

1.Introduction

1.1 Objectives

1.2 Overview

2. Design and implementation

2.1 OO Concepts and considerations

2.1.1 Inheritance(Is-a relationship)

2.1.2 Object composition(Has-a relationship)

2.1.3 Extensibility

2.1.4 Other considerations

2.2 Design principles

2.2.1 Single Responsibility Principle(SRP)

2.2.2 Open-Closed Principle(OCP)

2.2.3 Liskov Substitution Principle(LSP)

2.2.4 Interface Segregation Principle(ISP)

2.2.5 Dependency Injection Principle(DIP)

3.Diagrams

3.1 UML Class diagram

3.2 UML Sequence diagram

4. Testing

4.1 Test Cases

1.Introduction

1.1 Objectives

In this project, we are tasked to design a Console-based Student Automated Registration System for school's undergraduates and academic staff, namely MySTARS. The application enables both students and administrative staff to log in to their own account to make changes to information related to course registration.

1.2 Overview

In order to achieve the outcome, we have used 3 types of classes, namely boundary class, controller class and entity classes. Boundary classes include log-in interfaces for students and administrative staff respectively, allowing them to go to their respective accounts. Controller classes are specifically designed to implement the actions that users wish to perform after making a choice in the interface. Entity classes include all entities engaged in this application such as student, administrative staff, school, course, etc. These three types of classes work together to enable a fluent flow of the entire programme. Demonstration of the programme is on YouTube:<https://youtu.be/mpoIJKblbaA> .

2. Design and implementations

2.1 OO Concepts and considerations

2.1.1 Inheritance(Is-a relationship)

Both of the entity classes Admin and Student are subclasses of User class.

Instances of both classes have the same functionality as specified in the User class; however, they have specific functions to themselves as provided in the inherited classes respectively. With proper usage of inheritance, similar attributes such as username, password, firstname and lastname need not to be defined again inside subclasses, which enhances cleanliness of the code. For example, as mentioned above, besides the common attributes that both Student and Admin users have, both of them have specific attributes and functionalities as well. Admin has staffId which specifies the identity number of each Admin object. However, this is not relevant in

Student, where matriculation number, school, nationality and other student-related information is available.

2.1.2 Object composition(Has-a relationship)

In our entity classes, we have used many object compositions to demonstrate the has-a relationship between classes. This is essential as in our case, we need to construct a list of objects belonging to a certain entity class to demonstrate the idea of containment. For example, in reality, each course in NTU has several indexes which have different lesson types such as tutorials, labs or lectures. In our programme, we have implemented object composition to create a list of instances of index class as attribute in the Course class and a list of instances of class LessonType as attribute in the class index. Therefore, we have followed closely to the real-life composition idea and realized these compositions in our code.

2.1.3 Extensibility

Types of users are split into different classes, this means that when a new type of user is introduced into the system, it can readily extend from the User parent class. With the relatively loose coupling of multiple entity and control classes, adding on this new type of user's functionalities would be effortless without having to make major changes to our current programme.

2.1.4 Other considerations

We have made use of the function **readPassword()** from the **Console** output stream to "hide" the password and print "*" instead of the actual characters being printed. While the user inputs the password, we call the method `gethashedpassword()` defined on the main app to hash it with the digits {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'}. The password is then matched with the value of the password stored for each of the student and admin users that have been created. Note that we store only the value of the hashed password, in the database, for each of the student or admin objects.

2.2 Design principles

2.2.1 Single Responsibility Principle(SRP)

The single responsibility principle (SRP) states that a software component must have only a single task. This implies it is in charge of doing just one concrete thing, and it must have only one reason to change. In our case of the project, each controller class has its own purpose and any change in the controller class does not have any implications to the other controller classes. Take for example the two controller classes "AdminController" and "courseController". The "AdminController" class calls the methods `addCourse`, `printCourse` and `removeCourse` from the "courseController" class but both classes are in-charge of different tasks. The AdminController class can add and delete students, which is considered a specific task that cannot be done by the courseController class. Hence, once the methods in the courseController class are called, the courseController class will proceed to do the adding and removing of courses which cannot be done by the AdminController class. This shows that these two classes have their own set of specific functions. Our other controller classes are Database_Controller, StudentController and NotificationController.

