## SC2002 Object Oriented Design and Programming

## Project Title: Camp Application and Management System (CAMS)

### 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 be awarded for the assessed work. In addition, disciplinary actions may be taken.

| Name | Course<br>(CE2002 or CZ2002) | Lab Group | Signature /Date |
|---|---|---|---|
| Zhang Siyu | SC2002 | SCS5 | (25/11/23) |
| Phyo Sandar Win | SC2002 | SCS5 | Phyo Sandar Win (25/11/23) |
| Hong Zhi Hao | SC2002 | SCS5 | (25/11/23) |
| Harvande Advika Sandeep | SC2002 | SCS5 | Harvande Advika Sandeep (25/11/23) |
| Venus Ng Min Jia | SC2002 | SCS5 | (25/11/23) |

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

# 0 Executive Summary

This report focuses on explaining the design considerations and OOP concepts used in developing the Camp Application and Management System, a Java Command Line Interface (CLI) application. This report consists of a detailed UML Class diagram, solid design considerations, use of OOP concepts, and other design considerations like repository pattern and Observer Pattern. There is also a section to showcase the effectiveness of our system using test cases and results, as well as a reflection section entailing the challenges we faced, the knowledge we gained, and further improvements we can make. Overall, this report entails a comprehensive analysis of the design principles and OOP concepts used in the development of the CAM System.

# 1 Design Considerations

## 1.1 Overview of the Approach

We designed the Camp Management System with a **modular approach** while adhering to **object-orientated concepts** and **SOLID design principles**. This ensures that the CAM system is **easily testable**, **maintainable**, and **extensible**. Firstly, we split CAM into **different packages** according to its different functionalities. Secondly, each class within a package is aligned to **object-orientated concepts** to take a specific task according to the package's responsibility. Thirdly, we implemented **SOLID design principles** into the construction of these classes and their relationship with each other. Hence, with this approach, we designed CAM using a **Model-View-Controller (MVC)** architecture, with additional packages like `enums`, `interfaces`, `util`, `services`, and `stores`. This allows for **easy understanding** and **maintenance** by new developers if concerns are kept separate between different classes. We have included additional design considerations and patterns like a **simplified repository pattern** to address specific challenges when developing CAM.

## 1.2 Assumptions Made

**Firstly**, we assumed that the data stored in the data files are well-formed and would not require extensive data checking when importing the data during the initialization of CAM. Such an assumption has only been taken into consideration when data files are used for the first time during the initialization of CAMP since subsequent updates to the same data files will be well-formed as intended. **Secondly**, we have assumed that users to our application know what CAM is and understand its functionality and purpose according to their roles- Student, Staff, and Camp Committee. Thus, the CLI descriptions and instructions explaining each feature's actions were less descriptive and prevalent. This can be seen in our menu options and interfaces for each role. **Thirdly**, we are assuming that our users have the necessary resources to run CAMS.

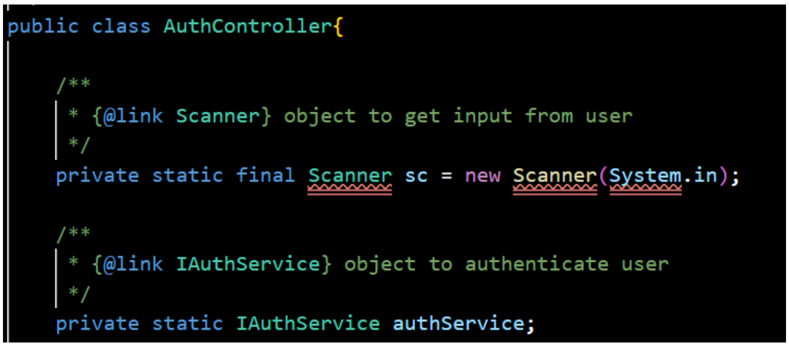
## 1.3 Use of Object-Orientated Concepts

### 1.3.1 Inheritance

**Hierarchical relationships** are created within classes using Inheritance, whereby child classes inherit behaviors and properties from parents. For example, the Student and Staff class inherits from the User class while Class Committee inherits from the Student class. The same is applied in the controller class where StudentController and StaffController inherit from UserController while CommitteeController inherits from StudentController. Inheritance was also applied in the services class where AuthStaffService and AuthStudentService inherit from AuthService. Our UML Class diagram also exhibits this concept of inheritance.

