Day 25

Employee ID: 201933938

Login ID: iamasif

Name: Shaik Asif

Excalidra link updated at 11.00

<https://excalidraw.com/#json=suv0yH5AB-GjV4ZQdPKo1,xOzVuTY2l6HZ08vr3p6nuQ>

Task 01:

Builder

public class Task01 {

// Product class with nested builder

static class Mobile {

private int storage;

private int memory;

// Private constructor

private Mobile() {}

public int getStorage() {

return storage;

}

public int getMemory() {

return memory;

}

@Override

public String toString() {

return "Mobile [Storage = " + storage + " GB, Memory = " + memory + " GB]";

}

// Static method to get builder

public static Builder builder() {

return new Builder();

}

// Nested static Builder class

public static class Builder {

private final Mobile mobile;

public Builder() {

this.mobile = new Mobile();

}

public Builder buildStorage(int storage) {

mobile.storage = storage;

return this;

}

public Builder buildMemory(int memory) {

mobile.memory = memory;

return this;

}

public Mobile build() {

return mobile;

}

}

}

// Main method

public static void main(String[] args) {

Mobile mobile = Mobile.builder()

.buildStorage(512)

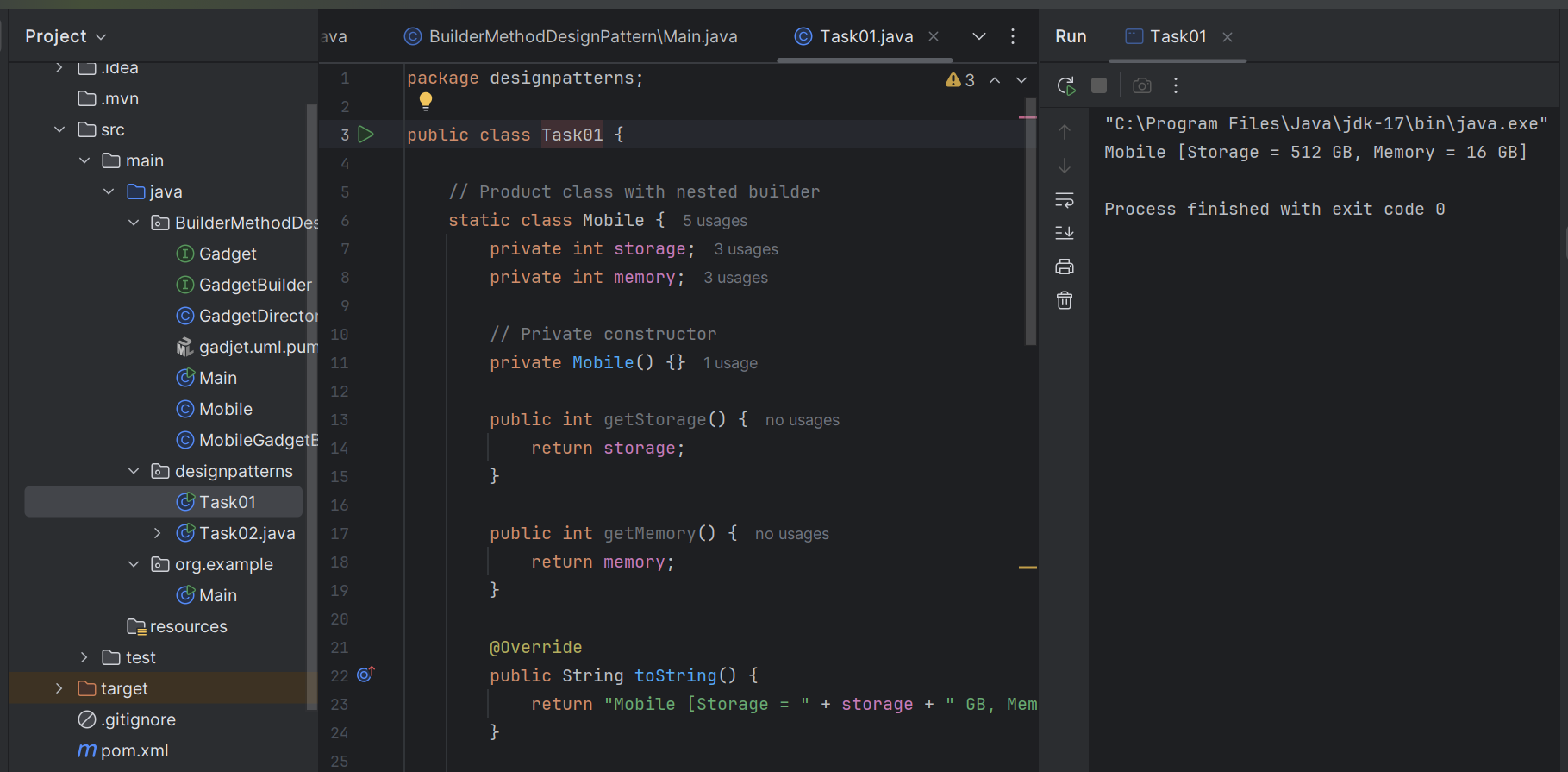
.buildMemory(16)

.build();

System.out.println(mobile);

}

}

f

Task 02:

Adaptor

public interface Iphone {

public void onCharge();

public void offCharge();

}

public class Iphone16Charger implements Charger{

Iphone16Charger() {......};

public void charge() {

sout(" i m acharging Iphone 16");

}

interface Charger {

void charge();

void removeCharge();

}

class Iphone16Adapter implements Charger {

Iphone16Charger iphone16Charger;

Iphone16Adapter() {

iphone16Charger = new Iphone16Charger();

