

TP08

Salma Bellaou

November 2025

1 Exercise 1

Task 1

1. The navigator is in this case the context.
2. Navigator depends on the RouteStrategy interface so it can use any routing algorithm through a single abstraction without depending on specific strategy implementations.
3. Strategy pattern.

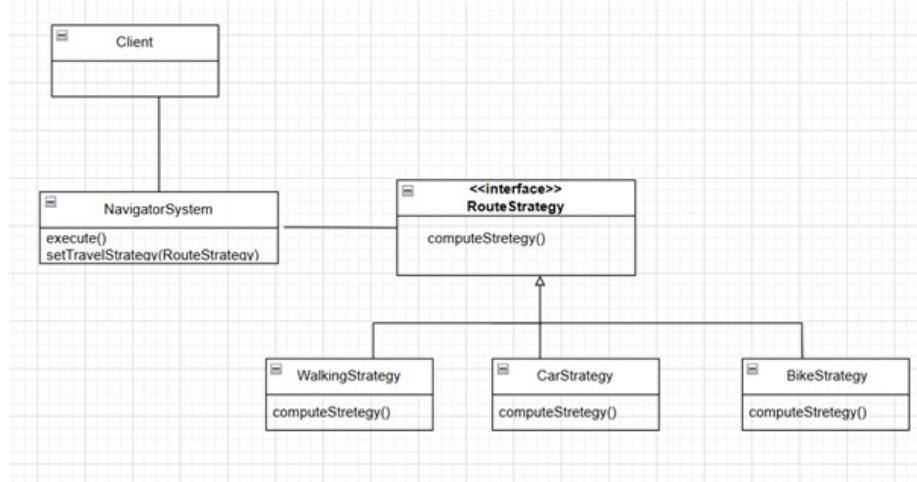


Figure 1: Class diagram

```
//exercise 01:

public interface RouteStrategy{ 6 usages 3 implementations
    public void ComputeStrategy();  no usages 3 implementations
}
class WalkingStrategy implements RouteStrategy{ no usages
    public void ComputeStrategy(){ no usages
        System.out.println("Walking strategy");
    }
}
class CarStrategy implements RouteStrategy{ 1 usage
    public void ComputeStrategy(){ no usages
        System.out.println("Car strategy");
    }
}
class BikeStrategy implements RouteStrategy{ 1 usage
    public void ComputeStrategy(){ no usages
        System.out.println("Bike strategy");
    }
}
```

Figure 2: Implementation

2 Exercise 2

1. We will use composite pattern.

```

class SystemNavigator{ 2 usages
    private RouteStrategy strategy; 2 usages

    void setRouteStrategy(RouteStrategy strategy){ 2 usages
        this.strategy = strategy;
    }
    public void execute(){ 2 usages
        strategy.ComputeStrategy();
    }
}

public class TP07 {
    public static void main(String[] args){
        SystemNavigator sn = new SystemNavigator();
        sn.setRouteStrategy(new CarStrategy());
        sn.execute();
        sn.setRouteStrategy(new BikeStrategy());
        sn.execute();
    }
}

```

Figure 3: Implementation

3 Exercise 3

1. We will use the adapter pattern.
2. Participants:
 - Client
 - Target: Payment Processor
 - Adapter 01: Quick Pay Adapter
 - Adapter 02: Safe Transfer Adapter
 - Adaptee 01: Quick Pay
 - Adaptee 02: Safe Transfer

