Assignment A1-student management App

Student: Razvan Pasca

**Group:30432**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 4

3. System Architectural Design 6

4. UML Sequence Diagrams 8

5. Class Design 9

6. Data Model 10

7. System Testing 11

8. Bibliography 11

1. Requirements Analysis

# Assignment Specification

This application serves as a management tool for teachers in the school department. It comes in the form of a desktop app with a database connection and contains all the functionality needed to reduce the workload and improve the management of the courses and students. The system has different types of users: admins and regular users, each being able to perform different activities.

# Functional Requirements

The system presents different use cases, each coming with its own functional requirements. However, some functional requirements are shared between the use cases:

* Before proceeding with the operations, each user must be logged in
* The password must be longer than 6 characters
* The student can choose to move only in existing groups, attend only existing courses and take only the exams corresponding to the courses he enrolled for
* The student can see only his own grades but can see the name/courses attended etc for the other students.
* The admins of the courses are notified when a student wants to enroll and accept/decline their request

# Non-functional Requirements

For the proposed system, I believe there are several non-functional requirements which are critical to implement, given the nature of the information stored, namely:

* Data integrity: the data maintained by the system should be accurate and exclude any logical incompatibilities (e.g. a student having a grade without an exam recorded etc)
* Security: the system should be protected from foreign access from the outside and malicious users. The main breaches are SQL injection and denial of service procedures. Data validation is a must when trying to cope with such events.
* Reusability: the system should be reusable in other apps or even in the case of extending the current application to different platforms or a web service
* Scalability: in case the system is going to be open to a larger and larger number of users, it should be able to accommodate them. This would be mostly solved by increasing the hardware resources of the system (either horizontally or vertically)

