UML

Alert Generation System

We designed this system to evaluate the patient data incoming in real time, it compares it with our personalised thresholds and generates alerts when anomalies are detected. The classes are AlertGenerator, Alert, and AlertManager.

1. AlertGenerator evaluates vital signals with the specific threshold of the patient. It has a relationship with Threshold, which holds the rules.
2. The Threshold object can be customised for different patients for modularity. This design follows a Strategy pattern, allowing different evaluation strategies.
3. When a threshold is activated, an Alert object is created. This class contains patient ID, condition, timestamp, and serves as a reliable carrier of alert data.
4. AlertManager splits these alerts to medical staff via multiple channels (email, pager, etc.). This separation ensures a clean limit between detection and communication.
5. The model also links to Patient and DataStorage; the system both reads patient information and interacts with stored data for decisions with context.
   This design is modular and enhances maintainability. If new alert types or thresholds need to be implemented, they can be added without changing the main alert logic.

Data Storage System

The Data Storage System is made for storing patient data from the signal generator. It ensures secure data.

1. The main class, DataStorage, manages storage, indexing, and retrieval. It holds a collection of PatientData instances, each representing a snapshot of patient vitals.
2. PatientData contains fields such as heart rate, blood pressure, timestamp, and patient ID. It is linked to Patient.
3. DataRetriever serves as the API for querying data. It abstracts query complexity and offers role-based access to ensure compliance with data privacy.
4. A DeletionPolicy class is introduced to define rules for deleting old data automatically, helping meet scalability and compliance needs.
5. Access control is modeled via AccessManager, ensuring only authorized personnel can query or delete data. This addresses both security and accountability.
   This architecture ensures modularity, scalability, and security. The data store can evolve without impacting other components.

Patient Identification System

The Patient Identification System ensures that each data is well linked to a patient in the hospital database.

1. PatientIdentifier is the main entry point. It receives raw data or ID from the data stream and attempts to resolve them to valid HospitalPatient entries.
2. The HospitalPatient class stores critical identity and demographic data.
3. The IdentityManager oversees this process, handling errors such as mismatches, missing data, or duplicate records. It stores incidents and raises alerts when something not normal happens.
4. PatientRegistry is made to maintain a centralized list of hospital patients.
5. And to handle anonymization, Anonymizer is optionally introduced to map real IDs to pseudonymous IDs in systems where data sensitivity

is high.
The system ensures traceability, compliance, and operational integrity. By separating the ID resolution from the logic, we enable flexible integration with external hospital systems. This modularity supports real-world requirements and identity de-duplication.

Data Access Layer

This system abstracts the retrieval and standardization of real-time data from the simulator to the internal data model of the CHMS system.

1. The DataListener is made as an abstract class, implemented by TCPDataListener, WebSocketDataListener, and FileDataListener, each handling a different input source.
2. Each subclass is responsible for managing the connection protocol, reading raw data, and passing it to the DataParser.
3. DataParser is a utility class that processes raw data into structured objects, converting it into a standardized PatientData object.
4. DataSourceAdapter acts as a bridge between this subsystem and DataStorage.
   This design respects the Open/Closed Principle: new listener types can be added without modifying the existing code. The use of a shared interface promotes testability and allows mocking in unit tests