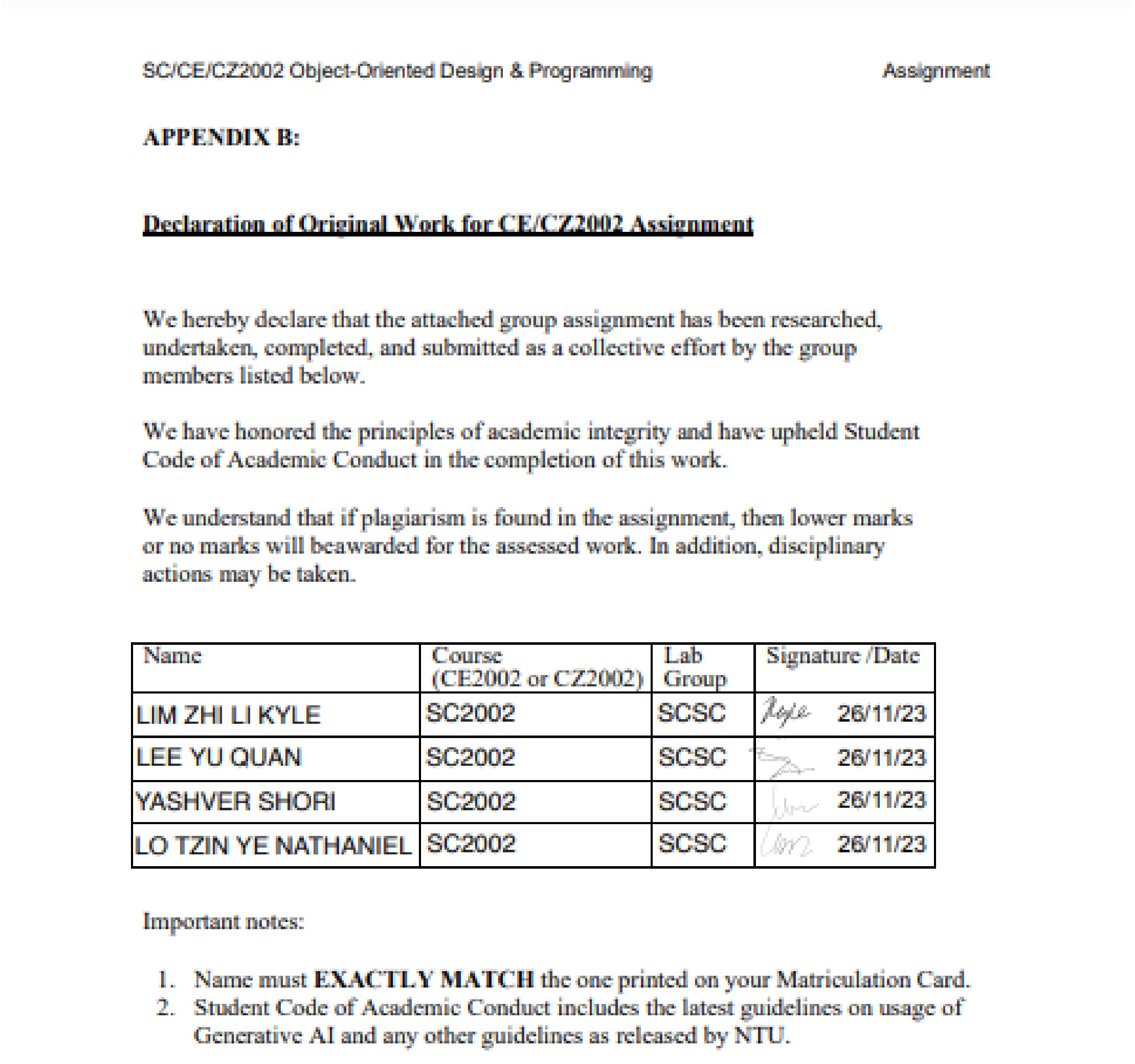
AY23/24 SC2002 Assignment Report

Group Members: Lim Zhi Li Kyle, Lee Yu Quan, Yashver Shori, Lo Tzin Ye Nathaniel

Tutorial Group: SCSC



SC/CE/CZ2002 Object-Oriented Design & Programming Assignment

APPENDIX B:

Declaration of Original Work for CE/CZ2002 Assignment

We hereby declare that the attached group assignment has been researched, undertaken, completed, and submitted as a collective effort by the group members listed below.

We have honored the principles of academic integrity and have upheld Student Code of Academic Conduct in the completion of this work.

We understand that if plagiarism is found in the assignment, then lower marks or no marks will beawarded for the assessed work. In addition, disciplinary actions may be taken.

Name	Course (CE2002 or CZ2002)	Lab Group	Signature /Date
LIM ZHI LI KYLE	SC2002	SCSC	[illegible] 26/11/23
LEE YU QUAN	SC2002	SCSC	[illegible] 26/11/23
YASHVER SHORI	SC2002	SCSC	[illegible] 26/11/23
LO TZIN YE NATHANIEL	SC2002	SCSC	[illegible] 26/11/23

Important notes:

1. Name must **EXACTLY MATCH** the one printed on your Matriculation Card.
2. Student Code of Academic Conduct includes the latest guidelines on usage of Generative AI and any other guidelines as released by NTU.

