Design Patterns – TP Report

Ait Aadi Youssef

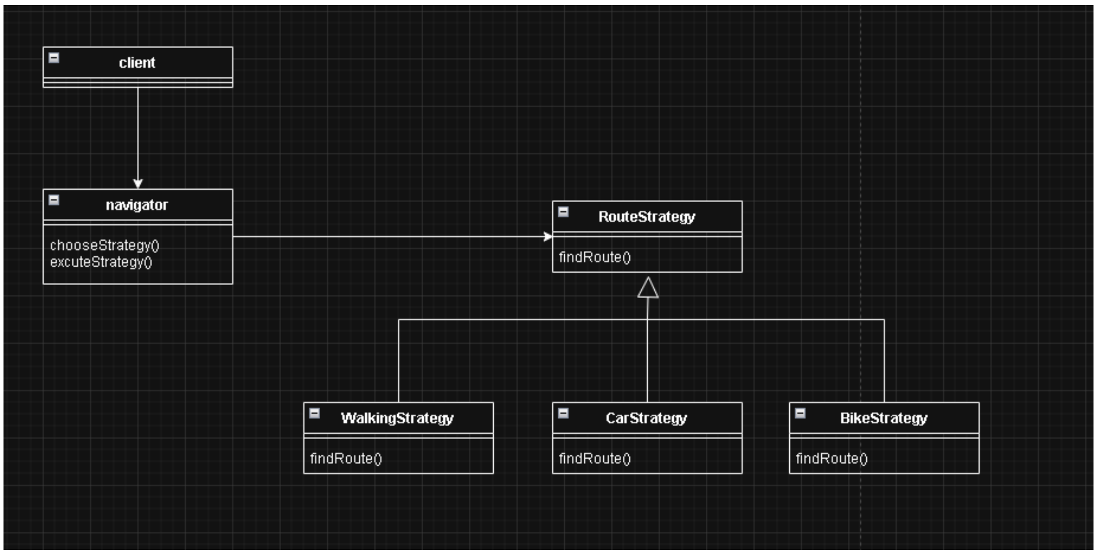
UM6P – College of Computing

November 27, 2025

Contents

1 Exercise 1 – Strategy Pattern

Class Diagram



Answers

1. Role of the Navigator The Navigator is the **Context**. It holds a reference to a routing strategy and delegates route computation to it.

2. Why Navigator depends on RouteStrategy To depend on the abstraction rather than concrete strategies, allowing runtime flexibility and respecting the Strategy Pattern structure.

3. Applied SOLID principles SRP, OCP, DIP.

Java Code

RouteStrategy Interface

```java
public interface RouteStrategy {
    void buildRoute(String start, String end);
}
```

Concrete Strategies

```java
public class WalkingStrategy implements RouteStrategy {
    @Override
    public void buildRoute(String start, String end) {
        System.out.println("Computing walking route...");
    }
}
```