2. Use-Case Model

Use case: Student view grades

Level: User goal level

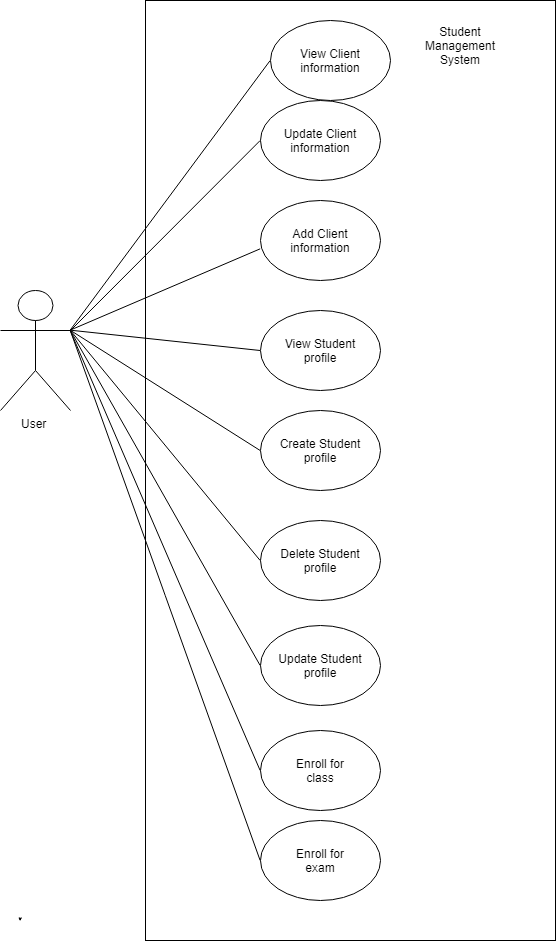
Primary actor: student

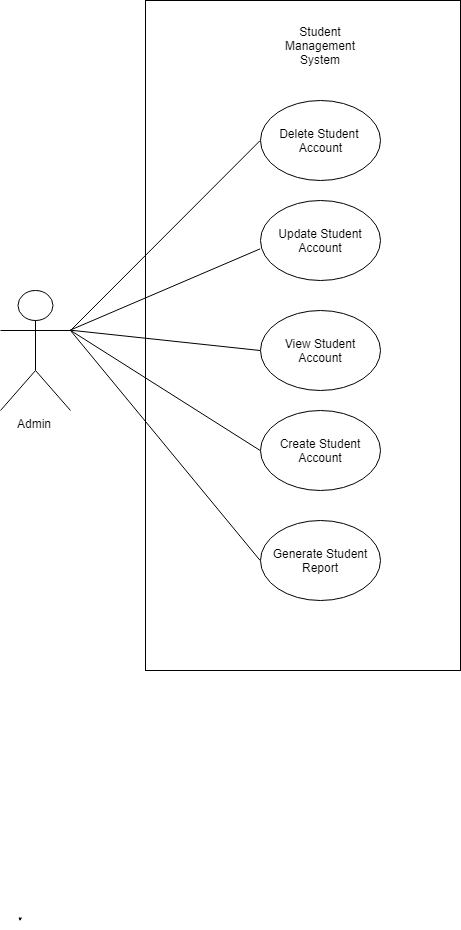
Main success scenario:

Pre-conditions: student is logged in

1. The user clicks “View student profile” button
2. A new window shows up with options for the student such as: view grades, view courses etc
3. The user clicks on “View grades” button
4. A list with the student’s grades shows up

Extensions: 4’. If the student has no grades yet, an appropriate message shows up





3. System Architectural Design

**3.1 Architectural Pattern Description**

For this application we are going to employ the layered architectural pattern, in its classical form, namely: Presentation, Business and Data layer. This approach was considered because it the app we are developing is a small one, and there is no need to develop into further components.

Moreover, the type of the application is the classical use of the layered pattern, since it implies:

* Connection with the database
* DAO components for the model classes
* Business logic to implement the functionalities
* Presentation layer for the GUI

Thus, we can see that it is easy to develop a logical separation between the layers based on their responsibilities and functionalities. This enables a lot of flexibility for the future, in case we opt to offer a web or mobile version of the system, since theoretically we would have to modify just the presentation layer.

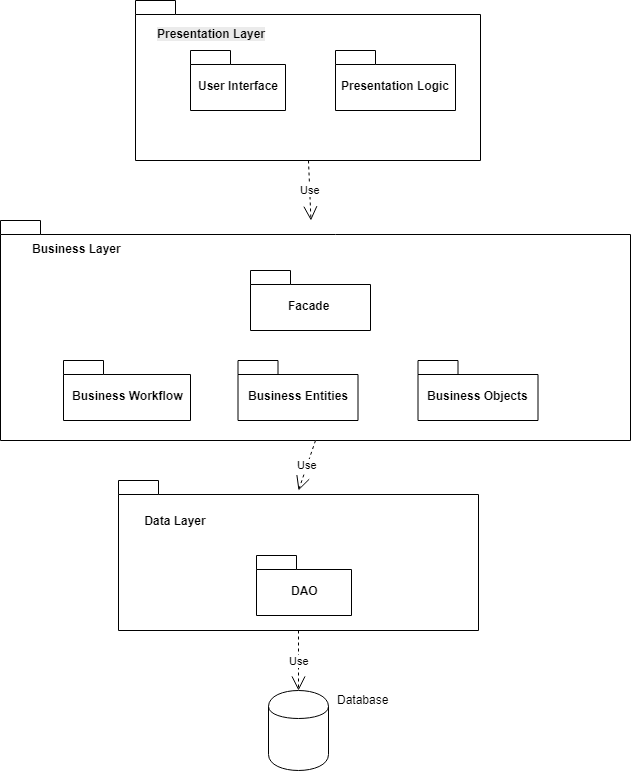
Another pattern which we are going to employ is the MVP pattern. This is a regular solution when developing user interfaces, since it splits the application logic into three separate parts. This increases the modularity of the system and gives us flexibility when it comes to extending the app and the user base. Moreover, we reduce the coupling between the model and the view, leaving the presenter to handle all the interactions and pass the messages between the layers.