Camp Application Management System (CAMS) Design Overview

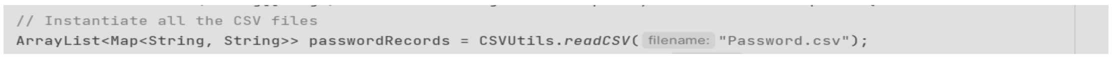
Step 1: Database Representation:

The foundation of our CAMS is a comprehensive database, structured into seven key CSV files. Each file serves a specific purpose:

- **UserCamp.csv:** This file outlines each camp member's responsibilities. Categories like UNINVOLVED, CREATOR, ATTENDEE, BLACKLISTED, and COMMITTEE are among them.
- **StudentList.csv:** In order to keep a comprehensive student profile, we collect here the necessary data about each student, including Name, NetworkID, and points.
- **StaffList.csv:** This file, which is devoted to staff members and contains their names, email addresses, and other pertinent information, is similar to the StudentList.
- **SuggestionList.csv:** This file is a compilation of all submitted suggestions
- **EnquiryList.csv**: All enquiries are systematically stored here, providing a clear record of questions and concerns raised by users.
- **Password.csv:** Essential for protection This file contains information about login authentication, such as user types, NetworkIDs, and hashed passwords.
- **CampList.csv:** This file contains the essential details about every camp, like Name, Start Date, and End Date.

Step 2: Initialising CSV and objects:

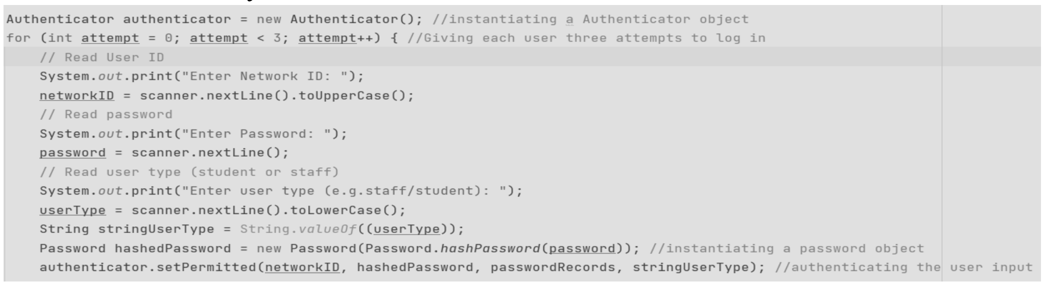
- Upon initialising the program, we instantiate the CSV files and convert their data into ArrayLists. These ArrayLists then form the basis for creating various objects ensuring a structured and efficient management system for all camp-related activities.Below is an example:



```
// Instantiate all the CSV files
ArrayList<Map<String, String>> passwordRecords = CSVUtils.readCSV( filename: "Password.csv");
```

Step 3 Authentication:

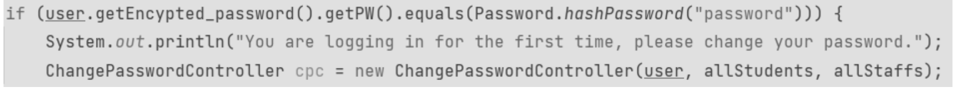
In the authentication step, we will prompt the user to enter the following inputs NetworkID, password and user type and it is authenticated by the Authenticator class.



```
Authenticator authenticator = new Authenticator(); //instantiating a Authenticator object
for (int attempt = 0; attempt < 3; attempt++) { //Giving each user three attempts to log in
    // Read User ID
    System.out.print("Enter Network ID: ");
    networkID = scanner.nextLine().toUpperCase();
    // Read password
    System.out.print("Enter Password: ");
    password = scanner.nextLine();
    // Read user type (student or staff)
    System.out.print("Enter user type (e.g.staff/student): ");
    userType = scanner.nextLine().toLowerCase();
    String stringUserType = String.valueOf((userType));
    Password hashedPassword = new Password(Password.hashPassword(password)); //instantiating a password object
    authenticator.setPermitted(networkID, hashedPassword, passwordRecords, stringUserType); //authenticating the user input
```

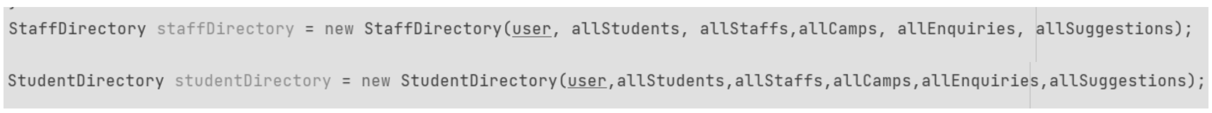
Step 4 Student/Staff directory:

After the user successfully login, we will obtain the specific user object from the ArrayList<Student> /ArrayList<Staff>. We check for first-time login and prompt the user to change his password.



```
if (user.getEncypted_password().getPW().equals(Password.hashPassword("password"))) {
    System.out.println("You are logging in for the first time, please change your password.");
    ChangePasswordController cpc = new ChangePasswordController(user, allStudents, allStaffs);
```

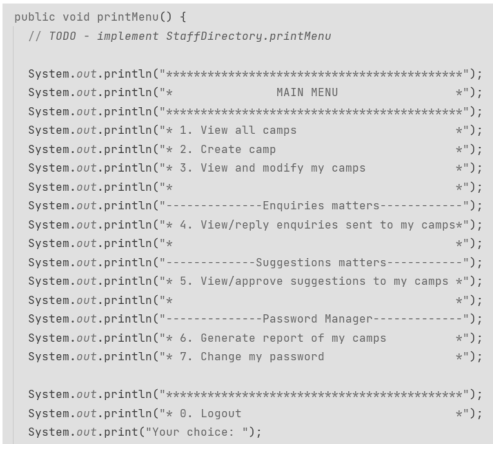
Based on the user type we will instantiate the student/staff directory.



```
StaffDirectory staffDirectory = new StaffDirectory(user, allStudents, allStaffs,allCamps, allEnquiries, allSuggestions);

StudentDirectory studentDirectory = new StudentDirectory(user,allStudents,allStaffs,allCamps,allEnquiries,allSuggestions);
```

