# SOFTWARE ARCHIETECTURE AND DESIGHN PATTERN

GROUP ACTIVITY

GROUP NO - 2

11- VAISHNAVI DATIR

23- PALLAVI INGOLE

46-TANMAY PAWAR

50 – VINAY SANAS

52 – ADESH SHELAR

- **CASE STUDY**

  CREATE A DATA MINING AGENT FRAMEWORK WHICH HAS CLIENT SERVER ARCHITECTURE.

  THERE WILL BE SORTING ALGORITHMS, BUBBLE SORT AND QUICKSORT, ARE
  IMPLEMENTED AND THE CLIENT CAN SELECT EITHER OF THE ALGORITHMS. USE APPROPRIATE DESIGN PATTERNS AND EXPLAIN WHY IT IS USED.

## DESIGN PATTERN USED IN CASE STUDY

- Design Pattern : Strategy Pattern
- The Strategy Pattern is suitable for this scenario because it allows you to define a family of algorithms, encapsulate each one of them, and make them interchangeable. In your case, the sorting algorithms (Bubble sort and Quicksort) are interchangeable strategies
- The Strategy pattern encapsulates the sorting algorithm behavior into separate classes. This makes it easy to switch between different sorting algorithms without modifying the client code. It also promotes code reusability and maintainability.

## • HOW CHOSEN DESIGN PATTERN IS APPLICABLE FOR THE CASE STUDY?

- The chosen design pattern, the Strategy Pattern, is highly applicable to the case study of developing a data mining agent framework with sorting algorithms (Bubble Sort and QuickSort).

1. Encapsulation of Algorithms : The Strategy Pattern allows you to encapsulate different sorting algorithms (Bubble Sort and QuickSort) into separate classes that implement a common interface (Sorting Algorithm Interface).
2. Interchangeable Algorithms : Data mining applications often require the flexibility to switch between sorting algorithms based on factors like dataset size, data distribution, and performance requirements.
3. Code Reusability : With the Strategy Pattern, you can create a library of sorting algorithms that can be reused across different data mining projects or even in other applications .
4. Scalability : Data mining often deals with large datasets, and the performance of sorting algorithms is crucial.

4. Flexibility in Client Choice : Clients of the data mining agent can choose the sorting algorithm that best suits their specific data and requirements. This flexibility is essential for adapting to varying data characteristics .
5. Maintainability and Readability : The Strategy Pattern enhances code maintainability and readability by making the sorting algorithm selection and execution explicit and modular.

In summary, the Strategy Pattern is an excellent choice for the data mining agent framework because it offers flexibility, maintainability, extensibility, and clean separation of concerns. It allows you to create a scalable and adaptable system capable of handling different sorting algorithms while ensuring that the core functionality of the data mining agent remains intact.

## • INTENT OF STRATEGY PATTERN

The primary intent of the Strategy Pattern is to:

- Define a set of algorithms, encapsulate each algorithm in a separate class, and make these algorithms interchangeable.
- Allow clients to choose an appropriate algorithm dynamically without changing the client's code.
- Promote code reusability by isolating algorithm-specific code from the context in which it is used.
- Enable a clean separation of concerns by decoupling the algorithm implementation from the client code