}

@Override

void charge() {

iphone16Charger.charge();

}

@Override

void charge() {

iphone16Charger.charge();

}

}

class Iphone16 implements Iphone {

Charger Iphone16Adapter;

Iphone16( Charge iphone16Adapter) {

this.iphone16Adapter= iphone16Adapter;

}

@Override

void onChrage() {

Iphone16Adapter.charge();

}

@Override

void offChrage() {

Iphone16Adapter.removecharge();

}

}

ClientAdapterDpMain {

psvm () {

Iphone16 iphoneobj = new Iphone16(new Iphone16Adapter());

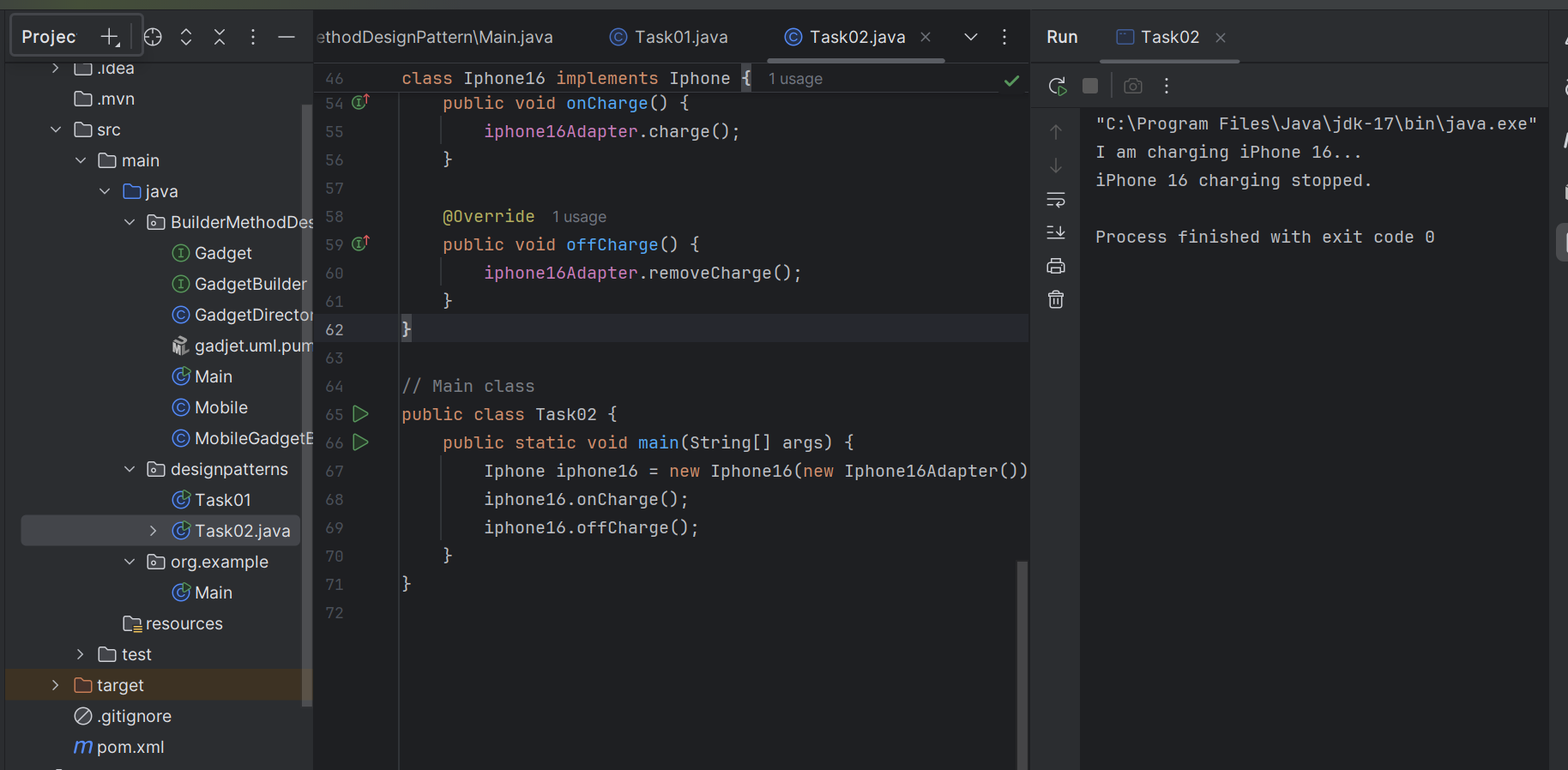
iphoneobj.onCharge();

iphoneobj.offCharge();

}

}

1.10 to 1.20



## **Gadget.java:**

package BuilderMethodDesignPattern;

public interface Gadget {

String getSpecs();

}

**Mobile.java:**

package BuilderMethodDesignPattern;

public 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 String getSpecs() {

return "Mobile Specs -> Storage: " + storage + "GB, Memory: " + memory + "GB";

}

@Override

public String toString() {

return getSpecs();

}

}

**GadgetBuilder.java:**

package BuilderMethodDesignPattern;

public interface GadgetBuilder {

GadgetBuilder buildStorage(int storage);

GadgetBuilder buildMemory(int memory);

Gadget build();

}

**MobileGadgetBuilder.java:**

package BuilderMethodDesignPattern;

public 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 Mobile build() {

return mobile;

}

}

**GadgetDirector.java:**

package BuilderMethodDesignPattern;

public class GadgetDirector {

private GadgetBuilder gadgetBuilder;

public GadgetDirector(GadgetBuilder gadgetBuilder) {

this.gadgetBuilder = gadgetBuilder;

}

public Gadget constructGadget() {

return gadgetBuilder

.buildStorage(256)