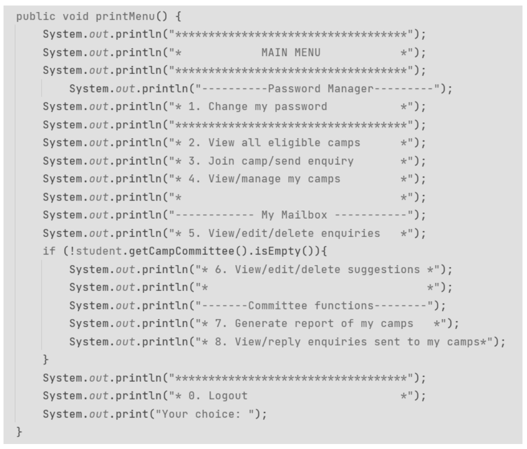
Step 5 Printing available functions menu and asking for input:

In the studentDirectory/staffDirectory constructor we will call the printmenu function in the respective class and hence we will not need to call the print menu method in the Main class.



```
public void printMenu() {
  // TODO - implement StaffDirectory.printMenu

  System.out.println("******************************************");
  System.out.println("*               MAIN MENU                *");
  System.out.println("******************************************");
  System.out.println("* 1. View all camps                      *");
  System.out.println("* 2. Create camp                         *");
  System.out.println("* 3. View and modify my camps            *");
  System.out.println("*                                        *");
  System.out.println("-------------Enquiries matters-----------");
  System.out.println("* 4. View/reply enquiries sent to my camps*");
  System.out.println("*                                        *");
  System.out.println("------------Suggestions matters----------");
  System.out.println("* 5. View/approve suggestions to my camps *");
  System.out.println("*                                        *");
  System.out.println("-------------Password Manager------------");
  System.out.println("* 6. Generate report of my camps         *");
  System.out.println("* 7. Change my password                  *");

  System.out.println("******************************************");
  System.out.println("* 0. Logout                              *");
  System.out.print("Your choice: ");
```

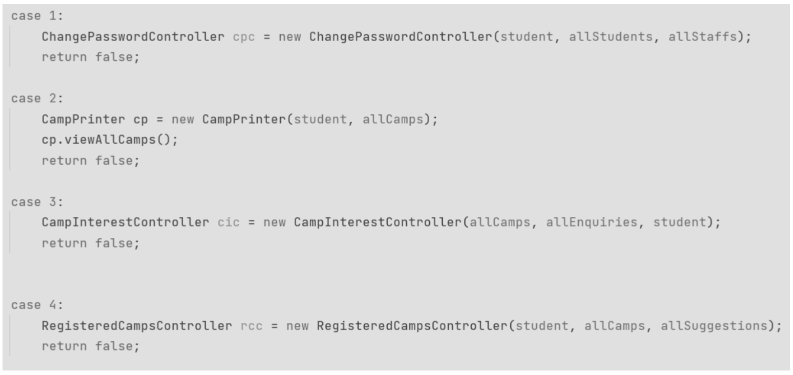
(Staff main menu)



```
public void printMenu() {
    System.out.println("***********************************");
    System.out.println("*           MAIN MENU             *");
    System.out.println("***********************************");
        System.out.println("---------Password Manager---------");
    System.out.println("* 1. Change my password           *");
    System.out.println("***********************************");
    System.out.println("* 2. View all eligible camps      *");
    System.out.println("* 3. Join camp/send enquiry       *");
    System.out.println("* 4. View/manage my camps         *");
    System.out.println("*                                 *");
    System.out.println("------------ My Mailbox -----------");
    System.out.println("* 5. View/edit/delete enquiries   *");
    if (!student.getCampCommittee().isEmpty()){
        System.out.println("* 6. View/edit/delete suggestions *");
        System.out.println("*                                 *");
        System.out.println("-------Committee functions-------");
        System.out.println("* 7. Generate report of my camps   *");
        System.out.println("* 8. View/reply enquiries sent to my camps*");
    }
    System.out.println("***********************************");
    System.out.println("* 0. Logout                       *");
    System.out.print("Your choice: ");
}
```

(Student main menu)

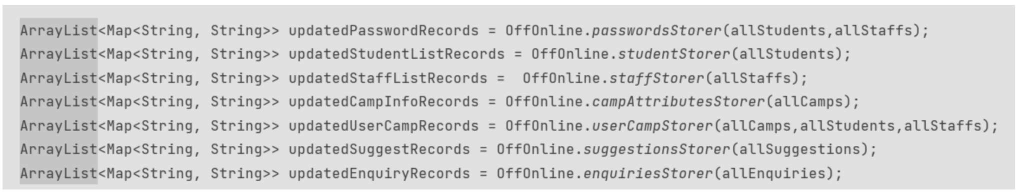
A switch case is applied for the user’s input. For each function listed, there is a delegated controller.



```
case 1:
    ChangePasswordController cpc = new ChangePasswordController(student, allStudents, allStaffs);
    return false;

case 2:
    CampPrinter cp = new CampPrinter(student, allCamps);
    cp.viewAllCamps();
    return false;

case 3:
    CampInterestController cic = new CampInterestController(allCamps, allEnquiries, student);
    return false;


case 4:
    RegisteredCampsController rcc = new RegisteredCampsController(student, allCamps, allSuggestions);
    return false;
```

