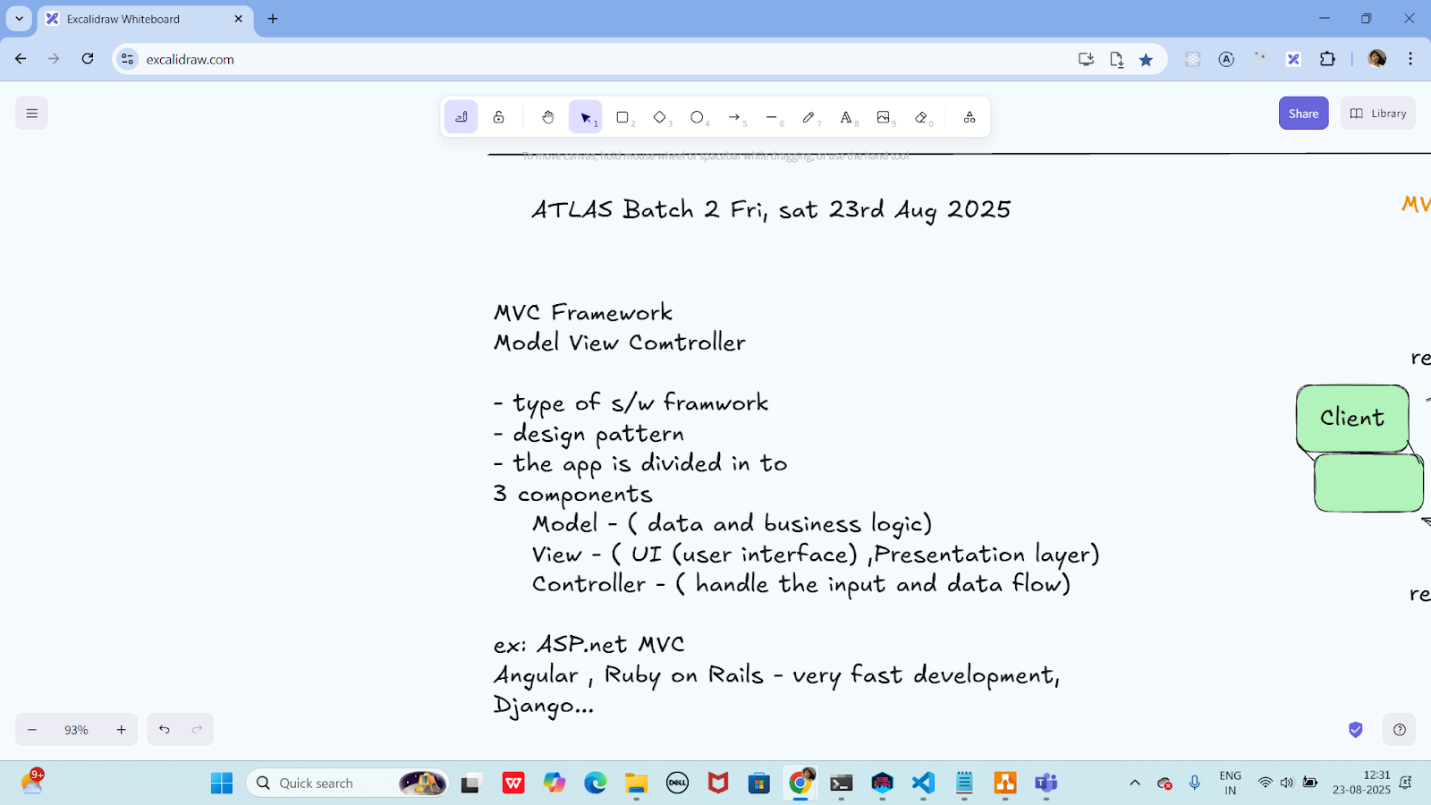
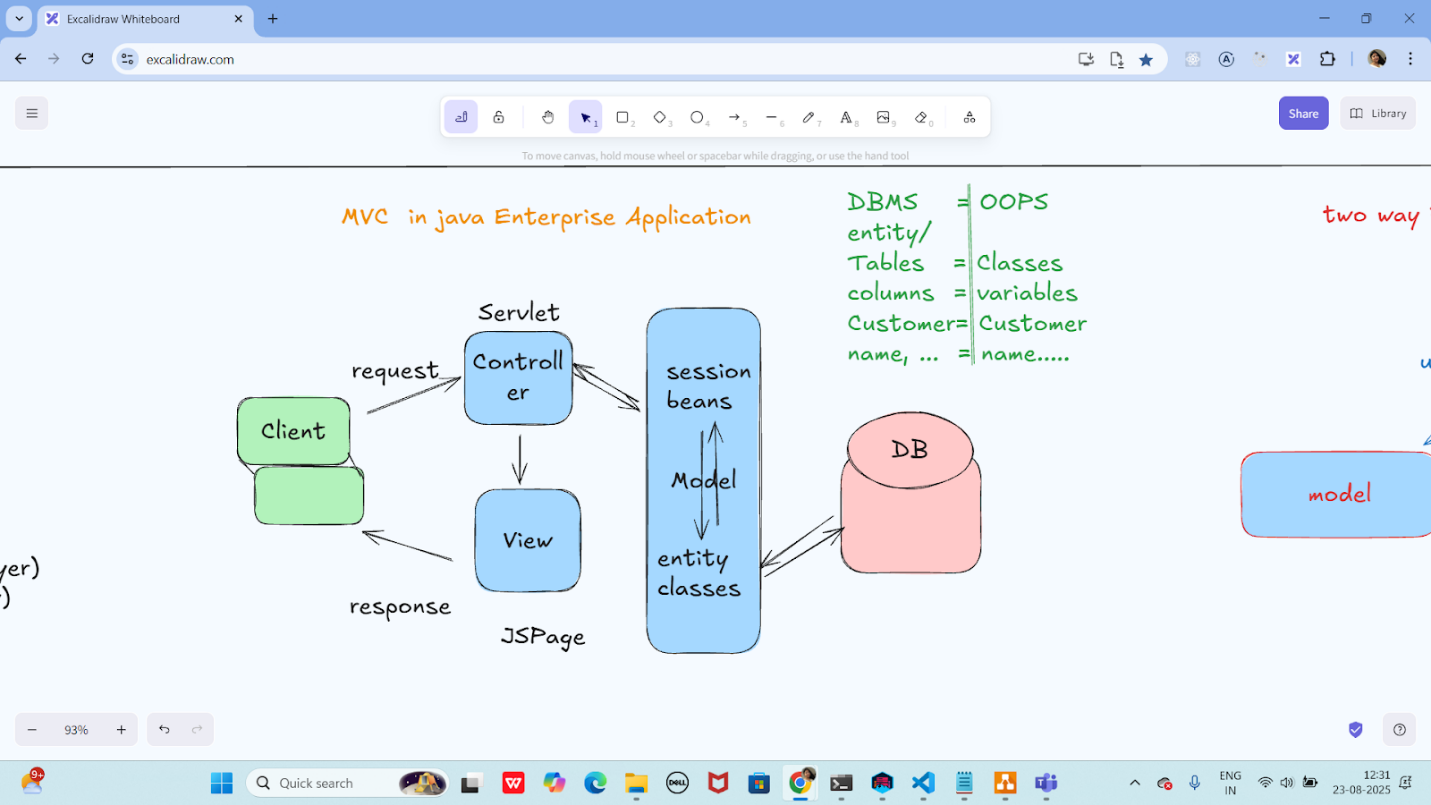
