Specification

An existing user can sign into their account using their existing login credentials. A new user can create an account by entering their personal information, course details, and personal interests. After signing in, user will be able to view their profile on the "Home" tab of the page; "Match Pool" shows the matches and profile alongside for the user to select a match; "Socials" let the user to add their social contact for the match; finally, "View Match" shows the profile of the chosen match; "Sign out" brings the user back to the log-in page. When the matching algorithm runs "Match Pool" will be updated. User credentials are stored and retrieved from a database. The database is extended to allow user enter up to 2 courses with tutorial section, lecture section, and course code.

Major Design Decisions

- At the end of phase 0, we realised UserManager was doing too much. Hence, we created User, Profile and then stored them into Database. We offload the creation of User and Profile to their respective factories.
- The text database was a good initial database for testing purposes but we needed a more robust and long-term solution. So the Azure Database for SQL replaces the text database.
- We chose to package by component over package by layer because this style of packaging would make it more intuitive to see how groups of classes are intra-related and interrelated.
- We chose to do a Web Application (webapp) over an Android App because of the flexibility of a webapp. A webapp can be accessed on any computer—desktop, laptop and mobile regardless of its OS.
- We chose to use the Spring framework with SpringBoot starters to code the webapp. We chose Spring because it facilitates dependency injection and web app design. We chose SpringBoot because it provides an easy way to manage dependencies and reduces the need for boilerplate code.

SOLID

Single Responsibility

The CRC cards outline each of the classes and were implemented such that each has a single duty. For example, the user class is only concerned with the user entity that stores all the data pertaining to the user such as the profile, unique identifier, and current matches. The profile class contains the user's information which includes name, year of study, program of study, courses for a term, interests, and contact information. The responsibility of storing the information in the profile class was not given to the user class in an effort to adhere to this single responsibility principle. The same idea can be extended to the course class containing a course's code, lecture time, and tutorial time in a separate class from the profile.

Open-closed principle

Our design is open for extension because currently, we have not added features like report and chat for users, but we plan to implement those features in Phase 2. Those features can be easily extended without changing the code. It is closed for modification because its basic function, which is matching users, stays the same.

Liskov substitution principle

We want to enforce the Liskov substitution principle starting in Phase2. For our project, this principle is demonstrated by the interchangeability of the user and admin classes. Since admin is a child of User, objects of admin should be replaceable by objects in User without breaking the program. Because User and Admin classes are interchangeable, it ensures that Admin adheres to the contact of User to ensure that subclass Admin does not override its superclass in unintended ways. For example, Admin should not modify .setCurrentMatches() and getId() from the User superclass.

Interface segregation principle

Vacuously satisfied since we don't have any interfaces.

Dependency inversion principle

CourseRepository implements the JpaRepository interface, which is used by some higher level service to interact with the course database. This implementation is necessary for Spring to auto-generate our repository code. This means that Spring need not worry about the specific type of data the database is working with, so changing the Course class doesn't impact the repository itself.

Clean Architecture

The CRC was implemented with a clear design of entities, use cases, interface adapters, and frameworks/drivers. In the center, the entities are separated from the other layers and do not have any knowledge of any of the other classes. For example, the user class has no information about the UserDatabase or the UserManager. ControlSystem, an interface adapter, takes the input and requests the UserManager, a Use Case, to make a new User, Profile and one or more Course objects, all of which are entities. The UserManager will also communicate with the UserDatabase to log in the new User. UserManager is a Use Case and UserDatabase is a database, so UserManager depending on UserDatabase is a violation of Clean Architecture because an inner layer is depending on an outer layer. However, this violation is an exception because UserManager has to interact with the database to add and edit users. In lecture, it was mentioned that this can be accepted. Similarly, once a day, the ControlSystem will request a list of User objects and send it to the Matcher, another use case. The Matcher will generate a hashmap of matches, based on each User's courses and interests, and return it back to

ControlSystem. The match hashmap has a key of a User object and the value would be that user's list of matches. This match hashmap is then sent to UserManager by ControlSystem. Lastly, UserManager takes the match hashmap and allocates the list of matches to each User.

The Dependency Rule is followed throughout the code except for the violation mentioned above. A concrete example is the interaction with the UI. The UI only displays information and gets input from the user. After that, the ControlSystem gets the input from the UI and communicates with the Use Cases to carry out functions.

Scenario Walkthrough Summary:

If the user creates a new profile, the UI will receive input from the user which includes name, year of study, courses and contact information and Control System requests the UserManager to make a new User, Profile and one or more Course object. Then, the UserManager stores the Course object(s) in Profile object, which is stored in User. UserManager will communicate with UserDatabase so that the new User is logged. Next, ControlSystem will request a list of User objects and send it to Matcher. The Matcher will generate a hashmap of matches, called match hashmap based on each User's courses and interests, and return it back to ControlSystem. The match hashmap has a key of a User object and the value would be that user's list of matches. This match hashmap is then forwarded to UserManager by the Controller. Lastly, UserManager takes the match hashmap and allocates the list of matches to each User.

