Design Patterns: Part 2

Zakariya Sabri

1 Exercise 1:

1.1 Task 1: Class Diagram

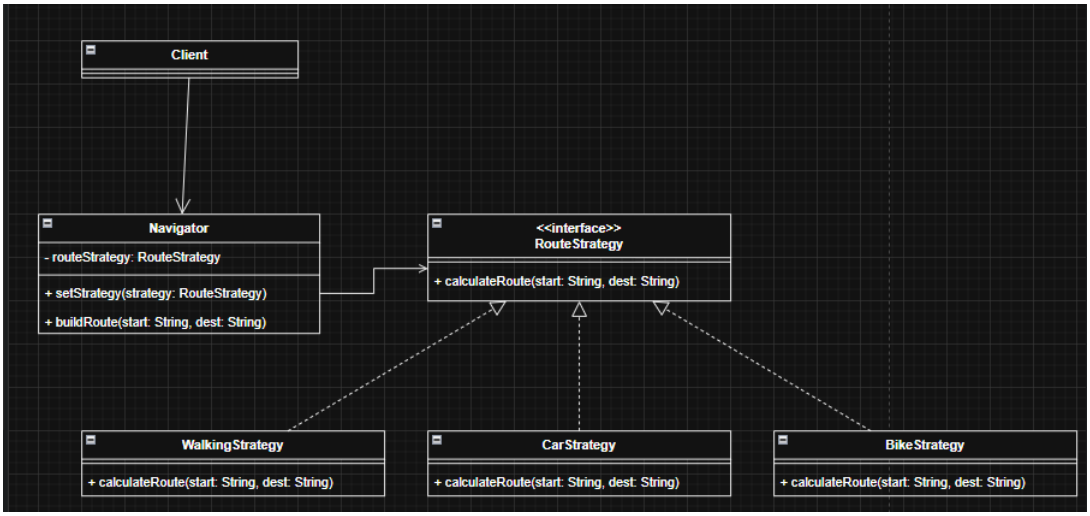

Figure 1: Class Diagram

1.1.1 Question 1: What role does the Navigator class play?

The Navigator class is the **Context** in the Strategy Pattern. It holds a reference to a RouteStrategy interface and delegates route calculation to the current strategy. The Navigator allows clients to change strategies at runtime without implementing routing algorithms itself.

1.1.2 Question 2: Why does Navigator depend on the RouteStrategy interface?

Navigator depends on the RouteStrategy interface for:

- **Loose coupling**: Navigator doesn't need to know specific implementations
- **Flexibility**: New strategies can be added without changing Navigator
- **Dependency Inversion**: Depends on abstraction, not concrete classes
- **Runtime switching**: Strategies can be swapped dynamically

1.1.3 Question 3: Which SOLID principles are applied?

Single Responsibility Principle (SRP): Each class has one responsibility. Navigator manages strategy selection, while each strategy implements its own routing algorithm.

Open/Closed Principle (OCP): The system is open for extension (new strategies can be added) but closed for modification (existing code doesn't change).

Liskov Substitution Principle (LSP): Any RouteStrategy implementation can substitute another without breaking functionality.

Dependency Inversion Principle (DIP): Both Navigator and concrete strategies depend on the RouteStrategy abstraction.

1.2 Task 2: Java Implementation

1.2.1 RouteStrategy Interface

Listing 1: RouteStrategy.java

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

1.2.2 WalkingStrategy Class

Listing 2: WalkingStrategy.java

```
public class WalkingStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String start, String destination)
        {
        System.out.println("Walking from " + start + " to " +
            destination);
        System.out.println("Time: 45 min, Distance: 3 km");
    }
}
```

1.2.3 CarStrategy Class

Listing 3: CarStrategy.java

```
public class CarStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String start, String destination)
        {
        System.out.println("Driving from " + start + " to " +
            destination);
        System.out.println("Time: 15 min, Distance: 12 km");
    }
```

```
}
```

1.2.4 BikeStrategy Class

Listing 4: BikeStrategy.java

```
public class BikeStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String start, String destination)
        {
        System.out.println("Biking from " + start + " to " +
            destination);
        System.out.println("Time: 25 min, Distance: 6 km");
    }
}
```

1.2.5 Navigator Class

Listing 5: Navigator.java

```
public class Navigator {
    private RouteStrategy routeStrategy;

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

    public void buildRoute(String start, String destination) {
        routeStrategy.calculateRoute(start, destination);
    }
}
```

