Day 26 - 22nd Aug 2025

Code for reference link

<https://drive.google.com/drive/folders/1LwhNov1s1-vHzF9GPAObLSnP9kAvipmw?usp=sharing>

Strategy Method Design pattern

package StrategyMP2;

public class Main {

   public static void main(String[] args) {

       System.*out*.println("Strategy Method Design PAttern - Behavioral DP");

   }

}

package StrategyMP2;

public interface PaymentStrategy {

   void process(double price);

}

package StrategyMP2;

public class PaymentUsingStrategy {

   private PaymentStrategy paymentStrategy;

   public PaymentUsingStrategy(PaymentStrategy paymentStrategy) {

       this.paymentStrategy = paymentStrategy;

   }

   public void process(double price) {

       paymentStrategy.process(price);

   }

}

package StrategyMP2;

public class CardPaymentConcreteStrategy implements PaymentStrategy {

   public void process(double price) {

       System.*out*.println("payment processing using card");

   }

}

package StrategyMP2;

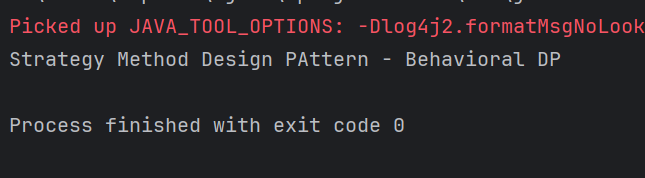
public class Main {

   public static void main(String[] args) {

       System.*out*.println("Strategy Method Design PAttern - Behavioral DP");

   }

}



Rating Qns:

Task 2:

What do you know about Design Work Flow…

A design workflow is the process of moving from an idea to a finished solution. It starts with research to understand the problem, then moves to brainstorming and developing concepts. These ideas are refined through prototypes, reviewed for feedback, and adjusted as needed. Finally, the design is implemented, tested, delivered, and later improved based on results and user feedback.

[Start]

   |

   v

[Problem & Requirements]

   |

   v

[High-Level Design (Architecture, Modules)]

   |

   v

[Detailed Design (UML, APIs, Data Models)]

   |

   v

[Implementation (Java Code)]

   |

   v

[Testing (Unit, Integration)]

   |

   v

[Review & Refactor]

   |

   v

[Deploy (Jar/War, Docker, Cloud)]

   |

   v

[Monitor & Iterate]

   |

   v

[End / Next Sprint]

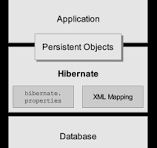
Task 3:

What do you know Persistent objects:

**Persistent objects** are objects in programming whose state **outlives the process that created them**. In other words, instead of disappearing when a program ends (like normal in-memory objects), persistent objects are stored in a way that they can be retrieved and reused later.

For example, if an object is saved to a database, a file, or some other permanent storage, it becomes persistent. When the program runs again, the object’s state can be reloaded so it continues from where it left off.

👉 In short: **persistent objects = objects stored beyond program execution (saved to disk, DB, etc.) so they can be reused later**.



Task 4:

Which of the following components is not typically part of the Command pattern?

a) Invoker

b) Receiver

c) Abstract Factory

d) Command (interface/abstract class)

The **Command Pattern** typically consists of:

* **Command (interface/abstract class)** → declares the action.
* **ConcreteCommand** → implements the action.
* **Receiver** → the actual object that performs the work.
* **Invoker** → calls the command.
* **Client** → creates and configures commands.

Task 5:

What role does the Invoker play in the Command pattern?

a) It knows how to perform the operations associated with a request.

b) It encapsulates the request as an object.

c) It asks the command to carry out the request.

d) It defines the interface for executing an operation.

**Receiver** → *Knows how to perform the operations* (**a**).

**Command (interface/abstract class)** → *Defines the interface for executing an operation* (**d**).

**ConcreteCommand** → *Encapsulates the request as an object* (**b**).

**Invoker** → *Holds a command and calls its execute() method → asks the command to carry out the request* (**c**).

Task 6:

A key benefit of using the Command pattern is its ability to support:

a) Lazy initialization

b) Undo/Redo functionality

c) Singleton instance creation

d) Compile-time polymorphism

The **Command Pattern** encapsulates requests as objects, which provides several benefits:

* **Undo/Redo functionality** → Since commands are objects, you can store them in a history list and reverse or reapply them later.
* **Queueing or logging requests** → Commands can be placed in queues or logs for later execution.
* **Decoupling sender (Invoker) from receiver** → The object that issues a request doesn’t need to know who performs it.

