## Updated Specification:

A medicine can either be prescribed or not. If the medicine is prescribed then each prescribed medicine should have the times to take the medication (or a number of times per day) when they should take the medicine, amount, type (pill, liquid, etc.), and any extra instructions or conditions. If the medicine is not prescribed, then it should just store the number of times you took it that day, how much you took, the type, and the time you took it. The system should track the time in order to remind users to take their medication, in a visual and auditory way.

Each user should be able to create an account, with a username and password. Users should be able to add, edit and remove prescriptions and medicine from their account. The user should be able to see when to take their medication, as a list or a calendar. Users should also be able to enter their meal times and sleep schedules so that they see if the times they chose to take the medicine work well. A user-friendly interface should be considered.

When a user looks at the schedule they have made, they should be able to see when they plan to take the medicine, what they are taking that day, how much to take, and any additional instructions. They should be able to edit their schedule as they see fit. They should be able to look at their schedule for the whole week.

Finally, the user should be able to logout and the program will save the user's username and password and all of the associated information. However, if the user wishes to login to their account, they should re-run the program and type login because the program will end when you log out but the information will still be there. The user just needs to rerun the program and they should be able to login using their login credentials.

**Highlights:**

- We added the functionality of being able to edit and remove medicines from the user's account. We also added the attribute type, to enhance the medicine description. Type, in this case, is the unit of measurement of the medicine.
- We added the functionality of being able to add, edit and remove prescriptions.
- Implemented the functionality of entering meal times and sleep schedules.
- We removed expiration dates and activities since they don't really add any new functionality to the app for their complexity.
- Added in the functionality of logging in and saving the User's account, except for PrescriptionMedicine. In order to save the state of the program, you must **log out**.

## UML Design:

The UML diagram for the project is included in the GitHub repository under the folder UML_Diagram. There are multiple images in that folder, each image corresponding to a UML diagram of a package in the program. There is also one pdf, which combines all the images in one document. If some images in the pdf file are too blurry, you can refer to the image with the same title in the folder.

## Major Changes:

One major design decision we made was to get rid of the activities part of the specification. Our group discussed what the purpose of activities was and how to implement it, and we determined that it would take a significant amount of time to implement this feature, however, it didn't contribute significantly enough to the program's purpose. Thus, we decided to not do the activities part of the prescription for now but could implement it in the future as a possible extension.

Another major design change was changing the way AppManager was implemented. We changed from calling a method at the end of each method to instead returning a string. The run method now has a while loop and a switch-case block. The switch-case block checks the returned string and calls the next appropriate method. Since each method calls a different window, this is analogous to calling the next window. The loop continues to run until the logout option is selected. This change makes the call stacks when errors occur much shorter and allows for us to be able to check the time at each iteration of the loop.

## Brief Description of Conformance to Clean Architecture:

As will be mentioned later, we decided to use the by-package design. We made sure that we did not import anything that would be in an outer layer. We made sure that it was testable, and that the entities were able to be tested without any external element such as a UI or database. One thing that was harder to adhere to was selecting what data crosses what boundary. For example, rather than have simple data types cross, we have managementSystem in App Manager. Other than this, we don't really pass in entities through boundaries. We also used a data access interface to save the state of the program and reload afterwards. We implemented this interface in frameworks and drivers, but the inner layers only depend on the interface, not on the implementation, which adheres to Clean Architecture. Overall, we really strived to adhere to Clean Architecutre and not violate any of the layer rules.

## Brief Description of Conformance with SOLID:

Our project adhered to the SOLID design principles. During phase 1, we either added new entities or added some extensions to the pre-existing ones which adhered to the OCP(Open/Closed principle) of the solid design principles. Most of our classes if not all had a single responsibility and we tried to break any class that achieved more than one responsibility into smaller ones. This adheres to the SRP(Single responsibility principle) of the solid design principles. One flaw we had, which has to do with the single responsibility principle, is with app manager. We had it as both a controller and presenter, which means that it should be broken up to separate classes. All of our subclasses can substitute their respective superclasses because they are only an extension of them and are not modified which makes our project meet the LSP (Liskov Substitution principle) of the solid design principles which also connects to OCP. Also, we made sure that all of our interfaces were segregated and only had one responsibility. We made sure that none of the classes implemented an interface with unnecessary methods, adhering to the ISP(Interface Segregation Principle) of the solid design pattern. Finally, Our project followed a clean architecture and no high-level modules were using low-level modules and instead, we used dependency inversion to create an abstraction between them on which both could depend on.

## Brief Description of Packaging Strategy:

We decided to go for the **by-layer** packaging strategy. There are 4 layers, each corresponding to a layer in Clean Architecture. This was the most obvious choice due to how we organized our CRC cards. Since most of the classes were made based on the CRC cards, we decided to use the by-layer strategy to keep things consistent.

