Palash Bajpai

Low Level Design Questions

Table of Contents

[**SOLID Principles** 2](#_Toc192340090)

[**1.** **Multi-Tier Employee Management System** 2](#_Toc192340091)

[**2.** **E-Commerce Payment System** 5](#_Toc192340092)

[**3.** **Ride-Sharing System** 8](#_Toc192340093)

# **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!");  
 }  
}