Course Management Application
Analysis and Design Document

Student: Varadi Robert
Group: 30432

Table of Contents

1. Requirements Analysis

1.1 Assignment Specification

Design and implement a Java application for the management of students in the CS Department at TUCN. The application should have two types of users (student and teacher/administrator user) which have to provide a username and password in order to use the application.

The regular user can perform the following operations:

- Add/update/view client information (name, identity, card number, personal numerical code, address, etc.).
- Create/update/delete/view student profile (account information: identification number, group, enrollments, grades).
- Process class enrolment (enroll, exams, grades).

The administrator user can perform the following operations:

- CRUD on student information.
- Generate reports for a particular period containing the activities performed by a student.

1.2 Functional Requirements

The functional requirements for the system are the following:

- For each operation, the user should be logged in with an existing account, except for the case of the account creation operation, when the user creates the account
- If a student wants to enroll to a course, he/she sends a request about his/her intention which the teacher will be notified about and if the teacher approves the student, the student will be enrolled to that course and implicitly to the exam associated to that course.
- A teacher will be able to give grades to a student.
- A teacher, once logged in, can assign a group to a student who has not been assigned yet to a group.

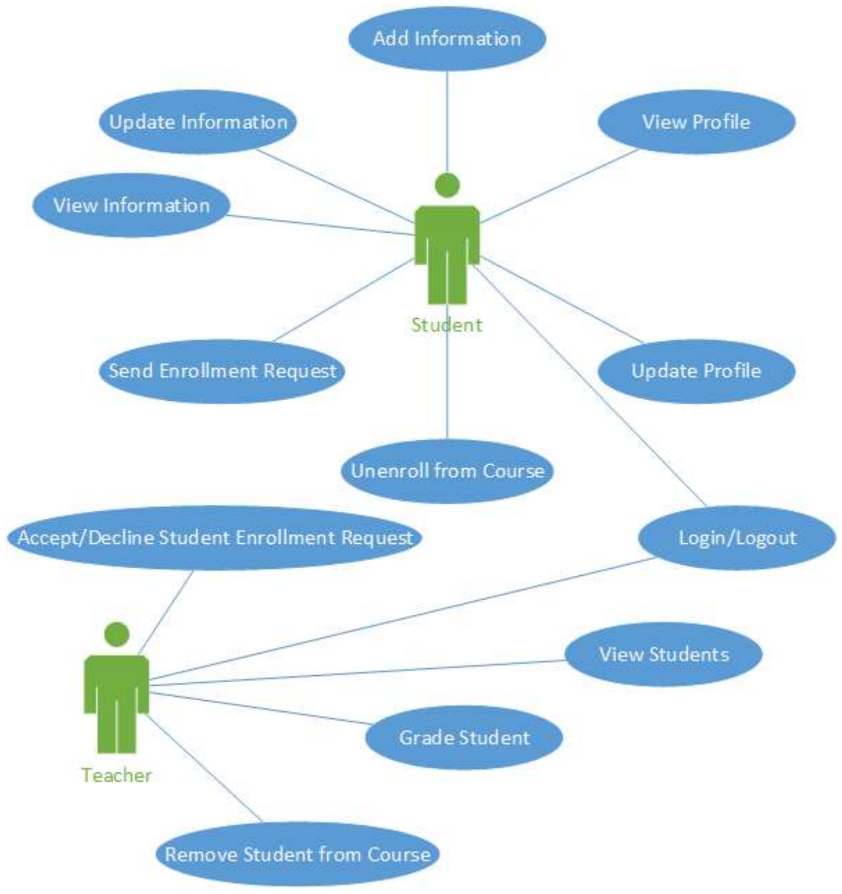
1.3 Non-functional Requirements

The non-functional requirements for the system that we will take into account are the following ones:

- Availability: The system must be available 100% of the time, except the periods of planned maintenance, when the user cannot access the system.
- Performance: The system must do each operation in less than 1s 95% of the time and under 2s 5% of the time.
- Reusability: The components of the system must be developed in such a way that they can be reused by other systems if necessary.
- Testability: The system's components must be written in such a way that they will be easy to test.
- Usability: The system must be easy to use, so that even people who are not very skilled with computers should be able to use it.

2. Use-Case Model

From the problem specification we deduce the existence of the following actors in the system: a Student actor and a Teacher actor. The Student actor is able to create an account, add information, update information, view information, create a profile (account), update the profile, view the profile and delete profile (a profile means the grades of a student, the courses he or she is enrolled in as well as the group to which the student belongs), and send an enrollment request for a particular course. A Teacher is able to login or log out from the system, accept the enrollment request of a student or decline that request, view all the students enrolled to his/her course, grade a student and finally, remove a student from his/her course. The use-case diagram for the given system is the following one:



Use case: *Enroll*
Level: *user-goal*
Primary actor: *Student*
Main success scenario:

- *Login*
- *Select a course*
- *Click the "Enroll" button*
- *Send the enrollment request*

Extensions:

- *The student failed to log in*
- *The student has already been enrolled*

3. System Architectural Design

3.1 Architectural Pattern Description

For the given system, we are going to use the Layered architectural pattern. From the point of view of the Layered architectural pattern, the application will be split into the following components:

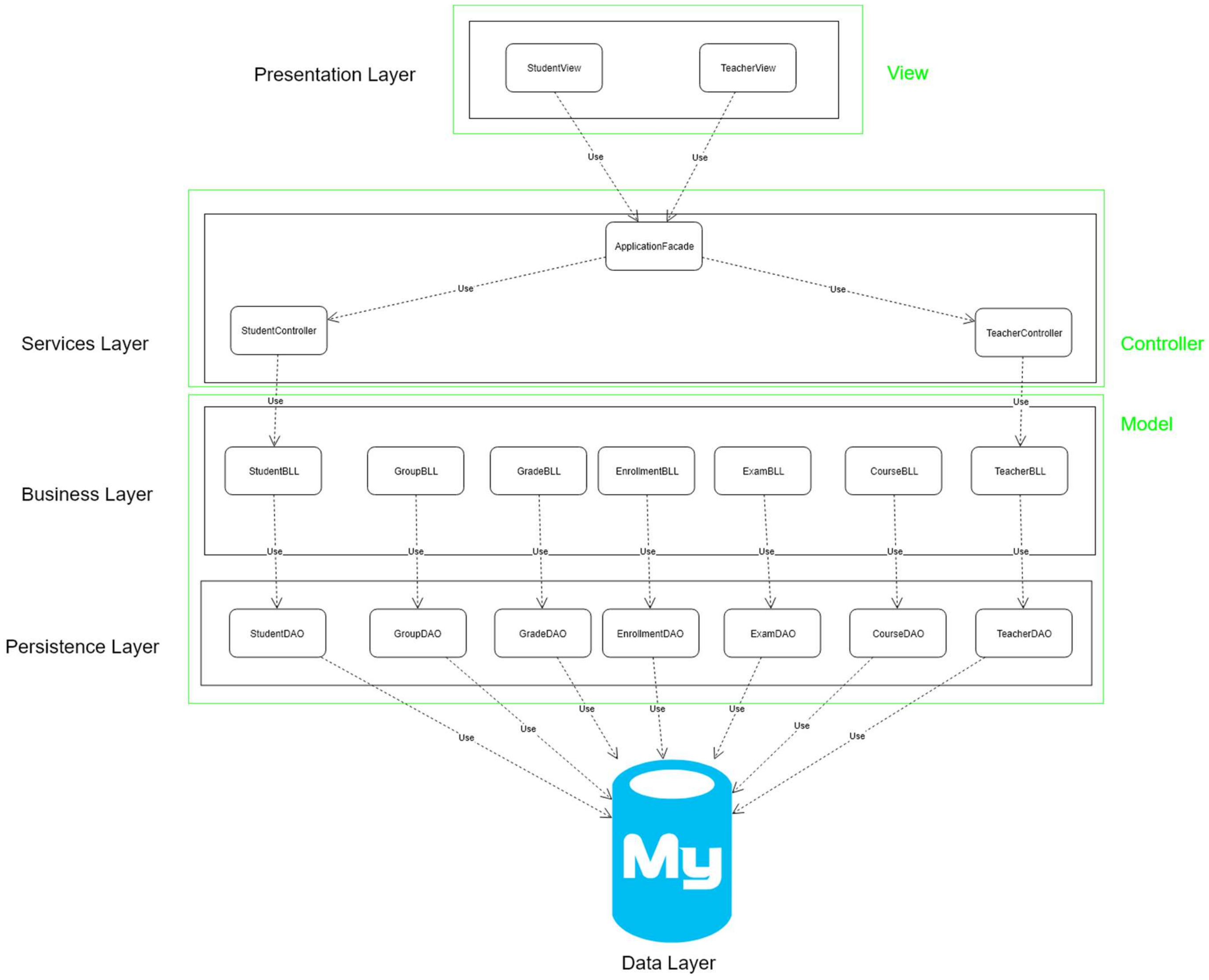
- The database layer, where we have all the data of the system stored;
- The persistence layer, responsible for connecting to the database and interchange objects among the database and the application;
- The business layer, where the higher-level functionalities are implemented which will be then used by the user. In this layer, we use one or more functionalities from the persistence layer in order to achieve

a task;

- The service layer, which is responsible for taking data from the view and sending it to the application as well as for sending data to be displayed from the application to the view.

