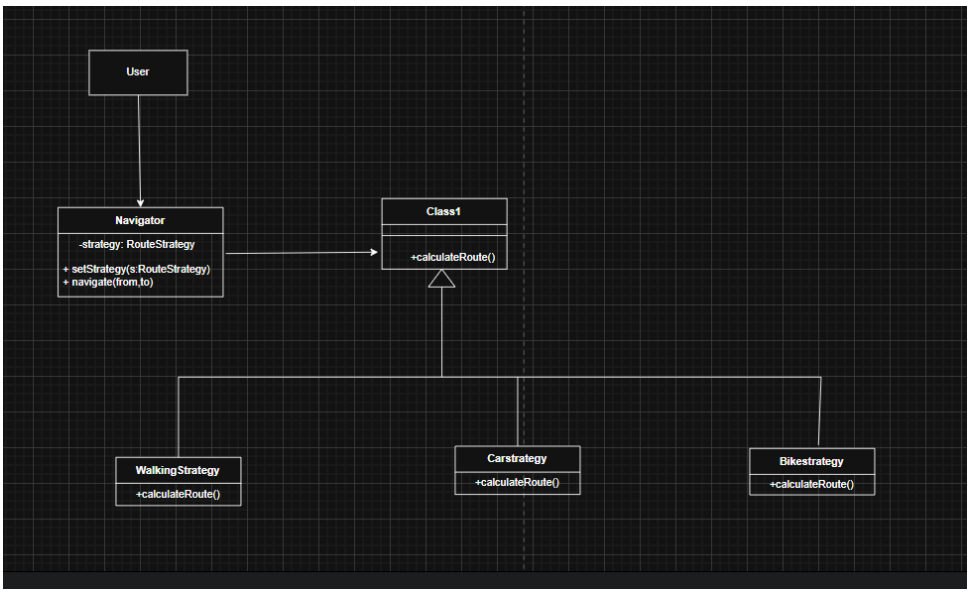
TP Design Patterns - Part 2
Solutions and Explanations

Exercise 1:

Task 1: Class Diagram and Questions

The Strategy pattern is perfectly suited for this navigation scenario because it allows different routing algorithms to be interchangeable, since we want to perform the same task but with different approaches .



Questions:

1. **What role does the Navigator class play?**
   The Navigator class plays the role of the **Context** in the Strategy pattern.
   2. **Why does Navigator depend on the RouteStrategy interface?**
   The Navigator uses the RouteStrategy interface to follow the Dependency Inversion Principle. This way, it doesn't depend on any specific algorithm. Instead, it can work with any strategy you plug in. That makes the whole system easier to extend, update, and adapt without changing the Navigator itself.
   3. **Which SOLID principles are applied in this design?**

- **Open/Closed Principle**
- **Dependency Inversion Principle**
- **Single Responsibility Principle**

Task 2: Java Implementation

```
public interface RouteStrategy {
    void calculateRoute(String from, String to);
}

public class WalkingStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String from, String to) {
        System.out.println("Calculating␣walking␣route␣from␣" +
           from + "␣to␣" + to);
    }
}

public class CarStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String from, String to) {
        System.out.println("Calculating␣car␣route␣from␣" + from +
           "␣to␣" + to);
    }
}

public class BikeStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String from, String to) {
        System.out.println("Calculating␣bike␣route␣from␣" + from +
            "␣to␣" + to);
    }
}

public class Navigator {
    private RouteStrategy strategy;

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

    public void calculateRoute(String from, String to) {
        if (strategy == null) {
            System.out.println("No␣strategy␣set!");
            return;
        }
        strategy.calculateRoute(from, to);
    }
}

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

        navigator.setStrategy(new WalkingStrategy());
```

```
        navigator.calculateRoute("Home", "Office");

        navigator.setStrategy(new CarStrategy());
        navigator.calculateRoute("Home", "Airport");

        navigator.setStrategy(new BikeStrategy());
        navigator.calculateRoute("Park", "Mall");
    }
}
```