.buildMemory(12)

.build();

}

}

**Main.java:**

package BuilderMethodDesignPattern;

public class Main {

public static void main(String[] args) {

GadgetBuilder builder = new MobileGadgetBuilder();

GadgetDirector director = new GadgetDirector(builder);

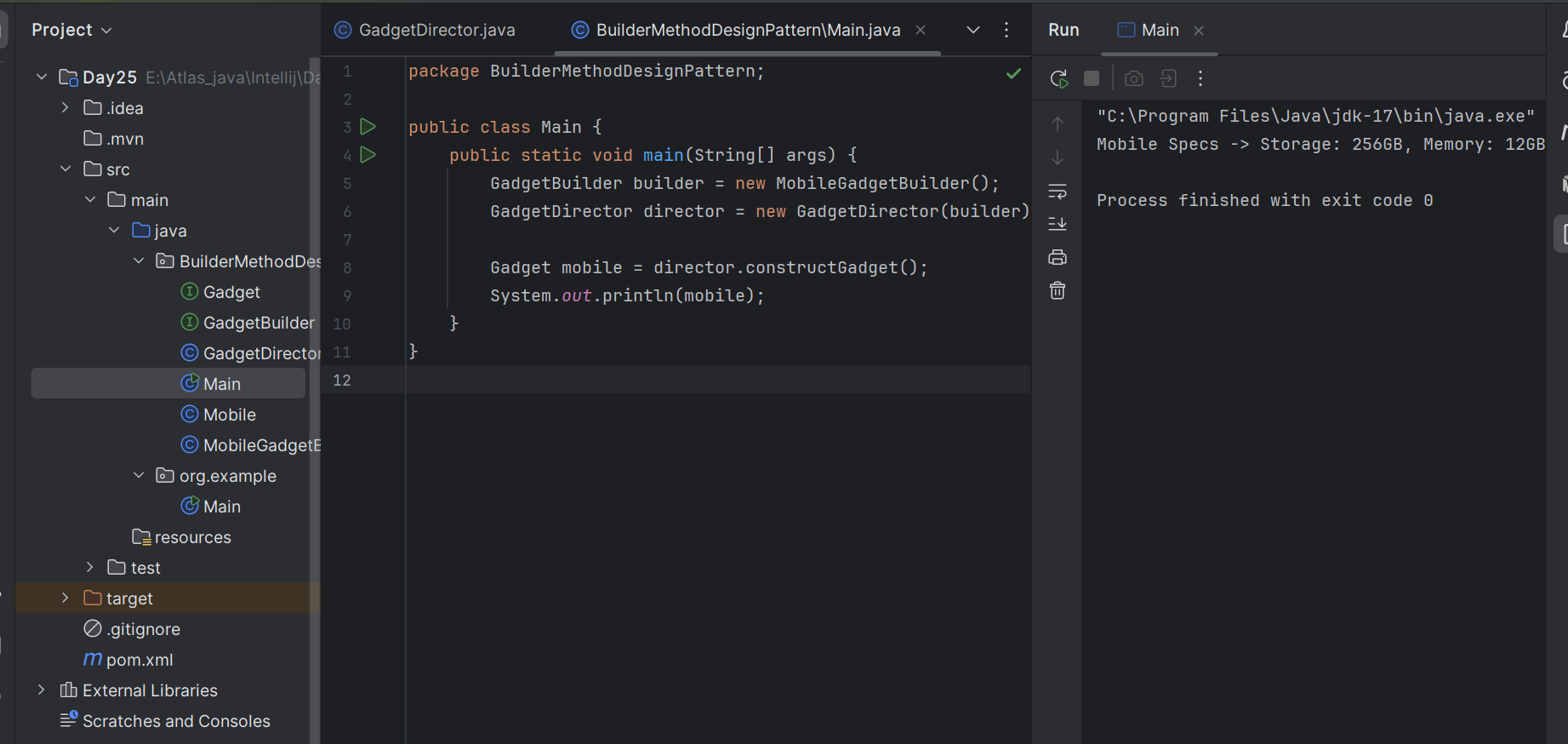
Gadget mobile = director.constructGadget();

System.out.println(mobile);

