COSC2440: Software Architecture Design and Implementation

Student enrolment management system overview

Student: Mai Lê Anh Huy (s3836278)

Contents

[1. Introduction 3](#_Toc70846706)

[1.1. Value 4](#_Toc70846707)

[1.2. Scope 5](#_Toc70846708)

[2. System Overview 3](#_Toc70846706)

[2.1. Conventions 5](#_Toc70846708)

[2.2. Features list 5](#_Toc70846708)

[3. Functional Requirements 8](#_Toc70846710)

[3.1. Class diagrams 9](#_Toc70846711)

[3.2. Use case diagrams 9](#_Toc70846711)

[3.3. ER diagrams 17](#_Toc70846713)

[4. Non Functional Requirements 8](#_Toc70846710)

I use no other resources than writing this myself

**https://www.quora.com/How-did-people-manage-student-enrollments-in-the-past-without-a-database-like-nowadays?\_\_filter\_\_=all&\_\_nsrc\_\_=notif\_page&\_\_sncid\_\_=25523425834&\_\_snid3\_\_=34827428696**

1: Introduction

In the modern world, the job of managing student enrolments in schools, universities is one of the top crucial requirements that must be performed precisely. Moreover, there are several protocols in regard to this job from the need of storing adequate data, the need of performing operations on data to the need of efficiency of time. With the protocols, an academic assistant will have a hard time to handling such requirement if the tools, system that they use is not appropriate.

In regard to the limitations of managing student enrolments in the past. Enrolments were stored in paper and modified by pen. There would be a teacher in a large room categorized sheets of appropriate enrolment of various classes into specific desks and students had to walk to the correct desk to fill in the data and to the schedule they carried. All of the limitations in the past raised a request for building more intricate softwares nowadays and they are used even more efficient, precise.

There are various software that manage student enrolments but in our project we also decided to build a Student Enrolment Management System.

1.1 Value.

Our aim is to make the software well-known to everyone so that we can further process our future directions which are advertising, monitizing and so on. However there is one thing that need to be considered.

This software is still on the experimental level meaning that we can not publish the software without the completed implementation of other components. By experimental, we meant that the software is still ready to be used however for instance, the backend of the software has not fully developed, you can understand this clearer in **Scope** section

1.2 Scope.

|  |  |
| --- | --- |
| In Scope | Out Scope |
| Implementations of standard CRUD functionaties. | Data persistence: Saves modified list of enrolment in a file when the program terminates. Load that file in to the enrolment list when the program loads |
| Calculates and obtains precise results | Use other commands than the in-built commands to query. |
| Saves to CSV file (report.csv) | Minor features such as printing to paper, sending data, etc. |

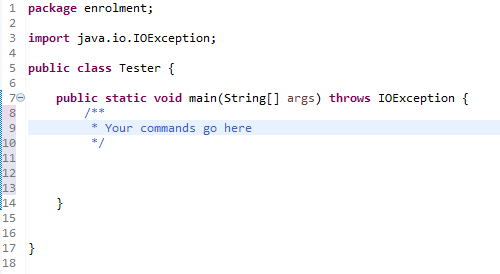
Note that in regard to the Data Persistence. This limitation has caused this specific issue:

Each time a user input a value, that value will modified the list of enrolments if certain commands are used. However as mentioned, when the program stop executing, the list will erase all changes meaning that when the user load the software again, he can not query the previous list and the list will be the same as the very first list. He has to input the same value again as a result.

*Best practice:* As the limitation mentioned, the best try is actually to imagine what will happen after a piece of command executed only then you chain another command below it. More example will be explained but for now because the system behave in that way, we called our system a **Procedural System.**

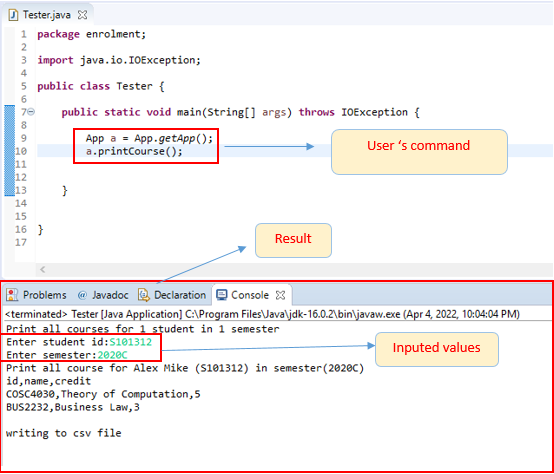
2.System Overview

The user enters the Tester class in order to use the system. Here is what the user area should look like.



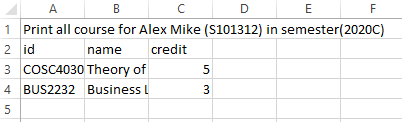
When he is in the Tester class he is now able to use commands which will be discussed later on. He has to get the App then executes the commands. After doing so, a result will display on the console.

