Day 26

Employee ID: 201933938

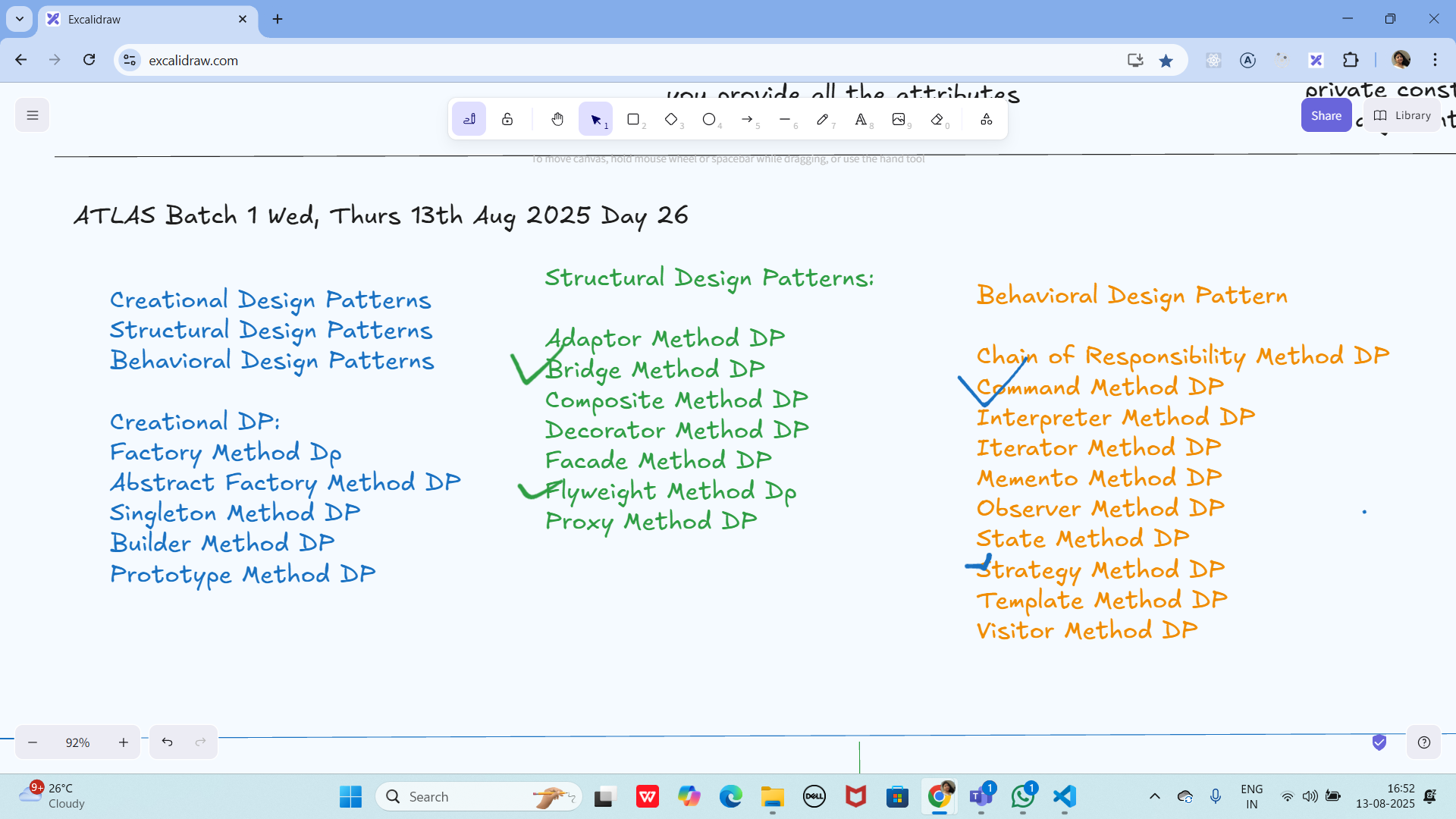
Login ID: iamasif

Name: Shaik Asif

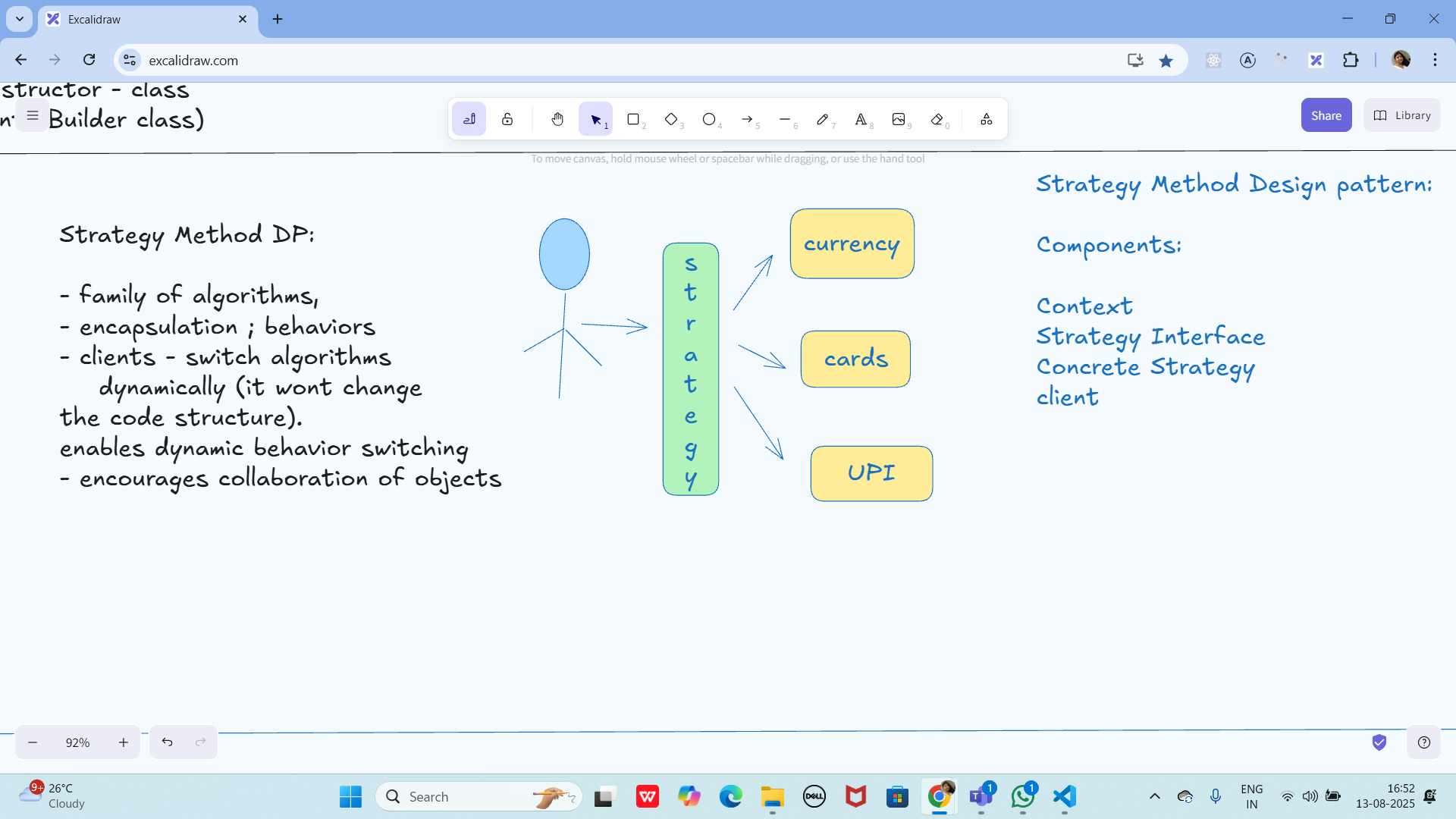
Creational

Structural DP

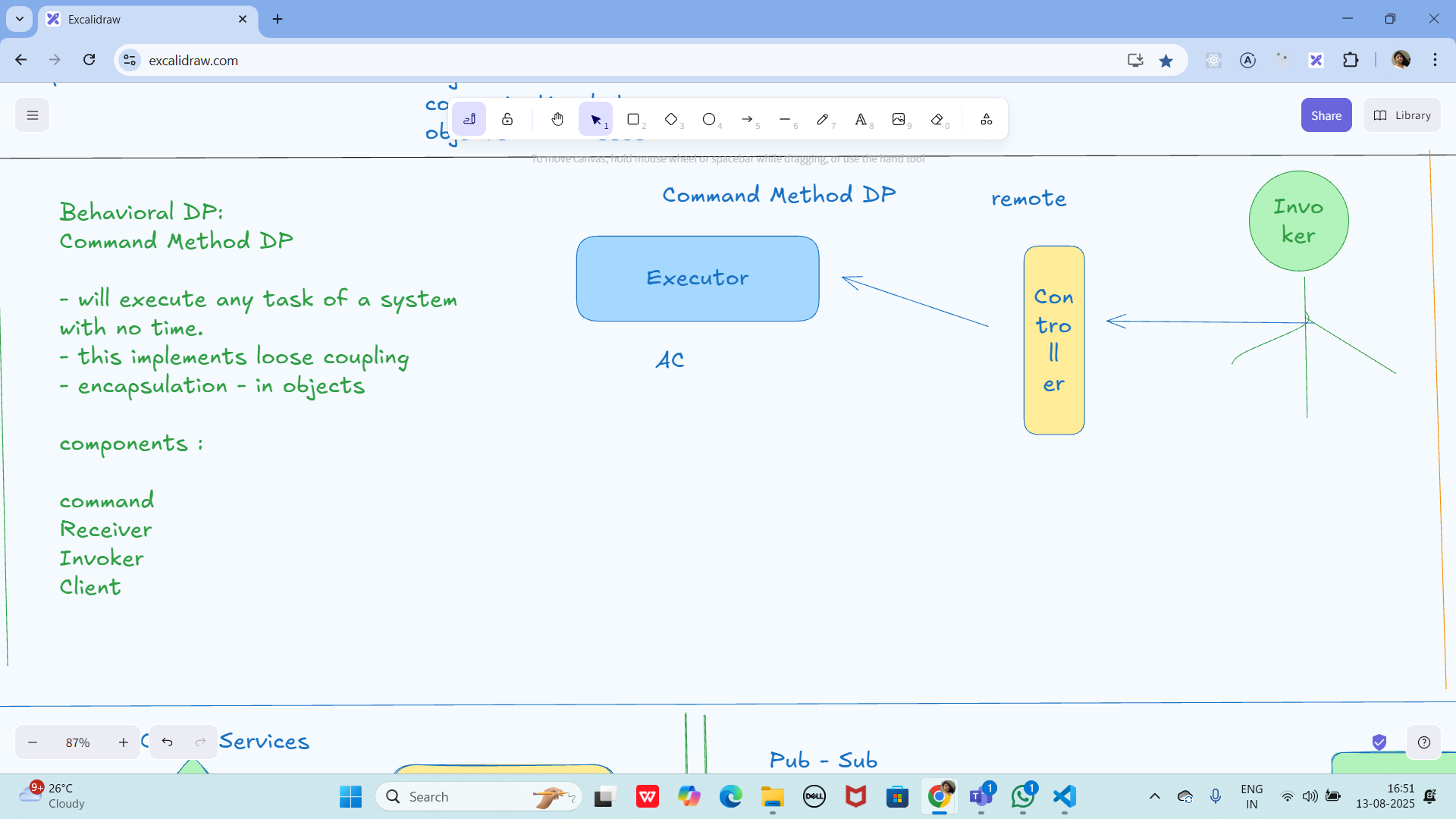
