Bring ideas to life
VIA University College

PROJECT REPORT

Students

Flemming Vindelev	251398
Ionel-Cristinel Putinica	266123
Mihail Rumenov Kanchev	266106

Supervisors

Ib Havn
Joseph Chukwudi Okika
Knud Erik Rasmussen
Mona Wendel Andersen

Date

08-06-2018

Table of Contents

Table of Figures

Abstract

Education plays an important role in the human development, thus, this multi-user system has been fully designed in order to facilitate a better and more efficient communication between parents, teachers and students leading to a higher quality educational experience. This project report has the aim to describe all the methods and stages of the implementation of this system in the Java & SQL programming languages. All the features have been designed with the end-users in mind, giving teachers the possibility to manage students, parents, grades and attendance, and allowing parents and students to view the grades and the attendance inside the application. The application also gives the teachers and parents the opportunity to communicate with each other, via a message feature. The entire system is maintained by an admin, which has the possibility to create, manage, and delete teachers from the database and has been designed so it is easy to expand, maintain and to update, offering all the users a straight-forward experience.

Introduction

In human development, education plays a very important role. This, in turn, is the duty of a teacher, in any educational institute – to shape pupils' behavior and attitude from a young age while they are still easily influenceable. The way that these individuals are modelled as children is the way they will grow up as adults, given that they will naturally associate parents and teachers as role models. Parents, being the most present people around youth, have a very strong capacity for influence, as children, as well as teenagers, seem to observe and imitate the behavior of people around them. If repeated enough, different behavior and attitude 'can become character for them'.

A consequence can be drawn from the statements above – parents and teachers alike make the greatest difference in early human development, probably until puberty. For this reason alone, these two parts need to communicate and report any problems, in order to be able to fix flaws or create improvement in children's behavior.

This project would like to facilitate parent-teacher communication in an educational context, with the goal of fixing and preventing behavioral issues, but also encouraging children to find their character and personality. Contact between these two parts is already possible through phone calls or text messages, using mobile applications, such as WhatsApp, to keep in touch. These have proven fast and efficient over the years, as they can boost the trust between parent and teacher, as well as offer the parents a sense of involvement.

These methods appear successful, although it would seem as though they are not always convenient and sometimes do not provide a detailed-enough overview of the pupil's behavior, grades or problems. Hence the reason why this project wants to offer parents a closer, personalized, more intimate look into their child's day to day life at school by giving access to a full analysis of grades, complaints, class updates, attendance, teacher's comments, and the possibility to schedule a 1-on-1 meeting. Such digital systems exist already around Europe – the Danish school portals or Romanian online gradebooks – yet, they do not offer nearly the same number of features as the project introduced here.

Therefore, a larger-scale solution, including more detailed and personalized experiences for both teachers and parents, has the potential of boosting interpersonal trust and helping pupils grow into better and more successful individuals, through communication.

To see a more in-depth introduction to the project, see appendix 8.

Analysis

Requirements

In the requirements section of the paper, it is described what *had* to be done by the deadline and what *could* be done if time allowed it. It also includes some requirements about *how* it should work or some technical requirements so to say. This is split into two categories, Functional Requirements and Non-Functional Requirements. In the functional section the requirements are categorized by priority.

Functional Requirements
Top Priority
1. Admin must be able to create, delete and modify Teachers. 2. Admin must have only one account in the Database. 3. Admin account will be hardcoded within the System. 4. Every Class must have only one Teacher as a Headmaster. 5. Teacher must be able to add Students to his/her class. 6. Teacher must be able to create and assign Parents to Students. 7. Teacher must be able to change grades and attendance of every Student. 8. Teacher must be able to modify name of every Parent/Student. 9. Parent must be able to view grades and attendance of his/her own Student. 10. Student must be able to view his/her own grades.
Non-Priority
1. Teacher must be able to respond to a Parent's message or contact a Parent. 2. Parent must be able to send messages to Teacher. 3. A single Parent must be able to be assigned to multiple Students.

Non-Functional Requirements

1. System must be maintainable and easy to update.
2. System must store data into a Database.
3. System will use Client-Server connection of type RMI.
4. System will use CPR number as a login.
5. System will use connection of type LAN.
6. System will be coded in Java and SQL.
7. System will not include a live-chat feature.

Use Case Model

The requirements of the Class Book System have been summed up and made into a use case model. This model explains the interaction between all types of users and the system in an easy to understand diagram. To get a better view of the use case model, see appendix 1.

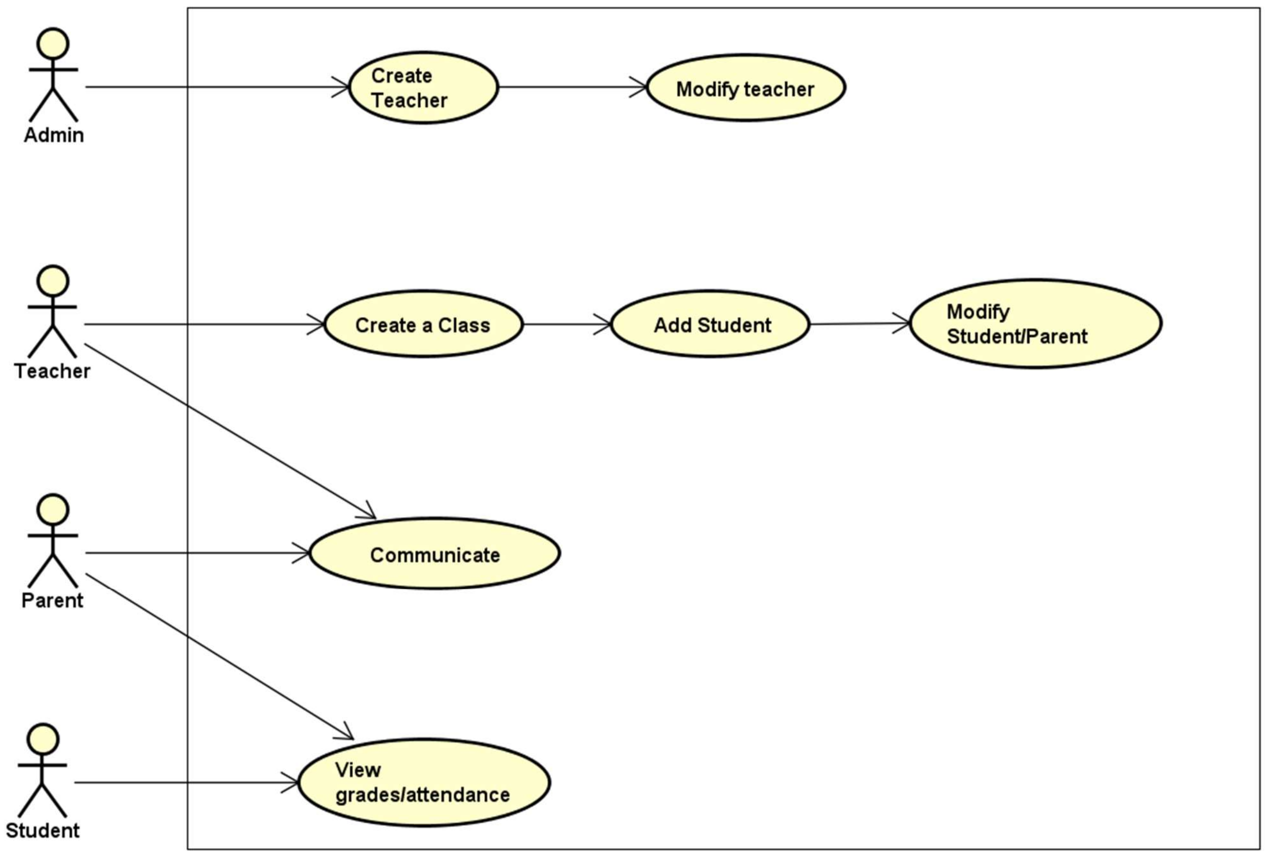


Figure 1. Use case model

Use Cases

Below is a list of different use case descriptions, which explains in depth how each function from the use case diagram works. This can help the user to use the system, similar to guidelines or user manual.

Use Case	Create a Class
Actor	Teacher
Pre-Condition	The server must have an existing teacher stored in its data.
Post-Condition	The data is stored in the server
Base sequence	1)Teacher logs-in 2)Teacher writes the name of the class 3)Teacher clicks the „Create Class” button
Exception sequence	If data is invalid: 1-3

Use Case	Create Teacher
Actor	Admin
Pre-Condition	The server must be running
Post-Condition	The data is stored in the server
Base sequence	1)Admin clicks the „Create teacher” button 2)Admin inputs the name of the teacher 3)Admin inputs the CPR of the teacher 4)Admin clicks the „Create” button 5)The new teacher is added to the teacher list
Exception sequence	If data is invalid: 1-5

Use Case	Modify Teacher
Actor	Admin
Pre-Condition	The server must be running
Post-Condition	The data is stored in the server
Base sequence	1)Admin selects a teacher from the teacher list 2Admin clicks the „Modify teacher" button 3)Admin inputs the new name of the teacher 4)Admin clicks the „Create" button 5)The changes to the teacher are saved, data is updated all around the system.
Exception sequence	If data is invalid: 1-5

Use Case	Add student
Actor	Teacher
Pre-Condition	There must be an existing teacher and class to add a student
Post-Condition	The data is stored in the server
Base sequence	1)Teacher clicks the „Add Student" button 2)Teacher inputs the name and CPR of the student and the parent as well as the grades and attendence of the student 3)Teacher clicks the „Create" button 4)The new student is added to the teacher's class
Exception sequence	If data is invalid: 1-4

Use Case	Modify student
Actor	Teacher
Pre-Condition	There must be an existing student in the class to modify it
Post-Condition	The data is stored in the server
Base sequence	1)Teacher selects a student from the class 2)Teacher clicks the „Modify Student” button 3)Teacher inputs the new name of the student and the parent as well as the new grades and attendence of the student 4)Teacher clicks the „Save” button 5)The new data of the student is saved
Exception sequence	If data is invalid: 1-5

Domain Model

In the following diagram, the domain model is shown. This model is like a map over the different non-tech specific objects, which will be instantiated in the system. The model shows the relation between the different objects, since some of them uses information from each other. This model is the foundation of what the system should be able to do, based on the requirements from the customer. To get a better view of the domain model, see appendix 2.

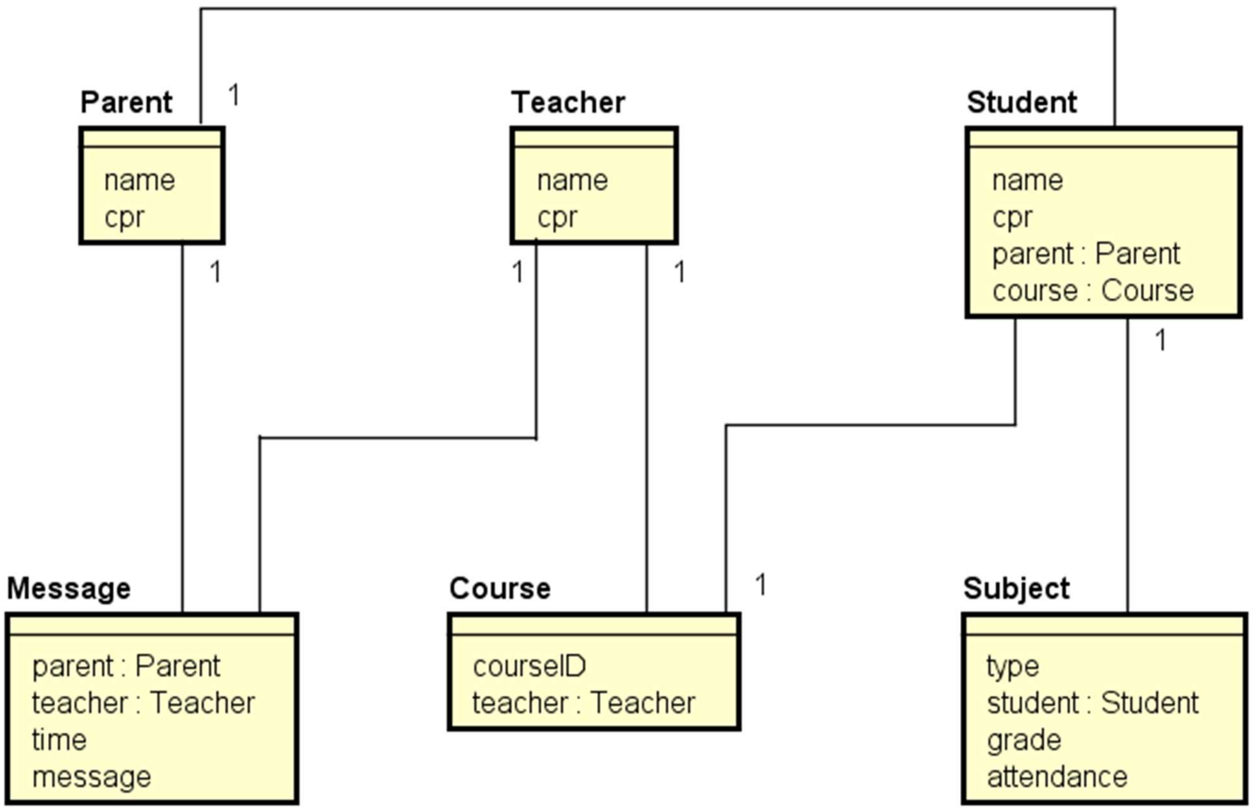


Figure 2. Domain model

Design

UML Diagram

Client Class Diagram

The following figure is the Client Class Diagram, which has been shaped up in order to give a better understanding of the Client side of the system. In the Client Class Diagram, it can be examined how the responsibilities in the Client have been split using the Model-View-Controller pattern. The model contains data and provides operations to manipulate data, the view shows the model's state and gets input from the user and the controller takes care of the data flow into the model object and updates the view whenever data changes. For a better look at the diagram, see appendix 3.

