TrackMe project Giorgio Cozza and
Barbara Ferretti

# Data4Help and AutomatedSOS

## Design Document

| | |
|---:|---|
| **Deliverable:** | DD |
| **Title:** | Data4Help and AutomatedSOS - Design Document |
| **Authors:** | Giorgio Cozza, Barbara Ferretti |
| **Version:** | 1.0 |
| **Date:** | 10-December-2018 |
| **Download page:** | https://github.com/GiorgioCozza/CozzaFerretti.git |
| **Copyright:** | Copyright © 2018, Giorgio Cozza and Barbara Ferretti – All rights reserved |

# Contents

## List of Figures

# 1 Introduction

## 1.1 Purpose

This Design Document will provide an overall definition of the main system components and the relative interactions, an in-depth discussion about the chosen architectural patterns and the plan for integration, verification and validation steps. The audience for this artifact is tighter than the one of the RASD due to the more technical language. In particular, it is oriented to all the people involved in engineering the software-to-be. In any case, in order to facilitate the comprehension of this document a previous RASD consultation is strongly suggested.

## 1.2 Scope

As already explained in the previous RASD, Data4Help and AutomatedSOS are physically separated applications, since even if they handle the same set of data types, they concern with different purposes. The first one represents a privacy-guaranteed service to manage third party requests of access to user-provided health data, the second is a service conceived to ensure first assistance whenever dangerous conditions for registered users are detected. The discussion at this point should be focused specifically on the architecture behind these two services, analyzing the way both applications will collect health data from the monitoring devices, how TrackMe servers will manage such data and more. To this end, since TrackMe wants to not depend strictly on external cloud infrastructures for commercial and data-security reasons, it decided to host its own server farm completely. For the moment the system will be centralized, but easily distributable for future needs, according to the growth factor of the customer number. This in order to form a private cloud infrastructure built on top of a distributed database system.

## 1.3 Definitions, Acronyms, Abbreviations

### 1.3.1 Definitions

**User:** both Data4Help and AutomatedSOS users.
**Third Party:** more frequently are companies that looks for health information for their market research, but they could also be facilities like hospitals or other activities interested in the health field.
**Anomaly condition:** alteration of health monitored parameters that causes an alert to the user.
**Emergency condition:** alteration of the health monitored parameters that directly triggers the sent of an ambulance.
**Supervisor user:** every user that has access to another user information and location.
**Sensor/Monitoring Device:** all the health sensor devices that can be registered in the Data4Help system.
**Supervised User:** an AutomatedSOS user who is monitored by the application and has one or more Supervisors.
**ASOS Monitor Mode:** flag that tells the system if the user must be monitored or not, it can be changed from client-side application.
**Emergency Resource Manager:** is a system already implemented in all the hospitals subscribed to the AutomatedSOS service, it handles emergency calls and AutomatedSOS notifications. For each case manages the resources required by the emergency.

**RescueSquad:** general term used to refer a registered group of rescuers provided with the equipment required for the specific type of emergency.

### 1.3.2 Acronyms

**ASOS:** AutomatedSOS
**D4H:** Data4Help
**API:** Application Programming Interface
**RASD:** Requirement Analysis Specification Document
**HW:** Hardware
**BLE:** Bluetooth Low Energy
**TLS:** Transport Layer Security
**HTTPS:** HTTP over TLS
**ERM:** Emergency Resource Manager
**TP:** Third Party

### 1.3.3 Abbreviations

**[Gn]:** n-th Goal
**[Rn]:** n-th Requirement

## 1.4 Revision History

Up to now, there are no revision of this document as this is the first release.

## 1.5 Reference Documents

- Mandatory Project Assignment AY 2018-2019
- *Requirements Analysis and Specification Document*, version 2.0

## 1.6 Document Structure

### 1.6.1 Chapter 1

In the first chapter is given a general presentation of the aim of this document. Moreover are given other basic information in order to read easily the entire work, like the dictionary for example.

### 1.6.2 Chapter 2

In the second chapter the entire architecture is defined, starting from an high-level view of the system.

### 1.6.3 Chapter 3

In chapter 3, the GUI system is clearly represented by the UX diagrams. A short description about the adopted notation is provided in the beginning of the relative section.

### 1.6.4 Chapter 4

This chapter is dedicated to explain how the architecture components are mapped with the requirements defined in the previous artifact.

### 1.6.5 Chapter 5

This part is dedicated to discuss about implementation, integration and testing strategies and plans for the further stages.

# 2 Architectural Design

## 2.1 Overview

Both Data4Help ad AutomatedSOS are complex services due to the large set of technologies involved in their functionalities. To have a full comprehension of the overall architecture, it is necessary to abstract the entire system, starting the analysis from the customer perspective. Here, in fact, the main challenge is related to device management and data representation, since a large variety of sensors (supported by the service), can be connected and send data simultaneously. Each health indicator sample provided to the app should be processed at first in order to match a common data structure and then sent to the server. Since a D4H user is automatically an ASOS user (once the first login is performed), it may happen that both applicationswill be used in tandem. This leads to a problem of data redundancy, since the server could receive two samples from the same monitoring device. It has been chosen to avoid either the effort foo the server to analyze and discard in real time duplicated samples or to store identical dataset parts in the database system: as anticipated in the introduction, both the services provided by TrackMe for several reasons will rely on the same dataset for each monitored user. The solution to the mentioned problem is based on a messaging system operating between the apps and the server to synchronize the monitoring sessions and avoid any conflict. The mechanism will be discussed later. Since client and server architectures are characterized by some different aspects and responsibilities, both of them will be defined and carefully analyzed in this document. An high level view of the entire system, anyway, can be provided by the following picture:

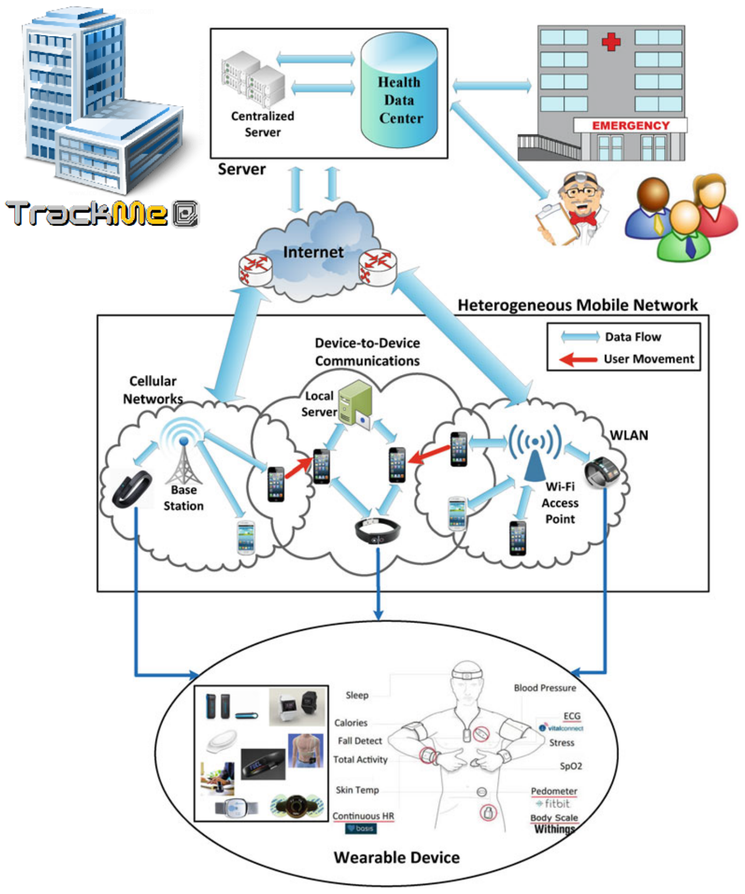

Figure 1: High-level View of the System

## 2.2 Component View

In this section is provided a brief description of the common components in D4H and ASOS apps for what concerns hardware interaction and about the mechanism that regulates the messages and data exchange between the client and the server. In the analysis of server-side architecture in subsection *2.2.4*, the component view has been splitted in two parts in order to help the comprehension of the way the architecture works. The first part is related to user data collection and health status observation, the second one presents the functionalities that deal mostly with Data4Help mansions (device and third party request management, privacy and security constraint evaluator and so on).

### 2.2.1 Common Components

Starting from the client-side architecture, there are common aspects between the two applications for what concerns the hardware interaction part and activity synchronization. Respectively, in the first case the system provides:

- **DeviceProxy:** a software interface for the control of each specific device. It implements different methods to directly interact through the proper API with the real device. It encapsulates also all the essential information about the smart sensor;
- **DeviceManager:** that handles with all the connected devices. It is in charge of:
  - instantiate a device when it is required;
  - activate and deactivate a single device when an event occurs;
  - give information about the status of the device set.
- **Data Collector:** that implements all the algorithms required to perform data collection according to specific patterns.

For both client and server architectures, in order to synchronize all the activities carried on by the system, a mechanism of message exchange has been designed. This includes:

- **EventObserver:** that mediates the interaction among the manager components (all those components responsible of managing a specific set of functionalities in the system) and the server;
- **EventHandling:** an interface implemented by manager components. It includes the essential methods required to send notifications when a particular event occurs and to perform specific activities when messages are received by other manager components and the server (through the EventObserver).

### 2.2.2 Data4Help Client Components View

Specifically to Data4Help on client side, the component in addition to the above mentioned is the **DataRequestManager**, it implements all those basic functionalities that allow the user to receive and evaluate third party requests.

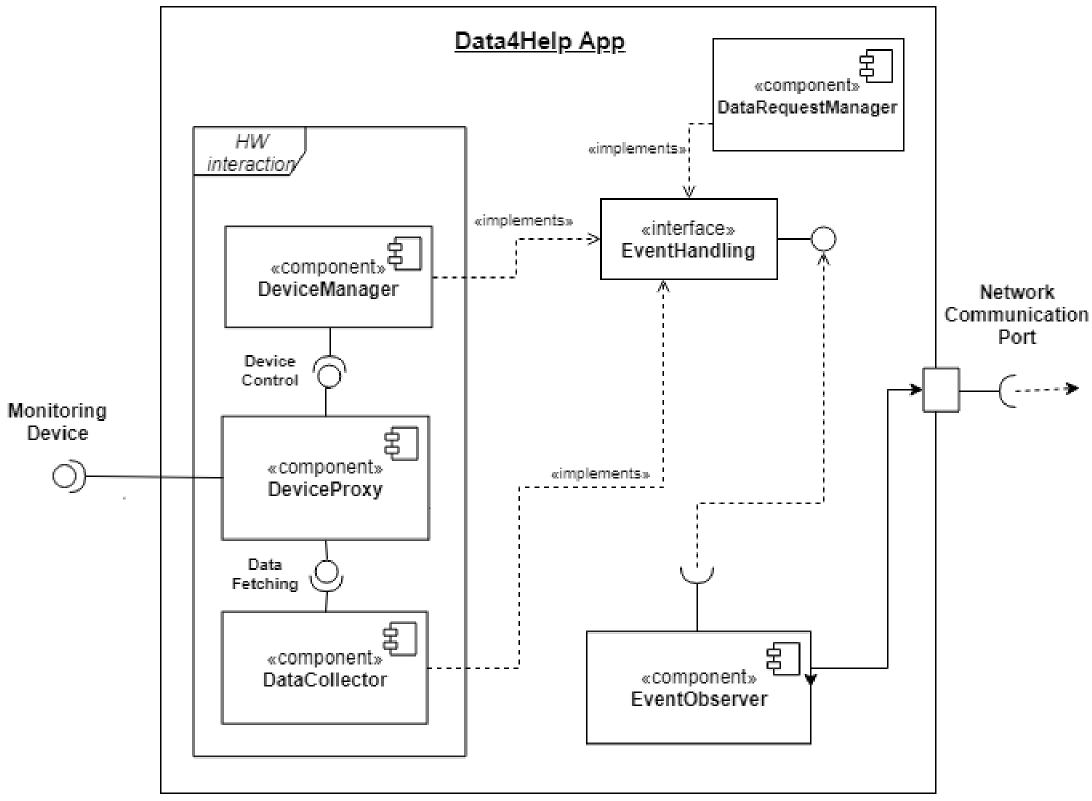
Data4Help App
«component» DataRequestManager
«implements»
HW interaction
«component» DeviceManager
«implements»
«interface» EventHandling
Device Control
Monitoring Device
«component» DeviceProxy
«implements»
Network Communication Port
Data Fetching
«component» DataCollector
«component» EventObserver

Figure 2: D4H Client App, Component Diagram

### 2.2.3 AutomatedSOS Client Component View

The ASOS application is characterized in addition by:

- **UserHealthProfileManager:** that implements all those operations required to create and modify a health profile for the user. It is a basic component, the information provided by the patient is requested by the server to understand which health parameters to watch over;
- **SupervisorRequestManager:** that deals with Supervisor tasks from the client perspective. It is used to keep track of the supervised users or to send, receive and manage external requests of supervision.