Behavioural



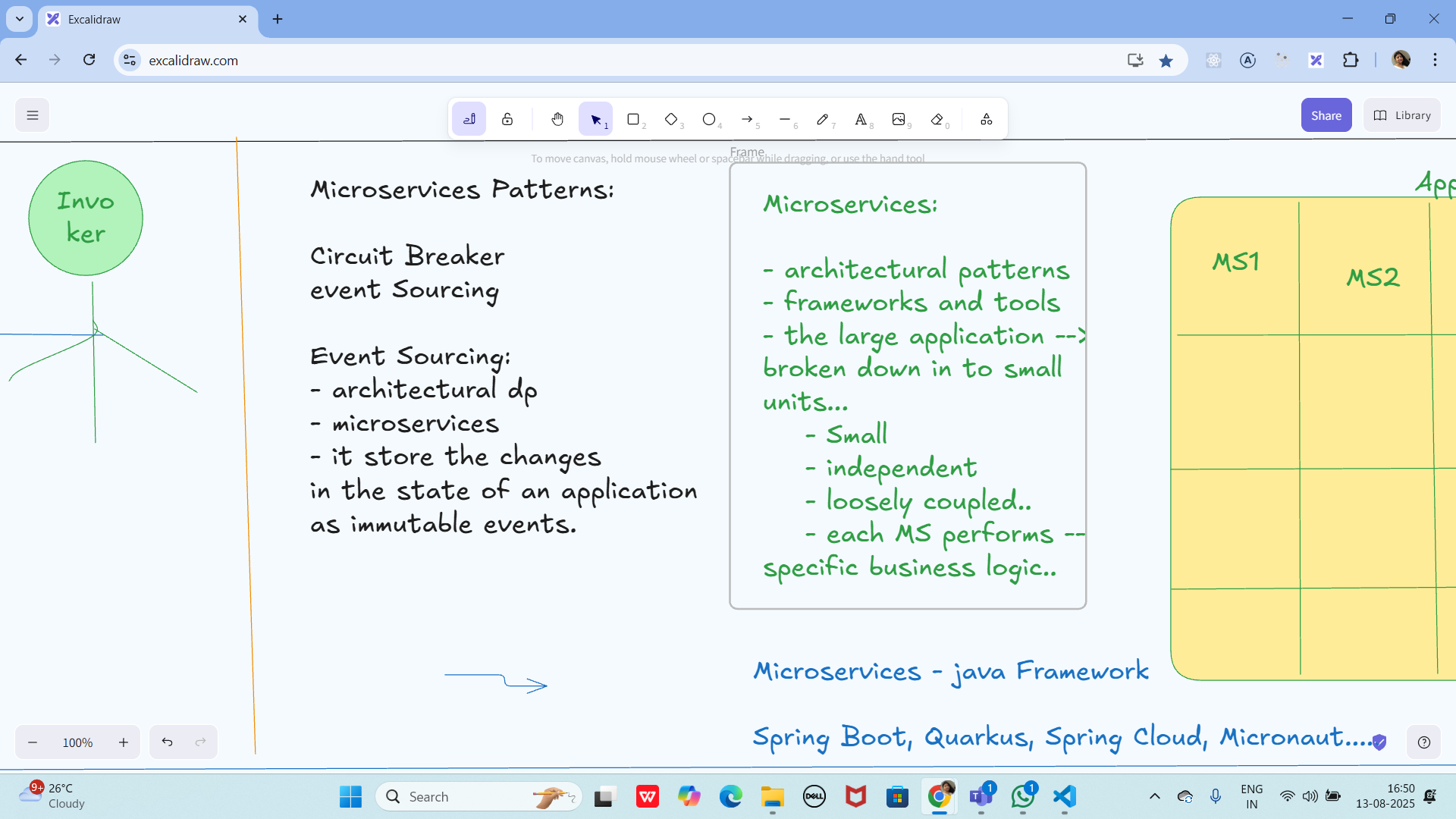
Strategy Method Dp → Bahavioral

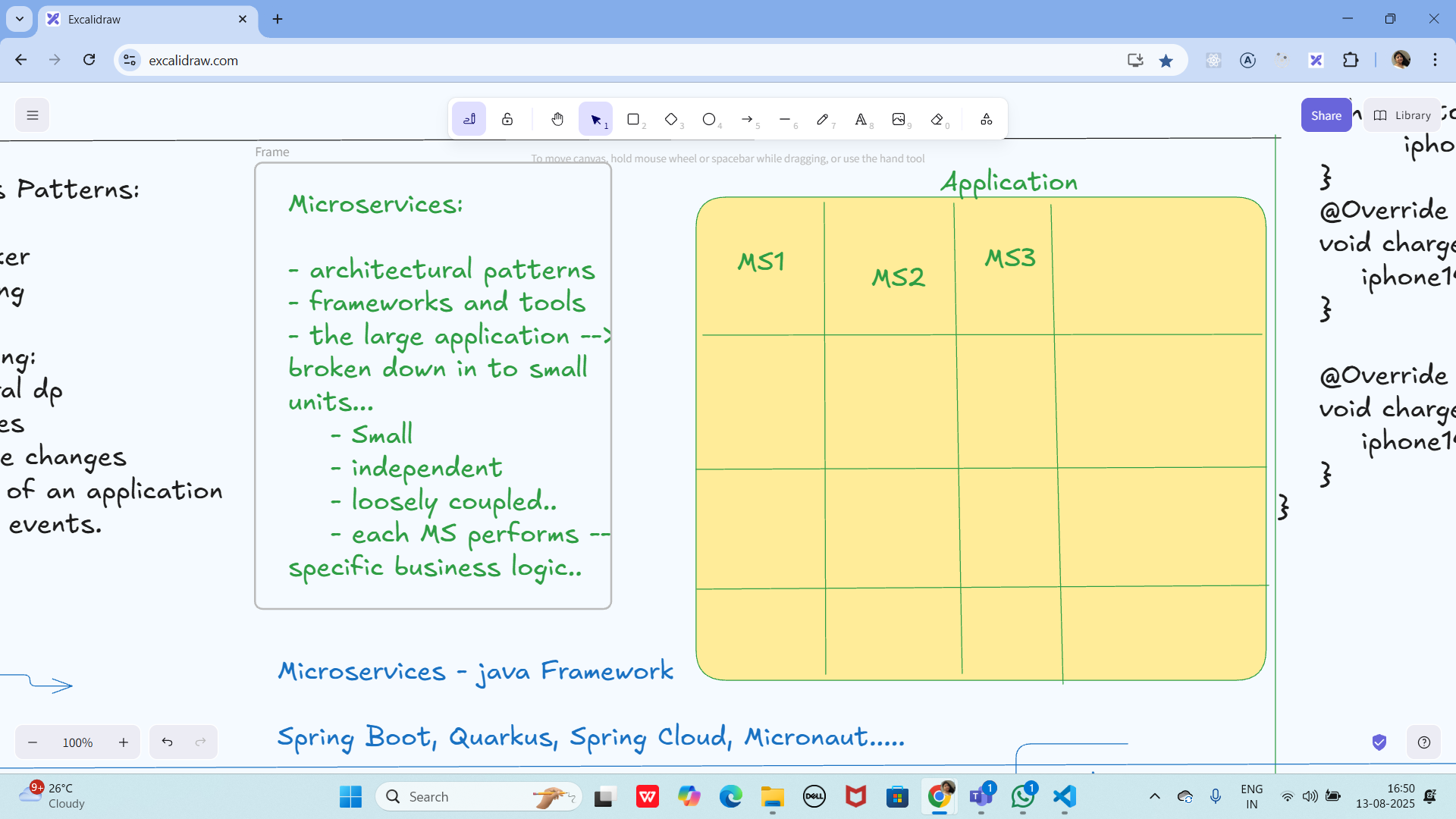


Command method DP → Behavioural

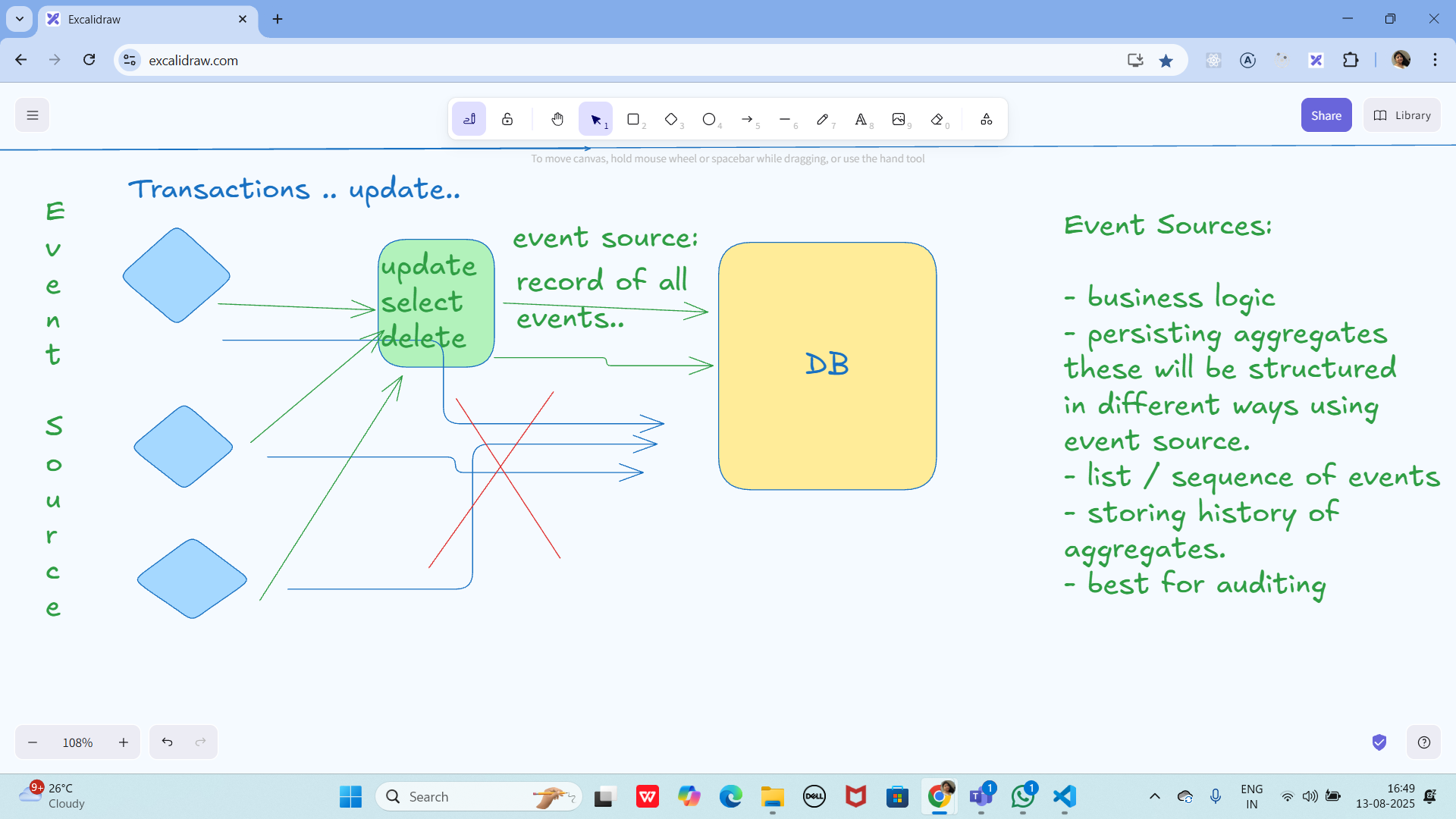


Microservices:

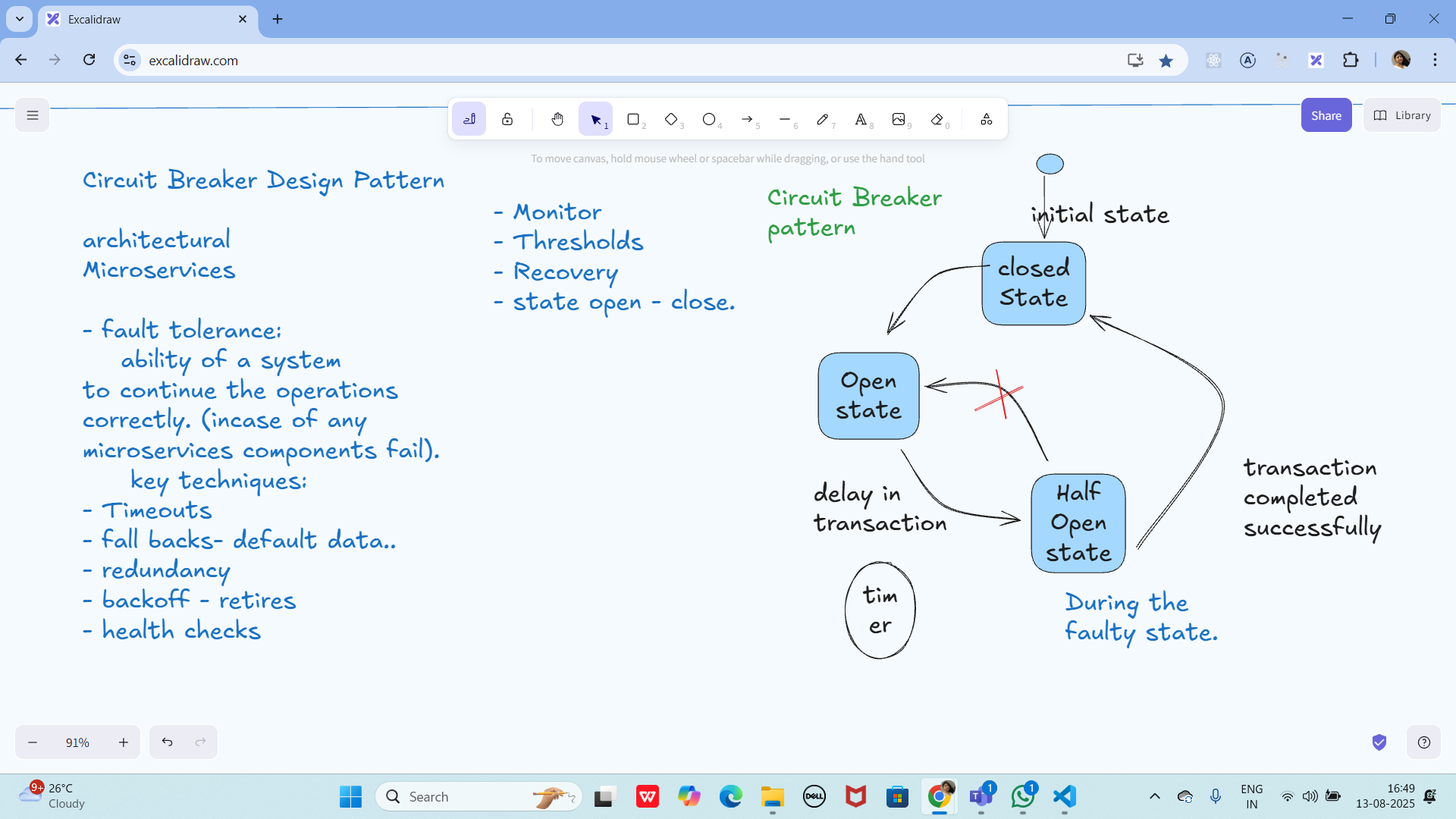


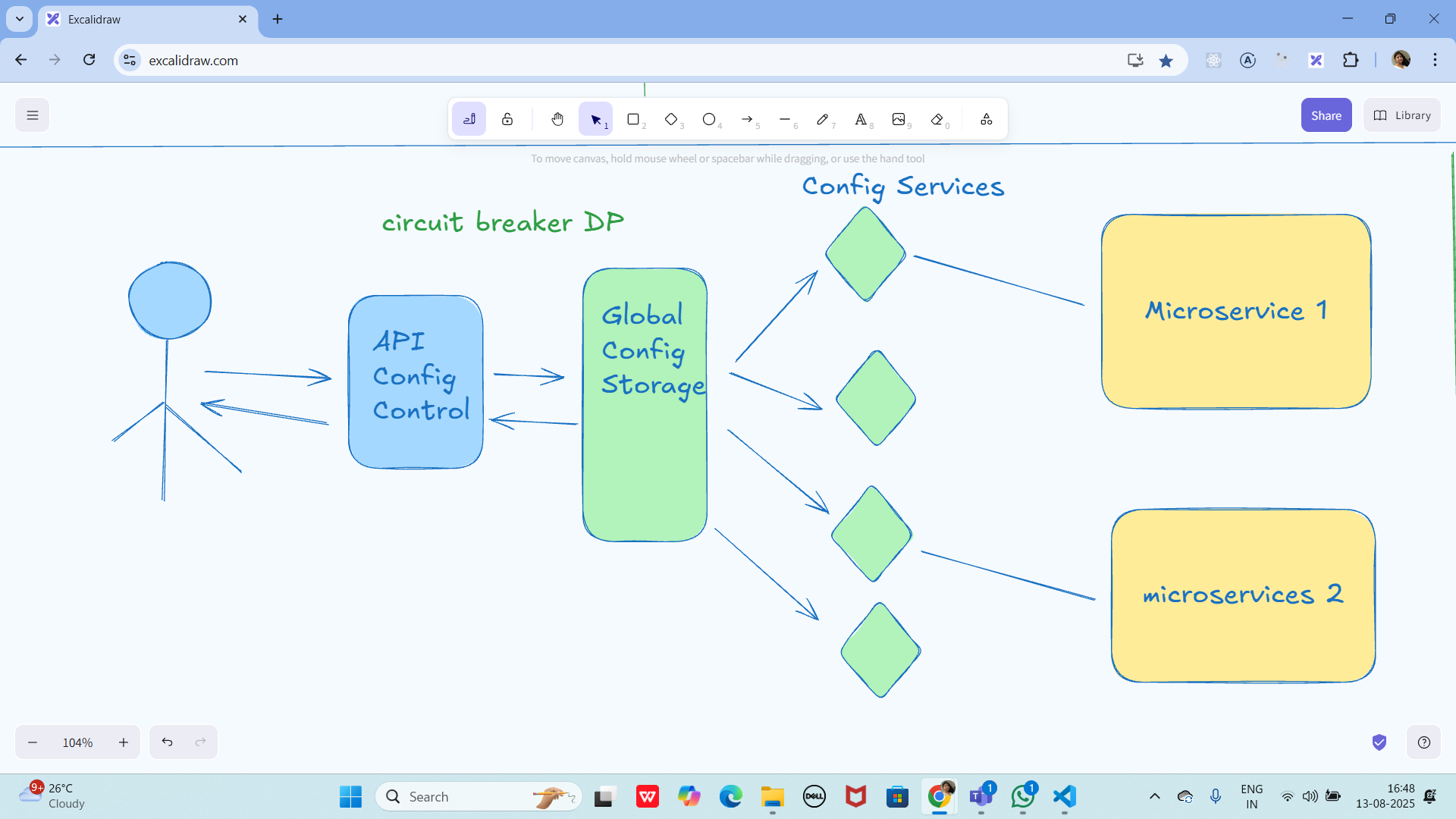


Event source Pattern

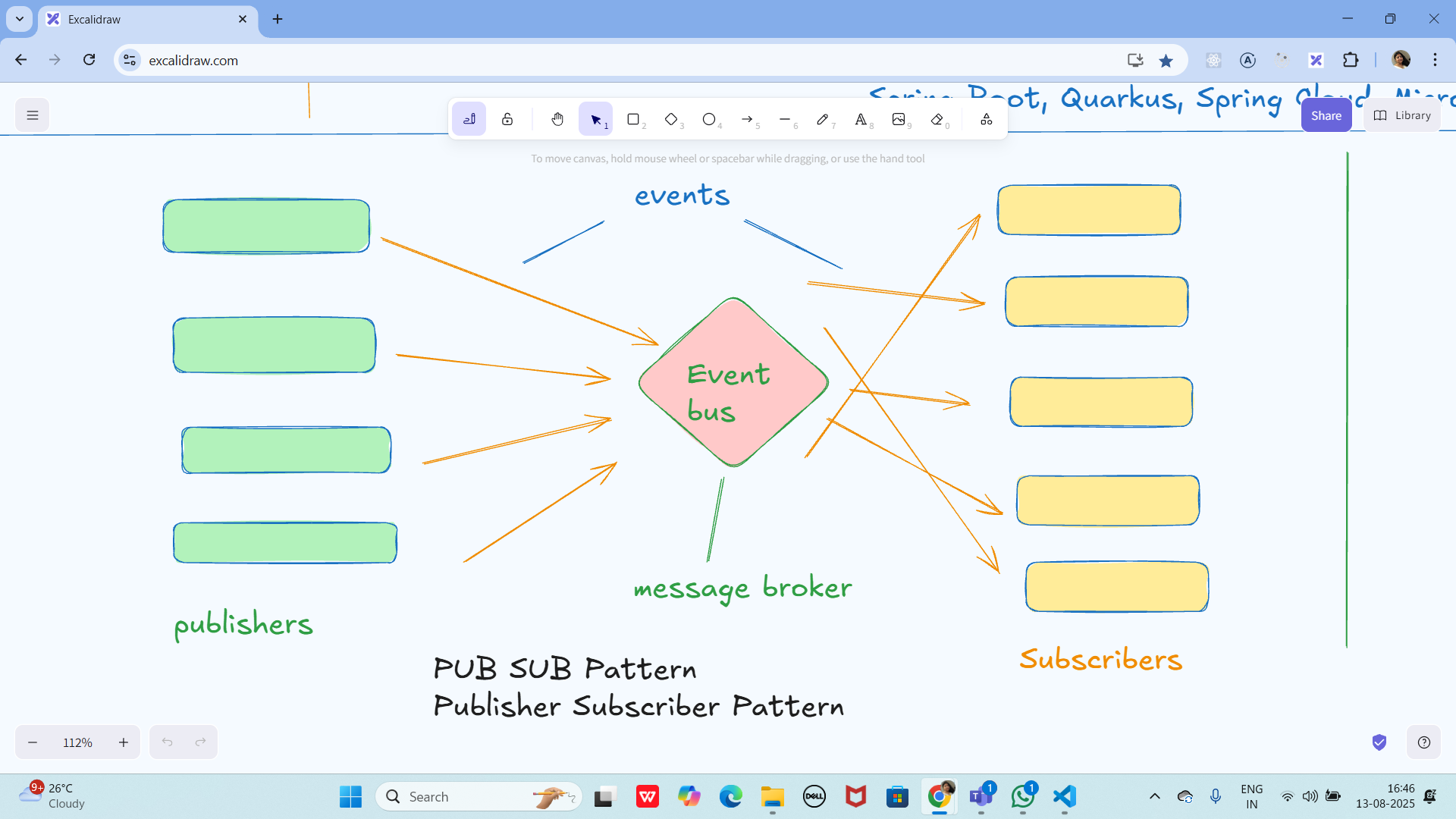


Circuit Breaker Pattern

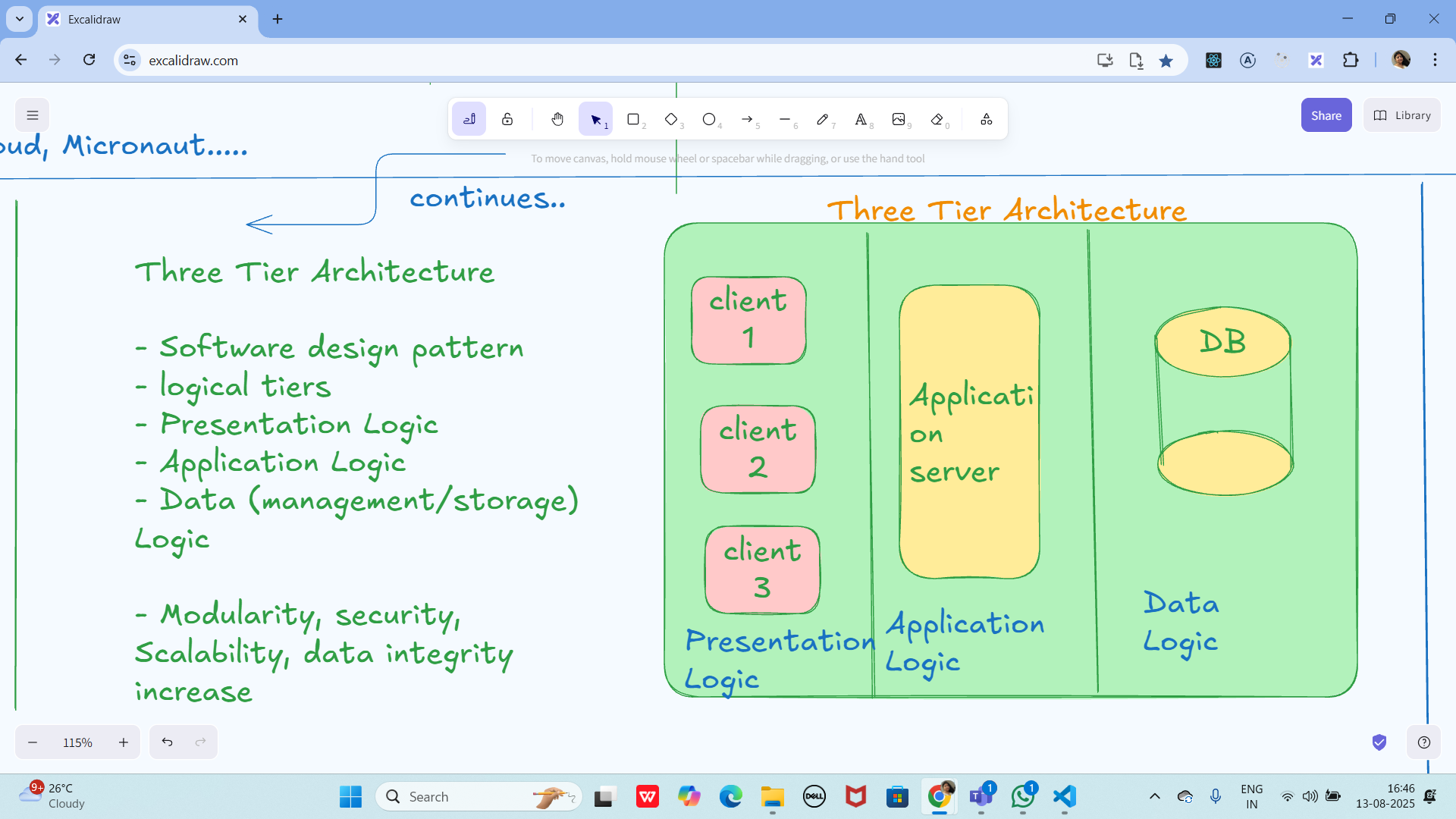
****



Pub sub Pattern:



