Day 28

Employee ID: 201933938

Login ID: iamasif

Name: Shaik Asif

Clas name DManager

Add , remove , retrieve.. Methods (list)

→ Declare private static variable ..- type Dmanager - make sure it holds singleton instance

→ create private constructor to prevent direct instantiation of class

→ create instance..

→ throw an illegalStaticException , create singleton instance if not created .. use getInstance()

→ public static synchronized method – getInstance()

→ return existing instance..

→ create a new instance if doesnot exist..

In the thread safe list management:

→ declare a private variable

→ initialize the list inside the constructor using new..

→ also implement public sync methods

→ addItem

→ removeitem

→ list

10.18 to 10.23 to 10.25

Input 👍

The user

Toys

Groceries

Fruits

Done

Groceries

Output:

Toys

Fruits

**PROGRAM:**

**DataManager**

package org.example;

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class DataManager {

private static DataManager *instance*;

private List<String> items;

private DataManager() {

items = Collections.*synchronizedList*(new ArrayList<>());

}

public static synchronized DataManager getInstance() {

if (*instance* == null) {

*instance* = new DataManager();

}

return *instance*;

}

public synchronized void addItem(String item) {

items.add(item);

}

public synchronized void removeItem(String item) {

items.remove(item);

}

public synchronized List<String> listItems() {

return new ArrayList<>(items); // return a copy

}

}

**DataManagerMain**

**package org.example;**

**import java.util.Scanner;**

**public class DataManagerMain {**

**public static void main(String[] args) {**

**DataManager manager = DataManager.*getInstance*();**

**Scanner scanner = new Scanner(System.*in*);**

**System.*out*.println("Enter items (type 'Done' to stop):");**

**// Take input until "Done"**

**while (true) {**

**String input = scanner.nextLine();**

**if (input.equalsIgnoreCase("Done")) {**

**break;**

**}**

**manager.addItem(input);**

**}**

**// Show list before removal**

**System.*out*.println("\nItems before removal:");**

**for (String item : manager.listItems()) {**

**System.*out*.println(item);**

**}**

**// Ask which item to remove**

**System.*out*.println("\nEnter item to remove:");**

**String toRemove = scanner.nextLine();**

**manager.removeItem(toRemove);**

**// Show final list**

**System.*out*.println("\nFinal Items:");**

**for (String item : manager.listItems()) {**

**System.*out*.println(item);**

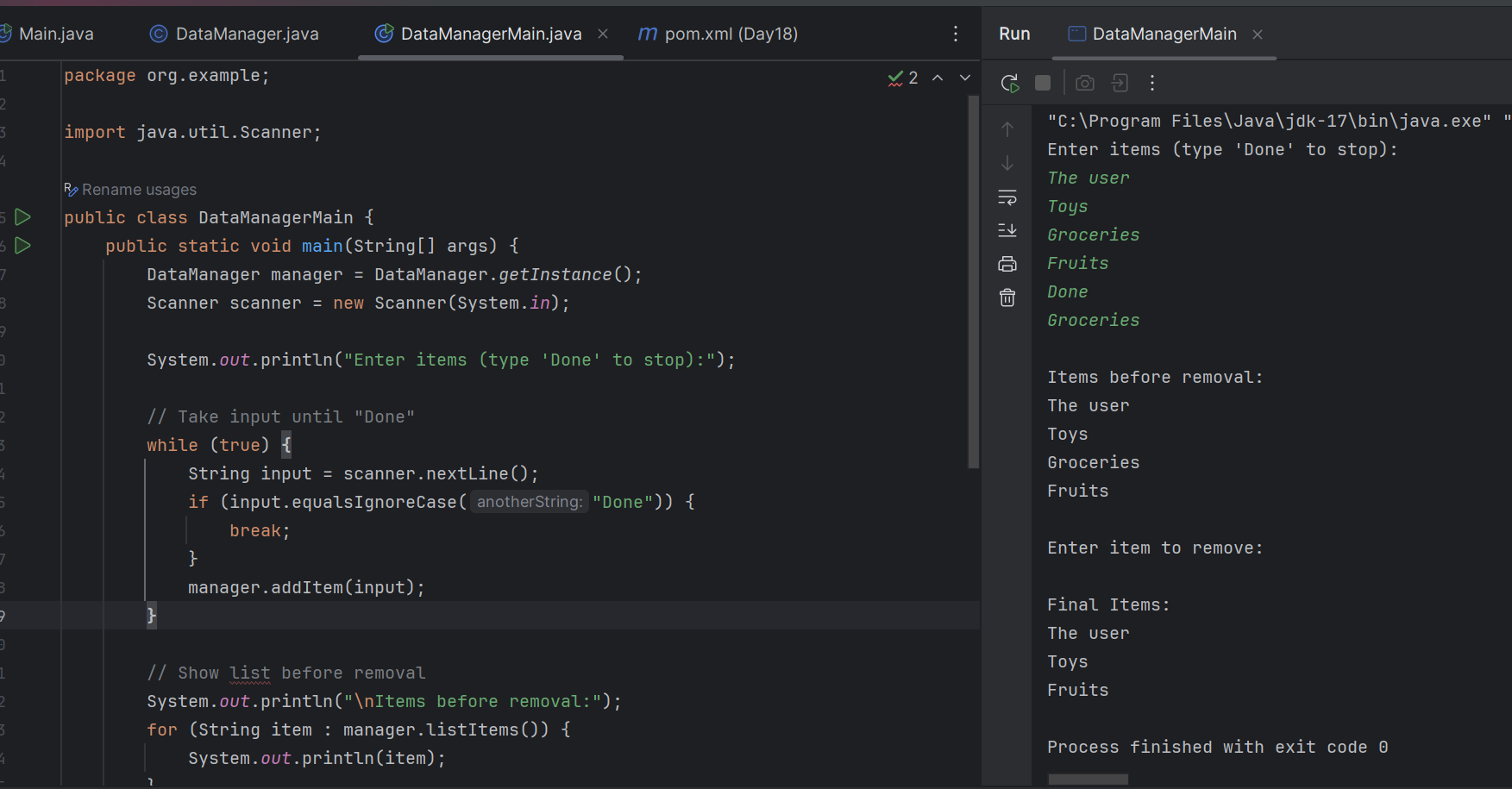
**}**

**scanner.close();**

**}**

**}**

**OUTPUT:**

****

Task 2:

Give output:

Class A {

psvm() {

Int a = 5;

Int b = 10;

Int c = 15:

sout((a > b ) && (b < c))

}

}

1. **Compilation error**
2. True
3. False
4. Runtime error