}

}

**OUTPUT:**



Excalidraw updated at 13.08

<https://excalidraw.com/#json=QURJ6oE0ck3nlsez82HVI,qiWEIN4OR56V2751XiD8TA>

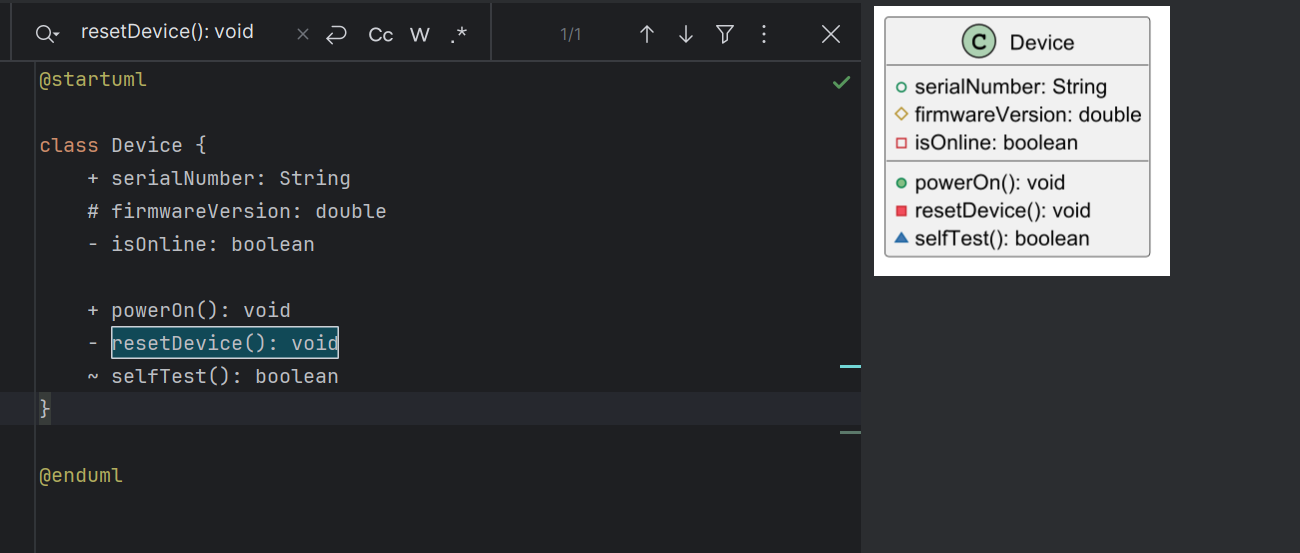
Facade 👍

Code by Vivek: need to verify

compsite:<https://codefile.io/f/2WmAlULYgy>

Task03:

Create a class diagram in uml..



Task 04:

What are the commonly used visibility Notations in class diagrams?

Eg: =,-, ~, #,

| **Symbol** | **Visibility** | **Description** | **Equivalent in Java** |
| --- | --- | --- | --- |
| + | **Public** | Accessible from anywhere | public |
| - | **Private** | Accessible only within the class | private |
| # | **Protected** | Accessible within the class and subclasses | protected |
| ~ | **Package** (default) | Accessible within the same package | *(no modifier in Java)* |

**For expl:**

@startuml

class User {

- id: int

+ name: String

# email: String

~ phone: String

+ getName(): String

- validateId(): boolean

# sendEmail(): void

~ internalLog(): void

}

@enduml

**Task 05:**

What do you know about Parameter Directionality?

List them

**Types of Parameter Directionality in UML:**

| **Direction** | **Keyword** | **Description** |
| --- | --- | --- |
| **in** | **Input** | **Data flows into the operation. The parameter is passed to the method. Default in most languages like Java.** |
| **out** | **Output** | **Data flows out of the operation. The method is expected to assign a value to this parameter.** |
| **inout** | **Input/Output** | **Data flows both into and out of the operation. The parameter is passed in, possibly modified, and returned.** |

**Example in UML Method Signature:**

+ processData(in input: int, out result: String, inout flag: boolean): void

Task 06:

Class Diagram Relationships

**UML Class Diagram Relationships:**

| **Relationship Type** | **Symbol in UML** | **Description** |
| --- | --- | --- |
| Association | — | A general “has-a” relationship (e.g., a student has a course). |
| Directed Association | → | Same as above, but with a direction. |
| Aggregation | ◇— | “Whole-part” relationship (e.g., a team has players). |
| Composition | ◆— | Stronger form of aggregation (e.g., a house contains rooms). |
| Inheritance / Generalization | ▲— | “Is-a” relationship (e.g., Dog is a type of Animal). |
| Realization (Interface implementation) | ▻— | A class implements an interface. |
| Dependency | - - - > | A temporary or “uses” relationship (e.g., a method parameter). |

## **Diagram Symbols Quick View**

* Inheritance: ChildClass --|> ParentClass
* Realization: Class ..|> Interface
* Association: Class1 --> Class2
* Aggregation: Class1 o-- Class2
* Composition: Class1 \*-- Class2
* Dependency: Class1 ..> Class2

Task 07:

Package notation

In UML, a package is used to group related classes, interfaces, or components together. It's a way to organize complex systems.

**UML Package Symbol**

* Shape: A tabbed folder
* It contains:  
  + The package name
  + Optionally, contained elements (classes, interfaces, etc.)

**Example in PlantUML Syntax**

@startuml

package "Banking System" {

class Account

class Customer

class Transaction

}

package "Utilities" {

class Logger

class Validator

}

Account --> Customer : owns

Transaction --> Account : affects

Validator --> Transaction : checks

@enduml

Task 08:

what is state notation .. initial and final ?

## **Initial State Notation**

* Symbol: Solid black circle ●
* Meaning: The starting point of the state machine; where the object begins its lifecycle.
* Example: An order process begins in the Created state.

**expl:**

**[\*] --> Created**

## **Final State Notation**

* Symbol: A black circle with a surrounding white ring ◎ (bullseye)
* Meaning: The end of the lifecycle or behavior of the object.
* Example: When the order is Delivered, it’s complete.

**expl:**

**Delivered --> [\*]**

================================

Home task:

Home task01:

Facade code:

class BankFacade {

//list of all services

private FdService fdService;

//constructor

this.fdService = new FdService();

void getfdServiceDetails(String ...) {

fdservice.getfdServiceDetails(accoundno);

}

}

**FdService.java:**

package hometask01;

public class FdService {

public void getFdServiceDetails(String accountNo) {

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

// You can add mock logic here

}

}

**BankFacade.java:**

package hometask01;

public class BankFacade {

// Subsystem class

private FdService fdService;

// Constructor initializes subsystem services

public BankFacade() {

this.fdService = new FdService();

}

// Simplified method for clients to access FD service

public void getFdDetails(String accountNo) {

fdService.getFdServiceDetails(accountNo);

