Design Patterns Laboratory Report
Strategy, Composite, Adapter & Observer Patterns

Mohamed Amine El Bacha
Course: Software Engineering
Institution: UM6P - College of Computing

November 25, 2025

Contents

0.1 Introduction

This laboratory report presents the implementation and analysis of four fundamental design patterns in software engineering: the Strategy pattern, the Composite pattern, the Adapter pattern, and the Observer pattern. These patterns offer solutions to common design problems in object-oriented programming, promoting code reusability, maintainability, and scalability.

The Strategy pattern enables runtime selection of algorithms without modifying client code. The Composite pattern unifies the treatment of single objects and compositions of objects. The Adapter pattern allows incompatible interfaces to work together. The Observer pattern supports event-driven communication between objects.

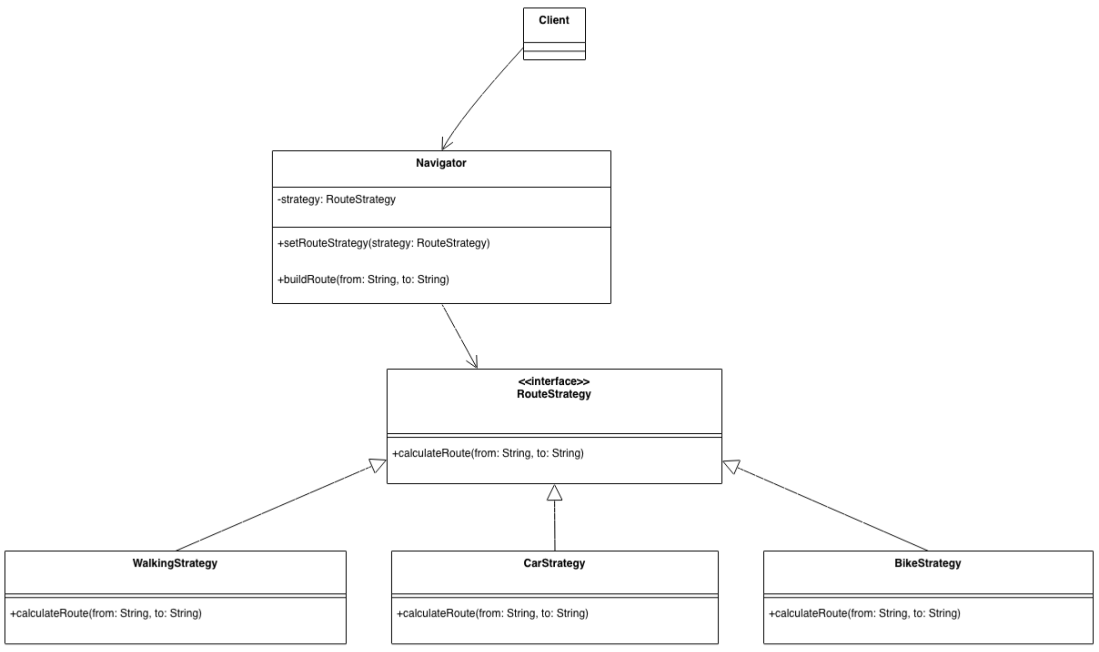
This report demonstrates practical implementations of these patterns through four progressive exercises, each addressing real-world software design challenges.

0.2 Exercise 1: Strategy Pattern - Navigation System

0.2.1 Objective

Implement a navigation system that can calculate routes using different algorithms (walking, car, bike). The system must allow switching strategies at runtime without modifying the Navigator class. The goal is to apply the Strategy Pattern to make the system flexible and maintainable.

0.2.2 Task 1: Class Diagram


Client
Navigator
-strategy: RouteStrategy
+setRouteStrategy(strategy: RouteStrategy)
+buildRoute(from: String, to: String)
<<interface>>
RouteStrategy
+calculateRoute(from: String, to: String)
WalkingStrategy
+calculateRoute(from: String, to: String)
CarStrategy
+calculateRoute(from: String, to: String)
BikeStrategy
+calculateRoute(from: String, to: String)

0.2.3 Task Questions

Question 1: What role does the Navigator class play?

The Navigator class plays the role of the **Context** in the Strategy design pattern. Specifically, the Navigator:

- Holds a reference to a RouteStrategy object (the strategy interface)
- Delegates the route calculation task to the current strategy object
- Does not implement the routing algorithm itself
- Provides methods to set or change the strategy at runtime (setRouteStrategy)
- Maintains the client-facing interface (buildRoute method)

The Context acts as the intermediary between the client and the concrete strategies, decoupling the algorithm implementation from the code that uses it.

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

The Navigator depends on the RouteStrategy interface rather than concrete implementations for several important reasons:

- **Flexibility:** By depending on an abstraction (interface), the Navigator can work with any routing algorithm that implements RouteStrategy without knowing the specific implementation details
- **Interchangeability:** Different concrete strategies can be swapped at runtime without modifying the Navigator's code
- **Dependency Inversion Principle:** High-level modules (Navigator) depend on abstractions (RouteStrategy), not on low-level concrete implementations
- **Loose Coupling:** The Navigator and concrete strategies are loosely coupled, making the system more maintainable and testable
- **Extensibility:** New routing strategies can be added by implementing the interface without changing existing code

This design allows the algorithm to vary independently from clients that use it, which is the core principle of the Strategy pattern.

Question 3: Which SOLID principles are applied?

This design applies several SOLID principles:

- **Single Responsibility Principle (SRP):** Each concrete strategy class has one responsibility - implementing its specific routing algorithm. The Navigator has one responsibility - managing route calculation by delegating to strategies. The client has one responsibility - selecting and configuring the appropriate strategy.

- **Open/Closed Principle (OCP):** The system is open for extension but closed for modification. New routing strategies (e.g., PublicTransportStrategy, ScooterStrategy) can be added by creating new classes that implement RouteStrategy, without modifying the Navigator class or existing strategies.
- **Liskov Substitution Principle (LSP):** Any concrete strategy (WalkingStrategy, CarStrategy, BikeStrategy) can be substituted for the RouteStrategy interface without affecting the correctness of the Navigator's behavior.
- **Interface Segregation Principle (ISP):** The RouteStrategy interface is focused and contains only the method(s) necessary for route calculation, preventing clients from depending on methods they don't use.
- **Dependency Inversion Principle (DIP):** The high-level Navigator class depends on the RouteStrategy abstraction, not on concrete strategy implementations. Both high-level and low-level modules depend on the same abstraction.

0.2.4 Task 2: Implementation

RouteStrategy Interface

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

Listing 1: RouteStrategy.java - Strategy Interface

Explanation: The RouteStrategy interface defines the contract that all routing algorithms must implement. This abstraction allows the Navigator to work with any routing strategy without knowing the specific implementation details. The calculateRoute method takes origin and destination parameters and encapsulates the algorithm for computing routes.