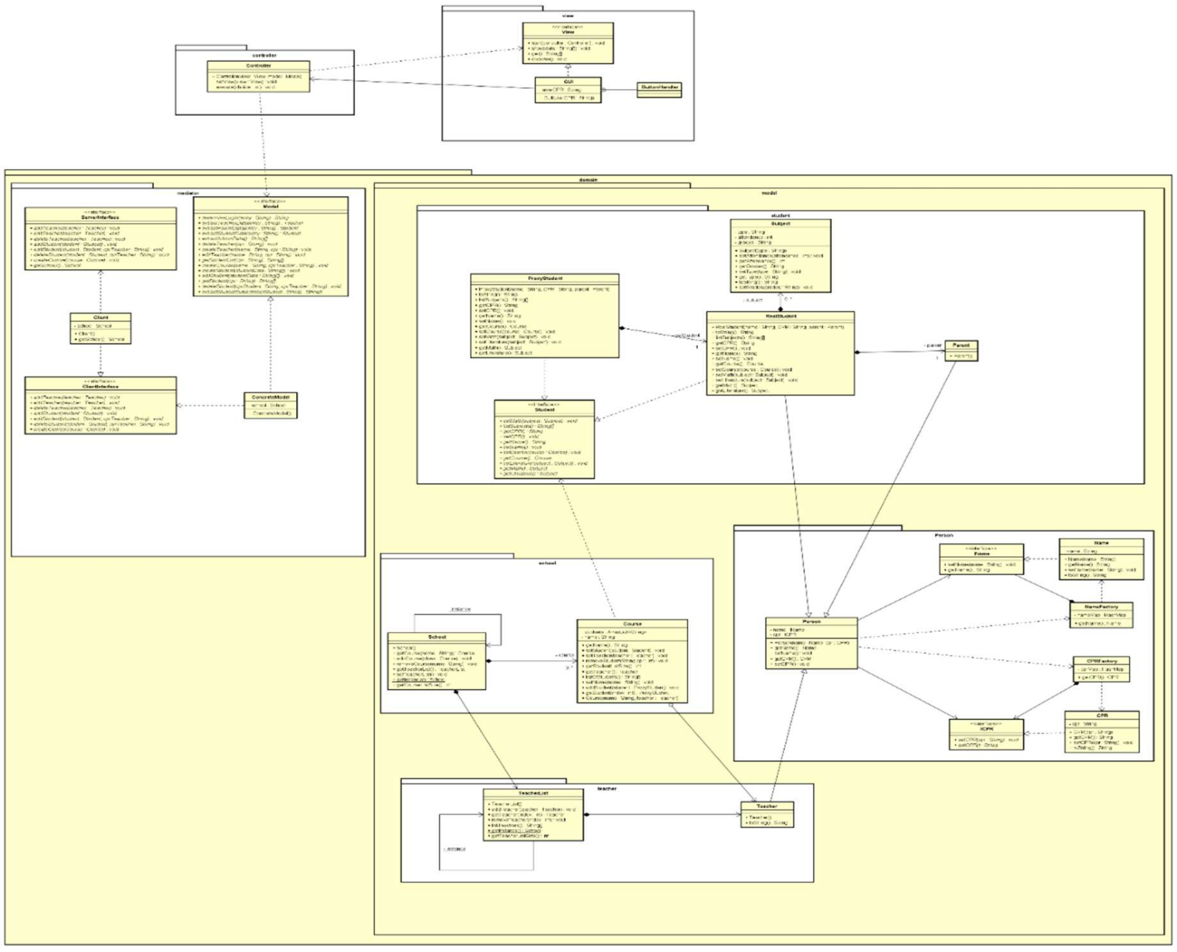


Figure 3. Client UML diagram

Server Class Diagram

The following figure represent the Server Class Diagram, its purpose being a better comprehension of the Server side of the system. In the Server Class Diagram, an accurate representation of the connection between the server and the database using the Adapter pattern can be found. Another reason for which the Server Class Diagram exists is to give a better overview on how data and objects are handled in the system with the help of the Proxy, Singleton and Flyweight patterns. For a better look at the diagram, see appendix 3.

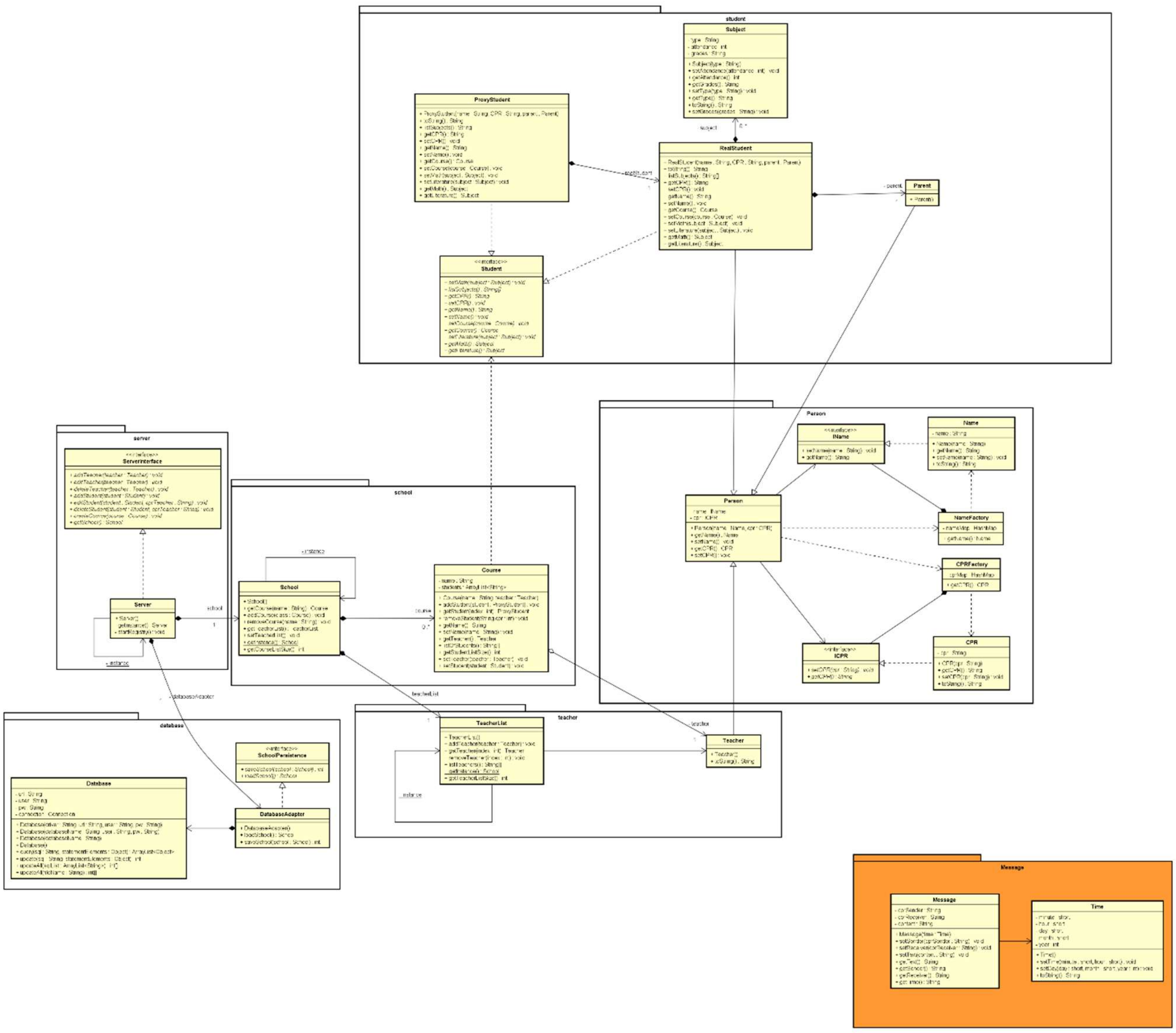


Figure 4. Server UML diagram

Sequence Diagram

Modify Student sequence diagram is a diagram that represents the flow of operations in a method, and the order in which they take place.

Firstly, the Teacher triggers the modifyStudent method, which is called upon the system, that displays back to the teacher a list of attributes. After that, the teacher, via the inputAttributes method, fills up the data, which is then sent back to the system, which uses saveStudent to store all the information, and then the student list is going to be updated. To see all the sequence diagrams, see appendix 4.

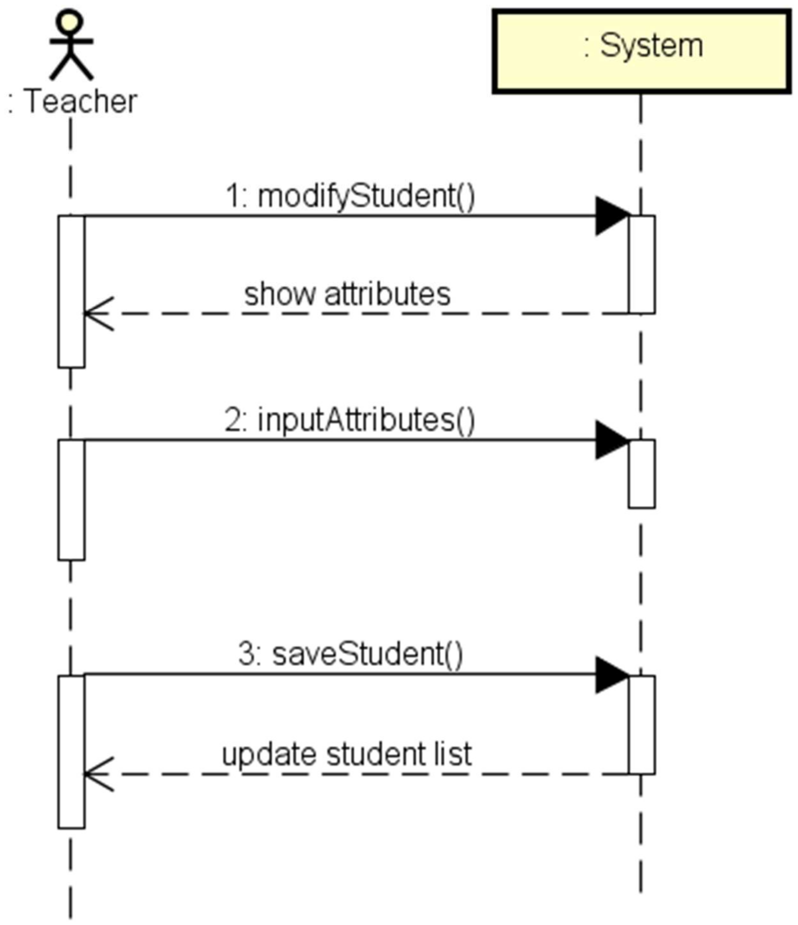


Figure 5. Modify student sequence diagram

Activity Diagram

Modify Student/Parent activity diagram shows the process handled when the teacher-user modifies the data for a student and for a parent. The purpose of this diagram is to guide the user through the steps behind actually modifying the data for the student and for the parent.

