**Task - 1:**

**What do you think is the need for Refactoring?**

The need for refactoring arises from the desire to maintain clean, efficient, and maintainable code over time. Software development is iterative, and as code evolves, it can become messy, duplicated, or overly complex. Refactoring addresses these issues. Here’s a detailed explanation:

1. Improve Code Readability

* Code must be understandable not only to the original developer but also to others who maintain it later.
* Clean and readable code reduces misunderstandings, errors, and onboarding time for new developers.  
   *Example:* Renaming variables or extracting methods with meaningful names makes logic self-explanatory.

2. Reduce Complexity

* Over time, code may accumulate nested conditions, duplicated logic, or long methods.
* Refactoring simplifies complex structures, making them easier to understand and maintain.  
   *Example:* Replacing nested conditionals with polymorphism or guard clauses.

3. Eliminate Code Duplication

* Duplicate code increases maintenance overhead—any change must be applied in multiple places.
* Refactoring removes duplication through methods, classes, or inheritance strategies.

4. Facilitate Maintenance and Extension

* Well-structured code allows developers to add features, fix bugs, or improve performance without breaking existing functionality.
* Refactoring helps create modular, loosely coupled code, which is easier to extend.

5. Improve Performance and Resource Management

* Refactoring can optimize algorithms, data structures, or memory usage without changing the software’s behavior.
* This is often a side benefit of cleaning up inefficient or redundant code.

**Task -2**

**What are the Principles of refactoring?**

**Key principles:**

1. Preserve Behavior

* The program should work exactly the same after refactoring.
* Only the internal structure changes; functionality stays the same.

2. Improve Readability

* Code should be easier to read and understand.
* Use meaningful names, clean formatting, and simple logic.

3. Reduce Complexity

* Simplify complicated code, loops, or conditions.
* Break long methods into smaller, focused methods.

4. Remove Duplication

* Avoid repeating code in multiple places.
* Reuse methods, classes, or objects where possible.

5. Keep Code Modular

* Organize code into small, independent, and reusable pieces (methods, classes, modules).

6. Continuous Improvement

* Refactoring is done regularly, not just once.
* Small, frequent improvements prevent the code from becoming messy.

**Task -3:**

**What are the steps for performing code refactoring?**

### **Steps for Performing Code Refactoring:**

Step 1. Identify the Code to Refactor

* Look for code smells: long methods, duplicated code, complex conditionals, large classes, or poor naming.
* Focus on parts of the code that are hard to read, maintain, or extend.

Step 2. Understand the Code

* Make sure you understand what the code does before changing it.
* Run the program, read the logic, and check dependencies.

Step 3. Write or Run Tests

* Ensure there are unit tests covering the functionality.
* This helps confirm that the program works the same after refactoring.

Step 4. Plan the Refactoring

* Decide which refactoring techniques to apply:  
  + Extract Method
  + Rename Variable
  + Replace Conditional with Polymorphism
  + Remove Duplicated Code
* Plan changes in small steps to avoid introducing errors.

Step 5. Apply Refactoring in Small Steps

* Make one change at a time.
* Refactor a method, class, or module incrementally.
* After each change, run tests to ensure behavior is unchanged.

Step 6. Test After Each Change

* Run all unit and integration tests after each refactoring step.
* Ensure nothing breaks and the program behaves the same as before.

Step 7. Review and Improve

* Check the refactored code for:  
  + Readability
  + Maintainability
  + Performance
* Make additional small improvements if needed.

Step 8. Repeat Regularly

* Refactoring is an ongoing process, not a one-time task.
* Continuously improve code as new features are added.

**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
3. Large Class
4. 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. Usses 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
2. Primitive Obsession
3. Large Class
4. 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

**Task 12**

What is a key benefit of using the Facade design pattern in application architecture?

1. It provides a way to eliminate middle layers and reduce abstraction in software components.

2. It allows access to the low level subsystems directly for debugging and testing

3. It offers a mechanism for injecting multiple implementations into a core algorithm dynamically

4. It simplifies access to a complex system by providing a unified interface over a set of interfaces in a subsystem

**Task 13:**

How does the Proxy Design Pattern support performance or access control?