WalkingStrategy Class

```java
public class WalkingStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String from, String to) {
        System.out.println("Calculating WALKING route from "
            + from + " to " + to);
        System.out.println("- Using pedestrian paths");
        System.out.println("- Avoiding highways");
        System.out.println("- Including stairs and shortcuts");
    }
}
```

Listing 2: WalkingStrategy.java - Concrete Strategy

Explanation: WalkingStrategy implements the RouteStrategy interface with logic specific to pedestrian navigation. It considers factors like sidewalks, pedestrian crossings, stairs, and walking-only paths. This strategy would typically optimize for shortest distance rather than speed.

CarStrategy Class

```java
public class CarStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String from, String to) {
        System.out.println("Calculating CAR route from "
            + from + " to " + to);
        System.out.println("- Using highways and main roads");
        System.out.println("- Optimizing for speed");
        System.out.println("- Checking traffic conditions");
    }
}
```

Listing 3: CarStrategy.java - Concrete Strategy

Explanation: CarStrategy implements vehicle-specific routing that prioritizes highways and faster routes suitable for car travel. It considers traffic conditions, road types, and speed limits to optimize travel time.

BikeStrategy Class

```java
public class BikeStrategy implements RouteStrategy {
    @Override
    public void calculateRoute(String from, String to) {
        System.out.println("Calculating BIKE route from "
            + from + " to " + to);
        System.out.println("- Using bike lanes and paths");
        System.out.println("- Avoiding steep hills");
        System.out.println("- Preferring scenic routes");
    }
}
```

Listing 4: BikeStrategy.java - Concrete Strategy

Explanation: BikeStrategy implements cycling-specific routing that considers bike lanes, terrain difficulty, and elevation changes. It balances distance with ride difficulty to provide cyclist-friendly routes.

Navigator Class (Context)

```java
public class Navigator {
    private RouteStrategy routeStrategy;

    // Constructor accepts initial strategy
    public Navigator(RouteStrategy routeStrategy) {
        this.routeStrategy = routeStrategy;
    }

    // Allows strategy to be changed at runtime
    public void setRouteStrategy(RouteStrategy routeStrategy) {
        this.routeStrategy = routeStrategy;
    }

    // Delegates route calculation to current strategy
    public void buildRoute(String from, String to) {
        if (routeStrategy == null) {
            throw new IllegalStateException(
                "RouteStrategy not set. Please set a strategy first.");
        }
        System.out.println("\n=== Building Route ===");
        routeStrategy.calculateRoute(from, to);
        System.out.println("==================\n");
    }
}
```

Listing 5: Navigator.java - Context Class

Explanation: The Navigator class acts as the Context in the Strategy pattern. It maintains a reference to a RouteStrategy object and delegates the route calculation to it. The key features are:

- The constructor accepts an initial strategy, ensuring the Navigator is always in a valid state
- The setRouteStrategy() method allows the strategy to be changed at runtime, demonstrating the pattern's flexibility
- The buildRoute() method delegates to the current strategy without knowing implementation details
- Error checking ensures a strategy is set before attempting to build a route

Client Implementation

```java
public class NavigationDemo {
    public static void main(String[] args) {
        System.out.println("========================================");
        System.out.println("   NAVIGATION SYSTEM - STRATEGY DEMO   ");
        System.out.println("========================================\n")
            ;

        // Create navigator with initial car strategy
        Navigator navigator = new Navigator(new CarStrategy());

        // Trip 1: Using car
        System.out.println("--- Trip 1: Traveling by Car ---");
        navigator.buildRoute("UM6P", "Rabat Center");

        // Trip 2: Change to walking strategy at runtime
        System.out.println("--- Trip 2: Traveling by Walking ---");
        navigator.setRouteStrategy(new WalkingStrategy());
        navigator.buildRoute("UM6P", "Rabat Center");

        // Trip 3: Change to bike strategy
        System.out.println("--- Trip 3: Traveling by Bike ---");
        navigator.setRouteStrategy(new BikeStrategy());
        navigator.buildRoute("UM6P", "Rabat Center");

        // Demonstrate flexibility: same destination, different routes
        System.out.println("--- Trip 4: Different destination by Car
            ---");
        navigator.setRouteStrategy(new CarStrategy());
        navigator.buildRoute("Casablanca", "Marrakech");
    }
}
```

Listing 6: NavigationDemo.java - Client

Explanation: The client demonstrates the Strategy pattern's key benefits:

- Creates a Navigator instance with an initial strategy
- Changes strategies at runtime without modifying the Navigator
- Uses the same Navigator interface regardless of the chosen strategy
- Shows that the same navigator can handle multiple trips with different strategies

This implementation showcases the flexibility and maintainability that the Strategy pattern provides.

0.2.5 Program Output

```
========================================
   NAVIGATION SYSTEM - STRATEGY DEMO
========================================

--- Trip 1: Traveling by Car ---

=== Building Route ===
Calculating CAR route from UM6P to Rabat Center
- Using highways and main roads
- Optimizing for speed
- Checking traffic conditions
===================


--- Trip 2: Traveling by Walking ---

=== Building Route ===
Calculating WALKING route from UM6P to Rabat Center
- Using pedestrian paths
- Avoiding highways
- Including stairs and shortcuts
===================


--- Trip 3: Traveling by Bike ---

=== Building Route ===
Calculating BIKE route from UM6P to Rabat Center
- Using bike lanes and paths
- Avoiding steep hills
- Preferring scenic routes
===================


--- Trip 4: Different destination by Car ---

=== Building Route ===
Calculating CAR route from Casablanca to Marrakech
- Using highways and main roads
- Optimizing for speed
- Checking traffic conditions
==================
```

Listing 7: Output of NavigationDemo.java

0.3 Exercise 2: Composite Pattern - Company Maintenance Costs

0.3.1 Objective

Within a vehicle sales system, we want to represent client companies, in particular to offer them maintenance packages for the vehicles they own. To do this, we need to calculate the maintenance cost of their vehicles, which depends on the number of vehicles a company owns, and the unit maintenance cost per vehicle. Companies can either be independent, or parent companies to which we can add independent companies. It must be possible to calculate the maintenance cost for both parent companies and independent companies in a uniform way.

