

How to remember Structural Design patterns:

BD🡪Budedu , PFAC-🡪PF acoount

BD 🡪(Behaviroul, Decorator)Budedu, PFFAC🡪 (Proxy, Flyweight, Façade,Adapter,Composite)PF account

1. **Façade design pattern**

## Real-World Analogy

* 🏨 **Hotel Front Desk** = Facade
* They internally talk to:
  + Housekeeping
  + Kitchen
  + Billing

But you (the customer) only deal with the **front desk**.

**public class TravelFacade {**

**private HotelBooking hotel;**

**private FlightBooking flight;**

**private CarRental car;**

**public TravelFacade() {**

**this.hotel = new HotelBooking();**

**this.flight = new FlightBooking();**

**this.car = new CarRental();**

**}**

**public void bookTrip() {**

**hotel.bookRoom();**

**flight.bookFlight();**

**car.rentCar();**

**System.out.println("Trip booking completed.");**

**}**

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

**TravelFacade facade = new TravelFacade();**

**facade.bookTrip(); // Client doesn't know how the internals work**

**}**

**}**

1. **Adapter**

**🧩 Real-World Analogy: Charger Plug Adapter**

**Scenario:**

* You have a **Laptop** that only accepts **Type-C** chargers.
* You have an **old USB-A charger**.
* You need an **Adapter** that converts the USB-A output to fit into the Type-C port.

**🎯 Goal in Code**

* Laptop expects a TypeCCharger interface
* You only have a USBACharger (legacy)
* You use USBAtoTypeCAdapter to convert the interface

**💻 Java Code**

**✅ Step 1: Target Interface (Type-C)**

// The expected interface by the laptop

public interface TypeCCharger {

void chargeViaTypeC();

}

**✅ Step 2: Adaptee Class (USB-A charger)**

// The incompatible legacy class

public class USBACharger {

public void chargeViaUSBA() {

System.out.println("Charging with USB-A charger.");

}

}

**✅ Step 3: Adapter Class**

// The adapter that makes USBA work with Type-C interface

public class USBAtoTypeCAdapter implements TypeCCharger {

private USBACharger usbaCharger;

public USBAtoTypeCAdapter(USBACharger usbaCharger) {

this.usbaCharger = usbaCharger;

}

@Override

public void chargeViaTypeC() {

System.out.println("Adapter converting USB-A to Type-C...");

usbaCharger.chargeViaUSBA();

}

}

1. **Proxy design pattern.**

Great question again! The **Proxy Design Pattern** may *look similar* to Adapter and Facade (since it also wraps another object), but it has a **very different purpose**.

Let’s explain it fully with:

* ✅ Concept
* ✅ Real-world analogy
* ✅ Java code example
* ✅ Comparison with Adapter and Facade

**🧱 What is the Proxy Design Pattern?**

The **Proxy Pattern** provides a **placeholder or surrogate** for another object to **control access** to it.

Instead of directly calling the real object, clients go through a **proxy object** that can add **extra behavior** like:

* Security checks 🔐
* Lazy loading 💤
* Caching 🗃️
* Access control 🚧
* Logging 📝

**💡 Real-World Analogy**

🕵️ **Credit card = Proxy for your bank account**

* You don’t give your **bank account directly** to the merchant.
* You give your **credit card**, which acts as a **proxy**.
* It controls:
  + ✅ Who can charge you
  + ✅ How much
  + ✅ When

public interface Payment {

void pay(double amount) throws Exception;

}

public class BankAccount implements Payment {

private double balance;

public BankAccount(double balance) {

this.balance = balance;

}

@Override

public void pay(double amount) throws Exception {

if (amount > balance) {

throw new Exception("Insufficient funds in bank account.");

}

balance -= amount;

System.out.println("Paid ₹" + amount + " using bank account. Remaining: ₹" + balance);

}

}

public class CreditCardProxy implements Payment {

private BankAccount bankAccount;

public CreditCardProxy(BankAccount bankAccount) {

this.bankAccount = bankAccount;

}

@Override

public void pay(double amount) throws Exception {

System.out.println("Attempting payment via credit card proxy...");

// Add logging, fraud checks, limits etc.

bankAccount.pay(amount); // delegate to real subject

System.out.println("Transaction logged successfully.\n");

}

}

**Saga Orchestration Pattern**

**🎯 What is it?**

A **central coordinator** (orchestrator) manages the entire saga by telling each service **what to do next**.