### 1.3.2 Abstraction

We implemented both data and procedural abstraction and simplified our system by separating interface and implementation by abstracting details of services and view folder classes to **interface** folder classes. One example is SuggestionView in view and ISuggestionView in the interface. We also use an **abstract model class** to hide implementation details of how a function or method is called sucha as shown in Fig 1.3.2a.

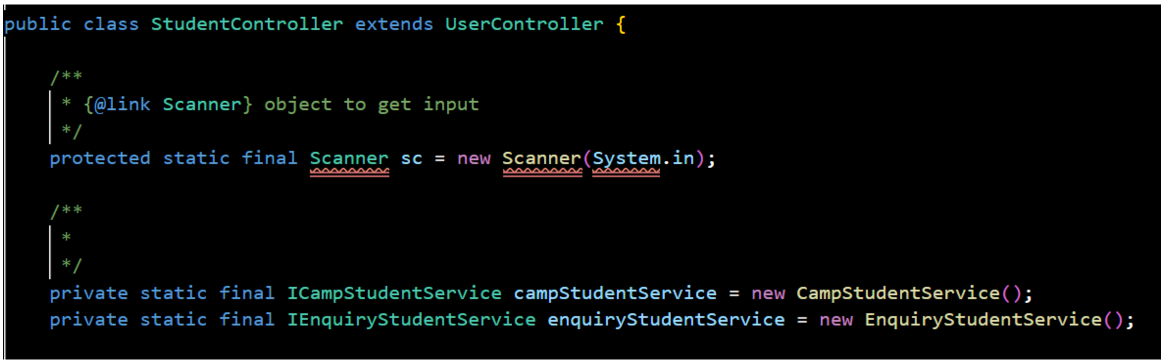


```
public class AuthController{

    /**
     * {@link Scanner} object to get input from user
     */
    private static final Scanner sc = new Scanner(System.in);

    /**
     * {@link IAuthService} object to authenticate user
     */
    private static IAuthService authService;
```

*Fig 1.3.2a shows the abstraction of AuthService and use of IAuthService as an interface for the different IAuthService implementations.*

### 1.3.3 Encapsulation

Encapsulation observed through the building of barriers to an object's private data and allowing access only through **public getter and setter methods**. An example is displayed in Fig 1.3.3a.



```
public class StudentController extends UserController {

    /**
     * {@link Scanner} object to get input
     */
    protected static final Scanner sc = new Scanner(System.in);

    /**
     *
     */
    private static final ICampStudentService campStudentService = new CampStudentService();
    private static final IEnquiryStudentService enquiryStudentService = new EnquiryStudentService();
```

*Fig 1.3.3a Implementation of private campStudentService methods under Student Controller*

### 1.3.4 Polymorphism

Polymorphism allows for more **flexible** and **reusable** code by promoting a **common interface** for various classes and by allowing objects to take on multiple forms and behaviors. 'displaycamp' has been overridden, for example, allowing for different behaviors in classes like CampAvailableView and CampRegisteredView.

## 1.4 Design Principles

### 1.4.1 Single Responsibility Principle

The main concept revolves around the idea that ‘each responsibility is an axis of change’. The use of this concept in our design allows for each responsibility to be allocated specifically to each package and singular tasks to each class.

#### *1.4.1.1 Controllers*

Controller classes enable us to carry out **singular control tasks**. It makes use of service and user classes based on inputs to control output and flow. For instance, AuthController controls the user authentication experience, while StudentController, StaffController and CommitteeController controls the CAMS experience for students, staff and camp committee members respectively.

#### *1.4.1.2 Stores*

Stores classes **manage the global state of the application** with **direct interaction with the database**. DataStore interacts directly with the database and manages global object data state. AuthStorex manages the global authentication state and hence classes in the store manage global state application.

