Palash Bajpai

Low Level Design Questions

Table of Contents

[**SOLID Principles** 2](#_Toc195079233)

[**1.** **Multi-Tier Employee Management System** 2](#_Toc195079234)

[**2.** **E-Commerce Payment System** 5](#_Toc195079235)

[**3.** **Ride-Sharing System** 8](#_Toc195079236)

[**Creational Design Pattern** 12](#_Toc195079237)

[**1.** **Singleton Pattern** 12](#_Toc195079238)

[**2.** **Abstract Factory Pattern** 14](#_Toc195079239)

[**Structural Design Pattern** 18](#_Toc195079240)

[**1.** **Facade Pattern** 18](#_Toc195079241)

[**Behavioural Design Pattern** 20](#_Toc195079242)

[**1. Strategy Pattern** 20](#_Toc195079243)

[**2. Command Pattern** 22](#_Toc195079244)

# **SOLID Principles**

1. Single Responsibility Principle (SRP): A class should have only one reason to change.
2. Open/Closed Principle (OCP): Code should be open for extension but closed for modification.
3. Liskov Substitution Principle (LSP): Derived classes should be substitutable for their base classes.
4. Interface Segregation Principle (ISP): Avoid forcing classes to implement unnecessary methods of an interface.
5. Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules. Both should depend on abstractions

## **Multi-Tier Employee Management System**

You are designing an Employee Management System where there are Permanent Employees, Contract Employees, and Interns.

**The system should support:**

* Permanent employees getting salary and bonus.
* Contract employees getting only salary.
* Interns only having an ID and name, but no salary or bonus.

Update the existing code so that new code follows SOLID Principles:

**Existing Code**

class Employee {  
 String name;  
 double salary;  
  
 public double getSalary() {  
 return salary;  
 }  
  
 public double getBonus() {  
 return salary \* 0.1;  
 }  
}

class PermanentEmployee extends Employee {  
 public double getBonus() {  
 return salary \* 0.2; // Higher bonus  
 }  
}

class ContractEmployee extends Employee {  
 // No bonus for contract employees  
}  
  
class Intern extends Employee {  
 public double getSalary() {  
 throw new UnsupportedOperationException("Interns do not receive a salary");  
 }  
}

**Tasks:**

* Apply LSP so that an Intern does not inherit unnecessary methods.
* Apply ISP by creating separate interfaces for salary-based and non-salary employees.
* Apply OCP so that we can add new employee types without modifying existing classes

SOLUTION:-

Above code do not follow SOLID as:

❌ Liskov Substitution Principle (LSP) because Intern extends Employee but has a getSalary() method that throws an exception.

❌ Interface Segregation Principle (ISP) because Employee has methods that Interns don’t need.

❌ Open/Closed Principle (OCP) because every time a new employee type is added, we must modify Employee.

// ISP - Separate interfaces for Salary and Bonus  
interface SalariedEmployee {  
 double getSalary();  
}  
  
interface BonusEligible {  
 double getBonus();  
}

// Permanent employees implement both interfaces  
class PermanentEmployee implements SalariedEmployee, BonusEligible {  
 private double salary;  
  
 public PermanentEmployee(double salary) {  
 this.salary = salary;  
 }  
  
 public double getSalary() {  
 return salary;  
 }  
  
 public double getBonus() {  
 return salary \* 0.2;  
 }  
}

// Contract employees implement only SalariedEmployee  
class ContractEmployee implements SalariedEmployee {  
 private double salary;  
  
 public ContractEmployee(double salary) {  
 this.salary = salary;  
 }  
  
 public double getSalary() {  
 return salary;  
 }  
}

// Intern does not inherit unnecessary methods  
class Intern {  
 private String name;  
  
 public Intern(String name) {  
 this.name = name;  
 }  
}

// OCP - PayrollProcessor can handle new employee types without modification  
class PayrollProcessor {  
 public void processSalary(SalariedEmployee employee) {  
 System.*out*.println("Processing salary: $" + employee.getSalary());  
 }  
  
 public void processBonus(BonusEligible employee) {  
 System.*out*.println("Processing bonus: $" + employee.getBonus());  
 }  
}

// Client Code  
public class EmployeeApp {  
 public static void main(String[] args) {  
 PermanentEmployee perm = new PermanentEmployee(5000);  
 ContractEmployee contract = new ContractEmployee(3000);  
 Intern intern = new Intern("John Doe");  
  
 PayrollProcessor processor = new PayrollProcessor();  
 processor.processSalary(perm);  
 processor.processSalary(contract);  
 processor.processBonus(perm);  
 }  
}

## **E-Commerce Payment System**

You are designing an e-commerce payment system where customers can pay using Credit Card, PayPal, or Cryptocurrency.

