Updated Specification (Benjamin):

A medicine can either be prescribed or not. If the medicine is prescribed then each prescribed medicine should be grouped together. Regardless of whether or not it is a prescribed medicine or regular medicine, it should store the name, time to take or taken, when to take it and how often to take it.
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 log out and the program will save the user's username and password and all of the associated information. However, if the user wishes to log in into their account, the program should be restarted 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 log in using their login credentials.
Highlights:

- We fixed AppManager and MedicineManagement so that it uses the facade design pattern and it makes other classes do the work.
- We added the GUI and the ability to use it outside of the console

UML Design (Sujoy):

The UML diagram for Phase 2 is under UML_Diagram_Phase_2 in the Github repo. There is a pdf file with all the images of the UML diagram, but if some images are blurry, there are images by the same name in the document

Major Changes (Mouaid & Sujoy):

A Major change was implementing the Facade design pattern for the ManagementSystem and AppManager classes so that they now delegate tasks to specialized classes in their facade which greatly reduced the size of the classes. In previous phases, we noticed that AppManager and ManagementSystem where getting increasing big as more features were added, and thus we decided to use the Facade design pattern to more clearly adhere to the single responsibility principle.

Another major change was the decision to use Java Swing and AWT libraries instead of Android. When we started to research into Android, we noticed it heavily depended on XML files and didn't have an actual main method. Rather, it had a main activity and used that to call other activities. The way we implemented the Main class was to initialize all of our windows and call AppManagerFacade to run the program. In order to migrate to Android, we would have had to majorly restructure the way our outer layers operated, and we felt that this shift was too big and not feasible for the amount of time we had available. Hence, we decided to use Java Swing and AWT instead, as Sujoy had previous experience with these libraries and thus, believed it would be faster to implement than Android. Mohamed and Sujoy also spent an entire week trying to understand Android and were struggling, which in addition to time constraints influenced our decision to use Java Swing and AWT instead.

Brief Description of Conformance to Clean Architecture (Mohamed):

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 Architecture and not violate any of the layer rules.

Brief Description of Conformance with SOLID (Mohamed):

Our project adhered to the SOLID design principles. During the past two phases, 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. In one example, some extensions needed to be made in ManagementSystem so that it would work. In order to follow the Single Responsibility principle, we broke up AppManager and ManagementSystem to smaller classes to adhere to the SRP(Single responsibility principle) of the solid design principles. 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). This can be seen in the Window class as any subclass can be replaced with Window . Also, we made sure that all of our interfaces were segregated and only had one responsibility. One window, AddPrescriptionWindow, wasn't adhering to that, so we fixed it in this phase. 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, like Frame Observer where it only has one method.. Finally, our project followed Clean Architecture and no high-level modules were using low-level modules and instead, we used the dependency inversion principle to create an abstraction between them on which both could depend on.

Brief Description of Packaging Strategy (Bora):

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. One disadvantage of this, and what we didn’t have time to change was going by something else because it got cluttered in the end.

Summary of Design Patterns (Mouaid)

One major design pattern we have used in the past two phases 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 the AppManagerFacade. THe AppManagerFacade only depends on the Window class and DisplayEntityInformation interface, and not the individual Window subclasses. We create and inject these subclasses into the AppManagerFacade in Main, through the run method. We also use this pattern to inject the data access implementation.

Another Major design pattern that was implemented was the facade design pattern. We used this design pattern on two different classes, ManagementSystem in the Application Business rules layer and AppManager in the Interface Adaptors layer. These two classes were the largest two classes in our project and were responsible for carrying out all the tasks related to taking info coming from the windows and taking it to the entities. We implemented the facade pattern by creating Accounts, ActivitySetter, Medicine, and Prescription classes for each facade and delegated it the tasks related to its name. This greatly reduced the sizes of these major classes, for example, the size of AppManager was cut by more than half.

