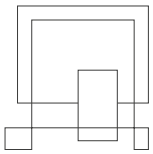
Bring ideas to life
VIA University College



eNTe Management System

Daniela Koch, 266502

Matej Michalek, 266827

Michaela Golhova, 266099

Michal Karol Pompa, 266494

Supervisors: Joseph Okika, Ib Havn

VIA University College

ente

57 850 characters

Information and Communication Technology Engineering

2nd semester

June 2018

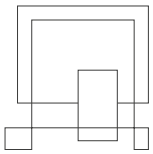
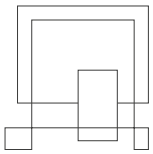
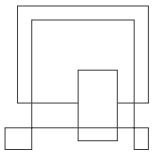


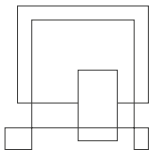
Table of content





Abstract

The purpose of this system was to make a maintainable system for eNTe (a small non-public primary modern free school basing on internal motivation of its students) that provides communication between students and teachers in the term of carrying out discussions, posting and submitting homework, as well as arranging meetings with parents. However, this project's focus is on managing users and depending on the type, providing specific functionalities concerning homework. The document presents the development of the system in particular sections: analysis, design, implementation, test and plans for future development. One can find information concerning the needs of the company both as a list of requirements and graphical representations on diagrams, the chosen architecture for the system and the reason for it (i.a. model-view-controller pattern and Sockets, in order to have a client-server system), follow the development of the database, the way the view was created, the chosen solution for the logging in system, including solutions for the security issues and much more. Another interesting component described is the builder pattern used to create users.



1 Introduction

„Children are our future" sings Whitney Houston in her song "Greatest Love of All". This quote is repeated more and more often, however, it is not indicating the importance of children on their own, but the importance of shaping them. In the world of today being educated is one of the most vulnerable qualifications (M. Schulte-Markwort, 2015). Bearing that in mind, a great attention to schools is being paid. The most desired competences in the 21st century, which should be cultivated by the educational system are: complex problem solving, critical thinking, creativity, people management, coordinating with others, emotional intelligence, judgment and decision making, service orientation, negotiation and cognitive flexibility (Alex Gray, 2016).

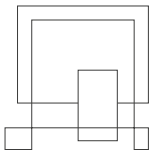
Unfortunately the traditional schooling in Poland is not adjusted to teaching the competences needed in the 21st century and is not preparing for life nor teaching vulnerable skills (Michał Pasterski, 2014). That is the reason why innovative, modern educational concepts and schools are growing in importance.

One of those schools is eNTe. eNTe is a small non-public primary modern free school basing on internal motivation of its students. It is increasing both critical and scientific thinking of its students. Moreover its focus is on teaching about democracy and with means of democratic tools. The rules in the school are established on the basis of consensus after discussions of the school community. Each week there is a plenum organized where the discussions are being held.

However, this indicates an issue. It is expected to establish the rules as a school community, i.e. all students and teachers together. Due to the fact that the meetings are being held at school, absent members (e.x. sick students or busy teachers) are not involved.

That leads to the needs of eNTe. It requires a system, where the discussions concerning the rules could be held. It would not only enable all students and teachers to participate in them, but also involve the parents. Moreover it needs a place to gather arguments and perform discussions on a variety of topics to enable the members of the community to acknowledge different opinions and basing on them come to compromises, as well as to increase their critical thinking and tolerance.

Another issue being faced by eNTe is homework. It is easy to forget the deadlines for assignments when they are being announced in advance of one or two weeks. That is why the school is in need of a place to store all the most important information concerning them: the topic, criteria, the number of students to deliver it together and the hand in date. Additionally, in the thought of being green, the institution desires a possibility for the students to deliver their homework electronically, in order not to waste paper.



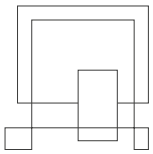
Regarding the matter of teamwork, which is one of the core skills in eNTe, a place to hold chat groups would be valuable. Not only would it simplify work in groups at home, but also would leave a footprint for the teachers concerning how each person performed, as well as how the whole group cooperated on a homework.

Another issue is the parental matter. As the fundamental of eNTe is communication, parent-teacher-child meetings are being held approximately twice a semester. What is causing difficulties in this area is the logistics. The school needs a solution that would simplify scheduling meetings.

Last but not least the school is striving for excellence, in the meaning of exploring new means and solutions constantly. What that indicates, is the need of a maintainable system, in which there could be implemented new functionalities and made changes continuously.

Even though school dedicated systems already exist, none of them meets all of the needs of eNTe. eNTe is an individual and original school with specific problems, what resolves in a need of a unique system dedicated specifically for it.

More details can be found in the project description appended in Appendix A.



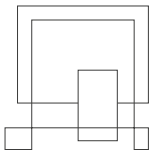
2 Requirements

Bearing in mind the needs of eNTe presented in the introduction, the requirements concerning the needed system are divided into functional, non-functional and special requirements.

2.1 Functional Requirements

All functional requirements are stated in the list below:

1. The user must be able to log in to the server using his/her email as the login
2. The user must be able to log out from the server
3. The system must divide users into 4 groups (students, teachers, parents and an administrator)
4. The system must divide students into classes
5. The administrator must be able to manage (create, edit, delete) users accounts (parent/student/teacher)
6. The teacher must be able to post homework in specific classes (one or more)
7. The teacher must be able to post discussion topics in chosen classes (one or more)
8. The teacher must be able to post an announcement for chosen classes
9. The teacher must be able to see who did the homework and access it
10. The teacher must be able to see who has seen a post
11. The student must not be able to see discussions and announcements marked as parental.
12. The system must display all posts marked as important on the top of the list of all posts.
13. The system must send emails regarding posts that are marked as important or parental to all users it concerns
14. The user must be able to comment a discussion that he/she has access to and edit it afterwards
15. The administrator and the teacher must be able to edit his/her own post
16. The teacher has to have a profile with a name and an email
17. The student has to have a profile with a name, email, class and history of activity, as well as his/her parents' names and emails
18. The system must contain a calendar with meetings*
19. The student must be able to see post for his/her class
20. The student must be able to post and edit his/her own solution to the homework before deadline
21. The user must be able to send private messages to other users
22. The user must be able to see private messages sent to and by him/her
23. The teacher must be able to create a chat group from students for a specific homework
24. The student must be able to send private messages in chat groups he/she is enrolled in
25. The teacher must be able to access every chat group and see the messages in it
26. The parent must be able to see posts posted for his/her child and marked as parental
27. The system must archive one year back
28. The administrator must be able to choose students that have not passed before the beginning of each year.
29. The system must change the class of each student that has passed before the beginning of each year



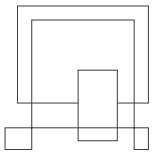
However, this projects focus is on requirements listed below:

1. The user must be able to log in to the server using his/her email as the login
2. The system must divide users into 4 groups (students, teachers, parents and an administrator)
3. The system must divide students into classes
4. The administrator must be able to manage (create, edit, delete) users accounts (parent/student/teacher)
5. The teacher must be able to post homework in specific classes (one or more)
6. The teacher must be able to see who did the homework and access it
7. The administrator and the teacher must be able to edit his/her own post
8. The student must be able to see post for his/her class
9. The student must be able to post and edit his/her own solution to the homework before deadline

To understand the requirements even better a data glossary has been attached in Appendix B. Use Case Descriptions for all functional requirements and Activity diagrams for the most complicated ones are attached in Appendix C.
An example of a use case description is presented below on Figure 1.

ITEM	VALUE
UseCase	Manage a post
Summary	The user creates/edits/deletes a homework/annoucment/discussion.
Actor	Teacher
Precondition	Edit/delete: a post already exists
Postcondition	A post is created and accessible for a choosen group of users/ Chosen information concerning the post has been changed/ A post has been deleted from the server and is not accessible anymore
Base Sequence	Create homework: 1. The user clicks "homework" button 2. The user clicks "add" button 3. The user fills a displayed form with the topic, criteria, number of students to deliver it together, chooses date from a calendar, enters the time for the deadline and chooses classes by selecting checkboxes. 4. The user clicks "save" button 5. The system notifies students from the chosen class that a homework has been added
Branch Sequence	Edit homework: 1. The user clicks "homework" button 2. The user chooses a homework from the list 3. The user clicks "edit" button 4. The user changes chosen fields from the displayed filled form 5. The user clicks "save" button 6. The system notifies students from the chosen class that a homework has been changed Delete homework: 1. The user clicks "homework" button 2. The user chooses a homework from the list 3. The user clicks "delete" button 4. The system displayes an "are you sure you want to delete this homework from the system?" message 5. The user clicks the "yes" button 6. The system deletes the specific homework from the server
Exception Sequence	IF any of the fields is empty THEN the system displays a message "all fields must be filled" IF the user clicks the logo before clicking the "save" buuton THEN the post is not saved IF the user clicks "cancel" button, no action is taken and the dialog box is closed
Sub UseCase	
Note	Frequency: high

Figure 1



2.2 Non-Functional Requirements

The non-functional requirements are as follows:

1. The system must be implemented in Java
2. Secondary storage needs to be done with databases

2.3 Special Requirements

The special requirements are defined below:

1. The main panel for students must contain a wall with information(homework, discussion and announcements)

3 Analysis

The first stage of creating a project is analysis. This is where the needs of the customer are analyzed and defined. It is the most important part of the system, as it creates the base of it. Therefore, next stages as Design, Implementation and Test were derived from deep analysis of this stage.

As the purpose was to make a maintainable system for eNTe, that would provide communication between students and teachers in the term of carrying out discussions, posting and submitting homework, as well as arranging meetings with parents, the main questions considering the problem are as follows:

- What communication should be provided between users?
- What information should be displayed for specific user types?
- What is the needed security level of the login system?
- What are the needed information to arrange a meeting?

Extended analysis takes into consideration every issue, but sometimes there are some difficulties that are not possible to overcome and that is why creating a list of delimitations is needed. Due to specific requirements from the company and the lack of time the delimitations are summarized as follows:

- The system will be adjusted only for eNTe's needs.
- The system will not fulfill the requirements of the company with a medium or extra importance, which are:
    - The user must be able to log out from the server
    - The teacher must be able to post discussion topics in chosen classes (one or more)
    - The teacher must be able to post an announcement for chosen classes
    - The teacher must be able to see who has seen a post
    - The student must not be able to see discussions and announcements marked as parental.
    - The system must display all posts marked as important on the top of the list of all posts.
    - The system must send emails regarding posts that are marked as important or parental to all users it concerns
    - The user must be able to comment a discussion that he/she has access to and edit it afterwards
    - The teacher has to have a profile with a name and an email
    - The student has to have a profile with a name, email, class and history of activity, as well as his/her parents' names and emails
    - The system must contain a calendar with meetings*
    - The user must be able to send private messages to other users
    - The user must be able to see private messages sent to and by him/her
    - The teacher must be able to create a chat group from students for a specific homework

- The student must be able to send private messages in chat groups he/she is enrolled in
- The teacher must be able to access every chat group and see the messages in it
- The parent must be able to see posts posted for his/her child and marked as parental
- The system must archive one year back
- The administrator must be able to choose students that have not passed before the beginning of each year.
- The system must change the class of each student that has passed before the beginning of each year

3.1 System analysis

Basing on requirements, one can proceed with analyzing the main parts a system needs. In the case of a system for the school eNTe, it can be seen that a few actors are needed. This is due to having different functionalities depending on the type of a user. The use case diagram below (Figure 2) presents the distinguished actors and the actions each of them can perform. Moreover, the actions are divided into those that are to be fulfilled in this project (yellow background) and those that are a part of the projects delimitations (blue background). Each action is a use case and all use case descriptions of the use cases that are to be realized can be found in Appendix C.

