**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 | Lab Group | Signature/Date |
| --- | --- | --- | --- |
| Ng Yao Hong | SC2002 | SCS4 | 26 Nov '23 |
| Shannamae Yang-Tay | SC2002 | SCS4 | 26 Nov '23 |
| Thomas Tan | SC2002 | SCS4 | 26 Nov '23 |
| Tyler Goh | SC2002 | SCS4 | 26 Nov '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.

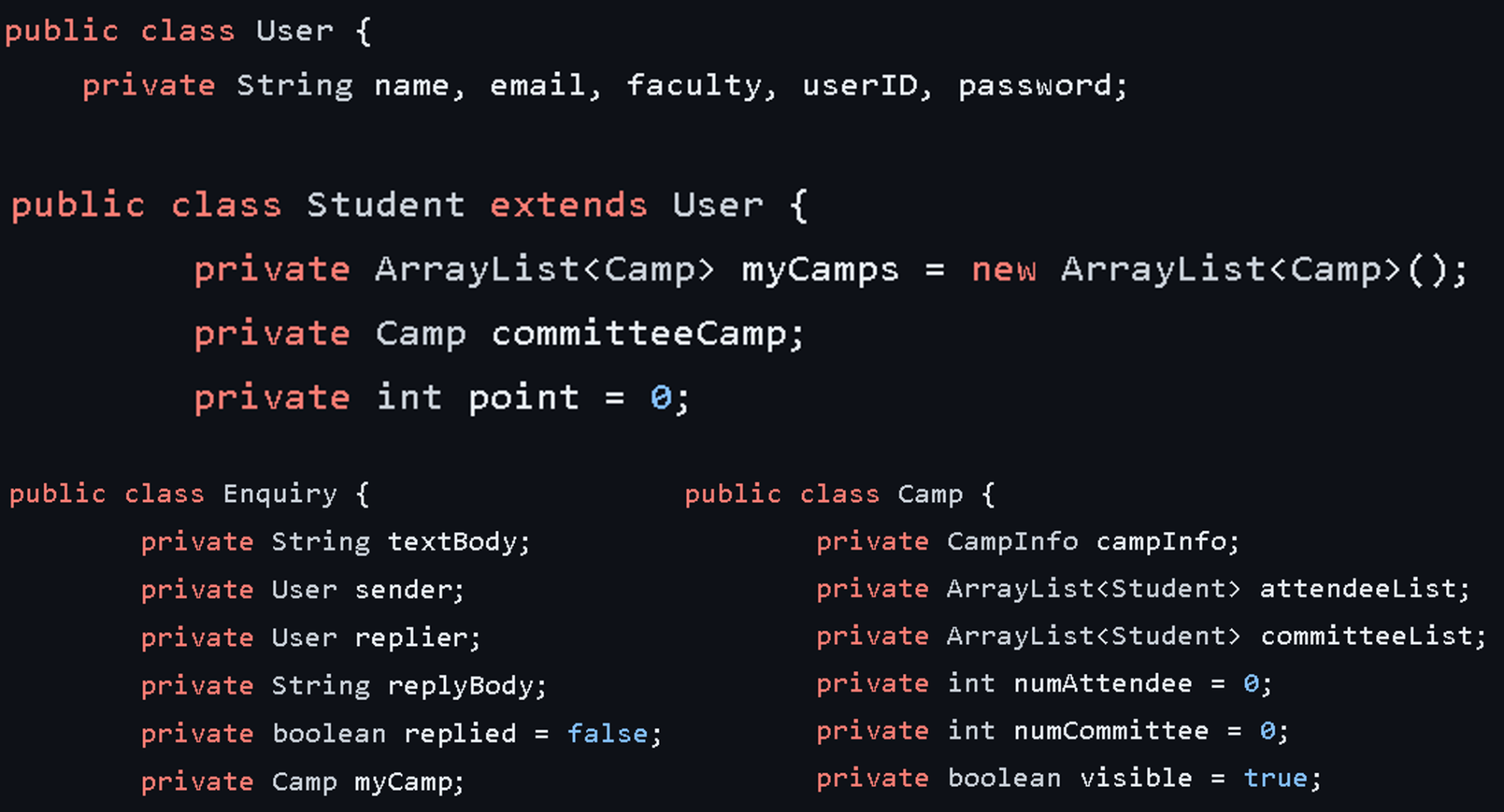
# I. Design Considerations and Use of OO Concepts

