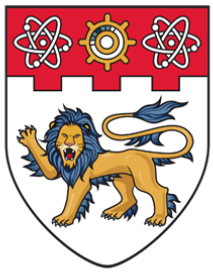


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

AY2023/24 SEMESTER 1

SC2002: Object Oriented Design and Programming

Title: Camp Application and Management System (CAMs)

Lab Group: SCSA (Group 2)

Group members	Matriculation Number
Khant Zaw	U2221075E
Jayda Foo Kai Yen	U2222333C
Phoebe Lee Jia Ying	U2223881K
Chan Wei Ting	U2221415E
Chia Yi Xuan	U2221955D

Declaration of Original Work for SC2002 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 be awarded for the assessed work. In addition, disciplinary actions may be taken.

Name	Course	Lab Group	Signature /Date
Khant Zaw	**SC2002**	**SCSA**	Khant Zaw
Jayda Foo Kai Yen	**SC2002**	**SCSA**	
Phoebe Lee Jia Ying	**SC2002**	**SCSA**	Phoebe Lee
Chan Wei Ting	**SC2002**	**SCSA**	Wei Ting
Chia Yi Xuan	**SC2002**	**SCSA**	Yi Xuan

Table Of Contents:

1. Introduction..... 5
2. Design Considerations..... 5
3. UML Class Diagram..... 5
4. Testing..... 5
5. Reflection..... 5

1. Introduction

CAMs is an application for staff and students to manage, view and register for camps within NTU. The application will act as a centralized hub for all staff and students. In this report, design considerations, design principles and the use of object-oriented programming (OOP) concepts are demonstrated. The report also includes a detailed UML Class Diagram for the CAMs, followed by screenshots of the testing done using the test data that we have provided.

2. Design Considerations

2.1 Approach Taken

The CAM application was designed to focus on cohesiveness and loose coupling. The application is divided into three types of classes, Entity, Controller and Boundary.

Entities: User, Student, Staff, Camp, Enquiry, Suggestion
Controller: UserController, StudentController, StaffController, CampController, EnquiryController, SuggestionController
Boundary: UserBoundary, StudentBoundary, StaffBoundary, CampBoundary, EnquiryBoundary, SuggestionBoundary

When a user interacts with the boundaries of the system, the controller is called upon to carry out desired activities, such as altering an object or retrieving data for display. Each of these categories work together to complete our system while minimizing our reliance on one another. As a result, our system is highly adaptable, extensible, and simple to maintain. When extending our system, for example, introducing a new system function will require minimal work.

2.2 Implementation of SOLID design principles

2.2.1 Single Responsibility Principle (SRP)

The Single Responsibility Principle emphasizes that each class should have a clear and singular responsibility and only focus on related tasks. By following the SRP principle, the process of testing, modifying and reusing code can be simplified which results in a robust software design. For example, each Boundary class menu is dedicated to only one

purpose such as ‘suggestions’ and ‘enquiry’. Therefore, only one class needs to be modified if the ‘suggestions’ or ‘enquiry’ needs to be changed.

2.2.2 Open/Closed Principle (OCP)

According to the available/Closed Principle (OCP), classes should be available for extension but closed for modification, allowing classes to easily extend without modifying the class itself. Abstraction, inheritance, and polymorphism can all be used to achieve OCP.

For our project, we applied OCP by creating a ‘CentralManager’ that allows us to ingest and write data for different entities. If new entities are created, the class can be extended without modifying existing methods. With the same concept in mind, we also created ‘BaseController’, ‘BaseBoundary’ and ‘BaseInterface’ classes so we are able to easily add new controllers and boundaries without changing current classes.

2.2.3 Liskov Substitution Principle (LSP)

The Liskov Substitution Principle implies that objects of a superclass should be replaceable with objects of its subclasses without destroying the application. It requires the objects of the subclasses to behave in the same way as the objects of the superclass. For example, the principle was used in the inheritance of the ‘User’ class and subclasses ‘Student’, ‘Staff’ and ‘CampCM’. The ‘currentUser’ attribute is of type ‘User’, but it can hold an instance of any of the ‘User’ subclasses. The rest of the ‘UserController’ code would work correctly regardless of which subclass of ‘User’ the ‘currentUser’ is.

2.2.4 Interface Segregation Principle (ISP)

The Interface Segregation Principle refers to many specific interfaces being better than one general interface. In other words, we should always avoid designing a ‘fat interface’. As such, when developing our system, we noticed that this is important to promote maintainability, flexibility and modularity. For example, the ‘CampController’ class provides methods for handling operations related to camps, such as ‘getCampByID()’ and

‘getAvailCamps()’. This adheres to the principle as each controller class (which can be thought of as an interface) is tailored to a specific set of operations.

