PROJECT REPORT

CZ2002 OBJECT ORIENTED DESIGN AND PROGRAMMING

PREPARED BY:

SS5, GROUP 1

|  |  |  |
| --- | --- | --- |
| TAN PEI LUN | U2022841A | |
| AHMAD ALEENA | |  | |
| ANANDARAJAN SINDINI |  | |
| LIM JIEXIAN |  | |
| TAY LU KANG |  | |

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

# Table of Contents

[Table of Contents 3](#_bookmark0)

1. [Introduction 4](#_bookmark1)
2. [Design Considerations 4](#_bookmark2)
   1. [Approach Taken 4](#_bookmark3)
   2. [Design Principles Used 4](#_bookmark4)
      1. [Single Responsibility Principle (SRP) 4](#_bookmark5)
      2. [Open-Closed Principle (OCP) 5](#_bookmark6)
      3. [Liskov Substitution Principle (LSP) 5](#_bookmark7)
      4. [Interface Segregation Principle (ISP) 6](#_bookmark8)
      5. [Don’t Repeat Yourself (DRY) 6](#_bookmark9)
   3. [Object-Oriented Concepts (Explanation of UML Diagram) 6](#_bookmark10)
      1. [Composition 6](#_bookmark11)
      2. [Association Class 7](#_bookmark12)
      3. [Inheritance 7](#_bookmark13)
      4. [Encapsulation/ Information Hiding 7](#_bookmark14)
      5. [Abstraction 8](#_bookmark15)
      6. [Polymorphism 8](#_bookmark16)
   4. [Data Structure 8](#_bookmark17)
3. [Assumption 8](#_bookmark18)
4. [UML Diagram 9](#_bookmark19)
5. [UML Sequence Diagram 10](#_bookmark20)
6. [Test Cases 11](#_bookmark21)

**I. Introduction**

**Restaurant Reservation and Point of Sale System (RRPSS)** is a console-based application designed for restaurant staff to manage restaurant operations. The application covers the key features such as making reservations, recording orders, and displaying sale records.

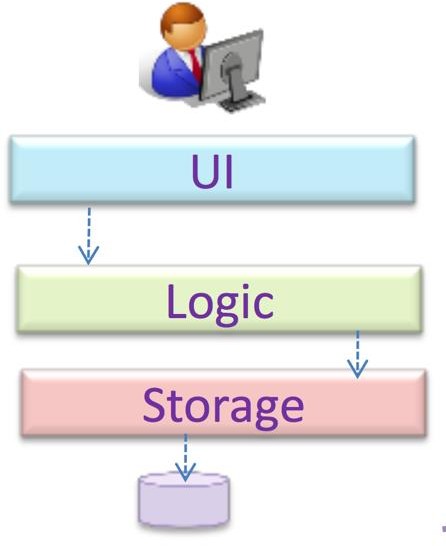
This report covers the object-oriented programming (OOP) concepts and key design considerations used to implement the application. The design will also be represented in a UML Class Diagram showing the interaction and relationship between the objects, and UML Sequence Diagram for one of the functions “Check/Remove reservation booking”. Several test cases are also included to ensure that the application meets the requirements stated.

## Approach Taken

**II. Design Considerations**

OOP concepts are applied comprehensively in this project, in both the design and the implementation of the RRPSS application. Our team aims to make our application easy to maintain and extend.

The architectural style that we have taken is the **N-tier architectural style** where higher layers make use of services provided by lower layers, but lower layers are independent of higher layers. As seen in *Figure 1*, our application’s *UI* classes depend on the *Manager* classes which then interact with the *Data* file for storage. This further reinforces separation of concerns and ensures that the design is easy to maintain and extend.



