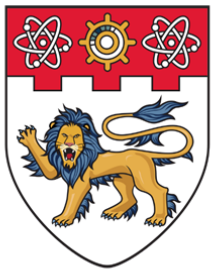


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC2002 Object Oriented Design & Programming

Title: Team Assignment (CAMS) Report

Lab: SCEC

Group: 1

Date of Submission: 26 November 2023

Group Members	**Matriculation Number**
Gokul Ramesh	U2222182H
Jared Pek	U2220146G
Qiang Zhiqin	U2222407C
Rachel Phuar Yi Ling	U2220042B
Tan Yoong Ken	U2220061G

Table of Contents

1 - Design Considerations

1.1 - Approach

The Camp Application and Management System (CAMs) implements a camp enrolment and management system for both students and staff members of NTU. It is split into both high-level and low-level packages that are responsible for different aspects of the application.

The high-level packages include the view package, which is responsible for providing both student and staff users with a user-friendly interface to interact with, whereas low-level packages include the user, camp, enquiry and suggestion packages, which are responsible for implementing their respective object entities, and for users to seamlessly interact with.

We also implemented control classes that store an array of their respective objects, and manage the interaction between the user and the objects. These control classes also enforce permission control and exception handling to ensure security and manipulation of the objects appropriately.

This splitting of each object implementation into different packages encapsulates them from each other, and reduces coupling of these services in the system, increasing overall scalability and extensibility.

1.2 - Assumptions

- There is no concurrent usage of the CAMs system.
- Staff members in charge of the camp can generate both a participant report, as well as a performance report of the committee members.
- Staff members can create camps for any faculty in NTU.
- Students can only ever be involved in 1 camp as a committee member.

1.3 - Design Principles

1.3.1 - SOLID Principles

S - Single Responsibility Principle

Under this principle, each class will only have 1 responsibility, and hence only has 1 reason to change. This minimises the coupling of different services, which improves the scalability and extensibility of our classes, and our application.

To demonstrate this principle, each package has a control class whose sole responsibility is to manage the list of objects composed within it via API function calls. Helper classes such as the build, edit and select classes are also implemented to fulfil individual responsibilities and further enforce this principle, and ensure that each class is only responsible for one action.

```
public void add(User user) {
    if (!isStaff(user)) return;
    camps.add((new CampBuild()).build(user));
    (new CampSort()).sortByAlphabetical(camps, orderOption:0);
}
```

```
public void edit(User user) {
    if (!isStaff(user)) return;
    ArrayList <Camp> createdCamps = getByStaff(user);
    Camp camp = (new CampSelect()).select(createdCamps, user.getUserID());
    (new CampEdit()).edit(camp);
}
```

Fig 1: add() and edit() methods in the CampControl class, which simply make the required function calls, instead of having to handle the building and editing of objects

O - Open Closed Principle

Under this principle, each class should be open for extension, but closed for modification, so we can add more features to the class without changing the original source code of our modules.

To demonstrate this principle, we implemented an abstract Message class, which closes it for modification, and extend via the Enquiry and Suggestion subclasses which inherits from the Message class. The Enquiry and Suggestion classes also implement their own class-specific methods and features.

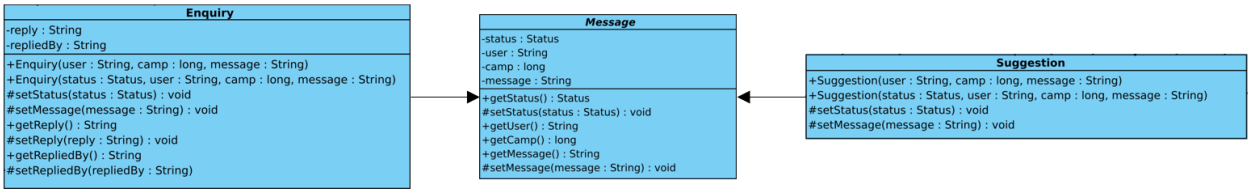


Fig 2: The abstract (indicated by *italics* in Visual Paradigm) Message class is closed for modification and is open for extension since it is inherited by the Enquiry and Suggestion subclasses

L - Liskov Substitution Principle

Under this principle, objects of superclasses should be replaceable with objects of their subclasses without breaking the application.

