Design Document

Group13-Utalk

Hanqi Zhang, Chuanyang Qiao, Zhijun Wang, Zeyuan Li, Kuang Jiang, Junhao Saw

Specification

Our program is dedicated to providing a platform for faculty to post and update course information, for students and professors to share thoughts and ideas about a specific course.

All three types of users should go through the registration process to access this program. Users will need to decide their usernames, passwords and clarify their identities (Faculty/Student/Professor).

We also have administrator privileges at the main user interface. Once you type in the correct developer password(ADMINISTRATOR), the administrator can check the information of all users and delete all the user information.

We have different functionalities for different identities. After logging in, the Faculty can create a new course with the course code, course description and start the semester. For those courses which already exist, faculty can still modify their course information and update the new semester.

Student and Professor have similar functionalities. Both of them can enroll in courses based on the course code. After enrolling, they can view the course page where the course code, course information and several other information related to that course are displayed. Also, they can view the comment page of the selected semester of that course. For each comment, it has the unique comment id, content and user name.

On the comment page, Students and professors can create a new comment or reply to an existing comment by comment id. They can also edit or delete the comment based on that id. The student can only modify the comment made by themselves but the professor has the authority to delete any comment.

List of functionalities

- user sign in (student / professor / faculty)
- log in as (student / professor / faculty)
- faculty can add courses and update courses
- The professor can add course and enrol to the course he/she taught and see all the comment they made
- they can add and reply to comment also they are able to delete any comment on that course page
- Students enter course code to enrol and drop the course and see all the comments they made
- they can add and reply to comment, they are able to delete any comment made by themselves.

<u>Additional features we implemented for phase 2:</u>

1. Reading the csvs and restore all the states of java objects.
2. Writing all the information in java storage to corresponding csv files.
3. Implemented an interface for Student and Professor, as well as updating the one for Faculty.
4. Implemented CommentUI.
5. Implemented the presenter class to present Coursepage and Postpage

Major decisions

- Embedded template design pattern into project provided by Junhao, and make the Student and Professor inherit the same abstract class Commentable User. This strategy saves us from writing duplicate codes. A getClassString method provides a convenient access to the type of CommentableUser (i.e. Student or Professor).

- For data storage, most of our information regarding action made is stored in csv files. We have java storage classes when we are running the program, and have csv files after the program ends. Qiao wrote CSVReader and CSVWriter to read and write Arraylist<Arraylist<String>> objects to csv files with a certain path. The Arraylist<Arraylist<String>> objects are obtained by calling the record methods in java storage. We choose this because it is easy to read and write lists into csvs, therefore we just adjust other parts for it.

- We choose to restore the state of the program as soon as the program begins so that we can avoid duplicate readings.