The first step is for the teacher to log-in inside the system with his/hers associated CPR number. The next step after that is for the teacher to select one of the students from the "List of students" by clicking on the desired one. Afterwards, the teacher has to input all the new practical information related to the student, and either cancel the process or save the new data. To see all the activity diagrams, see appendix 5.

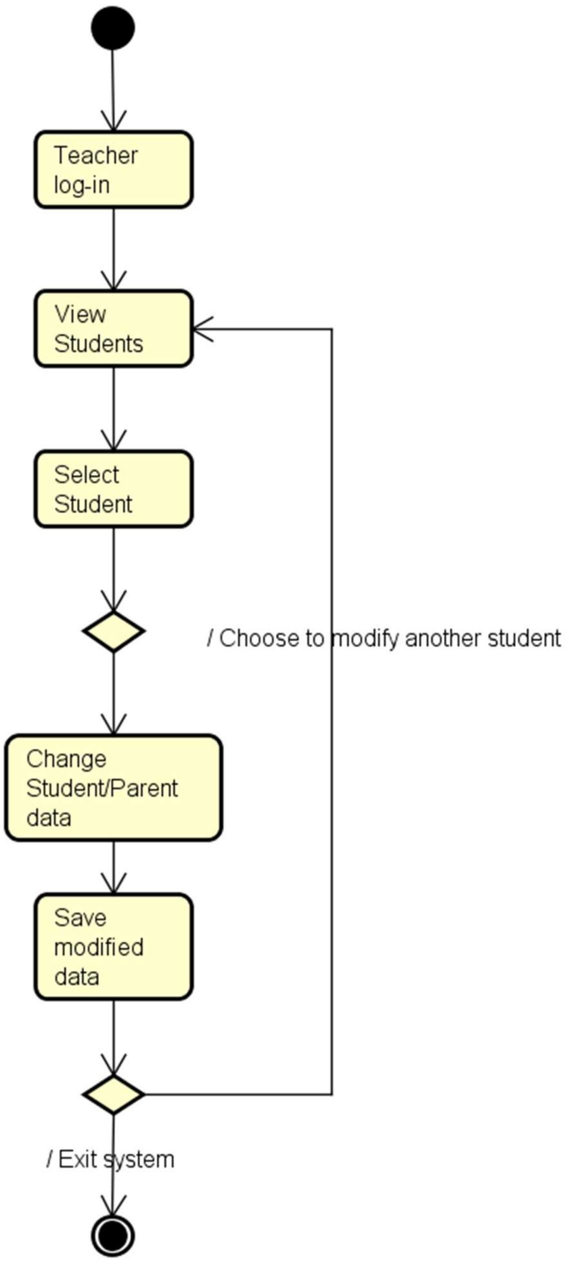


Figure 6. Modify student/parent activity diagram

Design Patterns

Singleton

The Singleton pattern is a software design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects. The term comes from the mathematical concept of a singleton.

Adapter

The Adapter pattern is a software design pattern (also known as Wrapper, an alternative naming shared with the Decorator pattern) that allows the interface of an existing class to be used as another interface. It is often used to make existing classes work with others without modifying their source code.

Flyweight

A flyweight is an object that minimizes memory usage by sharing as much data as possible with other similar objects; it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory. Often some parts of the object state can be shared, and it is common practice to hold them in external data structures and pass them to the objects temporarily when they are used.

Proxy

A proxy, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate. In short, a proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes. Use of the proxy can simply be forwarding to the real object, or can provide additional logic. In the proxy, extra functionality can be provided, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked. For the client, usage of a proxy object is similar to using the real object, because both implement the same interface.

RMI

Java Remote Method Invocation (Java RMI) is a Java API that performs remote method invocation, the object-oriented equivalent of remote procedure calls (RPC), with support for direct transfer of serialized Java classes and distributed garbage-collection.

The original implementation depends on Java Virtual Machine (JVM) class-representation mechanisms and it thus only supports making calls from one JVM to another.

Model-View-Controller

Model–view–controller is commonly used for developing software that divides an application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. The MVC design pattern decouples these major components allowing for efficient code reuse and parallel development.

Observer

The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

EER Diagram

The diagram shown below is a map over how the database stores data. More specific it shows the different entities which is created in the system, and how they are stored. It also shows how the entities are related to each other, which helps making the database more effective. To have a better look at the EER diagram, see appendix 6.

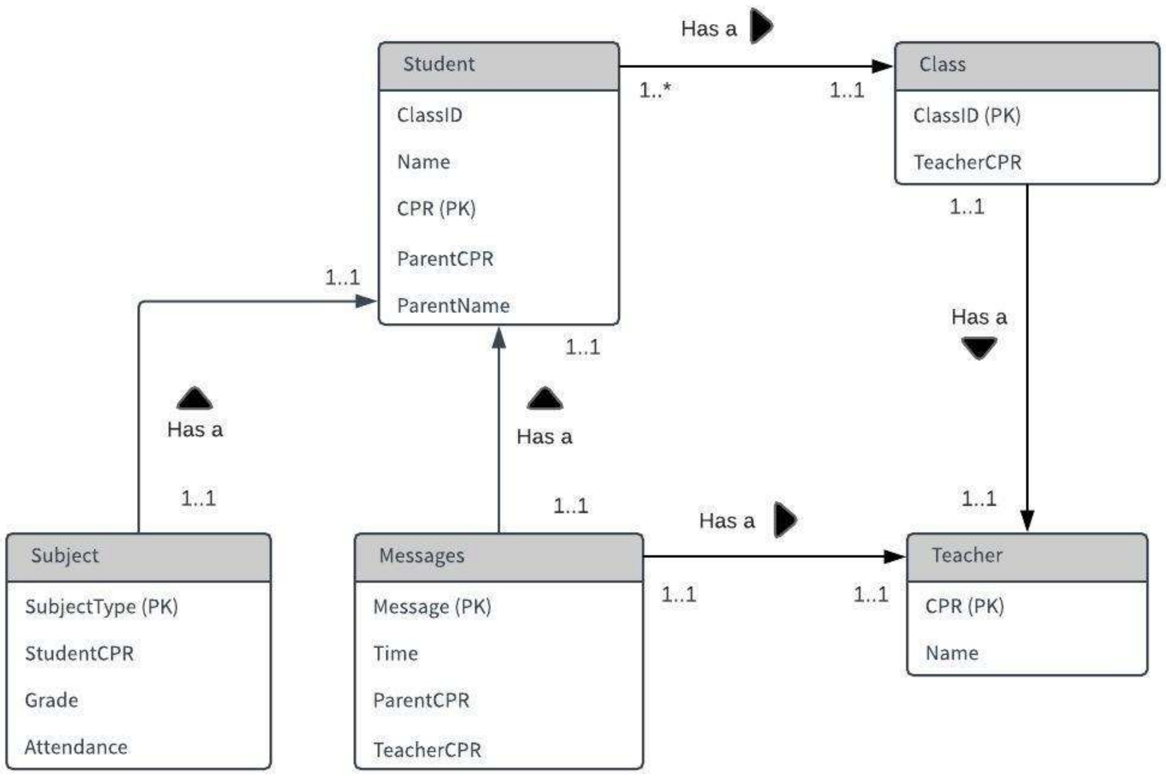


Figure 7. Database EER diagram

UI Design Choices

To see a more in-depth guide on the system, see appendix 7.

The first time any user runs the application a login window will pop-up, in which he will have to input the associated login data.

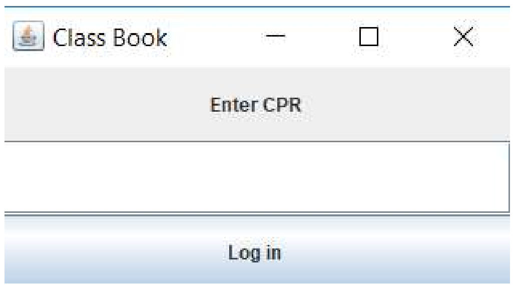


Figure 8. GUI log-in

Depending on the data the user inputs, he will be sent to a new page, where he will have access to his features. To access the admin features, a passcode has to be typed, rather than a CPR code. Once the passcode is entered and validated, the admin features will be unlocked and a window will appear. This window contains a "List of teachers" and 3 buttons "Create teacher", "View teacher" and "Delete teacher" that control different functions.

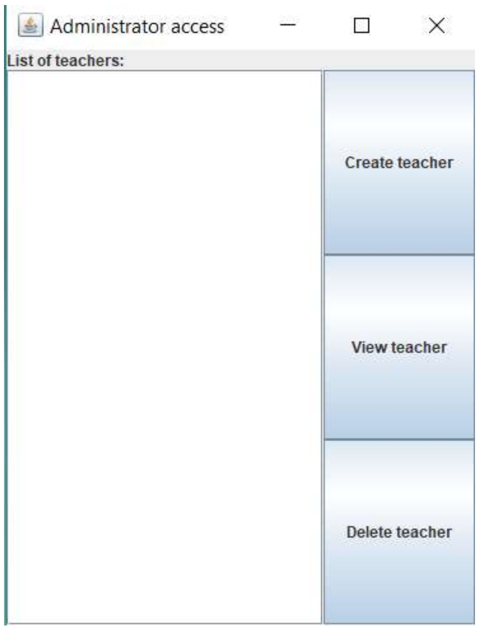


Figure 9. GUI Administrator

The "Create teacher" button opens another window with 2 text fields and 2 buttons: "Save" and "Cancel". The admin can input the name and CPR code of the teacher, press the "Save" button, which will update the "list of teachers" or to press the "Cancel" button to cancel the whole process.

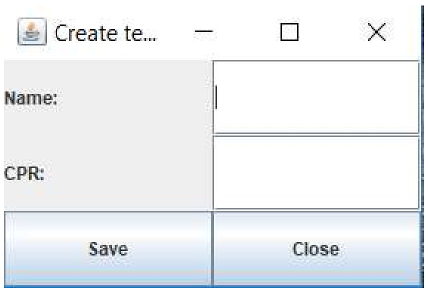


Figure 10. GUI Create teacher

To delete a teacher, the admin has to select him from the "list of teachers" and simply click the "Delete teacher" button.

When a teacher logs-in for the first time, he will be prompted with a "Create Class" window that contains a text field where he has to input the name of his class. There is also a "Create Class" button that will save the class and send him to the main teacher window, with all the features that are related to the teacher.

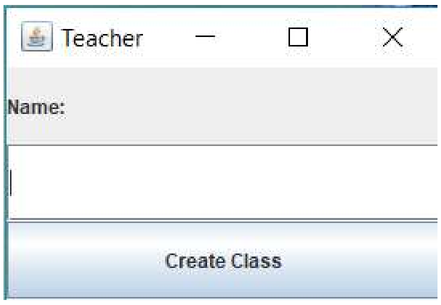
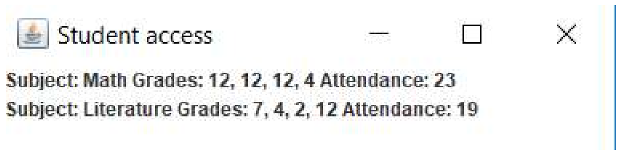


Figure 11. GUI Create class

Similar to the admin interface, the teacher window contains a "list of students", but this time 4 buttons: "Create student", "View student", "Delete student" and "View messages/Contact parent".

The teacher's features work similar to the admin's features, clicking the "Create student" button will open a page with multiple text fields and "Save" and "Cancel" buttons. The "View Student" button works in a similar way, for the "Delete Student" button, a student has to be selected from the list first, and once it is clicked, the student will be removed from the list.

The student interface is straight-forward, it displays the grades and the attendance of the said student.



Subject: Math Grades: 12, 12, 12, 4 Attendance: 23
Subject: Literature Grades: 7, 4, 2, 12 Attendance: 19

Figure 12. GUI Student access

The parent interface is very similar to the one for the student – it displays the grades and attendance for the assigned student, but the only difference is that the parent contains a button "View messages/Contact teacher".

