Design Patterns: Part 2
Lab Report

Mohamed Amine El Kalai

November 27, 2025

Contents

1 Exercise 1: Strategy Pattern

1.1 Task 1: Questions

Question 1: What role does the Navigator class play?

Navigator play the role of **Context**. It hold a reference to RouteStrategy and delegate the route calculation to the strategy.

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

Navigator depend on RouteStrategy interface to use different strategy without knowing the concrete implementation. We can change strategy at runtime without modify Navigator code.

Question 3: Which SOLID principles are applied?

Open/Closed Principle: Navigator is open for extension but closed for modification. **Dependency Inversion**: Navigator depend on abstraction not concrete class. **Single Responsibility**: Each strategy have one responsability.

1.2 Task 2: Implementation

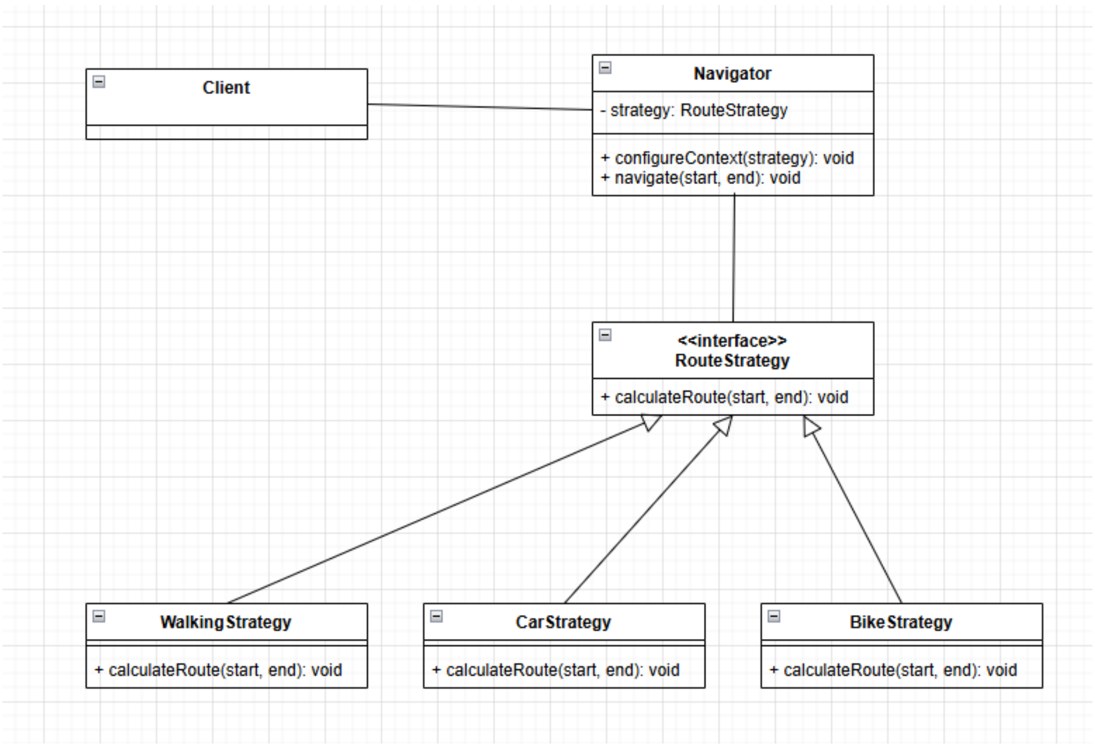
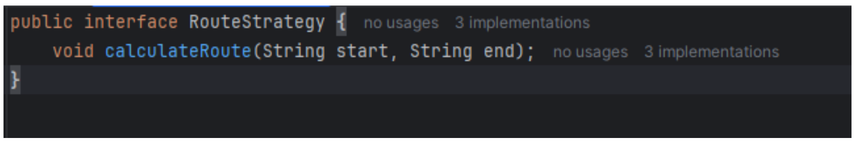
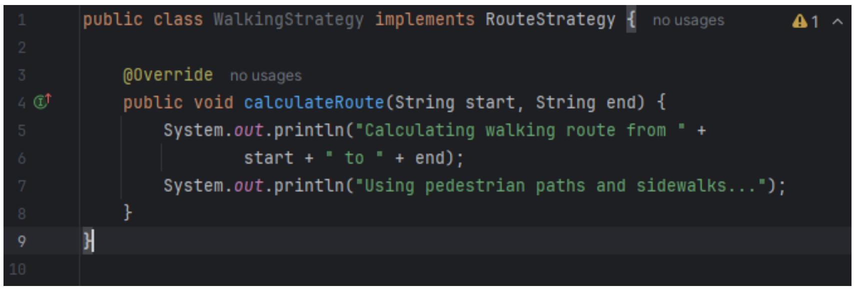