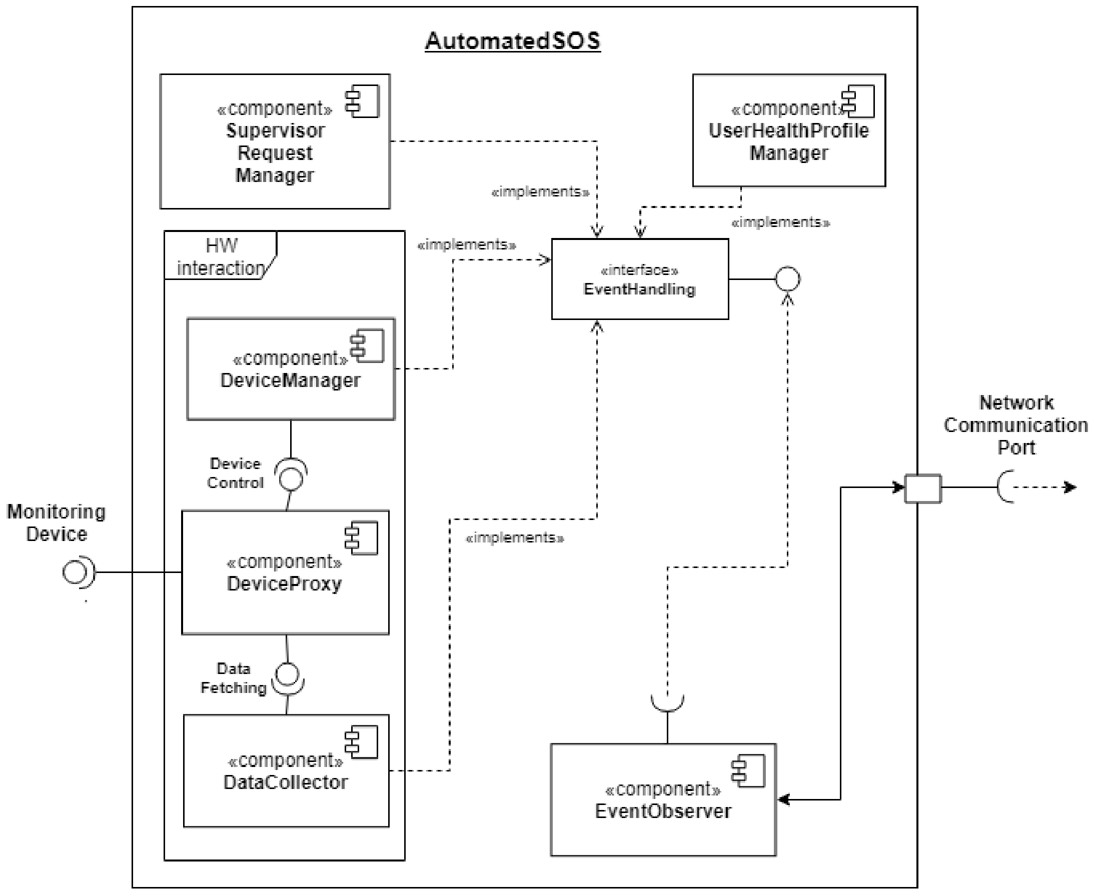
AutomatedSOS
«component» Supervisor Request Manager
«component» UserHealthProfile Manager
«implements»
«implements»
«implements»
«interface» EventHandling
HW interaction
«component» DeviceManager
Device Control
Monitoring Device
«component» DeviceProxy
Data Fetching
«component» DataCollector
«implements»
Network Communication Port
«component» EventObserver

Figure 3: ASOS Client App, Component Diagram

### 2.2.4 Server Side Component View

From the server perspective the situation is different. The application logic is almost completely deployed in this part of the system. In order to facilitate the comprehension of the architecture, the component diagram has been splitted.
As said, Data4Help and AutomatedSOS performs different tasks but they work on the same dataset, so the part that deals with data collection must be considered as common to both the applications that in some other cases will communicate exclusively with specific components.
Before describing the architecture it is needed to introduce some key concepts. In particular, the time in which the system collects and provides personal data from user monitoring devices will be divided in intervals, each one called *session*. A session, from server perspective, can be thought as a group of data queues with a proper ID, associated to all the health indicators monitored from a specific user and a data stream directly connected to a component called, Analyzer. Since for real-time requirements ASOS needs to process each received sample as fast as possible, a software component will dispatch a copy of each value in a data stream (at first) and consequently in the proper queue. The Analyzer, in charge of processing data to monitor user condition, will receive data only from the stream. Periodically and according to the frequency of sample arrivals, each queue will be stored in the database.
The following high-level diagram clearly represents the described mechanism.

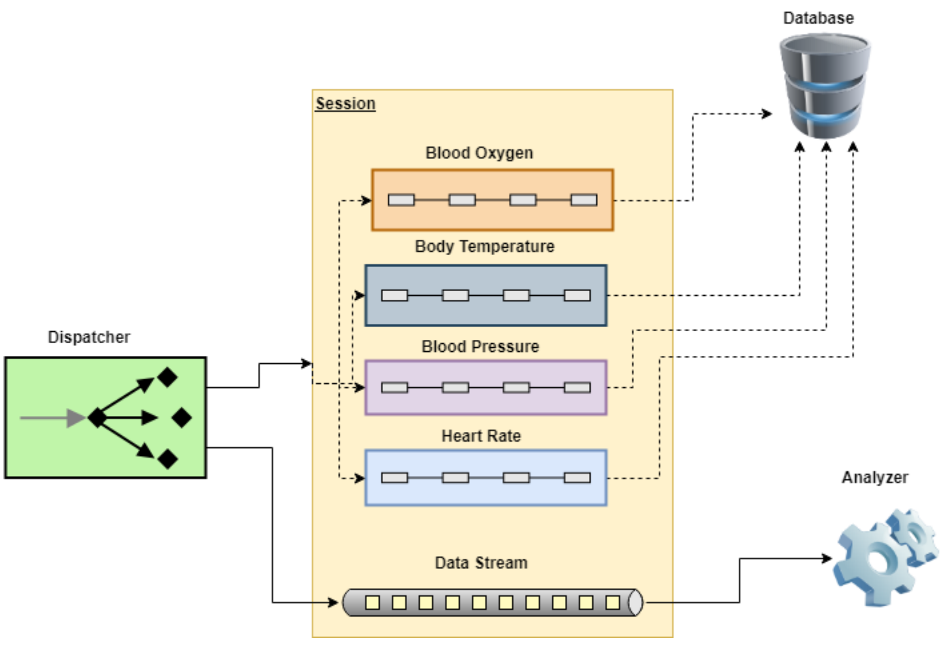

Figure 4: High-level View of Session Management

Other software artifacts regarding this side of the system are:

- **SessionManager:** that is responsible to create and manage each session. In particular it will handle the situations in which the apps running in tandem will require a context switch for the current session;
- **Analyzer:** one of the most complex parts of the architecture, implements all the algorithms and mechanisms required to process ASOS data stream and establish with appreciable certainty the situation of the user;
- **EmergencyCondition:** that encapsulates all the information regarding a detected urgency (health status, user, location and so on), for which immediate assistance is required;

- **AnomalyCondition:** that encapsulate the mechanisms and information to handle a suspicious state of urgency;
- **SupervisorManager:** in charge of managing supervisor operations at server side;
- **ERMManager:** an interface component that allows the system to communicate with external ERMs;
- **Resource:** a part of the architecture that is used to keep track of the resource status instantiated by the ERM, once dispatched;
- **DatabaseManager:** a common element of both the diagrams, it will provide an interface efficiently implemented to the database system.

Here is the first part of the server side component:

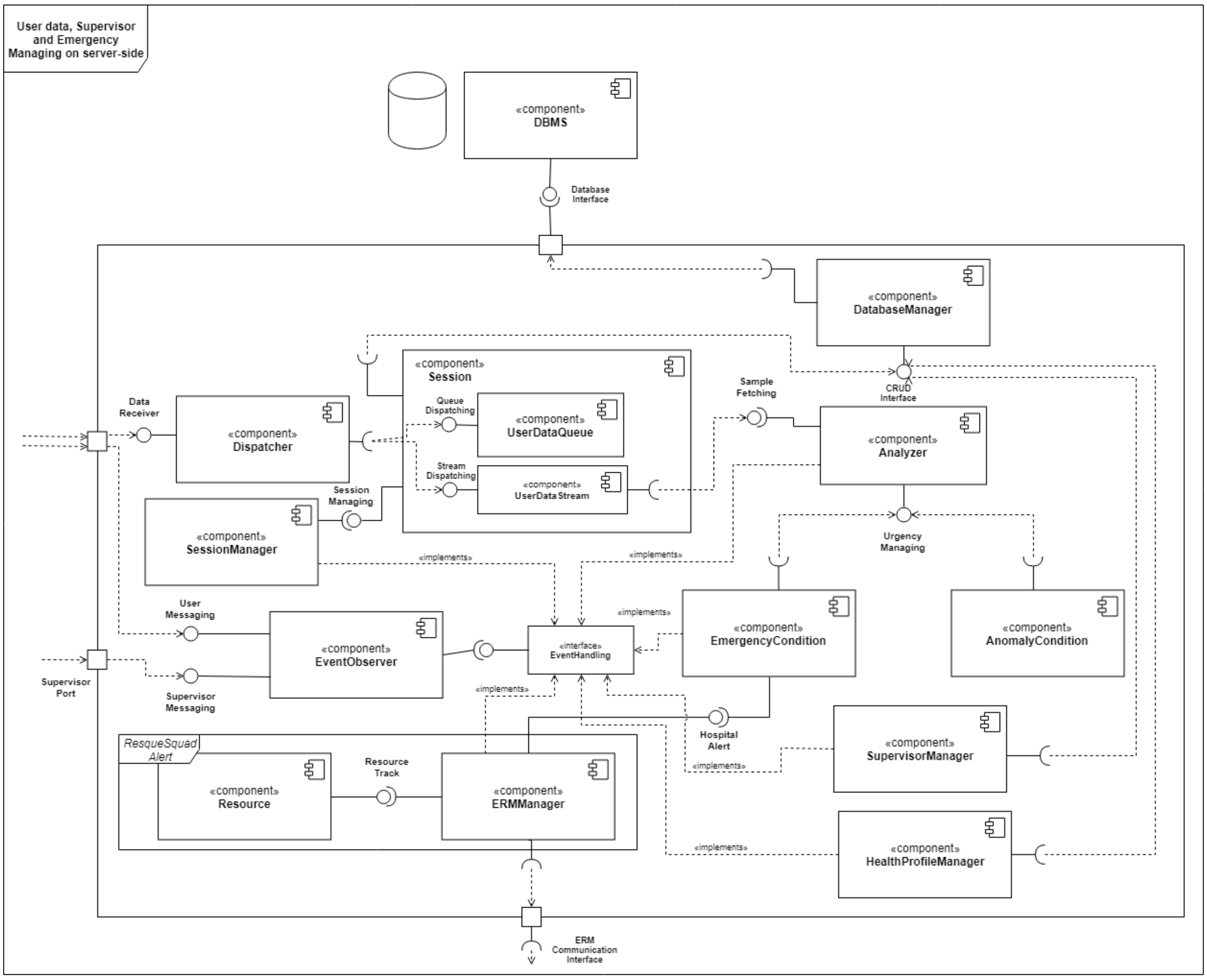
User data, Supervisor and Emergency Managing on server-side
«component» DBMS
Database Interface
«component» DatabaseManager
CRUD Interface
«component» Session
Queue Dispatching
«component» UserDataQueue
Stream Dispatching
«component» UserDataStream
Data Receiver
«component» Dispatcher
Session Managing
«component» SessionManager
Sample Fetching
«component» Analyzer
Urgency Managing
«implements»
«component» EmergencyCondition
«component» AnomalyCondition
User Messaging
«component» EventObserver
«interface» EventHandling
Supervisor Port
Supervisor Messaging
Hospital Alert
«component» SupervisorManager
ResqueSquad Alert
«component» Resource
Resource Track
«component» ERMManager
«component» HealthProfileManager
ERM Communication Interface

Figure 5: Emergency Handling and User Monitoring on Server Side, Component Diagram

Of course, both the communication system composed by the EventObserver and the EventHandling and the DatabaseManager should be considered as common components in the two component diagrams.