2.2.5 Dependency Injection Principle (DIP)

The Dependency Injection Principle suggests that higher modules must not depend on lower modules, but both should depend on abstraction. DIP helps to make our applications loosely coupled, extendable, and maintainable. We applied DIP by passing dependencies through constructors (Constructor Injection). For example, our different controller and boundary classes are passed through the ‘CentralManager’ object through the constructor.

2.3 Object-Oriented Concepts

2.3.1 Abstraction

Abstractions mean hiding internal details and focusing on showing functionality only, focusing on what the object does instead of how it does it. In our application, the boundary classes often call methods defined in the controller classes to perform a function, while the details and implementation of the method itself are written in the controller classes.

For example, when a ‘Student’ wants to view all the camps that they are participating in, the ‘StudentBoundary’ class calls the ‘viewCamps()’ method, which calls ‘this.getCampBoundary().viewCamps(studentID);’, which calls ‘this.getCampController().getAvailCamps(studentID);’, while the actual implementation is hidden in the ‘getAvailCamps()’ method in the ‘CampController’ class.

2.3.2 Polymorphism

Polymorphism refers to the ability of an object reference to be referred to different types. In our application, the ‘getCurrentStudent()’ function executes downcasting by:
‘public Student getCurrentStudent() {return (Student) this.currentUser;}’
where the ‘User’ reference was used to accommodate the types of users such as ‘Student’ and ‘Staff’. The ‘studentController’ and ‘StaffController’ classes are thus only able to

execute methods according to their own specific types.

2.3.3 Inheritance

Inheritance is a mechanism that allows a class to inherit properties and behaviors from another class, absorbing their attributes and behaviors, as well as adding new capabilities in the sub-classes. Inheritance is mainly used in the boundary and controller classes. For example, our boundary classes extend the 'BaseBoundary' class so that they are able to share the 'get()' functions as well as the different 'getBoundary()' functions, which allows our code to be more reusable and organized.

2.3.4 Encapsulation

Encapsulation refers to the bundling of data with the methods that operate on that data into a class. It is used to hide the details or implementation of a class from its users, preventing unauthorized parties' direct access to them. Users of a class are only able to access their private attributes through getter and setter methods. In our application, encapsulation is used in almost every entity class to protect their private attributes and data. For example, the 'User' entity class has private attributes such as 'name' and 'email', which can only be accessed by other classes by calling the 'getName()' and 'getEmail()' functions, or modified by calling the 'setName()' and 'setEmail()' classes.

