Entwicklung eines automatisierten Pillenspenders

Bachelor-Arbeit

zur Erlangung des akademischen Grades
Bachelor of
Science in Engineering

Eingereicht am
Bachelorstudiengang Medizintechnik, Linz
Fachhochschule Oberösterreich

von
Roman Iumatov

Linz, am May 24, 2025

Begutachter:
FH-Prof. Dipl.-Ing. Dr. Robert Merwa

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Linz, Austria, May 24, 2025

Roman Iumatov

Abstract

English

The goal of this project is to develop a prototype of an automatic pill dispenser, a device that would assist caretakers with scheduling and dispensing pills for people with limited motor capabilites. This document will cover the details of how it was done, the problems that have arisen as well as solutions to them. We will go step-by-step through each stage of development of such device and highlight challenges and solutions to the task.

Deutsch

Ziel dieses Projekts ist die Entwicklung eines Prototyps eines automatischen Pillenspenders, eines Geräts, das Pflegekräfte bei der Einnahme von Tabletten für Menschen mit eingeschränkten motorischen Fähigkeiten unterstützen soll. In diesem Dokument wird detailliert beschrieben, wie das Projekt durchgeführt wurde, welche Probleme aufgetreten sind und welche Lösungen es dafür gibt. Wir gehen Schritt für Schritt durch jede Phase der Entwicklung eines solchen Geräts und zeigen die Herausforderungen und Lösungen für diese Aufgabe auf.

Executive Summary

The end result of the project is a working prototype of the device. The specifications of the device were defined at the beginning of the project and are as follows:

1. **The device contains 21 chambers, 3 for each day of the week.**

   This requirement is a necessary condition for every pill dispenser, not just the automatic ones, because even the simple dispensers (normally) have a separate chamber for each day.

2. **The device contains a pill disposal system.**

   This is a place where unused pills are stored. This location is also divided into sections so that the caregiver is able to investigate which pills have not been taken and when. Meeting this requirement makes it easier for the caregiver to manage the device, as it is easy for them to analyze unused pills and dispose of the remaining pills.

3. **The device has an accompanying app for remote control.**

   This requirement enables the caregiver to control the device remotely. This is useful for the following reasons:

   (a) The device lacks a control panel on the housing. Therefore, the function cannot be interrupted by accidentally pressing a button.

   (b) With a single Android app, the caregiver may be able to connect to and manage multiple devices remotely. This is useful in hospitals or retirement homes, for example, where there may be several such devices in a confined space.

   (c) Alternatively, for use at home, multiple apps can be connected to a single device (not simultaneously) if there are multiple independent caregivers for a single device.

Contents

List of Figures

List of Abbreviations

1 Introduction

1.1 Problem Definition and Motivation

The main goal of this project is to explore the possibilities of developing such a device that can be mass produced in a decentralized way using conventional tools (3D printing) and at the same time be widely available due to the low manufacturing costs. The question is whether such a device can exist at all. While there are already ready-made solutions on the market, they are not cheap and lack certain features that are present in the prototype in question, be it the lack of remote controls, the lack of flexible scheduling or the lack of a pill disposal system.

1.2 Goals and Approach

The aim of this project is to develop a device that makes it easier for people with motor or mental disabilities to access their medication. This is to be achieved through automation, ease of use and ergonomics. A working prototype of this device will be developed as part of this project. Here are the initial requirements for this project:

1. The device should contain 21 chambers, 3 for each day of the week. This would be the default setting, but the schedule could also be configured to dispense pills either twice or once a day. In this case, the 21-chamber system does not correspond exactly to the 1-week dispensing time. The dispensing information would be available to the caregiver

2. The device should include a pill disposal system. The purpose of this system is to store the unused pills for safety reasons, to prevent accidental overdosing on previously unused pills and to enable monitoring of unused pills. Therefore, this system is also divided into 21 functional chambers.

3. The device will have a companion app. This is an Android-based application that the caregiver can use to configure, monitor and adjust the device.

As this project involves the development of a prototype, activities such as marketing, certification and industrial mass production are not part of the project. User testing is also not part of this project, as it requires the production of several devices, the collection of test subjects (i.e. establishing contacts with hospitals, nursing homes, etc.) and the statistical analysis of this data.

All of these are obviously necessary next steps after the development of the prototype, but for this project they are out of the ordinary.

1.3 Structure and Organization of the Work

This project is divided into several major steps:

1. **Design and Development of the Body**

   This is the first task: developing the basic structure. At the end of this step, a complete device should be brought into shape. We will look at the different options for developing the structure, discuss their pros and cons and choose the most suitable one for the next step.

2. **Construction and 3D-Printing of the Design**

   This is the step where our design takes on a physical form. This is the prototyping step where we iteratively develop the device to make sure the design works. This is also the step where the material and mechanical properties of our device are our biggest concern. At the end of this step, the device is functional but not yet configurable. This means that it will be able to rotate one chamber at a time.

3. **Development of the Android APP**

   This step is similar to steps 1 and 2, but for an Android app. We design and develop the template for the app, which can then be filled with functions. In this step, we will also determine which functions the microcontroller on our device must output to the app. At the end of this step, we would have a working device and an app, but they are not yet connected, which brings us to the next step.

4. **Development of the interface between the device and the app** This is the final step where everything comes together. The device will communicate with the Android app. It will be able to send (usage statistics, current time, time until next delivery, etc.) and receive (configuration settings, forced delivery command, etc.) information.

The table contains more detailed and precise steps to be taken during this project. Please note that it has been expanded and therefore includes more steps. This means that several steps from the table below are combined into one step from the list above. More specifically, steps 1.0 and 2.0 belong to the Design and development of the Body section, and steps 3.0 and 4.0 belong to the Construction and 3D-Print of Design section.

Work Package	Input	Activity	Goal
1.0 Initialization	Definition of the precise project requirements, identification of the target audience, execution of the bureaucratic process, definition of the project scope.	Communication with stakeholders	Clearly formulated objective, structured development plan.
2.0 Physical Requirements	Requirements for the device's physical properties: materials, mechanisms, size, robustness requirements, ergonomics.	Analysis of the target audience and communication with stakeholders.	Definition of the boundaries for the proposed designs.
3.0 3D Printing and Construction of a Functional Prototype	Requirements for the materials used, 3D models of the prototype.	Adapting the models for 3D printing, printing the prototypes.	Physically existing device with all functional features.
4.0 Implementation of the Delivery System	Constraints from the previous step, ideas for implementation, feasibility of different approaches.	Defining and finalizing a 3D model of the delivery system, 3D design, optional 3D printing of a prototype.	Functional delivery mechanism.
5.0 Implementation of the Control System	Requirements for ergonomics and functionality.	Selecting the microcontroller, connecting the microcontroller to the delivery system, creating skeleton functions for the required device features.	Delivery system and control system are interconnected and can be programmed, making the delivery system controllable.
6.0 Programming the Remote App	Requirements for the remote app's functionality.	Programming the app (in a high-level programming language).	Application capable of controlling the device over a wireless network.
7.0 Programming the Control System	Skeleton functions implemented in the previous steps.	Programming the microcontroller.	The device functions correctly.

Table 1: Project planning table

1.4 Tools and Technologies

For each step there is a different toolset that will be covered here and then mentioned in the chapter of the particular step if necessity arises. Here are the tools that have been used:

1. **Organisation and Project Management**

   - **GitHub**
     Is a version control tool that has been used for bug-fixing, change tracking and logging of the work done. It is used extensively for Frontend and Backend development steps. It is also used to synchronize work from different machines. The project is organized in such a way, that source code for Microcontroller, Android App are all within the same repository.
   - **AI Tools**
     These tools are used mostly for troubleshooting and debugging. The process is usually this: When there is an issue that is hard to resolve, the description of the problem and all relevant information is then fed into all 3 of the used Language Models, the final solution is typically derived from multiple suggestions, rather than adopted verbatim from a single model. This is done so because these models can be confidently incorrect at times, which, if their advice is taken without understanding, can lead to a deeper level of confusion. It will also be mentioned in the work whenever AI was used to generate solution. All AI-generated code was subsequently reviewed and adapted by the author; responsibility for final content remains with the author

     These three are used together in a way described earlier:
     - **OpenAI ChatGPT o3**
     - **Google Gemini 2.5 Pro + AI Studio**
     - **Deepseek R1**

     GitHub Copilot is integrated within Visual Studio Code (VSCode) as a plugin. It is sparingly used whenever there are smaller issues, mostly syntax errors to be fixed, as unlike the three models mentioned above it is not a thinking model.

2. **3D Model Prototyping and Design**

   - **Fusion 360** was the primary Computer Assisted Design (CAD) software used for all stages of 3D modeling. It is the main and only tool used for Prototyping Earlier Sketches and creating 3D models of the finalized prototype. Unlike other 3D-Modeling software that relies primarily on meshes (Blender, 3DMax), Fusion 360 Uses parametric design which is more precise, with further slicing for 3D-Printing (further facilitated by built-in export to .stl feature) and contains a wide toolset of analytical features (such as Interference, Center of Mass and Section Analysis which were used extensively). Its parametric nature also allows easy adjustment of dimensions of components whenever an overlap between them happens.
   - **PrusaSlicer** This software is used as an intermediary step between a 3D-model and 3D-printer. Since printers require a set of commands to execute and a .stl is a geometry data, there needs to be an interpreter software which

would translate this data into an instruction set (G-Code, although PrusaSlicer has a very convenient feature of transferring this code directly to the printer if computer with the slicer and 3D-Printer are connected to the internet). All of the prints have been done with the default built in template for Prusa MK4S called **0.20mm STRUCTURAL**. Some models required support material to be added and since original template doesn't have them enabled (either for autogeneration or for printing) this template would then be adjusted to include printing supports.

3. **Frontend Development**

   - **Android Development Studio + Kotlin** This combination is a industry standard in Android development. Google has established that Kotlin is preferred language for developing an Android apps [1].

     Android Development Studio Integrated Development Environment (IDE) is a joint project of Google and JetBrains, an innovative IDE development company. it has many powerful features such as Previews, Device Emulators, Layout Editor and Gradle Build System. These features have alleviated the development process.

     Kotlin was chosen because an alternative would be Java, which is not personally preferred development language. Besides being industry standard, Kotlin has very easy-to-understand syntax, *Null-Safety* (a feature of the language, which prevents certain variables to have a null values i.e. value, the pointer of which doesn't have an address), that prevents very common Null-pointer exception runtime errors and also google's official trainings and tutorials which were essential in learning the language.

4. **Backend Development**

   - **VSCode + Platform.io + C++** Similar to the Frontend development step, this is also quite common choice for programming. VSCode offers a wide variety of plugins, due to being open source, and Platform.io is one of them.

     VSCode is chosen mostly because of ease of use and flexibility. It is Open-Source, therefore there has been a huge community effort to develop plugins and also one can adjust almost every component of the editor as one sees fit, for example by creating and running custom scripts.

     Platform.io has many important features, such as Microcontroller templates, built-in C++ and Arduino libraries, Device monitoring (data from connected device can be logged into terminal) and Upload of compiled code. As mentioned before, it is a standard for Arduino software development and enjoys a wide support from the community. This is important not only for the development of a device, but also for maintenance of its code in the future, should a project be further expanded.

     C++ Was chosen because of efficiency, due to being one of the lower-level programming languages, close to hardware, while also being able to abstract away, reducing the need to manually flip bits on the hardware. Furthermore, unlike C, another commonly used language for hardware programming, C++ supports Object-Oriented Programming (OOP), whose features (Incapsulation, Polymorphism, Inheritance) offer aid in programming a device.

2 Design and development of the Body

The development of a device usually involves several steps, starting with the design phase. In this phase, a decision must first be made as to how the device should look in view of the requirements that are set. The requirement to have 21 chambers is of particular interest in this context, as it significantly restricts the design options. However, this restriction should be viewed positively, as the requirements for the first steps seem to be too low. There are no requirements regarding the physical characteristics of the device, such as size or weight. This opens up a remarkably wide scope for design with regard to the external appearance of the device.

Based only on the information we have at hand, there are already some initial ideas about what the device might look like, each with its own advantages and disadvantages. Here is a brief overview of the possible designs we have come up with.

2.1 Early Ideas

2.1.1 Pill Dispense System

When thinking about how the pills should be dispensed, a number of ideas inspired by the environment emerged. Below you will find brief descriptions of the systems and also an image 1 of what they would look like. Please note that we are not going into too much detail here as they were the result of a brainstorming session and after all that, only one idea would be developed moving forward, albeit with certain inspirations from ideas that were dismissed.

- The **Revolver** system was the simplest, and it didn't take long to come up with this idea, because such devices had already been developed[2]. Its advantages are the simplicity of the mechanism, the robustness and the low price, while a disadvantage is its size and it requires a motor to drive a fairly large wheel.

- The second design is that of the **Carousel**. The main idea is that it is much smaller and uses space more efficiently than the revolver system. The smaller wheels would rotate at a lower speed than the main arm that holds them, as they are attached to the main axle with a reduction gear and a belt. Although it is slightly more space efficient than a revolver system, it is much more complex and still requires a motor to control the entire structure.

- The third design is that of the **Conveyor Belt**. On the image 1 you can see that it is round, but it could theoretically also be designed elliptically or with other, more complex curves. The principle is as follows: An output chamber is attached to the belt, which moves a series of these chambers along a specific path. It is similar to the revolver system, but the chambers would be removable and the conveyor belt would allow some flexibility in the design of the device. The main disadvantages are the need for a motor, the need for a chamber management system and the overall complexity of the device.

- The fourth design is inspired by **Vending Machines**. It would have 21 chambers, each with an electromagnet attached. Each time a pill is to be dispensed, the electromagnet is activated, causing the chamber to move and the pills to fall onto the dispensing area. In this design, the stepper motor becomes redundant, but

simplicity is lost as a logical circuit is required to connect 21 chambers in parallel. Theoretically, a parallel circuit would be advantageous, but this is not the case for us, as we cannot leverage this advantage sensibly.

