Object Oriented Design

Table of contents

> Tip
>
> Some of the these question only contain a code snippet which maybe incomplete, if that is the case, you can assume rest of the structure of the program (classes and methods) based on the provided code.

1 SOLID

1. The `UserProfile` class below handles both user data and user persistence logic. Refactor it to adhere to the Single Responsibility Principle.

```
public class UserProfile {
    private String name;
    private String email;

    public UserProfile(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // User data handling methods
    public void updateEmail(String newEmail) {
        this.email = newEmail;
    }

    // User persistence logic
    public void saveUser() {
        System.out.println("User saved: " + this.name);
        // Logic to save user to a database
    }
}
```

2. The `DiscountCalculator` class needs to be extended to support different types of discounts without modifying its existing code. Refactor it to comply with the Open/Closed Principle.

```
public class DiscountCalculator {
    public double calculateDiscount(double price) {
        return price * 0.2; // 20% discount
    }
}
```

3. The `Bird` class is extended by `Duck` and `Ostrich` classes. However, calling the fly method on an Ostrich instance doesn't make sense. Refactor the code structure to adhere to the Liskov Substitution Principle.

```
class Bird {
    public void fly(){
        System.out.println("Flying");
    }
}

class Duck extends Bird {
    // Duck specific behavior
}

class Ostrich extends Bird {
    // Ostrich specific behavior
}
```

4. The `SmartDevice` interface contains functionalities not applicable to all smart devices. Split it according to the Interface Segregation Principle.

```
interface SmartDevice {
    void print();
    void fax();
    void scan();
}

class SmartPrinter implements SmartDevice {
    public void print() {
        // Print logic
    }

    public void fax() {
        // Fax logic
    }

    public void scan() {
        // Scan logic
    }
}
```

5. The `UserManager` class directly depends on the `MySQLDatabase` class for data storage. Refactor it to adhere to the Dependency Inversion Principle.

```
class MySQLDatabase {
    public void store(Object data) {
        System.out.println("Storing data in MySQL database");
    }
}

class UserManager {
    private MySQLDatabase database;

    public UserManager() {
        this.database = new MySQLDatabase();
    }

    public void saveUser(Object user) {
        database.store(user);
    }
}
```

6. The `OrderProcessor` class handles both order processing and logging of order processing errors. Refactor this class to adhere to the Single Responsibility Principle.

```
public class OrderProcessor {
    public void processOrder(Order order) {
        try {
            // Process order logic
        } catch (Exception e) {
            logError(e);
        }
    }

    private void logError(Exception e) {
        // Log error to a file
    }
}
```

7. Refactor the following classes to ensure that substituting a base class object (`Payment`) with a derived class object (`CreditCardPayment` or `PayPalPayment`) does not break the functionality.

```
class Payment {
    void initiatePayments() {}
    boolean validatePayment() { return true; }
}

class CreditCardPayment extends Payment {
    @Override
    boolean validatePayment() { return true; }
}

class PayPalPayment extends Payment {
    @Override
    void initiatePayments() {}
}
```

8. The `MultifunctionPrinter` class implements the `Printer` interface, which has too many responsibilities. Apply the Interface Segregation Principle to refactor the interface.

```
interface Printer {
    void printDocument();
    void scanDocument();
    void faxDocument();
}

class MultifunctionPrinter implements Printer {
    public void printDocument() { /* Implementation */ }
    public void scanDocument() { /* Implementation */ }
    public void faxDocument() { /* Implementation */ }
}
```

9. The `UserSettings` class manages both user preferences and user authentication. Split the class to adhere to the Single Responsibility Principle.

```
public class UserSettings {
    public void changeSetting(String setting, String value) { /* Change settings logic */
    public boolean login(String username, String password) { /* Login logic */ }
}
```

10. The `ProductFilter` class needs enhancement to support filtering products by color and size simultaneously without modifying its existing code.

```
class ProductFilter {
    public Stream<Product> filterByColor(List<Product> products, Color color) {
        return products.stream().filter(p -> p.getColor() == color);
    }
    // Existing methods...
}
```

11. Ensure that the `ElectricCar` class can replace the Car class without altering the expected behavior, focusing on the refuel method.

```
class Car {
    void refuel() { /* Refueling logic */ }
}

class ElectricCar extends Car {
    @Override
    void refuel() {
        // Electric charging logic
    }
}
```

12. The `Worker` interface is used by both `Manager` and `Technician` classes but contains methods that are not applicable to both. Apply the Interface Segregation Principle.

```
interface Worker {
    void work();
    void manage();
}

class Manager implements Worker {
    public void work() { /* Manager-specific work */ }
    public void manage() { /* Management tasks */ }
}

class Technician implements Worker {
    public void work() { /* Technical tasks */ }
    public void manage() { /* Irrelevant for Technician */ }
}
```

2 GRASP

1. Refactor the code to adhere to the Information Expert principle for calculating the total price of items in an order.

```
class Order {
    Item[] items;

    public Order(Item[] items) {
        this.items = items;
    }
}

class Item {
    double price;

    public Item(double price) {
        this.price = price;
    }
}

class Calculator {
    double calculateTotalPrice(Order order) {
        double total = 0;
        for (int i = 0; i < order.items.length; i++) {
            total += order.items[i].price;
        }
        return total;
    }
}
```

