UNIVERSITE SAINT-JOSEPH
FACULTE D’INGENIERIE

DESIGN PATTERNS LAB

Session 4 – Duration 45 minutes

In this lab, you'll explore 2 ways to apply the Strategy Design Pattern using a simple sorting example.
For each approach, copy the source code into Notepad, compile execute and explain how the array to sort was passed to the strategy.

Approach 1: The Context Passes Data to the Strategy

In this approach, the context class (Sorter) passes the data (the array to be sorted) explicitly to the strategy.

```
// Define the SortingStrategy interface
interface SortingStrategy {
   void sort(int[] array);
}

// Implement concrete sorting strategies
class BubbleSort implements SortingStrategy {
   public void sort(int[] array) {
      System.out.println("Sorting using Bubble Sort");
      // Implementation of Bubble Sort
      // ...
   }
}

class QuickSort implements SortingStrategy {
   public void sort(int[] array) {
      System.out.println("Sorting using Quick Sort");
      // Implementation of Quick Sort
      // ...
   }
}

// Context class that passes data to the selected sorting strategy
class Sorter {
   private SortingStrategy strategy;

   public Sorter(SortingStrategy strategy) {
      this.strategy = strategy;
   }
```

```
    public void setStrategy(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void performSort(int[] array) {
        strategy.sort(array);
    }
}

public class StrategyPatternExample {
    public static void main(String[] args) {
        int[] numbers = {5, 1, 4, 2, 8};

        // Create sorting strategies
        SortingStrategy bubbleSort = new BubbleSort();
        SortingStrategy quickSort = new QuickSort();

        // Create a sorter with the BubbleSort strategy
        Sorter sorter = new Sorter(bubbleSort);

        // Perform a sort using the BubbleSort strategy
        sorter.performSort(numbers);

        // Switch to the QuickSort strategy and perform another sort
        sorter.setStrategy(quickSort);
        sorter.performSort(numbers);
    }
}
```

In this approach:

We define the SortingStrategy interface with a sort method.

We implement two concrete sorting strategies, BubbleSort and QuickSort, each implementing the sort method with their respective sorting algorithms.

The Sorter class is the context class that uses the selected sorting strategy. It has a reference to the current strategy and a method performSort to execute the sorting.

In the main method, we create an array of numbers and two sorting strategies (BubbleSort and QuickSort). We initialize the Sorter with the BubbleSort strategy, perform a sort, and then switch to the QuickSort strategy and perform another sort.

This example demonstrates how you can easily switch between different sorting strategies using the Strategy pattern without changing the client code.

Approach 2: The Strategy Retains a Reference to the Context

In this approach, the context class (Sorter) is passed to the sorting strategy during its creation, and the strategy retains a reference to the context for accessing the data.

```
// Define the SortingStrategy interface
interface SortingStrategy {
    void sort();
}

// Implement concrete sorting strategies
class BubbleSort implements SortingStrategy {
    private Sorter context;

    public BubbleSort(Sorter context) {
        this.context = context;
    }

    public void sort() {
        int[] array = context.getData();
        System.out.println("Sorting using Bubble Sort");
        // Implementation of Bubble Sort using context.getData()
        // ...
    }
}

class QuickSort implements SortingStrategy {
    private Sorter context;

    public QuickSort(Sorter context) {
        this.context = context;
    }

    public void sort() {
        int[] array = context.getData();
        System.out.println("Sorting using Quick Sort");
        // Implementation of Quick Sort using context.getData()
        // ...
    }
}

// Context class that retains data and passes itself to the selected sorting strategy
class Sorter {
    private SortingStrategy strategy;
    private int[] data;
    public Sorter(int[] data) {
        this.data = data;
    }

    public void setStrategy(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public int[] getData() {
        return data;
    }

    public void performSort() {
        strategy.sort();
    }
}

public class StrategyPatternExample {
    public static void main(String[] args) {
        int[] numbers = {5, 1, 4, 2, 8};

        // Create a sorter with the BubbleSort strategy
        Sorter sorter = new Sorter(numbers);
        SortingStrategy bubbleSort = new BubbleSort(sorter);
        sorter.setStrategy(bubbleSort);

        // Perform a sort using the BubbleSort strategy
        sorter.performSort();

        // Switch to the QuickSort strategy and perform another sort
        SortingStrategy quickSort = new QuickSort(sorter);
        sorter.setStrategy(quickSort);
        sorter.performSort();
    }
}
```

These two approaches demonstrate different ways to implement the Strategy pattern, depending on whether the context class passes data explicitly or the strategy retains a reference to the context. Both approaches adhere to the Strategy pattern and allow you to switch between sorting strategies without modifying the client code.