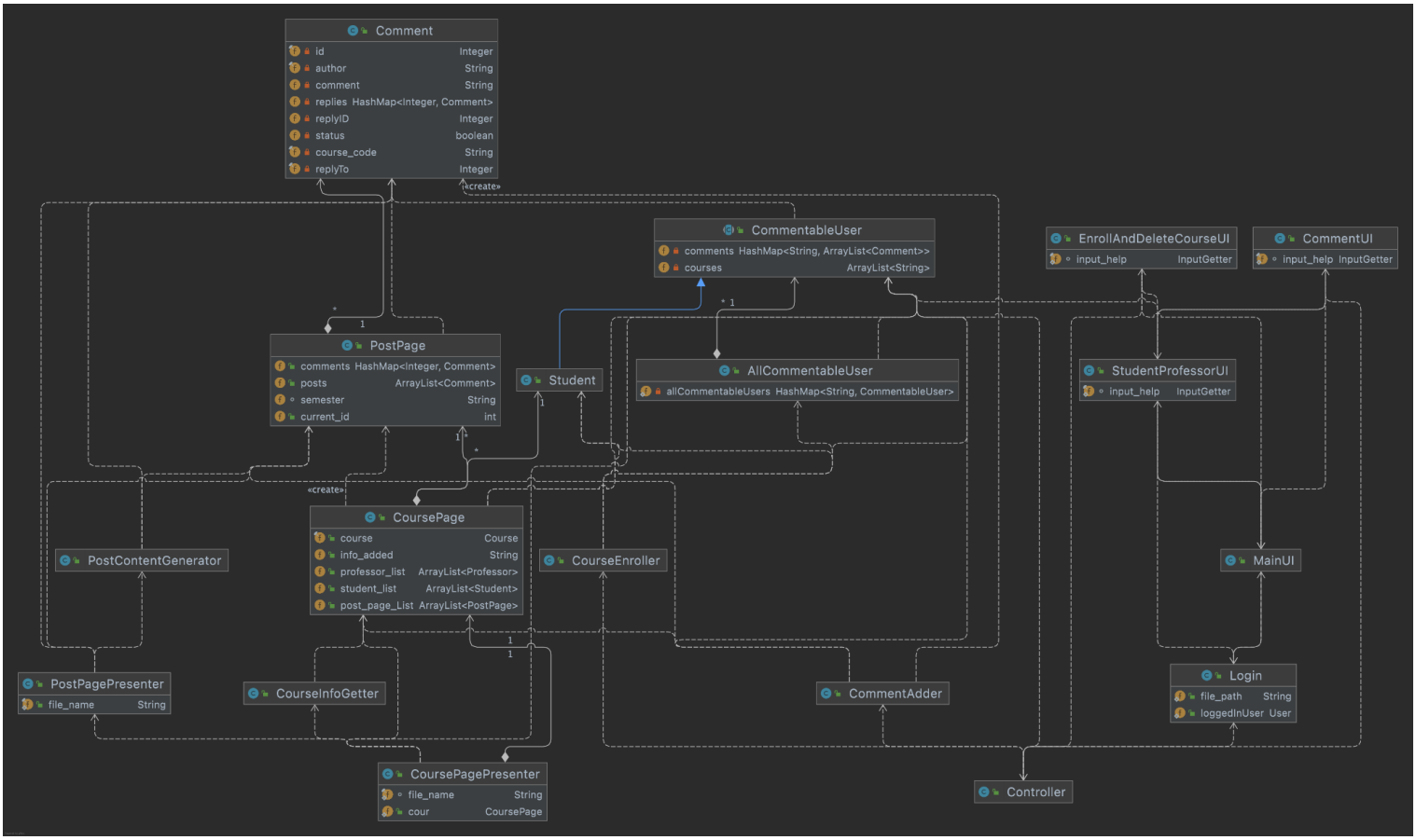
- Controllers help us call on relevant methods instead of a bunch of if else statements in UIs.

- For comment class, we discussed a lot of its data structure to ensure program efficiency. Each comment object has an attribute reply which is a hashmap containing all replies. The whole structure is similar to a tree, which enables high running efficiency.

Clean Architecture

We followed strictly to the clean architecture and dependency rule. The classes in out layers are dependant on which in inner layers.

Class diagram for main function:



For example, we can see the four layers here in the class diagram.

- outer layer: CommentUI, Enroll and Delete Course IU, StudentProfessorUI, MainUI
- interface adaptors: Controller, Presenter(CoursePagePresenter, PostPagePresenter)
- use case: CourseEnroller, CommentAdder, CourseInfoGetter, PostContentGenerator, AllCommentableUser.
- entity: Comment, PostPage, CommentableUser, Student, CoursePage

scenario walkthrough

Suppose you are a student and you want to enrol in csc207. First you need to Login. The MainUI in the outer layer would take you to the Login function which ask you for your name and password and match your information with information stored in inner layers(javaStorage in useCase).

After you have logged in, the MainUI will direct you to the student professor UI in the same layer, which then calls the presenter in interface adapter to print all courses that you have enrolled in and ask you what action you want to take. After you choose to enroll in the course, you will be directed by the student professor UI to the Enroll and Delete Course iu, which then calls the Controller in interface adapter layer to enroll you. The controller would then call use case courseEnroller to enroll you in based on the course code you typed.

The courseEnroller updates the attribute: studentList of entity course page and updates attributes courses and attribute comments of commentableUser in entity layer.

After that the Use Interface will ask you what other actions you want to make. If you type in that you want to add a comment, the student UI would ask you for the course code and lead you to the comment UI. The comment UI ask presenter to print all comments now for specific course and ask you for the information about your comment(basically the content and the comment id you want to reply to) and pass it to controller, the controller call CommentAdder in use case layer which change the commentable and postpage in entity layer.