In the following diagram is shown the second part of the server architecture, that is more related to D4H functionalities.
It is characterized by:

- **DeviceHandler:** to manage system supported devices (add, remove and modify smart sensors that are supported by the services) and allow users to associate devices to their account through the registration mechanism;
- **Login:** to authenticate users and third parties;
- **TP_RequestManager:** to process and manage third party requests from the server perspective. This component encapsulates also the methods required to allow third parties to create and manage all the requests and to allow the target users to receive such requests as soon as they are formulated;
- **PrivacyAndSecurityEvaluator:** to enforce an access control policy to user datasets, checking privacy constraint violations for search criteria specified in anonymous requests and to certificate third party identity and reliability;
- **DatasetGenerator:** to generate anonymous or specific user datasets from evaluated and accepted requests.

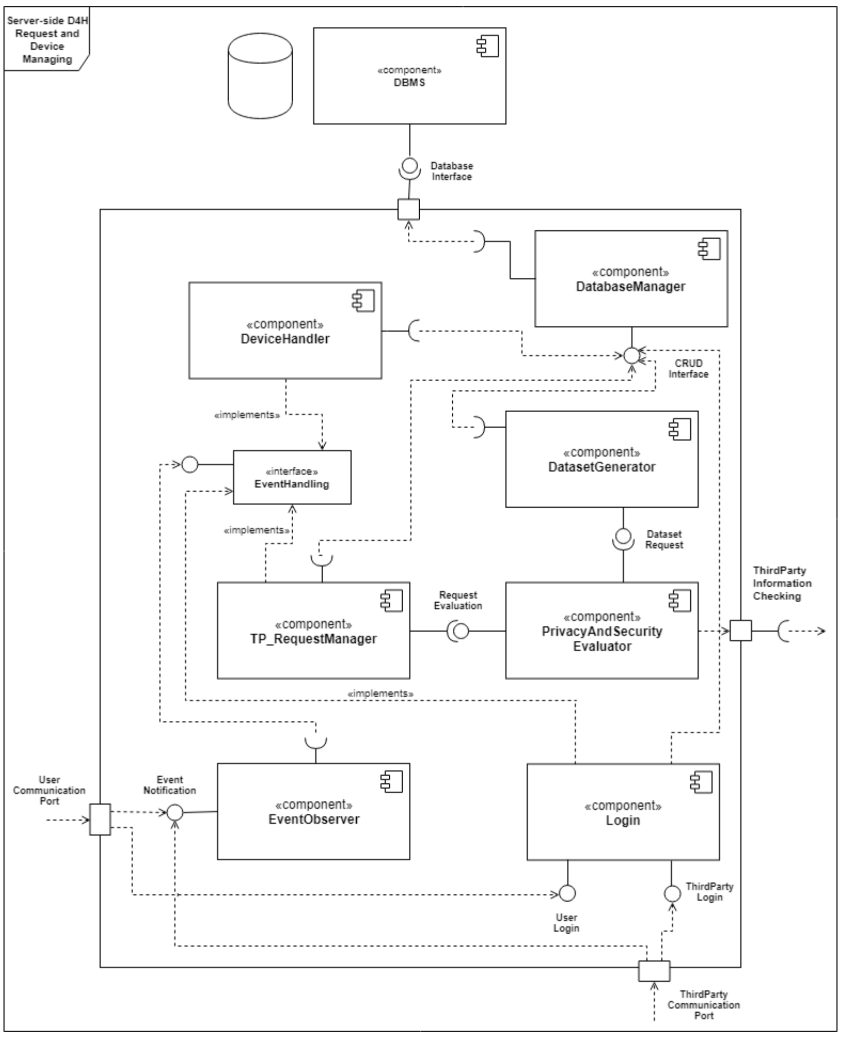
Server-side D4H Request and Device Managing
«component» DBMS
Database Interface
«component» DatabaseManager
«component» DeviceHandler
CRUD Interface
«implements»
«interface» EventHandling
«component» DatasetGenerator
«implements»
Dataset Request
ThirdParty Information Checking
Request Evaluation
«component» TP_RequestManager
«component» PrivacyAndSecurity Evaluator
«implements»
User Communication Port
Event Notification
«component» EventObserver
«component» Login
User Login
ThirdParty Login
ThirdParty Communication Port

Figure 6: Device and TP Request Managing on Server Side, Component Diagram

As explained before, the reason for event notification through message exchanging between client and server is due to the need to synchronize system activities . A **Message** can be thought in the design context as an element that generalizes two concepts:

- an **Action**
- an **Event**

The first one is used when an architectural component must communicate a state change to the rest of the system. When the notification needs to be sent to the server, the EventObserver, that is the mediator of each communication, is called by the notifier component (e.g: DeviceManager), it creates a Message and sends it to the server side. When a message is received by the opposite observer, this can be turned either into an Event, if something is required to be communicated to other components, or into an Action, if some operation must be performed. The EventObserver establishes which component is the receiver, calls the method *onReceive* (implemented by each manager component through the interface **EventHandling**) and passes as parameter an Action. In any case, all these elements are characterized by the same format:

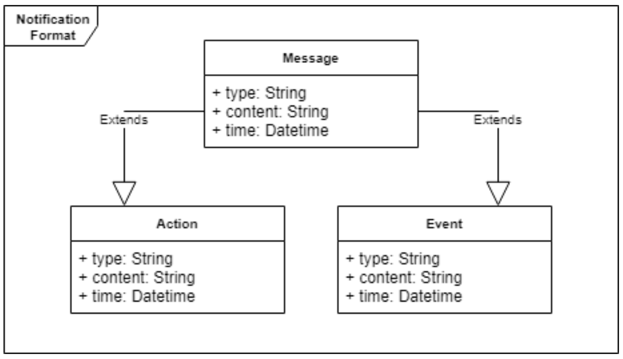

Figure 7: Notification Format

The *type* is used by the EventObserver to understand which is the message receiver, the *content*, to establish which is the operation that the specific component must perform after being notified and the *time* should be used for further implementation of a time scheduling functionality for such component. In many case this mechanism can be useful:

- When a user triggers a conflict event such as:
  - He/She turns on ASOS when Data4Help is running. In this case the current session is suspended to allow the system to change the context (stream some health parameters used by D4H to the analyzer), notifying the need to switch off some devices in the D4H app in order to not transmit sample copies;
  - He/She attempts to register/unregister devices when ASOS Monitor Mode is switched on. In this case the server should avoid this condition in order to not raise unexpected conflicts. A warning must be shown to the client;
  - He/She tries to change the own health profile while ASOS is running in MonitorMode.
- During normal activities, such as:
  - Register/Unregister a device;
  - Notify a decision on a specific request (ThirdParty/Supervisor);
  - Notify changes in health profile;
  - Notify emergency communication (from both the parts);
  - Other notifications.

### 2.2.5 Entity-Relationship Diagram

The following diagram represents the database model of the defined architecture:

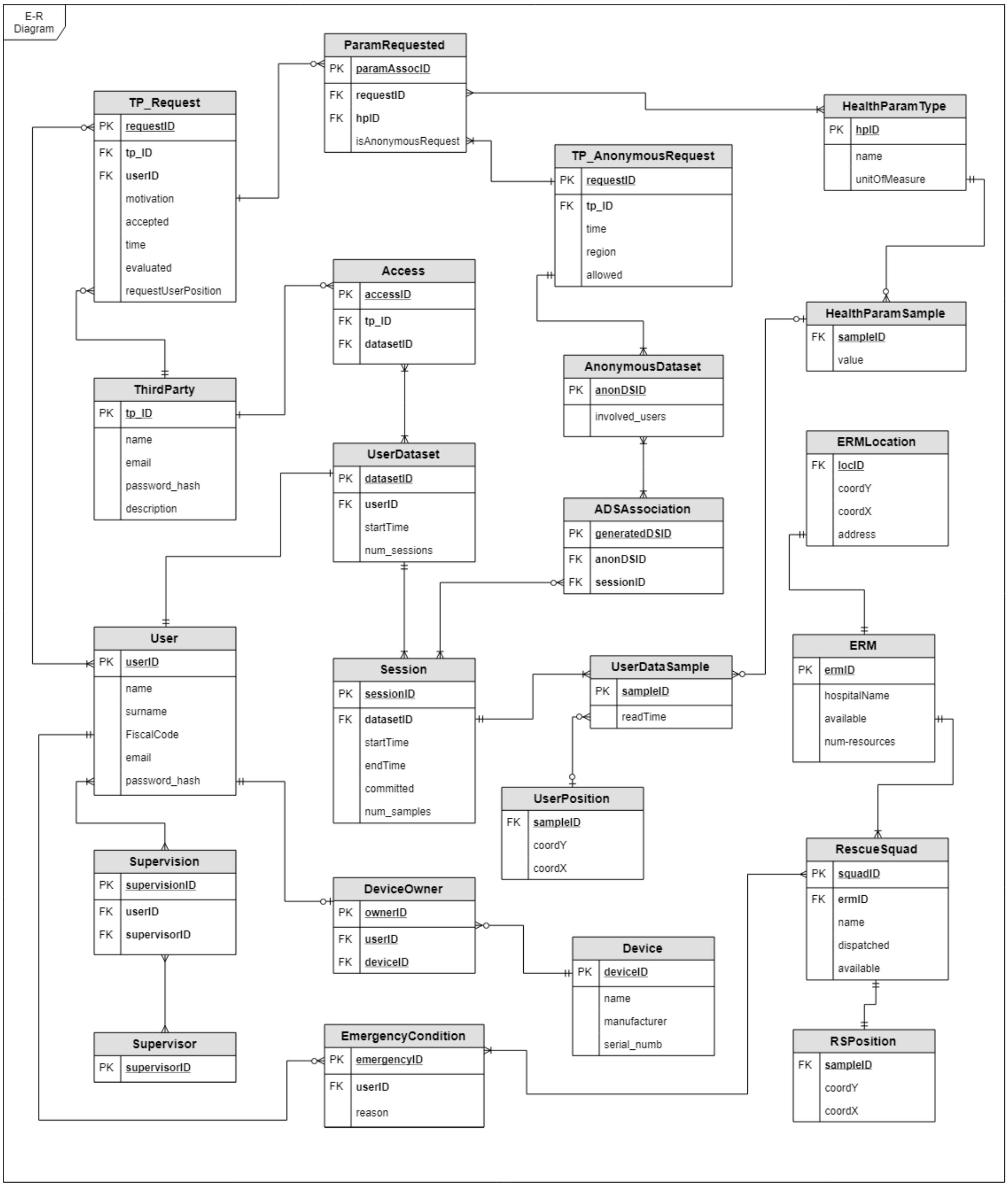

Figure 8: E-R Diagram for Database Modeling

## 2.3 Deployment View

The deployment diagram represents the displacement of all the physical components involved in the system, how the application logic is inserted in relative execution environments and the communication with the external services (ERM system and ThirdParty database).
In the entire system deployment, the following components are involved:

- **Application tier:** in which the system artifacts interact among them in their respective execution environments, in particular:
  - **Data4Help** and **AutomatedSOS**: that interacts with the application server, by data and message exchange using HTTP over TLS;
  - **ThirdParty Web-Browser:** used to allow companies and organizations to access the server logic in order to formulate requests, download datasets, manage their own profile.
- **External Service tier:** composed by the external services with relative APIs and the system logic components required to interact with them, also in this case over an encrypted channel:
  - **ERMSystem:** running on the ERMServer, that is in charge of manage the Rescue Squads (in this document called Resources), it provides the interface required to the ASOS server application to interact with it;
  - **ThirdPartyDatabaseServer:** on which the system relies to retrieve information about each company and organization that wants to register or formulate a request to the service.
- **Database tier:** that includes all those logic and physical components used to interface persistent data to the server application (for security reasons through a firewall);
- **SmartSensor:** connected physically to the smartphone using a communication interface such as Bluetooth, BLE, NFC and controlled by the applications through the Android OS.