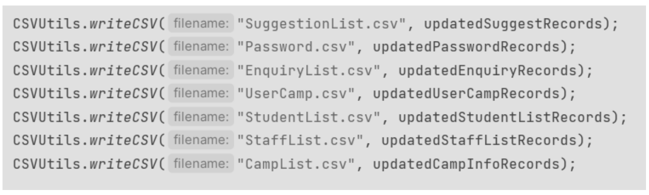
Step 6 updating of ArrayList of objects:

Since objects are called by reference, functions can permanently modify Arraylists. Using the ArrayList of objects as parameters we will call the various Storer functions in the OffOnline class to recreate the ArrayList<Map<String, String>> of various records.



```
ArrayList<Map<String, String>> updatedPasswordRecords = OffOnline.passwordsStorer(allStudents,allStaffs);
ArrayList<Map<String, String>> updatedStudentListRecords = OffOnline.studentStorer(allStudents);
ArrayList<Map<String, String>> updatedStaffListRecords =  OffOnline.staffStorer(allStaffs);
ArrayList<Map<String, String>> updatedCampInfoRecords = OffOnline.campAttributesStorer(allCamps);
ArrayList<Map<String, String>> updatedUserCampRecords = OffOnline.userCampStorer(allCamps,allStudents,allStaffs);
ArrayList<Map<String, String>> updatedSuggestRecords = OffOnline.suggestionsStorer(allSuggestions);
ArrayList<Map<String, String>> updatedEnquiryRecords = OffOnline.enquiriesStorer(allEnquiries);
```

Lastly, we will use all the records obtained from the ArrayList of objects and write back as the updated CSV using the static writeCSV method in the CSVUtils class.



```
CSVUtils.writeCSV( filename: "SuggestionList.csv", updatedSuggestRecords);
CSVUtils.writeCSV( filename: "Password.csv", updatedPasswordRecords);
CSVUtils.writeCSV( filename: "EnquiryList.csv", updatedEnquiryRecords);
CSVUtils.writeCSV( filename: "UserCamp.csv", updatedUserCampRecords);
CSVUtils.writeCSV( filename: "StudentList.csv", updatedStudentListRecords);
CSVUtils.writeCSV( filename: "StaffList.csv", updatedStaffListRecords);
CSVUtils.writeCSV( filename: "CampList.csv", updatedCampInfoRecords);
```

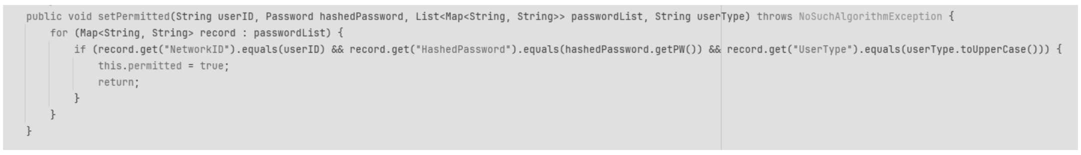
<u>Design Principle Consideration/Application:</u>

1) Single Responsibility Principle

The Single Responsibility Principle states that objects should not have more than one reason to change..)We separated our controllers to each perform only a unique function. For example, the View All Camps controller will only print all available camps, the Camps Manager controller will only be responsible to perform modifications to a staff's camp, within the Staff's available functions.

2) Open - Closed Principle

Our application would be open for an additional implementation should there be an additional user type without much change. Say for example that Camp Supervisor is an additional user that has authority to perform advisory duties, on top of replying to enquiries, no modification will be needed to deal with the addition. Instead, a separate controller to provide advice, and the current controller to reply to enquiries will need to be added to the Camp supervisor's directory class. In the case shown below, the authenticator class has a function setPermitted which only takes the general userType instead of the specific Staff or Student users and is thus flexible for more user types.



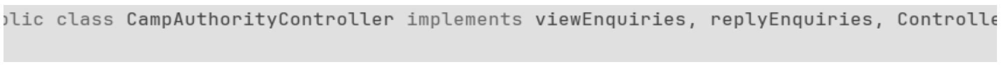
```
public void setPermitted(String userID, Password hashedPassword, List<Map<String, String>> passwordList, String userType) throws NoSuchAlgorithmException {
    for (Map<String, String> record : passwordList) {
        if (record.get("NetworkID").equals(userID) && record.get("HashedPassword").equals(hashedPassword.getPW()) && record.get("UserType").equals(userType.toUpperCase())) {
            this.permitted = true;
            return;
        }
    }
}
```

3) Liskov Substitution Principle.

Subclasses should be substitutable for the classes from which they were derived from. One instance of LSP can be seen in the Student, Staff and User class. The Student and Staff are subclasses whereas the parent class is the User class. The Student and Staff class can perform methods of the parent class without causing any runtime error. In our application. The User class has the general important methods and attributes that are needed in both student and staff classes, example: getFaculty, getNetworkID,getencryptedPassword. All methods were made public and inherited by both Student and Staff so that the subclass will be able to do everything the parent class can.

