SC2002 Object Oriented Design & Programming

SCSC Group 3

CAMs Assignment Report

1.Design Considerations and Design Approach Taken

We approached the design of this project application by employing the use of “Controller” classes to segregate the method definitions from the entity classes, which contain the data information attributes. The entity classes depend (use) on the “Controller” classes to manipulate the data. We also used Interface classes whose responsibility is to operate the terminal output to display the application output to the user. This way, we are able to deploy a pseudo Model-View-ViewModel (MVVM) Architecture where instead of a GUI, the program output is displayed in the terminal interface. We chose to use csv files to store and extract the data to and from our application. Our program reads the data from the csv files to create the necessary objects needed for the program and writes and creates csv files to store user instructed data. We also attempted to implement full error checking and to adopt a user-friendly approach, by having a simple and easy-to-read user interface flow and prompting for correct user input whenever erroneous input is received.

1. Single Responsibility Principle (SRP)

In the design of our application, we ensured that every class has a sole responsibility. We achieved this by ensuring all our methods are first implemented in “Controller” classes, each controller class is responsible for implementing methods related to its role. For example, the “StaffCampOperationController” only implements methods relating to a staff’s operations on a camp, this is illustrated in the figure below.

Figure 1.1 Demonstration of using “Controller” classes

A diagram of a computer flowchart

Description automatically generated

This allows us to have high cohesion and loose coupling. High cohesion is realised in our design where each “Controller” class does one thing (and does one thing well). Loose coupling is realised in our design where there is low dependency between classes by using dependency relationships between entity class and their respective “Controller” classes for the methods.

Maintainability is observed when changing one implementation of a method in a “Controller” class. The change will not affect other “Controller” classes and will not affect how the corresponding classes use the methods. For example, a change in the implementation of “edit” in “StaffCampOperationController” will not change how “Staff” calls the method. If there are changes to how any method is implemented, “Staff” class is not affected, and the controller classes implementation will be modified instead.

This way, there is also better encapsulation in our design where “Controller” classes are responsible for their own validation and persistence. Our entity classes will create the “Controller” object temporarily and use it to access necessary methods. The entity classes do not need to know the inner workings of the methods, and only need to temporarily create the “Controller” object to access the relevant methods.

There is also better reusability and extensibility in our design where if a new class needs to be added and has same or more privileges as existing classes (Eg. “Staff”), the new class can just use the respective existing “Controller” classes that contain the relevant methods. Furthermore, having “Controller” classes allows the flexibility for any entity class to only use the methods that they need from the “Controller”.

1. Open-Closed Principle (OCP)

Open-close principle is applied in this case where (idea of extending but do not change). For example, “Staff” and “Student” inherit from “User”, so that if any new special type of “User” is needed, it can extend from the “User” class while “User” is closed for modification. Adding a new special type of “User” does not affect the existing “Staff” and “Student” classes and we can easily apply inheritance by extending the “User” class. This improves reusability of existing classes, extensibility of the superclasses and maintainability.

1. Liskov Substitution Principle (LSP)

All our subclasses can do what their superclass does, because there is no method overriding, all the methods are inherited and remain with the same functionality. “Staff” and “Student” classes have all the functions that “User” class has.

1. Interface Segregation Principle (ISP)

To avoid creating “FAT” interfaces, we separated our interfaces into its most general forms. For more specialised interfaces, we will extend it from the general interfaces. The “Controller” classes will only implement interfaces containing only methods that they will use and with their own implementation as seen in Figure 1.2.

Figure 1.2 “Controller” implementing interfaces that they need

A diagram of a computer program

Description automatically generated

Both “CommitteeSuggestionController” and GeneralEnquiryController implements all 4 interfaces while “CommitteeCampOperationController” only implements the “IView” interface as the “CommitteeCampOperationController” class does not require the methods of the other interfaces.

This reflects on one of the cases where we employed the OO concept of polymorphism. In this scenario, the CommitteeCampOperationController, CommitteeSuggestionController and GeneralEnquiryController have different implementations of the IView interface view method.

This approach also makes use of Java’s feature of multiple interfaces and applies abstraction by creating an Interface that declares the common method among all the “Controller” classes.

By applying ISP, “Controller” classes that do not use the method will not be forced to implement it. Whenever there is a need to add new methods into the application, we will place it in the respective interface. Furthermore, the “Controller” classes can have their own implementation of the same method declared in the interface.

