System Design Document

UKE MLAB

Corvin Biebach
Maximilian Brosius
Fynn-Ole Menk
Anni Reinert
Noah Scheld
Arne Struck
Mudassar Zahid

9. Dez. 2021

Table of Contents

Table of Contents 2
Introduction 3
Overview 3
Design Goals 3
Subsystem decomposition 4
Hardware/software mapping 5
Persistent data management 6
Access control and security 7
Global software control 7
Boundary conditions (if applicable) 7

1. Introduction

This document shall give an overview of the design goals of the MLab UKE patient monitoring, ventilator, and defibrillation 3-in-1 application. The goal of this project is to build a proof of concept for such a 3-in-1 user interface and alarm management.

1.1. Overview

This application will use a modern tablet device to simulate the actual screen of a real-world patient monitor. However, this application running on a tablet will not meet the actual target such a developed 3-in1 device would have. A real commercial product would run on an embedded system with components specifically designed to run the patient monitoring, ventilation, and defibrillation. Nevertheless, the scope in this project is to build and sketch out ideas to redesign the user interface, reimagine the alarm management and generally rethink how the user interaction with such a device should or could look like. This entails that goals and requirements for designing such systems need to be carefully distinguished.

On the following pages, we will consider different aspects of the application that is developed, delimit its differences from the real world scenario and show what stands out to fulfill the goals for this showcase.

2. Design Goals

The system at hand is embedded in a context that requires the handling of very sensitive information while it has to deliver its capabilities at all times and under the most unlikely circumstances. Thus, besides the requirements that have been stated and agreed on some fundamental and overreaching design goals emerge. However, the design scope of this project needs some specification. The system at hand is a mockup and proof of concept of the existing three devices (monitoring, defibrillation, and ventilation) combined in one device. This proof of concept is supposed to sketch out ideas and inspire the development of a potential real three-in-one solution of the mentioned devices. Thus, some of the design goals at hand that are not discussable for the real device play an inferior role in our proof of concept. This means that showcasing certain ideas and feature solutions are preferred in the scope of this project although these ideas would not meet the design goals and requirements of a real-world device. For example, a real device would have to meet certain requirements to be certified meeting a certain medical standard. Focussing on such requirements and prerequisites would narrow down the room for innovation and ideas that are the goal for this proof of concept. This does not of course mean that unrealistic ideas and innovation or the neglect of the underlying design goals are preferred at all times. All innovation should with some adaptation or further refinement have the possibility of being implemented in the real world device.

Since doctors use the system in critical situations and most of the time are fighting for lives with the help of it, robustness and reliability are fundamental for building a System that fits the needs of the situations and stakeholders. By robustness, it is meant that the system can handle faulty user inputs, detect them, and in necessary cases assist the user in correcting them. Further robustness also means that unexpected malfunction of subparts of the system does not lead to a crash of the whole system. Reliability on the other hand shall reflect the extended need of the users, that the system needs to be available and responsive at all times the user intends to use the system. In addition, this also means that users can sincerely rely on the fact that each functionality of the system does precisely, and nothing more than the function entails doing.
Since critical medical conditions also inherit time criticality and the potential to change rapidly, another goal of the design is the high performance of the system in all situations. This means that the user does not encounter loading screens or situations while performing standard tasks with the system. In addition, the system does always deliver the expected screens and settings when the user asks for them without having to wait.
An improvement in design was a general goal from the beginning of the project. However, ease of use and user-friendliness has not played the biggest role in the design of the old systems that are currently used. The fact that fundamental settings of alarm borders for example are not reachable within 2-3 taps or clicks alone emphasizes the need for better usability and ease of use.