Other options:

* **a) Lazy initialization** → Related to performance optimization, not Command pattern.
* **c) Singleton** → Creational pattern, unrelated.
* **d) Compile-time polymorphism** → Related to method overloading, not Command pattern.

Task 7:

In the Strategy pattern, what role does the "Context" play?

A. It defines the interface for the algorithms.

B. It implements a specific algorithm.

C. It maintains a reference to a Strategy object and delegates the task to it.

D. It creates the Concrete Strategy objects.

Task 8:

In which of the following mechanisms, types of all variables and expressions are fixed at compilation time.

a) Strong Typing

b) Weak Typing

c) Static Binding/ early binding

d) Dynamic Binding/ late binding

**Static Binding / Early Binding** → The types of all variables and expressions are determined **at compile time**. Example: method overloading in Java.

**Dynamic Binding / Late Binding** → The method to call is resolved **at runtime**. Example: method overriding (polymorphism) in Java.

**Strong Typing** → Language enforces strict type rules (e.g., Java, C#).

**Weak Typing** → Language is more flexible with types (e.g., JavaScript, Python).

Task 9:

In which pattern does a class represent the functionality of another class, providing a simplified interface to a complex subsystem?

a) Decorator Pattern

b) Facade Pattern

c) Proxy Pattern

d) Composite Pattern

**Explanation**

* **Facade Pattern** → Provides a **simplified interface** to a complex subsystem. It hides complexity and makes the subsystem easier to use.
* **Decorator Pattern** → Dynamically adds responsibilities/behaviors to an object.
* **Proxy Pattern** → Acts as a placeholder or surrogate for another object, controlling access.
* **Composite Pattern** → Composes objects into tree structures to represent part-whole hierarchies.

Task 10:

 Which of the following statements about Persistence is correct?

a) It is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most they may be interchanged only in very restricted ways.

b) It is the property of an object through which its existence transcends time and/or space.

c) It is the property that distinguishes an active object from one that is not active.

d) All of the mentioned

Task 11:

 What is that concept in type theory in which a single name may denote objects of many different classes that are related by some common super class referred to \_\_\_\_\_\_

a) Monomorphism

b) Type Checking

c) Polymorphism

d) Generalization

Task 12:

Which of the following patterns is used to create a single instance of a class and provide a global point of access to it?

a) Factory Pattern

b) Singleton Pattern

c) Builder Pattern

d) Prototype Pattern

**Explanation**

* **Singleton Pattern** → Ensures that only **one instance** of a class is created and provides a **global point of access** to it.  
    
  + Example: Logger, Database Connection Pool, Configuration Manager.
* **Factory Pattern** → Creates objects without exposing the instantiation logic.
* **Builder Pattern** → Constructs complex objects step by step.
* **Prototype Pattern** → Creates new objects by copying existing ones (c

Task 13:

 The Adapter pattern is a type of \_\_\_\_\_\_ pattern.

a) Creational

b) Structural

c) Behavioral

d) Concurrency

**Explanation**

* **Adapter Pattern** → A **structural pattern** that allows incompatible interfaces to work together by acting as a bridge (adapter).  
    
  + Example: A memory card reader works as an adapter between a memory card and a laptop USB port.
* **Creational Patterns** → Deal with object creation (Factory, Singleton, Builder).
* **Behavioral Patterns** → Deal with communication between objects (Strategy, Observer, Command).
* **Concurrency Patterns** → Deal with multi-threaded programming (like Thread Pool).

Task 14:

Which design pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically?

a) Strategy Pattern

b) Command Pattern

c) Observer Pattern

d) Mediator Pattern

**Explanation**

* **Observer Pattern** → Defines a **one-to-many dependency** between objects.  
    
  + When the **Subject (observable)** changes state, all registered **Observers (listeners)** are automatically notified and updated.
  + Example: Notification systems, event listeners in GUIs, stock price updates.
* **Strategy Pattern** → Encapsulates algorithms and makes them interchangeable.
* **Command Pattern** → Encapsulates requests as objects.
* **Mediator Pattern** → Defines an object that centralizes communication between other objects to reduce dependencies.