The class Database_Controller is responsible for access to the database as it interacts directly with the .dat files. It has functions such as `getCourseByCourseCode()` and `getIndexByCourseIndex()` that are accessible by both the AdminController and the studentController.

StudentController is responsible for the functions that will be accessible to the users who login as students. It has functions like `addCourse()`, `dropCourse()`, `changeIndex()`,

swapIndex(), etc. These functions will eventually be invoked by the boundary class StudentMenu that will directly interact with the users who login as Students. NotificationController is responsible for notifying the student via email when they enrolled in a particular Course.

Hence, we see that our classes follow **high cohesion** and **less coupling** as each of the controller classes have **one responsibility**.

2.2.2 Open-Closed Principle(OCP)

This principle states that the model should be open for extension but closed for modifications. Open for extension – This means that the behavior of the module can be extended. As the requirements change for the application, we can extend the module with new behaviors that satisfy those changes. In our files, we made use of this feature. Take for instance, AdminMenu and StudentMenu implements Menu which is an interface. If there are further changes to the program that requires us to create a new class of user interface, it can implement the Menu interface which proves our point about the extensibility of our program.

Closed for modification – Extending the behavior of a module does not result in changes to the source or binary code of the module. For example, we managed to use encapsulations involving private data members and also get and set methods to access or change the respective variables that we have used for each of the entity classes. This means that foreign users will not get access to the data members directly and hence are “hidden” from the data members. As stated earlier the getter or setter methods have to be used in orders to access and change the data respectively.

```
public class User implements Serializable {
    /**
     * The user type of the user, either Admin or Student.
     */
    private char usertype;
    /**
     * The username of this User which will be used for login.
     */
    private String username;
    /**
     * The account password of this User which will be used for login.
     */
    private String password ;
    /**
     * The first name of this user.
     */
    private String firstName;
    /**
     * The last name of this user.
     */
    private String lastName;
```

Figure of code showing the use of private data members

```
public void setFirstName(String firstName) {
    this.firstName = firstName;
}

/**
 * Changes to the last name of this Student.
 * @param lastName
 * this User's new last name.
 */

public void setLastName(String lastName) {
    this.lastName = lastName;
}
```

```
/**
 * Gets the last name of this User.
 * @return this User's last name.
 */

public String getFirstName(){
    return firstName;
}

public String getLastName() {
    return lastName;
}
```

```
public boolean isTimeClashBIndexes(String oldindex, String newindex) {
    boolean isClash = false;
    ArrayList<LessonType> list = db.getLessonTypeByIndex(oldindex);
    ArrayList<LessonType> newList = db.getLessonTypeByIndex(newindex);
    if (list != null & newList != null) {
        for (int i = 0; i < list.size(); i++) {
            if (list.get(i).isTimeClashBLesson(newList))
                return true;
        }
    }
    if (isClash)
        return true;
    else
        return false;
}
```

Figure of code showing the use of get and set methods to access unauthorised or data that cannot be directly accessed.

The implementation has shown that the design and writing of the code has been done such that new functionality should be added with minor changes in the existing code.

As shown above the design should allow adding of new functionality to new classes, retaining as much as possible existing code.

2.2.3 Liskov Substitution Principle(LSP)

The Liskov Substitution Principle states that, if S is a subtype of T, then objects of type T should be replaced with the objects of type S. In other words, we can say that objects in an application should be replaceable with the instances of their subtypes without modifying the correctness of that application. It states that “subclasses should be substitutable for their base classes”, meaning that code expecting a certain class to be used should work if passed any of this class’ subclasses.