Our implementation of the Message abstract class and its Enquiry and Suggestion subclasses are compliant with this principle since Message objects can be easily substituted by Enquiry or Suggestion objects, while providing their own implementation of common methods like the print() method without any modifications to the Message class.

I - Interface Segregation Principle

Under this principle, each class should only implement interfaces that they will use since they would be more client specific, instead of being one general purpose interface.

To demonstrate this principle, our EnquiryControl classes implements multiple interfaces – IControl and IEnquiry, which outline the methods that it should implement to perform its responsibilities. Other interfaces like ICamp are not relevant to it and are not implemented.

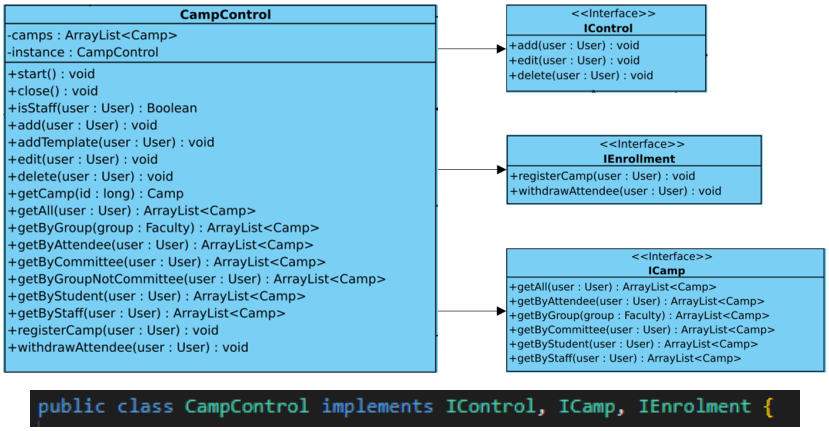


Fig 3: CampControl implements the IControl, IEnrollment and ICamp interfaces, which are the only interfaces relevant for its responsibilities

D - Dependency Injection Principle

Under this principle, high-level modules should not depend on low-level modules, but instead on abstraction, making high-level modules more flexible and less coupled.

To demonstrate this principle, we implemented a common interface called IControl for our higher-level control classes, which outlines relevant methods that these control classes must implement to manage their respective collections of objects. These control classes need not know exactly how the objects will be manipulated, and will only be concerned with implementing the required functionality.

1.3.2 - Singleton Principle

Under this principle, only 1 instance of a class should exist throughout the entire Java Virtual Machine, and there should be a global access point to access the object.

To demonstrate this principle, we implemented static and public instances of our control classes within themselves, for example, in the SuggestionControl class, there is a static and public attribute to a SuggestionControl object, which could then be accessed anywhere in the

application. No new instances of the control classes would be instantiated anywhere in the application, ensuring that only 1 instance of each control class exists.

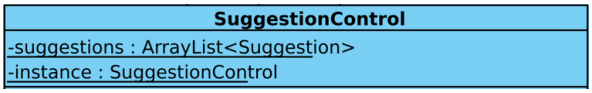


Fig 4: The SuggestionControl class includes a static 'instance' to a SuggestionControl object, ensuring global access

1.3.3 - Enumeration Based State Tracking

In addition to the SOLID principles, we also utilised enumerables to enforce a standardised state throughout the application. This allows for easy data input and code debugging, as well as enhanced code readability and maintainability.

This is demonstrated via our implementation of multiple enumerables, such as the User Faculty, Message Status and Camp Roles.