Every time we start our program the Allreader reads from our database and stores the present information about user, comment and courses, all information will be read to the entity and java. And every time we end our program the writeall function in DatabaseWriter in the interface adapter will write everything into the csvs.

Solid Principle

We did try hard to follow the SOLID principle. I will discuss each principle separately.

1. Single Responsibility Principle
This principle states that “A class should have just one reason to change.” Our program is basically consistent with this principle. For instance, in phase one, most of our classes were super big and contained many functionalities. After we read the single responsibility principle, we separated most of our classes so that one class is facing one functionality. For instance, we used to put all the methods associated with one class inside the class, causing large chunks of code. We now divided the functions into smaller classes responsible for different functionalities such as Comment Adder and Comment Editer.

2. Open/closed Principle.
This principle states that “Classes should be open for extension but closed for modification.” One example is the CommentableUser class. We let the Student and Professor classes extend them, while including only common parts in the CommentableUser superclass. This makes the class closed for modification as the CommentableUser only includes methods that a user who can make comments can do, while it is open for extension, where we can add a third class called TA and let it inherits CommentableUser as well.

3. Liskov Substitution principle.
This principle states that “When extending a class, remember that you should be able to pass objects of the subclass in place of objects of the parent class without breaking the client code”. One example that we followed this principle is the CommentableUser and its subclasses. CommentableUser can delete its own comment, while both Student and Professor can delete comment methods, with Professor also able to delete others’ comment. There is no conflicts between the methods in superclasses and subclasses, which adheres to the Liskov Substitution principle.

4. Interface segregation principle.
This principle states that “Clients shouldn’t be forced to depend on methods they do not use.” In our program we have a parent class user and a commentableUser class extends it. We can notice that the CommentableUser class has some special methods such as getcomment, getCourses, which are needed by any users who can make comments. We decided to write these methods in commentableUser instead of writing it in User because faculty (subclass of user) do not use these methods.

5. Dependency inversion principle.
This principle states that High-level modules should not depend on low-level modules. Both should depend on abstractions. And abstractions should not depend on details. Details should depend on abstractions. As was mentioned before, we make most superclasses abstract. Take presenter class as an example, presenter use entity transfer package to load information from entity and this does not violate dependency inversion principle.

Packaging Strategies

1.The first layer of package is edu, just included here to follow packaging conventions.

2.The second layer of packages are entity; usecase; interfaceadaptor; outerlayer; runner, which are grouped by clean architecture layers.
This helps us find the functions easier by first looking at the layers. For example, for an action like enroll course, we will just search for relevant methods in the usecase layer first.

3.The third layer packages are functional packages such as boundaries, controllers, createupdate, which group the classes based on there functionality.

This makes it easier to find methods with similar properties after locating the correct layer. For example, you can easily find methods associated with faculty(create and update courses) in the createandupdate folder.
(We used lower case letter without any separation mark for the package name based on this StackOverflow Question:
https://stackoverflow.com/questions/49890803/naming-conventions-of-composed-package-names)

