

**CZ2002 OODP Final Project**

**Lab Group: SS5**

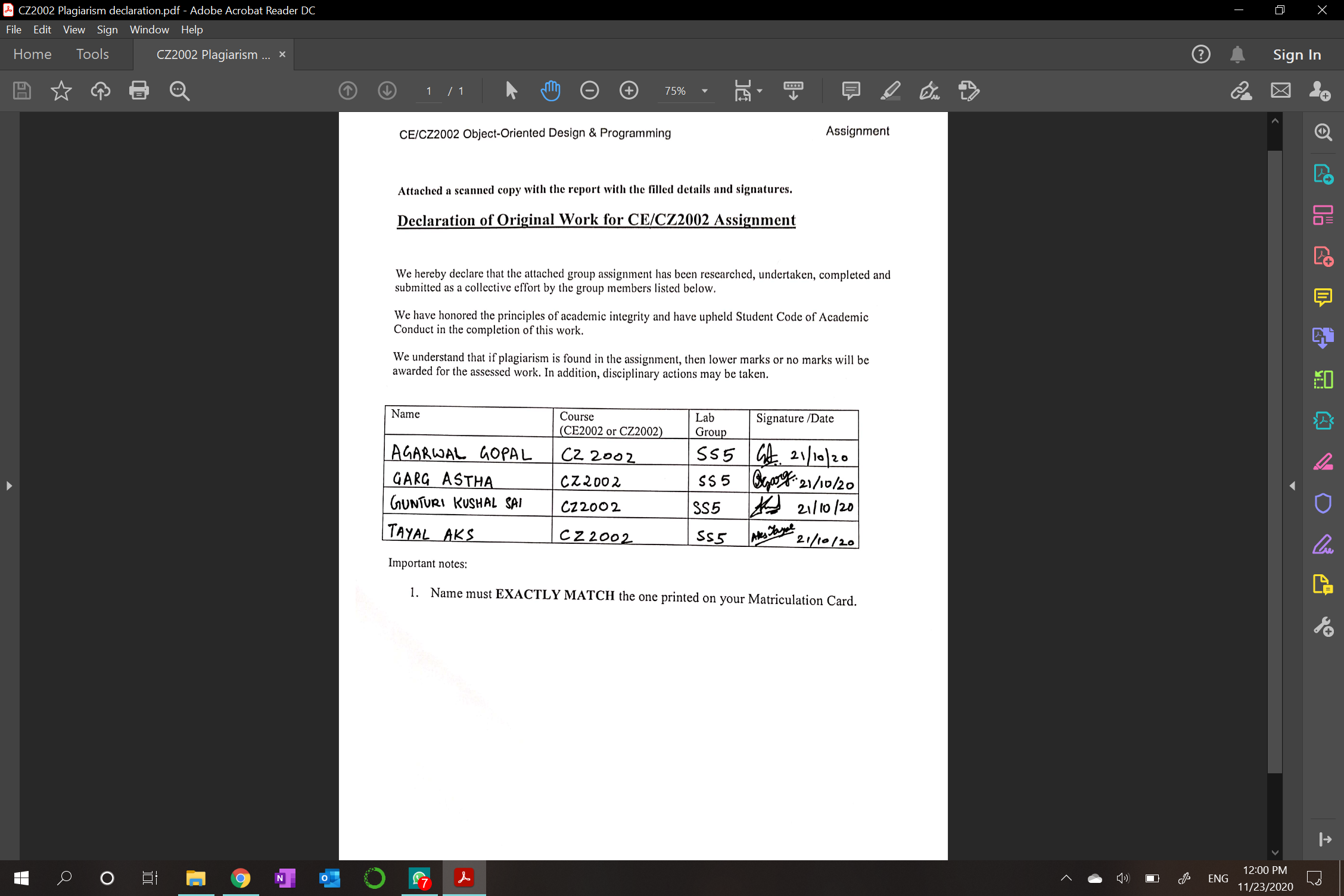
**Group Members:**

**Agarwal Gopal**

**Garg Astha**

**Gunturi Kushal Sai**

**Tayal Aks**



**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. The application covers 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.

**Approach Taken**

Our main aim in designing the application was modularity, cohesiveness between the modules (with loose coupling between classes), and space for modification and extension. An application like this needs to allow for updates and changes. Therefore, we kept this goal in mind.

We divided it into three parts- UI, control and entities. Every part is hidden from others and only knows what is required for carrying out the necessary requirements.