1. Dependency Injection Principle (DIP)

We apply DIP by depending on interfaces more than concrete classes, therefore our “Controller” classes are all ‘splitted’ based on interfaces. This is because interfaces are less likely to change, and they also give the flexibility for each controller to implement the same method with their own functionality. As much as possible, we do not have dependencies between concrete classes unless necessary. “Controller” classes are prevented from depending on modules that often change and promote loose coupling. We also exploit the ‘uses’ relationship as much as possible as it depicts a temporary relationship and loose bonding.

For example, both “ManagerEnquiryController” and “StaffCampOperationController” depend on the interface “IView”. When requirements for “ManagerEnquiryController” change, only that controller is modified and “StaffCampOperationController” is not affected. This would not be possible if both classes depended on a general “StaffController” concrete class.

Interfaces ‘shield’ changes made in each “Controller” method from each other. Interfaces also allow the flexibility to plug in a new “Controller”, say an externalOrganiserController.

1. Assumptions

The following are the additional assumptions made while building our UML:

* Students can only submit one enquiry and one suggestion per camp, they can only edit their existing enquiry or suggestions on the condition where it is not replied or approved.
* Staff will not create a Camp with an existing CampID.
* Once a staff approves a suggestion, he/she cannot unapproved it.
* Staff can only edit all camp details (less campID) only if there no registered participants (empty camp).
* Each Camp can only be open to either one faculty or the entire NTU (all faculty)

2. Detailed UML Class Diagram

We start our design in interfaces, interfaces will be a group of methods needed at a high level by the application. The “Controller” classes will implement the methods of the interfaces. “Staff”, “Student”, “CampAttendee” and “CampCommittee” will use their corresponding “Controller” classes to access the specific methods they need. Our UML class diagram is illustrated in figure 1.1, a more viewable UML Class Diagram is available in PDF. (UML Diagram.pdf).

Figure 2.1 Detailed UML Class Diagram

A diagram of a computer flowchart

Description automatically generated

The main entity classes are “User”, “Staff”, “Student”, “CampAttendee”, “CampCommittee”. Each entity class depends (use-a) on their respective “Controller” classes for any operation. Each “Controller” class will implement interfaces with the required declared methods. “Camp”, “CampList”, “CampEnquiry” and “CampSuggestion” are classes that contain the necessary information. This approach to our design improves cohesion and reduces coupling.

3. Testing

Some of the test cases are shown below:

1. Committee members gain one point whenever
2. he/she submits a suggestion
3. he/she replies an enquiry
4. submitted suggestion is approved by a staff

**A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**

The increment in points are reflected in the top half of the images, the bottom half shows the inputs.

1. Staff cannot delete a Camp
2. without creating a camp or

A screenshot of a computer program

Description automatically generated

1. if camp is filled with participants

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated

1. Camp registration deadline must be before camp start date

A screenshot of a computer program

Description automatically generated

1. Staff can view all camps while student can only view camps visible to him/her based on camp and visibility.

A screen shot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screen shot of a computer

Description automatically generated

First image shows the “*camp\_list.csv*” file which shows that faculty camps are not open to NTU. UOC and Hall orientation camps are open to NTU, but Hall orientation camp is not visible. Second image shows a Staff’s Interface, third image shows an SCSE camp on the Student’s page.

1. Camp Committee members can only edit and delete details of suggestions that are not yet approved.

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

First image on the left shows that the camp committee member can create a new suggestion “*Suggestion 1*” and then edit it to “*Suggestion 1 Edited*” before it is approved. However, after it is being approved by a staff member as shown in the middle image, the committee member cannot edit/delete their previous suggestion, shown in the third image on the right.

4. Reflection

1. Implementation of Design Principles

Realising design principles into our UML took a huge amount of time. It required many iterations of verifying correct implementation of design principles while applying OO concepts. We had to ensure we did not violate any principles in the structure of the UML.

1. Knowledge learnt from this course

Appreciation of OO concepts together with SOLID design principles. By designing using SOLID, whenever changes needed to be made in our project, we could easily do so with minimal impact. Implementation of a relatively large project in a short amount of time led us to learning how to divide and conquer a task efficiently. We also had to self-learn how to use libraries and in-built functions such as “LocalDate” and CSV read/write methods.

1. Area of improvements for the course

Provided project specifications and descriptions are vague and unclear. This led to confusion and many reiterations to adjust our project to newly clarified requirements. This added to our stress load as we only completed the content required for the project in Week11 (Design Principles), which was one of the core knowledge areas needed for the project. Suggested sample test cases are not aligned to the features required of the application. Even the report requirements are misaligned. An example screenshot is attached in Figure 4.1. The figure on the left does not mention the “Testing” section to be included.

Figure 4.1 Misalignment of instructions

A close-up of a document

Description automatically generatedA document with text and blue text

Description automatically generated with medium confidence