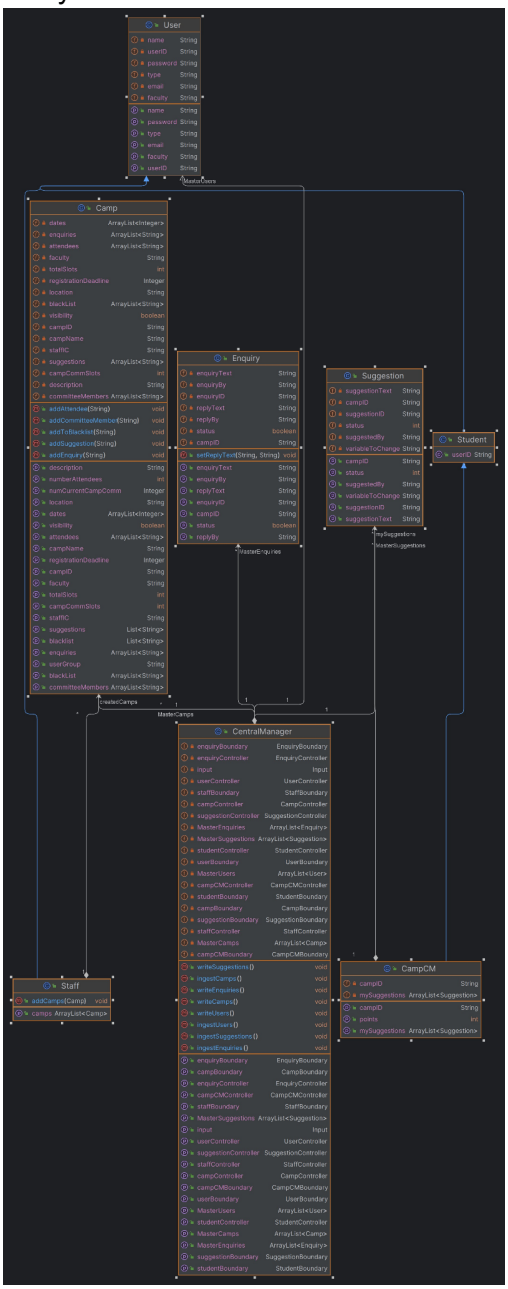
2.4 Assumptions Made

a. The application is intended to serve one user at a time, not more than one user concurrently.
b. Registration of camp and camp committee is automatic as long as there is a vacancy.
c. Number of camp committee members is counted into total slots.
d. Camps that are open to all faculties are classified as user group 'ALL'.
e. Staff cannot delete a camp once there is one attendee.
f. Staff can only edit camp description, location and visibility. Staff can only extend the registration deadline. Editing other features would affect existing attendees so it is disallowed.

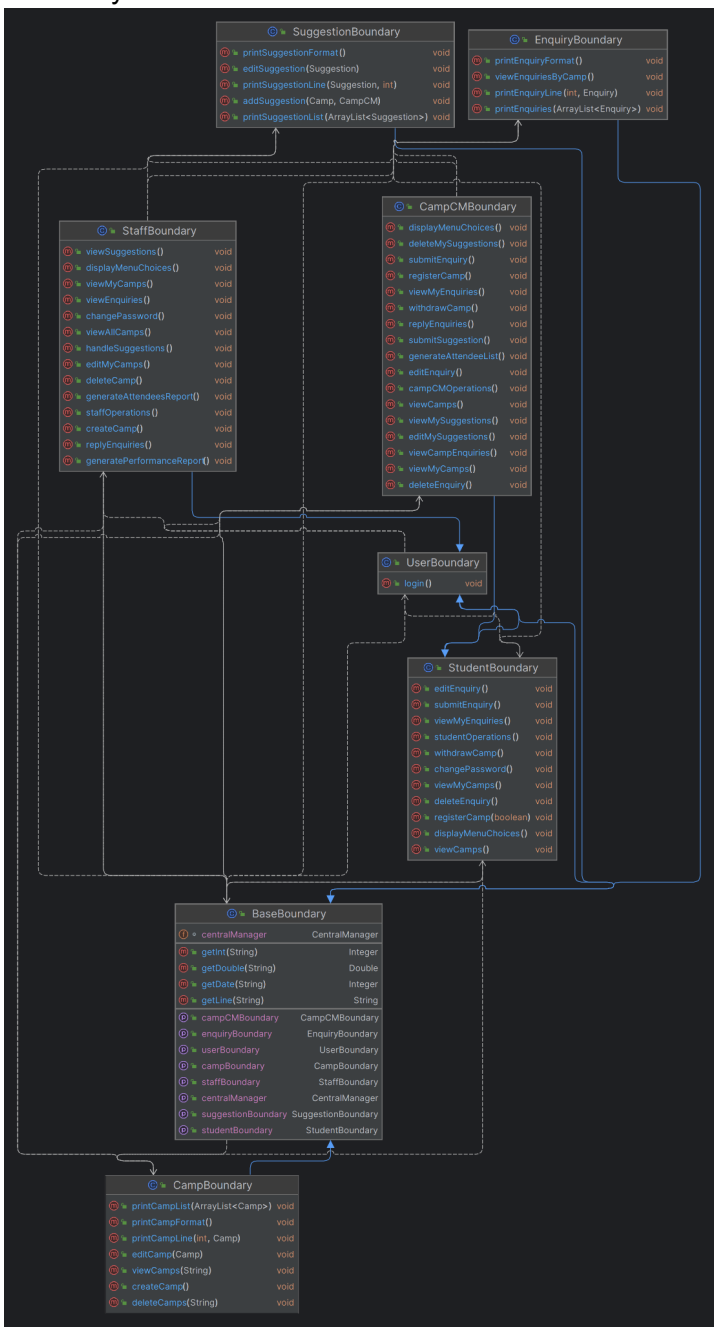
g. Likewise, for the above reasoning, camp committee members can only make suggestions on camp description and location.