Thus, the UI depends on the control classes which take care of different functionalities including the database handling, hence, hiding the data and preventing it from being directly accessed.

**Design Considerations**

**Design Principles Used**

**Single Responsibility Principle (SRP)**

The principle implies that every class should have only one responsibility. As the class would perform only one job, if there is any change in the implementation of that job, only then would the code have to be updated. Therefore, the code will be changed only for one reason. This ensures cohesiveness and makes error checking easier.

We have used this principle in assigning only one responsibility to each class. For example, one of the many manager classes - DataListManager has just one responsibility of handling all the requests from other classes when they want to retrieve data. Also, The FileHandler class has just one responsibility of interacting with the text files. The ValidationManager class also assumes responsibility of only validating certain inputs when entered by the User. Additionally, there are separate packages for handling UI, control and entities, with separate classes responsible for a set of related functions, thereby following this design principle.

**Open-closed Principle (OCP)**

“A model should be open for extension but closed for modification” defines this principle. In other words, we want to be able to change what a module does without changing the original source code.

For applying this principle, the User class has been made abstract, which contains the attributes common to all the users like username, password, gender, etc. and the corresponding getter and setter functions. Therefore, for introducing a new category of users, we can directly extend from this class and implement the functionality without making any changes to it, as it is done for Student and Staff.

**Liskov Substitution Principle (LSP)**

In simple words, the principle states that subtypes must be substituted for their base types.

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, all the classes implementing NotificationManager - SMSNotification Manager and EmailNotificationManager, can sendNotification as required, without the need for additional information from the calling method.

**Interface Segregation Principle (ISP)**

This principle states that classes should not depend on interfaces that they do not use. To follow this, we have avoided the usage of FAT interfaces. Instead, we have implemented many different interfaces for specific clients.

For e.g. we have separate UI for Login, Student, Staff. Only the appropriate menu is shown for specific users. The StudentUI itself has different methods for calling the appropriate function from the control classes for performing different tasks.

**Dependency Injection Principle (DIP)**

High level modules should not depend on low level modules rather both should depend upon abstraction. This allows the high level modules to be re-used quite simply. We have implemented this principle by making the NotificationManager an interface. Therefore, if new methods of notification have to be implemented, then a new class can be created which defines the sendNotification method according to the required mode. This allows the high level module to be reused simply.

**The UML Class Diagram**

The project is divided into three packages - UI, manager and entities.

The UI package consists of classes that are concerned with interacting with the user, the different manager classes take care of the functionalities and make use of the entity classes.

There are three separate UI classes, for login, Student and Staff, which show the respective menu. These UI classes are dependent on the classes in the manager package.

The different manager classes perform different functions for e.g.

* LoginManager handles logging in,
* the ValidationManager checks for the correctness of the input,
* FileHandler and DataListManager perform file I/O and return the appropriate information,
* CalendarManager handles all date related functions,
* StaffCourseManager is for the staff to do course related functions,
* StudentCourseManager handles registration and other course methods for students while StaffCourseManager does that for the Staff,
* StudentStaffManager performs the student functions for the staff,
* NotificationManager handles the sending of notification in the required format, after an update in the waitlist.

The User is an abstract class which is extended by Student and Staff. Other classes in this package include- Course, Index, Lesson, CourseRegistration (which is a record of all the students which registered for the course along with the indexNumber)

