1. Abstract Factory Pattern

Usage: To provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Application: The action of adding a new doctor to the hospital. Every doctor has their degree and specialization where they might be a very competent doctor with multiple and maybe specialized skills. We need to create a new doctor object with all the skills, information and dependencies.

- Hospital has doctors
- Doctor has Degree(s)
- Degree has a specific Degree Factory extending default (general physician) degree factory
- Degree factory returns degree containing multiple skills given specific factory class
- For a new degree, we add new degree factory with composeSkills() method containing all the required skills

2. Builder Pattern

Usage: To construct a complex object step by step, allowing for the creation of different types and representations of an object using the same construction process.

Application: Building patient medical records, which might have multiple optional attributes (e.g., medical history, surgeries, allergies, medications).

Director:
Manages the construction process. It holds a reference to the Builder and constructs the PatientRecord by calling the builder's methods.

Methods:
setBuilder(Builder builder): Sets the builder to use.
construct(): Orchestrates the construction of the PatientRecord.

Builder:
An abstract class or interface that defines the methods for setting the different parts of the PatientRecord and for building the final product.

ConcreteBuilder:
Implements the Builder interface and constructs the PatientRecord step by step.
Holds a PatientRecord instance that it configures through its set methods.

Methods:
Implements all the Builder methods to set parts of the PatientRecord.
build(): Returns the fully constructed PatientRecord.

PatientRecord:
The complex object that is being built.

Fields:

name: String
medicalHistory: String
surgeries: String
allergies: String
medications: String

Constructor:

Takes a Builder as a parameter to initialize its fields.

3. Singleton pattern

Usage: To ensure that a class has only one instance and provide a global point of access to it.

Assignment: We need to manage the assignment of rooms to patients in the hospital management system. The class diagram must ensure that only one instance of the RoomAssignmentManager exists, which handles the assignment of rooms to patients, ensuring that each patient is assigned only one room.

RoomAssignmentManager:
Singleton class that manages the assignment of rooms to patients.

Fields:

- instance: RoomAssignmentManager: A private static variable that holds the single instance of the class.
- roomAssignments: Map<Patient, Room>: A private map that stores the association between patients and rooms.

Methods:

- getInstance(): Public static method that returns the single instance of the class, ensuring only one instance exists.
- assignRoom(patient: Patient, room: Room): Public method to assign a room to a patient.
- getRoom(patient: Patient): Public method to get the room assigned to a patient.
- releaseRoom(patient: Patient): Public method to release the room assigned to a patient.

Constructor:

- RoomAssignmentManager(): A private constructor to prevent instantiation from outside the class.