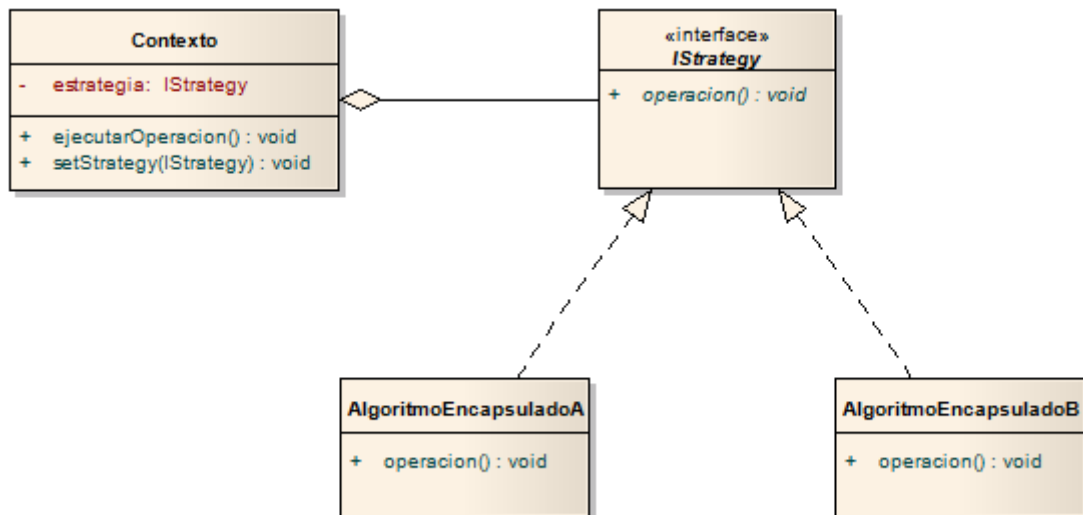


PATRÓN STRATEGY

Objetivo:

«Definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. Strategy permite cambiar el algoritmo independientemente de los clientes que lo utilicen».



Un patrón de diseño puede ayudarnos a “encapsular distintos algoritmos” a partir de una «plantilla» cuyos hijos se encargaban de especializar. El patrón *Strategy*, se caracteriza por **encapsular un algoritmo completo ignorando los detalles de su implementación**, permitiendo intercambiarlo en tiempo de ejecución para permitir actuar a la clase cliente (consumidora), con un comportamiento distinto.

El nombre de este patrón evoca la posibilidad de **realizar un cambio de estrategia en tiempo de ejecución** sustituyendo un objeto que se encargará de implementarla. No nos preocupará el «cómo». De hecho, ni siquiera nos importará «el qué»: la clase que actúa como interfaz del patrón únicamente tendrá que exponer el método o métodos que deberá invocar el cliente.

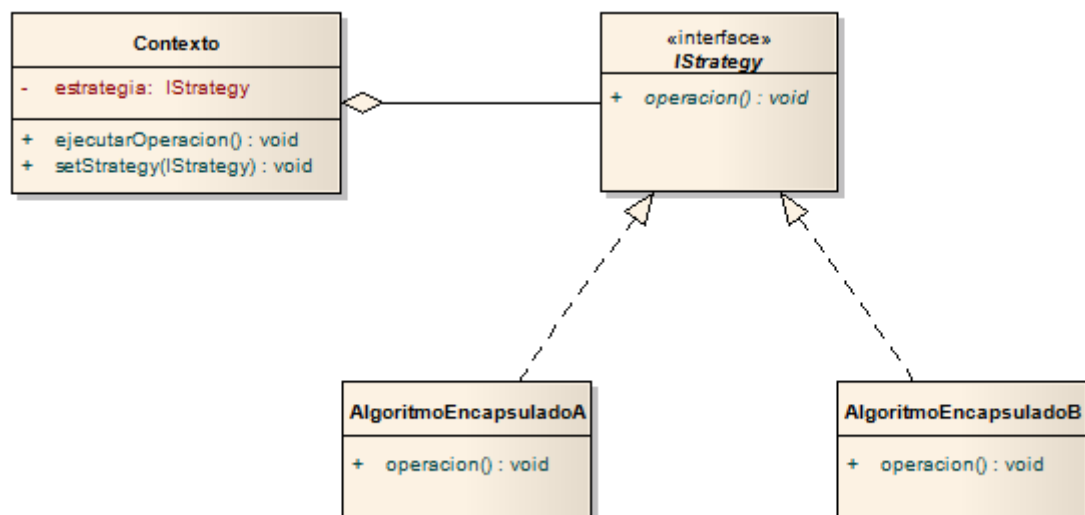
Patrón Strategy y relación con los videojuegos



