorator* vs *Strategy*
- ***Strategy** vs **Template Method.***
  - *Template Method* usa la herencia para modificar una parte de un algoritmo. *Strategy* usa delegación para evitar el algoritmo completo