**Existing Code:**

class PaymentProcessor {  
 public void processPayment(String paymentType, double amount) {  
 if (paymentType.equals("CreditCard")) {  
 System.*out*.println("Processing credit card payment: $" + amount);  
 } else if (paymentType.equals("PayPal")) {  
 System.*out*.println("Processing PayPal payment: $" + amount);  
 } else if (paymentType.equals("Crypto")) {  
 System.*out*.println("Processing cryptocurrency payment: $" + amount);  
 }  
 }  
  
 public void logTransaction(double amount) {  
 System.*out*.println("Transaction logged: $" + amount);  
 }  
  
 public void sendEmailConfirmation(String email) {  
 System.*out*.println("Email sent to: " + email);  
 }  
}

**Tasks:**

* Refactor the design to follow SRP by separating payment processing, logging, and email notifications.
* Apply OCP to allow new payment methods to be added without modifying existing code.
* Use DIP so the PaymentProcessor uses abstractions, not concrete implementations.

SOLUTION:-

Above code do not follow SOLID as:

❌ The system currently violates the Single Responsibility Principle (SRP) because a single PaymentProcessor class does everything, including logging transactions and sending payment confirmation emails.

❌ It also violates Open/Closed Principle (OCP) because adding a new payment method requires modifying the existing class.

❌ It violates Dependency Inversion Principle (DIP) because PaymentProcessor directly depends on concrete classes instead of abstractions.

// Abstraction for Payment Methods  
interface PaymentMethod {  
 void process(double amount);  
}

// Concrete Implementations  
class CreditCardPayment implements PaymentMethod {  
 public void process(double amount) {  
 System.*out*.println("Processing Credit Card payment: $" + amount);  
 }  
}  
  
class PayPalPayment implements PaymentMethod {  
 public void process(double amount) {  
 System.*out*.println("Processing PayPal payment: $" + amount);  
 }  
}  
  
class CryptoPayment implements PaymentMethod {  
 public void process(double amount) {  
 System.*out*.println("Processing Cryptocurrency payment: $" + amount);  
 }  
}

// SRP - Separate Email Notification Responsibility  
class EmailNotifier {

public void sendEmail(String email) {  
 System.*out*.println("Email sent to: " + email);  
 }

}

// DIP - Payment Processor depends on abstraction  
class PaymentProcessor {  
 private PaymentMethod paymentMethod;  
 private TransactionLogger logger;  
 private EmailNotifier notifier;  
  
 public PaymentProcessor(PaymentMethod paymentMethod, TransactionLogger logger, EmailNotifier notifier) {  
 this.paymentMethod = paymentMethod;  
 this.logger = logger;  
 this.notifier = notifier;  
 }  
  
 public void processPayment(double amount, String email) {  
 paymentMethod.process(amount);  
 logger.log(amount);  
 notifier.sendEmail(email);  
 }  
}

// Client Code  
public class ECommerceApp {  
 public static void main(String[] args) {

PaymentProcessor processor = new PaymentProcessor(new PayPalPayment(), new TransactionLogger(), new EmailNotifier());

processor.processPayment(100.0, "customer@example.com");  
 }  
}

## **Ride-Sharing System**

You are designing a ride-sharing system like Uber/Ola where users can book different types of rides (Car, Bike, Auto).

**The System should:**

* Allow adding new ride types without modifying existing code.
* Ensure some rides support extra features (e.g., Cars have Air Conditioning but Bikes do not).
* Decouple payment from ride booking and ride notifications.

