

Project 8:

MedC Management System

Salil Sameer Godbole (2021A7PS2004P)

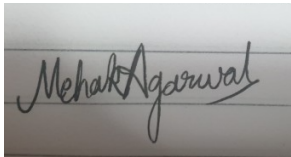
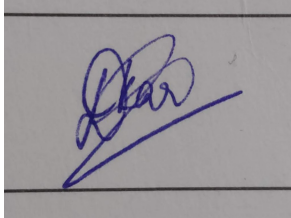
Debjit Kar (2020A7PS0970P)

Mehak Agarwal (2020B3A70868P)

Vedant Verma (2020A3PS0356P)

Plagiarism Statement

We certify that this assignment is our own work, based on our personal study and research and that we have acknowledged all material and sources used in its preparation, whether they be books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. I also certify that this assignment has not previously been submitted for assessment in any other unit, except where specific permission has been granted from all unit coordinators involved, or at any other time in this unit, and that we have not copied in part or whole or otherwise plagiarised the work of other students and persons.

Name	Contribution	Signature
Salil Sameer Godbole	Handling the inputs, taking user list, notice board, medicine list, appointment list, order list inputs in the given format (from a .csv file) and returning them in the correct data type. Creating updated UML diagrams. Class description documentation, Solid Principles of OOP documentation.	
Mehak Agarwal	Choice of collections and their implementation in AppData class, Doctor class (getter setter methods), Student class (methods implemented: checkNoticeBoard, purchaseMeds, checkDues), Admin class (addNotice, checkAllDues), Design patterns documentation, multithreading implementation and documentation.	
Debjit Kar	Developed the classes and functionalities required for handling booking of appointments, designed and developed the command line interface, implemented multithreading, coded the logic for decision making in the program	
Vedant Verma	Worked on logic behind input handling, built the file-based database system for storing input between sessions, integrated everybody's code into a uniform final programme	Vedant

Design Pattern page

If we had to redo the project, we would use a Singleton design pattern for the app data class as we only need one shared instance of it across all classes. At the moment we are using shared fields of the class through the static method. All the collections in the AppData class are static and being statically accessed and updated by the methods that need them.

We would have also used the Builder design pattern for the registration of the student by reducing the required parameters for the construction of the object. There can be two concrete builder classes admin and the student, which can ask the client for the necessary information like the name of the student, BITS ID and can fill in for the voids such as mobile number without explicitly specifying 'NULL' for it.

We could have also implemented the Command Design pattern on our GUI as it would reduce the coupling between the invoker and the receiver of the command. We would have prevented the object from our command line to directly be called and added some classes which in turn would have handled the implementation of deferred execution of operations.

This document gives a brief idea about the classes that have been implemented and their attributes and methods. It also provides a small idea about how the input and output work. GitHub and GDrive Links at the end.

Description of Classes:

1. Student

This class holds the data of the student. It has attributes such as Student ID, Name, Email and Mobile number. The Open-Closed principle is followed here wherein any new functionality of the student can be extended to the student

class. Different methods are present to book appointments, view the notice board, purchase medicines, check the student's dues, display the students information and store the student's information to a file. It has a constructor that assigns values to the variables when an object is instantiated. It has the following methods implemented

- **takeAppointment**
It takes in the timestamp, doctor id, day of appointment and time of appointment as input from the user and instantiates an appointment object with said values and sees if the appointment can be added to the doctor's slot
- **checkNoticeBoard**
It shows the Notice board describing the details of doctors available. Each notice corresponds to a doctor.
- **purchaseMeds**
It takes in the Medicine ID, quantity required and the mode of payment as input from the user and updates the inventory if the medicine is available. It adds the cost of the medicine to the SWD account of the user if the user chooses to pay later.
- **checkDues**
It displays the dues of the user associated with the SWD account.
- **getName**
It displays the name of the student.

2. Appointment

This class creates an appointment object with the data inputted by the user if it is not clashing with any other appointments for the given slot with the given doctor. This process is explained in the multithreading portion of the documentation. It has attributes such as timestamp at the booking time, student id, doctor id, the day of appointment required and the time of appointment. Its constructor adds the appointment object to the appointment data structure of the doctor in question.