#### *1.4.1.3 Interfaces*

Interface classes **declare a set of methods that require concrete classes** to implement and **provide a way to achieve multiple inheritance of a type**. ICampView provides an interface to be implemented by multiple classes like CampAvailableView and CampRegisteredView to display details of selected camps.

#### *1.4.1.4 View*

These classes **display information to users** regarding specific objects. SuggestionView, for example, displays suggestions for the camp using CLI, while CampRegisteredView displays camps registered for.

#### *1.4.1.5 Enum*

These classes **store constant values** to be used throughout the application such as ‘Staff’, ‘Student’, ‘Committee’ for UserRole, and ‘CCEB’, ‘SCSE’, etc. for schools.

#### *1.4.1.6 Model*

Model is for **entity classes** and the **definition of their relationship**. We split it into two different packages for better ease of understanding. For instance, model.camp has entities like Camp and Suggestion while model.user has Student and Staff.

#### *1.4.1.7 Services*

Service classes are concrete classes that implement specific business logic functionalities. This included services like ReportGeneratorService that generates camp report and can be used by other abstract classes like StaffController.

### 1.4.2 Open-Closed Principle (OCP)

Implementation of OCP allows us to change what the modules do, without changing the source code of the module. This ensures **loose coupling** between our classes to ensure easy maintainability by using interface and abstract classes.

#### *1.4.2.1 Interface*

We created specific interfaces for view and service classes which enable **multiple interfacing** in our application, allowing for simpler maintainability and switching out or modification of abstract or concrete classes. `ICampView` provides a simpler interface to display camp for multiple classes like `CampAvailableView` and `CampRegisteredView`.

#### *1.4.2.2. Abstract*

Abstract classes containing both concrete and abstract methods or functions have been implemented such that it enables multiple subclasses to extend from them. `AuthService` is a class implemented by multiple subclasses easily extendable like `AuthStudentService` and `AuthStaffService`, providing a more definite authentication service.

### 1.4.3 Liskov Substitution Principle

By using this principle, we ensured that a user of a base class should continue to function properly if a derivative of that base class is passed to it, meaning pre-conditions should be no weaker than the base class, and post-conditions should be no stronger than base class either. This is attained through **polymorphism** and is applicable to all **generalisation** and **realisation** relationships. For instance, `AuthService` is extended by `AuthStudentService` and `AuthStaffService`.

### 1.4.4 Interface Segregation Principle

Client-specific interface promotes non-dependency of classes on interfaces that they do not use. For Enquiry related interfaces, we implemented multiple specific interfaces like `IEnquiryStudentService`, `IEnquiryStaffService`, `IEnquiryCommitteeService`, and `IEnquiryView`. This enables loose coupling between classes and a more user-friendly interface.

### 1.4.5 Dependency-Injection Principle

An external supply of dependencies is available rather than having a component create them internally through the constructor, setter, and method injection.

#### *View*

Controller classes interact with all views via an interface like ICampView instead of CampView.

#### *Service*

Controller and Store classes interact with service classes via interfaces and hence they implement interfaces instead of directly interacting with concrete service classes. E.g. `StudentController` utilises `ICampStudentService` instead of `CampStudentService`.

## 1.5 Other Design Considerations

### 1.5.1 Strategy Pattern

Strategy pattern was implemented in our design through classes that contain a reference to a strategy object and might define an interface that lets the strategy access its data. Basically, an interface common to all supported algorithms. One such implementation of this would be under view whereby `EnquiryView` is interfaced by multiple classes like staff, student, and camp committee.

### 1.5.2 Simplified Repository Pattern

By separating the data layer, our design promotes consistency in data access patterns through the application and includes high-level abstraction over the data access layer, making the application more modular and maintainable.