Client
Navigator
- strategy: RouteStrategy
+ configureContext(strategy): void
+ navigate(start, end): void
<<interface>>
RouteStrategy
+ calculateRoute(start, end): void
WalkingStrategy
+ calculateRoute(start, end): void
CarStrategy
+ calculateRoute(start, end): void
BikeStrategy
+ calculateRoute(start, end): void

Figure 1: Strategy Pattern Class Diagram



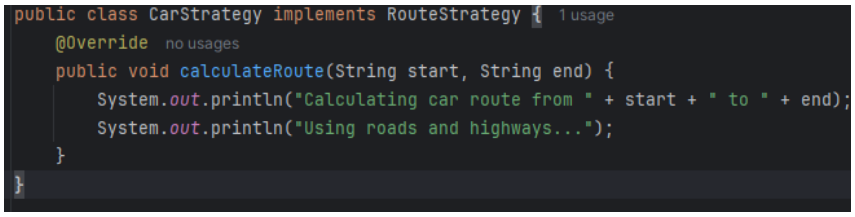
```
public interface RouteStrategy {  no usages  3 implementations
    void calculateRoute(String start, String end);  no usages  3 implementations
}
```

Figure 2: RouteStrategy Interface



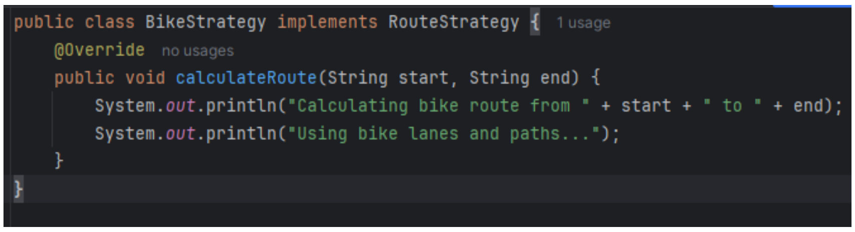
```java
public class WalkingStrategy implements RouteStrategy {  no usages

    @Override  no usages
    public void calculateRoute(String start, String end) {
        System.out.println("Calculating walking route from " +
                start + " to " + end);
        System.out.println("Using pedestrian paths and sidewalks...");
    }
}
```

Figure 3: WalkingStrategy



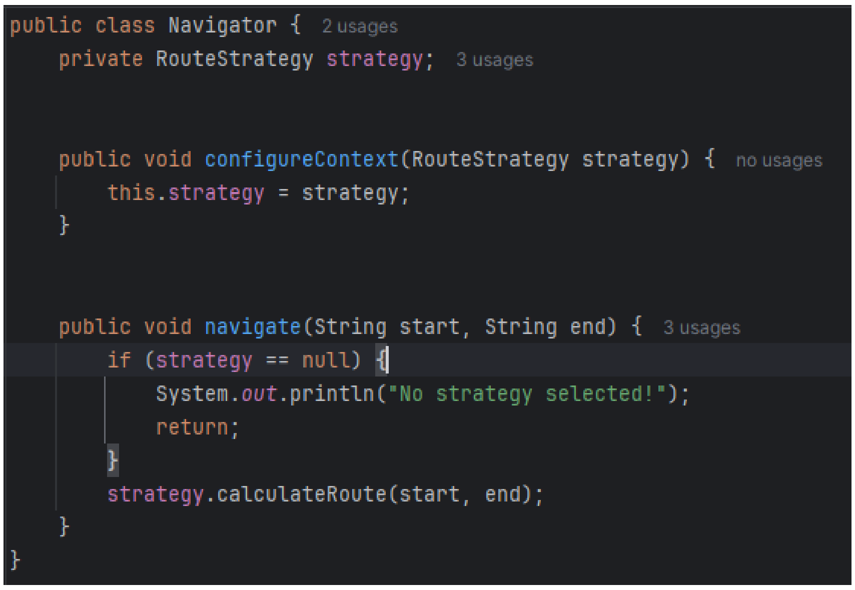
```java
public class CarStrategy implements RouteStrategy {  1 usage
    @Override  no usages
    public void calculateRoute(String start, String end) {
        System.out.println("Calculating car route from " + start + " to " + end);
        System.out.println("Using roads and highways...");
    }
}
```