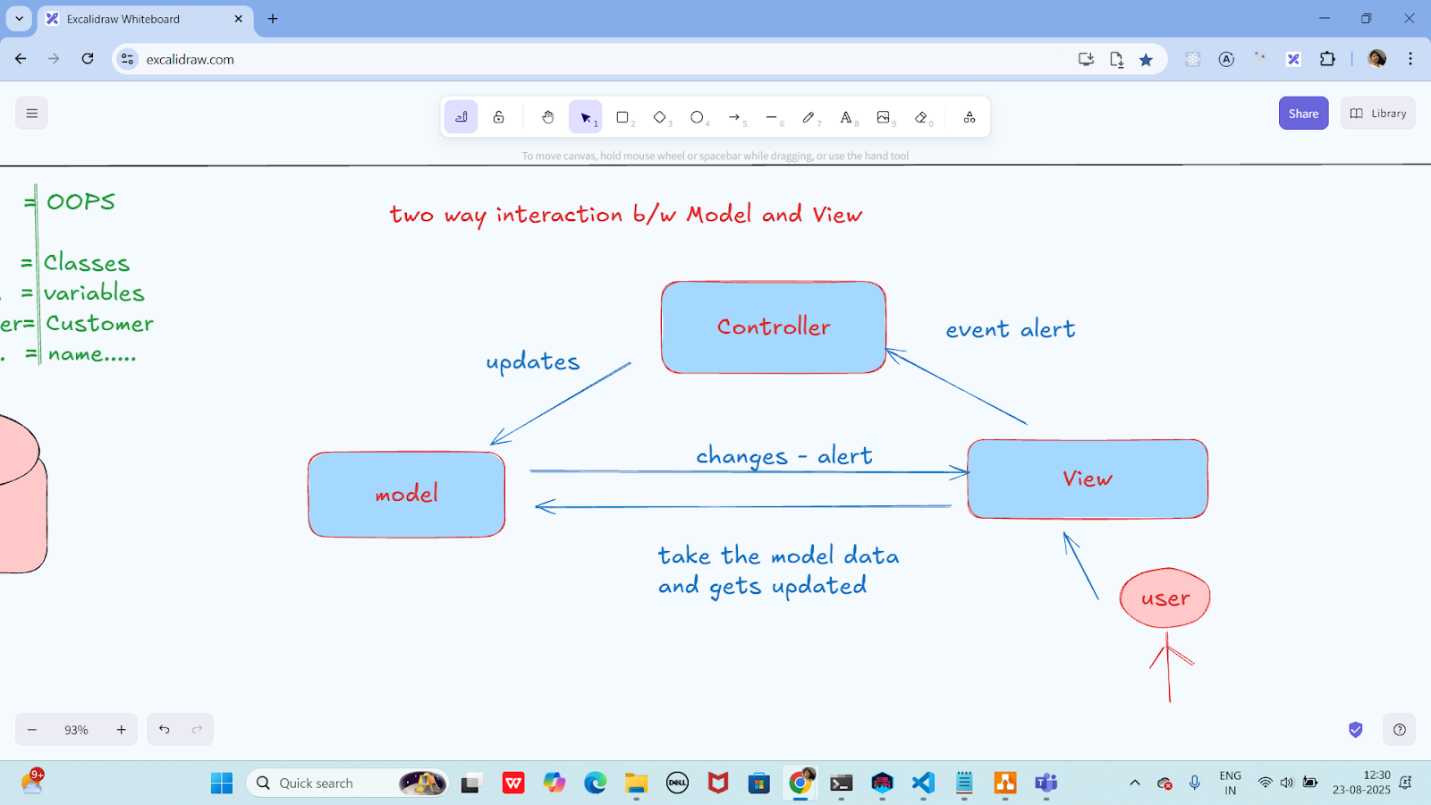
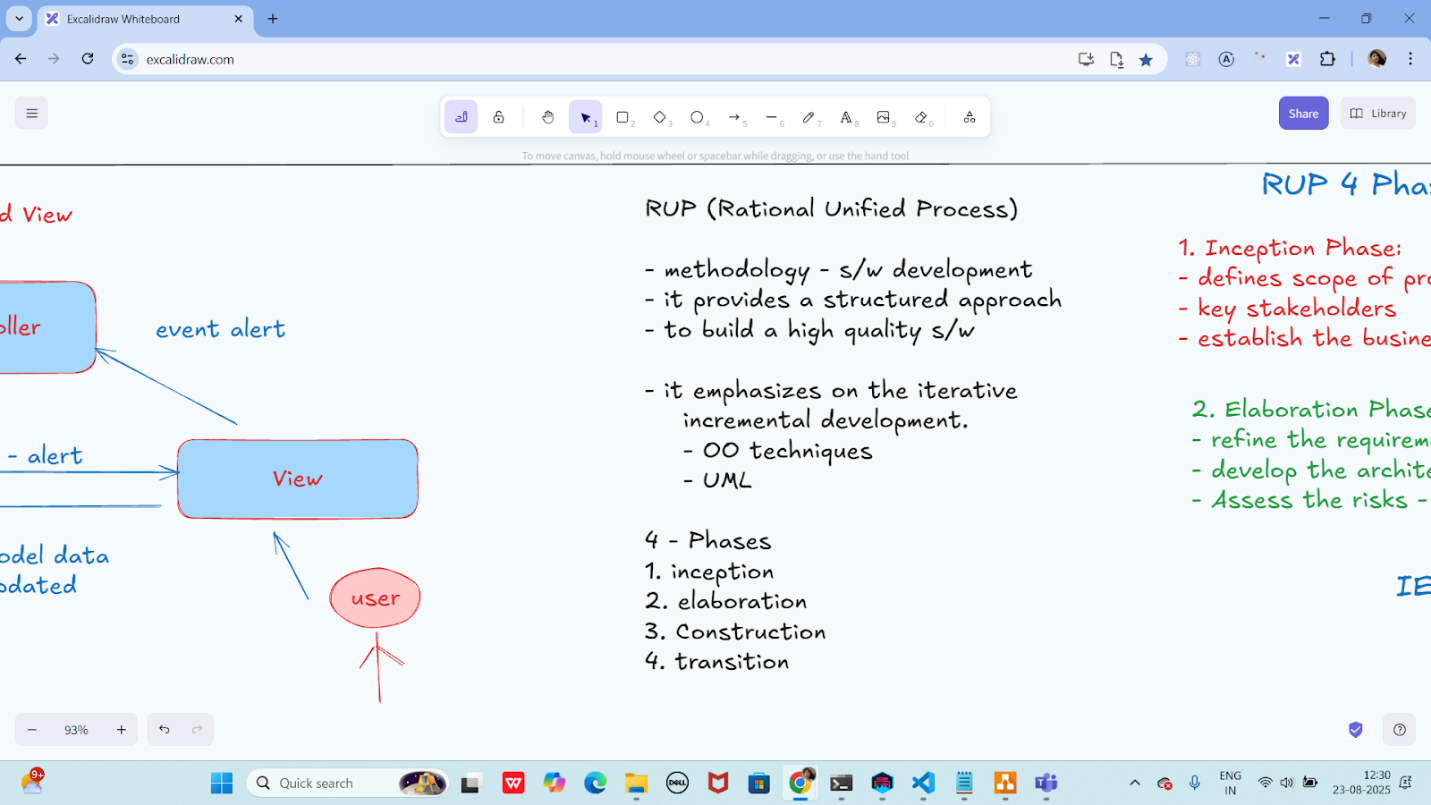
