Design Patterns Practical Session Report

Younes Menfalouti

November 23, 2025

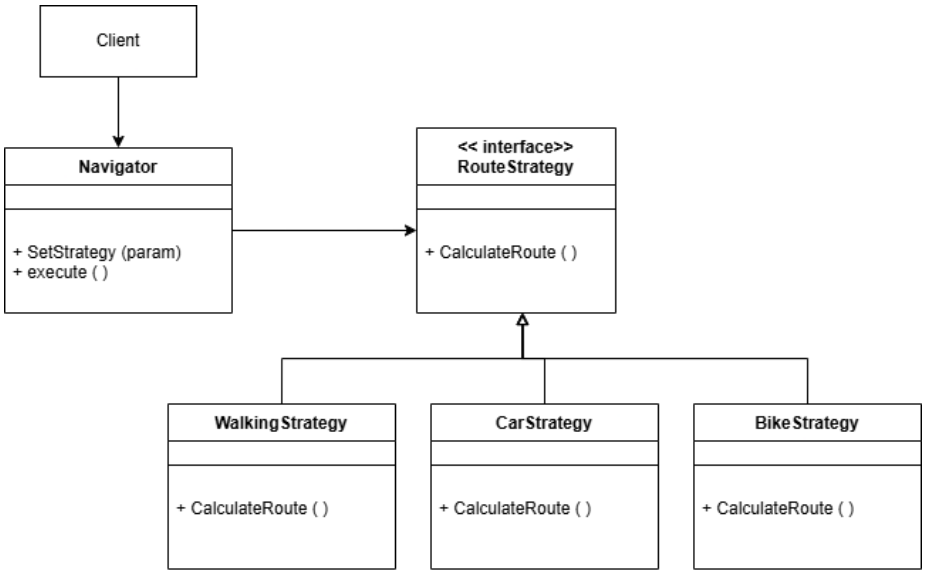
0.1 Introduction

This report documents the implementation of four design patterns in Java: Observer, Strategy, Composite, and Adapter. Each pattern is showcased in a separate package with a main class demonstrating its usage.

0.2 Strategy Pattern - Navigator

The Strategy pattern allows selecting different route calculation strategies.

0.2.1 Class Diagram



0.2.2 Code Implementation

0.2.3 Output Screenshot

```
public class Navigator {

    public void setRouteStrategy(int choice) {  1 usage
        switch (choice) {
            case 1:
                this.routeStrategy = new WalkingStrategy();
                break;
            case 2:
                this.routeStrategy = new CarStrategy();
                break;
            default:
                System.out.println("Invalid choice");
                break;
        }
```

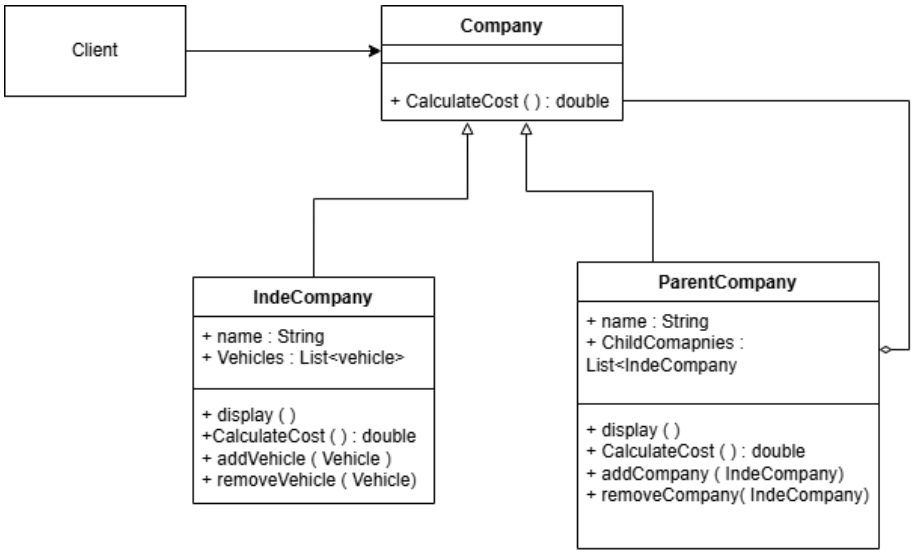
Navigator

```
"C:\Program Files\Java\jdk-25\bin\java.exe" "-javaagent:C:\Program Files\JetBrains
Welcome to the Route Strategy
Please enter the Route Strategy:
1) Walking
2) Car
1
Enter Route Distance in km
100
Navigator executing...
Route Strategy: WalkingStrategy
Time : 12.5 hours

Process finished with exit code 0
```

