**A lion with a red and white background

Description automatically generated**

**SC2002: OBJECT-ORIENTED DESIGN & PROGRAMMING**

**AY 2023/2024 SEMESTER 1**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Project: Building an OO Application**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Lab Group:** SCEC Group 4

**Lab Instructor:** LAN MENGCHENG

|  |  |
| --- | --- |
| **Submitted By:** | **Matriculation Number** |
| Tejeswara Nehru | U2220197K |
| Jeyaprakash Sri Vikass | U2222682D |
| Shankar Mathangi | U2223221G |
| Krishna Nageswaran Jeevika | U2223941D |

**Table of Contents**

[Academic Declaration 3](#_Toc151889924)

[1. Introduction 4](#_Toc151889925)

[2. Design Considerations 4](#_Toc151889926)

[**2.1.** **Approach Adopted** 4](#_Toc151889927)

[2.2. SOLID Design Principles 4](#_Toc151889928)

[**2.2.1.** **Single Responsibility Principle (S)** 4](#_Toc151889929)

[**2.2.2.** **Open-Close Principle(O)** 5](#_Toc151889930)

[**2.2.3.** **Liskov Substituion Principle (L)** 6](#_Toc151889931)

[**2.2.4.** **Interface Segregation Principle (I)** 6](#_Toc151889932)

[**2.2.5.** **Dependency Inversion Principle (D)** 7](#_Toc151889933)

[2.3. Assumptions 8](#_Toc151889934)

[3. UML Class Diagram 8](#_Toc151889935)

[4. Program Testing 9](#_Toc151889936)

[5. Reflection 13](#_Toc151889937)

# Academic Declaration

**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 (CE2002 or CZ2002) | Lab Group | Signature/ Date |
| Tejeswara Nehru | SC2002 | SCEC | Tejeswara / 25-11-2023 |
| Jeyaprakash Sri Vikass | SC2002 | SCEC | Vikass / 25-11-2023 |
| Shankar Mathangi | SC2002 | SCEC | Mathangi / 25-11-2023 |
| Krishna Nageswaran Jeevika | SC2002 | SCEC | Jeevika / 25-11-2023 |

Important notes:

Name must **EXACTLY MATCH** the one printed on your Matriculation Card.

# Introduction

This project explores the integration of object-oriented programming and design principles in creating a robust Camp Management application. The report outlines our methodology, highlights key test cases, and reflects our learning process throughout the development journey. Join us as we navigate building an Object-Oriented Application for effective Camp management.

# Design Considerations

## **Approach Adopted**

In pursuit of enhancing the **maintainability, comprehensibility, and scalability** of our object-oriented programming, we implemented **SOLID design principles** alongside key object-oriented concepts such as **Encapsulation, Abstraction, Inheritance**, and **Polymorphism**. This approach fosters high cohesion and loose coupling within our codebase, ensuring that future improvements can be seamlessly integrated into the application. By effectively managing dependencies between classes and packages, we mitigate the potential ripple effects of changes, minimizing impacts on other parts of the software. For example, the password check functionality is implemented inside the user class instead of Data Manager which helps to enhance the security of the code and provide better encapsulation. This also provides high cohesion to the Data Manager as now it only needs to manage the login accounts and user objects.

# SOLID Design Principles

## **Single Responsibility Principle (S)**

To adhere to this principle, we ensure that each class is dedicated to a single responsibility, allowing changes to be driven by a singular reason. This approach minimizes dependencies, resulting in lower coupling. Precise and descriptive names are assigned to each class to enhance implementation clarity. For example, distinct classes are created for specific request types, like "*ViewAllCamps*" and "*WithdrawFromCamps*," ensuring that each class is tailored to handle only one type of request. The following is a compilation of our various classes created for their specific purposes.

Classes which are created to handle Single Responsibility Principle:

1. ***AvailableCamps*** to check the available camps for students.
2. ***RegistractionCamps***to deal with registration functions associated with students and Camp Committee Members
3. ***Authority*** to deal with the enquiries and suggestions submitted by the attendees.
4. ***ViewAllCamps*** to list all the camps which are ongoing in NTU.
5. ***SuggestionsHandler*** to handle the Suggestions Inbox available to the staff.
6. ***ReplyHandler*** for camp committee members and staffs to reply to the enquiries in his Suggestions Inbox and also for the students to view replies.
7. ***WithdrawFromCamps*** to handle the withdrawing feature for the students to withdraw from the camps registered by them.
8. ***DateConfiguration*** for validating accurate dates and handling errors in date input.
9. ***ReportGenerator*** to generate the available reports according to the format selected
10. ***Query*** to deal with all kinds of enquiries and suggestions feature submitted.
11. ***CampListGenerate*** helps to list the camps in a more sophisticated manner.
12. ***ViewingCamps*** to view and list all the camps created by the staff.
13. ***InputErrorChecking*** to check and catch all kinds of input errors from user.