Progress Report (Bora):

- **Member Report:**

  - Benjamin:

    Benjamin changed it so that we would use LocalDateTime rather than a map so that we could use days of the month and months rather than the days of the week like Monday, Tuesday, etc. (This was in the last phase but the pull request was merged in this phase). He also fixed the unit tests so that they would show these changes and made AddMedicineWindow, and SelectTimesWindow so that it would work with the GUI.
    https://github.com/CSC207-UofT/course-project-mouaid-and-associates/pull/54/files
    This pull request has the Unit Tests that Benjamin has changed and some work that he has done on AddPrescriptionWindow to show part of his work on the GUI. It also contains a checker to make sure that the inputs in the windows he implemented, AddMedicineWindow and SelectTimesWindow to make sure the times are valid.
    https://github.com/CSC207-UofT/course-project-mouaid-and-associates/pull/52/files
    This pull request shows a significant portion of work that Benjamin did in making the GUI as he was in charge of AddMedicineWindow, AddPrescriptionWindow, and SelectTimesWindow.

  - Bora:

    Implemented the Choose Medicine To Edit Window and Choose Prescription To Edit Window, which are the windows before EditPrescription and EditMedicine and also tested some parts of the code. Made necessary changes in both AppManagerMedicine and AppManagerPrescription.
    https://github.com/CSC207-UofT/course-project-mouaid-and-associates/pull/49/
    https://github.com/CSC207-UofT/course-project-mouaid-and-associates/pull/59
    Both pull requests show a significant amount of his work on the features that are described above.

- Eren:

  Eren did the EditPrescriptionWindow, ViewAccountWindow, and changed the necessary functions in AppManagerPresenters, and AppManagerPrescription, being showAccountWindow() and editPrescription(). EditPrescriptionWindow had 2 more implicit layers, it being the changing name of the prescription and removing medicines from that prescription. At first he implemented those two as different classes but then used them as helper functions inside the same class. https://github.com/CSC207-UofT/course-project-mouaid-and-associates/pull/51/files This pull request shows a significant portion of his work on the features described above. He also contributed significantly to the graphics of ChooseMedicineToEditWindow and ChoosePrescriptionToEditWindow.

- 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 and he serialized the PrescriptionMedicine entity. One major pull request that Mohamed made was that he compiled all of the code from all the different branches into one final code project. https://github.com/CSC207-UofT/course-project-mouaid-and-associates/pull/61
- Mouaid:

  One of the main things that Mouaid worked on was implementing design patterns to improve the structure of our code. He implemented the Facade design pattern for both the ManagementSystem and AppManager classes and converted them into the ManagementSystemFacade and AppManagerFacade classes, each with their respective classes. These included creating AppManagerAccounts, AppManagerActivitySetter, AppManagerHelpers, AppManagerMedicine, AppManagerPrescription, AppManagerPrescriptions and refactoring the AppManger class into AppManagerFacade. The pull request below shows most of the work that went into implementing the AppManagerFacade https://github.com/CSC207-UofT/course-project-mouaid-and-associates/pull/36

It also included creating ManagementSystemAccounts, ManagementSystemActivitySetter, ManagementSystemMedicine, ManagementSystemPrescription and refactoring ManagementSystem into ManagementSystemFacade. The pull request below shows most of the work that went into implementing the ManagementSystemFacade https://github.com/CSC207-UofT/course-project-mouaid-and-associates/pull/39 Mouaid also implemented the GUI for both the SetSleepTimingsWindow and the SetMealTimingsWindow so that there are actual graphical windows for the user to set their sleep and meal times. The pull request below shows most of the work that went into implementing the GUI in SetSleepTimingsWindow and the SetMealTimingsWindow https://github.com/CSC207-UofT/course-project-mouaid-and-associates/pull/53

- Sujoy:

  Since the end of Phase 1, Sujoy has been collaborating with Mohamed on migrating to Android and implementing a graphical user interface for the app. After a lot of research and struggle with Android, he and Mohamed had decided to move onto Java Swing in order to implement a GUI for the project. He and Mohamed spearheaded the development of the GUI, by building the base framework for others to build GUIs for their windows, and helping the rest of the group with implementing the GUI, through long sessions of debugging and explanations. Personally, Sujoy implemented the GUI for LoginWindow, StartScreenWindow, TimeTableWindow, and EditMedicineWindow. Sujoy implemented the full functionality of edit medicine (https://github.com/CSC207-UofT/course-project-mouaid-and-associates/pull/19) for the project. This is a significant contribution due to him implementing the full functionality of a critical feature described in the specification. He also, with the help of Mohamed, implemented the base framework on which the GUI was built, as well as provided example cases for the rest of the group to use (https://github.com/CSC207-UofT/course-project-mouaid-and-associates/pull/40). Sujoy also spent a significant amount of effort in leading the group project, through scheduling and leading meetings, delegating tasks, and communicating with the rest of the group about deadlines and progress on a weekly basis. For future plans, Sujoy plans to look more into multi-threading and UI design. While working with Java Swing, having multiple threads run simultaneously was confusing and a struggle, thus Sujoy plans to learn more about this concept, to better improve the GUI performance, and make it more accessible.

