CS207 - Design Document (Phase 2):

Specification:
See the specification.md file in this directory.

Major Design Decisions:
Expressing criteria (for entry to a facility or for discount applicability) as a string. For example, "(1/2/3).(Math/CompSci):(3/4).(Music)" means "students in years 1, 2, or 3 in the math or computer science department, and faculty in years 3 and 4 in the Music department". The symbols "/", ".", "(", ")", and ":" help with parsing over the string to decode it. We decided to do the criteria like this because we wanted a compact way to store it in the database.

Creating a "check access" option in the facilities menu of the app. Selecting this option for a given facility will display a message indicating if you have TCard access to the facility or not. This is determined by checking if your user information satisfies the criteria for access listed by the facility. We did this because we wanted a reasonable way for users of the app to interact with facilities virtually

.
Adding merchants into the app, and deciding on their features. Merchants and their features work similarly to facilities. Instead of a "check access" button, each merchant has a "see your discouns" button that you can select to see which discounts the merchant offers apply to you.

Adding a functional UCheck feature in the app, which has the UCheck questionnaire. The UCheck saves and stores the data in the app for future reference and displays a date from the last completion of the UCheck questionnaire. You must complete all the UCheck questionnaire questions before submitting it (enforced). The UCheck dashboard also has a similar feature to UCheck, providing a clickable button that links to a UofT resource page.

Code Style and Documentation:
We fixed most of the warnings that existed in phase 1. We still got some warnings in the UserDBhelper, FacilityAdapter and the User class. But we will fix that very soon. Warnings regarding the gradle version should be ignored because the current version of gradle is the only one that can run on everyone's computer. Updating the gradle may result in issues.

There was inconsistency in our naming style. We've tried to use the gradle plugin "checkstyle" to automatically check the errors. But it didn't work. So we had to check it manually. Most of the inconsistencies should be removed now.

Documentation will be done by Wednesday, and they will be for all methods.

Testing:(Line Coverage)

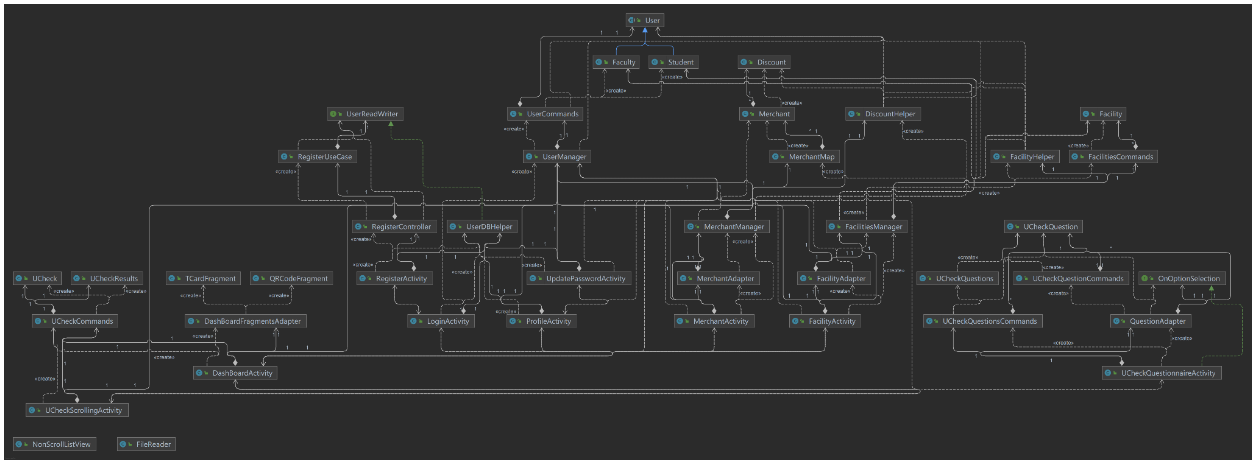
- Entities(100%)
- Usecases(33%)
- Controllers(29%)
- The testing coverage was higher. But unfortunately some tests were not successfully committed to Github and we lost those tests by accident. We've added a lot of code after phase 1 and that also lowered our test coverage. We will improve the coverage to 50% before submitting the final version of our project.