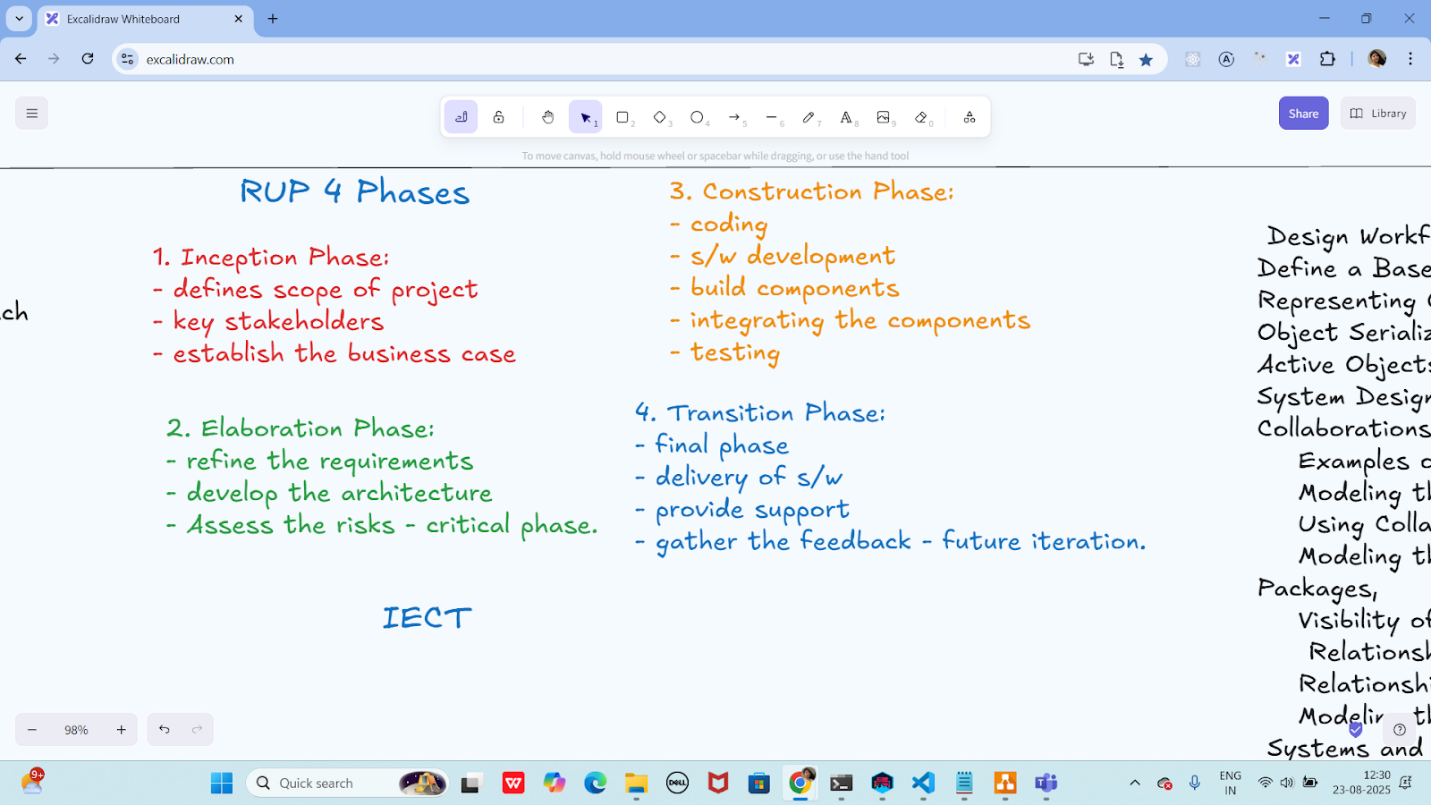
Day 27 - 23rd Aug 2025