As in our case, the user object has two subtypes: student and admin, which can replace the user object since all the necessary parameters to define a user object are defined by both the student and admin objects. Thus, by the definition of liskov substitution principle which states that if for example an object T has a subtype S then, the objects of type T can be replaced by the object of type S in the respective methods that use the object T, our program satisfies Liskov substitution principle. We have implemented the polymorphism since for each of the entity classes we override the function .equals in order to check if the given parameter is equal to a specific value. Thus the value for which the equality condition is checked changes according to the scenario in which it is applied.

2.2.4 Interface Segregation Principle(ISP)

The ISP principle states that a specific interface shouldn’t be responsible to implement methods that are not required to be implemented together with the other methods i.e., the user should not be exposed to the methods that he/she does not require. This means that the methods should be split across different interfaces that have to be implemented. Thus the principle advocates the necessity of using smaller and cohesive interfaces known as role interfaces. The main function for each of these methods will be to implement methods of a particular behaviour instead of implementing all the methods in one single interface.

Viewing the above conditions to satisfy the rules to implement ISP , we did not find the need to implement ISP since the boundary class ‘menu.java’ is the only interface that has been implemented for our course registration system. Note that the interface ‘menu.java’ is used to call the display method to print all the choices that the student or admin has if he or she were to login in to the system. (explains the use of interfaces since both the student and admin implement menu to display the choices.)

2.2.5 Dependency Injection Principle(DIP)

The Dependency Inversion Principle states that the high level modules in an application should not depend on the low level modules; both should rather depend on abstractions. Both inversion of control and dependency injection are ways that enable you to break the dependencies between the components in your application. The Dependency Injection Principle has various disadvantages. Some of them being: Implementing dependency injection principle increases the complexity of the program as it increases the number of classes with single responsibility which is not always beneficial. The code becomes coupled to the Dependency Injection framework that has been used.

Not only does it have the aforementioned disadvantages but it also has containers that perform type resolving which usually incur a slight runtime penalty which also proves to be disadvantageous. Decoupling makes each task readable as well as understandable, however, it increases the complexity of orchestrating the more complex tasks.

Viewing the above disadvantages , we were able to build the course registration system without the use of this principle making our implementation robust, since each of the class methods and functionalities can be easily changed without having to change multiple classes.

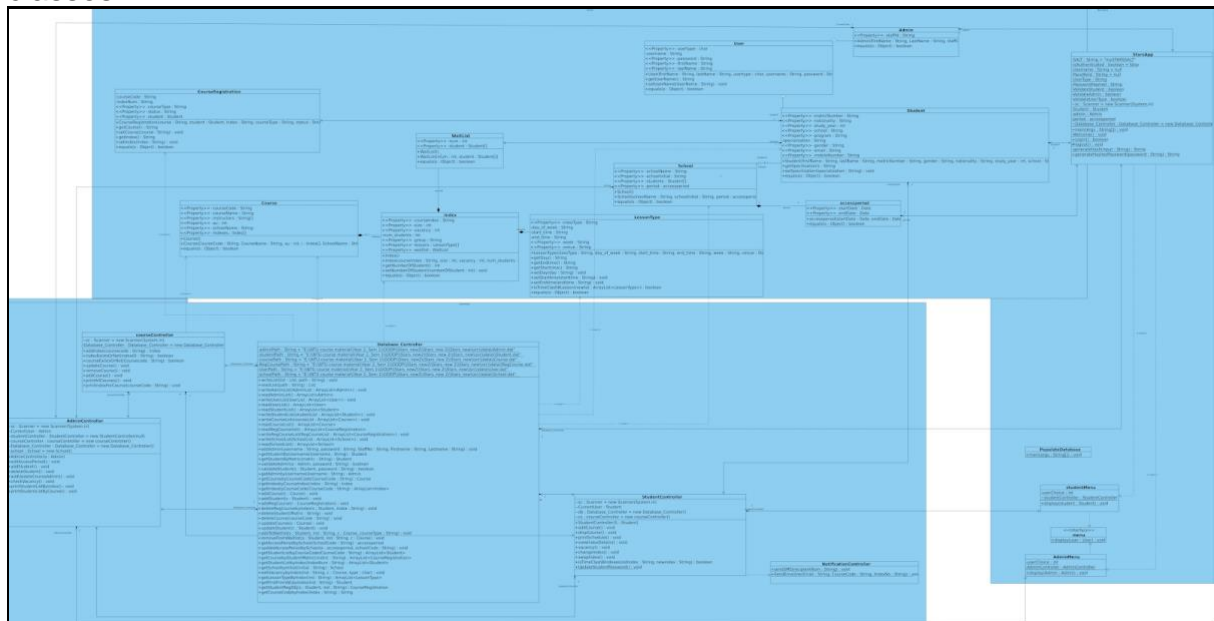
Point to note here is that, with the exclusion of the above principle we were able to achieve high cohesion with each of the controller classes implementing only specific and necessary types of actions.

