

**AY2023/24 SEMESTER 1: SC2002 Objected-Oriented Programming 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 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  (CE2002 or CZ2002) | Lab Group | Signature /Date |
| Nguyen Phuong Linh (U2220531J) | SC2002 | SCMA Team 1 | 24 November 2023 |
| Soo Qi Yang (U2221761L) | SC2002 | SCMA Team 1 | 24 November 2023 |
| Teo Kai Xuan (Zhang Kaixuan) (U2222468D) | SC2002 | SCMA Team 1 | 18 November 2023 |
| Tee Jeeng Yee (U2222052F) | SC2002 | SCMA Team 1 | 21 November 2023 |
| Merwyn Masagca  (U2222856A) | SC2002 | SCMA Team 1 | 24 November 2023 |

**1. Design Considerations**

**1.1 Approach**

Our primary observation from the assignment requirements is that there are several multiple-to-multiple associations to be handled. For example, a student can register for multiple camps, and a camp can be registered for by many students. This applies to enquiries and suggestion information as well, among other examples. Hence, in order to reduce the coupling between the aforementioned classes, we try to avoid object composition and unnecessarily storing objects within each other as an array. Instead, we seek to create a more structured, extensible and flexible system.

We implement “databases” to store each type of information. Namely: Users, Enquiries, Registration Information, Suggestions and Camps. These databases create manager classes that provide interfaces with specified contracts that can be seamlessly utilised by users or other databases. The use of databases minimises coupling between the User and Camp classes, and the provision of interfaces reduces coupling between the clients and the databases.

The high-level classes that provide the UI for the users will then utilise the lower-level interfaces to query and manipulate information as desired, in a manner allowed by the interface and manager classes.

This approach is also robust and extensible. If we seek to add a new form of interaction between User and Camp classes, beyond enquiries, suggestions and registration details, we will be able to extend the code base easily, by simply adding a new database with appropriate interfaces for that new interaction class, with minimal impact to existing code.

**1.2 Assumptions**

Assumptions made when designing the system:

1. All users can use filters to view the camp list or to generate reports (date, location, presence of a particular student as camp attendee/committee etc.).
2. The default filter for a user is none.
3. All users can use sorters to sort the camp list or to generate reports (sort by camp details like date, location, etc.)
4. The default sorting order of viewing a camp list or generating a report is by alphabetical order of the camp name.
5. Registration of camp and camp committee is automatic as long as there is a vacancy.
6. The number of camp committee slots is counted in the total slots.
7. A camp’s name is unique. No two camps are allowed to have the same name.
8. A suggestion is automatically rejected if it is not approved 5 days after it is created.
9. Students are able to see details of a camp when registering so they can make a better decision on whether they want to participate or not. Examples of important information in the camp details include dates, location, and camp descriptions.
10. A student can register for multiple camps, but must do it one camp at a time.
11. A camp cannot be deleted or have its visibility changed once a student has registered.
12. A camp cannot have its faculty changed after creation.

**1.3 Design Principles**

**1.3.1 Design Principles**

Most of the design principles are implemented in the Camp and Registration databases for demonstration purposes, and the principles would ideally be extended and implemented in our other databases and classes in a real-world application. At the present moment, in the interest of time, the other databases simply prioritise core functionalities over implementing such detailed design principles.

**Single Responsibility Principle**

Each class should be in charge of a single responsibility, which will reduce the number of reasons that its code would need to be changed whenever a change in functionality or implementation is desired in the system. In our system, we split the responsibility of the classes into a storage class and multiple manager classes. The storage class is only responsible for storing the information, and each manager class is only responsible for interacting with the storage class in one specific manner.

For example, within the Camp package, the CampDataBase class stores an array of camps, and is not responsible for much else. It provides manager classes, such as StaffViewAllCamps, which is responsible for printing all the staff’s camps details to a console, and StaffPerformanceReportGenerator, which is responsible for generating a report of the camp committee member’s in the staff camps to a text file. Because these manager classes are only responsible for a single task, there is only one reason that they might each need to be changed.