**Existing Code:**

class RideService {  
 public void bookRide(String type) {  
 if (type.equals("Car")) {  
 System.*out*.println("Booking Car Ride...");  
 } else if (type.equals("Bike")) {  
 System.*out*.println("Booking Bike Ride...");  
 }  
  
 // Payment Logic (SRP Violation)  
 System.*out*.println("Processing payment...");  
  
 // Notification Logic (SRP Violation)  
 System.*out*.println("Sending SMS Notification...");  
 }  
}

**Tasks:**

* Apply OCP to allow new ride types to be added without modifying RideService.
* Apply ISP so only Cars have ACControl().
* Apply DIP so RideService depends on abstractions.
* Apply SRP by separating ride booking, payments, and notifications.

SOLUTION:-

Above code does not follow SOLID as:

❌ Open/Closed Principle (OCP) because adding new ride types requires modifying the RideService class.

❌ Interface Segregation Principle (ISP) because all rides implement Ride even though only Cars have AC control.

❌ Dependency Inversion Principle (DIP) because RideService depends on concrete classes instead of abstractions.

❌ Single Responsibility Principle (SRP) because RideService handles ride booking, payments, and notifications.

// OCP - Ride interface  
interface Ride {  
 void book();  
}  
  
// ISP - Separate interface for AC rides  
interface ACControl {  
 void turnOnAC();  
}

// Concrete Ride Implementations  
class CarRide implements Ride, ACControl {  
 public void book() {  
 System.*out*.println("Booking Car Ride...");  
 }  
  
 public void turnOnAC() {  
 System.*out*.println("Turning on AC in Car...");  
 }  
}  
  
class BikeRide implements Ride {  
 public void book() {  
 System.*out*.println("Booking Bike Ride...");  
 }  
}  
  
// OCP - Adding Auto ride without modifying existing code  
class AutoRide implements Ride {  
 public void book() {  
 System.*out*.println("Booking Auto Ride...");  
 }  
}

// DIP - Payment abstraction  
interface PaymentMethod {  
 void processPayment(double amount);  
}

// Concrete Payment Methods  
class CreditCardPayment implements PaymentMethod {  
 public void processPayment(double amount) {  
 System.*out*.println("Processing Credit Card payment of $" + amount);  
 }  
}  
  
class WalletPayment implements PaymentMethod {  
 public void processPayment(double amount) {  
 System.*out*.println("Processing Wallet payment of $" + amount);  
 }  
}

// DIP - Notification abstraction  
interface Notifier {  
 void sendNotification(String message);  
}  
  
// Concrete Notification Implementations  
class SMSNotifier implements Notifier {  
 public void sendNotification(String message) {  
 System.*out*.println("Sending SMS: " + message);  
 }  
}

// SRP - RideService only handles booking rides  
class RideService {  
 private Ride ride;  
  
 public RideService(Ride ride) {  
 this.ride = ride;  
 }  
  
 public void bookRide() {  
 ride.book();  
 }  
}

// SRP - PaymentService handles payments  
class PaymentService {  
 private PaymentMethod paymentMethod;  
  
 public PaymentService(PaymentMethod paymentMethod) {  
 this.paymentMethod = paymentMethod;  
 }  
  
 public void pay(double amount) {  
 paymentMethod.processPayment(amount);  
 }  
}

// SRP - NotificationService handles notifications  
class NotificationService {  
 private Notifier notifier;  
  
 public NotificationService(Notifier notifier) {  
 this.notifier = notifier;  
 }  
  
 public void sendAlert(String message) {  
 notifier.sendNotification(message);  
 }  
}

// Client Code  
public class RideSharingApp {  
 public static void main(String[] args) {  
 RideService ride = new RideService(new CarRide());  
 ride.bookRide();  
  
 PaymentService payment = new PaymentService(new CreditCardPayment());  
 payment.pay(50.0);  
  
 NotificationService notification = new NotificationService(new SMSNotifier());  
 notification.sendAlert("Ride booked successfully!");  
 }  
}

# **Creational Design Pattern**

## **Singleton Pattern**

**Problem Statement:** Design a Printer Spooler System using the Singleton Design Pattern.

**Requirements:**

1. Only one printer spooler instance should exist in the system.

1. The system should support basic operations:

* addJob(String job) → Adds a print job to the queue.
* processJob() → Processes the next job in the queue.
* displayQueue() → Displays pending print jobs.