## *Introduction* and Approach Taken

For the development of our Camp Application and Management System (CAMs), we chose to use a modular approach to increase extensibility and maintainability. Our application is designed with a focus on encapsulation, low coupling and high cohesion through the application of SOLID principles. We also make use of inheritance when necessary to reduce code reuse.

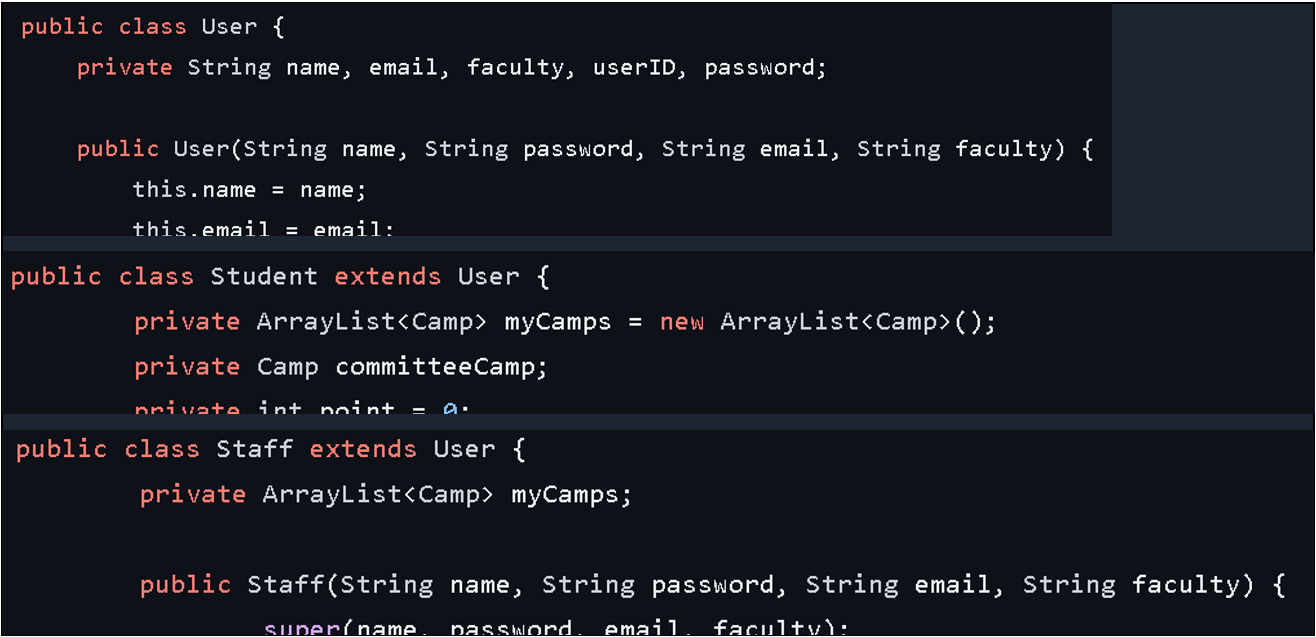
## Design Principles

To protect data confidentiality, our design encapsulates all individual object's data through the *private* accessibility modifier in Java. For example, our User class which holds a user's sensitive private details such as email and password are all hidden and can only be accessed and modified through the public *get* and *set* methods. This is also evident throughout the remaining classes, as can be seen from Fig. 1. Through encapsulation, we ensure that only Classes with a "need to know" requirement can access such private information. An instance of this would be camp committee Student or Staff objects accessing a Camp's information only through special method calls, thus guaranteeing security of the hidden attributes.