*Figure 1: Architectural Design*

## Design Principles Used

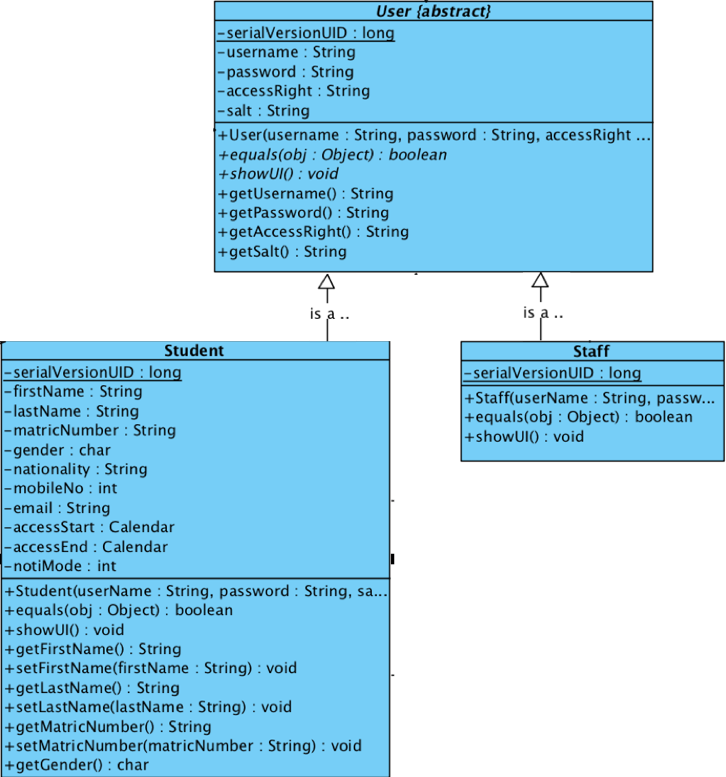
### Single Responsibility Principle (SRP)

The principle states that every software module should have only one reason to change. This means that every class, or similar structure, in the code should have only one job to do.

As an example, the Course class only manages the attributes of a single course – index, course code, course name, etc. Similarly, the Student class only manages the attributes of a single student. This principle is continuously applied in the design, in the UI and Manager classes as well, ensuring cohesiveness. This reduces functional overlaps and also limits the ripple effect when changes are introduced to a specific part of the system.

### Open-Closed Principle (OCP)

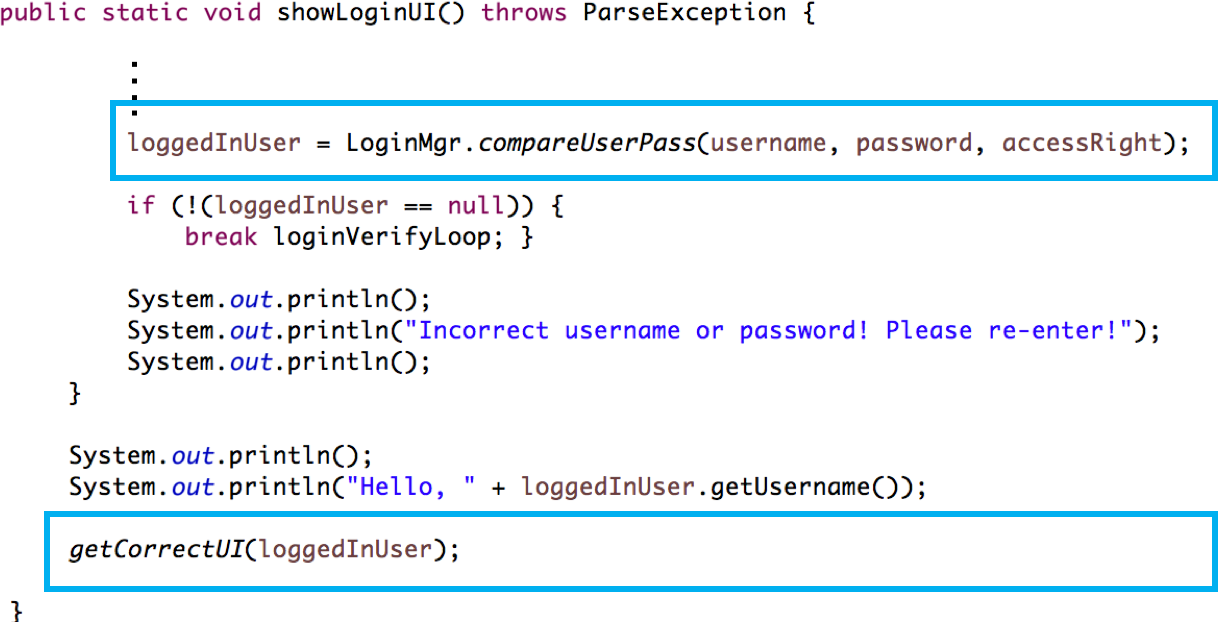
Our group applied OCP where we ensured that modules were open for extension and closed for modification – designing our classes in such a way that extends the functionality of the class without going into the source code and and modifying it.