Implementation

The system implementation was done gradually following the loose coupling concept. Although the process went without any critical complications, loose coupling was not used to its full potential. Almost every pattern planned, found its respective place inside the system with the exception of Observer pattern, which was left in its conception and did not fit in the time schedule. Most of the bugs were fixed with the help of the supervisors. The finished product fits almost perfectly into the envisioned initial design, with the exception of the message feature, which was left in its design stage, due to poor planning. Package structure of the server does not match the design diagram because of last minute bug fixing.

Server Implementation

RMI connection was chosen as it satisfies the needs of the customer and the limited time frame of the team as it does not require a strict protocol to be followed.

A **Singleton pattern** is used in the server class to avoid the instantiation of another server object in the system.

A School object is generated inside the Server constructor and a DatabaseAdapter object is used to load all the data from the Database and assign it to the newly instantiated School object.

NOTE!

***Observer Pattern** was planned to be implemented, with the Server class, to notify and print out changes made by clients to the School object.*

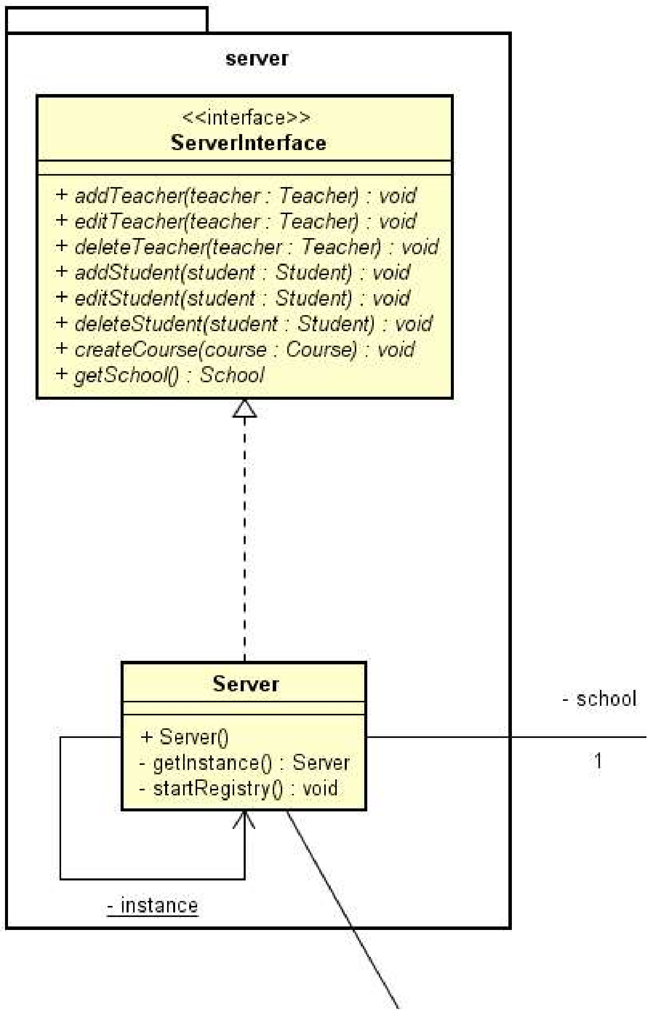
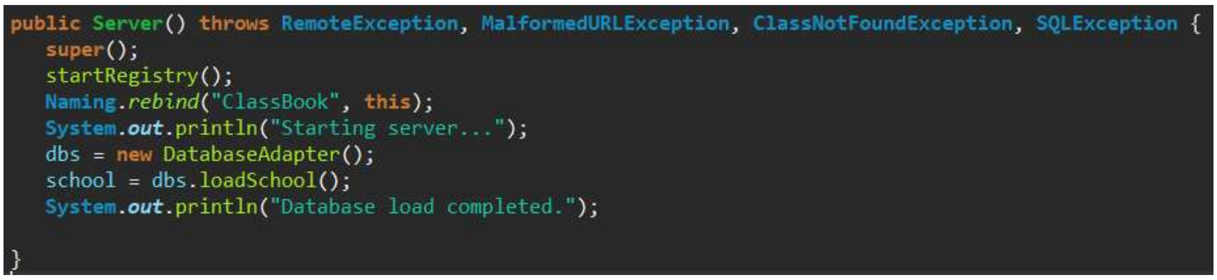

server
<<interface>>
ServerInterface
+ addTeacher(teacher : Teacher) : void
+ editTeacher(teacher : Teacher) : void
+ deleteTeacher(teacher : Teacher) : void
+ addStudent(student : Student) : void
+ editStudent(student : Student) : void
+ deleteStudent(student : Student) : void
+ createCourse(course : Course) : void
+ getSchool() : School
Server
+ Server()
- getInstance() : Server
- startRegistry() : void
- instance
- school
1

Figure 13. Server UML diagram

Ultimately all the data from the Server is saved back to the Database whenever the server program is terminated. This proves to be an ineffective way of data transfer as data can be easily lost in different cases of system crashes. The mentioned implementation is done due to time restrictions.

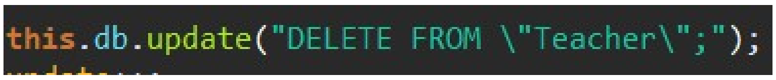
The methods used to implement it are Runtime calls, assigning threads that save back the Server data to the Database whenever the system terminates.



```
public Server() throws RemoteException, MalformedURLException, ClassNotFoundException, SQLException {
   super();
   startRegistry();
   Naming.rebind("ClassBook", this);
   System.out.println("Starting server...");
   dbs = new DatabaseAdapter();
   school = dbs.loadSchool();
   System.out.println("Database load completed.");

}
```

Figure 14. Server constructor integration



```
this.db.update("DELETE FROM \"Teacher\";");
```

Figure 15. Delete function in saveSchool() method

Adapter pattern implementation was done with the purpose of loading and saving data from/to the Database. Due to previously mentioned time limitations two main methods were used to interact with the Database – respectively loadSchool() and saveSchool(). A specific detail in the save method is the annihilation of all data stored inside the Database. The preferred implementation of that method was to generate SQL code to look and modify already existing data by its primary key.

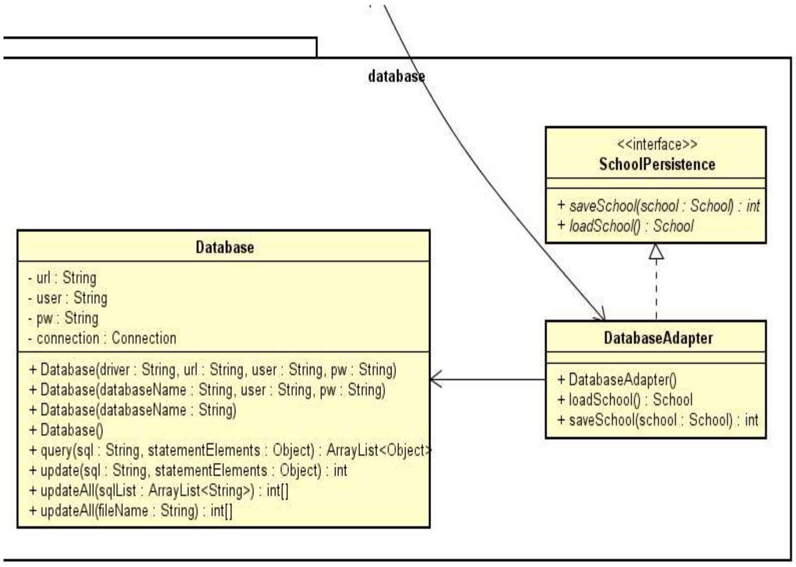
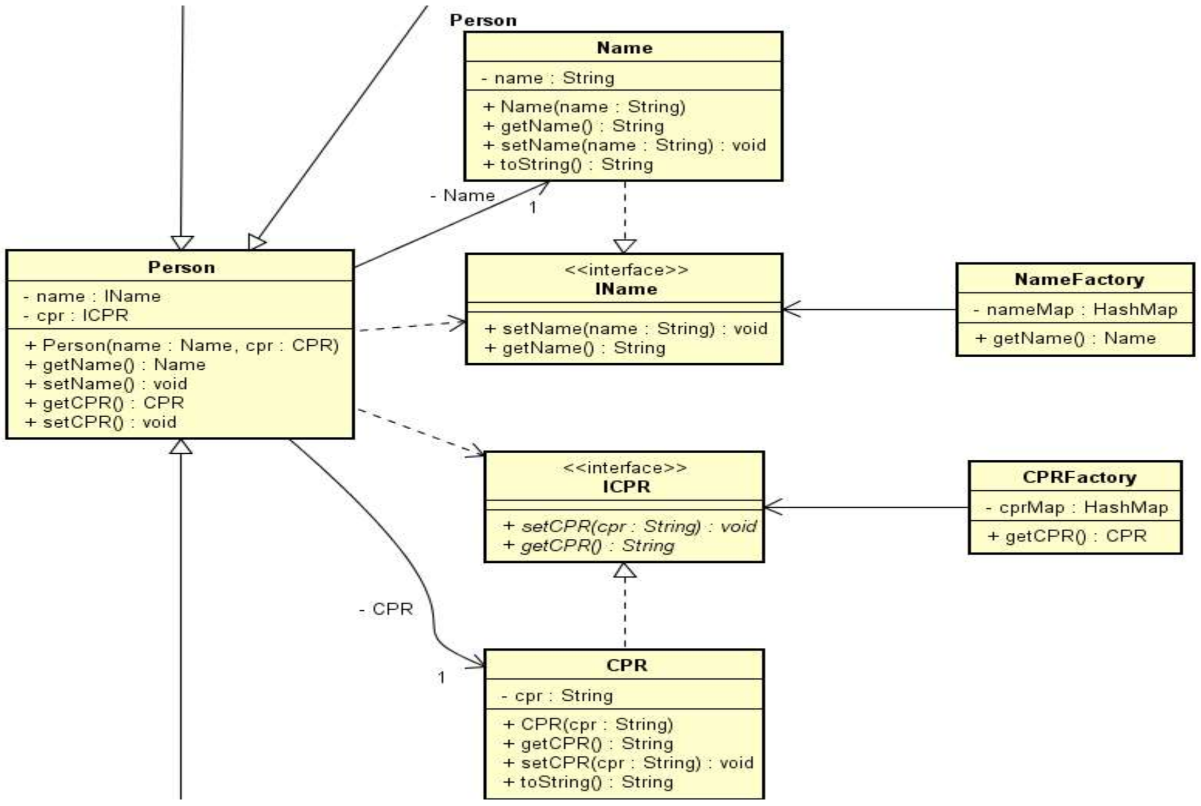

database
<<interface>>
SchoolPersistence
+ saveSchool(school : School) : int
+ loadSchool() : School
Database
- url : String
- user : String
- pw : String
- connection : Connection
+ Database(driver : String, url : String, user : String, pw : String)
+ Database(databaseName : String, user : String, pw : String)
+ Database(databaseName : String)
+ Database()
+ query(sql : String, statementElements : Object) : ArrayList<Object>
+ update(sql : String, statementElements : Object) : int
+ updateAll(sqlList : ArrayList<String>) : int[]
+ updateAll(fileName : String) : int[]
DatabaseAdapter
+ DatabaseAdapter()
+ loadSchool() : School
+ saveSchool(school : School) : int

Figure 16. Adapted patter UML diagram



Figur 17. Person package showing UML diagrams of Flyweight implementation

Flyweight pattern was chosen to reduce the data usage of the system. Person class is extended by Teacher, Parent and Student alike. Flyweight is implemented into Person as two Flyweight classes – Name and CPR. CPR uses Flyweight as the team had predicted that Parents may also be Teachers in the school, avoiding the duplication of objects. Name class implements Flyweight as to avoid duplication of “clone” Name objects sharing similar values.

Proxy pattern was used in order to “embrace” Loose coupling in Student as Student class holds roughly 40% of all data. As previously mentioned Loose coupling is not used to its full potential. The purpose of a class is to hold 2 to 3 functions which is not the case with our implementation. The data of a Subject class is hardcoded into Student as a String array due to previously mentioned time limitations. Two instances of the Subject class are generated inside Student and each holds its respective grades and attendance. The Student constructor requires a Parent object, helping us easily find parents inside the system, by using students CPR for searching. The ProxyStudent calls its respective methods from the RealStudent class, limiting the access the system has to RealStudent objects. Both Parent and Student classes extend Person class.