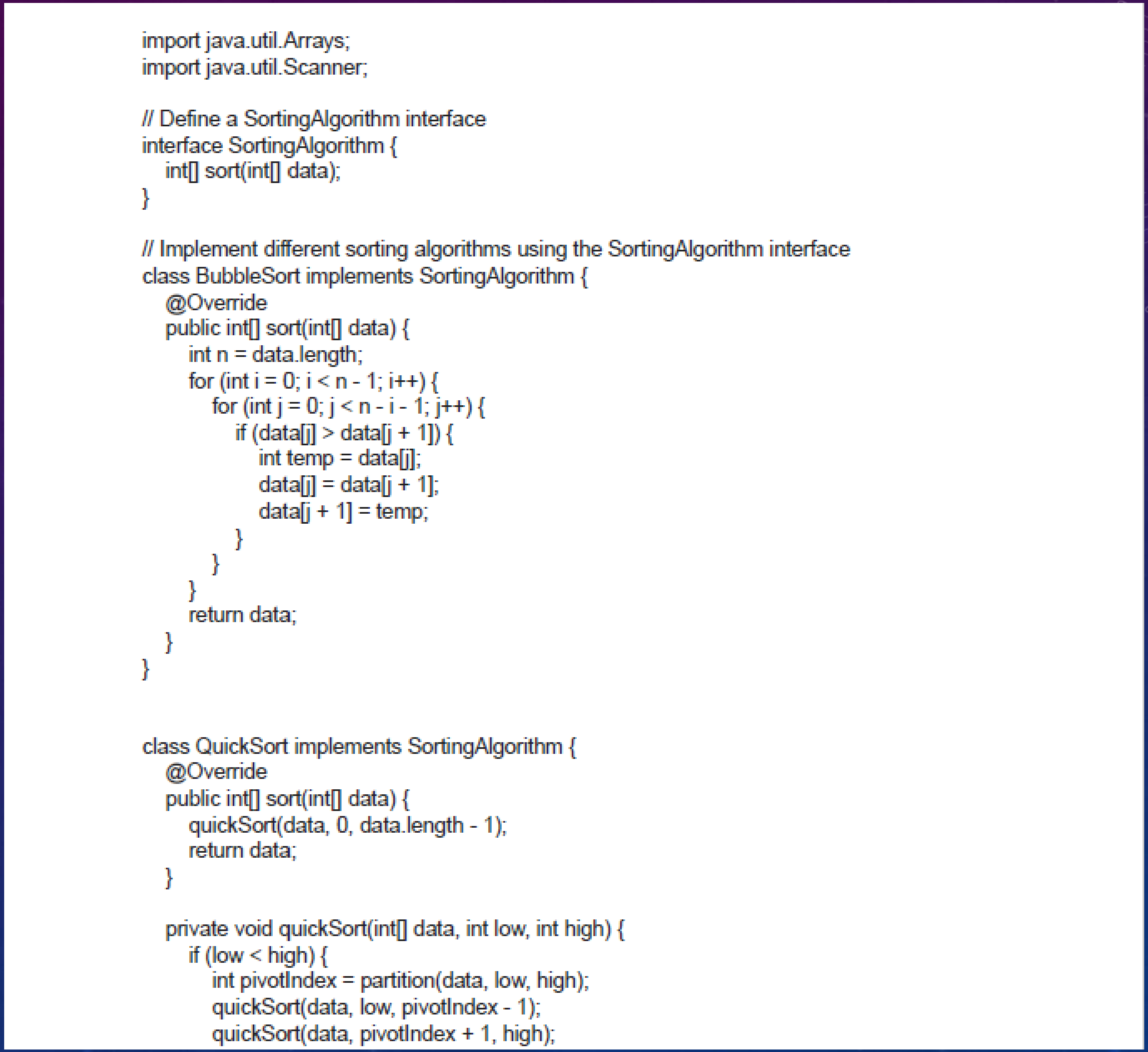
## • STRUCTURE OF STRATEGY PATTERN

1. **Context:**
   - The Context class maintains a reference to the selected strategy (algorithm).
   - It uses this reference to delegate the algorithm's execution to the concrete strategy class.
2. **Strategy (Algorithm Interface):**
   - The Strategy interface defines a common set of methods or an abstract class that represents the algorithm's contract.
   - Concrete strategy classes implement this interface or extend the abstract class to provide specific algorithm implementations.
3. **Concrete Strategies:**
   - Concrete strategy classes implement the Strategy interface or extend the Strategy abstract class.
   - Each concrete strategy class represents a specific algorithm.