1. The printer spooler should be thread-safe to allow multiple threads to add jobs.
2. Implement lazy initialization to create the instance only when needed.

**Solution**

public class PrinterSpooler {

private static PrinterSpooler *instance*;  
 private Queue<String> printQueue;  
  
 private PrinterSpooler(){  
 printQueue = new PriorityQueue<>();  
 }  
  
 // Public method to get the singleton instance with thread safety  
 public static synchronized PrinterSpooler getInstance() {  
 if (*instance* == null) {  
 *instance* = new PrinterSpooler();  
 }  
 return *instance*;  
 }

// Add a job to the print queue  
 public synchronized void addJob(String job) {  
 printQueue.offer(job);  
 System.*out*.println("Job added: " + job);  
 }  
  
 // Process the next job in the queue  
 public synchronized void processJob() {  
 if (printQueue.isEmpty()) {  
 System.*out*.println("No print jobs in the queue.");  
 return;  
 }  
 String job = printQueue.poll();  
 System.*out*.println("Processing job: " + job);  
 }  
  
 // Display all jobs in the queue  
 public synchronized void displayQueue() {  
 System.*out*.println("Current Print Queue: " + printQueue);  
 }  
  
}

Client code

public class PrinterTest {  
 public static void main(String[] args) {  
 PrinterSpooler printer = PrinterSpooler.*getInstance*();  
 PrinterSpooler printer2 = PrinterSpooler.*getInstance*();  
  
 System.*out*.println(printer2==printer); //true as only once instance created  
  
 printer.addJob("Notes.pdf");  
 printer2.addJob("World.png");  
  
 printer.displayQueue();  
  
 printer.processJob();  
 printer.processJob();  
 printer.processJob();  
 }  
}

In a multithreaded environment, the Singleton pattern often uses synchronized methods of object creation to ensure that only one instance is created, even if multiple threads attempt to create an instance at the same time.

## **Abstract Factory Pattern**

**Problem Statement:** Design a Smart Home Automation System using the Abstract Factory Pattern.

**Requirements:**

1. The system should support multiple home types:

* Luxury Home
* Budget Home

1. Each home should have three types of smart devices:

* Smart Light (e.g., LuxuryLight, BudgetLight)
* Smart Door Lock (e.g., LuxuryLock, BudgetLock)

1. The system should be extendable so that new home types or devices can be added in the future.
2. Clients should not need to know which specific device implementation is used.

🔹 **Code**

// Step 1: Define Abstract Product Interfaces for Smart Devices  
interface SmartLight {  
 void turnOn();  
}  
  
interface SmartDoorLock {  
 void lock();  
}

// Step 2: Implement Concrete Products for Luxury Home  
class LuxuryLight implements SmartLight {  
 public void turnOn() {  
 System.*out*.println("Luxury Light is turned on with soft dimming effect.");  
 }  
}  
  
class LuxuryLock implements SmartDoorLock {  
 public void lock() {  
 System.*out*.println("Luxury Lock is secured with biometric authentication.");  
 }  
}

// Step 3: Implement Concrete Products for Budget Home  
class BudgetLight implements SmartLight {  
 public void turnOn() {  
 System.*out*.println("Budget Light is turned on with basic functionality.");  
 }  
}  
  
class BudgetLock implements SmartDoorLock {  
 public void lock() {  
 System.*out*.println("Budget Lock is secured with a standard key lock.");  
 }  
}

