**tep-by-Step Guidance for Assignment #1:**

This assignment involves refactoring existing Java code using SOLID principles and developing additional functionality. Let's go step by step to ensure we achieve a modular and maintainable design.

**Step 1: Single Responsibility Principle (SRP) in PatientManagement.java**

The first task is to examine PatientManagement.java and refactor it according to the **Single Responsibility Principle** (SRP). This principle states that a class should have only one reason to change, meaning it should only be responsible for one task.

1. **Current Problem**:
   * The PatientManagement class likely manages patients and handles both treatment planning and prescription generation, which violates SRP.
2. **Refactoring**:
   * Split the responsibilities:
     + Create a new PatientService class that handles patient-related operations.
     + Use a separate TreatmentPlanService class for treatment plans.
     + Use a separate PrescriptionService for prescription generation.
   * PatientManagement.java should only coordinate these services.

**Example**:

java

Copy code

public class PatientService {

public void managePatient(Patient patient) {

// Logic to manage patient data

}

}

public class TreatmentPlanService {

public void createTreatmentPlan(Patient patient) {

// Logic to create a treatment plan based on patient type

}

}

public class PrescriptionService {

public void generatePrescription(Patient patient) {

// Logic to generate prescription

}

}

**Step 2: Liskov Substitution Principle (LSP) for Patient, Inpatient, and Outpatient**

Next, we need to ensure that **Liskov Substitution Principle (LSP)** is followed, meaning subclasses should be replaceable with their parent class without affecting the program's correctness.

1. **Current Problem**:
   * If Inpatient or Outpatient classes override methods in ways that violate the expected behavior of the Patient class, it breaks LSP.
2. **Refactoring**:
   * Ensure both Inpatient and Outpatient subclasses only extend Patient where it makes sense.
   * Avoid overriding methods that change the behavior expected by Patient.

**Example**:

java

Copy code

public class Patient {

public void getDetails() {

// Basic patient details

}

}

public class Inpatient extends Patient {

@Override

public void getDetails() {

super.getDetails();

// Add inpatient-specific details

}

}

public class Outpatient extends Patient {

@Override

public void getDetails() {

super.getDetails();

// Add outpatient-specific details

}

}

**Step 3: Dependency Inversion Principle (DIP) for createTreatmentPlan() and generatePrescription()**

Now, we will address violations of the **Dependency Inversion Principle (DIP)**. The high-level PatientManagement class should not depend directly on low-level implementations like createTreatmentPlan() or generatePrescription().

1. **Current Problem**:
   * PatientManagement likely uses concrete methods for treatment plans and prescriptions, which violates DIP.
2. **Refactoring**:
   * Use abstractions (TreatmentPlan.java and PrescriptionService.java) to decouple the high-level logic from the low-level implementations.

**Example**:

java

Copy code

public class PatientManagement {

private TreatmentPlan treatmentPlan;

private PrescriptionService prescriptionService;

public PatientManagement(TreatmentPlan treatmentPlan, PrescriptionService prescriptionService) {

this.treatmentPlan = treatmentPlan;

this.prescriptionService = prescriptionService;

}

public void managePatient(Patient patient) {

treatmentPlan.createPlan(patient);

prescriptionService.generatePrescription(patient);

}

}

**Step 4: Implement MedicationTreatmentPlan.java and SurgeryTreatmentPlan.java**

We need to create two new classes that implement the TreatmentPlan interface.

1. **Business Rules**:
   * For MedicationTreatmentPlan:
     + Different medications and durations based on age and condition.
   * For SurgeryTreatmentPlan:
     + Surgery details including fasting and follow-up based on age.

**Example**:

java

Copy code

public class MedicationTreatmentPlan implements TreatmentPlan {

@Override

public void createPlan(Patient patient) {

// Logic based on patient age and medical condition

}

}

public class SurgeryTreatmentPlan implements TreatmentPlan {

@Override

public void createPlan(Patient patient) {

// Logic for surgery details like fasting and follow-up

}

}

**Step 5: Implement OnlinePrescriptionService.java and PrintablePrescriptionService.java**

Next, we implement two classes to handle different prescription formats.

1. **OnlinePrescriptionService**:
   * Generates prescription in .html format.
2. **PrintablePrescriptionService**:
   * Generates prescription in .txt format.

**Example**:

java

Copy code

public class OnlinePrescriptionService implements PrescriptionService {

@Override

public void generatePrescription(Patient patient) {

// Logic to generate .html prescription file

}

}

public class PrintablePrescriptionService implements PrescriptionService {

@Override

public void generatePrescription(Patient patient) {

// Logic to generate .txt prescription file

}

}

**Step 6: Implement Physician.java**

Finally, create the main class Physician to simulate the complete solution.

1. **Instantiate Patients**:
   * Create two patient objects (one Inpatient and one Outpatient).
2. **Utilize PatientManagement**:
   * Use the management system to create treatment plans and generate prescriptions.

**Example**:

java

Copy code

public class Physician {

public static void main(String[] args) {

Patient inpatient = new Inpatient();

Patient outpatient = new Outpatient();

TreatmentPlan medicationPlan = new MedicationTreatmentPlan();

PrescriptionService onlineService = new OnlinePrescriptionService();

PatientManagement patientManagement = new PatientManagement(medicationPlan, onlineService);

patientManagement.managePatient(inpatient);

patientManagement.managePatient(outpatient);

}

}

**Step 7: JUnit Tests**

Develop **JUnit tests** to verify the business rules. Ensure each new class adheres to the requirements and is functioning as expected.

**Example**:

java

Copy code

@Test

public void testCreateMedicationPlan() {

Patient patient = new Outpatient();

TreatmentPlan plan = new MedicationTreatmentPlan();

plan.createPlan(patient);

// Assertions to verify the logic

}

**Step 8: UML Class Diagram and Report**

Finally, generate a **UML Class Diagram** that includes:

* Patient, Inpatient, Outpatient
* TreatmentPlan, MedicationTreatmentPlan, SurgeryTreatmentPlan
* PrescriptionService, OnlinePrescriptionService, PrintablePrescriptionService
* PatientManagement, Physician

Explain how each SOLID principle was applied.

**Final Notes:**

* Ensure **clean, commented code** following Java standards.
* Demo your solution during the lab session.
* Submit your code and UML diagram in time.