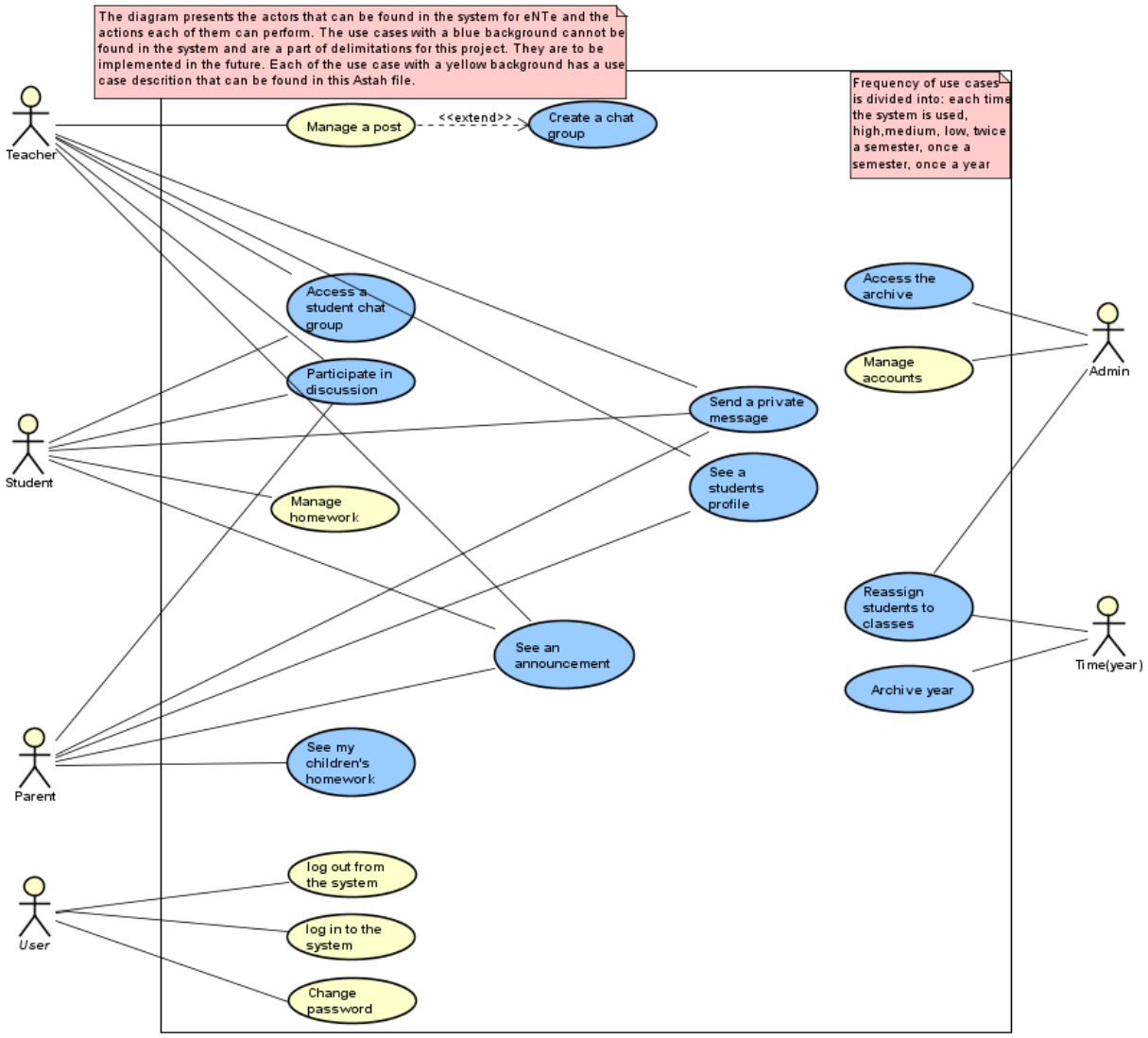


Figure 2

The following analysis of the system has been divided into a few parts. Those are: different user types and managing them, logging in, posts which has a subpart of homework, connection, secondary storage and view.

3.1.1 Different user types and managing them

As it can be seen on the use case diagram above (Figure 2), there are four types of users distinguished in the system: Administrator, Teacher, Student and Parent. Each user type has access to different functionalities of the system. The Administrator is responsible for managing users. The teacher uploads posts (see paragraph 2.2.3). The student can respond to homework uploaded by a teacher (see paragraph 2.2.4). Taking under consideration the delimitations, the parent cannot do anything in the system at the moment. However, in the future the parent is supposed to see posts concerning its children and also see and/or participate in posts dedicated for parents.

This indicates that parents and students are bounded, because a parent must know who its child is (to see posts dedicated specifically for its child) and a student must know its parents (see Requirement number 17). Moreover, a student can have many parents and a parent can have many children. The relations between different user types are shown on the diagram below (Figure 3).

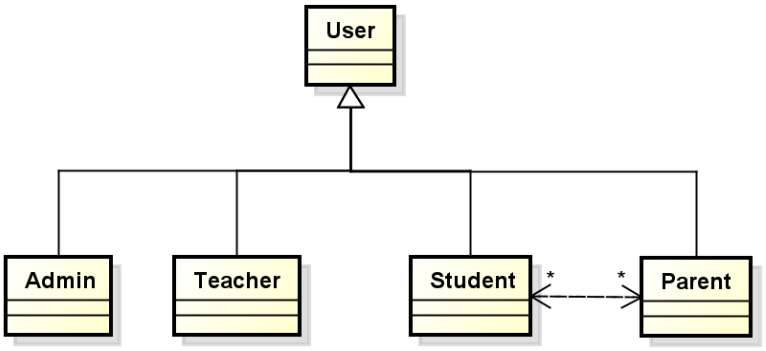


Figure 3

This kind of a relation between a Student and a Parent has a multiplicity many to many, which is a sign of a bad design. That could result in problems while adding/deleting/updating users. While making a change in one student/ parent, it would have to be changed in all related users as well. To deal with this problem, Family has been introduced. A Family stores information about connections within Parents and Students. As a result, the user must know only its Family to know the other Family members. When changes are made, they are made only within a Family, there is no longer a need to make changes in all family members. The solution is illustrated by Figure 4.

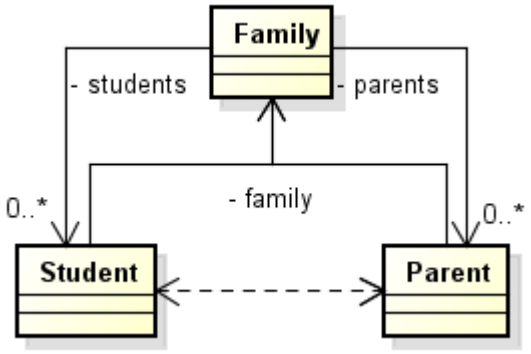


Figure 4

The only user that is authorized to create, edit and delete users in the system is the Administrator. In the application can be created 3 types of users: Student, Parent and Teacher. In the case of creating students and parents, the administrator has to create a new family to assign students and parents to it.

3.1.2 Logging in

Considering requirements 16 and 17 each user should have a unique profile (parents inside their children's profiles). In order to identify and distinguish specific users, a logging in system is needed. In the process of logging in, a user must enter his/her email (login) and password.

When a user account is created, its password is generated randomly and is temporary. That password is sent to the email specified during the creation. When a user wants to log in for the first time, he/she is asked to change the temporary password to his/her own.

When the user forgot the password, his/her password is changed to a generated one and sent by email. During the next login, the user is asked again to change the password.

In order to secure the stored passwords, they are encrypted using SHA-256 asymmetric hash function. Because of this, in the case of a data leak of an unauthorized access to the database, the attacker can only see hashes, which cannot be used to logging into the system.

Another aspect of the login process is that the client application should receive information about the current state of the model. Data about users, families, posts must be delivered with so called “WelcomingData” object (elaborated in 2.2.4). The process of logging in is presented on the activity diagram below (Figure 5).

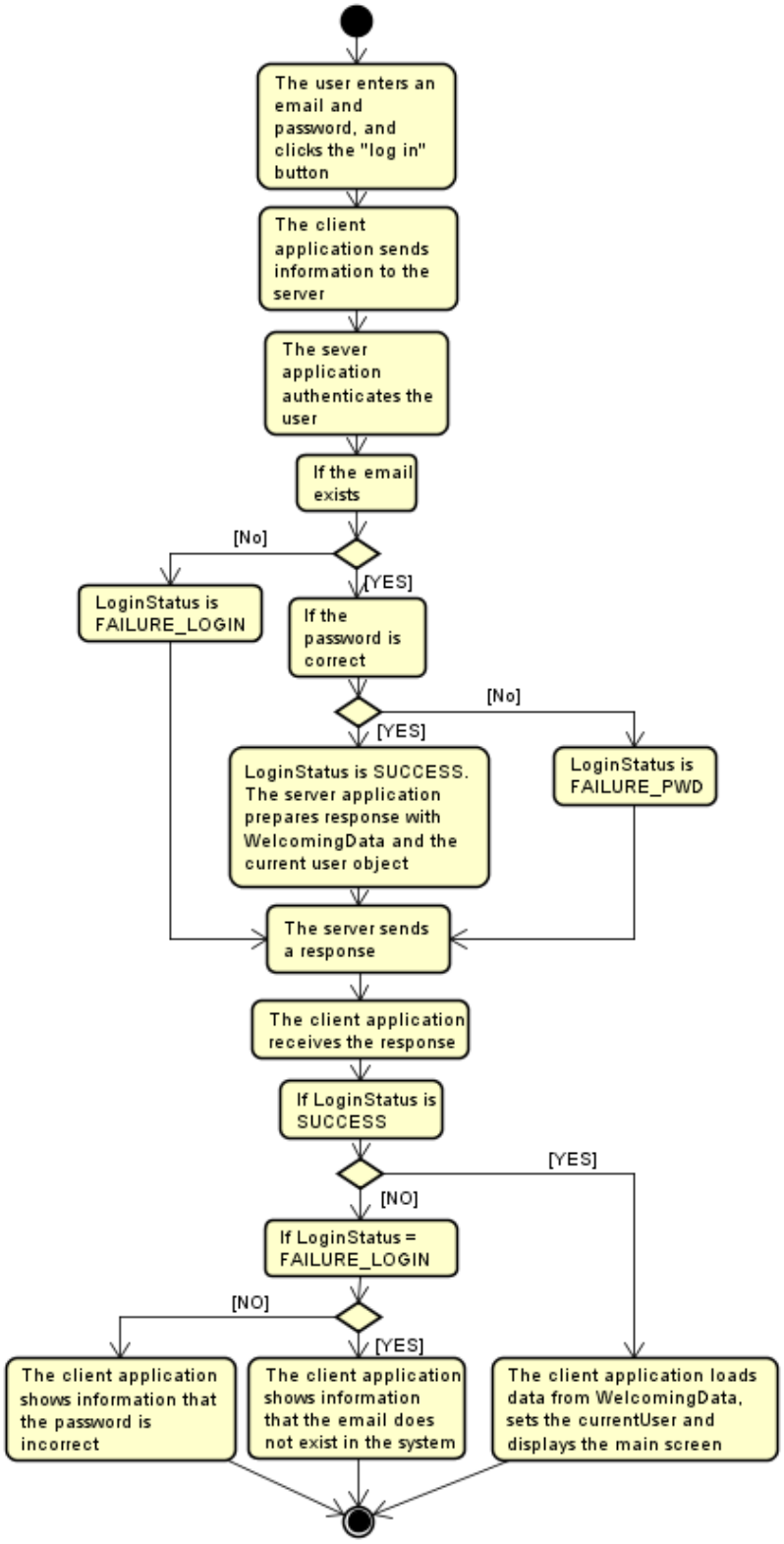

The user enters an email and password, and clicks the "log in" button
The client application sends information to the server
The sever application authenticates the user
If the email exists
[No]
[YES]
LoginStatus is FAILURE_LOGIN
If the password is correct
[No]
[YES]
LoginStatus is SUCCESS. The server application prepares response with WelcomingData and the current user object
LoginStatus is FAILURE_PWD
The server sends a response
The client application receives the response
If LoginStatus is SUCCESS
[YES]
[NO]
If LoginStatus = FAILURE_LOGIN
[NO]
[YES]
The client application shows information that the password is incorrect
The client application shows information that the email does not exist in the system
The client application loads data from WelcomingData, sets the currentUser and displays the main screen

Figure 5

3.1.3 Posts

Another crucial part of the system are posts. A post is a field that contains information that is displayed for chosen users. Each post has a title, some content, an author, which is the name of the teacher that has uploaded the post, and the date of publication. The requirements (requirements number 6-8) state three different types of posts: homework, discussion and announcement. Besides the listed elements, each of them differs, has an other function and may have more fields unique for the type. The relation between them can be seen on Figure 6.

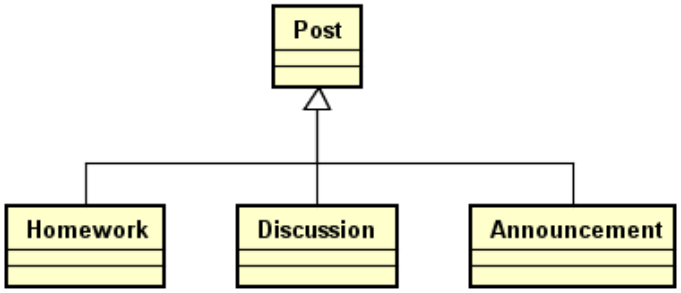


Figure 6

Bearing in mind the delimitations, the focus is only on Homework, which is described more in detail in the following part.