### 1.5.3 Adapter Pattern

The adapter pattern allowed us to make incompatible interfaces to work together without changing their source code. In this pattern we adopted a target, an adaptee, and a adapter. The target is an interface that client uses that defines the specific methods. Adaptee is a class that has an interface incompatible with the client. Adapter is a class that bridges the gap between the target and adaptee by implementing the target interface and translate client requests to adaptee's interface.

### 1.5.4 Observer Pattern

Observer pattern enables a one-to-many dependencies between objects, so that when one object changes its state, all it's dependents are notified and updated automatically. Entity classes like Student and Staff are implemented by multiple interfaces or abstract classes so that all dependents like `AuthStudentService` are immediately updated.

### 1.5.5. Foreign Key Association

To ensure data integrity, foreign keys are used to link related objectsd together instead of composition/ aggregation. This ensures that when details about an object changes, the other objects that contain the updated object will still have the updated information about the object. This means that there is always only one instance of an object in the whole CAMS application. For example, `Camp` class is related to `Student`, `Staff` and `Committee` classes. To ensure data integrity, the private attributes of `Camp` related to `Student`, `Staff` and `Committee` are 'students', 'campCommittee' and 'staffIC' strings or list of strings instead of objects. To retrieve the `Student`, `Staff` and `Committee` objects, public getter methods are used to directly access the `DataStore` and retrieve them.

# 2 UML Class Diagram

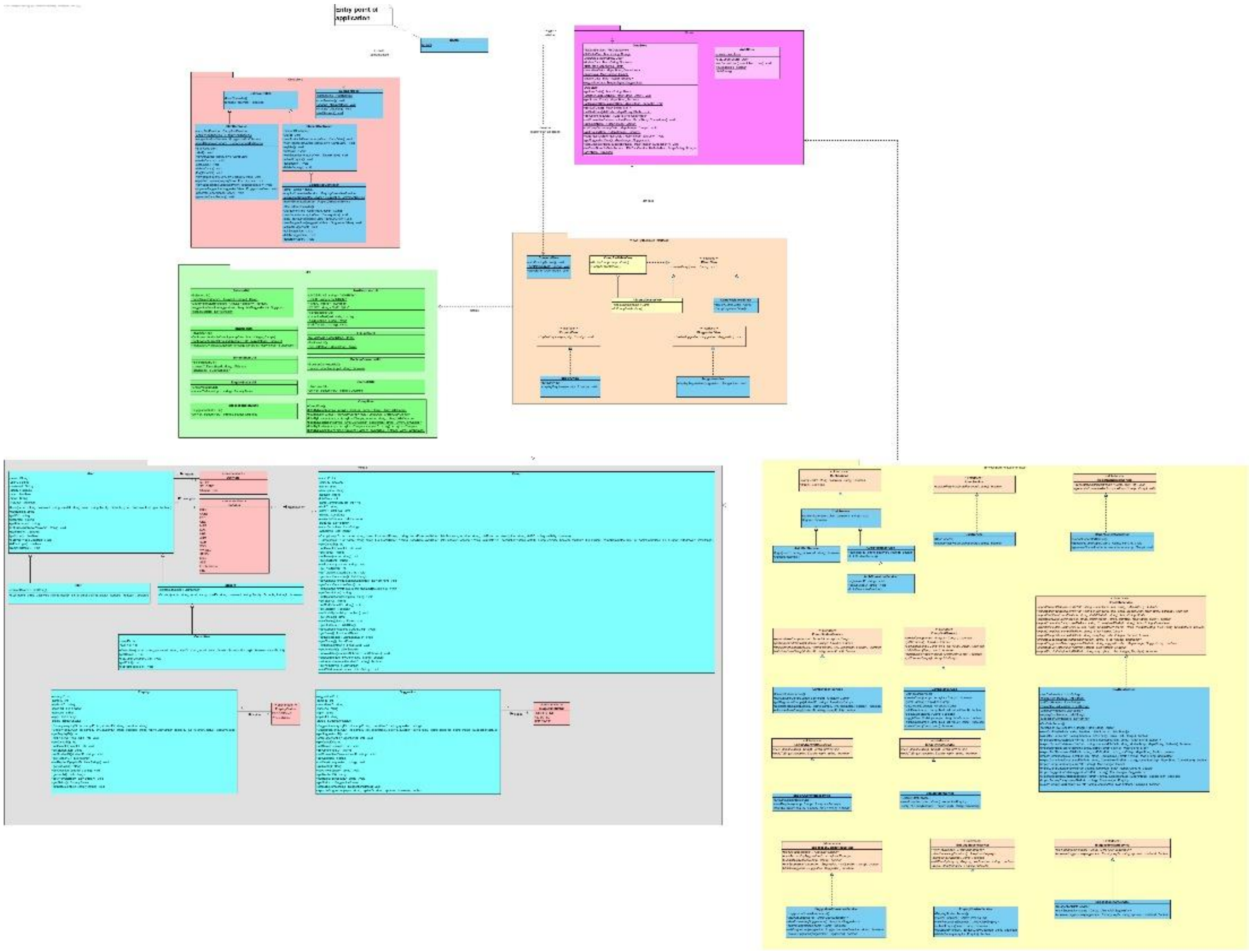
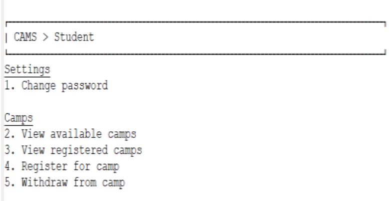