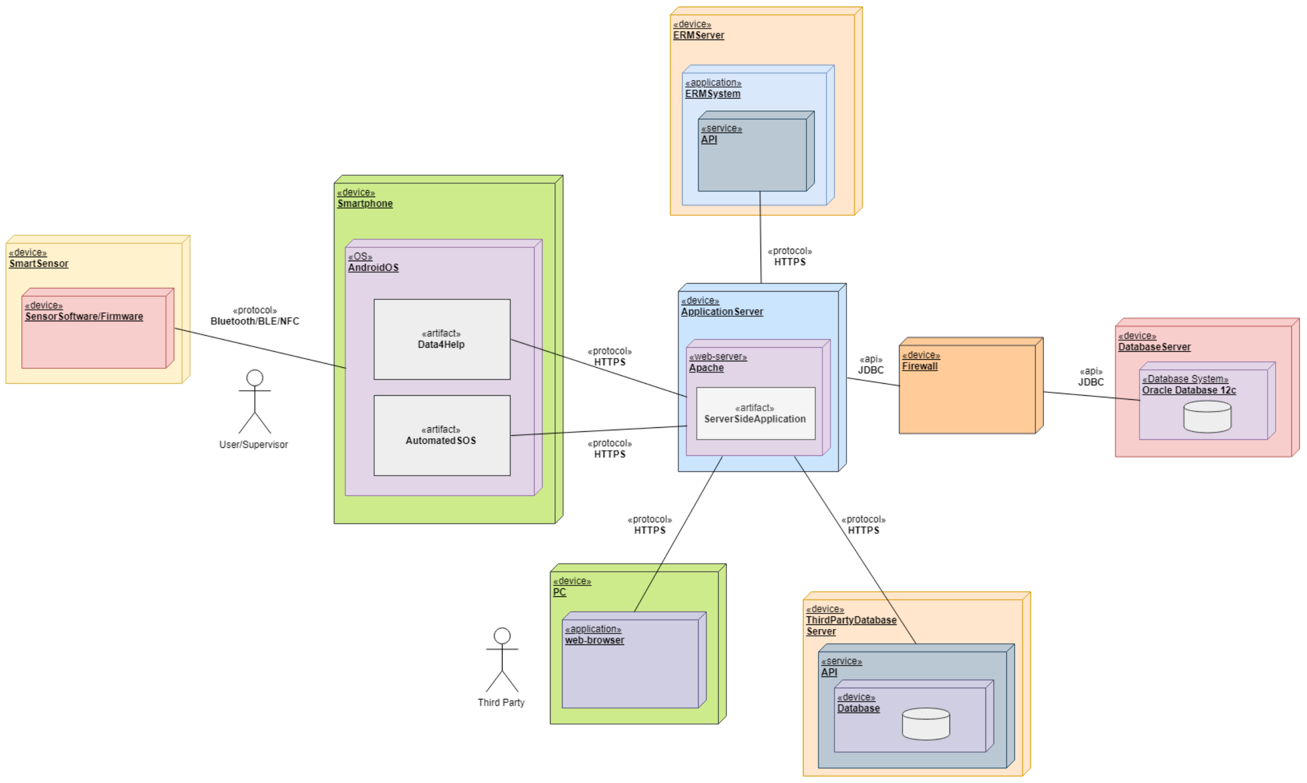

Figure 9: Deployment Diagram

## 2.4 Runtime View

The Runtime View provides the interaction sequence for four use cases also mentioned in the RASD document (see *Section 3.5* in the *Requirements Analysis and Specification Document*). No description is provided since the detail level by which the interactions are represented, is enough to understand the event flows.

