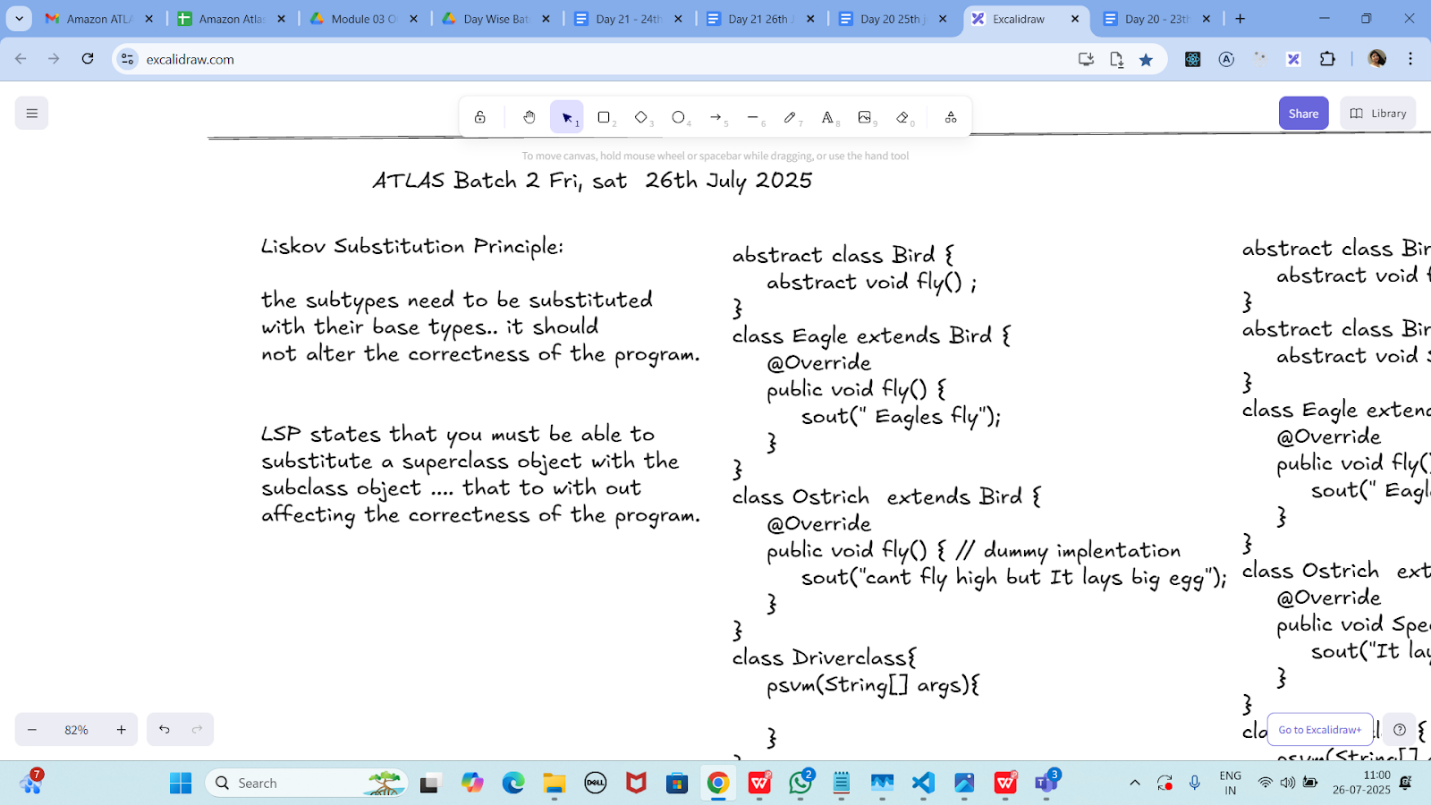
Day 21 26th July 2025

• S: One class = One job.  
  •  O: Add new features without modifying old code.  
  •  L: Subclasses should work where parent works.  
 • I: Small, specific interfaces > big ones.  
 • D: Depend on abstractions, not concrete classes.

| **Letter** | **Principle Name** | **Description** |
| --- | --- | --- |
| S | Single Responsibility | One reason to change, one responsibility per class. |
| O | Open/Closed | Open for extension, closed for modification. |
| L | Liskov Substitution | Subtypes are substitutable for their base types. |
| I | Interface Segregation | Prefer multiple specific interfaces to one general-purpose interface. |
| D | Dependency Inversion | Depend on abstractions, not concrete implementations |

Liskov Substitution Principle



Task 01:

Violation of Liskov

abstract class Bird {

    abstract void fly() ;

}

class Eagle extends Bird {

    @Override

    public void fly() {

        sout(" Eagles fly");

    }

}

class Ostrich  extends Bird {

    @Override

    public void fly() { // dummy implentation

        sout("cant fly high but It lays big egg");

    }

}

class Driverclass{

    psvm(String[] args){

    }

}

Task 02:

Implementation of Liskov

abstract class BirdsthatFly {

    abstract void fly() ;

