EL PATRÓN STRATEGY

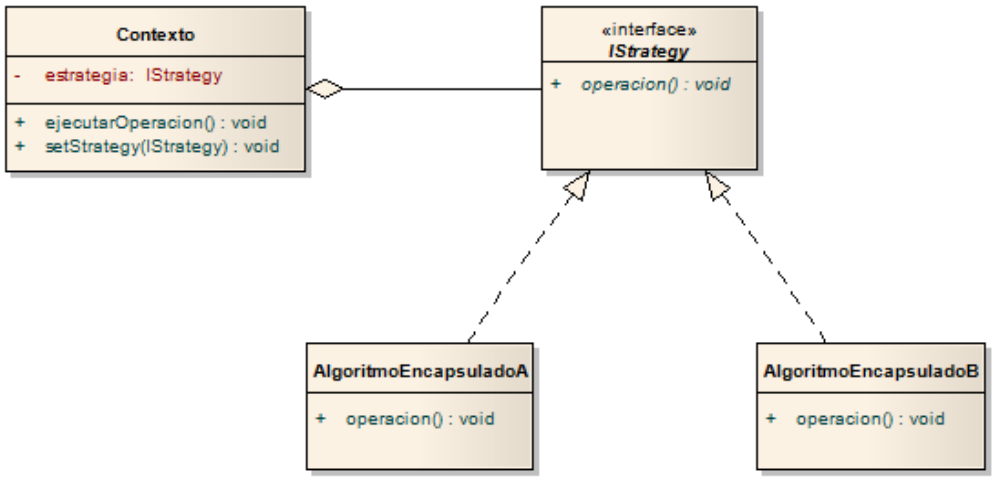
Diseño de clases.
En una aplicación compleja, antes de programar hay que diseñar las clases, es decir, detectar las clases necesarias y describir su estructura y funcionalidad básica. Así, para resolver un problema del mundo real, después de mucho pensar, decido usar un conjunto de clases concreto. Ese conjunto de clases que tengo pensado pero que aún no llegué a codificar es "mi diseño de clases". El diseño se puede hacer con notas manuscritas o con sofisticadas herramientas de diseño. Uno de los recursos de diseño de clases más utilizado son los diagramas UML que nos permiten hacer una representación gráfica de nuestro diseño de una manera standard. El diagrama de clases, que ya conocemos, es un tipo de diagrama UML.

¿Qué es un patrón de diseño?
Hay problemas, que se repiten una y otra vez. Un diseñador experto genera una solución genérica pero que se puede aplicar a muchos problemas concretos del "mismo tipo". A esa solución genérica se le llama Patrón, porque es un patrón o molde genérico a partir del que se puede generar una solución concreta. Por tanto, los patrones me permiten obtener rápidamente una estructura de clases adecuada para resolver mi problema y ahorrarte mucho tiempo *reinventando la rueda*. Los patrones de diseño se describen normalmente gráficamente con UML pero a menudo necesitan de texto adicional para su adecuada interpretación.

Patrones de diseño y principios solid
Para ver si mi diseño de clases está bien hecho le puedo "pasar el test" de los principios SOLID. Así, teniendo en cuenta que los patrones de diseño son soluciones "bien hechas", si a un patrón le pasamos "el test" de los patrones solid, veremos que efectivamente cumple todos los principios solid relevantes para la solución.

EL 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". *Design Patterns: Elements of Reusable Object-Oriented Software*



PUNTOS CLAVE:

- Estrategia (IStrategy en el ejemplo)
    - Declara la interfaz común a todos los algoritmos permitidos.
    - El contexto usa esa interfaz para llamar al algoritmo definido por una estrategia (en el ejemplo tienen una referencia de tipo IStrategy)
- Estrategia Concreta
    - Implementa el algoritmo concreto
- Contexto

  Intermediario entre estrategias y clientes. El contexto puede tener un formato variable Y típicamente, pero no necesariamente, contiene:
    - un método setStrategy() para que los clientes puedan cambiar de estrategia o algoritmo.
    - un método ejecutarOperacion() o similar que a su vez es el que invoca al algoritmo en cuestión, es decir tendrá una instrucción con el aspecto

      estrategia.operacion()

      donde estrategia es el atributo que inicializa setStrategy()

Esta estructura es utilizada por otras clases que son los *Clientes*. A menudo el contexto se puede comportar como mix de contexto-cliente.