*Fig 1. Examples of encapsulation in our implementation used throughout our design*

In the creation of our User class, we made use of inheritance in OOP to prevent extensive duplication of code. For any User in our CAMS, they will share some common attributes such as email and password, and have common functions such as change password. Hence, we created a general User superclass from which we can derive unique subclasses from, such as Staff and Student. Doing so allows changes to be made to the User superclass and be inherited by its subclasses. This also allows us to focus on creating functions that are unique to these subclasses rather than reusing old code. We showcase this in Fig 2.



*Fig 2. Inheritance of User superclass through Java's* ***extends***

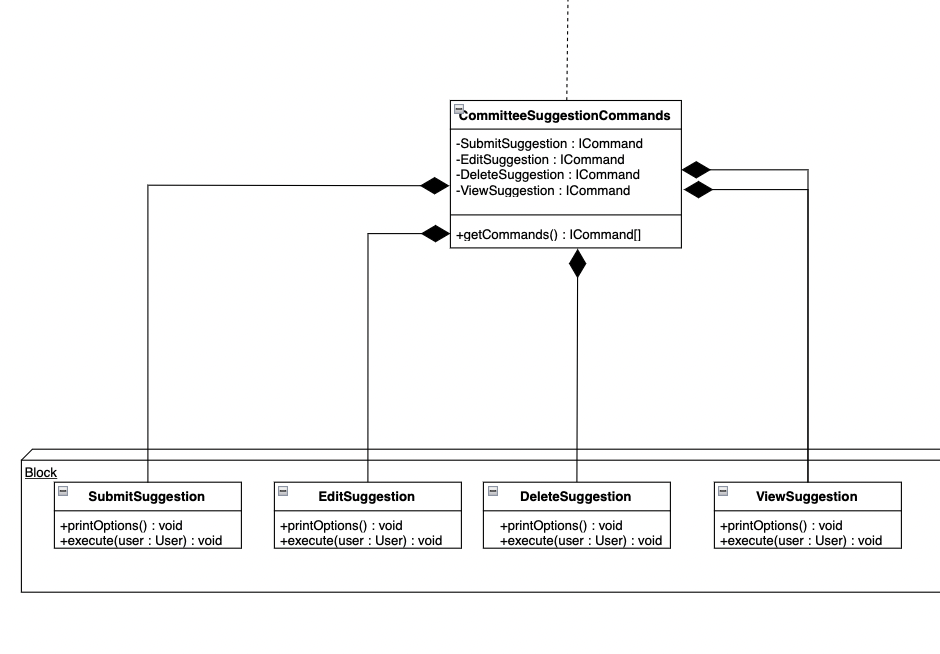
Additionally, our design also makes extensive use of SOLID principles to further enhance the extensibility and maintainability of our application.

We made use of the Open-Closed Principle (OCP) to allow for flexibility in user types and our application's functionalities. This is done through the use of abstraction via extending the abstraction layer. For instance, should there be a need for another new user type, such as an external 3rd party vendor to provide services for the camps, we can simply extend the User class to provide for specialised functions for vendors. This allows us to manipulate what each user type does or add more user types without modifying the fundamental functionality and identity of User, such as changing passwords.

Furthermore, this also adheres to Liskov Substitution Principle (LSP) by ensuring that objects of the superclass (e.g. User) can be replaced by its subclasses (e.g. Student or Staff) without breaking the functionality of our application. This polymorphism in allowing different user types to be used interchangeably improves the scalability and flexibility of our program.

We also made use of the Single Responsibility Principle (SRP) to ensure that our classes have high cohesion. Our classes are defined in such a way that it is focused in its purpose and not versatile in what they do. The attributes and methods defined for our classes exist only for the sole purpose of fulfilling the class's responsibility. For instance, our User classes such as Student or Staff do not handle the creation and management of enquiries although it is in their purview. Instead, they call upon the Enquiry class which is designed specifically to handle enquiries only.