Three tier Architecture



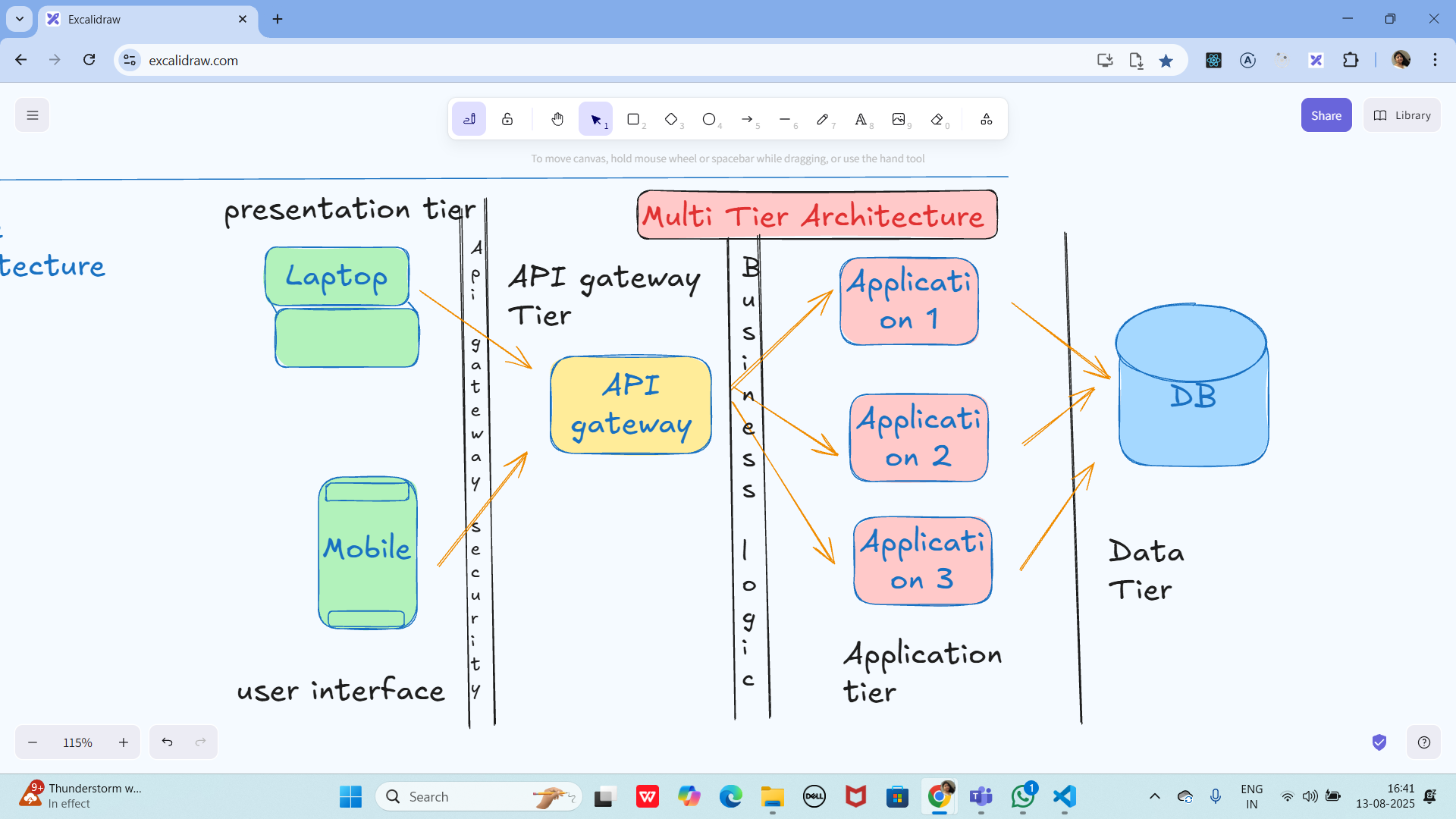
Multi tier Architecture

Multi Tier Architecture

=> client - server architecture

- improved scalability

-



**Task 01:**

What are the Advantages of Multi tier Architecture?

