






**SC2002 OBJECT ORIENTED DESIGN & PROGRAMMING**  
**FINAL PROJECT REPORT (CAMP APP)**  
**AY23/24 Sem 1 | SCSD, Group 6**



**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 honoured 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
DEXTER NG JUN HENG (U2221630G)	SC2002	SCSD	 25/11/23
JESSICA DANIELLA GIRSANG (U221579J)	SC2002	SCSD	 25/11/23
KAPOOR ANANYA (U2222748K)	SC2002	SCSD	 25/11/23

MARCUS KOH ZHI YONG (U2222814E)	SC2002	SCSD	 25/11/23
RAPHAEL LIEW JIN CHENG (U2220628L)	SC2002	SCSD	 25/11/23

## 1. DESIGN CONSIDERATIONS

### 1.1 Approach taken

Our CAMs system serves as a comprehensive application tailored to implement NTU's very own Camp Management System. We approached system design by first identifying the entity classes that we believed were necessary. For instance, Student and Staff were necessary to store their related information and access permissions. CampController, CampRegistrator and ViewData are the main classes that handle the bulk of the functionality that Users, i.e. both Students and Staff, require.

We then used the design principles covered in Section 1.2, to guide our decisions on the remaining classes, interfaces, and references which we require.

### 1.2 DESIGN CONSIDERATIONS

#### 1.2.1 SOLID Design Principles

The following principles, collectively acronymized as **SOLID**, were referenced during our approach. They emphasise greatly on minimising dependencies and maximising cohesion within

classes, which are key goals to meet when it comes to design. This section briefly describes each of the principles with regards to our understanding. You can see them being used extensively in section 1.3, where possible.

### **Single Responsibility Principle (SRP)**

SRP states simply that each class should have only one purpose or responsibility that it does well. A class that has multiple, or tries to accomplish too much, violates this principle.

### **Open-Closed Principle (OCP)**

OCP states that any further modifications to already existing classes should be avoided, but classes could be instead extended, in the event that new functionality is added.

### **Liskov Substitution Principle (LSP)**

LSP, informally states that the subclass should have the same capabilities as the superclass, and the subclass should not throw exceptions where the superclass has not, such that, at any point in time, if object references to the superclass are replaced by that of the subclass, there are no differences as to the way the current existing system operates.

For example, in our code, we have the Page class which is extended by the subclasses, Login, Exit, SetPassword, StudentMenu, StaffMenu, MainMenu, and CampCommitteeMenu. These subclasses implement the same functionality as the startExecution() method in the superclass, meaning that at any point, we can replace a Page reference with a reference to any of its subclasses, and the code should still work.

### **Interface Segregation Principle (ISP)**

ISP states that large, unwieldy interfaces should be broken down into separate interfaces instead. Each interface should have one purpose or responsibility much like SRP for classes. For example,

### **Dependency Inversion Principle (DIP)**

DIP, informally states that there should be no direct associations between concrete classes. Concrete classes should instead interact through abstractions, i.e. interfaces or abstract classes.

## 1.2.2 Object-Oriented Programming Principles

### Encapsulation

Encapsulation is a fundamental concept that involves tying down a class' methods and attributes to only that class. In doing so, we hide these members from other classes in what is known as data hiding. This concept aligns with **SRP**, as each class encapsulates only a specific set of responsibilities, ensuring a clear and modular design.

### Polymorphism

Polymorphism allows object references to be referred to as a different type. This is achieved through method overloading and method overriding. Polymorphism supports **OCP** by allowing subclasses to modify or extend the methods they override, instead of a full replacement. It further plays a crucial role in achieving **LSP**, as it allows objects of derived classes to be used interchangeably with objects of their base classes.

## 1.3 Application of Principles

In this section, we explain how the above mentioned design and programming principles are applicable to our application.

CampController, which we intend to implement for staff to create, edit and delete camps, if left as it is, will be a violation of several principles:

- **SRP**, because the class has too many responsibilities.
- **OCP**. When we want to add new operations regarding camps, we will have to make several direct modifications to this class.
- **DIP**, Staff will have to directly use or depend on CampController.

Then, the resolution to these violations would be to partition this class into smaller classes, each one taking just one responsibility, i.e. one class focuses on creating camps, CreateCamp, another on deleting camps, DeleteCamp, and so on. These classes are performing some operation so they should extend CampController by overriding some method, or modifyCampInfo() in our case.

CampController therefore becomes an abstract class, extended by CreateCamp, EditCamp, DeleteCamp, and ToggleVisibility.

The next important instance to design has to do with the registration functionality for students. The logic for this class should be able to keep track and enable students to either register for a Camp or withdraw from one. Having one sole class to implement everything will again violate the same principles as mentioned above. We therefore applied a similar tactic, but used an interface instead, IEnrolls, in which the Enroll() method is implemented by the RegisterCamp, and WithdrawCamp classes. This usage of interfaces further enhances loose coupling between classes.