*Fig 2: UML diagram contains Models, Controllers, Stores, Utils, Services, Views*

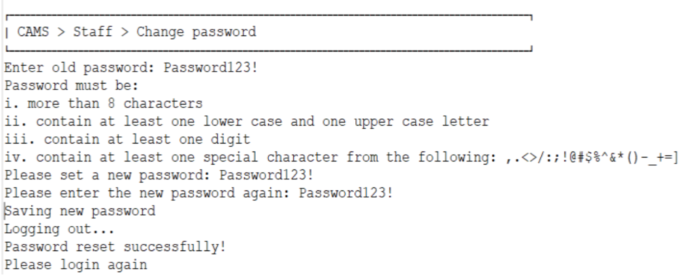
# 3 Functional Tests and Results

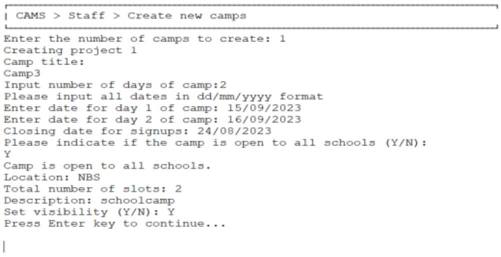
## 3.1 Authentication

| Test Case | Test Description | Expected Result | Pass/Fail |
|---|---|---|---|
| 1 | [Cannot Login]<br>(a) Entering Wrong UserID for the selected role.<br>(b) Entering Correct UserID but wrong | User fails to log in and is prompted to key in credentials again. | Pass |

| | | | |
|---|---|---|---|
| | password for the selected role. | | |
| 2 | [Successful Login] User enters correct UserID and password for the selected role. | User logs in successfully and a different menu is displayed for both Staff and Student, with different functions available.<br><br>\| CAMS > Student<br>Settings<br>1. Change password<br>Camps<br>2. View available camps<br>3. View registered camps<br>4. Register for camp<br>5. Withdraw from camp<br>*Figure 3.1.1a Student Menu upon log in* | Pass |
| 3 | [Change Password] | When password meets requirements: User changes password successfully and is asked to log in again | Pass |