Refactoring:

- Made changes to variable types to satisfy the open closed principle. For example, we refactored ArrayList<String> var = new ArrayList<>(); to List<String> var = new ArrayList<>(); wherever it was applicable.
- Corrected the naming of variables and methods so that they have the correct format (camel case with first letter lowercase)
- Removed unused import statements
- Refactored the user database accessor class UserDBHelper. This class is a Gateway (it belongs in the interface adapters layer), so use cases that depend on this class will violate clean architecture. To solve this issue, we created a use case-level interface UserReadWriter (which lists all the methods from UserDBHelper that the use cases will use) and made UserDBHelper implement this interface. This way, the use cases will depend only on UserReadWriter, which is in the use case level, so clean architecture is satisfied.
- Refactored RegisterActivity. I made a controller class RegisterController that controls the user registration process and a use case class RegisterUseCase which is the place that takes the bulk of the responsibility of checking validity of user input.
- Refactored facilities so that it follows clean architecture. At first it was using entities inside the activity classes, so I made use cases and controllers. For the facility check to see if the user can enter, the use case gets the information from the facility and the user entities, a controller calls the use-case to get the criteria of the facility and the user to check if they are the same, and the activity calls the controller when the user presses the button to check access.
- Refactored UpdatePasswordActivity so that it follows clean architecture. At first it was checking the validity of the new and old password in the activity classes, after refactoring it does these operations at the use case level.
- Refactored the color and strings in the app. Before refactoring, the string content and some color hex codes were directly in the layout xml or java activity classes. Now the string content and hex codes are stored in the values folder (string.xml and color xml) so that the layouts and activities call the strings and colors needed by their recorded id's (all

the layout strings were moved, the activities strings and colors will be moved by Wednesday). This makes it easier to implement accessibility features like dark mode and multi language options and makes the app easier to extend.
- Initially we plan to use a file reader to set up all the information for facilities and merchants. We spent a lot of time on testing it and all the tests passed but when we ran the app the reader didn't work. We then decided to hard code the information into facilities and merchants.
- Refactored User according to Chris's feedback. We now implement it using HashMap instead of a list. This allows us to get the attributes of the user without knowing the order of the user's file.
  Link:https://github.com/CSC207-UofT/course-project-the-frank-lloyd-bytes/commit/c4874e9109ca8b634dcdcc1733433301966ed20c

Clean Architecture:
- We believe that our code is organized very meaningfully and clearly. To follow clean architecture, we created packages for entities, use cases, controllers, data base, and UI in our code. This way, we always know when we can directly call from another class and if it violates clean architecture.
- It is very easy to find things in this package structure. Similar classes and interfaces this way are packaged together and they are named intuitively enough to know which package they belong to.
- The tests are packaged the same. For example, the tests in the entities package in the testing contains unit tests for the entity classes.
- As one of our main focuses in phase 2 was refactoring, we believe our project satisfies clean architecture well over all, except possibly for a few violations. We plan to deal with any outstanding violations before the project deadline.
- Talk about violations
  - Smelly code UCheckQuestions:
    - This code has hard coded questionnaire for UCheckQuestionnaireActivity (Jesse)
- UML:

- 

SOLID:

- The user class adheres to the single responsibility principle well. User class and its subclasses: Student and Faculty are only responsible for storing and displaying the information of the user.
- We only have one use case class for User, which takes all responsibilities like changing password, showing profiles and creating user objects. And this problem exists in our other higher layers classes too. This is because we don't follow the dependency inversion principle. We designed the low level classes first and therefore all our higher level classes had to depend on the low level class. It was not hurting much since the program isn't big. But we will try to solve this in the future.

Design Patterns:

Adapter: We used the adapter design pattern for the recycler view for facilities and merchants. These adapter classes in activity are subclasses of the RecyclerView.Adapter. The activity classes don't know what the superclass does, but it takes the required methods for the class to work as we want it to work. In other words, it takes two completely unrelated classes and allows them to work together by converting a class's interface into the interface the client expects. We also used an adapter for UCheckQuestionnaireActivity and for ViewPager2 in the DashboardActivity to display TCard and QR Code fragments.

