Specification:

UofT Campus Location Review System:

Specification: i. The user creates a new profile with his name and student number. The profile stores his ratings and reviews of various buildings. The profile also stores his preference list which is a collection of ranked lists of buildings ranked according to different criteria like: study, relaxation, quiet.

ii. Running the program **creates a shell** that prompts the user to input his location

iii. The user inputs his location based on the alphanumerically gridded map.

iv. Each building has attributes like: private study space, group study space, bathrooms, water fountain, food, **accessibility**, grid location

v. The user can choose which attributes they do and don't want from the specific buildings. These would be implemented as filters in the search. For example, the user can specify if he only wants buildings with bathrooms, or if he wants a quiet study area, or buildings with above 4 star ratings, or a search radius.

vi. The program will return locations with those attributes that best fit the users specifications

vii. The user can leave a review of a building in stars **and a written review**.

Progress Report:

We all have been gathering some data on buildings around campus to use

David: Has worked on making the Dialogue controller class and making it so that the main class follows clean architecture. Implemented the method initiate review, and made it so that the user can input their preferences in a search function. Also made the UI window for the user to input a review. Allowed the user to choose a building to review. Two key commits are https://github.com/CSC207-UofT/course-project-comfortivity/commit/ad1a2bca76d6034254ad5fe328891d3edbd02c7b
And
https://github.com/CSC207-UofT/course-project-comfortivity/commit/ded09de5096eab11394c7d3817388de8b1748e3c
Since they see the user being able to choose what building to review and the addition of a few UI windows.

Edmund: Created database and added collections and documents using MongoDB Atlas Database. Implemented UseCases classes for Entities User, Building, Review, and schoolMap and their corresponding Gateway classes which retrieved data from the aforementioned database. Used the Dependency Injection design pattern to ensure clean

architecture in the case of the Dependency Inversion Principle was followed between the Usecase and Gateway classes.

Gesikeme:
Wrote the original code for the CLI, which has now been updated and replaced with our GUI. Implemented the Builder pattern for the User class; with this I implemented a slew of getter and setter methods to facilitate the function of the design pattern.

I implemented a UserBuilder interface which the UserBuilderClass implements. I also implemented the UserBuilderClass and implemented builder methods that enable the client code(usually controllers and gateways) to update the user instance without a direct dependency. I also implemented a director class for the Builder design pattern that allows client code to perform more complicated and varied actions on the User class with a single method call, e.g. initialising a user with only his student number, or with his name, student number and preference list. This would give us more flexibility in future coding, in case we needed to create a user instance in more complex ways, or in different capacities, and still need to update that instance throughout the running of the application.

I also implemented the searchUseCase class which uses a percentage points system to rank how good a choice each building is based on the filters given by the User's search filters.
Commits:
https://github.com/CSC207-UofT/course-project-comfortivity/actions/runs/1518058270
This is when I had implemented the builder class design pattern for the User class
https://github.com/CSC207-UofT/course-project-comfortivity/actions/runs/1547124851
This covers me writing the search method for the search usecase class, and replacing the previous search method we had during phase 1.

Nori: created unit tests

Gurman(?):

Design Document:

- Solid

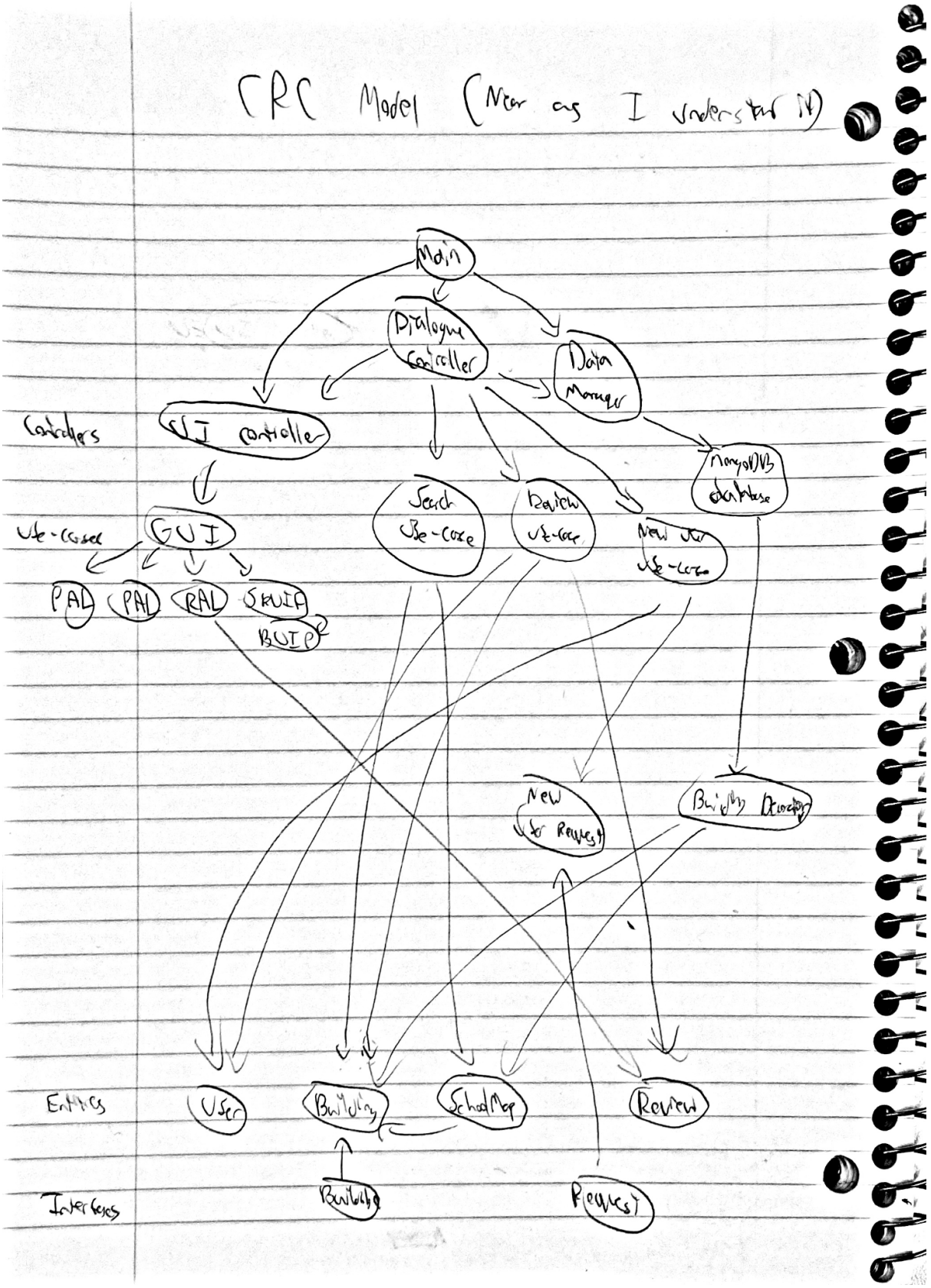
Single-responsibility principle: This, I think, we followed quite well. Each of our classes has, near as I can tell, only one responsibility. When we found some of our classes were getting too cluttered in terms of responsibilities, our 'request' classes arose to combat this. An object that clearly serves the purpose in its name helped hold us to that mark. For example, NewUserRequest makes a new user and does that only. We sort of use a facade design pattern for this as well. There's only one class managing communication with the user, but it passes into different use-cases and different request classes based on whether the user wants to search or review (or add?) buildings. Further, as our CRC model which we have posted later in the document reveal, most of our classes have only one reason to change, which comes as a result of them obeying the single responsibility principle. Our datamanager function would need to change only if we were changing the way we use our data, for example. Our database would need to change only if we want to change our datamanager. Things like this

Open/closed principle: Our ‘request’ objects are a good example of contribution to this, I think. A request class has our request interface, meaning it should have the necessary criteria of a request (be performed, etc) while it’s possible to, behind the scenes, change how this is done, though impossible to change what it does. Otherwise, we don’t use terribly many interfaces. In other places, the open/closed principle presents itself as a bit of a problem for our code. For example, some of our user’s preference criteria are hard-coded, so one could be unable to add or remove preferences as easily as you’d like, and you would need to change code in several places to, say, add an option to filter buildings by hours they’re open.
Liskov Substitution Principle: We don’t have many subclasses or superclasses here. Our buildings can be made differently, though, but each still has the fundamental traits of ‘building’, so this adheres to that I suppose.
Interface Segregation Principle: Again we don’t have terribly many interfaces, which may be a bit of an oversight, they could have been useful. What interfaces we do have are small, easy to parse, and fulfill a function clearly and well, at the very least. For example our request interface.
Dependency inversion Principle: This has a lot to do with clean architecture so I won’t talk about it too too much here. One problem I will raise with our code is that there isn’t always so many levels of abstraction between higher and lower level classes. For example the dialogue controller passes phrases into the GUI to be displayed as is and are hard-coded, which I believe circumvents the use of a presenter. Also, in some of our code higher-level code depends on the implementation of low-level code more than you would want. It sort of arose as entities are tempting to program first and understand well. For example, the frame that displays search results requires a building to have some specific attributes and it decides how to lay those out. Maybe in a perfect world, the building entity would have defined how to represent itself instead and would create a frame that it could return back to the higher-level GUI controller so low-level code wouldn’t depend on high-level code.