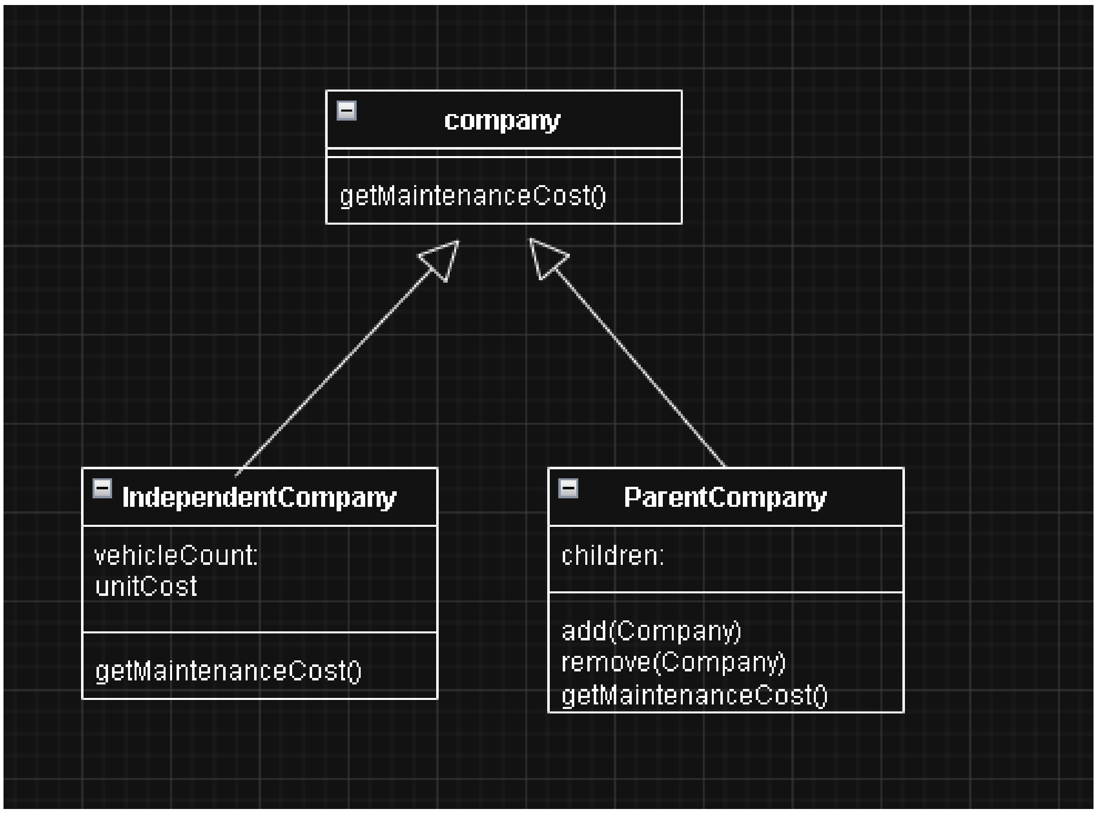
```java
public class CarStrategy implements RouteStrategy {
    @Override
    public void buildRoute(String start, String end) {
        System.out.println("Computing car route...");
    }
}
```

Navigator

```java
public class Navigator {
    private RouteStrategy strategy;

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

    public void navigate(String start, String end) {
        strategy.buildRoute(start, end);
    }
}
```

2 Exercise 2 – Composite Pattern

Class Diagram



Answer

This problem requires a **Composite Pattern** to represent parent companies that contain independent companies in a uniform hierarchy.