Our group used inheritance to do this, where the User class is an abstract class that extends to Staff class and Student class which can be seen in *Figure 2*. Now, if for example we want to add a class to represent Student Teacher Assistant (TA) who may need to do both course registration and modification of courses details, it can be done without changing the source code of the modules.

*Figure 2: Abstraction of User Class*

### Liskov Substitution Principle (LSP)

LSP states that any derived class should be able to be used as if it were the parent class, without modification. It ensures that a derived class does not affect the behavior of the parent class.



*Figure 3: showLoginUI()*

For example, the derived classes (*Student* and *Staff)* are **substitutable** for the base class, *User,* while retaining the original behaviour of the class. We made sure that the derived class’s pre-conditions are no stronger than the base class method and its post- conditions are no weaker than the base class method. For example, the method *getCorrectUI* takes in user class as well as the staff and student staff to get the correct UI for the right type of user. This shows that the derived classes are substitutable to the base class.

### Interface Segregation Principle (ISP)

In our design, we ensure that there are no classes that are forced to depend on methods and interfaces they do not use.

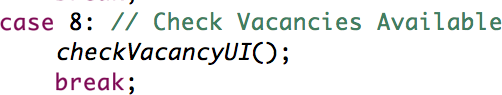
For example, the different *manager* classes are segregated by their functions and they do not depend on any interface that they do not need. This is mainly because of the segregation of responsibilities that we have established for each classes that allows for a good segregation of duties.

### 

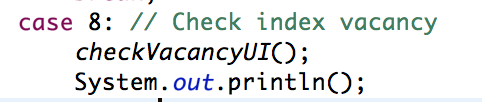
### Don’t Repeat Yourself (DRY)

DRY states that every piece of knowledge must have a single, unambiguous, authoritative representation within a system. This means that there should be no duplication of code and functionality, encouraging code reuse and efficiency.

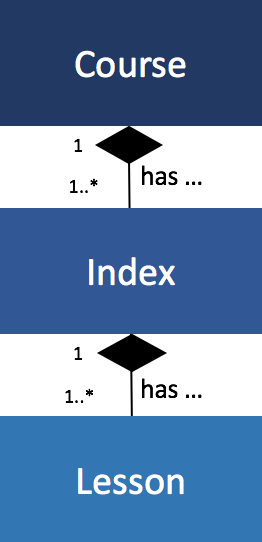
In our application, we used inheritance with the parent class *CourseMgr* so that the methods which are applicable can be accessed by both the children *StudentCourseMgr* and *StaffCourseMgr*. This removes the need for duplication as methods like *printIndexInfo* and *getCourseByCode* can be used by both the child classes.

Graphical user interface, text, application, email

Description automatically generated



*Figure 4: checkVacancyUI is reused multiple times in the application*

Moreover, even in our UI, for functions that are repeated a lot, we make sure that there is only a single representation within the system. For example, the methods *checkVacancyUI* and *getValidCourseCodeUI* are available in the *CouresUI* (parent class) which makes sure that the the functions are represented once in the entire application, instead of repeating the code.

## Object-Oriented Concepts (Explanation of UML Diagram)

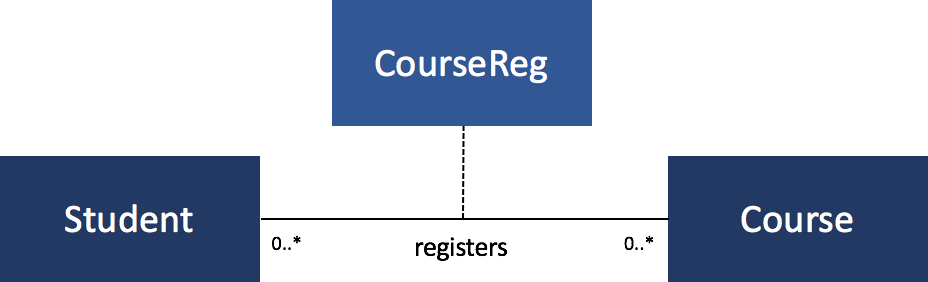
### Composition

We implemented composition relationship between *Course* with *Index* with *Lesson.* This is because index will not exist without course and lesson will not exist without index. In detail, *Course* has a one-to-many relationship with *Index* and *Index* has a one-to-many relationship with *Lesson*. This represent a ‘whole-part’ relationship. As such, in implemented *removeCourse()*, all the indexes will be removed automatically

and the same principle applies to removing an index of a course. *Figure 5: Composition relationship*

