Design Patterns: Part 2 - Solutions

UM6P

1 Exercise 1: Strategy Pattern - Navigation System

1.1 Explanation

The Strategy pattern is used here becuase we need different route calculation algorithms that can be swapped at runtime. The Navigator acts as the Context, holding a refrence to the RouteStrategy interface. Each concrete strategy (Walking, Car, Bike) implements its own routing logic.

1.2 Answers to Questions

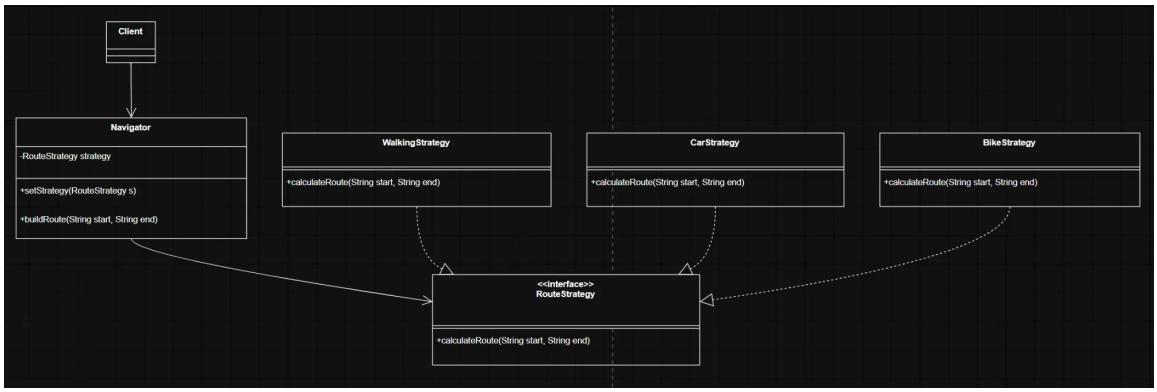
1. **What role does Navigator play?**
   Navigator is the Context class. It holds a refernce to the RouteStrategy and delegates the routing calculation to it without knowing the details of the algorithm.

2. **Why does Navigator depend on RouteStrategy interface?**
   This allows flexibility and interchangability. Navigator doesn't depend on concrete implementations, so strategies can be changed at runtime without modifing the Navigator code.

3. **Which SOLID principles are applied?**

   - Open/Closed Principle: Open for extension (add new strategies) but closed for modification (no need to change Navigator)
   - Dependency Inversion: Navigator depends on abstraction (RouteStrategy) not concrete classes
   - Single Responsibility: Each strategy has one reason to change - its routing algorithm

1.3 Class Diagram



1.4 Java Implementation

1.4.1 RouteStrategy Interface

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

1.4.2 WalkingStrategy

```
public class WalkingStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String start, String end) {
        System.out.println("Calculating walking route from " + start + "
    to " + end);
        System.out.println("Route: pedestrian paths, shortest distance")
    ;
        System.out.println("Estimated time: 45 mins");
    }
}
```

1.4.3 CarStrategy

```
public class CarStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String start, String end) {
        System.out.println("Calculating car route from " + start + " to
    " + end);
        System.out.println("Route: highways and main roads, fastest
    route");
        System.out.println("Estimated time: 15 mins");
    }
}
```

1.4.4 BikeStrategy

```
public class BikeStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String start, String end) {
        System.out.println("Claculating bike route from " + start + " to
    " + end);
        System.out.println("Route: bike lanes and quiet streets");
        System.out.println("Estimated time: 25 mins");
    }
}
```

1.4.5 Navigator

```
public class Navigator {
    private RouteStrategy strategy;

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

```

```
    public void buildRoute(String start, String end) {
        if(strategy == null) {
            System.out.println("Please set a route stratgy first");
            return;
        }
        strategy.calculateRoute(start, end);
    }
}
```