## 3.2 Staff Functionality

| Test Case | Test Description | Expected Result | Pass/Fail |
|---|---|---|---|
| 1 | [Change Password] | User is prompted to enter old password for verification and confirmation then directed to enter new password after being given a list of password requirements.<br><br>\| CAMS > Staff > Change password<br>Enter old password: Password123!<br>Password must be:<br>i. more than 8 characters<br>ii. contain at least one lower case and one upper case letter<br>iii. contain at least one digit<br>iv. contain at least one special character from the following: ,.<>/:;!@#$%^&*()-_+=]<br>Please set a new password: Password123!<br>Please enter the new password again: Password123!<br>Saving new password<br>Logging out...<br>Password reset successfully!<br>Please login again<br>*Figure 3.2.1a Changing password experience for Staff* | Pass |
| 2 | [Create/Edit/Delete Camps]<br>1. Creating 1 camp<br>2. Editing Camp | 1. The user is first asked how many camps to create and then asked to key in camp details for implementation of the camp. Camps are created successfully and there is an increment in the number of camps.<br>2. User is asked to enter Camp ID he/she wishes to edit and then allowed to make edits to camp details. | Pass |

| | | | |
|---|---|---|---|
| | 3. Deleting Camp created | 3. User is asked to enter camp ID he/she wished to delete and then asked to key in Y/N to reconfirm.<br><br>\| CAMS > Staff > Create new camps<br>Enter the number of camps to create: 1<br>Creating project 1<br>Camp title:<br>Camp3<br>Input number of days of camp:2<br>Please input all dates in dd/mm/yyyy format<br>Enter date for day 1 of camp: 15/09/2023<br>Enter date for day 2 of camp: 16/09/2023<br>Closing date for signups: 24/08/2023<br>Please indicate if the camp is open to all schools (Y/N):<br>Y<br>Camp is open to all schools.<br>Location: NBS<br>Total number of slots: 2<br>Description: schoolcamp<br>Set visibility (Y/N): Y<br>Press Enter key to continue...<br>*Fig3.2.2b Staff experience for creating camps* | |
| 3 | [View and Approve Suggestions]<br>1. View suggestions<br>2. Approve Suggestions | User is asked to choose whether he/she wants to view or approve suggestions first<br>1. List of suggestions is displayed<br>2. User is asked to key in suggestion number he/she wishes to approve and is then allowed to approve suggestion chosen | Pass |
| 4 | [View and reply to enquiries]<br>1. View Enquiries<br>2. Reply Enquiries | User is prompted to choose whether to view or reply to enquiries<br>1. User is able to view the whole list of enquiries submitted<br>2. User is asked to enter enquiry no. he/she wished to reply to and is able to give a reply. | Pass |
| 5 | [Generate report of list of students attending each camp] | User prompted to choose which camp to generate a report of list of students for and report is generated. | Pass |
| 6 | [Generate performance report of camp committee members] | User prompted to enter camp ID and then performance report of committee members for that camp is generated. | Pass |

## 3.3 Student Functionality

| **Test Case** | **Test Description** | **Expected Result** | **Pass/Fail** |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 1 | [Change Password] | User is prompted to enter old password and then given a set of requirements for new password. Student is asked to re-enter new password to validate and then redirected to relogin. | Pass |
| 2 | [View Available camps] | List of available camps is displayed with Camp ID and name | Pass |
| 3 | [Register for a camp] | Student is given list of camp ID and camp name and asked to enter ID of camp he/she wishes to register for and is then registered | Pass |
| 4 | [View Registered camps] | a. Student has registered for camps: List of registered camp ID and name is displayed<br>b. Student has not registered for any camps: Message is displayed that student has not registered for any camps. | Pass |
| 5 | [View Enquiry status and enquiries sent] | List of enquiries sent by student along with status is displayed. | Pass |
| 6 | [Withdraw from camps] | Student is prompted to enter campID of camp he wants to withdraw from list of camps he registered for along with their CampID and names | Pass |
| 7 | [Edit enquiries] | Student is asked to enter enquiry no. of enquiry he wants to edit and is then allowed to enter his updated enquiry. | Pass |