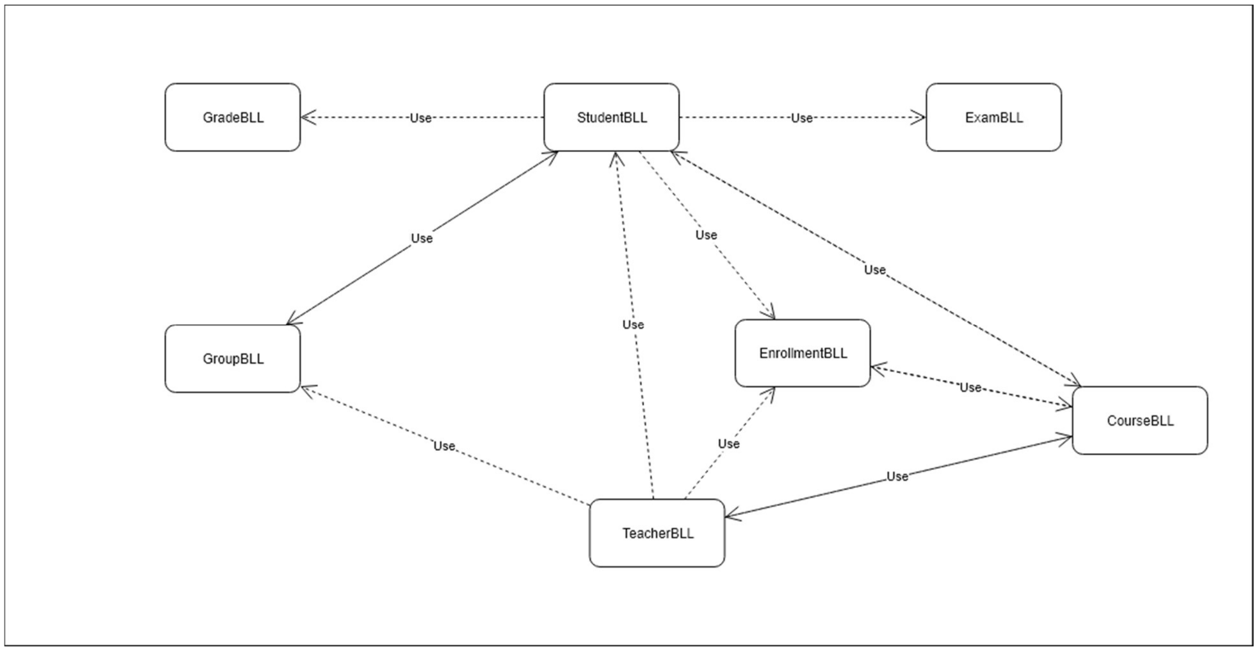
Since the application must also have a graphical user interface, a good approach is to use also the MVC architectural patterns which consists of splitting the application into 3 components, the Model where the domain model data and the operations on those data are defined, the View which is the component of the application which directly interacts with the user and finally the Controller component, responsible for connecting the View of the application to the Model so that each time the user interacts with the View, the Model will be updated and vice versa, any change in the Model will be propagated to the View.

3.2 Diagrams



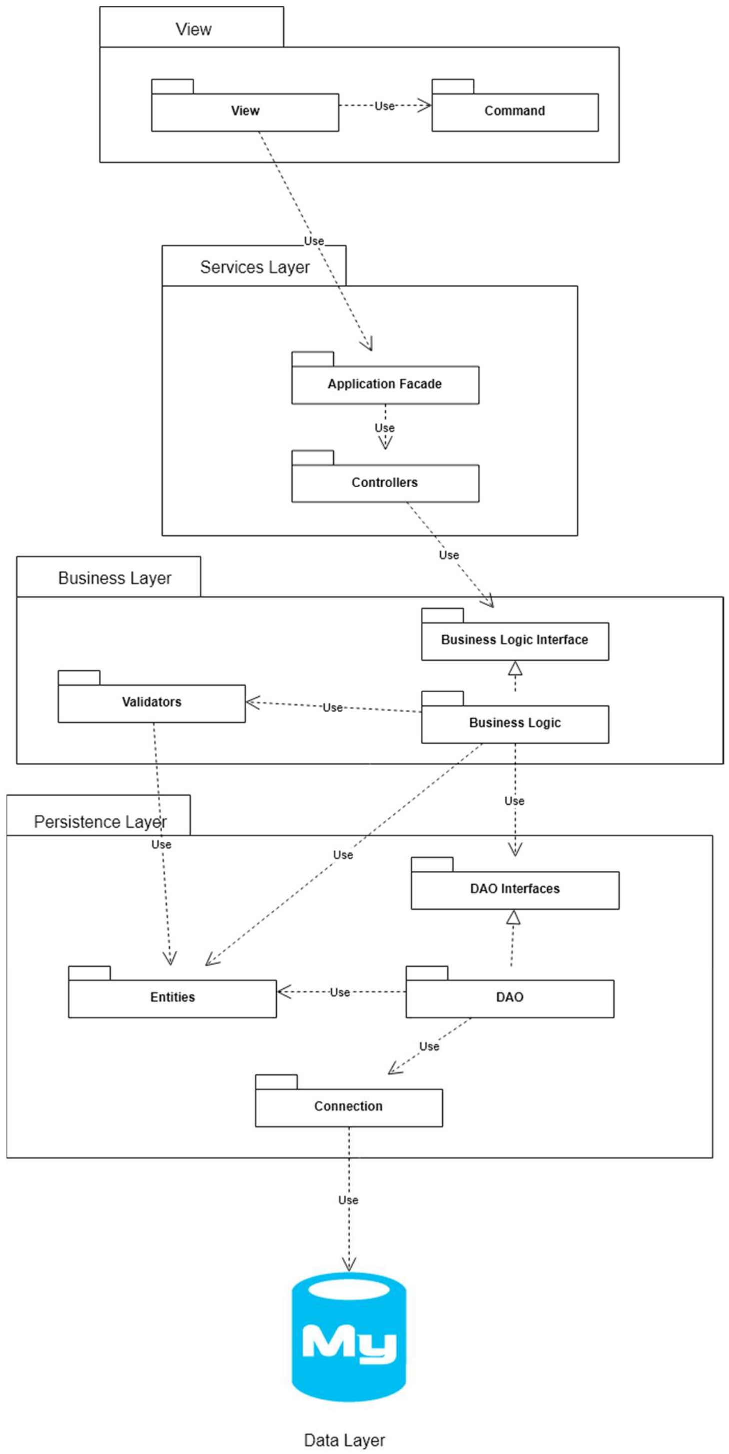
Each layer will be assigned a package and may contain more sub-packages. The first layer is the persistence layer, containing 3 sub-packages: the package containing the domain model classes (Teacher, Student, Course, etc), the connection package which is responsible only for establishing a connection with the database server, so the application will be able to exchange information with the data source, and finally the data access layer, which is responsible for sending queries to the database, once a connection to the database has been established.