A picture containing timeline

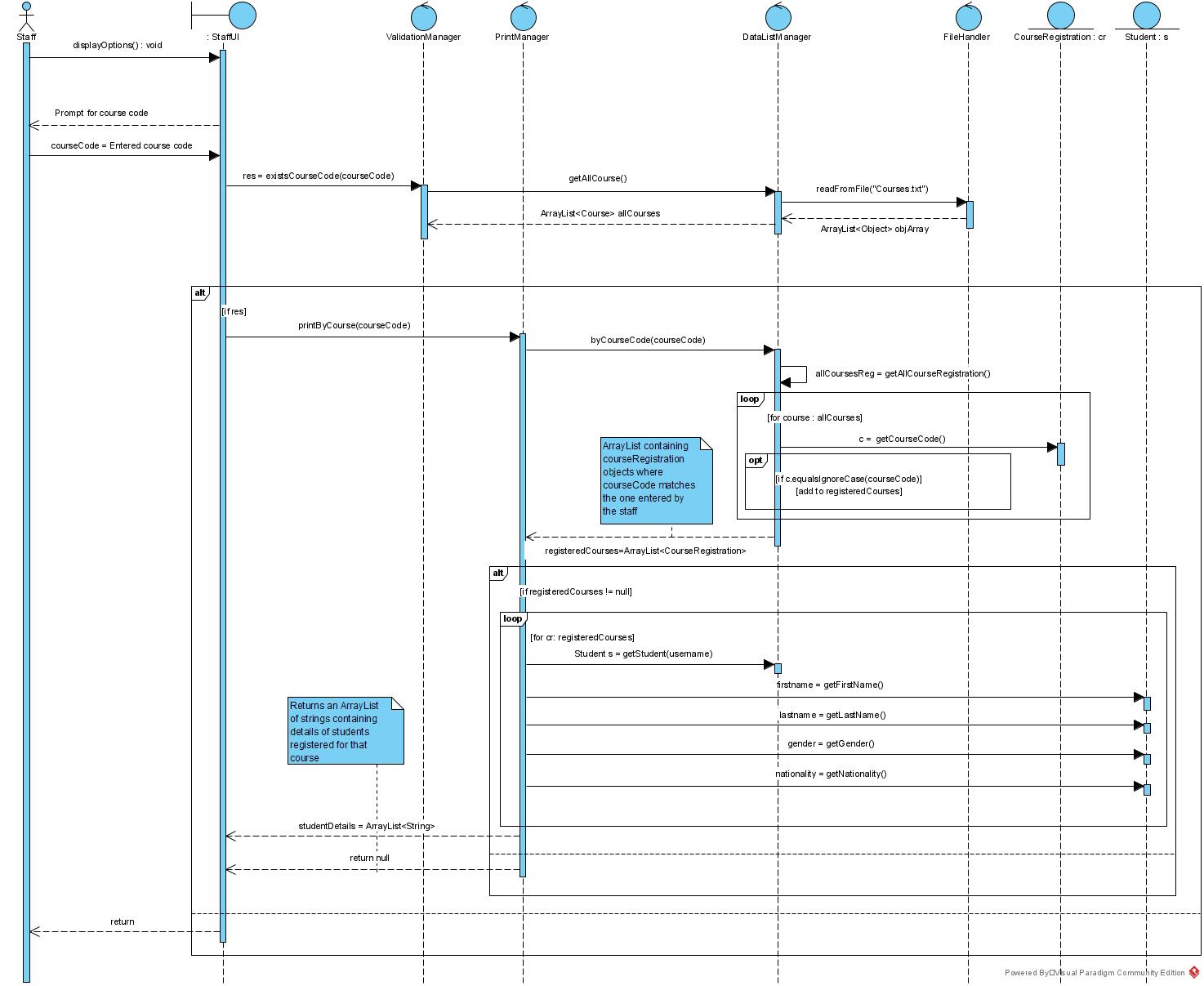
Description automatically generated

Entity

UI

Manager

**UML Sequence Diagram**



**Flow of the diagram**

After logging in as a staff, the StaffUI displays the menu. Upon selecting the option for printing the student list by courseCode, the user is asked to enter the courseCode.

The courseCode is passed a parameter to the function existsCourseCode() in ValidationManager to check if this courseCode is valid. Inside this function, getAllCourse() is called from the DataListManager in order to retrieve the ArrayList or Course type objects called allCourses. The getAllCourse() function calls the readFromFile() function from the FileHandler class which returns to it an ArrayList of Object type objects. After explicitly converting them to Course type objects it is returned to the Validation Manager which loops through it to find out if the courseCode is present as the course code of any of the objects in allCourses. This function returns a boolean value which is true if the courseCode is present and false otherwise. If the received res(boolean) is true, the printByCourse function from the PrintManager is invoked and the courseCode is passed as the parameter. This function first calls byCourseCode() from the DataListManager which in turn calls getAllCourseRegistration() from its own class to retrieve all records of registered courses. We then loop over the retrieved ArrayList of courseRegistration type objects and append those objects which have the course code same as the one entered by the user. The new ArrayList to which we appended all these objects is then returned to the PrintManager. If this ArrayList called registeredCourses is not null, we loop through it and for each CourseRegistration type object CR, we get the Student S whose username matches the one in the CR. We then retrieve its first name, last name, gender and nationality from the respective getter function in the Student class. All these values are concatenated into a String and an ArrayList of String is finally returned to the StaffUI for displaying. A null ArrayList is returned in case registeredCourses is null. In the case that the courseCode does not exist, meaning that res(boolean) is false, an appropriate error message is displayed, and Staff is once again presented the menu.

