Ministry of Education and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work No. 3

Discipline: Techniques and Mechanisms of Software Design

Elaborated: FAF-222,
Grosu Renat

Checked: asist. univ.,
Furdui Alexandru

Chișinău 2024

Topic: Behavioral DPs

Task: Implement atleast 3 structural design patterns in your project.

Objectives:

1. Study and understand the Behavioral Design Patterns.

2. As a continuation of the previous laboratory work, think about what communication between software entities might be involed in your system.

3. Implement some additional functionalities using behavioral design patterns.

THEORY

Behavioral design patterns help with the interaction and communication between objects or classes in a system. They are responsible with defining the ways objects collaborate, handle several responsibilities, and execute tasks efficiently.

1. **Chain of Responsibility Pattern**
   The Chain of Responsibility pattern defines a chain of handlers that pass a request along to each other. The request is processed by each handler which then has the option to either handle it or give it down the chain.
   - **Key Idea:** Allows multiple handlers to process a request in a sequence.
   - **Use Cases:** Event handling systems, logging frameworks.
2. **Command Pattern**
   The Command pattern encapsulates a request as an object, allowing parameterization of requests, queuing, and undoable operations.
   - **Key Idea:** Turns requests into objects which allows delaying and queueing request executions and supporting undoable operations.
   - **Use Cases:** GUIs with buttons for undo/redo or menu commands.
3. **Interpreter Pattern**
   The Interpreter pattern defines a language's grammar and an interpreter which processes statements in that language.
   - **Key Idea:** Maps a domain-specific language into a hierarchical class structure for interpretation.
   - **Use Cases:** Expression parsing, configuration file processing.
4. **Iterator Pattern**
   The Iterator pattern is used to iterate through elements of a collection without exposing its internal representation.

- **Key Idea:** It separates the iteration logic from the collections themselves.
- **Use Cases:** List/Tree/Hashmap Collections and others.

5. **Mediator Pattern**
   This pattern encapsulates how objects interact by having them communicate with a mediator instead of each other, which reduces dependencies. Using it, objects don't refer to each other directly, but instead talk through the mediator.

   - **Key Idea:** Reduce chaotic dependencies between objects.
   - **Use Cases:** Chat applications, traffic control systems.

6. **Observer Pattern**
   The Observer pattern allows you to make a subscription mechanism that notifies multiples objects when an event happened to the object they are observing.

   - **Key Idea:** Provides a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
   - **Use Cases:** Event listeners, notifications.

7. **Strategy Pattern**
   The Strategy pattern lets you define a family of algorithms, puts them into separate classes, and makes them interchangeable.

   - **Key Idea:** Separates behavior from its context which in turn promotes the Open/Closed Principle.
   - **Use Cases:** Sorting algorithms, payment strategies.

8. **Visitor Pattern**
   The Visitor pattern lets you execute an operation over a set of objects with different classes by having a visitor object implement several variants of the same operation, which correspond to all target classes, without altering them.

   - **Key Idea:** Separates the algorithms or behaviors from the objects on which they operate.
   - **Use Cases:** Compilers, object tree traversal.

INTRODUCTION

In this project, I continued my order processing system and I implemented the Strategy and Observer behavioral design patterns inside it. The Observer pattern enables communication between the order processor and its observers, making sure that any changes in the order status are automatically sent to all of its subscribed observers. Meanwhile, the Strategy pattern is used for handling various types of payment methods, which allows my system to easily switch between payment strategies like credit cards, PayPal, or cash.

OBSERVER IMPLEMENTATION

The Observer pattern was implemented to send notifications and automatic updates from an OrderProcessor (subject) to its observers. The pattern makes sure that any changes in the OrderProcessor (such as a payment or an order being processed) are communicated immediately to all its observers.

```
public interface Observer {  8
    void update(String event);
}
```

```
public class EmailNotificationObserver implements Observer {
    @Override  1 usage  Renat03
    public void update(String event) {
        System.out.println("[Email Notification] " + event);
    }
}
```

```
public class LoggingObserver implements Observer {  2 usages  Renat03
    @Override  1 usage  Renat03
    public void update(String event) { System.out.println("[Log] " + event); }
}
```