As a result of each class's specialisation, coupling is reduced. Since each class is responsible for only 1 function, we can edit a specific class without affecting the functionality of unrelated classes. This simplifies debugging and updating of the code as fewer classes are affected. For example, in Fig 3, SubmitSuggestion, Edit Suggestion, DeleteSuggestion and ViewSuggestion are separated into individual classes so that editing the function of SubmitSuggestion will not affect the remaining 3 unrelated classes.



*Fig 3. Application of Single Responsibility Principle in our commands class*

Another consideration we had for our application is the simplification of the menu. Rather than displaying a long-list of options for the users to choose from, we chose to create separate menu bars, each with its own commands. This allows for better readability and user-friendliness. To do so, we made use of Interface Segregation Principle (ISP). We implemented each command for each user type separately, through the ICommand interfaces (ref. to Fig. 4.). This allows for each specific User class to use only the specific methods they require under the ICommand interface. This avoids wastage of computational resources by keeping unnecessary methods in system memory whilst the code is running.

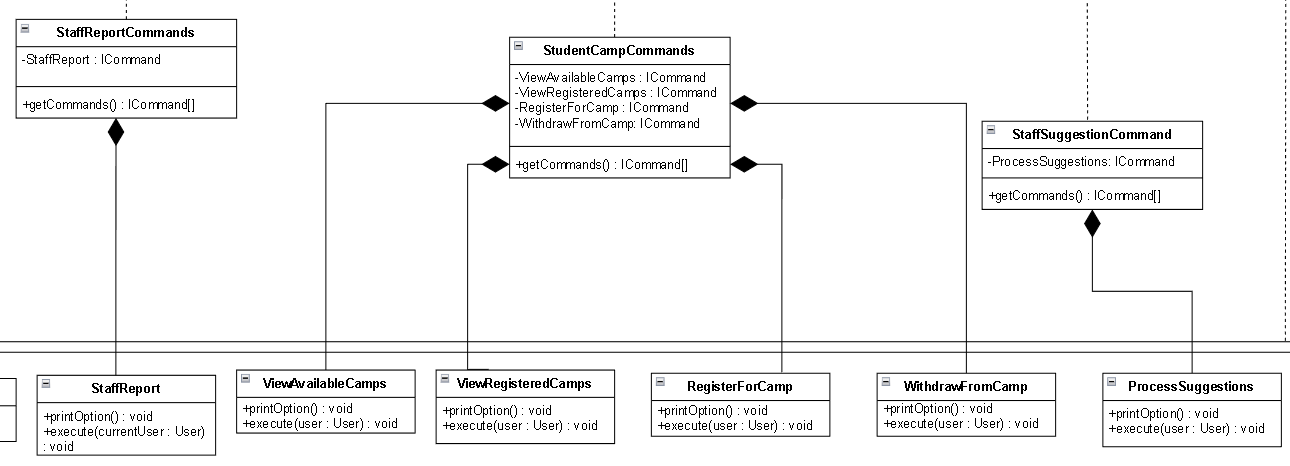


Fig. 4. Commands classes under the ICommand interface

Finally, through the use of the Dependency Injection Principle, our higher level modules, such as the Commands and ui classes, are unaware of the actual implementations of the lower level classes' methods. This allows our higher level modules to be used independently of the lower level modules. Without such dependencies, our design ensures low coupling, and any changes to lower level modules will not affect the higher level ones. This promotes reusability without needing to import all the unnecessary baggage. Overall, our design approach follows a framework-esque design, with little dependencies and coupling between classes, as can be seen from our UML diagram. This promotes high modularity in our application's various functions.