**Object-oriented Concepts (Explanation of the UML Diagram)**

**Encapsulation**

This refers to binding the data and functions so as to protect them from outside. This has been used everywhere in the design. The only way two classes can interact is by the way of calling the member methods. The method performs the function and returns the result. This way, the internal functioning is hidden from the class which called the method ensuring data protection.

**Abstraction**

The main goal of this concept is to hide unnecessary details from the user and show only what is required. For e.g., when a student chooses to register for a course, the UI class calls the function from the StudentCourseManager without knowing how exactly it will be done. The StudentCourseManager in turn calls the DataListManager to obtain data from the files. The data control class just returns the required information. This way, the StudentCourseManager does not need to know how the actual reading of the file is performed as it is not concerned with it.

**Inheritance**

This allows one class to inherit properties from another class while defining its own attributes or methods. This was used in implementing Student and Staff classes that extend the User class. Since both the classes have common attributes and functions like username, password, gender etc, having a common parent class prevents duplication of these attributes and methods. Furthermore, introducing a new user is also simple as the common features can be inherited from User.

**Polymorphism**

This concept allows overloading of methods i.e. two methods with the same name but different parameters will behave differently. In our case, we have implemented this at several places, one of them being - updateAccessTime(String, Calendar, Calendar) and updateAccessTime(Calendar, Calendar). In the first case the access time of a particular student is changed and hence the String parameter of username is required. On the other hand, the latter function changes access time for all students present in the database and no username is needed.

**Data Structure**

Firstly, for implementing File I/O, we have used Serialization. This way we can easily save entire objects with all the attributes and methods. Hence, on updating something specific we can just save the updated object itself instead of reading the specific information and replacing it. There are separate text files (.txt) for every entity.

Furthermore, a queue has been used to implement the waiting list, so new students can be added from the back end and the topmost student can be removed easily to maintain the first-come-first-serve rule.

**Assumptions**

* The student is not given the option to edit any of his/her details as the application is for the sole purpose of registration of courses.
* The course code, index number and lesson ID are unique across all courses.
* For swapping the index number for a course with another student, the password of the other student is needed.
* When adding a student, the default password is the username which is case sensitive.
* Throughout the application, the username is not case sensitive.
* AU limit has been set to 10 for all students.
* Staff cannot be added from the application.

**Demo and Test Cases**

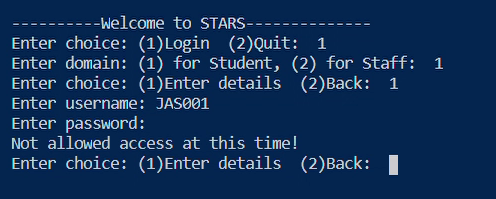
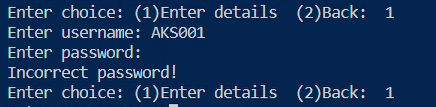
The youtube link for the video - https://www.youtube.com/watch?v=RjbVyXoEavQ&feature=youtu.be

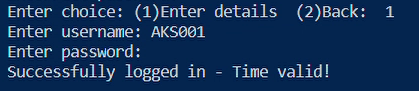
The admin already existing has username - ADMIN1, and password ADMIN1

A README text file has been added which specifies the database information.

