Day 23 aug 7  
  
Name: Aravind Kasanagottu

Id: mvsnarav  
  
  
**Task 1**   
public class BuilderPatternDemo {

// Product interface

interface Gadget {

void showSpecs();

}

// Concrete product

static class Mobile implements Gadget {

private int storage;

private int memory;

public void setStorage(int storage) {

this.storage = storage;

}

public void setMemory(int memory) {

this.memory = memory;

}

@Override

public void showSpecs() {

System.out.println("Mobile Specifications:");

System.out.println("Storage: " + storage + " GB");

System.out.println("Memory: " + memory + " GB");

}

}

// Builder interface

interface GadgetBuilder {

GadgetBuilder buildStorage(int storage);

GadgetBuilder buildMemory(int memory);

Gadget build();

}

// Concrete builder

static class MobileGadgetBuilder implements GadgetBuilder {

private Mobile mobile;

public MobileGadgetBuilder() {

this.mobile = new Mobile();

}

@Override

public GadgetBuilder buildStorage(int storage) {

mobile.setStorage(storage);

return this;

}

@Override

public GadgetBuilder buildMemory(int memory) {

mobile.setMemory(memory);

return this;

}

@Override

public Gadget build() {

return mobile;

}

}

// Director

static class GadgetDirector {

private GadgetBuilder gadgetBuilder;

public GadgetDirector(GadgetBuilder gadgetBuilder) {

this.gadgetBuilder = gadgetBuilder;

}

public Gadget constructGadget() {

return gadgetBuilder

.buildStorage(512)

.buildMemory(16)

.build();

}

}

// Main method (Client code)

public static void main(String[] args) {

GadgetBuilder builder = new MobileGadgetBuilder();

GadgetDirector director = new GadgetDirector(builder);

Gadget mobile = director.constructGadget();

mobile.showSpecs();

}

}   
output:- Mobile Specifications:

Storage: 512 GB

Memory: 16 GB

**Task 2**  
// Iphone interface

public interface Iphone {

void onCharge();

void offCharge();

}

// Charger interface

interface Charger {

void charge();

void removeCharge();

}

// Concrete class implementing Charger

class Iphone16Charger implements Charger {

Iphone16Charger() {

System.out.println("Iphone16Charger is ready.");

}

@Override

public void charge() {

System.out.println("I am charging Iphone 16");

}

@Override

public void removeCharge() {

System.out.println("Stopped charging Iphone 16");

}

}

// Adapter class implementing Charger and adapting Iphone16Charger

class Iphone16Adapter implements Charger {

Iphone16Charger iphone16Charger;

Iphone16Adapter() {

iphone16Charger = new Iphone16Charger();

}

@Override

public void charge() {

iphone16Charger.charge();

}

@Override

public void removeCharge() {

iphone16Charger.removeCharge();

}

}

// Iphone16 class implementing Iphone and using the adapter

class Iphone16 implements Iphone {

Charger iphone16Adapter;

Iphone16(Charger iphone16Adapter) {

this.iphone16Adapter = iphone16Adapter;

}

@Override

public void onCharge() {

iphone16Adapter.charge();

}

@Override

public void offCharge() {

iphone16Adapter.removeCharge();

}

}

// Client class to test the Adapter pattern

public class ClientAdapterDpMain {

public static void main(String[] args) {

Iphone iphoneObj = new Iphone16(new Iphone16Adapter());

iphoneObj.onCharge();

iphoneObj.offCharge();

}

}

Output:- Iphone16Charger is ready.

I am charging Iphone 16

Stopped charging Iphone 16   
  
 **Task 3**   
public class FacadePatternDemo {

// Subsystem class

static class FdService {

public void getFdServiceDetails(String accountNo) {

System.out.println("Fetching FD details for account: " + accountNo);

}

}

// Facade class

static class BankFacade {

private FdService fdService;

// Constructor

public BankFacade() {

this.fdService = new FdService();

}

// Facade method to access FD service

public void getFdServiceDetails(String accountNo) {

fdService.getFdServiceDetails(accountNo);

}

}

// Client code (main method)

public static void main(String[] args) {

BankFacade bank = new BankFacade();

bank.getFdServiceDetails("ACC1234567890");

}

}  
  
output:- Fetching FD details for account: ACC1234567890

**Task 4**import java.util.HashMap;

// Flyweight interface

interface Shape {

void draw(int x, int y, int radius);

}

// Concrete Flyweight class

class Circle implements Shape {

private String color; // intrinsic state

public Circle(String color) {

this.color = color;

}

@Override

public void draw(int x, int y, int radius) {

System.out.println("Drawing Circle [Color: " + color +

", X: " + x + ", Y: " + y + ", Radius: " + radius + "]");

}

}

// Flyweight Factory

class ShapeFactory {

private static final HashMap<String, Shape> circleMap = new HashMap<>();

public static Shape getCircle(String color) {

Circle circle = (Circle) circleMap.get(color);

if (circle == null) {

circle = new Circle(color);

circleMap.put(color, circle);