The Business layer is responsible for combining the functionalities of the persistence layer in order to generate a response for the user request which comes from the higher-level layers. This package will contain two sub-packages, namely the Validators package which is responsible for validating the data coming from the user interface, the Business Logic classes, responsible for combining the functionality of the persistence layer in order to achieve some piece of functionality. The connections between the components of the Business Layer are not shown in the above diagram because of the complexity of the relations. Here is a more detailed picture about how these classes connect to each other.

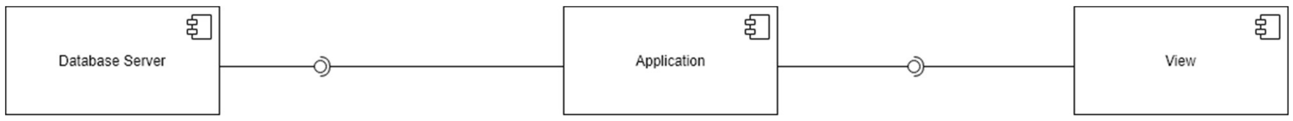


The next layer is the Services Layer which contains a Façade class, encapsulating the whole functionality of the system. This layer also contains a set of controllers which are responsible for interpreting the commands coming from the view and sending them to the lower level, the business logic layer. The last sub-package contains a set of interfaces which the controller classes will use and the business logic layer classes will implement. This way we commit to an interface, not to an implementation and the controller classes will not have to figure out how to call the methods provided by the lower-level classes, instead the lower-level classes will have to adapt to the needs of the controller classes.

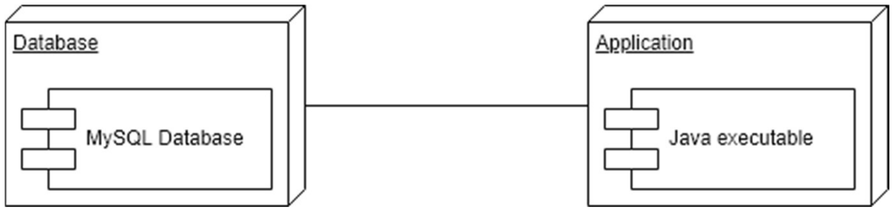
Finally, the presentation layer contains two sub-packages, the View, where the different views of the system are defined and the Command sub-package where the commands which may be sent to the application will be defined, one command per use-case.



From the perspective of the component diagram, the system can be split into three components, the database server, which provides an interface for the application (SQL language queries), the application which provides an interface for the GUI of the application (the one involving the command design pattern), and finally the view (GUI) of the application.