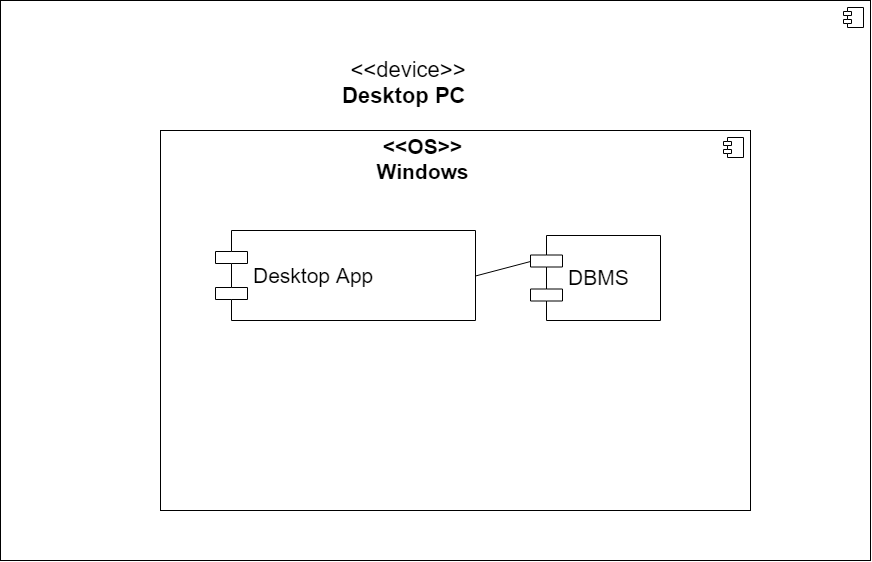
**3.2 Diagrams**

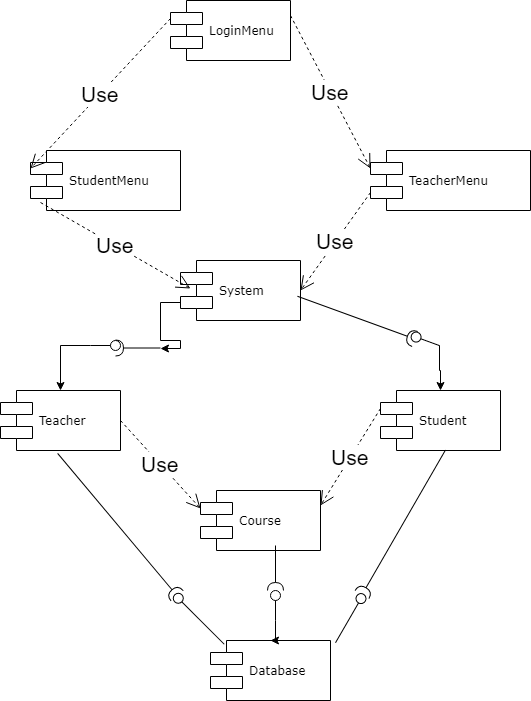
For the layered architecture, we are going to employ the classical package division of the components. Moreover, in the initial phase, the whole system is going to be placed on one component. At further iterations, we can employ a multi-tier architecture, where the database and the BLL are deployed on one component and the desktop app is delivered independently.

For the MVC, we can also employ different implementations of the view component, depending on the type of the host device: desktop or mobile. Talking about the dependencies, the controller is responsible for handling the interactions with the business layer through the façade provided.

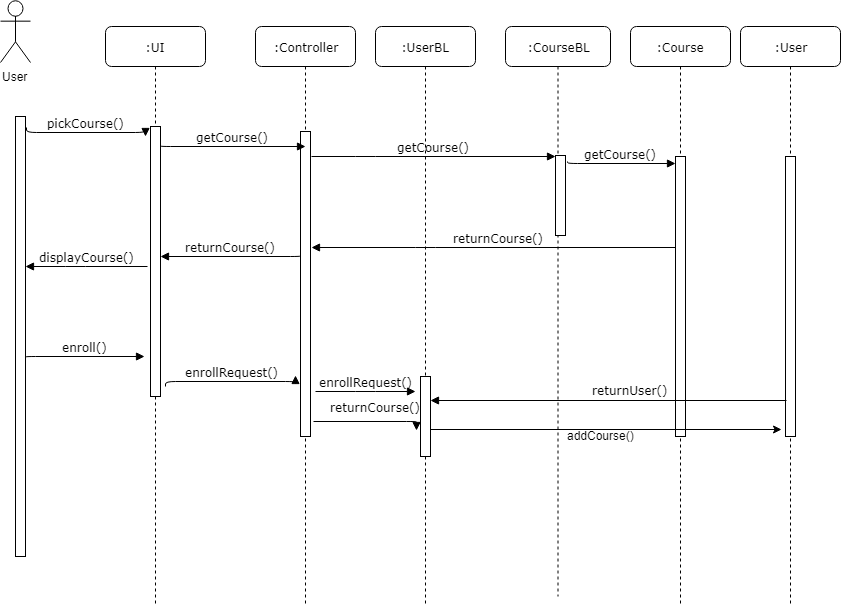
Below we have the following diagrams, in order: Package, Deployment and Component.