IN THE FUTURE, ***Strategy*** can be used for facilities and merchants. The addresses, and hours of the facilities and merchants are recorded right now. If a location and time feature is added to the app (so the user will have to allow their location to be known by the app), there can be algorithms that organize and sort the merchant/facility list based on:

- Proximity
- How far their closing time is from now
- How close their opening time is from now

- What type or merchant/facility they are (eg. library or gym)
- Discount percentage (for the merchants)

With strategy design patterns each of these sorter algorithms can have one Sorter interface that they all use and then have their own classes where individual algorithms are stored in. The user can select which strategy to use to see the list of merchants and facilities. This pattern will also fulfill the open/close principle as we can easily add a new algorithm to sort the merchants or facilities out by adding a new strategy class that uses the same Sorter interface (e.g. UofT gives us access to live data of available spaces left in facilities, so we implement an available percentage strategy). So we'll be extending the app without modifying other classes.

Builder: We tried using the builder pattern for the user, student,

Use of GitHub Features:

- Merging from each other manually due to the pull request issues
- Used token cards more effectively this time. Since we were already used to how it worked and everyone mostly already knew what they were doing, we could use the token cards as a way to update other team members on what was done and whether someone else can merge the code onto their version.
- Used smart merging to merge only files that work as intended and not the ones that may have bugs. This way, we don't have to wait a long time for each person to completely finish their task before someone else can start theirs.

Functionality:

- A function that we couldn't get to work in phase one was facilities. Due to the lack of time, we weren't able to finish it and get it to work in time. However, for phase two, we were able to complete it. It now checks the logged in user and the facility the user presses on to see if the user has access to that facility.
- We also added merchants. With this tab, the user can see a list of all merchants available near them in the university. Pressing on the button in the merchants page will allow them to see what discounts are available for each merchant.
- We also added UCheck to the app. It is very similar to the UCheck at our school as it gives a specific survey to the student everytime it is launched. However, unlike a site, we have integrated it in our app so that everything is in one place when the student wants to go to a facility to show them tCard and their UCheck. It shows whether the student is able to enter campus after completion of the self-check survey.
- Change password
- User registration
- Upload profile picture and dark mode

Accessibility & Universal Design Principles:

- ***Day/Night Mode*****:** We have implemented dark and day night modes for the app that can be used by switching the viewMode switch found in the Dashboard. This feature minimizes the fatigue the user can experience by looking at a bright white screen at a low lighting environment and provides a method of choice to the user. So it follows the **flexibility** and **low physical effort** universal design principles.
- ***MultiLanguage Option:*** We have set up the bases for implementing multiple language options for the app. Right now all the strings displayed in the xml layouts are stored in an English and French string.xml. Ideally, with a radio button group in the login page, the user can switch between languages (though this doesn't work yet). We decided to implement this feature as Canada is a bilingual country and most official organizations are required to offer French options as well in the services they provide to their users. We are also considering implementing a Mandarin language option as a large portion of the student body has Mandarin as their mother tongue. Implementation of this feature will make the app **equitable to use** as users not comfortable with English can still easily use the app. And even if they're comfortable with English, they'll have a choice to have the app in different languages which increases the **flexibility** of the app.
- ***UCheck CardView Colors****:* Once the user logs in, they'll be faced with the dashboard where the UCheck Cardview is displayed. While this card has the functional use of taking the user to the UCheck test + results, it also changes color and text content based on whether the user has taken the test yet or the results they received from the test. Having these vibrant colors and large fonted UCheck results displayed in the dashboard maximizes the "legibility" of the UCheck results and increases the **perceptibility of information**. It also acts as a warning to users who have not taken their UCheck test yet or users who failed their UCheck test. In the future a notification feature can be added if the user has entered the campus area (they need to allow the app to access their location) but haven't completed the UCheck test yet to increase the implementation of **Tolerance for Error principl**e.

Open Questions and what has worked well with our design:

- For dark mode, I thought about having an entity class that would store the app settings (because in the future of the app there'll be more settings to store eg. language of the app, location enabled or not, access to gallery granted or not…) Would having an AppSettings entity class + use cases + controllers be the right way to go for this, or would using Shared Preferences for each of them be enough?
- We have been having troubles with reading the merchant and facility data from their csv's while the app is running. Before, we were reading the data every time we opened the facilities or merchant page. The app would crash after clicking on their menus from the dashboard. Our current solution is to just hard code their data into the app. Is there a correct way to read their data from csv that will not cause errors?

