A picture containing indoor, computer, computer, table

Description automatically generated

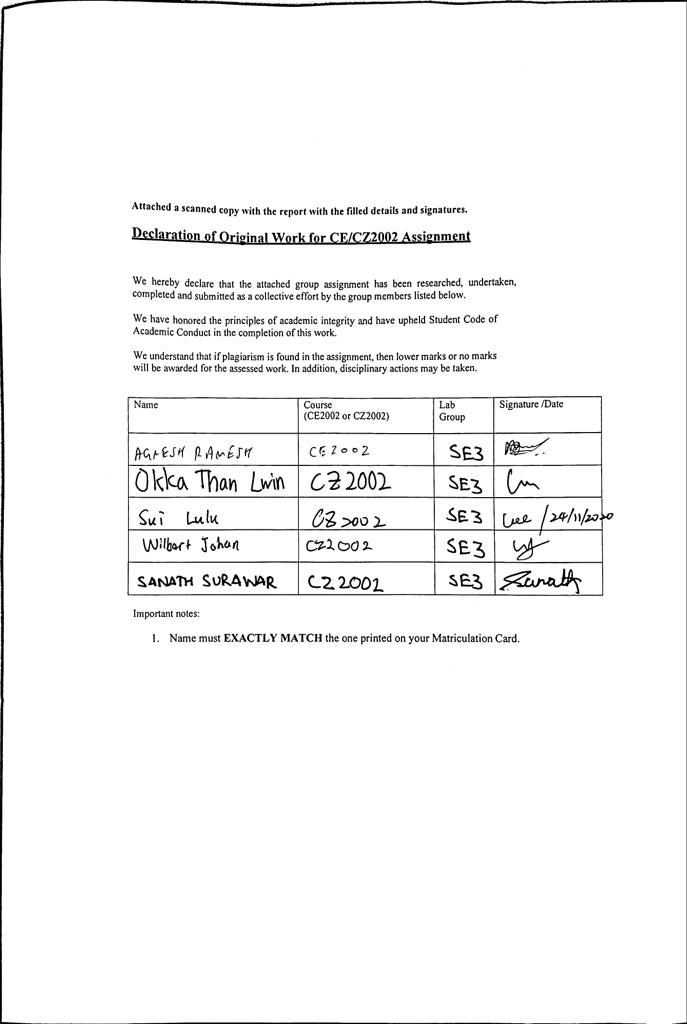
**TEAM MEMBERS**

|  |  |
| --- | --- |
| **Name** | **Matriculation Number** |
| **Surawar Sanath Sachin** | **U1922592K** |
| **Agnesh** | **U1921141K** |
| **Sui Lulu** | **U1921987A** |
| **Okka Than Lwin** | **U1920751J** |
| **Wilbert** | **U1922167K** |

**SE3 GROUP 7**

**CZ2002 Final Report**

**Nanyang Technological University**

**DECLARATION OF ORIGINAL WORK FORM**

# Table of Contents

[Table of Content](#_bookmark0) 3

1. [Introduction](#_bookmark1) 4
2. [Design Considerations](#_bookmark2) 4
   1. [Approach Taken](#_bookmark3) 4
   2. [Design Principles Used](#_bookmark4) 4
      1. [Single Responsibility Principle (SRP)](#_bookmark5) 4
      2. [Open-Closed Principle (OCP)](#_bookmark6) 5
      3. [Liskov Substitution Principle (LSP)](#_bookmark7) 5
      4. [Interface Segregation Principle (ISP)](#_bookmark8) 5
      5. [Dependency Injection Principle (DIP)](#_bookmark9) 5
   3. [Object-Oriented Concepts (Explanation of UML Diagram) 6](#_bookmark10)
      1. [Abstraction](#_bookmark12) 6
      2. [Encapsulation/ Information Hiding](#_bookmark14) 6
      3. [Inheritance](#_bookmark15) 6
      4. [Polymorphism](#_bookmark16) 6
      5. Composition 7
      6. Interface Realization 7
   4. [Data Structure](#_bookmark17) 7
3. [Assumption](#_bookmark18) 7
4. [UML Diagram](#_bookmark19) 8
5. [UML Sequence Diagram](#_bookmark20) 9
6. [Test Cases 1](#_bookmark21)0

**INTRODUCTION**

My STudent Automated Registration System (MySTARS) is a console-based application designed and developed for both staff and students to manage registration of courses. This application encompasses the key features such as creation of new courses, registration of courses and addition of student records.

This report covers the object-oriented programming (OOP) concepts and key design considerations used to implement the application. The design will also be represented in a UML Class Diagram and UML Sequence Diagram for one of the features, showing the relationships and interactions between the objects. Moreover, several test cases have been included to ensure that the requirements of the application that had been set initially, are met.