1. It executes logic inside core components without any delegation.

2. It logs method calls without executing them.

3. It provides a placeholder to control access to another object, often adding lazy loading, access control, or caching.

4. It permanently replaces the original object with a faster mock implementation

**Task 14**:

Which of the following best represents the "Open/Closed Principle from the SOLID principles?

1. Software components should be designed to be open for direct modification but closed to extension for maintaining rigidity

2. Entities should be open for extension through mechanisms like inheritance or composition, but closed for modification to avoid breaking existing behavior.

3. Code should be able to accept runtime parameter changes without altering any class behavior or interface

4. Code must be completely static to avoid any modification or future maintenance overhead

**Task 15:**

What distinguishes the Builder pattern from the Prototype pattern in object creation?

1. The Builder pattern focuses on shallow copying of objects while Prototype deals with constructing complex objects step by step

2. The Builder pattern separates the construction of a complex object from its representation, while Prototype allows creation of duplicate objects by copying an existing one.

3. The Builder pattern helps clone objects quickly whereas Prototype builds objects using various helper methods

4. The Builder and Prototype serve similar purposes but Builder is used at compile time and Prototype at runtime

**Task 16:**

You've joined a legacy insurance product where changes in one module often result in failures in unrelated modules. There's a lack of clear ownership and multiple responsibilities per class. You're tasked with improving stability and maintainability without breaking functionality. What is the first approach you should take?

1. Merge related classes into one for tighter control

2. Rewrite all modules from scratch using latest Java frameworks

3. Refactor classes to follow the Single Responsibility Principle and identify code smells

4. Move business logic to the frontend to reduce complexity in backend

**Task 17:**

Analyze the code below. What anti-pattern or refactoring opportunity is present here?

class UserManager {

public void processUser(String username) {

if (username.equals("admin")) {

// Admin-specific logic

}else if (username.equals("guest")) {

// Guest-specific logic

} else {

// Default logic

}

}

1. The method violates the Open Closed Principle, consider using polymorphism instead of hard-coded conditions.

2. No refactoring is required since all roles are covered

3. The method property uses polymorphism by branching based on user roles

4. The logic should be moved to the database to improve separation of concerns

**Task 18:**

You're designing a microservice-based inventory system where changes in product details should notify multiple services like pricing, recommendation, and search. These dependent services should act independently and not affect the source servicer's behavior. How should you model this behavior?

1. Use a centralized database to keep all services in sync

2. implement direct service-to-service RPC calls on update

3. Use asynchronous messaging with Publish Subscribe to notify downstream services

4. Add retry logic in all dependent services for error recovery

**Task 19:**

A logistics company's platform must scale to millions of requests per day. The design should separate data handling, business logic, and presentation, allowing independent scaling of layers. Which architectural model should be applied?

1. Use Decorator to wrap all business logic for better scaling

2. Use a 3-tier Architecture to decouple UI, Business, and Data layers

3. Implement Singleton in each layer to reduce memory usage

4. Implement Proxy classes to replace all direct DB interactions

**Task 20:**

What characteristic of a well-written unit test makes it valuable in Test Driven Development?

1. It should test only one method but involve multiple objects and rely on external systems.

2. It must execute complex test scenarios using mock networks and full integrations

3. It should be independent of the code and unrelated to the software behavior

4. It should be repeatable, focused on a single responsibility and clearly define expected outcomes for each condition

**Task 21:**

A project has high unit test coverage but frequent production bugs. On investigation, the tests mostly validate getters, setters, and trivial logic. How can the test suite be improved to catch real-world issues?

1. Add more assertions to the existing tests without changing test focus

2. Refactor tests to coverage cases, boundary conditions, and business logic paths

3. Migrate unit tests to performance tests

4. Replace unit tests with mocks to simulate data better

**Task 22:**

A team is building a financial analytics platform where data needs to be fetched from multiple sources like APIs, files, and databases. These sources require different logic but return results in a similar format. The lead architect wants to design it in a way that supports adding new data sources in the future without modifying the core system. What pattern is most appropriate?