}

abstract class BirdsthatDontFly {

    abstract void Speciality() ;

}

class Eagle extends BirdsthatFly {

    @Override

    public void fly() {

        sout(" Eagles fly");

    }

}

class Ostrich  extends BirdsthatDontFly {

    @Override

    public void Speciality() {

        sout("It lays big egg");

    }

}

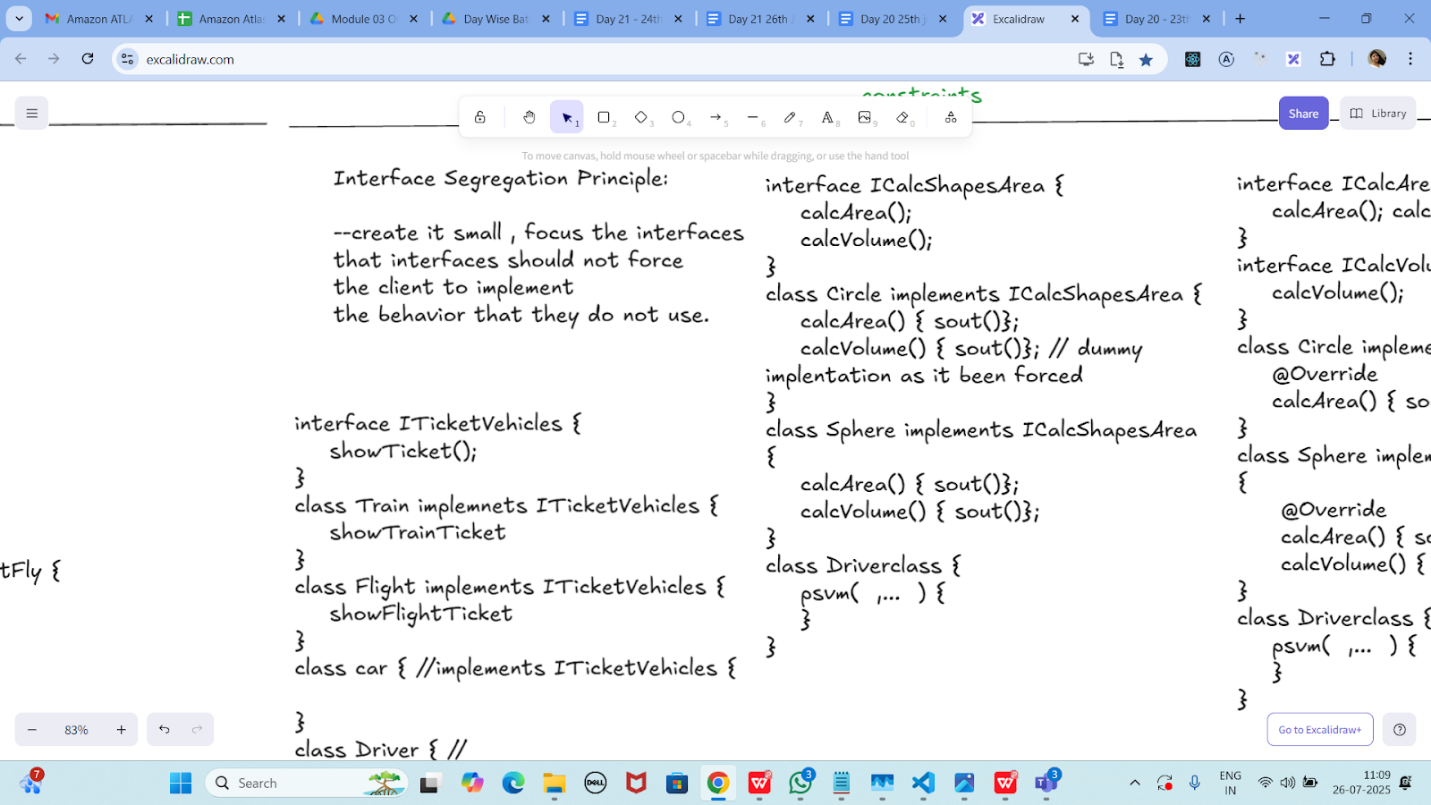
class Driverclass{

    psvm(String[] args){

    }

}

I - Interface Segregation Principle



Task 03:

Violation of Interface Segregation principle

interface ICalcShapesArea {

    calcArea();

    calcVolume();

}

class Circle implements ICalcShapesArea {

    calcArea() { sout()};

    calcVolume() { sout()}; // dummy implentation as it been forced

}

class Sphere implements ICalcShapesArea {

    calcArea() { sout()};

    calcVolume() { sout()};

}

class Driverclass {

    psvm(  ,...  ) {

    }

}

Task 04:

Implementation of Interface Segregation Principle

interface ICalcArea {

    calcArea(); calcPerimeter();

}

interface ICalcVolume {

    calcVolume();

}

class Circle implements ICalcArea {

    @Override

    calcArea() { sout()};

}

class Sphere implements ICalcArea, ICalcVolume {

     @Override

     calcArea() { sout()};

     calcVolume() { sout()};