## **Open-Close Principle(O)**

Our Camp Management Application design utilises **abstraction** and the **open-closed principle** to achieve **reusability** and **extensibility**. For example, our interface “*ReportSelector*” have subclasses “*EnquiryReport*”, “*AttendanceReport*” and “*PerformanceReport*” which implements it. If we want to add another kind of report to our system, instead of modifying the report generator or the staff or camp committee member class who generates the report, we can just add another class which implements the “*ReportSelector*” interface. This adheres to the OCP because you can extend the system by adding new classes (through interface implementation) without modifying the existing classes or the main class. In this way, the use of an interface promotes a design that is open for extension (by adding new subclasses) while remaining closed for modification (since changes to existing classes are not necessary).

Classes which are created to handle Open-Close Principle:

1. ***User*** to allow us to add new types of users to join the camp.
2. ***ReportGenerator*** to allow students to also generate reports.
3. ***ReportSelector*** as mentioned in the example above.
4. ***EnquiryReportSelector*** to add another format for generating enquiry report.
5. ***AttendanceReportSelector*** to add another format for generating attendance report.
6. ***PerformanceReportSelector*** to add another format for generating performance report in future.
7. ***Authority*** to give authority to students in future to perform certain tasks.
8. ***ReplyHandler*** if we want to add another feature like replying to suggestions.
9. ***WithdrawFromCamps*** to add another feature by allowing the camp committee member to withdraw from camps.

## **Liskov Substituion Principle (L)**

In upholding the Liskov Substitution Principle, it is essential that every derived subclass is interchangeable with its base class. In our design, both "*Student*" and "*Staff*" act as subclasses of the base class "*User*," while "*CampCommitteeMember*" is a subclass of "*Student*." All subclasses implement the necessary methods inherited from "*User*"; for instance, each subclass's constructor incorporates the parameters of the "*User*" constructor. This design ensures seamless interchangeability between instances of the derived user subclasses and the "*User*" base class, maintaining the program's overall functionality without any adverse effects.

## **Interface Segregation Principle (I)**

In adhering to the Interface Segregation Principle (ISP), our design prioritizes specificity in the functionality of each interface. For example, we dedicated interfaces "*EnquiryTextReportFormatSelector*" and "*EnquiryCsvReportFormatSelector*" exclusively designed for generating enquiry reports in text and CSV formats, respectively. Similarly, the "*RegistrationCamps*" interface serves the sole purpose of registering users for the camp.

To further comply with ISP, we rely on the "*ReportSelector*" interface, which encompasses three distinct classes: "*EnquiryReportSelector*," "*AttendanceReportSelector*," and "*PerformanceReportSelector*." Each class is tailored to a specific type of report, ensuring a focused and modular structure. Additionally, we address diverse formatting needs through dedicated report format selector interfaces like "*TextFormatSelector*" and "*CSVFormatSelector*." Each of these interfaces consists of two classes, accommodating the varied formatting requirements within our project. ISP infusion is evident in the strategic design of two specialized interfaces, namely, "*StaffViewableReportSelector*" and "*CommitteeMemberViewableReportSelector*." Both interfaces extend the more general "*ReportSelectorInterface*," ensuring that each interface aligns precisely with the specific functionalities required by staff members and committee members. The "StaffViewableReportSelector" interface provides methods for generating three distinct types of reports, catering to the expanded functionality available to staff members. Conversely, the "*CommitteeMemberViewableReportSelector*" interface is tailored to include methods for generating only two types of reports, reflective of the specific capabilities granted to committee members.

This meticulous structuring not only promotes a clear and concise interface for each user role but also prevents the imposition of unnecessary methods on classes that do not require them. Consequently, our design maintains a higher degree of cohesion, reducing the likelihood of unintended dependencies and contributing to a more modular and scalable system. The purpose-specific nature of our interfaces aligns seamlessly with the Interface Segregation Principle, enhancing the maintainability and extensibility of our project.

## **Dependency Inversion Principle (D)**

The Dependency Inversion Principle is applied to ensure that higher level modules do not depend on lower-level modules, but instead depend on abstractions and not concrete methods. we introduced interfaces like "*AvailableCamps*" to establish a separation between higher-level modules, such as "Camp," and the lower-level modules like "*StudentAvailableCamps*." This design choice allows the higher-level module to depend on the abstraction provided by the interface rather than directly on the specific implementations in the lower-level classes. Consequently, the dependencies are inverted, aligning with the principles of DIP, and facilitating a more flexible and maintainable system.

Another illustration involves the hierarchy comprising the "User" base class and its subclasses: "*Staff*," "*Student*," and "*CampCommitteeMember*," with the latter extending the "*Student*" class. Adhering to the Dependency Inversion Principle, this principle allows for the extension of additional subclasses without necessitating modifications to the parent base class. In this scenario, concrete classes reside at the lower-level modules, and the base classes remain independent of the subclasses. This design facilitates future enhancements by introducing more subclasses under "*Student*" (e.g., distinct types like undergraduate or postgraduate) without requiring alterations to the higher-level code, specifically the "*User*" or "*Student*" base class. Throughout the project, we have consistently applied these principles to foster a modular and extensible architecture, promoting code flexibility and ease of maintenance.