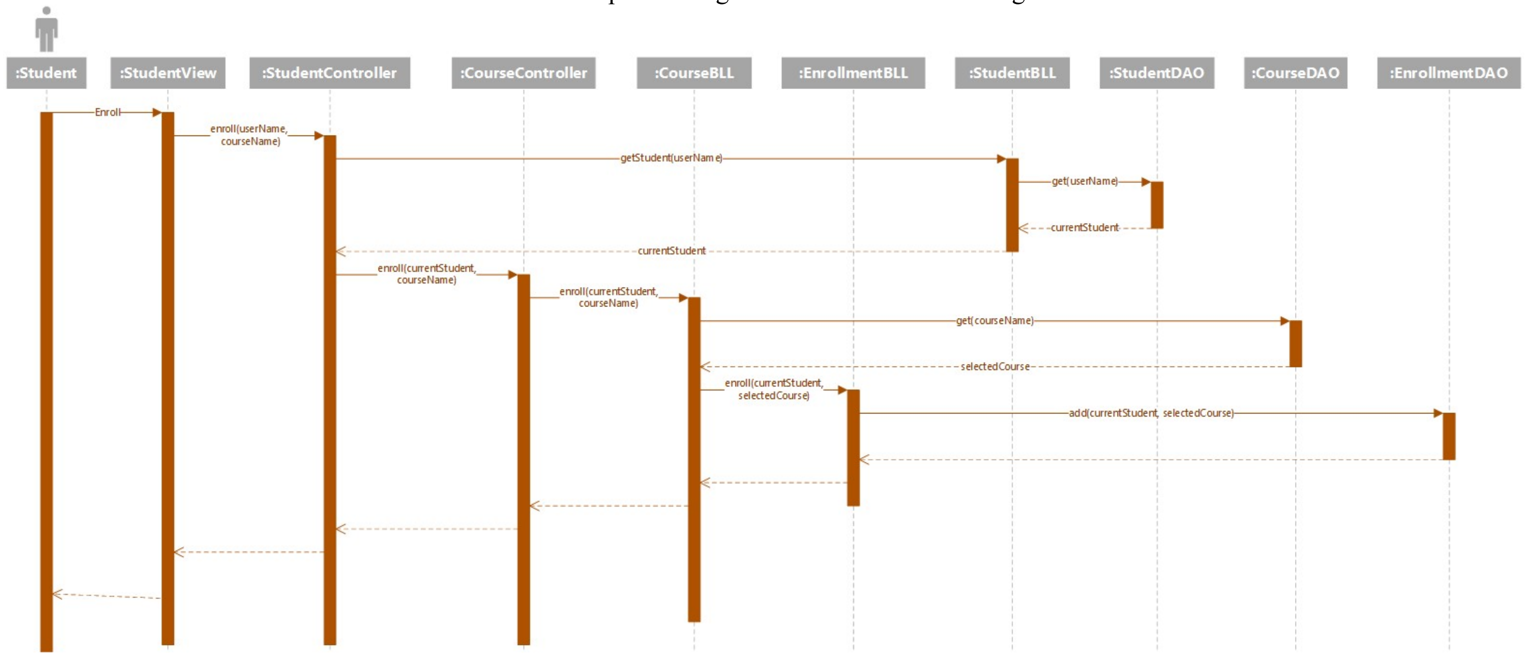


The deployment diagram for this system is a rather simple one, which consists only of two nodes, namely the database and the application itself which uses the database server. From this perspective the architecture could be interpreted as a 2-tier architecture which may be represented in the following way:



4. UML Sequence Diagrams

An example of relevant a scenario from the perspective of our system would be the enrollment of a student to a course. For this purpose, the student, once logged in, selects a course and presses the "Enroll" button, after which the enrollment request will be stored by the system. Then, whenever a teacher logs into the application, he/she will see all the students who wish to enroll to the corresponding course and the teacher will be able to either accept or decline the enrollment of the student. The sequence diagram would be the following one:



5. Class Design

5.1 Design Patterns Description

One design pattern which we may use is the Factory design pattern applied for the connections to the database. There will be a ConnectionFactory object responsible for creating connections to the database each time we want to do an operation. Also, since such a connection factory should be only one per application, we can use the Singleton pattern as well to limit the number of ConnectionFactory objects to at most 1.

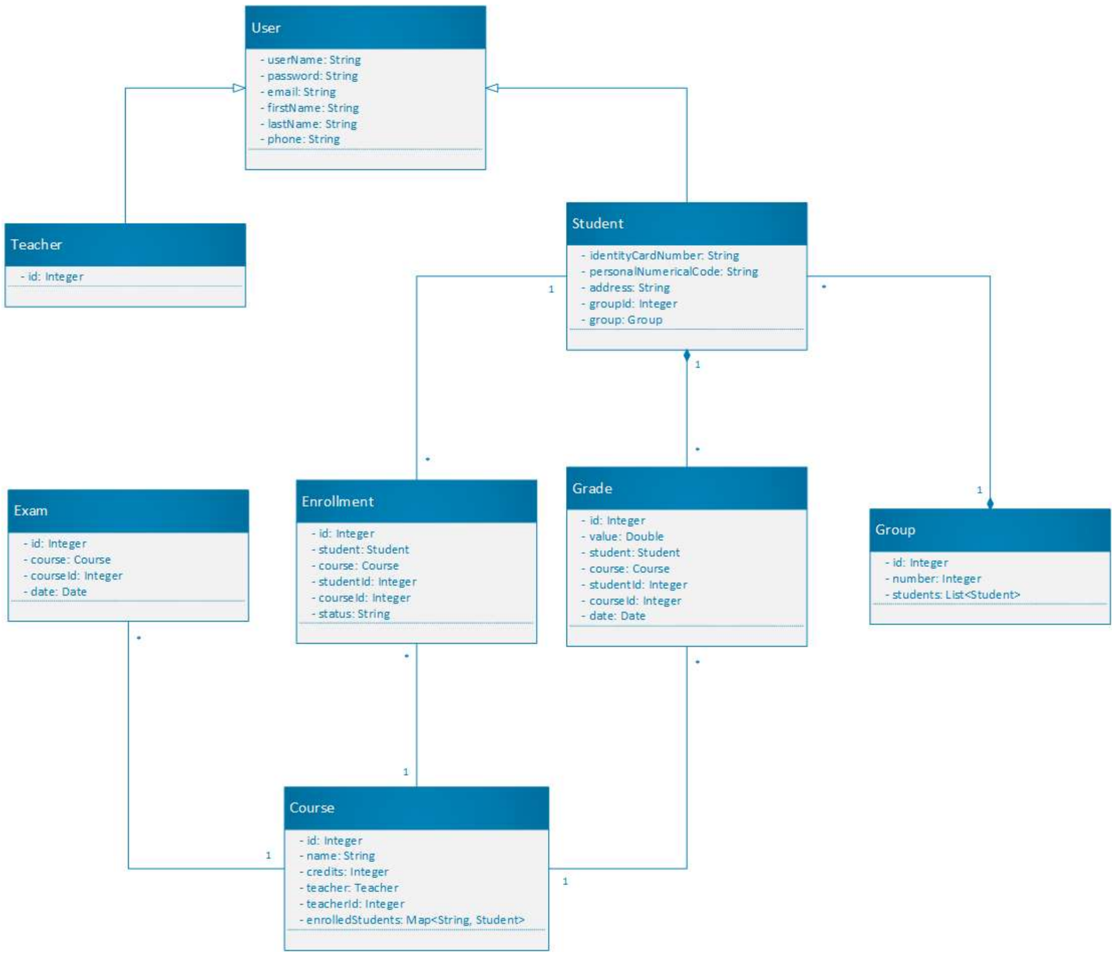
Another design pattern which may be used is the Command design pattern in which the UI will build a Command object for each user operation and send it to the application which will decide, through polymorphism, which is the command to be executed.

