D :

We have a Car class that depends on the concrete Engine class; therefore, it is not obeying DIP.

public class Engine {  
    public Engine() {  
    }  
    public void start() {  
        System.out.println(“Engine Started”);  
    }  
}

public class Car {  
    private Engine engine;  
    public Car(Engine e) {  
        engine = e;  
    }  
    public void startCar() {  
        engine.start();  
    }  
}

public class DriverApp

{

    public static void main(String[] args)

{

        Engine e1 = new Engine();

        Car c1 = new Car(e1);

        c1.startCar();

}

}

 but what if we wanted to add another engine type, let’s say a diesel engine? This will require refactoring the Car class.

Solution:

solve this by introducing a layer of abstraction. Instead of Car depending directly on Engine, let’s add an interface:

public interface Engine {  
    public void start();  
}

*// Class for PetrolEngine*

class PetrolEngine implements Engine {

    public void start() {

        System.out.println("Petrol engine started.");

    }

}

*// Class for PetrolEngine*

class DieselEngine implements Engine {

    public void start() {

        System.out.println("Diesel engine started.");

    }

}

class Car {

    private Engine engine;

    public Car(Engine e) {

        this.engine = e;

    }

    public void startCar() {

        engine.start();

    }

}

public class Main {

    public static void main(String[] args) {

        Engine petrolEngine = new PetrolEngine();

        Engine dieselEngine = new DieselEngine();

        Car petrolCar = new Car(petrolEngine);

        Car dieselCar = new Car(dieselEngine);

        petrolCar.startCar();

        dieselCar.startCar();

    }

}