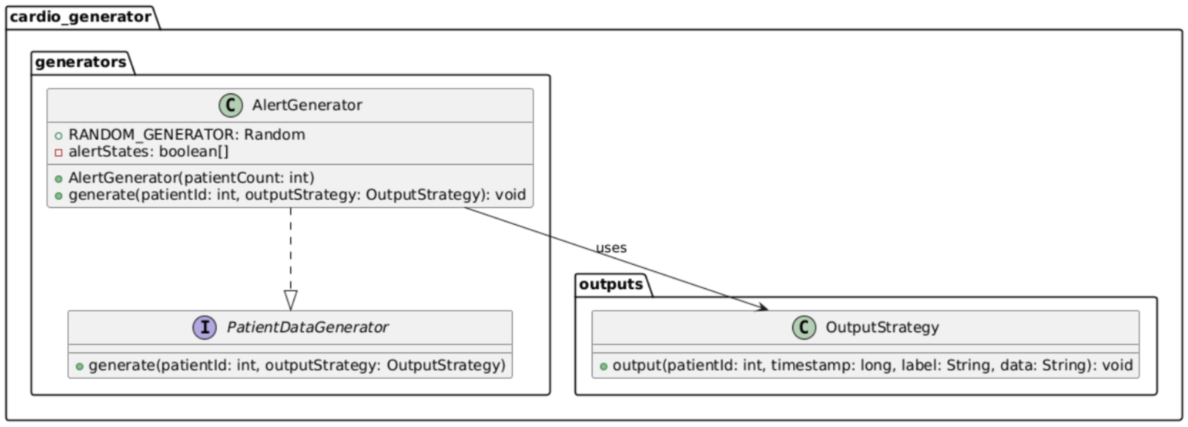
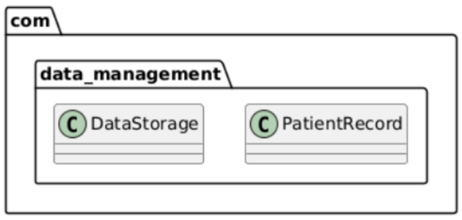
Alert Generator


cardio_generator
generators
AlertGenerator
RANDOM_GENERATOR: Random
alertStates: boolean[]
AlertGenerator(patientCount: int)
generate(patientId: int, outputStrategy: OutputStrategy): void
uses
PatientDataGenerator
generate(patientId: int, outputStrategy: OutputStrategy)
outputs
OutputStrategy
output(patientId: int, timestamp: long, label: String, data: String): void



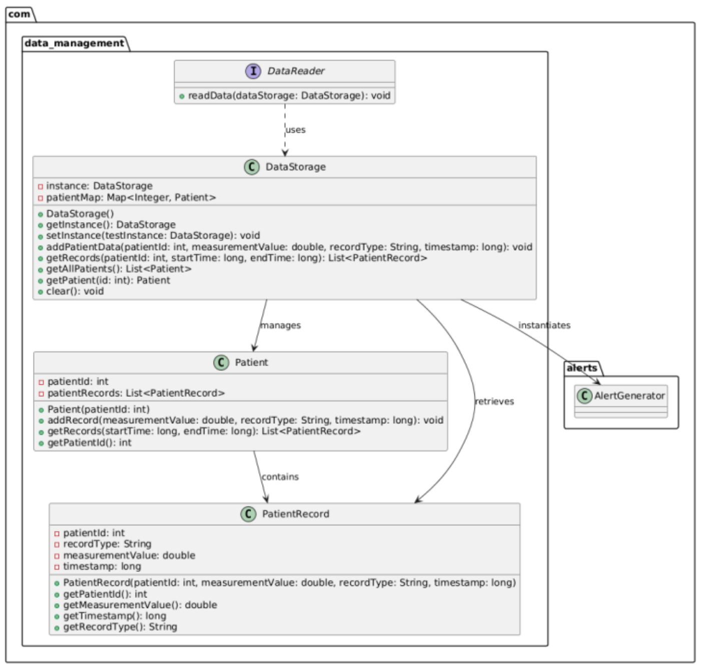
The Alert Generation System is the core of the CHMS that is persistently monitoring patient health information in real time and raising alerts upon detection of any critical condition. The system offers timely medical response through the detection of abnormalities and passing them on to the concerned healthcare personnel. The system has four major components: the AlertGenerator, AlertManager, Doctor, and Alert.