System.out.println("Creating circle of color: " + color);

}

return circle;

}

}

// Client Code

public class FlyweightPatternDemo {

private static final String[] colors = { "Red", "Green", "Blue", "White", "Black" };

public static void main(String[] args) {

for (int i = 0; i < 10; ++i) {

Shape circle = ShapeFactory.getCircle(getRandomColor());

int x = getRandomX();

int y = getRandomY();

int radius = 100;

circle.draw(x, y, radius);

}

}

private static String getRandomColor() {

return colors[(int)(Math.random() \* colors.length)];

}

private static int getRandomX() {

return (int)(Math.random() \* 100);

}

private static int getRandomY() {

return (int)(Math.random() \* 100);

}

}   
output:- Creating circle of color: Red

Drawing Circle [Color: Red, X: 38, Y: 92, Radius: 100]

Creating circle of color: Green

Drawing Circle [Color: Green, X: 12, Y: 67, Radius: 100]

Drawing Circle [Color: Red, X: 91, Y: 4, Radius: 100]

**Task 5**// Bridge Pattern Demo

public class BridgePatternDemo {

// Implementor interface

interface Color {

String fill();

}

// Concrete Implementors

static class RedColor implements Color {

@Override

public String fill() {

return "Red";

}

}

static class BlueColor implements Color {

@Override

public String fill() {

return "Blue";

}

}

// Abstraction

abstract static class Shape {

protected Color color; // Bridge to implementor

public Shape(Color color) {

this.color = color;

}

abstract void draw();

}

// Refined Abstraction

static class Circle extends Shape {

public Circle(Color color) {

super(color);

}

@Override

public void draw() {

System.out.println("Drawing Circle filled with color: " + color.fill());

}

}

static class Square extends Shape {

public Square(Color color) {

super(color);

}

@Override

public void draw() {

System.out.println("Drawing Square filled with color: " + color.fill());

}

}

// Client code

public static void main(String[] args) {

Shape redCircle = new Circle(new RedColor());

Shape blueSquare = new Square(new BlueColor());

redCircle.draw(); // Output: Drawing Circle filled with color: Red

blueSquare.draw(); // Output: Drawing Square filled with color: Blue

}

}

Output:- Drawing Circle filled with color: Red

Drawing Square filled with color: Blue

**Task 6**

Create a class diagram in uml..

@startuml

class Calculator {

+ calculateSum(in num1: int, in num2: int, out result: int)

+ updateStatus(inout status: String)

}

@enduml

**Task 7**

What are the commonly used visibility Notations in class diagrams?

Answer:- In UML **class diagrams**, **visibility notations** define how class members (attributes and methods) are accessed. These notations specify the **access level** of a field or method.

**✅ Commonly Used Visibility Notations:**

| **Symbol** | **Visibility** | **Description** |
| --- | --- | --- |
| + | **Public** | Accessible from **anywhere** (any class) |
| - | **Private** | Accessible **only within the class** |
| # | **Protected** | Accessible within the class **and subclasses** |
| ~ | **Package** (default) | Accessible by **classes in the same package/module** |

**🧠 Example UML Class with Visibility Notations:**

@startuml

class Example {

- id: int // private attribute

+ name: String // public attribute

# calculate(): double // protected method

~ logData(): void // package-private method

}

@enduml

**✅ Summary:**

| **Notation** | **Keyword Equivalent in Java** |
| --- | --- |
| + | public |
| - | private |
| # | protected |
| ~ | *(default access)* |

**Task 8**What do you know about Parameter Directionality?

Answer:- **✅ Parameter Directionality in UML**

**Parameter directionality** describes how data flows **into and out of operations (methods)** in a **UML class diagram**. It is used to clarify whether a parameter is:

* An **input**
* An **output**
* Or **both**

This concept is helpful when modeling systems, especially in **interface design**, **method documentation**, and **collaborative development**.

**🎯 Common Directional Keywords**

| **Direction** | **Keyword** | **Meaning** |
| --- | --- | --- |
| Input | in | Data is passed **into** the operation only. |
| Output | out | Data is passed **out** of the operation (the method sets it). |
| Input/Output | inout | Data is passed in, possibly **modified**, and returned. |
| Return | *(return type)* | Returned from the method (not a parameter, but part of signature). |

**🧠 Example (UML Method Signature)**

class Calculator {

+ calculateSum(in num1: int, in num2: int, out result: int)

+ updateStatus(inout status: String)

}

* calculateSum: takes two numbers (in), calculates sum, stores it in result (out)
* updateStatus: modifies the passed-in status string (inout)

**🔁 Usage in Programming Languages**

| **Language** | **Native Support for in, out, inout** |
| --- | --- |
| **Java** | ❌ No direct support (use return or mutable objects) |
| **C++** | ✅ Use pointers or references |
| **C#** | ✅ Has ref, out, and in keywords |
| **Python** | ❌ Pass-by-object-reference can simulate inout |
| **Swift** | ✅ Has inout keyword |

**✅ Summary**

Parameter directionality:

* Improves method clarity in UML diagrams.
* Defines **how data flows** between the caller and the operation.
* Is especially important in **interface design**, **distributed systems**, and **multi-language modeling**.