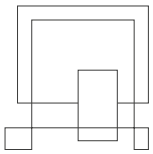
3.1.3.1 Homework

One of the types of posts is homework. A homework is an assignment for students either individual or done in groups. Beyond the elements enumerated in post, it also gives information about the number of students to deliver it together, a deadline and sometimes extra information. In the case of a homework, the content is criteria. To a homework each student that it concerns has to upload a solution before the deadline. If homework is uploaded after the deadline, it is marked as late. The uploaded solution can be seen only by the student who has uploaded it and the teachers. In the current version of the system, the parent cannot perform any actions associated with homework.

3.1.4 Connection

In the system, the communication between the client and the server application is provided by Java Sockets. This solution has been chosen over the RMI API, because the server application is used mostly as a source of data and does not provide any server-side-only operations or services.

The communication proceeds as follows. The client during starting the application makes a connection with the server. The user logs in and the client sends an authentication message to the server. If the user is authenticated correctly, the server replies with a message containing the “WelcomingData”, which involves information about the current state of users, families and posts in the system. When a change is made in the client application, information about what has been changed is sent to the server. Other users are notified about the change if needed.



Objects are sent through sockets using Java Object Serialization. The object that is being sent is an Message instance. It contains the type of the message and data, which depends on the type.

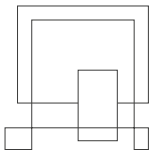
3.1.5 Secondary Storage

Another part of the system that requires specific analysis is the secondary storage. In order to make the system maintainable, there is a need to create a database for storing data from the system about users and posts. The list of eNTe's business rules is derived from the functional requirements, such that the exact tasks for the database are following:

1. For every of 4 types of users (Administrator, Teacher, Student and Parent) eNTe wants to keep track of the user's id, name, email and password. Moreover, it is required to store the value of whether the user's password is temporary or not, it means whether it is needed to be changed (see paragraph 2.2.2).
2. For every student and parent it is needed to keep track of the family that he/she belongs to. Student or parent can belong only to one family and a family can consist of multiple students (children) and parents.
3. For every student, eNTe needs to hold the student's current class. Therefore, a student can only be a part of one class and a class can consist of multiple students.
4. For every post eNTe wants to keep track of post's id, title, content, name of the author and date of publication.
5. For every homework, which is a specific type of post, it is needed to store the number of students that can work on it together, the date of deadline, the list of classes that it belongs to and the value of whether the homework is already closed or still open for submitting solutions.
6. For every homework reply (student's solution) the system needs to hold the id of the homework that it corresponds to, the id of the student that it is elaborated by, the date of hand in, the solution and the value of whether the solution was submitted after the deadline or not.

3.1.6 View

The view has been written in **JavaFX**. Compared to Swing library, the JavaFX application is easier to maintain. The design of the application can be easily done thanks to scene builder which generates an FXML document. There is no need to code component's positions anymore, one can just drag and drop them. CSS can be used for changing the style of components.



4 Design

Having done analysis, one can proceed with designing the system. This is the part where specific classes are distinguished and relations between them are being specified. In this part of the working process, the analysis of the problem statements and requirements is converted into an overview of the whole system. This is the last step before formulating the code.

4.1 Architecture

The base of a successful system is a well-designed architecture. As it is a skeleton for all features of the system, it must follow certain rules. The most important are two SOLID rules, Open-Close Principle, as the once chosen architecture should not be easily modified, and Dependency Inversion Principle, so that different parts of the system are loosely coupled and can be easily exchanged.

Moreover, the whole system has been divided into 4 main parts (packages): client, server, model and utility. The client and server packages are containing classes of respectively the client and the server applications. The common part within them is the model, where the whole domain is described. The utility package contains classes such as SendEmail or Password, which have functionalities not associated with neither the model, client nor server applications.

All those parts, besides the utility one, are described more in detail further in the document.

4.1.1 Client-Server

The system must allow interactions between multiple users. That is why Client-Server architecture has been chosen. Users are connecting to the server, which provides all data stored in the system. When a change is made by a client, the information about it is sent to the server, which notifies other users and updates their data.

4.1.2 Model-View-Controller pattern

The Model-View-Controller pattern has been chosen to be the base of the architecture of both the client and the server application. That means that these parts should contain three inner parts: model, view and controller.

The first component, the model, has been detached as an independent part. That is due to the fact, that it is one of the biggest parts of the system and both the client and server are in need of the same model. The model contains all classes describing the logic of the system, among others the types of users and classes needed to manage homework.

The view segment on the client side contains all classes needed for creating a user interface. It is developed further in paragraph 3.7- View. On the other side, the server application does not fully follow the model view controller rules, as it does not provide an user interface for the server. However, it has another part instead called persistence. The persistence provides a connection to the database. It is described more in detail in paragraph 3.6- secondary storage.

Last but not least is the controller. The main aim of the controller is to ensure a connection between the rest of the parts of the system.

An illustration of the model-view-controller can be seen on the class diagram in Appendix C.

That approach allows to develop parts of the system independently. What that indicates is each component of the application being able to be easily replaced with another one, which fulfills some specific rules (ex. implements a required interface). Another advantage is a rapid development of the system. Separate parts of the system can be developed parallel without appearance of merge conflicts.

4.2 Singleton

The purpose of a singleton is to ensure that there is only one instance of a chosen class in the whole system as well as providing a global access to it. That is what the ClientController class needs. As mentioned before, the ClientController is a class that controls the client side of the system and provides a connection between the model and the view. These are the reasons why only one instance of it should exist and why a global access from other classes is needed. The class is illustrated on Figure 7.

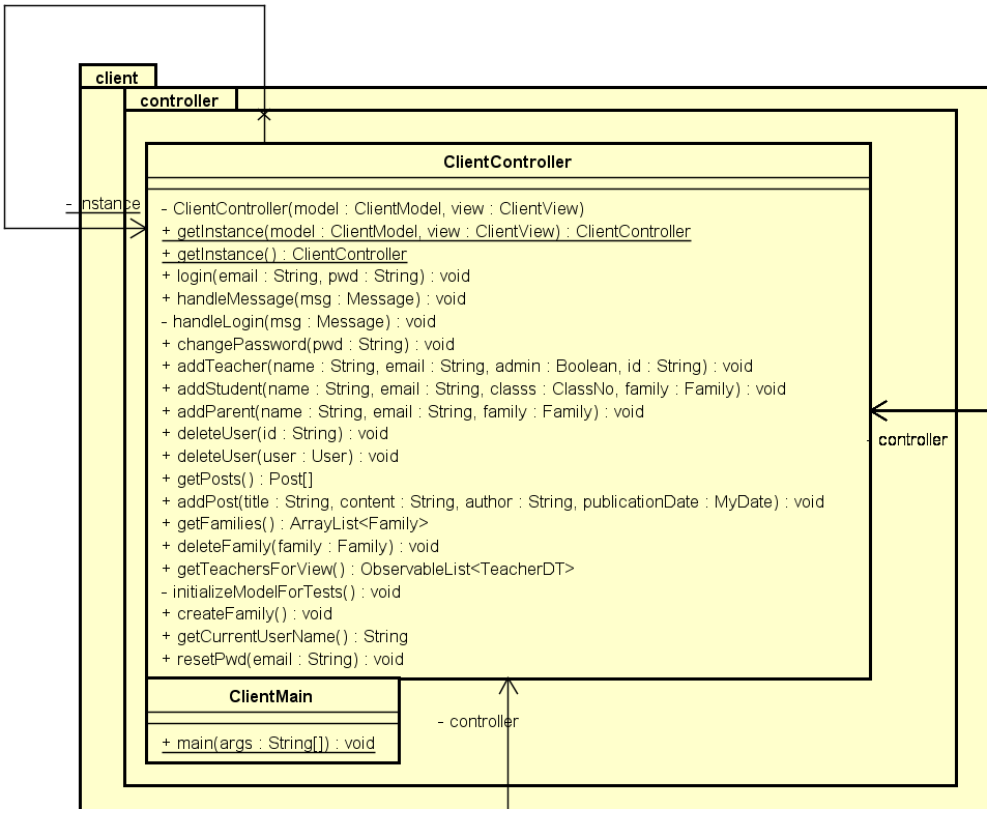
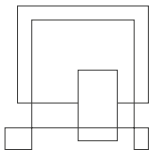


Figure 7



4.3 Managing users

4.3.1 The builder pattern

At some point of developing the business model, the system ended up with a few separate ways of creating users, especially Parents and Students. For example the way how the user is created in the view (specifying the ID is not needed) is much different from the one used when retrieving data from a database (the User must be created with a specific ID and be assigned to a correct family). This resulted in a situation where all User subclasses had three or more constructors and it was confusing which one should be used.

The chosen solution to the problem is the Fluent Builder API (Per Lundholm, 2013). The idea behind the Builder pattern is to provide API for a class with a complex instantiating process in order to make it easier. This can be achieved by implementing inside the target class an inner static final class Builder, with builder methods which are preparing the object. Then in the target class, a static method which creates and returns an instance of the builder is needed. However this is not the original Builder pattern described in the book ***Design Patterns: Elements of Reusable Object-Oriented Software***. There, the Builder is used to separate the construction of a complex object from its representation. What makes Fluent Builder special is clever usage of interfaces for Builder.

It can be shown on the example of the Student class. However, in all users classes builders are designed in a similar way.

The Student object has three fields (name, email, class) that are necessary to be specified during the creation of the object. They cannot be empty/null! The rest of the fields are optional (family, password, id, historyOfActivity), they can be specified later, or created inside a Student. Beside the Builder inner class, four interfaces are defined in the Student class. Three of them are responsible for the necessary fields, while the last one specifies that the object can be build.

All of those interfaces are creating a chain of methods, that allows to specify each needed field of a Student and also set only the wanted optional fields. Figure 8 shows the used interfaces.

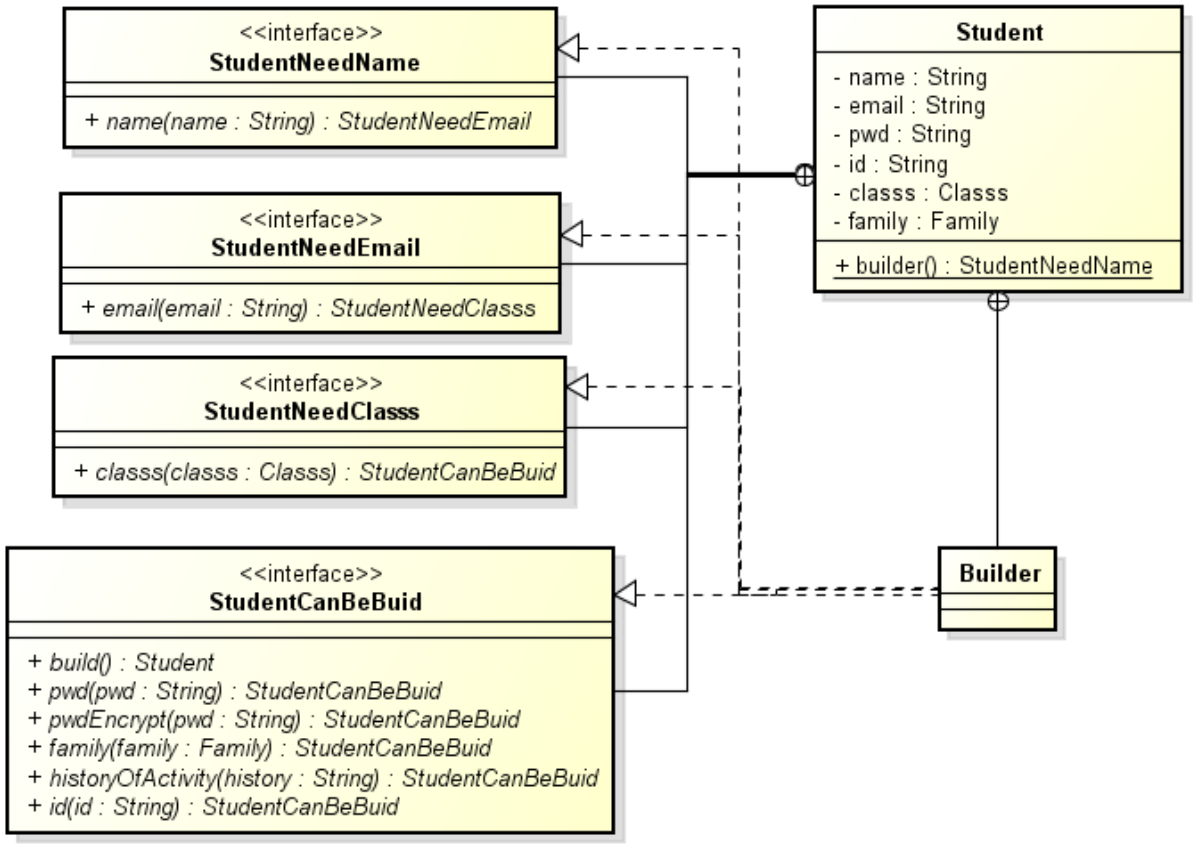


Figure 8

The flow of the Builder can be illustrated by a situation where a programmer, who was not present during the implementation of the Student class, wants to create a Student. The constructor for a Student is not accessible, but the static method Student.builder() is available. This method returns an object of Builder, however it is of the type StudentNeedName and right now, the only accessible method to call is name(). name() returns the type StudentNeedEmail, thus he has to call email(), and after this the only option is classNo() (thanks to StudentNeedClassNo). At this point, the programmer has reached the point, where he has specified all the necessary fields of the Student, and the object can be created. To ensure that the object can actually be created and does not contain any nulls, the build method throws an IllegalStateException when a passed argument is null. The method classNo() returns the type StudentCanBeBuild and afterwards the method build(), which returns the desired Student, is accessible. Eventually, he can also specify the id, family and password.