Figure 4: CarStrategy



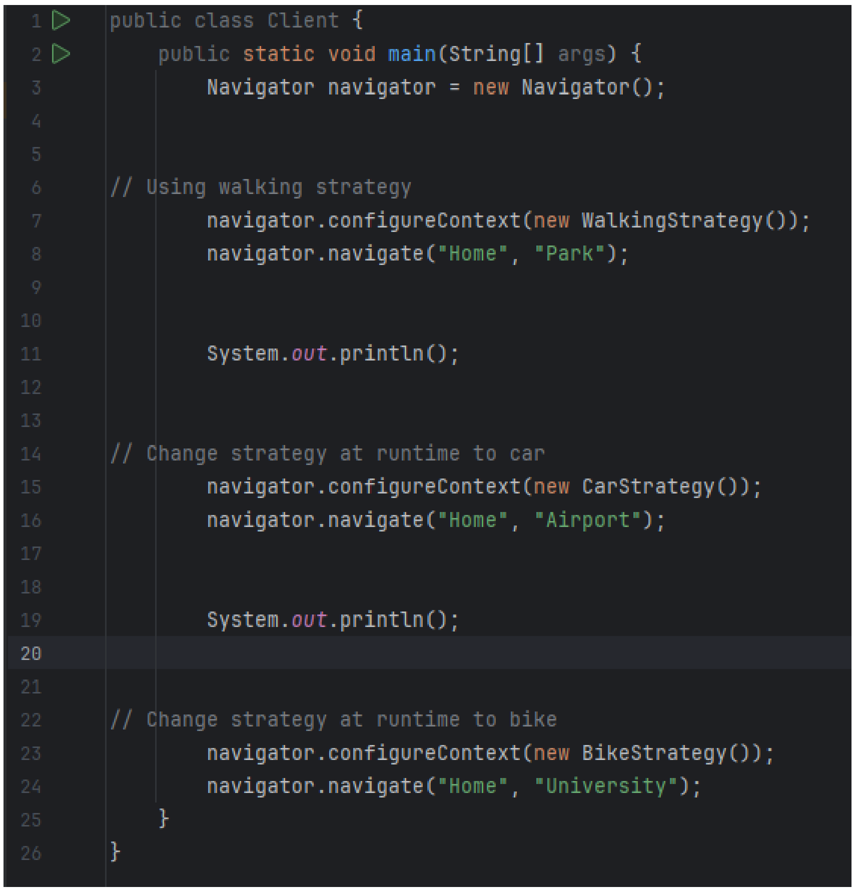
```java
public class BikeStrategy implements RouteStrategy {  1 usage
    @Override  no usages
    public void calculateRoute(String start, String end) {
        System.out.println("Calculating bike route from " + start + " to " + end);
        System.out.println("Using bike lanes and paths...");
    }
}
```

Figure 5: BikeStrategy



```java
public class Navigator {  2 usages
    private RouteStrategy strategy;  3 usages


    public void configureContext(RouteStrategy strategy) {  no usages
        this.strategy = strategy;
    }


    public void navigate(String start, String end) {  3 usages
        if (strategy == null) {
            System.out.println("No strategy selected!");
            return;
        }
        strategy.calculateRoute(start, end);
    }
}
```

Figure 6: Navigator Class



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


// Using walking strategy
        navigator.configureContext(new WalkingStrategy());
        navigator.navigate("Home", "Park");


        System.out.println();


// Change strategy at runtime to car
        navigator.configureContext(new CarStrategy());
        navigator.navigate("Home", "Airport");


        System.out.println();