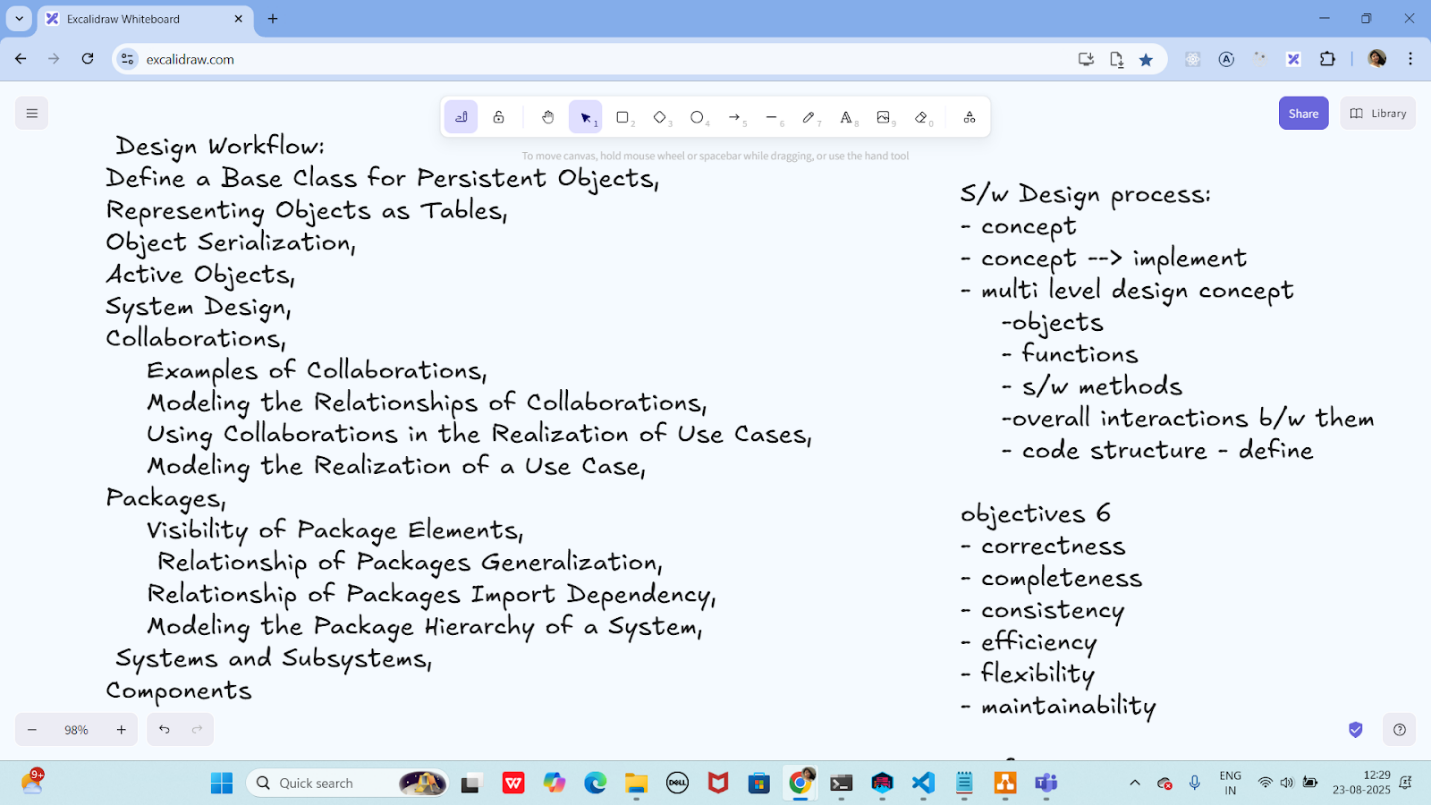


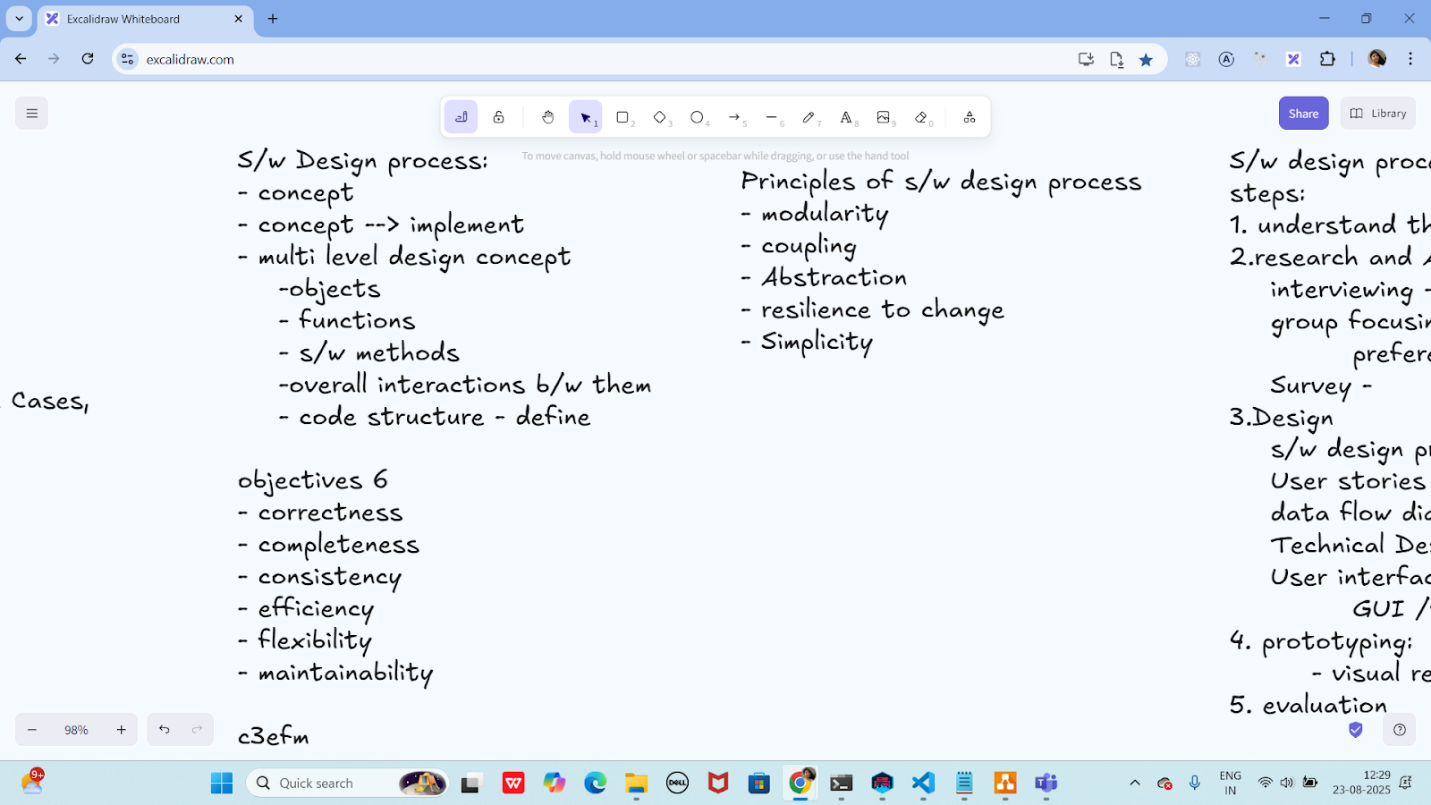


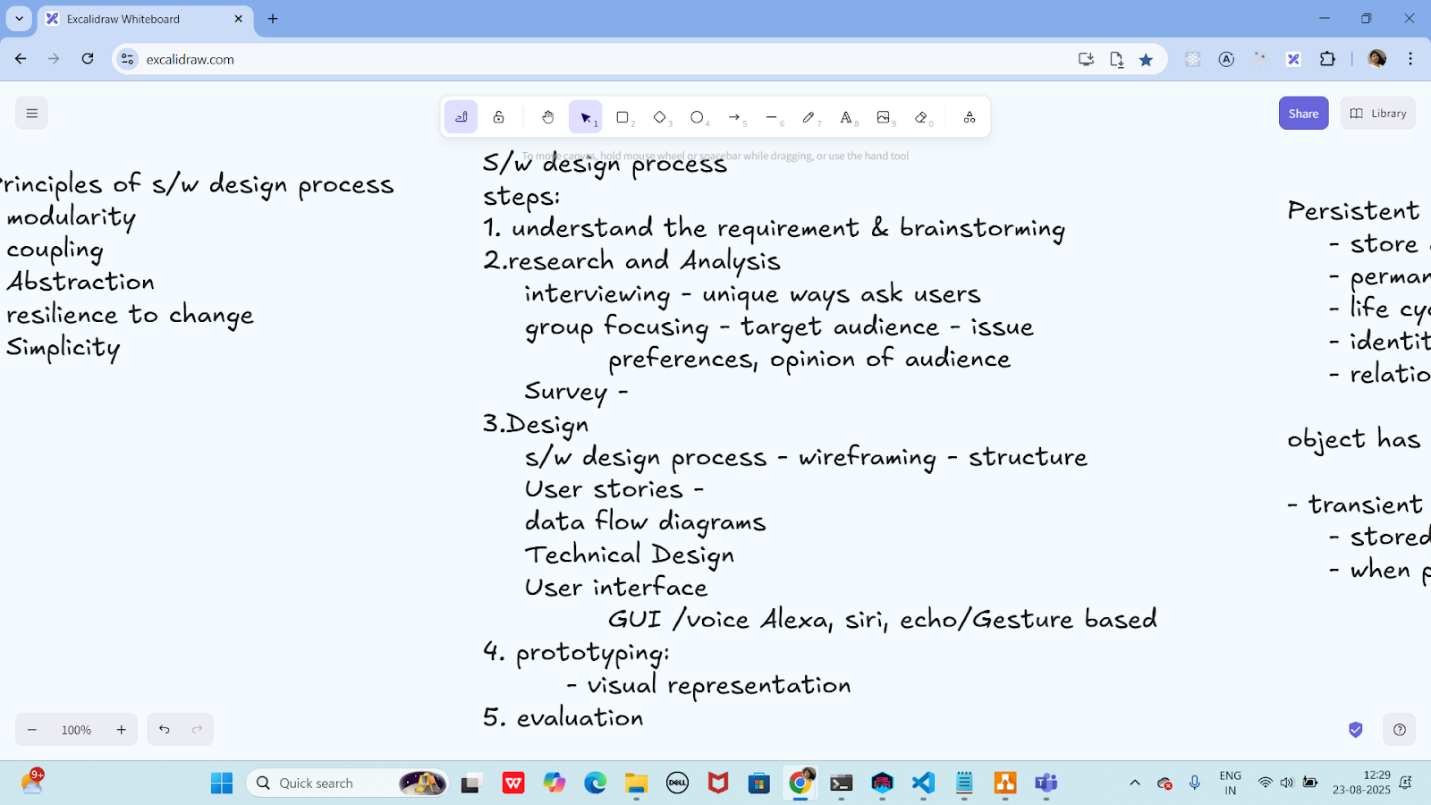


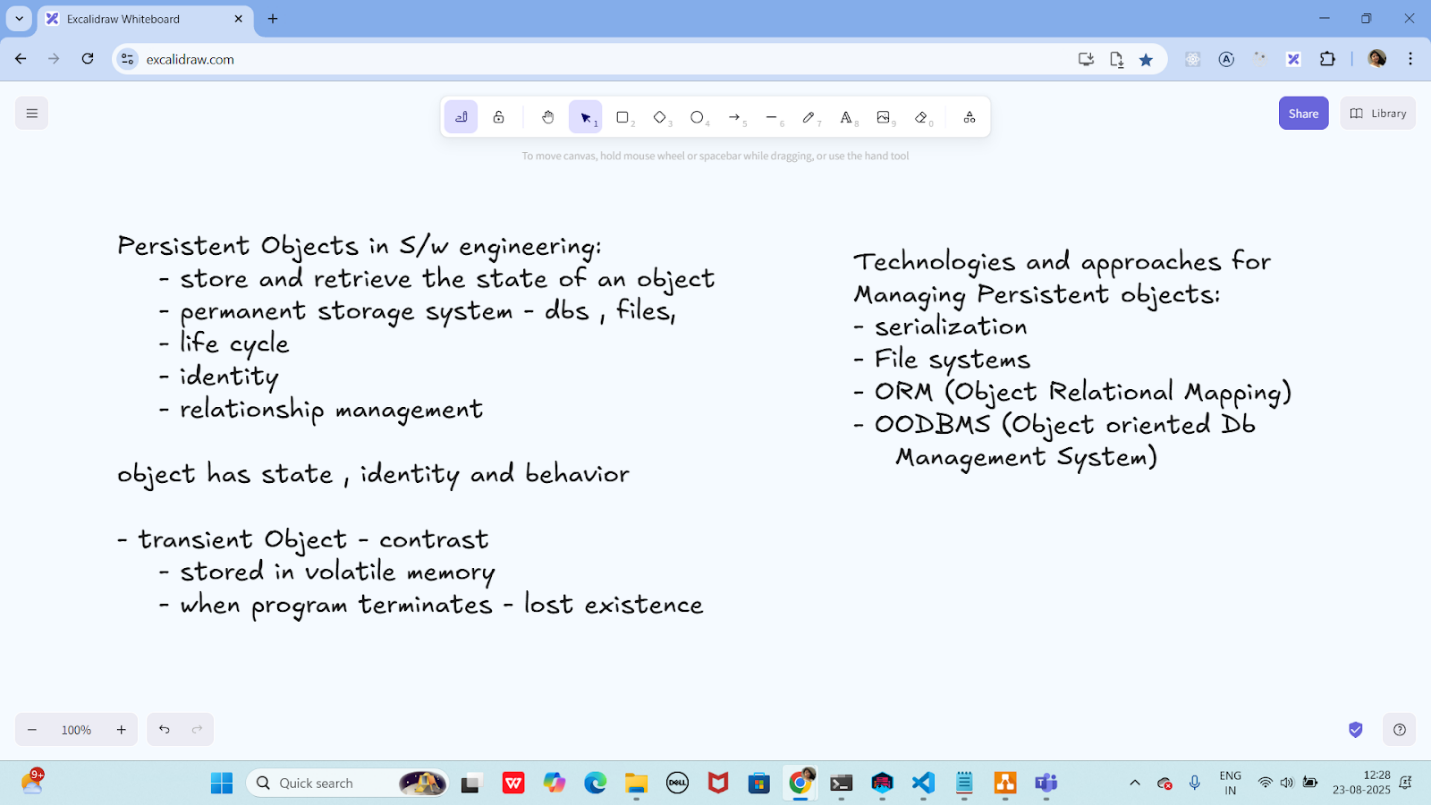


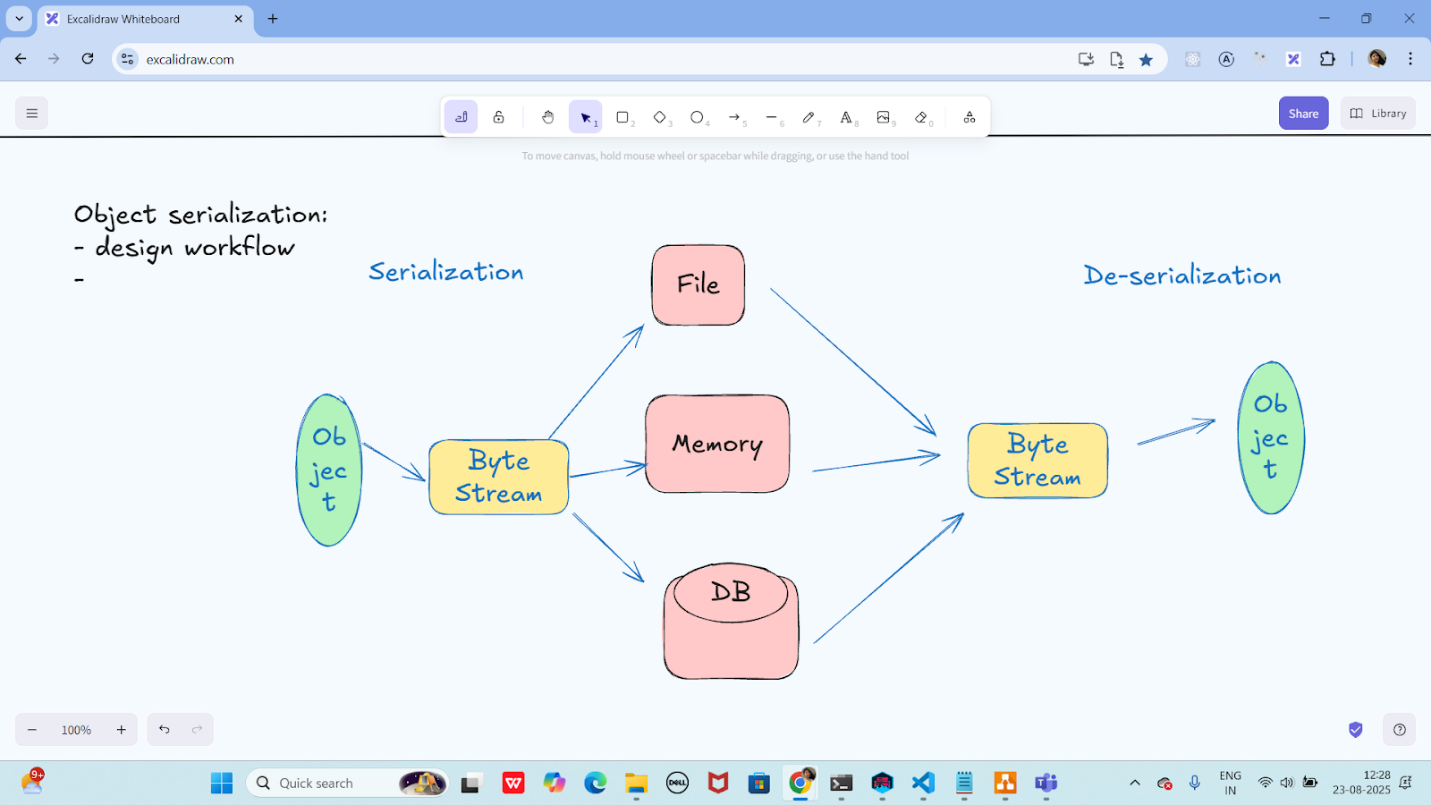


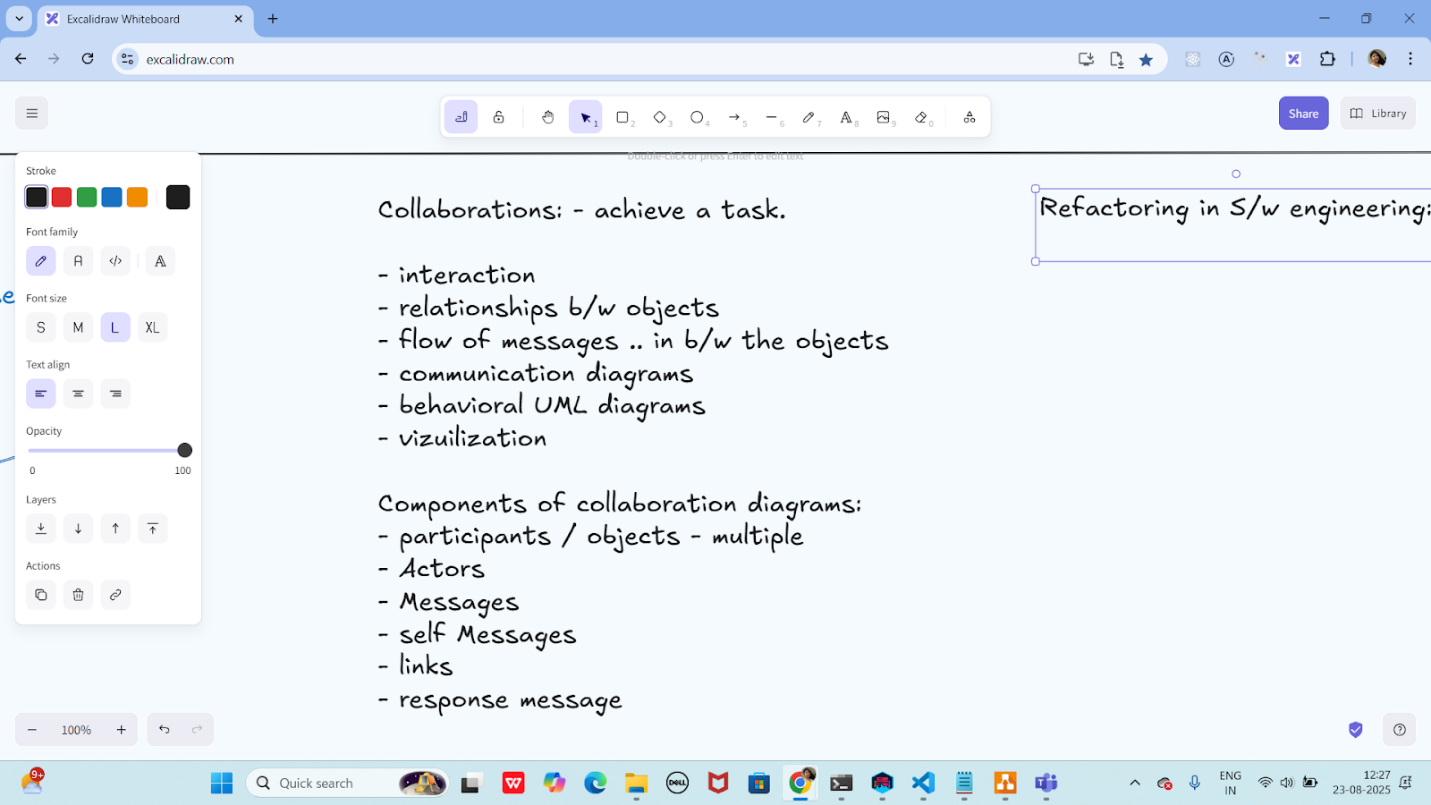


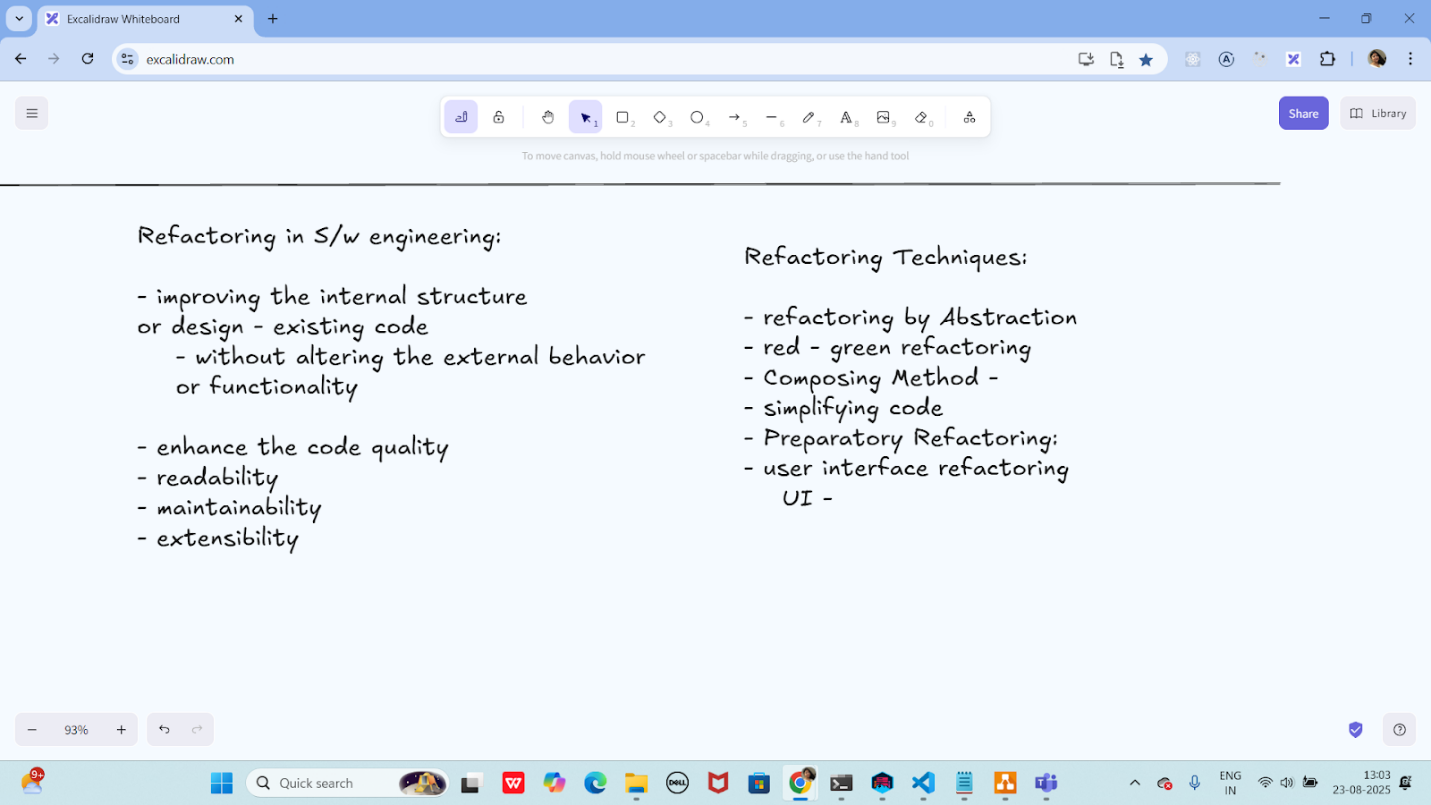












Task 1:

What do you think is the need for Refactoring?

Refactoring is the process of **restructuring existing code without changing its external behavior**. The need for refactoring arises from several important software development goals:

1. Improve Code Readability

2. Reduce Complexity

3. Enhance Maintainability

4. Remove Redundancy

5. Improve Performance (sometimes)

6. Facilitate Testing

Task 2:

What are the Principles of refactoring?

**Preserve Behavior** – Refactoring should not change the program’s external functionality.

**Improve Readability** – Make code easier to read and understand.

**Reduce Complexity** – Simplify logic, break down large methods, and minimize nested structures.