Thanks to this approach, there is no way to create an incomplete object. Moreover, the programmer is guided in how to make a Student. He does not have to know how many fields he must specify, which are important and must be given. There is even no need to remember the order of parameters, because of the clever use of interfaces and return types.

4.4 Connection

The connection in the system has been made with the use of sockets. The object that is serialized and sent through the socket connection is a Message. The message has a type and a data object, which depends on the type. There are eleven different types: Login, Auth, NoType, CheckE-mail, EmailStatus, ChangePwd, ManageFamily, ManagePost, ManageUser, Fail, Success. Each of them is responsible for a different activity and has a corresponding class, which encloses the data object and provides basic operations. For example the ManageUser class has three public static final fields: “ADD”, “DELETE”, “EDIT”, which are used as a value for the “action” field. Moreover, the ManageUser class is also used to hold the user object.

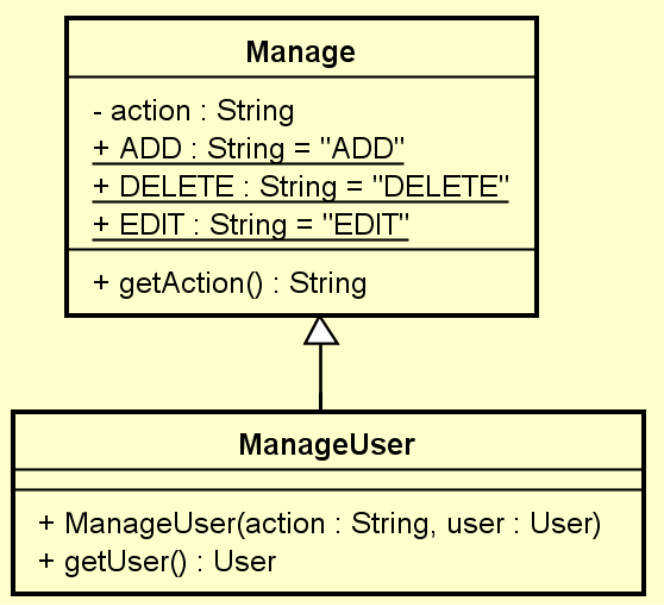


Figure 9

The communication with Sockets is accomplished by the means of two connection proxy classes and a Message class. The ClientProxy and the ServerProxy are responsible for sending and receiving Messages. The ServerProxy class listens for incoming clients connections. When a connection is established, a socket is passed into the HandleClient class, which in a new thread communicates with the client. Both of those classes are associated with corresponding Controllers, which handle each message or reply. The described classes are shown on Figure 10.

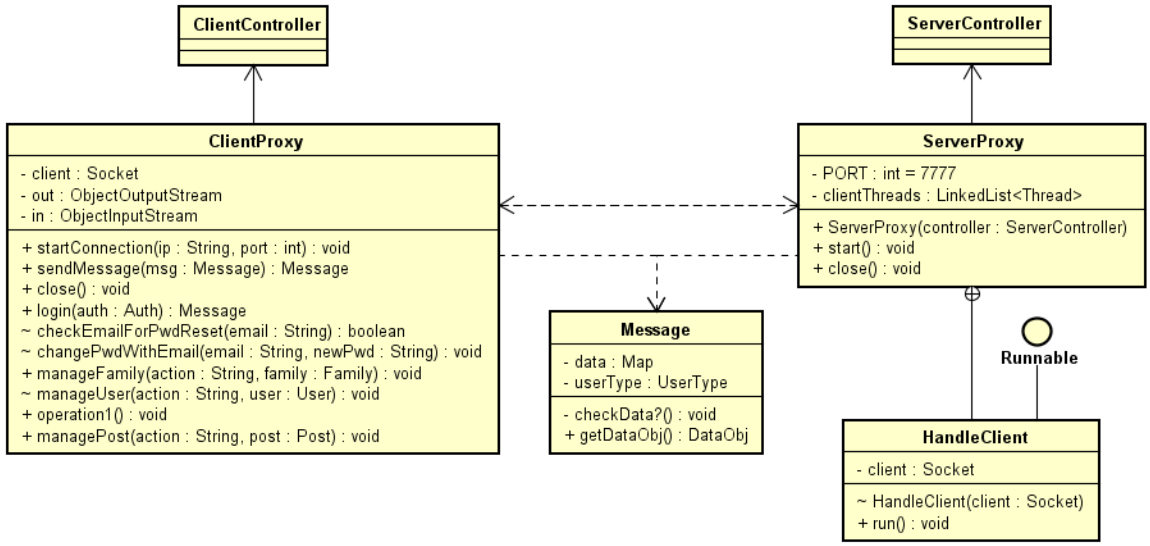


Figure 10

4.5 Logging in

The logging in process starts when a user has entered his/her email and password on the login screen and clicked the “log in” button. The handler for the “log in” button collects values from fields and calls the “login(email, password)” method in the ClientController, which passes it to the ClientModelManager and further to the ClientProxy. There, the Message object containing the Auth object (a wrapper for the email and password), is prepared and sent to the ServerProxy.

At the server side, the ServerProxy receives the Message and passes it to the ServerController calling handleMessage(). The ServerController passes it to the ServerModelManager to check the email and password in the list of users. Basing on the result of the authentication, a response in the form of a Login object (containing the LoginStatus, WelcomingData and an object of the User that has logged in), can be created in three ways, depending on the LoginStatus returned from the model. If the user exists and its password is correct, the Login object is made with WelcomingData and an object of the user that has logged in. If the email does not exists in the system, the Login object is made with the LoginStatus set to FAILURE_LOGIN. In the case when the email exists, yet the password is incorrect, the LoginStatus is set to FAILURE_PWD. The Login object is packed into a Message and sent back to the ClientProxy, which passes it to the ClientController. Depending on the LoginStatus, the ClientController is showing either the main application screen (if the LoginStatus is SUCCESS), or shows information that the email does not exist (FAILURE_LOGIN) or that the password is incorrect (FAILURE_PWD).

A sequence diagram which shows this process in details can be found in Appendix C.

4.6 Secondary storage

4.6.1 Database design

The design of the database requires the elaboration of the EER diagram in order to avoid having complicated and inconsistent design of the database. All diagrams concerning the database design can be found in Appendix C.

4.6.1.1 EER diagram

The EER diagram below presents a possible solution for tasks mentioned in the paragraph 2.2.5.

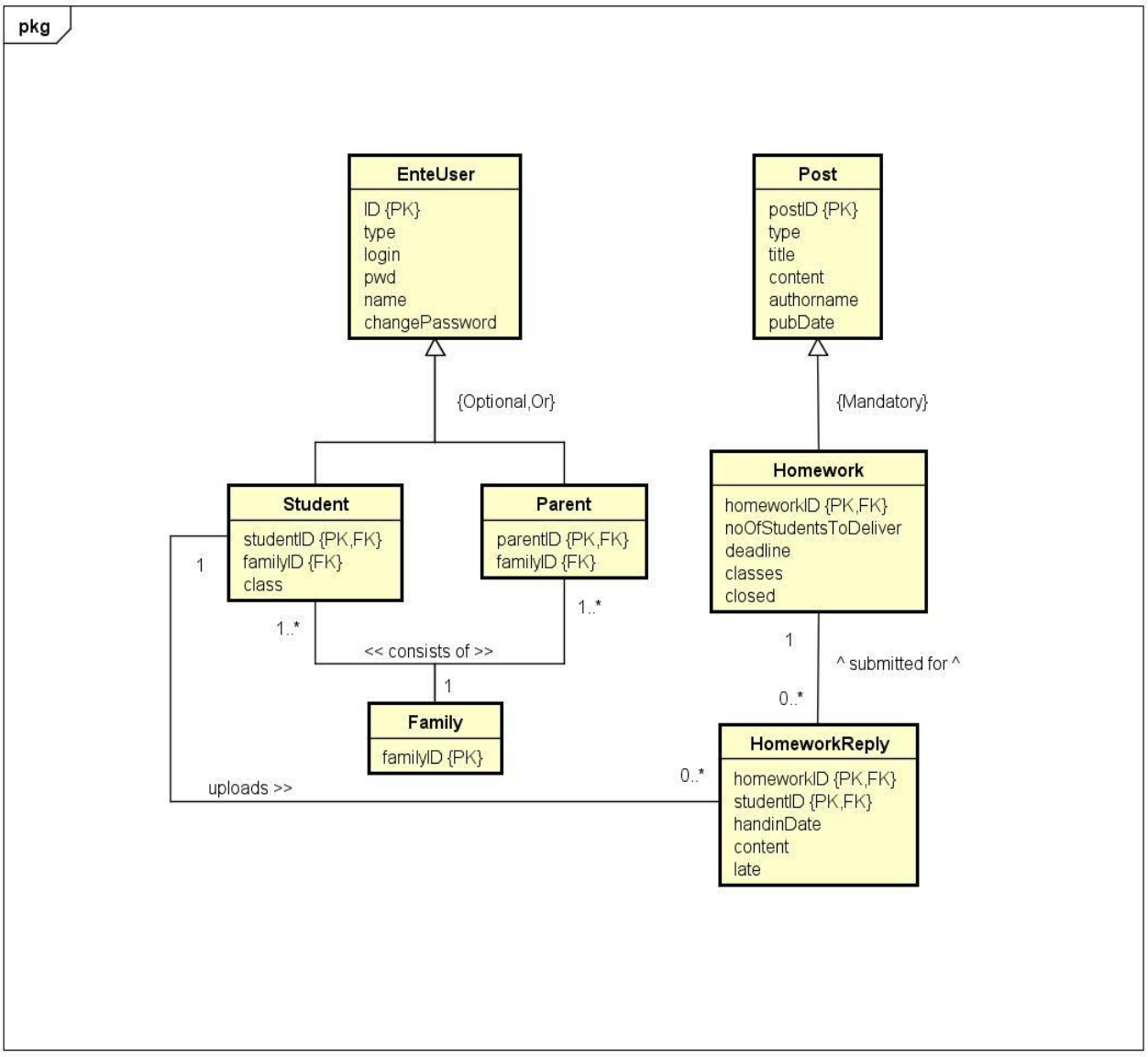
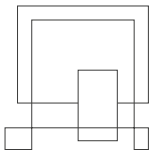


Figure 11



At the moment, there is no need to make Teacher or Administrator as an entity as they do not hold any specific data in comparison with the enteuser. Therefore, only Student and Parent are made as separate entities that hold specific data about family and also about class in the case of a student.

One of the biggest issues regarding the database model is the many to many relationship. This kind of a relationship can be found in this project between Student and Parent entities. As the Family has been introduced in the model of the system (see paragraph 2.2.1), the Family entity is supposed to solve this problem in the database model as well. The family entity holds just the family id which can represent the column referenced from both Student and Parent entities.

As for the part of the database regarding posts and replies, in the actual version of this project is provided only one type of post - homework. At the moment it is not needed to make two separate entities for Post and Homework with a superclass/subclass relationship between them as the participation constraint is mandatory. Nevertheless, this kind of superclass/subclass relationship has been chosen for entities Post and Homework because of the further system development that is supposed to operate with more types of posts, as for example discussions and announcements.

The HomeworkReply entity is the one that connects two parts of the database, one that controls users and another one that controls posts, in the way that it contains references to both the Student and Homework entities.

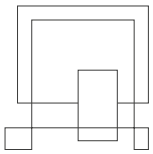
4.6.1.2 Domain table

The table below shows the attribute's, name, meaning and definition of each of required domains.