Everything begins with the AlertGenerator, which takes in patient data and compares it to a list of custom ThresholdRule objects. These define precise conditions under which alerts must be generated—for example, "heart rate above 130" or "oxygen saturation below 92%". If a rule is met, an Alert is created. If the patient has an existing alert already, then there is a 90% chance that the alert will be cleared in the next cycle; else, there is still a small possibility that a new alert will be triggered.

Each Alert object has the patient's ID, the cause of the alert, and a timestamp. The alert is forwarded to the AlertManager, which maintains a queue of active alerts and is responsible for sending them out. The alert is forwarded to the Doctor as soon as it is received, where it is evaluated and the proper medical response is initiated based on the condition of the patient.

By separating the data analysis task, the alert tracking task, and the medical response task neatly from each other, the Alert Generation System is scalable and modular. As a result, updating or extending the system is easy, and hospitals can adapt it easily to different monitoring needs while making sure that significant patient events are never missed.

Data Storage


com
data_management
DataReader
readData(dataStorage: DataStorage): void
uses
DataStorage
instance: DataStorage
patientMap: Map<Integer, Patient>
DataStorage()
getInstance(): DataStorage
setInstance(testInstance: DataStorage): void
addPatientData(patientId: int, measurementValue: double, recordType: String, timestamp: long): void
getRecords(patientId: int, startTime: long, endTime: long): List<PatientRecord>
getAllPatients(): List<Patient>
getPatient(id: int): Patient
clear(): void
manages
instantiates
retrieves
Patient
patientId: int
patientRecords: List<PatientRecord>
Patient(patientId: int)
addRecord(measurementValue: double, recordType: String, timestamp: long): void
getRecords(startTime: long, endTime: long): List<PatientRecord>
getPatientId(): int
contains
PatientRecord
patientId: int
recordType: String
measurementValue: double
timestamp: long
PatientRecord(patientId: int, measurementValue: double, recordType: String, timestamp: long)
getPatientId(): int
getMeasurementValue(): double
getTimestamp(): long
getRecordType(): String
alerts
AlertGenerator

The Data Storage System will store, retrieve, and manage all patient health information in a secure and organized manner. When receiving new readings—heart rate, blood pressure, or oxygen level, for instance—the DataStorage class links each item of data with the corresponding Patient based on their unique ID. If the patient does not already exist within the system, a new Patient object is created and added. Each Patient has a list of PatientRecord records, and each record contains the patient ID, the type of measurement, its value, and the timestamp.

In order to keep the system as streamlined and up-to-date, DataStorage also works together with a DeletionPolicy, an abstract class which supplies the mechanism for the removal of outdated records. A standard implementation is TimeBasedDeletionPolicy, in which records older than a specified time are removed automatically. This ensures that the system remains trim and up-to-date without keeping unnecessary data.