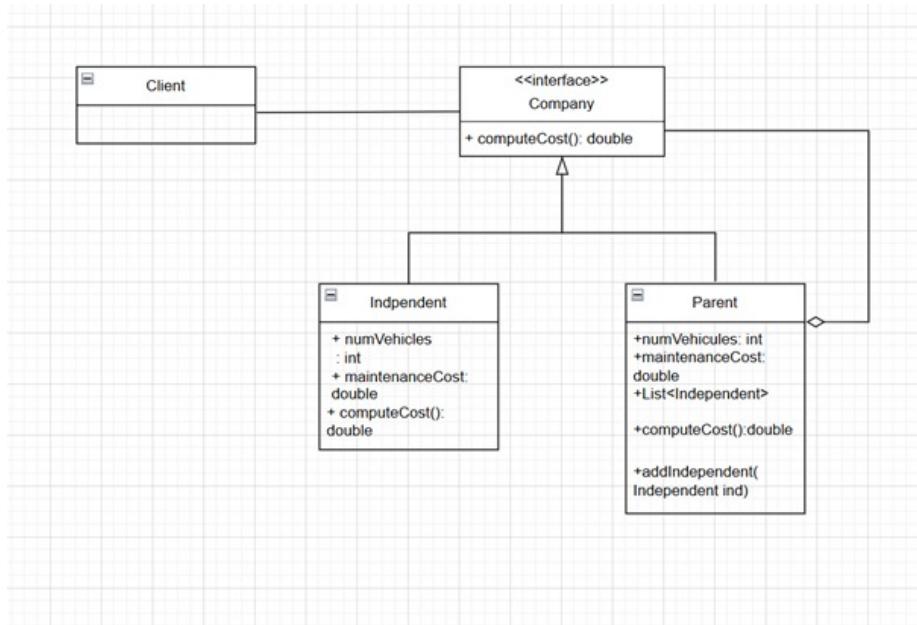


Figure 4: Class diagram(exercise 2)

```

import java.util.*;
interface Company{ 2 usages 2 implementations
    double computeCost(); 2 usages 2 implementations
}
class Independent implements Company{ 6 usages
    int numVehicles; 2 usages
    double MaintenanceCost; 2 usages
    public Independent(int numVehicles, double MaintenanceCost){ 2 usages
        this.numVehicles = numVehicles;
        this.MaintenanceCost = MaintenanceCost;
    }

    public double computeCost(){ 2 usages
        return numVehicles*MaintenanceCost;
    }
}

```

Figure 5: Composite and leaf implementation(exercise2)

```

class Parent implements Company{ 2 usages
    int numVehicles; 2 usages
    double MaintenanceCost; 2 usages
    List<Independent> independents; 1 usage
    public Parent(int numVehicles, double MaintenanceCost){ 1 usage
        this.numVehicles = numVehicles;
        this.MaintenanceCost = MaintenanceCost;
    }
    public double computeCost(){ 2 usages
        return numVehicles*MaintenanceCost;
    }
    public void addIndependent(Independent ind){ no usages
        independents.add(ind);
    }
}
public class TP07 {
    public static void main(String[] args){
        Parent p = new Parent( numVehicles: 100, MaintenanceCost: 50000 );
        Independent i1 = new Independent( numVehicles: 3, MaintenanceCost: 30000 );
        Independent i2 = new Independent( numVehicles: 7, MaintenanceCost: 30000 );
        System.out.println(i1.computeCost());
        System.out.println(p.computeCost());
    }
}

```

Figure 6: Component implementation(exercise 2)

4 Exercise 04

1. We will choose the observer pattern, because Multiple components (Logger, LabelUpdater, NotificationSender) need to react to changes in GUI elements (Buttons, Sliders)

