"Heaven's Light is Our Guide"

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Lab Report On Design Patterns

Course No. : CSE 3206

Course Title : Software Engineering Sessional

Index :

Submitted by -	*Submitted To -*
Group 04 *Consists of* *2003130,2003131,2003132*	**Farjana Parvin** **Lecturer** **Department of CSE, RUET**

1. Facade Design Pattern

Introduction:

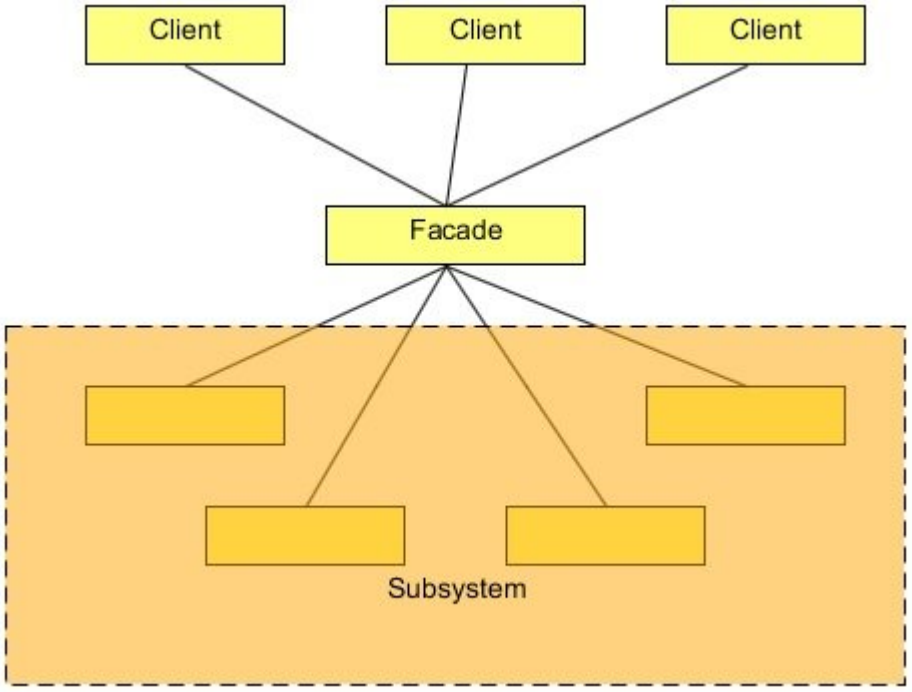
The Facade Design Pattern is a structural design pattern that provides a unified and simplified interface to a set of interfaces within a subsystem. It hides the complexities of the subsystem and exposes only the essential functionality required by the client.

Objective:

The objective of this lab is to understand and implement the Facade Design Pattern in Java by designing an e-commerce application. This pattern simplifies interactions between clients and complex subsystems, thereby enhancing code readability, maintainability, and scalability.

Scenario:

In an e-commerce system, users interact with multiple subsystems to complete an order:
InventoryService - Manages stock checks.
PaymentService - Processes payments.
ShippingService - Arranges shipping for orders.
To streamline these interactions, a Facade (EcommerceFacade) is introduced. It acts as a single point of communication between the client and the subsystems, reducing complexity.



1. Without Using the Facade Design Pattern

In this approach:

The client directly interacts with the subsystems (InventoryService, PaymentService, ShippingService).

The client must manage the coordination between subsystems, leading to tightly coupled code.

Code:

```
// Subsystem 1: Handles inventory management
class InventoryService {
    // Checks if the specified item is in stock
    public boolean checkStock(String item) {
        System.out.println("Checking stock for " + item);
        return true; // Assume the item is always in stock for simplicity
    }
}
// Subsystem 2: Handles payment processing
class PaymentService {
    // Processes payment using the specified payment method and amount
    public void processPayment(String paymentMethod, double amount) {
        System.out.println("Processing payment of $" + amount + " using " + paymentMethod);
    }
}

// Subsystem 3: Handles shipping
class ShippingService {
    // Ships the specified item to the provided address
    public void shipItem(String item, String address) {
        System.out.println("Shipping " + item + " to " + address);
    }
}

// Main class to simulate an e-commerce order process without a facade
public class EcommerceWithoutFacade {
    public static void main(String[] args) {
        // Order details
        String item = "Laptop";
        double amount = 1200.00;
        String paymentMethod = "Credit Card";
        String address = "123 Main St, Springfield";

        // Create subsystem instances
        InventoryService inventoryService = new InventoryService();
```

```
        PaymentService paymentService = new PaymentService();
        ShippingService shippingService = new ShippingService();

        // Client directly interacts with all subsystems
        if (inventoryService.checkStock(item)) { // Check if the item is in stock
            paymentService.processPayment(paymentMethod, amount); // Process the payment
            shippingService.shipItem(item, address); // Arrange for shipping
        } else {
            System.out.println("Item is out of stock!");

    }
    }
}
```

2. With the Facade Design Pattern

Using the Facade Design Pattern:
A single interface (EcommerceFacade) simplifies client interaction.
The facade coordinates with the subsystems internally, isolating the client from their complexity.