}

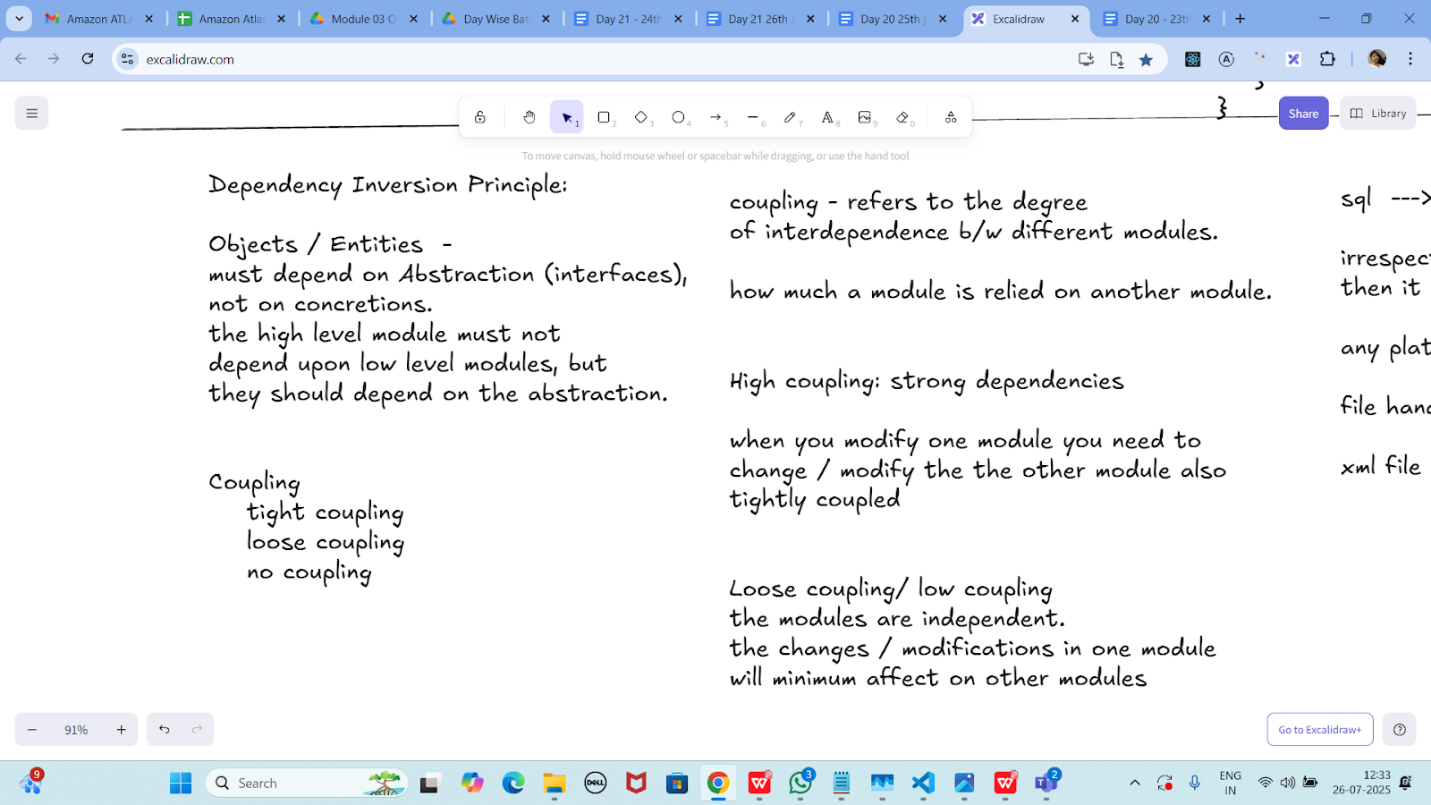
class Driverclass {

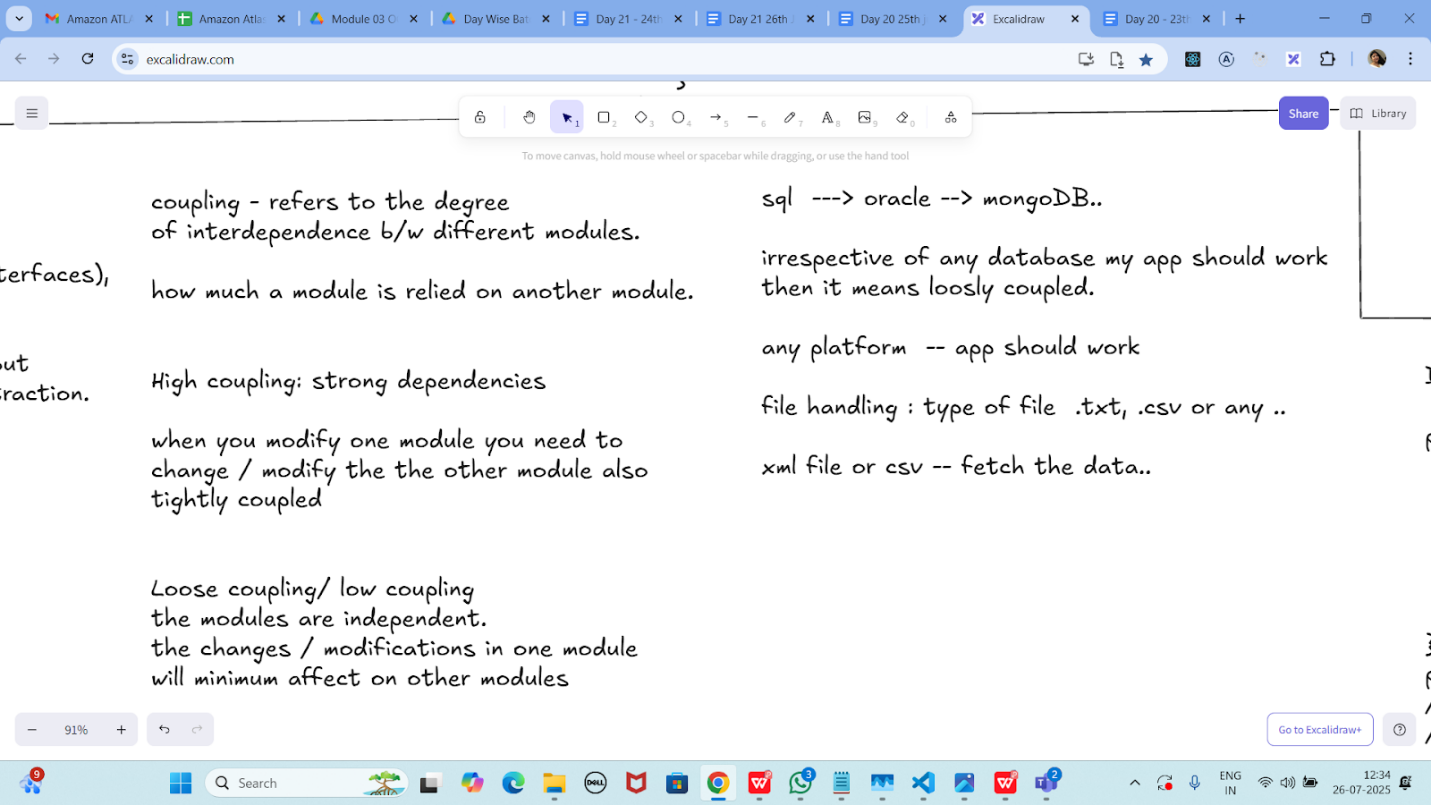
    psvm(  ,...  ) {

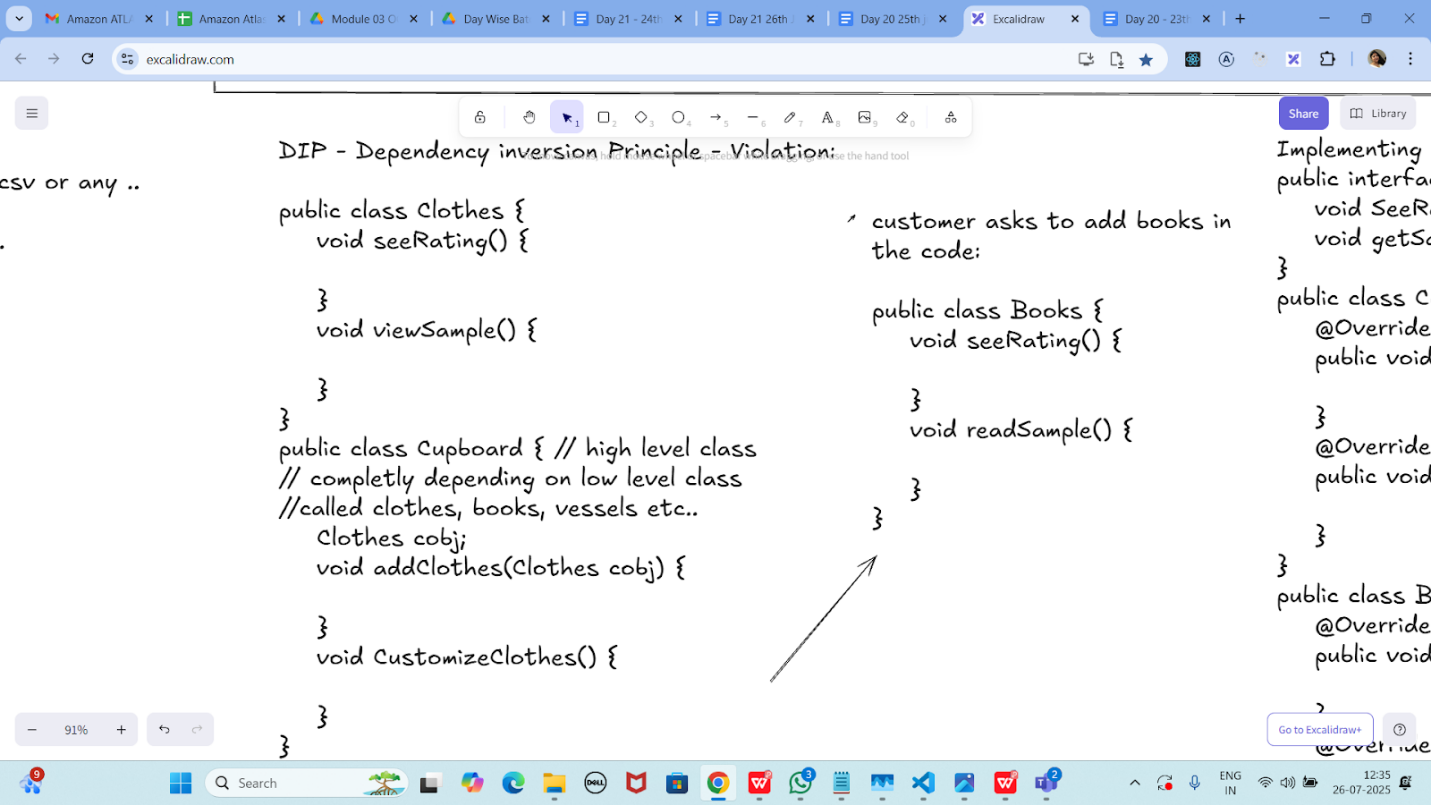
    }

}

Dependency Inversion Principle:







Dip violation code:

DIP - Dependency inversion Principle - Violation:

public class Clothes {

    void seeRating() {

    }

    void viewSample() {

    }

}

public class Cupboard { // high level class

// completly depending on low level class

//called clothes, books, vessels etc..

    Clothes cobj;

    void addClothes(Clothes cobj) {

    }

    void CustomizeClothes() {

    }

}

customer asks to add books in the code:

public class Books {

    void seeRating() {

    }

    void readSample() {

    }

}

Now DIP implementation:

Implementing Dependency Inversion Principle

public interface IProduct {

    void SeeReviews();

    void getSample();

