1. Alert Generation System

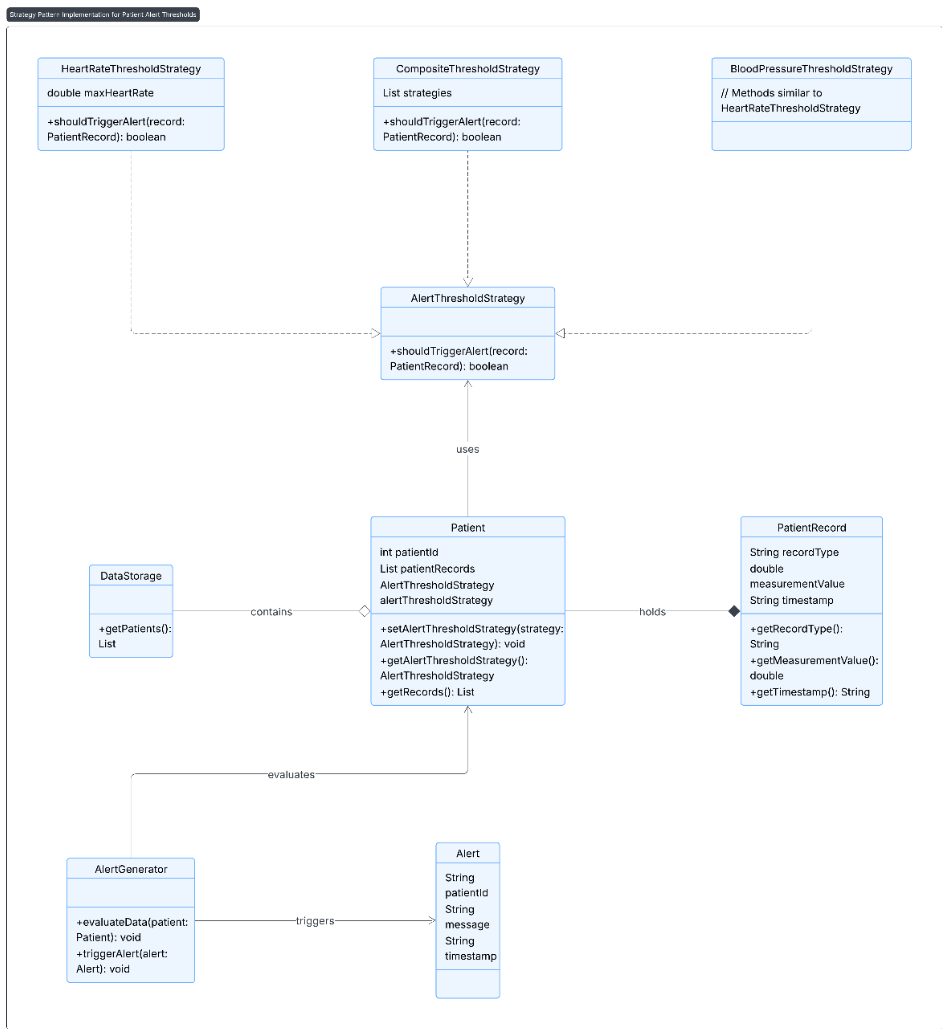
This UML diagram models the Alert Threshold System, which is responsible for evaluating patient data against predefined thresholds and triggering alerts when necessary. The design follows the Strategy Pattern, enabling flexibility and modularity in defining alert thresholds for different patients.

The central interface, AlertThresholdStrategy, defines the contract for evaluating whether a patient record should trigger an alert. Concrete implementations, such as HeartRateThresholdStrategy, BloodPressureThresholdStrategy, and CompositeThresholdStrategy, encapsulate specific threshold logic. The CompositeThresholdStrategy allows combining multiple strategies, making the system extensible for handling complex conditions.

The Patient class holds a reference to an AlertThresholdStrategy, enabling per-patient customization of alert logic. This ensures that each patient can have unique thresholds tailored to their medical needs. The PatientRecord class represents individual health measurements, such as heart rate or blood pressure, and is evaluated against the strategy.