1.4.6 Main Client

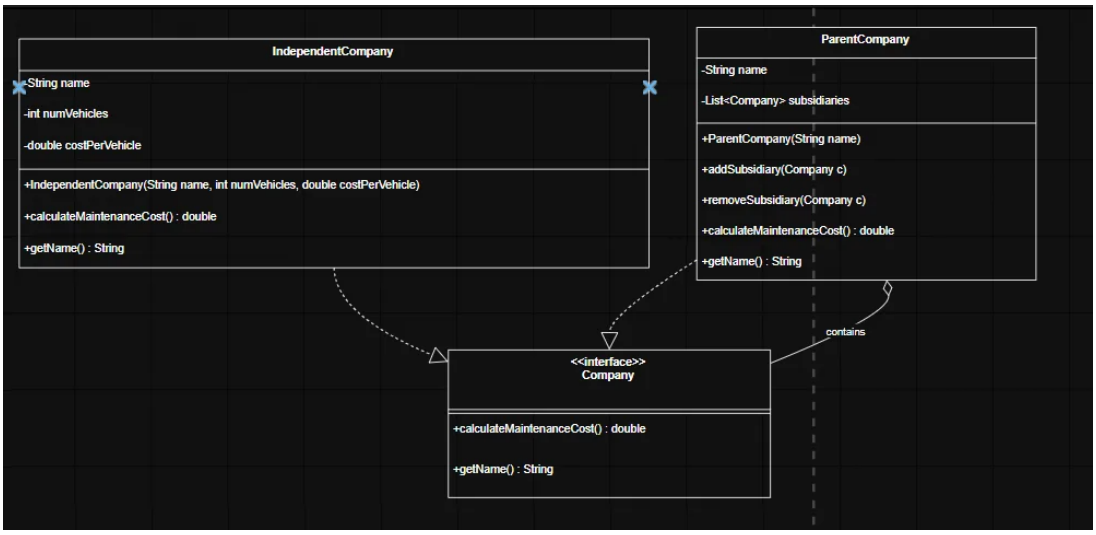
```
public class Main {
    public static void main(String[] args) {
        Navigator navigator = new Navigator();

        // Walking
        navigator.setStrategy(new WalkingStrategy());
        navigator.buildRoute("Home", "Office");

        System.out.println("\n--- Switching to Car ---\n");

        // Car
        navigator.setStrategy(new CarStrategy());
        navigator.buildRoute("Home", "Office");

        System.out.println("\n--- Switching to Bike ---\n");

        // Bike
        navigator.setStrategy(new BikeStrategy());
        navigator.buildRoute("Home", "Office");
    }
}
```

2 Exercise 2: Composite Pattern - Vehicle Maintenance

2.1 Explanation

This problem needs the Composite pattern because we have a tree structure where parent companies contain independent companies. We need to treat both individual companies and groups of companies uniformly when calculating maintanence costs. The pattern allows recursive composition and uniform treatment of leaf and composite objects.

2.2 Class Diagram



2.3 Java Implementation

2.3.1 Company Interface

```java
public interface Company {
    double calculateMaintenanceCost();
    String getName();
}
```

2.3.2 IndependentCompany

```java
public class IndependentCompany implements Company {
    private String name;
    private int numVehicles;
    private double costPerVehilce;

    public IndependentCompany(String name, int numVehicles, double
   costPerVehicle) {
        this.name = name;
        this.numVehicles = numVehicles;
        this.costPerVehilce = costPerVehicle;
    }

    @Override
    public double calculateMaintenanceCost() {
        return numVehicles * costPerVehilce;
    }

```

```
    @Override
    public String getName() {
        return name;
    }
}
```

2.3.3 ParentCompany

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

public class ParentCompany implements Company {
    private String name;
    private List<Company> subsidiaries;

    public ParentCompany(String name) {
        this.name = name;
        this.subsidiaries = new ArrayList<>();
    }

    public void addSubsidary(Company company) {
        subsidiaries.add(company);
    }

    public void removeSubsidiary(Company company) {
        subsidiaries.remove(company);
    }

    @Override
    public double calculateMaintenanceCost() {
        double totalCost = 0;
        for(Company company : subsidiaries) {
            totalCost += company.calculateMaintenanceCost();
        }
        return totalCost;
    }