**Task 9**

Class Diagram Relationships

**✅ UML Class Diagram Relationships**

In **UML class diagrams**, relationships between classes help describe **how classes are connected** and **interact** with each other. Understanding these relationships is essential for designing object-oriented systems.

**📘 Common UML Class Diagram Relationships**

| **Relationship** | **Symbol** | **Meaning** | **Example** |
| --- | --- | --- | --- |
| **Association** | → or line | "Uses" or "has a" relationship between two classes | A Customer places an Order |
| **Multiplicity** | 1, 0..1, \* | How many objects are involved | A Company has \* Employees |
| **Aggregation** | ◇ | Whole-part (weak ownership) | A Team has Players (players can exist independently) |
| **Composition** | ◆ (filled diamond) | Whole-part (strong ownership) | A House has Rooms (rooms cannot exist without the house) |
| **Inheritance / Generalization** | ▲ (triangle) | "Is-a" relationship | A Dog is a Animal |
| **Realization (Interface implementation)** | ▭▶ (dashed triangle) | Class implements an interface | Bird implements Flyable |
| **Dependency** | ──▷ (dashed line) | Temporary use of a class | ReportGenerator uses Logger |
| **Association Class** | Class connected to a relationship | Used when the association has attributes | A Membership between User and Club has role |

**🧠 Example in PlantUML**

@startuml

class Customer

class Order

class Product

Customer "1" --> "\*" Order : places

Order "1" --> "\*" Product : contains

class Team

class Player

Team o-- "11" Player : has

class House

class Room

House \*-- Room : composed of

class Animal

class Dog

Animal <|-- Dog

interface Flyable

class Bird

Bird ..|> Flyable

class Logger

class ReportGenerator

ReportGenerator ..> Logger : logs with

@enduml

**✅ Visual Symbols Summary**

| **Relationship** | **Symbol in Diagram** |
| --- | --- |
| Association | Solid line with arrow or none |
| Aggregation | Open diamond o-- |
| Composition | Filled diamond \*-- |
| Inheritance | Triangle arrow `< |
| Realization | Dashed triangle `.. |
| Dependency | Dashed line with arrow ..> |

**Task 10**represent a Package diagram

Sure! Here's how you can **represent a Package Diagram** in **UML (Unified Modeling Language)** using both a **text-based description** and a **PlantUML example**, which is widely used for generating UML diagrams.

**✅ What is a Package Diagram?**

A **package diagram** in UML is used to organize and group related classes or elements into **packages**, making complex systems more manageable. It shows **how packages depend on or relate to each other**.

**📦 Simple UML Package Diagram Structure**

+-----------------+ +------------------+

| Package A |-------->| Package B |

|-----------------| |------------------|

| ClassA1 | | ClassB1 |

| ClassA2 | | ClassB2 |

+-----------------+ +------------------+

|

v

+-----------------+

| Package C |

|-----------------|

| ClassC1 |

+-----------------+

* Solid arrow → indicates **dependency** between packages.

**🌱 PlantUML Example of a Package Diagram**

@startuml

package "User Module" {

class User

class Profile

}

package "Order Module" {

class Order

class Invoice

}

package "Product Module" {

class Product

class Category

}

' Dependencies

User --> Order : places

Order --> Product : contains

User --> Profile : has

@enduml

**📌 Explanation:**

* **Packages:** User Module, Order Module, and Product Module group related classes.
* **Arrows:** Show how one class depends on or communicates with others across packages.

**Task 11**   
what is state notation .. initial and final ?

Answer:- **✅ What is State Notation in UML?**

**State Notation** is used in a **UML State Machine Diagram** to represent the **different states** an object can be in during its lifecycle and how it **transitions** between these states.

**🔄 Key Elements of State Notation**

**1. State**

* Represents a condition or situation in the life of an object.
* Notation: A **rounded rectangle** with the state name inside.

Example:

+-------------+

| Processing |

+-------------+

**2. Initial State**

* Represents the **starting point** of the state machine.
* Notation: A **filled black circle** (●)
* Only one per diagram.

Example:

● ----> Idle

**3. Final State**

* Represents the **end** of the object’s lifecycle or state process.
* Notation: A **circle with a dot inside** (◎)
* Can be zero or multiple depending on the diagram.

Example:

Shutdown ----> ◎

**🧠 Example State Transition**

● ----> Idle

Idle ----> Processing : start()

Processing ----> Completed : finish()

Completed ----> ◎

**📘 Use Case Example**

For an **Order** object:

* Initial State: ●
* States: New, Processing, Shipped, Delivered
* Final State: ◎