1.2.6 Main Class

Listing 6: Main.java

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

        navigator.setStrategy(new WalkingStrategy());
        navigator.buildRoute("Home", "Work");

        navigator.setStrategy(new CarStrategy());
        navigator.buildRoute("Home", "Work");

        navigator.setStrategy(new BikeStrategy());
        navigator.buildRoute("Home", "Work");
    }
}
```

2 Exercise 2:

2.1 Question 1: Which design pattern is best suited?

The **Composite Design Pattern** is best suited for this problem because we need to represent a tree structure where individual companies (independent companies) and compositions of companies (parent companies) must be treated uniformly. The pattern allows us to calculate maintenance costs for both types in the same way.

2.2 Question 2: Class Diagram

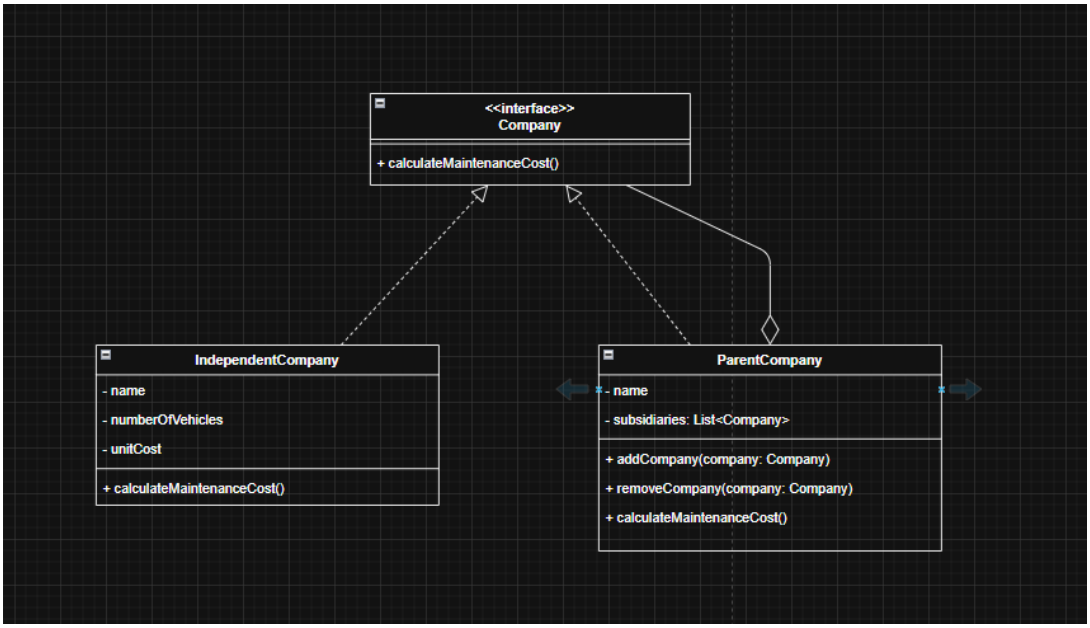

Figure 2: Class Diagram for Composite Pattern

2.3 Question 3: Java Implementation

2.3.1 Company Interface

Listing 7: Company.java

```
1 public interface Company {
2     double calculateMaintenanceCost();
3 }
```

2.3.2 IndependentCompany Class

Listing 8: IndependentCompany.java

```
1 public class IndependentCompany implements Company {
2     private String name;
3     private int numberOfVehicles;
4     private double unitCost;
```

```
5
6       public IndependentCompany(String name, int numberOfVehicles,
            double unitCost) {
7           this.name = name;
8           this.numberOfVehicles = numberOfVehicles;
9           this.unitCost = unitCost;
10      }
11
12      @Override
13      public double calculateMaintenanceCost() {
14          return numberOfVehicles * unitCost;
15      }
16  }
```

2.3.3 ParentCompany Class

Listing 9: ParentCompany.java

```
1   import java.util.ArrayList;
2   import java.util.List;
3
4   public class ParentCompany implements Company {
5       private String name;
6       private List<Company> subsidiaries;
7
8       public ParentCompany(String name) {
9           this.name = name;
10          this.subsidiaries = new ArrayList<>();
11      }
12
13      public void addCompany(Company company) {
14          subsidiaries.add(company);
15      }
16
17      public void removeCompany(Company company) {
18          subsidiaries.remove(company);
19      }
20
21      @Override
22      public double calculateMaintenanceCost() {
23          double total = 0;
24          for (Company company : subsidiaries) {
25              total += company.calculateMaintenanceCost();
26          }
27          return total;
28      }
29  }
```

2.3.4 Main Class

Listing 10: Main.java