Attribute	Domain Name	Meaning	Domain Definition
ID, postID, homeworkID, studentID, parentID, familyID	IDs	The set of all possible ids for all kinds of objects	character : size 36
class	enteClasses	The set of all possible values of the eNTe's classes	character: size 10, values : First, Second,Third, Fourth, Fifth, Sixth, Seventh, Eighth
pubDate, deadline, handinDate	dates	The set of dates represented as timestamp	timestamp
email	emails	The set of all possible emails	character: size 80
type	postTypes	The set of possible post types	character: size 20, values: Homework, Announcement, Discussion
pwd	pwds	The set of possible passwords	character: size 64
usertype	userTypes	The set of all possible types of eNTe users	character: size 15 values: Administrator, Teacher, Student, Parent
classes	classesForHomework	The set of arrays of eNTe's classes	array of characters: character size 7

Table 1

Even though the current state of the system provides only posts of the type Homework, this table indicates that the domain with the name "postTypes" can contain 3 different values - Homework,Announcement,Discussion, this is due to the further development.



4.6.1.3 Logical data model

The following logical data model shows the primary and foreign keys for the given entities.

EnteUser (ID,usertype,email,pwd,name,changePassword)
PK: ID

Family (familyID)
PK: familyID

Student (studentID,familyID,class)
PK: studentID
FK: studentID REFERENCES EnteUser (ID)
FK: familyID REFERENCES Family (familyID)

Parent (parentID,familyID)
PK: parentID
FK: parentID REFERENCES EnteUser (ID)
FK: familyID REFERENCES Family (familyID)

Post (postID,type,title,content,authorName,pubdate)
PK: postID

Homework (homeworkID,noOfStudentsToDeliver,deadline,classes,closed)
PK: homeworkID
FK: homeworkID REFERENCES Post (postID)

HomeworkReply (homeworkID,studentID,handinDate,content,late)
PK: homeworkID,studentID
FK: homeworkID REFERENCES Homework (homeworkID)
FK: studentID REFERENCES Student (studentID)

In the logical data model can be seen the relations between different entities in the following way.

- EnteUser is a parent entity for both Student and Parent where the key is the user's id.
- Family is a parent entity for both Student and Parent where the key is the family's ID.
- Post is a parent entity for Homework where the key is post's ID
- HomeworkReply is a child as of Homework (referencing by homework's ID) as well as of Student entity (referencing by student's ID)

4.6.1.4 Database visualization

The figure below represents the visualization of the database system based on the EER diagram, domains table and the logical data model. There can be seen relations between all tables, primary and foreign keys, columns of all tables and also types of all columns.

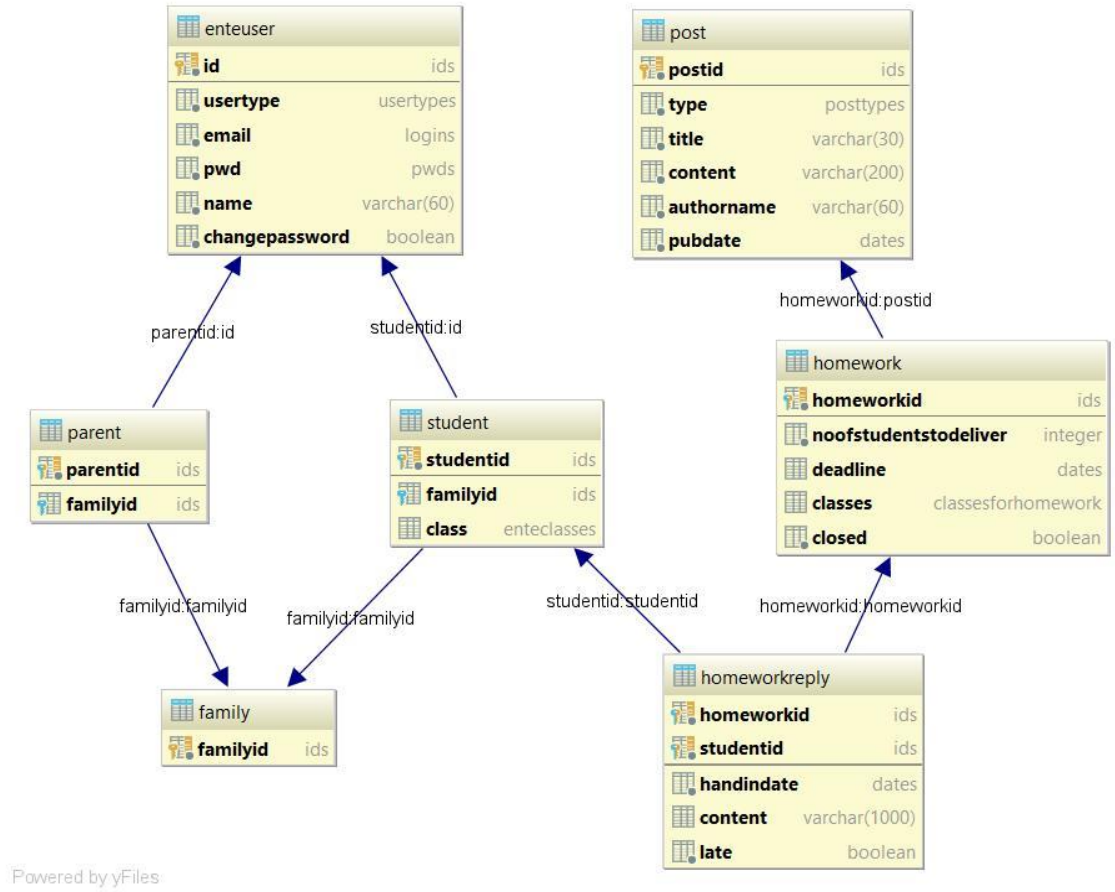


Figure 12

4.6.2 Adapter pattern

The adapter design pattern has been chosen for transforming data to and from the database. The main reason for using this design pattern is making the system clearer, dividing responsibility and providing a possibility to change the database without having to make any changes in other parts of the system.

It is implemented in the following way. The adapter class (DBAdapter) implements the interface DBPersistence that is like a target that holds methods needed for the model on the server's side. Therefore the ServerModelManager has no direct relations to the database (SQL language). Moreover, the Dependency Inversion Principle is applied to this part of the system by implementing this design pattern, such that ServerModelManager does not depend on a lower-level-modul(DBAdapter), but on an abstraction(DBPersistence) that is implemented by that lower-level-modul.

The SQL strings are created by methods in the DBAdapter class, which is responsible for calling these SQL statements on the Database class that implements the DBInterface. This interface represents the Adaptee in the adapter design pattern, such that each class, that implements this interface, is responsible for accessing data from a specific kind of database that it is made for. In the case of this system, the class that implements DBInterface is the Database class that provides the communication between the system and the PostgreSQL database system.

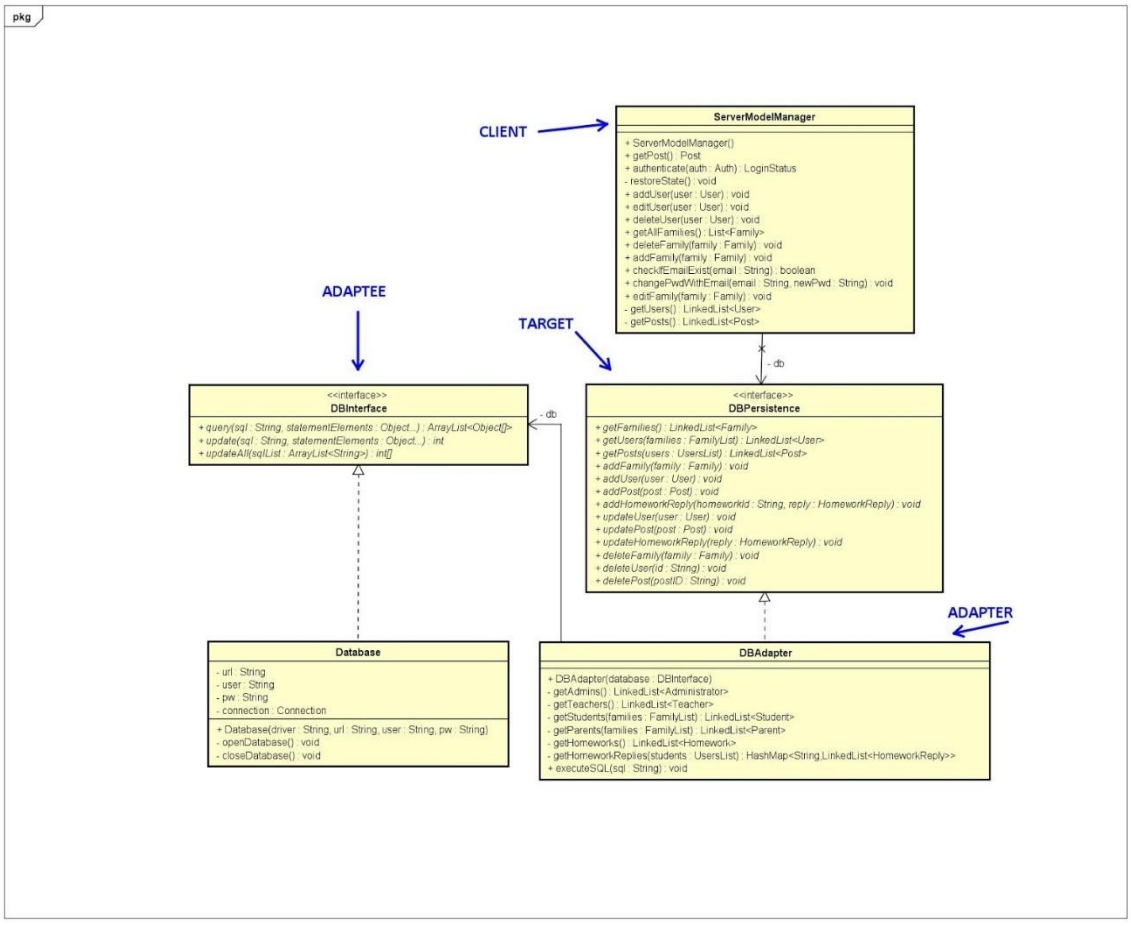


Figure 13

4.7 View

The graphical design of the ente application is based on the customers vision. The customer wish was to keep a simple design, without many details on it.

Figure 14

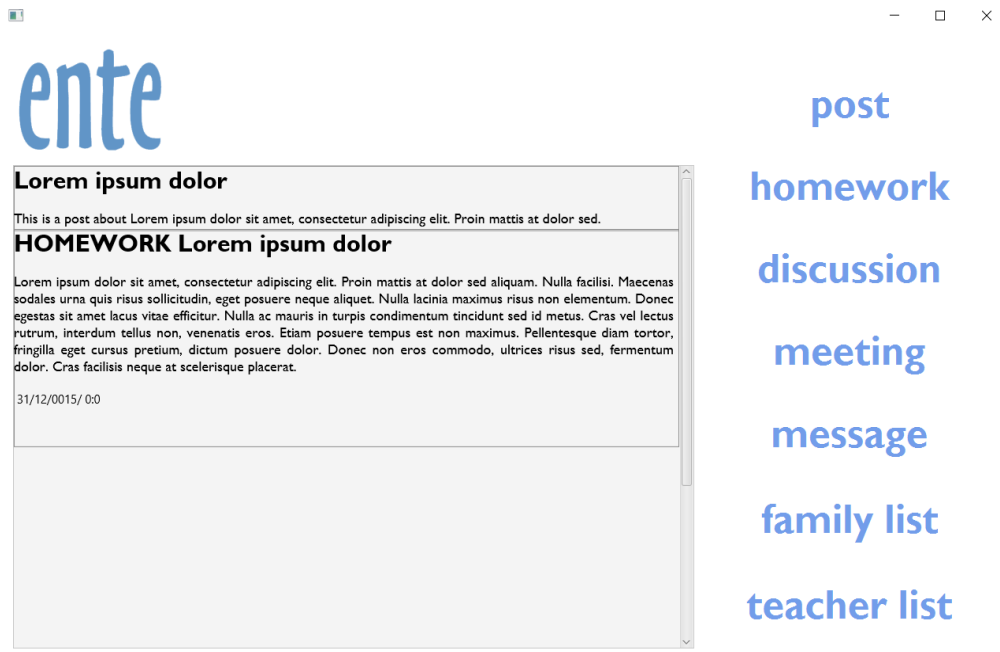


Figure 15

The connection between classes can be seen in the diagram below. It takes just a few clicks to get anywhere in the system. The main functionalities of the system can be found on the home page(AdminMainHandler or StudentMainHandler). From there, the user(admin) can access the family list or teacher list and go to creation forms for new users. Moreover, he can edit or delete users. In the case of creating homework, admin has access to the HomeworkHandler, which leads him to the crea-

tion form for new homework. Editing and deleting homework can be found in the same place. Student has access to homework from the home page. A user guide describing how to use the system has been attached in Appendix D.

