**Task 1**

**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 2**

**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){

}

}

**Task 3**

**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 4**

**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( ,... ) {

}

}

**Task 5**

**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() {

}

}

**Task 6**

**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() {

}

}

**Task 7**

public class Engine {

public void start() {

System.out.println("Engine starting");

}

}

public class Car {

public void drive() {

Engine engine = new Engine(); // Dependency: used only inside this method

engine.start();

System.out.println("Car is driving");

}

}

public class Main {

public static void main(String[] args) {

Car myCar = new Car();

myCar.drive();

}

}

**Task 8**

public class Driver {

private String name;

public Driver(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

public class Car {

private Driver driver; // Aggregation

public Car(Driver driver) {

this.driver = driver;

}

public void showDriver() {

if (driver != null) {

System.out.println("Car is driven by: " + driver.getName());

} else {

System.out.println("No driver assigned.");

}

}

}

public class Main {

public static void main(String[] args) {

Driver driver = new Driver("John"); // Driver exists independently

Car myCar = new Car(driver); // Car has-a Driver (Aggregation)

myCar.showDriver(); // Output: Car is driven by: John

}

}

**Task 9**

public class Wheel {

public Wheel() {

System.out.println("Wheel created");

}

}

public class Car {

private Wheel[] wheels; // Composition: Car owns the wheels

public Car() {

// Car creates and owns 4 Wheel objects

wheels = new Wheel[4];

for (int i = 0; i < 4; i++) {

wheels[i] = new Wheel();

}

System.out.println("Car created with 4 wheels");

}

}

public class Main {

public static void main(String[] args) {

Car myCar = new Car(); // Car and its wheels are created together

}

}