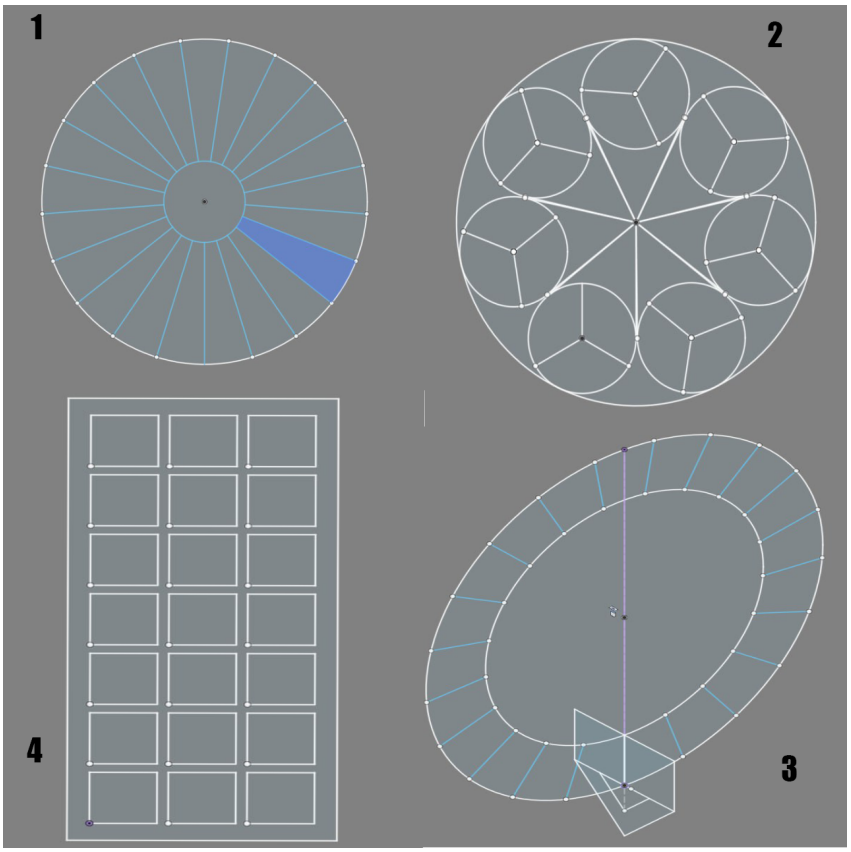

Figure 1: Initial ideas on how the main structure (dispenser) of the device should look. 1. Revolver system. 2. carousel system. 3. conveyor system (running sushi). 4 Vending machine system

2.1.2 Pill Disposal System

The idea for the Disposal system at the time was that it would mirror the dispensing system, either by being positioned directly underneath it or in such a way that the tablets would first fall onto a ramp, which after a certain time would be transported into the return mechanism by another action. It is not difficult to see that the choice of disposal system is highly dependent on the dispensing system we develop, so a more detailed description will be given later when we look in more detail at the chosen dispense system.

2.2 Selection of the Design

Of all the proposed designs, the revolver system seemed to be the most sensible choice, but it required some improvements in the design. A brief market research (e.g. [2] [3]) showed that there is not really such a system that would meet the above requirements of having a built-in disposal mechanism. For this reason, the design would have to be improved.

2.2.1 Improvements to the Revolver system

Now that we have chosen the main mechanism, we need to work out the details. The design philosophy was to make it as simple as possible, with as few moving details as possible. The first ideas were not very successful in this respect. In the picture 2 you can see what the earlier ideas would look like. In such a design, each chamber would have its own door, under that door would be a notch for the lever to fall into to open it. The pills would then fall into the disposal tray. After a certain amount of time (15 minutes), the S-shaped arm on the disposal platform would rotate and drop the pills further down the ramp into the disposal storage. This solution had numerous disadvantages:

1. **Too many moving parts**

   Each chamber has 21 doors. The doors need a lever to open (22), this lever needs to be translationally engaged and disengaged in the notch (23), the revolver mechanism needs to rotate (24), the disposal arm also needs to rotate (25, but the idea was to synchronize it with the main axis using belts), the disposal chambers would also need to rotate (26). Although this was a solution, it was not optimal given the number of moving parts. Although the dispensing chambers, the disposal chambers and the disposal arm would rotate around the same axis, the rotation and translation of the lever for an opening would require a complex mechanism that would further increase the effort.

2. **Requires a large dispenser bowl**

   The disposal mechanism itself is already quite large (a diameter of 300 mm was measured for this prototype). If another horizontal surface were added next to this surface, even if it were only half the size (a reduction in size would make it difficult for people with motor disorders to remove the pills safely), the entire system would become extremely large.

3. **Contains several small objects**

   This is not an obvious disadvantage, but each trapdoor must rotate around a specific axis. This axis must consist of either a small metal rod inserted into the openings of the door or 3D printed teeth on the design of the door itself. The first solution requires a daring and precise assembly of 21 doors, the second requires a high level of 3D printing precision that cannot be achieved by conventional means.

Due to these disadvantages, the system was discarded and never further improved. However, one important lesson was learned: if we want to have the simplest design, we have to utilize the power of gravity.

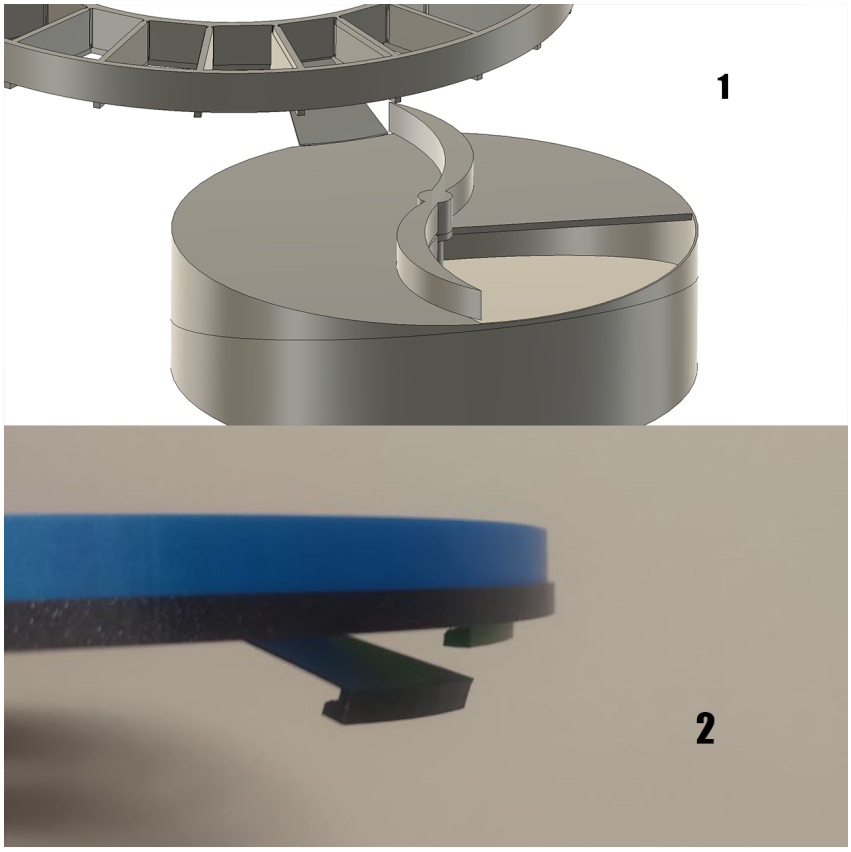

Figure 2: First revolver prototype. Here you can see that initial idea was to have a trapdoor in each chamber which will release the pills onto the disposal dish by opening it. (1) Is a designed prototype, (2) is a photo of 3D-printed prototype

2.2.2 Redesigning the Revolver system. A new approach

Having realized that we need to get rid of the extra space that the separate body at the side of the main body takes, the new approach was required. This time, the disposal chambers would be directly underneath the dispensing chambers and we would use the power of gravity to move from one chamber to another, as you can see in the image 3. This would drastically reduce the amount of space the pill dispenser takes, significantly reduce complexity, setting all the rotating objects onto a single axis.

Because this system doesn't have a separate mechanism for removing unused pills, another new method would have to be developed for this too. In the image 4 you can see how the chambers of the disposal system look like. The cover of the disposal system has an opening (part 2 of the same image) that is shut when the pills have to be taken. When the upper mechanism rotates to dispose new pills, the lower mechanism rotates to send pills into disposal chamber simultaneously. Ridges are added to the cover of disposal system to prevent pills from accidentally rolling away.

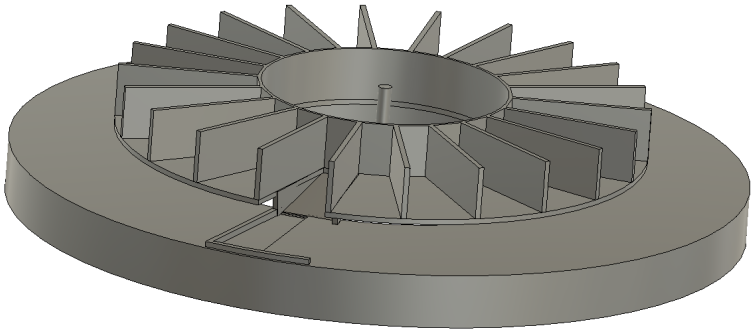

Figure 3: Second variant of the pill dispenser, with unevenly sized chambers

However, there are still ways to improve this approach. While much better than the previous one, this one also have some disadvantages:

1. **Chambers of the upper mechanism are smaller.** The chambers of pill storage are significantly smaller, as clearly seen on image 3. We can calculate chamber volume using the following formulas:

$$A = \pi r_1^2 - \pi r_0^2,$$

$$V = A \cdot h$$

   For the upper chamber: $r_1 = 100\,\text{mm}$,$r_0 = 47.5\,\text{mm}$, $h = 18\,\text{mm}$, the result is:

$$V = 437.90\,\text{cm}^3$$

   For the lower chamber: $r_1 = 135\,\text{mm}$, $r_0 = 80\,\text{mm}$, $h = 18\,\text{mm}$, the result is:

$$V = 668.69\,\text{cm}^3$$

   As we can see, the lower chambers are almost 1.5 times bigger than the upper ones, which is a waste that we might want to get rid of.

2. **Synchronous movement of both chambers** This approach makes future development less flexible. independence of the chambers is a degree of freedom that we might want to keep somehow. The biggest problem with current approach is that it might lead to situations where some pills might fall too fast from the top to fall directly into an opening in time.

3. **Reliance on a ramp for pills to fall down.** This can introduce an undesired behavior where some sticky pills might stick to the ramp and not fall down. This can lead to very dangerous scenario where a pill, not properly dispenced in time, would fall down with the next batch which would then lead to overdose.

As mentioned earlier, this design is already what we would like to see as a final result, but we can improve it further, by redesigning it a bit and adressing the disadvantages we mentioned above.

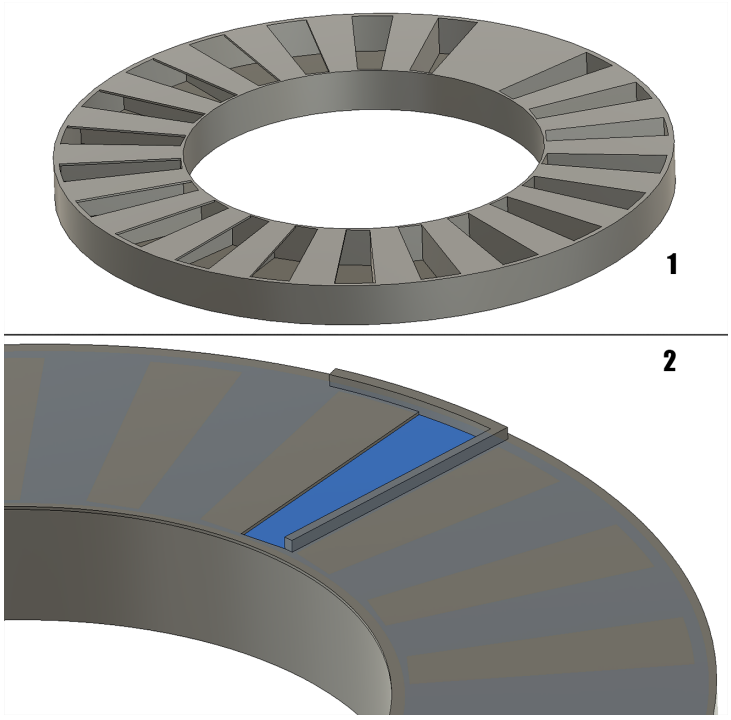

Figure 4: Decomposed Disposal system of 2nd revolver system. (1) Shows the Lower Mill without cover. the whole construction would be rotating, covered by a ring with a cutout, seen on (2)

2.2.3 The final Redesign. Improving on what's good.

A few thoughts come to mind as a potential approach to the redesign. Using gravity is obviously good, we need to continue doing, however we might change how we use it. Having 2 systems one above another is also a great idea, what if we can make the chambers the same size? We could reposition the holes. Regarding synchronous movement, we have 2 types of direction of pill movement: From dispense to patient (intended behavior) and from dispense to disposal (backup behavior). The intended behavior implies human interaction, we can make some simple movement to deliver pills that would remove the need for synchronicity from our system. All these ideas are implemented in the final design, however it is worth mentioning that there has also been the intermediary step, where the dispense mechanism was rotated 90 degrees (see image 5), so that it is completely vertical. It is worth giving a short overview of why this idea wasn't chosen as final:

- **Too tall** the wheel with diameter of 100mm would be vertically positioned which takes a lot of space.
- **Requires more torque** A motor would have to directly resist the force of gravity to lift the pills up a mill
- **Doesn't fix synchronicity** The mechanisms are still locked together, this issue remains unresolved.

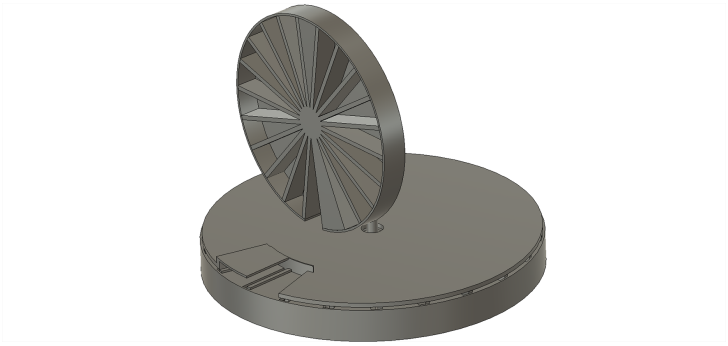

Figure 5: Pill dispenser with dispense system rotated vertically.

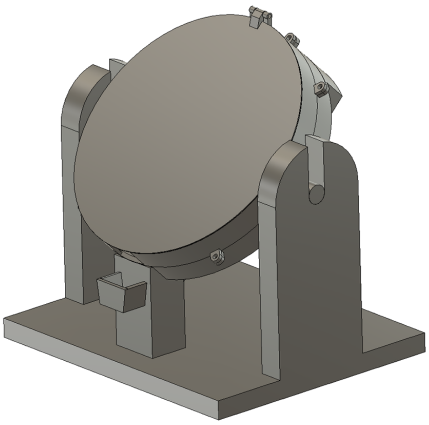

Figure 6: The final design of the Pill Dispenser.

For the final design, the following steps have been taken to address the issue:

1. **Resize the chambers** Chambers size were changed so that they match. The result is device now has a cylindrical form where the chamber mechanisms are located directly on top of each other, maximising the space used

2. **Remove the ramp, change disposal mechanism** The disposal mechanism was also changed. It works in tandem with dispense mechanism, which was also changed. The pills are now dispensed by the same movement by which they are made available, namely by rotation of the dispense mill.

3. **Introduce a separate axis of rotation to dispense pills** We will use tilting movement of the whole device for dispense of the pills into a cup or a hand of a patient, but for that we would have the whole construction elevated.

4. **Expand the device to include the stand** As mentioned above, it needs to be elevated. The stand will provide not only the elevation, but also another axis of rotation, therefore decoupling the disposal and dispense mechanisms from one another.

In the image 6 you can see the final design. It has all the features mentioned above already covered. In the next chapter we will go step by step into the details of the design and see how the proposed solutions were implemented.

3 Construction and 3D-Print of Design

The final design differs significantly from the prototypes. It has seen many iterative improvements and thus deserves its' own chapter to cover all the changes to each component of the design. The final design consists of the following bodies:

1. **Stand** Is the stand upon which the device resides. Needed for introduction an another axis of rotation and decoupling dispense and disposal mechanisms

2. **UpperBody** Consists of a walls and a floor of a dispense mechanism with a cutout for dispense and disposal paths as well as connection mechanisms.