}

}

**Client.java:**

package hometask01;

public class Client {

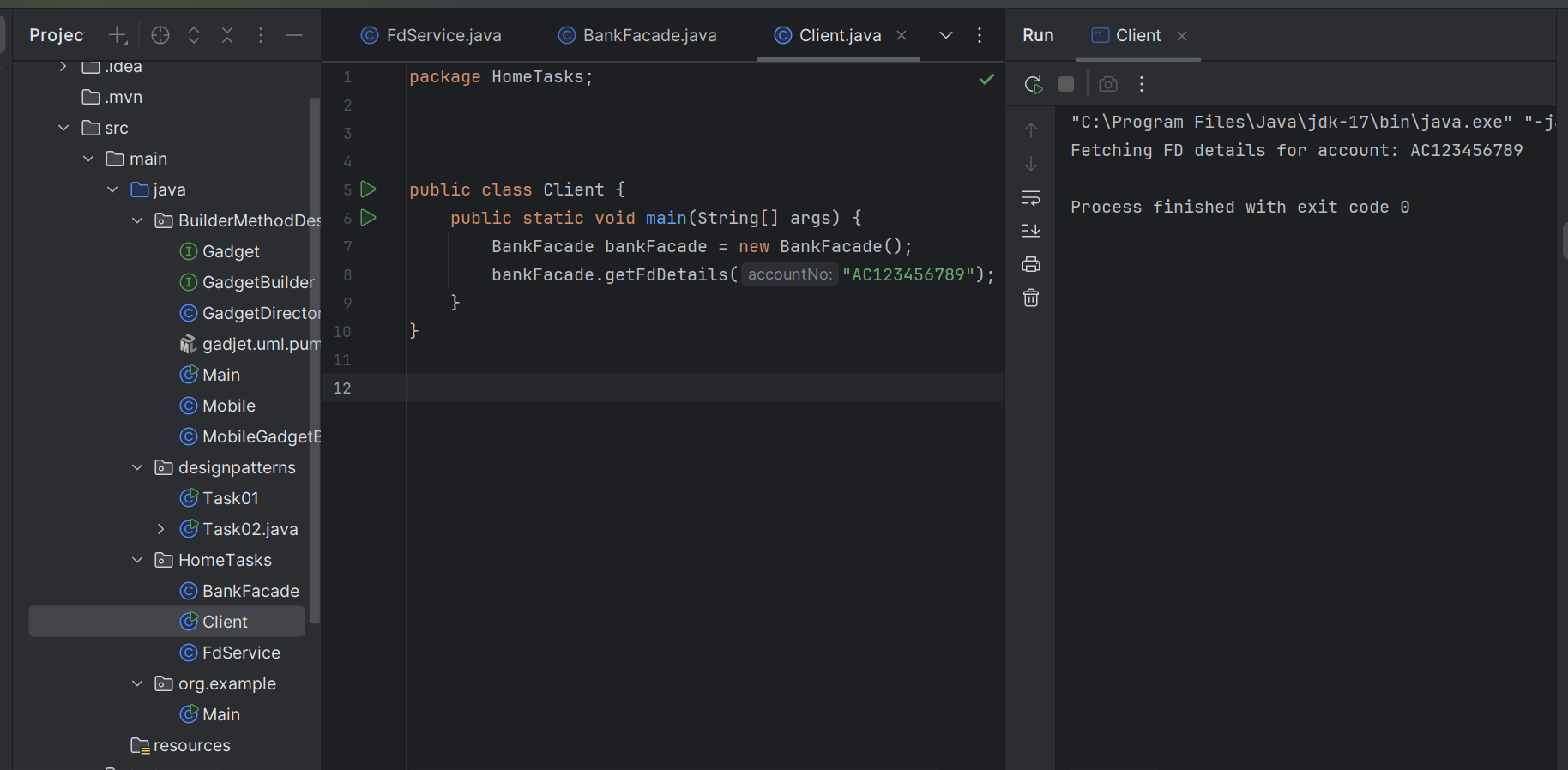
public static void main(String[] args) {

BankFacade bankFacade = new BankFacade();

bankFacade.getFdDetails("AC123456789");

}

}



Excalidraw updated at 17.08:

<https://excalidraw.com/#json=vc4FTtHNUf8rUYzdF4Kng,JQLY2aD4fRzDFyBm4yW2WA>

Home task 02:

Composite Method Design Patteren:

interface Component {

void brandName();

}

class leaf implements Component {

// declare variables

construtor

@Override

brandName()

}

class Composite implements Component {

//variable declaration

Composite() {

}

Composite(...params) { }

@Override

brandName () {

for each loop - for all components {

compobj.brandName();

}

}

add(Component subComponent) {

componets.add(subComponent);

}

}

#### **Component.java:**

package compositedemo;

public interface Component {

void brandName();

}

Leaf.java

package compositedemo;

public class Leaf implements Component {

private String name;

public Leaf(String name) {

this.name = name;

}

@Override

public void brandName() {

System.out.println("Leaf Node (Brand): " + name);

}

}

**Composite.java:**

package compositedemo;

import java.util.ArrayList;

import java.util.List;

public class Composite implements Component {

private List<Component> components = new ArrayList<>();

private String groupName;

public Composite(String groupName) {

this.groupName = groupName;

}

@Override

public void brandName() {

System.out.println("Composite Group: " + groupName);

for (Component c : components) {

c.brandName();

}

}

public void add(Component component) {

components.add(component);

}

public void remove(Component component) {

components.remove(component);