### **1. Separation of concerns**

Each layer (UI, business logic, data) has its own responsibility. This makes the code cleaner and easier to maintain.

### **2. Easier maintenance and updates**

You can change one tier without touching the others — for example, change the UI without rewriting the database code.

### **3. Scalability**

You can scale each tier independently. If your database is overloaded, you can add more database servers without touching the business logic or UI.

### **4. Reusability**

The middle (business) layer can be reused for different applications (e.g., both a web app and a mobile app can use the same backend).

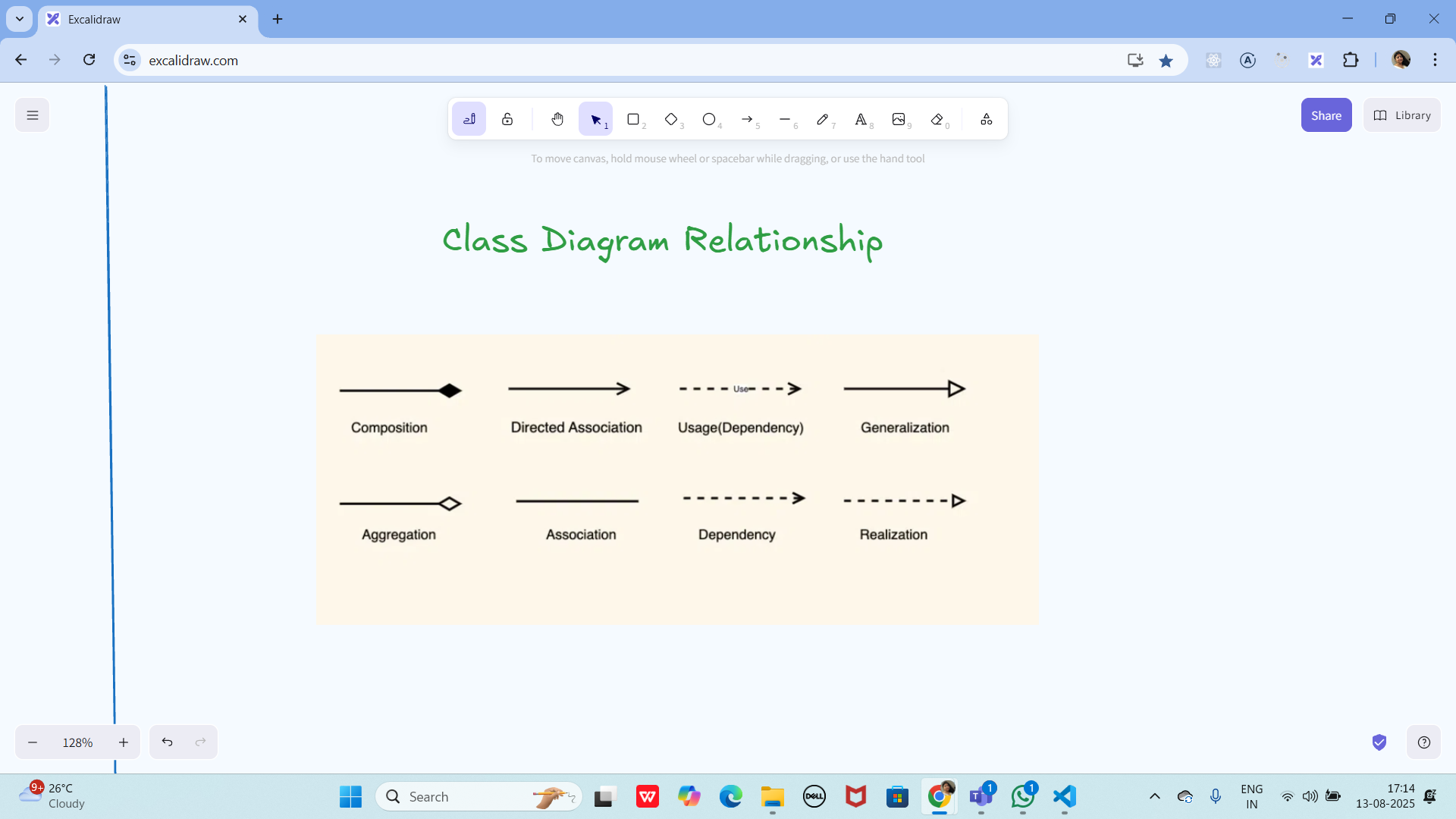
### **5. Security**

Sensitive data stays in the data tier, and users only interact with the UI tier. Each layer can have its own security rules.

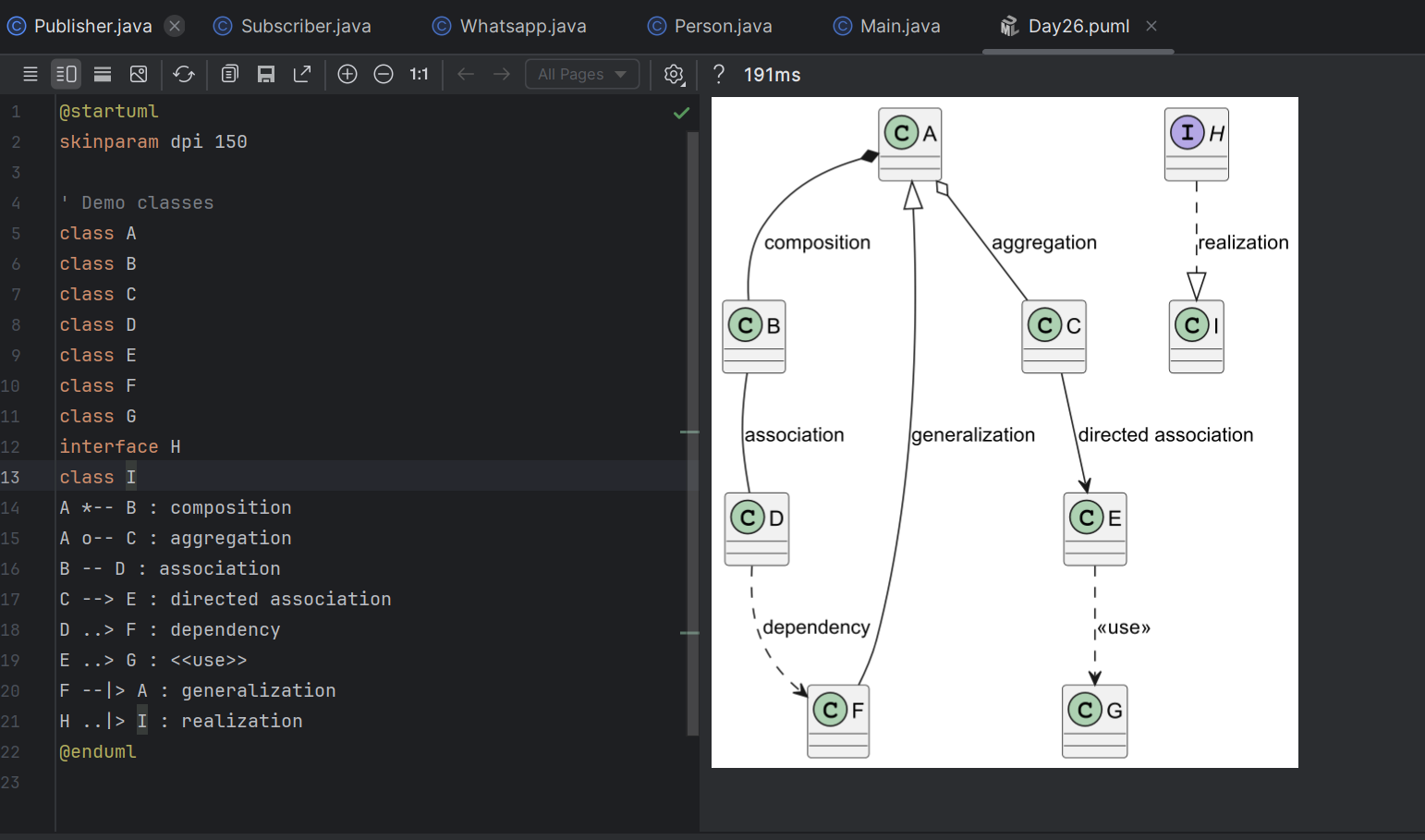
### **6. Flexibility in technology**

Different tiers can be built with different technologies — for example, React for the UI, Java Spring Boot for business logic, and MySQL for the database.