3. UML Class Diagram

Entity:



Boundary:



(UML Diagrams are attached separately in the ‘UML Diagrams’ Folder for easier viewing)

4. Testing

```
Do you want to login?

Please enter 'Y' or 'N' only
Y
Input UserID: LE51
Input Password: password
```

Fig 2: Student and Staff Login

Student:	Staff:

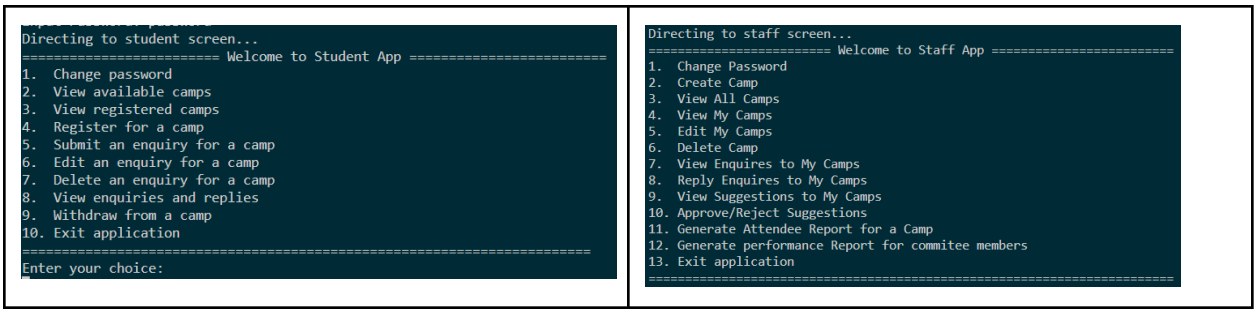
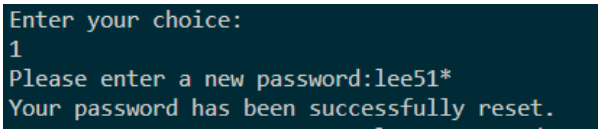


Fig 3: Main Menus

4.1 Student



Prompt for change:

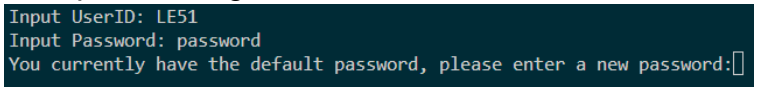
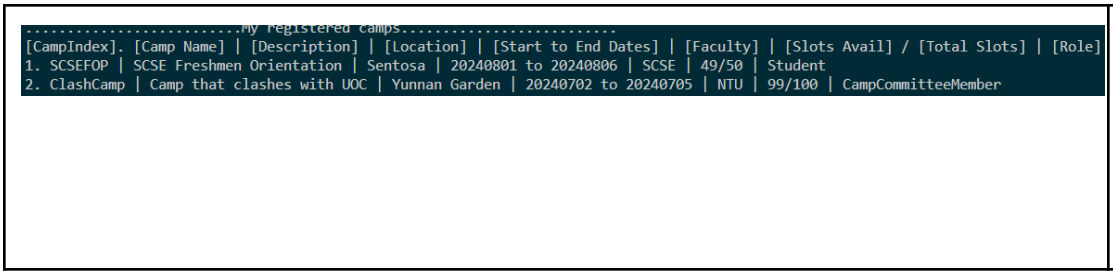


Fig 4: Student can change password (will be prompted to change password if password is default)



Remaining Slots:

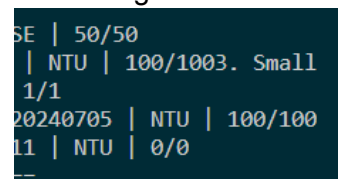


Fig 5: Student can view the list of camps that are open to his/her user group and remaining slots

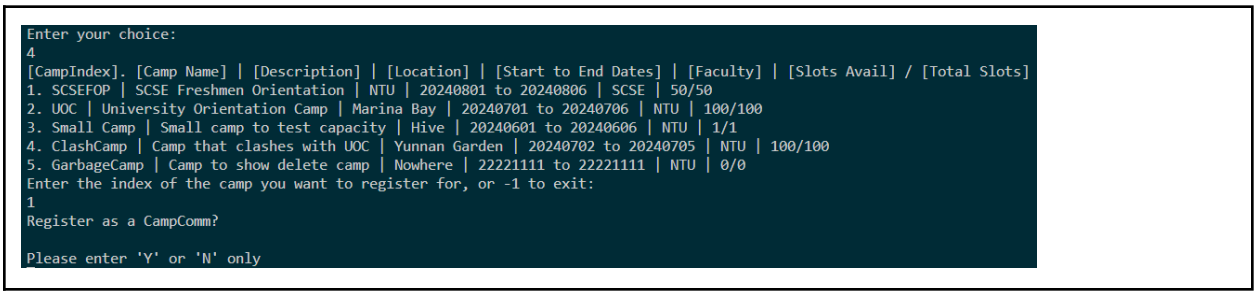
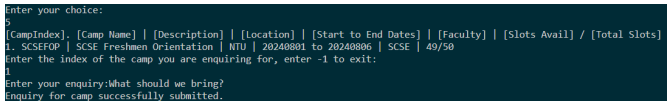


Fig 6: Student can register for camps either as a camp attendee or camp committee.

