**Task 1:**

What do you think is the need for Refactoring?

1. Improve Code Quality: Makes code cleaner and easier to understand

2. Reduce Complexity: Simplifies complicated code

3. Fix Technical Debt: Addresses poor design choices made earlier

4. Enhance Maintainability: Makes future changes easier

5. Better Performance: Can improve code efficiency

6. Remove Code Smells: Fixes bad coding practices

7. Easier Bug Finding: Makes it simpler to spot and fix issues

**Task 2:**

What are the Principles of refactoring?

1. Keep It Simple: Don't make things more complex than necessary

2. Small Steps: Make gradual changes rather than big rewrites

3. Test Often: Ensure changes don't break existing functionality

4. Don't Add Features: Focus only on improving existing code

5. Maintain Behavior: Code should work the same way after refactoring

6. Code Review: Have others check your refactored code

7. Clean Code: Follow coding standards and best practices

**Task 3:**

What are the steps for performing code refactoring?

1. Identify Problems

- Look for areas that need improvement

- Find complicated or messy code

2. Create Tests

- Write tests before making changes

- Ensure existing functionality works

3. Make Small Changes

- Refactor one piece at a time

- Don't try to fix everything at once

4. Test Again

- Run tests after each change

- Make sure nothing broke

5. Review Changes

- Check if code is better now

- Ensure it's easier to understand

6. Document

- Note important changes made

- Update any documentation

7. Commit Changes

- Save your work

- Keep track of what was changed

**Task 4:**

What makes Composite pattern useful when designing complex tree structures?

1. It replaces the use of collections to store children

2. allows treating individual objects and compositions uniformly through a common interface.

3. It automatically serializes tree objects for persistence

4. optimizes memory by removing duplicate nodes in the tree

**Task 5:**

Identify the code smell:

public class Order {

private String orderid;

private String customerName;

private String customerAddress;

private String customerPhone;

public String getOrderld() {

return orderid,

}

public void setOrderld(String orderid) {

this.orderid orderid,

}

public String getCustomerName() {

return customerName;

}

public void setCustomerName(String customerName) {

this.customerName = customerName;

}

public String getCustomerAddress() {

return customerAddress;

}

public void setCustomerAddress(String customerAddress) {

this.customerAddress = customerAddress;

}

public String getCustomerPhone() {

return customerPhone;

}

public void setCustomerPhone(String customerPhone) {

this.customerPhone = customerPhone;

}

}

1. Long Method
2. Primitive Obsession

1. Large Class

1. Feature Envy

**Task 6:**

In the context of the Three-tier architecture, what role does the 'Business Logic Layer play?

1. It is responsible for managing physical data storage and retrieval mechanisms from database systems.

2. It processes commands from the user interface, performs validations, and implements the core functional Logic.

3. It defines how the system behaves under network traffic and handles load balancing

4. it renders the UI elements and sends them directly to database procedures for execution

**Task 7:**

What is the role of Packages in representing subsystems?

1. Packages are used only to store deprecated classes for backward compatibility

2. Packages group related elements and can be used to modularize large systems into manageable subsystems with defined interfaces

3. Packages represent reusable libraries only and are not part of design architecture

4. Packages define the runtime performance model of subsystems

**Task 8:**

You are building a system that maintains a cache of user sessions. The session data must be accessed globally and initialized once, lazily. Which implementation is the most thread-safe and efficient?

public class SCache {

private static volatile SCache instance;

private SCache() {}

public static SCache getinstance() {

if (instance == null) {

synchronized (SCache.class) {

if (instance == null) {

instance = new SCache();

}

}

}

return instance;

}

}

1. Implements Command pattern for caching logic

2. Uses double checked locking Singleton, ensures lazy and thread-safe initialization

3. Applies Factory pattern with static holder

4. Uses Prototype pattern with unnecessary locking

**Task 9**

Identify the code smell :

public class Customer {

private String name;

private String address;

private String phoneNumber;

public void printCustomer Details() {

System.out.println("Name: " + name);

System.out.println("Address: " + address);

System.out.println("Phone Number: " + phoneNumber);

}

}

1. Long Method

1. Primitive Obsession

1. Large Class

1. Feature Envy

**Task 10:**

Consider the following set of interfaces and classes for a payment system. What principle is violated and how would you improve it?

interface PaymentService{

void makePayment();

void cancelPayment();

void generatelnvoice();

}

class CreditCardPayment implements PaymentService {

@Override

public void makePayment() {

Implementation for making credit card payment

}

@Override

public void cancelPayment() {

//Implementation for canceling credit card payment

}

@Override

public void generatelnvoice() {

// Not applicable for credit card

}

}

1. Liskov Substitution Principle is violated due to missing default behavior

2. Dependency Inversion is violated, introduce abstraction for the payment handler

3. Open Closed Principle is violated by not supporting extension for other payment types

4. Interface Segregation Principle is violated spit the interface into more specific ones for better adherence to roles.

**Task 11:**

Consider the following class hierarchy. What major design issue exists and how would you refactor it?

class Notification {

public void send(String message) {

System.out.println("Sending generic notification: message);

}

}

class EmailNotification extends Notification }

@Override

public void send(String message) }

System.out.println("Sending email:+message);

}

}

class SMSNotification extends Notification {

@Override

public void send(String message) {

throw new Unsupported OperationException("SMS not supported");

}

}

1. Violates Interface Segregation, merge all notifications into one abstract class

2. Violates Liskov Substitution Principle: use interfaces and split behaviors per notification type

3. No issue, the design is extensible and allows overriding

4. Follows Open-Closed Principle; hence no refactoring is needed