System Design Document

Corvin Biebach

Maximilian Brosius

Fynn-Ole Menk

Anni Reinert

Noah Scheld

Arne Struck

Mudassar Zahid

27.01.2022

Table of Contents

1 Introduction

1.1 Overview

This document provides an overview of the design goals for the M-Lab UKE project. The project aims to develop an application for an emergency medical care device that combines monitoring, ventilation and defibrillation. The scope of this project is to build and sketch out ideas to redesign the user interface, reimagine the alarm management and generally rethink how the user could interact with a 3-in-1 interface.

This application will use a modern tablet device to simulate the screen of a patient monitor. Running on a tablet this application will not meet the actual target of a future 3-in1 device. This entails that goals and requirements for designing the system need to be carefully distinguished. On the following pages, this document will consider the design goals and different aspects of the application. Furthermore, it will delimit its differences from the real world scenario and shows what the goals for this showcase are.

1.2 Definition, acronyms and abbreviations

3-in-1	Future combination device of monitoring, ventilation and defibrillation
AED	Automated external defibrillator
API	Application Programming Interface
chartData	Provides data for the chart in the Syncfusion library
Flutter	Open-Source UI Development Kit
GetX	State management package for Flutter
Jupyter Lab	Web-based interactive computing platform
Python	An interpreted high-level programming language
MVC Pattern	Model-View-Control Pattern
NIBP	Non-invasive blood pressure
UML	Unified Model Language
Weinmann Emergency	Manufacturer for the current device for ventilation and future 3-in-1 device

1.3 References

The requirements for the M-Lab project have been clarified in the Requirement Analysis Document. In the following pages, those requirements in the context of the System Design will be defined further with the design goals.

The app is implemented in Flutter and the software code will be delivered via GitLab. The generated data for the patient scenarios are written with Python in Jupyter Lab. The Figma prototype serves as a template for the design of the user interface and should be implemented as well as possible.

2 Design Goals

The system is going to provide a combined interface of a monitoring, ventilation and defibrillation device. By combining the devices it is possible to develop an alarm management system that can offer new approaches for a future device that will help medical personnel. The system must perform its functions and process sensitive information at any time. Therefore, in addition to the requirements already defined for the project, fundamental and overreaching design goals arise.

The scope of this project is a mockup on a tablet and is going to be a proof of concept of the existing three devices, previously mentioned. The required specifications for the future device is 10 inches with a resolution of 1920 x 1200 in a format of 16:10. These specifications are not fulfilled by currently emulatable tablets nor by reference hardware owned by the team, which is why the application will be designed for an iPadPro (11 inches) and an Android Pixel C (10.2 inches).

The project as proof of concept is supposed to sketch out ideas and inspire the development of a potential real 3-in-1 solution of the mentioned devices. This means that showcasing certain ideas and novel feature solutions is preferred in this project's scope. Those ideas may not meet the design goals and requirements of a real-world device. Focussing on such requirements and prerequisites would narrow down the space for innovation and ideas which are the goal for this proof of concept. This does not mean that the degree of realism of an idea or innovation is not considered. All innovation should, with some adaptation or further refinement, have the possibility of being implemented in the future 3-in-1 device.

The design of the alarm management needs to prevent sensory overload for the user. Simultaneously occurring alarms are required to be handled with specified rules, like prioritising alarms according to their importance. For those rules, the changing values of parameters need to be considered. In addition to that, the alarm management should give the user the ability to change parameter's alarm limits with one button press to offer smart adjustment behaviour. Since the app works with parameters from different modes (monitoring, ventilation, defibrillation), the alarm

management should include those system states. This for example is necessary to mute all alarms in defibrillation mode or to separate alarms according to their mode.

Medical personnel use the system in critical situations, so 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 incorrect 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 system. Reliability on the other hand should reflect the extended need of the users, that the system needs to be always available and responsive. This also means that users can rely on the functionalities of the system.

While the system will be used in critical medical conditions, another goal of the design is high performance in all situations. This means that the user does not encounter loading screens and graphs or situations while performing standard tasks with the system. In addition to that, the system does always deliver the expected screens and settings when the user asks for them without having to wait. On the other hand, this means that the required graphics and data should be displayed even if there is a delay in reading the data.