Observers such as EmailNotificationObserver and LoggingObserver implement the Observer interface, which defines the update method for handling notifications.

```
public class OrderProcessor {  6 usages  Renat03 *
    private static domain.singleton.OrderProcessor instance;  3 us
    private final List<Observer> observers = new ArrayList<>();
```

```
public void processOrder(Order order) {  3 usages  Renat03
    System.out.println("\nProcessing order...");
    order.displayOrderDetails();
    notifyObservers( event: "Order processed for customer: " + order.getCustomerName());
}

public void processPayment(PaymentStrategy paymentStrategy, double amount) {  3 usages  Renat03
    if (paymentStrategy == null) {
        System.out.println("No payment strategy selected. Payment cannot be processed.");
    } else {
        paymentStrategy.pay(amount);
        notifyObservers( event: "Payment of $" + amount + " processed.");
    }
}
```

My OrderProcessor holds a list of all of its observers and when the state of it changes (for example a payment is being processed), it calls the notifyObservers method, which triggers the update method on all of its observers.

```
OrderProcessor processor = OrderProcessor.getInstance();
processor.addObserver(new EmailNotificationObserver());
processor.addObserver(new LoggingObserver());
processor.addObserver(new DatabaseObserver());
```

Below is the result, which shows how the Observer pattern works inside my system.

```
Processing order...
Electronics Order: 2 units of Smartphone for John Smith at Street Name 123
[Email Notification] Order processed for customer: John Smith
[Log] Order processed for customer: John Smith
[Database] Updating record: Order processed for customer: John Smith
Processing PayPal payment...
PayPal Account: alice@example.com
Amount Paid: $299.99
[Email Notification] Payment of $299.99 processed.
[Log] Payment of $299.99 processed.
[Database] Updating record: Payment of $299.99 processed.
```

STRATEGY IMPLEMENTATION

The Strategy pattern was implemented with the purpose of dynamically handling different payment methods in my order processing system. It allows my OrderProcessor to select a specific payment strategy at runtime without it being coupled with one in particular. It also allows me to easily add new payment methods without modifying existing code.

```
public interface PaymentStrategy {
    void pay(double amount);  1 usage
}
```

The PaymentStrategy interface defines the pay method for all payment strategies.

```
public class CashPayment implements PaymentStrategy {  2 us
    @Override  1 usage  Renat03
    public void pay(double amount) {
        System.out.println("Processing cash payment...");
        System.out.println("Amount Paid: $" + amount);
    }
}
```

```java
public class PayPalPayment implements PaymentStrategy {  2 us
    private String email;  2 usages

    public PayPalPayment(String email) {  1 usage  Renat03
        this.email = email;
    }

    @Override  1 usage  Renat03
    public void pay(double amount) {
        System.out.println("Processing PayPal payment...");
        System.out.println("PayPal Account: " + email);
        System.out.println("Amount Paid: $" + amount);
    }
}
```

CashPayment and PayPalPayment are the concrete implementations of the PaymentStrategy, which provide payment processing logic specific to them.

```java
public void processPayment(PaymentStrategy paymentStrategy, double amount) {  3 usages  Re
    if (paymentStrategy == null) {
        System.out.println("No payment strategy selected. Payment cannot be processed.");
    } else {
        paymentStrategy.pay(amount);
        notifyObservers( event: "Payment of $" + amount + " processed.");
    }
}
```

The OrderProcessor dynamically selects a payment strategy and processes the payment according to the selection of the strategy.

```java
PaymentStrategy paypalPayment = new PayPalPayment( email: "alice@example.com");
processor.processPayment(paypalPayment,  amount: 299.99);
```

```java
PaymentStrategy cashPayment = new CashPayment();
processor.processPayment(cashPayment,  amount: 950.0);
```

Below are the results of my strategy implementation, which shows how you can easily and dynamically switch between different payment strategies.

```
Processing order...
Electronics Order: 2 units of Smartphone for John Smith at Street Name 123
Processing PayPal payment...
PayPal Account: alice@example.com
Amount Paid: $299.99

Processing order...
Clothing Order: 19 units of Barca Jersey for Lamine Yamal at Cabra 304 St
Processing cash payment...
Amount Paid: $950.0
```

CONCLUSION

In this laboratory work, I implemented two behavioral design patterns: Observer and Strategy. Through their implementation I have gained a deeper understanding of how these patterns work, how they enhance communication and flexibility within a system, and how to apply them effectively in practice. This laboratory work helped me see how these patterns can make the code more modular, easier to maintain, and adaptable for future extensions.