3. **UpperMill** Is a mechanisms that divides the whole dispense system into 21 functional + 1 service chamber (22 in total). Inner side contains a gear and teeth for connecting upper mill with the lower mill (synchronicity between them is still retained)

4. **LowerMill** Is a mechanisms that divides the whole disposal system into 22 chambers. Unlike dispense system, they are all the same, since chambers need to be of the same size as dispense system, but there is no need for the service chamber. it also contains grooves to reduce the surface contact with lower body and thus friction

5. **LowerBody** is a very complex detail. Firstly, it contains cutouts for the stepper motor, driver and the deck that holds further electronics. Secondly, it has "ears" that define the axis of rotation for the dispense pathway. Thirdly, it contains grooves for the lower mill to travel on rotationally, to optimize power spent on overcoming friction.

6. **LowerDeck** Is a cover for all the electronics inside. It contains grooves on the sides for it to slide into electronics chamber.

7. **Electronics** is the housing for microcontroller, battery and charger. it also contains cutout for the deck to slide into so that electronics is covered from the outside.

8. **Holder** is a mechanism of additional mechanical fixation of the electronics. Although the Electronics compartment was designed so, that all the components are fixed in place, it might come to the situations where excess movement (e.g. during transportation) might cause electronics to fall out. this mechanism prevents it.

9. **UpperDeck** Is an upper cover of the device.

Moving forward, we will go step-by-step into the design of each of those components and also will cover the nuances of 3D-printing them all. For now, however, we will compare the final design with the goals set and why this design was chosen in the end. The goals can be divided into 2 types: Initial requirements and the problems arisen during previous designs. Initial requirements are:

1. **The device contains 21 chambers, 3 for each day of the week.** This requirement is satisfied. The final design has 22 chambers, 21 of which can contain pills.

2. **The device contains a pill disposal system.** This requirement is also satisfied.

3. **The device has an accompanying app for remote control.** This is not implemented yet. Alternatively, this requirement tells us, that an user interface on the body of device itself is not required. The electronics of this device are chosen so that this requirement can be implemented later on.

The problems that arisen during the design are solved:

1. From first prototype:
   (a) **Too many moving parts** Final design only has 2 mills that are moving simultaneosly on the same axis and nothing else. This is significant improvement over the first design
   (b) **Requires a large dispenser bowl** the disposal system of the final design is located directly underneath the dispensing system and they have the same spatial dimensions.
   (c) **Contains several small objects** The only small object is the holder, all the other ones are big and static, which reduces the amount of potential failure points.
2. From first prototype:
   (a) **Chambers of the upper mechanism are smaller** Since the dispense and disposal mechanisms are of the same size, this is not the problem
   (b) **Synchronous movement of both chambers** While both mechanisms still move synchronically, the pathways of dispense and disposal were decoupled, therefore their synchronous movement is not a problem anymore.
   (c) **Reliance on a ramp for pills to fall down.** The final design doesn't have a ramp. the pills would fall directly down in the disposal pathway. In dispense pathway, the device would be tilted 45 degrees which makes certain that the pills would fall down.
3. From third prototype:
   (a) **Too tall** The device will never be completely vertical. Although the device is now at an elevation it is still shorter as when having to have one wheel aways be vertical.
   (b) **Requires more Rotational momentum** since both mills are now horizontal, there is no extra power spent to battle gravity. the mills will only normally rotate in horizontal position.

3.1 Components of the final design

3.1.1 UpperBody

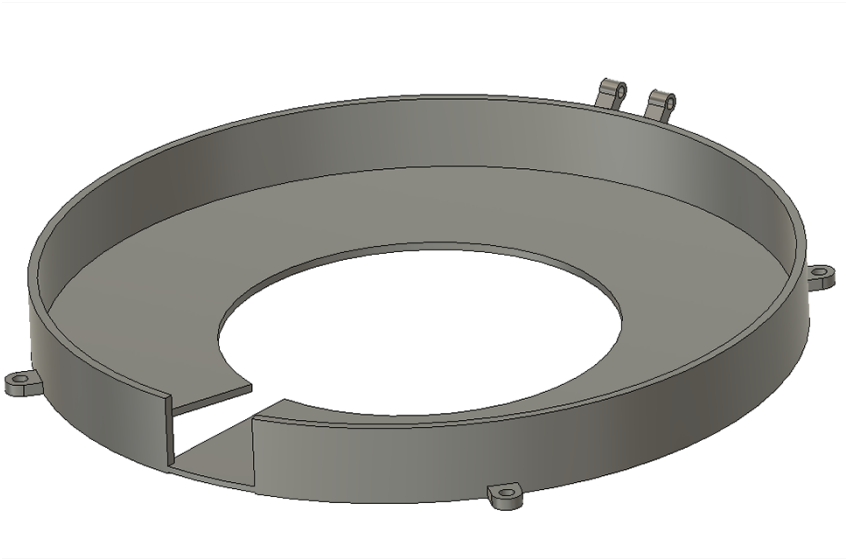

Figure 7: Upper Body

Upper body is a simple design consisting of a cylindrical shape with multiple cutouts. First one is on the floor, through this hole the pills that have not been taken would fall. The other cutout is in the wall, through this cutout the pills that will have to be taken would fall. In the middle there is a circular hole for housing the mill. under the cutout for the dispense pathway, there is a notch to prevent accidental rotation

The measurements are:

- outer radius = 100mm
- inner radius = 52 mm
- height = 20 mm
- thickness of floor = 2 mm
- thickness of wall = 2.5 mm
- notch depth = 0.4 mm

3.1.2 UpperMill

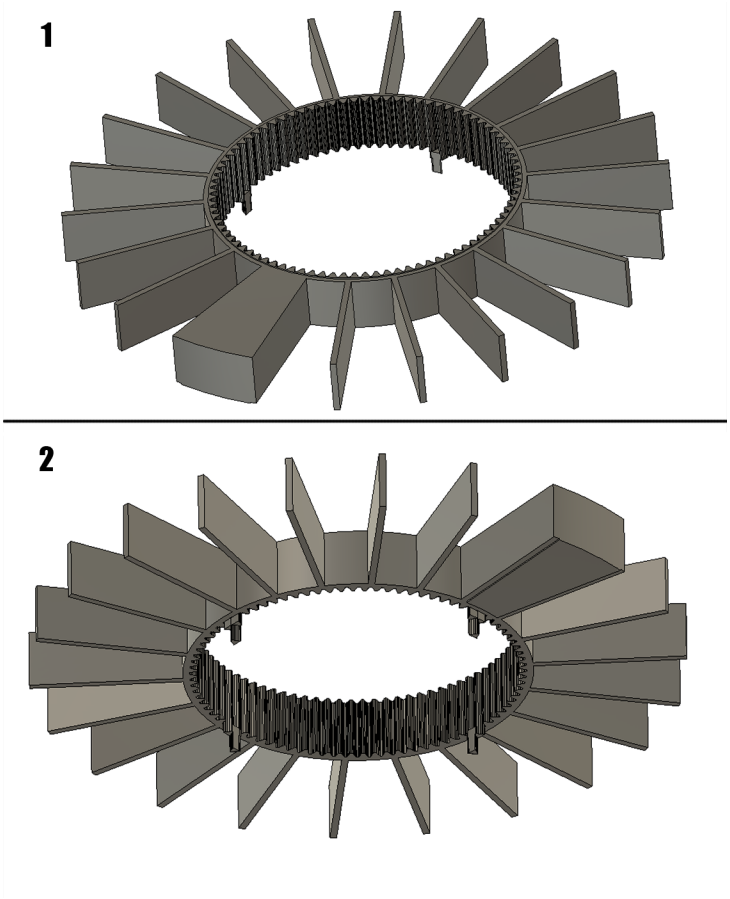

Figure 8: Upper Mill. (1) View from above. (2) view from below

Depicted on image 8, Upper mill consists of 21 functional chambers and one service chamber. the service chamber is needed to cover the hole in the upper body, so that at any time, 21 chambers are available. Inner rim is designed as a cogwheel. It has 88 teeth, it will be important later for the stepper motor movement calculation. The gear was designed using a free tool called DXF and SVG GEAR DXF GENERATOR [4]. These parameters were chosen to match the size of the cog and amount of teeth needed:

- Gear 1 Tooth Count: -88 (internal gear)
- Gear 2 Tooth Count: 22
- Module: 1.11 (mm)
- Pressure Angle: 20°
- Clearance: 0.15 mm
- Gear 1 Center Hole Diameter: 0 mm (no hole)
- Gear 2 Center Hole Diameter: 0 mm (no hole)

The lower side contains 4 teeth to connect the upper mill to the lower mill. This implementation makes sure that the two mills will move at the same time, removing need for a separate stepper motor for each of them or a connection directly to the upper mill.

There are certain design decisions that have been taken to reduce friction. Firstly, on the upper side the ring is offset by 0.4 mm so that the upper deck will not touch the whole mechanism, but only this ring. At the bottom side, the teeth are complex shaped. the upper part of the tooth has a length of 2.4 mm, which is 0.4 mm taller than the thickness of the floor of the upper body. The idea behind it is that the wings of the mill will not touch the floor directly, leaving 0.2 mm opening between the floor and the upper mill.

The measurements of the upper mill are:

- chamber height = 17 mm
- chamber length = 45.05 mm
- inner ring diameter = 100 mm
- outer ring diameter = 193 mm
- gear tooth depth 2.7 mm
- connecting tooth length = 2.4 mm + 5 mm

3.1.3 LowerMill

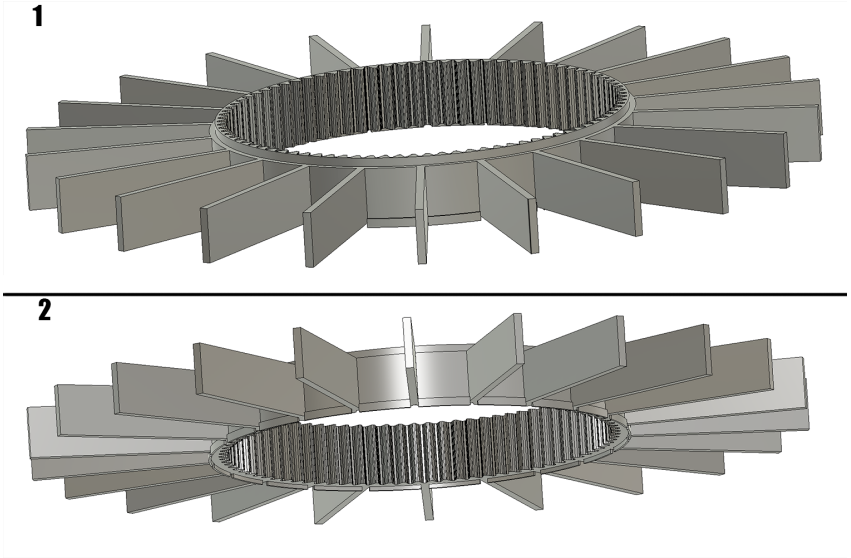

Figure 9: Lower mill. 1 view from above. 2 view from below

Depicted on image 9, Lower mill consists of 22 chambers which are the same. This is done to make sure that the sizes of upper and lower mill chambers are the same. Just as with the upper mill, inner ring is also a cogwheel, designed the same way and with the same parameters as upper mill. Unlike the upper mill, lower mill doesn't have teeth to connect. The teeth of upper mill are inserted into teeth of cogwheel of the lower mill.

The upper part also contains a ring that is 2 mm above the rest of the construction. This ring exists to eliminate the friction between the ceiling (which is a floor of the upper mill) and the chambers.

The lower part is also made in such a way as to reduce friction. it contains rims that would travel rotationally along the ring located on the lower body. they are rather tall, measured at 1 mm to make sure that they don't accidentally fall out during movement.

The measurements of lower mill are:

- chamber height = 13.6 mm
- chamber length = 45.05 mm
- inner ring diameter = 100 mm
- outer ring diameter = 193 mm
- gear tooth depth 2.7 mm

3.1.4 LowerBody

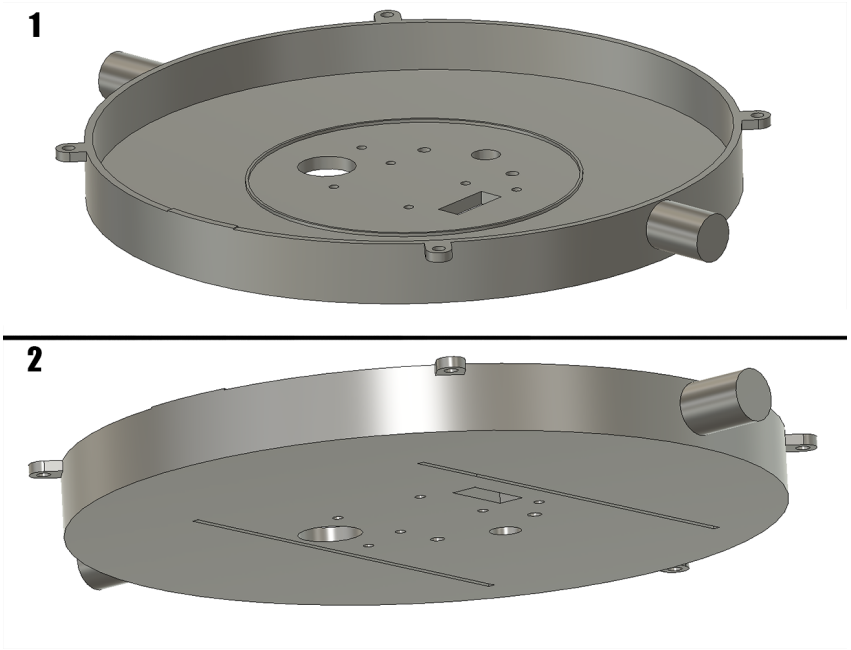

Figure 10: Lower Body. 1 view from above. 2 view from below

Depicted on image 10, Lower body is a bit more complex than the upper one. It is designed to connect multiple electronic components, Stepper motor, motor driver - directly, charger and microcontroller - indirectly, through the lower deck.

On the inner side there is a ring around which the lower mill rotates. This ring also separates the chamber area and electronics area. inside this ring there are cutouts for stepper motor, driver and wires.

On the outer side, there are "ears" upon which the whole device would be positioned on a stand. These ears define the axis of rotation for the dispense pathway. Outer rim contains a ridge to fix the the upper body and prevent accidental rotation.

On the inner side there is also grooves to append lower deck, these grooves are needed to remove the possibility of attaching the lower deck the wrong way. Grooves are 125 mm long, 2 mm wide and 2 mm deep.