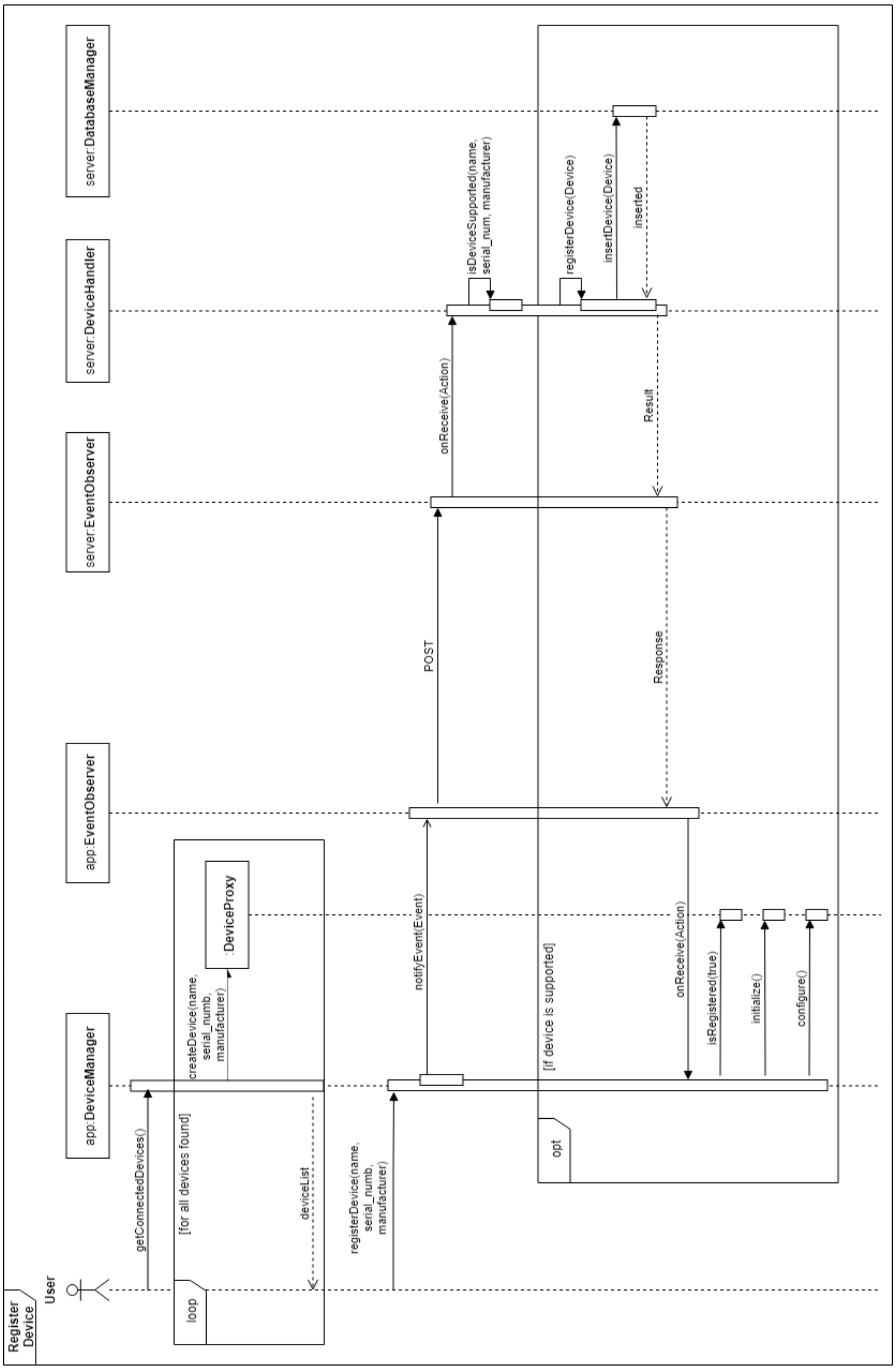

Figure 10: Register Device, Sequence Diagram

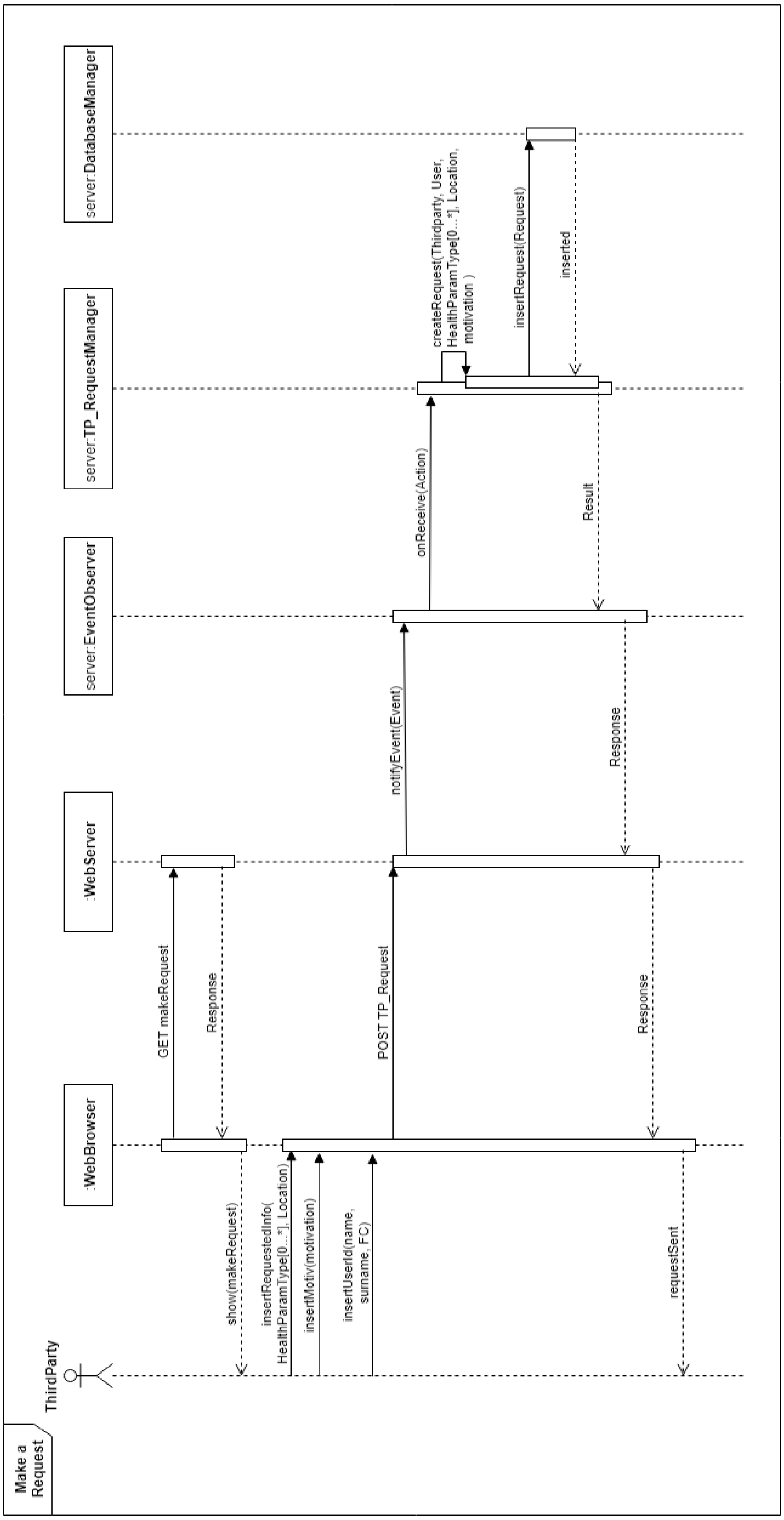

Figure 11: Make a Request, Sequence Diagram

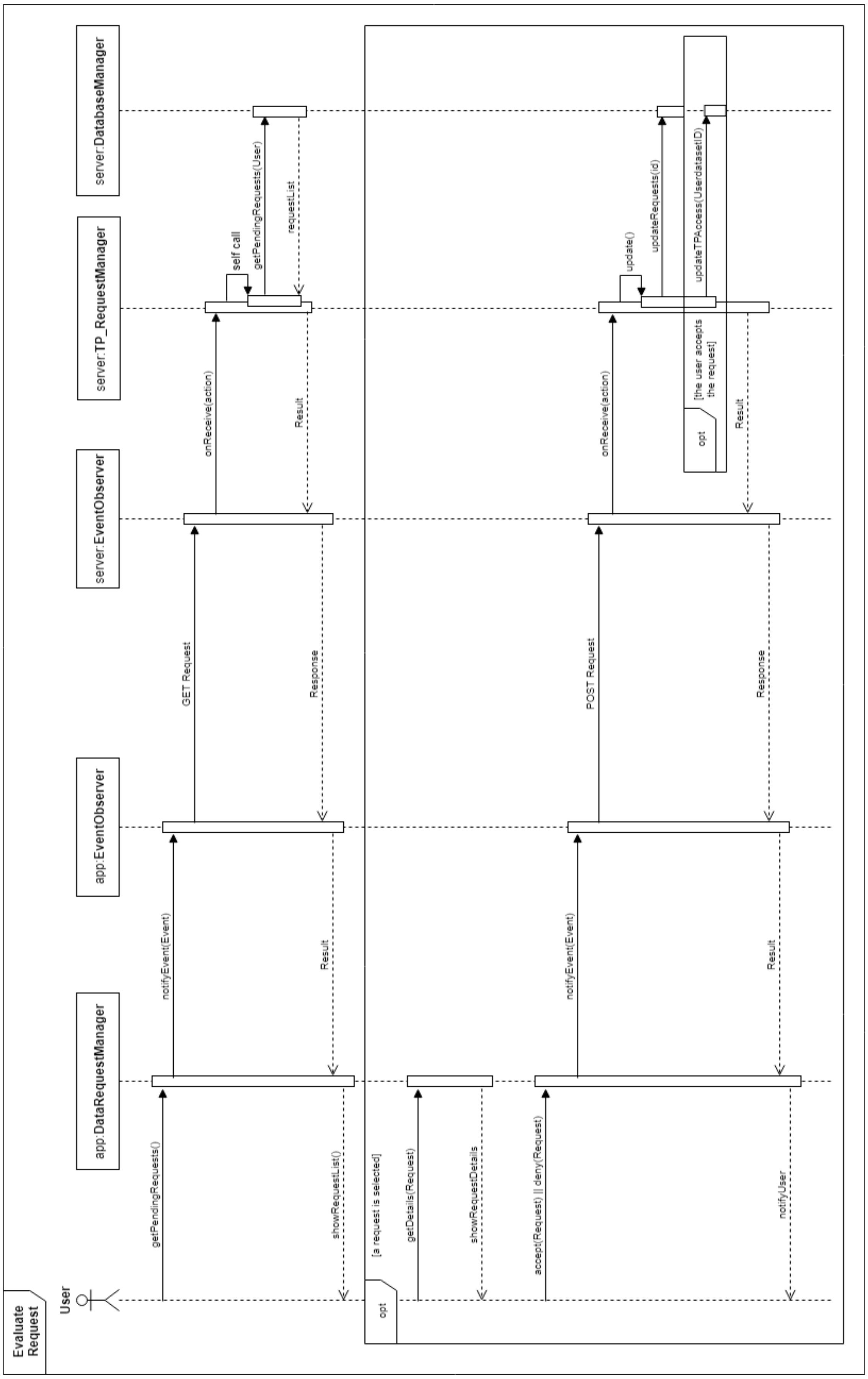

Figure 12: Evaluate a Request, Sequence Diagram

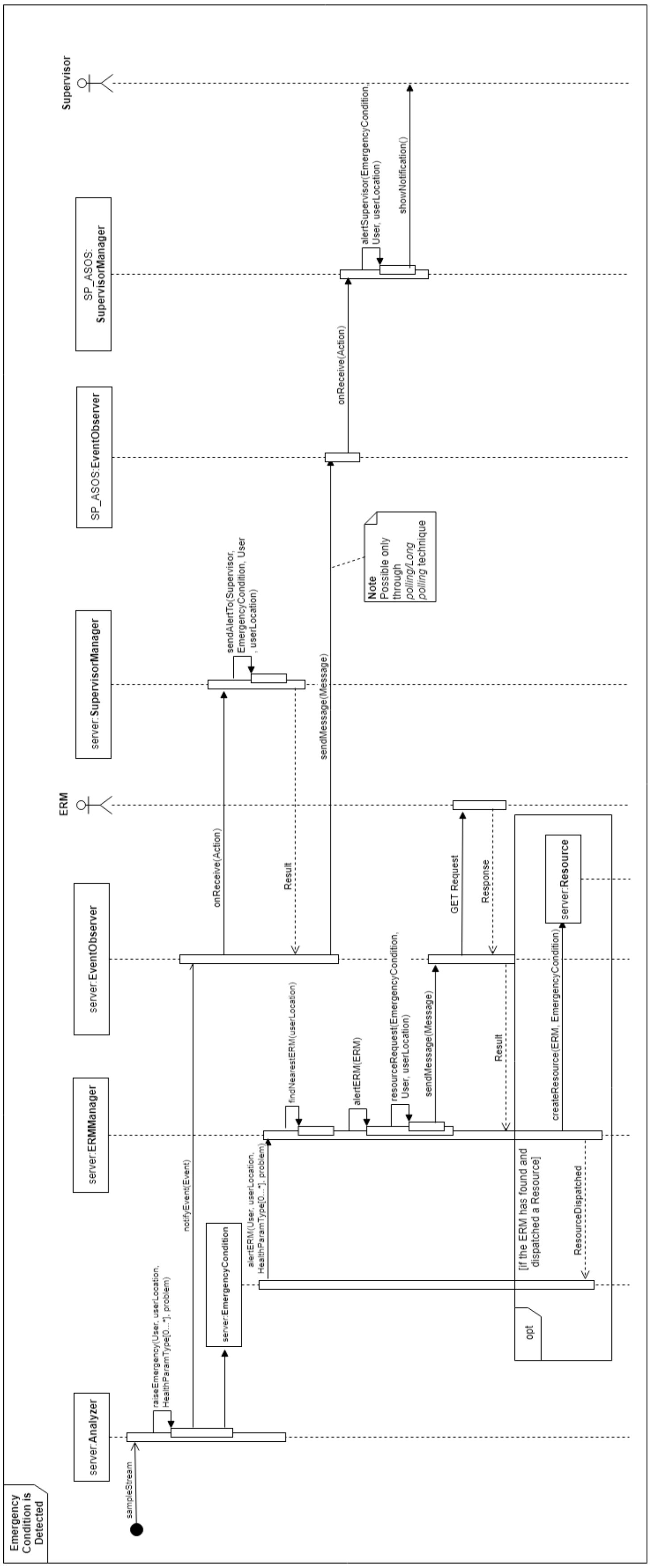
Emergency Condition is Detected
server:Analyzer
server:EmergencyCondition
server:ERMManager
server:EventObserver
ERM
server:SupervisorManager
SP_ASOS:EventObserver
SP_ASOS: SupervisorManager
Supervisor
sampleStream
raiseEmergency(User, userLocation, HealthParamType[0...*], problem)
notifyEvent(Event)
alertERM(User, userLocation, HealthParamType[0...*], problem)
findNearestERM(userLocation)
alertERM(ERM)
resourceRequest(EmergencyCondition, User, userLocation)
sendMessage(Message)
GET Request
Response
Result
createResource(ERM, EmergencyCondition)
server:Resource
ResourceDispatched
opt
[if the ERM has found and dispatched a Resource]
onReceive(Action)
Result
sendAlertTo(Supervisor, EmergencyCondition, User , userLocation)
sendMessage(Message)
Note
Possible only through polling/Long polling technique
onReceive(Action)
alertSupervisor(EmergencyCondition, User, userLocation)
showNotification()

Figure 13: Emergency Condition, Sequence Diagram

## 2.5 Component Interfaces

This section highlights the dependencies among the various interfaces in the system. As in the previous case, it has been made for both client and server architectures.

### 2.5.1 D4H Client Component Interfaces

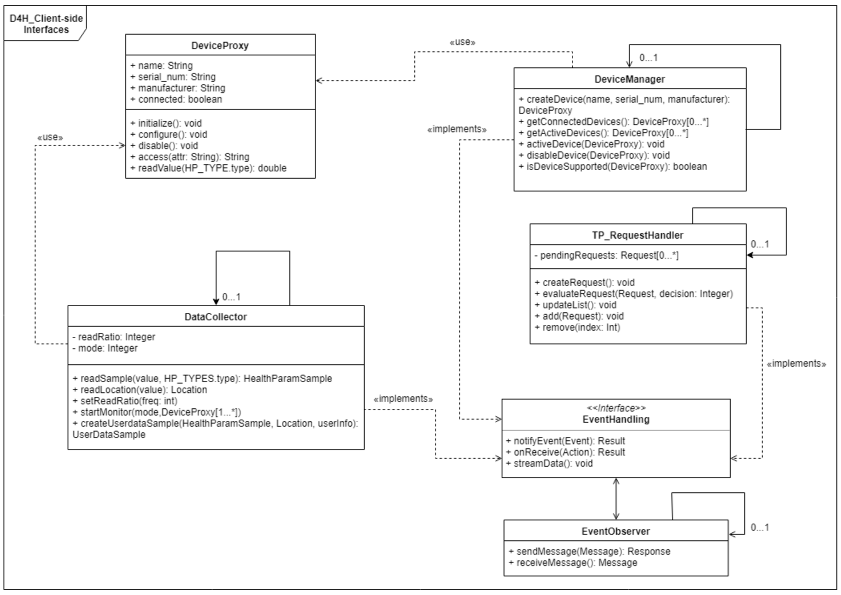
D4H_Client-side Interfaces
DeviceProxy
+ name: String
+ serial_num: String
+ manufacturer: String
+ connected: boolean
+ initialize(): void
+ configure(): void
+ disable(): void
+ access(attr: String): String
+ readValue(HP_TYPE.type): double
«use»
«use»
0...1
DeviceManager
+ createDevice(name, serial_num, manufacturer): DeviceProxy
+ getConnectedDevices(): DeviceProxy[0...*]
+ getActiveDevices(): DeviceProxy[0...*]
+ activeDevice(DeviceProxy): void
+ disableDevice(DeviceProxy): void
+ isDeviceSupported(DeviceProxy): boolean
«implements»
TP_RequestHandler
- pendingRequests: Request[0...*]
+ createRequest(): void
+ evaluateRequest(Request, decision: Integer)
+ updateList(): void
+ add(Request): void
+ remove(index: Int)
0...1
0...1
DataCollector
- readRatio: Integer
- mode: Integer
+ readSample(value, HP_TYPES.type): HealthParamSample
+ readLocation(value): Location
+ setReadRatio(freq: int)
+ startMonitor(mode,DeviceProxy[1...*])
+ createUserdataSample(HealthParamSample, Location, userInfo): UserDataSample
«implements»
«implements»
<<Interface>>
EventHandling
+ notifyEvent(Event): Result
+ onReceive(Action): Result
+ streamData(): void
EventObserver
+ sendMessage(Message): Response
+ receiveMessage(): Message
0...1

Figure 14: D4H Client App, Interface Dependencies

### 2.5.2 ASOS Client Component Interfaces

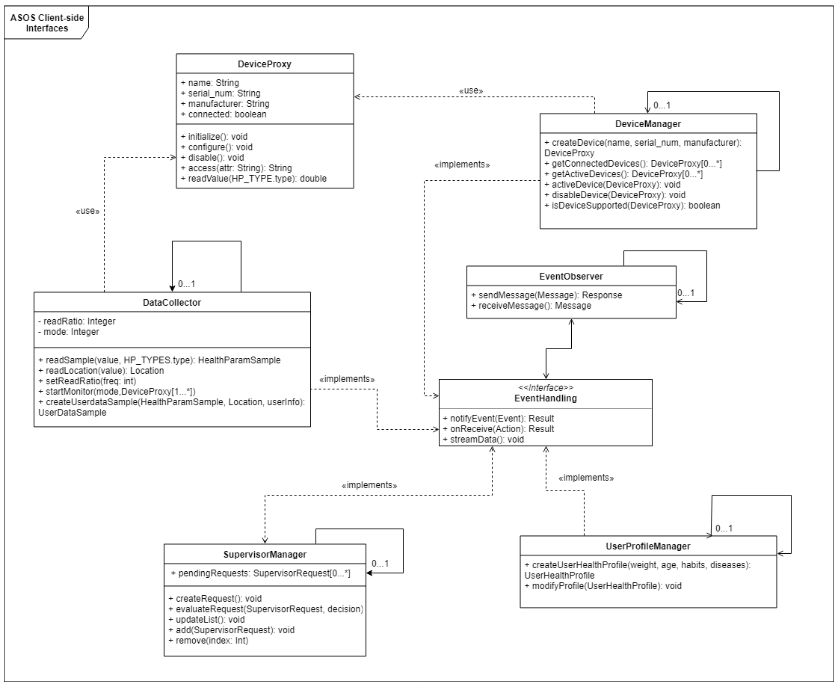
ASOS Client-side Interfaces
DeviceProxy
+ name: String
+ serial_num: String
+ manufacturer: String
+ connected: boolean
+ initialize(): void
+ configure(): void
+ disable(): void
+ access(attr: String): String
+ readValue(HP_TYPE.type): double
«use»
«use»
0...1
DeviceManager
+ createDevice(name, serial_num, manufacturer): DeviceProxy
+ getConnectedDevices(): DeviceProxy[0...*]
+ getActiveDevices(): DeviceProxy[0...*]
+ activeDevice(DeviceProxy): void
+ disableDevice(DeviceProxy): void
+ isDeviceSupported(DeviceProxy): boolean
«implements»
EventObserver
+ sendMessage(Message): Response
+ receiveMessage(): Message
0...1
0...1
DataCollector
- readRatio: Integer
- mode: Integer
+ readSample(value, HP_TYPES.type): HealthParamSample
+ readLocation(value): Location
+ setReadRatio(freq: int)
+ startMonitor(mode,DeviceProxy[1...*])
+ createUserdataSample(HealthParamSample, Location, userInfo): UserDataSample
«implements»
<<Interface>>
EventHandling
+ notifyEvent(Event): Result
+ onReceive(Action): Result
+ streamData(): void
«implements»
«implements»
SupervisorManager
+ pendingRequests: SupervisorRequest[0...*]
+ createRequest(): void
+ evaluateRequest(SupervisorRequest, decision)
+ updateList(): void
+ add(SupervisorRequest): void
+ remove(index: Int)
0...1
0...1
UserProfileManager
+ createUserHealthProfile(weight, age, habits, diseases): UserHealthProfile
+ modifyProfile(UserHealthProfile): void

