In the PatientManagement.java class, we can see violations of the **Dependency Inversion Principle (DIP)**. The class is directly dependent on the concrete classes SurgeryTreatmentPlan, MedicationTreatmentPlan, OnlinePrescriptionService, and PrintablePrescriptionService. This means the high-level module (PatientManagement) is tightly coupled with these low-level classes, making the code less flexible and harder to maintain.

**Violation of DIP:**

* **createTreatmentPlan()**: The method directly instantiates SurgeryTreatmentPlan and MedicationTreatmentPlan, which violates DIP. Instead, it should depend on the abstraction TreatmentPlan.
* **generatePrescription()**: Similarly, it instantiates OnlinePrescriptionService and PrintablePrescriptionService, which should also be abstracted using PrescriptionService.

**Refactoring to Comply with DIP:**

We will refactor the PatientManagement class so that it depends on abstractions (TreatmentPlan and PrescriptionService) rather than concrete implementations. This will make the class more flexible and allow for the easy addition of new types of treatment plans and prescription services without modifying the PatientManagement class.

**Refactored Code:**

**PatientManagement.java (Refactored):**

package com.algonquin.management;

import com.algonquin.patient.Patient;

import com.algonquin.prescription.PrescriptionService;

import com.algonquin.treatment.TreatmentPlan;

import com.algonquin.utilities.Utilities;

import java.io.IOException;

public class PatientManagement {

private TreatmentPlan treatmentPlan;

private PrescriptionService prescriptionService;

// Constructor to inject dependencies

public PatientManagement(TreatmentPlan treatmentPlan, PrescriptionService prescriptionService) {

this.treatmentPlan = treatmentPlan;

this.prescriptionService = prescriptionService;

}

// Create treatment plan based on injected treatment plan type

public void createTreatmentPlan(Patient patient) throws IOException {

// Using injected TreatmentPlan dependency

patient.setAge(Utilities.calulatePatientAge(patient.getDateOfBirth()));

patient.setLifeStage(Utilities.determineLifeStage(patient.getAge()));

String plan = treatmentPlan.createTreatmentPlan(patient);

patient.setPlannedTreatment(plan);

}

// Generate prescription based on injected PrescriptionService

public void generatePrescription(Patient patient, String filename) throws IOException {

prescriptionService.generatePrescription(patient, filename);

}

}

**Explanation of Refactoring:**

* **Dependency Injection**:
  + Instead of directly instantiating SurgeryTreatmentPlan, MedicationTreatmentPlan, OnlinePrescriptionService, and PrintablePrescriptionService inside the method, we inject these dependencies into the PatientManagement class through its constructor. This is an example of **dependency injection**, which allows the PatientManagement class to depend on abstractions (TreatmentPlan and PrescriptionService) instead of concrete implementations.
* **Flexible Design**:
  + Now, we can pass any implementation of TreatmentPlan (such as SurgeryTreatmentPlan or MedicationTreatmentPlan) or PrescriptionService (such as OnlinePrescriptionService or PrintablePrescriptionService) when creating a PatientManagement object. This makes the code more flexible and adheres to the DIP principle.

**Example Usage:**

Here’s how we would use the refactored PatientManagement class:

java

Copy code

// In your main program or service class

// Inject SurgeryTreatmentPlan and OnlinePrescriptionService

TreatmentPlan treatmentPlan = new SurgeryTreatmentPlan();

PrescriptionService prescriptionService = new OnlinePrescriptionService();

PatientManagement patientManagement = new PatientManagement(treatmentPlan, prescriptionService);

Patient patient = new Inpatient("001", "John Doe", "john.doe@example.com", "1234567890", "1987-10-17", "Room 101");

patientManagement.createTreatmentPlan(patient);

patientManagement.generatePrescription(patient, "john\_doe\_prescription.txt");

// For another patient with MedicationTreatmentPlan and PrintablePrescriptionService

TreatmentPlan medicationPlan = new MedicationTreatmentPlan();

PrescriptionService printService = new PrintablePrescriptionService();

PatientManagement patientManagement2 = new PatientManagement(medicationPlan, printService);

Patient patient2 = new Outpatient("002", "Jane Smith", "jane.smith@example.com", "9876543210", "1990-05-23", "2024-10-10");

patientManagement2.createTreatmentPlan(patient2);

patientManagement2.generatePrescription(patient2, "jane\_smith\_prescription.txt");

**Summary:**

By applying the **Dependency Inversion Principle (DIP)**:

* **High-level modules (PatientManagement)** now depend on **abstractions** (TreatmentPlan, PrescriptionService) instead of concrete implementations.
* The code is now more **modular**, **flexible**, and **extensible**, making it easier to add new treatment plans or prescription services in the future without changing the PatientManagement class.
* We used **dependency injection** to pass the specific implementations of TreatmentPlan and PrescriptionService, further decoupling the high-level logic from low-level implementations.