**DESIGN CONSIDERATIONS**

1. **Method of Approach**

* This project was a comprehensive application of OO concepts, both in terms of ensuring proper design and efficiency of code.
* The architectural style that we have adopted is thus the Object-Oriented Architecture, which is a newer form of the call-and-return architecture

1. **Design considerations and principles**

In our project design, SOLID design principles are extensively applied to make sure the project is easy to maintain and modify the codes with minimal cost by minimizing the impact of changes. The SOLID design principles are namely Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Injection Principle. Detailed explanation and implementation examples will be elaborated one by one.

***b. 1 Single Responsibility Principle***

* This principle means that there should never be more than ONE reason for a class to change, which means high cohesion. To achieve this objective, we try to make all our classes to bear only one responsibility.
* For example, the Menu class only print the menu of the actions that our users can take and then pass on the input of the user to the respective controller classes or the application classes. Since the Menu class has no access to the actual functionality to the respective classes, but merely uses them, any modification done on other classes or functions will not affect the Menu class and vice versa.
* As such, the ripple effect of changes on the codes will be extensively reduced and the minimal modification efforts will be required in the entire STAR system.

***b.2 Open/Closed Principle***

* This principle means that a module should be open for extension but closed for modification. To achieve this, we make sure subclasses can only extend the functionalities of the superclass but cannot modify the source codes in our project.
* For example, the Student class and Admin class are both extended from the User superclass. However, the User class is closed for the two subclasses to modify it, though the two classes make modification. For instance, the Student class has the extended functions such as setStudent\_id, which is not present in the User class.
* Following this principle in our project allows us great flexibility in the future when additional needs arise. For example, if the university needs a class of Deans where this group of staff can access all information on the Admin class, the User class can be easily extended again for a new subclass without great influence on many other classes.

***b.3 Liskov Substitution Principle***

* This principle means that subtypes must be substitutable for their base types. In terms of class, the derived class is substitutable for its base class if its pre-conditions are no stronger than the base class method, and its post-conditions are no weaker than the base class method.
* In our project, this principle is best illustrated by the method of dropCourse in the fileController. This dropCourse is also used in the studentController and the adminController with no greater expectation and providence than what is in the fileController.

***b.4 Interface Segregation Principle***

* This principle means that many specific interfaces are better than one general purpose interface so that classes should not depend on interfaces that they do not use.
* To avoid those fat interfaces, we use the MenuUI interface to display menu but the studentMenuUI and the adminMenuUI to display the specific menu for each group. This segregation makes sure fat interfaces are avoided.

***b.5 Dependency Injection Principle***

* This principle has a two-part meaning. Firstly, a high-level module should not depend upon low level modules, but both should depend upon abstractions. Secondly, abstractions should not depend upon details, but details should depend upon abstractions. In our project, many of our classes depend on abstraction, the abstract interface of Serializable.
* This dependence is beneficial for us to store necessary information in the binaryio file easily. For example, the User class is the higher-level module, and the Student class is the lower-level module. Both classes depend on the abstraction of Serializable. Thus, using abstraction of Serializable allows us to change behaviors and future code evolutions easily.

1. **Use of OO concept**

***c.1 Abstraction***

* Abstraction is constantly applied in our project. Besides the User superclass and its Student and Admin subclasses discussed in the Open/Closed Principle, another good illustration is the menuUI and its implementation of studentMenuUI and adminMenuUI.
* Using such interface implementation, we can apply the method displayMenu() more easily in the implementation class using object reference.

***c.2 Encapsulation***

* Encapsulation or information hiding is essential in our project as the STAR system needs to deal with many private and confidential information such as emails and passwords. This information should be encrypted and not accessible to other classes.
* For example, in the notificationController class, the sender\_email and sender\_pw attributes are built to be private so that such private information should be encapsulated within the class. Thus, critical information will not be accessible to classes such as fileController and data safety is granted.

***c.3 Inheritance***

* Inheritance is applied in our project to enhance code reuse. For example, the Student and Admin classes inherit from the User class and use the methods available inside the User class like setUsername() and setPassword() so that repetition is minimized.
* This method overriding make sure that no new methods must be created every time a subclass is created.

***c.4 Polymorphism***

* Polymorphism allows us to perform a single action in different ways. This is a useful concept in our STAR project as there are some occasions where the same methods are called to perform the tasks slightly different in some classes.
* For example, the function below shows that the menuUI object reference is referred to different types as studentMenuUI and adminMenuUI. Such polymorphism allows you to define one interface of menuUI and have multiple implementations.