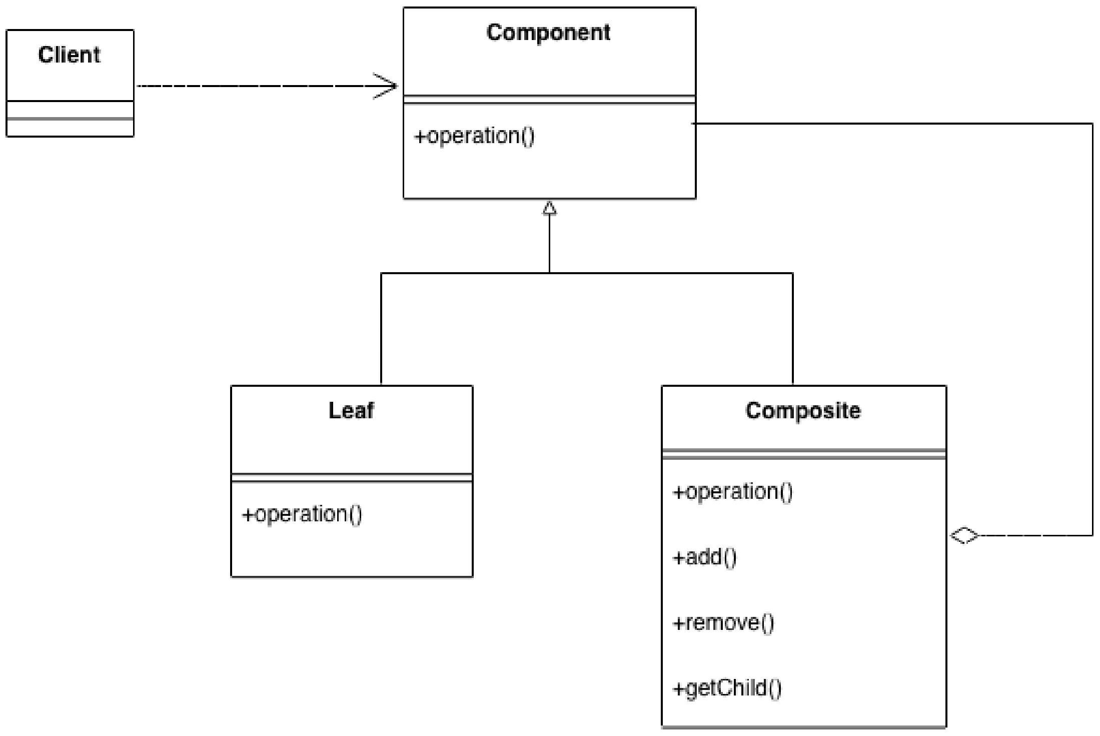
0.3.2 Question 1: Which design pattern is best suited?

The **Composite Pattern** is best suited to model this problem for the following reasons:

- **Part-Whole Hierarchy:** The problem involves a tree structure where parent companies contain other companies (either independent or parent companies), forming a hierarchical relationship
- **Uniform Treatment:** Both independent companies (leaf nodes) and parent companies (composite nodes) must be treated uniformly when calculating maintenance costs
- **Recursive Composition:** Parent companies can contain other parent companies, creating arbitrary depth hierarchies that need recursive cost calculation
- **Single Interface:** Clients should interact with both types of companies through the same interface without needing to distinguish between them
- **Aggregation:** Parent companies aggregate the maintenance costs of all their child companies, which is a classic composite pattern behavior

The Composite pattern allows us to build complex tree structures from simple components and treat individual objects and compositions uniformly through a common interface.

0.3.3 Question 2: Class Diagram



0.3.4 Question 3: Implementation

CompanyComponent Interface

```java
public interface CompanyComponent {
    double getMaintenanceCost();
    String getName();
    void displayInfo(int level);
}
```

Listing 8: CompanyComponent.java - Component Interface

Explanation: The CompanyComponent interface defines the common operations that both leaf nodes (independent companies) and composite nodes (parent companies) must implement. This allows uniform treatment of individual and composite objects. The displayInfo method with a level parameter supports hierarchical display of the company structure.

IndependentCompany Class (Leaf)

```java
public class IndependentCompany implements CompanyComponent {
    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 getMaintenanceCost() {
        return numberOfVehicles * unitMaintenanceCost;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public void displayInfo(int level) {
        String indent = "  ".repeat(level);
        System.out.println(indent + "- " + name);
        System.out.println(indent + "  Vehicles: " + numberOfVehicles);
        System.out.println(indent + "  Unit Cost: $"
            + unitMaintenanceCost);
        System.out.println(indent + "  Total Maintenance: $"
            + getMaintenanceCost());
    }
}
```

Listing 9: IndependentCompany.java - Leaf Node

Explanation: IndependentCompany represents a leaf node in the composite structure. It:

- Stores the number of vehicles and unit maintenance cost
- Calculates its maintenance cost as vehicles × unit cost
- Has no children (cannot contain other companies)
- Implements displayInfo to show its details at the appropriate indentation level

This class represents the base case in the recursive structure.

ParentCompany Class (Composite)

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

public class ParentCompany implements CompanyComponent {
    private String name;
    private List<CompanyComponent> children = new ArrayList<>();

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

    // Add a child company (independent or parent)
    public void add(CompanyComponent company) {
        children.add(company);
        System.out.println("Added " + company.getName()
            + " to " + name);
    }

    // Remove a child company
    public void remove(CompanyComponent company) {
        children.remove(company);
        System.out.println("Removed " + company.getName()
            + " from " + name);
    }

    // Get all children
    public List<CompanyComponent> getChildren() {
        return children;
    }

    @Override
    public double getMaintenanceCost() {
        double total = 0;
        for (CompanyComponent company : children) {
            total += company.getMaintenanceCost();
        }
        return total;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public void displayInfo(int level) {
        String indent = "  ".repeat(level);
        System.out.println(indent + "+ " + name
            + " (Parent Company)");
        System.out.println(indent + "  Total Maintenance: $"
            + getMaintenanceCost());
        System.out.println(indent + "  Children:");

        for (CompanyComponent company : children) {
            company.displayInfo(level + 1);
        }
    }
}
```

Listing 10: ParentCompany.java - Composite Node

Explanation: ParentCompany represents a composite node that can contain other companies (both independent and parent companies). Key features:

- Maintains a list of child companies
- Provides add() and remove() methods for building the hierarchy
- Calculates total maintenance cost by recursively summing children's costs
- Implements displayInfo to show the hierarchy with proper indentation
- Can contain any mix of independent and parent companies

The recursive nature of getMaintenanceCost() allows it to handle arbitrary depth hierarchies.

Client Implementation

```java
public class CompanyDemo {
    public static void main(String[] args) {
        System.out.println("========================================");
        System.out.println("  VEHICLE MAINTENANCE COST CALCULATOR   ");
        System.out.println("========================================\n
            ");

        // Create independent companies (leaf nodes)
        IndependentCompany transportCoA =
            new IndependentCompany("Transport Co A", 10, 500.0);
        IndependentCompany transportCoB =
            new IndependentCompany("Transport Co B", 15, 600.0);
        IndependentCompany transportCoC =
            new IndependentCompany("Transport Co C", 8, 550.0);
        IndependentCompany transportCoD =
            new IndependentCompany("Transport Co D", 12, 500.0);
        IndependentCompany transportCoE =
            new IndependentCompany("Transport Co E", 20, 450.0);