# II. Detailed UML Class Diagram

*Refer to pdf titled "SCS4-NgYaoHongGrp-UML" in attached folder*

# 

# III. Testing (Test Cases and Results)

**a. Login**

| 1. Cannot login: invalid user or valid ID but wrong password |  | | |
| --- | --- | --- | --- |
| 2. After login successfully, different menu list (main page) displayed for different user category, staff, students, and camp committee. | Student | Staff | Camp Committee |
| 3. Change password for 1st time user |  | | |

**b. main page for a Student**

| Change password: |  | Submit enquiries for a camp: |  |
| --- | --- | --- | --- |
| View available camps:  +  Select camp to register as attendee or committee |  | View registered camp: |  |
| View reply to enquiry: |  | Request to withdraw from camp: |  |

**c. main page for a Staff**

| Change Password |  | | |
| --- | --- | --- | --- |
| Create/edit/view camps all camps |  | | |
| Edit: (only able to edit camps he/she has created; a staff should not be able to edit camps from other staff).  Input choice from ‘Create’, ‘Edit’, ’View’ to choose function |  | | |
| View list of students that have registered for the camp as attendees or committee can be viewed.(from report) |  | | |
| View Suggestions to changes to camp details from camp committee.  +  Accept/Reject suggestion |  | | |
| Generate report for each camp  Camp report with the list of students attending each camp |  | | |

**d. main page for a Camp committee member**

| 1. All the functions that student main page have | ref. to a. 2. for the login page functions between a Student and Camp Committee | |
| --- | --- | --- |
| 2. Submit suggestions to staff for changes to camp details |  | |
| 3. View and reply to enquiries from students from the camp he/she oversee |  | |
| 4. View, edit and delete the details of his/her unprocessed and/or processed suggestions | View my suggestions | Edit |

# 

Our test cases are robust and demonstrate the full functionality

# IV. Reflection

## Difficulties Encountered

Our system initially violated the fundamental 5 design principles due to our classes carrying multiple responsibilities and a lack of interface utilisation. As we aimed to enhance our system and implement changes, the editing process became challenging, leading to a cascade of modifications across related classes for each alteration made. Consequently, a comprehensive restructuring was necessary to align with these essential design principles.

To enhance the system's design, a significant amount of time was devoted to addressing the five core design principles. We meticulously edited and restructured our classes to align with the single responsibility and open-closed principles. Additionally, we introduced interfaces for our commands, ensuring compliance with the interface segregation and dependency injection principles.

Once these principles were integrated, the system underwent substantial improvements with greater ease of editing and enhancement.

## 

## Overcoming Challenges

In response to our system's initial violation of design principles, we opted to restructure our system’s UML Class Diagram. Our collective group discussions helped identify the most effective approach to enhance the system's alignment with these principles.

Upon revising the UML Class diagram, we meticulously segregated responsibilities among the team members according to our system’s single-responsibility classes. This restructuring paved the way for a clearer depiction of tasks for each class, enabling us to significantly improve the system with greater ease thereafter.

## 

## Knowledge Gained

We acknowledge that the utilisation of UML Class diagram, with adherence to the 5 design principles, is highly beneficial as it offers readers an overview of the system's functionality and significantly aids programmers during system implementation. When our system aligns with these design principles, extending or modifying it becomes more manageable without necessitating changes to other related classes. Additionally, it streamlines the segregation of responsibilities, especially within a team setting, enabling each member to concentrate on their dedicated single-responsibility classes. Moreover, this approach enhances program readability as each class on our system depicts a single specific task clearly.

## Improvement Suggestions

While our system adheres to the 5 design principles, there's room for improvement in further segregating our classes into more specific tasks. Increasing the use of more segregated interfaces could further bolster compliance with the interface segregation and dependency injection principles.

For our menu functionalities, we implemented a change password function twice, once in the LoginApplication class and again in the ChangePassword class under the ICommand package. This is to avoid needing to import both the entire menu ui and the ICommand package into LoginApplication for a trivial and short function. Nevertheless, there is a minor reuse of code here and could be refactored.

Additionally, to enhance user privacy, implementing hashed passwords for each user would significantly improve our system's security. This approach ensures that staff members do not have direct visibility into users' passwords, thereby fortifying user privacy and data security.