Creational Design Patterns

Github Link : <https://github.com/axatgupta/lld>

They provide various object creation mechanisms, which increase the flexibility and reuse of existing code.

Problem : E-commerce platforms (Amazon, Flipkart, Myntra ..)

1. Payment Gateway Integration :

Singleton Design Pattern : Payment Gateway (ensuring only one payment processor instance)

Razorpay, Stripe, PhonePe …

package org.example.creational.singleton;

public class PaymentGateway {

private static volatile PaymentGateway *paymentGatewayInstance*;

public PaymentGateway(){

System.*out*.println("Initializing one instance ...");

}

// Double locking mechanism

public synchronized static PaymentGateway getPaymentGatewayInstance() {

if(*paymentGatewayInstance* == null){

synchronized (PaymentGateway.class){

if(*paymentGatewayInstance* == null){

*paymentGatewayInstance* = new PaymentGateway();

}

}

}

return *paymentGatewayInstance*;

}

public void processPayment(String orderId, double amount){

System.*out*.println(" Processing the amount of : " + amount + " against the order id : " + orderId);

}

}

Benefits:

✔ Prevents multiple payment processor instances.

✔ Ensures thread safety in multi-threaded payment transactions.

Factory Design Pattern

Problem : you want to build a notification system to inform all the users about the upcoming sale. Email, SMS, Mobile Notification.

package org.example.creational.factory;

public interface Notification {

void notifyUser(String message);

}

package org.example.creational.factory;

public class SMSNotification implements Notification{

private static volatile SMSNotification *smsNotificationInstance*;

SMSNotification(){

System.*out*.println(" Initializing the smsNotificaitonInstance ... ");

}

public static SMSNotification getSmsNotificationInstance(){

if(*smsNotificationInstance* == null){

synchronized (SMSNotification.class){

if(*smsNotificationInstance* == null){

*smsNotificationInstance* = new SMSNotification();

}

}

}

return *smsNotificationInstance*;

}

@Override

public void notifyUser(String message) {

System.*out*.println("Sending an SMS to the user about the GRAND SALE !!! XOXO" + message);

}

}

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.creational.factory;

public class NotificationFactory {

public static Notification createNotification(String type){

if(type.equals("Email")){

return new EmailNotification();

} else if ( type.equals("SMS")) {

return new SMSNotification();

} else if(type.equals("Whatsapp")) {

return new WhatsappNotification();

} else if (type.equals("Telegram")){

return new TelegramNotification();

}else {

throw new IllegalArgumentException("Invalid notification type");

}

}

}

Benefits:

✔ Reduces tight coupling between the system and notification types.

✔ Easily add new notification types without modifying the existing system.

Builder Design Pattern

Problem : Customers can customize size, color, bankOffer, giftWrap to their respective products, how will this need be accommodated at a code level.

Benefits:

✔ Step-by-step construction of an order.

✔ Avoids constructor explosion for optional features.

package org.example.creational.builder;

import lombok.Builder;

@Builder

public class Order {

private String product;

private String size;

private String bookType;

private boolean giftWrapped;

private String color;

private boolean bankOfferApplicable;

private boolean primeDelivery;

public Order(OrderBuilder orderBuilder){

this.product = orderBuilder.product;

this.size = orderBuilder.size;

this.bookType = orderBuilder.bookType;

this.giftWrapped = orderBuilder.giftWrapped;

this.color = orderBuilder.color;

this.bankOfferApplicable = orderBuilder.bankOfferApplicable;

this.primeDelivery = orderBuilder.primeDelivery;

}

public static class OrderBuilder {

private String product;

private String size;

private String bookType;

private boolean giftWrapped;

private String color;

private boolean bankOfferApplicable;

private boolean primeDelivery;

public OrderBuilder(String product){

this.product = product;

}

public OrderBuilder size(String size){

this.size = size;

return this;

}

public OrderBuilder bookType (String bookType){

this.bookType = bookType ;

return this;

}

public OrderBuilder color (String color){

this.color = color;

return this;

}

public OrderBuilder giftWrapped (boolean giftWrapped ){

this.giftWrapped = giftWrapped;

return this;

}

public Order build(){

return new Order(this);

}

}

}

package org.example;

import org.example.behavioral.command.Command;

import org.example.behavioral.command.OrderService;

import org.example.behavioral.command.PlaceOrderCommand;

import org.example.creational.builder.Order;

import org.example.creational.factory.\*;

import org.example.creational.singleton.PaymentGateway;

import org.example.structural.composite.Item;

import org.example.structural.decorator.Product;

public class Main {

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

Order laptop = new Order.OrderBuilder("Laptop")

.size("15 inch")

.color("Black")

.build();

Order book = new Order.OrderBuilder("Harry Potter")

.color("Blue")

.bookType("Fiction")

.bankOfferApplicable(true)

.giftWrapped(true)

.primeDelivery(true)

.build();

}

}