Task 15:

The Model-View-Controller (MVC) is an example of a \_\_\_\_\_\_ pattern.

a) Creational

b) Structural

c) Behavioral

d) Architectural

**Explanation**

* **MVC (Model–View–Controller)** → An **architectural pattern** used to separate an application into three main components:  
    
  + **Model** → Manages data, logic, and rules.
  + **View** → Displays information (UI).
  + **Controller** → Handles user input and updates the model/view.
* **Creational Patterns** → Object creation (Factory, Singleton, Builder).
* **Structural Patterns** → Class/object composition (Adapter, Facade, Composite).
* **Behavioral Patterns** → Object interaction and responsibilities (Observer, Strategy, Command).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Home Tasks:

Task 1 :

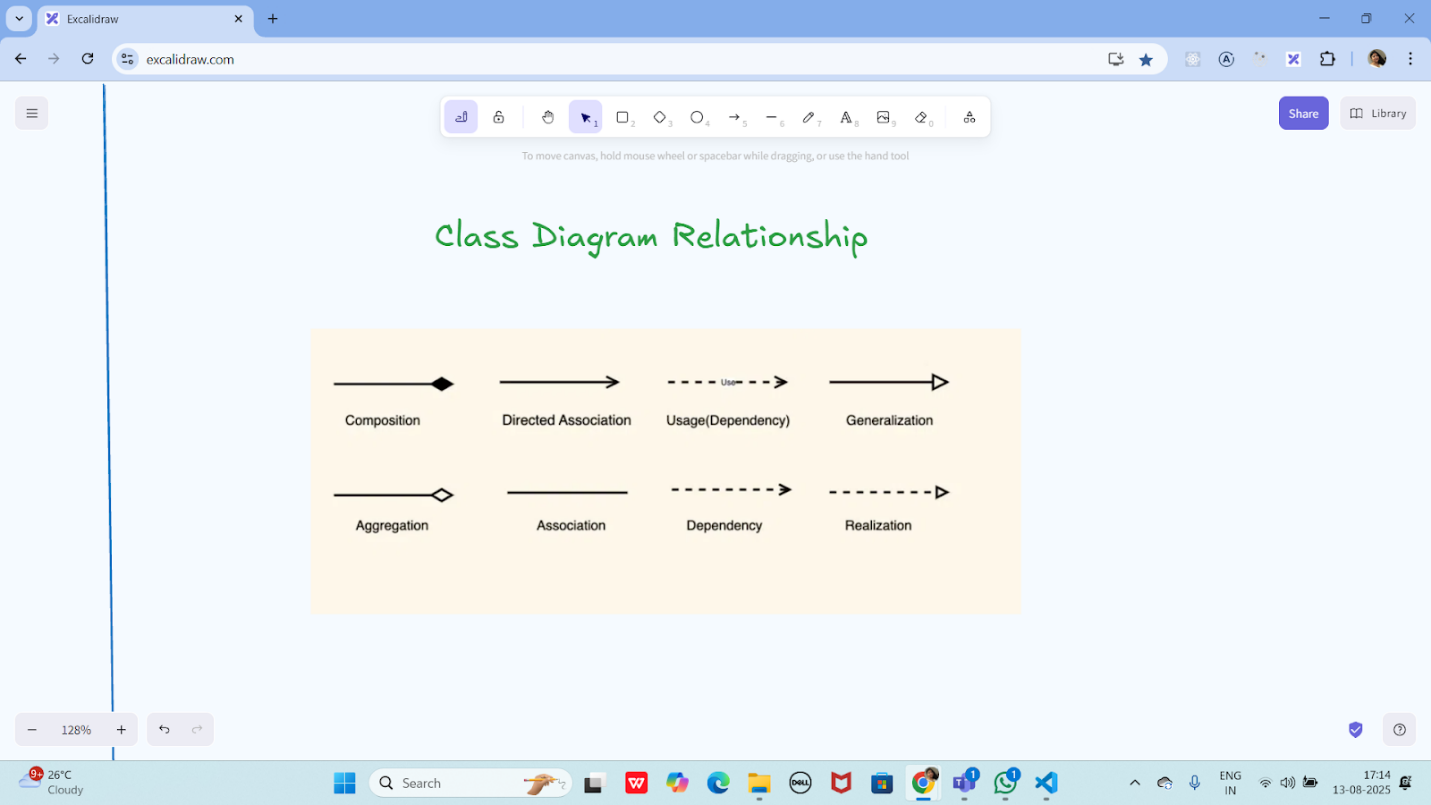
Pub Subs code – complete it

Task 2:

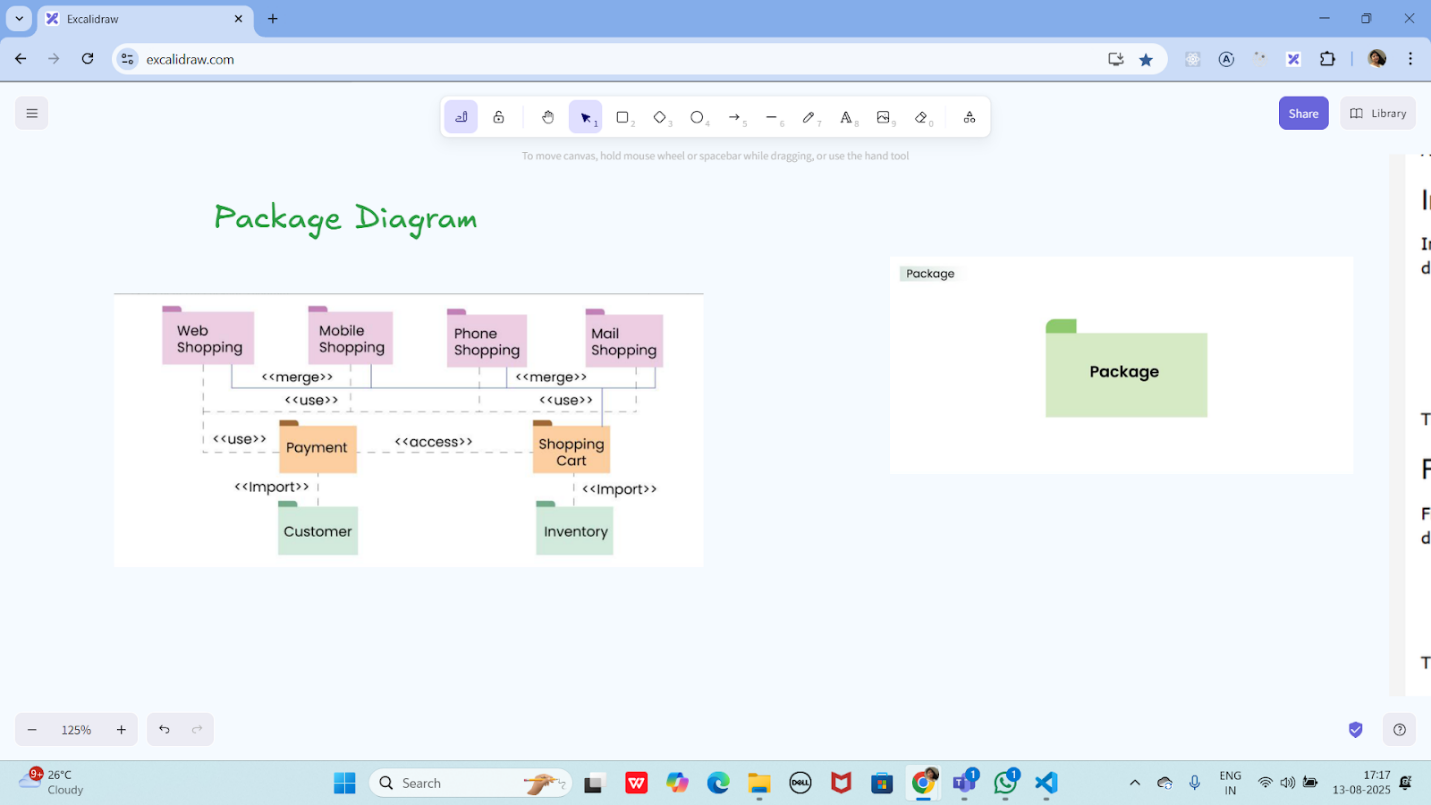
Practice singleton and Strategy design patterns …

You need to explain by sharing the screen..