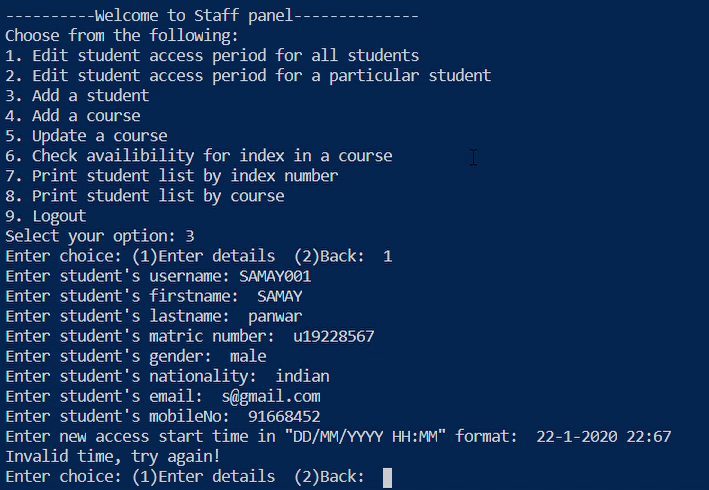
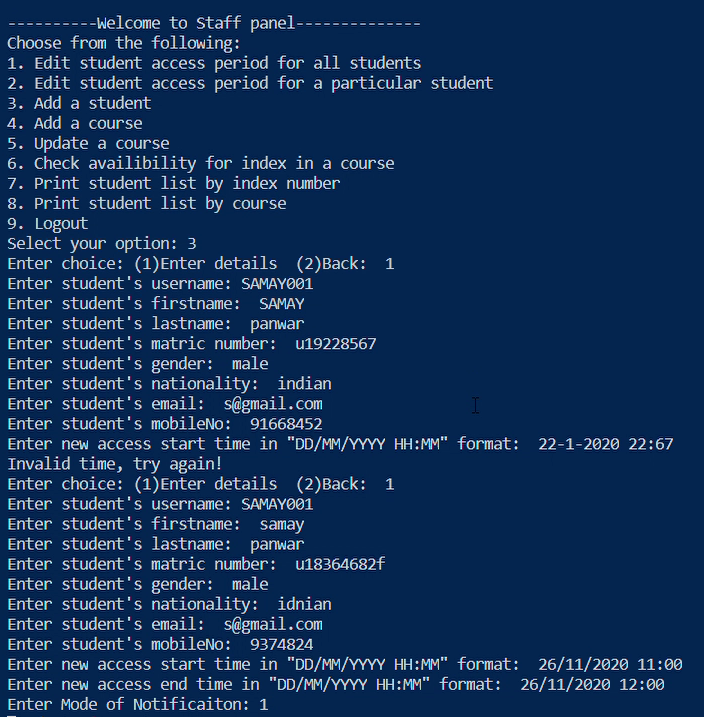
**Test Case 1**

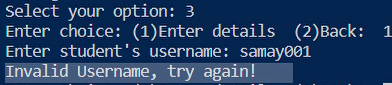
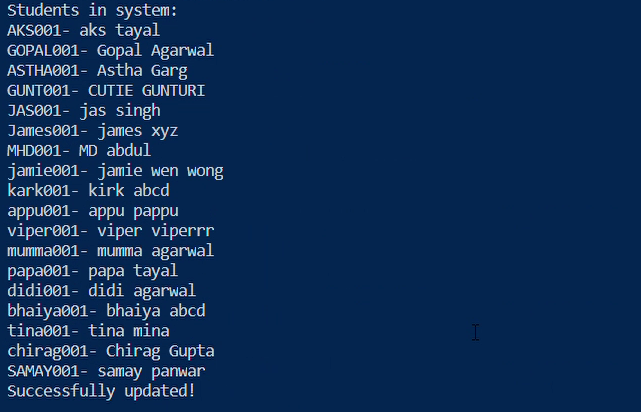
1. **Student Login**
2. Login beyond access time b. Incorrect password



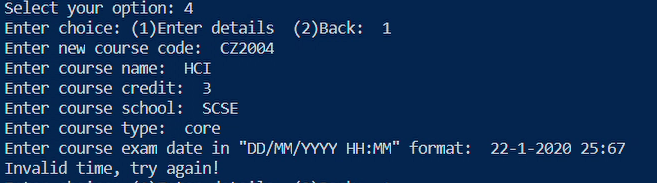
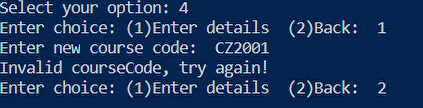
c. Successful login

**2. Add a student**

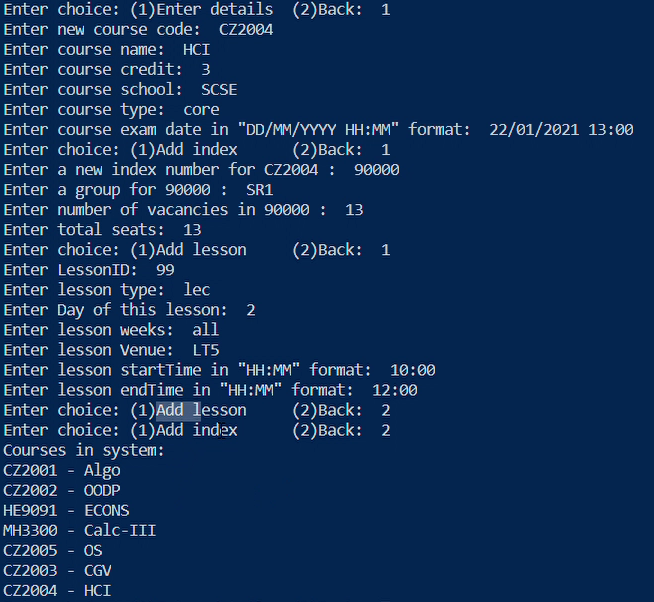
1. Wrong date format in input c. Adding a student