*between course, index and lesson*

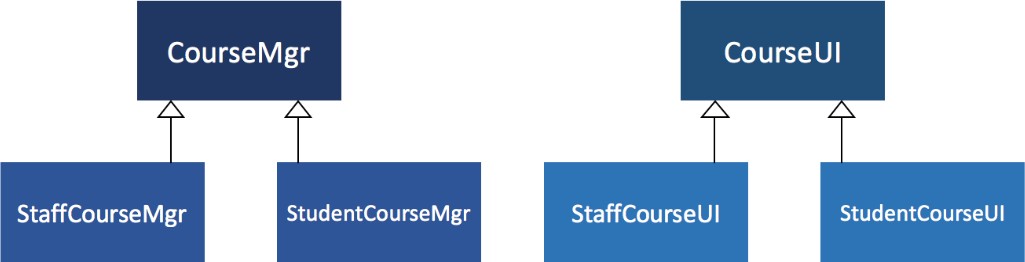
### Association Class



*Figure 6: CourseReg Association Class*

Association class *CourseReg* is used to keep track of the courses registered by each student. This is because there is a need to store additional information about an association but registration data is not specifically owned by either the *Student* or the *Course* object. As such, an additional class is introduced in this case, the *CourseReg* class.

### Inheritance



*Figure 7: Inheritance*

Inheritance is implemented in several classes in the design. For example, *CourseMgr* is the the parent class of *StaffCourseMgr* and *StudentCourseMgr* as both implement common methods so as to encourage code reuse. For instance, both child classes can use methods such as *printIndexDetail* and *printIndexList* which are inherited from the parent class.

The same is applied to *CourseUI* as the parent class of *StaffCourseUI* and *StudentCourseUI*. Both are able to use methods which are inherited from the parents and override any method as needed.

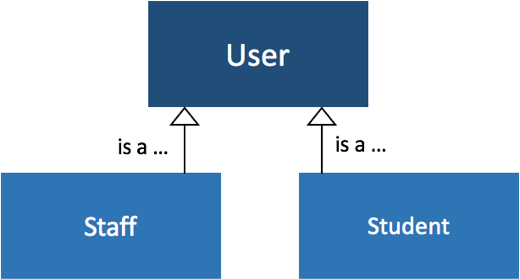
### Encapsulation/ Information Hiding

Encapsulation builds a conceptual barrier to protect an object’s private data. The entities created have private attributes which are only accessible through the get and set methods, making them a read-only class. Information hiding also hides details of the class from the users.

For example, the *Student* class has private attributes such as firstName, lastName, matricNumber, nationality, etc. As they are private, they are only accessible through their respective getter and setter methods. As such, *Student* class has full control of what is stored in its private fields.

### Abstraction

Abstraction is giving information and essential characteristics that distinguish the class from other objects, relative to the point of view of the viewer, making the coding process much easier by reducing the general complexity of the code. For example, the *User* class is an abstract class and the implementation of *getUI()* method is provided by the derived classes *Student* and *Staff.* Moreover, as stated in section b, this further allows for extensibility in the future as



*Figure 8: Abstraction*

new types of users can be added without modifying the method that calls the classes, reinforcing OCP.

### Polymorphism

Polymorphism is when different types of objects are treated as a single general type, but yet each type of object exhibits a different kind of behavior. This is implemented in the *User* class where the *getUI()* method are called depending on the the child classes (*Staff* and *Student)*. The child classes will override the methods in the parent class during runtime. This enhances the functionality of the program as the behaviour of the subclasses can be extended in the future to ensure that extension can be done in the future.