5.2 UML Class Diagram

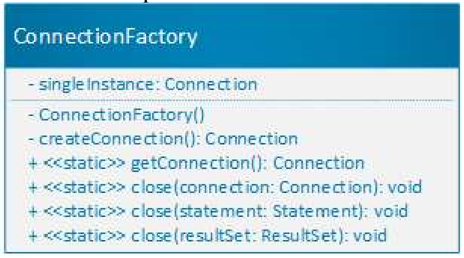
The domain model for the system will consist of the following classes: User, Student, Teacher, Course, Enrollment, Post, Exam, Grade and Group. The User class will contain information regarding the account of each user, which are common for both the Student and the Teacher entities, like username, password, email address, first name, last name and phone number. The Student class will contain all the latter attributes and additional ones, like identity card number, personal numerical code, address and grades (no grade may exist without belonging to a student). The teacher will contain all the fields related to the User class and the additional taught courses field, which represents all the courses the teacher teaches. Both the Student and the Teacher class will extend the User class.

The Course class will contain the subject name, credits and a collection of enrolled students. The Enrollment class will contain the student who has enrolled, the course to which he/she has enrolled, the enrollment date and also the enrollment status (request, accepted, declined, deleted).

The next class from the domain model is the Post class and it will be used by the teacher to post messages related to the Course he/she teaches and will contain the Course, the Teacher, the subject of the post, the content and the post date. The Exam class will contain the Course to which it is associated and the date of the exam. The Grade will have a value, a weight, the date when it has been assigned, the subject at which the grade was given and the type of the grade (exam, lab, assignment, bonus). Finally, the Group class will contain a number and a HashMap of students for the case when we wish to retrieve the group of a given student. The class diagram of the domain model is the following one:

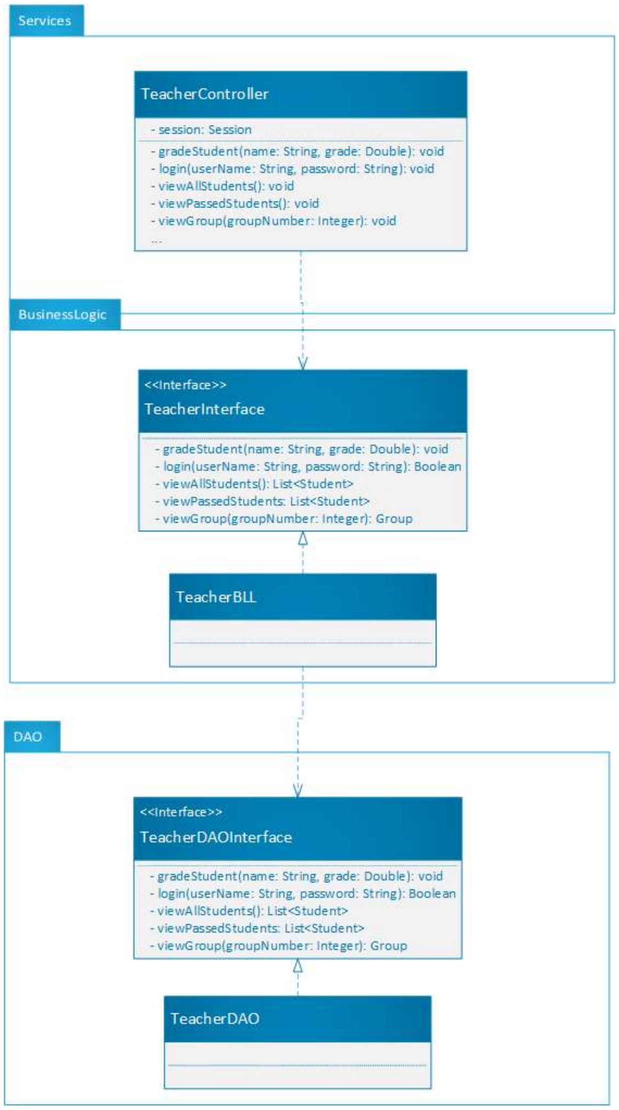


The Factory and Singleton patterns will be implemented as shown in the following class diagram:

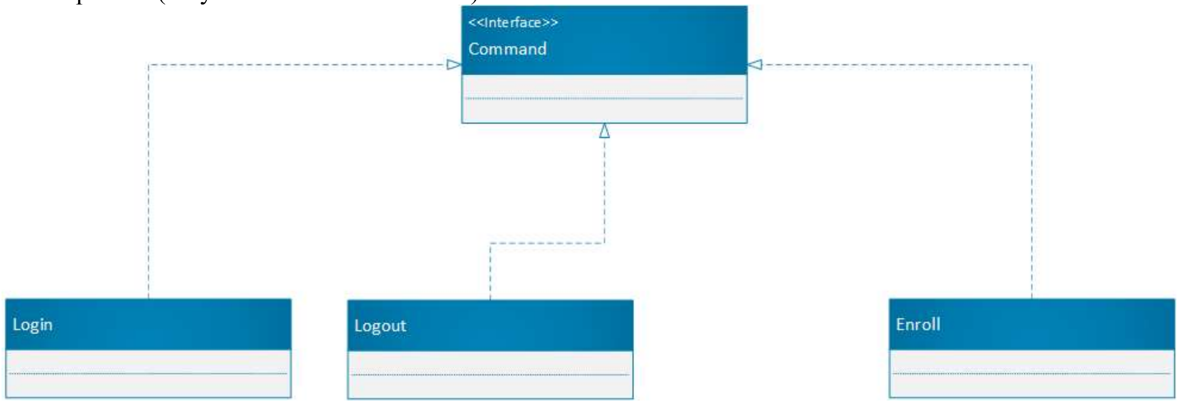

ConnectionFactory
- singleInstance: Connection
- ConnectionFactory()
- createConnection(): Connection
+ <<static>> getConnection(): Connection
+ <<static>> close(connection: Connection): void
+ <<static>> close(statement: Statement): void
+ <<static>> close(resultSet: ResultSet): void

The constructor for the ConnectionFactory object will be private, which will ensure that only one such object will be instantiated, namely the singleInstance object. The factory then will be able to create connections and close connections as well as result sets and statements, functionalities which are implemented as static methods.

The following step in to provide an interface for a layer, at the layer immediately lower, so that the lower level layer can be replaced without affecting the application, the only condition is that the new component must implement the interface. The diagram showing this approach is the following one (only the one for the Teacher entity is shown, the Student follows the same pattern and the other classes do not have controllers, but the interfaces for the DAO still exist):


Services
TeacherController
- session: Session
- gradeStudent(name: String, grade: Double): void
- login(userName: String, password: String): void
- viewAllStudents(): void
- viewPassedStudents(): void
- viewGroup(groupNumber: Integer): void
...
BusinessLogic
<<Interface>>
TeacherInterface
- gradeStudent(name: String, grade: Double): void
- login(userName: String, password: String): Boolean
- viewAllStudents(): List<Student>
- viewPassedStudents: List<Student>
- viewGroup(groupNumber: Integer): Group
TeacherBLL
DAO
<<Interface>>
TeacherDAOInterface
- gradeStudent(name: String, grade: Double): void
- login(userName: String, password: String): Boolean
- viewAllStudents(): List<Student>
- viewPassedStudents: List<Student>
- viewGroup(groupNumber: Integer): Group
TeacherDAO