3. Subsystem decomposition

The original system consists of 3 individual subsystems. However, our task is to unify these systems, which is why we have a different approach for our subsystem division. The subsystems now will build on each other and support each other.
We implement this idea first and foremost with a division into monitoring, ventilation, and defibrillation. Coupling cannot be avoided completely. Each subsystem will still have to know some subparts from the other system. This is also the reason why we call the relation between them hierarchical. Nevertheless, to reduce coupling we will therefore try to shrink our flutter widgets into smaller and smaller subparts. So that we avoid redundant code and increase reusability.
For instance, we need to use graphs in the monitoring subsystem, which are also used in the ventilation subsystem. In addition, the ventilation subsystem will also have other graphs like a flow graph which can also be generated from the same graph widgets. These flutter widget decisions will be more common in our project.

The following interaction and the MVC pattern are shown in our UML on page 9.

We also follow the architecture style MVC. With the help of the MVC, we will separate these subsystems from each other and let them communicate only coherently. Our views, by and large, make up our subsystems. Controllers are used to switch between the views and to insert alarms into the views. Each of these views is provided with data or controlled by these controllers. We also have a data model, which provides mockup data and alarm settings among others.

In addition, there is the scenario Subsystem. This subsystem is special in that it interacts with the MVC pattern.
The purpose of the scenario subsystem is to provide mockup data within a time history. We thus provide input to the system which is, in the big picture, decoupled from the actual subsystem relationship. It has to interact with the model and the controller, but this interaction is only unilaterally from the scenario subsystem.

4. Hardware/software mapping

In the big picture, our application is involved in a system that provides our application with data. This data is fed into our system via an API. This is fairly open-ended and vaguely described. See also page 10 Deployment Diagram Full System.

For our proof of concept, we restrict ourselves to an interaction that we can describe quite precisely. See also page 10 Deployment Diagram Our App. A tablet that simulates the screen with mockup data. We will not get real-time data from an external system. In this case, we will also not be able to plan a proper interface for this exact connection with a medical device.

Our application should still be able to receive data via an API. First of all, for the reason that at least in case the external system is developed, it can already adapt to specifications from our system. Secondly, we want to set the scope to build a real system, as it could be used.
This is also the reason why we use simulation data from Weinmann Emergency, presumably to recognize patterns. With this data, we can prepare our model and more generally our core app for the future data structure. This helps us to bring our simulation data as well prepared as possible into our core app, as the future device from Weinmann Emergency would do.

The advantage is that everything that happens in the core app does not have to be redesigned by the customer again.

Accordingly, the most important point in our mapping is that the model is provided with correct mockup data, which can also be replaced via an API interface.

Another point is that we will design the core app as a tablet application. Here we have certain requirements, whereby our software will function exclusively within this app.

5. Persistent data management

Data management plays a fundamental role within the system that is developed in the scope of this system. At all times it is necessary to show and update the screen of the system with correct, coherent, and actual data. Showing wrong or outdated data has to be eliminated at the best of possibilities.

The screens of the system house multiple graphs and absolute value parameters. These graphs and absolute values are usually (in the real world device) fed into the system using sensors, analog to digital transformers, filtering, and some logical interpretation of the read data. For the scope of this system, we will not have the possibility to read and work with real live data from actual sensors since this is a proof of concept running on a tablet device.

However, mocking and showcasing the ideas as close to the real-world application as possible is key to showing the feasibility of the innovation implemented in this project. To achieve this we have agreed on implementing an interface that can interpret and read real sensor data. To ensure this, we're in contact with one of the engineers at Weinmann Emergency who agreed on sharing some recorded real sensor log data of an actual monitoring/ventilation device. By building our application's interface around these datasets we ensure realistic processing and interpretation of live data.

The advantage of using real log data of such systems is that we can ensure that our application is capable of handling the data streams and rates that are necessary for the correct representation of certain medical parameters. Further, a handover to a real-world prototype with actual sensors can be simplified since no translation middleware needs to be implemented. Lastly, this also brings the possibility to test our application with the same simulators that are used to test the actual Weinmann devices.

To give a more detailed view of how the graphs and absolute values get the necessary data, we will consider the Entity-Relationship diagram of our project. Please refer to the ER Diagram on Page 11.