The AlertGenerator class evaluates patient data by iterating through their records and using the assigned strategy to determine if an alert should be triggered. If a threshold is exceeded, an Alert object is created, encapsulating details such as the patient ID, condition, and timestamp.

This design ensures separation of concerns by decoupling the alert logic from the data storage and patient management systems. It is also highly extensible, as new alert strategies can be added without modifying existing code. The use of the Strategy Pattern ensures that the system is both flexible and maintainable.



Strategy Pattern Implementation for Patient Alert Thresholds
HeartRateThresholdStrategy
double maxHeartRate
+shouldTriggerAlert(record: PatientRecord): boolean
CompositeThresholdStrategy
List strategies
+shouldTriggerAlert(record: PatientRecord): boolean
BloodPressureThresholdStrategy
// Methods similar to HeartRateThresholdStrategy
AlertThresholdStrategy
+shouldTriggerAlert(record: PatientRecord): boolean
uses
DataStorage
+getPatients(): List
contains
Patient
int patientId
List patientRecords
AlertThresholdStrategy alertThresholdStrategy
+setAlertThresholdStrategy(strategy: AlertThresholdStrategy): void
+getAlertThresholdStrategy(): AlertThresholdStrategy
+getRecords(): List
holds
PatientRecord
String recordType
double measurementValue
String timestamp
+getRecordType(): String
+getMeasurementValue(): double
+getTimestamp(): String
evaluates
AlertGenerator
+evaluateData(patient: Patient): void
+triggerAlert(alert: Alert): void
triggers
Alert
String patientId
String message
String timestamp

2. Data Storage System

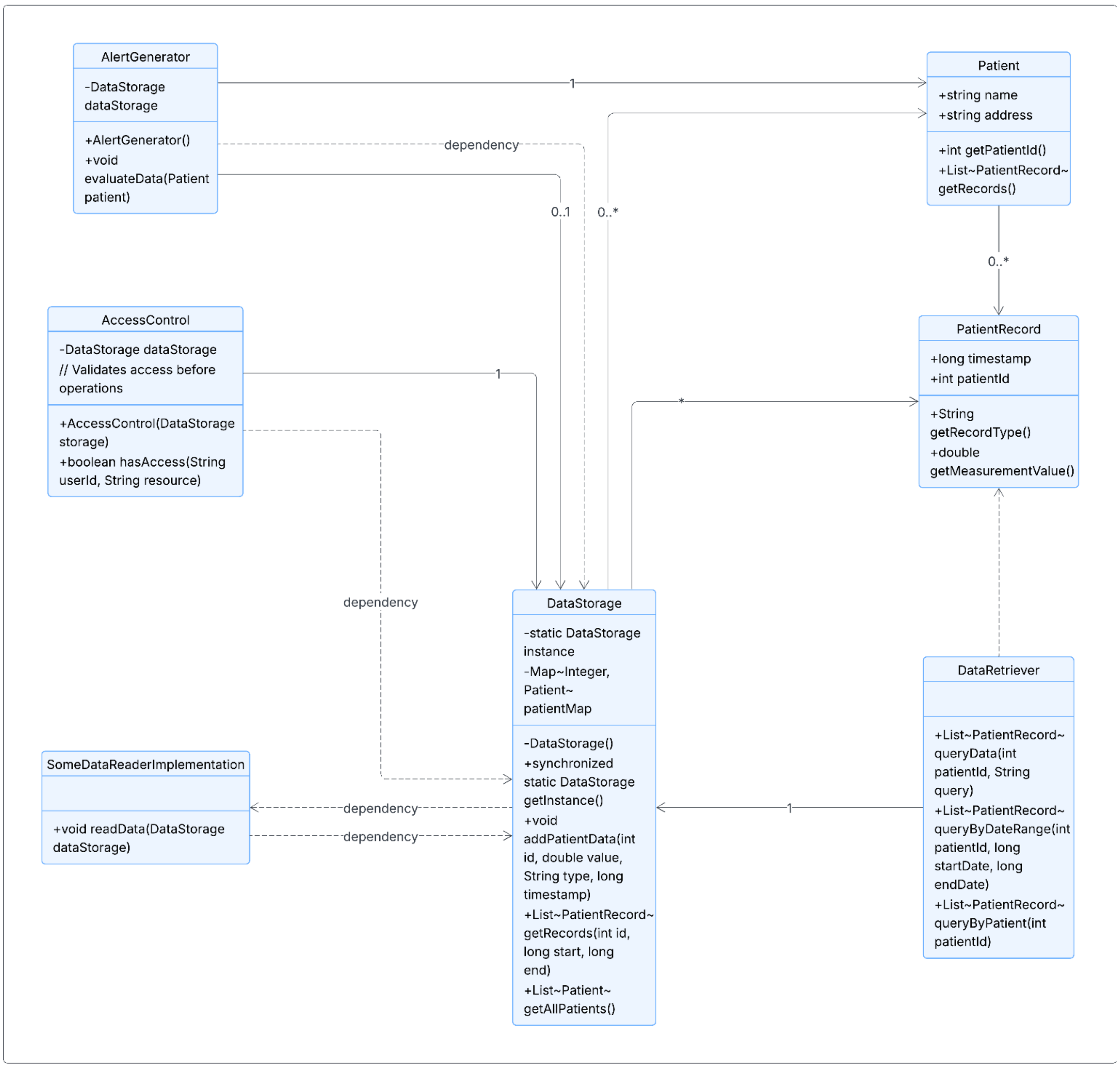
This UML diagram represents the Data Storage System and its integration with Access Control. The DataStorage class serves as the central repository for patient data, implementing the Singleton Pattern to ensure a single instance is used throughout the system. It stores patient records and provides methods for adding, retrieving, and managing data.