    @Override
    public String getName() {
        return name;
    }
}
```

2.3.4 Main Client

```
public class Main {
    public static void main(String[] args) {
        // Create independent companies
        Company company1 = new IndependentCompany("TechCorp", 50, 200.0)
    ;
        Company company2 = new IndependentCompany("LogisticsPro", 100,
    150.0);
        Company company3 = new IndependentCompany("DeliverFast", 30,
    180.0);

        // Create parent company
```

```
        ParentCompany parentCompany = new ParentCompany("GlobalGroup");
        parentCompany.addSubsidary(company1);
        parentCompany.addSubsidary(company2);

        // Create another parent company
        ParentCompany megaCorp = new ParentCompany("MegaCorp");
        megaCorp.addSubsidary(parentCompany);
        megaCorp.addSubsidary(company3);

        // Calculate costs
        System.out.println(company1.getName() + " maintenance cost: $" +
                           company1.calculateMaintenanceCost());
        System.out.println(parentCompany.getName() + " maintenance cost:
    $" +
                           parentCompany.calculateMaintenanceCost());
        System.out.println(megaCorp.getName() + " total maintenance cost
    : $" +
                           megaCorp.calculateMaintenanceCost());
    }
}
```

3 Exercise 3: Adapter Pattern - Payment Processing

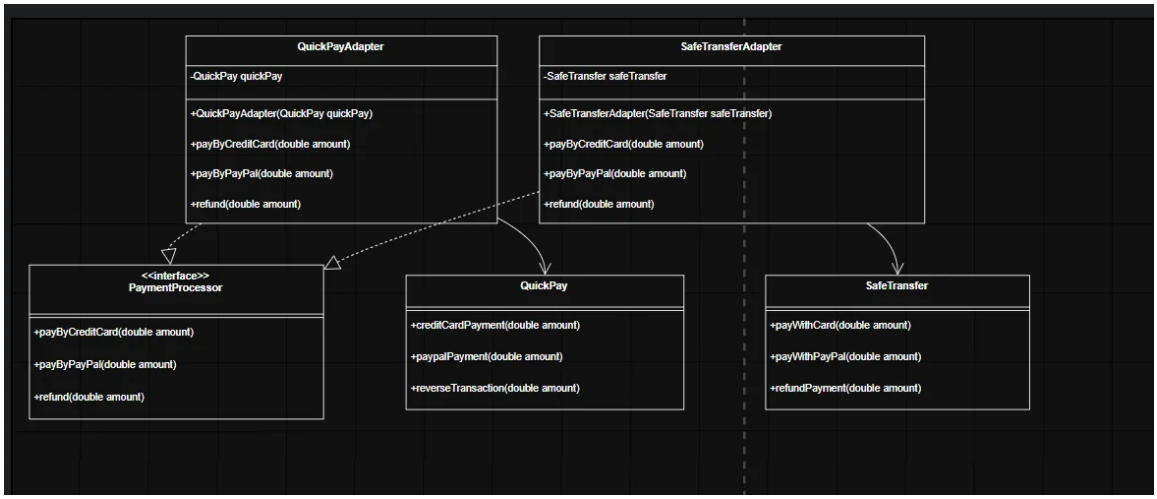
3.1 Explanation

The Adapter pattern is needed because we have incompatible interfaces. The system expects PaymentProcessor interface, but the third-party services (QuickPay and SafeTransfer) have different method names. The adapters wrap these services and translate the interface calls to match what the system expects.

3.2 Participants

- **Target**: PaymentProcessor interface (what the system expects)
- **Adaptee**: QuickPay and SafeTransfer (existing services with incompatible interfaces)
- **Adapter**: QuickPayAdapter and SafeTransferAdapter (make adaptees compatible with target)
- **Client**: The e-commerce system that uses PaymentProcessor

3.3 Class Diagram



3.4 Java Implementation

3.4.1 PaymentProcessor Interface

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

3.4.2 QuickPay (Existing Service)

```
public class QuickPay {
    public void creditCardPayment(double amount) {
        System.out.println("QuickPay: Processing credit card payment $"
   + amount);
    }

```