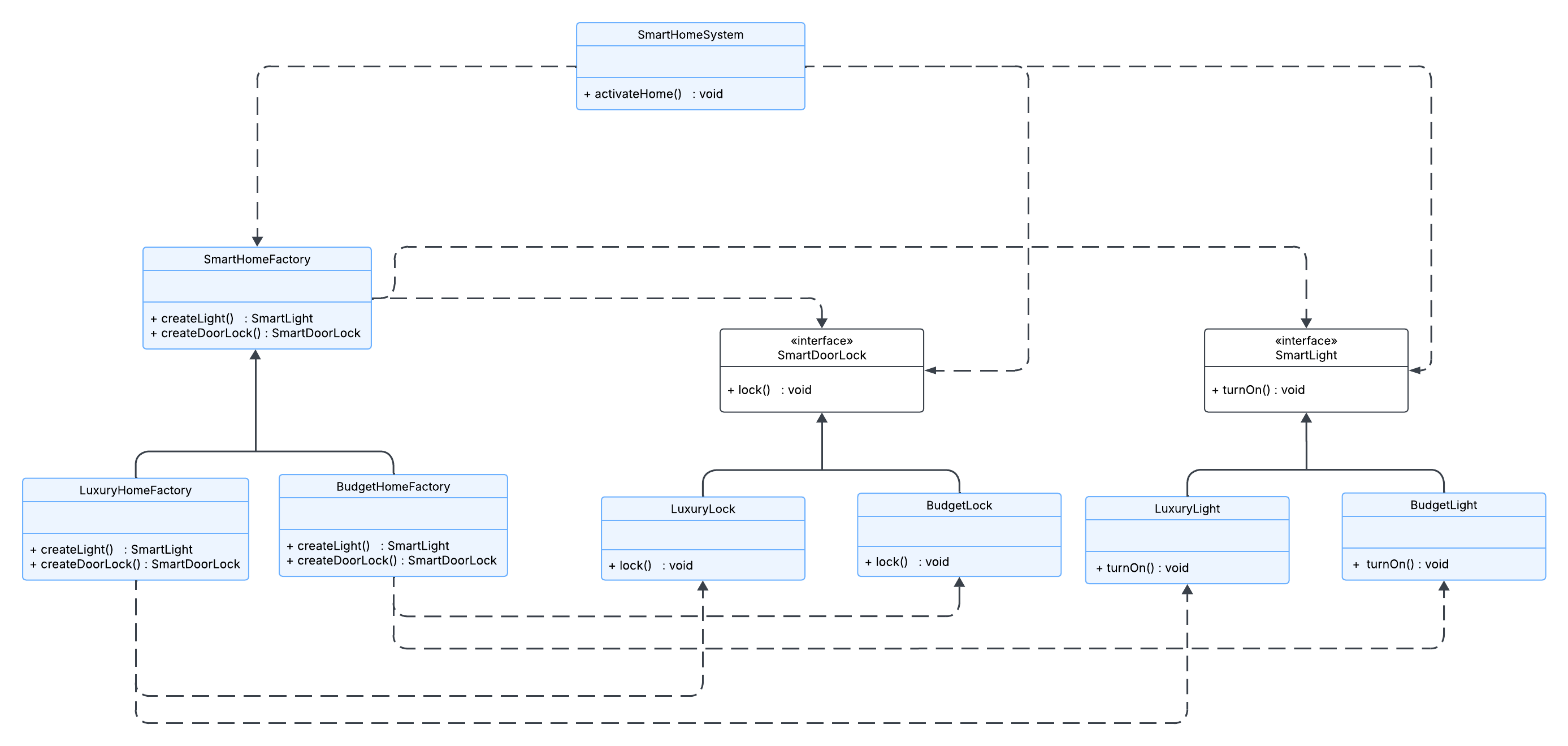
// Step 4: Define Abstract Factory Interface  
interface SmartHomeFactory {  
 SmartLight createLight();  
 SmartThermostat createThermostat();  
 SmartDoorLock createDoorLock();  
}

// Step 5: Implement Concrete Factories for Different Home Types  
class LuxuryHomeFactory implements SmartHomeFactory {  
 public SmartLight createLight() {  
 return new LuxuryLight();  
 }  
  
 public SmartDoorLock createDoorLock() {  
 return new LuxuryLock();  
 }  
}  
  
class BudgetHomeFactory implements SmartHomeFactory {  
 public SmartLight createLight() {  
 return new BudgetLight();  
 }  
  
 public SmartDoorLock createDoorLock() {  
 return new BudgetLock();  
 }  
}

// Step 6: Client Code that Uses the Abstract Factory  
class SmartHomeSystem {  
 private final SmartLight light;  
 private final SmartDoorLock doorLock;  
  
 public SmartHomeSystem(SmartHomeFactory factory) {  
 this.light = factory.createLight();  
 this.doorLock = factory.createDoorLock();  
 }  
  
 public void activateHome() {  
 System.*out*.println("Activating Smart Home...");  
 light.turnOn();  
 doorLock.lock();  
 }  
}