The AccessControl class enforces security by checking permissions for read and write operations. It ensures that only authorized users can access or modify patient data, addressing privacy and security concerns. The DataRetriever class interacts with DataStorage to query patient data based on specific criteria, such as time ranges or patient IDs.

The PatientRecord class represents individual health data points, while the Patient class aggregates these records. The relationship between DataStorage and Patient is modeled as a composition, indicating that DataStorage owns and manages the lifecycle of Patient objects.

This design emphasizes modularity and security. By separating data storage, retrieval, and access control, the system ensures that each component has a clear and focused responsibility. The Singleton Pattern in DataStorage guarantees consistency and prevents duplication, while the inclusion of AccessControl addresses critical privacy requirements.

Implementing Singleton Pattern for DataStorage


AlertGenerator
-DataStorage dataStorage
+AlertGenerator()
+void evaluateData(Patient patient)
Patient
+string name
+string address
+int getPatientId()
+List~PatientRecord~ getRecords()
1
dependency
0..1
0..*
0..*
AccessControl
-DataStorage dataStorage
// Validates access before operations
+AccessControl(DataStorage storage)
+boolean hasAccess(String userId, String resource)
1
PatientRecord
+long timestamp
+int patientId
+String getRecordType()
+double getMeasurementValue()
*
dependency
DataStorage
-static DataStorage instance
-Map~Integer, Patient~ patientMap
-DataStorage()
+synchronized static DataStorage getInstance()
+void addPatientData(int id, double value, String type, long timestamp)
+List~PatientRecord~ getRecords(int id, long start, long end)
+List~Patient~ getAllPatients()
DataRetriever
+List~PatientRecord~ queryData(int patientId, String query)
+List~PatientRecord~ queryByDateRange(int patientId, long startDate, long endDate)
+List~PatientRecord~ queryByPatient(int patientId)
1
SomeDataReaderImplementation
+void readData(DataStorage dataStorage)
dependency
dependency

3. Patient Identification System

This UML diagram models the Patient Identification System, which links incoming data to the correct patient by matching patient IDs with hospital records. The design ensures accurate identification and handles mismatches or anomalies effectively.