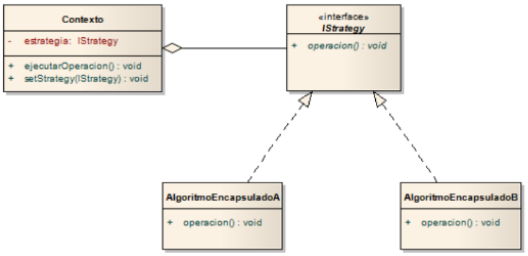
EJEMPLO:
Supón que deseo escribir un video juego en el que un personaje tiene la capacidad de atacar con armas. Las armas que usa el personaje se pueden cambiar a lo largo del juego. Las armas son espada, arco y magia. Necesitaría en un caso real muchos más detalles pero con esto es suficiente para nosotros para ejemplificar Strategy.
Mi proceso de trabajo debe ser:

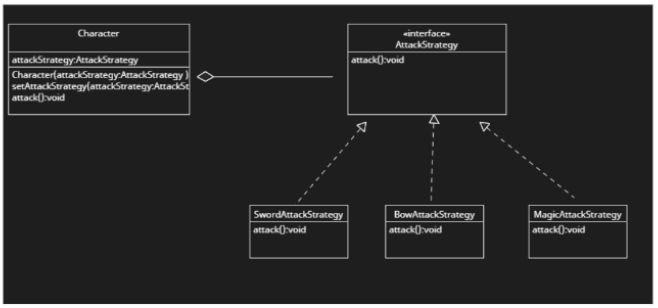
1. Diseñar la solución all problema
2. Escribir el diseño en un lenguaje de programación

Diseñar la solución al problema.
En estos momentos, debes de tener absolutamente que un patrón es "modelo de solución genérica a un tipo de problema"

En nuestro ejemplo, al comenzar a diseñar nuestra aplicación y gracias a nuestros conocimientos de patrones nos damos cuenta que aplicar el patrón Strategy es lo apropiado, esto nos permite alcanzar un diseño correcto en mucho menos tiempo

DISEÑO





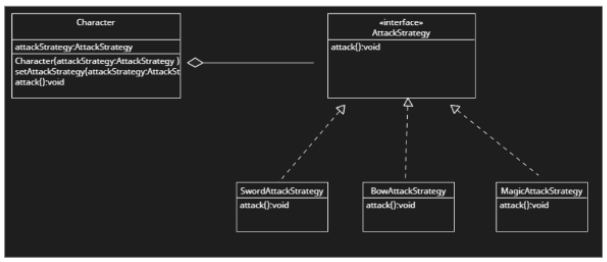
PATRÓN

APLICACIÓN DEL PATRÓN A UN CASO CONCRETO

Implementación del diseño

Quedaría implementar el diseño en un lenguaje de programación, ya más fácil y mecánico salvo que se necesite escribir complicados algoritmos. En el código ejemplo añadimos además un cliente App que usa la estructura anterior.