```
    public void paypalPayment(double amount) {
        System.out.println("QuickPay: Processing PayPal payment $" +
    amount);
    }

    public void reverseTransaction(double amount) {
        System.out.println("QuickPay: Reversing transaction $" + amount)
    ;
    }
}
```

3.4.3 SafeTransfer (Existing Service)

```
public class SafeTransfer {
    public void payWithCard(double amount) {
        System.out.println("SafeTransfer: Paying with credit card $" +
    amount);
    }

    public void payWithPayPal(double amount) {
        System.out.println("SafeTransfer: Paying with PayPal $" + amount
    );
    }

    public void refundPayment(double amount) {
        System.out.println("SafeTransfer: Refunding payment $" + amount)
    ;
    }
}
```

3.4.4 QuickPayAdapter

```
public class QuickPayAdapter implements PaymentProcessor {
    private QuickPay quickPay;

    public QuickPayAdapter(QuickPay quickPay) {
        this.quickPay = quickPay;
    }

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

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

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

3.4.5 SafeTransferAdapter

```
public class SafeTransferAdapter implements PaymentProcessor {
    private SafeTransfer safeTransfer;

    public SafeTransferAdapter(SafeTransfer safeTransfer) {
        this.safeTransfer = safeTransfer;
    }

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

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

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

3.4.6 Main Client

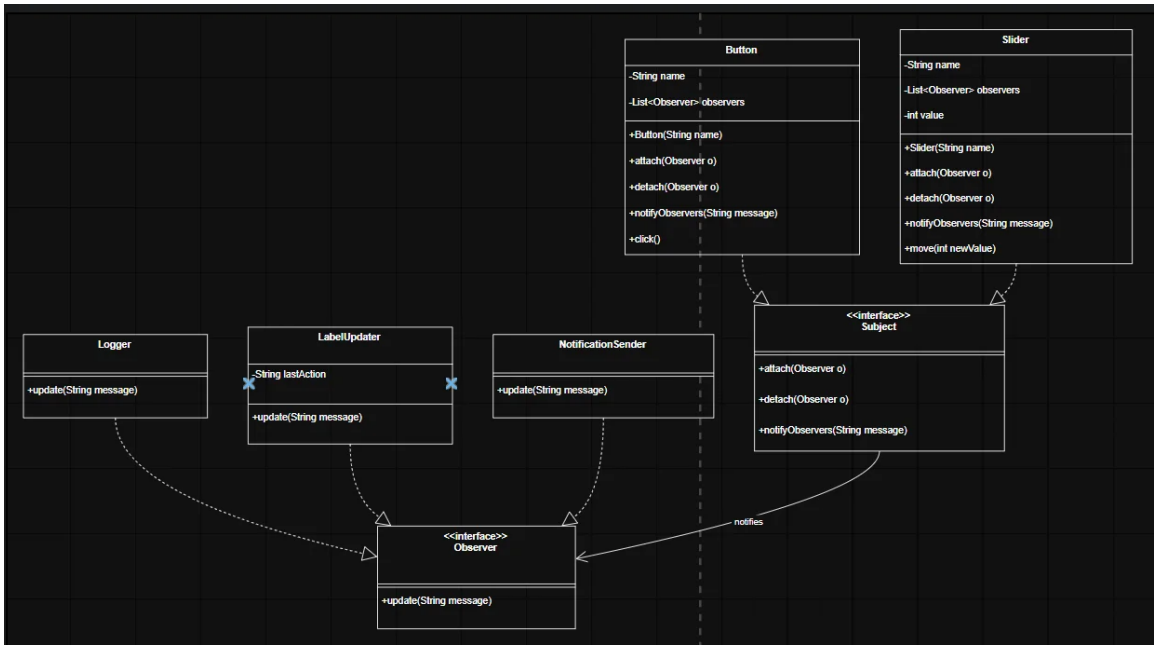
```
public class Main {
    public static void main(String[] args) {
        // Using QuickPay through adapter
        PaymentProcessor processor1 = new QuickPayAdapter(new QuickPay()
    );
        processor1.payByCreditCard(100.0);
        processor1.payByPayPal(50.0);
        processor1.refund(25.0);