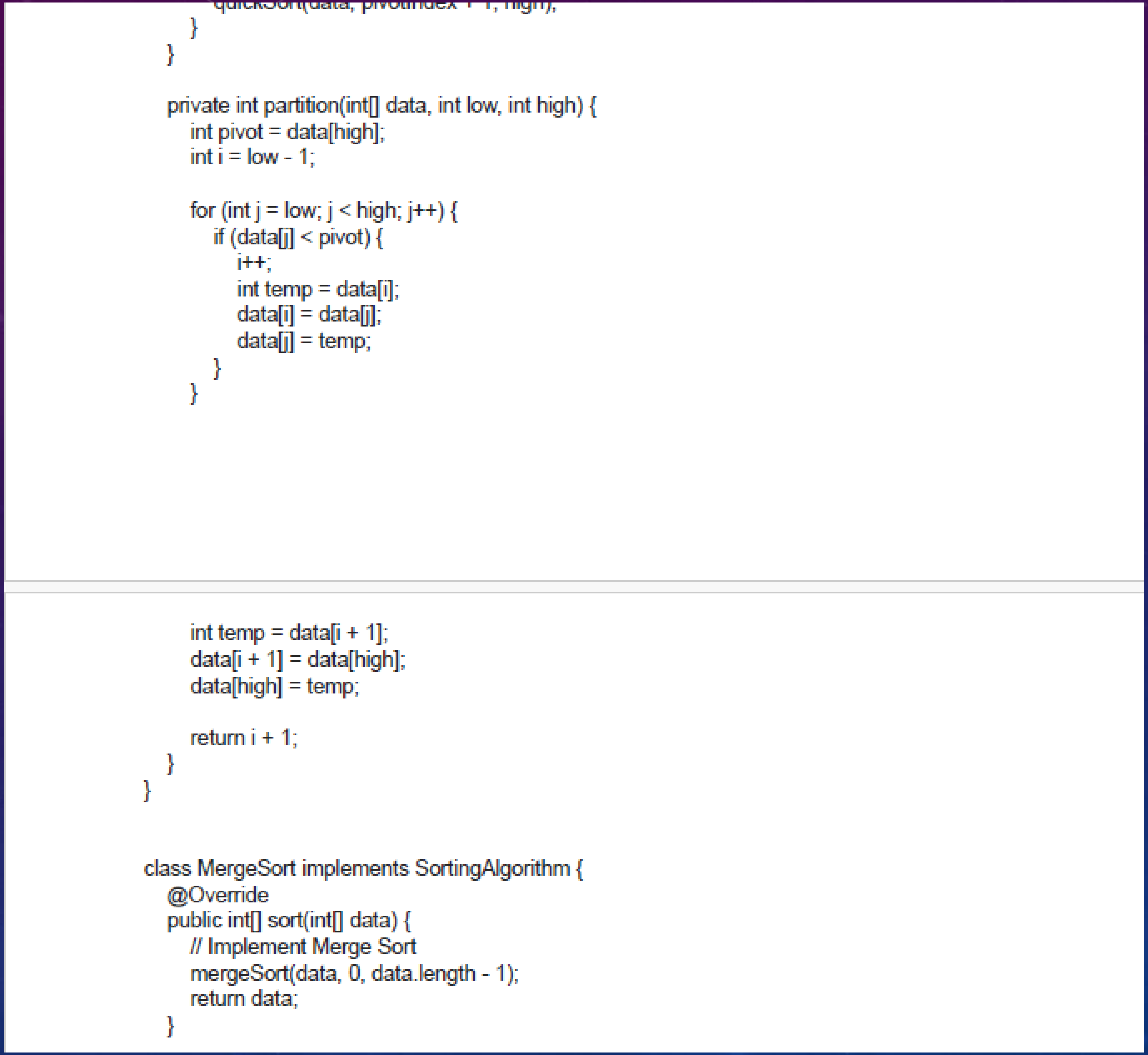
## • PARTICIPANTS OF STRATEGY PATTERN

1. **Client:**
    - The Client is responsible for creating a Context object and selecting the appropriate strategy (algorithm) dynamically.
    - The Client interacts with the Context to perform operations without having to know the details of the algorithms.
2. **Context:**
    - The Context maintains a reference to a Strategy object.
    - It provides an interface for clients to set the current strategy and execute the algorithm.
3. **Strategy (Algorithm Interface):**
    - The Strategy interface defines the methods that concrete strategies must implement.
    - It establishes a common contract for all concrete strategy classes, ensuring that they have the same set of methods.
4. **Concrete Strategies:**
    - Concrete strategy classes implement the Strategy interface or extend the Strategy abstract class.
    - Each concrete strategy encapsulates a specific algorithm and provides the implementation details.