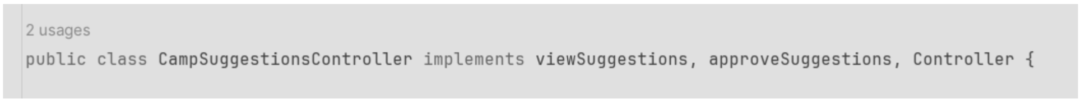
4) Interface Segregation Principle

Our Program implements multiple interfaces that each have specific functions. Different controllers that correspond to different functionalities and authorities implement different interfaces. For example, the class CampAuthorityController implements interfaces like viewEnquiries and replyEnquiries:



```
blic class CampAuthorityController implements viewEnquiries, replyEnquiries, Controlle
```

Similarly, the authority to view and approve suggestions is only given to the staff in charge of the camp, Hence, the class that implements the interfaces of viewSuggestions and approveSuggestions is the CampSuggestionController.. The previous controller of CampAuthorityController is not proposed to handle suggestions, hence the CampAuthorityController does not implement the viewSuggestions and approveSuggestions interfaces.

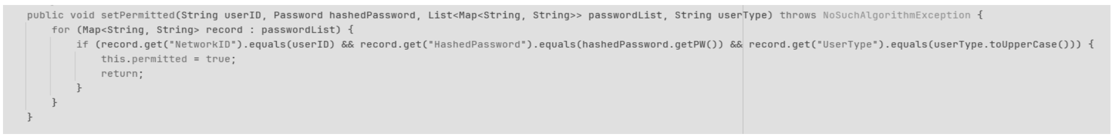


```
2 usages
public class CampSuggestionsController implements viewSuggestions, approveSuggestions, Controller {
```

In this example, both these example classes adhere to the Interface Segregation Principle because they implement interfaces with cohesive methods that are relevant to their functionalities and there is no unnecessary dependency on methods that are not used by the classes.

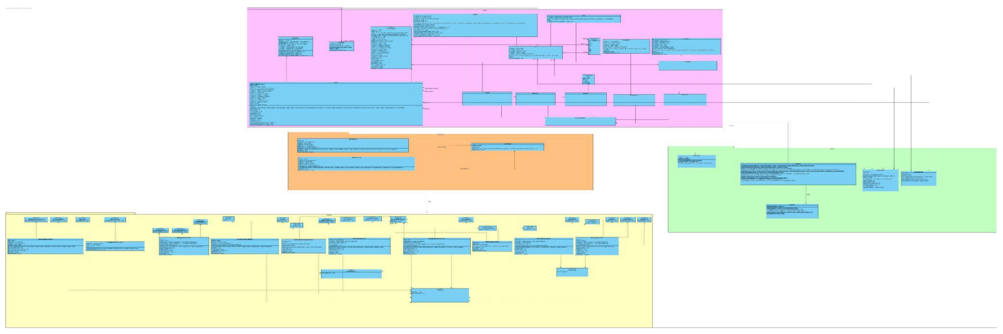
5) Dependency Inversion Principle

For most non-role based controllers, we pass in the User object as a parameter such that the controllers only depend on the User object and not its children. Another example would be the Authenticator class, once again it is not dependent on the staff and student class but only the user class.This makes it so that the Authenticator class can work with any usertype created as long as the information aligns



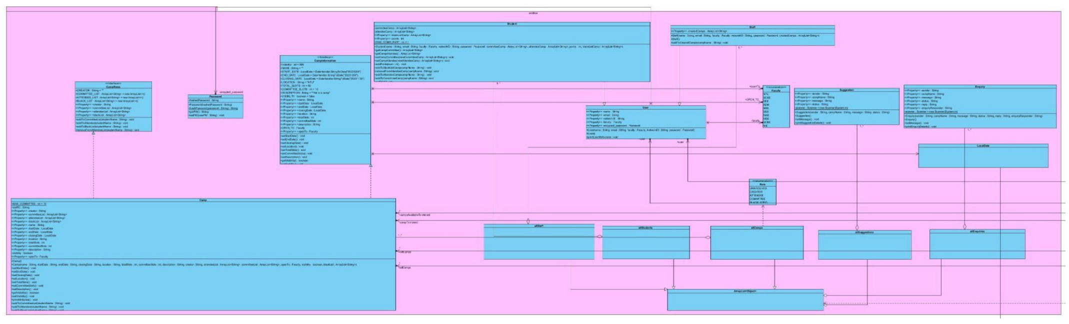
```
public void setPermitted(String userID, Password hashedPassword, List<Map<String, String>> passwordList, String userType) throws NoSuchAlgorithmException {
    for (Map<String, String> record : passwordList) {
        if (record.get("NetworkID").equals(userID) && record.get("HashedPassword").equals(hashedPassword.getPW()) && record.get("UserType").equals(userType.toUpperCase())) {
            this.permitted = true;
            return;
        }
    }
}
```

<u>Our UML</u>



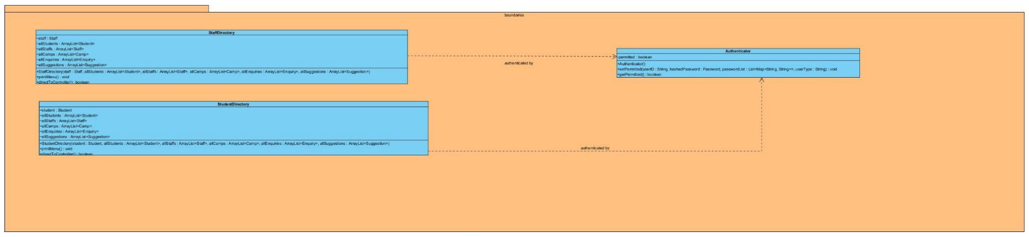
Our UML is separated into 4 parts with each part still having a relationship with the other. Our motives for separation came from the ENTITY-BOUNDARY-CONTROL principle.(ECB). Following the principle, our entities are typically independent of the user interface, our boundaries provide a UI for the user to interact and finally our controllers control the flow of data in our program. Additionally, we have a helpers department which is in charge of converting our databases into data usable by controllers and entities.