Figure 15: ASOS Client App, Interface Dependencies

### 2.5.3 Server Component Interfaces

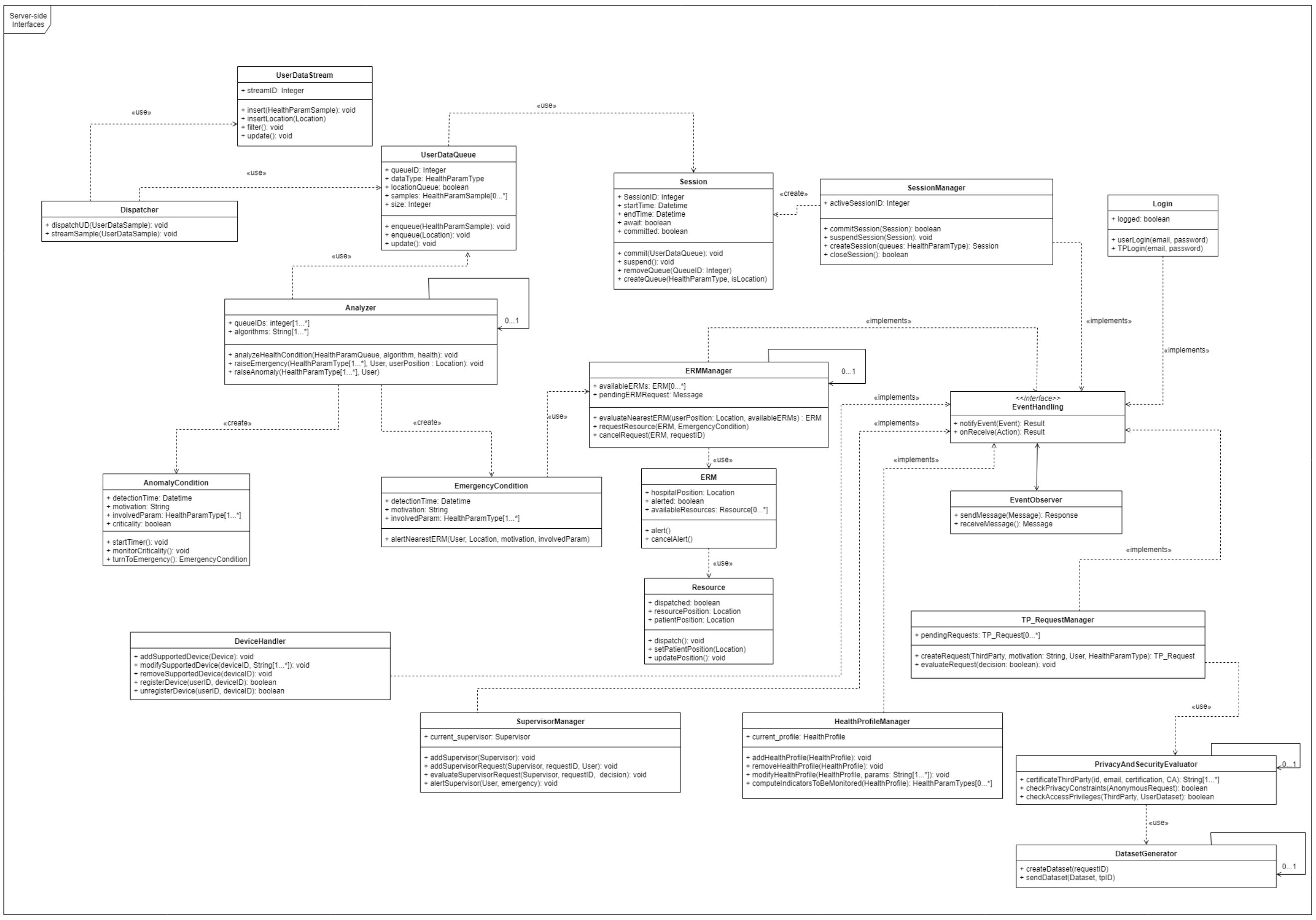
Server-side Interfaces
UserDataStream
+ streamID: Integer
+ insert(HealthParamSample): void
+ insertLocation(Location)
+ filter(): void
+ update(): void
«use»
Dispatcher
+ dispatchUD(UserDataSample): void
+ streamSample(UserDataSample): void
«use»
UserDataQueue
+ queueID: Integer
+ dataType: HealthParamType
+ locationQueue: boolean
+ samples: HealthParamSample[0...*]
+ size: Integer
+ enqueue(HealthParamSample): void
+ enqueue(Location): void
+ update(): void
«use»
«use»
Session
+ SessionID: Integer
+ startTime: Datetime
+ endTime: Datetime
+ await: boolean
+ committed: boolean
+ commit(UserDataQueue): void
+ suspend(): void
+ removeQueue(QueueID: Integer)
+ createQueue(HealthParamType, isLocation)
«create»
SessionManager
+ activeSessionID: Integer
+ commitSession(Session): boolean
+ suspendSession(Session): void
+ createSession(queues: HealthParamType): Session
+ closeSession(): boolean
Login
+ logged: boolean
+ userLogin(email, password)
+ TPLogin(email, password)
Analyzer
+ queueIDs: integer[1...*]
+ algorithms: String[1...*]
+ analyzeHealthCondition(HealthParamQueue, algorithm, health): void
+ raiseEmergency(HealthParamType[1...*], User, userPosition : Location): void
+ raiseAnomaly(HealthParamType[1...*], User)
0...1
«create»
«create»
AnomalyCondition
+ detectionTime: Datetime
+ motivation: String
+ involvedParam: HealthParamType[1...*]
+ criticality: boolean
+ startTimer(): void
+ monitorCriticality(): void
+ turnToEmergency(): EmergencyCondition
EmergencyCondition
+ detectionTime: Datetime
+ motivation: String
+ involvedParam: HealthParamType[1...*]
+ alertNearestERM(User, Location, motivation, involvedParam)
«use»
«implements»
«implements»
«implements»
ERMManager
+ availableERMs: ERM[0...*]
+ pendingERMRequest: Message
+ evaluateNearestERM(userPosition: Location, availableERMs) : ERM
+ requestResource(ERM, EmergencyCondition)
+ cancelRequest(ERM, requestID)
0...1
«use»
ERM
+ hospitalPosition: Location
+ alerted: boolean
+ availableResources: Resource[0...*]
+ alert()
+ cancelAlert()
«use»
Resource
+ dispatched: boolean
+ resourcePosition: Location
+ patientPosition: Location
+ dispatch(): void
+ setPatientPosition(Location)
+ updatePosition(): void
«implements»
«implements»
«implements»
<<Interface>>
EventHandling
+ notifyEvent(Event): Result
+ onReceive(Action): Result
EventObserver
+ sendMessage(Message): Response
+ receiveMessage(): Message
«implements»
TP_RequestManager
+ pendingRequests: TP_Request[0...*]
+ createRequest(ThirdParty, motivation: String, User, HealthParamType): TP_Request
+ evaluateRequest(decision: boolean): void
DeviceHandler
+ addSupportedDevice(Device): void
+ modifySupportedDevice(deviceID, String[1...*]): void
+ removeSupportedDevice(deviceID): void
+ registerDevice(userID, deviceID): boolean
+ unregisterDevice(userID, deviceID): boolean
SupervisorManager
+ current_supervisor: Supervisor
+ addSupervisor(Supervisor): void
+ addSupervisorRequest(Supervisor, requestID, User): void
+ evaluateSupervisorRequest(Supervisor, requestID, decision): void
+ alertSupervisor(User, emergency): void
HealthProfileManager
+ current_profile: HealthProfile
+ addHealthProfile(HealthProfile): void
+ removeHealthProfile(HealthProfile): void
+ modifyHealthProfile(HealthProfile, params: String[1...*]): void
+ computeIndicatorsToBeMonitored(HealthProfile): HealthParamTypes[0...*]
«use»
PrivacyAndSecurityEvaluator
+ certificateThirdParty(id, email, certification, CA): String[1...*]
+ checkPrivacyConstraints(AnonymousRequest): boolean
+ checkAccessPrivileges(ThirdParty, UserDataset): boolean
0...1
«use»
DatasetGenerator
+ createDataset(requestID)
+ sendDataset(Dataset, tpID)
0...1

Figure 16: Server Logic, Interface Dependencies

## 2.6 Selected Architectural Styles and Patterns

This part is dedicated to discuss about some design decision not explicitly represented in the rest of the document. Several design patterns have been considered for both the architecture, in particular:

- **Hardware Proxy Pattern**: that creates a software element responsible for access to a piece of hardware. It uses a class to encapsulate device information and all access to the hardware device, regardless of its physical interface;
- **Mediator Pattern**: to coordinate the interactions among different set of elements, a Mediator figure has been chosen. In particular, the best candidates, in this case, the *EventObserver* and the *EventHandling* interface have been conceived for allowing some components to notify events and perform specific operations when specific events occur, send and receive messages to and from the server, according to a predefined format;
- **Strategy Pattern**: since some components will implement different algorithms to perform specific tasks, this pattern has been chosen. To not complicate too much the UML diagrams and since many of this algorithms are not known at this stage of the development, the pattern has not been represented explicitly. However, the candidate classes are:
  - **Analyzer**: different types of analysis following the same abstract model could be performed according to specific risk factors;
  - **DataCollector**: that could collect data samples in many different ways from all the active devices.
- **MVC (Model-View-Controller)**: at client side, this pattern allows separation between graphical representation of data, the business logic deployed to process it and data models;
- **Chain of Responsibility Pattern**: considered in the server architecture, in particular to:
  - regulate the interaction between the server and the ERMs for dispatching a rescue squad (Resource) when an emergency condition is detected. When the *Analyzer* catches a dangerous situation, creates and delegates the responsibilities to alert the ERM to the *EmergencyCondition* component, that, in turn, calls the *ERMController* for alerting the hospital which is the nearest to the user;
  - prevent third parties to access user datasets before allowing the system to certificate their identity and check the status of the relative request. The *RequestManager* once ensured the second mentioned condition, delegates to the *PrivacyAndSecurityEvaluator* the responsibilities to certificate the identity of the requester and to ensure that all the privacy constraints are respected by anonymous requests.
- **Singleton:** some classes, for several reasons (e.g: avoid resource access conflict) will be implemented as singleton (only the existance of a static instance is allowed)

## 2.7 Other Design Decisions

On client side the defined architecture will interface with a little SQLite database hosted on the smartphone. It is well known how efficient SQLite Engine is, for smart embedded devices. Moreover, Android provides a full support in managing such database system in Android-based applications. In this document it is not explicitly inserted. After a team discussion, it has been decided to postpone the development of such part directly in the implementation stage.

# 3 User Interface Design

## 3.1 Mock-ups

For what concern the software-to-be mockups, is possible to find them in *Section 3.1* in the *Requirements Analysis and Specification Document*.
As for reminder for the next section, for the Data4Help part, the user section is intended to be a mobile application and the Third Party section a website. On the other hand for AutomatedSOS part the idea is to have only a mobile application.

## 3.2 UX Diagrams

The following diagrams are intended to give a general idea about the interactions between the various screens and the actions that are permitted. For the sake of clarity here are explained the notation of the interfaces:

- ***«Screen»:*** the screen is what the final user is going to see.
- ***«Input Form»:*** everything that the final user is going to interact with (empty space to fill with text, checkboxes, on/off buttons...).
- ***«Screen Compartment»:*** part of the screen that contains plain text.