        System.out.println("\n--- Building Company Hierarchy ---\n");

        // Create regional parent companies
        ParentCompany northRegion = new ParentCompany("North Region");
        northRegion.add(transportCoA);
        northRegion.add(transportCoB);

        ParentCompany southRegion = new ParentCompany("South Region");
        southRegion.add(transportCoC);
        southRegion.add(transportCoD);

        // Create national parent company
        ParentCompany nationalOffice =
            new ParentCompany("National Office");
        nationalOffice.add(northRegion);
        nationalOffice.add(southRegion);
        nationalOffice.add(transportCoE);

        // Create headquarters (top level)
        ParentCompany headquarters =
            new ParentCompany("Headquarters");
        headquarters.add(nationalOffice);

        // Display complete hierarchy
        System.out.println("\n--- Complete Company Hierarchy ---\n");
        headquarters.displayInfo(0);

        // Display costs at different levels
        System.out.println("\n--- Cost Summary ---\n");
        System.out.println("North Region Total: $"
            + northRegion.getMaintenanceCost());
        System.out.println("South Region Total: $"
            + southRegion.getMaintenanceCost());
        System.out.println("National Office Total: $"
            + nationalOffice.getMaintenanceCost());
        System.out.println("Headquarters Grand Total: $"
            + headquarters.getMaintenanceCost());
    }
}
```

Listing 11: CompanyDemo.java - Client

Explanation: The client demonstrates the Composite pattern's capabilities:

- Creates multiple independent companies with different vehicle counts and costs
- Builds a multi-level hierarchy: Headquarters → National Office → Regions → Companies
- Treats all companies uniformly through the CompanyComponent interface
- Calculates costs at any level of the hierarchy without special handling
- Shows that both leaf and composite nodes respond to the same operations

0.3.5 Program Output

```
=========================================
  VEHICLE MAINTENANCE COST CALCULATOR
=========================================


--- Building Company Hierarchy ---

Added Transport Co A to North Region
Added Transport Co B to North Region
Added Transport Co C to South Region
Added Transport Co D to South Region
Added North Region to National Office
Added South Region to National Office
Added Transport Co E to National Office
Added National Office to Headquarters

--- Complete Company Hierarchy ---

+ Headquarters (Parent Company)
  Total Maintenance: $32400.0
  Children:
  + National Office (Parent Company)
    Total Maintenance: $32400.0
    Children:
    + North Region (Parent Company)
      Total Maintenance: $14000.0
      Children:
      - Transport Co A
        Vehicles: 10
        Unit Cost: $500.0
        Total Maintenance: $5000.0
      - Transport Co B
        Vehicles: 15
        Unit Cost: $600.0
        Total Maintenance: $9000.0
    + South Region (Parent Company)
      Total Maintenance: $9400.0
      Children:
      - Transport Co C
        Vehicles: 8
        Unit Cost: $550.0
        Total Maintenance: $4400.0
      - Transport Co D
        Vehicles: 12
        Unit Cost: $500.0
        Total Maintenance: $6000.0
    - Transport Co E
      Vehicles: 20
      Unit Cost: $450.0
      Total Maintenance: $9000.0

--- Cost Summary ---

North Region Total: $14000.0
South Region Total: $9400.0
National Office Total: $32400.0
Headquarters Grand Total: $32400.0
```
Listing 12: Output of CompanyDemo.java

0.4 Exercise 3: Adapter Pattern - Payment Processor Integration

0.4.1 Objective

You are developing a modern e-commerce platform. The system expects a standard PaymentProcessor interface for payment processing. However, the platform must integrate with two existing third-party payment services (QuickPay and SafeTransfer), each with a different API. You want the user to be able to use the new PaymentProcessor interface without modifying the existing payment services.

0.4.2 Question 1: Which design pattern should you use?

The **Adapter Pattern** (also known as Wrapper pattern) is the appropriate solution for this scenario because:

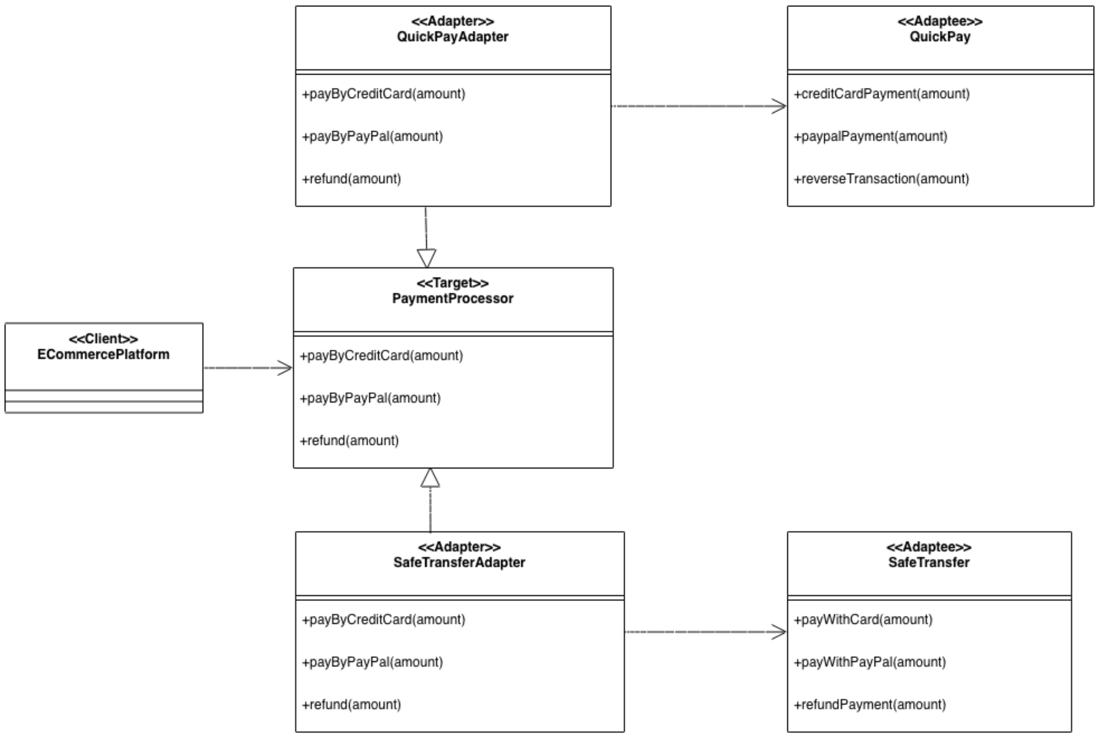
- **Incompatible Interfaces:** The third-party services (QuickPay and SafeTransfer) have different method signatures than the expected PaymentProcessor interface
- **Cannot Modify Existing Code:** The third-party services are external libraries that cannot be modified
- **Interface Translation:** We need to translate calls from the PaymentProcessor interface to the specific methods of each payment service
- **Multiple Adaptees:** We have two different services that need to work with the same target interface
- **Reusability:** The adapter allows us to reuse existing, tested payment service code without modification

The Adapter pattern acts as a bridge between incompatible interfaces, allowing classes to work together that couldn't otherwise due to incompatible interfaces.

0.4.3 Question 2: Participants and Class Diagram

Participants in the Adapter Pattern:

- **Target (PaymentProcessor):** The interface that the client expects to use
- **Adaptee (QuickPay, SafeTransfer):** The existing classes with incompatible interfaces
- **Adapter (QuickPayAdapter, SafeTransferAdapter):** Classes that implement the Target interface and wrap the Adaptee, translating calls from the Target interface to the Adaptee's methods
- **Client (PaymentDemo):** Uses the Target interface without knowing about the adapters



0.4.4 Question 3: Implementation

Target Interface

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

Listing 13: PaymentProcessor.java - Target Interface

Explanation: PaymentProcessor defines the standard interface that our e-commerce platform expects. All payment methods should conform to this interface, providing consistent method names and signatures for credit card payments, PayPal payments, and refunds.

Adaptee Classes (Third-Party Services)