Entities:



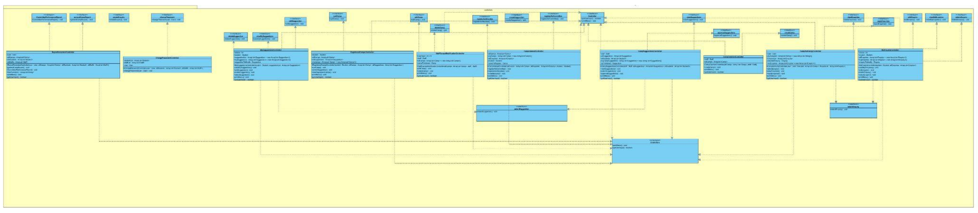
On the left and top of the entity uml diagram is all our basic entities such as Camp , Student/Staff which inherits from User and Enquiries/Suggestions. On the bottom right is our individual Arraylists containing these objects, created by the helper object OffOnline, from a csv. One thing to note is the allCamps Arraylist combines 2 Arraylists, CampRoles and CampInformation as these information are usually handled at the same time, thus increasing the program efficiency.

Boundaries:


boundaries
StaffDirectory
-staff : Staff
-allStudents : ArrayList<Student>
-allStaffs : ArrayList<Staff>
-allCamps : ArrayList<Camp>
-allEnquiries : ArrayList<Enquiry>
-allSuggestions : ArrayList<Suggestion>
+StaffDirectory(staff : Staff, allStudents : ArrayList<Student>, allStaffs : ArrayList<Staff>, allCamps : ArrayList<Camp>, allEnquiries : ArrayList<Enquiry>, allSuggestions : ArrayList<Suggestion>)
+printMenu() : void
+directToController() : boolean
StudentDirectory
-student : Student
-allStudents : ArrayList<Student>
-allStaffs : ArrayList<Staff>
-allCamps : ArrayList<Camp>
-allEnquiries : ArrayList<Enquiry>
-allSuggestions : ArrayList<Suggestion>
+StudentDirectory(student : Student, allStudents : ArrayList<Student>, allStaffs : ArrayList<Staff>, allCamps : ArrayList<Camp>, allEnquiries : ArrayList<Enquiry>, allSuggestions : ArrayList<Suggestion>)
+printMenu() : void
+directToController() : boolean
Authenticator
-permitted : boolean
+Authenticator()
+setPermitted(userID : String, hashedPassword : Password, passwordList : List<Map<String, String>>, userType : String) : void
+getPermitted() : boolean
authenticated by
authenticated by

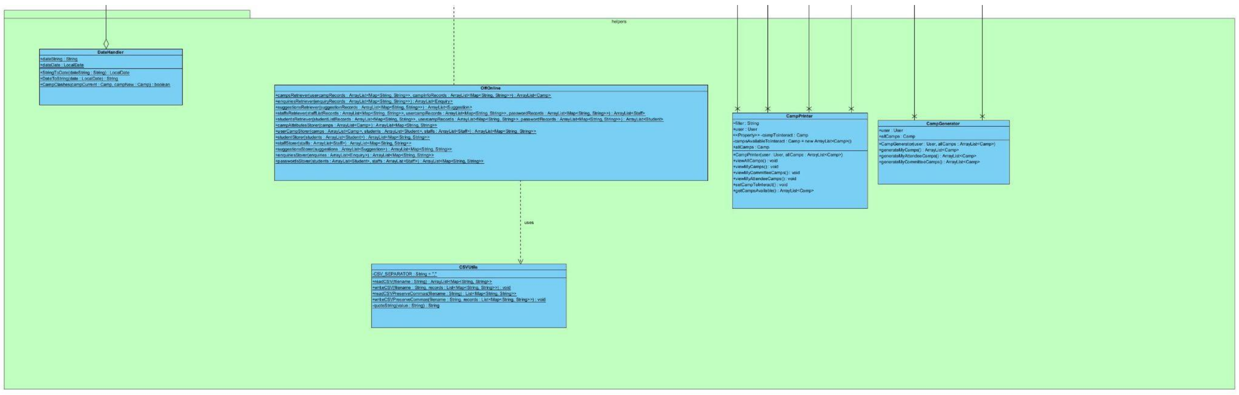
Our boundaries only consist of three classes, Authenticator, StaffDirectory and StudentDirectory. The Authenticator compares the user's login info with the records to validate before allowing them to proceed to the directories.StaffDirectory and Student Directory prints out the respective UIs for the staff and student user type as well as the switch case to call the respective controllers.

Control:



These are our controllers, which are in charge of the data flow of our project. Most of them implement a general interface called Controllers which has the necessary function for our controllers to work,getUserInput. The boundaries allow the user to provide an input to the program and the function getUserInput provides it to the controllers. This makes it so that each controller can work independently to perform their various tasks. All our necessary methods are made into individual interfaces as well, following the ISP principle so that each controller only needs to implement necessary methods.