Java Code

Company Interface

```
public interface Company {
    double getMaintenanceCost();
}
```

IndependentCompany

```
public class IndependentCompany implements Company {
    private int vehicleCount;
    private double unitCost;

    public IndependentCompany(int vehicleCount, double unitCost) {
        this.vehicleCount = vehicleCount;
```

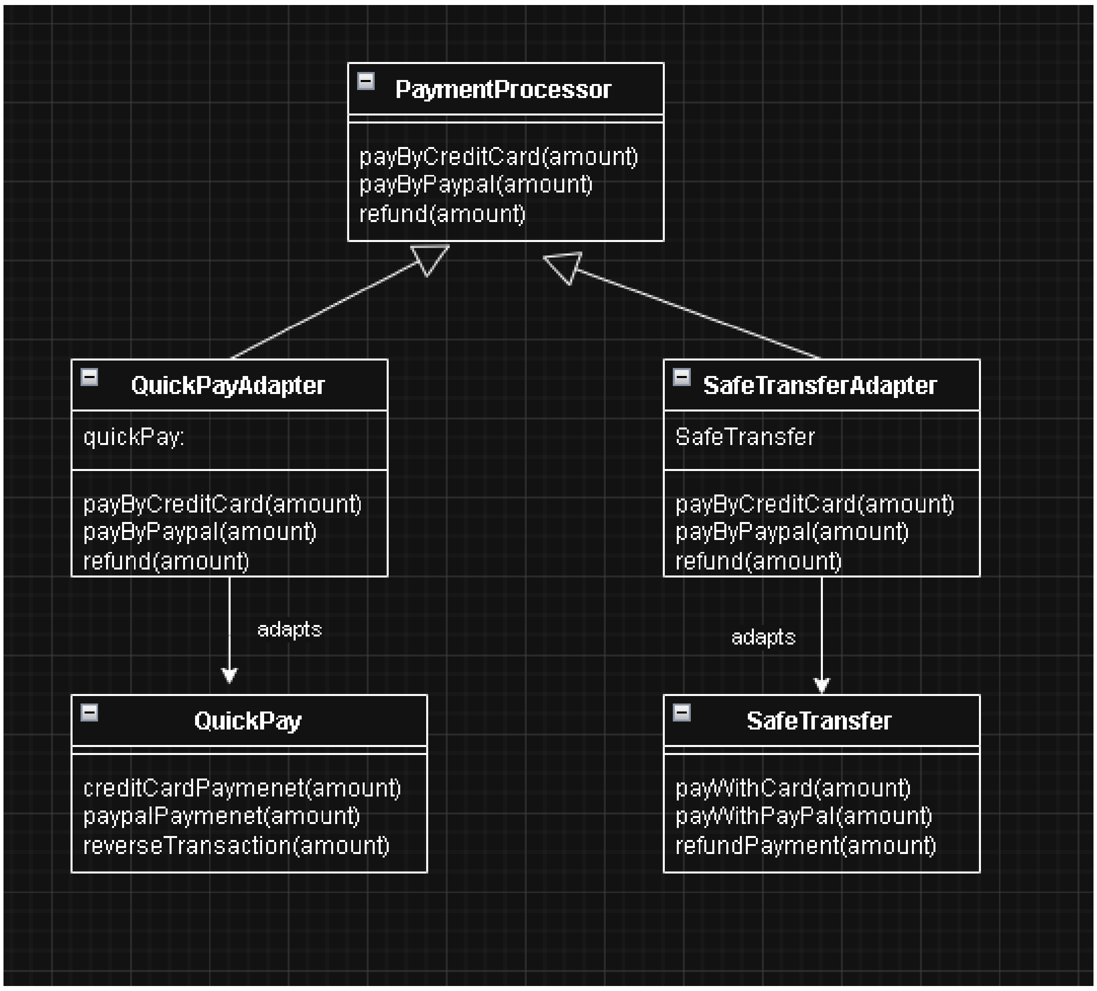
```java
        this.unitCost = unitCost;
    }

    @Override
    public double getMaintenanceCost() {
        return vehicleCount * unitCost;
    }
}
```