### 3.2.1 Data4Help

**User**

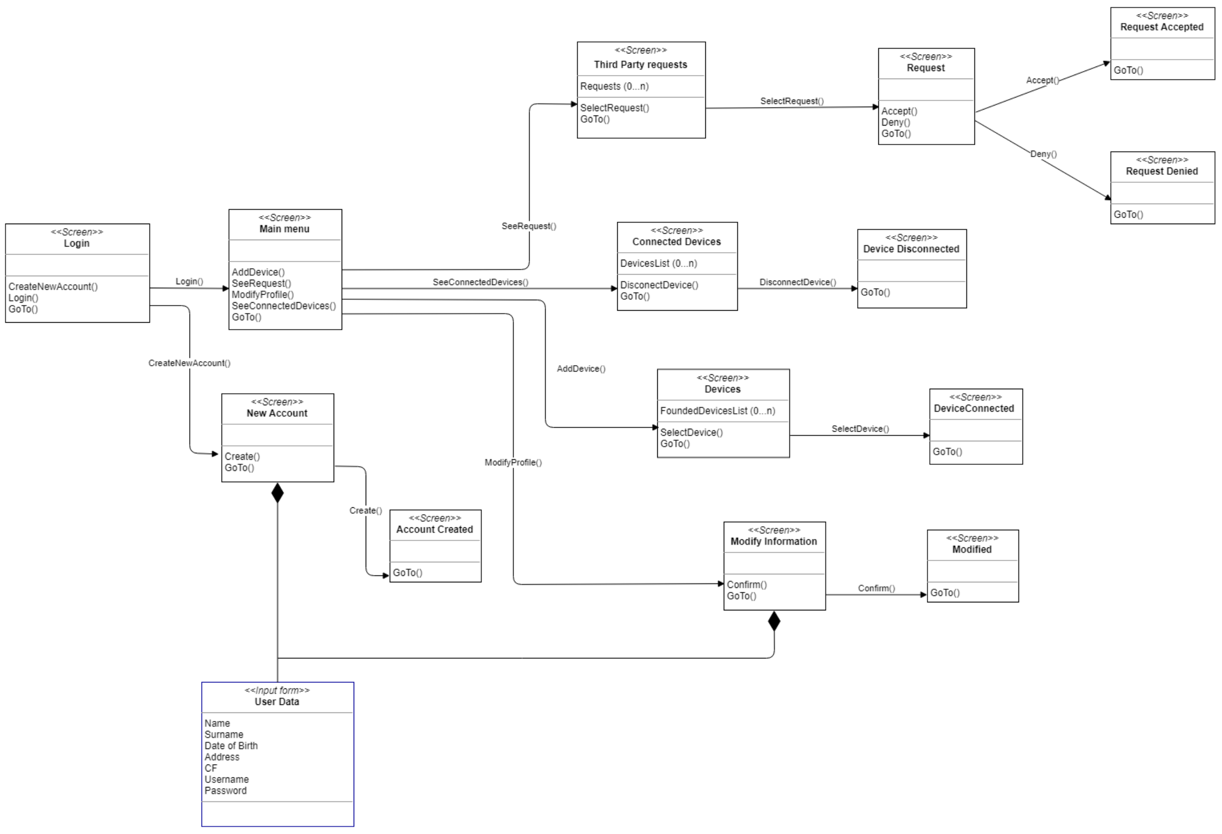

Figure 17: Data4Help User UX diagram

## Third Party

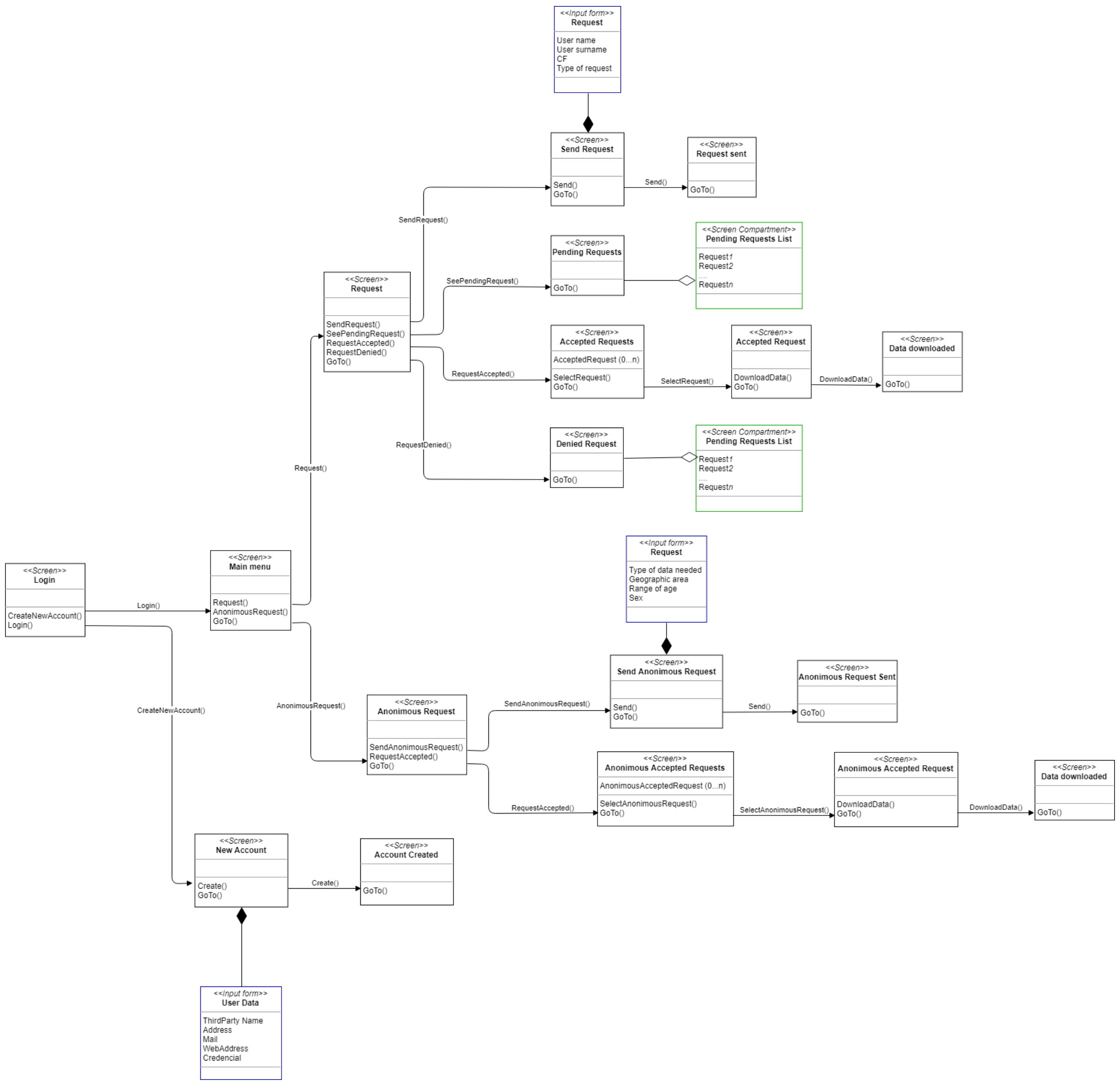

Figure 18: Data4Help Third Party UX diagram

### 3.2.2 AutomatedSOS

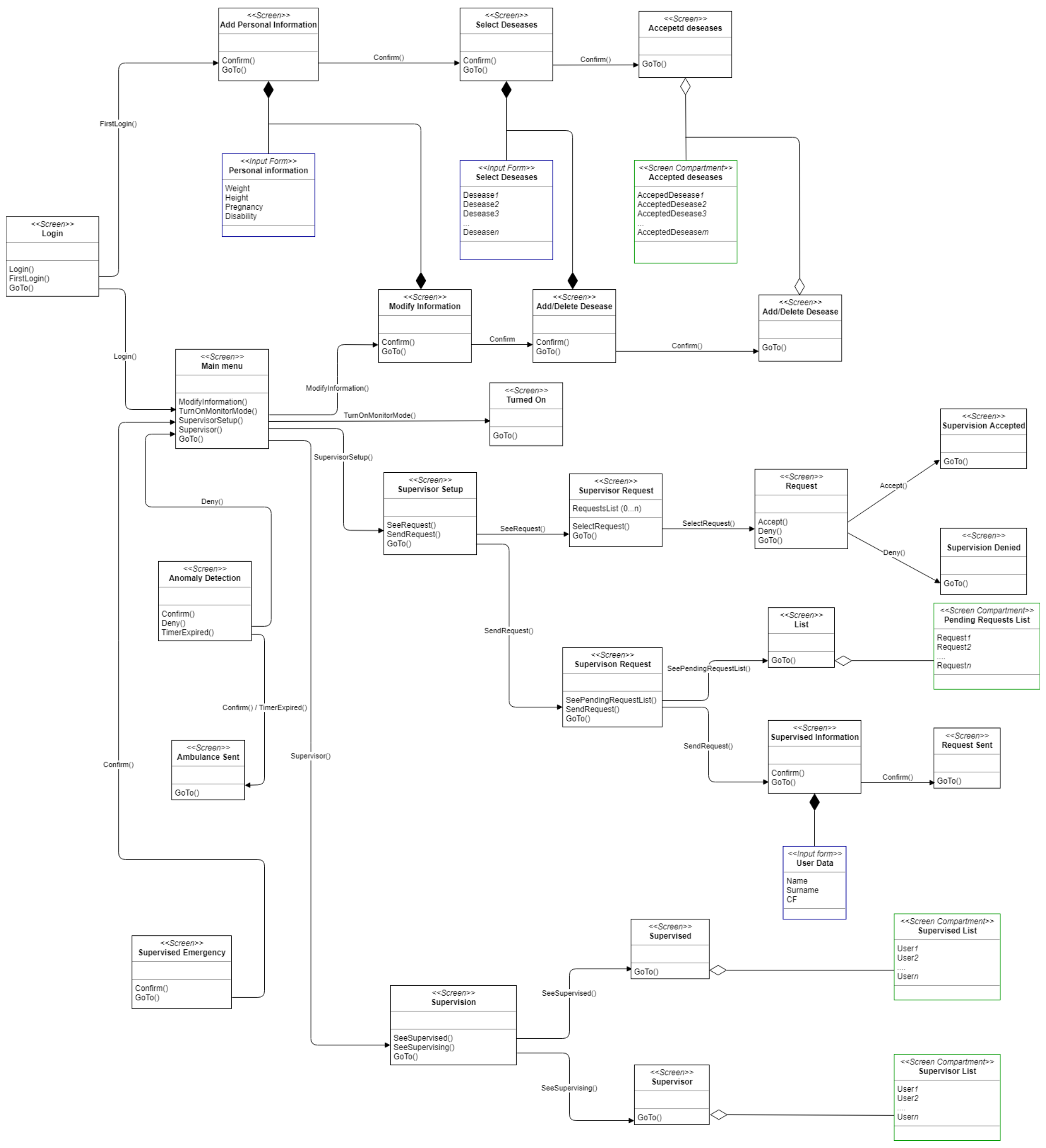

Figure 19: AutomatedSOS user UX diagram

# 4 Requirements Traceability

## 4.1 Data4Help

**[G1]: A user can register personal health monitoring devices in the system**

- [R1]: The system must allow a registered user to associate a sensor device to the service.
  - **DeviceManager** (Client)
  - **DeviceHandler** (Server)

**[G2]: TrackMe acquires periodically health parameters specifically related to a user**

- [R2]: Each user is uniquely identified by the system.
  - **Login** (Server)
- [R3]: The system can acquire health data by specific sensors connected to the user's main device.
  - **DeviceProxy** (Client)
  - **DataCollector** (Client)
  - **Dispatcher** (Server)
  - **Session** (Server)
  - **UserDataQueue** (Server)
  - **UserDataStream** (Server)
  - **SessionManager** (Server)

**[G3]: Third parties can access health data of specific users, if expressively authorized by the them**

- [R5]: The system must be able to identify and certificate the reliability of each organization that wants to request user data.
  - **PrivacyAndSecurityEvaluator** (Server)
- [R6]: Each registered organization that wants to access health data of specific users must be able to formulate a request providing information related to the purpose of the request.
  - **TP_RequestManager** (Server)
- [R7]: The system must notify each user of a third party request as soon as it is formulated, and allow him/her to accept or reject the request.
  - **DataRequestManager** (Client)
- [R8]: For each third party request the system should provide to the user information related to the requestor and the purpose of the request.
  - **TP_RequestManager** (Server)
  - **DataRequestManager** (Client)
- [R9]: Once a third party request is accepted by the user, the third party must have full access to the entire collection of data of the user.
  - **TP_RequestManager** (Server)
  - **DatasetGenerator** (Server)

**[G4]: Third-parties can request anonymous information about groups of users**

- [R10]: Third parties must be able to request health data of groups of anonymous users, according to several criteria without being expressively authorized.
  - **TP_RequestManager** (Server)
  - **DatasetGenerator** (Server)
- [R11]: The system must prevent third parties to trace back to specific user information through anonymous requests.
  - **PrivacyAndSecurityEvaluator** (Server)

## 4.2 AutomatedSOS

**[G5]: The user has the possibility to specify the diseases he/she has, so the system can evaluate which health parameters to monitor**