A design improvement is one of the main goals of the project. Ease of use and a user-friendly interface did not play a major role in the development of the old systems. The interface design aims to display a modern style with fitting colour schemes to give the user an enjoyable and intuitive user experience. Another aspect of the design of the user interface and -experience is to increase the ease of learning. The future device will be used daily by trained doctors and paramedics. To prevent confusion while displaying parameters with only absolute values, the parameter widgets will have a consistent, recognizable design to enable familiarity, also in challenging situations. For example, it is an explicit design decision to add a title for the absolute parameters but not to the graphs. Graphs are always displayed with an absolute value next to them where the name of the value is displayed. Considering the future screen size of 10 inches, the displayed widgets should be readable and buttons should have a size easy to tap.

In the established systems basic settings, like the alarm limits, cannot be reached effortlessly, which shows the need for better usability and ease of use. This applies to all other setting options and touch inputs. For the M-Lab project, the system will be available for touch only in contrast to the future device, which will also have hardware buttons. Those hardware inputs will be neglected for the application. The whole system should be implemented modifiable and extensible. Physical sensors, that are part of the showcase, should be added easily with fitting data for absolute and graphical values.

3 Subsystem decomposition

The original system consists of 3 individual subsystems. In this project, it is the task to unify these systems, which is why there are different approaches for the subsystem division.

The first idea to implement the application with a division into monitoring, ventilation and defibrillation was neglected because coupling cannot be avoided completely. Each subsystem has to know some subparts from the other systems. Furthermore, the application needs to combine the three previous devices and should not convey a feeling that the devices are split into individual screens on one device. Nevertheless, to reduce coupling the interface will be divided with Flutter widgets into smaller subparts. With that, it will avoid redundant code and increase reusability. For instance, the heart frequency graphs in the monitoring subsystem can also be used in the ventilation or defibrillation mode. In addition to that, the ventilation subsystem will also have different graphs, for example, a flow graph, which can also be generated from the same graph widgets. Those Flutter widget decisions will be more common in the project.

Since that updated data needs to be displayed on the screen directly, the Model-View-Controller pattern will serve as the basis for the application. Furthermore, the decision was made, because the input of adjusting alarm limits needs to be bound to each parameter and the alarm management also needs to communicate with the data for playing sounds. The following interactions and the MVC pattern are shown in the UML Class Diagram on page 10.

With the help of the MVC pattern, subsystems will be separated from each other and let them communicate coherently. Controllers are used to switch between the views and to insert alarms or sounds onto the screen. Each of these views is provided with data or controlled by controllers. Data models provide mockup data and alarm settings among others. In addition to that, the application will offer demo scenarios. This subsystem is special for the project and needs to be inserted within the MVC pattern. The purpose of the scenarios is to provide mockup data within a timeframe to make the app more testable by participants of usability studies. This provides input to the system which is decoupled from the actual subsystem relationship. It has to interact with the model and the controller, but its interaction is only one-way from the scenario subsystem.

4 Hardware/software mapping

In a productive environment with the future 3-in-1 device, the application would be involved in a system that provides data. In this case, data would be fed into the system via an API. In the research project of the UKE and the other project partners, this scenario is still unclear and therefore not completely defined. As a reference see also page 9 for the Deployment Diagram.

The App simulates every graph or absolute value with mockup data. This means the app will not get real-time data from an external system. For the graphical presentation and showcase that are not part of the patient scenarios, Weinmann Emergency provided extracts from real-life sensor data. These extracts will be restructured to fit in the application and they will provide a good structure for the future 3-in-1 device. With this data, the model and more generally the core app can be prepared for the future data structure. The structure of this data serves as the basis to generate the datasets for the patient scenarios. These datasets will not have the accuracy of real data. But they will fit the content of the given patient scenarios. The advantage of this decision is that everything which happens in the core app does not have to be redesigned by the customer again. Accordingly, the most important point in the mapping is that the model is provided with correct mockup data, which can also be replaced via an API interface.

5 Persistent data management

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

The screen of the system contains multiple graphs and absolute value parameters. These graphs and absolute values are usually fed into the system using sensors, analogue to digital transformers, filtering, and some logical interpretation of the read data. For the scope of this system, it is not possible to read and work with live data from actual sensors since this is a proof of concept running on a tablet device with predefined data. Still, 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. For this task, at least semi-realistic data is required and provided by Weinmann Emergency as mentioned in section 4. Furthermore, for the simulated data of the patient scenarios, certain values are needed.