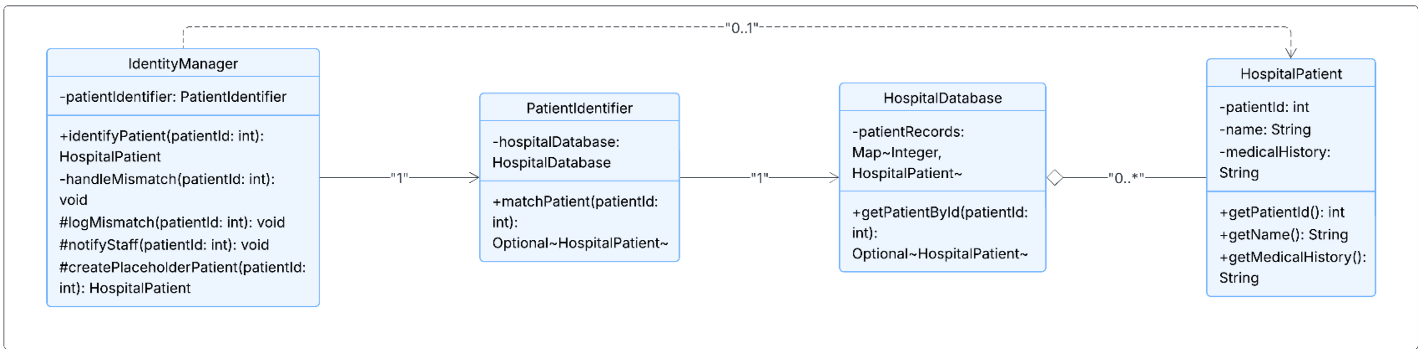
The IdentityManager class oversees the identification process. It uses the PatientIdentifier class to match incoming patient IDs with records in the HospitalDatabase. If a match is found, the corresponding HospitalPatient object is returned. In case of a mismatch, the IdentityManager invokes the handleMismatch method to address the issue, such as logging the error or notifying administrators.

The HospitalDatabase class stores patient records in a Map for efficient lookup by patient ID. Each record is represented by a HospitalPatient object, which contains details such as the patient's name and medical history.

This design ensures separation of concerns by delegating specific responsibilities to distinct classes. The PatientIdentifier focuses on matching logic, while the IdentityManager handles higher-level operations and error management. The use of a Map in HospitalDatabase ensures efficient data retrieval, making the system scalable for large datasets.

By addressing edge cases like mismatches, this design ensures robustness and reliability in linking data to the correct patient, which is critical for maintaining data integrity in a healthcare system.

UML Diagram for Patient Identification System


"0..1"
IdentityManager
-patientIdentifier: PatientIdentifier
+identifyPatient(patientId: int): HospitalPatient
-handleMismatch(patientId: int): void
#logMismatch(patientId: int): void
#notifyStaff(patientId: int): void
#createPlaceholderPatient(patientId: int): HospitalPatient
"1"
PatientIdentifier
-hospitalDatabase: HospitalDatabase
+matchPatient(patientId: int): Optional~HospitalPatient~
"1"
HospitalDatabase
-patientRecords: Map~Integer, HospitalPatient~
+getPatientById(patientId: int): Optional~HospitalPatient~
"0..*"
HospitalPatient
-patientId: int
-name: String
-medicalHistory: String
+getPatientId(): int
+getName(): String
+getMedicalHistory(): String