}

public class Clothes implements IProduct {

    @Override

    public void SeeReviews() {

    }

    @Override

    public void getSample() {

    }

}

public class Books implements IProduct {

    @Override

    public void SeeReviews() {

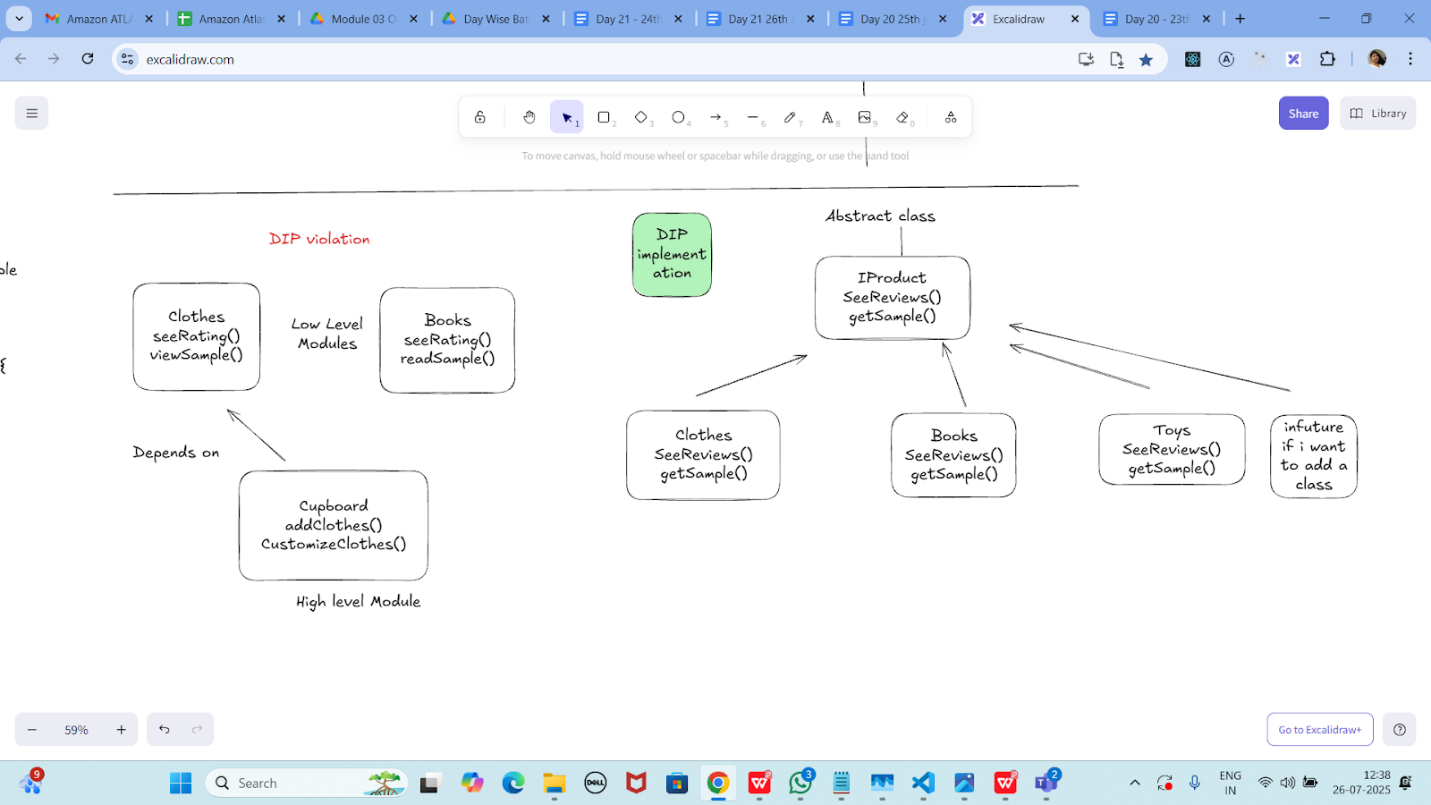
    }

    @Override

    public void getSample() {

    }

}



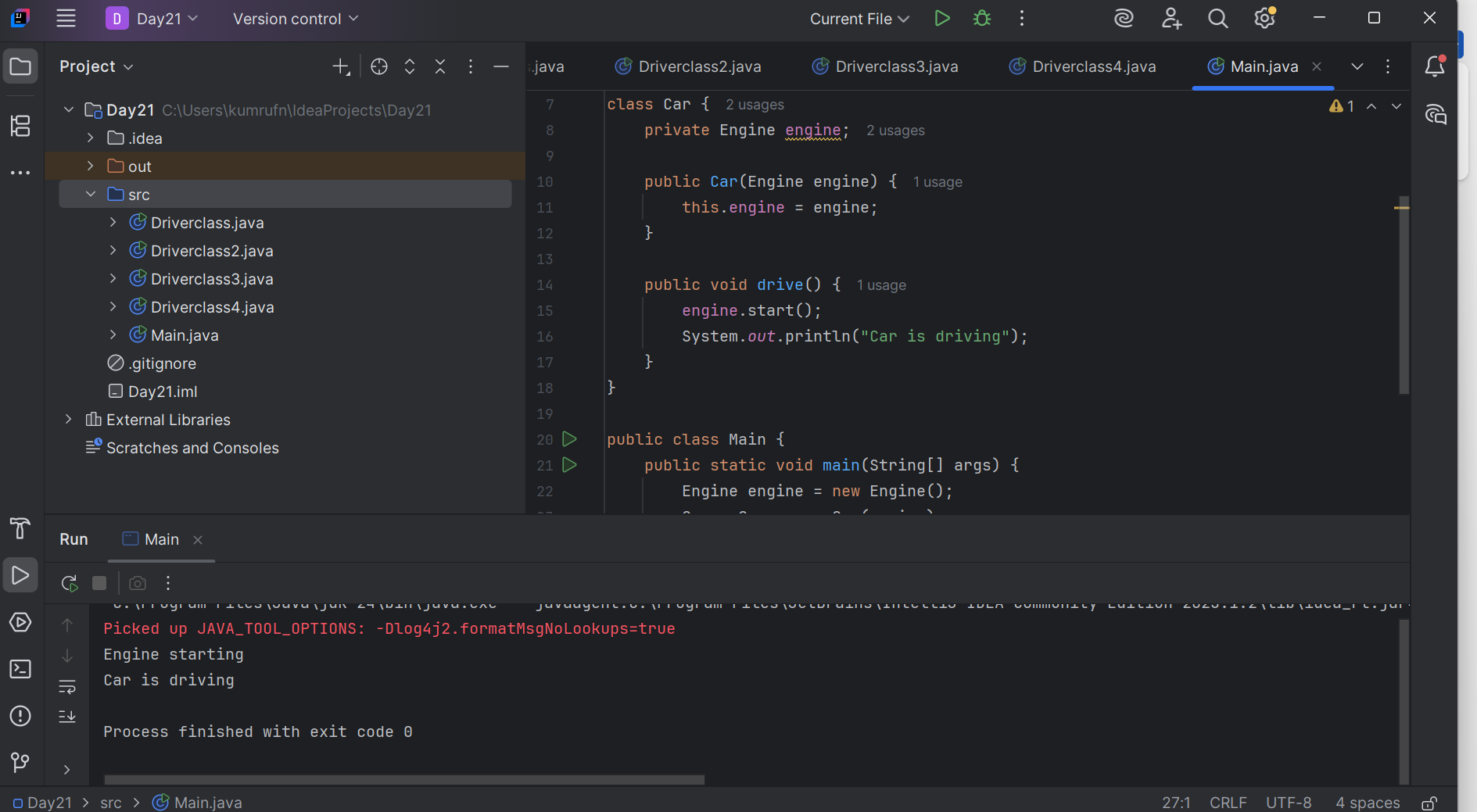
**Document :**

https://docs.google.com/document/d/1-z9nrbDEnY0gdbUgTKADWljkYHup-gBTU-PhO7XwNPE/edit?tab=t.0#heading=h.pyjp6j9k1my6