ParentCompany

```java
import java.util.ArrayList;
import java.util.List;

public class ParentCompany implements Company {

    private List<Company> subsidiaries = new ArrayList<>();

    public void add(Company c) { subsidiaries.add(c); }
    public void remove(Company c) { subsidiaries.remove(c); }

    @Override
    public double getMaintenanceCost() {
        double total = 0;
        for (Company c : subsidiaries) total += c.getMaintenanceCost();
        return total;
    }
}
```

3 Exercise 3 – Adapter Pattern

Class Diagram



Answer

The **Adapter Pattern** is required to unify QuickPay and SafeTransfer under one PaymentProcessor interface.

Java Code

PaymentProcessor

```java
public interface PaymentProcessor {
    void payByCreditCard(double amount);
    void payByPayPal(double amount);
    void refund(double amount);
}
```

QuickPayAdapter

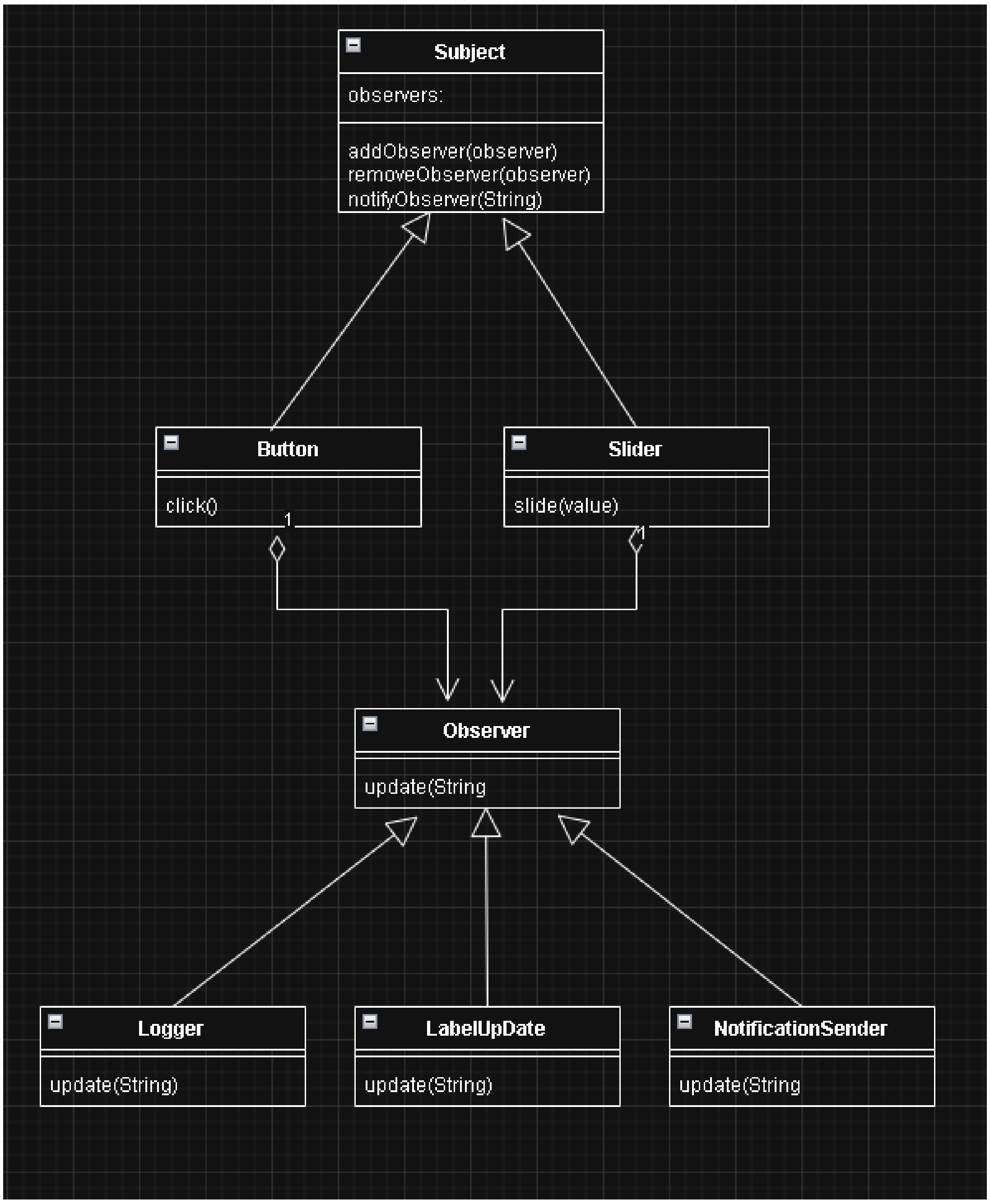
```java
public class QuickPayAdapter implements PaymentProcessor {
    private QuickPay qp = new QuickPay();

    @Override
    public void payByCreditCard(double amount) {
        qp.creditCardPayment(amount);
    }

    @Override
    public void payByPayPal(double amount) {
        qp.paypalPayment(amount);
    }

    @Override
    public void refund(double amount) {
        qp.reverseTransaction(amount);
    }
}
```

SafeTransferAdapter

```java
public class SafeTransferAdapter implements PaymentProcessor {
    private SafeTransfer st = new SafeTransfer();

    @Override
    public void payByCreditCard(double amount) {
        st.payWithCard(amount);
    }

    @Override
    public void payByPayPal(double amount) {
        st.payWithPayPal(amount);
    }

    @Override
    public void refund(double amount) {
        st.refundPayment(amount);
    }
}
```

4 Exercise 4 – Observer Pattern

Class Diagram



Answer

The **Observer Pattern** is appropriate since multiple GUI components must react dynamically to events.

Java Code

Observer Interface

```java
public interface Observer {
    void update(String event);
}
```

Subject Class

```java
import java.util.*;

public abstract class Subject {
    protected List<Observer> observers = new ArrayList<>();

    public void addObserver(Observer o) { observers.add(o); }
    public void removeObserver(Observer o) { observers.remove(o); }

    public void notifyObservers(String event) {
        for (Observer o : observers) o.update(event);
    }
}
```

GUI Elements

```java
public class Button extends Subject {
    private String name;
    public Button(String name) { this.name = name; }
    public void click() { notifyObservers("Button clicked: " + name); }
}

public class Slider extends Subject {
    private String name;
    public Slider(String name) { this.name = name; }
    public void slide(int value) {
        notifyObservers("Slider moved: " + name + " -> " + value);
    }
}
```

Concrete Observers

```java
public class Logger implements Observer {
    @Override
    public void update(String event) {
        System.out.println("Logger: " + event);
    }
}

public class LabelUpdater implements Observer {
    @Override
    public void update(String event) {
        System.out.println("Label updated: " + event);
    }
}

public class NotificationSender implements Observer {
    @Override
    public void update(String event) {
        System.out.println("Notification: " + event);
    }
}
```