The measurements of lower body are:

- radius = 100mm
- ring inner radius = 50.5 mm
- ring inner radius = 51.4 mm
- ring height = 1 mm
- body height = 20 mm
- thickness of floor = 4 mm
- thickness of wall = 2.5 mm
- notch depth = 0.4 mm

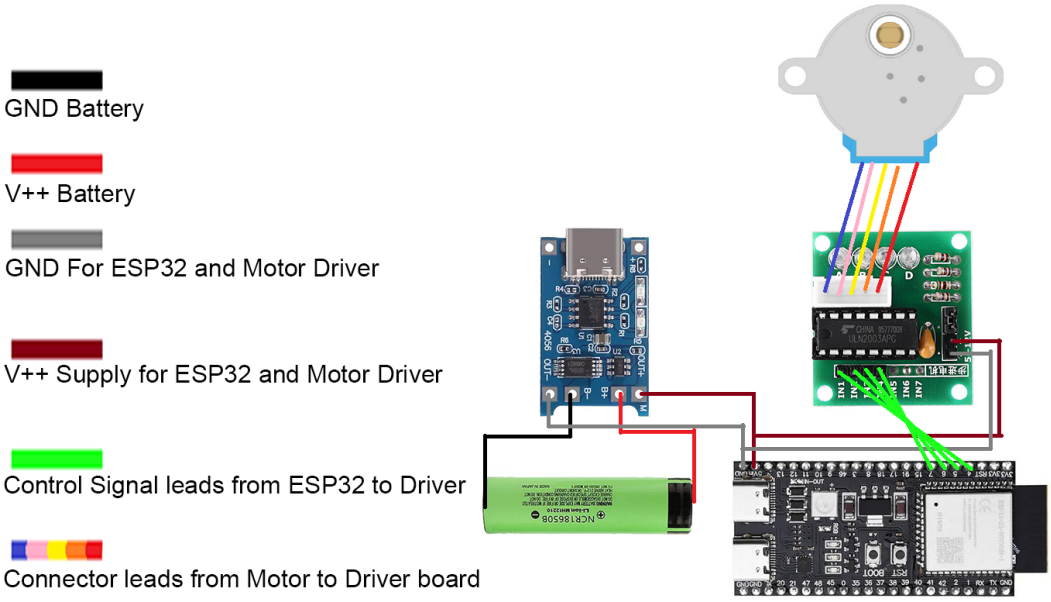

Figure 11: Connection schema of electronic components. Components from top to bottom: 1)28BYJ Stepper motor 2) Left: Charger Module 3)Right: ULN2003 Motor Driver 4)Left:Panasonic NCR18650B Battery 5)Right ESP32 S3 DevKitC-1 N16R8 S3 Microcontroller

3.1.5 Electronics

The device needs a microcontroller and a stepper motor for control. We need to know what we need for our electronics to filter out possible choices. Microcontroller needs to be able to control a stepper motor and also to have a network interface, Wi-Fi or Bluetooth, so that we can host an user interface somewhere else. Stepper motor needs to have enough torque to rotate 2 mills and a load. We don't need a high performance stepper motor, since our rotational speed is not that important. also it should be noticed that there is a gear reduction of 4:1 on the gears which results in 4 times higher torque for 4 times lower speed. From the measurements above we know that the diameter of mill is 200 mm, the weight is hard to calculate, but we can access the calculation used by the slicer software. The results are 44 g for Lower mill, and 61 g for Upper mill for 105g in total. The extra weight from pills would be around 100g for 205g in total. While there has been no calculation involved, the setup was tested with the coins to serve as a load, however it was tested only with the lower mill attached. We will come back to discuss the results in the later part of the document. Considering also that the one of the goals of the project was to develop the device for as cheap as possible, a 28BYJ [5] stepper motor was chosen, together with the ULN2003[6] Motor driver.

For the microcontroller, an ESP32 S3 DevKitC-1 N16R8 ESP32 S3 was selected [7]. it is a common choice to pair this microcontroller and stepper motor and it has the network interface (both Wi-Fi and Bluetooth). For us a Bluetooth Low Energy (BLE) is of particular interest. It allows us to save battery, since we don't need a constant stream of data exchange between a smartphone and device.

Another addition to the electronics is inclusion of battery. While not a requirement, battery-powered device is much more convenient to use as it makes it independent from being located near a socket. For this, Panasonic NCR18650B [8] Battery was selected. Image 11 Shows the connection schema of selected components.

3.1.6 Lower Deck(Electronics compartment)

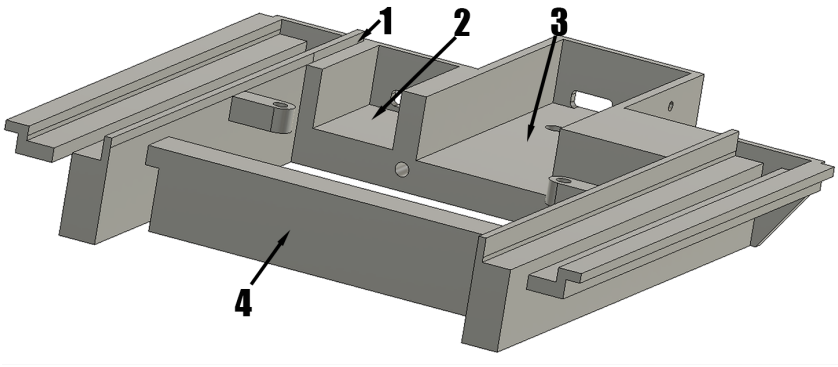

Figure 12: Lower Deck and its' features: (1)Ridge (2)Microcontroller housing (3)Charger Module housing (4)Battery Wall

The lower deck houses three electronic components: Microcontroller, battery charger and battery. It also serves as a cover for the stepper motor, whereas the driver would be located on the opposite side of the lower body, inside the ring. The shape of the lower deck is quite more complex than that of the previous bodies, therefore it makes sense to divide the description of it into multiple parts. on the image 12 you can see the lower deck and a numbers that point to the parts of components. Here is a description of all the components of this body:

1. **Ridge** Lower body has 2 parallel grooves on its lower part. These grooves will house these ridges on the lower deck. The ridges are designed to actually be a tiny bit smaller (1.6 mm vs 2 mm) than the notches of the lower body, we will discuss why in the later chapter. Ridges are 125mm long, 1.6 mm thick and 2 mm deep.

2. **ISP32 Microcontroller housing** This is where the microcontroller would be placed. We know the dimensions of the microcontroller (63.5 mm x 28 mm) and this opening is made to be around the same size to fit it (67.6 mm x 28 mm). It has cutouts on the body for 2 USB ports present on a microcontroller, as well as a LED.

3. **Charger Module housing** In this part the charger circuit will be nested. The cutout matches exactly the dimensions of the charging module (28 mm x 17.5 mm). This compartment contains also the cutout on the body for the USB port.

4. **Battery Wall** Battery holder is attached to this wall. at the sides one can notice that the connection to the side walls does not go all the way. The reason is that wires will go from the battery holder through these openings.

Lower deck also features a long hole for a slide-in lid which you can see to the sides from the Ridges that connect Deck to the lower body.

3.1.7 Stand

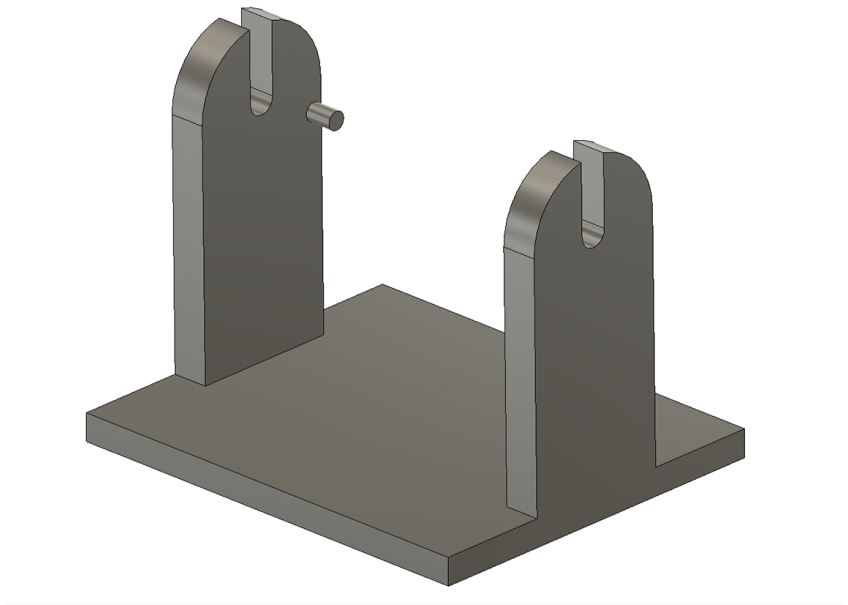

Figure 13: Stand

Depicted on 13,Stand is a simple design that features 2 distinct components: Pillars to the sides to hold the device and a dispensing dock in the front and center of it. The pillars have a cutout to slide down the device into it and notches to prevent the device from tilting backwards. The dispensing dock is designed to match the shape of the device so that it can easily be tilted into it at a 45 degree angle. The dimensions are:

- Stand height from bottom to the peak of stand: 190 mm
- Stand Width: 240 mm
- Stand Length: 200 mm
- Stand floor thickness: 15 mm
- Pillar thickness: 20 mm

It should be noted, however, that although this Stand is the default solution for the sake of completeness of the project, ideally, it would be manufactured not with a plastic, but rather with metal. The purpose of the stand (besides providing pathway for dispense of the pills) is also to provide stability of the construction so that people suffering from tremors would be able to lean firmly onto it and stabilize their hand. In this case, plastic might not be the optimal solution.

3.2 3D printing specifics

As mentioned earlier, one of the goals of the project is to make the device buildable with consumer-grade devices, among them 3D-Printers. This requirement introduces first physical limitations for the device, as 3D-Printers have a limited printing space. For this project as a reference printer, a Prusa MK4S 3D-Printer [9] was selected. From the Specification we can see that the build volume is 250 x 210 x 220 mm, putting a limitation on how large our device can be. Theoretically, of course, we could circumvent this limitation by designing the components in such a way that they would consist of multiple parts interlocking with themselves, however this adds unnecessary complexity to an already extensive project. The dimensions of each component mentioned in previous section conform to this limitation.

As for material selection, Prusa MK4S (and many other printers) support printing with Polylactic Acid (PLA) plastic. This material has many advantages: it is biodegradable (it is a polymer of lactic acid), lightweight, sterile [10], doesn't produce as much fumes(but still dues) as standard Acrylonitrile–Butadiene–Styrene (ABS) plastic [11], due to lower printing temperature and is easy to print with. However, certain disadvantages should be mentioned: Heat deflection at around 55-60 degrees [12] means it cannot be washed with boiling water as it would warp the body. Pure PLA is food safe [13], however the pigments might not be, so any wear might introduce microparticles that might not be as safe to consume if colored PLA is chosen.

Printing components that are exactly equally sized (e.g. a mill with diameter of 180mm to fit inside a ring with inner diameter) has a certain problem associated with it, which must be considered when designing and printing the device. The current models have already had this issue addressed: all interlocking components are made with certain tolerance (1mm for Mills, 0.2mm for Lower Deck). The Printer has nozzle diameter of 0.4mm [9], which means that horizontal (x-y axis) resolution is exactly that. It cannot print features smaller than its nozzle diameter, therefore, usage of small body features, extrusions etc was avoided during design process.

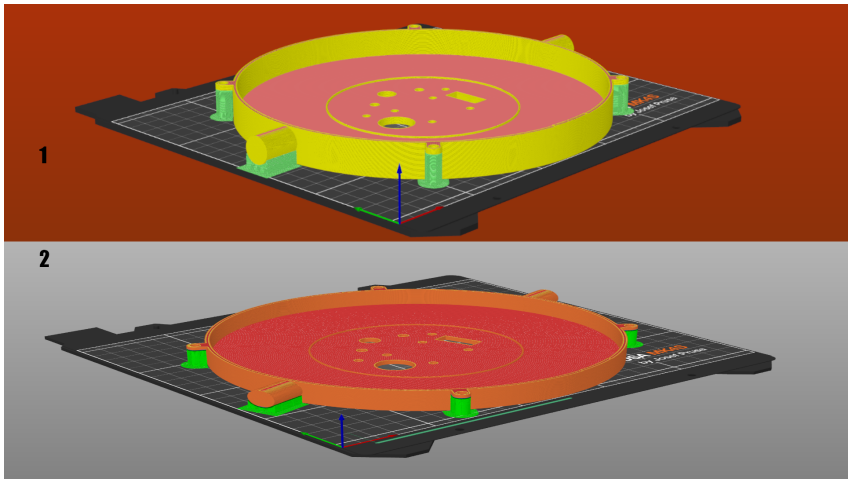

Figure 14: Issue when slicing. On the upper image (1) the supports are out of printing bounds, as the ominous red aura highlights at this problem. On the lower image (2) the problem is fixed. The component was rotated, so that the sticking-out parts of it are given a certain distance from the border to be able to draw support material

4 Development of a Remote-APP

AI Disclaimer: AI was extensively used to generate code for the App. The file structure, descriptions of main functions and architecture of the app as a whole were not designed by AI, because this information has laid down the groundwork for efficient use of AI to write this said code. Considering this, the chapter will not focus as much on code snippets with in-depth explanation of how the code functions, but rather on a structural overview and architecture of the project. File and app structure, most important functions and state selection will be covered here. The code provided by AI was reviewed and tested for its functionality, however it was not tested for the security (A common concern and weak spot of AI-Generated code), because of lack of such requirement and time constraints. Motivation and methodology for the use of AI will be discussed in the Results Analysis and Discussion section.

Existence of a remote app is one of the main requirements and a hardest one. The problem here lies in the amount of choices one needs to do upfront, which also contributes to uncertainty about the extent of the project, which will be talked about in Results Section in greater detail. For platform, Android was chosen. Alternatives were Web-based User Interface with Javascript+HTML or IOS. Android was chosen over WebUI or IOS solely from personal preference. A significant amount of time was also spent designing the structure of an app to make it ergonomic, to separate all the functions into different context windows.

The App is structurally subdivided into multiple parts:

1. **Bluetooth connection interface** (`com.example.dispenser.ble`) package Consists of 2 files: **BluetoothLeManager.kt** and **GattAttributes.kt**.

   BluetoothLeManager.kt contains all the functions needed to establish bluetooth communication with the microcontroller using BLE protocol, such as environment scan, dispenser identification, reading and writing data using BLE.

   GattAttributes.kt contains all the Universally Unique Identifier (UUID)s used by the microcontroller with names assigned to them, to avoid magic numbers in the code and replace them with named constants. This is a common practice everywhere, especially in microcontroller programming (where even a value of 0 is often rather depicted as LOW in the code, to avoid confusion)

2. **User Interface** (`com.example.dispenser.screens`) package consists of 3 files that represent their assigned context windows:

   - **DevConnScreen.kt** Contains the code for the UI of the "Device Connection" view. It contains the button to scan the environment for all the bluetooth devices, which are listed in the list under the **Found Devices**. At the very bottom there is also a connection status of the app to the device. When There is also a detection method (based on the device name) to detect the pill dispenser. The app will not attempt to connect to any other device.
   - **DevConfScreen.kt** Contains the code for the UI of the "Device Configuration" view. It contains all the settings regarding the configuration of device, such as: Time of dispense, schedule template load/save, view current schedule, ability to edit or delete current schedule, as well as adding new schedule template and the schedule save/load indicator. It also contains the information about on-board clock and the ability to set or change it

- **DevMonitoringScreen.kt** Contains the code for the UI of the "Monitoring & Status" view. It contains all the miscellaneous information about the device such as: Connection status, Last dispensed time, time until next dispense. The device logs are also available there. Logs contain information about bluetooth connection, information about past dispense events.
- `com.example.dispenser.screens.devconnection` package contains a class called **DevConnViewModel.kt** that contains functions related to updating the state of the connection view. It provides live update on information sent and received between the Device and the App, using `BluetoothLeManager`.
- `com.example.dispenser.ui.theme` package defines the application style: colors, fonts, themes etc.