4. Data Access Layer

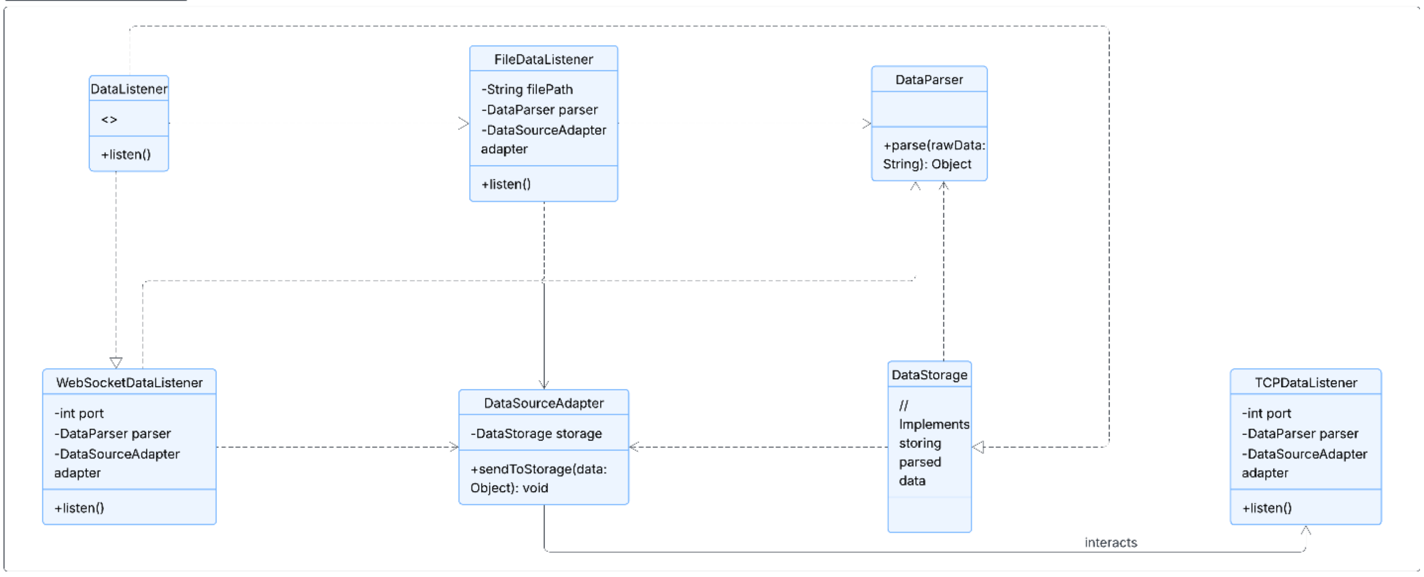
This UML diagram models the Data Access Layer, which connects the external world (e.g., signal generator) to the system by retrieving and parsing data into a standardized format. The design ensures modularity and flexibility by using the Adapter Pattern and a shared interface for data listeners.

The DataListener interface defines a common contract for all data sources. Its concrete implementations—TCPDataListener, WebSocketDataListener, and FileDataListener—handle data input from TCP connections, WebSocket streams, and log files, respectively. Each listener depends on a DataParser to standardize raw input data (e.g., JSON, CSV) into a usable format.

The DataSourceAdapter acts as a bridge between the parsed data and the DataStorage system. It ensures that parsed data is handed off to DataStorage in a consistent manner. The DataStorage class, implemented as a Singleton, stores the parsed data for further processing.

This design emphasizes separation of concerns by decoupling data retrieval, parsing, and storage. Adding a new data source (e.g., HTTP) requires only a new implementation of DataListener, without modifying existing code. The use of the Adapter Pattern ensures that the system remains flexible and extensible, while the centralized DataParser promotes code reuse and maintainability.

UML Diagram for Data Access Layer of CHMS



DataListener
<>
+listen()
FileDataListener
-String filePath
-DataParser parser
-DataSourceAdapter adapter
+listen()
DataParser
+parse(rawData: String): Object
WebSocketDataListener
-int port
-DataParser parser
-DataSourceAdapter adapter
+listen()
DataSourceAdapter
-DataStorage storage
+sendToStorage(data: Object): void
DataStorage
// Implements storing parsed data
TCPDataListener
-int port
-DataParser parser
-DataSourceAdapter adapter
+listen()
interacts