1. Use Singleton to manage shared resource access to these sources

2. Use Strategy Pattern to encapsulate source specific logic and switch at runtime

3. Use Prototype to clone existing logic for each data source

4. Use Decorator Patten to layer additional features on top of each data source

**Task 23:**

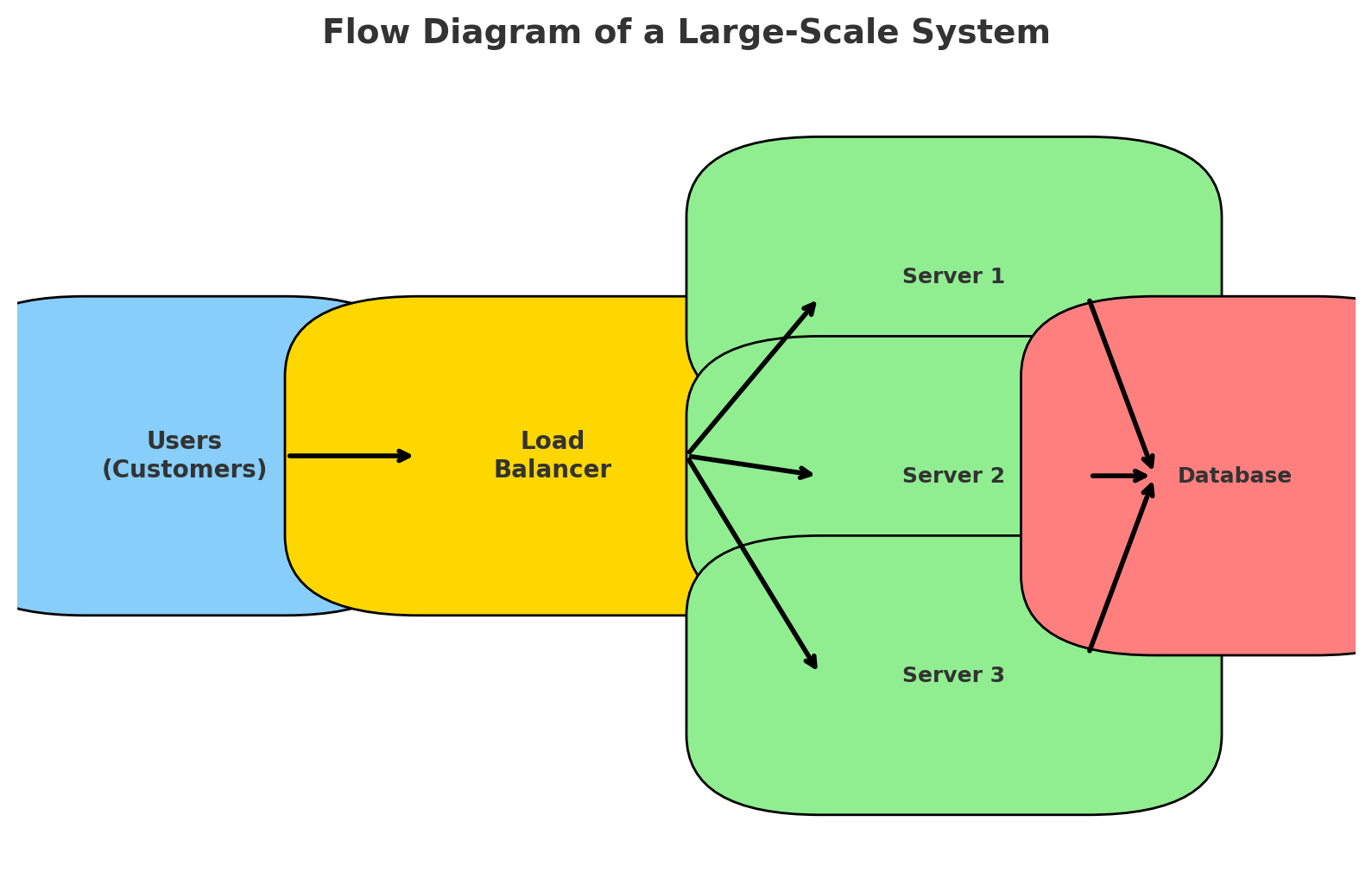
While working on a distributed messaging system, a team is facing challenges with tightly coupled modules. The event producers and consumers are directly referencing each other, causing deploy-time dependencies. What design adjustment would decouple them efficiently?

1. Introduce direct REST calls instead of asynchronous messaging

2. Use the Publish Subscribe Pattern to decouple producers from consumers

3. Add shared database access between both modules

4. Use Adapter Pattern to hide implementation details

****