For instance, we would only need to change StaffPerformanceReportGenerator when we wish to change the way a performance report is generated. This would not affect StaffViewAllCamps, since StaffViewAllCamps is only responsible for printing the staff’s camps details to the console, and the performance report is not under its responsibility.

**Open-Closed Principle**

The Open-Closed Principle states that a module should be open for extension but closed for modification. We want to be able to extend our modules and functionality without making changes to existing code.

We apply this concept to various interfaces within our system. For example, in the camp package, the manager classes StaffViewAllCamps and StudentViewAllCamps have different functionalities, as students can only view camps that are visible and within their faculty, while staff can view all the camps in the database. Hence we declare an interface IViewAllCamps that provides an abstract viewAllCamps method to the client, and close this interface from modification. StaffViewAllCamps and StudentViewAllCamps then extend different functionalities of viewAllCamps from this interface. The client can then call and obtain the desired functionality from the IViewAllCamps interface based on whatever behaviour is appropriate.

If we desire to add a different functionality or way of viewing all camps, the interface is open for extension. Hence we can simply add a new class implementing the interface which overrides the abstract viewAllCamps method with a different functionality, without affecting the existing classes, or the closed interface. It also would not affect the client class, which simply relies on the closed IViewAllCamps interface.

Another noteworthy application of this principle is within the registration package, where an interface ICheckRegistration that is responsible for checking if a student is registered in the registration database, is closed for modification, but open to extension. Various manager classes extend from it to provide different functionalities for the client, namely CommitteeRegistrationChecker and AttendeeRegistrationChecker to check if the student is registered as a camp committee member, or to check if the student is registered as a camp attendee.

**Liskov Substitution Principle**

Liskov Substitution Principle states that an object of a superclass should be replaceable for an object of its subclass without causing the program to work in an unexpected manner. This principle can be extended to interfaces as well, with the idea that any class implementing an interface can be substituted for any other class implementing the same interface, without causing the code to run unexpectedly.

In the camp package, the manager classes SortCampByCampName, SortCampByStartDate, amongst many others, implement the interface ISortCamps, which has the abstract method sortCamps(). The contract of this interface method is to take in no parameters, and to sort the camps in the Camp DataBase by some specified order.

For all these manager classes that implement ISortCamps, they do not request for any new input parameters or additional error checking, and the expected results would all be a sorted Camp Database. Hence, since the pre-conditions of these manager classes are not stronger than one another (manager classes expect no more), and their post-conditions do not give less than expected (manager classes provide no less), they can be substitutable for each other, as they do not violate the contract, and would not cause the code to work in unexpected ways.

**Interface Segregation Principle**

Interface Segregation Principle states that many client-specific interfaces are better than one general-purpose interface, and the classes should not depend on interfaces that they do not use. If we created a FAT interface that contains all the methods available for each database, there would be two main issues: the client would have access to every method available in the database, and every manager class that implements that FAT interface would have to provide a blank implementation for the methods it does not intend to provide functionality for.

Hence, instead, we provide multiple interfaces for each database, each containing a specific method that a client might want to use. In this manner, the client can selectively utilise interfaces that are only relevant to them. Manager classes will also only need to implement the interfaces that contain methods that are specifically relevant to them.

For example, the camp package provides multiple interfaces, such as IDeleteCamp and ISortCamps. In the case of these 2 interfaces, a Student client will only be able to utilise ISortCamp interface, while a Staff would be able to use both IDeleteCamp and ISortCamps. This implementation of multiple interfaces is flexible and secure as it allows us to only give each client the interfaces that they require, without exposing critical interfaces, such as IDeleteCamp, to less privileged users.

**Dependency Injection Principle**

According to the lecture, classes should depend on abstraction but not on concretion. Interfaces have the lowest chance to be modified, unless the signature of its abstract member function needs to change. This implies that clients should depend on interfaces instead of concrete classes, as this reduces the likelihood of modifications needing to be made to the client, since there is a lower chance that the interface which it depends on will experience any changes.

In our code, we strongly demonstrate the Dependency Injection Principle in the Camp and Registration Databases. Every functionality that these databases provide are implemented into an interface by a manager class. Hence, when interacting with these 2 databases, any client that seeks to interact with the manager classes will always do so via the interfaces.