# CODE

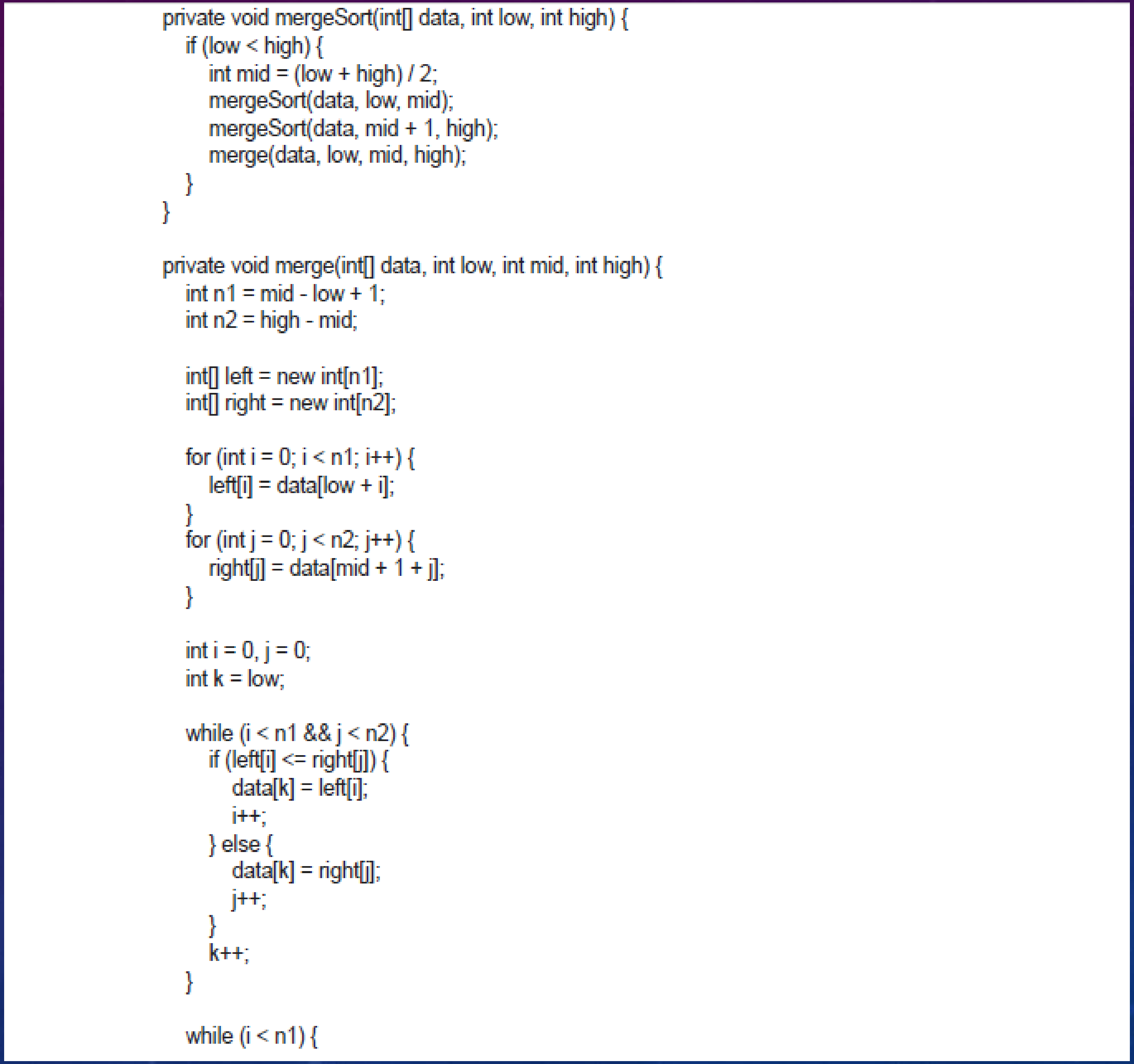


```
import java.util.Arrays;
import java.util.Scanner;

// Define a SortingAlgorithm interface
interface SortingAlgorithm {
    int[] sort(int[] data);
}

// Implement different sorting algorithms using the SortingAlgorithm interface
class BubbleSort implements SortingAlgorithm {
    @Override
    public int[] sort(int[] data) {
        int n = data.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (data[j] > data[j + 1]) {
                    int temp = data[j];
                    data[j] = data[j + 1];
                    data[j + 1] = temp;
                }
            }
        }
        return data;
    }
}


class QuickSort implements SortingAlgorithm {
    @Override
    public int[] sort(int[] data) {
        quickSort(data, 0, data.length - 1);
        return data;
    }

    private void quickSort(int[] data, int low, int high) {
        if (low < high) {
            int pivotIndex = partition(data, low, high);
            quickSort(data, low, pivotIndex - 1);
            quickSort(data, pivotIndex + 1, high);
```