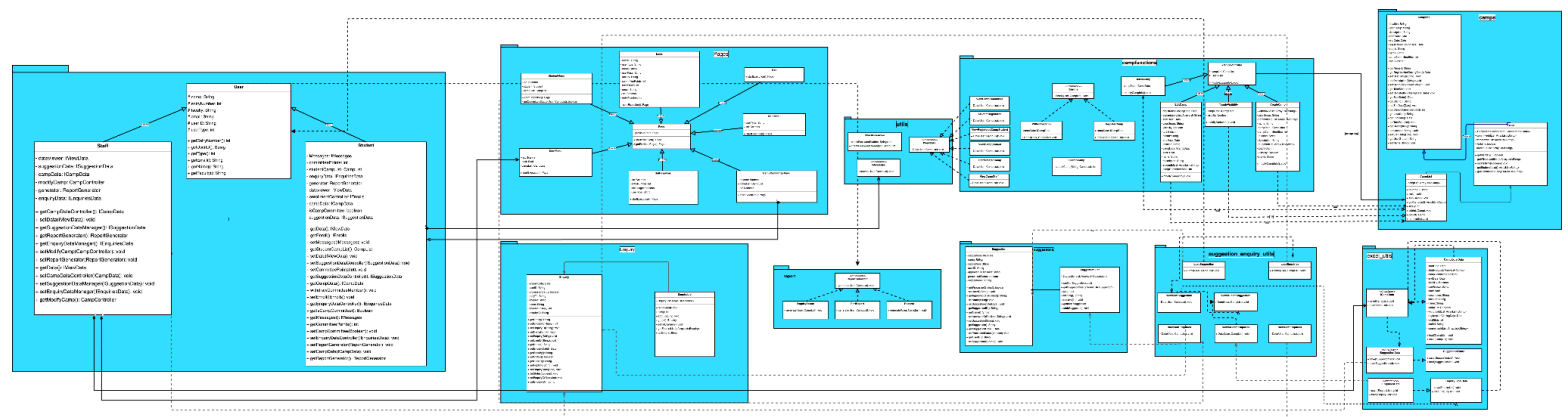
This is the general approach which we have applied extensively and used to justify the necessary creation of new classes separate from the main entity classes, in order to fulfil the design principles we've chosen. We also took additional care in ensuring that the **ISP** was not violated. In particular, when designing the interface for staff and students to be able to view the camps that they have created or registered for respectively, we realised that there were different variations of some view operation types, i.e. a student can only view the camps that are available to him/her, whereas a staff can view all camps, and the camps that he/she has created. We first considered adding all these view methods to the interface, but decided against it as it was a violation of **ISP**. We later discovered a workaround where we could simply abstract an arbitrary viewData() method from these different variations, and override it in separate classes. For instance, ViewCampCommittee, and StudentRegistered are classes that implement the IViewData interface, while overriding the Data() method.

#### **1.4 Assumptions made/other design choices**

- Instead of directly using ArrayLists, we made separate classes, like CampList, SuggestionList, and EnquiryList, which is essentially a recontextualisation of ArrayList to better communicate the object type to our other members within the team. In retrospect, this becomes redundant due to the impracticality when it comes to showing all the dependencies in the UML from the methods using the CampList class.

#### **1.5 Additional Features**

- ## 2. Detailed UML Class Diagram



Overall, our design allows for appropriate encapsulation for each object entity for the packages. While designing the UML, we also aimed to achieve loose coupling between classes by creating interfaces to reduce the relationship from association to dependency.

## Page 6 of 10

We came out with test cases based on the different menus (Main Menu, Student Menu, Staff Menu and Camp Committee Menu). From each menu, our test cases are from each option of the menu. For the complete and detailed test cases with images, refer to SC2002 TestCases.pdf attached together with this file. *"SCSD\_grp6\Report Submissions\SC2002 TestCases.pdf"*

## **4. Reflection**

Throughout this project, several challenges and learning experiences emerged, providing valuable insights into the complexities of designing and implementing an application. One common struggle was achieving concise Java documentation. Determining the appropriate level of detail proved to be a challenge, as providing too much could result in the document being difficult to read, whereas providing too little could lead to confusion among team members as to the functionality of a method or class. To overcome this, we embraced iterative feedback, regularly seeking input from team members to ensure that the documentation remained comprehensible yet comprehensive.

Figuring out the intricacies of the implementation of read and write from our application to Excel proved to be a massive pain point as well, especially in trying to debug this segment of our code that we were unfamiliar with. Even after searching extensively and looking at the provided sample code, it still was difficult in the areas of accessing, and formatting the file.