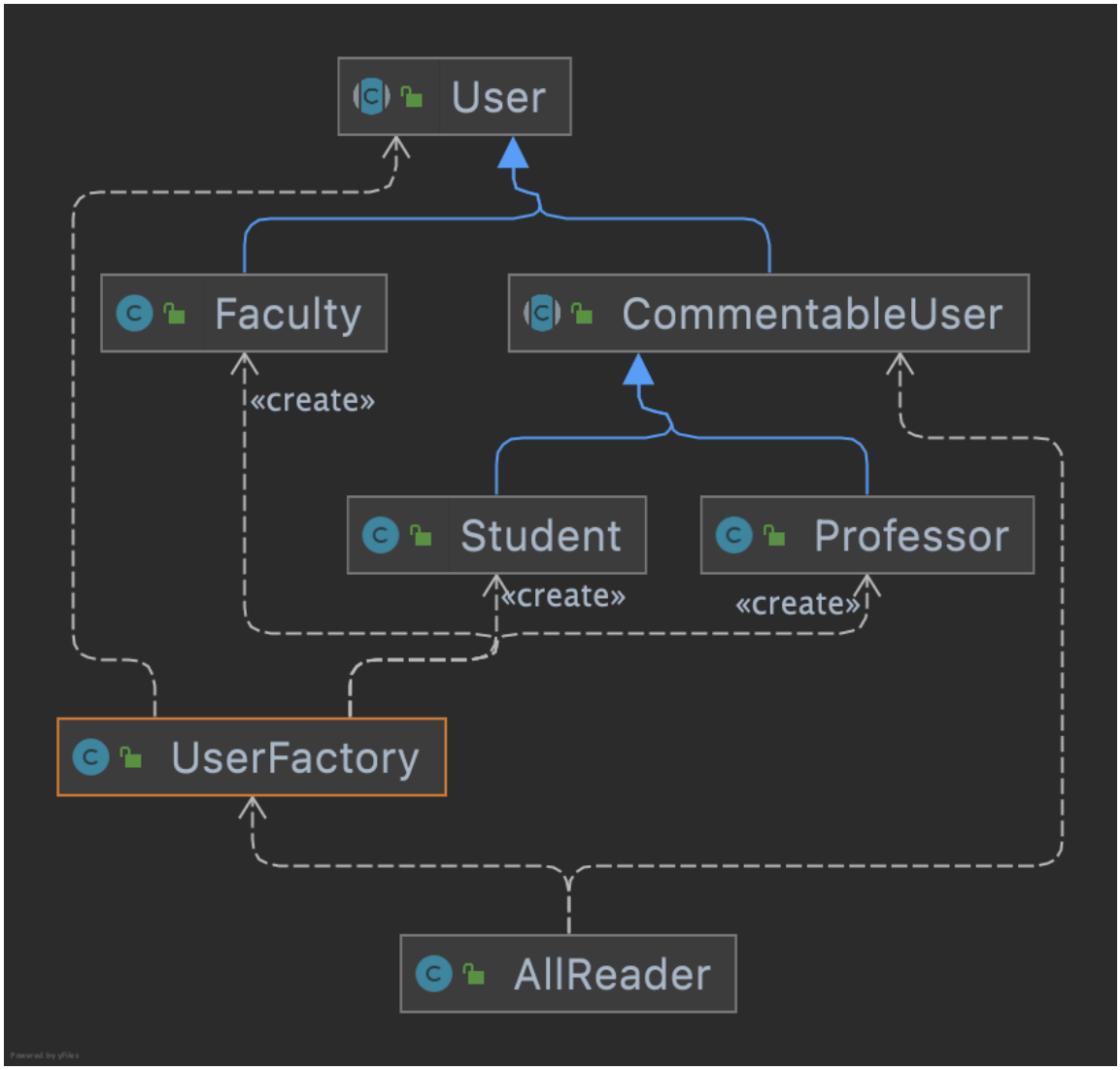
Refactoring

For phase 2, we extracted the methods from entity classes so that they will not have long method lists, which fixed the previous large class code smell. However, some classes like MainUI still have long methods due to the usage of switch statements and lots of printing statements, which may be fixed by extracting each case (e.g., “1”) into different classes. It has not been done due to time constraints.

We also repackaged our codes so that they will be grouped in a meaningful way by layers and functionalities.

We also implemented the template design pattern and simple factory pattern to deal with the duplicate code code smell and remove hard dependencies. The CommentableUser class simplified our codes a lot while the UserFactory simplified the reading process.