3. `AndroidManifest.xml` Declares information about the App: Permissions, hardware requirements for the device, **MainActivity.kt** as an executive function (similar to main.c in C).

The next section will cover in specific detail the initial requirements given to the AI to generate the code, as well as the results of the code review and how it actually functions.

4.1 Bluetooth Connection Interface

The requirements for bluetooth connection on the side of Android App are as follows:

1. Scan for devices, identify the device that is a Pill Dispenser (using Name as an identificator)
2. Connect to the device using BLE
3. Read Characteristics(Generic Attribute Profile (GATT) Attributes):
    - **CHAR_GET_DEVICE_TIME_UUID** to get the current time set on the dispenser.
    - **CHAR_GET_DISPENSE_SCHEDULE_UUID** to get the currently programmed dispense schedule from the dispenser.
    - **CHAR_GET_LAST_DISPENSE_INFO_UUID** to get information about the most recent dispense event
    - **CHAR_GET_TIME_UNTIL_NEXT_DISPENSE_UUID** to get the remaining time until the next scheduled dispense.
    - **CHAR_GET_DISPENSE_LOG_UUID** retrieve a log of past dispense events.
4. Write characteristics (Also GATT Attributes):
    - **CHAR_SET_DEVICE_TIME_UUID** to send the current time from the App to the dispenser.
    - **CHAR_SET_DISPENSE_SCHEDULE_UUID** to send a new dispense schedule (time) to the dispenser.
    - **CHAR_TRIGGER_MANUAL_DISPENSE_UUID** to send a command to make the dispenser dispense pills immediately.

5. Manually disconnect from device

All these initial requirements were implemented in code. GATT attributes form an interfacing protocol contract between the mobile application and the microcontroller. This shared definition ensures both systems can correctly interpret the BLE services and characteristics. Consequently, these attributes are mirrored in the microcontroller's firmware, as detailed in Section Microcontroller Programming. As mentioned above, the Bluetooth connection interface consists of 2 files, **BluetoothLeManager.kt** and **GattAttributes.kt**. The first one contains all the functions and classes for the Backend-Frontend interface, while the second one contains GATT attributes and their corresponding UUIDs. Code block 1 is the full **GattAttributes.kt** file. This file contains all the Attributes that were present in the requirements, as well as some service attributes. The standard way of defining it is `val NAME_UUID: UUID = UUID.fromString("UUID")`, where **val** defines a read-only nature of the value (similar, but not quite the same as `const` keyword in C++), **UUID** = UUID... is the Java UUID class.

Listing 1: GATT attributes defined in a separate file

```kotlin
// GattAttributes.kt
package com.example.dispenser.ble

import java.util.UUID

object GattAttributes {
  // Service UUID
  val SERVICE_UUID: UUID = UUID.fromString("03339647-3f4e-43df-abff-
      ↪ fac54287cf1a")

  // Writable Characteristics
  val CHAR_SET_DEVICE_TIME_UUID: UUID = UUID.fromString("65232f1d-618
      ↪ a-4268-9050-0548142a4536")
  val CHAR_SET_DISPENSE_SCHEDULE_UUID: UUID = UUID.fromString("999
      ↪ c584e-06c0-49a1-995a-66b7c802ac1b")
  val CHAR_TRIGGER_MANUAL_DISPENSE_UUID: UUID = UUID.fromString("36
      ↪ bb95f2-e57e-4db9-b9aa-fb6541ee784e")

  // Readable Characteristics
  val CHAR_GET_DEVICE_TIME_UUID: UUID = UUID.fromString("272ee276-
      ↪ e37e-4d78-8c5e-bb7225d35074")
  val CHAR_GET_DISPENSE_SCHEDULE_UUID: UUID = UUID.fromString("
      ↪ b53c2ed4-ae26-476d-8414-011a025dddfc")
  val CHAR_GET_LAST_DISPENSE_INFO_UUID: UUID = UUID.fromString("40
      ↪ d3b5d8-5480-4b7b-a115-5fe86bf17d7d")
  val CHAR_GET_TIME_UNTIL_NEXT_DISPENSE_UUID: UUID = UUID.fromString(
      ↪ "4b14acc4-768a-43e1-9d6c-0d97307e2666")
  val CHAR_GET_DISPENSE_LOG_UUID: UUID = UUID.fromString("6f182da7-
      ↪ c5a8-40ab-a637-f97ed6b5777b")

  // Descriptor for enabling notifications/indications
  val CLIENT_CHARACTERISTIC_CONFIG_UUID: UUID = UUID.fromString("
      ↪ 00002902-0000-1000-8000-00805f9b34fb")
}
```

One interesting property of Kotlin as a programming language can be seen in this

code. Line 4 contains an import of a Java library, this means that Kotlin is interoperable with Java and can use its libraries.

Moving forward, some properties of the Kotlin language need to be established:

1. **Mutability**: References to immutable objects or variables cannot be changed once created. When modification appears to happen, a new object (with different reference) is created with the modification applied to new object only [14] [15].

2. **Null-safety**: default variable declaration forbids it to have a `null` value i.e. definition strictly follows declaration and memory gets immediately allocated to the variable. **Nullable** (i.e. those which are able to hold `null` value) can be created by appending a '?' symbol e.g.: `var b: String? = null`. It is compile-time safety feature aimed at reduction of runtime Null-Pointer exceptions [16].

3. **Coroutines and Flows** allow multithreading and simultaneous execution of multiple functions. Coroutines Provide means for writing asynchronous, non-blocking code in a sequential style [17]. Flows are asynchronous data streams [18].

4. **StateFlow** is a special type of flow that is often used in the code below. It has 3 important Properties: State-Holder (always contains a value), Observable (all state collector functions receive information about state change of this object) and "Hot Flow" (A flow is hot because its active instance exists independently of the presence of collectors. This is opposed to a regular Flow, which is cold and is started separately for each collector)[19].

5. **Scope Functions (`let, run, with, apply, also`)** allow execution of block of code within the context of a given object. Allow referring to an object in different ways and run operations with it as a reference [20].

BluetoothLeManager.kt Contains all the logic of the Android app related to interacting with the device. The code block 2 is the top-level structure of the file. At the highest level there is package definition, library imports, service constant TAG used for logging and main. This file itself is too big (740 lines of code) to do an overview line-by-line and its exact functionality is not the scope of this thesis. In this case, a top level overview of the contents of the file will provide the context for understanding the project as a whole, without digging too deep in technical implementation, specifics of the language and the Platform and their interaction with BLE protocol.

`BluetoothLeManager` is a class that contains all functions and values for the interface. It relies heavily on built-in Android Package called `android.bluetooth` [21].

The class takes 2 input parameters:

1. `private val context: Context` to get the `BluetoothManager` (and subsequently the `BluetoothAdapter`) using `context.getSystemService(Context.BLUETOOTH_SERVICE)` as an entry point to all Bluetooth operations. Consequently, also handles Permissions and GATT connections.

2. `private val coroutineScope: CoroutineScope` defines the lifecycle and context for any new coroutines launched within it. Since BLE operations are asynchronous, this handles it by defining a coroutine on the Android side for each operation.

Next, within the class, objects meant for the correct function of BLE are created:

- **_characteristicUpdate** is a private(visible only inside the class), mutable active flow. It's used internally by `BluetoothLeManager` to emit events when a characteristic's value is updated.
- **characteristicUpdate** is the public(visible outside the class), read-only version of `_characteristicUpdate`. External classes can read flow information from this variable to be notified of characteristic value changes. **Note:** StateFlows are paired like that in the code, in the next instances where public and private instance of the same StateFlow are present, both of them will be described under the same bullet point
- **discoveredCharacteristics** is a private mutable map that stores the objects called `BluetoothGattCharacteristic` that are discovered on the connected peripheral. Values received from the Device are written within this variable.
- **bluetoothManager** is an instance of the Android Built-in `BluetoothManager` system service[21]. This is the primary entry point for accessing Bluetooth functionality.
- **bluetoothAdapter** is the local device's Bluetooth adapter(hardware) object, also from built-in Android package [21]. Nullable in case when a device with this App on it doesn't have Bluetooth. With this all standard Bluetooth operations, such as switching on, scanning and connection can be made.
- **bleScanner** is an instance of BluetoothLeScanner(a method of built-in Android Bluetooth class) used specifically for scanning for BLE devices. `by lazy` is a keyword indicating that this object(`blescanner`) will initialize only when its parent (`bluetoothAdapter?.bluetoothLeScanner`) is initialized.
- **scanJob** is a nullable(notice the '?' symbol) Coroutine Job used to manage the lifecycle of an ongoing BLE scan operation. Allows asynchronous scan operation to be started, monitored and cancelled.
- **connectJob** is a nullable Coroutine Job used to manage the lifecycle of an ongoing BLE connection attempt. Allows asynchronous GATT connection.
- **_foundDevices** is private, mutable StateFlow that holds the current list of discovered BLE devices. it gets internally updated by the `leScanCallback` as new devices are found during a scan. While the public version of it Exposes the list of discovered devices to the `DevConnViewModel`, which allows the UI to display found devices.
- **_isScanning** is a private, mutable StateFlow indicating whether a scan is currently active. Used internally to track the scanning state and prevent multiple concurrent scans. Public version of it allows the UI to observe the current scanning status.
- **_connectionStatus** is a private, mutable StateFlow which holds the current connection status (e.g., Disconnected, Connecting, Connected, Error). Used internally to reflect the real-time state of the BLE link. Public version of it allows UI to display the current connection status and to make decisions based on this state (e.g., enabling/disabling certain actions).

- **_connectedDevice** A private, mutable StateFlow, it holds information not only about whether a device is connected, but also about the currently connected device. Public version of it Allows the UI to know which device is currently connected.

- **currentGatt** is a nullable instance of built-in android BluetoothGatt class from the android.bluetooth package[21] that provides GATT client functionality. When a device is not connected, this class can be `null`. It is a main object of all interactions with connected device.

The next 3 functions are made for permission checks:

- **hasPermission** is an utility function to check if a specific permission (passed as a String) has been granted to the application.

- **hasRequiredScanPermissions** Checks if all necessary permissions for BLE scanning are granted. It also contains security features that prevent the App from asking for permissions that might not exist on the device (e.g. when Android version is too old).

- **hasRequiredConnectPermissions** Serves the same purpose as function above, but this time for connection permissions.

The following two functions and one object handle the scanning function of the App:

- **startScan** is a function that initiates a scan to discover all advertising devices in the reach. It performs permission checks, Checks if Bluetooth is enabled, Sets `_isScanning` to true and clears `_foundDevices`, Starts the scan, Launches a `scanJob` to manage the scan, including a timeout (30 seconds) after which the scan is automatically stopped and finally, Updates `_connectionStatus` with an error if prerequisites are not met.

- **stopScan** is a function that Stops an ongoing BLE scan. It Checks if a scan is actually in progress, Calls `bleScanner?.stopScan` using the `leScanCallback`, Sets `_isScanning` to **FALSE**, Cancels the `scanJob` and finally, if there is an error during the stopping process, Handles potential Exceptions during the stop operation.

- **leScanCallback** is an instance of `ScanCallback` class provided by the bluetooth.le package (it is different package that is made specifically for BLE connection) [22]. Its methods are invoked by the system when scan events occur. Using this class, device filtering is handled. If scan detects a device called "PillDispenserESP32" it will be marked with the `isPillDispenser` flag, outlining it and putting it at the top of the list of scanned devices.

The next functions handle the connection logic:

- **connectToDevice** is a function that Initiates a GATT connection to a BLE peripheral specified by its Media Access Control (MAC) address. First, it performs permission and bluetooth availability checks, then retrieves the `BluetoothDevice` object using the provided address, then launches a `connectJob` to handle the asynchronous connection process and finally, calls `deviceToConnect.connectGatt()` with the `gattCallback` to establish the connection.

- **disconnect** is a function that disconnects from the currently connected BLE peripheral and closes the GATT client. First, it checks if `currentGatt` is not null i.e. it is connected, then it performs permission checks, afterwards it calls the command `currentGatt?.disconnect()`, however the actual disconnection and resource cleanup (closing GATT) are typically handled in the `onConnectionStateChange` callback and finally it Cancels the `connectJob` if it was related to an ongoing connection attempt.

- **writeCharacteristic** function is used to write into the writable characteristics on the device. First, it checks for connect permissions and if `currentGatt` is valid. Then, retrieves the `BluetoothGattCharacteristic` from `discoveredCharacteristics`, so that it will know what writable characteristics are on the device. Then, it verifies that the characteristic supports the specified writeType (verifies that it is writable), calls `currentGatt?.writeCharacteristic()` and finally, if the write operation has successfully been initiated, it returns **TRUE**.

- **readCharacteristic** is in nature very similar to `writeCharacteristic`, but for read-only characteristics on the device. First, it checks for connect permissions and retrieves `BluetoothGattCharacteristic` (same as previous function), then verifies the read-only property of the characteristic, Calls `currentGatt?.readCharacteristic()` and finally, returns **TRUE** if read operation was initiated successfully.

- **setCharacteristicNotificationEnabled** is a function that enables or disables notifications (or indications) for a given characteristic. Currently implemented, but not used, might be used for debugging and fine-tuning notifications.

- **enableNotificationsFor** is a private helper function specifically called during the `onServicesDiscovered` phase. This is a function that is more simple and robust than the one above. It centralizes the logic for enabling notifications for all relevant "GET_..." characteristics as soon as they are discovered, simplifying the connection setup process

The next object is a key component for the functionality of the whole BLE communication logic.

gattCallback is an instance of built-in Android class called `BluetoothGattCallback`[21] provided to the `connectGatt` method. This is the heart of managing an active BLE connection and its asynchronous events. To accomplish this task, the class has methods built in it:

- **onConnectionStateChange** is called when the connection state changes. If connected successfully, it triggers service discovery(`gatt?.discoverServices()`), on disconnection, it closes the GATT client.

- **onServicesDiscovered** is called after `discoverServices()` completes. If successful, it iterates through the services and characteristics of the connected device. It populates the `discoveredCharacteristics` list with relevant characteristics defined in `GattAttributes` (SET/GET/TRIGGER UUIDs). It also calls `enableNotificationsFor()` function for readable characteristics to automatically subscribe to updates. If the required service or critical characteristics are not found, it updates `_connectionStatus` to an error and disconnects.

- **onCharacteristicRead** is called when a characteristic read operation completes.
- **handleCharacteristicRead** If the read was successful (GATT_SUCCESS), it emits the characteristic's UUID and value to the `_characteristicUpdate` flow.
- **onCharacteristicWrite** Called after a characteristic write operation completes. Logs the success or failure. If failed, updates `_connectionStatus` with an error.
- **onCharacteristicChanged** is called when the peripheral sends a notification or indication for a characteristic to which the client has subscribed.
- **handleCharacteristicChanged** is a method that emits the characteristic's UUID and updated value to the `_characteristicUpdate` flow.
- **onDescriptorWrite** is called after a descriptor write operation completes (e.g. to enable/disable notifications). It logs the outcome.

The final function of the class is **cleanup** function. It releases all resources used by the `BluetoothLeManager`. this function is called when the model is no longer needed. it resets all states related to Bluetooth to their default state (disconnected, not scanning).