Task 03:



Task 04:



Create a package diagram representation of Command Design Pattern code..

Create uml diagrams to represent different kinds of relation ships

package SingletonMP;

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

class DataManager {

   private static DataManager *instance*;

   private List<String> itemList;

   // Private constructor for singleton; throws exception if already initialized

   private DataManager() {

       if (*instance* != null) {

           throw new IllegalStateException("Singleton instance already created. Use getInstance() method.");

       }

       itemList = new ArrayList<>();

   }

   // Thread-safe singleton instance getter

   public static synchronized DataManager getInstance() {

       if (*instance* == null) {

*instance* = new DataManager();

       }

       return *instance*;

   }

   // Synchronized list operations

   public synchronized void addItem(String item) {

       itemList.add(item);

   }

   public synchronized void removeItem(String item) {

       itemList.remove(item);

   }

   public synchronized List<String> getList() {

       return new ArrayList<>(itemList);

   }

}

// --- Non-editable starts here ---

package SingletonMP;

import java.util.Scanner;

public class Main {

   public static void main(String[] args) {

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

       try {

           // This first call tests the exception if you do something weird in tests

           DataManager dataManager = DataManager.*getInstance*();

       } catch (IllegalStateException e) {

           System.*out*.println("Caught IllegalStateException: " + e.getMessage());

       }

       DataManager dataManager = DataManager.*getInstance*();

       // Read items continuously without "done" sentinel

       String input;

       while (scanner.hasNextLine()) {

           input = scanner.nextLine();

           if (input.isEmpty()) break; // stop if user enters empty line

           dataManager.addItem(input);

       }

       // Read item to remove

       String itemToRemove = scanner.nextLine();

       dataManager.removeItem(itemToRemove);

       // Output final list

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

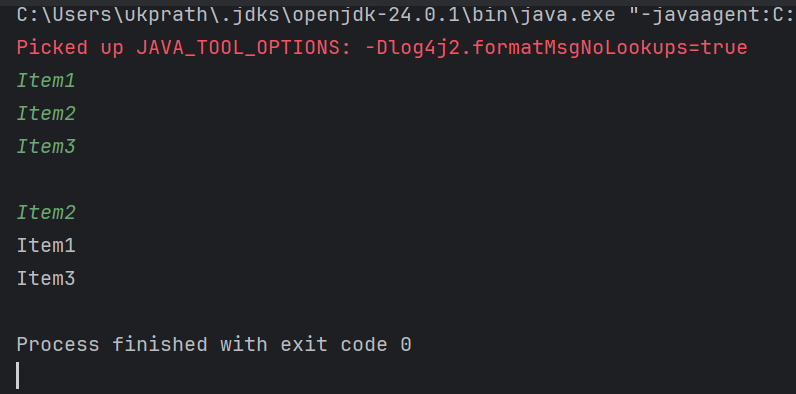
           System.*out*.println(item);

       }

       scanner.close();

   }

}



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Info Box

Excalidraw updated at 9.41am

<https://excalidraw.com/#json=BzWUiyDQJSrM3H4fnMZmA,8ku7hxm4MdwGtTl8l9xBAg>

Excalidraw updated at 12.57

<https://excalidraw.com/#json=wKUoOyPWlHWl88tFfcZLN,2peOceri73zO1fANtEIRXg>

Exaclidraw updated at 13.20

<https://excalidraw.com/#json=FvOSdghWg2edjx8IdkC_6,Jmc2IB9qmnTVmDtKPUgvZA>

Excalidraw updated at 15.38

<https://excalidraw.com/#json=ducQ4gsj0ERH6C5R1FeAX,3jVSsU8c1C5rxj9f9KMtBw>

Excalidraw updated at 16.01

<https://excalidraw.com/#json=203GR_BNJIcTIEDfwxLgl,H0yHSbcWxSqo1IJs35Cmfw>

For MCQ practice

<https://mcqmate.com/topic/object-oriented-software-engineering>

Mcqmate:

Git hub.. Link update sheet

<https://docs.google.com/spreadsheets/d/1y1753RPMqal4WxHqvdBuV4jqh5bMFWr_N4tcavWOmH8/edit?usp=sharing>

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%