        System.out.println("\n--- Switching to SafeTransfer ---\n");

        // Using SafeTransfer through adapter
        PaymentProcessor processor2 = new SafeTransferAdapter(new
    SafeTransfer());
        processor2.payByCreditCard(200.0);
        processor2.payByPayPal(75.0);
        processor2.refund(50.0);
    }
}
```

4 Exercise 4: Observer Pattern - GUI Dashboard

4.1 Explanation

The Observer pattern is best for this scenario because multiple components need to react to changes in GUI elements. The buttons and sliders are the subjects (observables) that notify observers (Logger, LabelUpdater, NotificationSender) when their state changes. This creates a loose coupling where subjects don't need to know the concrete types of their observers.

4.2 Class Diagram



4.3 Java Implementation

4.3.1 Observer Interface

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

4.3.2 Subject Interface

```
public interface Subject {
    void attach(Observer observer);
    void detach(Observer observer);
    void notifyObservers(String message);
}
```

4.3.3 Button

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

public class Button implements Subject {
    private String name;
    private List<Observer> observers;
```

```
7
8       public Button(String name) {
9           this.name = name;
10          this.observers = new ArrayList<>();
11      }
12
13      @Override
14      public void attach(Observer observer) {
15          observers.add(observer);
16      }
17
18      @Override
19      public void detach(Observer observer) {
20          observers.remove(observer);
21      }
22
23      @Override
24      public void notifyObservers(String message) {
25          for(Observer observer : observers) {
26              observer.update(message);
27          }
28      }
29
30      public void click() {
31          System.out.println("\n[" + name + " clicked]");
32          notifyObservers(name + " was clicked");
33      }
34  }
```

4.3.4 Slider

```
1   import java.util.ArrayList;
2   import java.util.List;
3
4   public class Slider implements Subject {
5       private String name;
6       private List<Observer> observers;
7       private int value;
8
9       public Slider(String name) {
10          this.name = name;
11          this.observers = new ArrayList<>();
12          this.value = 0;
13      }
14
15      @Override
16      public void attach(Observer observer) {
17          observers.add(observer);
18      }
19
20      @Override
21      public void detach(Observer observer) {
22          observers.remove(observer);
23      }
24
25      @Override
26      public void notifyObservers(String message) {
```

```
        for(Observer observer : observers) {
            observer.update(message);
        }
    }

    public void move(int newValue) {
        this.value = newValue;
        System.out.println("\n[" + name + " moved to " + value + "]");
        notifyObservers(name + " moved to " + value);
    }
}
```

4.3.5 Logger

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

4.3.6 LabelUpdater

```
public class LabelUpdater implements Observer {
    private String lastAction;

    @Override
    public void update(String message) {
        this.lastAction = message;
        System.out.println("LabelUpdater: Updating label to show - " +
   lastAction);
    }
}
```

4.3.7 NotificationSender

```
public class NotificationSender implements Observer {
    @Override
    public void update(String message) {
        System.out.println("NotificationSender: Sending alert for - " +
   message);
    }
}
```

4.3.8 Main Client

```
public class Main {
    public static void main(String[] args) {
        // Create GUI elements
        Button submitButton = new Button("SubmitButton");
        Button cancelButton = new Button("CancelButton");
        Slider volumeSlider = new Slider("VolumeSlider");
        Slider brightnessSlider = new Slider("BrightnessSlider");

        // Create observers
```

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

        // Attach observers to SubmitButton
        submitButton.attach(logger);
        submitButton.attach(labelUpdater);

        // Attach observers to VolumeSlider
        volumeSlider.attach(logger);
        volumeSlider.attach(notificationSender);

        // Attach observers to CancelButton
        cancelButton.attach(logger);

        // Attach observers to BrightnessSlider
        brightnessSlider.attach(logger);
        brightnessSlider.attach(labelUpdater);

        // Simulate user interactions
        submitButton.click();
        volumeSlider.move(75);
        cancelButton.click();
        brightnessSlider.move(50);
    }
}
```