Un ejemplo clásico para entender este patrón es el de un protagonista de un videojuego en el cual manejamos a **un soldado que puede portar y utilizar varias armas distintas**.

La clase (o clases) que representan a nuestro soldado no deberían de preocuparse de los detalles de las armas que porta: debería bastar, por ejemplo, con un método de interfaz **«atacar»** que dispare el arma actual y otro método **«recargar»** que inserte munición en ésta (si se diera el caso). En un momento dado, otro método **«cambiarArma»** podrá sustituir el objeto equipado por otro, manteniendo la interfaz intacta. Da igual que nuestro soldado porte un rifle, una pistola o un fusil: **los detalles de cada estrategia estarán encapsulados dentro de cada una de las clases intercambiables que representan las armas**. Nuestra clase cliente (el soldado) únicamente debe preocuparse de las acciones comunes a todas ellas: atacar, recargar y cambiar de arma. Éste último método, de hecho, será el encargado de realizar la operación de **«cambio de estrategia»** que forma parte del patrón.

Por lo tanto, el patrón *Strategy* es, como podemos imaginar, uno de los patrones más utilizados a la hora de diseñar software. Siempre que exista una posibilidad de realizar una tarea de distintas formas posibles, el patrón *Strategy* tendrá algo que decir al respecto.



El diagrama de clases del patrón muestra una interfaz denominada *IStrategy* que expone un método *operacion()*. Las clases que implementen esta interfaz serán

aquellas que implementen las distintas estrategias a realizar por el cliente; es sencillo: sólo una interfaz, y varias clases que la implementan.

La filosofía del patrón, por lo tanto, radica en el enlace entre la llamada **clase de contexto y la propia interfaz**. Esta clase de contexto será el intermediario entre el cliente y las clases que implementan la estrategia, y por lo tanto, sus funciones serán simples: cambiar la estrategia actual y ejecutarla.

Implementando el patrón

Armaremos un ejemplo de aplicación del patrón, pero en el contexto de los autos, a través de un hipotético módulo de la centralita del vehículo que nos permitirá alternar entre una conducción normal y deportiva. La diferencia entre ambas será simple: mayor consumo, mayor potencia y mayor velocidad. Podríamos agregar más comportamientos «personalizados», como el endurecimiento de la suspensión, pero con estos dos elementos será suficiente para captar la idea.

A) Creamos la interfaz *ITipoConduccion*, que simbolizará la interfaz de la estrategia (*IStrategy*). Le agregamos tres métodos:

1. uno para obtener la **descripción del tipo de conducción actual**,
2. otro que proporcione el **incremento de velocidad** en relación al combustible inyectado y
3. un tercer método que indique la **cantidad de potencia suministrada por el motor**, también en proporción al combustible que recibe.

Vea el código de la interfaz *ITipoConduccion*.

B) Agregamos las estrategias en sí, es decir, las clases que implementan la interfaz y dotan de distintos comportamientos que serán seleccionados por el contexto. Mencionamos que utilizaríamos dos: conducción normal y conducción deportiva, por lo tanto, crearemos dos clases que proporcionen distintos comportamientos para los mismos métodos:

Vea el código de las clases *ConduccionNormal* y *ConduccionDeportiva*.