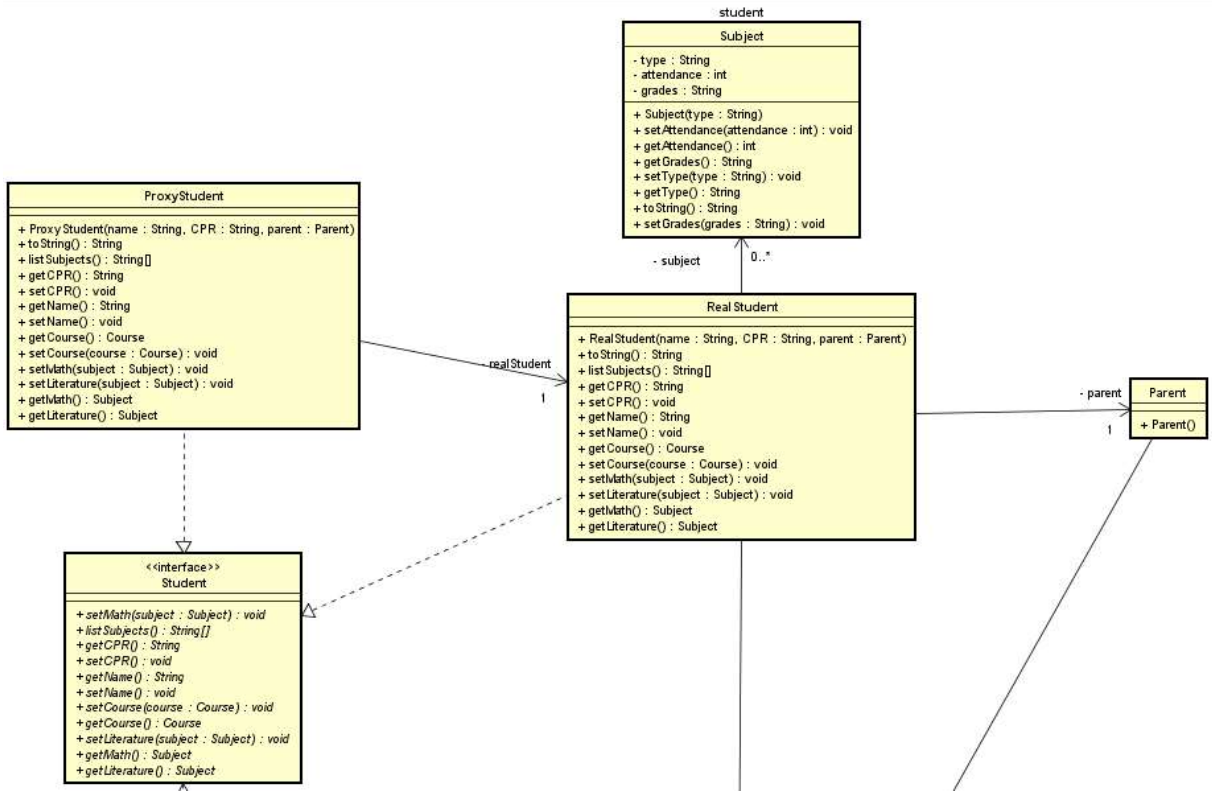
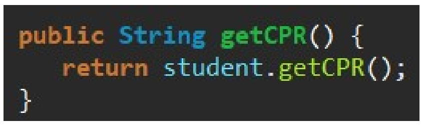

student
Subject
- type : String
- attendance : int
- grades : String
+ Subject(type : String)
+ setAttendance(attendance : int) : void
+ getAttendance() : int
+ getGrades() : String
+ setType(type : String) : void
+ getType() : String
+ toString() : String
+ setGrades(grades : String) : void
- subject
0..*
ProxyStudent
+ ProxyStudent(name : String, CPR : String, parent : Parent)
+ toString() : String
+ listSubjects() : String[]
+ getCPR() : String
+ setCPR() : void
+ getName() : String
+ setName() : void
+ getCourse() : Course
+ setCourse(course : Course) : void
+ setMath(subject : Subject) : void
+ setLiterature(subject : Subject) : void
+ getMath() : Subject
+ getLiterature() : Subject
- realStudent
1
Real Student
+ RealStudent(name : String, CPR : String, parent : Parent)
+ toString() : String
+ listSubjects() : String[]
+ getCPR() : String
+ setCPR() : void
+ getName() : String
+ setName() : void
+ getCourse() : Course
+ setCourse(course : Course) : void
+ setMath(subject : Subject) : void
+ setLiterature(subject : Subject) : void
+ getMath() : Subject
+ getLiterature() : Subject
- parent
1
Parent
+ Parent()
<<interface>>
Student
+ setMath(subject : Subject) : void
+ listSubjects() : String[]
+ getCPR() : String
+ setCPR() : void
+ getName() : String
+ setName() : void
+ setCourse(course : Course) : void
+ getCourse() : Course
+ setLiterature(subject : Subject) : void
+ getMath() : Subject
+ getLiterature() : Subject

Figure 18. UML Diagram of Proxy pattern implemented into Student, as well as Parent and Subject class implementation



```java
public String getCPR() {
    return student.getCPR();
}
```

Figure 19. Example of the Proxy class calling a method

Teachers are stored in the system with a Teacher class, extending Person. TeacherList class has the purpose of storing all the teachers in the school into an ArrayList. It implements Singleton in order to avoid multiple instances being created.

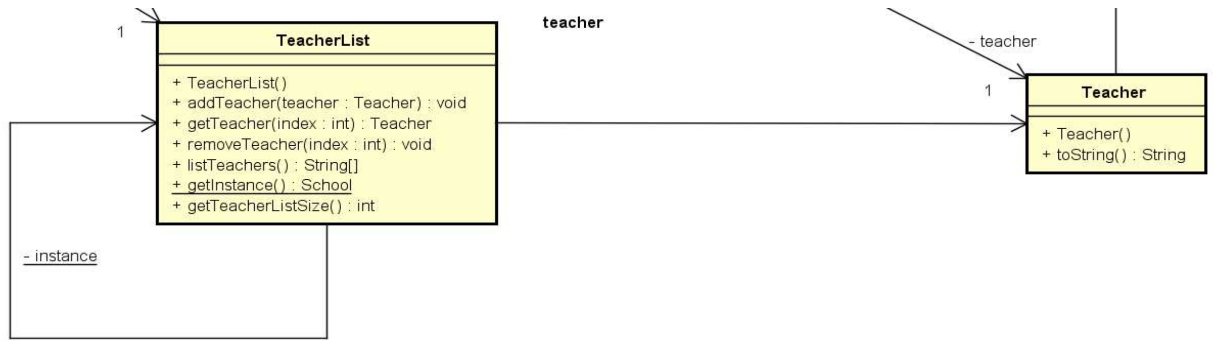

teacher
1
TeacherList
+ TeacherList()
+ addTeacher(teacher : Teacher) : void
+ getTeacher(index : int) : Teacher
+ removeTeacher(index : int) : void
+ listTeachers() : String[]
+ getInstance() : School
+ getTeacherListSize() : int
- instance
- teacher
1
Teacher
+ Teacher()
+ toString() : String

Figure 20. UML Diagram o Teacher class implementation

Course class is designed to take the responsibility of storing a Teacher for a headmaster and an ArrayList of Student objects. Its purpose is to separate school classes from one another. In order for a Course to be created, a Teacher object should be included in the constructor.

School class implements Singleton for the same purpose previous Singleton implementations have. The class stores an instance of TeacherList as well as an ArrayList of type Courses. The use of School object revolves around Database loads and saves as well as Server – Client data transfers.

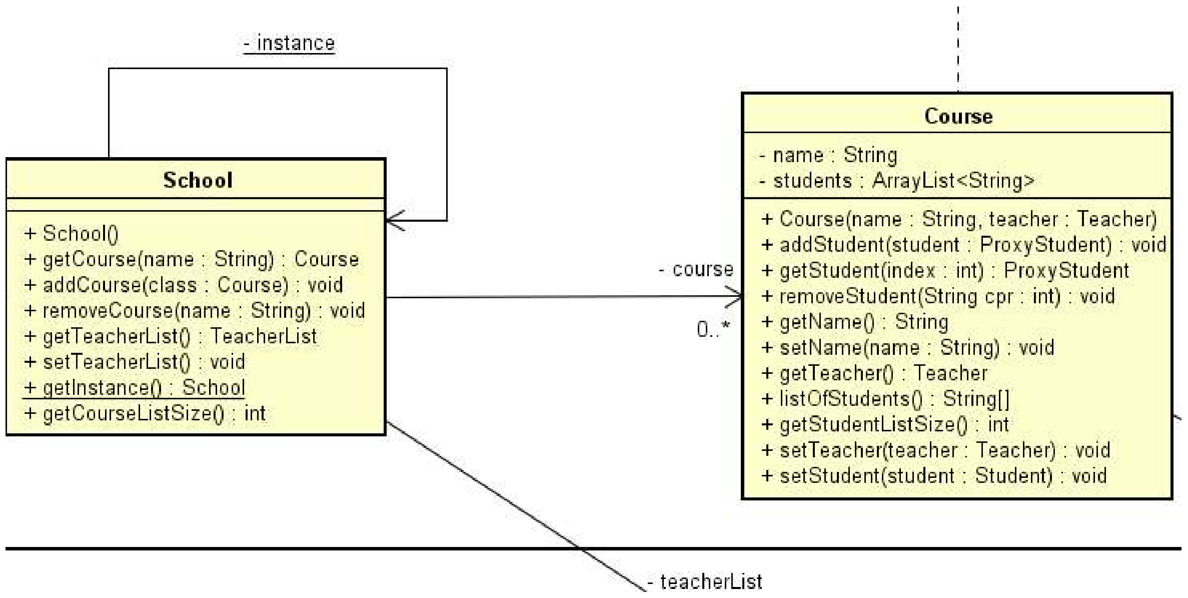
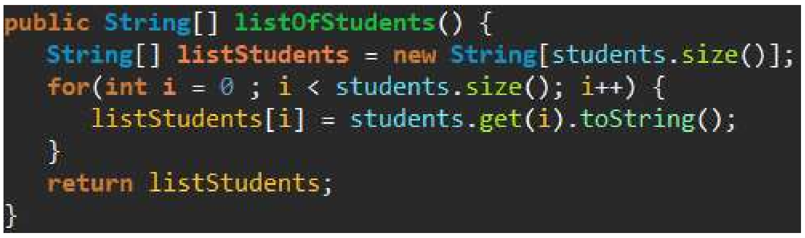

- instance
School
+ School()
+ getCourse(name : String) : Course
+ addCourse(class : Course) : void
+ removeCourse(name : String) : void
+ getTeacherList() : TeacherList
+ setTeacherList() : void
+ getInstance() : School
+ getCourseListSize() : int
- course
0..*
Course
- name : String
- students : ArrayList<String>
+ Course(name : String, teacher : Teacher)
+ addStudent(student : ProxyStudent) : void
+ getStudent(index : int) : ProxyStudent
+ removeStudent(String cpr : int) : void
+ getName() : String
+ setName(name : String) : void
+ getTeacher() : Teacher
+ listOfStudents() : String[]
+ getStudentListSize() : int
+ setTeacher(teacher : Teacher) : void
+ setStudent(student : Student) : void
- teacherList

Figure 21. UML Diagram Example of Course and School class implementation

Methods such as listOfStudents() have the return type of String[], which is used to display data in JLists on the client side.



```
public String[] listOfStudents() {
   String[] listStudents = new String[students.size()];
   for(int i = 0 ; i < students.size(); i++) {
      listStudents[i] = students.get(i).toString();
   }
   return listStudents;
}
```

Figure 22. listOfStudents() method example

Client Implementation

The client side of the system makes use of the Model View Controller framework as it provides a user interface. MVC makes the perfect fit for our client goals separating functionality from visualization.

The Client class was implemented to instantiate ServerInterface in order to call methods from the server. It also implements its own ClientInterface in order for the Model to make a connection to the Client and send back data to the Server.

With the instantiation of a Client object inside the Models constructor, the getSchool() method is called. This concept is not the best outcome, because if another client makes changes to the server data, the current client will not be able to access those changes. The only way to get a hold of data updated by other clients is to log- in again. The implementation was done that way because of decisions made by the Product Owner with the goal of ultimately providing a fully functional product.