The heart of our system design is the Data Model that contains a so-called sensor that creates a connection between the ChartData – our data provider – and the devices that utilize that data. There are two types of data: regular chart data with a value and timestamp, and NIBD data that additionally has a systolic pressure, diastolic pressure, and mean arterial pressure (derived from the two aforementioned attributes). This data is then visualized by the graph entities and value boxes. The system state closely works with the data model to ensure persistency. It stores vital information about the states of each entity, i.e. the device settings for ventilation and defibrillation and the preset values for each patient type.

6. Access control and security

Since the proof of concept for this 3-in-1 patient monitoring device does not require presentation and curation of user- or group-specific data and content, something like authentication or data management and encryption within a database is not necessary. If we consider the context and scope of such devices again, one can observe that these systems are presenting and analyzing sensor data that is fed into the system. Implementing something like user accounts and authentication would thus complicate and prolong the immediate use of our system. Which are an absolute necessity for devices that are relied on in emergencies. Further, the application we're developing will be only available for medical personnel and won't be accessible for unauthorized persons. Of course, the data and pieces of information that are fed into the system from the sensors are very sensitive and shall under no circumstances get into the hands of unauthorized people. Since our proof of concept won't have any communication interfaces that allow or facilitate communication with other systems like a hospital database, factors like encryption or secure communication are negligible in the context of our project. Clearly, these are design factors that are unavoidable to be implemented once our proof of concept reaches a stage of actually being produced as a commercially available product that should be capable of communicating the patients' information to a hospital. What this all amounts to is again, that we're developing a showcase of ideas for presenting patient information and implementing smart alarm management for a future device that might be developed. This entails that our system will never be used in a real emergency scenario. Rather it will be used with simulated data within simulations of emergencies to get an idea of how alarms can be managed and presented as an alternative to the currently available approach.

7. Global software control

We mainly control our flow with our MVC pattern, which we have illustrated as a sequence diagram on page 12.