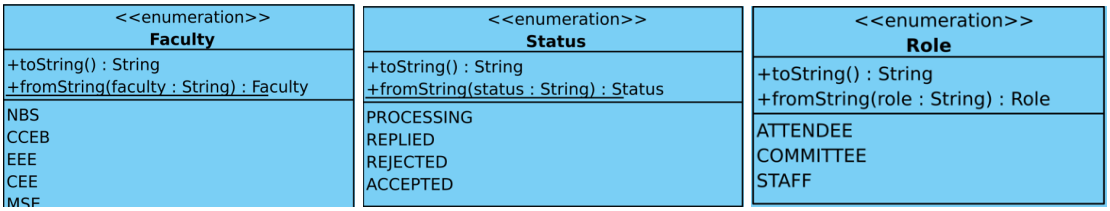


Fig 5: The User Faculty, Message Status and Camp Role enumerables (from left to right)

1.4 - OOP Principles

1.4.1 - Abstraction

In the CAMS system, control classes act as an abstraction layer between the view layer and the underlying object implementations. They provide a simplified interface for users to interact with the system without needing to know the complicated details of how the objects are stored and manipulated. For example, our EnquiryControl, SuggestionControl, CampControl and UserControl classes handle the main functionalities of the system, while the user does not need to know the inner workings of our object implementations.

1.4.2 - Encapsulation

Encapsulation plays a crucial role in the CAMS system by ensuring that each package is self-contained and independent, minimising dependencies between packages and promoting modularity. This approach enhances the system's maintainability, extensibility, and reusability.

This is done via our package structure, where each package encapsulates its own data and methods. For example, the user package encapsulates user-related data and methods, including user authentication, password and profile management, providing a clear separation of concerns and allowing other packages to interact with user objects without knowing the specifics of user storage and manipulation. This facilitates efficient development and testing of the system.

This is also done via protected, package-level object mutators, which enhances security and encapsulation. This approach ensures that both developers and users have no way to directly modify the attributes of these objects, and must go through the required permission handling in the respective control classes for any object modifications.

1.4.3 - Polymorphism

This principle allows objects of different classes to take on multiple forms, or respond to the same method call in different ways. It is particularly useful for handling diverse object types in a consistent manner and promoting code reusability.

This is demonstrated via method overloading. For example, the build methods in UserBuild class have different implementations according to different parameters in creating new user objects with different roles.

This is also done via interfaces, where different classes implement a general functionality that is relevant to them. This encourages loose coupling, which allows the system to adapt to changes in object types without disrupting the overall code structure.
For example, our control classes implement a general IControl interface which outlines the add, edit and delete methods for handling object collections. Control classes like CampControl, SuggestionControl and EnquiryControl then implement the interface and the required methods to reliably manage their respective collections of objects, while ensuring a standardised control class functionality with specific implementations.

