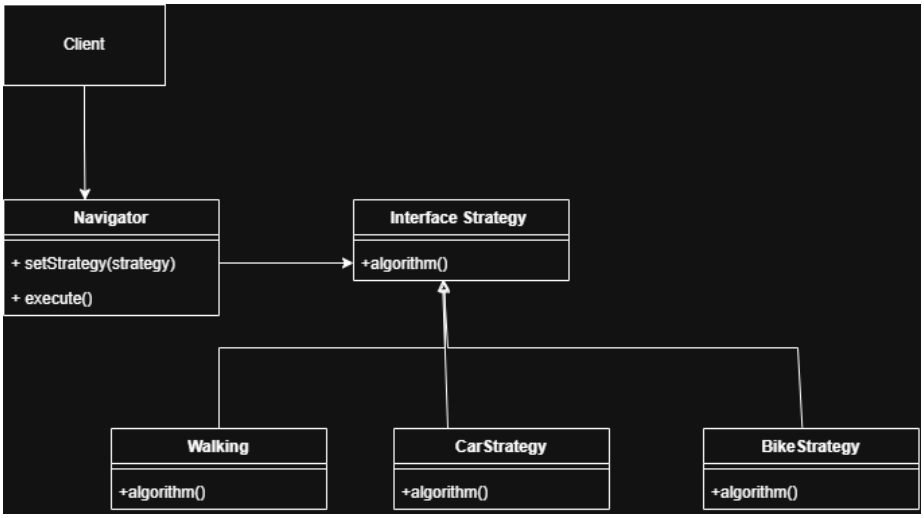
Design Patterns TP

El Gourji Nasreddine

November 2025

1 EX 1 :

1.1 Class diagram :



1.2 Questions :

[label=b)]

1. Navigator : Context
2. because depending on the User inputs the execution changes
3. Three main ones :
   (a) Single Responsibility Principle : each startegy class implements
   (b) OCP we can add new features as strategies

1.3 Code :

```java
public class Navigator{
    private RouteStrategy route ;
    public void setStrategy(RouteStrategy route){
        this.route = route ;
    }

    public void execute(){
        if (route == null){
            System.out.println(x: "Choose the strat first ");
        }
        route.algorithm();
    }


}
```

```java
public interface  RouteStrategy {
    void algorithm();
}
```

```java
public class Walking implements RouteStrategy{
    @Override
    public void algorithm(){
        System.err.println(x: "Walking  calcul .....");
    }
}
```

```java
public class Car implements RouteStrategy{
    @Override
    public void algorithm(){
        System.err.println(x: "Car  calcul .....");
    }
}
```

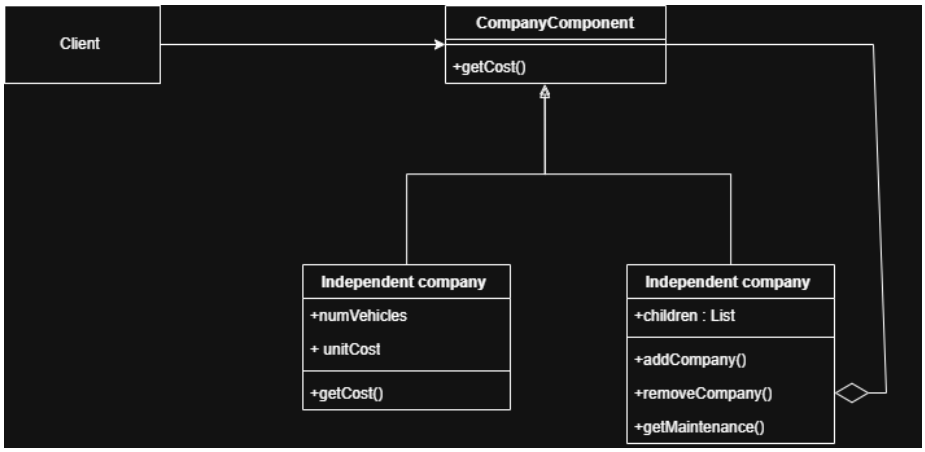
```java
public class Bike implements RouteStrategy{
    @Override
    public void algorithm(){
        System.err.println(x: "Bike calcul .....");
    }
}
```

2 EX 2 :

2.1 Best design pattern :

The most suitable design pattern for this case is the Composite pattern, since the maintenance cost must be computed from nested components.