Task 3:

Finding Inheritance during requirement analysis is it important in OOAD .. why so?

1. It removes the need for encapsulation in the system design
2. **It helps identify objects with the shared behavior to promote code reuse and logical hierarchy**
3. It forces a flat class design improving performance by reducing polymorphic calls
4. Ensures all classes are instantiated using interfaces.

Task 4:

Which characteristics best defines polymorphism in OOP?

1. Ensures each class has its own copy of data members
2. It restricts method access to specific roles within a system
3. **It allows a single function or operator to behave differently based on its parameters or calling object**
4. It serialized different objects into a common file format for persistence

Task 5:

Which design pattern is being used in the following code snippet?

public interface ABC (

}

int doOperation(int num1, int num2);

public class OperationAdd implements ABC {

@Override

public int doOperation(int num1, int num2) (

return num1 + num2;

<

}

}

public class OperationSubtract implements ABC (

@Override

public int doOperation(int num1, int num2) { return num1- num2;

e:37.46 Mbps

}

}

public class Context (

private ABC abc;

public Context(ABC abc) {

this.abc = abc;

46 Mbps

}

public int executeABC(int num1, int num2) {

return abc.doOperation(num1, num2):

………

// more code ..

**The code is implementing the Strategy Pattern.**

Task 6:

Which of the following best explains the concept of data hiding in Object-Oriented Programming?

1. Data hiding means removing data from memory when no longer in use to ensure memory efficiency.
2. Data hiding involves using access specifiers to restrict direct access to class members, enabling controlled interaction through methods.
3. Data hiding refers to storing object data in secure databases during runtime.
4. Data hiding is achieved by deleting unused attributes from objects after object creation.

Task 7 : In OOAD, what is the primary value of Requirements Analysis?

1. It helps define class inheritance structure before testing

**2. It identifies system behavior and user needs to model objects and interactions meaningfully**

3. It configures application deployment scripts for testing

4. It automatically generates interface documentation from class files

Task 8:

Which design pattern is implemented in the following code snippet?

public class ClassName {

private static ClassName instance;

private ClassName() {

public static ClassName getinstance() ( if (instance = null) {

instance = new ClassName();

}

return instance;

}

}

1. Factory Method

**2. Singleton**

3. Prototype

4. Builder

Task 9:

Why is Interface preferred in Java when applying polymorphism over using abstract classes in many designs?