When starting the application and choosing a patient type, a standard scenario will be displayed by the system. The data extracts provided by Weinmann represent the graphs data and values for the application. The extracts can be lined up or looped very well due to their basic structure. The absolute values are generated by the team itself fitting the standard or the explicit patient scenarios. They are generated with Python in Jupyter Lab using a Random-Walk-Function, which is an algorithm to generate randomised numbers, for example, to simulate movement in a graph. The standard scenario is 13 minutes long and will be looped. The main purpose of the standard scenario is to provide a sandbox to testers to explore the functions of the app. The patient scenarios are about 8 minutes long and will not be looped, because of the real-life situation they represent. Depending on the scenarios, the team calculated with Python how many data points are needed. A

special aspect of the project is that the graphical presentations are real-life extracts from real sensors provided by Weinmann, but lined up and looped by the team. They are not medically accurate, but they do meet the requirements of a prototype and represent a proof of concept. For example, the average number of data points for a graph is between 25.000 and 500.000. Those will be stored in Jsons, a standardised file format to store structured data, on the device and will be read by the system when needed. By building the application's interface around these datasets it is ensured that realistic processing and interpretation of live data is possible. The advantage of using real log data 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. Furthermore, a transition to a future 3-in-1 device with actual sensors can be simplified since no translation middleware needs to be implemented. Lastly, this also opens the possibility to test our application with the same simulators that are used to test the actual Weinmann devices.

The core of our system design is the Data Model which contains a sensor key representing a real sensor. This sensor establishes a connection between ChartData - our data provider - and the devices that use such data. There are two types of data: regular chart data with a value and timestamp, and NIBP data that additionally has a systolic pressure, diastolic pressure, and mean arterial pressure (derived from the two attributes). This data is visualised by the graph entities and value boxes. The system state closely works with the data model to ensure persistence. It stores vital information about the states of each entity, for example, the device settings for ventilation and defibrillation and the preset values for each patient type.

6 Access control and security

Since the project is a proof of concept, mechanisms like authentication or data storage and encryption within a database are not necessary. Further, the application will be only available for medical personnel and won't be accessible for unauthorised people. Of course, the data and pieces of information which are fed into the system from the sensors are very sensitive corporate secrets of Weinmann and should therefore not get into the hands of unauthorised people. To minimise the possibility of this sensible data leaking, all future users of the application should be part of the development team of the future 3-in-1 device.

Since the 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 the project. Which differs from the requirements of a future 3-in-1 device. This entails that the 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

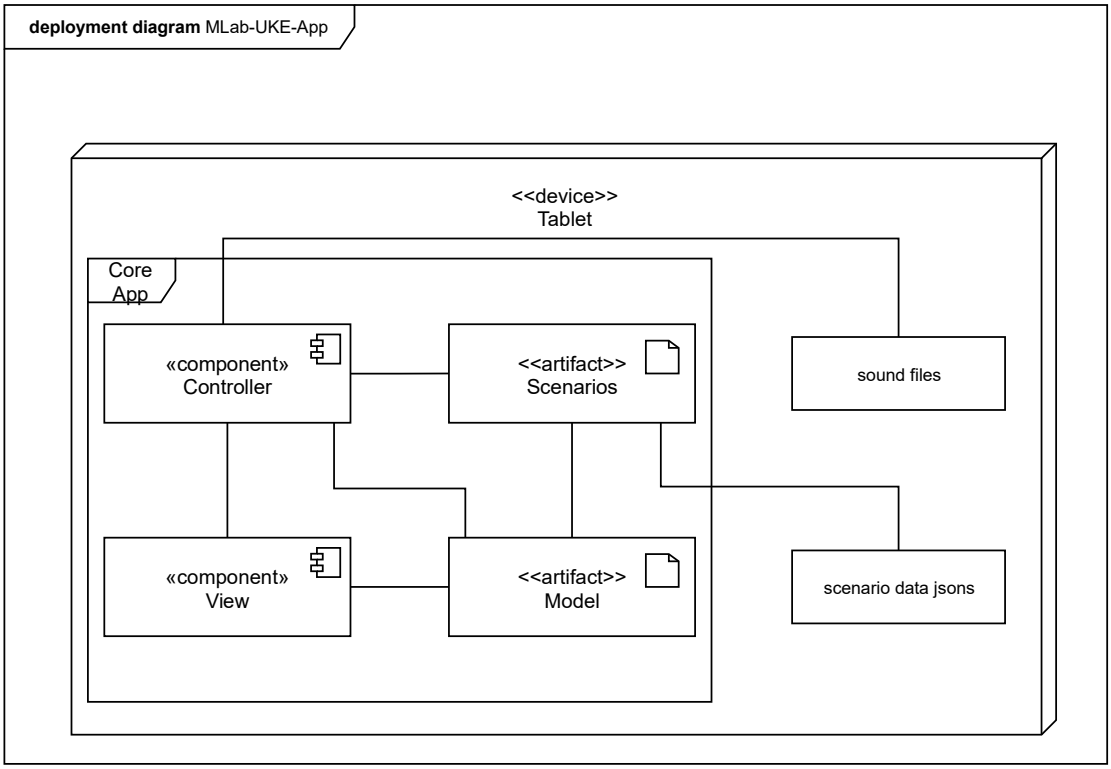
The system mainly controls the flow with the MVC pattern, which is illustrated as a sequence diagram on page 9. With the help of the MVC pattern, the team decided to use an event-driven architecture. An advantage of this decision is, for example, that the alarm management can be implemented easily and it can update the views using observer patterns. This pattern then acts for the mockup data as well as the alarms. The control flow is implemented using different controllers in the MVC pattern. The synchronisation of the graphs within the views is realised using listeners and the Flutter package GetX. For this purpose, each created graph has the listeners implemented directly.