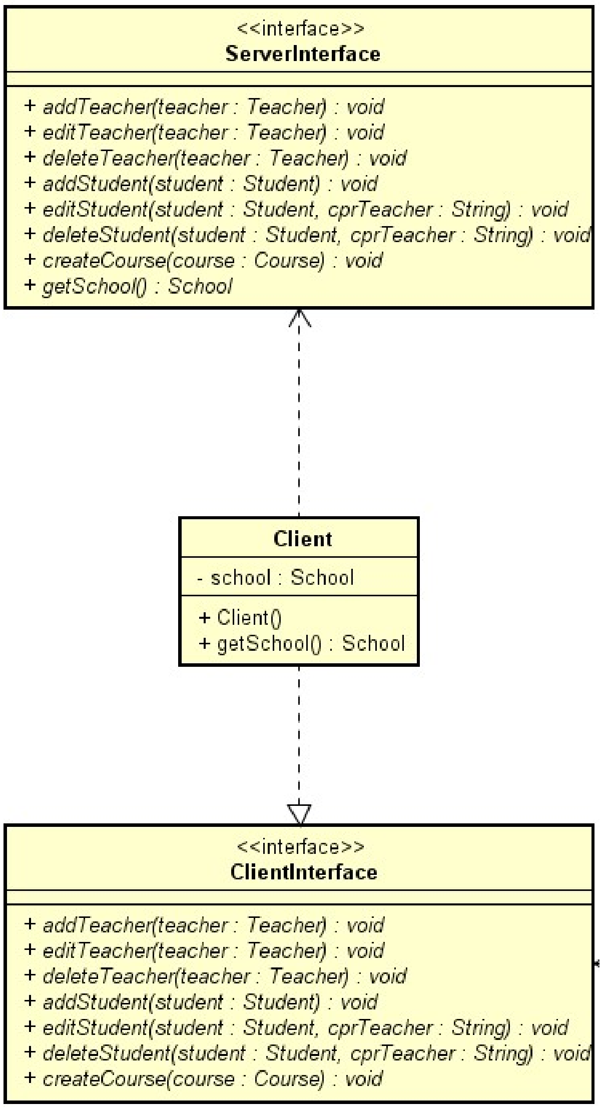

<<interface>>
ServerInterface
+ addTeacher(teacher : Teacher) : void
+ editTeacher(teacher : Teacher) : void
+ deleteTeacher(teacher : Teacher) : void
+ addStudent(student : Student) : void
+ editStudent(student : Student, cprTeacher : String) : void
+ deleteStudent(student : Student, cprTeacher : String) : void
+ createCourse(course : Course) : void
+ getSchool() : School
Client
- school : School
+ Client()
+ getSchool() : School
<<interface>>
ClientInterface
+ addTeacher(teacher : Teacher) : void
+ editTeacher(teacher : Teacher) : void
+ deleteTeacher(teacher : Teacher) : void
+ addStudent(student : Student) : void
+ editStudent(student : Student, cprTeacher : String) : void
+ deleteStudent(student : Student, cprTeacher : String) : void
+ createCourse(course : Course) : void

Figure 23. UML Diagram of client side RMI

Model has the goal of sending data to the controller, receiving data from the controller and sending it to the Server. The first method determines the log- in of a user. The log- in is done through a 10 digit long String(CPR) received from the Controller which initially got it from the View. That String is passed to the determineLogin method which goes through the CPR values of every single object, extending Person in the system. Afterwards it returns a String with the type of user logging in. It is also designed to return an error if a match to an existing object had not been found.

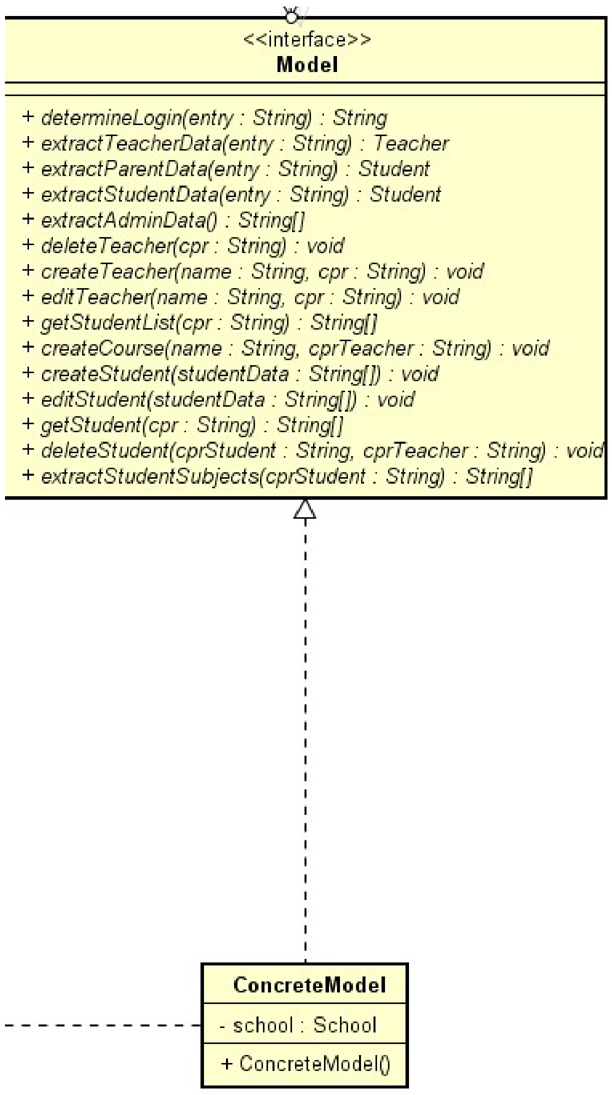
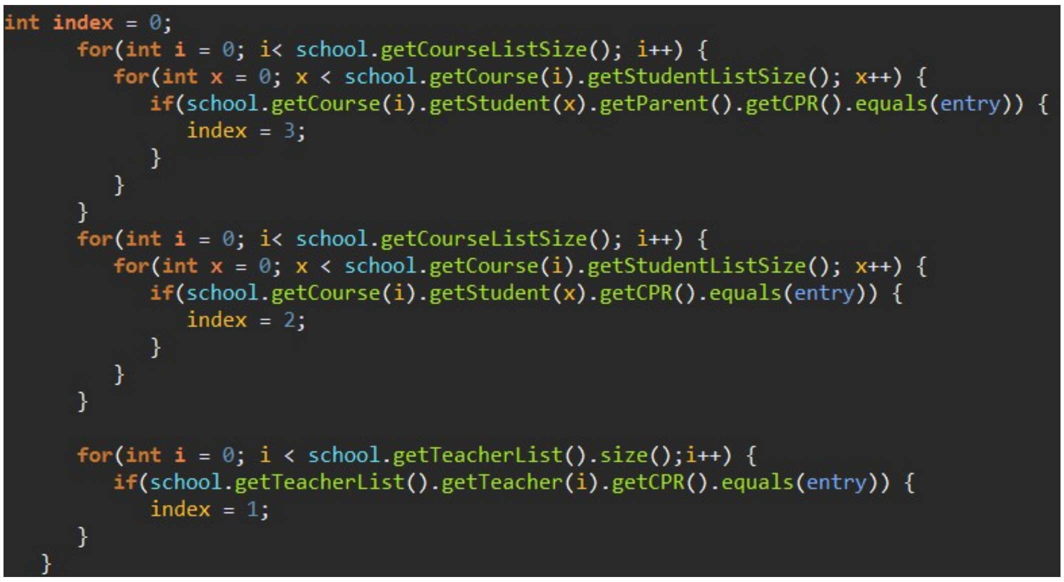

<<interface>>
Model
+ determineLogin(entry : String) : String
+ extractTeacherData(entry : String) : Teacher
+ extractParentData(entry : String) : Student
+ extractStudentData(entry : String) : Student
+ extractAdminData() : String[]
+ deleteTeacher(cpr : String) : void
+ createTeacher(name : String, cpr : String) : void
+ editTeacher(name : String, cpr : String) : void
+ getStudentList(cpr : String) : String[]
+ createCourse(name : String, cprTeacher : String) : void
+ createStudent(studentData : String[]) : void
+ editStudent(studentData : String[]) : void
+ getStudent(cpr : String) : String[]
+ deleteStudent(cprStudent : String, cprTeacher : String) : void
+ extractStudentSubjects(cprStudent : String) : String[]
ConcreteModel
- school : School
+ ConcreteModel()

Figure 24. UML Diagram of the Model interface

After a log in has been determined, extractData methods are called depending on the value received from determineLogin. These methods are burdened with the job of finding the specific user trying to log into the system, using the user CPR as an argument. After they find the user, they send back all the user data in the form of Strings.

The rest of the methods are logically similar. They receive a CPR as an argument, use it to find/change specific data in the system and afterwards they send the new data to the server. The same data is send to the server interface to acheive relevance as well as the CPR of user to keep up the log-in identity. The ConcreteModel class has the purpose of keeping a client instance and transferring data from/to it.



```
int index = 0;
      for(int i = 0; i< school.getCourseListSize(); i++) {
         for(int x = 0; x < school.getCourse(i).getStudentListSize(); x++) {
            if(school.getCourse(i).getStudent(x).getParent().getCPR().equals(entry)) {
               index = 3;
            }
         }
      }
      for(int i = 0; i< school.getCourseListSize(); i++) {
         for(int x = 0; x < school.getCourse(i).getStudentListSize(); x++) {
            if(school.getCourse(i).getStudent(x).getCPR().equals(entry)) {
               index = 2;
            }
         }
      }

      for(int i = 0; i < school.getTeacherList().size();i++) {
         if(school.getTeacherList().getTeacher(i).getCPR().equals(entry)) {
            index = 1;
      }
   }
```

Figure 25. Loop logic inside determinedLogin() method

NOTE!

The lack of proper loose coupling implementation begins to stand out the deeper we get into the Model logic.

The Controller class makes a connection between Model and View. It stores instances of both and calls different methods on them. The method setView() is used to assign a new View instance whenever a new GUI window is opened. Execute method holds a switch with 13 cases inside it, executing the combined logic of Model and View.

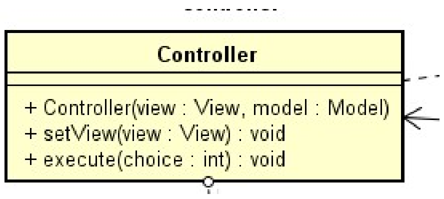
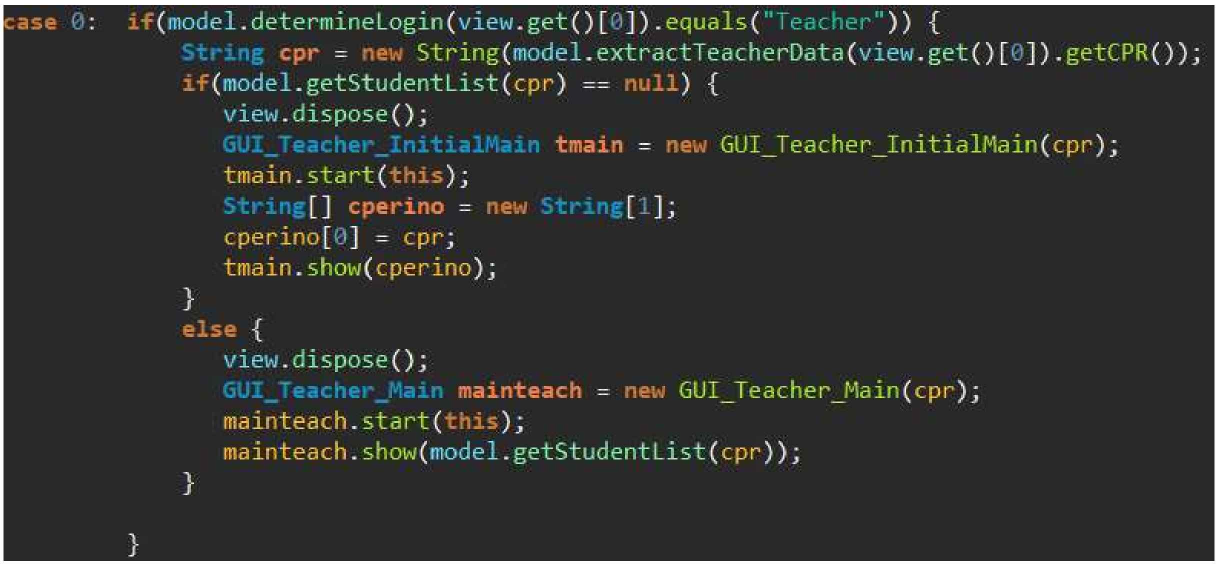

Controller
+ Controller(view : View, model : Model)
+ setView(view : View) : void
+ execute(choice : int) : void

Figure 26. UML Diagram of the Controller class

The logic inside the switch cases holds the same concept of receiving data from the View(user CPR being part of it in order to keep log-in relevance), calling methods from the Model with the data received and finally applying the returned data from the Model to a newly instantiated View.