What each member has been doing and plans to work on next:
The document should include everything outlined in phase 1, except the progress report should contain:

- a brief summary of what each group member has been working on since phase 1
- Each group member should include a link to a significant pull request (or two if you can't pick just one) that they made throughout the term. Include a sentence or two explaining why you think this demonstrates a significant contribution to the team.

Yanbin:

- Refactored User.
- Created tests for the project.
- Correct the naming style issues
- I have been merging everybody's work manually because the pull request was not working properly. I've also been fixing all of the conflicts when a new feature was added.
- This commit is a significant commit I made this term. With this commit, and several other commits close to this one, I've built the framework of our project and linked our app to a database, which many of our functions are based on.

Kristal:

- Got picture from gallery -> extracted picture uri -> stored picture uri in user and database -> displayed picture uri in imageView (Yanbin helped me in this section because we needed to add a new column to the database to store picture uri and I didn't quite know how the database worked)
- Refactored update password to have a cleaner architecture (moved some code into use case and controller classes)
- Fixed a small bug in register activity page (it used to crash when the user tried to sign in without filling any empty field because no radio button getting checked created problems)
- Implemented dark mode for accessibility (it has some problems tho, but better than nothing)
- Started implementing multi-language option but currently it's not working (apparently it wasn't as easy as it looked) + refactored how the string values are stored: previously string content was directly in the layout xml's, now it's in the string.xml in the values and layouts call the string id to get the content
- My ~~pull request~~ commit-and push action: Last step of implementing the upload profile photo picture feature. With this commit we fulfilled our TCard feature of our specs by completing the virtual TCard visuals with a profile picture. It was also an important commit-push because we now have a code section we can follow/copy paste if we ever want to import other pictures (like vaccine passport pictures) and store them in our app.

Jonah:
I did some refactoring of various parts of the project to ensure that clean architecture is satisfied. In particular, I refactored UserDBHelper and RegisterActivity. Here is the link to the associated pull request: UserDBHelper and RegisterActivity refactoring. More significantly, I added the merchants feature to the app in phase 2. This includes the actual design of merchants (in all layers of clean architecture), providing testing (for the entities, use cases, and controllers associated to the merchants feature), writing the java doc, and creating sample merchants data stored in a csv. The merchants feature satisfies clean architecture well. Here is the link to the pull request associated to the merchants feature: Merchants. I think these changes demonstrate a contribution to the team because:

- The RegisterActivity class was one of the places that needed to be refactored the most, as expressed in the phase 1 feedback.
- The Merchants feature is something we had hoped to do since phase0. I feel that adding this feature in phase1 demonstrated that we were able to extend our program well.

Jesse:

- Refractored User, UserManger and UserCommands (so interface controller does not call directly from entity).
- Finished UCheckScrollingActivity, UCheckQuestionnaireActivity and created entities, use cases, an adapter, and an interface for functionality and behaviour.
    - Also did UI design for UCheck (beside DashboardActivity).
    - Created a SharedPreference to store UCheck data with USER.
    - Created a Non Listview for UCheckScrollingQuestionnaire since Scrollingview looked and performed awful.
    - Created XML files, string values, and additional resources.
    - Fixed bugs.
    - Tested functionality in emulator with questionnaire.
- Helped Kristal with UCheck in DashboardActivity.
- UCheck functions (although I did not implement a 24-hour expiry).
- Implemented test cases for UCheck entities, use cases. Implemented additional test cases for User and UserCommands.
- Cleaned up my code, added documentation, and did my best to adhere to clean architecture.
- Significant pull request
    - Pull Request 1 Made the questionnaire implementation work. UCheckScrollingActivity works with entities and use cases hidden in some layers. Needed some more improvements here though.
    - Pull Request 2 Did some polishing, cleaned up for clean architecture, and introduced test cases. Refractored User class, UserManager and UserCommands

to work with my implementation on the front-end with UCheckScrollingActivity and QuestionAdapter.

Ming: refactored facilities before implementing everything. Previous version didn't work so I had to redo everything for facility. First I made sure that the classes followed clean architecture. Then I wrote methods for these classes for facility to create the facility entities from the database, or in this case what we hard coded in when the app runs. Last, I coded the facility activity classes so that the GUI works. Refactoring Facility, for writing the actual facility activity classes, I used Yanbin's computer so I pushed with his account, but everything pushed onto my branch was made by me even though it's under Yanbin's account.