// Step 7: Testing the System  
public class AbstractFactoryExample {  
 public static void main(String[] args) {  
 System.*out*.println("Initializing Luxury Home:");  
 SmartHomeSystem luxuryHome = new SmartHomeSystem(new LuxuryHomeFactory());  
 luxuryHome.activateHome();  
 }  
}

🔹 **UML diagram**

­­­

**There is an dependency because LuxuryHomeFactory uses LuxuryLight and LuxuryLock to create and return instances.** **Specifically, this is a dependency relationship, where LuxuryHomeFactory depends on these classes but does not own them permanently.**

**Q. Why relationship between SmartHomeSystem and smartDoorLock and Smartlight is not composition, though objects are created inside its constructor?**

Composition (◆———>) means the contained objects (SmartLight, SmartThermostat, SmartDoorLock) must be created inside the class AND be destroyed with it.

However, in this code:

* The SmartHomeSystem does not create the objects directly—it gets them from SmartHomeFactory.
* The objects can exist independently of SmartHomeSystem (Factory is responsible for their creation).
* If SmartHomeSystem is destroyed, these objects may still exist (Factory creates them, so ownership is not exclusive).

This is special case when even the objects are attribute of parent class and are created inside it’s constructor. Still it is not a composition.

# **Structural Design Pattern**

## **Facade Pattern**

**Problem Statement:** Design a system where a customer can order different types of coffee (Espresso, Cappuccino, Latte) without worrying about the complex steps involved in coffee preparation.

**Requirements:**

* Create subsystems (Grinder, Brewer, MilkFrother).
* Create a Facade class (CoffeeMachineFacade) to simplify interactions.
* Provide a single method (makeCoffee()) for customers.

**Solution**

//Sub Systems  
class Grinder {  
 public void grindBeans() {  
 System.*out*.println("Grinding coffee beans..."); }  
}  
  
class Brewer {  
 public void brew() {  
 System.*out*.println("Brewing coffee..."); }  
}  
  
class MilkFrother {  
 public void frothMilk() {  
 System.*out*.println("Frothing milk..."); }  
}

class CoffeeMachineFacade {  
 private Grinder grinder;  
 private Brewer brewer;  
 private MilkFrother milkFrother;  
  
 public CoffeeMachineFacade() {  
 this.grinder = new Grinder();  
 this.brewer = new Brewer();  
 this.milkFrother = new MilkFrother();  
 }  
  
 public void makeCoffee(String type) {  
 System.*out*.println("\nPreparing " + type + "...");  
 grinder.grindBeans();  
 brewer.brew();  
 if (type.equalsIgnoreCase("Cappuccino") || type.equalsIgnoreCase("Latte")) {  
 milkFrother.frothMilk();  
 }  
 System.*out*.println(type + " is ready!\n");  
 }  
}

//Client code  
public class CoffeeShop {  
 public static void main(String[] args) {  
 CoffeeMachineFacade coffeeMachine = new CoffeeMachineFacade();  
 coffeeMachine.makeCoffee("Espresso");  
 coffeeMachine.makeCoffee("Cappuccino");  
 }  
}

# **Behavioural Design Pattern**

## **1. Strategy Pattern**

**Problem Statement:** You are building a Navigation App (like Google Maps). It supports different routes between a source and a destination.

Users should be able to select different routing strategies such as:

* Fastest Route
* Shortest Distance
* Avoid Traffic
* Scenic Route

The strategy must be selected dynamically at runtime, and new strategies should be easy to add without modifying the existing code.