## Data Structure

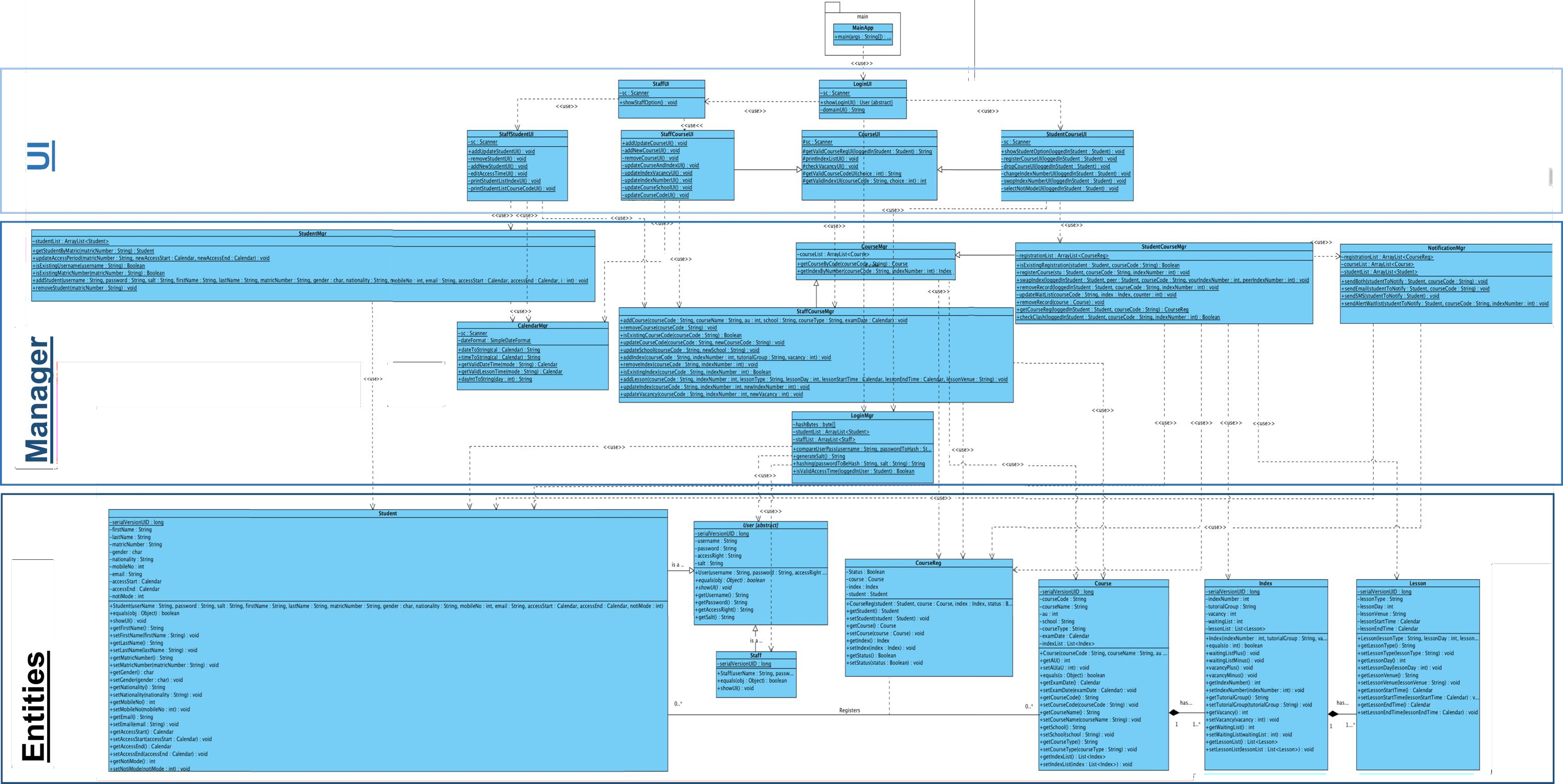
Firstly for file IO, we serialize and deserialize objects to file (.dat). This is preferred over the text file as the code implemented is cleaner and only one .dat file is needed for the entire system.

In implementing the waitlist, a **Stack** is implemented where the top of the list is popped out when a new vacancy is present. As such, the waitlist is implemented on a first-come-first-serve basis.