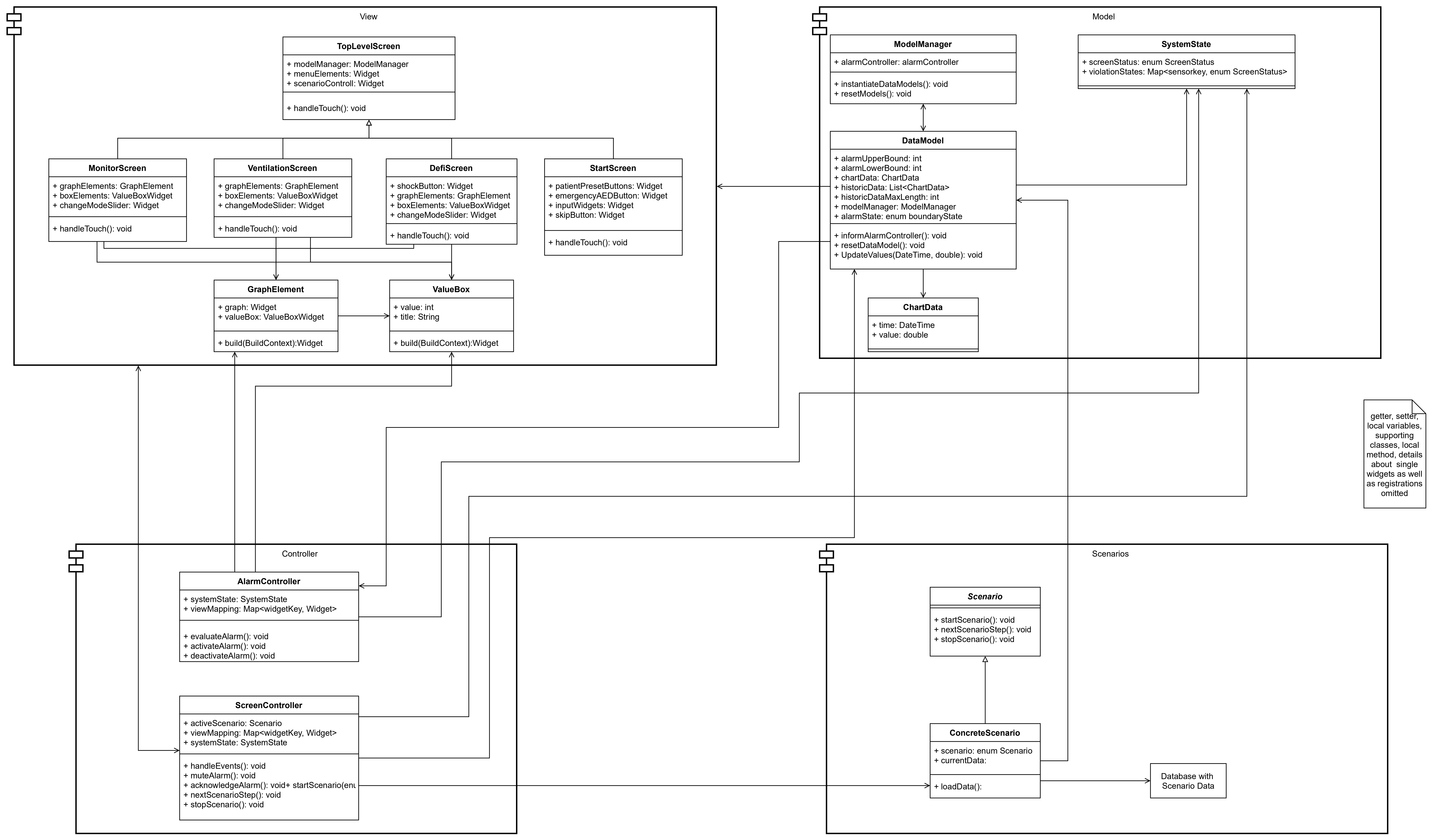
With the help of the MVC pattern, we decide against a monolithic and in favor of an event-driven architecture. We see the advantage that, for example, alarm management can be implemented easily and we can update our views using observer patterns. This pattern can then act for the mockup data as well as with the alarms.

The control flow is implemented using different controllers in the MVC pattern.

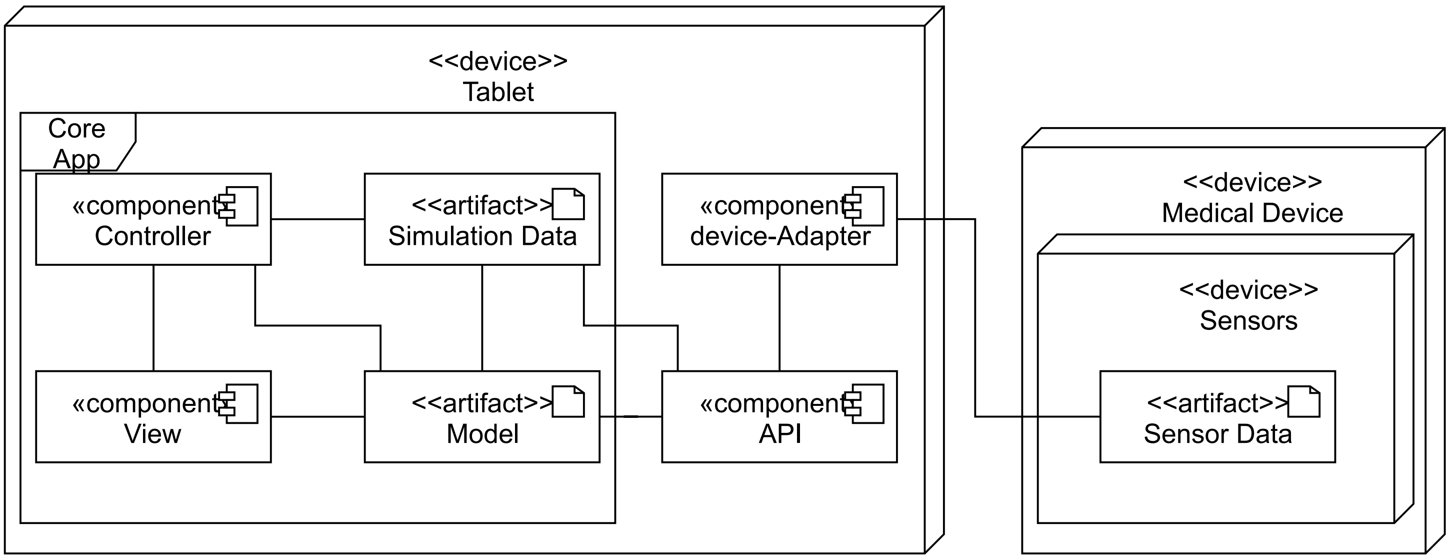
The synchronization of the graphs within the views is realized using listeners and the Flutter package GetX. For this purpose, each created graph has the listeners implemented directly. Alarms can also trigger these graphs via the listeners.