For example, when the student class only depends on interfaces when interacting with the Registration Database, through interfaces such as IRegisterCamp and IDeregisterCamp. It also only depends on interfaces, and never concrete classes, when interacting with the Camp Database, through interfaces such as IViewAllCamps and ISortCamps.

The same can be said for the Staff class, or any client that interacts with the Camp and Registration database. The client is always dependent on the interface provided by the database, and never reliant on a concrete class, hence reducing coupling between the clients and the databases.

**1.3.2 Object-Oriented Model Principles**

**Abstraction**

We abstract the various objects in the system based on their defining characteristics and behaviours. For instance, some defining variables of a student include their faculty and name. The characteristic behaviours for a Student object in this CAMs would be the ability to register and deregister for a camp, or to view all the camps that are available to him.

**Encapsulation**

We ensure appropriate data protection of the classes in our system, by setting all their instance variables to private, unless a specific need dictates otherwise. Data would then only be accessed and changed through their respective getter and setter functions, which enforces security and allows us to maintain a level of control over how these data can be accessed or altered.

**Polymorphism**

We implement polymorphism in the camp package’s IViewAllCamps interface. Both the StaffViewAllCamps manager class, and the StudentViewAllCamps manager class derive from the IViewAllCamps interface. When the User object uses the IViewAllCamps interface, it does not need to know the identity of the manager class that is extending it. The IViewAllCamps interface will simply invoke the appropriate function call based on the manager class that the User has.

# 

**4. Reflection**

**4.1 Difficulties Encountered**

Initially, the project seemed daunting at first as we were uncertain about where to start, as we had never come across a project of this scale before. It was difficult to come up with a complete, well-designed UML diagram before starting on the code, as we were unsure about the practicality of the implementation that we were conceptualising. We then decided to take on a more general approach to the UML class diagram, so as to visualise the required classes and to allocate classes for each member to work on. This helped us to split the workload into more manageable portions, and kick-started our progress.

It was also difficult to integrate our parts as a whole due to slight variations in ideas, but we managed to fix all the pieces of the puzzle together by working closely with one another and communicating our ideas clearly.

**4.2 Knowledge Learnt From This Course**

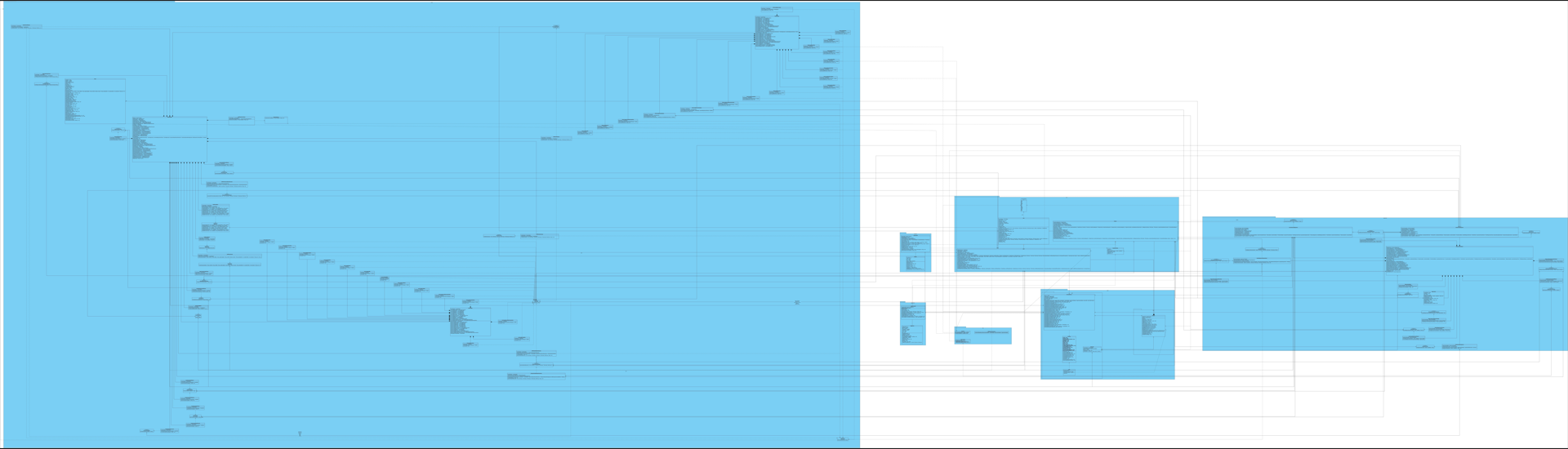
Through this project, we have learnt how to collaborate and work as a team on a relatively complicated project. We used version control such as Git and used Github in order to store our codes, and so that we would be able to make changes to and work together on the same codes. Additionally, this project allowed us to be more familiar with Java as an object-oriented programming language, as well as introduced us to the different features Java offers, like Javadoc for documentation purposes. Finally, we were able to apply what was taught in the lectures like SOLID design principles and the four main object-oriented concepts and incorporate these considerations into both our design and UML class diagram, in order to achieve loose coupling between different classes, and high cohesion within the same class.