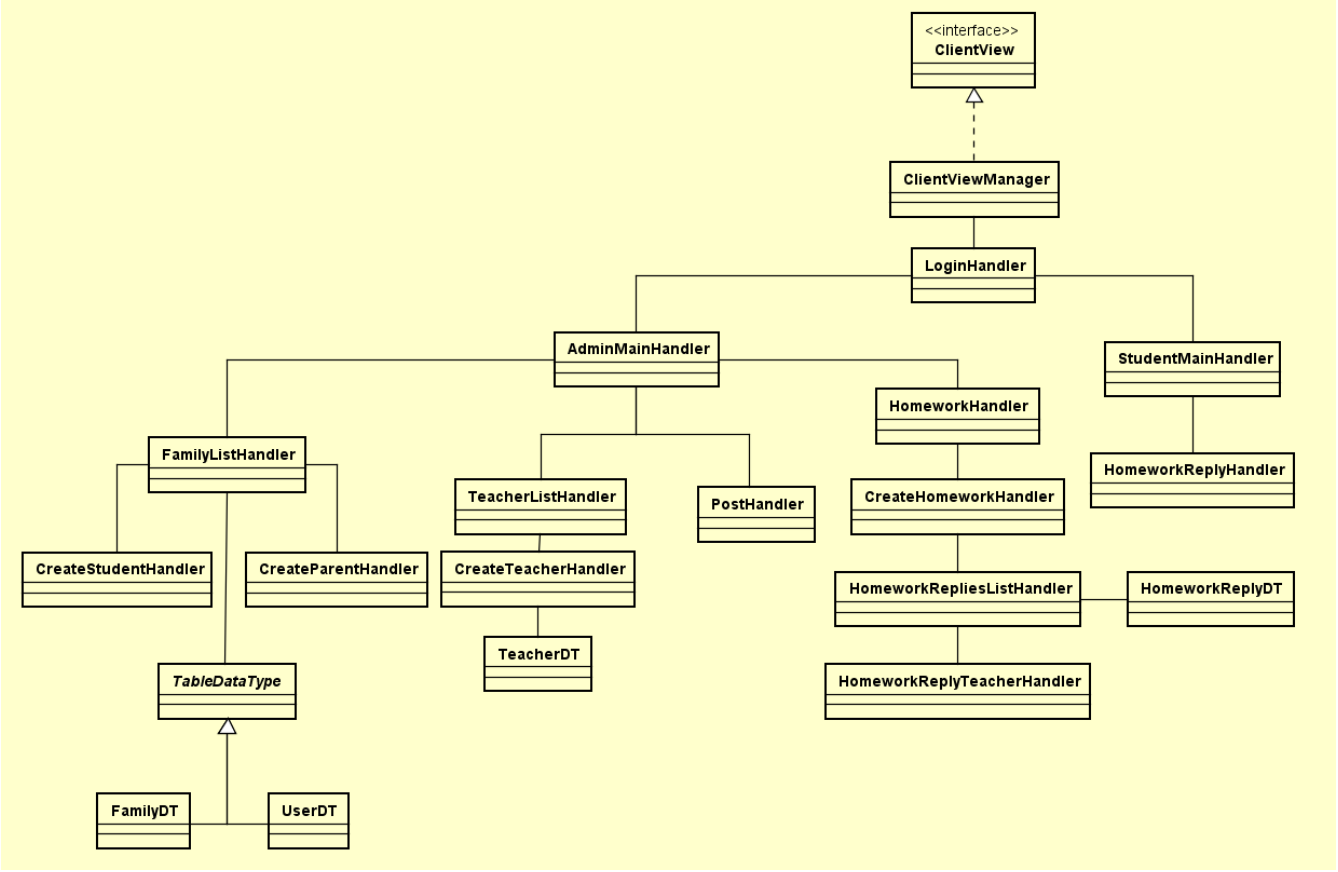


Figure 16

5 Implementation

Basing on analysis and design a code can be formulated. The following part shows the most interesting parts of the implementation. However, the whole source code can be found in Appendix E. Moreover, the deployed system in the form of jar files can be found in appendices F and G.

5.1 Connection to the database

The connection between Java and the database is established by using standard Java API - JDBC (Java Database Connectivity). The latest version of JDBC Driver (PostgreSQL JDBC 4.2 Driver, 42.2.2) is imported to the system in order to access the database's data that is placed on eNTe's private server with IP address - 207.154.237.196. The source SQL code of the database generated by DataGrip can be found in the Appendix H. The way how the Adaptee - Database class connects to get and update the data stored in the database is as follows.

When the object of the Database class is created, String fields (url, user, pwd) are set by constructor's parameters, the connection is set to null and the driver is loaded from the imported jar file that contains the current version of the JDBC Driver.

```
public Database(String driver, String url, String user, String pw)
        throws ClassNotFoundException {
    this.url = url;
    this.user = user;
    this.pw = pw;
    connection = null;
    Class.forName(driver);
}
```

Figure 17

The connection is established in the "openDatabase()" method. Thanks to the DriverManager class the connection is set by using the url, user and password.

```
private void openDatabase() throws SQLException {
    connection = DriverManager.getConnection(url, user, pw);
}
```

Figure 18

Once the connection is set up, the system can interact with the database by calling the overridden methods from the DBInterface. Having called one of those overridden methods, the connection to the database has to be closed by calling the "closeDatabase()" method.

Code snippets below represent the example of how the system gets desired data currently stored in the database. Firstly, the Database class transforms data (by calling the query() method) received as an object of the ResultSet interface into an ArrayList<Object[]> object. As the ResultSet object maintains the cursor in the current row, it is creating an array of objects from the current row and then the next() method is used for moving the cursor. This is how the data is transformed into Java objects.

```
@Override
public ArrayList<Object[]> query(String sql, Object... statementElements)
        throws SQLException {
    openDatabase();

    PreparedStatement statement = null;
    ArrayList<Object[]> list = null;
    ResultSet resultSet = null;
    if (sql != null && statement == null) {
        statement = connection.prepareStatement(sql);
        if (statementElements != null) {
            for (int i = 0; i < statementElements.length; i++)
                statement.setObject( parameterIndex: i + 1, statementElements[i]);
        }
    }
    resultSet = statement.executeQuery();
    list = new ArrayList<Object[]>();
    while (resultSet.next()) {
        Object[] row = new Object[resultSet.getMetaData().getColumnCount()];
        for (int i = 0; i < row.length; i++) {
            if (resultSet.getMetaData().getColumnType(i+1) == Types.ARRAY) {
                row[i] = resultSet.getArray( columnIndex: i+1).getArray();
            }
            else {
                row[i] = resultSet.getObject( columnIndex: i + 1);
            }
        }
        list.add(row);
    }
    if (resultSet != null)
        resultSet.close();
    if (statement != null)
        statement.close();
    closeDatabase();
    return list;
}
```

Figure 19

Secondly, the DBAdapter class needs to transform the ArrayList<Object[]> objects into the needs of the model. In the snippet below can be observed how the transformation to the model's objects is done.

```
private LinkedList<Homework> getHomeworks(HashMap<String, LinkedList<HomeworkReply>> replies) {
    LinkedList<Homework> list = new LinkedList<>();
    try {
        String sql = "SELECT p.postid, p.title, p.content, p.authorname, p.pubDate," +
                " h.noOfStudentsToDeliver, h.deadline, h.classes, h.closed FROM Post p," +
                " Homework h WHERE p.postid=h.homeworkid ORDER BY p.postid";
        ArrayList<Object[]> resultSet = db.query(sql);
        for (Object[] e : resultSet) {
            String postID = (String) e[0];
            String title = (String) e[1];
            String content = (String) e[2];
            String authorName = (String) e[3];
            Timestamp pubDateStamp = (Timestamp) e[4];
            int noOfStudentsToDeliver = (int) e[5];
            Timestamp deadlineStamp = (Timestamp) e[6];
            String[] classesString = (String[]) e[7];
            List<ClassNo> classes = new ArrayList<>();
            for (String a : classesString) {
                classes.add(ClassNo.valueOf(a));
            }
            boolean closed = (boolean) e[8];
            list.add(new Homework(postID, title, content, authorName,
                    MyDate.convertFromTimestampToMyDate(pubDateStamp),
                    MyDate.convertFromTimestampToMyDate(deadlineStamp),
                    classes, noOfStudentsToDeliver, replies.getOrDefault(postID,
                    new LinkedList<>()), closed));
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return list;
}
```

Figure 20

The transformation is mostly done by casting the objects into the desired ones, but in the case of the Timestamp, a static method implemented in MyDate class that converts Timestamp expressions into MyDate objects is needed.

5.2 Encryption

To secure stored passwords, they are encrypted right after the creation. The used encryption is SHA-256 hash function, which is implemented in the Java library (java.security.MessageDigest). As the digest() method takes an array of bytes as a parameter, conversion is needed. A hexadecimal string has been chosen to store the passwords in the database. A code snippet of encryption and conversion function is shown below (Figure 21).

```java
public static String encryptPwd(String pwd) {
    MessageDigest dig;
    String encrypted = "";
    try {
        dig = MessageDigest.getInstance("SHA-256");
        dig.update(pwd.getBytes());
        encrypted = toHex(dig.digest());
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
    return encrypted;
}

private static String toHex(byte[] byteData) {
    StringBuilder hexString = new StringBuilder();
    for (byte aByteData : byteData) {
        String hex = Integer.toHexString( i: 0xff & aByteData);
        if (hex.length() == 1)
            hexString.append('0');
        hexString.append(hex);
    }
    return hexString.toString();
}
```

Figure 21

5.3 Sending emails

The system is sending an information about new/or changed passwords via email. This feature has been implemented using an external Java library, Java Mail API (Oracle, 2017).

In order to send an email, the email server properties are configured for SMTP protocol. Then, a message is created and a connection with a Gmail account is established. After authentication, an email is send. A fragment of the SendEmail class, which contains the generateAndSendEmail() method, is shown below (Figure 22).

```
public static void generateAndSendEmail(String to,String subject, String emailBody) throws MessagingException {
    // Step1 - prepare mail server properties
    System.out.println("\n 1st ===> setup Mail Server Properties..");
    Properties mailServerProperties = System.getProperties();
    mailServerProperties.put("mail.smtp.port", "587");
    mailServerProperties.put("mail.smtp.auth", "true");
    mailServerProperties.put("mail.smtp.starttls.enable", "true");
    System.out.println("Mail Server Properties have been setup successfully..");

    // Step2 - create message
    System.out.println("\n\n 2nd ===> get Mail Session..");
    Session getMailSession = Session.getDefaultInstance(mailServerProperties,  authenticator: null);
    MimeMessage generateMailMessage = new MimeMessage(getMailSession);
    generateMailMessage.addRecipient(Message.RecipientType.TO, new InternetAddress(to));
    generateMailMessage.setSubject(subject);
    generateMailMessage.setContent(emailBody,  type: "text/html");
    System.out.println("Mail Session has been created successfully..");

    // Step3 - get session
    System.out.println("\n\n 3rd ===> Get Session and Send mail");
    Transport transport = getMailSession.getTransport( protocol: "smtp");

    // Step 4 - connect and send message
    transport.connect( host: "smtp.gmail.com",  user: "enteemailservice@gmail.com",  password:              );
    transport.sendMessage(generateMailMessage, generateMailMessage.getAllRecipients());
    transport.close();
}
```

Figure 22

In addition, the SendEmail class contains two more utility methods.

The method sendPasswordEmail(String email) prepares an email about a new password and sends it to the passed email. The implementation is shown on Figure 23.

```
public static void sendPasswordEmail(String email, String pwd) {
    String body = "This is your temporary password to eNTe system: "
            + pwd + "<br>Please reset it during next login to the system." +
            "<br>Regards, eNTe";
    try {
        generateAndSendEmail(email, subject: "Change your password", body);
    } catch (MessagingException e) {
        e.printStackTrace();
    }
}
```

Figure 23

The isValidEmailAddress(String email) method is verifying if the passed argument is an email address. It is using the InternetAddress class from java.mail (Java Mail API). It is shown on Figure 24.

```
public static boolean isValidEmailAddress(String email) {
    boolean result = true;
    try {
        InternetAddress emailAddr = new InternetAddress(email);
        emailAddr.validate();
    } catch (AddressException ex) {
        result = false;
    }
    return result;
}
```

Figure 24

5.4 UUID

Each user and post in the system must be uniquely identified, not only by it's content, as it is possible to create two objects of the same content, but also by an ID number.

To omit the problem of finding the correct format of ID for users and posts and ensure that it is unique in the system, UUID (Wikipedia, 2018) has been chosen. One of the advantages of using it is that generating a new ID is already implemented in Java Library. Another one is that the probability of collision (generating a duplicate ID) is very low. Moreover, it eliminates the problem of inventing special ID formats for this system. However, UUID is not easily readable and can not be used for displaying in GUI.