- **Design Decisions: What worked and why? (Bora)**
  - 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.
  - Due to a small amount of time, we have decided to switch to Java AWT graphics from Android. We agreed on the process of developing the basic GUI.
  - We have decided to use the Facade **design-pattern** for both the AppManager and Management system classes. It was a great decision because it hides the complex part of the code of the larger systems and provides a simpler interface to the user.
  - We tried to implement the Observer Design Pattern but since the classes mutually update each other we didn't implement it.

Accessibility Report (Eren)

For each Principle of Universal Design, write 2-5 sentences or point form notes explaining which features your program adhere to that principle. If you do not have any such features you can either:

(a) Describe features that you could implement in the future that would adhere to principle or

(b) Explain why the principle does not apply to a program like yours.

Principle 1: Equitable Use

The extent for privacy, security are equally available to all users, since the user information is saved in the same place. However, since all information is saved in one file it has low security. Initially, we had the labels above text boxes turn red if the input was invalid, but we thought that this may put colourblind people at a disadvantage, so we decided to use a popup window to notify the user instead.

Principle 2: Flexibility in Use

We did not allow much flexibility for the user. We didn't have anything like a dark mode and there were no preferences that a user could select to change the design or layout. The layout was neutral in the sense that both right handed and left handed people could use it easily.

Principle 3: Simple and Intuitive Use

After the user logs in everything is handled and directed by ViewAccountWindow, and every window goes back to ViewAccountWindow, so it will be very hard for the user to get lost. Our program just uses text fields and buttons for user input so it is very straightforward. We tried to make the language clear when asking for information from the user like what the desired input was. We tried not to clutter it with unnecessary information and decided to go with a simple layout. We did give feedback when the user added incorrect information and tried to be clear as to what was causing the problem.

Principle 4: Perceptible Information

We haven't addressed people with visual disabilities. In future extensions, we can add a voiceover to our console version of the program -which already exists- so that the person could use the program easily through the keyboard without having to see anything. We did highlight incorrect input, and made sure that it would grab users' attention by making it appear in the middle of the screen.

Principle 5: Tolerance for Error

One idea for future implementations is that we may store all the information the user enters so that, in case they accidentally delete medicines/prescriptions, they can be recovered or we can ask the user if they are sure they want to delete it. We did provide warnings of incorrect input and made sure that it wouldn't cause the program to crash.

Principle 6: Low Physical Effort

If a user wants to remove a medicine from a prescription, they can remove any number of them in one window. Our old implementation of this feature was that the user kept coming back to EditPrescriptionWindow from the Account window to remove medicines one by one. We tried to minimize repetitive actions, by allowing the users to input times for an entire week at once. In the future, we can make it so that the user can select how long to add the information for.

Principle 7: Size and Space for Approach and Use

The buttons are big enough so that it reduces the risk of misclicking. We tried not to clutter it, and make the boxes big and clear so that they would be able to be seen easily.

Write a paragraph about who you would market your program towards if you were to sell or license your program to customers. This could be a specific category such as "students" or more vague, such as "people who like games". Try to give a bit more detail along with the category.

We are targeting people who have more than one prescription, or people who have a lot of medicines that they need to keep track of.. Our program keeps track of numerous medicines which can be hard for the user themselves to track. We schedule the times to take medicines according to the user's meal and sleeping schedule. Furthermore, old people will also be targeted with this program with its easy-to-use and straightforward user interface, and since old people have a higher possibility of having to keep track of many medicines.

Write a paragraph about whether or not your program is less likely to be used by certain demographics. For example, a program that converts text files to files that can be printed by a braille printer is less likely to be used by people who do not read braille.

Our program would not be used by people who don't have a lot of medicines they need to keep track of, or if they don't have any at all. These people do not need to keep track of medicine or it may be too tedious to use if they only have to keep track of one or two medicine(s) for a short period of time. Although this may be the case, it is possible that we could make the program create schedules for general use rather than medicine specifically, so that it is more flexible. If people have appointments or want to use it as an agenda that is something we could do in the future.