```java
public class QuickPay {
    public void creditCardPayment(double amount) {
        System.out.println("QuickPay: Processing credit card payment of $"
            + amount);
        System.out.println("QuickPay: Transaction ID: QP"
            + System.currentTimeMillis());
    }

    public void paypalPayment(double amount) {
        System.out.println("QuickPay: Processing PayPal payment of $"
            + amount);
        System.out.println("QuickPay: PayPal Transaction ID: PP"
            + System.currentTimeMillis());
    }

    public void reverseTransaction(double amount) {
        System.out.println("QuickPay: Reversing transaction of $"
            + amount);
        System.out.println("QuickPay: Refund processed successfully");
    }
}
```

Listing 14: QuickPay.java - First Adaptee

Explanation: QuickPay is a third-party payment service with its own method naming conventions. Note that:

- It uses camelCase with different method names (creditCardPayment vs payByCreditCard)
- The refund operation is called reverseTransaction
- This is existing code that cannot be modified

```java
public class SafeTransfer {
    public void payWithCard(double amount) {
        System.out.println("SafeTransfer: Card payment of $"
            + amount + " completed");
        System.out.println("SafeTransfer: Confirmation code: ST"
            + System.currentTimeMillis());
    }

    public void payWithPayPal(double amount) {
        System.out.println("SafeTransfer: PayPal payment of $"
            + amount + " completed");
        System.out.println("SafeTransfer: PayPal reference: PP"
            + System.currentTimeMillis());
    }

    public void refundPayment(double amount) {
        System.out.println("SafeTransfer: Refund of $"
            + amount + " processed");
        System.out.println("SafeTransfer: Refund will be credited in 3-5
             days");
    }
}
```

Listing 15: SafeTransfer.java - Second Adaptee

Explanation: SafeTransfer is another third-party service with completely different method names and signatures from both QuickPay and our target interface. It uses "payWith" prefix for payments and "refundPayment" for refunds.

Adapter Classes

```java
public class QuickPayAdapter implements PaymentProcessor {
    private QuickPay quickPay;

    public QuickPayAdapter(QuickPay quickPay) {
        this.quickPay = quickPay;
        System.out.println("[Adapter] QuickPay adapter initialized");
    }

    @Override
    public void payByCreditCard(double amount) {
        System.out.println("[Adapter] Translating payByCreditCard " +
            "to QuickPay.creditCardPayment");
        quickPay.creditCardPayment(amount);
    }

    @Override
    public void payByPayPal(double amount) {
        System.out.println("[Adapter] Translating payByPayPal " +
            "to QuickPay.paypalPayment");
        quickPay.paypalPayment(amount);
    }

    @Override
    public void refund(double amount) {
        System.out.println("[Adapter] Translating refund " +
            "to QuickPay.reverseTransaction");
        quickPay.reverseTransaction(amount);
    }
}
```

Listing 16: QuickPayAdapter.java - First Adapter

Explanation: QuickPayAdapter wraps QuickPay and implements the PaymentProcessor interface. It:

- Holds a reference to a QuickPay instance
- Translates PaymentProcessor method calls to QuickPay's specific method names
- Provides logging to show the translation happening

- Allows QuickPay to work with code expecting PaymentProcessor

```java
public class SafeTransferAdapter implements PaymentProcessor {
    private SafeTransfer safeTransfer;

    public SafeTransferAdapter(SafeTransfer safeTransfer) {
        this.safeTransfer = safeTransfer;
        System.out.println("[Adapter] SafeTransfer adapter initialized")
            ;
    }

    @Override
    public void payByCreditCard(double amount) {
        System.out.println("[Adapter] Translating payByCreditCard " +
            "to SafeTransfer.payWithCard");
        safeTransfer.payWithCard(amount);
    }

    @Override
    public void payByPayPal(double amount) {
        System.out.println("[Adapter] Translating payByPayPal " +
            "to SafeTransfer.payWithPayPal");
        safeTransfer.payWithPayPal(amount);
    }

    @Override
    public void refund(double amount) {
        System.out.println("[Adapter] Translating refund " +
            "to SafeTransfer.refundPayment");
        safeTransfer.refundPayment(amount);
    }
}
```

Listing 17: SafeTransferAdapter.java - Second Adapter

Explanation: SafeTransferAdapter wraps SafeTransfer and provides the same interface translation capability. It demonstrates that multiple adapters can implement the same target interface while wrapping different adaptees with completely different APIs.

Client Implementation

```java
public class PaymentDemo {

    // Client method that works with PaymentProcessor interface
    public static void processPayments(PaymentProcessor processor,
                                       String serviceName) {
        System.out.println("\n--- Processing Payments with "
            + serviceName + " ---");

        System.out.println("\n1. Credit Card Payment:");
        processor.payByCreditCard(100.0);

        System.out.println("\n2. PayPal Payment:");
        processor.payByPayPal(50.0);

        System.out.println("\n3. Refund:");
        processor.refund(25.0);
    }

    public static void main(String[] args) {
        System.out.println("==========================================");
        System.out.println("   E-COMMERCE PAYMENT PROCESSING SYSTEM   ");
        System.out.println("==========================================\n
            ");

        // Create instances of third-party payment services
        QuickPay quickPay = new QuickPay();
        SafeTransfer safeTransfer = new SafeTransfer();

        // Wrap them with adapters
        System.out.println("--- Initializing Payment Adapters ---\n");
        PaymentProcessor quickPayProcessor =
            new QuickPayAdapter(quickPay);
        PaymentProcessor safeTransferProcessor =
            new SafeTransferAdapter(safeTransfer);