Code:

```
// Subsystem 1: Handles inventory management
class InventoryService {
    // Checks if the specified item is in stock
    public boolean checkStock(String item) {
        System.out.println("Checking stock for " + item);
        return true; // Assume the item is always in stock for simplicity
    }
}

// Subsystem 2: Handles payment processing
class PaymentService {
    // Processes payment using the specified payment method and amount
    public void processPayment(String paymentMethod, double amount) {
        System.out.println("Processing payment of $" + amount + " using " + paymentMethod);
    }
}

// Subsystem 3: Handles shipping
class ShippingService {
    // Ships the specified item to the provided address
    public void shipItem(String item, String address) {
        System.out.println("Shipping " + item + " to " + address);
    }
}

// Facade: Provides a simplified interface for the client to interact with the subsystems
```

```
class EcommerceFacade {
    private InventoryService inventoryService; // Handles inventory-related tasks
    private PaymentService paymentService;     // Handles payment-related tasks
    private ShippingService shippingService;   // Handles shipping-related tasks

    // Constructor: Initializes the subsystems
    public EcommerceFacade() {
        inventoryService = new InventoryService();
        paymentService = new PaymentService();
        shippingService = new ShippingService();
    }

    // Simplified method to place an order
    public void placeOrder(String item, double amount, String paymentMethod, String address) {
        System.out.println("Starting the order process...");
        if (inventoryService.checkStock(item)) { // Check stock availability
            paymentService.processPayment(paymentMethod, amount); // Process the payment
            shippingService.shipItem(item, address); // Arrange for shipping
            System.out.println("Order placed successfully!"); // Confirm order
        } else {
            System.out.println("Item is out of stock!");
        }
    }
}

// Main class(client) to simulate an e-commerce order process with a facade
public class EcommerceWithFacade {
    public static void main(String[] args) {
        // Order details
        String item = "Laptop";
        double amount = 1200.00;
        String paymentMethod = "Credit Card";
        String address = "123 Main St, Springfield";

        // Create a facade instance
        EcommerceFacade ecommerceFacade = new EcommerceFacade();

        // The client interacts only with the facade
        ecommerceFacade.placeOrder(item, amount, paymentMethod, address);
    }
}
```

Output:

Without Facade

Checking stock for Laptop

Processing payment of $1200.0 using Credit Card

Shipping Laptop to 123 Main St, Springfield

With Facade

Starting the order process...

Checking stock for Laptop

Processing payment of $1200.0 using Credit Card

Shipping Laptop to 123 Main St, Springfield

Order placed successfully!

Comparison:

.

Aspect	Without Facade	With Facade
Client Interaction	Direct interaction with all subsystems.	Single point of interaction via the facade.
Complexity	High, as the client handles coordination.	Low, as the facade handles coordination.
Coupling	High coupling between client and subsystems.	Low coupling; client depends only on the facade.
Readability	Reduced due to scattered subsystem logic.	Simplified with a unified interface.
Scalability	Changes in subsystems affect client logic	Subsystem changes are isolated from the client.

2. Flyweight Design Pattern

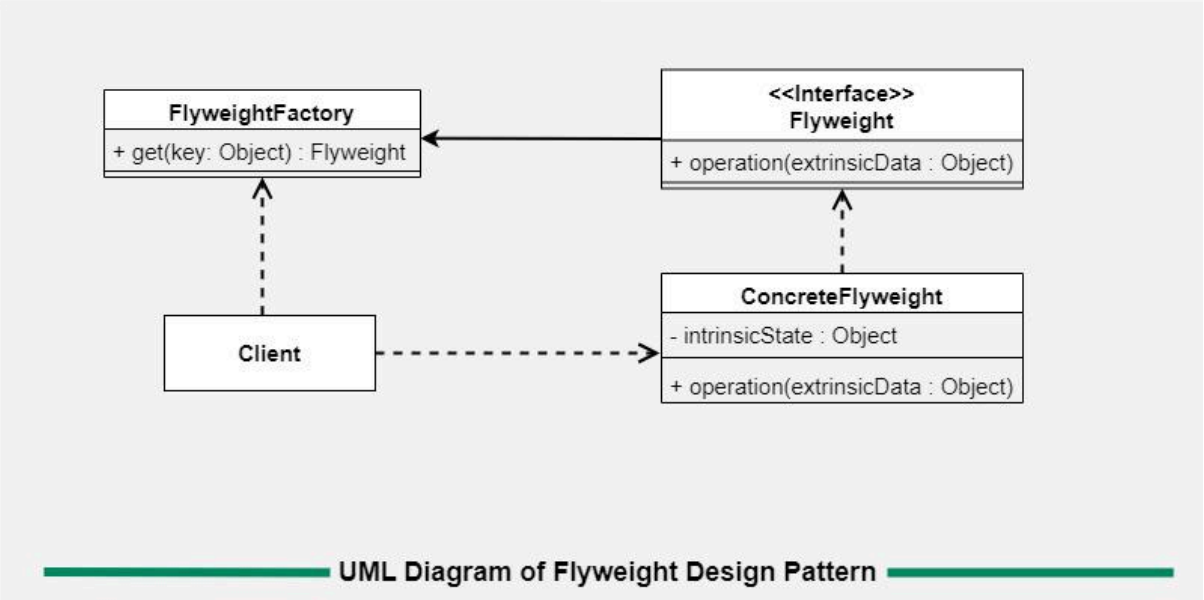
Title: Implementing the Flyweight Design Pattern in a Bookstore Application

Objective :

To demonstrate the use of the Flyweight Design Pattern in reducing memory usage by sharing common data across objects in a bookstore application.