The struggle to maintain consistency between UML diagrams and the evolving codebase revealed the importance of UML design. This is an area that we believe we can further improve on, in retrospect. We should have finalised the design of the UML before working on the implementation, which would have reduced the need for major modifications and reworks to the UML in order for it to properly reflect the code.

This project illuminated how the SOLID design principles are not just theoretical constructs but practical guidelines that significantly influence the development process. Their impact was evident in the clarity of our documentation, the adaptability of our code, and the overall resilience of our software architecture. Moving forward, this experience reinforces the notion that adherence to SOLID principles is not just good practice, but a cornerstone for building robust, maintainable, and scalable software systems.

Another key lesson from this project is the crucial importance of two foundational software design principles: loose coupling and high cohesion. The synergy between loose coupling and high cohesion proved instrumental in creating a robust, maintainable, and adaptable system. This insight emphasises the ongoing relevance of these principles, underscoring their practical imperative in software design for scalability and resilience.

Looking ahead, there are several areas for improvement. Implementing more comprehensive testing procedures throughout the development lifecycle could catch potential issues earlier, reducing the time spent on debugging. This project highlighted the importance of not just coding proficiency but also a robust version control strategy to manage the evolution of the codebase effectively.

In summary, this project served as a profound technical odyssey, offering a deep dive into the meticulous nuances of Java documentation and the dynamic landscape of UML diagrams and the complex implementation, each phase providing a deepened understanding of technical intricacies.



## **APPENDIX A**

### **5.1 INDIVIDUAL CONTRIBUTION**

#### **MARCUS KOH ZHI YONG (U2222814E)**

Mainly responsible for creating the UML diagram with the creation of classes, packages, associations, contributed to the creation of the report. Vetted the source code, adjusted naming conventions, logic, and checked whether it accurately reflects the UML and the corresponding design principles.

#### **DEXTER NG JUN HENG (U2221630G)**

Mainly responsible for coming up with the source coding. For example, created CampExcelData class with methods like save() to write data to the excel and load() to read data from the excel. Additionally, used inheritance principle when creating the user class, student class and staff class. Furthermore, tried to implement the code with SOLID principles especially single responsibility principle such that a class only has a single responsibility. Contributed to the testing section of the report.

#### **JESSICA DANIELLA GIRSANG (U221579J)**

Mainly responsible for the generating report into .txt files source code. In the context of generating reports, implemented Single Responsibility Principle (SRP) specifically responsible for handling the creation and formatting of reports. This class does not have additional responsibilities, such as file handling or data retrieval since we created a separate class for that matter. Contributed to the testing of the project.

#### **KAPOOR ANANYA (U2222748K)**

Contributed in the writing of the source code. Helped with various packages such as the suggestions\_excel\_utils and pages package. Contributed to implementing camp functions and their utilities/functionalities. Implemented OOP concepts such as inheritance in these packages as well used interfaces to promote loose coupling. Ensured that there is high cohesion within classes. Contributed to the testing and overall editing of the report.

## **RAPHAEL LIEW JIN CHENG (U2220628L)**

Contributed to producing source code mainly for classes in camp, campfunctions, staff & student menu page. Assisted in integrating and vetting overall source code to ensure functionalities are properly implemented. Worked closely with Marcus to ensure consistency between UML class diagrams and our source code. Demonstrated various OOP concepts in our implementation such as polymorphism to override superclass methods like 'public abstract void modifyCampInfo()' in our 'campController' class with various methods like 'CreateCamp', 'DeleteCamp', etc, in its respective subclasses. This example also demonstrates the use of inheritance and abstraction and serves to reflect a wider implementation of these OOP concepts throughout our source code.

Refer to "*SCSD\_grp6\Report Submissions\WBS\_SCSD\_grp6.xlsx*" for the detailed work distribution.

## **APPENDIX B: Code access**

To view our code, access the src folder by following this path: "SCSD\_grp6\src". From there go to the main folder and run the MainApp.java. Default password is **password** for all users regardless if he/she is a student or staff. Password change will be stored at student\_list.xlsx and staff\_list.xlsx depending on user.

### **To view location of file**

To access the student list, follow this path: "SCSD\_grp6\src\excel\student\_list.xlsx"

To access the staff list, follow this path: "SCSD\_grp6\src\excel\staff\_list.xlsx".

To access the camp list, follow this path: "SCSD\_grp6\src\excel\camp\_list.xlsx",

To access the enquiry list, follow this path: "SCSD\_grp6\src\excel\enquiry\_list.xlsx"

To access the suggestion list, follow this path: "SCSD\_grp6\src\excel\suggestion\_list.xlsx"

To access the report generated, follow this path: "SCSD\_grp6\src\excel\reports"

### **Additional instruction:**

Please install Apache POI for excel library to work.

The download link: <https://poi.apache.org/download.html#POI-5.2.4>