# Assumptions

1. In cases where there are no available slots, the student will not have visibility into the camp's existence, making it unviewable and unavailable for inquiries.
2. When entering dates, any deviation from the specified format (yyyy-mm-dd) may result in the system assuming a default interpretation. For instance, inputting "23-11-23" will be interpreted as "2023-11-23."
3. To maintain consistency for date entries, users are advised to always input the date in the full format with leading zeros for single-digit days or months. For instance, it is recommended to input "05" instead of "5," as the latter, although accepted, might not be optimal for subsequent computations.
4. The system prompts users to change their default password only during the initial login. Subsequent logins will not trigger this prompt to prevent user irritation.
5. Staff members have the capability to only modify certain camp details—such as the registration closing date, camp name, slots exceeding the current number of enrolled attendees, location, and descriptions after students and camp committee members have initiated their participation. Editing other details at this stage is restricted due to potential implications on other camp restrictions.

# UML Class Diagram

A UML class diagram depicts a static, object-oriented system. It defines projects by classes, attributes, and functions; as such, it is a fundamental building block of any object-oriented solution. It shows the classes within a system and the operations of each one. In our object oriented program we have used 55 classes/interfaces, which serve a variety of purposes. These classes are somehow related to one another through various relationships such as: Dependency, Association, Composition, Aggregation, Generalisation and Realisation.

# Program Testing

|  |  |
| --- | --- |
| **(1) Login** | **(2) Camp creation** |
| 1. Invalid login credentials 2. Successful login, prompt to change password | 1. Staff creates camp 2. Edit camp details |
| A screen shot of a computer  Description automatically generated    A screenshot of a computer screen  Description automatically generated | A screenshot of a computer program  Description automatically generated  A screenshot of a computer program  Description automatically generated  A screenshot of a computer screen  Description automatically generated |
| **(3) Camp registration** | **(4) Enquiries and Suggestions** |
| 1. Student registers as Attendee 2. Another student registers as Camp Committee Member | 1. Student submits enquiry→Committee member replies 2. Committee member submits suggestion→Staff approves |
| A computer screen shot of a program  Description automatically generated    A screenshot of a computer program  Description automatically generated | A screenshot of a computer program  Description automatically generated  A screenshot of a computer program  Description automatically generated  A screenshot of a computer program  Description automatically generated  A screen shot of a computer  Description automatically generated |
| **(5) Report generation (Committee)** | **(6) Report Generation (Staff)** |
| 1. Student attendance for each camp 2. Students’ enquiry report | 1. Camp committee performance report |
| A screenshot of a computer  Description automatically generatedA screenshot of a computer  Description automatically generated | A screenshot of a computer  Description automatically generated |

# Reflection

While undertaking this project, our team faced multiple challenges and difficulties that we had to overcome. These hindrances came in different forms and forced us to do the relevant research in order to solve it, thus allowing us to learn new skills and knowledge in the process. One of the main challenges we faced was figuring out how to start on the project, and how to approach it, given the sheer size of it. Along with multiple roles, each with differing functions, there were also many classes to create, and therefore this caused some confusion on which one to begin with. To overcome this, our group decided to break it down into simpler and smaller parts and focus on creating an overarching structure for each part and outlining the class diagram for it. This allowed us to see how the individual parts functioned on their own, thus making it easier to link everything together to form the whole system. This also taught us how to approach such big projects in future, as breaking it down into smaller subproblems will make it clearer to understand the roles and objectives of different classes.

Another difficulty that we faced was debugging errors when they came up in compilation time / run time. Due to the multiple classes and extensive methods created, it was initially troublesome to figure out what was the root cause for the error, and thus solving it took a significant amount of time. To conquer this, our group decided to refine the way we implemented the classes, by splitting up certain bigger classes into further subclasses and allocating the methods into the newer and smaller subclasses respectively based on the more specific requirements of the classes. This ensured that the responsibilities of each class were kept to a minimum, thus allowing easy tracking of which class or method is the cause of the error, as based on the type and location of the error, we can narrow down the possible classes that might be responsible for it. Through this experience, we understood the importance of the Single Responsibility Principle (SRP), and how essential it is, especially for projects of a bigger scale. While working together as a group on a relatively big project such as this, we learnt how important it is to understand the different aspects and requirements of the project as a whole, even before we begin working on it. Our group decided to meet up, split up the different parts, delegate them accordingly depending on the manpower needed, and also set realistic deadlines to ensure progress. Through this we also learnt the importance of sticking to the deadlines set, as certain parts of the project might have to be completed first before the next part can be worked on.

Looking back, there are certain ways that our group could have further improved on our current project. For instance, our group realized that user friendliness is also quite a major part of how well a code runs, and thus we could have improved on the way the code looks during run time. Making the menu options and titles more visible and easier to read, for example, can help in ensuring a more pleasant overall user experience. Another feature that we could have possibly implemented is an undo feature, as there are possibilities of mis-inputs from users, which might require them to go through a lengthy process to rectify it. As such, an undo feature would save them a lot of time, and also make the experience better.