**Task 1: What do you think is the need for Refactoring?**

Refactoring is needed to:

- Improve code readability and maintainability

- Reduce technical debt

- Eliminate code duplication

- Improve performance

- Make code easier to understand and modify

- Fix poor design choices without changing functionality

- Prepare code for new features

- Reduce bugs and improve code quality

**Task 2: What are the Principles of refactoring?**

Key principles include:

1. Don't break functionality - Tests should pass before and after

2. Small, incremental changes - Make one change at a time

3. Test continuously - Run tests after each change

4. Refactor before adding features - Clean code first

5. Follow the Boy Scout Rule - Leave code better than you found it

6. Don't refactor and add functionality simultaneously

7. Keep refactoring separate from feature commits

**Task 3: What are the steps for performing code refactoring?**

1. Identify code that needs refactoring (code smells)

2. Write/ensure tests exist for the code

3. Plan the refactoring approach

4. Make small, incremental changes

5. Run tests after each change

6. Review the refactored code

7. Document significant changes

8. Commit the refactoring separately

**Task 4: What makes Composite pattern useful when designing complex tree structures?**

Allows treating individual objects and compositions uniformly through a common interface.

The Composite pattern's key benefit is providing a uniform interface to work with both individual objects (leaves) and compositions of objects (branches), making tree traversal and operations consistent.

**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;

}

}

Feature Envy

The Order class has multiple customer-related fields (customerName, customerAddress, customerPhone) that should belong to a separate Customer class. This is Feature Envy - when a class is more interested in another class's data than its own.

**Task 6: In the context of the Three-tier architecture, what role does the 'Business Logic Layer play?**

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

The Business Logic Layer (middle tier) contains the application's business rules, validations, calculations, and core processing logic.

**Task 7: What is the role of Packages in representing subsystems?**

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

Packages help organize code into logical subsystems, providing modularity and clear boundaries between different parts of the system.

**Task 8: Thread-safe and efficient implementation**

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;

}

}

Uses double checked locking Singleton, ensures lazy and thread-safe initialization\*\*

The code shows the double-checked locking pattern with volatile keyword, which is a thread-safe way to implement lazy initialization of a singleton.

**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);

}

The code doesn't exhibit any of the listed smells:

- Not a Long Method (it's quite short)

- Not Primitive Obsession (uses appropriate String types)

- Not a Large Class (only 3 fields and 1 method)

- Not Feature Envy (works with its own data)

The method name has a space issue (printCustomer Details), but that's a syntax error, not a code smell.

**Task 10: What principle is violated and how would you improve it?**

Interface Segregation Principle is violated split the interface into more specific ones for better adherence to roles.\*\*

The PaymentService interface forces CreditCardPayment to implement generateInvoice() which it doesn't need. ISP states that clients shouldn't be forced to depend on interfaces they don't use.

**Task 11: 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");

}

}

Violates Liskov Substitution Principle: use interfaces and split behaviors per notification type\*\*

SMSNotification throws an exception in its send() method, violating LSP which states that subtypes should be substitutable for their base types without breaking the program's correctness.

**Task 12: Facade Design Pattern**

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

The Facade pattern provides a simplified interface to a complex subsystem, hiding the complexities and making the system easier to use.

**Task 13: Proxy Design Pattern**

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

The Proxy pattern acts as an intermediary that controls access to the real object, adding functionality like caching, access control, or lazy initialization.

**Task 14: Open/Closed Principle**

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

This SOLID principle ensures code can be extended without modifying existing, tested code.

**Task 15: Builder vs Prototype Pattern**

- 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

Builder constructs objects step-by-step, while Prototype clones existing objects.

**Task 16: Legacy System Improvement**

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

Starting with SRP helps create clearer boundaries and reduces coupling between modules.

**Task 17: Code Anti-pattern**

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

The if-else chain for user types should be replaced with polymorphism (e.g., different user classes).

**Task 18: Microservice Notification System**

- Use asynchronous messaging with Publish Subscribe to notify downstream services

Pub/Sub pattern ensures loose coupling and allows services to operate independently.

**Task 19: Scalable Architecture**

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

3-tier architecture allows independent scaling of presentation, business logic, and data layers.