Design Pattern
Simple Factory Design Pattern

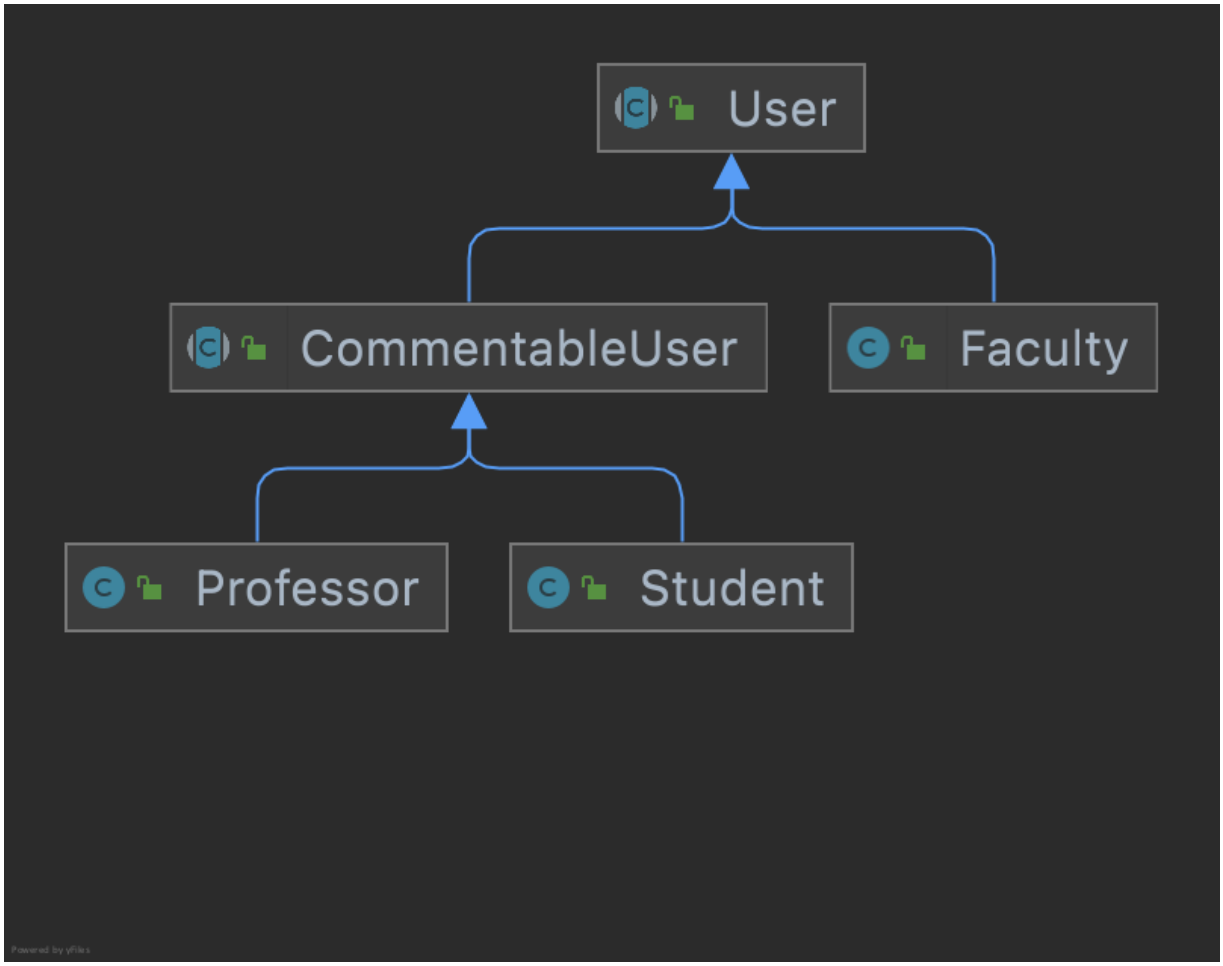


The simple Factory class is included to help create instances of different subtypes of User (Faculty, Student, Professor), which removes the hard dependency for those classes in the AllReader codes.

It also follows clean architecture so AllReader in the interface adapter layer will not directly depend on the User class.

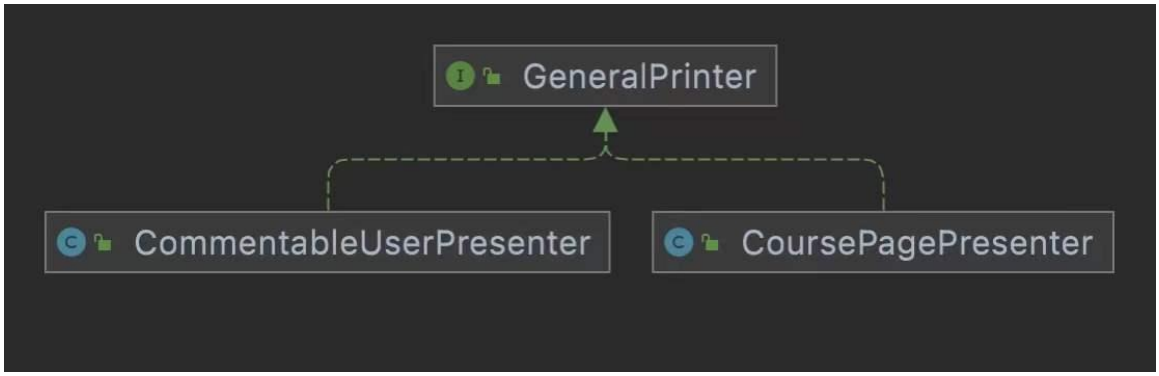
Associated pull request:
https://github.com/CSC207-UofT/course-project-utalk/pull/97