0.3 Composite Pattern - Vehicle Sales

The Composite pattern treats individual companies and groups uniformly.

0.3.1 Class Diagram



0.3.2 Code Implementation

0.3.3 Output Screenshot

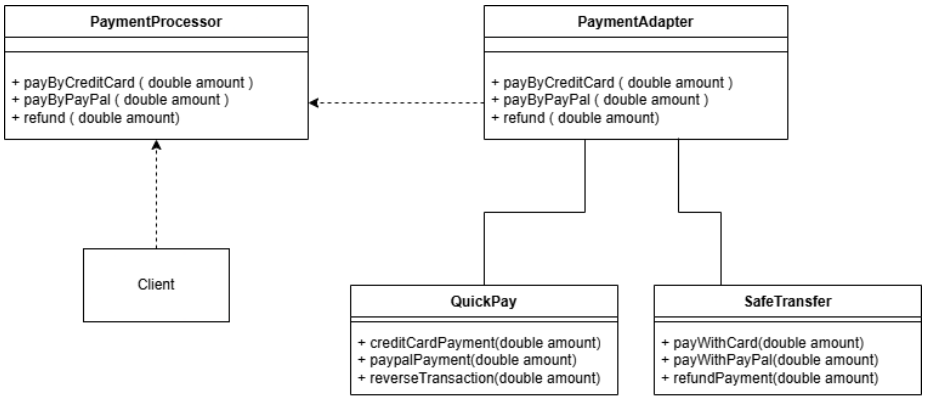
```java
package Software_Eng_TPs.Design_PatterTP.Vehicle_sales;

public class VehicleSalesDemo {
    public static void main(String[] args) {
        IndeCompany company1 = new IndeCompany( name: "Toyota Dealers");
        company1.addVehicle(new Vehicle( name: "Camry",  maintainanceCost: 500));
        company1.addVehicle(new Vehicle( name: "Corolla",  maintainanceCost: 400));

        IndeCompany company2 = new IndeCompany( name: "Honda Dealers");
        company2.addVehicle(new Vehicle( name: "Civic",  maintainanceCost: 450));
        company2.addVehicle(new Vehicle( name: "Accord",  maintainanceCost: 550));


        ParentCompany parent = new ParentCompany( name: "Auto Group");
        parent.addCompany(company1);
        parent.addCompany(company2);

        parent.display();

        System.out.println("Total maintenance cost: " + parent.CalculateCost());
    }
}
```

```
"C:\Program Files\Java\jdk-25\bin\java.exe" "-javaagent:C:\Program Files\JetBrains
Name: Auto Group
Companies:
Company Name: Toyota Dealers
Vehicles:
Camry
Corolla
Company Name: Honda Dealers
Vehicles:
Civic
Accord
Total maintenance cost: 0.0

Process finished with exit code 0
```

0.4 Adapter Pattern - E-commerce App

The Adapter pattern allows incompatible interfaces to work together.