**Task 20: Unit Test Characteristics**

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

Good unit tests are isolated, repeatable, and test one specific behavior.

**Task 21: Improving Test Coverage**

- Refactor tests to cover edge cases, boundary conditions, and business logic paths

Testing actual business logic and edge cases is more valuable than testing trivial code.

**Task 22: Financial Analytics Platform**

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

Strategy pattern allows easy addition of new data sources without modifying core logic.

**Task 23: Distributed Messaging Decoupling**

- Use the Publish Subscribe Pattern to decouple producers from consumers

Pub/Sub removes direct dependencies between producers and consumers in distributed systems.

**Task 24 👍**

**Debate on**

**Large Scale Systems**

**Load Balancers**

**Async Programming**

**Home Tasks:**

Strategy Pattern Implementation for Sorting

1. Strategy Interface

interface SortingStrategy {

void sort(List<String> items);

}

Strategy Interface: Defines the contract for all sorting strategies

2. Concrete Strategy Classes

// Concrete strategy for alphabetical sorting (case insensitive)

class AlphabeticalSortingStrategy implements SortingStrategy {

@Override

public void sort(List<String> items) {

Collections.sort(items, String.CASE\_INSENSITIVE\_ORDER);

}

}

// Concrete strategy for length-wise sorting

class LengthwiseSortingStrategy implements SortingStrategy {

@Override

public void sort(List<String> items) {

Collections.sort(items, new Comparator<String>() {

@Override

public int compare(String s1, String s2) {

// First compare by length

int lengthCompare = Integer.compare(s1.length(), s2.length());

// If lengths are equal, compare alphabetically

return lengthCompare != 0 ? lengthCompare : s1.compareTo(s2);

}

});

}

}

Concrete Strategies:

- AlphabeticalSortingStrategy: Sorts alphabetically (case-insensitive)

- LengthwiseSortingStrategy: Sorts by string length

3. Context Class

import java.util.\*;

class SortingContext {

private List<String> items;

private SortingStrategy strategy;

public SortingContext() {

this.items = new ArrayList<>();

}

public void setStrategyForSorting(SortingStrategy strategy) {

this.strategy = strategy;

}

public void addItem(String item) {

items.add(item);

}

public void addItems(List<String> newItems) {

items.addAll(newItems);

}

public void removeItem(String item) {

items.remove(item);

}

public void performSort() {

if (strategy == null) {

throw new IllegalStateException("Sorting strategy not set");

}

strategy.sort(items);

}

public List<String> getList() {

return new ArrayList<>(items); // Return a copy to maintain encapsulation

}

public void clearItems() {

items.clear();

}

}

Context Class:

- Maintains the list of items

- Allows dynamic strategy switching

- Provides methods to add/remove items

- Performs sorting based on selected strategy

4. Main Class with Test Implementation

public class SortingStrategyDemo {

public static void main(String[] args) {

// Create context

SortingContext context = new SortingContext();

// Add items

context.addItem("Stanford");

context.addItem("Ankit");

context.addItem("Watson");

context.addItem("Done");

// Test alphabetical sorting

System.out.println("Original list: " + context.getList());

System.out.println("\nAlpha sorting:");

context.setStrategyForSorting(new AlphabeticalSortingStrategy());

context.performSort();

List<String> alphaSorted = context.getList();

for (String item : alphaSorted) {

System.out.println(item);

}

// Reset and test length-wise sorting

context.clearItems();

context.addItem("Stanford");

context.addItem("Ankit");

context.addItem("Watson");

context.addItem("Done");

System.out.println("\nLengthwise sorting:");

context.setStrategyForSorting(new LengthwiseSortingStrategy());

context.performSort();

List<String> lengthSorted = context.getList();

for (String item : lengthSorted) {

System.out.println(item);

}

}

}

Output:

Original list: [Stanford, Ankit, Watson, Done]

Alpha sorting:

Ankit

Done

Stanford

Watson

Lengthwise sorting:

Done

Ankit

Watson

Stanford

Encapsulation: The context returns a copy of the list to maintain data integrity

Flexibility: New sorting strategies can be added without modifying existing code