A big part of the log- in happens in the Controller, as it has to determine if the Teacher is the headmaster of an already existing Course, If he/she is not, the controller initiates the Create Class GUI.

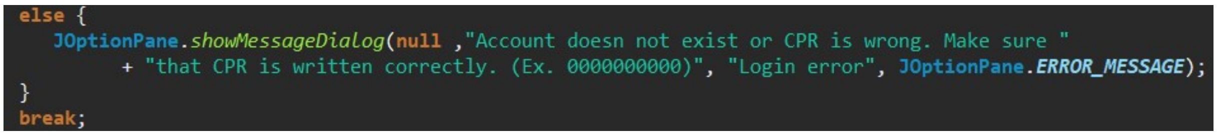


```
case 0:   if(model.determineLogin(view.get()[0]).equals("Teacher")) {
              String cpr = new String(model.extractTeacherData(view.get()[0]).getCPR());
              if(model.getStudentList(cpr) == null) {
                 view.dispose();
                 GUI_Teacher_InitialMain tmain = new GUI_Teacher_InitialMain(cpr);
                 tmain.start(this);
                 String[] cperino = new String[1];
                 cperino[0] = cpr;
                 tmain.show(cperino);
              }
              else {
                 view.dispose();
                 GUI_Teacher_Main mainteach = new GUI_Teacher_Main(cpr);
                 mainteach.start(this);
                 mainteach.show(model.getStudentList(cpr));
              }

          }
```

Figure 27. Code example of a teacher log-in from the execute method of the Controller class

The user input is strictly controlled through the GUI in order to avoid internal system errors. The controller takes the responsibility of providing error message in cases where provided data is invalid.



```
else {
   JOptionPane.showMessageDialog(null ,"Account doesn not exist or CPR is wrong. Make sure "
         + "that CPR is written correctly. (Ex. 0000000000)", "Login error", JOptionPane.ERROR_MESSAGE);
}
break;
```

Figure 28. Code example of an error message because of invalid data input

The View is held responsible for displaying the user interface. The start() method is used to assign a controller instance to a View object as well as to assign the View to the controller. Show() method is used to draw data from the controller. The get() method returns user input and is used in the controller and sent to the model. The dispose() method closes the interface window. A private nested ButtonHandler class is implemented in every GUI, handling the functions of different buttons inside the user interface.

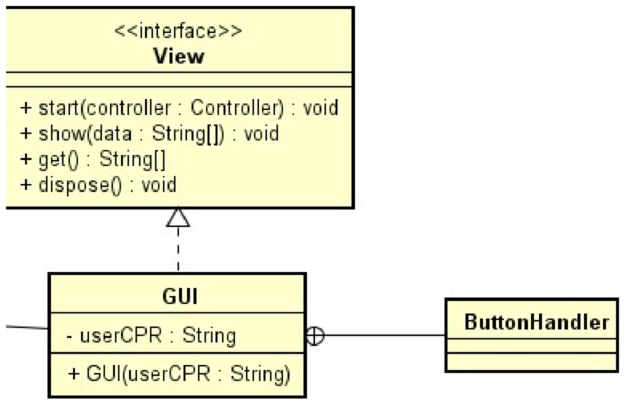
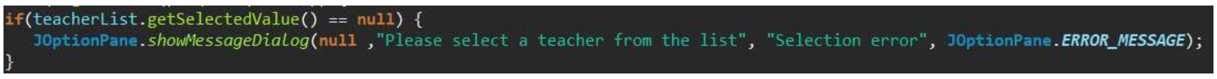


Figure 29. UML Diagram of the View package

A lot of GUI barriers have been put in order to assure that valid information is flowing to the Controller.



```
if(teacherList.getSelectedValue() == null) {
    JOptionPane.showMessageDialog(null ,"Please select a teacher from the list", "Selection error", JOptionPane.ERROR_MESSAGE);
}
```

Figure 30. Code example of an error message inside Admin_Main GUI

NOTE!

The value of most data in the system is of type String in order to avoid parsing and unpredicted exceptions.

Unimplemented Features

- **Observer pattern** as mentioned before, did not find its place in the time schedule.
- **Message function** had better luck being conceptualized and designed, but due to time cuts did not make it to the implementation.

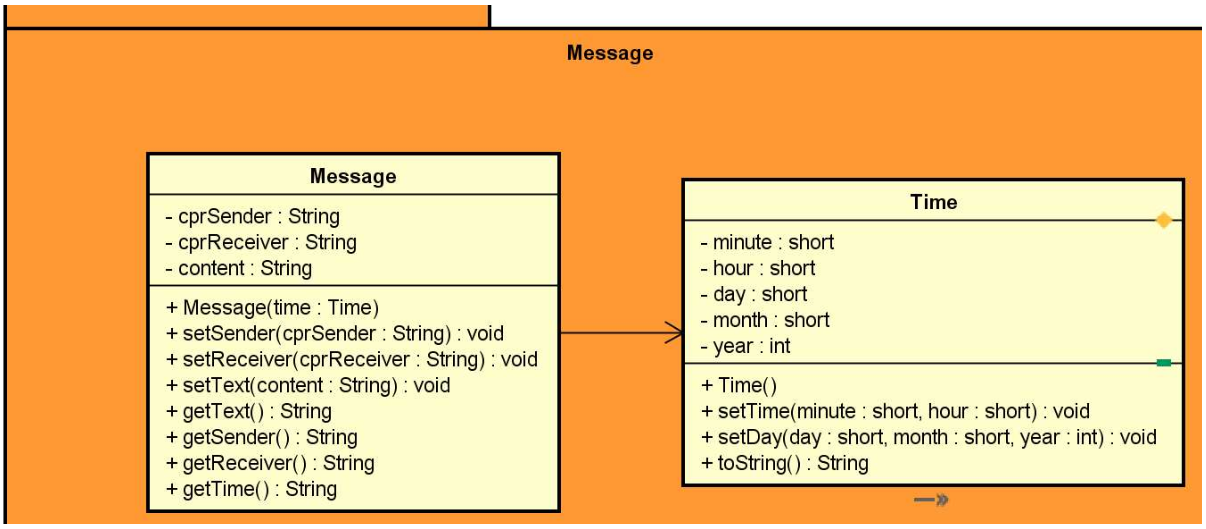
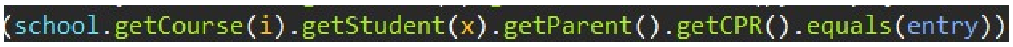


Figure 31. UML Diagram design of the Message feature

Implementation Retrospective

Although the system is working and implementation has finished, there are some points that the group wants to address.

- Loose coupling was not utilized to its full extend and cases such as the one shown below could have been avoided.
  Example



```
(school.getCourse(i).getStudent(x).getParent().getCPR().equals(entry))
```

- The patterns implemented help the system overall, but there are some cases where Flyweight could extensively be used (Subject class) as well as Proxy (School class).

- The big amounts of code and nested loops as well as the lack of comments make the logic of the system hard to follow.

Implementation Conclusion

Although implementation did not execute in the best case scenario, the structure of the system makes new features as Message to be implemented easily, without major complications.

Test

White Box Test

Case	Target	Method	Status
As a teacher I want to create a new student	RealStudent	RealStudent(String name, String cpr, Parent parent);	PASS
As a teacher I want to set/change the name of a student	RealStudent	setName(String name);	PASS
As a teacher I want to get the name of a student	RealStudent	getName();	PASS
As a teacher I want to add grades to the subject of a student	Subject	Subject(String type);	PASS
As a teacher I want to set the attendance of a subject	Subject	setAttendance(String attendance);	PASS
As a teacher I want to get the attendance of a subject	Subject	getAttendance();	PASS
As a teacher I want to set/change the grades of a subject	Subject	setGrades();	PASS
As a teacher I want to get the grades of a subject	Subject	getGrades();	PASS

Black Box Test

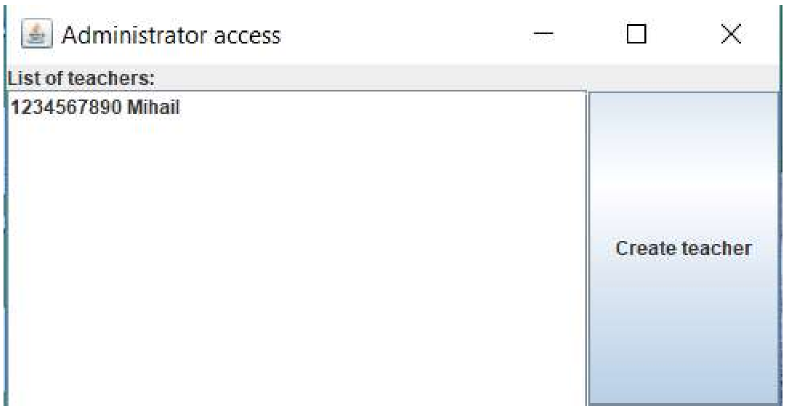
In this part of the report, the functionality of the application will be examined without emerging into its internal structures or workings.

Test case 1: Admin creating a teacher

The test will be conducted by accessing the admin features, clicking the "Create teacher" button from the interface, inputing the name as "Mihail" and the CPR as "1234567890" and then clicking the "Save" button.

Expected result: a teacher with the name of "Mihail" and the CPR of "1234567890" will be added to the List of teachers.

Test result: Success, feature works as intended.



1234567890 Mihail

Figure 32. GUI List of teachers

Test case 2: Admin deleting a teacher

The test will be conducted by starting with a list of four teachers.

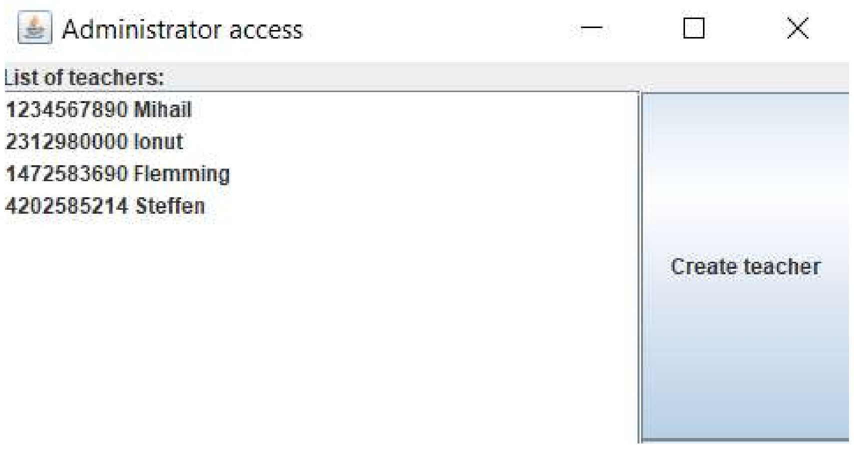

Administrator access
List of teachers:
1234567890 Mihail
2312980000 Ionut
1472583690 Flemming
4202585214 Steffen

Figure 33. GUI Updated list of teachers

An attempt to delete the 3rd teacher will be made(the one with the name "Flemming" and the CPR code" 1472583690"), by selecting him from the list of teachers and clicking the "Delete teacher" button.

Expected result: The teacher with the name "Flemming" and the CPR code "1472583690" will be deleted from the list of teachers.

Test result: Success, feature works as intended.