**III. Assumption**

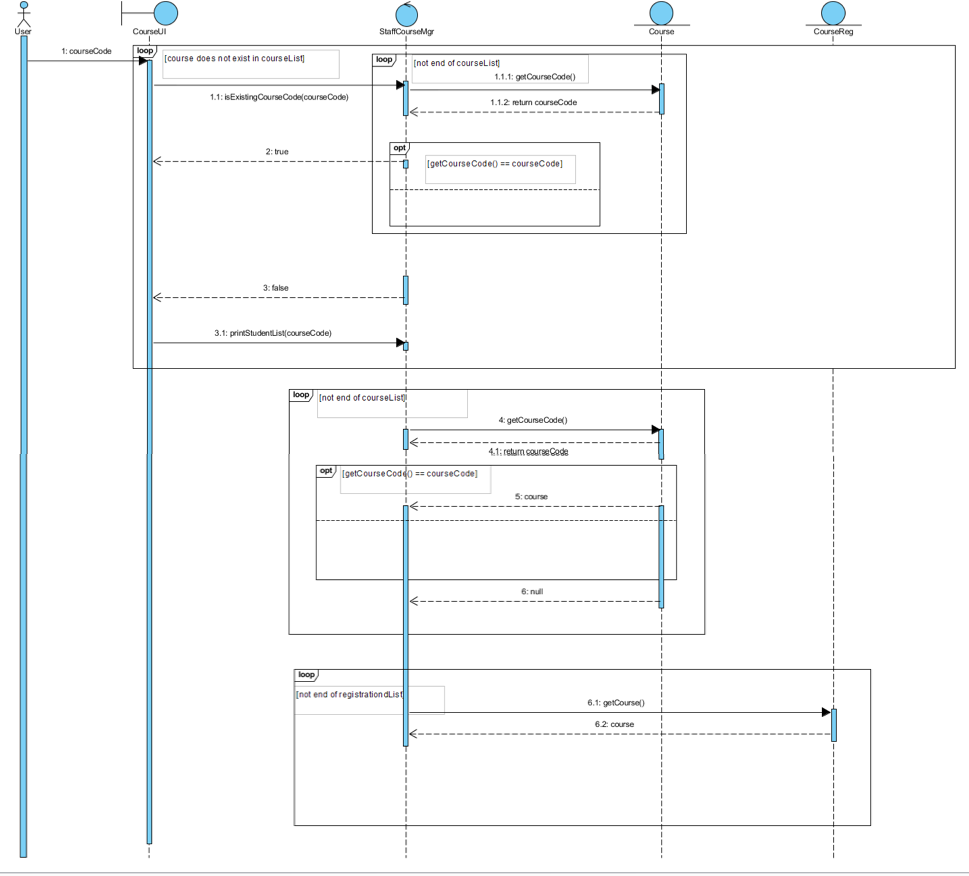
In implementing the code, several assumptions are made:

* All Staff holds the same level of authority as other staff in accessing the MySTAR application.
* Students are unable to update their particulars through the MySTAR portal as the portal is for them to register courses only. Students can, however, modify their notification modes.
* The default password for all students are the matriculation number. Students are assumed to be able to change this default password on another portal, not through MySTAR.
* Examination timings are the same everyday, meaning it is always at 9.30 am, 1.30 am and 4.30 pm from Monday to Friday.
* When swapping index number, user must input the other person’s username and password to continue.