IMPLEMENTACIÓN DEL DISEÑO




```java
interface AttackStrategy {
    void attack();
}
class SwordAttackStrategy implements AttackStrategy {
    @Override
    public void attack() {
        System.out.println("Atacando con espada!");
    }
}

class BowAttackStrategy implements AttackStrategy {
    @Override
    public void attack() {
        System.out.println("Atacando con arco!");
    }
}

class MagicAttackStrategy implements AttackStrategy {
    @Override
    public void attack() {
        System.out.println("Atacando con magia!");
    }
}

class Character {
    private AttackStrategy attackStrategy;

    public Character(AttackStrategy attackStrategy) {
        this.attackStrategy = attackStrategy;
    }

    public void setAttackStrategy(AttackStrategy attackStrategy) {
        this.attackStrategy = attackStrategy;
    }

    public void attack() {
        this.attackStrategy.attack();
    }
}
class App{
    Run | Debug
    public static void main(String[] args) {
        Character character = new Character(new SwordAttackStrategy());
        character.attack();

        character.setAttackStrategy(new BowAttackStrategy());
        character.attack();

        character.setAttackStrategy(new MagicAttackStrategy());
        character.attack();
    }
```

```
//LA ESTRATEGIA Y SUS ESTRATEGIAS CONCRETAS
 interface AttackStrategy {
    void attack();
}
class SwordAttackStrategy implements AttackStrategy {
    @Override
    public void attack() {
        System.out.println("Atacando con espada!");
    }
}

class BowAttackStrategy implements AttackStrategy {
    @Override
    public void attack() {
        System.out.println("Atacando con arco!");
    }
}

class MagicAttackStrategy implements AttackStrategy {
    @Override
    public void attack() {
        System.out.println("Atacando con magia!");
    }
}

//EL CONTEXTO
class Character {
    private AttackStrategy attackStrategy;

    public Character(AttackStrategy attackStrategy) {
        this.attackStrategy = attackStrategy;
    }

    public void setAttackStrategy(AttackStrategy attackStrategy) {
        this.attackStrategy = attackStrategy;
    }

    public void attack() {
        this.attackStrategy.attack();
    }
}
//EL CLIENTE
class App{
    public static void main(String[] args) {
        Character character = new Character(new SwordAttackStrategy());
        character.attack();

        character.setAttackStrategy(new BowAttackStrategy());
        character.attack();

        character.setAttackStrategy(new MagicAttackStrategy());
        character.attack();
    }

}
```

STRATEGY Y LOS PRINCIPIOS DE DISEÑO SOLID

El patrón Strategy nos obliga a cumplir de forma directa dos de ellos:

- *S – Single Responsibility Principle (SRP)*. Es decir: una clase, una responsabilidad. Al encapsular cada algoritmo distinto en una clase independiente estamos cumpliendo este principio
- *O – Open/Closed Principle (OCP).* El principio de "Cerrado para modificación, abierto para extensión" Ya que los clientes no contienen en su interior el código de las estrategias, no tienen que *modificarse* cada vez que se quiere crear una nueva estrategia, pero las estrategias se pueden extender creando una nueva clase

De forma más indirecta cumple los otros, por ejemplo, el uso del interface permite que el código del contexto esté aislado de los detalles de la clases concretas cumpliendo

entonces el *principio de inversión de dependencias* que recomienda depender de abstracciones.

STRATEGY EN JDK

Un ejemplo famoso es Comparator que estudiaremos más adelante. Cada programador puede ampliar las estrategias de ordenamiento implementando Comparator. Clientes como Collections.sort() no necesitan ser modificados cada vez que queremos un nuevo método de ordenación, simplemente le indicamos la nueva estrategia por parámetro y es capaz por tanto de aceptar nuevas formas de ordenación en tiempo de ejecución.

OTRAS SITUACIONES DE USO DEL PATRÓN STRATEGY

- Cifrado de archivos: los requisitos de comportamiento pueden cambiar según el tamaño del archivo. Los archivos más pequeños se pueden cifrar directamente en la memoria, mientras que los archivos más grandes se pueden almacenar parcialmente durante el proceso de cifrado.
- Los compresores funcionan a través de estrategias (se utiliza un algoritmo distinto para comprimir en zip o en rar)
- Almacenamiento de datos: es posible que el programa solo genere en JSON actualmente, pero ¿qué sucede si descubre que también necesita generar en XML?
- A los primeros ejemplos añadimos que en general, cualquier programa capaz de almacenar y transmitir datos en distintos formatos implementarán este patrón.
- Aplicación de indicaciones: se necesita ir del punto A al punto B, pero deberá modificar su algoritmo de ruta en función de si está conduciendo un automóvil, andando en bicicleta o caminando.
- y un largo etc.

Strategy pattern y videojuegos

Como ya utilizamos más arriba, 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.