In this case, the user used the command “printCourse()” which is one of the “save to file” commands. We will also see how result is stored externally.



Here is what the result file should look like. The result is saved to report.csv file

However, the user should **close the report.csv file before executes the program.**



To conclude, that is a quick overview of the system. To be more general, this is a student enrolments management system that allows users to interact between data and the system through the uses of commands. There are still many possibilities in the system but we just show only one for this example only, you can experiment more commands to see the output

2.1 Conventions.

There is another thing to keep in mind, commands should be used in meaningful ways. There are two main ways to achieve such task:

- Meaningful invocation : You can not modify an enrolment which technically doesn’t exist.( delete() and update())

- Valid input: Your input must be valid and in a correct pattern.

For example: invalid input student id: sid = helloworld

Should be: sid = S101312

Invalid pattern student id: sid = s101312

Should be: sid = S101312

So *inputs are case-sensitive*.

Besides the two rules here we would like to state out the conventions so that the user uses the system more careful and more safe.

+ **Built-in instances:**

When the program first loads, it will load the list of default instances which are Student, Course and Enrolment so that users can experiment without the need of adding instances.

*+***Relational approach:**

This system is built with the relational approach. Classes have to follow schemas, relation creation rules.

There are 3 *data classes* which are Student, Course, Enrolled designed with the relational approach.

*Here is the schemas and relation creations.*

Student:

\* <h1>Relation schema of student data</h1>

\*<code> Student(sid: CHAR(7),

\* sname: VARCHAR(30),

\* birthday: DATE); </code>

\* <h1>Creation of student data</h1>

\* <code> CREATE TABLE Student (

\* sid CHAR(7) PRIMARY KEY,

\* sname VARCHAR(30) NOT NULL,

\* birthday DATE NOT NULL

\* );</code>

In addition to the rules, in the Student class we also restrict user input to these convention:

Student id : id starts with uppercase "S" and follows by 6 digits.

Student name: name can contains up to 30 any characters.

Student birthday: birthday should be in MM/DD/YYYY format and it has to be valid.

Course:

\* <h1>Relation schema of course data</h1>

\*<code> Course(cid: VARCHAR(10),

\* cname: VARCHAR(50),

\* numOfCredits: BIT(10)); </code>

\* <h1>Creation of course data</h1>

\* <code> CREATE TABLE Course (

\* cid VARCHAR(10) PRIMARY KEY,

\* cname VARCHAR(50) NOT NULL,

\* numOfCredits BIT(10) NOT NULL

\* );</code>

Restrictions:

Course id : id can contains up to 10 any characters.

Course name: name can contains up to 50 any characters.

Course number of credit: A positive number smaller than 10

Enrolled:

\* <h1>Creation of Enrolled data</h1>

\* <code> CREATE TABLE Enrolled (

\* sid CHAR(7),

\* cid VARCHAR(10),

\* sem CHAR(5),

\* PRIMARY KEY(sid, cid),

\* FOREIGN KEY(sid) REFERENCES Student(sid),

\* FOREIGN KEY(cid) REFERENCES Course(cid)

\* );</code>

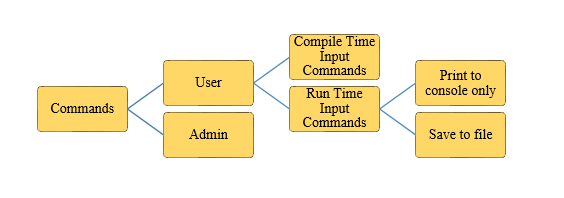
Restrictions:

Semester: A year follows by uppercase letter A,B or C.

Special rule: When a student enroll a course in a semester. Let “a” is the student birthyear, let “b” is the semester year. This expression must be satisfied:



2.2 Features list.

Once the user access the App, he is now allowed to used the commands. Commands are categorized by this hierachy.

**+User Commands are those which are used by the user and accessible to admin either.**

++Compile Time Input Commands are commands whose required inputs are yielded at compile time.:

Course.add(String id, String name,String numOfCredits): To add a new course.

Student.add(String id, String name,String birthday): To add a new student

++Run Time Input Commands are commands whose required inputs are yielded at run time.:

App a = App.getApp(); ---> A command to unlock the App commands

a.add() Requires (sid,sem,cid) To enroll a student to a course

a.delete() Requires (sid,cid) To remove an enrolment

a.update () Requires (sid,sem,option) To update an enrolment

+++ Print to console commands print results to the console and return a value:

StudentEnrolment a.getOne() Requires (sid, cid) To view a specific enrolment

+++ Save to file commands print results to the console, return a value and write to report.csv file:

ArrayList<Course> a.printCourse() Requires (sid, sem): Print all courses for 1 student in 1 semester.