// Change strategy at runtime to bike
        navigator.configureContext(new BikeStrategy());
        navigator.navigate("Home", "University");
    }
}
```

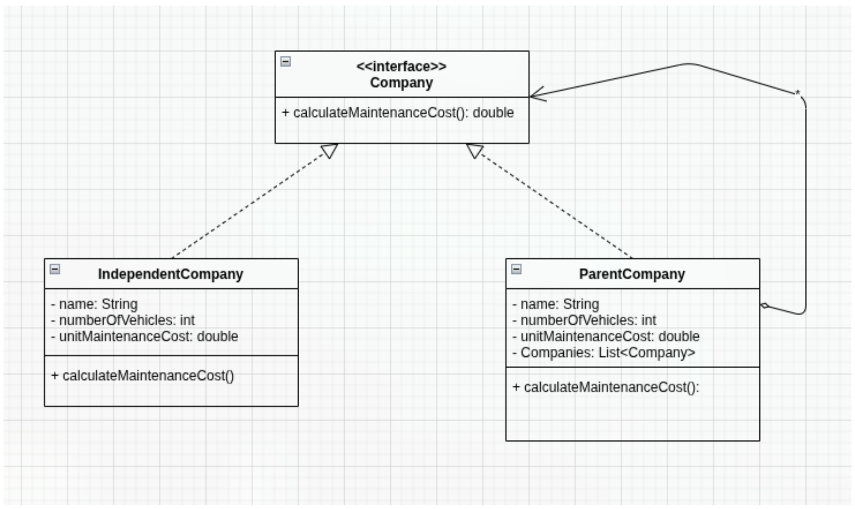
Figure 7: Client Class

2 Exercise 2: Composite Pattern

2.1 Question 1: Which design pattern is best suited?

The **Composite Pattern** is best suited. We have a tree structure where Independent companies are leaf nodes and Parent companies are composite nodes that contain other companies. We need to calculate maintenance cost in uniform way for both type.

2.2 Question 2: Class Diagram



<<interface>>
Company
+ calculateMaintenanceCost(): double
IndependentCompany
- name: String
- numberOfVehicles: int
- unitMaintenanceCost: double
+ calculateMaintenanceCost()
ParentCompany
- name: String
- numberOfVehicles: int
- unitMaintenanceCost: double
- Companies: List<Company>
+ calculateMaintenanceCost():
*

Figure 8: Composite Pqttern Class Diagram

2.3 Question 3: Java Implementation

2.3.1 Company Interface

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

2.3.2 IndependentCompany Class

```java
public class IndependentCompany implements Company {
    private String name;
    private int numberOfVehicles;
    private double unitMaintenanceCost;

    public IndependentCompany(String name, int numberOfVehicles,
       double unitMaintenanceCost) {
        this.name = name;
        this.numberOfVehicles = numberOfVehicles;
        this.unitMaintenanceCost = unitMaintenanceCost;
    }

    @Override
    public double calculateMaintenanceCost() {
        return numberOfVehicles * unitMaintenanceCost;
    }

}
```

2.3.3 ParentCompany Class

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

public class ParentCompany implements Company {
    private String name;
    private List<Company> Companies;
    private int numberOfVehicles;
    private double unitMaintenanceCost;

    public ParentCompany(String name, int numberOfVehicles,
       double unitMaintenanceCost) {
        this.name = name;
        this.numberOfVehicles = numberOfVehicles;
        this.unitMaintenanceCost = unitMaintenanceCost;
        this.Companies = new ArrayList<>();
    }

    public void addCompany(Company company) {
        Companies.add(company);
    }

    public void removeCompany(Company company) {
        Companies.remove(company);
    }

    @Override
    public double calculateMaintenanceCost() {
        double totalCost = numberOfVehicles * unitMaintenanceCost
            ;
        for (Company comp : Companies) {
            totalCost += comp.calculateMaintenanceCost();
        }
        return totalCost;
    }

}
```

3 Exercise 3: Adapter Pattern

3.1 Question 1: Which design pattern should you use?

The **Adapter Pattern** should be used. This pattern allow us to make incompatible interface work together. We have two existing payment service with different API, and we want them to work with our standard PaymentProcessor interface without modify the existing services.

3.2 Question 2: Participants and Class Diagram

Participants: **Target**: PaymentProcessor interface. **Adaptee**: QuickPay and SafeTransfer. **Adapter**: QuickPayAdapter and SafeTransferAdapter. **Client**: The code that use PaymentProcessor interface.