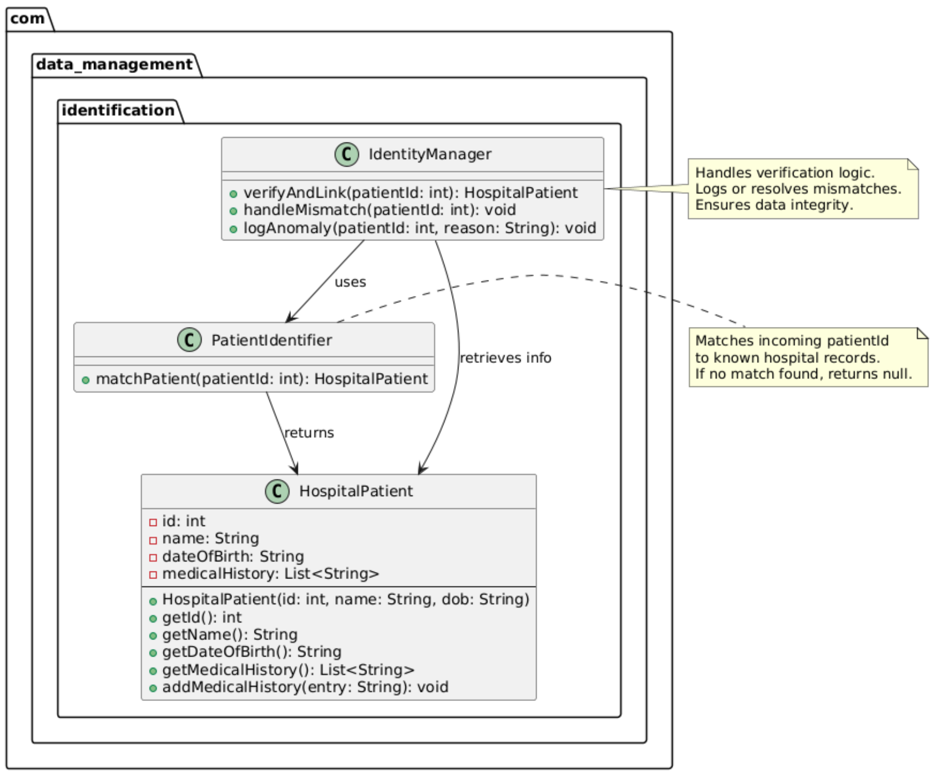
The AccessManager safely controls access to patient data. It keeps track of which users have permission to read or write specific patient records and blocks unauthorized personnel from viewing sensitive information. This access control maintains patient confidentiality and makes it easier to comply with data protection policies.

External data may be read into the system by implementations of the DataReader interface. This allows integration with various sources of data to enable populating the system with actual medical data.

The AlertGenerator is in direct contact with the stored data, continuously checking it for signs of medical concern—i.e., unsafe blood pressure or abrupt dips in oxygen saturation. When a problem is discovered, an alert is generated and forwarded to the appropriate medical staff, enabling quick and informed reactions.

Through real-time monitoring, automation of data cleanup, controlled access, and alert integration, the Data Storage System helps medical professionals track patient trends over time and respond quickly to life-threatening conditions, while ensuring accurate, secure, and manageable data.

Patient Identification System


com
data_management
identification
IdentityManager
+ verifyAndLink(patientId: int): HospitalPatient
+ handleMismatch(patientId: int): void
+ logAnomaly(patientId: int, reason: String): void
Handles verification logic.
Logs or resolves mismatches.
Ensures data integrity.
uses
retrieves info
PatientIdentifier
+ matchPatient(patientId: int): HospitalPatient
Matches incoming patientId
to known hospital records.
If no match found, returns null.
returns
HospitalPatient
- id: int
- name: String
- dateOfBirth: String
- medicalHistory: List<String>
+ HospitalPatient(id: int, name: String, dob: String)
+ getId(): int
+ getName(): String
+ getDateOfBirth(): String
+ getMedicalHistory(): List<String>
+ addMedicalHistory(entry: String): void

The "Patient Identification System" has the highest priority in ensuring each arriving patient health data is matched perfectly with the right patient. When new measurements come through—either real-time generators or batch imports—the system uses each patient's individual ID to match the data with their medical history. This ensures each patient receives each bit of data inserted correctly under them, keeping it accurate and safe for the clinician.