```
public class Main {
    public static void main(String[] args) {
        IndependentCompany company1 = new IndependentCompany("
            Company A", 10, 500);
        IndependentCompany company2 = new IndependentCompany("
            Company B", 5, 600);
        IndependentCompany company3 = new IndependentCompany("
            Company C", 8, 550);

        ParentCompany parent = new ParentCompany("Parent Corp");
        parent.addCompany(company1);
        parent.addCompany(company2);
        parent.addCompany(company3);

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

3 Exercise 3:

3.1 Question 1: Which design pattern should you use?

The **Adapter Design Pattern** should be used because we need to integrate existing third-party payment services (QuickPay and SafeTransfer) that have incompatible interfaces with our standard PaymentProcessor interface. The adapter wraps the incompatible interface and translates method calls to make the services compatible without modifying their source code.

3.2 Question 2: Identify participants and class diagram

Participants:

- **Target**: PaymentProcessor interface
- **Adaptee 1**: QuickPay class (existing third-party service)
- **Adaptee 2**: SafeTransfer class (existing third-party service)
- **Adapter 1**: QuickPayAdapter (adapts QuickPay to PaymentProcessor)
- **Adapter 2**: SafeTransferAdapter (adapts SafeTransfer to PaymentProcessor)

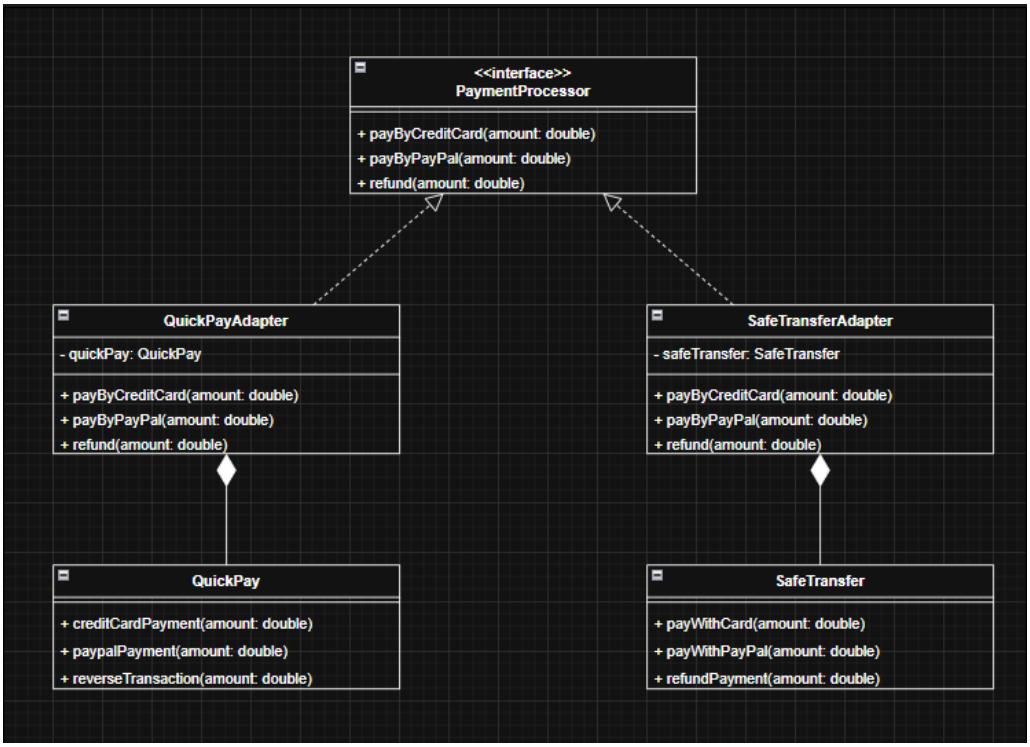

Figure 3: Class Diagram for Adapter Pattern

3.3 Question 3: Java Implementation

3.3.1 QuickPayAdapter Class

Listing 11: QuickPayAdapter.java

```
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);
    }
}
```

3.3.2 SafeTransferAdapter Class

Listing 12: SafeTransferAdapter.java

```
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) {
```

```
20          safeTransfer.refundPayment(amount);
21      }
22  }
```

3.3.3 Main Class

Listing 13: Main.java

```
1  public class Main {
2      public static void main(String[] args) {
3          QuickPay quickPay = new QuickPay();
4          PaymentProcessor quickPayProcessor = new QuickPayAdapter(
               quickPay);
5
6          SafeTransfer safeTransfer = new SafeTransfer();
7          PaymentProcessor safeTransferProcessor = new
               SafeTransferAdapter(safeTransfer);
8
9          quickPayProcessor.payByCreditCard(100.0);
10         quickPayProcessor.payByPayPal(50.0);
11         quickPayProcessor.refund(25.0);
12
13         safeTransferProcessor.payByCreditCard(200.0);
14         safeTransferProcessor.payByPayPal(75.0);
15         safeTransferProcessor.refund(30.0);
16     }
17 }
```