C) Lo siguiente será crear el contexto. *Esta clase será la encargada de establecer la conexión entre el cliente y las clases que implementan la estrategia, sustituyendo la clase que la implementa dependiendo del comportamiento esperado.* Se compondrá de

una referencia a la interfaz que implementarán las estrategias más un método que permita cambiar de instancia (es decir, una *property* o un *setter* de toda la vida). A partir de esta funcionalidad básica, el contexto podrá realizar otras operaciones relacionadas con la estrategia que pretende modelar, como por ejemplo la invocación de sus métodos o la encapsulación del cambio de estrategia.

En realidad, la propia clase cliente puede actuar como clase de contexto, pero siempre será mejor minimizar el acoplamiento entre las estrategias y las reglas de negocio. De este modo, respetaremos otro de los principios de la orientación a objetos: *una clase, una responsabilidad* (SOLID).

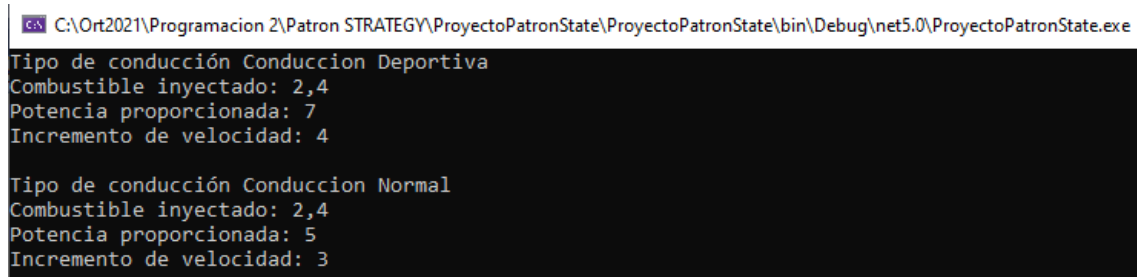
Vea el código de las clase Contexto.

D) Para comprobar el funcionamiento de nuestro cliente, bastará con utilizar este código que hará uso del contexto para cambiar de estrategia en tiempo de ejecución:

Vea el código de las clase Vehiculo.

E) Finalmente, el código que invoca a nuestro cliente será el siguiente:

Vea el código del Main.



```
C:\Ort2021\Programacion 2\Patron STRATEGY\ProyectoPatronState\ProyectoPatronState\bin\Debug\net5.0\ProyectoPatronState.exe
Tipo de conducción Conduccion Deportiva
Combustible inyectado: 2,4
Potencia proporcionada: 7
Incremento de velocidad: 4

Tipo de conducción Conduccion Normal
Combustible inyectado: 2,4
Potencia proporcionada: 5
Incremento de velocidad: 3
```

¿Cuándo utilizar este patrón? Ejemplos reales

Este patrón es aconsejable, en situaciones en los que una misma operación (o conjunto de operaciones) puedan realizarse de formas distintas.

El patrón *Strategy* realiza una tarea bastante similar al patrón *Template Method*, salvo porque en este caso el algoritmo no tiene por qué contar con pasos en común y porque *Strategy* confía en la composición mientras que *Template Method* se basa en la herencia.

Ejemplos reales de este patrón se aplican en:

- La serialización de objetos. Una interfaz que exponga un método *serialize()* podrá codificar un objeto en distintos formatos (String64, XML, JSON). El cliente no necesita saber cómo se realizará esta operación: bastará con que el contexto seleccione la estrategia adecuada y el resultado de la operación dependerá de la opción concreta que se haya seleccionado.
- Del mismo modo podemos pensar en una conexión a un servicio web: podremos realizarla mediante TCP/IP, HTTP, HTTPS, Named Pipes... todo esto deberá ser transparente para el cliente: El contexto será el encargado de adoptar una forma concreta de conexión.
- Los compresores funcionan a través de estrategias (se utiliza un algoritmo distinto para comprimir en *zip* o en *rar*), y en general, cualquier programa capaz de almacenar y transmitir datos en distintos formatos implementarán este patrón.