For example the student controller class takes care of all the functions that a student can perform namely registering for a course or drop course while the admin controller implements methods that an admin user can perform like, adding a new index for a course or changing the access period for the course registration. This example proves the above inference.

3. Diagrams

3.1 UML Class Diagram

In the class diagram, StarsApp depends on the users, studentMenu and AdminMenu, both of which implement menu interface. Hence, each has a realisation relationship with the interface, menu. The class 'studentMenu' depends on 'StudentController' class which inturn depends on the class Database_Controller and uses NotificationController in order to perform the operation of sending a notification. Similarly, AdminMenu depends on AdminController classes, which uses Database_Controller class and uses CourseController. Furthermore the class Database_Controller uses all the entity classes.



School and Student entity classes have an aggregation relationship as they have a whole-part relationship but Student can exist without the creation of School. School and accessperiod, Course and Index, Index and LessonType, Index and WaitList classes all have a composition relationship as in each pair of entity classes, the latter cannot exist on its own without the existence of the former. From this we can conclude that without the course existing none of the other entity classes namely Index, LessonType and WaitList, won't exist. Particularly, School can have 0 to 1 accessperiod, Course can have 1 to many Index, Index can have 1 to many

LessonType and Index owns 0 to 1 WaitList. Student and Admin classes extend to User class respectively, which shows a generalisation relationship.