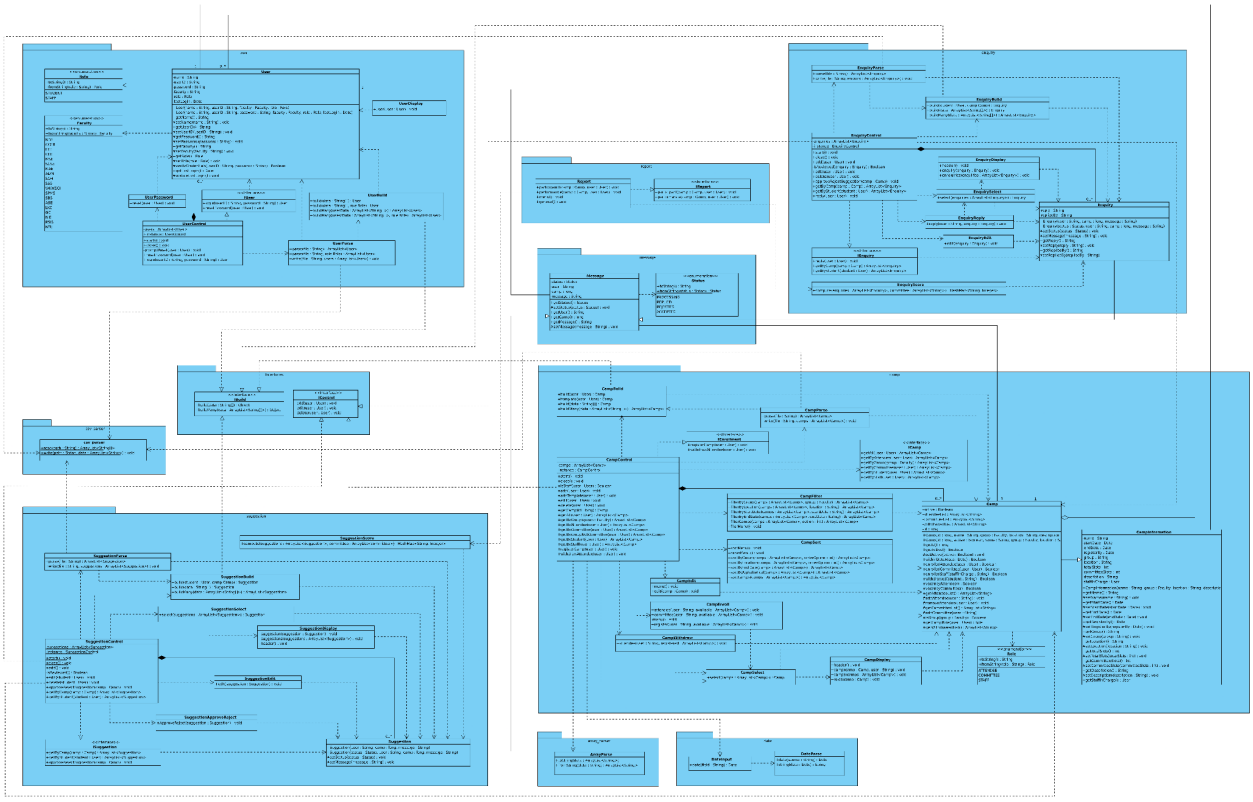
1.5 - Proposed New Features

One new feature we could include would be to allow students to enrol into multiple camps as a committee member. At the moment, each student can only ever be enrolled in 1 camp as a committee member, and permission control for this is enforced via the committee() method in the CampEnrol class.

To implement this feature, we only have to modify the if statements in the CampEnrol class's committee() method to ensure the relevant permission checking, and no other classes are affected. The convenience in implementing this modification is due to the Single Responsibility

Principle, where each class only handles 1 key responsibility, enhancing scalability and maintenance in scenarios like these.

2 - UML Diagram



2.1 - Explanation

The system is split into multiple packages, each representing a core feature of the system. Within each package, we have included 1 entity class, and 1 control class which implements a universal IControl interface for control classes, and respective entity-specific interfaces to interact with their collection of objects. This design appropriately encapsulates the data and functionality of each package, while minimising any coupling of classes by utilising a dependency relationship instead of an association relationship.

To view our UML diagram in detail, navigate to the ‘uml_diagram’ folder in the zipped file that was submitted, and select either the SVG or JPG version of the diagram.

3 - Reflections

When our group first met to discuss our UML class diagram, we mainly followed the guidelines provided and had each category of Student, User, Staff etc. as classes on their own. However, we soon encountered the problem where classes such as Student and Staff were completely empty with no attributes or methods. Certain classes like Camp would also be extremely big since it contained too many methods and attributes. In addition, all classes were created under the same package and all visibility modifiers were set to public.

While our UML class diagram looked very simple, the code was very badly organised, and due to the public visibility modifier, anyone would be able to directly access and modify attribute values of our objects. We also came to realise that most of the SOLID principles were not adhered to, leading to heavy coupling of classes and bad modularity. After encountering these problems, we made phased changes to our design, up to the one we currently have, which adheres to all the SOLID principles, and encourages scalability and modularity of our classes.