**Eliminate Duplication** – Remove repeated code and create reusable methods or classes.

**Enhance Maintainability** – Make code easier to modify, debug, and extend.

**Small, Incremental Changes** – Refactor in small steps and test frequently.

**Improve Flexibility** – Make code more adaptable to future changes.

**Test-Driven** – Always verify behavior with tests before and after refactoring.

Task 3:

What are the steps for performing code refactoring?

Identify → Understand → Test → Plan → Refactor → Test → Review → Document

**Identify the code to refactor** – Find complex, duplicated, or hard-to-maintain code.

**Understand the existing code** – Analyze what the code does and how it behaves.

**Write or verify tests** – Ensure unit or automated tests exist to check behavior.

**Plan the refactoring** – Decide on changes like renaming, extracting methods, or removing duplicates.

**Refactor the code** – Apply the planned changes carefully and incrementally.

**Test frequently** – Run tests after each small change to ensure behavior is preserved.

**Review and optimize** – Ensure code is clean, readable, and efficient.

**Document (optional)** – Update comments or documentation to reflect the new structure.

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;

}

}

Long Method

Primitive Obsession

Large Class

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

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

Home task

Task 01:

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

Lethwise sorting

Stanford

Watson

Ankit

package StrategyDesignPattern;

import java.util.\*;

// Strategy interface

interface SortStrategy {

   void sort(List<String> items);

}

// Concrete Strategy 1: Alphabetical Sorting (case insensitive)

class AlphabeticalSort implements SortStrategy {

   @Override

   public void sort(List<String> items) {

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

   }

}

// Concrete Strategy 2: Lengthwise Sorting

class LengthwiseSort implements SortStrategy {

   @Override

   public void sort(List<String> items) {