1. Interfaces enforce tight coupling between child and parent classes

2. Interfaces offer default constructors and static fields, which abstract classes cannot

**3. Interfaces allow a class to inherit from multiple sources of behavior, promoting decoupling and flexibility**

4. Interfaces provide direct access to private implementation logic

Task 10:

What is the role of the "Inception Phase in the Rational Unified Process?

1. It is the final phase where deployment and user training occur.

2. It defines the runtime environment for executing object oriented code

**3. It helps establish the business case, scope and feasibility of the proposed systems**

4. It focuses exclusively on UI design and database integration

Task 11:

What aspect of UML Diagrams makes them crucial in Object-Oriented Analysis and Design?

1. They provide detailed flowcharts for programming logic.

2. They represent runtime logs for system monitoring purposes.

3**. They visually capture the structure and behavior of systems through elements like classes, objects, and interactions.**

4. They replace testing frameworks by automatically generating code

Task 12:

Why is refactoring considered a continuous part of modern software development?

1. Refactoring is performed only at the end of a release cycle for documentation purposes

2. It replaces traditional debugging with automatic patching mechanisms

3. **Continuous refactoring ensures that the design evolves with changing requirements, reducing technical debt and improving code health**

4. Refactoring removes dependencies to minimize source control conflicts

Task 13:

OOAD, why is the Elaboration Phase important?

1. focuses on preparing production deployment pipelines

2**. The Elaboration Phase is where the major architectural decisions are validated through executable prototypes and risk mitigation**

3. is mainly used to finalize Uf designs and wireframes

4. is dedicated to refactoring legacy code to newer patterns

Task 14:

How are Active Objects represented in object modeling

1. As static utility classes for database access

2. **As objects that encapsulates encapsulate their own threat of control and asynchronously handle requests**

3. As serialized containers passed between processes

4. As Java Beans used solely for UI binding

11.20 to 11.35 Coffee Break

Task 15:

What makes Composite pattern useful when designing complex tree structures?

1. It replaces the use of collections to store children

**2. It 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 16:

Class SortingStrategy

– 2 concrete Strategies

* Sorting alphabetically (case insensitive)
* Sorting lengthwise

Interfaces - for both sorting

Define the strategy methods

Context is – SortingStretagy - class name

Methods to add, remove sort items based on the dynamically chosen strategy

* SetstretegyforSorting
* addItems
* PerformancSort
* getList

Input:

Stanford

Ankit

Watson

Done

Output:

Alpha sorting

Ankit

Stanford

Watson

**PROGRAM:**

package Sorting;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

// Step 1: Strategy Interface

interface SortingStrategy {

void sort(List<String> items);

}

// Step 2: Concrete Strategy - Alphabetical

class AlphabeticalSorting implements SortingStrategy {

@Override

public void sort(List<String> items) {

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

}

}

// Step 3: Concrete Strategy - Lengthwise

class LengthwiseSorting implements SortingStrategy {

@Override

public void sort(List<String> items) {

Collections.*sort*(items, Comparator.*comparingInt*(String::length));

}

}

// Step 4: Context Class

public class SortingContext {

private SortingStrategy strategy; // current strategy

private List<String> items; // list to sort

// Constructor

public SortingContext() {

this.items = new ArrayList<>();

}

// Set sorting strategy dynamically

public void setStrategy(SortingStrategy strategy) {

this.strategy = strategy;

}

// Add item

public void addItem(String item) {

items.add(item);

}

// Remove item

public void removeItem(String item) {

items.remove(item);

}

// Perform sorting using current strategy

public void performSort() {

if (strategy == null) {

System.*out*.println("No sorting strategy set!");

return;

}

strategy.sort(items);

}

// Get sorted list

public List<String> getList() {

return items;

}

// Main method for demo

public static void main(String[] args) {

SortingContext context = new SortingContext();

// Input

context.addItem("Shaik.Asif");

context.addItem("Teja");

context.addItem("B.Uday");

// Alpha sorting

context.setStrategy(new AlphabeticalSorting());

context.performSort();

System.*out*.println("Alpha sorting:");

for (String s : context.getList()) {

System.*out*.println(s);

}

System.*out*.println();

// Length sorting

context.setStrategy(new LengthwiseSorting());

context.performSort();

