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

insta

Structural patterns focus on how classes and objects are composed to form larger structures, ensuring flexible and efficient designs. Let’s explore them using real-world e-commerce examples:

**Problem:**

An e-commerce platform supports multiple payment providers (PayPal, Stripe, Razorpay, PhonePe, GPay, Jupiter), each with a different interface. We need a standardized way to integrate them.

### **Solution:**Use the **Adapter Pattern** to **convert** the interface of a third-party payment gateway into a **common interface** expected by the application.

### **Benefits:**

✔ Integrates **incompatible interfaces** without modifying code.  
✔ Adds **new payment providers** without changing the existing system.

package org.example;

import org.example.creational.Order;

import org.example.structural.\*;

public class Main {

public static void main(String[] args) {

PaymentPreprocessor razorpayPaymentPreprocessor = new RazorpayAdapter(new Razorpay());

razorpayPaymentPreprocessor.pay(1000.00);

PaymentPreprocessor stripePaymentPreprocessor = new StripeAdapter(new Stripe());

stripePaymentPreprocessor.pay(100.00);

}

}

package org.example.structural;

public class Razorpay {

public void makePayment(double amount){

System.*out*.println("Payment of : " + amount + " has been paid via Razorpay" );

}

}

package org.example.structural;

public class Stripe {

public void payment(double amount){

System.*out*.println("Payment of : " + amount + " has been paid via Stripe..");

}

}

package org.example.structural;

public class RazorpayAdapter implements PaymentPreprocessor{

private Razorpay razorpay;

public RazorpayAdapter(Razorpay razorpay){

razorpay = Razorpay.getInstance();

}

@Override

public void pay(double amount) {

razorpay.makePayment(amount);

}

}

package org.example.structural;

public class StripeAdapter implements PaymentPreprocessor{

Stripe stripe;

public StripeAdapter(Stripe stripe){

this.stripe = stripe;

}

@Override

public void pay(double amount) {

stripe.payment(amount);

}

}

### **Problem:**

Customers want to **customize products** (e.g., **gift wrap, extended warranty, personalization**). These features should be **added dynamically** without changing the core product class.

### **Solution:**

Use the **Decorator Pattern** to **add responsibilities** to objects **dynamically**.

package org.example;

import org.example.creational.Order;

import org.example.structural.\*;

public class Main {

public static void main(String[] args) {

Product laptop = new BasicProduct("Laptop", 10000.00);

laptop = new GiftWrap(laptop);

laptop = new ExtendedWarranty(laptop);

System.*out*.println(laptop.description() + " | " + laptop.getPrice());

}

}

package org.example.structural;

public interface Product {

String description();

double getPrice();

}

package org.example.structural;

public class BasicProduct implements Product{

private String description;

private double price;

public BasicProduct(String description, double price) {

this.description = description;

this.price = price;

}

@Override

public String description() {

return description;

}

@Override

public double getPrice() {

return price;

}

}

package org.example.structural;

public abstract class ProductDecorator implements Product{

protected Product product;

public ProductDecorator( Product product ){

this.product = product;

}

}

package org.example.structural;

public class GiftWrap extends ProductDecorator{

public GiftWrap(Product product){

super(product);

}

@Override

public String description() {

return product.description() + " has been decorated by a gift wrap";

}

@Override

public double getPrice() {

return product.getPrice() + 5.00;

}

}

package org.example.structural;

public class ExtendedWarranty extends ProductDecorator{

public ExtendedWarranty(Product product) {

super(product);

}

@Override

public String description() {

return product.description() + " has extended warranty now!";

}

@Override

public double getPrice() {

return product.getPrice() + 10.00;

}

}

### 

### 

### **Benefits:**

✔ **Dynamic customization** of products.  
✔ **Flexible addition** of features without modifying the base product.

### **Problem:**

An e-commerce site sells **product bundles** (e.g., **Gaming Bundle**: Console + Games + Controllers). Each bundle should behave **like a single product** in the shopping cart.

### **Solution:**

Use the **Composite Pattern** to treat **individual products and bundles uniformly**.

package org.example;

import org.example.structural.Item;

import org.example.structural.ProductBundle;

public class Main {

public static void main(String[] args) {

Item ps5 = new Item("Gaming console", 50000.00);

Item ps5Controller = new Item("Gaming Controller", 10000.00);

Item fifa = new Item(" Video Game FIFA ", 3000.00);

ProductBundle summerSaleGamingBundle = new ProductBundle(" Gaming Bundle for the summer season");

summerSaleGamingBundle.addProductToProductBundle(ps5);

summerSaleGamingBundle.addProductToProductBundle(ps5Controller);

summerSaleGamingBundle.addProductToProductBundle(fifa);

System.*out*.println(summerSaleGamingBundle.description()+ ", Total Price : " + summerSaleGamingBundle.getPrice());

}

}

package org.example.structural;

public interface Product {

String description();

double getPrice();

}

package org.example.structural;

public class Item implements Product{

private String description;

private double price;

public Item(String description, double price) {

this.description = description;

this.price = price;

}

@Override

public String description() {

return description;

}

@Override

public double getPrice() {

return price;

}

}

package org.example.structural;

import java.util.ArrayList;

import java.util.List;

public class ProductBundle implements Product{

private String description;

private List<Product> productList = new ArrayList<>();

public ProductBundle(String description){

this.description = description;

}

public void addProductToProductBundle(Product product){

productList.add(product);

}

@Override

public String description() {

return description;

}

@Override

public double getPrice() {

// double totalPrice = 0.0;

// for(Product product : productList){

// totalPrice+=product.getPrice();

// }

// return totalPrice;

return productList.stream().mapToDouble(Product::getPrice).sum();

}

}

### **Benefits:**

✔ Treat **individual products and bundles** uniformly.  
✔ Simplifies **cart and order processing** for complex product sets.

### **Problem:**

The **order processing** involves multiple subsystems (**payment, inventory, shipping, notifications**). Calling each subsystem separately increases **complexity**.

### **Solution:**

Use the **Facade Pattern** to provide a **simple interface** for order processing.

package org.example;

import org.example.structural.OrderFacade;

public class Main {

public static void main(String[] args) {

OrderFacade orderFacade = new OrderFacade();

orderFacade.placeOrder("tablet", 15000.00);

}

}

package org.example.structural;

public class OrderFacade {

private Inventory inventory;

private Payment payment;

private Shipping shipping;

public OrderFacade(){

inventory = new Inventory();

payment = new Payment();

shipping = new Shipping();

}

public void placeOrder(String item, double amount){

System.*out*.println("Processing the order for : " + item);

inventory.updateStock(item);

payment.processPayment(amount);

shipping.arrangeShipping(item);

System.*out*.println("Order successfully placed!");

}

}

package org.example.structural;

public class Inventory {

public void updateStock(String item){

System.*out*.println("Inventory updated for the item : " + item);

}

}

package org.example.structural;

public class Payment {

public void processPayment(double amount){

System.*out*.println(" Payment processed for the amount : " + amount);

}

}

package org.example.structural;

public class Shipping {

public void arrangeShipping(String item){

System.*out*.println(" Item : " + item + " has been shipped, it will reach you shortly!");

}

}

### **Benefits:**

✔ **Simplifies** complex subsystem interactions.  
✔ Provides a **single entry point** for the order process.