ArrayList<Course> a.printOffered() Requires (sem): Prints all courses offered in 1 semester

ArrayList<Student> a.printStudent() Requires (cid, sem): Print all students of 1 course in 1 semester.

++View object information commands:

Let std is a Student instance

String std.getId(): To get the student id as a string

String std.getName(): To get the student name as a string

String std.getBirthday(): To get the student birthday as a string

String std.toString(): To get a string representation of the student instance

Let crs is a Course instance

String crs.getId(): To get the course id as a string

String crs.getName(): To get the course name as a string

byte crs.getNumOfCredits (): To get the course number of credits as a byte

String crs.toString(): To get a string representation of the course instance

Let enr is a StudentEnrolment instance

Student enr.getStd(): To get the student of the enrolment

String enr.getCrs():To get the course of the enrolment

String enr.getSem (): To get the enrolment semester as a string

String enr.toString(): To get a string representation of the enrolment instance

++Other commands:

ArrayList<StudentEnrolment> a.getAll() To get all records in the Enrolled List

**+Admin Commands are those which are used by the system and unuseable to the user.**

App.setPlaceholder():

Course.getList():

Student.getList():

StudentEnrolment.getList():

3.Functional Requirements:

- Commands have to excecute properly

- Admin commands and encapsulated properties can not be exposed by user

- The system should guarantee stability, speed and high accuracy.

- The system should acquire all user requirements such as CRUD functionalities, save to csv file.

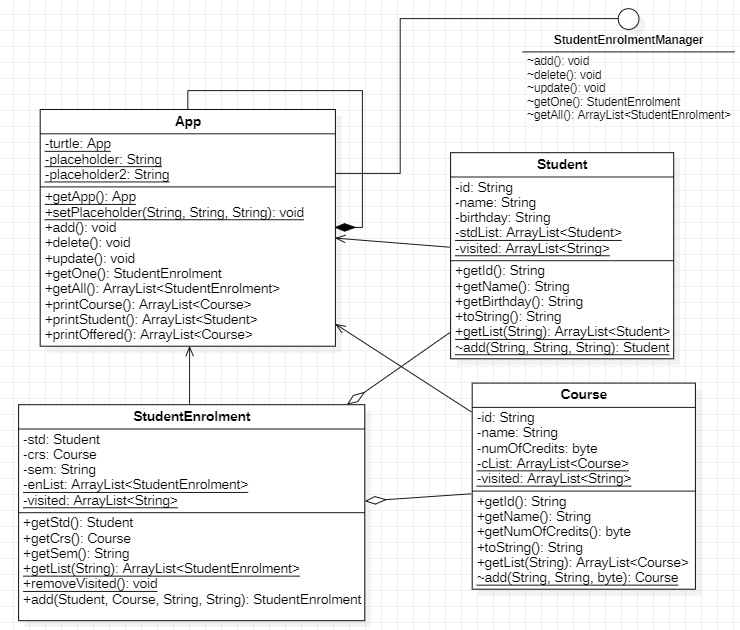
- The system should not expose data to intruder, all folders of files have to be kept internally

- The system should maintain security for admin, versatile commands so that user can not optionally use those commands.

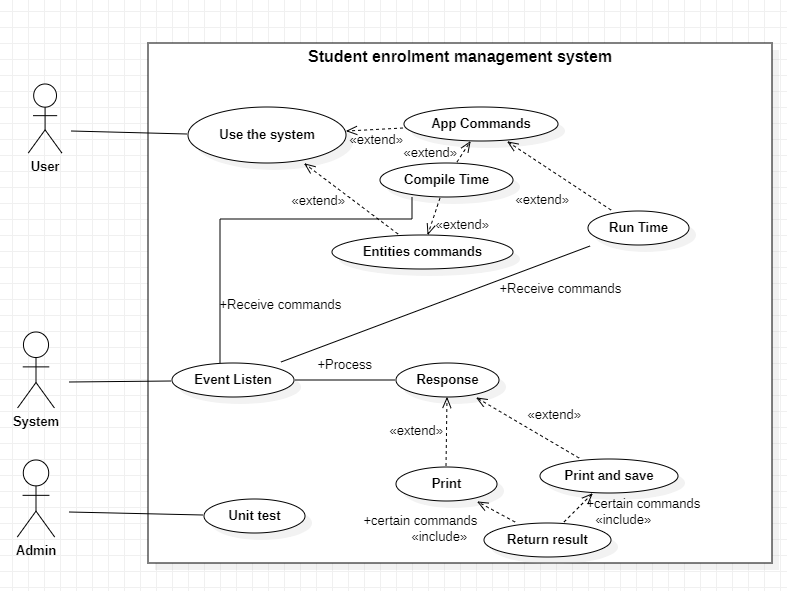
- Commands return a value if necessary so that the user can view more information about the object

- The system allows users to add more external students

3.1.Class diagrams



3.2. Use case diagrams



3.2. ER diagrams