**Dependency :**

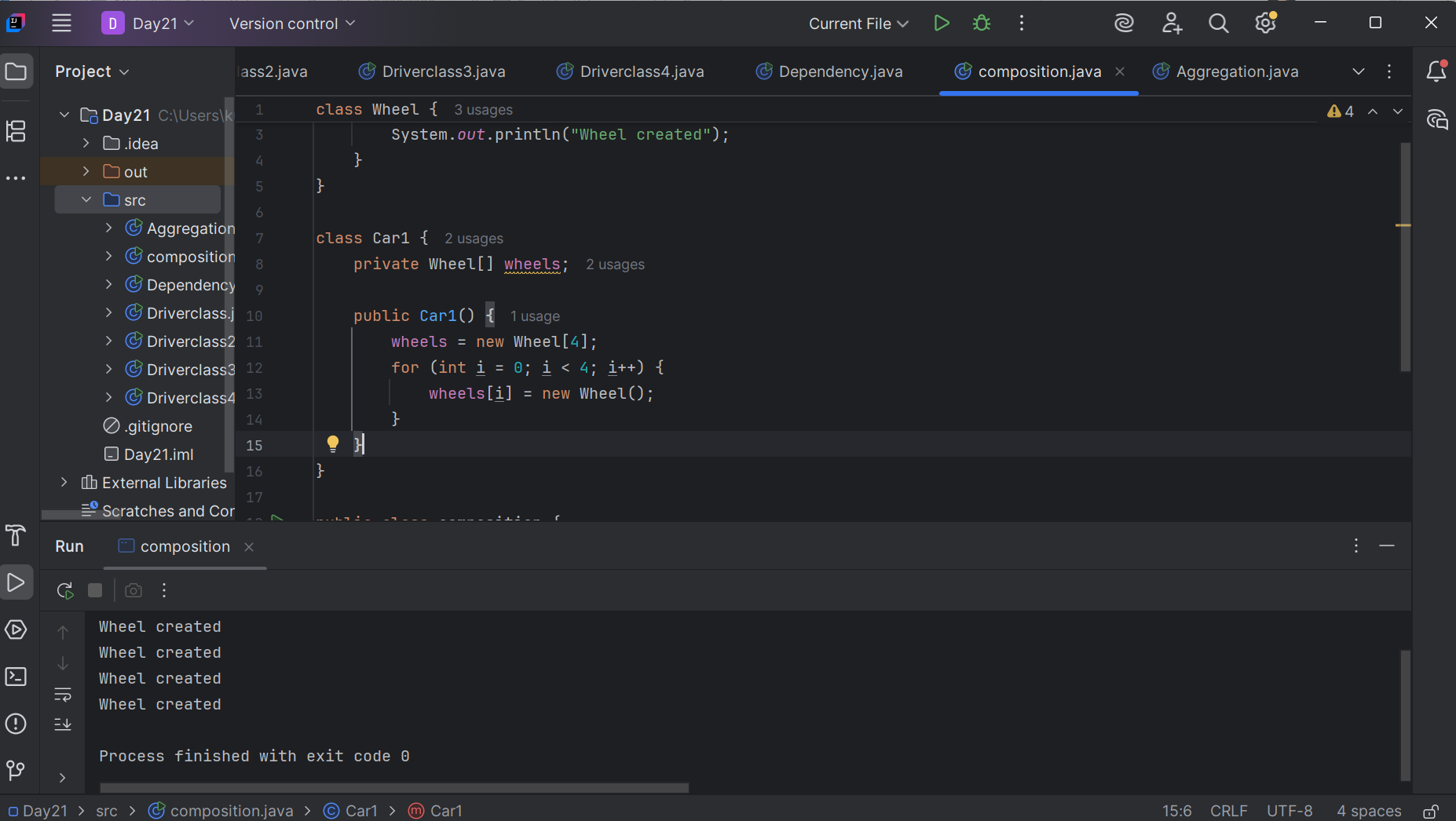
**Doc 1st code :**

class Engine {  
 public void start() {  
 System.*out*.println("Engine starting");  
 }  
}  
  
class Car {  
 private Engine engine;  
  
 public Car(Engine engine) {  
 this.engine = engine;  
 }  
  
 public void drive() {  
 engine.start();  
 System.*out*.println("Car is driving");  
 }  
}  
  
public class Dependency {  
 public static void main(String[] args) {  
 Engine engine = new Engine();  
 Car myCar = new Car(engine);  
 myCar.drive();  
 }  
}