Problem Statement

In a bookstore system, books have attributes such as title, author, genre, shelf number, and position on the shelf. Creating separate objects for each book leads to unnecessary memory consumption. The goal is to optimize memory usage using the Flyweight Pattern.



UML Diagram of Flyweight Design Pattern

Code Without Flyweight Pattern

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

class Book {
    private String title;
    private String author;
    private String genre;
    private int shelfNumber;
    private int position;

    public Book(String title, String author, String genre, int shelfNumber, int position) {
        this.title = title;
        this.author = author;
        this.genre = genre;
        this.shelfNumber = shelfNumber;
        this.position = position;
```

```
    }

    public void display() {
        System.out.println("Book: " + title + ", Author: " + author + ", Genre: " + genre +
                           ", Shelf: " + shelfNumber + ", Position: " + position);
    }
}

public class Bookstore {
    public static void main(String[] args) {
        List<Book> books = new ArrayList<>();
        books.add(new Book("Pather Panchali", "B. Bandopadhyay", "Novel", 1, 1));
        books.add(new Book("Pather Panchali", "B. Bandopadhyay", "Novel", 1, 2));
        books.add(new Book("Chokher Bali", "R. Tagore", "Novel", 2, 1));
        books.add(new Book("Chokher Bali", "R. Tagore", "Novel", 2, 2));

        for (Book book : books) {
            book.display();
        }
        System.out.println("Total book objects created: " + books.size());
    }
```

Output:

```
Book: Pather Panchali, Author: B. Bandopadhyay,Shelf: 1, Pos: 1
Book: Pather Panchali, Author: B. Bandopadhyay,Shelf: 1, Pos: 2
Book: Chokher Bali, Author: R. Tagore,Shelf: 2, Pos: 1
Book: Chokher Bali, Author: R. Tagore,Shelf: 2, Pos: 2
Total book objects created: 4
```

Code With Flyweight Pattern

```
import java.util.HashMap;
import java.util.Map;

class Book {
    private final String title;
    private final String author;
    private final String genre;

    public Book(String title, String author, String genre) {
        this.title = title;
        this.author = author;
        this.genre = genre;
    }

    public void display(int shelfNumber, int position) {
        System.out.println("Book: " + title + ", Author: " + author + ", Genre: " + genre +
                           ", Shelf: " + shelfNumber + ", Position: " + position);
    }
}

class BookFactory {
```

```
    private static final Map<String, Book> bookMap = new HashMap<>();

    public static Book getBook(String title, String author, String genre) {
        String key = title + "-" + author + "-" + genre;
        if (!bookMap.containsKey(key)) {
            bookMap.put(key, new Book(title, author, genre));
        }
        return bookMap.get(key);
    }

    // Public method to get the size of the bookMap
    public static int getBookCount() {
        return bookMap.size();
    }
}

public class Bookstore {
    public static void main(String[] args) {
        Book book1 = BookFactory.getBook("Pather Panchali", "B. Bandopadhyay", "Novel");
        Book book2 = BookFactory.getBook("Pather Panchali", "B. Bandopadhyay", "Novel");
        Book book3 = BookFactory.getBook("Chokher Bali", "R. Tagore", "Novel");
        Book book4 = BookFactory.getBook("Chokher Bali", "R. Tagore", "Novel");

        book1.display(1, 1);
        book2.display(1, 2);
        book3.display(2, 1);
        book4.display(2, 2);

        System.out.println("Total unique book objects created: " + BookFactory.getBookCount());
    }
}
```

```
Book: Pather Panchali, Author: B. Bandopadhyay, Genre: Novel, Shelf: 1, Pos: 1
Book: Pather Panchali, Author: B. Bandopadhyay, Genre: Novel, Shelf: 1, Pos: 2
Book: Chokher Bali, Author: R. Tagore, Genre: Novel, Shelf: 2, Pos: 1
Book: Chokher Bali, Author: R. Tagore, Genre: Novel, Shelf: 2, Pos: 2
Total book objects created: 2
```

Comparison of Metrics

Approach	*Total Book Objects Created*	*Memory Efficiency*
Without Flyweight	*4*	*Low*
With Flyweight	*2*	*High*

Conclusion

The Flyweight Design Pattern reduces memory usage by sharing common book attributes. In the bookstore application, the pattern reduced the number of unique book objects from 4 to 2, demonstrating its efficiency in optimizing memory usage.