If the user logs in, the UI receives the User ID as the input and ControlSystem requests UserManager to get the user using the UserID. If the User exists, they can edit their profile. The UI will receive the new information from the user, send it to ControlSystem, which will request the UserManager to change the information in the user's profile in Profile.

Packaging

For our packaging strategy, we chose to organize our classes according to the components they corresponded to. We chose to package it this way as it would help us in phase 2 when we add the Report classes and Chat functionality. They can be their own packages and not modify any of our current organization.

The component strategy was more intuitive as the relationships between classes were more prominent. Whereas in the layer packaging strategy, these wouldn't be as apparent and harder to notice.

However, since we are using Spring as a framework, we may modify our packaging strategy into a hybrid of the layer and component strategy.

Design Patterns

Factory Design Pattern:

The factory design pattern is implemented in our project for the classes UserFactory, ProfileFactory and CourseSet Factory. Each of these classes has only 1 method, which is to create and return User, Profile and a set of Courses, respectively. This reduces the responsibility of UserManager. When we add a new class Admin, we can add an interface that has "createUser()" and create and AdminFactory that implements the interface, so both UserFactory and AdminFactory implement the same interface and create Users or admins

The factory design pattern can also be implemented in our project for the abstract class Report. We currently have two kinds of Reports – HarassmentReport and DistressReport. Both take slightly different information and return different reports. We can create the interface Report with the method "createReport()" and leave the instantiation up to the subclasses.

Now, the HarassmentReport will implement the method "createReport()" and DistressReport will implement the method "createReport()". Both will return new HarassmentReport() and DistressReport() objects (or products) respectively.

This design pattern is particularly helpful in this case as we can create different types of reports in the future, say ReportUserProfile and not have to worry about changing the implementation of Report. We can create a new subclass and define ReportUserProfile individually.

Memento Design Pattern:

Currently, we have not implemented Memento as we plan to deploy that for the Chat class in Phase2, but this design pattern is not feasible to implement in Phase1. In Phase2, the users are able to communicate through the chat and the Chat class stores a collection of messages. If, by accident, a user wants to unsend a message, then a Memento can help restore the previous state of the chat conversation. This is useful to implement because in our project, we want to give the user control the ability to 'undo'. In Phase2, to implement the Memento design pattern for the Chat class, we will need to create the Originator, Memento, and Caretaker objects. The Originator should create a Memento class and use the Memento to capture the current chat state. For example, the originator can have functions such as restoreMessage, createMessagestate. The Memento can have functions such as getMessage, getMessagedate. Then, Caretaker can store all Memento objects in an array so that it can execute rollback to the Originator using saved states from the Memento. The Caretaker may have functions such as Chat.getMessage(), History.add(createMessagestate). In other words, we can keep track of a 'stack' of states and activate the rollback feature when needed.

Decorator Design Pattern:

In the project, currently we have an entity class called Message. It represents a single message in a chat. Each instance of Message contains information about the content of the message, who sent the message and the time/date of the message. This message needs to be represented in our GUI somehow. Here, we can use the decorator pattern. We can have MessageGUI as an interface with a method signature draw(). Message can implement MessageGUI and implement the draw method and provide a basic transparent box with the content of the message displayed in black inside the box. In addition, we can have multiple decorators such as MessageBorder and MessageFillColour that can wrap around an instance of Message to provide additional functionalities to the instance at runtime.

These functionalities could include creating a border around the basic white box in the chat or to change the fill colour of the box. The way this would be programmed is that the decorators like MessageBorder and MessageFillColour would be subclasses to the BaseDecorator that implements MessageGUI. The BaseDecorator would take an instance of Message as a parameter. In this way, we can add additional display features to each instance of Message at runtime without changing the class Message itself. This will allow easy customisation of the aesthetics of messages. Each user of our app can thus change how their messages look in their chats in their app as per their wishes.

Progress Report

Thus far, our group has diligently worked on implementing phase 1 of this project. The app was implemented as a webapp using Spring. Adrian used the designs from Phase 0 to create the webapp structure using HTML, CSS, and JavaScript. The backend team (Tony, Akshat, Rue and Dien) built upon the existing skeleton code in coding the additional classes that were not implemented before and implemented the database. Lawrence helped to create the design document and UML.