Submit:



View:

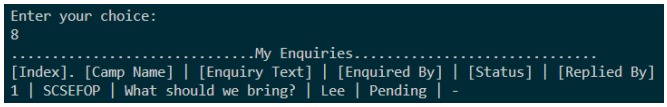


Fig 7: Student can submit and view enquiry

Edit:	Delete:

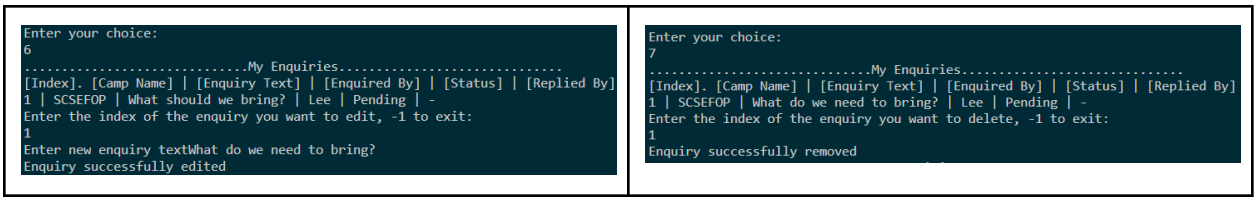


Fig 8: Student can edit and delete enquiry

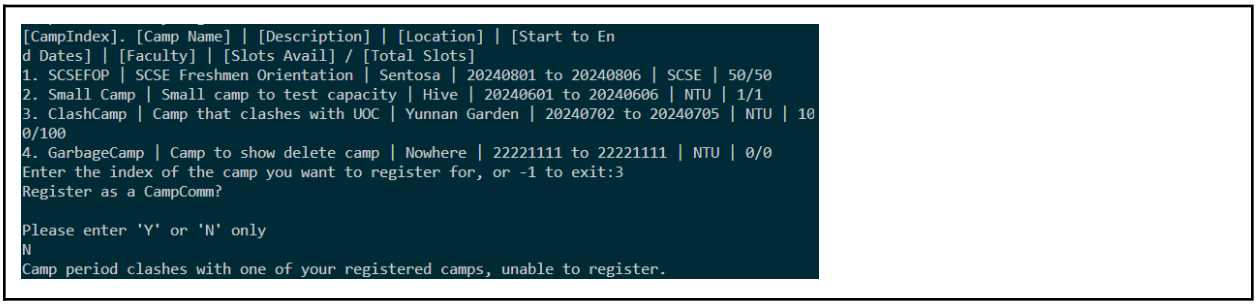


Fig 9: Student is not allowed to register for multiple camps if there are clashes in the dates