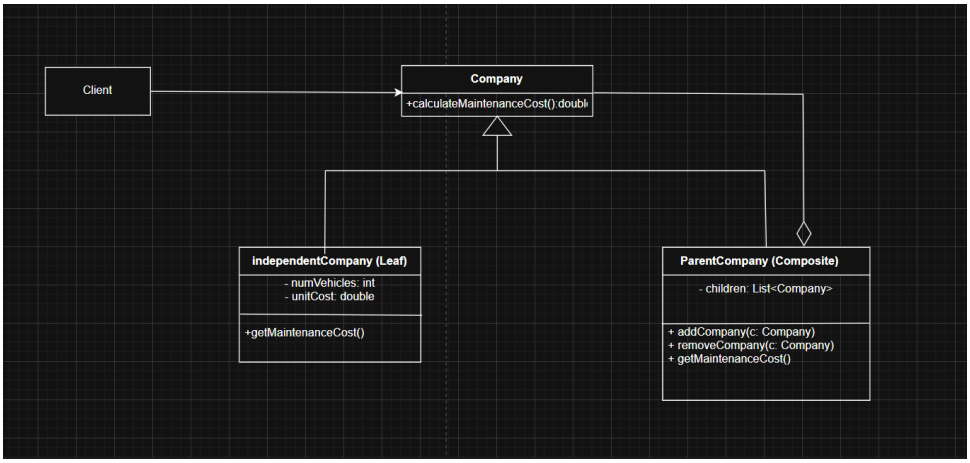
Exercise 2:

1. Design Pattern Identification

The **Composite Pattern** is best suited for this problem because:

- We need to treat individual companies and groups of companies uniformly
- Companies can contain other companies (parent-child relationship)
- We need to perform the same operation (calculate maintenance cost) on both individual and composite objects

2. Class Diagram



3. Java Implementation

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

public interface CompanyComponent {
    double calculateCost();
    void add(CompanyComponent company);
    void remove(CompanyComponent company);
    CompanyComponent getChild(int index);
}
```

```
10
11 public class Company implements CompanyComponent {
12     private String name;
13     private int vehicleCount;
14     private double unitCost;
15
16     public Company(String name, int vehicleCount, double unitCost)
           {
17         this.name = name;
18         this.vehicleCount = vehicleCount;
19         this.unitCost = unitCost;
20     }
21
22     @Override
23     public double calculateCost() {
24         return vehicleCount * unitCost;
25     }
26
27     @Override
28     public void add(CompanyComponent company) {
29         throw new UnsupportedOperationException("Cannot␣add␣to␣a␣
               leaf␣company");
30     }
31
32     @Override
33     public void remove(CompanyComponent company) {
34         throw new UnsupportedOperationException("Cannot␣remove␣
               from␣a␣leaf␣company");
35     }
36
37     @Override
38     public CompanyComponent getChild(int index) {
39         throw new UnsupportedOperationException("Leaf␣company␣has␣
               no␣children");
40     }
41
42     public String getName() { return name; }
43     public int getVehicleCount() { return vehicleCount; }
44     public double getUnitCost() { return unitCost; }
45 }
46
47 public class CompanyComposite implements CompanyComponent {
48     private String name;
49     private List<CompanyComponent> children;
50
51     public CompanyComposite(String name) {
52         this.name = name;
53         this.children = new ArrayList<>();
54     }
55
56     @Override
```

```
57      public double calculateCost() {
58          double totalCost = 0;
59          for (CompanyComponent child : children) {
60              totalCost += child.calculateCost();
61          }
62          return totalCost;
63      }
64
65      @Override
66      public void add(CompanyComponent company) {
67          children.add(company);
68      }
69
70      @Override
71      public void remove(CompanyComponent company) {
72          children.remove(company);
73      }
74
75      @Override
76      public CompanyComponent getChild(int index) {
77          return children.get(index);
78      }
79
80      public String getName() { return name; }
81      public List<CompanyComponent> getChildren() { return children;
            }
82  }
83
84  public class MaintenanceClient {
85      public static void main(String[] args) {
86          CompanyComponent company1 = new Company("SmallCorp", 5,
                100.0);
87          CompanyComponent company2 = new Company("MediumBiz", 10,
                90.0);
88          CompanyComponent company3 = new Company("LargeEnt", 20,
                80.0);
89
90          CompanyComposite parentCompany = new CompanyComposite("
                MegaCorp");
91          parentCompany.add(company1);
92          parentCompany.add(company2);
93
94          CompanyComposite holdingCompany = new CompanyComposite("
                GlobalHoldings");
95          holdingCompany.add(parentCompany);
96          holdingCompany.add(company3);
97
98          System.out.println("Company1␣cost:␣$" + company1.
                calculateCost());
99          System.out.println("Parent␣company␣cost:␣$" +
                parentCompany.calculateCost());
```