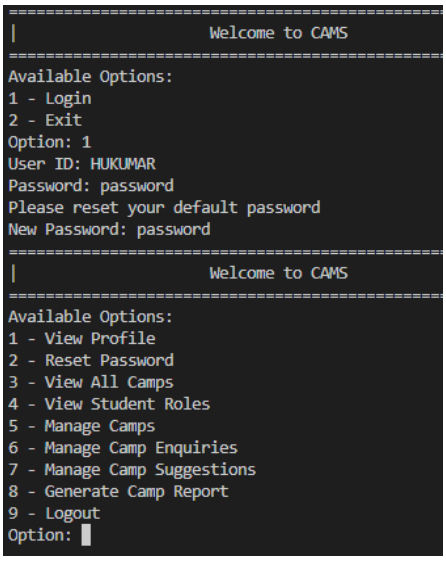
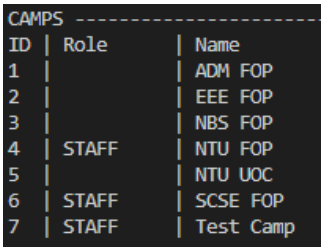
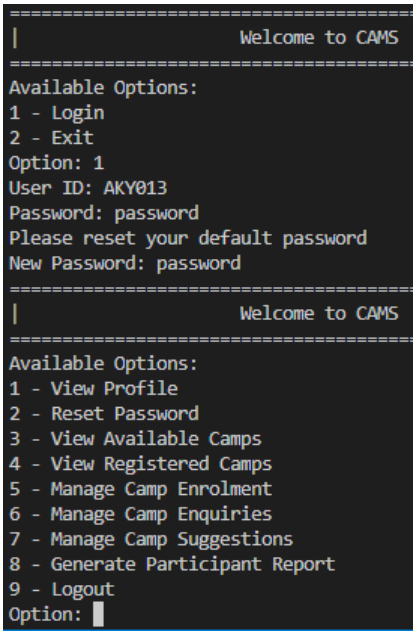
In addition to poor organisation, we also initially used object pointers as attributes in our entity classes. During our deliberation, we wanted to have easy access to the associated object's attributes when required. Comparing this to another approach, where we instead use a unique object ID, the object ID restricts the object reference to only its unique ID, and not the object itself, which was not what we had initially intended. Thus we decided to use pointers to objects as attributes.

This approach did work initially, but we soon encountered the problem where we were unable to write these pointer variables into a file for data persistence. To overcome this, we instead swapped to the other approach, where we use the objects' unique ID instead of pointers, then obtain the required object via the respective entity's control class, and change the visibility of all our mutator methods to protected. In hindsight, our new approach was much more secure and further enhanced the encapsulation of the data.