0.4.1 Class Diagram



0.4.2 Code Implementation

0.4.3 Output Screenshot

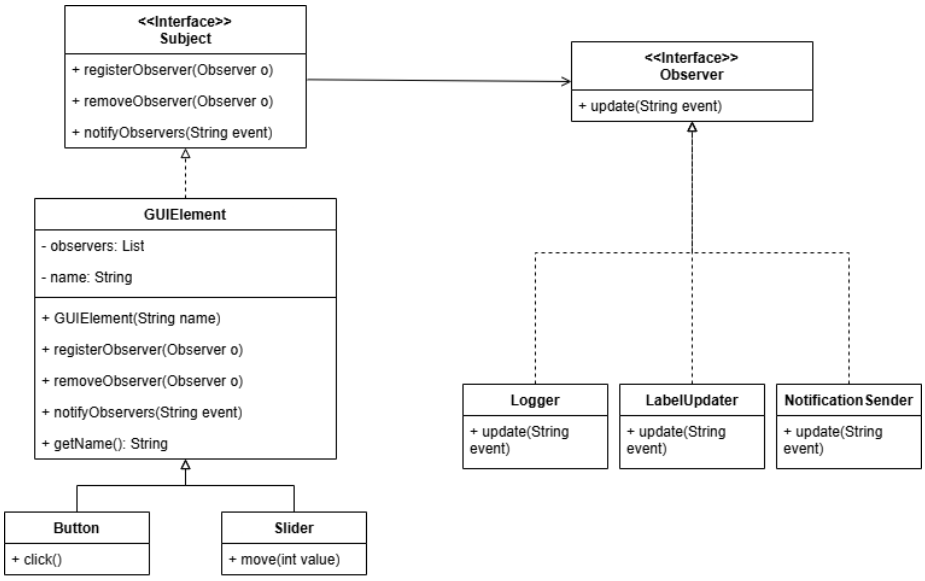
```
package Software_Eng_TPs.Design_PatterTP.e_commerceApp;

public class ECommerceDemo {
    public static void main(String[] args) {
        PaymentAdapter adapter = new PaymentAdapter();

        adapter.SetPaymentMethod(new QuickPay());
        adapter.payByCreditCard( amount: 100.0);


        adapter.SetPaymentMethod(new SafeTransfer());
        adapter.payByCreditCard( amount: 200.0);
    }
}
```

```
ECommerceDemo ×

"C:\Program Files\Java\jdk-25\bin\java.exe" "-javaagent:C:\Program Files\JetBrains
 QuickPay : Processing credit card payment $ 100.0
 SafeTransfer : Paying with credit card $ 200.0

Process finished with exit code 0
```

0.5 Observer Pattern - GUI Dashboard

The Observer pattern is used to notify multiple components when GUI elements change.

0.5.1 Class Diagram



0.5.2 Code Implementation

0.5.3 Output Screenshot

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

        Logger logger = new Logger();
        LabelUpdater labelUpdater = new LabelUpdater();
        NotificationSender notificationSender = new NotificationSender();


        Button submitButton = new Button( name: "SubmitButton");
        Slider volumeSlider = new Slider( name: "VolumeSlider");

        submitButton.registerObserver(logger);
        submitButton.registerObserver(labelUpdater);


        volumeSlider.registerObserver(logger);
        volumeSlider.registerObserver(notificationSender);

        System.out.println("--- Dashboard Initialized. Simulating user actions. ---");


        submitButton.click();
        volumeSlider.move( value: 75);


        System.out.println("\n--- Disabling notifications for the volume slider. ---");
        volumeSlider.removeObserver(notificationSender);
        volumeSlider.move( value: 50);
    }
```

```java
package Software_Eng_TPs.Design_PatterTP.GUI_dashboard;

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

        Logger logger = new Logger();
        LabelUpdater labelUpdater = new LabelUpdater();
        NotificationSender notificationSender = new NotificationSender();


```

Run Dashboard

```
"C:\Program Files\Java\jdk-25\bin\java.exe" "-javaagent:C:\Program Files\JetBrains
--- Dashboard Initialized. Simulating user actions. ---

--- User Action: Clicked SubmitButton ---
Logger: Logging user interaction SubmitButton: Clicked
LabelUpdater: Updating GUI label with last action  SubmitButton: Clicked

--- User Action: Moved VolumeSlider ---
Logger: Logging user interaction VolumeSlider: Moved to value 75
NotificationSender: Sending alert for VolumeSlider: Moved to value 75

--- Disabling notifications for the volume slider. ---

--- User Action: Moved VolumeSlider ---
Logger: Logging user interaction VolumeSlider: Moved to value 50
```

0.6 Conclusion

This practical session demonstrated the application of design patterns to solve common software design problems.