Helpers:


helpers
DateHandler
+dateString : String
+dateDate : LocalDate
+StringToDate(dateString : String) : LocalDate
+DateToString(date : LocalDate) : String
+CampClashes(campCurrent : Camp, campNew : Camp) : boolean
OffOnline
+campsRetriever(usercampRecords : ArrayList<Map<String, String>>, campInfoRecords : ArrayList<Map<String, String>>) : ArrayList<Camp>
+enquiriesRetriever(enquiryRecords : ArrayList<Map<String, String>>) : ArrayList<Enquiry>
+suggestionsRetriever(suggestionRecords : ArrayList<Map<String, String>>) : ArrayList<Suggestion>
+staffsRetriever(staffListRecords : ArrayList<Map<String, String>>, usercampRecords : ArrayList<Map<String, String>>, passwordRecords : ArrayList<Map<String, String>>) : ArrayList<Staff>
+studentsRetriever(studentListRecords : ArrayList<Map<String, String>>, usercampRecords : ArrayList<Map<String, String>>, passwordRecords : ArrayList<Map<String, String>>) : ArrayList<Student>
+campAttributesStorer(camps : ArrayList<Camp>) : ArrayList<Map<String, String>>
+userCampStorer(camps : ArrayList<Camp>, students : ArrayList<Student>, staffs : ArrayList<Staff>) : ArrayList<Map<String, String>>
+studentStorer(students : ArrayList<Student>) : ArrayList<Map<String, String>>
+staffStorer(staffs : ArrayList<Staff>) : ArrayList<Map<String, String>>
+suggestionsStorer(suggestions : ArrayList<Suggestion>) : ArrayList<Map<String, String>>
+enquiriesStorer(enquiries : ArrayList<Enquiry>) : ArrayList<Map<String, String>>
+passwordsStorer(students : ArrayList<Student>, staffs : ArrayList<Staff>) : ArrayList<Map<String, String>>
uses
CSVUtils
-CSV_SEPARATOR : String = ","
+readCSV(filename : String) : ArrayList<Map<String, String>>
+writeCSV(filename : String, records : List<Map<String, String>>) : void
+readCSVPreserveCommas(filename : String) : List<Map<String, String>>
+writeCSVPreserveCommas(filename : String, records : List<Map<String, String>>) : void
-quoteString(value : String) : String
CampPrinter
-filter : String
+user : User
<<Property>> -campToInteract : Camp
-campsAvailableToInteract : Camp = new ArrayList<Camp>()
+allCamps : Camp
+CampPrinter(user : User, allCamps : ArrayList<Camp>)
+viewAllCamps() : void
+viewMyCamps() : void
+viewMyCommitteeCamps() : void
+viewMyAttendeeCamps() : void
+setCampToInteract() : void
+getCampsAvailable() : ArrayList<Camp>
CampGenerator
+user : User
+allCamps : Camp
+CampGenerator(user : User, allCamps : ArrayList<Camp>)
+generateMyCamps() : ArrayList<Camp>
+generateMyAttendeeCamps() : ArrayList<Camp>
+generateMyCommitteeCamps() : ArrayList<Camp>

Helpers are in-charge of making data usable by all our entities and controllers.The OffOnline class uses our csv reader and writer to convert all the csv files into arraylists of objects that can be used much more easily to handle the data, helping to improve efficiency.The Datehandler helps convert string to Data object and back to improve versatility.CampPrinter and CampGenerator are used to output camps only specific to the user from the whole list of camps, either in array form or as an printed output.

Reflection:

In our project, we divided the work into two main stages: the initial UML diagram creation and the coding phase, which also included modifications to the UML diagram.

UML Creation:

1) First, our group started working on creating the UML diagram. Our limited knowledge of the various kinds of object relationships, such as aggregation, association, and composition, presented us with substantial difficulties. In order to solve this, we went over the lectures again and looked for outside sources, which helped make these ideas more clear.
2) We sought to develop our graphic as we gained a better understanding of these relationships, paying particular attention to entities like staff, students, users, and camps. Initially, we planned to have attendee and committee entities inherit from the student entity, while staff and student entities would inherit from the user entity. But when we learned that a student might be on the committee and present at the same time, we realised there was a problem with this plan. As a result, we decided against using the student class for these two classes' direct inheritance.
3) Another issue arose with the attributes of the student/staff and camp classes. Creating an array of Camp objects within the student class and an array of student objects within the camp class led to a recursive initialization problem, potentially causing a stack overflow. To circumvent this, we decided against reinitializing already initialised students/camps/staff. Instead, we chose to have the student/staff classes hold an array of camp names as strings, and vice versa for the camps.
4) Before learning about the SOLID principles, we planned for the staff/student classes to include various methods like viewing enquiries and joining camps. However, we recognized this as a violation of the SOLID principles, as entity classes should primarily contain getter and setter methods. We resolved this by creating separate classes for specific functions, like viewing camps and suggestions.
5) Determining how to distinguish between student and staff functionalities posed another challenge. Our initial idea was to use a permissions.csv file, but we found this approach impractical. Instead, we developed controllers with switch cases to restrict users to specific functions based on their roles.

Code Implementation:

1) An issue we faced was the initialising of the various objects as it is lengthy and it cluttered the application class. We therefore decided to create a class called the OffOnline class which has methods to create the various Array of objects from the Arraylist of strings

2) Another problem we thought we faced was the problem of having an updated ArrayList of objects. We initially thought that Java uses call by value and when we passed the Arraylist of objects in the constructor of our controllers, it would not update the global ArrayList. However after some testing, we realised that Java uses call by reference which helps simplify the implementation of our code.
3) Due to various changes we made to our code in the implementation phase, it was tedious to keep the UML and the code in sync.
4) We populated the various CSV files on our own and the careless mistakes we made caused the debugging of our code to be more tedious as the error lies in the data and not the code itself.
5) The requirements of the task were unclear and this made the code logic harder to implement at times. For instance, students are unable to join camps when they are involved in other camps during the same time period. Our initial values of storing dates as string were then not ideal and instead the date class was better as it has functions to check the range of date.