        // Use both services through the unified interface
        processPayments(quickPayProcessor, "QuickPay");

        System.out.println("\n" + "=".repeat(50) + "\n");

        processPayments(safeTransferProcessor, "SafeTransfer");

        // Demonstrate that client code doesn't need to know
        // about specific implementations
        System.out.println("\n" + "=".repeat(50));
        System.out.println("\nDemonstration: Client works with any " +
            "PaymentProcessor implementation");
        System.out.println("without knowing the underlying service.\n");
    }
}
```

Listing 18: PaymentDemo.java - Client

Explanation: The client demonstrates the Adapter pattern's key benefits:

- The processPayments() method works with any PaymentProcessor implementation
- Client code doesn't need to know about QuickPay or SafeTransfer's specific APIs
- Both incompatible services work through the unified interface
- Adding a new payment service only requires creating a new adapter
- The client code remains unchanged when switching between services

0.4.5 Program Output

```
==========================================
   E-COMMERCE PAYMENT PROCESSING SYSTEM
==========================================

--- Initializing Payment Adapters ---

```

```
[Adapter] QuickPay adapter initialized
[Adapter] SafeTransfer adapter initialized

--- Processing Payments with QuickPay ---

1. Credit Card Payment:
[Adapter] Translating payByCreditCard to QuickPay.creditCardPayment
QuickPay: Processing credit card payment of $100.0
QuickPay: Transaction ID: QP1732287456789

2. PayPal Payment:
[Adapter] Translating payByPayPal to QuickPay.paypalPayment
QuickPay: Processing PayPal payment of $50.0
QuickPay: PayPal Transaction ID: PP1732287456790

3. Refund:
[Adapter] Translating refund to QuickPay.reverseTransaction
QuickPay: Reversing transaction of $25.0
QuickPay: Refund processed successfully

==================================================

--- Processing Payments with SafeTransfer ---

1. Credit Card Payment:
[Adapter] Translating payByCreditCard to SafeTransfer.payWithCard
SafeTransfer: Card payment of $100.0 completed
SafeTransfer: Confirmation code: ST1732287456791

2. PayPal Payment:
[Adapter] Translating payByPayPal to SafeTransfer.payWithPayPal
SafeTransfer: PayPal payment of $50.0 completed
SafeTransfer: PayPal reference: PP1732287456792

3. Refund:
[Adapter] Translating refund to SafeTransfer.refundPayment
SafeTransfer: Refund of $25.0 processed
SafeTransfer: Refund will be credited in 3-5 days

==================================================

Demonstration: Client works with any PaymentProcessor implementation
without knowing the underlying service.
```

Listing 19: Output of PaymentDemo.java

0.5 Exercise 4: Observer Pattern - GUI Event System

0.5.1 Objective

We are building a GUI dashboard that has multiple interactive elements: Buttons (e.g., Submit, Cancel) and Sliders (e.g., Volume control, Brightness control). Multiple components need to react to changes in these GUI elements: Logger (logs user interactions), LabelUpdater (updates a GUI label showing the last action), and NotificationSender (pops up alerts for important interactions). You need to ensure that each component reacts efficiently and promptly to changes in the buttons and sliders.

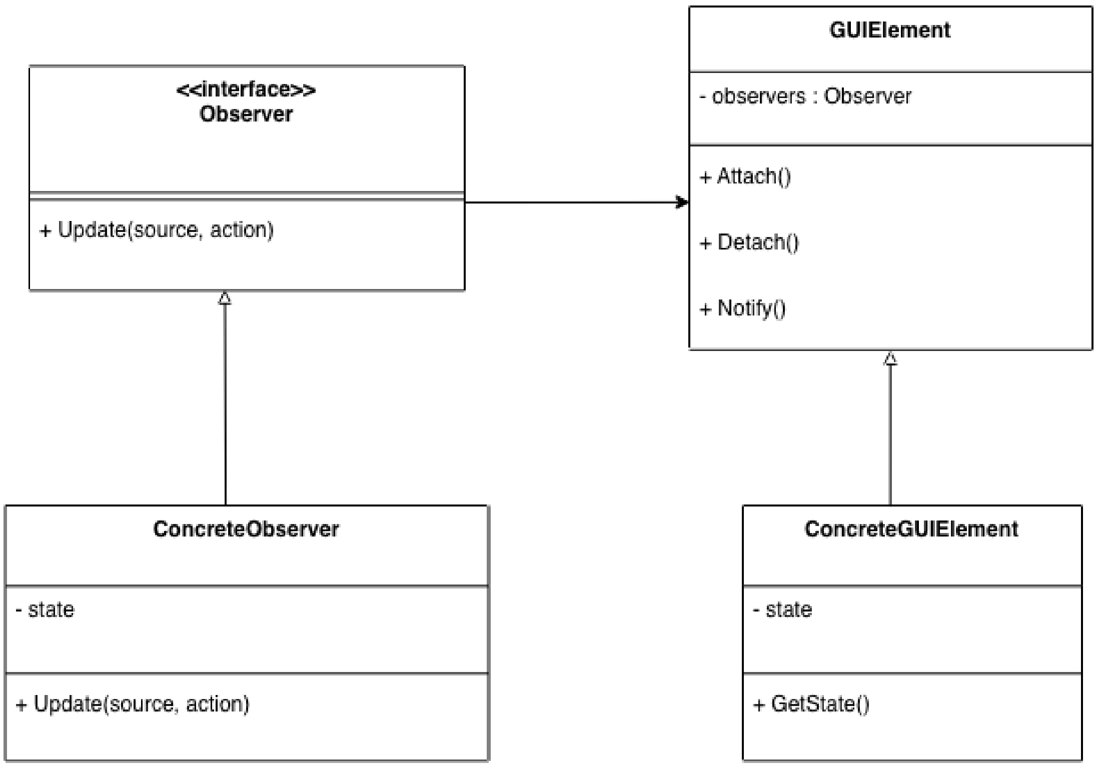
0.5.2 Question 1: Which design pattern is most suitable and why?

The **Observer Pattern** (also known as Publish-Subscribe pattern) is most suitable for this scenario for the following reasons:

- **One-to-Many Dependency:** One GUI element (subject) needs to notify multiple components (observers) about state changes
- **Loose Coupling:** GUI elements don't need to know the specific types of components observing them, only that they implement the Observer interface
- **Dynamic Relationships:** Observers can be added or removed at runtime without modifying the GUI elements
- **Event-Driven Architecture:** The system is inherently event-driven - user interactions trigger notifications to interested parties
- **Broadcast Communication:** A single user action (button click, slider movement) needs to notify all registered observers simultaneously
- **Separation of Concerns:** GUI elements focus on user interaction, while observers handle their specific reactions independently
- **Extensibility:** New observer types can be added without modifying existing GUI elements or other observers

The Observer pattern establishes a subscription mechanism that allows multiple objects to listen and react to events happening in the observed object, which perfectly matches the requirements of a GUI event system.

0.5.3 Question 2: Class Diagram



0.5.4 Question 3: Implementation

Observer Interface

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

Listing 20: Observer.java - Observer Interface

Explanation: The Observer interface defines the contract for objects that want to receive notifications from subjects. The update() method is called by the subject when an event occurs, passing information about what happened (source and action). This abstraction allows any class to become an observer by implementing this interface.

Subject Interface