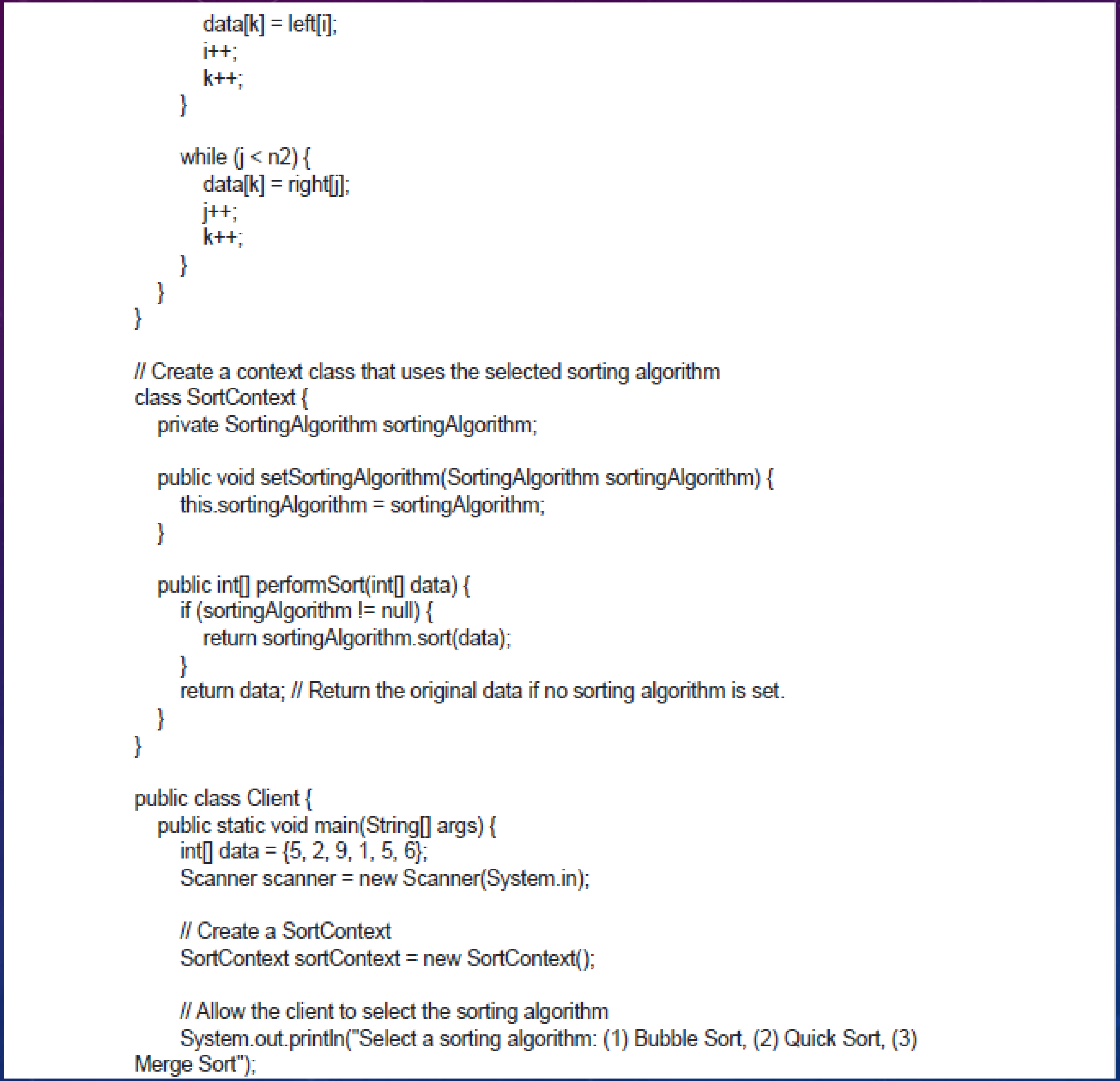


```
            quickSort(data, pivotIndex + 1, high);
        }
    }

    private int partition(int[] data, int low, int high) {
        int pivot = data[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (data[j] < pivot) {
                i++;
                int temp = data[i];
                data[i] = data[j];
                data[j] = temp;
            }
        }
```

