Sri Lanka Institute of Information Technology

Lab sheet No 7

Project title – Web-based boat safari trip management system.

Project group - 2025-Y2-S1-MLB-B10G2-05

Module name – Software engineering

Module code – SE2030

Group members –

Athapatthu A.M.P.S.	IT24102801
Dewmini G.D.C.P.	IT24102755

Bandara K.M.G.A.P	IT24102705
Hadhil.M.A.M.	IT24102785
Hettiarachchige D.S.	IT24102779
Hiranyada D M D	IT24102802

Design patterns used

1. **Strategy Pattern**- The Strategy pattern is a behavioural design pattern that allows selecting an algorithm's behaviour at runtime.
   It defines a family of algorithms, encapsulates each one, and makes them interchangeable within a system.
2. **Singleton pattern** – The **Singleton pattern** is a creational design pattern that ensures a class has only **one instance** and provides a **global point of access** to it.

In our system, this pattern is implicitly used in the OrderService class through Spring Boot's dependency injection mechanism. Spring automatically ensures that only one instance of each service or controller exists throughout the application's lifecycle.

3.**Model pattern**

Justification for each pattern

1. Strategy Pattern

In our system:
The Strategy pattern is used to handle different **payment options** dynamically when processing customer orders.
Currently, the system supports **Cash on Delivery**, but it can easily be extended to include other payment types like **Credit Card** or **PayPal** without modifying the core logic.

Why it was chosen:
Previously, if we had used simple conditional statements (if-else) to manage payments, adding a new payment method would require changing existing code violating the Open/Closed Principle. With the Strategy pattern, we can add new payment methods as new strategy classes, keeping the system flexible and maintainable.

How it improves the system:

1. **Flexibility** – New payment methods can be added without modifying existing logic.

2. **Clean structure** – Payment logic is separated from order management, improving readability.
3. **Reusability** – Each payment strategy can be reused across different modules or services.
4. **Maintainability** – Changes to one payment method don't affect others.

---

2. Singleton Pattern

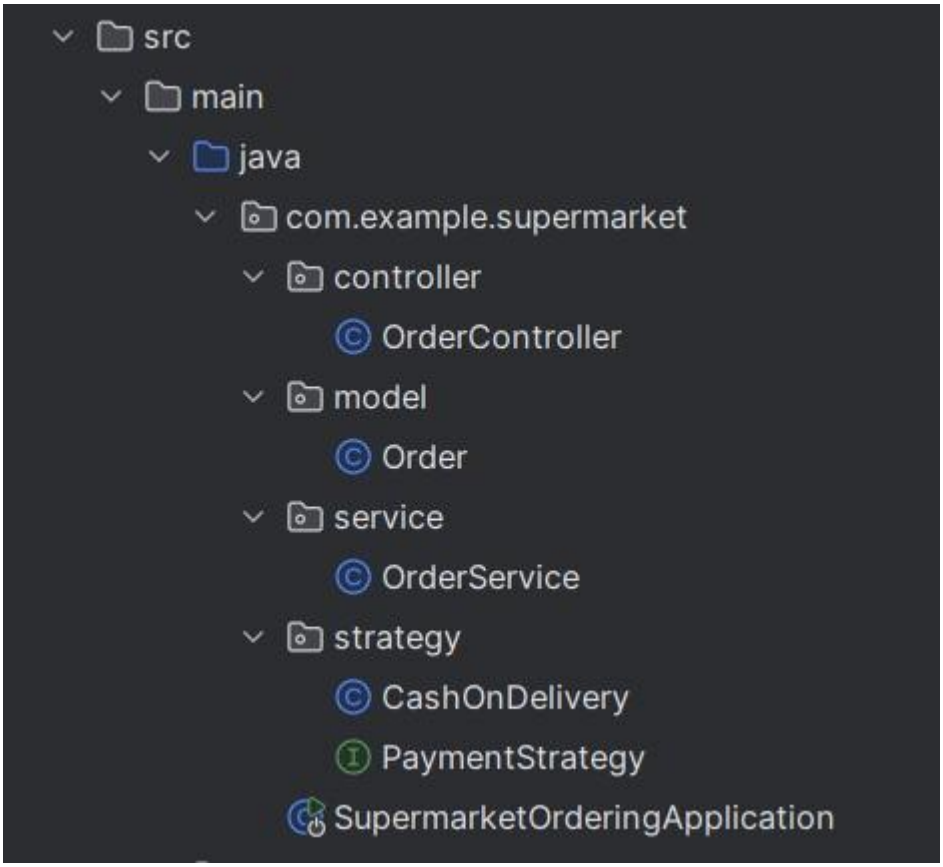
In our system:
The Singleton pattern ensures that only one instance of **OrderService** exists within the application.
This is handled automatically by Spring Boot each class annotated with @Service or @RestController is created once and reused everywhere.