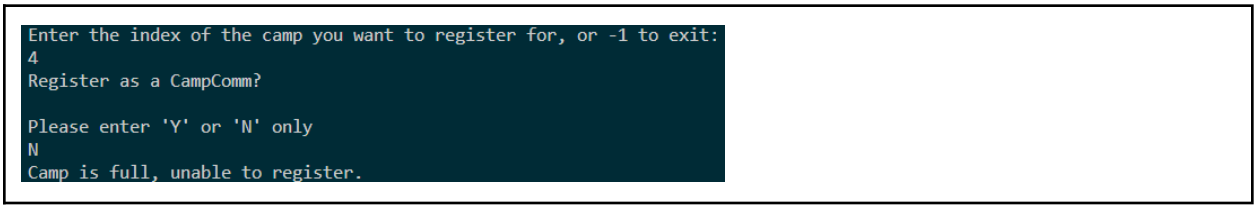


Fig 10: Student cannot register for a camp if it is full

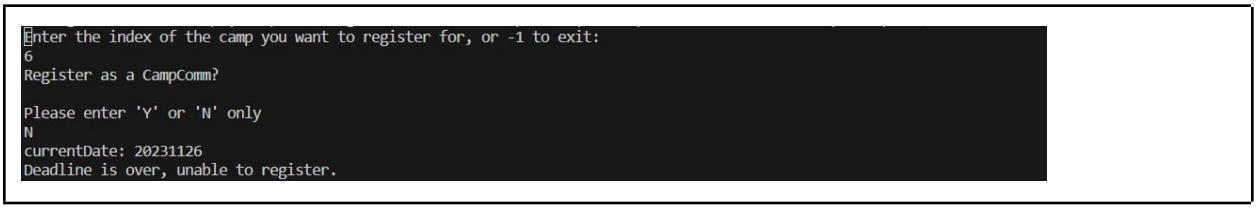


Fig 11: Student cannot register for a camp if it is past Registration Deadline

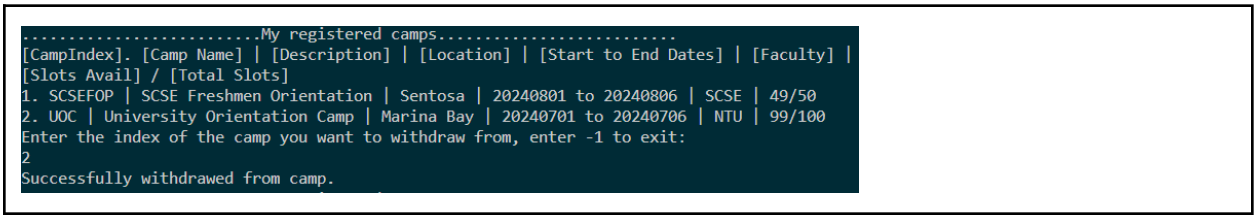


Fig 12: Student allowed to withdraw from camps that his/her has already registered

4.1.1 Camp Committee Member (CCM)

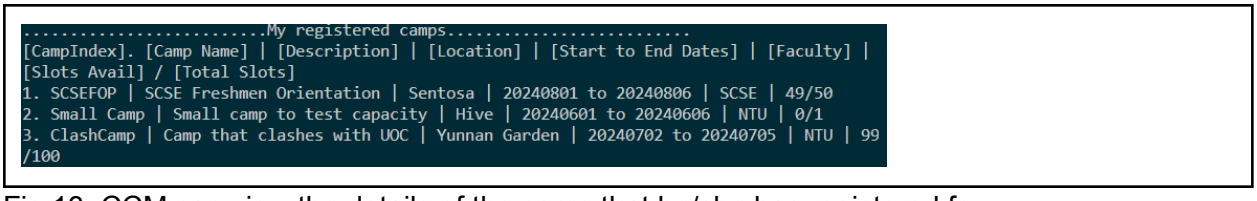


Fig 13: CCM can view the details of the camp that he/she has registered for

Fig 14: CCM can submit suggestions for changes to camp details to staff

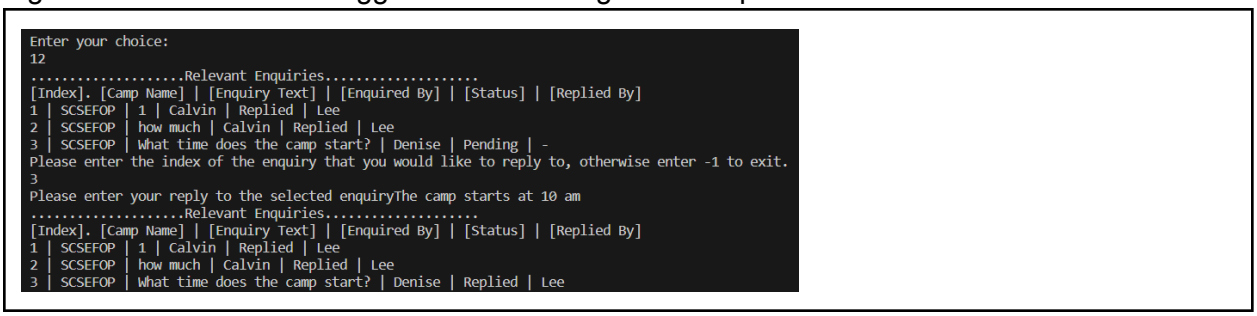


Fig 15: CCM can view and reply to enquiries from students to the camp they oversee

Fig 16: CCM can generate a report of the list of students attending

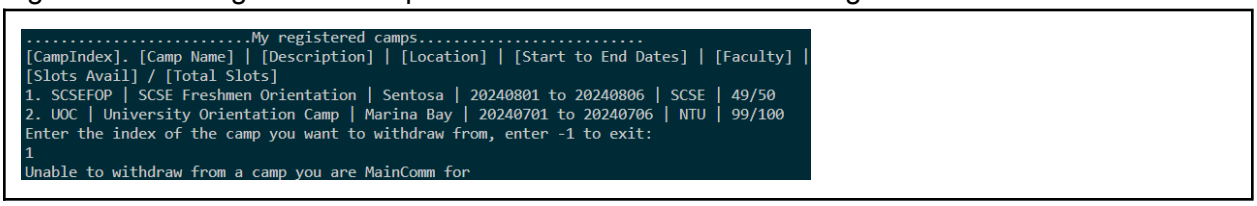
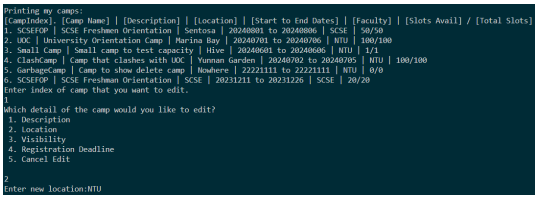
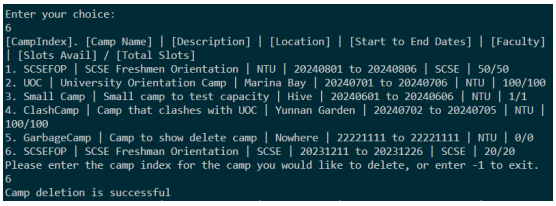