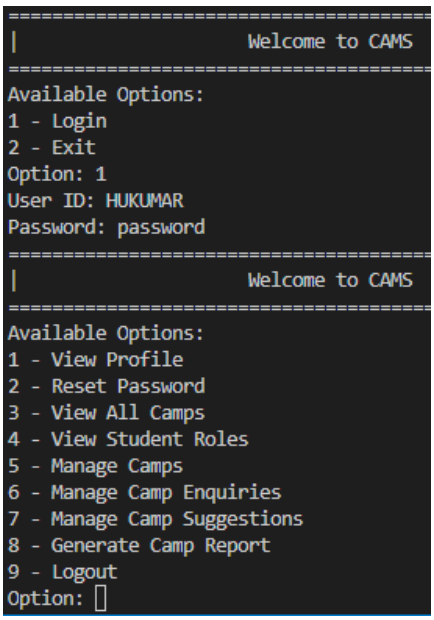
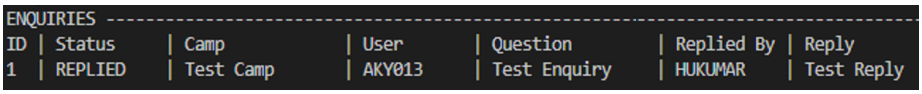
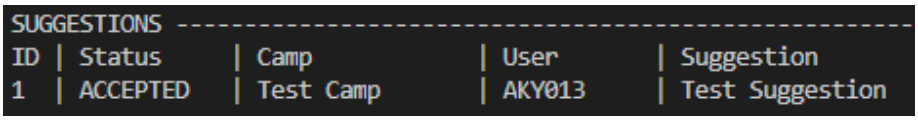
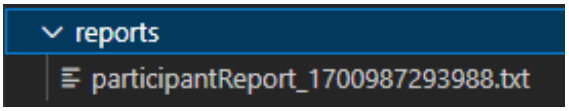
Through this project, we learnt that the design of an application is as important, if not more, in ensuring code modularity, and application scalability and extensibility. Good design is also crucial in enabling better collaboration between developers by making debugging much easier, and saving time in rewriting frequently repeated code.

To conclude, this project has taught us the importance of the designing process and we will definitely invest more time and effort into the designing process of any application in the future.

4 - Test Cases

Test Case	Expected Outcome and Results	
Login as ‘HUKUMAR’ user	Prompt for password reset if first login, then display the staff user interface.	
Adding a new Camp called ‘Test Camp’	New ‘Test Camp’ will be viewable when viewing all camps.	
Logging in as ‘AKY013’ user	Prompt for password reset if first login, then display the staff user interface.	

Create a new ‘Test Enquiry’ for ‘Test Camp’	New PROCESSING enquiry with ‘Test Enquiry’ as the question will be created for ‘Test Camp’. Available Options: 1 - View my Enquiries 2 - Add an Enquiry 3 - Edit an Enquiry 4 - Delete an Enquiry 5 - Reply an Enquiry Option: 1 ENQUIRIES ---------- ID | Status | Camp | User | Question 1 | PROCESSING | Test Camp | AKY013 | Test Enquiry
Register for ‘Test Camp’ as a committee member	Enrolled into ‘Test Camp’ as a committee member. Verifiable via the ‘View Profile’ feature. Option: 1 RAWAL (AKY013), STUDENT of SSS Last Login: 26-11-2023 15:37 Committee Of: Test Camp
Register for another camp as a committee member	Unable to select another camp to enrol as committee. Available Options: 1 - Register for a Camp 2 - Withdraw from a Camp Option: 1 Available Roles: 1 - Attendee 2 - Committee Option: 2 Already a committee member of 'Test Camp'
Create a new ‘Test Suggestion’ for ‘Test Camp’	New PROCESSING suggestion with ‘Test Suggestion’ as the suggestion will be created for ‘Test Camp’. Available Options: 1 - View my Suggestions 2 - Add a Suggestion 3 - Edit a Suggestion 4 - Delete a Suggestion Option: 1 SUGGESTIONS ---------- ID | Status | Camp | User | Suggestion 1 | PROCESSING | Test Camp | AKY013 | Test Suggestion

Re-Login as user ‘HUKUMAR’	Login successful without any prompt to change password.	
Reply to the ‘Test Enquiry’ enquiry for ‘Test Camp’	Reply to the enquiry and the userID of the person replying it appears when viewing all enquiries for the ‘Test Camp’. Status changes to ‘REPLIED’ 	
Approve the ‘Test Suggestion’ suggestion for ‘Test Camp’	Status of the suggestion will be changed to ‘ACCEPTED’ 	
Generate a participant report for ‘SCSE FOP’ without filters	A new ‘participantReport_xxxxx.txt’ will be created in the ‘reports’ folder. 	
Generate a performance report for ‘Test Camp’	A new ‘performanceReport_xxxxx.txt’ will be created in the ‘reports’ folder.