2. Decide where to place the creation logic of `Task` instances in a project management system to comply with the Creator GRASP principle.

```
class Task {
    String description;

    public Task(String description) {
        this.description = description;
    }
}
```

```
class Project {
    List<Task> tasks = new ArrayList<>();

    void addTask(String description) {
        Task newTask = new Task(description);
        tasks.add(newTask);
    }
}

class User {
    // User details
}
```

3. Identify the part of the following code that violates the Controller GRASP principle for handling user registration and refactor it.

```
class UserView {
    void onRegisterButtonClicked() {
        System.out.println("Registering a user...");
        // Logic to register a user
    }
}

class UserController {
    // Controller methods
}
```

4. Refactor the following code to reduce coupling between the `OrderManager` and `PaymentGateway` classes.

```
class OrderManager {
    void processOrder() {
        PaymentGateway paymentGateway = new PaymentGateway();
        paymentGateway.makePayment(100); // Example amount
    }
}

class PaymentGateway {
    void makePayment(double amount) {
        System.out.println("Processing payment of: " + amount);
    }
```

```
}
```

5. Improve the design of the following class to enhance its cohesion by applying the High Cohesion principle.

```
class ActivityManager {
    void startActivity() {
        System.out.println("Activity started.");
    }

    void stopActivity() {
        System.out.println("Activity stopped.");
    }

    void logActivity() {
        System.out.println("Activity logged.");
    }

    void sendActivityNotifications() {
        System.out.println("Activity notification sent.");
    }
}
```

6. Use polymorphism to eliminate conditional logic based on account type for calculating interest.

```
class Account {
    String type;
    double balance;

    public Account(String type, double balance) {
        this.type = type;
        this.balance = balance;
    }

    double calculateInterest() {
        if (type.equals("Savings")) {
            return balance * 0.03;
        } else if (type.equals("Checking")) {
            return balance * 0.01;
```

```
        }
        return 0;
    }
}
```

7. Add logging functionality to the `ProductService` class without violating SRP using the Pure Fabrication principle.

```
class ProductService {
    void addProduct(Product product) {
        System.out.println("Product added: " + product.getName());
        // Add product logic
    }
}

class Product {
    private String name;

    public Product(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

8. Reduce the direct dependency between `CustomerManager` and `EmailClient` by applying indirection.

```
class CustomerManager {
    void sendEmailToCustomer() {
        EmailClient client = new EmailClient();
        client.sendEmail("Thank you for your purchase!");
    }
}

class EmailClient {
    void sendEmail(String message) {
        System.out.println("Email sent: " + message);
    }
```

```
}
```

9. Apply the Information Expert principle to assign the responsibility of calculating the total number of orders for a customer.

```
class Customer {
    Order[] orders;

    public Customer(Order[] orders) {
        this.orders = orders;
    }
}

class Order {
    // Order details
}

class OrderCounter {
    int countOrders(Customer customer) {
        int count = 0;
        for (int i = 0; i < customer.orders.length; i++) {
            count++;
        }
        return count;
    }
}
```

3 Other Principles

1. The following code contains repetitive logic for calculating discounts on different types of products. Refactor it to adhere to the DRY principle.

```
class DiscountCalculator {
    double calculateBookDiscount(double price) {
        return price * 0.9; // 10% discount
    }

    double calculateToyDiscount(double price) {
        return price * 0.9; // 10% discount
    }
```

```
    // Other product discounts...
}
```

2. The following code for a simple banking application repeats the logic for logging and applying transaction fees. Refactor to eliminate the repetition.

```
class BankAccount {
    double balance;

    void deposit(double amount) {
        System.out.println("Deposit: " + amount);
        balance += amount;
        balance -= 2; // Transaction fee
    }

    void withdraw(double amount) {
        System.out.println("Withdraw: " + amount);
        balance -= amount;
        balance -= 2; // Transaction fee
    }
}
```

3. Simplify the following code that uses unnecessary complex logic to check if a user is logged in.

```
class UserSession {
    Boolean isLoggedIn;

    boolean checkIfUserIsLoggedIn() {
        if (isLoggedIn == null) {
            return false;
        } else {
            if (isLoggedIn) {
                return true;
            } else {
                return false;
            }
        }
    }
}
```

4. The following method signature is unnecessarily complex. Simplify it while maintaining its functionality.

```
class ReportGenerator {
    void generateReport(String title, String data, boolean includeGraphics, boolean includ
        // Report generation logic
    }
}
```

5. The `UserManager` class handles both user authentication and user data management. Refactor for separation of concerns.

```
class UserManager {
    void loginUser(String username, String password) {
        // Login logic
    }

    void saveUser(String username) {
        // Save user data
    }
}
```

6. The `calculateTotal` method unexpectedly changes the state of the order. Refactor to avoid surprise.

```
class Order {
    double total;
    List<Item> items;

    double calculateTotal() {
        total = 0; // Side effect
        for (Item item : items) {
            total += item.price;
        }
        return total;
    }
}
```

7. The following code violates the Law of Demeter by making a chain of method calls. Refactor to adhere to the principle.

```
class ShoppingSession {
    Cart cart;

    void checkout() {
        double total = cart.getItems().getTotalPrice();
        // Checkout logic
    }
}
```

8. Refactor the following code to avoid deep navigation of objects, in compliance with the Law of Demeter.

```
class Employee {
    Manager manager;

    void sendReport() {
        manager.getDepartment().submitReport("Report");
    }
}
```