**Solution**

**Step1: Define the RouteStrategy interface**

public interface RouteStrategy {  
 void calculateRoute(String from, String to);  
}

**Step2: Create Concrete Strategies**

public class FastestRouteStrategy implements RouteStrategy {  
 @Override  
 public void calculateRoute(String from, String to) {  
 System.*out*.println("Calculating the fastest route from " + from + " to " + to);  
 }  
}

public class ShortestDistanceStrategy implements RouteStrategy {  
 @Override  
 public void calculateRoute(String from, String to) {  
 System.*out*.println("Calculating the shortest distance route from " + from + " to " + to);  
 }  
}

public class AvoidTrafficStrategy implements RouteStrategy {  
 @Override  
 public void calculateRoute(String from, String to) {  
 System.*out*.println("Calculating a route avoiding traffic from " + from + " to " + to);  
 }  
}

public class ScenicRouteStrategy implements RouteStrategy {  
 @Override  
 public void calculateRoute(String from, String to) {  
 System.*out*.println("Calculating a scenic route from " + from + " to " + to);  
 }  
}

**Step3: Context Class — NavigationApp**

public class NavigationApp {  
 private RouteStrategy strategy;  
  
 public NavigationApp(RouteStrategy strategy) {  
 this.strategy = strategy;  
 }  
  
 public void setStrategy(RouteStrategy strategy) {  
 this.strategy = strategy;  
 }  
  
 public void navigate(String from, String to) {  
 strategy.calculateRoute(from, to);  
 }  
}

**Step4: Client Code**

public class StrategyNavigationDemo {  
 public static void main(String[] args) {  
 NavigationApp app = new NavigationApp(new FastestRouteStrategy());  
 app.navigate("Mumbai", "Pune");  
  
 app.setStrategy(new AvoidTrafficStrategy());  
 app.navigate("Mumbai", "Pune");  
  
 app.setStrategy(new ScenicRouteStrategy());  
 app.navigate("Mumbai", "Pune");  
 }  
}

## **2. Command Pattern**

**Problem Statement:** Design a remote control system for a TV that supports:

* Turning the TV ON and OFF
* Undoing the last action (e.g., if the user turns the TV ON, pressing undo will turn it OFF)
* Extensibility for more devices in the future
* Decoupling between button-press logic and actual device logic

**Solution**

**Step1: Command Interface**

public interface Command {  
 void execute();  
 void undo();  
}

**Step2: Receiver (TV)**

public class TV {  
 public void turnOn() {  
 System.*out*.println("TV is ON");  
 }  
  
 public void turnOff() {  
 System.*out*.println("TV is OFF");  
 }  
}

**Step3: Concreate Command Classes**

public class TVOnCommand implements Command {  
 private TV tv;  
  
 public TVOnCommand(TV tv) {  
 this.tv = tv;  
 }  
  
 public void execute() {  
 tv.turnOn();  
 }  
  
 public void undo() {  
 tv.turnOff();  
 }  
}

public class TVOffCommand implements Command {  
 private TV tv;  
  
 public TVOffCommand(TV tv) {  
 this.tv = tv;  
 }  
  
 public void execute() {  
 tv.turnOff();  
 }  
  
 public void undo() {  
 tv.turnOn();  
 }  
}

**Step4: Invoker (Remote Control)**

public class RemoteControl {  
 private Command currentCommand;  
 private Command lastCommand;  
  
 public void setCommand(Command command) {  
 this.currentCommand = command;  
 }  
  
 public void pressButton() {  
 currentCommand.execute();  
 lastCommand = currentCommand;  
 }  
  
 public void pressUndo() {  
 if (lastCommand != null) {  
 lastCommand.undo();  
 }  
 }  
}

**Step5: Client Code**

public class TVRemoteDemo {  
 public static void main(String[] args) {  
 TV tv = new TV();  
  
 Command tvOn = new TVOnCommand(tv);  
 Command tvOff = new TVOffCommand(tv);  
  
 RemoteControl remote = new RemoteControl();  
  
 remote.setCommand(tvOn);  
 remote.pressButton(); // Output: TV is ON  
  
 remote.pressUndo(); // Output: TV is OFF  
  
 remote.setCommand(tvOff);  
 remote.pressButton(); // Output: TV is OFF  
  
 remote.pressUndo(); // Output: TV is ON  
 }  
}