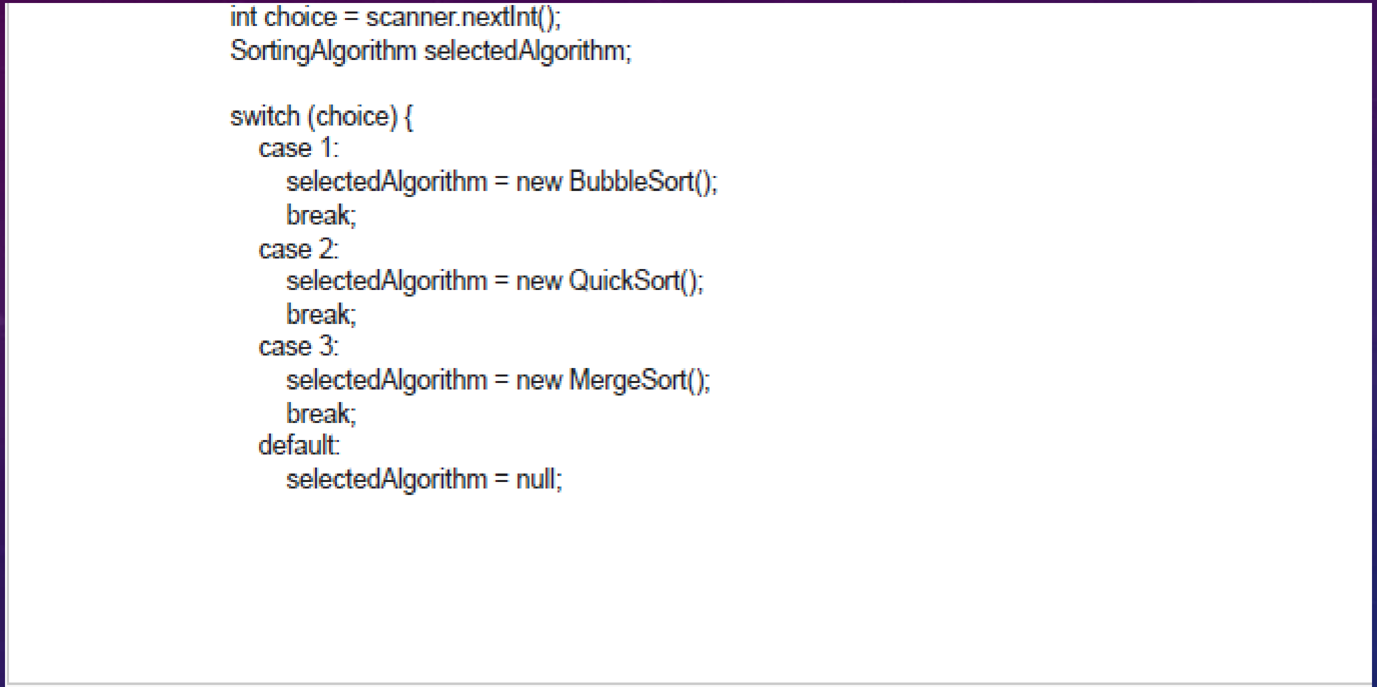
```
        int temp = data[i + 1];
        data[i + 1] = data[high];
        data[high] = temp;

        return i + 1;
    }
}


class MergeSort implements SortingAlgorithm {
    @Override
    public int[] sort(int[] data) {
        // Implement Merge Sort
        mergeSort(data, 0, data.length - 1);
        return data;
    }
```



