UML - Diagram explanation

1.

The designed alert system has the task of checking patient health records and sending alerts to staff members if something seems dangerous.

With the `AlertGenerator` the patient's health records get checked with the use of personal predefined thresholds, that define, if an alert should be triggered. To get the patient's data the `DataStorage` is used and the `AlertRule` checks the conditions.

The personal `AlertRule's` are based on different health types (ECG, Blood Saturation, etc.), which are in different classes and represented as a strategy pattern (`ECGRule`, `BloodSaturationRule`, `BloodPressureRule`, `BloodLevelsRule`) to check if a patient's records are safe or not. This polymorphic design ensures that new health conditions can be simply added.

If a risk condition appears and an `Alert` occurs, the `StaffMember's` get notified by the `AlertManager`. The `AlertManager` has a list of all staff members, and the Staff Member class can receive alarms and can take action, when needed, which makes it possible to send alarms to them out.

Overall, the system is clearly split into three different parts, which include: storing the data (`DataStorage`), checking it for problems (`AlertGenerator` and `AlertRule`) and sending alert notifications (`AlertManager`). By structuring the system in this manner, it is easy to manage, understand and even add health data.

2.

The designed data storage system has the goal to safely store and manage the patient's health data.

The `DataStorage` class, which contains a map of patient Ids to patients, can be seen as the middle point of the system. It contains a map of patient Ids and a list of their `PatientRecord` entries. The entries contain information about the blood pressure, heart rate with timestamps.

Then, the `DataRetriever` lets staff members look up patient data, before that can happen, the `StaffMember` gets checked by a method `canAccess()` based on patient ID and record type, to identify, if the staff member is allowed to see the certain types of data.

The `StaffMember` class represents staff members in the hospital like doctors and nurses. Each `StaffMember` has a list of patients they are allowed to see, and they can receive alerts if something is wrong with those patients.
Additionally, the alerts are taken care of by a patient-specific `AlertRules` map, which links `recordType` to `AlertRule` objects. This is not visible in the UML-diagram but is being used by classes.

All in all, the system's class structure is logically separated into data storage, retrieval, authorization and alerting. This design simplifies maintenance and future expansion.

3.

The design begins by isolating domain data from infrastructure concerns. `PatientRecord` represents a single measurement—timestamp, type, and value—while the `Patient` aggregate groups multiple records in a 1..* containment. A separate `PatientDatabase` uses a `Map<int, Patient>` to encapsulate persistence, ensuring that retrieval, storage, or deletion logic remains confined to one class and can be swapped out (e.g., in-memory or SQL) with minimal ripple effects.

Identification and verification responsibilities are cleanly separated. The `PatientIdentifier` class is the sole component that queries `PatientDatabase` (1–1 association), incorporates retry logic (`maxRetries`), and manages a simple cache. On top of that, the `IdentityManager` delegates all lookup and validation logic: its `verifyPatient(id)`: boolean method invokes `PatientIdentifier.matchPatientId()`, and any failures are passed to a dedicated `MismatchHandler`. By funneling all entry-point checks through `IdentityManager`, higher layers gain a single, consistent API for patient resolution.

Error handling is centralized to guarantee auditability and maintain system resilience. `MismatchHandler`—stereotyped «utility»—handles both missing and invalid IDs, recording every event in a singleton `AuditLogger` (1–1 "logs to" dependency). This logger aggregates all anomalies in a `List<String>`, satisfying traceability requirements while keeping business logic uncluttered.

All associations are directed with arrowheads indicating "uses," open diamonds denote non-owning aggregation (e.g., `Patient` ◇→ `PatientRecord`), and multiplicities are clearly annotated (1, 0..*). This layered, single-responsibility approach balances flexibility (easy extension of lookup strategies), clarity (each class has one well-defined role), and robustness (centralized logging and mismatch recovery).

4.

In this layer, the `DataListener` interface abstracts the connection lifecycle for external sources. Three concrete listeners—`TCPDataListener, WebSocketDataListener, and FileDataListener`—realize it (solid line + hollow triangle). Each listener "is-a" DataListener, so new protocols can plug in without altering adapter logic. For example, FileDataListener watches a log file, emitting new lines via receive(), while TCP and WebSocket listeners manage socket streams.

The `DataSourceAdapter` sits at the heart of this module. It aggregates multiple DataListener instances (association, 0..* listeners) and depends on exactly one `DataParser` (dependency, 1–1). On startup, it `registerListener()` and calls each listener's `connect().` Incoming raw strings trigger the private `onRawData(raw:String)` callback, which in turn invokes `parseAndStore(raw:String).` This design cleanly separates "how" data arrives from "what" happens to it.

Parsing logic is encapsulated in the `DataParser` interface, implemented by `JSONParser` and `CSVParser` (dashed realization). Each parser's parse(raw:String):DataRecord standardizes incoming formats into `DataRecord` instances, which carry timestamp, patientId, type, and measurementValue. Finally, parsed records flow into `DataStorage` (aggregation, 1..* DataRecord), decoupling ingestion from persistence. This modular structure ensures single responsibility, easy extension for new listeners or parsers, and clear ownership of parsed health data.