To generate UUID the system is using java.util.UUID, which provides basic operations on UUID.

```
id = UUID.randomUUID().toString();
```

Figure 25

5.5 Handling messages

Handling messages is a crucial task of the ClientController and the ServerController. These classes are responsible for correctly interpreting Messages objects which are sent between Client and Server applications. Basing on the type of the Message, controllers are deciding what actions should be taken. As an example, when the ClientController receives a Message, according to the Message type, different methods are called.

```
synchronized Message handleMessage(Message msg) {
    Message response;

    switch (msg.getType()) {
        case Auth:
            response = handleAuthentication(msg);
            break;
        case ManageUser:
            ManageUser manageUser = msg.getManageUser();
            response = handleManageUser(manageUser);
            break;
        case ManageFamily:
            ManageFamily manageFamily = msg.getManageFamily();
            response = handleManageFamily(manageFamily);
            break;
        case ManagePost:
            ManagePost managePost = msg.getManagePost();
            response = handleManagePost(managePost);
            break;
        case CheckEmail:
            String email = msg.getEmail();
            response = checkEmail(email);
            break;
        case ChangePwd:
            ChangePwd change = msg.getChangePwd();
            model.changePwdWithEmail(change.email, change.newPwd);
            response = Message.createSuccessfulResponse();
            break;
        default:
            response = Message.createFail();
            break;
    }
    return response;
}
```

Figure 26

When the Message has the type ManageUser, the handleManageUser() method is called, which is applying appropriate changes in the model.

```
private Message handleManageUser(ManageUser manageUser) {
    Message response;
    switch (manageUser.getAction()) {
        case ManageUser.ADD:
            model.addUser(manageUser.getUser());
            response = Message.createSuccessfulResponse();
            break;
        case ManageUser.EDIT:
            model.editUser(manageUser.getUser());
            response = Message.createSuccessfulResponse();
            break;
        case ManageUser.DELETE:
            model.deleteUser(manageUser.getUser());
            response = Message.createSuccessfulResponse();
            break;
        default:
            response = Message.createFail();
            break;
    }
    return response;
}
```

Figure 27

5.6 GUI

Every fxml document has its own handler with a method initialize, which is called by Javafx, to initialized components.

One of the interesting examples is the FamilyListHandler with a TreeTableView, which displays information in form of a table with tree elements. For every column, setting the cell value factory is needed. It specifies the content to be placed into this column for each row. To specify what kind of item will be placed into columns, an abstract class of TableDataType is created, which holds all fields that are meant to be displayed as Strigns. There are two classes which extend this class: UserDT, FamilyDT (and TeacherDT for TeacherListHandler), each for different type of displayed object. The source of information for these classes are user and family objects. The dataForTable() method is responsible for converting lists of families and users into suitable TreeItem objects, which are used in the TreeTableView. Eventually, columns and the table are initialized in the method initialize(). Implementation of those methods can be seen on Figures 28-30.

```
private TreeItem<TableDataType> dataForTable() {
    TreeItem<TableDataType> rows = new TreeItem<>();
    controller.getFamilies().forEach(family -> {
        TreeItem<TableDataType> parents = new TreeItem<>(new UserDT( name: "Parents"));
        parents.getChildren().addAll(family.getParents().stream()
                .map(parent -> new TreeItem<TableDataType>(new UserDT(parent,  classs: ""))).collect(Collectors.toList()));

        TreeItem<TableDataType> children = new TreeItem<>(new UserDT( name: "Students"));
        children.getChildren().addAll(family.getChildren().stream()
                .map(child -> new TreeItem<TableDataType>(new UserDT(child, child.getClassNo().toString()))).collect(Collectors.toList()));

        TreeItem<TableDataType> familyRoot = new TreeItem<>(new FamilyDT(family));
        familyRoot.getChildren().addAll(parents, children);
        rows.getChildren().add(familyRoot);
    });
    return rows;
}
```

Figure 28

```
@FXML
public void initialize() {

    name.setCellValueFactory(new TreeItemPropertyValueFactory<>("name"));
    email.setCellValueFactory(new TreeItemPropertyValueFactory<>("email"));
    className.setCellValueFactory(new TreeItemPropertyValueFactory<>("className"));

    familyTable.setRoot(dataForTable());
    familyTable.setShowRoot(false);
}
```

Figure 29

```java
public abstract class TableDataType {

    protected String name;
    protected String email;
    String className;
    String type;

    String getName() { return ""; }

    String getEmail() { return ""; }

    String getClassName() { return ""; }

    String getType() { return ""; }
}
```

Figure 30

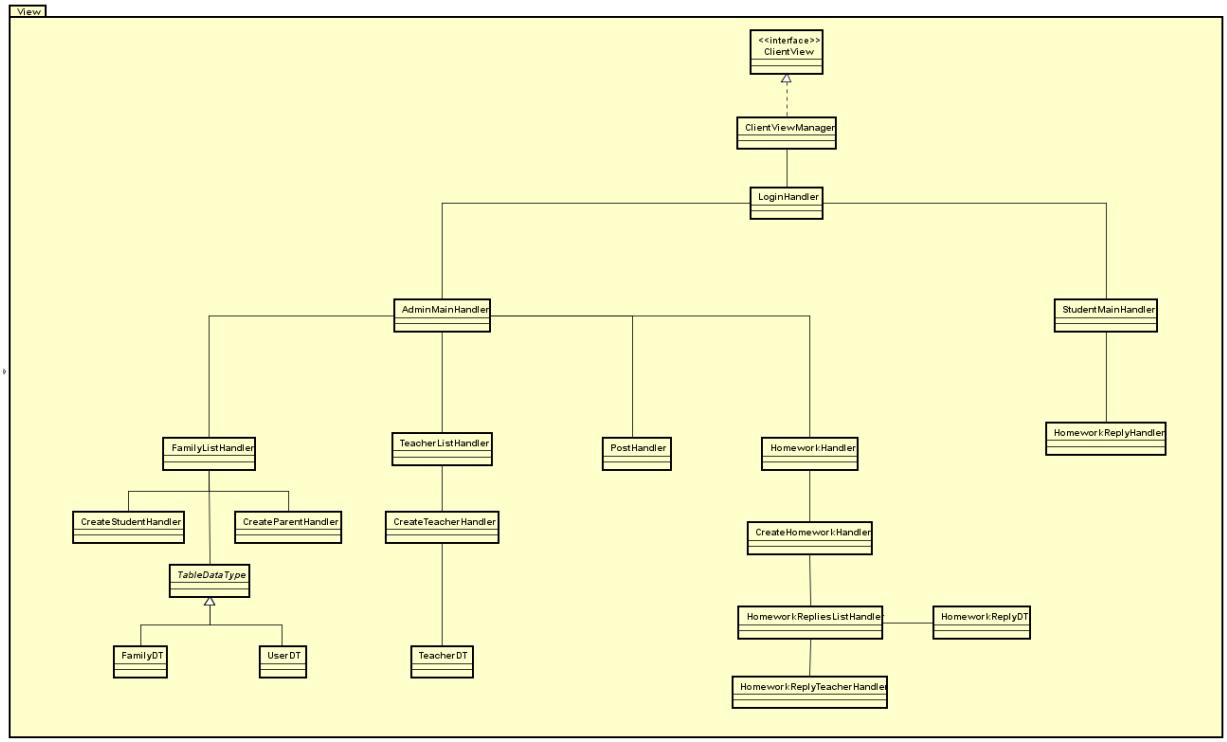


Figure 31

5.7 Threads

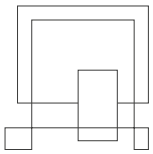
The system must be able to handle multiple clients at the same time. To do this, the server must create for each client a separate Thread for communication. In an infinite loop, the server thread is listening to incoming connections. When a connection is made, it creates a new Thread using the HandleClient class. This class is responsible for communication with the connected client and it implements the Runnable interface. Inside the run() method, the object is listening to incoming client requests. When a message is received, it is passed to the ServerController, which returns a response, that is send back to the client. Implementation of the server loop and the HandleClient class run() method can be seen below.

```
while (!serverSocket.isClosed()) {
    if (Thread.interrupted())
        break;

    System.out.println("Waiting for a client...");
    try {
        Socket client = serverSocket.accept();
        Thread clientThread = new Thread(new HandleClient(client));
        clientThread.start();
        clientThreads.add(clientThread);
    } catch (IOException e) {
        System.out.println("Connection error " + e.getMessage());
        e.printStackTrace();
    }
}
```

Figure 32

```java
public void run() {
    String clientIP;
    Message response, request;
    ObjectInputStream in = null;
    ObjectOutputStream out = null;
    try {
        clientIP = client.getInetAddress().getHostAddress();
        System.out.println("Welcome " + clientIP);
        out = new ObjectOutputStream(client.getOutputStream());
        in = new ObjectInputStream(client.getInputStream());
        while (!client.isClosed()) {
            if(Thread.interrupted())
                throw new InterruptedException();

            request = (Message) in.readObject();
            System.out.println(request);
            response = controller.handleMessage(request);
            out.writeObject(response);
            System.out.println("Send");
        }
    } catch (IOException | ClassNotFoundException e) {
```

Figure 33



6 Test

Having implemented the system, one has to conduct tests. Tests are needed not only to check whether the system is working and has been implemented correctly, but also to ensure that all requirements have been fulfilled and the system provides the end user with all the desired functionalities. Bearing that in mind, two kinds of tests have been executed: whitebox tests, in this case unit tests, and blackbox test, in this case tests following Requirement Test Descriptions. The results of the tests are shown further in this paragraph.

6.1 Unit test

To ensure that implementation of the system's features is working properly, a unit tests suite has been created. The tests are focused on validating separate methods and classes and are following the white box approach. The greatest attention was paid to testing the model, as it contains crucial operations on the domain, such as adding/deleting/editing users and posts. Other parts of system that the tests were focusing on were the Controller classes and the DBAdapter. As those classes are the connection between the other ones and are responsible for either converting or preparing data, the mocking technique has been chosen. For example, the DBAdapter class, that is preparing the SQL commands to be executed by the Database class on the database, is associated with the DBInterface (implemented by the Database class). However, there is no need to connect the physical database, as this is not a subject of the test. That is why a MockDatabaseTest class has been created. It is mocking the connection with the database and checking if the SQL commands have been created ly. This approach is very effective, as there is no need to connect to the remote server and tests are executed very fast (approximately 10 times faster than with connection).

In Appendix I can be found a unit test coverage report generated in InteliJ IDEA.

6.2 Requirement Test Descriptions

For this type of testing requirement test descriptions have been formulated, i.e. scenarios of what output the system should give after entering specific input. The descriptions and results are shown in Table 2.

No	Requirement Test Description	Passed (✔/X)	If failed, what?
Logging in			
1	An email of an administrator with a matching password is entered and the log in button is clicked. Afterwards, the system displays a welcoming screen.	✔	
2	An email of a student with a matching password is entered and the log in button is clicked. Afterwards, the system displays a welcoming screen).	✔	
3	An email of a parent with a matching password is entered and the log in button is clicked. Afterwards, the system displays a welcoming screen).	✔	
4	An email of a teacher with a matching password is entered and the log in button is clicked. Afterwards, the system displays a welcoming screen.	✔	
5	A wrong email is entered. The system displays a message: “wrong user name, try again”.	✔	
6	A wrong password is entered. The system displays a message: “wrong user password, try again”.	✔	
Changing password			
7	The administrator type of account is being used. A new account has been created. The system generates a random password and sends it to the typed email account.	✔	
8	The user enters the new added email as login and the received password. After pressing button “log in” the dialog window is displayed. The user writes a new password and confirms it by pressing button “ok”. Then the main page of user is loaded. During the next log in,	✔	

	only the new password is working (check both).		
9	The user presses the button “reset password”. The system generates a random password and sends it to the user’s email account. The user enters the email as login and the new received password. After pressing button “log in” the dialog window is displayed. The user writes a new password and confirms it by pressing button “ok”. Then main page of user is loaded. During the next log in, only the new password is working (check both)	✔	
Managing users			
10	The admin account is being used. After clicking the “teacher list” button, a “create teacher” button is visible. After clicking it, a form is displayed. In the form the user enters the name and the email of the new teacher. After clicking the “save” button, the new teacher can be found in the system.	✔	
11	The admin account is being used. After clicking the “teacher list” button, a “create teacher” button is visible. After clicking it, a form is displayed. In the form the user enters only the name and during the next test case only the email of the new teacher. After clicking the “save” button, there is a message displayed: “all fields must be filled". The user is not added to the system. Rest of the fields are filled. The procedure is as in requirement test description 1.	X	The message is not displayed, the teacher is added to the system.
12	The admin account is being used. After clicking the “teacher list” button, a “create teacher” button is visible. After clicking it, a form is displayed. In the form the user enters the name and the email of the new teacher. Afterwards the user clicks the logo. The main teacher list is displayed and the new teacher is not added to the system, all information have been discarded.	✔	
13	The admin account is being used. After clicking the “family list” button, all families are displayed as a tree table. A family from the list is selected. The “add student” button is clicked. The system displays a form containing: name and email fields and a dropdown list of classes. The user enters all the required information	✔	