## 3.4 Camp Committee Functionality

| **Test Case** | **Test Description** | **Expected Result** | **Pass/Fail** |
|---|---|---|---|
| 1 | [Change Password] | Same as test case in 3.3.1 | Pass |
| 2 | [View/Reply to Enquiries] | (a) List of enquiries is displayed for camp that camp committee member is assigned to<br>(b) User asked to input enquiry no. He/she wishes to reply to and is then prompted to enter reply. | Pass |
| 3 | [View/Edit/Delete Suggestions] | (a) User is able to view all suggestions and suggestion number. | Pass |

|  |  |  |  |
|---|---|---|---|
|  |  | (b) User is prompted to enter suggestion no. And allowed to re-enter updated suggestion<br>(c) User is prompted to key in suggestion no. and suggestion is deleted and suggestion number decreases by one. |  |
| 4 | [Generate a report of the list of students attending each camp that they oversee] | User able to view list of students attending each camp they oversee after report is generated. | Pass |
| 5 | [View Camp details of camp they have registered for] | User can view camp details like dates and camp name for camp they have registered for after being prompted for camp id. | Pass |
| 6 | [Get one point for each enquiry replied and each suggestion given. One extra point will be granted for each accepted suggestion.] | Point incremented by one for camp committee member after replying to an enquiry or a suggestion.<br>User point increments by one again when his/her suggestion is approved. | Pass |
| 7 | [Submit Suggestions] | User is prompted to enter suggestions and suggestion count is incremented by one. | Pass |

## 3.5 Error Handling

| Test Case | Test Description | Expected Result | Pass/Fail |
|---|---|---|---|
| 1 | [Invalid Inputs] Entering an invalid input when selecting userID, campID, enquiryID and suggestionID | A message that prompts the user to input a valid input is displayed | Pass |

# 4. Reflection

## 4.1 Difficulties encountered

**Firstly**, we found it difficult to ensure our design follows the single responsibility principle as many functions tend to overlap especially when they are related to the same object. For instance, viewing available camps and camps registered were two very similar functions but instead of compiling them into one class as two methods, we implemented two separate function-specific classes to stick to SRP. **Secondly**, we faced difficulty in maintaining data integrity while keeping global state of our application. Hence, we used foreign key associations to store the relationships between entity classes to ensure smooth modification of classes without having to update multiple external classes due to existence of a singular updated entity class. **Lastly**, we faced challenges in implementing different operations for methods of the same base function such as viewEnquiries and getAllCamps. Therefore, we used polymorphism to dynamically override the methods in subclasses to suit operations specific to the derived class.

## 4.2 Knowledge Learnt

Through this journey, we have developed a better understanding of OOP concepts and SOLID principles, and even design considerations not within our syllabus like observer or adapter pattern. We learned the importance of having a simplified, and well-structured model for our application as it facilitates better readability and maintainability among both users and programmers. Structure was needed not just within the application but in the development portion as well. Having a systematic approach by constructing a UML diagram beforehand allowed us to have a clearer vision of our product while working together and input clearer descriptions of specific functions and models. We also learned the importance of implementing proper naming conventions like 'ICampView' whereby the 'I' indicates that it is the interface of CampView. This is also applied in package naming like services and controllers to enable easier separation of function-specific classes. Lastly, we realized the need for abstraction of data layer to ease the process of accessing and updating data files anywhere in the program.

## 4.3 Further Improvement

**Firstly**, security features in our system can be improved, especially data protection. For example, like many other interfaces in today's world, we can implement the hashing system for our password. **Secondly**, our system only considers present users of the system. Hence, we can improve this by allowing the system to register new users. **Thirdly**, our interface can be made to be more user-friendly such that the functions for the respective users can be clearer. This can be through the inclusion of a more detailed description of various functions. **Lastly**, our design can follow abstraction and SRP more closely by creating more specific packages and classes. Classes in the interface package, for example, can be split into more specific packages like `EnquiryInterface` or `SuggestionInterface` to enable ease of maintainability.

# Contributions

| Student | Contribution: (Task) |
|---|---|
| Zhang Siyu | • Javadoc generation<br>• Service package |
| Phyo Sandar Win | • UML class diagram<br>• Stores package |
| Hong Zhi Hao | • Project manager<br>• Controller package, Model package, Interfaces |
| Harvande Advika Sandeep | • Report<br>• View package |
| Venus Ng Min Jia | • Javadoc<br>• Util package, Enums |