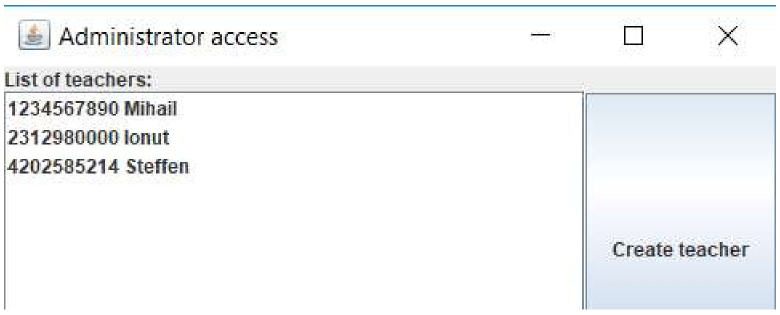

Administrator access
List of teachers:
1234567890 Mihail
2312980000 Ionut
4202585214 Steffen

Figure 34. GUI More teacher list

Test case 3: Teacher creating a class

The test case will be conducted by logging in with the data of a teacher, which will prompt up a “Create class” window. A name for the class will be imputed in the “name” text box, the “Create Class” button will be pressed, which then should lead the Teacher to the teacher panel, with its features.

Expected result: The class will be created and the teacher will be sent to the teacher panel, with all its features.

Test result: Success , feature works as intended.

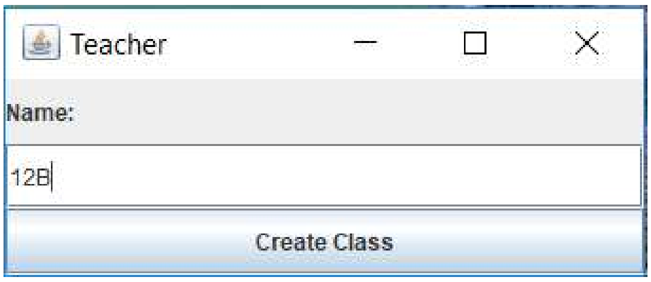


Figure 35. GUI Create class (again?)

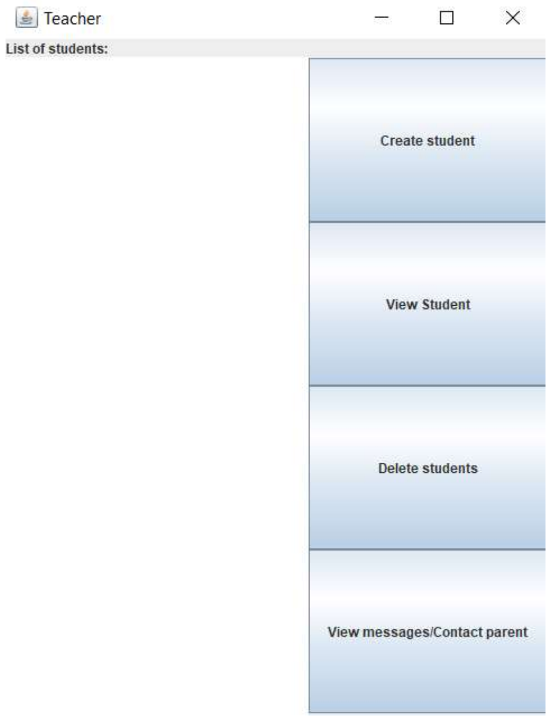


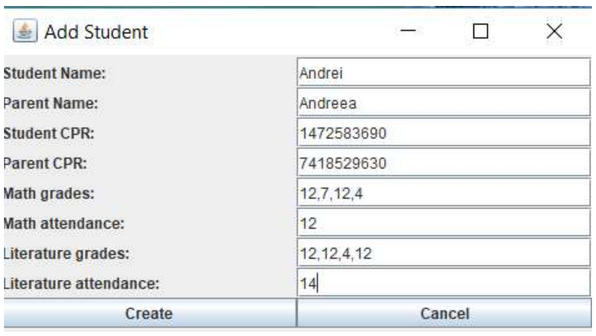
Figure 36. GUI List of students

Test case 4: Teacher creating a student

The test case will be conduced by clicking the "Create student" from the Teacher interface. The text boxes from the "Add student" window will be filled up with valid data and the "Create" button will be clicked.

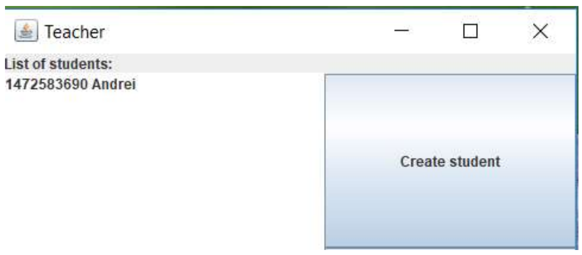
Expected result: The student and its data will be saved into the "List of students".

Test result: Success: Feature works as intended.


Add Student

Student Name:	Andrei
Parent Name:	Andreea
Student CPR:	1472583690
Parent CPR:	7418529630
Math grades:	12,7,12,4
Math attendance:	12
Literature grades:	12,12,4,12
Literature attendance:	14

Figure 37. GUI Add Student


Teacher

List of students:
1472583690 Andrei

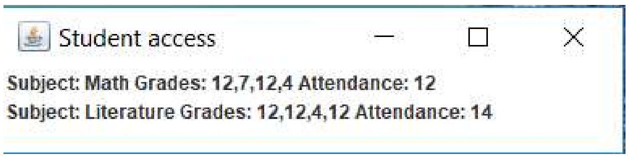
Figure 38. GUI List of students (again, again?)

Test case 5: Student viewing his grades and attendance

The test case will be conducted by logging in the system with the CPR of a student.

Expected result: A window should pop-up, with the grades and attendance of the students for his/hers subjects.

Test result: Success, feature works as intended.



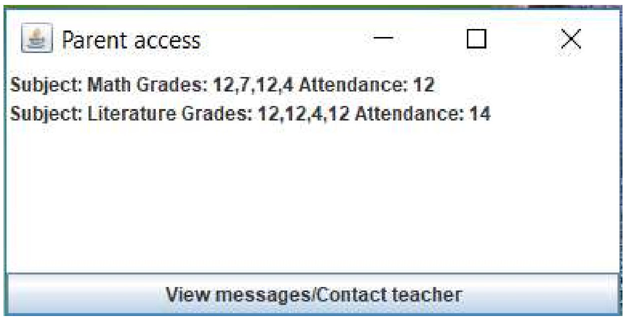
```
Subject: Math Grades: 12,7,12,4 Attendance: 12
Subject: Literature Grades: 12,12,4,12 Attendance: 14
```

Figure 39. GUI View of a student

Test case 6: Teacher deleting student – Deleted student's parent trying to log in

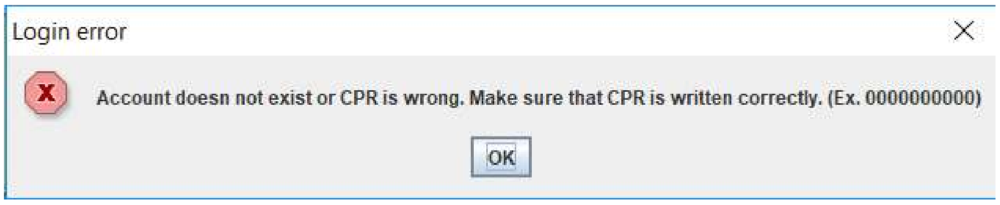
The test case will be conducted by first logging in as a parent. Then the teacher will select the assigned student of the parent from the list of students, pressing the "Delete button". Then another instance of the application will be run, and in the log-in window, the CPR of the deleted student's parent will be imputed.

Expected result: The first time the parent logs in, the grades and attendance of the student should show. Ultimately the student is deleted. If the parent logs-in, he/she should be granted a log-in error, because the student is no longer exists in the system, which means that the parent does not exist as well.
Test result: Success, feature works as intended.


Parent access

Subject: Math Grades: 12,7,12,4 Attendance: 12
Subject: Literature Grades: 12,12,4,12 Attendance: 14

Figure 40. GUI Another view of a student


Login error

Account doesn not exist or CPR is wrong. Make sure that CPR is written correctly. (Ex. 0000000000)

Figure 41. GUI Login error

Result

The Class Book System was developed with the purpose of meeting the requirements of a multi-user data management system, storing, modifying and removing information. The application allows for multiple types of users, each with their own unique set of features that have been fully implemented into a foolproof graphical user interface.

- Admin must be able to create, delete and modify Teachers.
- Admin must have only one account in the Database.
- Admin account will be hardcoded within the System.
- Every Class must have only one Teacher as a Headmaster.
- Teacher must be able to add Students to his/her class.
- Teacher must be able to create and assign Parents to Students.
- Teacher must be able to change grades and attendance of every Student.
- Teacher must be able to modify name of every Parent/Student.
- Parent must be able to view grades and attendance of his/her own Student.
- Student must be able to view his/her own grades.
- System must be maintainable and easy to update.
- System must store data into a Database.
- System will use Client-Server connection of type RMI.
- System will use CPR number as a log-in.
- System will use connection of type LAN.
- System will be coded in Java and SQL.
- System will not include a live-chat feature.

Discussion

As mentioned before, when the implementation phase started, the main focus was to follow the top-priority requirements, with the main goal that the final version of the application will integrate all of these features.

Conclusion

The purpose of this project was to create a multi-user system that facilitates a better educational experience. The application is used to keep track of a school's teachers, students, parents, grades and attendance, thus fulfilling all the top-priority requirements and the needs of the end-users. Along the way, during the project duration, some of the requirements have been changed or some of them have gained a different priority, new ones have been added, but every single change that occurred relating the system has been made for a better end-user experience.

Project Future

There are many ways the project could be expanded if the team decided to continue their work on it. For example, exception handling can be handled and implemented better, providing more information to the developer or creating a solution to the exception. The Database can be broadened and the values in the system can be controlled better, instead of saving data in strings. E-mail and chat functions can be easily implemented to the already existing System. Loose coupling can be emphasized on more as well as more patterns should be implemented in the system to boost its optimization. The team had had an idea of analyzed data gathered from all students, but the whole challenge was scrapped as it requires more advanced Data science knowledge. If eventually the team gets back on the project, the analysis feature can be further investigated and ultimately integrated into the system, providing a whole different approach to how the system is used and its place among educational institutions.

Sources of Information

Journal of Education and Practice, 2015, Vol.6, No.27, "Early Childhood Behavior Changing in Terms of Communication between Parents and Teachers" Available at: https://files-eric-ed-gov.ez-aaa.statsbiblioteket.dk:12048/fulltext/EJ1077393.pdf [Accessed March 04, 2018]

Egoza Wasserman, Yaffa Zwebner, International Journal of Learning, Teaching and Educational Research Vol. 16, No. 12, pp. 1-12, December 2017, "Communication between Teachers and Parents using the WhatsApp Application" Available at: http://ijlter.org/index.php/ijlter/article/view/1041/pdf [Accessed March 04, 2018]

Angelica Hobjilă, Procedia - Social and Behavioral Sciences, Volume 142, 14 August 2014, Pages 684-690, "Challenges in Continuing Education of Primary and Preschool Teachers in Romania: Teachers – Students' Parents Communication" Available at:

https://doi.org/10.1016/j.sbspro.2014.07.598 [Accessed March 03, 2018]

Philip J. Cook, Kenneth A. Dodge, Elizabeth J. Gifford, Amy B. Schulting, Children and Youth Services Review, Volume 82, November 2017, Pages 262-270, "A new program to prevent primary school absenteeism: Results of a pilot study in five schools" Available at: https://doi-org.ez-aaa.statsbiblioteket.dk:12048/10.1016/j.childyouth.2017.09.017 [Accessed March 04, 2018]

Blair Christopher Thompson, Joseph P. Mazer & Elizabeth Flood Grady, 23 February 2015, Pages 187-207, "The Changing Nature of Parent-Teacher Communication: Mode Selection in the Smartphone Era" Available at: https://www-tandfonline-com.ez-aaa.statsbiblioteket.dk:12048/doi/abs/10.1080/03634523.2015.1014382 [Accessed March 04, 2018]

http://www.itslearning.dk/foraeldreintra [Accessed March 04, 2018]

https://www.catalogulmeu.ro/ [Accessed March 04, 2018]

https://www.lectio.dk/ [Accessed March 04, 2018]

https://twitter.com/ [Accessed May 29, 2018]

Appendices