System.*out*.println("Lengthwise sorting:");

for (String s : context.getList()) {

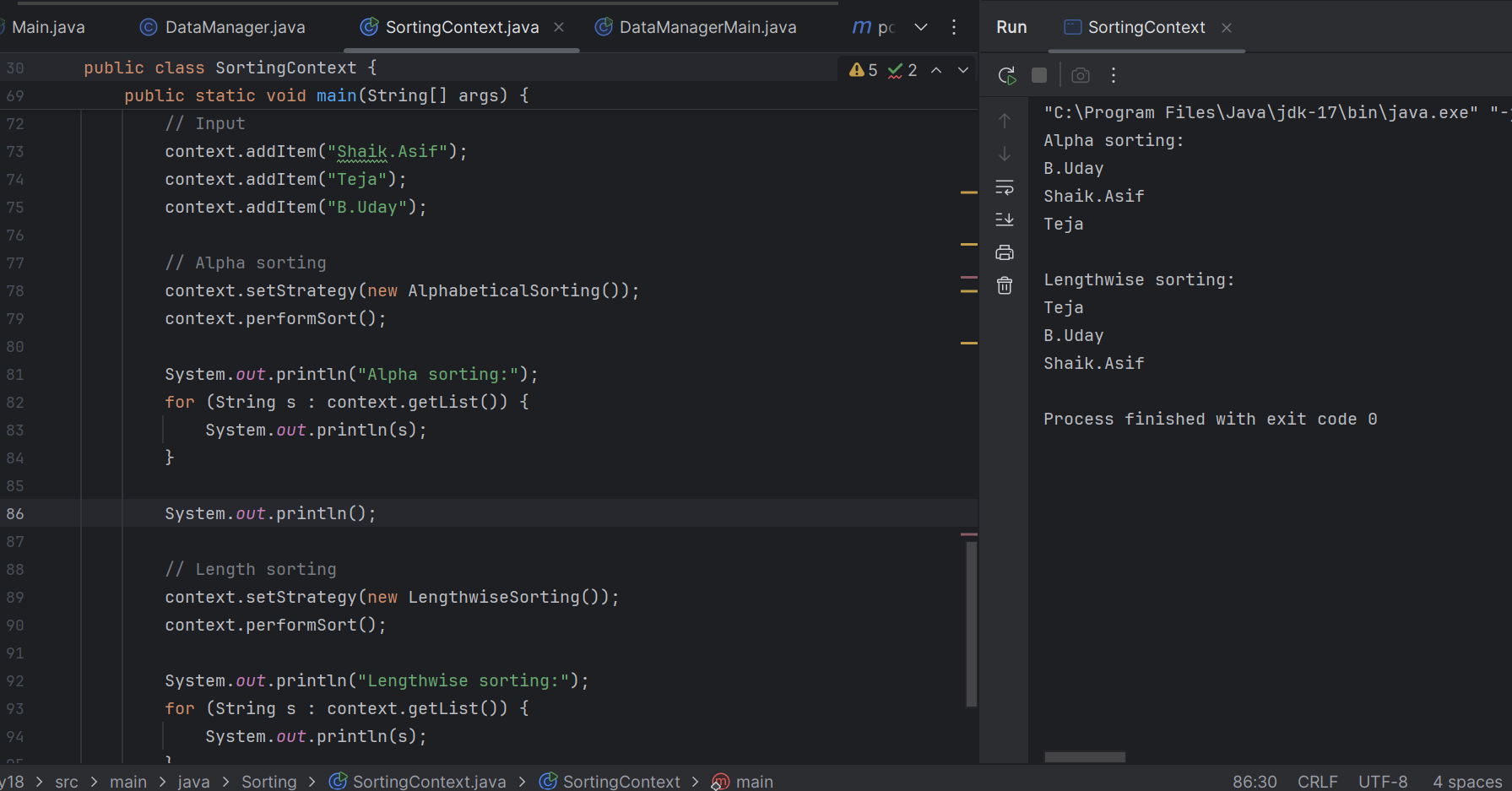
System.*out*.println(s);

}

}

}

**OUTPUT:**

****

Task 17:

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 18:

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 tubsystems 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 19:

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 20:

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 21:

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 22:

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 23:

5 Question Type, Multiple Choice Question (Single Correct Answer)

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 24:

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 25:

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 26:

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 27:

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 28:

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 29:

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 30:

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 31:

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 32:

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 33:

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 34:

**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**

Tsk 35:

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 Patten to hide implementation details