```
100             System.out.println("Holding company cost: $" +
                    holdingCompany.calculateCost());
101         }
102 }
```

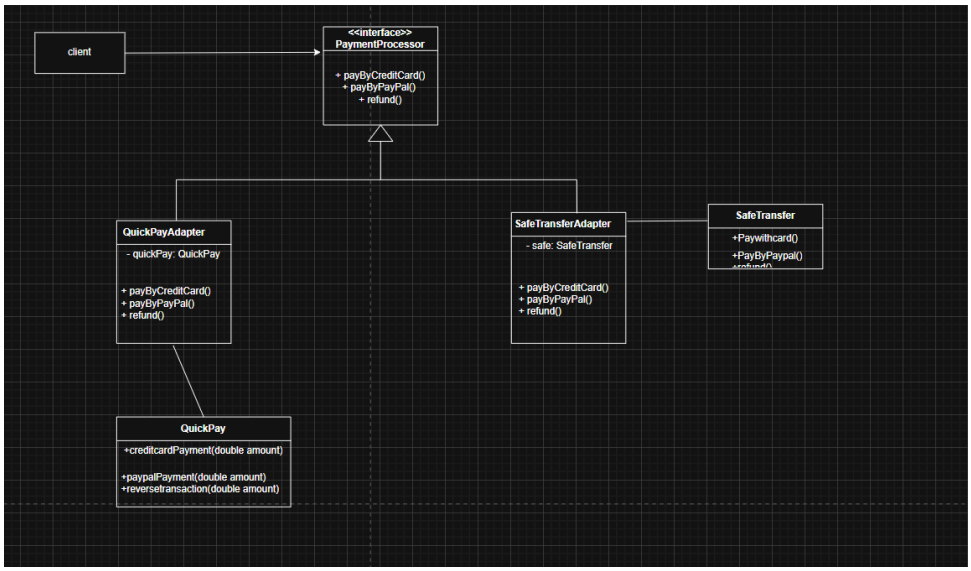
Exercise 3:

1. Design Pattern Identification

The **Adapter Pattern** should be used because:

- We need to adapt existing third-party payment services to a common interface
- We cannot modify the existing payment services
- We want clients to use a unified interface without knowing the underlying implementation details

2. Class Diagram and Participants



Participants:

- **Target**: PaymentProcessor interface
- **Adaptee**: QuickPay and SafeTransfer classes
- **Adapter**: QuickPayAdapter and SafeTransferAdapter
- **Client**: Uses the PaymentProcessor interface

3. Java Implementation

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

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);
    }
}

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);
    }
}

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);
    }
}

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);
    }
}

public class ECommerceClient {
    public static void main(String[] args) {
        QuickPay quickPay = new QuickPay();
        PaymentProcessor processor1 = new QuickPayAdapter(quickPay
            );

        processor1.payByCreditCard(100.0);
        processor1.payByPayPal(50.0);
        processor1.refund(25.0);

        System.out.println("---␣Switching␣payment␣service␣---");

        SafeTransfer safeTransfer = new SafeTransfer();
```

```
        PaymentProcessor processor2 = new SafeTransferAdapter(
            safeTransfer);

        processor2.payByCreditCard(200.0);
        processor2.payByPayPal(75.0);
        processor2.refund(30.0);
    }
}
```