}

}

**Client.java:**

#### package compositedemo;

#### 

#### public class Client {

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

#### // Leaf nodes

#### Component samsung = new Leaf("Samsung");

#### Component sony = new Leaf("Sony");

#### Component apple = new Leaf("Apple");

#### 

#### // Composite group 1

#### Composite phoneBrands = new Composite("Mobile Brands");

#### phoneBrands.add(samsung);

#### phoneBrands.add(sony);

#### 

#### // Composite group 2

#### Composite allBrands = new Composite("All Electronics Brands");

#### allBrands.add(phoneBrands);

#### allBrands.add(apple);

#### 

#### // Run

#### allBrands.brandName();

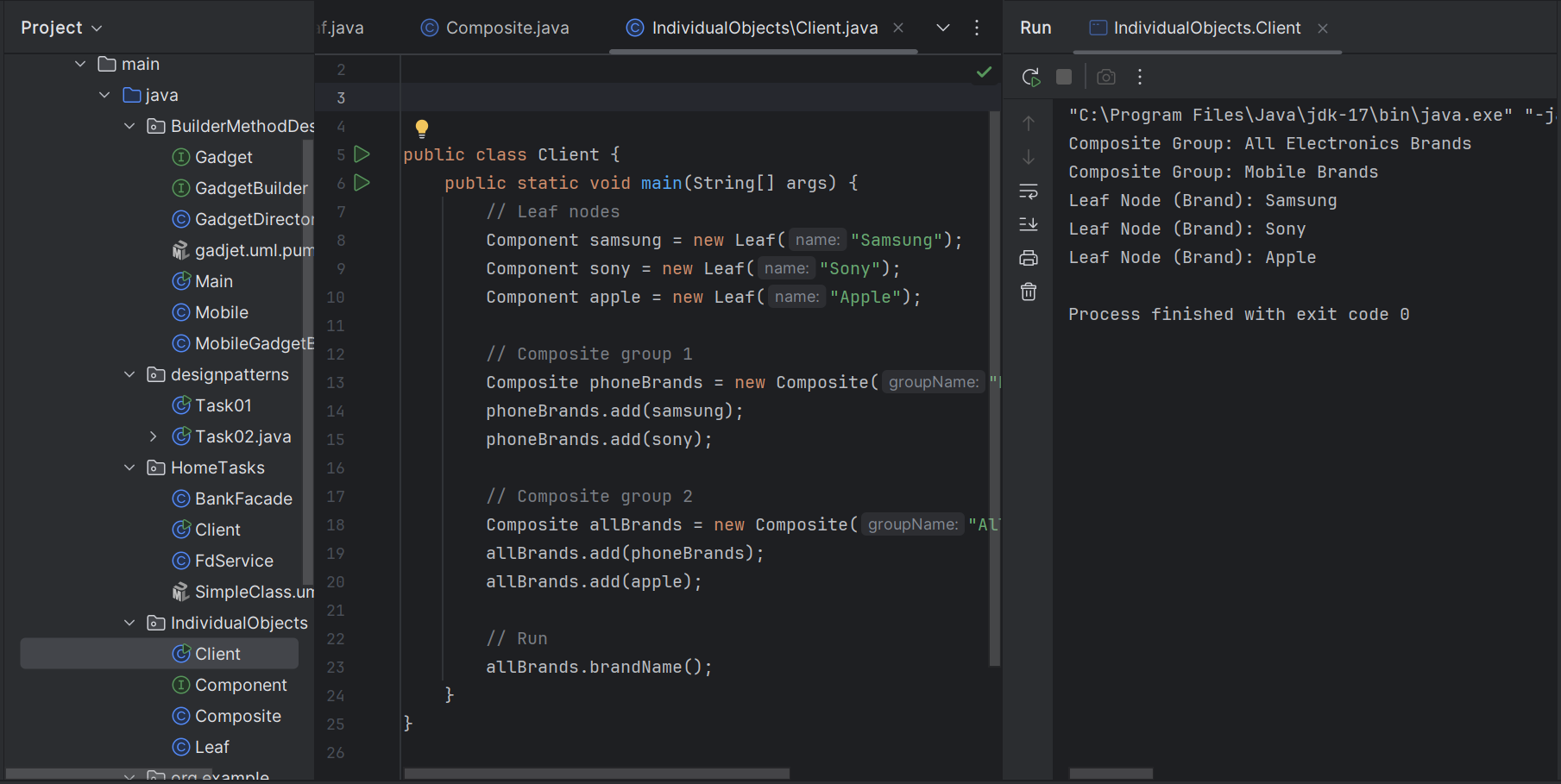
#### }

#### }

#### 

#### 

**OUTPUT:**

****

Home Task 03:

Proxy:

interface DBExecutor {

abstract void runQuery(String Type);

}

class DB{

DB(){ }

void runIt(String Type , String id) {

sout("we are running a query "+ Type);

}

}

class DBProxy implements DBExecutor {

String id;

DB db;

//constructor default

public DBProxy(String id) {

this.id = id;

db = new DB();

}

@Override

public void runQuery (String Type) {

if (Type.equals("Delete") && (!Objects.equals(this.id, "ADMIN")) {

sout("sorry access denied");

return;

}

db.runIt(Type, this.id);

}

}

class ClientProxyDP {

psvm(...) {

DBExecutor obj1 = new DBProxy("ADMIN");

obj1.runIt("READ");

obj1.runIt("UPDATE");

obj1.runIt("DELETE");

DBExecutor obj2 = new DBProxy("MANAGER");

obj2.runIt("READ");

obj2.runIt("UPDATE");

obj2.runIt("DELETE");

}

}

**DBExecutor.java:**

package proxydemo;

public interface DBExecutor {

void runQuery(String type);

}

DB.java:

package proxydemo;

public class DB {

public DB() {

// default constructor

}

public void runIt(String type, String id) {

System.out.println("[" + id + "] Running query: " + type);

}

}

**DBProxy.java:**

package proxydemo;

import java.util.Objects;

public class DBProxy implements DBExecutor {

private final String id;

private final DB db;

public DBProxy(String id) {

this.id = id;

this.db = new DB();