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

   }

}

// Context

class SortingStrategy {

   private SortStrategy strategy;   // current strategy

   private List<String> items;      // list of items

   public SortingStrategy() {

       items = new ArrayList<>();

   }

   // Set the sorting strategy dynamically

   public void setStrategy(SortStrategy strategy) {

       this.strategy = strategy;

   }

   // Add item to list

   public void addItem(String item) {

       items.add(item);

   }

   // Remove item from list

   public void removeItem(String item) {

       items.remove(item);

   }

   // Perform sorting

   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;

   }

}

package StrategyDesignPattern;

public class StrategyPatternDemo {

   public static void main(String[] args) {

       SortingStrategy context = new SortingStrategy();

       // Adding items

       context.addItem("Stanford");

       context.addItem("Ankit");

       context.addItem("Watson");

       context.addItem("Done");

       // Alphabetical sort

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

       context.setStrategy(new AlphabeticalSort());

       context.performSort();

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

           System.*out*.println(item);

       }

       // Lengthwise sort

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

       context.setStrategy(new LengthwiseSort());

       context.performSort();

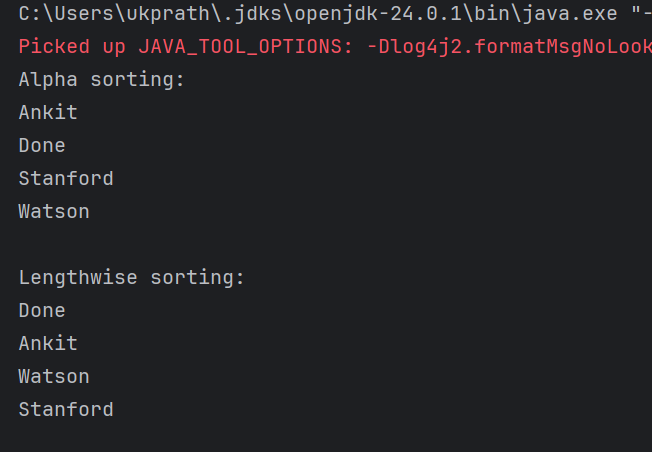
       for (String item : context.getList()) {

           System.*out*.println(item);

       }

   }

}



Info Box

Excaludraw updated at 10.36

<https://excalidraw.com/#json=o6fjfGm11LFTya5IpOBIf,6-fXCch1PyNqDjBH7er_vw>

Excalidraw updated at 11.47

<https://excalidraw.com/#json=2X41t78rqy31GWS_fdR85,hD8CBR_ZttwjoTL4D5aW1w>

Excalidraw updated at 12.23

<https://excalidraw.com/#json=HXkmPiMhiJJ8TAqRGU9ZQ,nEKpoZot-01V5BXSFl3DdQ>

Excalidraw updated at 12.51

<https://excalidraw.com/#json=CawldXC4D8kqUnsQIa1Cf,g4fLxLNdF3aTP4uxy1jbZQ>