8. Boundary conditions

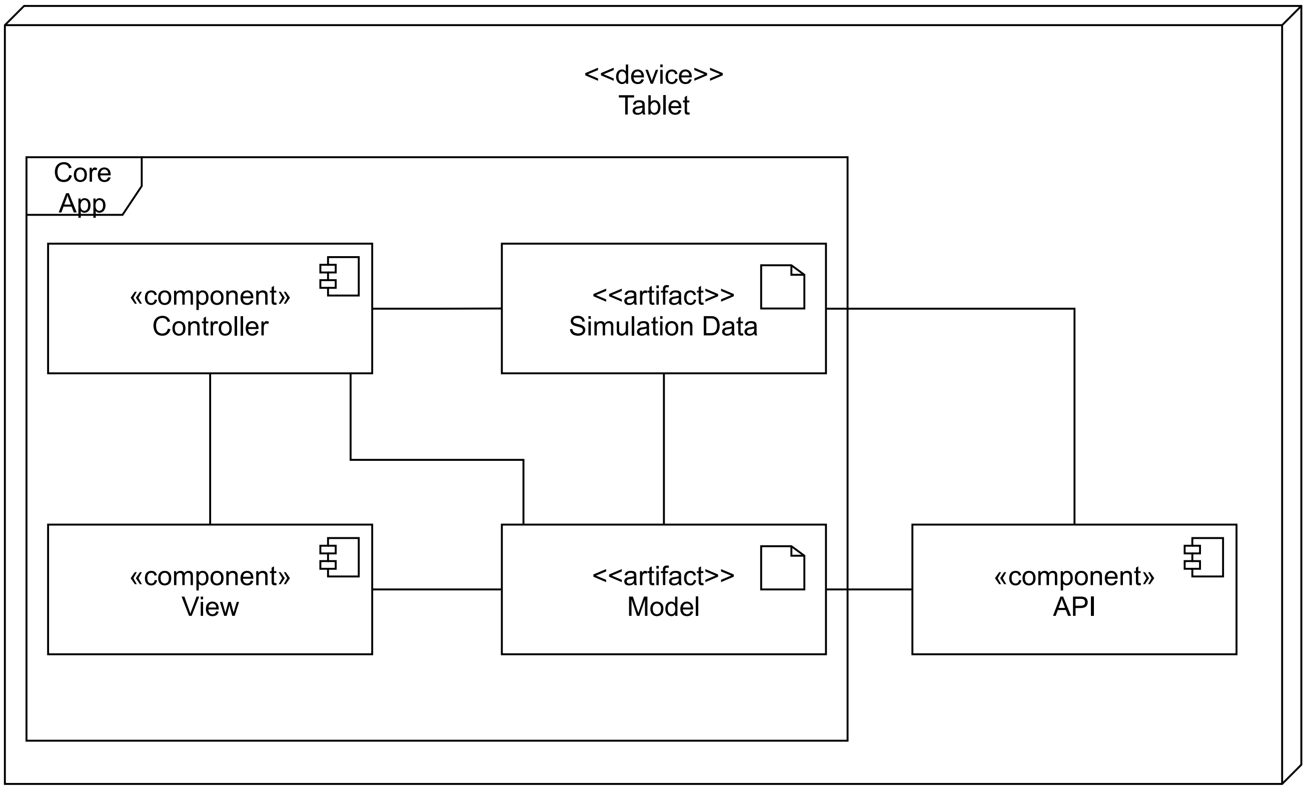
The app is run on the device, i.e. our tablet. The app is disabled by either turning off or locking the device If a component fails, the app is restarted automatically or manually if necessary. Because the data comes mainly from the mockup, this is not a problem. The same happens with a complete system failure. The only important thing is that the restart works fast.