Listing 2: Top level structure of the BluetoothLeManager.kt file

```
1  package com.example.dispenser.ble
2  import . . .
3  private const val TAG = "BluetoothLeManager"
4
5  class BluetoothLeManager(private val context: Context, private val
       ↪ coroutineScope: CoroutineScope)
6  {
7    private val _characteristicUpdate = MutableSharedFlow<Pair<UUID,
       ↪ ByteArray>>()
8    val characteristicUpdate: SharedFlow<Pair<UUID, ByteArray>> =
       ↪ _characteristicUpdate.asSharedFlow()
9
10   private val discoveredCharacteristics = mutableMapOf<UUID,
       ↪ BluetoothGattCharacteristic>()
11
12   private val bluetoothManager = context.getSystemService(Context.
       ↪ BLUETOOTH_SERVICE) as BluetoothManager
13   private val bluetoothAdapter: BluetoothAdapter? =
       ↪ bluetoothManager.adapter
14   private val bleScanner by lazy { bluetoothAdapter?.
       ↪ bluetoothLeScanner }
15
16   private var scanJob: Job? = null
17   private var connectJob: Job? = null
18
19   private val _foundDevices = MutableStateFlow<List<
       ↪ UiBluetoothDevice>>(emptyList())
20   val foundDevices: StateFlow<List<UiBluetoothDevice>> =
       ↪ _foundDevices.asStateFlow()
21
22   private val _isScanning = MutableStateFlow(false)
23   val isScanning: StateFlow<Boolean> = _isScanning.asStateFlow()
24
25   private val _connectionStatus = MutableStateFlow<ConnectionStatus
       ↪ >(ConnectionStatus.Disconnected)
```

```
26      val connectionStatus: StateFlow<ConnectionStatus> =
            ↪ _connectionStatus.asStateFlow()
27
28      private val _connectedDevice = MutableStateFlow<UiBluetoothDevice
            ↪ ?>(null)
29      val connectedDevice: StateFlow<UiBluetoothDevice?> =
            ↪ _connectedDevice.asStateFlow()
30
31      private var currentGatt: BluetoothGatt? = null
32
33      // Helper functions for permission checking
34      private fun hasPermission(permission: String): Boolean
35      private fun hasRequiredScanPermissions(): Boolean
36      private fun hasRequiredConnectPermissions(): Boolean
37
38      // Scan logic
39      fun startScan()
40      fun stopScan()
41      private val leScanCallback = object : ScanCallback { ... }
42
43      // Connection logic
44      fun connectToDevice(deviceAddress: String)
45      fun disconnect()
46
47      // Characteristic operations
48      fun writeCharacteristic(characteristicUUID: UUID, value:
            ↪ ByteArray, writeType: Int = ...): Boolean
49      fun readCharacteristic(characteristicUUID: UUID): Boolean
50      fun setCharacteristicNotificationEnabled(characteristicUUID: UUID
            ↪ , enable: Boolean): Boolean
51
52      // Notification helper
53      private fun enableNotificationsFor(characteristic:
            ↪ BluetoothGattCharacteristic)
54
55      // GATT callback
56      private val gattCallback = object : BluetoothGattCallback { ... }
57
58      // Cleanup
59      fun cleanup()
60
61  }
```

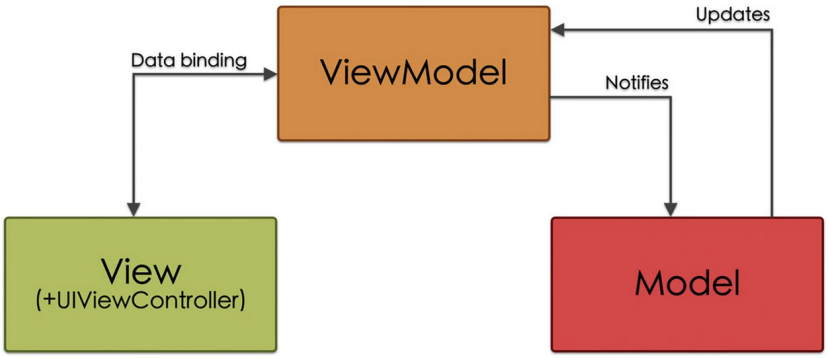

Figure 15: Model-View-ViewModel Structure. This image explains what model consists of and how different parts of model interact with each other. Credit [23]

4.2 User Interface

Having finished with describing how the App handles interface between microcontroller using BLE, the Graphical User Interface (GUI) of the app will be covered next. It consists of 3 main views (called **Device Connection, Device Configuration and Monitoring & Status**), each of them contains different information and functions. Switching between views is done via a drawer panel on the left-hand side of the app, which slides open when the button at the top left is pressed.

Once again, before moving forward, there is a certain terminology that needs to be covered. This thesis uses terminology used by official design guide[24] from **Android Developers** team, which is an industry standard.

- **Jetpack Compose** while not a technical term as the others are on the list, Jetpack Compose is the modern toolkit for building native Android User Interface (UI). It defines the paradigm under which all the other terms operate. It operates under the Model–view–viewmodel (MVVM) design pattern.
- **MVVM** is an architectural design pattern that separates an application into three (View, Model, ViewModel) components to enhance modularity, testability, and maintainability. Advantages of this pattern is that it separates logic from presentation, allowing easier testing, debugging and troubleshooting[23].
- **Model** is responsible for manipulation, access and storage of data.
- **View** is the part of the architecture that a user sees on the screen.
- **ViewModel** class is a business logic or screen level state holder. ViewModel is the core of the MVVM architecture and sits between the View and the Model. Its principal advantage is that it caches state and persists it through configuration changes. The alternative to a ViewModel is a plain class that holds the data displayed in the UI. This can become a problem when navigating between activities or Navigation destinations. Doing so destroys that data if it isn't stored using the saving instance state mechanism.
- **ViewModelFactory** is a class responsible for creating instances of ViewModels. It's needed because ViewModels often have dependencies that need to be provided to them when they are created.

- **UI State** represents what to show on the screen. For the Device Connection screen, this includes whether the app is currently scanning, the list of found Bluetooth devices, the current connection status (e.g., "Disconnected", "Connecting", "Connected"), and any error messages. This state typically lives in or is managed by the ViewModel.

- **Composable Function (or "Composable")** This is the fundamental building block of UI in Jetpack Compose. Instead of traditional XML layouts, UI is defined by writing functions annotated with @Composable. These functions describe what the UI should look like for a given state, not how to construct it step-by-step.

- **Recomposition** is the process of Jetpack Compose re-running Composable functions when their input **State** changes. If the `scanStatus` (part of the UI State) changes from "Disconnected" to "Scanning", Compose will re-run (recompose) only the parts of UI that depend on `scanStatus`, updating them automatically. UI elements don't need to be manually updated.

- **Layout Composables (Column, Row, Box)** are special Composables used to arrange other Composables on the screen. Column arranges its children vertically, Row horizontally, and Box allows stacking elements or positioning them relative to its edges.

- **Event Handling (Callbacks/Lambdas)** is based on user interaction with UI. Composables can trigger actions. This is usually done by passing lambda functions (callbacks) as parameters to the Composables.

- **StateFlow** has already been defined in the Bluetooth Connection Section. It is a state-holder observable "hot" flow that always has a value and emits updates to its collectors. This term is even more important here, as it creates a bridge for understanding how User Interface interacts with data. In context of User Interface, it is is a primary mechanism for the **ViewModel** to expose UI **State** to the Composable. Composable functions can observe (or "collect") these StateFlows, When the StateFlow emits a new value, collectAsState() receives this update and signals to the Compose runtime that the state read by a Composable has changed.

4.2.1 MainActivity.kt

The Main entry file to the whole app is called **MainActivity.kt**. The fundamental part of the UI is connected through this file. It is worth reviewing it here before delving deeper into the each view. The top level structure of the code can be seen in code block here 3.

Main application structure is defined in the **AppScaffoldWithDrawer** composable function. It contains top app bar, a navigation drawer, and the main content area where different views are displayed. This function is called within the `MainActivity` class's `onCreate` method to set up the UI. `AppScaffoldWithDrawer` has numerous tasks:

1. **Navigation:** Manages the ModalNavigationDrawer to allows switching between different sections of the app ("Device Connection", "Device Configuration", "Monitoring & Status").

2. **Screen Switching:** Uses a `when` (similar to `case` and `switch` commands in C/C++) statement to display the appropriate screen in the Scaffold's content area.

3. **ViewModel Access:** It receives the `devConnViewModelFactory` and uses it to obtain an instance of `DevConnViewModel`. This `ViewModel` is then passed down to the relevant screens.

4. **Conditional UI:** It conditionally renders content within the "Device Connection" screen path(e.g. Popups that prompt granting permissions and enabling Bluetooth).

5. **Callbacks:** Passes down callbacks to `MissingPermissionsScreen` and `BluetoothDisabledScreen`.

Besides `AppScaffoldWithDrawer` function, there are also various supplementary functions:

- **MissingPermissionsScreen** is a simple UI screen displayed when the necessary Bluetooth permissions have not been granted by the user.It provides a "Grant Permissions" button that triggers the `onRequestPermissions` callback (which is implemented in `MainActivity`) to launch the permission request.

- **BluetoothDisabledScreen** is similar to `MissingPermissionsScreen`, but for when Bluetooth is disabled. Also provides an "Enable Bluetooth" button to switch it on.

- **DeviceConfigurationScreen** This is a legacy device configuration view, left from older iterations of the code. Currently not used, still left because it might be used for debugging. Device configuration view is currently handled in a separate file `DevConfScreen.kt`

- **MonitoringStatusScreen** Same as above, legacy view for Device Monitoring & Status screen, currently not used. This View is currently handled by `DevMonitroingScreen.kt` file.

- **AppScaffoldWithDrawerPreview** is an assistance function for Prototyping. Android Studio has built-in app preview functionality, this helper function allows seeing how `AppScaffoldWithDrawer` looks without running the entire app on an emulator or device.

Next, `MainActivity` class ought to be reviewed next. It is a primary `Activity` of the application, handles lifecycle, Compose UI and manages permissions and Bluetooth initialization. It has 3 parameters:

- **private val applicationScope** is a `CoroutineScope` that lives as long as `MainActivity` is explicitly active. It consists of 2 components: `SupervisorJob` ensures that if one child coroutine fails, others are not cancelled. `Dispatchers.Default` is used for CPU-intensive work off the main thread.

- **private lateinit var bluetoothLeManager** holds the instance of `BluetoothLeManager`, the class that was extensively covered in the Bluetooth Connection Section.

- **private val blePermissions** is used to define an array of Bluetooth-related permissions required by the app, adapting to different Android versions (API 31/Android S and above have different permission requirements). Necessary for compatibility with older devices that have a different set of permissions.

Then, it has 4 different methods, each serving different purpose:

- **onCreate** is a Lifecycle method, which is called when activity is starting. It fulfills many important tasks: Initializes the `BluetoothLeManager`. Using `setContent { ... }` it creates entry point for Jetpack Compose, which defines the UI content. Besides that, it also sets the stage for the function of the app as a whole: it establishes **Permission Handling, Bluetooth Enabled Checking, Initial Permission/Bluetooth Requests, ViewModel Factory Creation** and finally **Main UI Setup**.
- **checkAllPermissionsGranted** is a private helper method that serves as a checklist to go through and check if all permissions (defined in `blePermissions`) have been granted. It initializes `hasPermissions` on creation from method above and can be called again to re-check permissions.
- **isBluetoothAdapterEnabled** is a private helper method that checks if Bluetooth adapter is currently enabled. It initializes `isBluetoothEnabled` in `OnCreate` and by `enableBluetoothLauncher` to check if Bluetooth is enabled.
- **onDestroy** is a lifecycle method for termination of the task that is used to release resources. It works together with the `bluetoothLeManager.cleanup()` method from the BluetoothLeManager class to clean up the resources.

Listing 3: Top-level structure of the `MainActivity.kt` file

```
1  package com.example.dispenser
2  import . . .
3
4  class MainActivity : ComponentActivity() {
5
6    private val applicationScope = CoroutineScope(/* SupervisorJob()
       ↪ + Dispatchers.Default */)
7    private lateinit var bluetoothLeManager: BluetoothLeManager   //
       ↪ Instantiate early
8
9    // List of permissions
10   private val blePermissions =
11   if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.S) { ... } else
       ↪ {
12     arrayOf(
13     Manifest.permission.BLUETOOTH,
14     Manifest.permission.BLUETOOTH_ADMIN,
15     Manifest.permission.ACCESS_FINE_LOCATION   // or coarse
16     )
17   }
18
19   // Lifecycle
20   override fun onCreate(savedInstanceState: Bundle?) { ... }
21   private fun checkAllPermissionsGranted(): Boolean
22   private fun isBluetoothAdapterEnabled(): Boolean
23   override fun onDestroy() { ... }
24 }
25
26 /* ---------- Composables ---------- */
27
```

```
28  @OptIn(ExperimentalMaterial3Api::class)
29  @Composable
30  fun AppScaffoldWithDrawer(
31  modifier: Modifier = Modifier,
32  permissionsGranted: Boolean,
33  isBluetoothEnabled: Boolean,
34  onRequestPermissions: () -> Unit,
35  onRequestEnableBluetooth: () -> Unit,
36  devConnViewModelFactory: DevConnViewModelFactory
37  ) { ... }
38
39  @Composable
40  fun MissingPermissionsScreen(onRequestPermissions: () -> Unit) {
       ↪ ... }
41
42  @Composable
43  fun BluetoothDisabledScreen(onRequestEnableBluetooth: () -> Unit) {
       ↪  ... }
44
45  // Place-holder composables for future screens
46  @Composable
47  fun DeviceConfigurationScreen() { ... }
48
49  @Composable
50  fun MonitoringStatusScreen() { ... }
51
52  /* ---------- Preview ---------- */
53  @Preview(showBackground = true)
54  @Composable
55  fun AppScaffoldWithDrawerPreview() { ... }
```

4.2.2 DevConnViewModel.kt

DevConnViewModel.kt is responsible for State management. Following the MVVM pattern it is the ViewModel part of the app. It contains:

- **DevConnViewModel** class is a ViewModel designed to manage the UI-related data for the device connection screen. This ViewModel is used by all views that interact with the connected BLE device. It inherits from Android built-in `AndroidViewModel`, which provides it with an Application context, useful for system services or application-wide resources.
- **scanStatus** is a StateFlow exposes the current status of the BLE scanning process to the UI.
- **foundDevices** is a StateFlow that exposes the current status of the connection to a BLE device.
- **connectingDevice** is a Stateflow that covers the specific device that is currently in the process of connecting or is already connected. It is nullable, therefore it can also be `null` if no device is connected.
- **_lastError** is a private, mutable state holder for any general error messages that the ViewModel itself might generate. Nullable, and is `null` when initialized.