Why it was chosen:
If multiple instances of the service existed, it could cause inconsistent order management and duplicate order processing.
With the Singleton pattern, all operations are managed through a single service instance, ensuring consistent data handling.

How it improves the system:

1. **Centralized management** – All order processing and payment handling go through one service instance.
2. **Consistency** – Prevents duplicate or conflicting operations.
3. **Efficiency** – Reduces memory overhead by avoiding multiple unnecessary instances.

Screenshots of Implementation



Showing the implementation of strategy pattern

```java
package com.example.supermarket.strategy;

public interface PaymentStrategy {  no usages  1 implementation
    void pay(double amount);  no usages  1 implementation
}
```

Strategy interface

```java
package com.example.supermarket.strategy;

import org.springframework.stereotype.Component;

@Component
public class CashOnDelivery implements PaymentStrategy {

    @Override  no usages
    public void pay(double amount) {
        System.out.println("Payment of Rs." + amount + " will be collected upon delivery (Cash on Delivery).");
    }
}
```

Concrete class (CashOnDelivery .java)

```java
package com.example.supermarket.strategy;

import org.springframework.stereotype.Component;

@Component
public class CashOnDelivery implements PaymentStrategy {

    @Override  no usages
    public void pay(double amount) {
        System.out.println("Payment of Rs." + amount + " will be collected upon delivery (Cash on Delivery).");
    }
}
```

Service class (OrderService.java)

```java
package com.example.supermarket.service;

import com.example.supermarket.model.Order;
import com.example.supermarket.strategy.CashOnDelivery;
import com.example.supermarket.strategy.PaymentStrategy;
import org.springframework.stereotype.Service;

@Service  no usages
public class OrderService {

    private final PaymentStrategy paymentStrategy;  2 usages

    public OrderService(CashOnDelivery cashOnDelivery) {
        this.paymentStrategy = cashOnDelivery;
    }

    public String processOrder(Order order) {  no usages
        paymentStrategy.pay(order.getTotalAmount());
        return "Order for " + order.getCustomerName() + " processed successfully with Cash on Delivery.";
    }
}
```

Model Pattern

```java
package com.example.supermarket.model;

public class Order {  no usages
    private Long id;  3 usages
    private String customerName;  3 usages
    private double totalAmount;  3 usages

    public Order(Long id, String customerName, double totalAmount) {  no usages
        this.id = id;
        this.customerName = customerName;
        this.totalAmount = totalAmount;
    }

    public Long getId() { return id; }  no usages
    public String getCustomerName() { return customerName; }  no usages
    public double getTotalAmount() { return totalAmount; }  no usages

    public void setId(Long id) { this.id = id; }  no usages
    public void setCustomerName(String customerName) { this.customerName = customerName; }  no usages
    public void setTotalAmount(double totalAmount) { this.totalAmount = totalAmount; }  no usages
}
```

Main class

```java
package com.example.supermarket;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SupermarketOrderingApplication {

    public static void main(String[] args) { SpringApplication.run(SupermarketOrderingApplication.class, args); }

}
```