**1. Patient Class Hierarchy**

The **Patient** class and its two subclasses **Inpatient** and **Outpatient** are central to your design.

* **Patient Class**: This is a base class representing general patient attributes such as:
  + id: String, representing patient ID.
  + name, email, phoneNumber, dateOfBirth, lifeStage, plannedTreatment: These are patient attributes needed for processing.
  + **Methods**: The methods like getId(), getName(), setLifeStage() allow accessing and modifying the patient's details.
* **Inpatient and Outpatient**: Both of these extend from **Patient**.
  + **Inpatient**: Adds attributes such as roomNumber and methods like admit().
  + **Outpatient**: Adds an attribute for appointmentDate and a similar admit() method.

**Explanation:**

The **Patient** class follows the **Inheritance** principle, where **Inpatient** and **Outpatient** inherit from the base **Patient** class. This represents the **Liskov Substitution Principle (LSP)** from the **SOLID** design principles—subtypes like Inpatient and Outpatient should be substitutable for their base type, Patient.

**2. TreatmentPlan Interface and its Implementations**

The **TreatmentPlan** interface and its two concrete classes **MedicationTreatmentPlan** and **SurgeryTreatmentPlan** handle different treatment plans for patients.

* **TreatmentPlan Interface**: Contains a method createTreatmentPlan(Patient patient): String. This method is used by different treatment plans to create the treatment details for each patient.
* **MedicationTreatmentPlan** and **SurgeryTreatmentPlan**: Both of these implement the **TreatmentPlan** interface and provide specific behavior for different treatments.

**Explanation:**

This setup is an application of the **Strategy Pattern**, where you can switch between different strategies (types of treatment plans) at runtime. For example, depending on whether a patient needs medication or surgery, a different treatment plan can be used. The PatientManagement class injects these strategies using the **Dependency Injection** pattern.

**3. PatientManagement Class**

The **PatientManagement** class is where the real "management" happens.

* **Attributes**:
  + treatmentPlan: This is of type TreatmentPlan and represents the treatment strategy that will be used.
  + prescriptionService: This handles the generation of prescriptions.
* **Key Methods**:
  + createTreatmentPlan(Patient patient): This method delegates the responsibility of creating a treatment plan to the TreatmentPlan instance.
  + generatePrescription(Patient patient, String filename): This uses the PrescriptionService to generate the prescription based on the treatment.

**Explanation:**

This class follows the **Single Responsibility Principle (SRP)**—it only manages the patient data and delegates the work of creating treatment plans and generating prescriptions to other classes. It uses **Dependency Injection** to inject the treatment strategy and prescription service at runtime, thus adhering to the **Open/Closed Principle (OCP)**, where you can extend behavior (e.g., adding new types of treatments or prescriptions) without modifying existing code.

**4. Prescription and PrescriptionService**

The **PrescriptionService** interface has two concrete implementations: **OnlinePrescriptionService** and **PrintablePrescriptionService**.

* **Prescription Class**: Contains detailed attributes about the prescription, such as:
  + dailyDosageCount, duration, diseaseType, riskFactor, followUpDate, etc.
* **PrescriptionService Interface**: This defines the method generatePrescription(Patient patient): String.
* **OnlinePrescriptionService** and **PrintablePrescriptionService**: These classes provide two different formats for generating prescriptions:
  + **OnlinePrescriptionService**: Generates prescriptions in HTML format.
  + **PrintablePrescriptionService**: Generates prescriptions in plain text format.

**Explanation:**

This setup is an example of the **Factory Pattern**—depending on the requirement, you can choose to generate a prescription either in online format or printable format. The **PrescriptionService** interface acts as the factory, while its concrete implementations handle the actual prescription creation.

**5. Utility and Validator Classes**

* **PatientUtilities**: Contains helper methods to calculate a patient’s age (calculatePatientAge) and determine their life stage (determineLifeStage).
* **PatientValidator**: Contains methods to validate patient data, such as isValidName, isValidEmail, and isValidPhone.

**Explanation:**

These utility classes ensure that common functions, such as validations and calculations, are extracted from the main patient management logic. This improves the **Single Responsibility Principle (SRP)**—keeping each class focused on a specific responsibility.

**6. Constants Class**

The **Constants** class holds static values for different constants used in the application, like CHILD, YOUTH, ADULT, and ACUTE\_DURATION.

**Explanation:**

This is a simple way to follow the **DRY (Don’t Repeat Yourself)** principle. By centralizing all the constants in one class, you avoid hard-coding the same values in multiple places, which would be error-prone.

**7. Physician Class**

The **Physician** class has a main() method and serves as the entry point to the application.

* It instantiates two patient objects and applies different treatment plans and prescriptions using the PatientManagement class.

**Explanation:**

This class demonstrates the actual running of the program. It uses the objects defined in the other classes and connects them to simulate a complete medical record system.

**Final Thoughts:**

* **Design Patterns in Use**:
  + **Strategy Pattern**: Used to apply different treatment plans (MedicationTreatmentPlan or SurgeryTreatmentPlan).
  + **Factory Pattern**: Used to generate different types of prescriptions (OnlinePrescriptionService or PrintablePrescriptionService).
  + **Single Responsibility Principle**: Each class has one specific job—**PatientManagement** manages patients, **PrescriptionService** generates prescriptions, and so on.
* **Relationships**:
  + **Inheritance**: **Inpatient** and **Outpatient** inherit from **Patient**.
  + **Interfaces and Implementations**: **TreatmentPlan** and **PrescriptionService** are interfaces, and their respective classes (like **MedicationTreatmentPlan**, **SurgeryTreatmentPlan**, etc.) implement these interfaces.

The relationships in the UML diagram look correct based on the code, and all the design patterns and principles are well applied to keep the code modular, maintainable, and extensible.