- [R12]: The system must give the possibility to the user to specify which diseases he/she has.
  - **UserHealthProfileManager** (Client)
  - **HealthProfileManager** (Server)
- [R13]: The system must prevent the possibility to use the service, if the minimum required sensors are missing.
  - **HealthProfileManager** (Server)

**[G6]: An ambulance may be sent within 5 seconds, if an emergency or anomaly condition is detected**

- [R14]: The system must allow the user to turn on and off the service each time he/she wants.
  - **Session** (Server)
  - **SessionManager** (Server)
- [R15]: The system must stop monitoring when minimum required sensors are missing.
  - **SessionManager** (Server)
  - **HealthProfileManager** (Server)
- [R16]: The system must be able to distinguish and detect emergency or anomaly conditions occurring to a specific user.
  - **Analyzer** (Server)
  - **EmergencyCondition** (Server)
  - **AnomalyCondition** (Server)
- [R17]: When an anomaly condition is detected, the system must send a notification to the user, asking if it is an emergency condition. If the user does not answer the notification within 30 seconds, the anomaly must become an emergency
  - **Analyzer** (Server)
  - **EmergencyCondition** (Server)

- **AnomalyCondition** (Server)
- **EventObserver**

- [R18]: When an emergency condition is detected, the nearest ambulance must be alerted, providing all the information about the situation.
  - **Analyzer** (Server)
  - **EmergencyCondition** (Server)
  - **AnomalyCondition** (Server)
  - **ERMManager** (Server)
  - **Resource** (Server)

**[G7]: A user designated as Supervisor of another one, is notified of both emergency and anomaly events occurring to the supervised user**

- [R19]: A user must have the possibility to request to become Supervisor of another one.
  - **SupervisorRequestManager** (Client)
  - **SupervisorManager** (Server)
- [R20]: Each supervised user must be able to accept or reject the possibility to have a Supervisor.
  - **SupervisorRequestManager** (Client)
  - **SupervisorManager** (Server)
- [R21]: The Supervisor must be notified by the system of all emergency and anomaly conditions occurring to the supervised user.
  - **SupervisorManager** (Server)
  - **EventObserver**

# 5 Implementation, Integration and Test Plan

## 5.1 Implementation

It has been decided that, as there is the presence of two partly distinguished applications, the implementation will be led concurrently by two different teams. In this way it will be possible to not lose time during the integration and testing part. Each team will focus on one client application, performing firstly implementation and then unit testing of each coded module.

### 5.1.1 Client Applications

Both the teams will work in a joint session to develop the part related to the sensor interaction. The first task will be the implementation of the DeviceProxy module that will interface with the real devices through the available APIs. The second will be the realization of the DeviceManager and consequent test. Then a week will be taken to discuss and choose the most efficient algorithms for collecting data, that could be handled by the DataCollector, implemented as third. It will be important to provide some simple test cases to perform unit testing of this component owing to its complexity in handling sensor data. After that, the two teams will work separately for the remaining components of the client application.

### 5.1.2 Server Application

Once finished the implementation of the smartphone apps, both the teams will manage the server architecture. Firstly, they will concentrate their efforts on the most complex module: the Analyzer. In collaboration with biomedical engineers and medical researchers, this component will be firstly defined in detail, then implemented. The whole process is expected to be completed in at least one month, a couple of weeks will be taken to create test cases and perform unit testing of the mentioned module. Completed this phase, the two teams will separate to code and test the following units:

- First Team, Session Management:
  - **SessionManager**
  - **Session, UserDataQueue, UserDataStream**
  - **Dispatcher**
- Second Team, Emergency Management
  - **EmergencyCondition, AnomalyCondition**
  - **ERMManager**
  - **Resource**

Then both the teams will work jointly on:

- **HealthProfileManager**
- **SupervisorManager**
- **DatabaseManager:** in particular, the part concerning ASOS functionalities;

- **EventObserver** and **EventHandling Interface**: at least to allow message exchanging for what concerns ASOS client-server communication.

After this working session the integration and testing phases could start for the implemented architecture, carried on by one of the two teams. The decision to following this order has been taken since it is almost intuitive that the mentioned phases will take much more time than the ones required for the rest of the architecture. Meanwhile, the other team will focus on the part of the server architecture that regards D4H functionalities, coding the remaining part of the DatabaseManager.

## 5.2 Integration and Testing

The integration and testing part will be performed bottom-up. This decision has been taken because these two applications have more critical parts in the business logic components and client-server communications, instead of the user/application interactions. The bottom-up model permits us to start by creating a solid base, starting to test the client architecture, then the server one and finally the communication between the two parts.

### 5.2.1 Client: Data4Help and AutomatedSOS Testing

The integration part will be done following the implementation part. This means that as soon as a component will be ready, the integration with the other ones already available will be performed.
The first test will be carried out on the sensor interaction part, involving the DeviceProxy, DataCollector and DeviceManager. Different test cases will feed the integrated modules in order to evaluate their response under stressing workloads and that of external devices when communicating with the DeviceProxy.
After the unit testing of the other client components and their integration with the EventObserver through the EventHandling interface, the client architecture can be tested feeding both the apps with patterns of messages to test the client reaction to external events. Few days could be required to decide which test cases are the most suitable for this testing phase.

### 5.2.2 Server-side Testing

The server logic is the hardest part for testing phase. As said in the previous section, as soon as the Analyzer unit is coded and tested, it must be integrated with the EmergencyCondition and AnomalyCondition components once they had been completed. In the while, if the first team coded the Resource and ERMManager components, a Driver and a Stub components could be created to simulate the EmergencyCondition block (based on the current work of the other team) and the behavior of the real ERM. All of this, to test the ERM communication part:

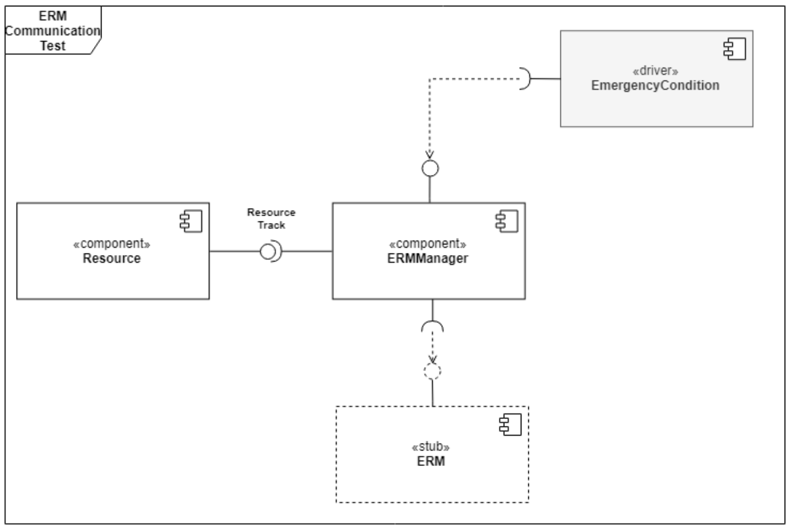

Figure 20: ERM Communication Testing

After this moment, for a couple of weeks, several test cases will be discussed and inserted in the test plan in order to optimize the resources spent during this phase. Once the previous tasks are completed the developed part of architecture will be tested using the chosen cases, after the relative stub and driver components are created:

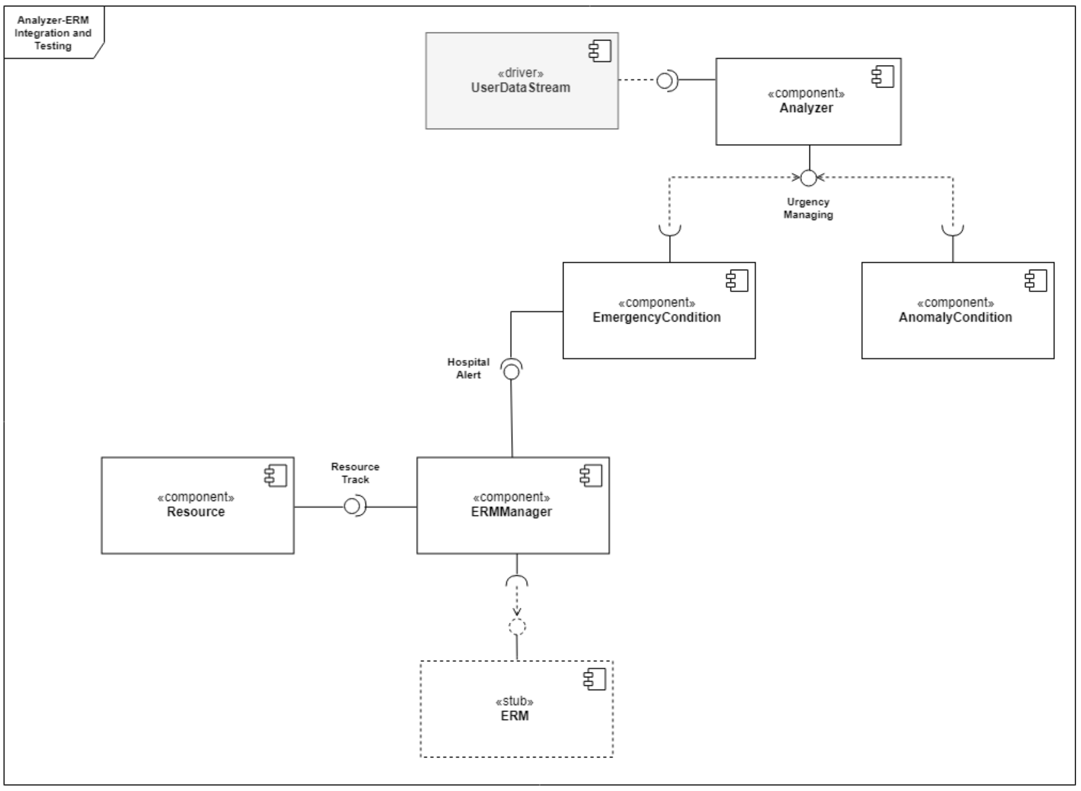

Figure 21: Analyzer-ERM Integration and Testing

After this long test step, the developed architecture will be integrated with the Session Management part and the other components, once these ones are completed and tested as well.

For what concerns D4H part of the architecture at server side, a particular attention should be spent to the interaction between the third parties and the user datasets and in particular to the PrivacyAndSecurityEvaluator. At this point, for few days, several test cases will be inserted in this part of the test plan with the support of computer security engineers. Once the TP_RequestManager, the mentioned security module and the DatasetGenerator are completed, the integration and consequent test of these components can be performed.
After that the DatabaseManager and EventObserver can be integrated to the rest of the architecture and lastly tested.
At the end of this long phase, the two teams are expected to work together on a set of test cases to stress the entire system and in particular the communication between client and server applications. This last phase is clearly explained in the next section.

### 5.2.3 Communication Testing

The last part of testing will be performed once every part of the two architectures will be completed. It consists on trying to make client and server working together preventing any conflict between them. Here is a list of special test cases required to probe the communication and reaction of the system when conflict events occur:

- **Register a device when AsosMonitorMode is activated:** in order to not create any conflict, the system must prevent the user to add a new device while ASOS is working.In this case a warning should be showed to the user.
- **Turn on AsosMonitorMode while Data4Help is collecting data:** the system should let the user to start an ASOS session. The result should consist on stopping the collection of data from a specific device in Data4Help, in order to not create sample copies collected by both the running applications, if one or more devices working with D4H are required by ASOS. At server side the expected result is a change in the state of the Session and the Dispatcher that will push the new set of samples into the UserDataStream while continuing to fulfill the relative queue. The mentioned set will be committed anyway to the database.
- **Turn off AsosMonitorMode while Data4Help is collecting data:** the system should return in the state it was before turning on the AsosMonitorMode. Samples should not be sent to the UserDataStream anymore.

## 6 Effort Spent

Here is the effort spent by each group member in working at this document.

| **Barbara Ferretti** | **Hours** |
|---|---|
| User Interfaces | 5 |
| Implementation, Integration and Test Plan | 4 |
| General | 3 |

| **Giorgio Cozza** | **Hours** |
|---|---|
| Introduction | 2 |
| Architectural Design | 20 |
| Implementation, Integration and Test Plan | 3 |

# 7 References

- Bruce Powel Douglass, PhD, "*Design Patterns for Embedded Systems in C: An Embedded Software Engineering Toolkit*"
- Gamma E., Helm R., Johnson R., Vlissides, J."*Design Patterns Elements of reusable Object-Oriented Software*"
- "*La Gestione dei Database in Android*", https://www.html.it/articoli/la-gestione-dei-database-in-android/
- Jim Conallen, ”*Building Web Applications With Uml*”