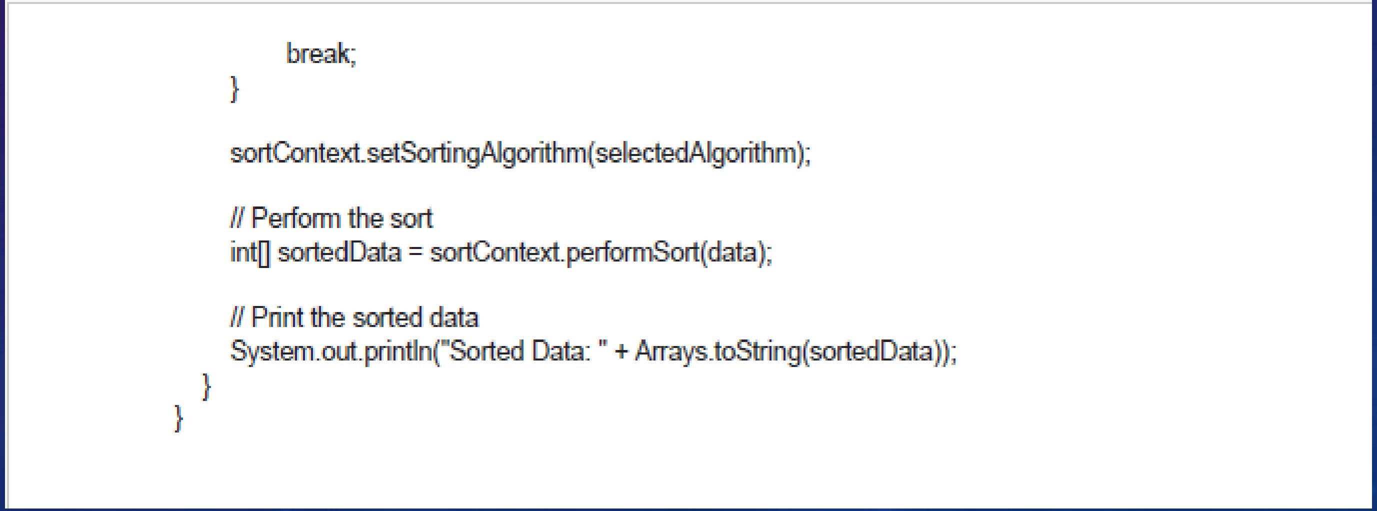
```
private void mergeSort(int[] data, int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
        mergeSort(data, low, mid);
        mergeSort(data, mid + 1, high);
        merge(data, low, mid, high);
    }
}

private void merge(int[] data, int low, int mid, int high) {
    int n1 = mid - low + 1;
    int n2 = high - mid;

    int[] left = new int[n1];
    int[] right = new int[n2];

    for (int i = 0; i < n1; i++) {
        left[i] = data[low + i];
    }
    for (int j = 0; j < n2; j++) {
        right[j] = data[mid + 1 + j];
    }

    int i = 0, j = 0;
    int k = low;

    while (i < n1 && j < n2) {
        if (left[i] <= right[j]) {
            data[k] = left[i];
            i++;
        } else {
            data[k] = right[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
```



```
            data[k] = left[i];
            i++;
            k++;
        }

        while (j < n2) {
            data[k] = right[j];
            j++;
            k++;
        }
    }
}

// Create a context class that uses the selected sorting algorithm
class SortContext {
    private SortingAlgorithm sortingAlgorithm;

    public void setSortingAlgorithm(SortingAlgorithm sortingAlgorithm) {
        this.sortingAlgorithm = sortingAlgorithm;
    }

    public int[] performSort(int[] data) {
        if (sortingAlgorithm != null) {
            return sortingAlgorithm.sort(data);
        }
        return data; // Return the original data if no sorting algorithm is set.
    }
}

public class Client {
    public static void main(String[] args) {
        int[] data = {5, 2, 9, 1, 5, 6};
        Scanner scanner = new Scanner(System.in);

        // Create a SortContext
        SortContext sortContext = new SortContext();

        // Allow the client to select the sorting algorithm
        System.out.println("Select a sorting algorithm: (1) Bubble Sort, (2) Quick Sort, (3)
Merge Sort");
```