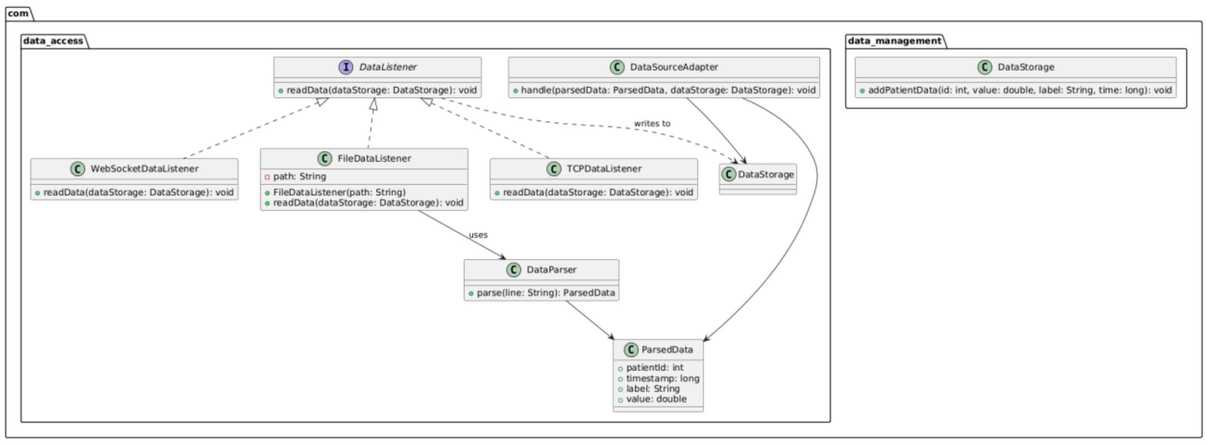
The IdentityManager is responsible for the matching operation, verifying each incoming patient ID against the PatientIdentifier. The module maintains a map of previously encountered patient data, thus can ensure the presence of the ID in the system in an efficient manner. All data for every patient are stored as PatientRecord objects, which contain key fields such as patient ID, type of measurement (e.g., "HeartRate"), value, and timestamp.

If there is a match, the system retrieves the corresponding HospitalPatient, linking the new data to the person's existing medical history. This makes available significant context like blood type, allergies, and chronic conditions, stored in the Patient class as a list of PatientRecord objects.

If there is no match, the IdentityManager forwards the issue to the MatchErrorHandler. This code logs the anomaly and provides structured functions for dealing with unmatched data. There are certain situations where the limit is relaxed to allow the limit to create a new patient entry so that no critical information gets lost.

When identified, the patient's data is saved in DataStorage and is included within their longitudinal health history. With this, AlertGenerator can determine their condition by considering all records available, hence enabling timely and accurate medical responses.

Data Access Layers


com
data_access
DataListener
readData(dataStorage: DataStorage): void
DataSourceAdapter
handle(parsedData: ParsedData, dataStorage: DataStorage): void
writes to
WebSocketDataListener
readData(dataStorage: DataStorage): void
FileDataListener
path: String
FileDataListener(path: String)
readData(dataStorage: DataStorage): void
TCPDataListener
readData(dataStorage: DataStorage): void
DataStorage
uses
DataParser
parse(line: String): ParsedData
ParsedData
patientId: int
timestamp: long
label: String
value: double
data_management
DataStorage
addPatientData(id: int, value: double, label: String, time: long): void

Data Access Layer is the interface of communications between external signal generators and internal systems of CHMS. It is responsible for accepting, interpreting, and transmitting incoming patient data in a format that is readable and processable for the rest of the system.

Signal generators may transmit the data via various channels, e.g., TCP connections, WebSocket streams, or file-based logs. For each input type, there exists a special listener class: TCPDataListener, WebSocketDataListener, and FileDataListener. All these listener classes have the same interface, DataListener, offering a generic listen() method. Such abstraction makes the system modular and flexible, and it is easy to add new input types without significant influence on the core architecture.

Each DataListener possesses a DataParser which converts raw data—whether JSON, CSV, or another format—and converts it into the standard PatientRecord object. This uniform structure enables consistent downstream processing regardless of how the data was received. The parsed data is forwarded to the DataSourceAdapter, which constitutes the interface between the access layer and the core system. The adapter calls the addPatientData() function of the DataStorage class, ensuring all records are stored correctly.

With concern separation of data input, parsing, and storage, the Data Access Layer allows for clean design and long-term maintainability. The Data Access Layer allows patient data to flow securely and reliably from the outside world into CHMS, supporting real-time monitoring and historical analysis alike.