	and clicks the “save” button. The new student can be found in the system and it is possible to log in into his/her account.		
14	The admin account is being used. After clicking the “family list” button, all families are displayed as a tree table. A family from the list is selected. The “add student” button is clicked. The system displays a form containing: name and email fields and a dropdown list of classes. The user does not enter all the required information and clicks the “save” button. There is a message displayed: “all fields must be filled". The user is not added to the system. Rest of the fields are filled. The procedure is as in requirement test description 4.	X	The message is not displayed, the student is added to the system. In the case of no chosen class nothing is happening.
15	The admin account is being used. After clicking the “family list” button, all families are displayed as a tree table. A family from the list is selected. The “add student” button is clicked. The system displays a form containing: name and email fields and a dropdown list of classes. The user enters all the required information. Afterwards the user clicks the logo. The family list is displayed and the new student is not added to the system, all information have been discarded.	✔	
16	The admin account is being used. After clicking the “family list” button, all families are displayed as a tree table. A family from the list is selected. The “add parent” button is clicked. The system displays a form containing: name and email fields. The user enters all the required information and clicks the “save” button. The new parent can be found in the system and it is possible to log in into his/her account.	✔	
17	The admin account is being used. After clicking the “family list” button, all families are displayed as a tree table. A family from the list is selected. The “add parent” button is clicked. The system displays a form containing: name and email fields. The user does not enter all the required information and clicks the “save” button. There is a message displayed: “all fields must be filled". The user is not added to the system. Rest of the fields are filled. The procedure is as in requirement test description 7.	X	The message is not displayed, the parent is added to the system.

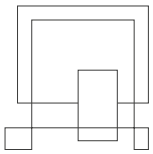
18	The admin account is being used. After clicking the "family list" button, all families are displayed as a tree table. A family from the list is selected. The "add parent" button is clicked. The system displays a form containing: name and email fields. The user enters all the required information. Afterwards the user clicks the logo. The family list is displayed and the new teacher is not added to the system, all information have been discarded.	✔	
19	The admin account is being used. The user clicks the "teacher list". Afterwards the system displays a list of all teachers. The user clicks one specific teacher and then the "edit teacher" button. The system displays a filed form with the name and email of the teacher. Next once a name, in the second test case an email is changed. The "save" button is clicked and the teacher has updated information.	✔	
20	The admin account is being used. The user clicks the "family list". Afterwards the system displays a list of all families. The user clicks on one specific family and student from the family and then the "edit user" button. The system displays a filed form with the name, email and chosen class of the student. Next once a name, in the second test case an email, in the third a class is changed. The "save" button is clicked and the student has updated information.	X	The user is not updated, new students are created in each test case.
21	The admin account is being used. The user clicks the "family list". Afterwards the system displays a list of all families. The user clicks on one specific family and parent from the family and then the "edit user" button. The system displays a filed form with the name and email. Next once a name, in the second test case an email is changed. The "save" button is clicked and the parent has updated information.	X	In both cases, just after going back to the main page and to the family list again, the user is updated, but also a new one is created.
22	The admin account is being used. The user clicks the "teacher list". Afterwards the system displays a list of all teachers. The user clicks one specific teacher and then the "delete teacher" button. A message: "are you sure you want to delete this teacher?" is shown. If "ok"	X	The message is not displayed. However, the teacher is deleted from the system.

	is being clicked, the system deletes the user from the system and the user cannot be found in any user lists, nor can log in to the system. If “cancel” is being clicked, then no changes in the system are made.		
23	The admin account is being used. The user clicks the “family list”. Afterwards the system displays a list of all families. The user clicks on one specific family and student from the family and then the “delete user” button. A message: “are you sure you want to delete this user?" is shown. If “ok” is being clicked, the system deletes the user from the system and the user cannot be found in any user lists, nor can log in to the system. If “cancel” is being clicked, then no changes in the system are made.	✔	The user is deleted just after going to the main page and back to the family list. (the list is updated)
24	The admin account is being used. The user clicks the “family list”. Afterwards the system displays a list of all families. The user clicks on one specific family and parent from the family and then the “delete user” button. A message: “are you sure you want to delete this user?" is shown. If “ok” is being clicked, the system deletes the user from the system and the user cannot be found in any user lists, nor can log in to the system. If “cancel” is being clicked, then no changes in the system are made.	✔	The user is deleted just after going to the main page and back to the family list. (the list is updated)
Managing homework			
25	A teacher type of account is being used. The “homework” button is being clicked, next the “create homework” button is being clicked, following that a form to fill is displayed. In the form there have to be entered the topic, criteria, number of students in to deliver it together, a deadline chosen from a calendar, a time chosen from the dropdown list and a class chosen from choice buttons. After clicking the “save” button, the homework is added to the system and the teacher can view it.	✔	
26	A student type of account is being used and a homework for that student’s class has been uploaded. The student got a notification and can see the uploaded homework.	✔	

27	A teacher type of account is being used. The "homework" button is being clicked, next the "create homework" button is being clicked, following that a form to fill is displayed. In the form there have to be entered the topic, criteria, number of students in to deliver it together, a deadline chosen from a calendar, a time chosen from the dropdown list and a class chosen from choice buttons. One has to leave a field empty (6 test cases, each with another field empty). After clicking the "save" button, an "all fields without a * must be filled" message can be seen. The homework is not added to the system.	✔	
28	A teacher type of account is being used. The "homework" button is being clicked, next the "edit" button on a specific homework is being clicked, following that a form with old information is displayed. After entering changes "save" button is clicked and the new information is overwritten and can be seen in homework.	X	The form doesn't contain old deadline and old classes are not selected in check boxes. The edited homework is added as new one and old one stays on the screen.
29	A teacher type of account is being used. The "homework" button is being clicked, next the "delete" button on specific homework is being clicked, following that the homework will be deleted from the system. The homework cannot be found in the system anymore (both from the teacher's and student's accounts).	X	The homework is deleted on the application where it has been deleted. Other online users can still see it.
Submitting homework			
30	A student type of account is being used. After clicking the "homework" button all homework for the specific class to which this student belongs to can be seen and no other homework is seen.	✔	
31	A student type of account is being used. The user enters a homework and clicks button "submit". Having clicked it, the user can type his/her answer in the text field. After clicking the "save" button the homework is saved and can be seen both for the student and the teachers (a need to log into a teacher's account and check).	X	The student is able to submit his/her solution, but teacher cannot see it
32	A student type of account is being used. The user enters	X	The student is able to edit

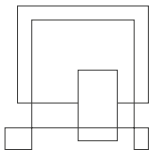
	a homework with an already written solution. The user presses the "edit reply" button and edits the answer. After clicking the "save" button, the answer is overwritten and can be seen for both the student and teacher.		his/her solution, but teacher cannot see it
33	A student type of account is being used. The user enters a homework, which's deadline has passed. The button "submit" is enable and student is able to write his/her solution. After clicking the "save" button the homework is saved and can be seen both for the student and the teachers (a need to log into a teacher's account and check, a note "after deadline" will be displayed).	X	The student is able to submit his/her solution, but teacher cannot see solution with the note "after deadline"
Checking homework			
34	A teacher type of account is being used. The "homework" button is being clicked, next the "done by" button is being clicked on a specific homework, following that a new window with homework replies list table is open. A teacher clicks two times on a student name. After that a new window with the student's reply is open.	X	The teacher is able to enter to the replies list, but student's solutions are not displayed

Table 2



7 Results and Discussion

The total outcome of the project can be stated as positive. From the beginning it was known, that not all of the customer's requirements would be met until the projects deadline. However, the core for the system has been done and all critical features implemented. That involves different types of users and managing them, analysis of posts with almost the whole part about managing homework and a logging system. Even though not all of the requirement tests passed, they gave an overview on what still needs to be done and corrected. The base of the system has been completed and the remaining tasks are clearly defined.



8 Conclusion

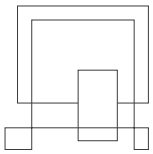
The purpose of the system was to make a maintainable system for eNTe that provides communication between students and teachers in the term of carrying out discussions, posting and submitting homework, as well as arranging meetings with parents. However, in the case of this project, most features were moved to delimitations and they were thought from the beginning to be implemented after the project's deadline (just for the customer).

In order to make the desired system, the creation has been divided into four parts: analysis, design, implementation and test. As the list of requirements from the customer was long, a new list of requirements was created just for the purpose of this project. Next a use case diagram was created and use case descriptions constructed, in order to develop the requirements further. Each part was analyzed and connections were established. Also the demands for the database were defined. The last thing was to create the skeleton for how to communicate, as the system is a client-server system.

The next stage was design. The design was extending the analysis by forming a blueprint for the implementation. Here, the main classes and connections between them were distinguished, design and architecture patterns chosen and a more specific overview of the database created. Moreover the connection has been specified and designed.

Following the design was the implementation phase. What this phase was about, was translating the diagrams created in design into code. Furthermore, the connection between the java application and database was established. Moreover, details were specified, e.x. choosing the best way for creating IDs and security issues.

Last but not least was testing. Two types of tests have been performed: blackbox and whitebox ones. Even though not all of them passed, they made it clear what has to be done in the future.



9 Project future

The future of this project consists primarily of finishing current tasks so that no of requirement test cases will still be failing. Furthermore, it is more than expected, that the features with medium and extra importance will be implemented and tested.

9.1 Medium and extra tasks

To make the system meet the customer's expectations, at least medium tasks must be implemented. Features such as adding announcements, checking homework replies, starting a discussion and participating in it, but also marking post as important or parental, will be taken into consideration first.

9.2 Remote observer

Other important features are notifying and updating currently connected users, when a change is made. This can be achieved with the Remote Observer design pattern. The server would have to keep track of active users (as a list of Threads) and send them appropriate information when needed.

9.3 Posts sending optimization

Another feature that can be implemented in the future (if needed) is optimization of sending posts (and generally data) from the server to the client. As during a semester, there could be created hundreds of posts, sending them all to the client with the WelcomingData can take some time. Another thing is, that the user may not need all of the posts, but only the most recent ones. To solve this problem, the client application could hold the number of posts that have been received. When older posts are needed, the client can make a request to the server with information how many posts have been already sent, so that the server can send more, without duplicating.

9.4 Deployment on a real server

To make the system accessible for all users, it must be deployed on a physical remote server, which can host an application all the time. In order to do that, a server must be bought and properly configured. Moreover, the server application must be adjusted to the server environment.

10 References

Michael Schulte-Markwort, 2015. *Burnout-kids. Wie das Prinzip Leistung unsere Kinder überfordert*.Maslow A.H. (1943). A theory of human motivation. Psychological Review, 50, 370–396. Crossref [Accessed 07 March 2018]

Alex Gray, 2016. The 10 skills you need to thrive in the Fourth Industrial Revolution. *WORLD ECONOMIC FORUM*. [e-journal], weforum https://www.weforum.org/agenda/2016/01/the-10-skills-you-need-to-thrive-in-the-fourth-industrial-revolution/ [Accessed 07 March 2018]

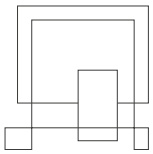
Michał Pasterski, 2014. *13 błędów polskiego systemu edukacji.* [online] http://michalpasterski.pl/2014/04/13-bledow-polskiego-systemu-edukacji/ [Accessed 13 March 2018]

Per Lundholm, 2013. Another builder pattern for Java. *Crisp's Blog*. [online] https://blog.crisp.se/2013/10/09/perlundholm/another-builder-pattern-for-java [Accessed 04 May 2018]

The Gang of Four: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* USA, Addison-Wesley.

Oracle, 2017. *JavaMail API.* [online] http://www.oracle.com/technetwork/java/javamail/index.html [Accessed 9 May 2018]

Wikipedia, 2018. *Universally unique identifier.* [online] https://en.wikipedia.org/wiki/Universally_unique_identifier [Accessed 4 April 2018]



11 List of Appendixes

Appendix A	Project Description – Project_Description.pdf
Appendix B	Data glossary – Data_glossary.pdf
Appendix C	Diagrams as Astah projects – eNTeDiagrams.zip
Appendix D	User guide – UserGuide.pdf
Appendix E	Code – sourceCode.zip
Appendix F	Client application – eNTe.jar
Appendix G	Server application – eNTeServer.jar
Appendix H	SQL code of the database – SQL.txt
Appendix I	Unit test coverage report – unitTestReport.zip