Fig 17: CCM cannot quit from camp

4.2 Staff

(Change password for Staff works the same as student)

Create:	Edit:	Delete:
		

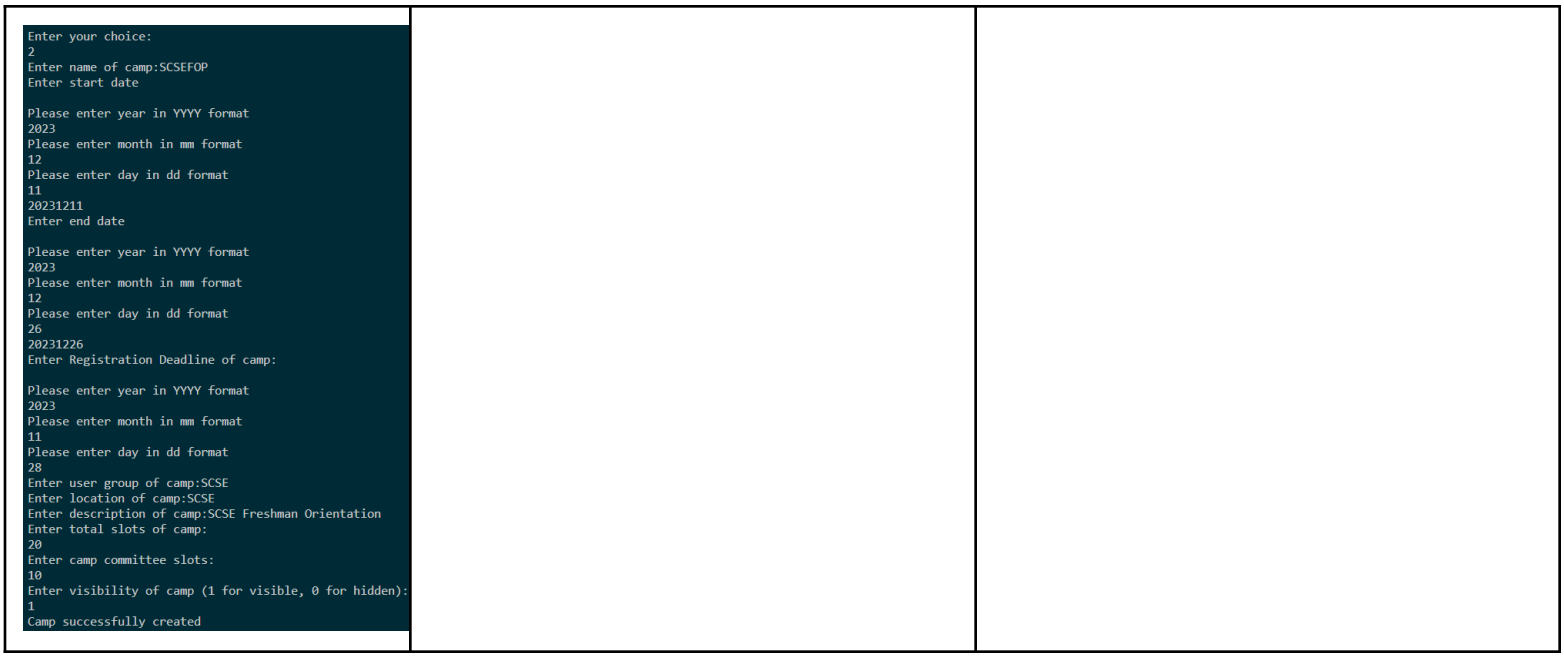
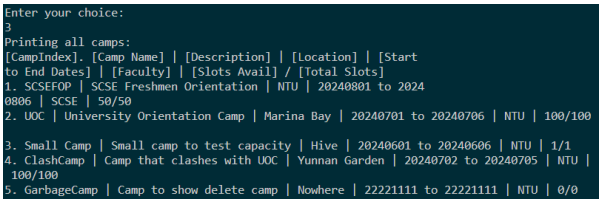