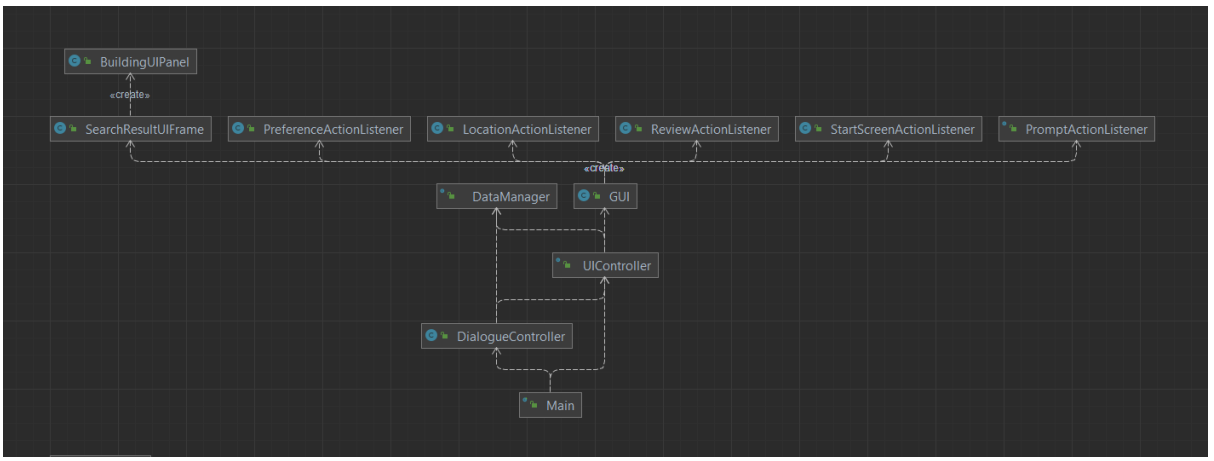
We were also cognizant of the Dependency Inversion Principle when implementing our Gateway and UseCase classes. Our original implementation had our UseCase classes tightly coupled to and dependent on our Gateway classes which was a violation of clean architecture as the lower-level classes should be dependent on the higher level classes. To circumvent this, we utilized the Dependency Injection design pattern which introduced an interface which was implemented by the lower-level classes. As the higher-level classes are dependent on the interface and not the actual lower-level classes, it adds a layer of separation to maintain SOLID principles.

- Clean Architecture

Following is the CRC model of our program, as near as I understand it. As is visible, we have use-cases, gateways, controllers, and entities very deliberately. The more abstract classes rely on the less abstract classes, with our databases, elements of our UI, and our entities at the bottom



Here is only part of it but definitely clearer



- Design Patterns
  We were supposed to use the decorator design pattern, but our teammate did not pull through on it. But we use the builder design pattern for the user class to allow us to make users with varying amounts of information to them. I suppose this contributes to privacy, as well
- Use of Github Features
  The branching helped us work on our own things at once, but maybe we should've merged sooner because we got some errors that we had a hard time sorting out in time just before the presentation.
- Code Style and Documentation
- Testing
- Refactoring

Our attribute capitalization has been all over the place the last two phases. Mostly we've gotten to work through it. Also, we finally implemented methods that had been abstract, unimplemented,, or hardcoded all this time. For example, in this specific commit, https://github.com/CSC207-UofT/course-project-comfortivity/commit/265f489979e7a4a229244a8ec2d2030ce8571f40 we finally made our search use case work how it was supposed to rather than be arbitrary, and this commit https://github.com/CSC207-UofT/course-project-comfortivity/commit/1775ca897c864a0d7d91dce7e067287783e6eec0
Made a lot of our methods from main smaller. If there was any main smell still in our code, though, it would also be long methods admittedly. But you can't win them all

- Code Organization
  We will get to packaging strategies later, one choice we made with our code that I think is good but is debatable enough to share is how some methods are one-line and exist to connect higher layers to lower layers. I'm not sure how consistent with clean architecture it is, but it makes it very intuitive what layer it's on when high level code calls low level code
- Functionality

Yeah, we have a database to save and load state. Yeah, I think the scope is sufficiently ambitious. We've sadly downsized as members of our project went awol but what the specification says, at least, we do. As shown in our presentation, I suppose

- Packaging Strategies:

We chose to package by layer, so that the architectural properties of each package stays together. This is in line with uncle bob's stance that classes should stay with other classes

that get used together, i.e. we keep the entities together because they’ll often be used by the same use-cases