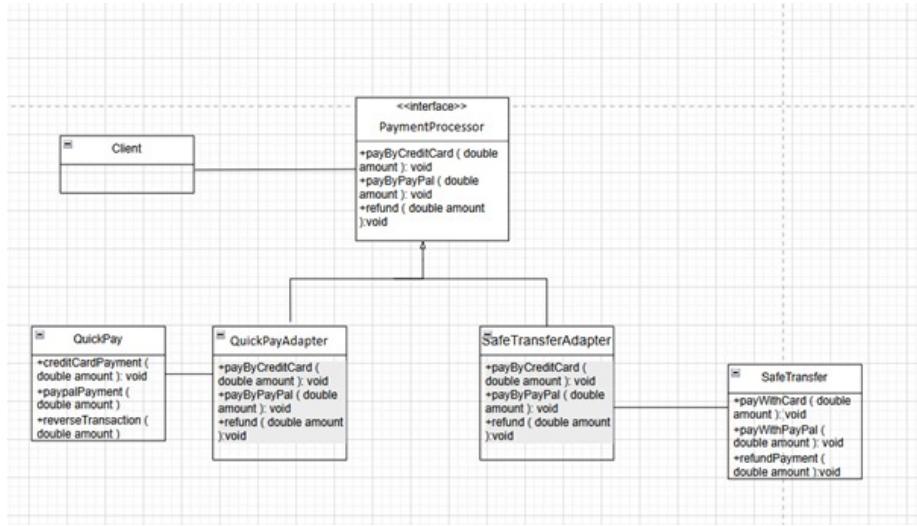


Figure 7: Class diagram(exercise 3)

```

interface PaymentProcessor { 3 usages 2 implementations
    void payByCreditCard ( double amount ) ; 2 usages 2 implementations
    void payByPayPal ( double amount ) ; 2 usages 2 implementations
    void refund ( double amount ) ; 2 usages 2 implementations
}
class QuickPay { 2 usages
    public void creditCardPayment(double amount) { 1 usage
        System.out.println(" QuickPay : Processing credit card payment $" +
                           amount);
    }

    public void paypalPayment(double amount) { 1 usage
        System.out.println(" QuickPay : Processing PayPal payment $" +
                           amount);
    }

    public void reverseTransaction(double amount) { 1 usage
        System.out.println(" QuickPay : Reversing transaction $" + amount);
    }
}

```

Figure 8: Target and adaptee 01 implementation(exercise 3)

```
class SafeTransfer { 2 usages
    public void payWithCard ( double amount ) { 1 usage
        System.out.println(" SafeTransfer : Paying with credit card $" +
                           amount);
    }
    public void payWithPayPal ( double amount ) { 1 usage
        System . out . println (" SafeTransfer : Paying with PayPal $" + amount );
    }
    public void refundPayment ( double amount ) { 1 usage
        System . out . println (" SafeTransfer : Refunding payment $" + amount ) ;
    }
}
```

Figure 9: Adaptee 02(exercise 3)

```
class QuickPayAdapter implements PaymentProcessor { 1 usage

    private QuickPay quickPay; 4 usages
    public QuickPayAdapter() { 1 usage
        this.quickPay = new QuickPay();
    }
    public void payByCreditCard(double amount) { 2 usages
        quickPay.creditCardPayment(amount);
    }
    public void payByPayPal(double amount) { 2 usages
        quickPay.paypalPayment(amount);
    }
    public void refund(double amount) { 2 usages
        quickPay.reverseTransaction(amount);
    }
}
```

Figure 10: Adapter 01(exercise 03)

```

class SafeTransferAdapter implements PaymentProcessor { 1 usage

    private SafeTransfer safeTransfer; 4 usages

    public SafeTransferAdapter() { 1 usage
        this.safeTransfer = new SafeTransfer();
    }

    public void payByCreditCard(double amount) { 2 usages
        safeTransfer.payWithCard(amount);
    }

    public void payByPayPal(double amount) { 2 usages
        safeTransfer.payWithPayPal(amount);
    }

    public void refund(double amount) { 2 usages
        safeTransfer.refundPayment(amount);
    }
}

```

Figure 11: Adapter 02(exercise 3)