```
public interface Subject {
    void attach(Observer observer);
    void detach(Observer observer);
    void notifyObservers(String action);
}
```

Listing 21: Subject.java - Subject Interface

Explanation: The Subject interface defines the contract for objects that can be observed. It provides methods to:

- attach(): Register a new observer to receive notifications

- detach(): Remove an observer from the notification list
- notifyObservers(): Notify all registered observers about an event

This interface ensures that all subjects follow the same protocol for observer management.

Button Class (Concrete Subject)

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

public class Button implements Subject {
    private String name;
    private List<Observer> observers = new ArrayList<>();

    public Button(String name) {
        this.name = name;
        System.out.println("[Button] " + name + " created");
    }

    @Override
    public void attach(Observer observer) {
        observers.add(observer);
        System.out.println("[Button] Observer attached to " + name);
    }

    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
        System.out.println("[Button] Observer detached from " + name);
    }

    @Override
    public void notifyObservers(String action) {
        System.out.println("[Button] " + name
            + " notifying " + observers.size() + " observer(s)");
        for (Observer observer : observers) {
            observer.update(name, action);
        }
    }

    // Simulate button click
    public void click() {
        System.out.println("\n" + "=".repeat(50));
        System.out.println("[USER ACTION] " + name + " clicked");
        System.out.println("=".repeat(50));
        notifyObservers("clicked");
    }

    public String getName() {
        return name;
    }
}
```

Listing 22: Button.java - Concrete Subject

Explanation: Button represents a clickable UI element that acts as a concrete subject. Key features:

- Maintains a list of observers interested in button events
- Implements attach/detach for observer management
- The click() method simulates user interaction and triggers notifications
- notifyObservers() iterates through all registered observers and calls their update() method
- The button doesn't know the concrete types of observers, only that they implement the Observer interface

Slider Class (Concrete Subject)

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

public class Slider implements Subject {
    private String name;
    private int value;
    private int minValue;
    private int maxValue;
    private List<Observer> observers = new ArrayList<>();

    public Slider(String name, int initialValue,
                  int minValue, int maxValue) {
        this.name = name;
        this.value = initialValue;
        this.minValue = minValue;
        this.maxValue = maxValue;
        System.out.println("[Slider] " + name + " created " +
            "(Range: " + minValue + "-" + maxValue +
            ", Initial: " + value + ")");
    }

    @Override
    public void attach(Observer observer) {
        observers.add(observer);
        System.out.println("[Slider] Observer attached to " + name);
    }

    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
        System.out.println("[Slider] Observer detached from " + name);
    }

    @Override
    public void notifyObservers(String action) {
        System.out.println("[Slider] " + name
            + " notifying " + observers.size() + " observer(s)");
        for (Observer observer : observers) {
            observer.update(name, action);
        }
    }

    // Simulate slider value change
    public void setValue(int newValue) {
        if (newValue < minValue || newValue > maxValue) {
            System.out.println("[Slider] Invalid value: " + newValue);
            return;
        }

        int oldValue = this.value;
        this.value = newValue;

        System.out.println("\n" + "=".repeat(50));
        System.out.println("[USER ACTION] " + name
            + " moved from " + oldValue + " to " + value);
        System.out.println("=".repeat(50));

        notifyObservers("moved to " + value);
    }

    public int getValue() {
        return value;
    }

    public String getName() {
        return name;
    }
}
```

Listing 23: Slider.java - Concrete Subject

Explanation: Slider represents a UI slider control that acts as another concrete subject. It:

- Maintains internal state (current value, min/max range)
- Validates value changes before accepting them
- Notifies observers when the value changes
- Passes the new value as part of the action string in notifications
- Demonstrates how subjects can include state information in notifications

EventLogger Class (Concrete Observer)

```java
public class EventLogger implements Observer {

    public EventLogger() {
        System.out.println("[Observer] EventLogger created");
    }

    @Override
    public void update(String source, String action) {
        String timestamp = java.time.LocalTime.now().toString();
        System.out.println("  [LOG] " + timestamp
            + " - " + source + " was " + action);
    }
}
```

Listing 24: EventLogger.java - Concrete Observer

Explanation: EventLogger is a concrete observer that logs all UI events with timestamps. It:

- Implements the Observer interface
- Records every event it's notified about
- Adds timestamps for event tracking
- Demonstrates a passive observer that doesn't modify system state

LabelUpdater Class (Concrete Observer)

```java
public class LabelUpdater implements Observer {
    private String labelText = "No events yet";

    public LabelUpdater() {
        System.out.println("[Observer] LabelUpdater created");
    }

    @Override
    public void update(String source, String action) {
        labelText = "Last event: " + source + " - " + action;
        System.out.println("  [LABEL] Display updated: \""
            + labelText + "\"");
    }

    public String getLabelText() {
        return labelText;
    }
}
```

Listing 25: LabelUpdater.java - Concrete Observer

Explanation: LabelUpdater is a concrete observer that maintains and updates a display label. It:

- Maintains internal state (the label text)
- Updates its state based on notifications
- Simulates updating a GUI label component
- Demonstrates an observer that maintains state

NotificationSender Class (Concrete Observer)

```java
public class NotificationSender implements Observer {
    private boolean enabled = true;

    public NotificationSender() {
        System.out.println("[Observer] NotificationSender created");
    }

    @Override
    public void update(String source, String action) {
        if (enabled) {
            System.out.println("  [NOTIFICATION] ALERT: "
                + source + " was " + action);
            System.out.println("  [NOTIFICATION] Push notification sent
                " +
                "to user's device");
        }
    }

    public void setEnabled(boolean enabled) {
        this.enabled = enabled;
        System.out.println("[NotificationSender] Notifications "
            + (enabled ? "enabled" : "disabled"));
    }

    public boolean isEnabled() {
        return enabled;
    }
}
```

Listing 26: NotificationSender.java - Concrete Observer

Explanation: NotificationSender is a concrete observer that sends push notifications. It:

- Can be enabled or disabled without detaching from subjects
- Simulates sending notifications for important events
- Demonstrates an observer with controllable behavior
- Shows how observers can filter or process notifications conditionally

Client Implementation

```java
public class GUIDemo {
    public static void main(String[] args) {
        System.out.println("==========================================");
        System.out.println("        GUI EVENT SYSTEM DEMO            ");
        System.out.println("==========================================\n
            ");

