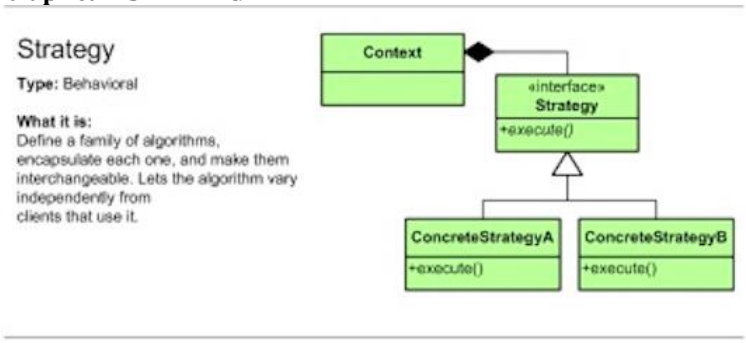
Exercici de Patrons de Disseny solucionat pas a pas

Exercici 1. Es tenen diferents maneres de fer una ordenació: el mètode de la bombolla, el mètode d'inserció, el mètode de QuickSort i el mètode de MergeSort. Es vol modelar una classe que permeti ordenar un vector d'enters segons el mètode anomenat `arrange`. Per a dissenyar aquest problema, quin patró podries fer servir? Com l'aplicaries? Escriu el diagrama de classes que dissenyaries sota aquest patró, identificant les diferents classes del patró amb el teu problema concret. Dona l'exemple d'un programa principal per a veure com utilitzaria el teu disseny. Analitza si vulneres algun principi de disseny amb la teva solució i explica per què.

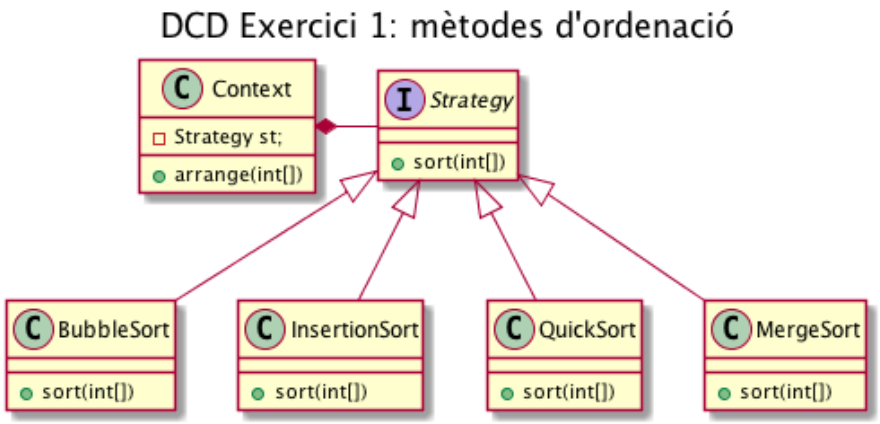
1. Problema identificat a solucionar:

El fet de tenir diferents estratègies d'ordenació que han de ser implementades des del mètode arrange() porta a un problema de modelar aquests diferents comportaments en una interfície on els diferents mètodes d'ordenació siguin les diferents implementacions.

2. Patró a aplicar: STRATEGY



3. Aplicació del patró:



4. Anàlisi del patró aplicat. (si s'escau)

En aquest cas, la classe Context té el mètode arrange() que, segons el seu atribut de tipus Strategy, cridarà a un mètode o un altre. La forma d'inicialitzar l'atribut st de la classe Context, es mitjançant

injecció de dependències en el seu constructor. Així s'evita la vulneració del principi Dependency Inversion Principle, tal i com es mostra en el llistat de sota.

```
public class Context {
        private final Strategy strategy;

        public Context(Strategy strategy) {
           this.strategy = strategy;
        }

        public void arrange(int[] input) {
            strategy.sort(input);
        }
}
```

Serà el Client del Context qui farà la tria de quin mètode cal utilitzar i en tot cas, serà ell qui vulnerarà el principi Open-Close.

5. Programa principal que mostra l'ús del patró utilitzat.

```
public class Test {

   public static void main(String args[]) {

        // we can provide any strategy to do the sorting
        int[] var = {1, 2, 3, 4, 5 };
        Context ctx = new Context(new BubbleSort());
        ctx.arrange(var);

        // we can change the strategy without changing Context class
        ctx = new Context(new QuickSort());
        ctx.arrange(var);
   }
}
```

Aquí només se'n dóna un exemple d'ús sense tenir en compte tots els casos.

6. Observacions addicionals

Si es tingués el Client que contempla tots els casos:

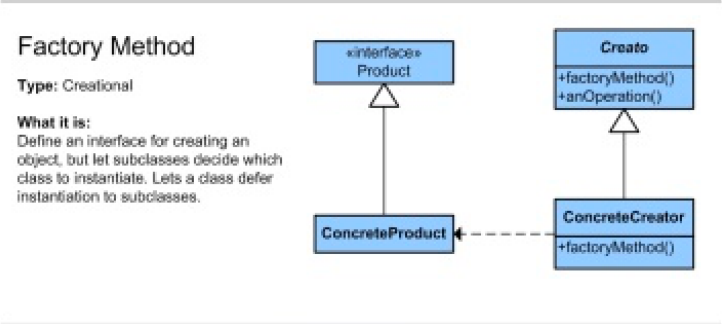
```
public class Client {

       public void order (SortType type, int[] var) {
          Context ctx;
         // SortType is an enum {BubbleSortType, InsertionSortType, QuicksortType,
MergeSortType}
          switch (type) {
            case BubbleSortType:
                 ctx = new Context(new BubbleSort());
                 break;
            case InsertionSortType:
                 ctx = new Context(new InsertionSort());
                 break;
            case QuickSortType:
                 ctx = new Context(new QuickSort());
```

```
                break;
           case MergeSortType:
                ctx = new Context(new MergeSort());
                break;
           default:
                break;
         }
        ctx.arrange(var);
        }
}
```

Es tindria el problema de vulnerabilitat de l'Open-Close Principle, ja que si es volguessin afegir més mètodes, aquí caldria canviar el codi.

Cal observar que s'està davant d'un problema llavors de construcció de les classes i podríem aplicar el patró de FactoryMethod per a solucionar-ho. De fet el Product és la classe Strategy i els concreteProduct seran les diferents classes de les ordenacions. Per a acabar d'aplicar bé el patró de FactoryMethod caldria definir una classe abstracte addicional (Factory) i la seva filla StrategyFactory.




```
public abstract class Factory {

        public void order (SortType type, int[] var) {

                Context ctx;
                Strategy st  = createStrategy(type);

                ctx.arrange(var);
        }
        public abstract Strategy createSetrategy(SortType type);
}
```

```
public class StrategyFactory implements Creator {
        public Strategy createStrategy(SortType type) {
          switch (type) {
                case BubbleSortType:
                ctx = new Context(new BubbleSort());
                break;
           case InsertionSortType:
                ctx = new Context(new InsertionSort());
                break;
           case QuickSortType:
                ctx = new Context(new QuickSort());
                break;
           case MergeSortType:
                ctx = new Context(new MergeSort());
```

```
                break;
            default:
                break;
          }
        }
}
```