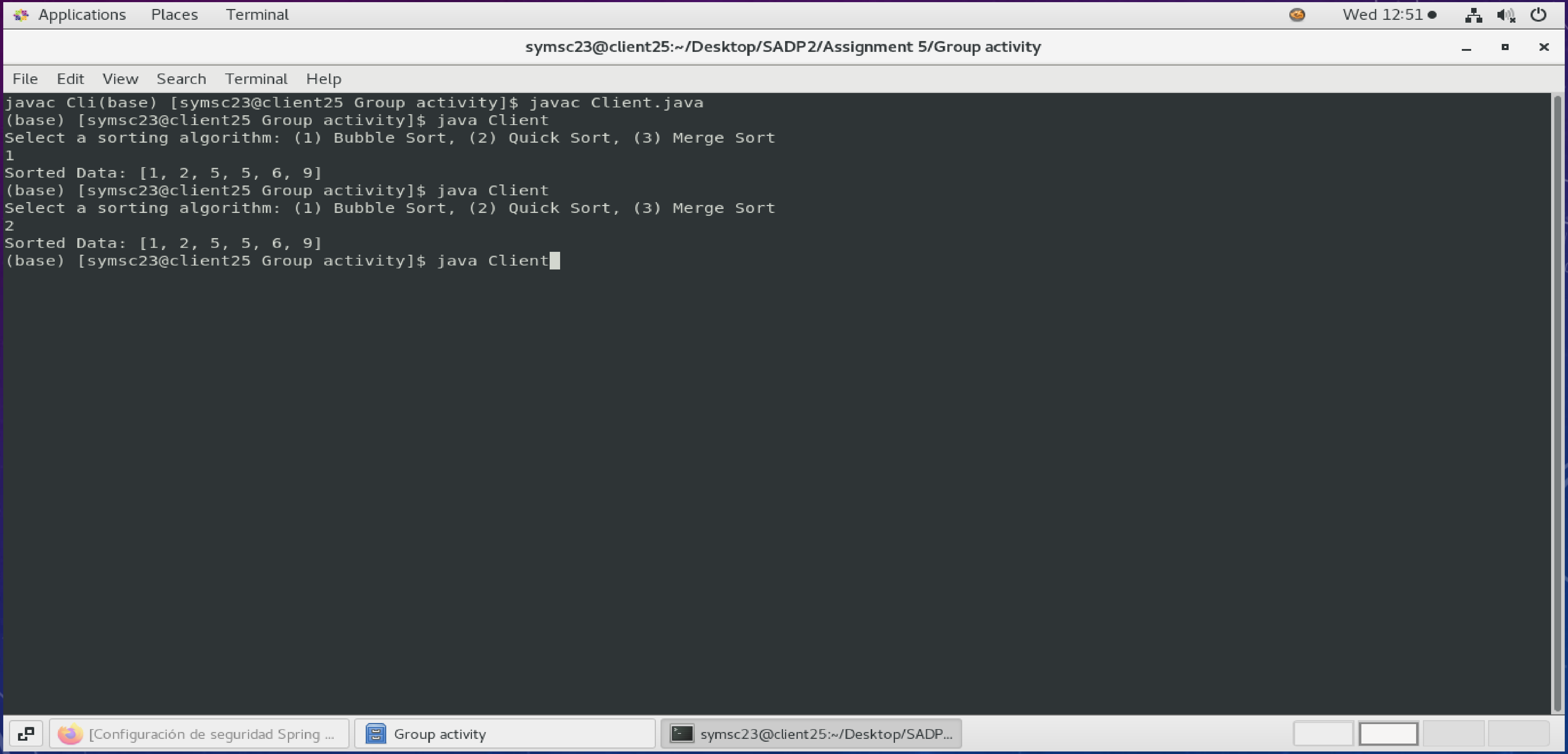


```
        int choice = scanner.nextInt();
        SortingAlgorithm selectedAlgorithm;

        switch (choice) {
          case 1:
            selectedAlgorithm = new BubbleSort();
            break;
          case 2:
            selectedAlgorithm = new QuickSort();
            break;
          case 3:
            selectedAlgorithm = new MergeSort();
            break;
          default:
            selectedAlgorithm = null;
```



```
            break;
        }

        sortContext.setSortingAlgorithm(selectedAlgorithm);

        // Perform the sort
        int[] sortedData = sortContext.performSort(data);

        // Print the sorted data
        System.out.println("Sorted Data: " + Arrays.toString(sortedData));
      }
    }
```

## OUTPUT


Applications
Places
Terminal
Wed 12:51
symsc23@client25:~/Desktop/SADP2/Assignment 5/Group activity
File
Edit
View
Search
Terminal
Help
[Configuración de seguridad Spring ...
Group activity
symsc23@client25:~/Desktop/SADP...

```
javac Cli(base) [symsc23@client25 Group activity]$ javac Client.java
(base) [symsc23@client25 Group activity]$ java Client
Select a sorting algorithm: (1) Bubble Sort, (2) Quick Sort, (3) Merge Sort
1
Sorted Data: [1, 2, 5, 5, 6, 9]
(base) [symsc23@client25 Group activity]$ java Client
Select a sorting algorithm: (1) Bubble Sort, (2) Quick Sort, (3) Merge Sort
2
Sorted Data: [1, 2, 5, 5, 6, 9]
(base) [symsc23@client25 Group activity]$ java Client
```

The Strategy Pattern separates the algorithms from the clients that use them, promoting flexibility, code reusability, and a clean separation of concerns. Clients can choose from a set of interchangeable strategies without modifying their code, and the strategies themselves can be extended or added easily, making it a powerful pattern for managing algorithmic variations.

# THANK YOU