Template Design Pattern



The Template Method Design Pattern is used here as two of our classes, Student and Professor, share a lot of functionalities except for minor variations in some of the steps.

For example, they all have add comments, reply to comments, enroll course and drop course methods, which have exactly the same implementations. For functionalities like deleting comments, they also share some codes in common, varying a little in common. Therefore, this kind of situation is suitable for a template design pattern so we create a common superclass CommentableUser for Professor and Student classes. By using this pattern, we do not need to duplicate our code.



In addition, for the Presenter package, we include factory design patterns for the presenter class. Since two classes are the core of the present package, factory design patterns restrict input and protect functionalities of classes. Frankly, this pattern is not versatile as we didn't need various present functions, but it will be useful as we implement other types of present functions later.

Testing

We wrote test cases for each use case class as well as readers and writers in the interface adaptor layers. We also add JUnit test for usecase

As most of the methods are extracted from the entity classes, we do not have many methods to test in the entity.

We also tried to test UIs to fix bugs through running them and hand code input. It is hard to do assertions for UI testing as they take in input from the command line.

Use of Github Features

We tried our best to follow the guideline of Github usage and Phase1 feedback this time. It helps us increase working efficiency during the Phase 2 process.

Use of Pull Request & Branches

For the push process, we first commit changes to our branches (most of them named by functionalities). Then, we named our pull request with a descriptive title such as add Course Enrol method. Also, we would add a comment to briefly describe the new content in this version. We will request our teammates to fully check our code. If teammates find some bugs, we will figure them out in the comments and solve them before pushing to the main.

Use of Issues

Moreover, this time we also made good use of Issues on Github. We divided the whole project into several issues and assigned them to the teammates in charge. Once we have finished them, we will close the relevant issues. If we encounter some problems during the process, which cannot be solved by ourselves. We will add new issues, briefly describe the problem and talk about it in group meetings.

Project Accessibility report

Principles of Universal Design

Principle 1: Equitable Use

- All users have the same level of security and privacy
- Our program is a command line interface and is visually neutral
- Different users have to have different privileges as there is a hierarchy in our target demographic

Principle 2: Flexibility in Use

- Our program is a command line interface so it is accessible to both left and right handed users
- The inputs required by the program are quite short, usually only a single character

Principle 3: Simple and Intuitive Use

- Our program will output effective feedback during and after task completion i.e. when adding a comment, the user will see a messaging saying the comment has been added successfully
- The level of language in our program is fairly low, reducing complexity

Principle 4: Perceptible Information

- Our program is a command line interface so essential information is legible
- We could extend the program by adding text-to-speech features for users with visual impairment

Principle 5: Tolerance for Error

- The program outputs messages that indicate when the user has inputted an incorrect variable or command to the command line interface

Principle 6: Low Physical Effort

- The inputs required by the program are quite short, usually only a single character
- The user does not have to make repetitive actions or exert physical force

Principle 7: Size and Space for Approach and Use

- It is easy to use our program whether the user is standing or sitting
- We could extend our program to accept speech input for users who struggle to type
- We could also add different coloring schemes to our program e.g. high contrast/color blind mode

Marketing the program

- If we were to market the program, we would mostly likely target institutions because this demographic is likely to have the hierarchical structure outlined by our program.
- Since students have lots of questions about course content, they are more likely to be adding comments and viewing previous questions asked by their seniors

Project usage by other demographics

- Our program could be used by other demographics with hierarchical structure that require an interface to communicate and have discussions with
- It is unlikely that the visually impaired are to use are program as it is displayed on a screen

Progress Report

Hanqi Zhang's contribution:

1. Implemented the addCourse and deleteCourse methods and modify their structure to separate the methods from the Faculty entity class.
2. Implemented the FacultyUI and StudentProfessorUI, and link the methods and UIs together using Controller
3. Implemented the record methods for AllCourses and AllCommentableUsers, so all relevant information will be recorded into Arraylists and written into csvs through CSVwriter.
4. Implemented the read methods for reading courses and commentable users in.
5. Repackaging all the classes and add a FilePathHelper to direct all csvs to the database folder.