Fig 18: Staff can create, edit and delete camp

All Camps:



Created List of camps:

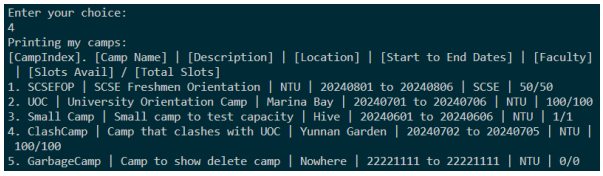
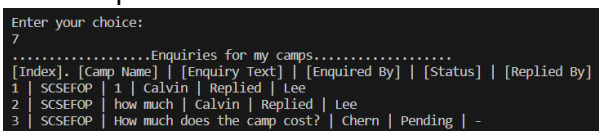


Fig 19: Staff can view all created and available camps

View Enquiries:



Reply Enquiries:

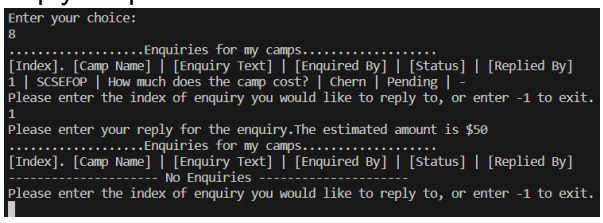
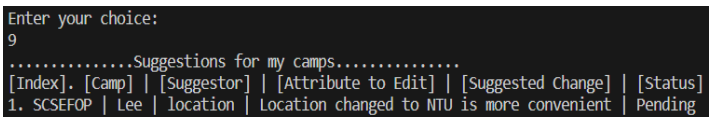


Fig 20: Staff can view and reply to all enquiries from the camps they created

View Suggestions:



Approve Suggestions:

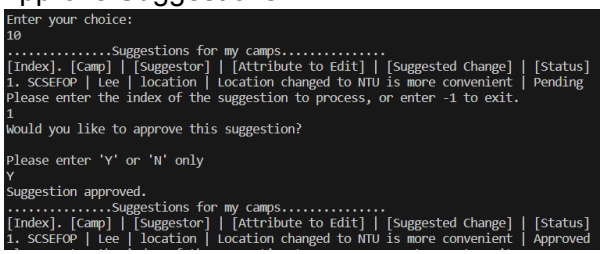


Fig 21: Staff view and approve suggestions to camp details from camp committee

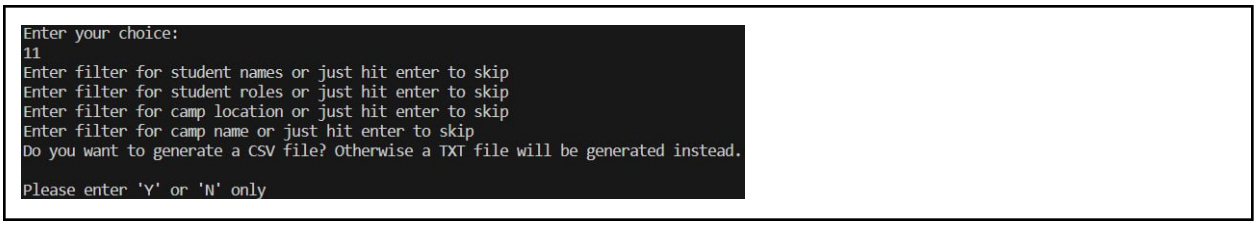


Fig 22: Staff can generate reports (csv or txt format)

5. Reflection

In our assignment, all of us witnessed the value of design concepts through real-world applications, bridging the gap between the theoretical knowledge learnt and its integration. Our main takeaway would be that software programs need high cohesion and loose coupling so that it is highly flexible, readily managed, and easily extensible. This assignment highlighted that a minor change in our code had a big effect on the other classes, requiring many areas of our code to be altered.

Simultaneously, we have learned how to develop software that is appropriate for its functions and real-world applications. We constantly altered our design with the different users given in mind, so that the program takes into account numerous potential scenarios and ensures that there are no conflicts between the users.

5.1. Further Improvements

1. **Security**

We could further enhance the application by implementing password masking and password restrictions. We could also add password encryption to improve the security of the user accounts.

2. **Extending to other student bodies**

On top of only grouping students via their faculties, we can also add other grouping types such as CCAs. The list of camps can also be grouped by CCA clubs for easier sorting and visibility. There could be more usage of interfaces in the application adhering to the ISP principle to maintain flexibility and maintainability.