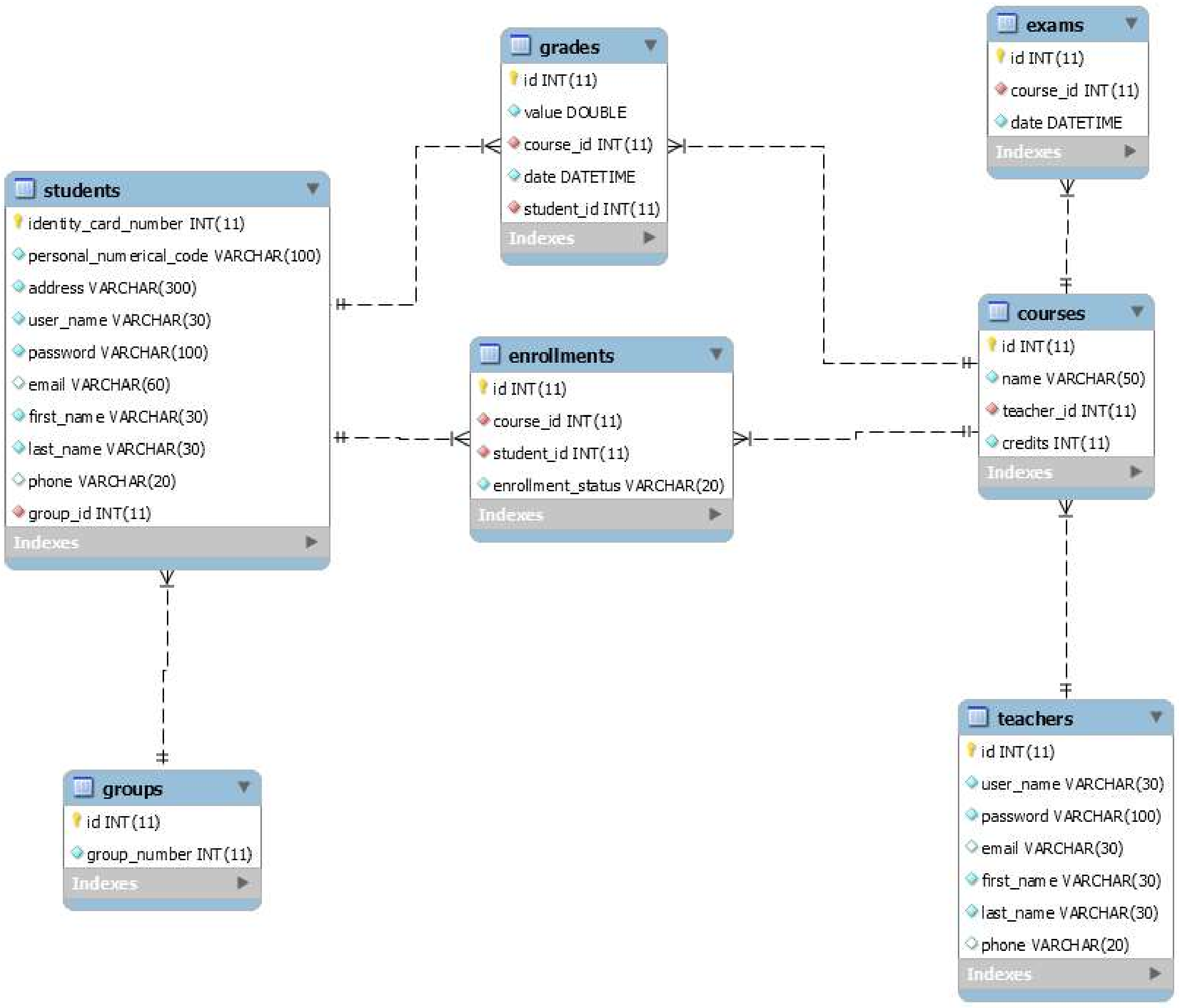
We will also use the Command design pattern to send commands from the view to the application. In this approach, the application will contain a hierarchy of commands, one for each user action, represented as a class hierarchy and each time a user performs an operation in the view, the corresponding command object will be sent to the controllers, which will figure out, through polymorphism, which command has to be executed. The class diagram for the pattern (only 3 commands are shown):



We will use the Façade design pattern in order to encapsulate the functionality of the system into a single class such that the view does not have to know how the methods of which class it should call, this way we achieve a low coupling between the front-end and the backend. Also, we are going to use Observer pattern in order for the view to notify the application Façade each time an action has been taken, by sending the proper command, which will be executed by the application. After executing the command, a response will be sent back to the view, based on which, a view factory will instantiate the new view the user needs.

6. Data Model

For the data model of the application, we start from the first class-diagram and deduce the presence of the following database entities: student, teacher, grade, course, group, exam, enrollment. The teacher and course are in one-to-many relationship. The enrollment table could be an intermediary table between the student and course entities. The entity relational diagram will be the following one:



7. System Testing

In order to test the application, we will use unit testing which consists of writing the methods to be tested into a JUnit file, in which each method is a test case and can be used to test some functionalities of the system. The test consists of providing some input data to the method, perform some operations and compare the result with the result expected by us, through an assertion. If the assertion fails, the unit test failed, so there could be a problem with our implementation.

The testing method will be testing based on partitioning in which we try to split input domain into disjoint subsets, whose union gives the initial input domain (partitions). This partition has the property that for all its elements, the method will behave in the same way (for example if, they are all invalid input data, the method should throw an exception). For testing the whole input domain, we could select want representative from each partition and run the test for that particular value. Together with this testing method, we should employ also the boundary analysis which means to test not only for the values from the particular partitions, but also for the boundaries between two partitions. This way we might detect inconsistencies in the program. For example, let us suppose that the input domain is the set of real numbers and the program should behave differently for positive real numbers and for negative real numbers. In this case we take a negative number, a positive number and test the output of the method, but it would be also useful to test the output for the boundary, that is, zero in this case, because the method might not behave as expected for boundary values.

8. Bibliography

- https://www.safaribooksonline.com/library/view/software-architecture-patterns/9781491971437/ch01.html
- https://en.wikipedia.org/wiki/Command_pattern