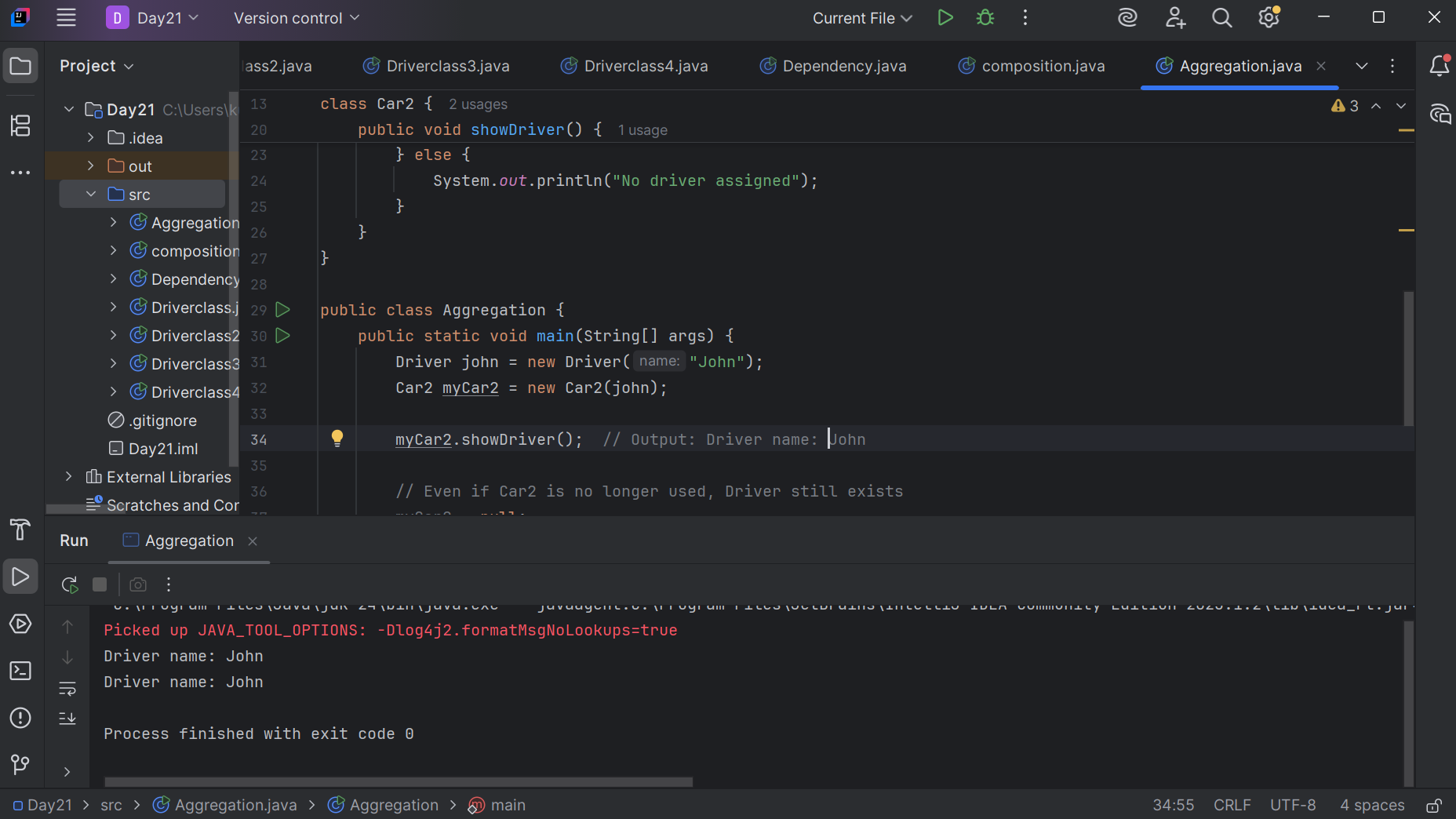
**Task 2 :**

class Wheel {  
 public Wheel() {  
 System.*out*.println("Wheel created");  
 }  
}  
  
class Car1 {  
 private Wheel[] wheels;  
  
 public Car1() {  
 wheels = new Wheel[4];  
 for (int i = 0; i < 4; i++) {  
 wheels[i] = new Wheel();  
 }  
 }  
}  
  
public class composition {  
 public static void main(String[] args) {  
 Car1 myCar = new Car1();  
 }  
}



**Task 3 :**

class Driver {  
 private String name;  
  
 public Driver(String name) {  
 this.name = name;  
 }  
  
 public void display() {  
 System.*out*.println("Driver name: " + name);  
 }  
}  
  
class Car2 {  
 private Driver driver; // Aggregation — Car2 has a Driver  
  
 public Car2(Driver driver) {  
 this.driver = driver;  
 }  
  
 public void showDriver() {  
 if (driver != null) {  
 driver.display();  
 } else {  
 System.*out*.println("No driver assigned");  
 }  
 }  
}  
  
public class Aggregation {  
 public static void main(String[] args) {  
 Driver john = new Driver("John");  
 Car2 myCar2 = new Car2(john);  
  
 myCar2.showDriver(); // Output: Driver name: John  
  
 // Even if Car2 is no longer used, Driver still exists  
 myCar2 = null;  
  
 john.display(); // Still works: Driver name: John  
 }  
}



Info Box

<https://excalidraw.com/#json=FvHREzZ_RxqrbO0_4iF3U,0WLjacoEfdym6RTsA9sGYA>