**Behavioral Design Patterns**

**Behavioral patterns** focus on how objects interact, ensuring responsibility distribution and communication flow among them. Let’s explore them using real-world e-commerce examples:

**Problem:**

An e-commerce site offers various discounts (seasonal sale, loyalty discount, coupon-based). These discount strategies **change frequently** and should be **interchangeable**

**Solution:**

Use the Strategy Pattern to define a family of discount algorithms, encapsulate them, and make them **interchangeable** at runtime.

Benefits:

✔ Easily switch between different discount strategies.

✔ Open/Closed Principle compliance (add new discounts without changing existing code).

package org.example;

import org.example.behavioral.strategy.\*;

public class Main {

public static void main(String[] args) throws InterruptedException {

ShoppingCart shoppingCart = new ShoppingCart(new SeasonalDiscountStrategy());

System.*out*.println(

"Final price after the seasonal discount is : " +

shoppingCart.checkout(1000.00)

);

shoppingCart = new ShoppingCart( new LoyaltyDiscountStrategy());

System.*out*.println(

"Final price after the loyalty discount is : " +

shoppingCart.checkout(1000.00)

);

shoppingCart = new ShoppingCart( new CouponDiscountStrategy());

System.*out*.println(

"Final price after the coupon discount is : " +

shoppingCart.checkout(1000.00)

);

shoppingCart = new ShoppingCart(new VIPMemberDiscountStrategy());

System.*out*.println(

"Final price after the VIP discount is : " +

shoppingCart.checkout(1000.00)

);

}

}

package org.example.behavioral.strategy;

public class NoDiscountStrategy implements DiscountStrategy{

@Override

public double applyDiscount(double amount) {

return amount;

}

}

package org.example.behavioral.strategy;

public class CouponDiscountStrategy implements DiscountStrategy{

@Override

public double applyDiscount(double amount) {

return amount \* 0.80;

}

}

package org.example.behavioral.strategy;

public class ShoppingCart {

private DiscountStrategy discountStrategy;

public ShoppingCart(DiscountStrategy discountStrategy) {

this.discountStrategy = discountStrategy;

}

public double checkout(double price){

return discountStrategy.applyDiscount(price);

}

}

package org.example.behavioral.strategy;

public class VIPMemberDiscountStrategy implements DiscountStrategy{

@Override

public double applyDiscount(double amount) {

return amount \* 0.65;

}

}

**Problem:**

After placing an order, customers want real-time notifications (via email, SMS, or app push notifications) as the order moves through different stages (packed, shipped, delivered).

**Solution:**

Use the Observer Pattern to automatically notify all registered observers (subscribers) when the order status changes.

package org.example.behavioral.observer;

import org.example.creational.factory.EmailNotification;

import org.example.creational.factory.Notification;

public class EmailNotifier implements OrderObserver{

Notification emailNotification;

@Override

public void update(String status) {

emailNotification = new EmailNotification();

emailNotification.notifyUser(status);

}

}

**package org.example.creational.factory;**

**public class EmailNotification implements Notification{**

**@Override**

**public void notifyUser(String message) {**

**System.*out*.println("Sending an email notification to the user about their order it's now ...." + message);**

**}**

**}**

**package org.example.behavioral.observer;**

**public interface OrderObserver {**

**void update(String status);**

**}**

**package org.example;**

**import org.example.behavioral.observer.\*;**

**public class Main {**

**public static void main(String[] args) throws InterruptedException {**

**Order order = new Order();**

**order.addObserver(new EmailNotifier());**

**order.addObserver(new SMSNotifier());**

**order.addObserver(new WhatsappNotifier());**

**order.addObserver(new TelegramNotifier());**

**order.updateStatus("Shipped");**

**System.*out*.println();**

**Thread.*sleep*(1000);**

**order.updateStatus("Out for delivery");**

**System.*out*.println();**

**Thread.*sleep*(1000);**

**order.updateStatus("Delivered");**

**}**

**}**

**package org.example.behavioral.observer;**

**public class SMSNotifier implements OrderObserver{**

**@Override**

**public void update(String status) {**

**System.*out*.println("SMS : Your shipment is now at: " + status);**

**}**

**}**

**package org.example.behavioral.observer;**

**public class TelegramNotifier implements OrderObserver{**

**@Override**

**public void update(String status) {**

**System.*out*.println("Telegram : the order is now : " + status);**

**}**

**}**

**Benefits:**

✔ Automatic updates to all interested observers.

✔ Loose coupling between the subject and observers.

**Problem:**

Users should be able to place, cancel, or reorder items. The system needs a history of actions for undo/redo functionality.

**Solution:**

Use the Command Pattern to encapsulate actions as objects, allowing undo/redo functionality.

package org.example.behavioral.command;

import org.example.structural.decorator.Product;

public class PlaceOrderCommand implements Command {

private OrderService orderService;

public PlaceOrderCommand(OrderService orderService) {

this.orderService = orderService;

}

@Override

public void execute(Product product) {

orderService.placeOrder(product);

}

@Override

public void undo(Product product) {

orderService.cancelOrder();

}

}

package org.example;

import org.example.behavioral.command.Command;

import org.example.behavioral.command.OrderService;

import org.example.behavioral.command.PlaceOrderCommand;

import org.example.structural.composite.Item;

import org.example.structural.decorator.Product;

public class Main {

public static void main(String[] args) throws InterruptedException {

Product ps5 = new Item(50000.00, "PS5 Console");

OrderService orderService = new OrderService();

Command placeOrderCommand = new PlaceOrderCommand(orderService);

placeOrderCommand.execute(ps5);

}

}

package org.example.behavioral.command;

import org.example.structural.decorator.Product;

public interface Command {

void execute(Product product);

void undo(Product product);

}

package org.example.behavioral.command;

import org.example.structural.decorator.Product;

import org.example.structural.facade.OrderFacade;

public class OrderService {

OrderFacade orderFacade = new OrderFacade();

public void placeOrder(Product product){

orderFacade.placeOrder(product);

}

public void cancelOrder(){

System.*out*.println("Cancelling the order...");

}

}

Benefits:

✔ Encapsulates requests as objects.

✔ Supports undo/redo operations.