3.2 UML Sequence Diagram

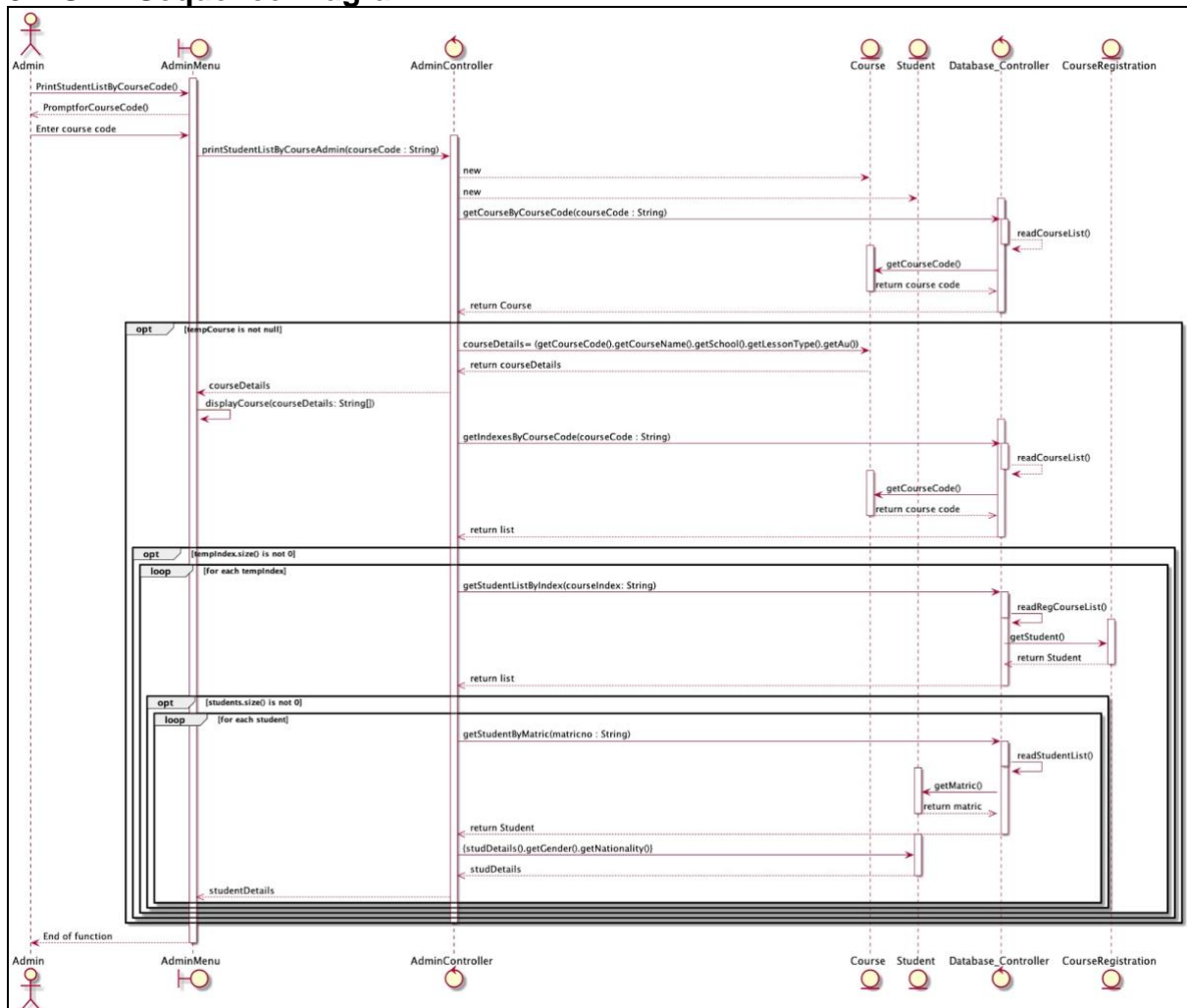


Figure showing the flow of PrintStudentbyList()

The above shows the flow of our program for an admin user. As clearly shown above after login, we prompt a user input for the action he would like to perform. Depending on the choice that the user selects, the particular method is called which inturn leads to the carrying out the specific operation.

4. Testing

4.1 Test Cases

1. Student Login

Test Case	Outcome

a	Login before allowed period (dates)	<pre> StarsApp [Java Application] C:\Program Files\Java\jdk-14.0.2\bin\javaw.exe (25-Nov-2020, 8:02:05 pm) # Welcome to Stars Planner # Username : Atul005 password : password enter the type of user , Student - S and Admin - AS You are not allowed to access now : Sat Sep 11 10:00:00 SGT 2021 Wed Sep 22 22:00:00 </pre>
b	Login after allowed period (dates)	<pre> Username : ZZ005 password : password enter the type of user , Student - S and Admin - AS You are not allowed to access now : Wed Sep 11 10:00:00 SGT 2019 Sun Sep 22 22:00:00 </pre>
c	Wrong password	<pre> Username : Kiran006 password : Password enter the type of user , Student - S and Admin - AS Invalid Username or password, please try again </pre>

2. Add a student

Test Case		Outcome
a	Add a new Student	<pre> Username : CarolinaTan001 password : password enter the type of user , Student - S and Admin - AA Enter your choice 1. Edit Student Access Period 2. Add a Student 3. Remove a Student 4. Add/update/delete a Course 5. Check for Available Vacancies 6. Show Student List by Index Number 7. Show Student List by Course 8. Print All Students 9. Logout </pre>
b	Add an existing Student	<pre> Enter the matriculation number of the student U1922759J Student already exists! </pre>