6. Github link:

https://github.com/CSC207-UofT/course-project-utalk/pull/18/files

Provided a new structure with proper packaging to make it easy to read and access different functionalities in different layers.

https://github.com/CSC207-UofT/course-project-utalk/pull/85/files

Implemented the readers and writers using the CommentableUser instead of separately for Professor and Student class, which makes it in line with the implementation of Template Design Pattern for common functionalities for Student and Professor class. It simplified the code as well as storages.

Chuanyang Qiao's contribution:

1. Implemented User Register and Register UI. Register a user (get from command line) and record this user into a csv file
2. Implemented Log in and Log in UI. Log in as a specific user (get from the command line) and return this user. Log in process will be recorded into a CSV file.
3. Implemented Log out and Log out UI. Log out a specific user (get from the command line) . Log out process will be recorded into a CSV file.
4. Implemented MainUI. The main interface of our program.
5. Implemented CSV Writer, CSV Reader, and csv_list transfer. These classes are controllers and presenters that link our database and java program.
6. Implemented administrator privileges. Only a developer with a valid password can enter. a) check all users that registered before. b) delete the whole database.
7. github pull request link:
   This is one of significant pull requests related to Main UI and login/logout functionality.
   https://github.com/CSC207-UofT/course-project-utalk/pull/31

Zeyuan Li

1. finish all the Comment related methods, canAccesscomment, canDeleteComment, canAddComent, Addcomment, deleteComment, EditComment with structure of Junhao's template pattern
2. implement entity classes commentable user, comments with wang
3. implement the method read all comment read comment and postpage from csv, realize basic set up for comments and postpage
4. implement writeall method realize functionality of write comment information in csv
5. Implement CommentUI with interact with user for all comment related functionality, connected with profstudUI
6. Try turning test in Junit
7. Github link: https://github.com/CSC207-UofT/course-project-utalk/pull/66/files
   this is link show mainly comment related function how I read the postpage information and use design pattern

Kuang jiang:

1. Implement the CourseDropper and CourseEnroller functionalities for both student and professor, add test cases for both of the methods.
2. Implement the Test case for CourseDropper and CourseEnroller
3. Implement the EnrollAndDeleteCourseUI which is used as an interface to help students and professors modify their course list.
4. Implement test cases for all the usecase.
5. Implement javadoc for all the methods in usecase, outerlayer, interfaceadaptor
6. Implement StudentProfessorUI with Hanqi Zhang.
7. Github Link:
   https://github.com/CSC207-UofT/course-project-utalk/pull/43
   Implement the CourseDropper and CourseEnroller functionalities for both student and professor.
   https://github.com/CSC207-UofT/course-project-utalk/pull/78
   Implement StudentProfessorUI

Zhijun Wang:

1. Implement all classes in the Presenter and entityInfoTransfer package.
2. Using Junhao's idea and draft code to implement comment and commentable user's class with Zeyuan
3. Implement CoursePagePresenter UI, which includes a feature that shows PostPage for different semesters.
4. Using online resources to implement TextFileCreator and WriteIntoTxtFile class class, which stores course page, post page, and comment information into the database.
5. Revising and debugging comment related UIs.

6. Implement Factory design pattern for CoursePagePresenter and CommentableUserPresenter class, it will be more versatile if we have more present functions added in the future.
7. GithubLink (Implementation of entityInfoTransfer package that solving dependency inversion of presenter class, the previous version calls on entity functions and I move it to Usecase package)

Junhao Saw

1. Design commentable user class and comment class, using comment status attribute to convenient delete comment implementation
2. Provide draft code for comment and commentable user class. (Link)
3. Embedded template design pattern into our project.
4. Refactor others' work to satisfy basic principles.
5. Formalizing and contributing to reports for all Phases.
6. GitHub