- **displayError** is a StateFlow that Provides a single, consolidated error message for the UI.
- **startScan(), stopScan(), connectToDevice(device: UiBluetoothDevice) and disconnect()** are functions managing interaction with the deivce. They use corresponding methods from the `BluetoothLeManager` class.
- **setGeneralError** is a helper function to allow setting a custom error message to **_lastError**

The next StateFlows within the `DevConnViewModel` class are related to interaction with Timer/Counter of the device, which plays a pivotal role in its configuration and interaction in general

- **_deviceTime** is a StateFlow that holds the current time of the connected device, as a string. **deviceTime** is a similar non-mutable StateFlow to display the time in UI.
- **_dispenseSchedule** is a StateFlow that contains the current dispense schedule of the connected dispenser device, as a string. **dispenseSchedule** is an immutable variant of this.
- **_lastDispenseInfo** Holds information about the last dispense event from the dispenser. **lastDispenseInfo** is an immutable variant.
- **_timeUntilNextDispense** Holds the calculated or received time until the next scheduled dispense. **timeUntilNextDispense** is an immutable variant.
- **_dispenseLog** Holds a log of dispense events from the dispenser. **dispenseLog** is an immutable variant.

The next code block **init {. . .}** is executed when an instance of `DevConnViewModel` is created. It is used to set up long-lived operations or observers. It launches a coroutine in `viewModelScope` to collect connection data from `bluetoothLeManager.connectionStatus`. If the status is not Connected, it resets all the device-specific data (`_deviceTime, _dispenseSchedule,` etc.) to their "N/A (Disconnected)" states.

After this block, functions related to sending data from the App to the device are defined:

- **triggerManualDispense** sends a command to the connected BLE device to perform a manual dispense.
- **setDeviceTime** sends a command to the connected BLE device to set its internal time.
- **setDispenseSchedule** sends a command to the connected BLE device to set its dispense schedule.
- **requestDeviceTime** sends a request to the connected BLE device to read its current time characteristic.
- **requestDispenseSchedule** sends a request to read the dispense schedule characteristic.

- **requestLastDispenseInfo** sends a request to read the last dispense info characteristic.
- **onCleared** This method is called when the ViewModel is no longer used and is about to be destroyed. It's the place to clean up any resources tied to the ViewModel's lifecycle.

Listing 4: Top-level structure of the `DevConnViewModel.kt` file

```
1  package com.example.dispenser.screens.devconnection
2
3  import ...
4
5  class DevConnViewModel(
6  application: Application,
7  private val bluetoothLeManager: BluetoothLeManager
8  ) : AndroidViewModel(application) {
9
10   // State for scan status (now reflects manager's scanning state)
11   val scanStatus: StateFlow<ConnectionStatus> = bluetoothLeManager.
        ↪ isScanning
12   .map { isScanning ->
13     if (isScanning) ConnectionStatus.Scanning else ConnectionStatus
         ↪ .Disconnected
14   }
15   .stateIn(viewModelScope, SharingStarted.Lazily, ConnectionStatus.
        ↪ Disconnected)
16
17   // State for found devices (directly from manager)
18   val foundDevices: StateFlow<List<UiBluetoothDevice>> =
        ↪ bluetoothLeManager.foundDevices
19
20   // State for device connection status (directly from manager)
21   val connectionStatus: StateFlow<ConnectionStatus> =
        ↪ bluetoothLeManager.connectionStatus
22
23   // State for the device currently being connected to/connected (
        ↪ directly from manager)
24   val connectingDevice: StateFlow<UiBluetoothDevice?> =
        ↪ bluetoothLeManager.connectedDevice
25
26   // State for any last error message (can be a combination or
        ↪ specific errors)
27   // For now, let's use the connectionStatus error or a general one
        ↪ .
28   private val _lastError = MutableStateFlow<String?>(null) // For
        ↪ other types of errors if needed
29   val lastError: StateFlow<String?> = _lastError.asStateFlow()
30
31   // Combine scan errors and connection errors into a displayable
        ↪ error
32   val displayError: StateFlow<String?> = combine(
33   bluetoothLeManager.connectionStatus, // For connection/scan
        ↪ errors from manager
34   _lastError // For other general errors set by ViewModel
35   ) { connStatus, generalError ->
36     if (connStatus is ConnectionStatus.Error) {
37       connStatus.message
```

```
38          } else {
39              generalError
40          }
41      }.stateIn(viewModelScope, SharingStarted.Eagerly, null)
42
43      fun startScan() { ... }
44      fun stopScan() { ... }
45      fun connectToDevice(device: UiBluetoothDevice) { ... }
46      fun disconnect() { ... }
47      fun setGeneralError(message: String?) { ... }
48
49      private val _deviceTime = MutableStateFlow<String?>("N/A")
50      val deviceTime: StateFlow<String?> = _deviceTime.asStateFlow()
51
52      private val _dispenseSchedule = MutableStateFlow<String?>("N/A")
53      val dispenseSchedule: StateFlow<String?> = _dispenseSchedule.
            ↪ asStateFlow()
54
55      private val _lastDispenseInfo = MutableStateFlow<String?>("N/A")
56      val lastDispenseInfo: StateFlow<String?> = _lastDispenseInfo.
            ↪ asStateFlow()
57
58      private val _timeUntilNextDispense = MutableStateFlow<String?>("N
            ↪ /A")
59      val timeUntilNextDispense: StateFlow<String?> =
            ↪ _timeUntilNextDispense.asStateFlow()
60
61      private val _dispenseLog = MutableStateFlow<String?>("N/A")
62      val dispenseLog: StateFlow<String?> = _dispenseLog.asStateFlow()
63      init { ... }
64      fun triggerManualDispense() { ... }
65      fun setDeviceTime(timeString: String) { ... }
66      fun setDispenseSchedule(scheduleString: String) { ... }
67      fun requestDeviceTime() { ... }
68      fun requestDispenseSchedule() { ... }
69      fun requestLastDispenseInfo() { ... }
70      override fun onCleared() { ... }
71  }
```

DevConnViewModelFactory.kt file contains contains a single *DevConnViewModelFactory* class. It is a custom factory responsible for creating instances of `DevConnViewModel` This is necessary because `DevConnViewModel` requires arguments in its constructor (`application` and `bluetoothLeManager`), which the default ViewModel provider mechanism cannot supply.

4.2.3 Device Connection

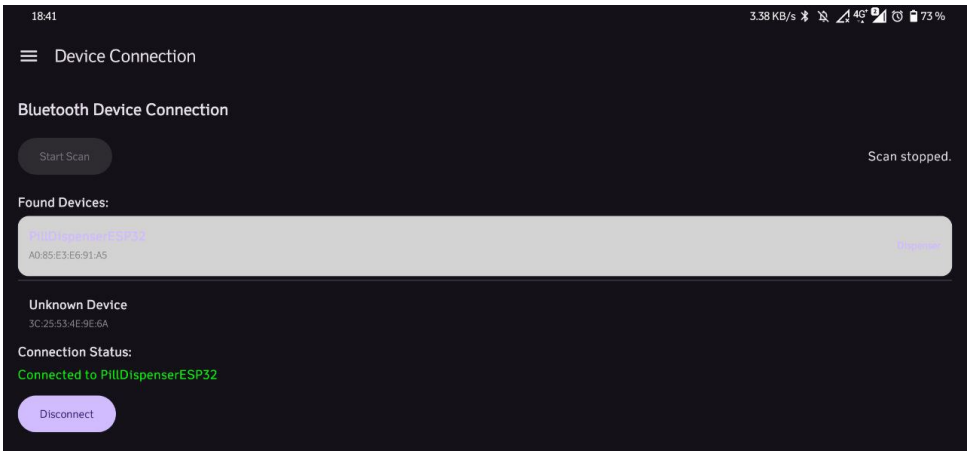

Figure 16: Device connection view.

The structure behind **Device Connection** view consist of multiple files that contribute to this view:

1. **DevConnScreen.kt** file contains the Composable function `DevConnScreen`, which defines the visual layout and user interface elements for the device connection screen. This file represents the **View** layer of MVVM pattern.

2. **DevConnViewModel.kt** is already covered before. This file contains State management logic. It acts as a bridge between the UI (`DevConnScreen`) and the data layer (`BluetoothLeManager`). It holds the state displayed by the UI and processes user actions. This file represents the **ViewModel** layer of MVVM pattern.

3. **BluetoothLeManager.kt** Covered in previous section, is a class responsible for handling BLE interactions between Device, Android Bluetooth Application Programming Interface (API) and GUI. Since it manages logic and background functionality, it is part of **Model** layer of MVVM pattern.

4. **MainActivity.kt** is the main entry point of this app. It sets up the overall UI structure, including the navigation drawer and the content area where all the screens are displayed.

5. **DevConnViewModelFactory.kt** is a standard ViewModel factory responsible for creating instances of `DevConnViewModel`

6. **GattAttributes.kt** Defines the unique UUIDs for BLE service and characteristics on the ESP32. Already covered in the Bluetooth Connection Section.

Many of these files (**BluetoothLeManager.kt**, **MainActivity.kt** and **GattAttributes.kt**) have already been previously covered and 3 files that we will cover next, to understand how the view is created. **DevConnScreen.kt** file is responsible for layout and UI elements. Its top level structure can be seen in code block 4 It contains:

- **UiBluetoothDevice** data class serves as a simplified representation of a Bluetooth device, made to be displayed in the UI. It's more convenient to use this

in Composable functions than the raw Android `BluetoothDevice` object directly. Instances of this class are used to populate the list of found devices displayed on the screen(MAC address, Device name and whether it is a Pill Dispenser).

- **ConnectionStatus** class defines a set of distinct states representing the current status of the Bluetooth scanning and connection process. It is a **sealed** class, which means that all direct subclasses of a sealed class must be declared in the same file as the sealed class itself. Here it is used for a state-switch (using an already known command `when`). It has 4 state objects: `Disconnected, Scanning, Connecting, Connected`, each representing current relationship of an App to the device.

- **DevConnScreen** is a primary composable function that defines the entire layout and behavior of the **Device Connection** screen. It is quite vast function in a sense that through it the whole view is displayed, therefore it is worth documenting it deeper. `DevConnScreen` takes many input parameters:

  - **scanStatus** is the current status of the scanning process
  - **foundDevices** is list of discovered Bluetooth devices.
  - **connectionStatus** is current status of the device connection.
  - **connectingDevice** is the device that is currently being connected to, or is already connected.
  - **lastError** general error message that might not be part of either `scanStatus` or `connectionStatus`
  - **onStartScanClick** is a callback function invoked when the "Start Scan" button is clicked.
  - **onStopScanClick** is a callback function invoked when the "Stop Scan" button is clicked.
  - **onDeviceClick** is a callback function invoked when a device in the list is clicked.
  - **onDisconnectClick**is a callback function invoked when the "Disconnect" button is clicked.

  In essence, this function is responsible for every element of the UI seen within the "Device Connection Screen" window.

- **DeviceListItem** is a helper Composable function responsible for rendering a single item in the list of found Bluetooth devices. In essence, this function is responsible for the contents of a single card from the list of scanned devices. It is called each time a new device is found to draw a card with information about this device within it.

- **DevConnScreenPreview_*** are multiple functions with similar names and purpose. Each of them serves the debugging previewing, allowing checking different states of this window within Android Studio without having to compile and emulate the whole app.

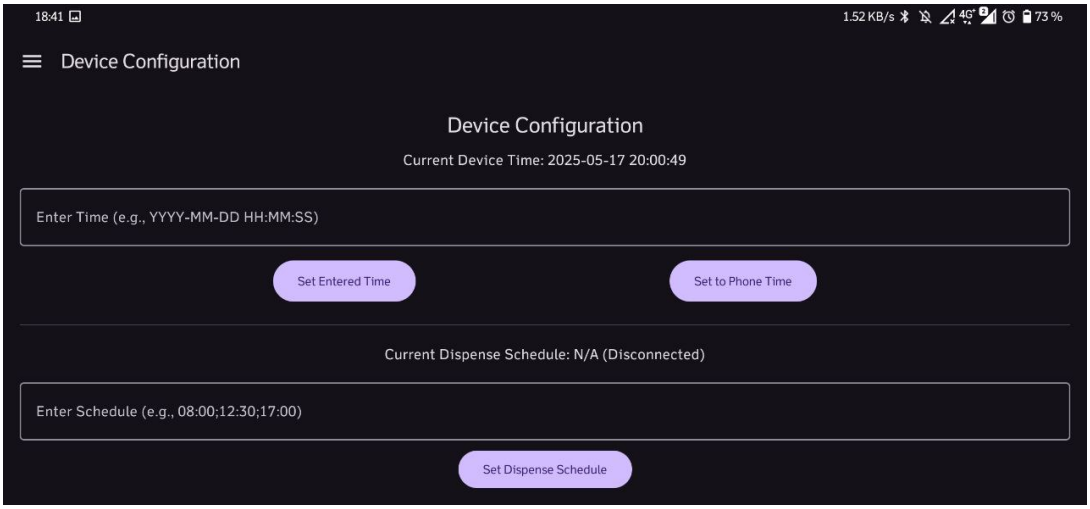

Figure 17: Layout of the device configuration view file

4.2.4 Device Configuration

The Device Configuration screen has the following functionality, as also seen on image 17:

- **View Current Device Time**: Displays the time currently set on the ESP32.
- **Set Device Time**: Manually enter a time string or set the ESP32's time to the phone's current time.
- **View Current Dispense Schedule:** Displays the dispense schedule set on the microcontroller.
- **Set Dispense Schedule**: Enter a new dispense schedule string.
- **Trigger Manual Dispense**: Command the ESP32 to perform a single dispense operation.

All these interactions need an activle Bluetooth connection to the device.

The files contributing to this view are **DevConfScreen.kt, DevConnViewModel.kt, BluetoothLeManager.kt, GattAttributes.kt, MainActivity.kt**. They serve the same purpose for this view as for the previous one. What differs the two views is the controller file, in this case **DevConfScreen.kt**. It is the file that defines the structure and functionality of this screen. Unlike **DevConnScreen** file, this one is much smaller, because **DevConnScreen** also serves as an initialiser for services, permissions etc. which is not present here. Essentially all the functionality is relegated to other files, classes and functions. This file only binds them together and lays down the structure of the UI. as seen in block 5, This file contains a sole function called `DeviceConfigurationScreen` which takes `DevConnViewModel` ViewModel as an input parameter to fetch necessary data from the device.

Listing 5: Top-level structure of the `DeviceConfigurationScreen.kt` file

```
1  package com.example.dispenser.screens
2
3  import ...
4
5
6  @OptIn(ExperimentalMaterial3Api::class)
7  @Composable
8  fun DeviceConfigurationScreen(devConnViewModel: DevConnViewModel) {
9    val connectionStatus by devConnViewModel.connectionStatus.
       ↪ collectAsState()
10   val isConnected = connectionStatus == ConnectionStatus.Connected
11
12   var timeToSetText by remember { mutableStateOf("") }
13   var scheduleToSetText by remember { mutableStateOf("") }
14
15   val currentDeviceTime by devConnViewModel.deviceTime.
       ↪ collectAsState()
16   val currentSchedule by devConnViewModel.dispenseSchedule.
       ↪ collectAsState()
17
18   LaunchedEffect(isConnected) {
19     if (isConnected) {
20       devConnViewModel.requestDeviceTime()
21       devConnViewModel.requestDispenseSchedule()
22     }
23   }
24
25   Column(
26   modifier = Modifier
27   .fillMaxSize()
28   .padding(16.dp),
29   horizontalAlignment = Alignment.CenterHorizontally,
30   verticalArrangement = Arrangement.spacedBy(10.dp)
31   ) { \UI Components here\  }
32   }
33 }
```

4.2.5 Monitoring & Status

a. Device Status Display: i. Show basic status received from the dispenser: 1. Connected/Disconnected 2. Battery Level (ESP32 Function) 3. Last dispensed time/status (Success/Failure) (Requires Sensor) (Sensor+T/C info) 4. Current time on the dispenser (to check sync) T/C Info 5. Time until next dispense T/C Info

b. Dispense History/Log: i. Display a log of past dispense events. (Need to configure Logging storage on ESP32, last 30 due to small memory size) ii. Information per entry: Timestamp, Scheduled Time, Medication Name (if set), Status (e.g., Dispensed Successfully, Failed, Skipped). This is crucial for tracking adherence.