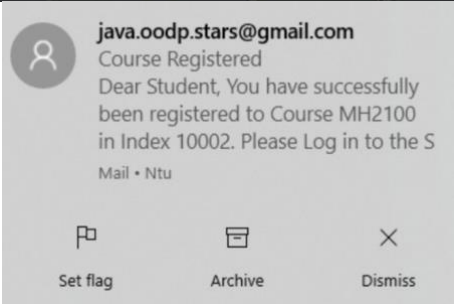
c	Invalid data entries	<pre> Enter your choice 1. Edit Student Access Period 2. Add a Student 3. Remove a Student 4. Add/update/delete a Course 5. Check for Available Vacancies 6. Show Student List by Index Number 7. Show Student List by Course 8. Print All Students 9. Logout 11 Invalid </pre>
d	Remove Student	<pre> Remove a Student ----- Enter the matriculation number of the student U1000000 Student removed! </pre>

3. Add a Course

Test Case		Outcome
a	Add a new course	<pre> Add Course ----- Course code: MH3500 Course name: Stats AU of the course: 4 School of the course: SPMS </pre> <div> Username : CarolinaTan001 password : password enter the type of user , Student - S and Admin - AA Enter your choice 1. Edit Student Access Period 2. Add a Student 3. Remove a Student 4. Add/update/delete a Course 5. Check for Available Vacancies 6. Show Student List by Index Number 7. Show Student List by Course 8. Print All Students 9. Logout </div>
b	Add an existing course	<pre> Add Course ----- Course code: Cz2002 Course already exists! </pre>

c	Invalid data entries	<pre> Enter the Index number of the course that you would like to add: 1 Error. Please Enter correct Index. </pre>
d	Update Course	Course will be updated successfully, with appropriate messages

4. Register student for a course

	Test Case	Outcome
a	Add a student to a course index with available vacancies.	<pre> Enter the Index number of the course that you would like to add: 10002 Select appropriate course type for 10002 1. Core 2. Prescribed 3. Unrestricted 4. Cancel and return to main menu 1 Registered successfully! Email Successfully sent to KIRAN006@e.ntu.edu.sg </pre> 
b	Add a student to a course index with 0 vacancies in Tut / Lab.	Error! There are no vacancies available!
c	Register the same course again	Error! You are already registered for the course!

5. Check available slot in a class (vacancy in a class)

	Test Case	Outcome
a	Check for vacancy in course index	<pre> Check for Available Vacancies ----- enter course code CZ2004 (Size/Vacancy/Waitlist) for 10011: 25/25/0 (Size/Vacancy/Waitlist) for 10012: 25/25/0 (Size/Vacancy/Waitlist) for 10013: 1/0/0 </pre>
b	Invalid data entries (eg course code, class code etc)	Invalid entry, try again!

6.Day/Time clash with other course

	Test Case	Outcome
a	Add a student to a course index with available vacancies.	Added successfully!

7. Waitlist notification

	Test Case	Outcome
a(i)	Add studentA to a course index with 0 vacancies	Error! No vacancies, adding course to the waitlist!
a(ii)	Drop studentB from the same course index	Dropped successfully! Updating Waitlist!
a(iii)	Display studentA timetable	Prints the registered courses.

8. Print student list by index number, course

	Test Case	Outcome
a.	Print list by Course	<pre>Show Student List by Course ----- Enter course code of the course that you wish to print student list: MH2500 Index : 10003 Name:ABHISHEKVAIDYANATHAN Gender:null Nationality:Indian Name:VARUNIYENGAR Gender:null Nationality:Indian Index : 10004 10004does not have any students.</pre>
b.	Print list by Index	Same as above. The only change is the list is printed according to the index instead of the course
c.	Invalid data entries (eg course code, index code etc)	Invalid entry, try again!

References

- Password Hashing : <https://www.baeldung.com/java-password-hashing>
- Password Masking:
<http://www.cse.chalmers.se/edu/year/2018/course/TDA602/Eraserlab/pwdmasking.html>
- Sequence Diagram: <https://plantuml.com/sequence-diagram>
- Class Diagram:
<https://circle.visual-paradigm.com/docs/uml-and-sysml/class-diagram/>