Strategy Pattern

Introducción y nombre

Strategy. De Comportamiento. Convierte algoritmos en clases y los vuelve intercambiables.

Intención

Encapsula algoritmos en clases, permitiendo que éstos sean re-utilizados e intercambiables. En base a un parámetro, que puede ser cualquier objeto, permite a una aplicación decidir en tiempo de ejecución el algoritmo que debe ejecutar.

También conocido como Policy, Estrategia.

Motivación

La esencia de este patrón es encapsular algoritmos relacionados que son subclases de una superclase común, lo que permite la selección de un algoritmo que varia según el objeto y también le permite la variación en el tiempo. Esto se define en tiempo de ejecución. Este patrón busca desacoplar bifurcaciones inmensas con algoritmos dificultosos según el camino elegido.

Solución

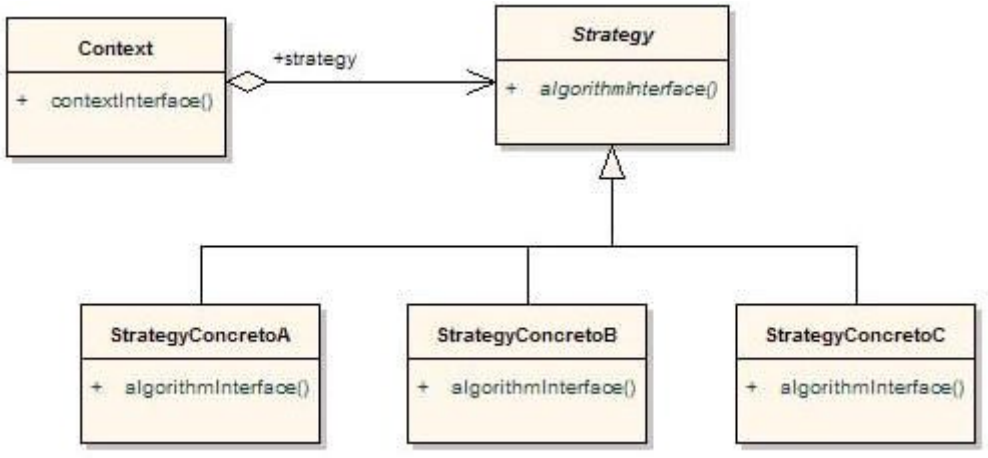
Este patrón debe utilizarse cuando:

Un programa tiene que proporcionar múltiples variantes de un algoritmo o comportamiento.

Es posible encapsular las variantes de comportamiento en clases separadas que proporcionan un modo consistente de acceder a los comportamientos.

Permite cambiar o agregar algoritmos, independientemente de la clase que lo utiliza.

Diagrama UML



Participantes

Strategy: declara una interfaz común a todos los algoritmos soportados.

StrategyConcreto: implementa un algoritmo utilizando la interfaz Strategy. Es la representación de un algoritmo.

Context: mantiene una referencia a Strategy y según las características del contexto, optará por una estrategia

determinada..

Colaboraciones

Context / Cliente: solicita un servicio a Strategy y este debe devolver el resultado de un StrategyConcreto.

Consecuencias

Permite que los comportamientos de los Clientes sean determinados dinámicamente sobre un objeto base.

Simplifica los Clientes: les reduce responsabilidad para seleccionar comportamientos o implementaciones de comportamientos alternativos. Esto simplifica el código de los objetos Cliente eliminando las expresiones if y switch.

En algunos casos, esto puede incrementar también la velocidad de los objetos Cliente porque ellos no necesitan perder tiempo seleccionado un comportamiento.

Implementación

Los distintos algoritmos se encapsulan y el cliente trabaja contra el Context. Como hemos dicho, el cliente puede elegir el algoritmo que prefiera de entre los disponibles o puede ser el mismo Context el que elija el más apropiado para cada situación.

Cualquier programa que ofrezca un servicio o función determinada, que pueda ser realizada de varias maneras, es candidato a utilizar el patrón Strategy. Puede haber cualquier número de estrategias y cualquiera de ellas podrá ser intercambiada por otra en cualquier momento, incluso en tiempo de ejecución.