5 Microcontroller Programming

Before going into the details of how the backend part of programming the device is structured, an overview of the general plan for the backend development is worth looking at. Before starting the implementation, the plan was drafted that would outline important functions that would be needed for the app to communicate properly with the device.

Writable

- `Set_Device_Time` — synchronize the ESP32 clock.
- `Set_Dispense_Schedule` — add, edit, or delete schedule entries.
- `Trigger_Manual_Dispense` — manually trigger a dispense (logged for traceability).

Readable

- `Get_Device_Time` — read the current device time.
- `Get_Dispense_Schedule` — retrieve the stored schedule.
- `Get_Last_Dispense_Info` — timestamp + status of the last event.
- `Get_Time_Until_Next_Dispense` — countdown to the next dose.
- `Get_Dispense_Log` — full dispense history.

Notify/Indicate

- `Notify_Dispense_Event` — real-time success/failure updates with timestamps.
- `Notify_Schedule_Change_Confirmation` (optional) — acknowledgement of schedule edits.

5.1 Important Functions

5.2 Backend-Frontend Integration

6 Results Analysis and Discussion

This section provides an analysis of the project's outcomes and a critical discussion of the development process. The project involved a wide range of tasks spanning multiple technical domains. While valuable experience was gained and certain components were progressed, such as 3D printed body, an existing prototype of an Android app and some backend code, the overall integration and functionality targeted by the initial objectives were not fully achieved. Consequently, the final result is evaluated as not meeting the predefined success criteria. The success criterion was this: A fully functioning Pill Dispenser with a remote app to remotely control it. This discussion will explore the contributing factors, focusing on challenges encountered in project management and technical execution, reflect on the lessons learned, and propose directions for future work. Contextualizing these challenges is crucial for understanding the project's trajectory and final state.

6.1 Challenges Encountered

The difficulties faced during the project can be broadly categorized into two interconnected areas: **Project Management and Execution**, and **Technical Complexity and Resource Constraints**.

6.1.1 Project Management and Execution

Effective project management proved to be a significant challenge, stemming primarily from a lack of prior experience in managing long-term, multi-faceted technical projects.

1. **Planning and Structuring** Early momentum was shaped by *over-confidence that past short, ad-hoc jobs had prepared the author for end-to-end project ownership.* . An initial underestimation of the need for rigorous, upfront planning led to difficulties. Specifically, the project lacked a sufficiently detailed breakdown of the overall task into manageable sub-components with clear milestones and dependencies. This absence of a granular structure hindered effective progress tracking and adaptation throughout the development cycle.

2. **Resource Availability** Ambiguity regarding the scope of permissible resources (e.g., access to university facilities, specific software licenses, or hardware) and their integration into the project plan was an early challenge. This lack of explicit definition led to a predominantly Do-It-Yourself approach to problem-solving, potentially precluding the use of more optimized or readily available solutions typical in industrial or well-resourced academic environments. This highlighted the necessity for proactive clarification of all available resources at the project's outset.

3. **Development Methodology and Feedback Loop**: An Agile-inspired iterative approach was initially assumed, anticipating regular feedback to guide development. However, the conditions **were not clearly defined** and there was some confusion about what role each participant (including me) would play. There was an absence of a clearly defined counterpart or stakeholder role dedicated to providing consistent, actionable feedback. This ambiguity made it challenging to effectively solicit, interpret, and integrate feedback, occasionally impeding development velocity. Furthermore, the lack of clearly delineated responsibilities within the project

(as it was executed solely by the author) made whatever feedback was given feel less integrated into a structured workflow.

4. **Task Management and Cognitive Load**: Executing all project tasks single-handedly – from conceptualization and design to implementation and testing across different domains – resulted in significant challenges related to **divided attention**. Constantly shifting between diverse tasks (e.g., 3D modeling, embedded programming, mobile development) proved cognitively demanding and potentially reduced efficiency and focus in each area. This experience highlighted a key learning: the critical importance of comprehensive documentation. Well-structured documentation serves as an essential tool for defining components, standardizing terminology, and maintaining a clear overview, thereby facilitating better intrapersonal communication (tracking one's own work across domains) and enabling potential future collaboration.

6.1.2 Technical Complexity and Resource Constraints

The project's ambitious scope required integrating expertise from several distinct technical fields, which posed considerable problems for a single developer.

1. **Scope and Expertise Mismatch**: The project necessitated skills in 3D modeling and printing, Android application development, and microcontroller programming with Bluetooth Low Energy (BLE) communication. While 3D design and printing were navigated successfully, providing valuable practical insights, they consumed a significant portion of the available time.

2. **Software Development Learning Curve**: The software development aspects presented steep learning curves:

   (a) Android Development: Required learning not only Kotlin syntax but also the OOP paradigm as applied to GUI development, Android-specific API, the application lifecycle, build toolchains (e.g., Gradle), and handling platform diversity (API levels, device compatibility). This differs substantially from procedural or scripting languages like C, Python, or MATLAB, particularly concerning memory management, data type handling, and runtime environments.

   (b) Microcontroller Programming & BLE: Required dedicated research into embedded C/C++, microcontroller architecture (referencing datasheets), and the complexities of the BLE protocol stack and its implementation on the target hardware. While manufacturer documentation and libraries provide support, integrating these components effectively remained time-intensive.

   (c) Effort Underestimation: The cumulative effort required to gain proficiency and implement solutions across these diverse domains was initially underestimated. The need to simultaneously learn and apply knowledge in unfamiliar areas significantly impacted the development timeline and the ability to achieve the desired level of integration and polish for the final system. Tackling these complex, disparate technical challenges single-handedly proved to be a major constraint.

6.2 Reflection and Lessons Learned

Despite the project not fully meeting its objectives, the process yielded significant learning outcomes. The development itself delivered a big amount of various domain knowledge in areas of 3D-Design and Printing, Android and Microcontroller development and Project Management:

1. **3D-Design and Printing**: Before starting the project, my experience designing a functional device was limited. Moreover, the designed components would then have to be 3D-printed, which meant that material properties of the filament would have to be taken into account, leading to research and adjustment of the design to facilitate the newly obtained knowledge.

2. **Android Development**: Through this project a lot of insights has been gained on how the Android development environment works. Since I had no experience in programming any User interface harder than basic Command-Line interface, I had to learn how to do it and also using OOP for that. Android platform has also added complexity, since one would need to learn how Application Packaging, Permissions, Device Compatibility and many other components interact together in an Android Application.

3. **Microcontroller Development** I have already had experience developing on an another microcontroller, which was also rather lower-level. Arduino and Platform.io development environments have made it easier to write a backend, however writing a BLE was new for me.

4. **Project Management** This project has had a profound impact on my view of the role and importance of Project Management. It is important to not also divide the tasks in a project-stucture plan (which has-been done), but also to clarify the participants, their roles and tasks. Since this hasn't been done initially it lead to difficulties estimating necessary amount of effort to complete the project successfully. It is also important to know in advance (and have it written down somewhere) which resources are made available for the project. There has been no requirements on budget laid down. Generalizing, the high amount of requirements is good because it reduces amount of creative freedom. There is a balancing act at play where if the requirements are too strict, there is only one solution or none at all, while on the other hand, if the requirements are too lax, the creative freedom is too wide which leads to kind of analysis paralysis, in which the developer would have to either lay down requirements themselves (effectively delegating the task of creating a Technical Task onto the developer) or operate on assumptions (which would then require a feedback from the client, whether these assumptions are adequate).

6.3 Next steps

A lot of work remains to be done. From the side of components it might be worth looking into a stronger stepper motor, as the current one has very narrow margin of error and its performance is highly dependent on battery level. What this means is that when the battery level is low, it might not be able to pull the load of 2 mills and extra weight. The construction of the device itself is mostly fine, although the chamber wheels are a bit loose to reduce amount of contact area and friction. If there is a change in components,

it would most certainly also require redisigning the lower deck to fit the new components, as the more powerful stepper motor would also require different (perhaps also bigger) driver. Replacing the motor would also require better power supply, as a single 18650 battery might not be enough to power it.

Another area of improvement would be the addition of sensors. Adding sensors would allow more precise control of the chambers (e.g. using hall sensors + magnet on each wing to check whether chambers are aligned), as well as tracking whether the pills were dispensed or disposed using an accelerometer at the bottom of dispense chamber. The reason it hasn't been initially done is the lack of documented requirements. On one hand, the project is already quite extensive as it is, on another hand, implementing any of the above mentioned changes would require allocating resources that have already been critically lacking. Doing this would mean revamping a substantial part of the device construction, therefore it is better left for the next iteration of development.

There is also a big area for improvement in the Android App. Due to limited experience and knowledge about current state-of-the-art frontend development paradigm, its usability could further be improved. For this, information like user journey(what functions does user use the most and how long does it take them to get there) need to be collected. Certain functions could also be implemented the other way. For example, bluetooth connection could be done through Android settings and device pairing, removing the need for the device connection screen completely.

References

[1] C. Haase. "Google i/o 2019: Empowering developers to build the best experiences on android + play." Blog post, Accessed: May 16, 2025. [Online]. Available: https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html.

[2] LiveFine, *Automatic Pill Dispenser with Clear Lid and Large Display*, https://www.livefineproduct.com/products/automatic-pill-dispenser-with-clear-lid-and-large-display, Accessed: 2025-04-21, 2024. [Online]. Available: https://www.livefineproduct.com/products/automatic-pill-dispenser-with-clear-lid-and-large-display.

[3] Zoksi. "Zoksi large pill organizer daily pill box for 7 or 14 days wall mountable pill dispenser hold for medicine vitamin or supplements weekly pill case with easy press button blue." Accessed on April 22, 2025, Accessed: Apr. 22, 2025. [Online]. Available: https://zoksi.shop/products/zoksi-large-pill-organizer-daily-pill-box-for-7-or-14-days-wall-mountable-pill-dispenser-hold-for-medicine-vitamin-or-supplements-weekly-pill-case-with-easy-press-button-blue.

[4] E. Design, *Spur gear generator*, Accessed: 2025-05-02, 2021. [Online]. Available: https://evolventdesign.com/pages/spur-gear-generator.

[5] Kiatronics, *28byj–48 – 5 v 4-phase stepper motor datasheet*, Accessed 2025-05-02, Welten Holdings Ltd., Tauranga, New Zealand. [Online]. Available: https://www.digikey.com/htmldatasheets/production/1839399/0/0/1/28byj-48.pdf.

[6] Texas Instruments Incorporated, "Uln200x, ulq200x high-voltage, high-current darlington transistor arrays," Texas Instruments Incorporated, Dallas, TX, USA, Datasheet SLRS027T, 2025, Revision T, accessed 2025-05-02. [Online]. Available: https://www.ti.com/document-viewer/uln2003a/datasheet.

[7] Espressif Systems, *Esp32-s3 development kits documentation, User guide v1.1*, Accessed: 2025-05-02, Espressif Systems, 2024. [Online]. Available: https://docs.espressif.com/projects/esp-dev-kits/en/latest/esp32s3/esp-dev-kits-en-master-esp32s3.pdf.

[8] Panasonic Corporation, "Ncr18650b lithium-ion rechargeable battery cell," Panasonic Industrial Devices, Datasheet NCR18650B, Accessed 2025-05-02. [Online]. Available: https://baterie18650.pl/wp-content/uploads/2023/12/panasonic_ncr18650b.pdf.

[9] Prusa Research. "Original Prusa MK4 S 3D Printer." Accessed 4 May 2025, Accessed: May 4, 2025. [Online]. Available: https://www.prusa3d.com/product/original-prusa-mk4-2/.

[10] S. A. Raj, E. Muthukumaran, and K. Jayakrishna, "A case study of 3d printed pla and its mechanical properties," *Materials Today: Proceedings*, vol. 5, no. 5, pp. 11 219–11 226, 2018.

[11] A. Manoj, M. Bhuyan, S. R. Banik, and M. R. Sankar, "Review on particle emissions during fused deposition modeling of acrylonitrile butadiene styrene and polylactic acid polymers," *Materials Today: Proceedings*, vol. 44, pp. 1375–1383, 2021.

[12] T. Tábi, A. Wacha, and S. Hajba, "Effect of d-lactide content of annealed poly (lactic acid) on its thermal, mechanical, heat deflection temperature, and creep properties," *Journal of Applied Polymer Science*, vol. 136, no. 8, p. 47 103, 2019.

[13] R. Conn et al., "Safety assessment of polylactide (pla) for use as a food-contact polymer," *Food and Chemical Toxicology*, vol. 33, no. 4, pp. 273–283, 1995.

[14] JetBrains s.r.o. "Basic syntax – variables." Accessed: May 22, 2025. [Online]. Available: https://kotlinlang.org/docs/basic-syntax.html#variables.

[15] JetBrains s.r.o. "Collections overview – collection types." Accessed: May 22, 2025. [Online]. Available: https://kotlinlang.org/docs/collections-overview.html#collection-types.

[16] JetBrains s.r.o. "Null safety – kotlin documentation." Accessed: May 22, 2025. [Online]. Available: https://kotlinlang.org/docs/null-safety.html.

[17] JetBrains s.r.o. "Coroutines, Language support for asynchronous, non-blocking programming." Accessed: 22 May 2025, Accessed: May 22, 2025. [Online]. Available: https://kotlinlang.org/docs/coroutines-overview.html.

[18] JetBrains s.r.o. "Asynchronous flow, Cold streams that emit multiple values asynchronously." Accessed: 22 May 2025, Accessed: May 22, 2025. [Online]. Available: https://kotlinlang.org/docs/flow.html.

[19] JetBrains s.r.o. "Stateflow — kotlinx.coroutines," Accessed: May 23, 2025. [Online]. Available: https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-state-flow/.

[20] JetBrains s.r.o. "Scope functions." Last modified 25 Sept. 2024, Accessed: May 23, 2025. [Online]. Available: https://kotlinlang.org/docs/scope-functions.html.

[21] Android Developers. "Bluetoothgatt — android api reference," Google LLC, Accessed: May 23, 2025. [Online]. Available: https://developer.android.com/reference/kotlin/android/bluetooth/package-summary.

[22] Android Developers. "Android.bluetooth.le — android api reference," Google LLC, Accessed: May 23, 2025. [Online]. Available: https://developer.android.com/reference/kotlin/android/bluetooth/le/package-summary.

[23] R. F. García, "Mvvm: Model–view–viewmodel," in *iOS Architecture Patterns: MVC, MVP, MVVM, VIPER, and VIP in Swift*. Berkeley, CA: Apress, 2023, pp. 145–224, ISBN: 978-1-4842-9069-9. DOI: 10.1007/978-1-4842-9069-9_4. [Online]. Available: https://doi.org/10.1007/978-1-4842-9069-9_4.

[24] Android Developers. "Ui layer — app architecture," Google LLC, Accessed: May 23, 2025. [Online]. Available: https://developer.android.com/topic/architecture/ui-layer/.