**if(acc.equals("student")) {**

**menuUI studentmenu=new studentMenuUI();**

**studentmenu.displayMenu(username);**

**}**

**else if(acc.equals("admin")) {**

**menuUI adminmenu=new adminMenuUI();**

**adminmenu.displayMenu(username); }**

***c.5 Composition***

* Composition is best illustrated in our Course-Index-Schedule association. This is because an index will not exist without a course. Similarly, a schedule will not exist without an index.
* This ‘whole-part’ relationship allows us to reuse code by modeling the “has-a” association between these objects. For example, getWaitlist() method is reused in both Course and Index.

***c.6 Generalization and Interface Realization***

* The Student class and the Admin class are two specific classifiers of their generalization, the User class. Similarly, the studentMenuUI and the adminMenuUI are the realization of the menuUI interface.
* Such usage of different relationship between the classes and interfaces allows a clearer understanding of the entire system and easy reuse of codes.

1. **Data structure**

* To read objects or write objects to a file, we use object serialization provided by Java. The data about each user is then stored and updated in a file called binaryio. Such binary file IO makes sure our data are stored and accessed as a sequence of bytes. Binary files are chosen because they are more efficient and the speed of access of data is faster.
* Additionally, as the data is stored using numeric formats, these files will also take up less memory spaces. This advantage is crucial in our project as the STAR system is needed to every member in a university and the huge amount of data resulted will require such efficient files.

1. **Assumptions & Limitations**

In our project, some of the assumptions and limitations are listed as follows.

* Multi-users concurrent login is not considered.
* Pre-requisite conditions when registering courses are not omitted.
* The major and school of each student is not accounted for. Therefore, all students can take all possible courses available on STAR.
* There are only two level of authorities in our STAR system, namely Admin and Student. There is discrepancy from the real life where more levels of authority are needed to handle the STAR properly.

[**UML CLASS DIAGRAM**](https://drive.google.com/file/d/1oWqhB-W-xSoHcEtwequ-F27Kc_mkqlty/view?usp=sharing)

A picture containing diagram

Description automatically generated

**Diagram, schematic

Description automatically generated**[**UML SEQUENCE DIAGRAM**](https://drive.google.com/file/d/1qLlOyLTtWiKg1sR_Sd0tGObaeYPgziih/view?usp=sharing)

**TEST CASES**

**Student Login**

|  |  |  |
| --- | --- | --- |
|  | Test Case | Expected Outcome |
| a | Login before allowed period (dates) |  |
| b | Login after allowed period (dates) |  |
| c | Wrong password |  |

**Add a student**

|  |  |  |
| --- | --- | --- |
|  | Test Case | Expected Outcome |
| a | Add a new student | Text  Description automatically generated |
| b | Add an existing student | Text  Description automatically generated |
| c | Invalid data entries |  |

**Add a course**

|  |  |
| --- | --- |
| Test Case | Expected Outcome |
| Add an existing course |  |
| Add a new course  (with combination of (ii) from above) |  |
| Invalid data entries |  |

**Register Student for a course**

|  |  |  |
| --- | --- | --- |
|  | Test Case | Expected Outcome |
| a | Add a student to a course index with available vacancies. |  |
| b | Add a student to a course index with 0 vacancies in Tut / Lab. |  |
| c | Register the same course again |  |
|  | Invalid data entries  (e.g. wrong student ID / course code, etc.) |  |

**Check available slot in a class(vacancy in a class)**

|  |  |  |
| --- | --- | --- |
|  | Test Case | Expected Outcome |
| a | Check for vacancy in course index |  |
| b | Invalid data entries  (e.g. course code, class code etc.) |  |

**Day/Time clash with another course**

|  |  |  |
| --- | --- | --- |
|  | Test Case | Expected Outcome |
| a | Add a student to a course index with available vacancies. |  |

**Waitlist Notification**

|  |  |
| --- | --- |
| Test Case | Expected Outcome |
| Add studentA to a course index with 0 vacancies |  |
| Drop studentB from the same course index |  |
| Display studentA timetable |  |

**Print student list by index number, course**

|  |  |  |
| --- | --- | --- |
|  | Test Case | Expected Outcome |
| a | Print list by   1. Course 2. index | (i)    (ii) |
| b | Invalid data entries  (e.g. course code, index code etc.) |  |

**Additional Provisions**

|  |  |  |
| --- | --- | --- |
|  | Test Case | Expected Outcome |
| a | Swap indexes with another student | Text  Description automatically generated |
| b | Change registered email | Text  Description automatically generated |
| c | Change password for student | Text  Description automatically generated |

**Link to Demonstration Video**: <https://youtu.be/97qAMecTb4k>