}

@Override

public void runQuery(String type) {

if (type.equalsIgnoreCase("DELETE") && !Objects.equals(this.id, "ADMIN")) {

System.out.println("[" + id + "] Access Denied for DELETE query.");

return;

}

db.runIt(type, this.id);

}

}

**ClientProxyDP.java:**

package proxydemo;

public class ClientProxyDP {

public static void main(String[] args) {

DBExecutor adminUser = new DBProxy("ADMIN");

adminUser.runQuery("READ");

adminUser.runQuery("UPDATE");

adminUser.runQuery("DELETE");

System.out.println("-----");

DBExecutor managerUser = new DBProxy("MANAGER");

managerUser.runQuery("READ");

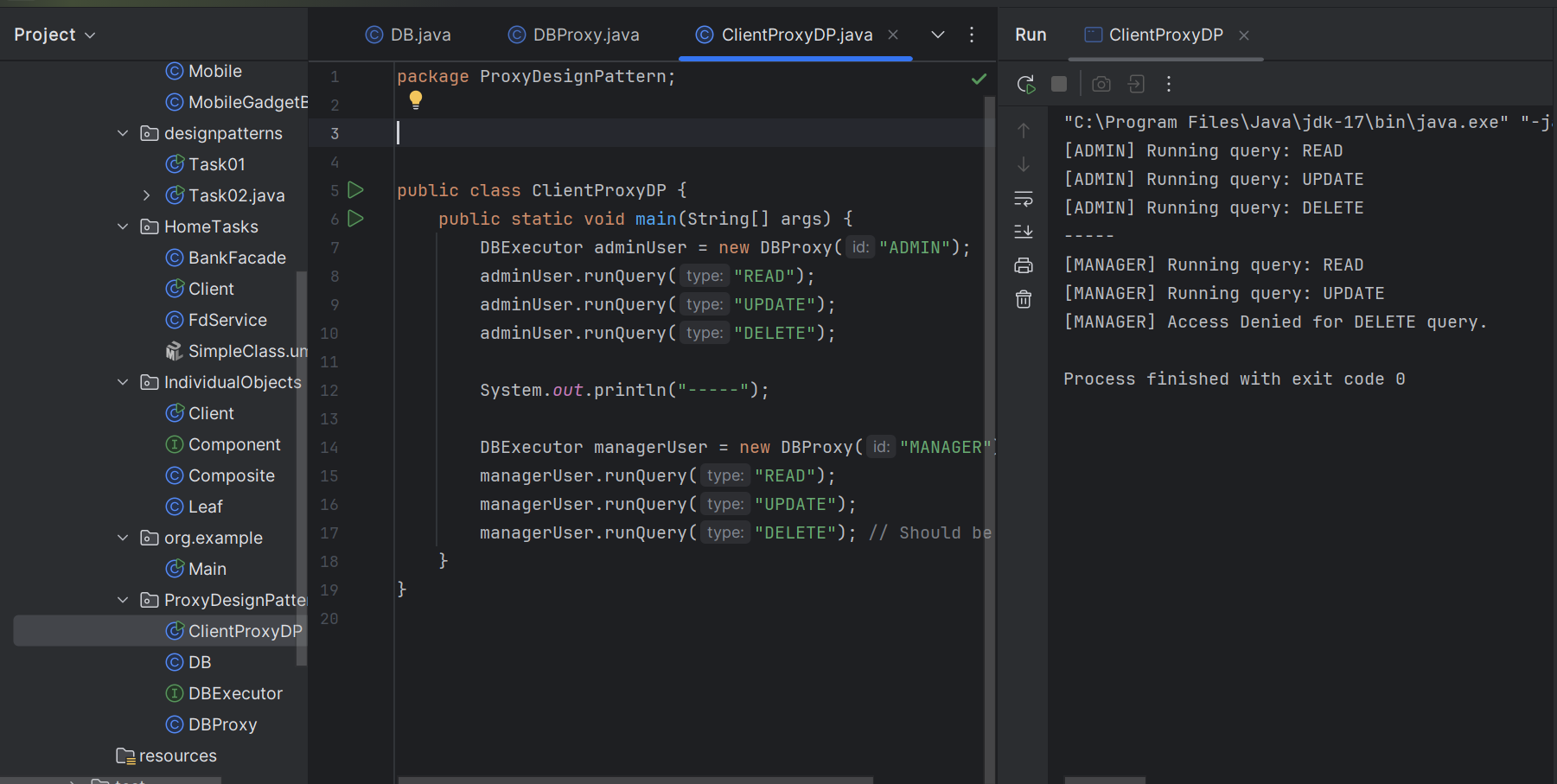
managerUser.runQuery("UPDATE");

managerUser.runQuery("DELETE"); // Should be denied

}

}

**OUTPUT:**



Hmotask 04:

Flyweight:

Flyweight is a structural design pattern used to minimize memory usage by sharing as much data as possible with similar objects.

Instead of creating a new object every time, Flyweight reuses existing ones when possible. This is especially useful when you have many similar objects (e.g., characters in a document editor).