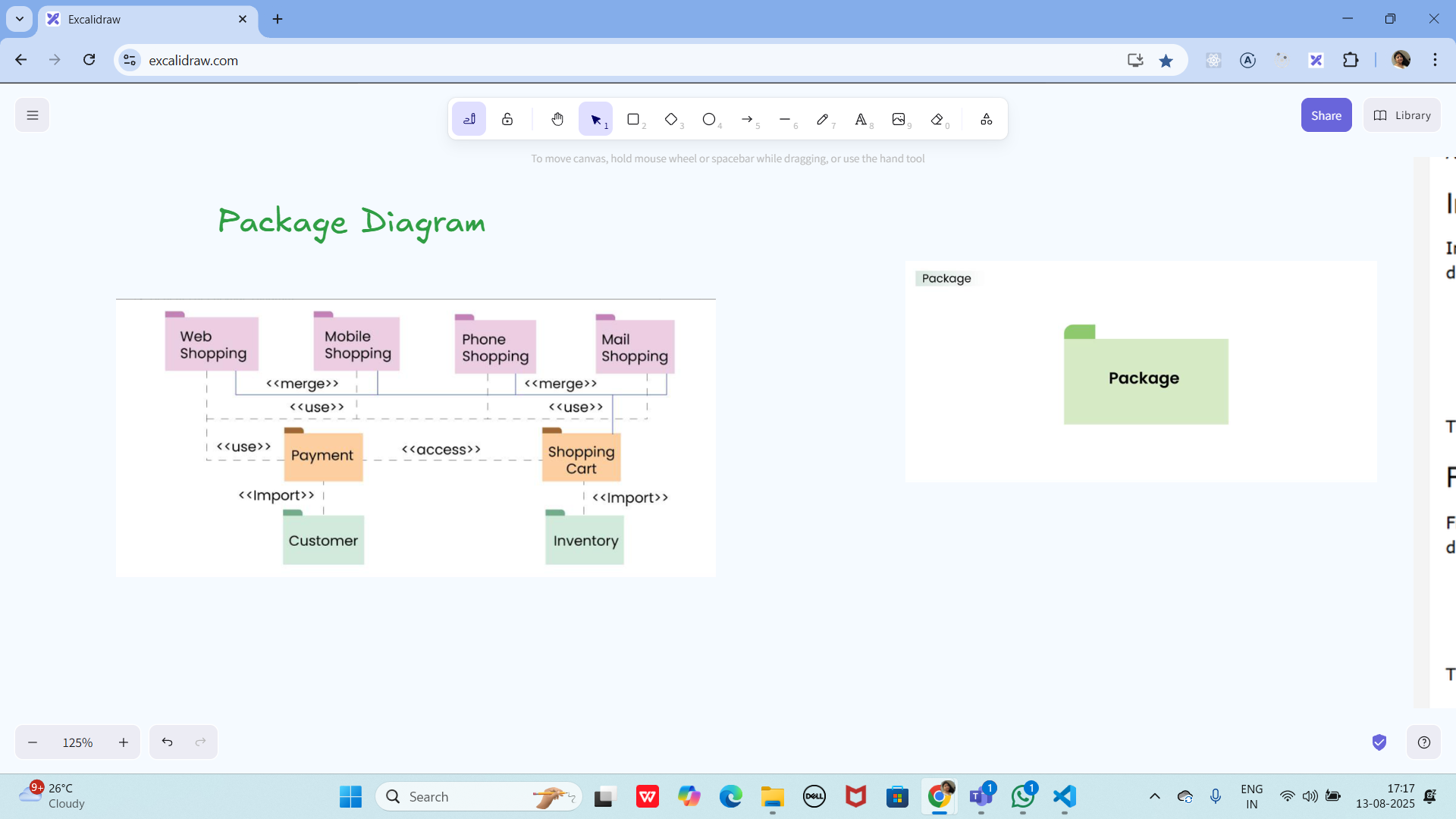
Task 03:



Create uml diagrams to represent different kinds of relation ships

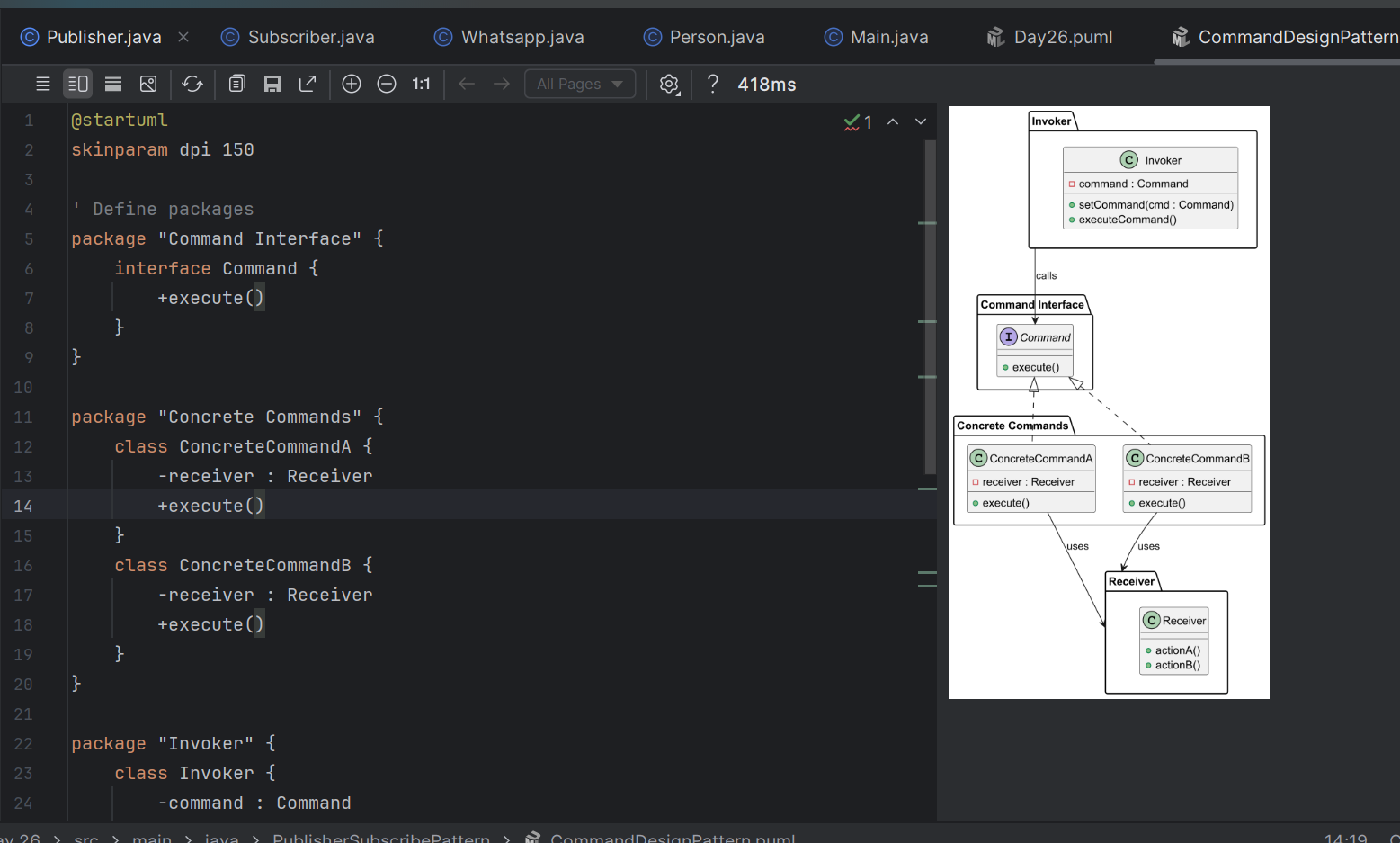


Task 04:



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

Create uml diagrams to represent different kinds of relation ships



Task 05:

Info 👍

Understand diff between

**done**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Info Box:

Codes for reference link is beloiw

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

**Code can be found in → Codes for reference → OOAD → Strategy Method DP.**

**(Command Method DP code in Codes for reference)**

Excalidraw link updated at 11.50:

<https://excalidraw.com/#json=ApYmKVfT4c0BbWs8y2EOe,UpLAw9RgWj4HgxdaMOC0xQ>

Excalidraw link updated at 13.23

<https://excalidraw.com/#json=mftMxEg5jjIXA5eVC7b-i,DVNKX2PvMsB8ZVkKc0CNuQ>

Excalidraw link updated at 16.44

<https://excalidraw.com/#json=aV6jajpxgU0LHxFWvmRy1,2HM5gvVCmTa5_wGXMTZc2A>

Exca;lidraw link updated at 18.05

<https://excalidraw.com/#json=4WkZ9xQ3_vqnQxofqOAGs,18EhyeE8wgYKq-kWm2xjmA>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Strategy Method DP:**

**PROGRAM: Payments.java**

package StratergyDesignPattern;

public class Payments {

public enum PaymentType {

*CASH*, *CARD*, *UPI*

}

private PaymentType paymentType;

public Payments(PaymentType paymentType) {

this.paymentType = paymentType;

}

public void process(double price) {

if (paymentType == PaymentType.*CASH*) {

System.*out*.println("Cash payment: " + price);

} else if (paymentType == PaymentType.*CARD*) {

System.*out*.println("Card payment: " + price);

} else if (paymentType == PaymentType.*UPI*) {

System.*out*.println("UPI payment: " + price);

} else {

System.*out*.println("Invalid payment type!");

}

}

public static void main(String[] args) {

Payments p1 = new Payments(PaymentType.*CASH*);

p1.process(100);

Payments p2 = new Payments(PaymentType.*CARD*);

p2.process(250.50);

Payments p3 = new Payments(PaymentType.*UPI*);

p3.process(75.25);

}

}

**PaymentStratergy.java:**

**package StratergyDesignPattern;**

**public interface PaymentStrategy {**

**void process(double price);**

**}**

**CashPayment.java:**

package StratergyDesignPattern;

public class CashPayment implements PaymentStrategy {

@Override

public void process(double price) {

System.*out*.println("Processing cash payment of " + price);