View
TopLevelScreen
+ modelManager: ModelManager
+ menuElements: Widget
+ scenarioControll: Widget
+ handleTouch(): void
MonitorScreen
+ graphElements: GraphElement
+ boxElements: ValueBoxWidget
+ changeModeSlider: Widget
+ handleTouch(): void
VentilationScreen
+ graphElements: GraphElement
+ boxElements: ValueBoxWidget
+ changeModeSlider: Widget
+ handleTouch(): void
DefiScreen
+ shockButton: Widget
+ graphElements: GraphElement
+ boxElements: ValueBoxWidget
+ changeModeSlider: Widget
+ handleTouch(): void
StartScreen
+ patientPresetButtons: Widget
+ emergencyAEDButton: Widget
+ inputWidgets: Widget
+ skipButton: Widget
+ handleTouch(): void
GraphElement
+ graph: Widget
+ valueBox: ValueBoxWidget
+ build(BuildContext):Widget
ValueBox
+ value: int
+ title: String
+ build(BuildContext):Widget
Model
ModelManager
+ alarmController: alarmController
+ instantiateDataModels(): void
+ resetModels(): void
SystemState
+ screenStatus: enum ScreenStatus
+ violationStates: Map<sensorkey, enum ScreenStatus>
DataModel
+ alarmUpperBound: int
+ alarmLowerBound: int
+ chartData: ChartData
+ historicData: List<ChartData>
+ historicDataMaxLength: int
+ modelManager: ModelManager
+ alarmState: enum boundaryState
+ informAlarmController(): void
+ resetDataModel(): void
+ UpdateValues(DateTime, double): void
ChartData
+ time: DateTime
+ value: double
getter, setter, local variables, supporting classes, local method, details about single widgets as well as registrations omitted
Controller
AlarmController
+ systemState: SystemState
+ viewMapping: Map<widgetKey, Widget>
+ evaluateAlarm(): void
+ activateAlarm(): void
+ deactivateAlarm(): void
ScreenController
+ activeScenario: Scenario
+ viewMapping: Map<widgetKey, Widget>
+ systemState: SystemState
+ handleEvents(): void
+ muteAlarm(): void
+ acknowledgeAlarm(): void+ startScenario(enu
+ nextScenarioStep(): void
+ stopScenario(): void
Scenarios
Scenario
+ startScenario(): void
+ nextScenarioStep(): void
+ stopScenario(): void
ConcreteScenario
+ scenario: enum Scenario
+ currentData:
+ loadData():
Database with Scenario Data