Example

**OrchestratorService is the orchestrator ✅**

java

CopyEdit

@Service

public class OrchestratorService {

@Autowired OrderService orderService;

@Autowired PaymentService paymentService;

@Autowired InventoryService inventoryService;

public String placeOrder(OrderRequest request) {

boolean orderCreated = orderService.createOrder(request);

if (!orderCreated) return "Order Failed";

boolean paid = paymentService.deductPayment(request);

if (!paid) {

orderService.cancelOrder(request);

return "Payment Failed";

}

boolean stock = inventoryService.reduceStock(request);

if (!stock) {

paymentService.refund(request);

orderService.cancelOrder(request);

return "Inventory Failed";

}

return "Order Successful!";

}

}

## 💃 Choreography Pattern

### 🎯 What is it?

There is **no central coordinator** — each service **reacts to events** from the previous service and decides what to do next.

## 🧩 Scenario: Order Placement Flow (Choreography Style)

**Services:**

1. OrderService → emits OrderCreatedEvent
2. PaymentService → listens to OrderCreatedEvent, emits PaymentCompletedEvent
3. InventoryService → listens to PaymentCompletedEvent, emits InventoryUpdatedEvent
4. ShippingService → listens to InventoryUpdatedEvent

## Shared Model (Used across services)

java

CopyEdit

// OrderEvent.java

public class OrderEvent {

private String orderId;

private String status; // CREATED, PAID, FAILED, etc.

private String source; // ORDER, PAYMENT, INVENTORY

// + Getters/Setters

}

## ✅ 1. Order Service

java

CopyEdit

@RestController

@RequestMapping("/order")

public class OrderController {

@Autowired KafkaTemplate<String, OrderEvent> kafkaTemplate;

@PostMapping

public ResponseEntity<String> placeOrder(@RequestBody OrderEvent event) {

event.setStatus("CREATED");

event.setSource("ORDER");

kafkaTemplate.send("order-events", event);

return ResponseEntity.ok("Order created and event published.");

}

}

## ✅ 2. Payment Service

java

CopyEdit

@Component

public class PaymentListener {

@Autowired KafkaTemplate<String, OrderEvent> kafkaTemplate;

@KafkaListener(topics = "order-events", groupId = "payment-group")

public void listen(OrderEvent event) {

if ("CREATED".equals(event.getStatus())) {

System.out.println("Processing payment for order: " + event.getOrderId());

// Simulate payment success

event.setStatus("PAID");

event.setSource("PAYMENT");

kafkaTemplate.send("order-events", event);

}

}

}

## ✅ 3. Inventory Service

java

CopyEdit

@Component

public class InventoryListener {

@Autowired KafkaTemplate<String, OrderEvent> kafkaTemplate;

@KafkaListener(topics = "order-events", groupId = "inventory-group")

public void listen(OrderEvent event) {

if ("PAID".equals(event.getStatus())) {

System.out.println("Reducing stock for order: " + event.getOrderId());

// Simulate stock update

event.setStatus("INVENTORY\_UPDATED");

event.setSource("INVENTORY");

kafkaTemplate.send("order-events", event);

}

}

}

## ✅ 4. Shipping Service

java

CopyEdit

@Component

public class ShippingListener {

@KafkaListener(topics = "order-events", groupId = "shipping-group")

public void listen(OrderEvent event) {

if ("INVENTORY\_UPDATED".equals(event.getStatus())) {

System.out.println("Shipping order: " + event.getOrderId());

// Final service, no new event needed

}

}

}