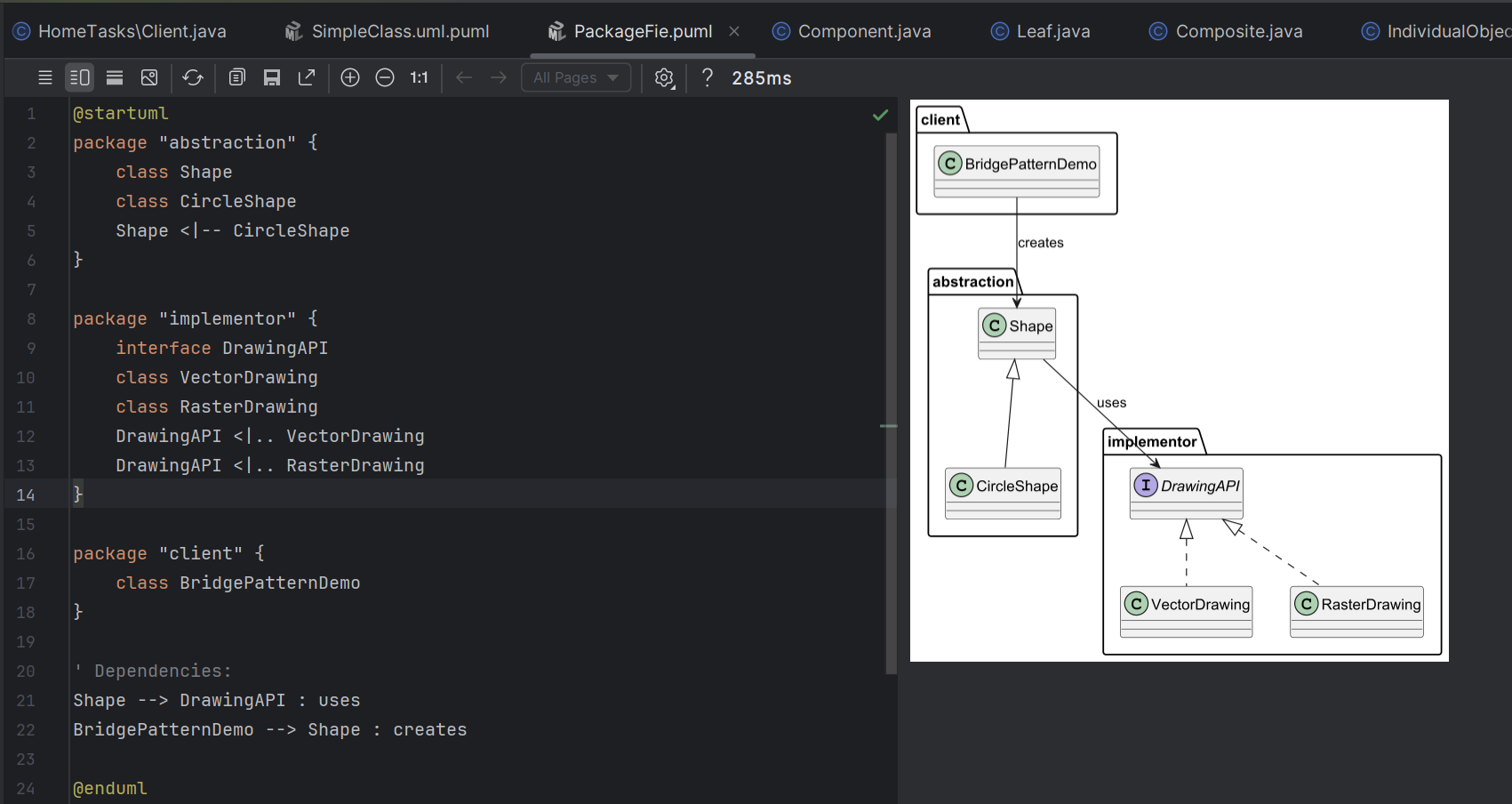
HomeTask 05:

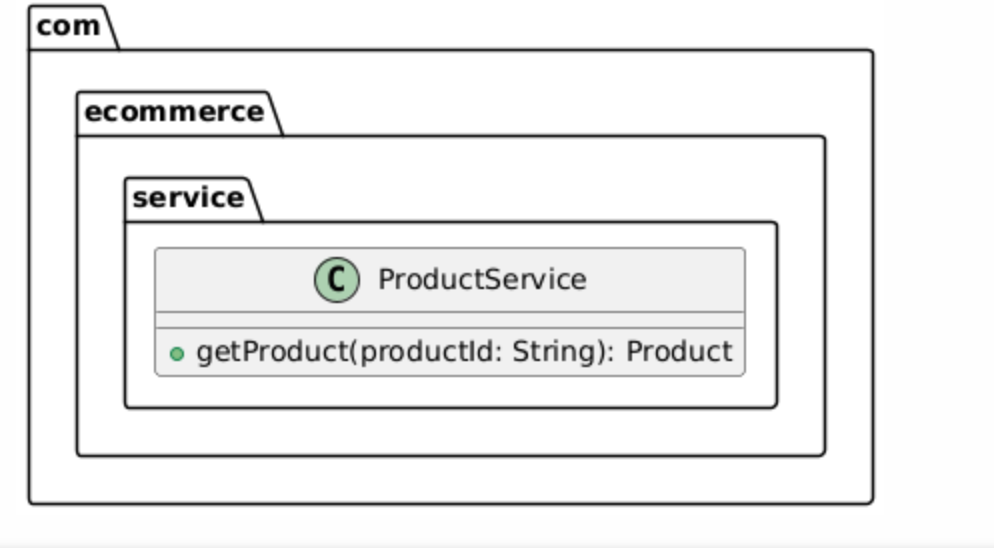
Bridge:

Bridge is a structural design pattern that **decouples an abstraction from its implementation**, allowing both to vary independently.

You use it when you want to avoid a permanent binding between an abstraction (e.g., a shape) and its implementation (e.g., drawing it with different APIs).

**Package File:**





@startuml

package com.ecommerce.service {  
 class ProductService {  
 + getProduct(productId: String): Product  
 }  
 }

@enduml

Excalidraw link updated at 11.00