        // Create GUI components (subjects)
        System.out.println("--- Creating GUI Components ---\n");
        Button submitButton = new Button("Submit Button");
        Button cancelButton = new Button("Cancel Button");
        Slider volumeSlider = new Slider("Volume Slider", 50, 0, 100);
        Slider brightnessSlider =
            new Slider("Brightness Slider", 75, 0, 100);

        // Create observers
        System.out.println("\n--- Creating Observers ---\n");
        EventLogger logger = new EventLogger();
        LabelUpdater labelUpdater = new LabelUpdater();
        NotificationSender notifier = new NotificationSender();

        // Attach observers to subjects
        System.out.println("\n--- Attaching Observers ---\n");

        // Submit button: all three observers
        System.out.println("Configuring Submit Button:");
        submitButton.attach(logger);
        submitButton.attach(labelUpdater);
        submitButton.attach(notifier);

        // Cancel button: logger and label only
        System.out.println("\nConfiguring Cancel Button:");
        cancelButton.attach(logger);
        cancelButton.attach(labelUpdater);

        // Volume slider: logger and notifier
        System.out.println("\nConfiguring Volume Slider:");
        volumeSlider.attach(logger);
        volumeSlider.attach(notifier);

        // Brightness slider: logger and label
        System.out.println("\nConfiguring Brightness Slider:");
        brightnessSlider.attach(logger);
        brightnessSlider.attach(labelUpdater);

        // Simulate user interactions
        System.out.println("\n\n" + "=".repeat(50));
        System.out.println("     SIMULATING USER INTERACTIONS");
        System.out.println("=".repeat(50) + "\n");

        // Interaction 1: Submit button click
        submitButton.click();

        // Interaction 2: Volume slider adjustment
        try { Thread.sleep(100); } catch (InterruptedException e) {}
        volumeSlider.setValue(75);

        // Interaction 3: Cancel button click
        try { Thread.sleep(100); } catch (InterruptedException e) {}
        cancelButton.click();

        // Interaction 4: Brightness adjustment
        try { Thread.sleep(100); } catch (InterruptedException e) {}
        brightnessSlider.setValue(90);

        // Demonstrate detaching observer
        System.out.println("\n\n" + "=".repeat(50));
        System.out.println("   DEMONSTRATING DYNAMIC OBSERVER MANAGEMENT
            ");
        System.out.println("=".repeat(50) + "\n");

        System.out.println("Detaching NotificationSender from " +
            "Submit Button...\n");
        submitButton.detach(notifier);

        System.out.println("\nClicking Submit Button again " +
            "(no notification should be sent):");
        submitButton.click();

        // Demonstrate disabling observer
        System.out.println("\n\nDisabling NotificationSender...\n");
        notifier.setEnabled(false);

        System.out.println("\nMoving Volume Slider " +
            "(notification disabled):");
        volumeSlider.setValue(100);

        // Final summary
        System.out.println("\n\n" + "=".repeat(50));
        System.out.println("Current label text: \""
            + labelUpdater.getLabelText() + "\"");
        System.out.println("=".repeat(50));
    }
}
```

Listing 27: GUIDemo.java - Client

Explanation: The client demonstrates the Observer pattern's key capabilities:

- Creates multiple subjects (buttons and sliders) and observers
- Establishes different observer relationships for different subjects
- Simulates user interactions that trigger notifications
- Demonstrates detaching observers at runtime
- Shows how observers can be disabled without detaching
- Illustrates the loose coupling between subjects and observers

0.5.5 Program Output

```
=========================================
        GUI EVENT SYSTEM DEMO
=========================================

--- Creating GUI Components ---

[Button] Submit Button created
[Button] Cancel Button created
[Slider] Volume Slider created (Range: 0-100, Initial: 50)
[Slider] Brightness Slider created (Range: 0-100, Initial: 75)

--- Creating Observers ---

[Observer] EventLogger created
[Observer] LabelUpdater created
```

```
[Observer] NotificationSender created

--- Attaching Observers ---

Configuring Submit Button:
[Button] Observer attached to Submit Button
[Button] Observer attached to Submit Button
[Button] Observer attached to Submit Button

Configuring Cancel Button:
[Button] Observer attached to Cancel Button
[Button] Observer attached to Cancel Button

Configuring Volume Slider:
[Slider] Observer attached to Volume Slider
[Slider] Observer attached to Volume Slider

Configuring Brightness Slider:
[Slider] Observer attached to Brightness Slider
[Slider] Observer attached to Brightness Slider


==================================================
     SIMULATING USER INTERACTIONS
==================================================


==================================================
[USER ACTION] Submit Button clicked
==================================================
[Button] Submit Button notifying 3 observer(s)
  [LOG] 14:23:45.123 - Submit Button was clicked
  [LABEL] Display updated: "Last event: Submit Button - clicked"
  [NOTIFICATION] ALERT: Submit Button was clicked
  [NOTIFICATION] Push notification sent to user's device

==================================================
[USER ACTION] Volume Slider moved from 50 to 75
==================================================
[Slider] Volume Slider notifying 2 observer(s)
  [LOG] 14:23:45.234 - Volume Slider was moved to 75
  [NOTIFICATION] ALERT: Volume Slider was moved to 75
  [NOTIFICATION] Push notification sent to user's device

==================================================
[USER ACTION] Cancel Button clicked
==================================================
[Button] Cancel Button notifying 2 observer(s)
  [LOG] 14:23:45.345 - Cancel Button was clicked
  [LABEL] Display updated: "Last event: Cancel Button - clicked"

==================================================
[USER ACTION] Brightness Slider moved from 75 to 90
==================================================
[Slider] Brightness Slider notifying 2 observer(s)
  [LOG] 14:23:45.456 - Brightness Slider was moved to 90
  [LABEL] Display updated: "Last event: Brightness Slider - moved to 90"

```

```

==================================================
    DEMONSTRATING DYNAMIC OBSERVER MANAGEMENT
==================================================

Detaching NotificationSender from Submit Button...

[Button] Observer detached from Submit Button

Clicking Submit Button again (no notification should be sent):

==================================================
[USER ACTION] Submit Button clicked
==================================================
[Button] Submit Button notifying 2 observer(s)
  [LOG] 14:23:45.567 - Submit Button was clicked
  [LABEL] Display updated: "Last event: Submit Button - clicked"


Disabling NotificationSender...

[NotificationSender] Notifications disabled

Moving Volume Slider (notification disabled):

==================================================
[USER ACTION] Volume Slider moved from 75 to 100
==================================================
[Slider] Volume Slider notifying 2 observer(s)
  [LOG] 14:23:45.678 - Volume Slider was moved to 100


==================================================
Current label text: "Last event: Submit Button - clicked"
==================================================
```
Listing 28: Output of GUIDemo.java