## Summary of Design Patterns

One major design pattern we have used since phase 0 is the Dependency injection design pattern. We used this pattern in the frameworks and drivers package, in which we created the different windows of the app in Main, and injected them into AppManager. AppManager only depends on the Window class and DisplayEntityInformation interface, and not the individual Window subclasses. We create and inject these subclasses into AppManager in Main, through the run method. We also use this pattern to inject the data access implementation.

# Progress Report:

## Member Report:

- Benjamin:
  Note that everything Benjamin did will be in Phase 2, as a communication error due to a pull request caused what he did to be incompatible. Everything he has done so far can be seen on his branch. Benjamin changed the instance attribute of Event to LocalDateTime. He made every class compatible with LocalDateTime, such as AppManager, Schedule, and AddMedicineWindow. He ensured that the medicine would be stored properly. He also helped implement the login system, which consists of serializing the entities, creating a data access interface, and implementing their calls and uses. He and Sujoy used Bora's code to implement the login system.
- Bora:
  Bora implemented the databases for Medicine and User classes by using Serialization. It creates a Users.txt and Medicines.txt file and writes in it when a user creates an account. If needed it reads the user or medicine information on the databases.
- Eren:
  Eren collaborated with Mohammed in creating the PrescriptionMedicine class and its related classes/functions in other files. He worked on EditPresciriptionWindow and the methods that are related with it in both AppManager and Management System. He also took advantage of pre-existing windows such as AddMedicine and RemoveMedicine windows.
- Mohamed:
  Mohamed implemented the prescription medicine feature where a user could add a medical prescription. Mohamed implemented the PrescriptionMedicine entity where it stores the prescription's name and all of its corresponding medicines. He made a data variable where all the prescriptions and their names are stored. In addition to that, Mohamed implemented the AddPrescriptionWindow and RemovePrescriptionWindow which are the windows the user is prompted to when adding and/or removing a prescription. He also largely contributed in implementing EditMedicineWindow where the user could edit the information of a specific prescription. Finally, he made some minor changes in some of the pre-existing files and resolved some bugs in them.
- Mouaid:
  Mouaid created the OtherActivities abstract class and used them to create the Sleep and Meal entities through inheritance. Using Sleep and Meal, Mouaid added the full

functionality of the user being able to add sleep and wake times, as well as meal times to their schedule. With OtherActivities, Mouaid also created the OtherActivitie's use case, OtherActivitiesManager, and created windows SetMealTimingsWindow and SetSleepTimingsWindow for the user to be able to input their sleep schedule and meal schedule. With adding 3 new entities, he made all the corresponding changes to User, UserManager, ManagementSystem, and AppManager, as well as adding unit tests for User, Sleep, Meal, and OtherActivitiesManager.

- Sujoy:

  Sujoy first changed the description of each event so that it included the amount of medicine to take, and added in an attribute which represents the unit the medicine is taken in. Both of these made the description of the events more descriptive and clear. He also added in the full functionality of editing and removing medicine from an account, and changed the implementation of AppManager so that it used a switch-case block to call the different methods rather than call a method at the end of another method *(see **Major Changes** for more information as to the reason of this change)*. Sujoy added unit tests for User, Medicine, Schedule, MedicineManager, and UserManager, as well as debugged the program and resolved conflicts when merging different branches. Sujoy also implemented the login system with Benjamin, using Bora's code as base for the serialization aspect. He and Benjamin debugged the code so that serialization works properly.

## ● Open Questions:

- Do you have any suggestions, tips, or things to look out for when migrating a project to Andriod? - *Sujoy*
- Is it better to use Android Studio or Android on IntelliJ? - *Sujoy*
- While implementing Serialization, we noticed there was a lot of emphasis on manually setting the serialVersionUID. I was a bit confused as to how to do this properly, and was hoping if you had any advice regarding this topic. I don't believe this was talked about in lecture, from what I have seen. - *Sujoy*

## ● Design Decisions: What worked and why?

- The idea to use the by-layer packaging strategy, was a great decision because it made deciding where to implement new classes easier, and it also made it easier to check to see if there are any violations to the layer rule of Clean Architecture.
- While we do need to add methods to a lot of classes when making a change to a class in the entities package, the path of information is clear and easy to follow, and

making changes mid-path is also really easy and quick. Such as when Sujoy went to implement a new attribute to Medicine, it was really easy for him to make the necessary changes easily and quickly by following the method calls and adding a small change where necessary. It took a lot less time than it would have if we hadn't used the by-layer packaging strategy to separate the classes

- I believe the idea of having an app manager and a management system was a good idea, as they act as the border between the application business rules and the frameworks and drivers. The app manager is the interface side and the management system is on the applications business rules side. It makes the process of adding new features, and making changes more structured and actually helps me plan out how I will implement and test the new feature. - *Sujoy*