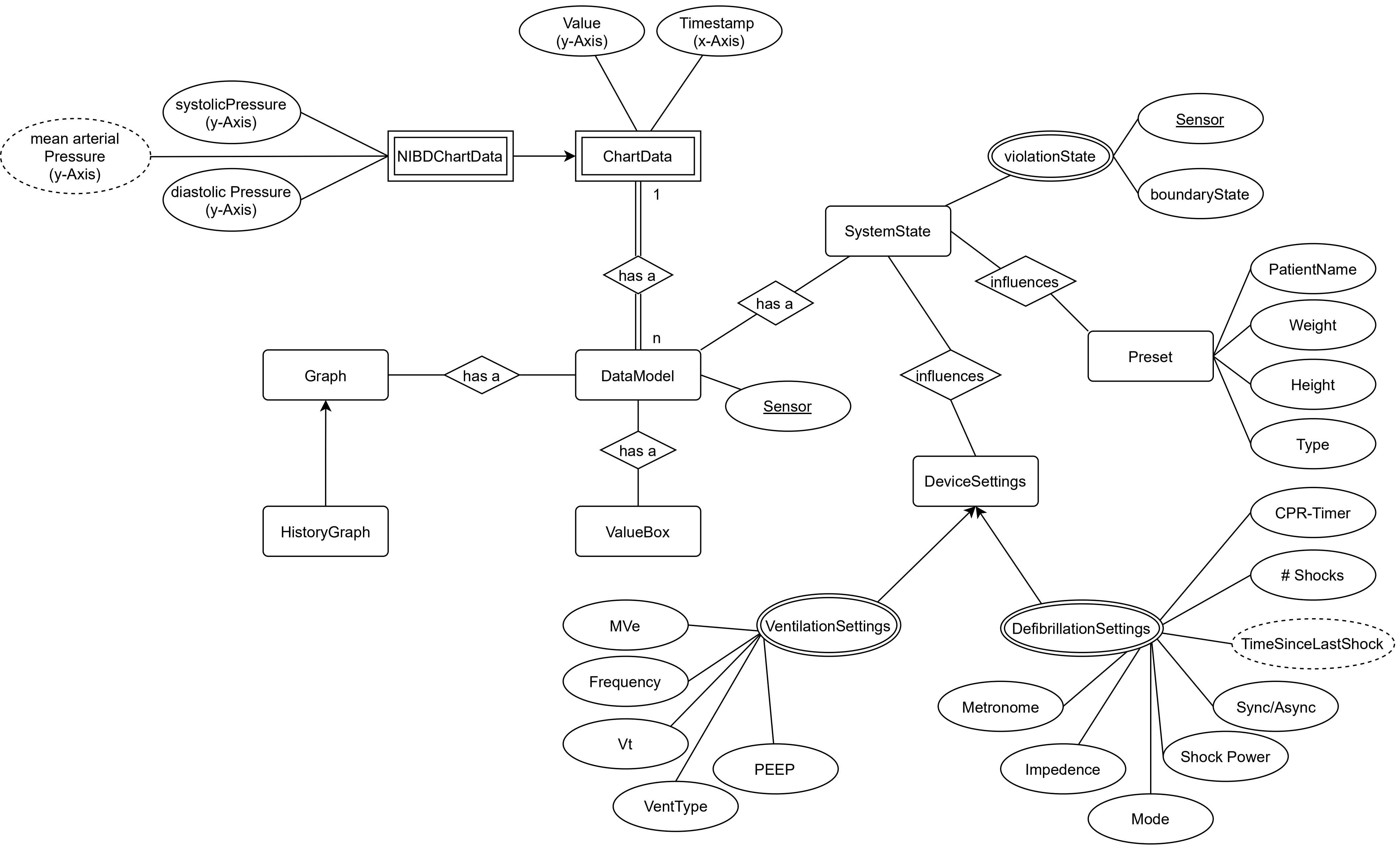
Deployment Diagramm Full

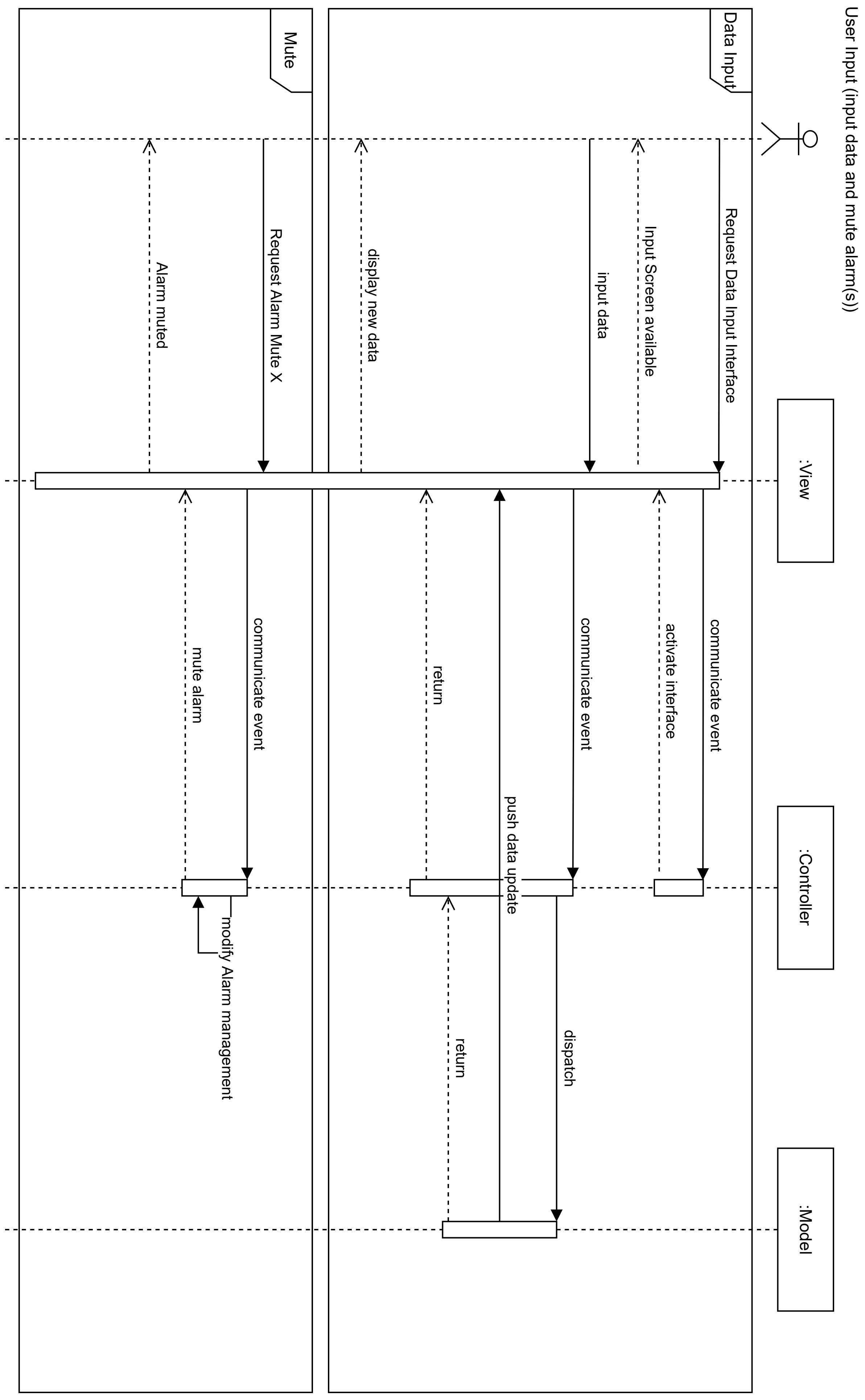


Deployment Diagramm Our App



Entity Relationship Diagram
MLab - UKE




User Input (input data and mute alarm(s))
:View
:Controller
:Model
Data Input
Request Data Input Interface
communicate event
activate interface
Input Screen available
input data
communicate event
dispatch
return
push data update
return
display new data
Mute
Request Alarm Mute X
communicate event
modify Alarm management
mute alarm
Alarm muted