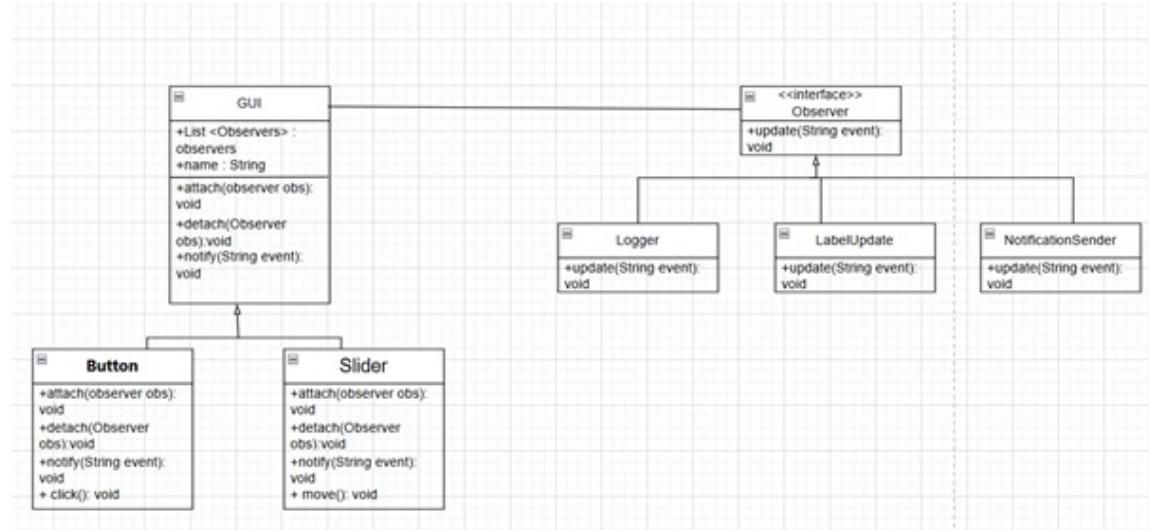


Figure 12: Class diagram (Exercise 04)

```
import java.util.*;

interface Observer { 7 usages 3 implementations
    void update(String event); 1 usage 3 implementations
}
class GUI { 2 usages 2 inheritors
    private List<Observer> observers; 4 usages
    private String name; 4 usages

    public GUI(String name) { 2 usages
        this.name = name;
        this.observers = new ArrayList<>();
    }

    public void attach(Observer obs) { 9 usages
        observers.add(obs);
        System.out.println("Observer attached to " + name);
    }
}
```

Figure 13: Observer and subject implementation(exercise 04)

```
public void attach(Observer obs) { 9 usages
    observers.add(obs);
    System.out.println("Observer attached to " + name);
}

public void detach(Observer obs) { 1 usage
    observers.remove(obs);
    System.out.println("Observer detached from " + name);
}

public void notify(String event) { 2 usages
    for (Observer observer : observers) {
        observer.update(event);
    }
}
```

Figure 14: Subject methods(exercise 4)

```
class Button extends GUI { 4 usages

    public Button(String name) { 2 usages
        super(name);
    }

    public void click() { 2 usages
        System.out.println("\n" + getName() + " was clicked!");
        notify( event: "CLICK");
    }
}
```

Figure 15: Concrete subject implementation(exercise 4)

```
class Slider extends GUI { 4 usages

    public Slider(String name) { 2 usages
        super(name);
    }

    public void move() { 3 usages
        System.out.println("\n" + getName() + " was moved!");
        notify( event: "MOVE");
    }
}
```

Figure 16: Concrete subject(exercise 4)

```

class Logger implements Observer { 2 usages

    @Override 1 usage
    public void update(String event) {
        System.out.println("Logger: Logging event - " + event);
    }
}

class LabelUpdate implements Observer { 2 usages
    public void update(String event) { 1 usage
        System.out.println("LabelUpdate: Updating label - " + event);
    }
}

class NotificationSender implements Observer { 2 usages

    public void update(String event) { 1 usage
        System.out.println("NotificationSender: Sending alert for " + event);
    }
}

```

Figure 17: concrete observers(exercise 4)

The screenshot shows a Java code editor with the following code:

```

73 class NotificationSender implements Observer { no usages
74
75     public void update(String event) { 1 usage
76         System.out.println("NotificationSender: Sending alert for " + event);
77     }
78 }
79
80
81

```

Below the code, there is a terminal window showing the output of a test run:

```

tests passed 2 tests total, 43 ms
\Users\PC GOLD\.jdks\ms-21.0.8\bin\java.exe" ...

```

Figure 18: valid test (exercise 4)