c. Adding an existing student 

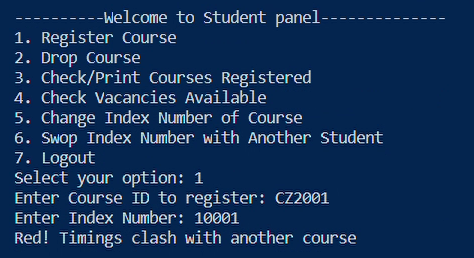
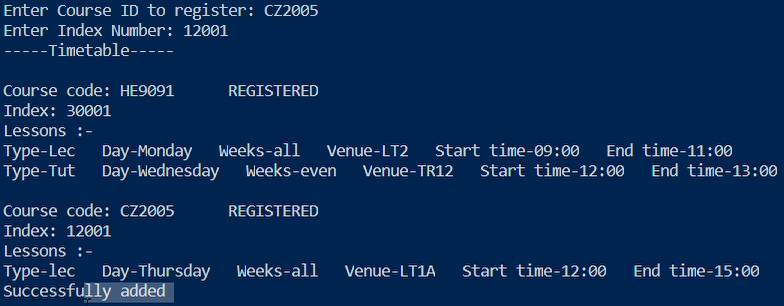
**3. Add a course**

1. Invalid data entries b. Adding an existing course

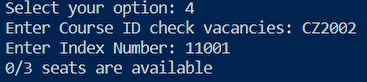
c. Adding course correctly

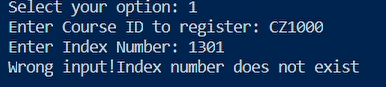
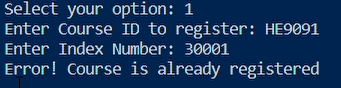


**4. Register student for a course**

1. Clash in timings b. Successful registration

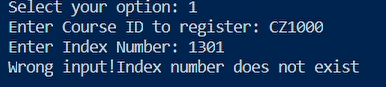
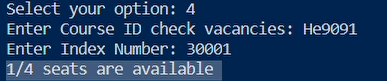
c. With 0 vacancy Timetable



d. Registering existing course e. Invalid data entry

**5. Check available slot in a class**

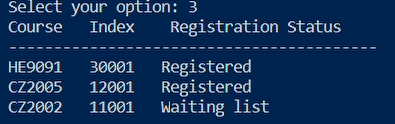
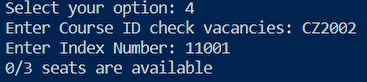
1. Output as vacancy/total b. Wrong data entry



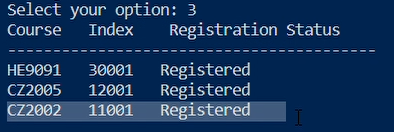
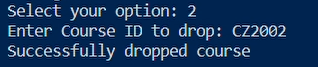
**6. Date/Time clash for course registration** - Demonstrated in 4 a.

**7. Waitlist Notification**

1. Add to a course with 0 vacancy to studentA



1. Dropping course from another studentB c. Timetable for studentA

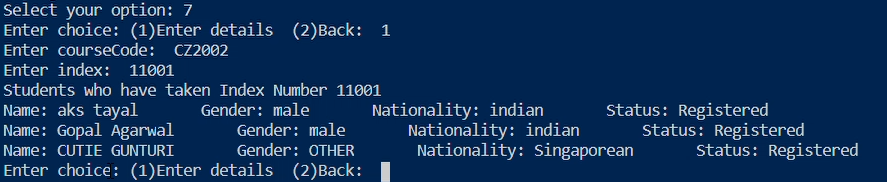


d. E-mail notification



**8. Print student list by index number, course code**

1. Printing by indexNumber



b. Printing by courseCode