**IV. UML Diagram**

**V. UML Sequence Diagram**

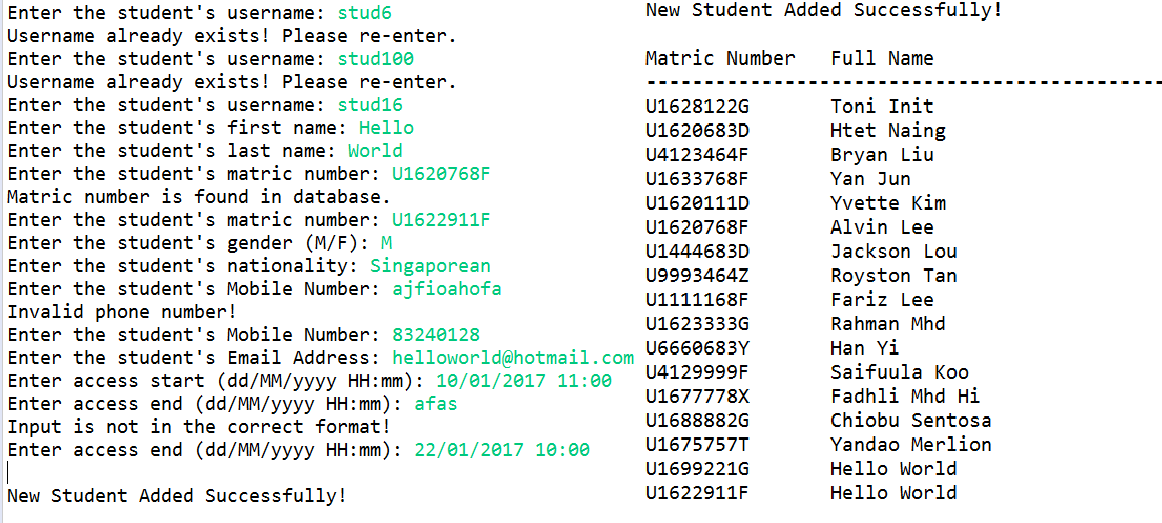


**VI. Test Cases**

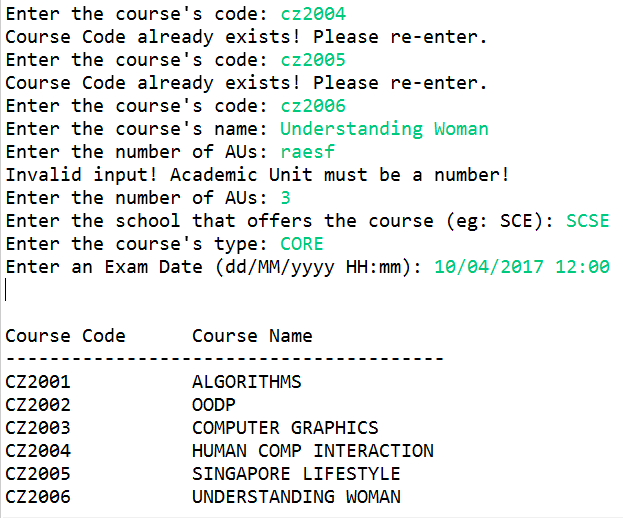
### Student Login

|  |  |  |
| --- | --- | --- |
| **a) Login before allowed period (dates)** | **b) Login after allowed period (dates)** | **c) Wrong Password** |
|  |  |  |

1. **Add a Student**



1. **Add a new student**
2. **Add an existing student**
3. **Invalid data entries**
4. **Add a Course**



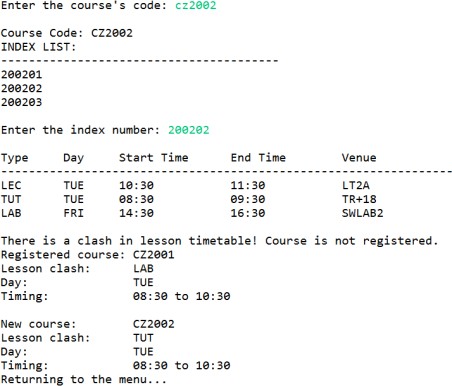
1. **Add a new course**
2. **Add an existing course**
3. **Invalid data entries**
4. **Register student for a course**

|  |  |
| --- | --- |
| **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** |  |
| **d) Invalid data entries (eg wrong student ID / course code, etc)** |  |

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

|  |  |
| --- | --- |
| **a) Check for vacancy in course index** | **b) Invalid data entries (eg course code, class code etc)** |
|  |  |

1. **Day/Time clash with other course**



**a) Add a student to a course index with available vacancies.**

1. **Waitlist notification**

|  |  |
| --- | --- |
| **ai) Add studentA to a course index with 0 vacancies** |  |
| **aii) Drop studentB from the same course index** |  |
| **aiii) Display studentA timetable** |  |

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

|  |  |
| --- | --- |
| **ai) Print list by Course** |  |
| **aii) Print list by Index** |  |
| **b) Invalid data entries (eg course code, index code etc)** |  |

1. **Swap Index Number With Another Student**

|  |  |
| --- | --- |
| **a) To swap index with peer, student required to enter peer’s login details** |  |
| **b) Student need to enter the indexes to swap** |  |

|  |  |
| --- | --- |
| **c) New index is reflected in student’s timetable** |  |

1. **Change Index Number Of A Course**

|  |  |
| --- | --- |
| **a) Staff select the index number to be changed and enter the new index number** |  |
| **b) New index is reflected in the Course** |  |