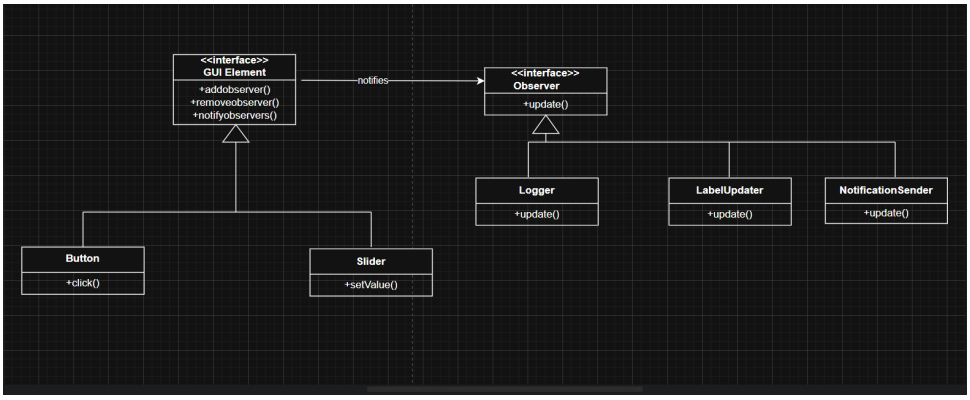
Exercise 4:

1. Design Pattern Identification

The **Observer Pattern** is most suitable for this scenario because:

- Multiple components need to react to changes in GUI elements
- GUI elements (subjects) don't need to know the specific details of their observers
- New observers can be added without modifying the subjects

2. Class Diagram



3. Java Implementation

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

public interface Observer {
    void update(String componentName, String action);
}

public interface Subject {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers(String action);
}

```

```
14 public abstract class GUIComponent implements Subject {
15     protected String name;
16     protected List<Observer> observers;
17 
18     public GUIComponent(String name) {
19         this.name = name;
20         this.observers = new ArrayList<>();
21     }
22 
23     @Override
24     public void addObserver(Observer observer) {
25         observers.add(observer);
26     }
27 
28     @Override
29     public void removeObserver(Observer observer) {
30         observers.remove(observer);
31     }
32 
33     @Override
34     public void notifyObservers(String action) {
35         for (Observer observer : observers) {
36             observer.update(name, action);
37         }
38     }
39 
40     public String getName() {
41         return name;
42     }
43 }
44 
45 public class Button extends GUIComponent {
46     public Button(String name) {
47         super(name);
48     }
49 
50     public void click() {
51         System.out.println("Button ’" + name + "’ clicked");
52         notifyObservers("clicked");
53     }
54 }
55 
56 public class Slider extends GUIComponent {
57     public Slider(String name) {
58         super(name);
59     }
60 
61     public void slide() {
62         System.out.println("Slider ’" + name + "’ moved");
63         notifyObservers("slid");
64     }
```

```
}

public class Logger implements Observer {
    @Override
    public void update(String componentName, String action) {
        System.out.println("Logger: " + componentName + " was " +
            action);
    }
}

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

public class NotificationSender implements Observer {
    @Override
    public void update(String componentName, String action) {
        if (componentName.contains("Volume") || componentName.
            contains("Submit")) {
             System.out.println("NotificationSender: Sending alert
                for " +
                                componentName + " " + action);
        }
    }
}

public class GUIClient {
    public static void main(String[] args) {
        Button submitButton = new Button("SubmitButton");
        Slider volumeSlider = new Slider("VolumeSlider");
        Slider brightnessSlider = new Slider("BrightnessSlider");

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

        submitButton.addObserver(logger);
        submitButton.addObserver(labelUpdater);

        volumeSlider.addObserver(logger);
        volumeSlider.addObserver(notificationSender);

        brightnessSlider.addObserver(logger);

        System.out.println("=== User clicks SubmitButton ===");
        submitButton.click();
```

```
112
113         System.out.println("\n=== User moves VolumeSlider ===");
114         volumeSlider.slide();
115
116         System.out.println("\n=== User moves BrightnessSlider ===");
117         brightnessSlider.slide();
118
119         System.out.println("\n=== Adding NotificationSender to SubmitButton ===");
120         submitButton.addObserver(notificationSender);
121         submitButton.click();
122     }
123 }
```

Conclusion

This TP covered four essential design patterns:

- **Strategy Pattern**: For interchangeable algorithms in navigation
- **Composite Pattern**: For hierarchical company structures
- **Adapter Pattern**: For integrating incompatible payment services
- **Observer Pattern**: For reactive GUI components