Summary of each member's progress

Adrian:

For Phase 1, I was responsible for front-end development and wrote some test cases. I designed all the screens for the webapp, such as the welcome page, create account & sign-in page, user's account page, socials page, and finally the matching screens. To present to the user functionalities that we wanted to implement, such as the matching algorithm and log-in account functions, we decided that a webapp is better than making a mobile app we originally planned in Phase0. I enjoy the simple mobile app design from Phase0 and did my best to extend that theme to the webapp. I also wrote test cases for entity User and Course; made local branches to edit code, used pull requests, raised issues, and approved pull requests to work with the team. One aspect of the front-end design that has worked well is the ease of use for the user and code organization. Connecting frontend code to the backend is still work in progress and will take a bit more time, but much of it has been completed.

Dien:

I built upon the existing skeleton code by adding methods to allow users to edit their profile. I also implemented the Factory design pattern after discussing with the team what would be a suitable design pattern to use. Creating the class diagrams helped me understand the structure of our whole project and how the classes interact with each other. This made implementing the Factory design pattern easier. The new Factory classes made the code cleaner and easier to fix if there were any bugs in the process of creating a User. It also lessens the responsibility of UserManager. I wrote test cases for the Factory classes, however I wasn't able to write tests for methods that edit profiles in UserManager because we haven't finished implementing the database. Rue and I tried to link Spring Boot with HTML using Thymeleaf so Tony had to help us. While writing methods to create Users and Profiles in UserManager, Adrian and I did not communicate, so the information to be input in the Java class did not match with the information to be input in the UI. In Phase 2, I will communicate with frontend more and make sure the whole team finalizes each screen.

Akshat:

I along with Tony set up the Microsoft Azure server which currently hosts our entire codebase and also runs our web application. In addition, I set up the Azure Database for MySQL which is meant to replace our text database from Phase 0. Currently, it is able to store courses that are inputted as part of registering one's courses. While it is unable to store Users and Profiles at the moment, this is a good proof-of-concept that showcases the functionality of the database as well as the interaction between the frontend and the database. After consulting with Rue, Dien and Tony, I repackaged all the java classes into components to have a more intuitive packaging strategy rather than just having all classes in one package.

Rue:

Continuing with our designs from Phase 0, I helped Adrian modify the mobile designs into a web app design. I was involved in the deciding what design pattern to use, the implementation of the design pattern, packaging strategy and the MySQL database. I also helped test the UserFactory and CourseSetFactory and refractor our code to include documentation. I worked with Dien and Tony to link SpringBoot and Thymeleaf. I also helped create the design document. I will continue to help the backend team clean up the implementation of the database and refractor the code.

Tony:
I worked with Akshat to set up the Microsoft Azure server. I also worked with the Spring framework and SpringBoot and attempted to integrate the front- and backend. Specifically, I modified the HTML files by adding Thymeleaf (the template

engine of our choice) annotations and wrote and annotated Java classes used by Spring. However, the complexity of our project means that there are many unfinished parts. I list some of them here:

- The majority of the frontend is still disconnected from the backend.
- Too much is hard coded into the HTML right now (e.g. variable names, etc.). This needs to be fixed by introducing application properties.
- The HTML is still messy, primarily because it is filled with Webflow related markups that we don't need in our project.
- Code for validating forms is not done.

etc.
I will continue working on Spring for project phase 2 and hopefully address some of the leftover problems from this phase. Also, like I have done always, I will continue to help my teammates solve problems.

Lawrence:

I worked on frontend with Adrian in the development of the webapp. I created the UML for the project using graphic design. I also worked on this design document and wrote some test cases for the back end development. I assisted others with issues that had arisen as well. Overall, I believe that the group is working well together and we have a solid plan ahead of us.

Open questions your group is struggling with
We are currently trying to decide how to store our data in our database and ensure we're following Clean Architecture as we do so.
We are also trying to figure out how to implement the Chat functionality.

What has worked well so far with your design?
Entity and use case classes follow Clean Architecture and work well with each other teo create and edit users and user profiles. We have refactored many parts of the design, such as adding UserFactory without having to change any entities (User, Profile, Course) or ControlSystem.
We used Springboot's decorators and simplified our code a lot for the azure database. We did not need to write MySQL commands since the decorators automatically handled that for us.

Use of GitHub Features

We used the following github features:

- Issues
  - We used issues to keep track of pending to-dos and bugs as we came across them.
- Actions
  - We used actions in Github Workflows to run the tests every time a commit onto main or pull request to a main was made.
  - It also compiled the code each time and helped us identify compilation errors.
- Pull Requests
  - We used pull requests to ensure code reviews were conducted prior to pushing to main.
  - We also used pull requests to highlight commits that used a design pattern or refactoring. For example,
    - We modified UserManager and introduced User, Profile and CourseSet Factory.
    - We also added documentation for most of phase 0 code.