}

}

**CardPayment.java:**

**package StratergyDesignPattern;**

**public class CardPayment implements PaymentStrategy {**

**@Override**

**public void process(double price) {**

**System.*out*.println("Processing card payment of " + price);**

**}**

**}**

**UpiPayment.java:**

**package StratergyDesignPattern;**

**public class UpiPayment implements PaymentStrategy {**

**@Override**

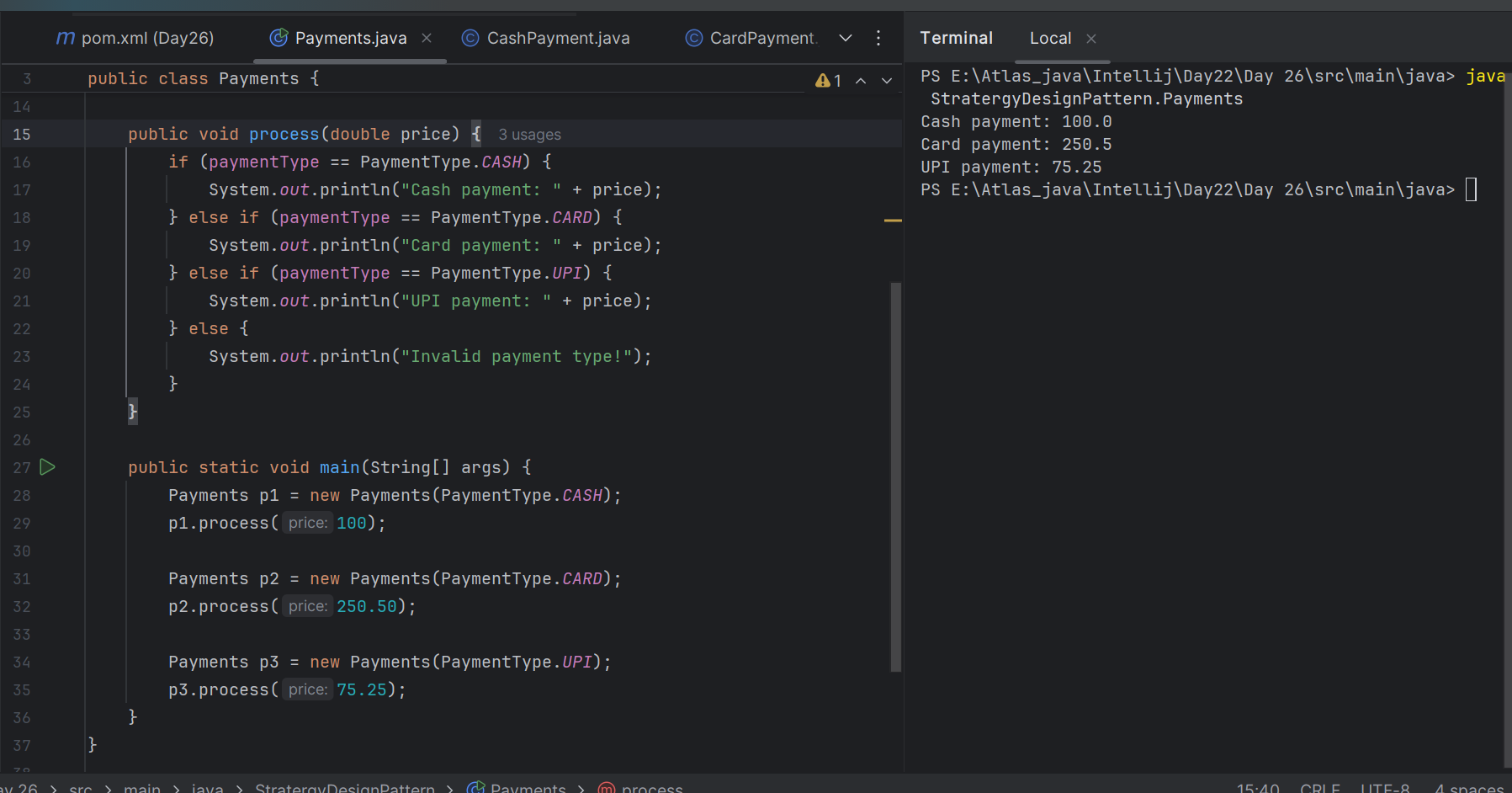
**public void process(double price) {**

**System.*out*.println("Processing UPI payment of " + price);**

**}**

**}**

**OUTPUT:**

****

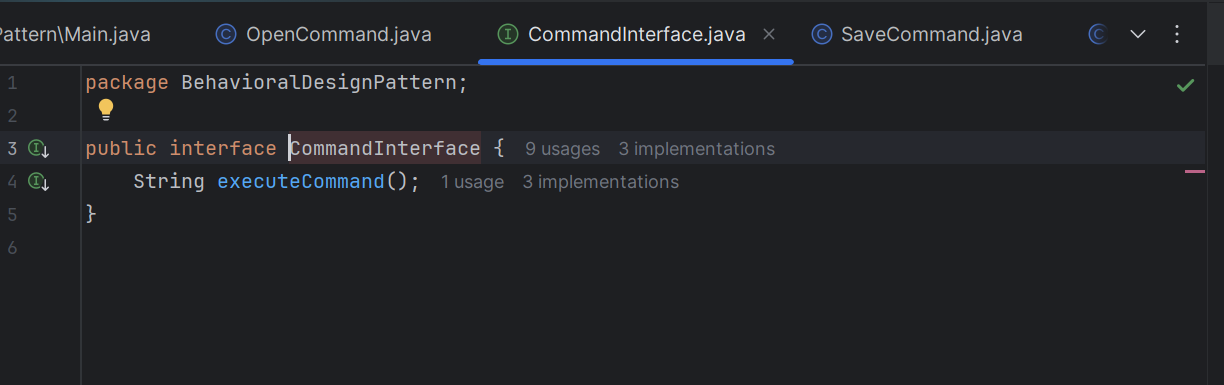
**Behavioral Design pattern:**

Behavioral patterns focus on how objects communicate and interact with each other.

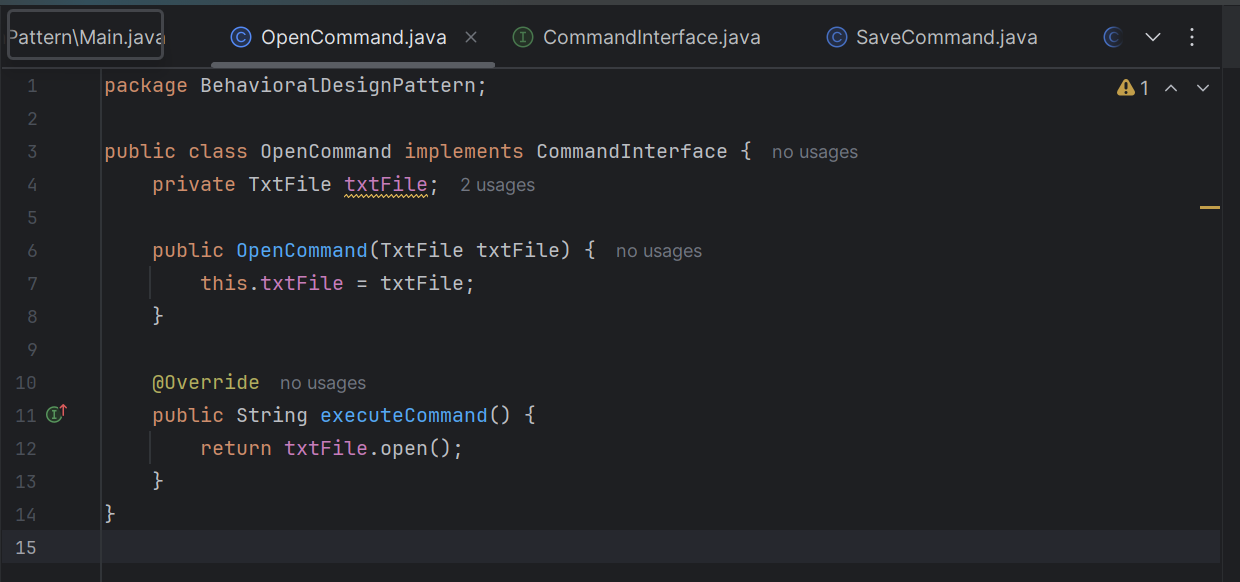
They deal with responsibility assignment and message passing between classes or objects.