8 Boundary conditions

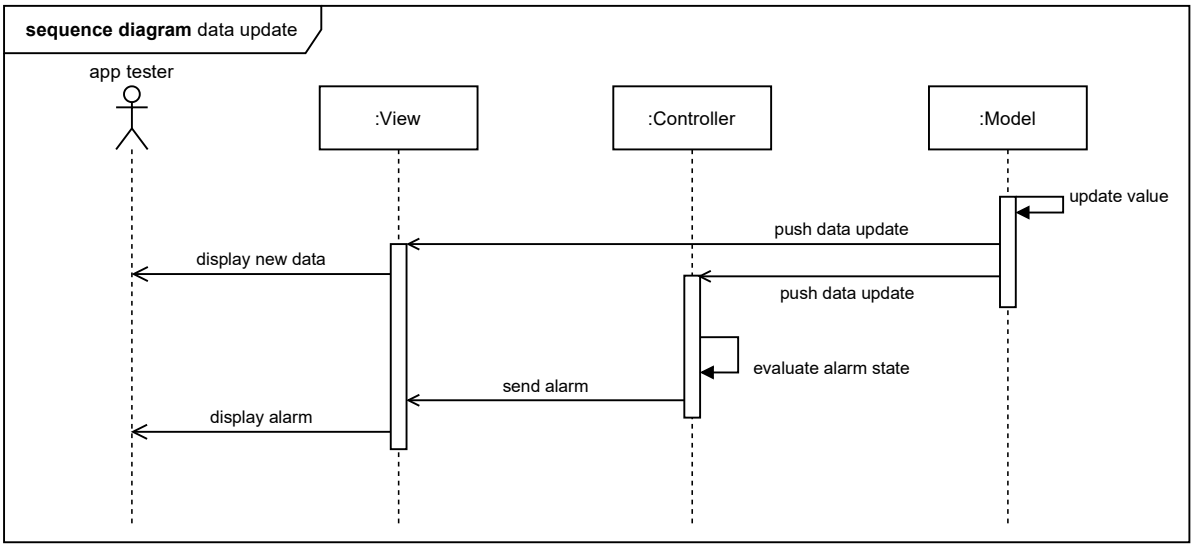
The app created in this project will run in landscape mode on a tablet device. The app is disabled by either turning off or locking the device. If a component fails, the app is restarted automatically or manually if necessary. In the scope of this project, this will not be a problem, because the needed data is provided as a mockup and not by living people. The same principle applies to a complete system failure. It is only important that the device's restart will be fast. This is necessary because the usage of the application is meant to reflect a real environment, where fast rebooting will be extremely important.

9 Diagrams

9.1 Deployment Diagram



9.2 Sequence Diagram



class diagram MLab-UKE-App

due to amount subclasses representing screen Elements are omitted, view Figma clickdummy to get an overview

display Alarm if set in SystemState

View

StartScreen
+ build(BuildContext context): Widget

AlarmLimitScreen
+ build(BuildContext context): Widget

1

display updates

MainScreen
+ systemState: SystemState
+ showMonitorScreen()
+ showVentilationScreen()
+ showDefiScreen()
+ displayAlarm()
+ build(BuildContext context): Widget

DemoScreen
+ build(BuildContext context): Widget

1 1

display updates

display updates

Model

holds reference to alarm Controller to pass to dataModels on init

ModelManager
+ alarmController: AlarmController
+ registerAlarmController(AlarmController): void
+ initDataModels(): void
+ loadPatientPresets(): void
+ loadModelEnvironmentValues(): void
+ resetAllModels(): void