4 Exercise 4:

4.1 Question 1: Which design pattern is most suitable?

The **Observer Design Pattern** is most suitable because we have GUI elements (subjects) whose state changes need to be communicated to multiple components (observers). The pattern ensures that when a button is clicked or a slider is moved, all registered observers are notified automatically. This provides loose coupling between the GUI elements and the components that react to them.

4.2 Question 2: Class Diagram

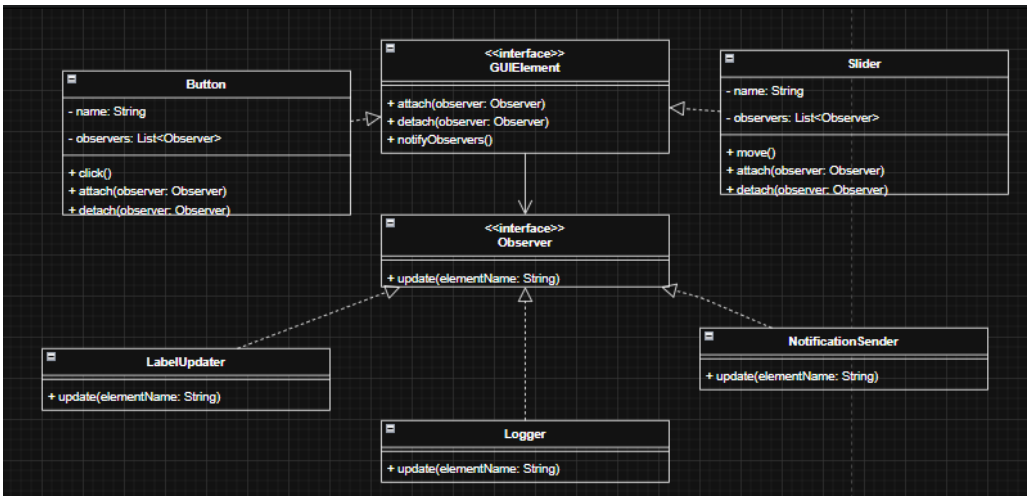

Figure 4: Class Diagram for Observer Pattern

4.3 Question 3: Java Implementation

4.3.1 Observer Interface

Listing 14: Observer.java

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

4.3.2 GUIElement Interface

Listing 15: GUIElement.java

```java
public interface GUIElement {
    void attach(Observer observer);
    void detach(Observer observer);
    void notifyObservers();
}
```

4.3.3 Button Class

Listing 16: Button.java

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

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

    public Button(String name) {
        this.name = name;
        this.observers = new ArrayList<>();
    }

    public void click() {
        System.out.println(name + " clicked!");
        notifyObservers();
    }

    @Override
    public void attach(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
    }

```

```
    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(name);
        }
    }
}
```

4.3.4 Slider Class

Listing 17: Slider.java

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

public class Slider implements GUIElement {
    private String name;
    private List<Observer> observers;

    public Slider(String name) {
        this.name = name;
        this.observers = new ArrayList<>();
    }

    public void move() {
        System.out.println(name + " moved!");
        notifyObservers();
    }

    @Override
    public void attach(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(name);
        }
    }
}
```

4.3.5 Logger Class

Listing 18: Logger.java

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

4.3.6 LabelUpdater Class

Listing 19: LabelUpdater.java

```
public class LabelUpdater implements Observer {
    @Override
    public void update(String elementName) {
        System.out.println("LabelUpdater: Updating label for " +
            elementName);
    }
}
```

4.3.7 NotificationSender Class

Listing 20: NotificationSender.java

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

4.3.8 Main Class

Listing 21: Main.java

```
public class Main {
    public static void main(String[] args) {
        Logger logger = new Logger();
        LabelUpdater labelUpdater = new LabelUpdater();
        NotificationSender notificationSender = new
            NotificationSender();

        Button submitButton = new Button("SubmitButton");
        submitButton.attach(logger);
        submitButton.attach(labelUpdater);

        Slider volumeSlider = new Slider("VolumeSlider");
        volumeSlider.attach(logger);
```

```
13          volumeSlider.attach(notificationSender);
14
15          submitButton.click();
16          System.out.println();
17          volumeSlider.move();
18      }
19  }
```