4. UML Sequence Diagrams



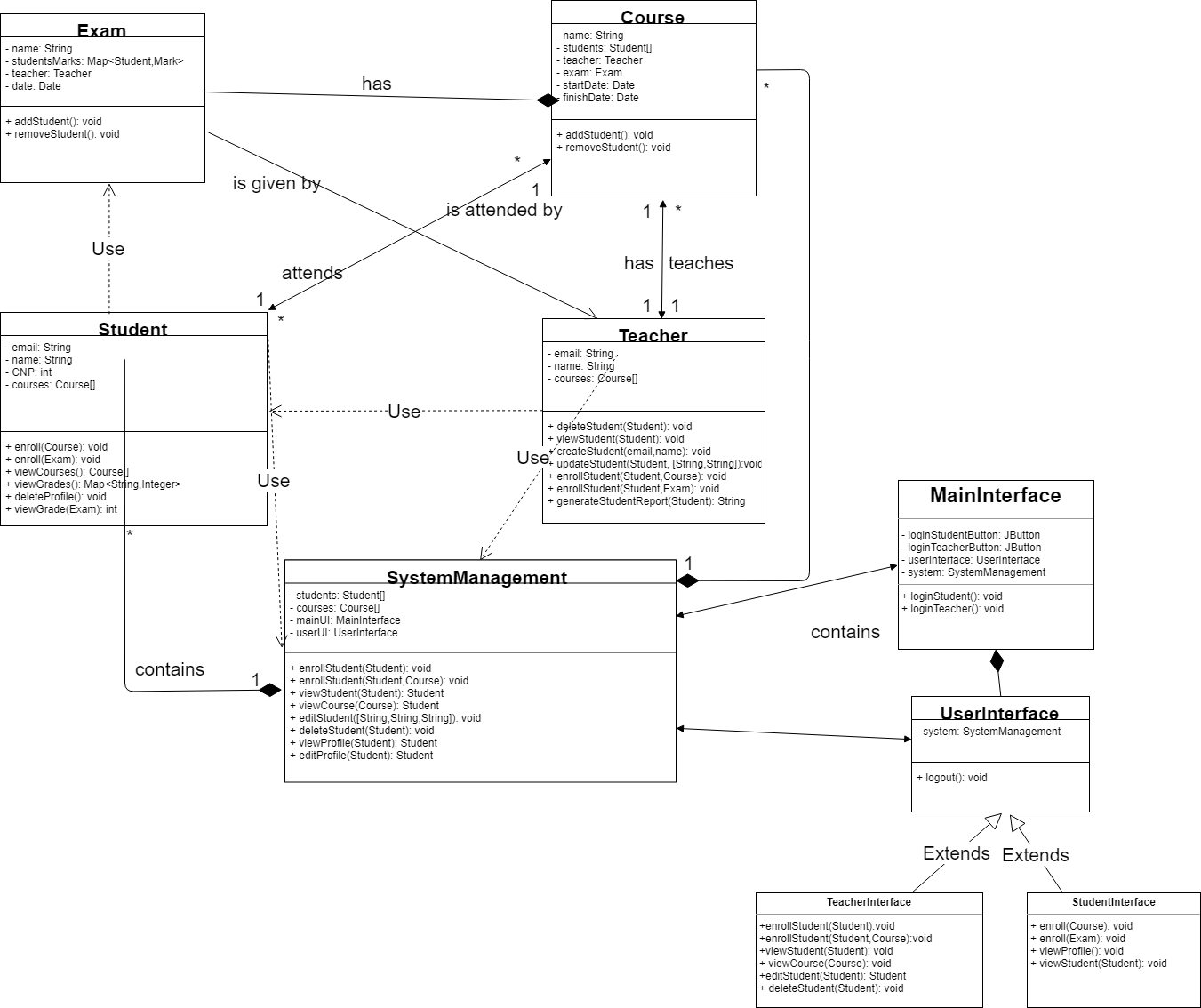
5. Class Design

**5.1 Design Patterns Description**

Façade Design Pattern: this pattern has been employed to simplify the interface provided by the business logic. In this way, the interaction with the system is simplified, and reduces the outside dependencies to the BLL.

Observer Design Pattern: this pattern has been employed to notify the admins about any changes in the courses they teach (e.g. a student wants to enroll). Also, the students are notified about changes in the courses they are enrolled in (e.g. an exam is announced or the date is changed).

**5.2 UML Class Diagram**



As it can be seen from the above, we employ the MVP pattern, where we have a clear separation between the model classes (Exam, Course, Student, Teacher), the Presenter(System Management) and the View classes. I have also opted for separating the interfaces of the Teacher and the Student, based on the login option they choose, applying the Interface Segregation Principle. Moreover, there is no coupling between the View and the Model classes, which increases the modularity of the system.

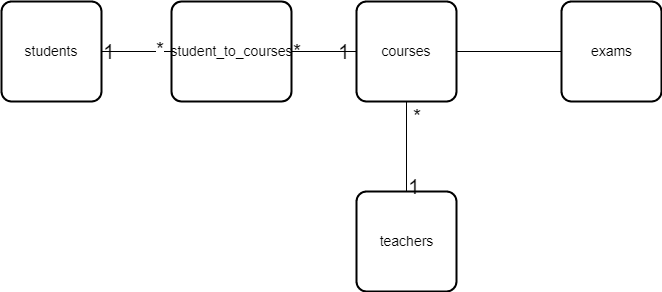
Also, the presenter handles all the interaction with the business layer. These functionalities are hidden behind a façade, which only provides the interface and the main operations of the system, as it can be seen from above. The smaller classes, which take care of implementing the specific logic are in the background.

Finally, we have the DAO classes which have not been represented on this diagram. They are responsible for low level interaction with the database and usually have a 1 to 1 mapping with the tables in the database.

6. Data Model

As said above, the data model classes are straightforward in this initial version of the app. Coming from the functional requirements, we have ended with a design where:

1. A student can enroll in several courses
2. A course can have several students participating
3. Each course has an associated exam (this is an assumption, there are no make-up sessions in this version of the faculty)
4. A teacher can teach more courses (usually 2, maximum 3, but no constraints are placed on this)
5. The link between the students and the exams is also done through the student\_to\_courses table, which holds data such as enroll date, mark obtained and so on. I believe this is the point where we can split the table in two, in case the student can participate in more exams for one course (in a further version).
6. Finally, the teacher doesn’t care that much about the exams, and have no direct access to them, only through the courses table.



7. System Testing

For testing the system, we are going to employ 2 strategies: Junit tests and Use case driven testing. We are going to use Junit for testing mostly at class/method level, by checking if the expected output matches the actual output given by the method.

Use case testing is going to be used to test the system thoroughly, covering the major use cases encountered by the users. This is going to help us discover integration defects which cannot be covered by Junit tests, since they cover just a simple module/component. This also helps us to diagnose the behavior on the alternative scenarios such as when entering wrong details or trying to damage the system.

8. Bibliography

1. https://en.wikipedia.org/wiki/Facade\_pattern#UML\_class\_and\_sequence\_diagram

2. <https://msdn.microsoft.com/en-us/library/ee658109.aspx>

3. <https://stackoverflow.com/questions/2056/what-are-mvp-and-mvc-and-what-is-the-difference>

4. https://en.wikipedia.org/wiki/Scalability