**(Command Method DP code in Codes for reference)**

**CommandInterface.java:**

****

**OpenCommand.java:**

****

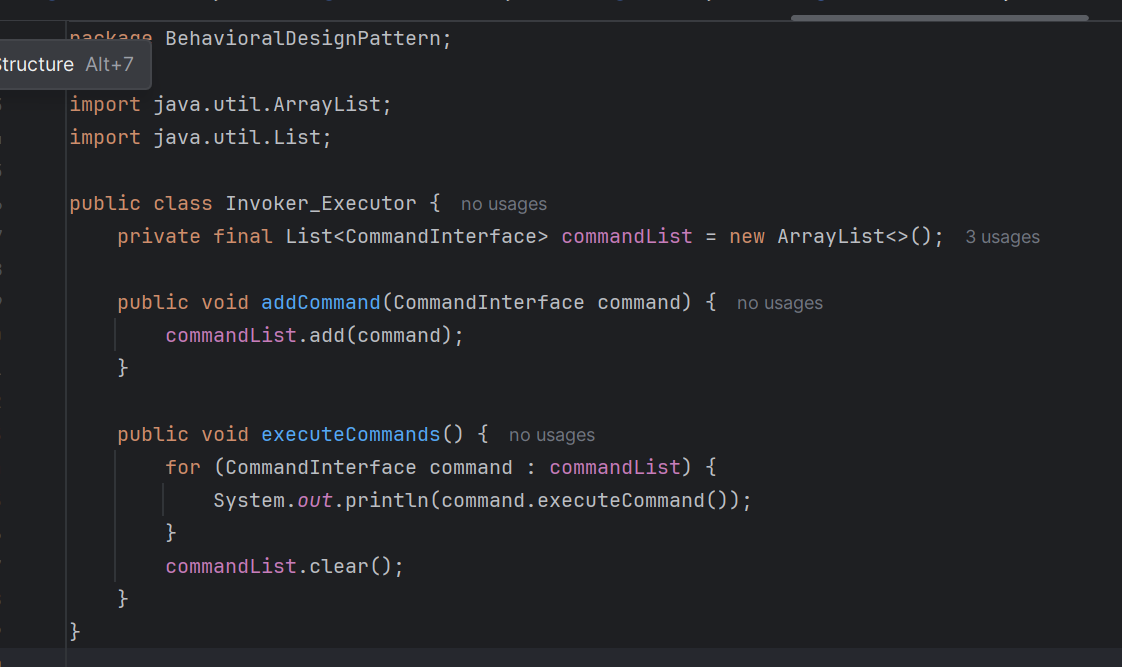
**SaveCommand.java:**

****

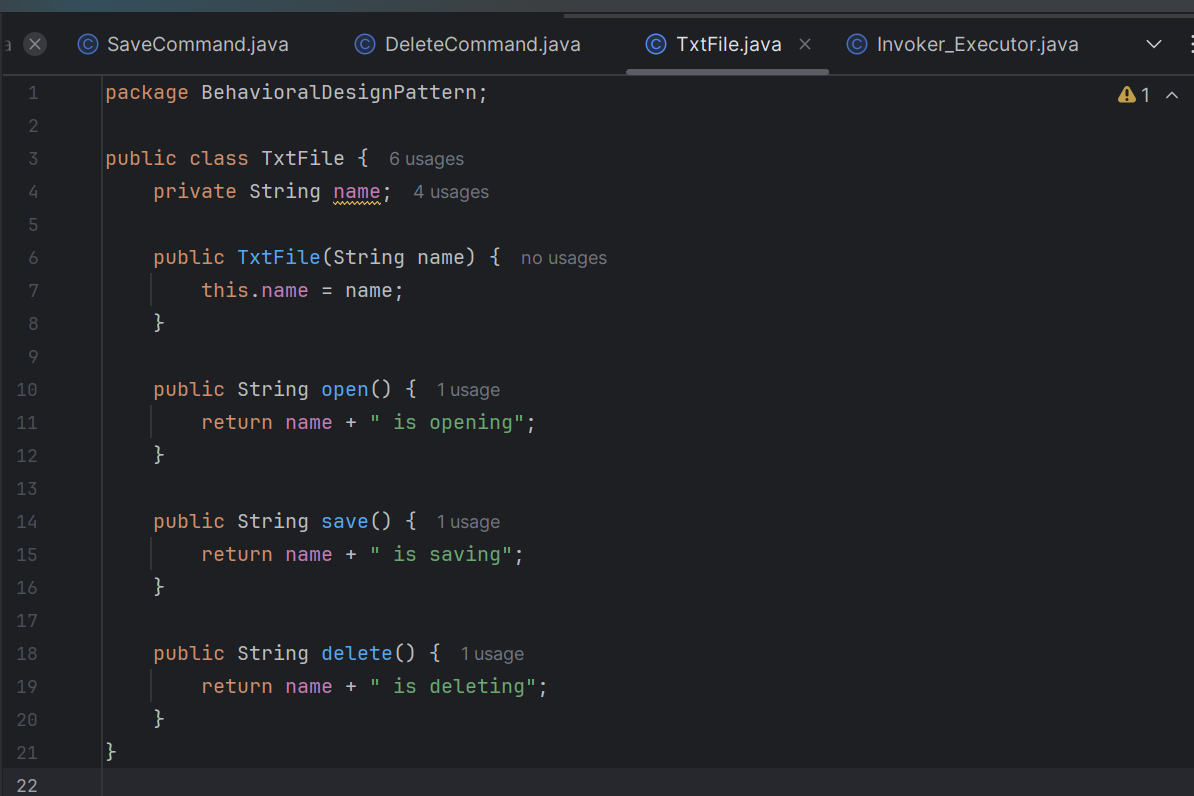
**DeleteCommand:**

****

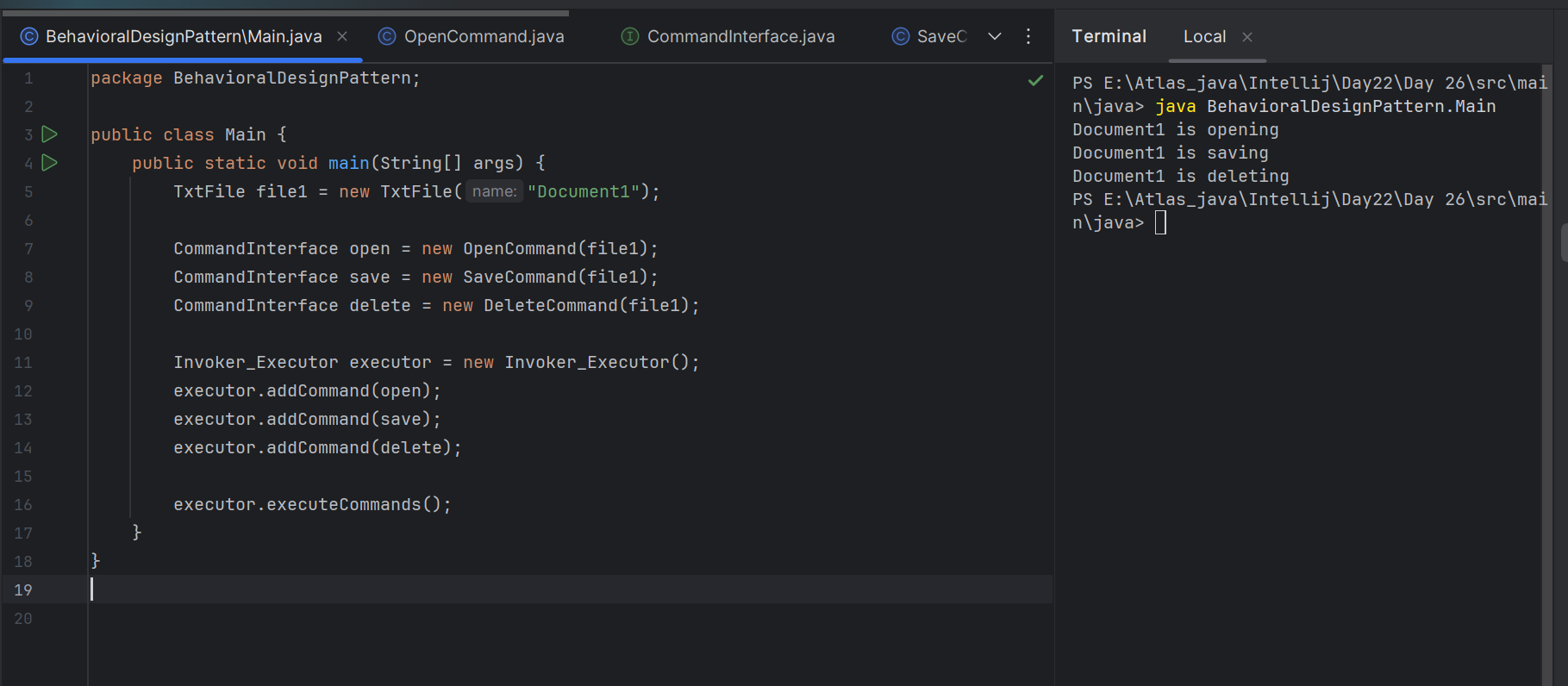
**Invoker\_Exceutor.java**

****

**TxtFile.java:**

****

**Main & outPut:**

****

**Microservices pattern:**

**PROGRAM:**

**Subscriber.java:**

**package PublisherSubscribePattern;**

**public interface Subscriber {**

**void update(String message);**

**}**

### **Publisher.java:**

package PublisherSubscribePattern;

public interface Publisher {

void subscribe(Subscriber subscriber);

void unsubscribe(Subscriber subscriber);

void notifySubscribers(String message);

}

**Whatsapp.java:**

**package PublisherSubscribePattern;**

**import java.util.ArrayList;**

**import java.util.List;**

**public class Whatsapp implements Publisher {**

**private List<Subscriber> subscribers = new ArrayList<>();**

**@Override**

**public void subscribe(Subscriber subscriber) {**

**subscribers.add(subscriber);**

**System.*out*.println("Subscriber added: " + subscriber);**

**}**

**@Override**

**public void unsubscribe(Subscriber subscriber) {**

**subscribers.remove(subscriber);**

**System.*out*.println("Subscriber removed: " + subscriber);**

**}**

**@Override**

**public void notifySubscribers(String message) {**

**for (Subscriber s : subscribers) {**

**s.update(message);**

**}**

**}**

**// Simulate sending a message in a microservices-like flow**

**public void sendMessage(String message) {**

**System.*out*.println("WhatsApp Publisher sending message: " + message);**

**notifySubscribers(message);**

**}**

**}**

**Person.java:**

**package PublisherSubscribePattern;**

**public class Person implements Subscriber {**

**private String name;**

**public Person(String name) {**

**this.name = name;**

**}**

**@Override**

**public void update(String message) {**

**System.*out*.println(name + " received: " + message);**

**}**

**@Override**

**public String toString() {**

**return name;**

**}**

**}**

**Main.java:**

**package PublisherSubscribePattern;**

**public class Main {**

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

**Whatsapp whatsappPublisher = new Whatsapp();**

**Person p1 = new Person("Alice");**

**Person p2 = new Person("Bob");**

**whatsappPublisher.subscribe(p1);**

**whatsappPublisher.subscribe(p2);**

**whatsappPublisher.sendMessage("Hello from WhatsApp Microservice!");**

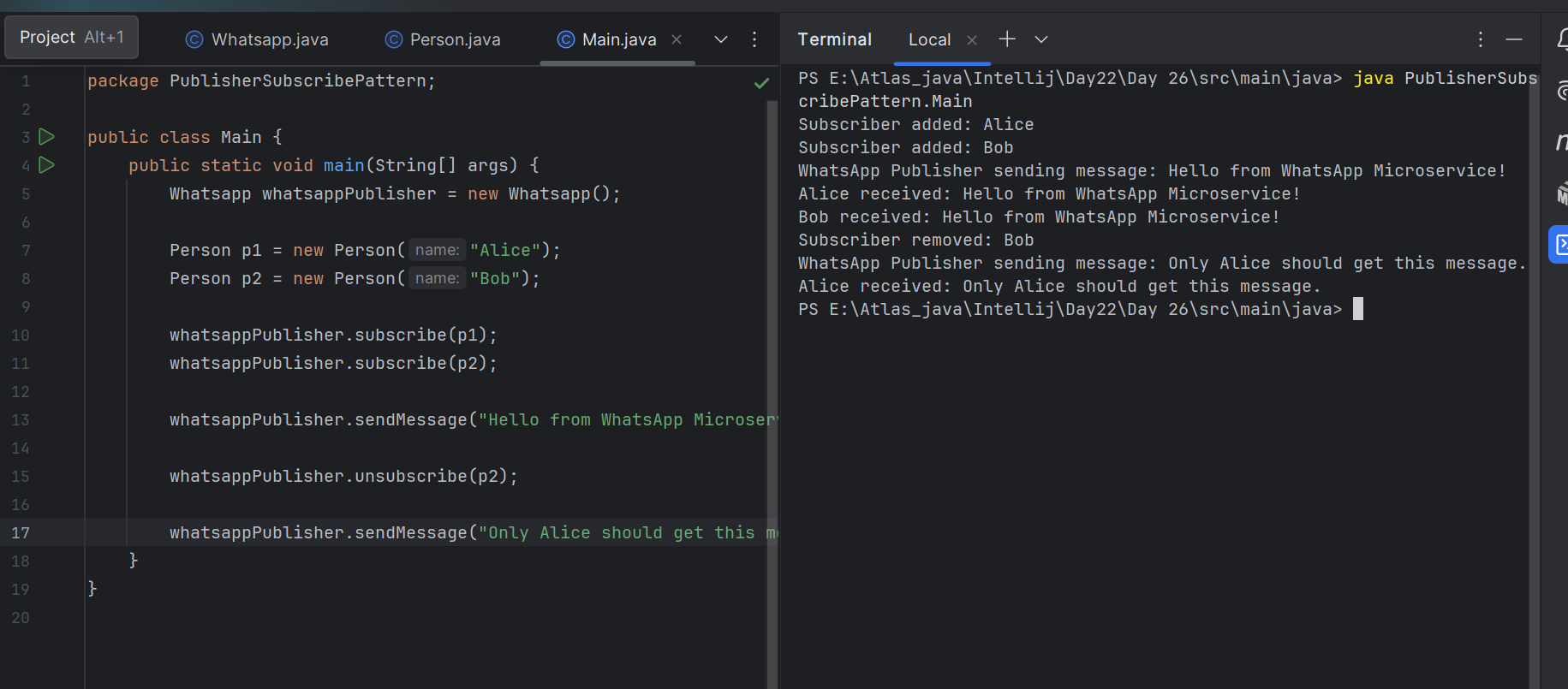
**whatsappPublisher.unsubscribe(p2);**

**whatsappPublisher.sendMessage("Only Alice should get this message.");**

**}**

**}**

**OUTPUT:**

****

3. 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)

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.

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

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.