Código de muestra

Supongamos un caso donde un instituto educativo tiene una lista de alumnos ordenados por promedio. Dicho instituto suele competir en torneos intercolegiales, en campeonatos nacionales y también en competencias internacionales.

Obviamente la cantidad de gente que participa en cada competencia es distinta: por ejemplo, para los campeonatos locales participan los 3 mejores promedios, pero para las competencias internacionales, sólo se envía al mejor promedio de todos.

[code]

```
public class Alumno {

   private String nombre;
   private double promedio;

   public Alumno(String nombre, double promedio) {
      this.nombre = nombre;
      this.promedio = promedio;
   }

   public String getNombre() {
      return nombre;
   }

   public void setNombre(String nombre) {
      this.nombre = nombre;
   }

   public double getPromedio() {
      return promedio;
   }

   public void setPromedio(double promedio) {
      this.promedio = promedio;
   }
```

```
}

public interface ListadoStrategy {
    public List getListado(List lista);
}

public class CompetenciaInternacional implements ListadoStrategy {
    @Override
    public List getListado(List lista) {
        List resultado = new ArrayList();
        resultado.add(lista.get(0));
        return resultado;
    }
}

public class CompetenciaNacional implements ListadoStrategy {
    @Override
    public List getListado(List lista) {
        return lista.subList(0, 3);
    }
}

public class InterColegial implements ListadoStrategy {
    @Override
    public List getListado(List lista) {
        return lista.subList(0, 5);
    }
}

public class Colegio {

    private List<Alumno> alumnos;

    public Colegio() {
        alumnos = new ArrayList<Alumno>();
        Alumno a1 = new Alumno("Juan", 10);
        Alumno a2 = new Alumno("Sebastian", 9.5);
        Alumno a3 = new Alumno("Mario", 9);
        Alumno a4 = new Alumno("Pedro", 8.5);
        Alumno a5 = new Alumno("Matias", 8);
        Alumno a6 = new Alumno("Diego", 7.8);
        alumnos.add(a1);
        alumnos.add(a2);
        alumnos.add(a3);
        alumnos.add(a4);
        alumnos.add(a5);
        alumnos.add(a6);
    }

    public List<Alumno> getAlumnos() {
        return alumnos;
    }

    public void setPersonas(List<Alumno> alumnos) {
        this.alumnos = alumnos;
    }
}

public class Main {
    public static void main(String[] args) {

        Colegio colegio = new Colegio();
        List<Alumno> alumnos = colegio.getAlumnos();

        ListadoStrategy st = new CompetenciaNacional();
        // Se puede evitar que el cliente conozca los strategy concretos
```

```
        List rta = st.getListado(alumnos);

        // Veamos el resultado del patrón
        System.out.println("Los participantes son:");
        for (int i = 0; i < rta.size(); i++) {
            Alumno alumno = (Alumno) rta.get(i);
            System.out.println(alumno.getNombre());
        }
    }
}
```

La salida por consola es:
Los participantes son:
Juan
Sebastian
Mario
[/code]

Cuándo utilizarlo

Este patrón debe ser utilizado cuando un algoritmo es cambiado según un parámetro. Imaginemos una biblioteca de un instituto educativo que presta libros a los alumnos y profesores. Imaginemos que los alumnos pueden asociarse a la biblioteca pagando una mensualidad. Con lo cual un libro puede ser prestado a Alumnos, Socios y Profesores.

Por decisión de la administración, a los socios se les prestará el libro más nuevo, luego aquel que se encuentre en buen estado y, por último, aquel que estuviese en estado regular.

En cambio, si un Alumno pide un libro, ocurre todo lo contrario. Por último, a los profesores sólo se les puede otorgar libros buenos o recién comprados.

Este caso es ideal para el patrón Strategy, ya que dependiendo de un parámetro (el tipo de persona a la que se le presta el libro) puede realizar una búsqueda con distintos algoritmos.

Patrones relacionados

Flyweight: si hay muchos objetos Cliente, los objetos StrategyConcreto puede estar mejor implementados como Flyweights.

El patrón Template Method maneja comportamientos alternativos a través de subclases más que a través de delegación.