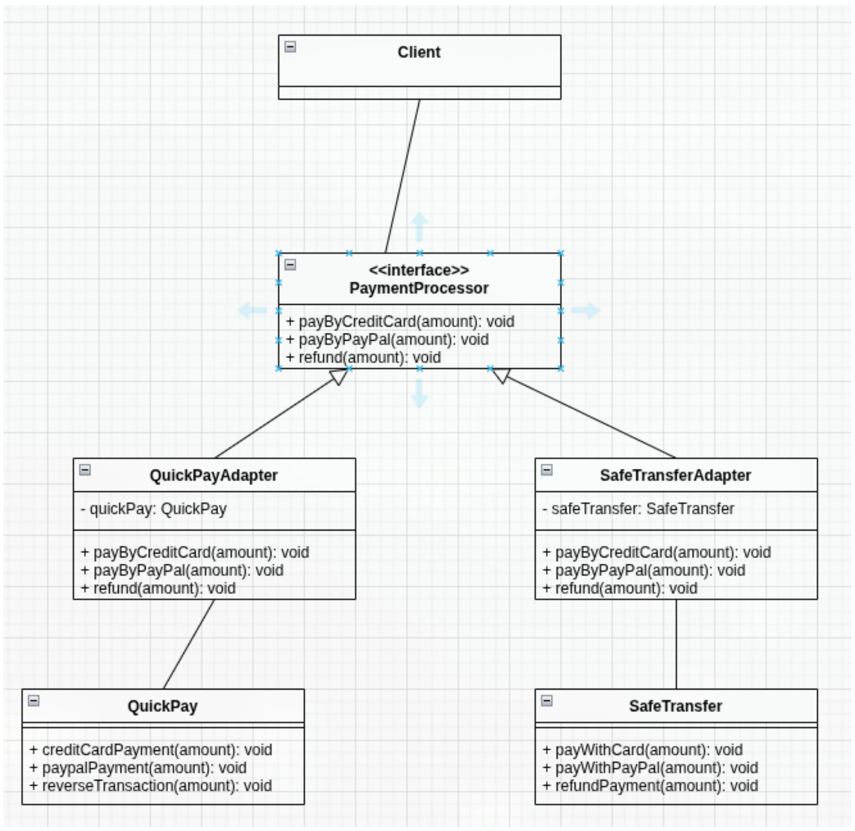

Client
<<interface>>
PaymentProcessor
+ payByCreditCard(amount): void
+ payByPayPal(amount): void
+ refund(amount): void
QuickPayAdapter
- quickPay: QuickPay
+ payByCreditCard(amount): void
+ payByPayPal(amount): void
+ refund(amount): void
SafeTransferAdapter
- safeTransfer: SafeTransfer
+ payByCreditCard(amount): void
+ payByPayPal(amount): void
+ refund(amount): void
QuickPay
+ creditCardPayment(amount): void
+ paypalPayment(amount): void
+ reverseTransaction(amount): void
SafeTransfer
+ payWithCard(amount): void
+ payWithPayPal(amount): void
+ refundPayment(amount): void

Figure 9: Class Diagram

3.3 Question 3: Java Implementation

3.3.1 PaymentProcessor Interface

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

3.3.2 QuickPayAdapter Class

```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);
    }
}
```

3.3.3 SafeTransferAdapter Class

```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 Exercise 4: Observer Pattern

4.1 Question 1: Which design pattern is most suitable and why?

The **Observer Pattern** is most suitable. We have GUI elements that change state and multiple component need to react to these change. We want loose coupling between the GUI element and the component. The Observer pattern define a one-to-many dependency so when one object change state, all its dependent are notify automatically.

4.2 Question 2: Class Diagram

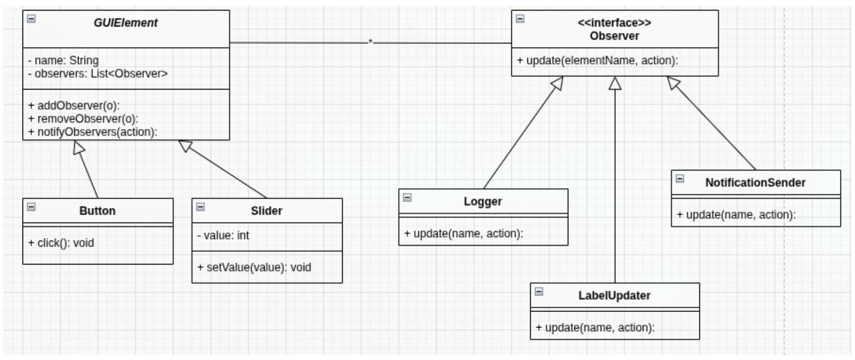


Figure 10: Class Diagram

4.3 Question 3: Java Implementation

4.3.1 Observer Interface

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

4.3.2 GUIElement Abstract Class

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

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

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

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    public void notifyObservers(String action) {
        for (Observer observer : observers) {
            observer.update(name, action);
        }
    }
}
```

4.3.3 Button Class

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

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

4.3.4 Slider Class

```java
public class Slider extends GUIElement {
    private int value;

    public Slider(String name) {
        super(name);
        this.value = 0;
    }

    public void setValue(int newValue) {
        this.value = newValue;
        System.out.println(name + " moved to " + newValue);
        notifyObservers("moved to " + newValue);
    }
}
```

4.3.5 Logger Class

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

4.3.6 LabelUpdater Class

```
public class LabelUpdater implements Observer {
    @Override
    public void update(String elementName, String action) {
        System.out.println("LabelUpdater: Updated label - Last
           action: " + elementName + " " + action);
    }
}
```

4.3.7 NotificationSender Class

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