Knowledge learnt:

From this project, we learn how to implement a simple database using CSV, the various solid principles and how it enhances maintainability, facilitates scalability, improves code readability and understandability. Also by segmenting the code up, it helps in identifying and reducing potential bugs. We also learn how to implement various downcasting of objects from User to either Staff or Student.

Thoughts of the assignment:

Due to the limited time and the complexity of the project, it is difficult to produce an application that obeys the SOLID principle entirely. The lack of clarity in the requirements of the project further increases the difficulty. However, it has proven fruitful in our learning of OODP and our coding and debugging skills have significantly improved. With additional time, we will definitely make amendments to the implementation of our camps, Students and Staff class. Instead of the bidirectional object composition which will definitely lead to stack overflow, we will consider a single directional object composition where Camps will have Staff and student objects in its attributes unlike what we have at the moment which is just solely ArrayList of string.

Test 1.

anwit, password, staff <prompted to change password> change to 123

2 (view all camp)

enter (skip filter)

3 (create camp)

Makers (CampName)

18 (committee slot) error checking

2 (committee slot)

1 (total slots) error checking

3 (total slots)

making pasta (description)

Jurong (location)

20231202 (start date) error checking

20240102 (start date)

20231231 (end date) error checking

20240105 (end date)

20240701 (registration closed date) error checking

20240101 (registration closed date)

T (opento)

T (visibility)

1 (create camp)

bakers (campName)

10 (committee slot)

20 (total slot)

baking muffin (description)

hall 7 (location)

20240606 (start date)

20240606 (end date)

20231101 (registration date)

F(opento)

F(visibility)

0 (return to main menu)

2 (view all camp)

aker (filter for aker)->> should see the 2 created camps

4 (view my camp)

enter(skip filter)

6(view enquiries)

Scrollup (No enquiry found)

7(view suggestion)

Scrollup (No suggestion found)

0 (to logout)

<u>Test 2.</u>

dl007, password, student <prompted to change password> 123

2 (view all camps)

enter (skip filters) , scroll to see camps

3 (join camp)

enter(skip filter)

5 (select maker camp)

3 (submit enquiry)

what are we making (enquiry)

5 (select maker camp)

1 (sign up as committee)

5 (select maker camp)

3 (submit enquiry) // cannot send enquiry anymore...

0 (quit to main)

0 (quit to main)

4 (go to camp manager)

3 (create suggestion)

1 (select makers camp)

lets reduce to total slots to 2(suggestion message)

0(return to main menu)

6 (view edit suggestion)

1 (edit suggestion)

1 (choose suggestion 1)

reduce committee slot to 1 (suggestion message)

0 (return to main menu)

0 (log out)

Test 3

anwit, password, staff (error checking) -> prove that the database is updated

anwit, 123, staff -> mention that the password has changed

7 (view suggestion menu)

1 (mark as read)

2 (approve suggestion)

1 (select the suggestion for maker) -> limitation of our system is that manual ammendment

1 (view all) // now status: {approved}

0 (return to main menu)

4 (view my camps)

enter (skip filter)

2 (choose camp number 2 baker)

1 (edit camp)

7 (select visibility to change)

T (change visbility to T)

enter(skip filter)

0 (to quit)

5 (generate report menu)

1 (generate camp report)

1 (select camp maker)

Y,Y,Y....... N, N (select attributes to show in report)-> show onscreen the report generated

2 (generate performance report)

enter (skip filter)

1 (select camp maker)-> show onscreen the points of the committee member denise add points

0 (logout)

Test 4

ct113, password,student

password (change password to password)

3 (join camp menu)

enter (skip filter)-> see baker is available

5 (select makers)

3 (submit enquiry)

what is the pasta (enquiry message)

5 (select makers)

2 (sign up as attendee)

0 (to quit)

0 (to return to main menu)

4 (view and manage my camp)

2 (withdraw from camp)

3 (invalid choice)-> error checking

2 (withdraw from camp)

1 (remove user from maker camp and added to blacklist)

0 (return to main menu)

3(join camp menu)

enter (skip filter) -> show that slots taken up is reduce to 1

5 (select maker camp)

1 (sign up as committee)-> prompt user that he cannot join as he is blacklisted

5 (select maker camp)

2 (sign up as attendee)-> prompt user that he cannot join as he is blacklisted

0 (return to menu)

0 (logout)

Test 5

dl0007, 123, student

Not prompted to reset password

8 (view reply enquiry)

1 (view enquiry)-> look at the enquires printed and mention status is unviewed

1(view enquiry)-> look at enquiries and mention now that status is viewed

2 (reply enquiry)

2 (select enquiry number 2)

yes (reply yes to the enquiry)

1 (view enquiry) -> mention that the reply is reflected with the responder name dl007

0 (return to menu)

0 (logout)

Test 6

anwit,123, staff(login as anwit)

5 (generate report controller)

2 (performance report)

enter (skip filter)

1 (camp makers)-> show that densie have total points of 2

0 (quit to main menu)

4 (view and modify camp)

enter (skip filter)

2 (camp bakers)

2 (delete camp)

enter(skip filter)-> bakers no longer exists

1 (camp makers)

2 (delete camp) -> Cannot delete camp as there are student already in the camp!

0 (return to main menu)

0 (quit)

0 (logout)