**The Alert Generation System**

The Alert Generation System is designed to actively manage patient-specific medical alerts within a cardiovascular ward in a hospital.

**Central Classes:**

The **“AlertGenerator”** class is responsible for continuous evaluation of data incoming from various medical devices and initializing action, when thresholds are crossed.  
It has two methods: evaluateData() and triggerAlert(). Separation of data monitoring and alert triggering ensures clear distinction between two tasks of the class, providing clarity and maintainability.

The **“Alert”** class captures all necessary information about an alert within four fields: patientId, alertId, timestamp and condition. By providing detailed information about each alert, the class ensures that all alerts are easily traceable and analysed.

The **“AlertManager”** class manages all “Alert” instances sent by the “AlertGenerator”. There are two methods within the class: sendAlert() and addToPatientHistory(). They are needed for alert transmission and creation of historical record. Alerts are managed using a Priority Queue, ensuring prioritising of critical alerts.

**Supporting Classes:**

The **“PatientRecord”** class maintains detailed patient information, including personalized thresholds (“heartRateThreshold”, “bloodPressureThreshold”) required for alert analysis.  
The class also stores history of all alerts triggered for the patient (“alertHistory”). Collecting all data concerning a patient within one class simplifies the access to the data and performing updates on the data.

The **“PatientData”** class represents real-time patient data. It includes “heartRate” and “bloodPressure” and boolean indicators for whether they fit the established norm. Separation of dynamically changing data from static patient data of “PatientRecord” is important for real- time evaluation performed by the system.

The **“BloodPressure”** class is needed, because of a structure of blood pressure interpretation.

The **“BloodPressureThreshold”** and **“HeartRateThreshold”** support the “PatientRecord” class by defining how the readings of health indicators are performed. Their separation from the more central class “PatientRecord” ensures a distinct declaration and highlights their separateness.

**Data Storage System Requirements**

DataStorage class is responsible for storing patient data and managing it. Regarding privacy, only authorized pesonnel should have access to sensitive patient data, which is why the instances of the class remain mostly private. The DataStorage class interacts with instances of PatientData to store, retrieve and delete patient data. It is represented by the composition relation, because the class DataStorage doesn’t exist if we don’t have any patient data. DataRetriever allows the stuff to quickly access basic information about a patient, such as their name, date of birth or medical history. It has methods to search those attributes based on the patient’s unique ID, which makes the information easily accessible. Data storage can call PatientRecord’s methods, however not vice versa. That’s because PatientRecord does not need to interact directly with the data storage mechanism, as it merely provides the necessary details for identification and retrieval purposes. By that design, we ensure that patient identification details remain separate from the storage logic, enhancing encapsulation and protecting data integrity. The HistoricalDataAPI class serves as an interface for accessing historical patient data stored in DataStorage. DataStorage and AlertGenerator are concrete implementations of HistoricalDataAPI.

**Patient Identification System**

The Patient Identification System is responsible for making sure that data from various sources is correctly linked with a corresponding patient it concerns.

The **“PatientIdentifier”** class is the central class of this system. It contains a void method matchPatientID(), that uses another boolean helper method isMatched() to compare identifiers from incoming data to patient IDs in hospital’s database. It’s a void method, because it matches and links data without returning a value. When there is a match, it links the data to a right patient and communicates this information to another class called “DataStorage”, that can use it to start a process of adding data to patient’s history.

The **“IdentityManager”** class oversees the data matching process and hadles any occurring discrepancies. It has two methods: checkData() and hadleDiscrepancy(). CheckData() verifies, if all links made between data are correct, calling the second method if a mismatch occurs. The second method is programmed to manage the mistake. This process is crucial for maintaining data accuracy and patient safety.

The **“DataStorage”** class contains a PatientRecords HashMap and methods to store and retrieve objects of the “PatientRecord” class from the hospital’s database. A HashMap, where data can be quickly identified using patient’s ID, supports quick access and updates and easy retrieval.

The **“PatientRecord”** is a class that stores all relevant information about a patient (explained in detail above). And the **“DataStream”** class was added to simulate the reception of data that needs to be processed and matched.

**Data Access Layer**

The UML class diagram shows the data access later, starting with the DataListener class, which serves as a generic data listener. DataListener is responsible for actively listening for incoming data with a listen() method. It is an abstract class that provides a common base for different implementations tailored to specific data sources. WebSocketDataListener, FileDataListener and TCPDataListener are all concrete implementations of this abstract class. After receiving data from different sources, DataParser converts this raw data into a standardised format. Next, DataSourceAdapter **t**akes standardised data from DataParser and processes it, preparing it for storage in DataStorage. DataStorage stores processed data and provides methods for storing, retrieving, and deleting patient data. The relationships in the diagram are mostly dependencies, because in that way each class encapsulates its own functionality and relies on other classes to perform a specific task. For example, the DataSourceAdapter focuses on processing data, while the DataStorage is responsible for storing it. This separation enhances code readability, understandability, and maintainability.