2.2 Class diagram :



2.3 Code :

```java
public abstract class CompanyComponent {
    public abstract double getMaintenanceCost();
}
```

```java
public class IndependentCompany extends CompanyComponent {

    private int numberOfVehicles;
    private double unitCost;

    public IndependentCompany(int numberOfVehicles, double unitCost) {
        this.numberOfVehicles = numberOfVehicles;
        this.unitCost = unitCost;
    }

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

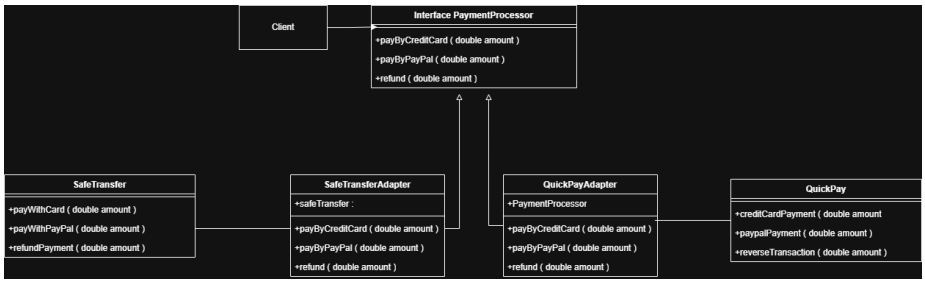
```java
public class IndependentCompany extends CompanyComponent {

    private int numberOfVehicles;
    private double unitCost;

    public IndependentCompany(int numberOfVehicles, double unitCost) {
        this.numberOfVehicles = numberOfVehicles;
        this.unitCost = unitCost;
    }

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

3 EX 3 :

3.1 Best design pattern :

The most suitable design pattern for this case is the Adapter one since we have two new methods that we should adapt to the existing interface .

3.2 Class diagram :



Participants — Adapter Pattern

- **Target:** `PaymentProcessor` — expected interface.
- **Adaptees:** `QuickPay`, `SafeTransfer` — existing payment services.
- **Adapters:** `QuickPayAdapter`, `SafeTransferAdapter` — adapt services to `PaymentProcessor`.
- **Client:** E-commerce platform using `PaymentProcessor`.

3.3 Code :

```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);
    }
}
```

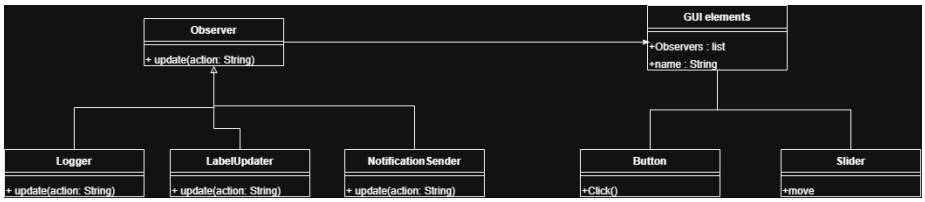
```java
public class SafeTransferAdapter implements PaymentProcessor {

    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) {
        safeTransfer.refundPayment(amount);
    }
}
```

4 EX 4 :

4.1 Best design pattern :

The most suitable design pattern for this case is the Adapter one since we have two new methods that we should adapt to the existing interface .

4.2 Class diagram :



4.3 Code :

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

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

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

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

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

public abstract class GUIElement {
    private List<Observer> observers = new ArrayList<>();
    protected String name;

    public GUIElement(String name) {
        this.name = name;
    }

    public void attach(Observer o) {
        observers.add(o);
    }

    public void detach(Observer o) {
        observers.remove(o);
    }

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

```java
public class Button extends GUIElement {
    public Button(String name) {
        super(name);
    }

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

class Slider extends GUIElement {
    public Slider(String name) {
        super(name);
    }

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