3. Notice

This class is used to make notice objects. It has an object of Doctor class (the doctor to which the notice corresponds) as an attribute and has a method called

'show' which shows the information regarding the doctor in the form of a string representing Doctor ID, Doctor name, consultation type, days available and time of availability. These notice objects are stored in an arraylist called noticeboard in AppData.

4. Admin

This class handles the responsibilities of the admin of updating the notices, managing the record of every student, and facilitating the medical store owner to see the summary of purchases done. The Open-Closed principle is followed here wherein any new functionality of the admin can be extended to the admin class. It has the following methods implemented

- **createNotice**
It creates notices to be displayed on the notice board.
- **deleteNotice**
It deletes a notice from the notice board.
- **checkAllDues**
It displays all the dues generated.
- **netSales**
It displays the record of all the sales conducted.

5. Doctor

This class holds the data of the doctor. It has attributes such as Doctor ID, Consultation Type, Days Available, Start Time, End Time and Appointments. It follows the Open-Closed principle, any new functionality of doctor can be extended to the doctor class. There are different getter and setter methods to display and update the doctor's information, appointment details. The dependency inversion principle is also followed. It has a constructor that assigns values to the variable when an object is instantiated. It has the following methods implemented

- **getID**
It displays the ID of the doctor.
- **getAppts**
It takes in a day and displays the doctor's appointments on that day.

- **getName**
It displays the name of the doctor.
- **getConsultationType**
It displays the type of consultation.
- **getDaysAvailable**
It displays the days the doctor is available.
- **setConsultationType**
It takes a mode of consultation and assigns it as the consultation type.
- **setStartTime**
It takes in a time of the day and assigns it as the doctor's start time.
- **setEndTime**
It takes in a time of the day and assigns it as the doctor's end time.

6. Medicine

This class holds the data of the medicines. It has attributes such as Name, Medicine ID, Price and Quantity. It has a constructor that assigns values to the variable when an object is instantiated. It follows the Open-Closed principle, any new functionality can be extended to the medicine class. There are different methods to update and display the medicine details. It has the following methods implemented

- **getName**
It displays the name of the medicine.
- **getID**
It displays the ID of the medicine.
- **getPrice**
It displays the price of the medicine.
- **getQuantity**
It displays the quantity of the medicine.
- **setQuantity**
It takes in an integer and assigns it as the quantity of the medicine.

7. AppData

It holds the appointment related information, like the doctor, student and medicine, notice and dues generated databases. It has attributes such as docs, studs, inventory, notices, dues and sale.

Solid Principles of OOP

- The Single Responsibility Principle has not been used since the classes perform multiple functionalities.
- The Open-Closed principle has been used, since different functionalities of classes can be implemented using subclasses, without disturbing the parent class. It is used in the Student, Doctor, Admin and Medicine classes.
- The Liskov Substitution Principle has not been implemented since we have not used subclasses.
- The Interface Segregation principle has not been used since purpose-specific interfaces have not been used.
- The Dependency Inversion Principle has not been used, since abstract classes and interfaces have not been implemented.

Multithreading

Every student and admin object that is created is essentially a new thread.

Multithreading is required in areas in which multiple users might try to simultaneously update a collection where data is stored, i.e., the `addNotice()` method in admin and the `takeAppointment()` method.

The `addNotice` method in the admin class contains a synchronized block that takes the noticeboard, an arraylist that is stored in `AppData`, as its lock. This way only one admin thread at a time can update the noticeboard.

To understand the way the `takeAppointment()` method works, we need to understand how the appt (appointment) data structure in the doctor class works. Each doctor object will have a field called `appt` which is an arraylist of hash sets of the `Appointment` type. Each hashset corresponds to one day of the week. So the arraylist contains seven hash-sets of the `Appointment` type corresponding to each day of the week.

Now when a student is trying to book an appointment, they enter the Doctor's name, the time slot they want, and the day of the appointment. This data is parsed and the doctor's name is used to fetch the appt arraylist and the desired day for the appointment is used to get the hashset of appointments corresponding to that day. Now this hashset is used as a lock in the booking of appointments synchronized block. Therefore, two students can book appointments with the same doctor concurrently as long as the day for the appointment is different. Two students cannot simultaneously book an appointment with the same doctor on the same day.

Video Drive Link

https://drive.google.com/drive/folders/1me8Nh6mBYZFcRUd3Riqx5b-i2-R45aqY?usp=share_link