1 initializes 1..*

1 initializes 1..*

single models addressed via Getx dependency injection using sensorEnumAbs as key

SystemState
+ screenStatus
+ patientType
+ dataBasedAlarmStates
+ generalAlarmStates
+ activeTheme
+ alarmState
+ themeModel
+ graphList
+ absAlarmTiles
+ smartAdjustmentMap
+ toggleTheme
+ setAlarmState(
sensorKey: sensorEnumAbs,
priority : int,
message: String,
alarmStatus: alarmStatusEnum
color: Color)

GraphDataModel
+ sensorKey: sensorEnumGraph
+ environmentValues
+ singleData: ChartData
+ graphData: List<ChartData>
+ updateValues(double): void
+ resetModel(): void

1..* 1..*

AbsoluteDataModel
+ sensorKey: sensorEnumAbs
+ environmentValues
+ singleData: double
+ alarmBoundaries: double
+ alarmController : AlarmController
+ updateValue(double): void
+ resetModel(): void
+ setLowerAlarmBoundary(double): void
+ setUpperAlarmBoundary(double): void

1..* 1..*

1..* 1..* 1..* 1..*

update Value (alarm Boundaries)

handle UI Events

update Value (e.g: screen status, theme)

set alarm status of the respective sensorEnumAbs

evaluate alarm on data update

play saturation HfBeep on update of spo2 or hf model

update values in Models

Controller

ScreenController
+ runningScenario: AbstractScenario
+ systemState: SystemState
+ modelManager: ModelManager
+ setLowerAlarmBoundary(AbsoluteDataModel, value): void
+ setUpperAlarmBoundary(AbsoluteDataModel, value): void
+ changeScenario(scenario: scenarioEnum): void
+ continueButton(addtitionalInfo:String): Future?
+ skipButton(): Future?
+ aedButton(): Future?
+ changeMainScreen(index: int): void
+ alarmSettingsButton(): Future?
+ scenarioMenuButton(): Future?
+ scenarioButton(scenarioKey: scenarioEnum): Future?
+ MenuExitButton(): Future?

AlarmController
+ _confirmMap: Map<sensorEnum , dynamic>
+ _soundList: List<SoundListEntry>
+ _alarmTypes: List<bool>
+ _soundCtronller : SoundController
+ _systemState: SystemState
+ _soundController : SoundController
+ initListeners : void
+ evaluateAlarmState(sensor: sensorEnumAbs): void
+ triggerAlarmState(sensor: sensorEnumAbs, priority): void
+ triggerConfirm(sensor: sensorEnumAbs): void
+ endConfirm(sensor: sensorEnumAbs): void
+ evaluateBoundaryAdjustment(): void
+ evaluateSoundList(sensor: dynamic, priority : int) : void
+ updateSoundList(sensor: dynamic, priority : int) : void
+ evaluateAlarmSound(priority : int) : void
+ evaluateSmartAdjustment(sensorKey: sensorEnumAbs, message: AlarmMessage)
+ triggerSoundController(sensorKey: sensorEnumAbs, status: alarmStatusEnum)

1

SoundController
+ alarmPlayer: AudioCache
+ ecgPlayer: AudioCache
+ alarmSoundFilesMap
+ ecgSoundFilesMap
+ alarmPlayer: Audioplayer
+ ecgPlayer: Audioplayer
+ play(soundKey: Enum): void
+ stop(): void
+ stopAudioPlayer(audioPlayer: AudioPlaye : void
+ stopAlarmTimer() : void
+ saturationHfBeep(bpm: int, spo2: int): void
+ playDefiSound(): void
+playAlarmSound(alarmSound : SoundIdentifier, timerDuration : int) : void

play alarm sound on evaluation

start/stop data model run

specific low level controllers omitted (e.g.: controllers for specific widgets)

Scenarios

AbstractScenario
+ isScenarioRunning: bool
+ scenarioType: scenarioEnum
+ loadData(): List<Map<sensorEnumGraph, data>, Map<sensorEnumAbs, data>>
+ startScenario(): void
+ stopScenario(): void

PatientScenario
+ runScenario(List<Map<sensorEnumGraph, data>, Map<sensorEnumAbs, data>>)
+ updataDataModels()

runs the called scenario onces

StandardScenario
+ runScenario(List<Map<sensorEnumGraph, data>, Map<sensorEnumAbs, data>>)
+ updateDataModels()

loops over Scenario Data