**4.3 Suggestions for Further Improvement**

**Camp feedback and evaluation**

We propose the addition of a feature that will enable staff to collect feedback and data about the camp from camp participants and camp committee members after the conclusion of the camp. This would allow staff to use the feedback provided to continuously improve the organisation, planning and execution of future camps and provide a better experience for everyone involved.

**5. UML Diagram**

****

The .vpp and .jpg file for our implementation are attached along with this report.

**5.1 Explanation**

The related classes are grouped and packaged together. Users are in a package, and miscellaneous classes that provide utilities like parsing date formats are in another. The Camp and Registration packages have a database object, and multiple manager classes that provide interfaces to encapsulate the database object from classes belonging to other packages.

**6. Essential Test Cases**

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Explanation** | **Result** |
| Security feature: Prompt user to change password when logging in for the first time | The default password is overridden. The new password is saved and written back to the database file upon exit |  |
| Staff creates a camp with camp’s end date set to be earlier than camp’s start date | **Error Checking:**  Displays error message and asks for staff’s input again. When all details are correctly entered, new camp is created |  |
| Staff views all camps | 1. Input ‘1’ to view camps 2. Input ‘a’ to view all camps OR ‘b’ to view staff’s own camps |  |
| Staff approves suggestions by camp committee | 1. Prints the list of suggestions 2. Input the suggestion number to approve 3. Prints success message and committee member gains a point |  |
| Staff generates student report | 1. Input the name of output file (‘student’) 2. student.txt report is generated, which is unique to each of the staff 3. Prints success message |  |
| Staff generates performance report | 1. Input the name of output file (‘performance’) 2. performance.txt report is generated, which is unique to each of the staff 3. Prints success message |  |
| Staff generates enquiries report | 1. Input the name of output file (‘enquiries’) 2. enquiries.txt report is generated, which is unique to each of the staff 3. Prints success message |  |
| Student views registered camps | 1. Input ‘5’ to view registered camps 2. A list of registered camps is printed, including the role of the student in the camp (In this example, Camp Attendee) |  |
| Student registers for a camp after registration closing date | **Error Checking:**  Registration closing date (10/10/2023) has passed, unable to register for camp, prints error message |  |
| Non-committee student attempts to access Committee Commands | **Error Checking:**  Checks that student is not a committee member, and denies access. Error message printed and program returns to main menu |  |
| Student registers for a camp as camp committee | 1. Input ‘3’ to register as camp committee 2. From the list of camps visible to the student, input the camp name (‘explore’) to register |  |
| Student sends enquiry | 1. Input camp name (‘explore’) 2. Input the enquiry |  |
| Student views own enquiry sent and the corresponding reply | 1. Input ‘7’ to view own enquiries submitted by the student 2. The list of enquiries submitted and the corresponding replies (if any) are printed |  |
| Camp committee replies to enquiry | 1. In Committee Commands, input ‘6’ to reply to enquiries 2. The list of enquiries submitted is displayed 3. Input the corresponding enquiry number to reply to the enquiry 4. Enter the reply and gain a point |  |

