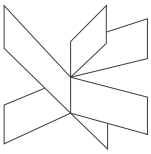


Employees management system

Students

Balkis Ibrahim 260092

Dziugas Austys 280144

Przemyslaw Regulski 280196

Ronald Johnson 279987

Supervisor

Troels Mortensen

ICT ENGINEERING

SECOND SEMESTER 2019

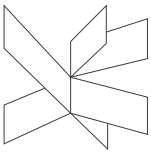
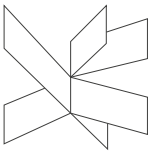
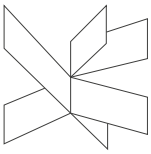


Table of content





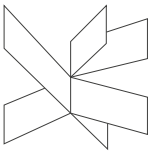
1 Introduction

Introduction of time management system into the companies, where employees working hours were tracked based on them logging into the system and being able to review their schedule helped not only the workers with being paid exactly for how much they've worked but also helped the companies with organizing their work force and making it easier to pay salaries without spending countless time figuring out who worked for how long and when he wasn't at work.

Currently, various recruitment agencies generally known as "Vikar" agencies in Denmark find there type of system perfectly suited for their needs. They juggle their employees across many companies, without any time management system employees would have hard time keeping with their schedule and companies would find it problematic to pay the salaries. That's why as a target for developed system those "Vikar" agencies were chosen as a target audience and system tries to meet their various needs.

The proposed system makes it possible for employers to add new employees to the database and assign responsibilities for them. Employees on the other hand can denote their beginning and ending of the shift. Each employee has his own account protected by a password so that there are no problems with accidentally logging on a wrong account.

The reason behind the project was to make a system that would be implemented as a Client/Server program. To achieve this RMI(Remote method invocation) was used. The data from and to database can be read and changed by two types of users in the system – administrators and employees.
To begin developing the system, firstly user stories had to be created where all of the functionalities the system will need were defined. Once those requirements were



achieved the analysis part begins. At this stage use case description and conceptual diagrams will be discussed.

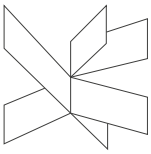
After this stage the design part begins in which activity diagrams, sequence diagrams and Class diagrams are made. When this stage is finished, finally the implementation part can take place, where code with examples gets discussed. Lastly going through testing of the system takes place with JUnit and scenario testing.

2 User stories and requirements

2.1 User stories

In this subchapter the user stories from the customer will be presented since the requirements for this report will be made based on them.

1. As an admin, I want to add employees to the system so that all the employees will have access to the system.
2. As an admin, I want to delete employees from the system so that all employees can no longer access the system.
3. As an admin, I want to edit employee's information in the system, so that all employee's data can be modified.
4. As an admin, I want to be able to assign shifts to employees, so that employees can view their work plan.
5. As an admin, I want to be able to remove shifts from employees work plan, so that employees can view their work plan.
6. As an admin, I want to be able to view employee's data, so that I have access to pertinent information regarding employees.
7. As a user, I want to be able to view my work schedule so that the schedule can be adhered to.
8. As a user, I want to able to denote my time of arrival and departure from work, so that my working hours can be recorded.
9. As a user, I want to be able to check my work-related statistics, so that I can calculate my income.



2.2 Functional requirements

Based on the information above the following requirements have been created.

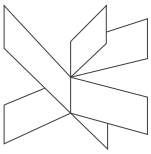
1. The system must be able to allow the admins to add employees.
2. The system must be able to allow the admins to remove employees.
3. The system must allow admins to modify employee's data.
4. The system must be able to allow the admins to assign shifts for employees.
5. The system must be able to show the work plan for the employees.
6. The system must allow employees to modify their data.
7. The system must allow the admin to view employee's data.
8. The system must allow the employees to specify the days they want to work.
9. The system must allow employees to view details regarding specific shift.
10. The system must allow employees to view their work statistics
11. The system must allow the employees to denote the time of arrival and departure from work.

2.3 Non-functional requirements

1. The System must follow the Client/Server architecture (RMI).
2. The system must be implemented in Java.
3. The usability of the system must be tested by end-users.
4. The system must store information in a database.

3 Analysis

3.1 Scenarios



Add employee scenario:

1. Manager provides name and relevant information for an employee.
2. System check the employee existence.

Classes:

- User

Methods:

- adduserBtnPressed
- checkEmail

3.2 use case diagram

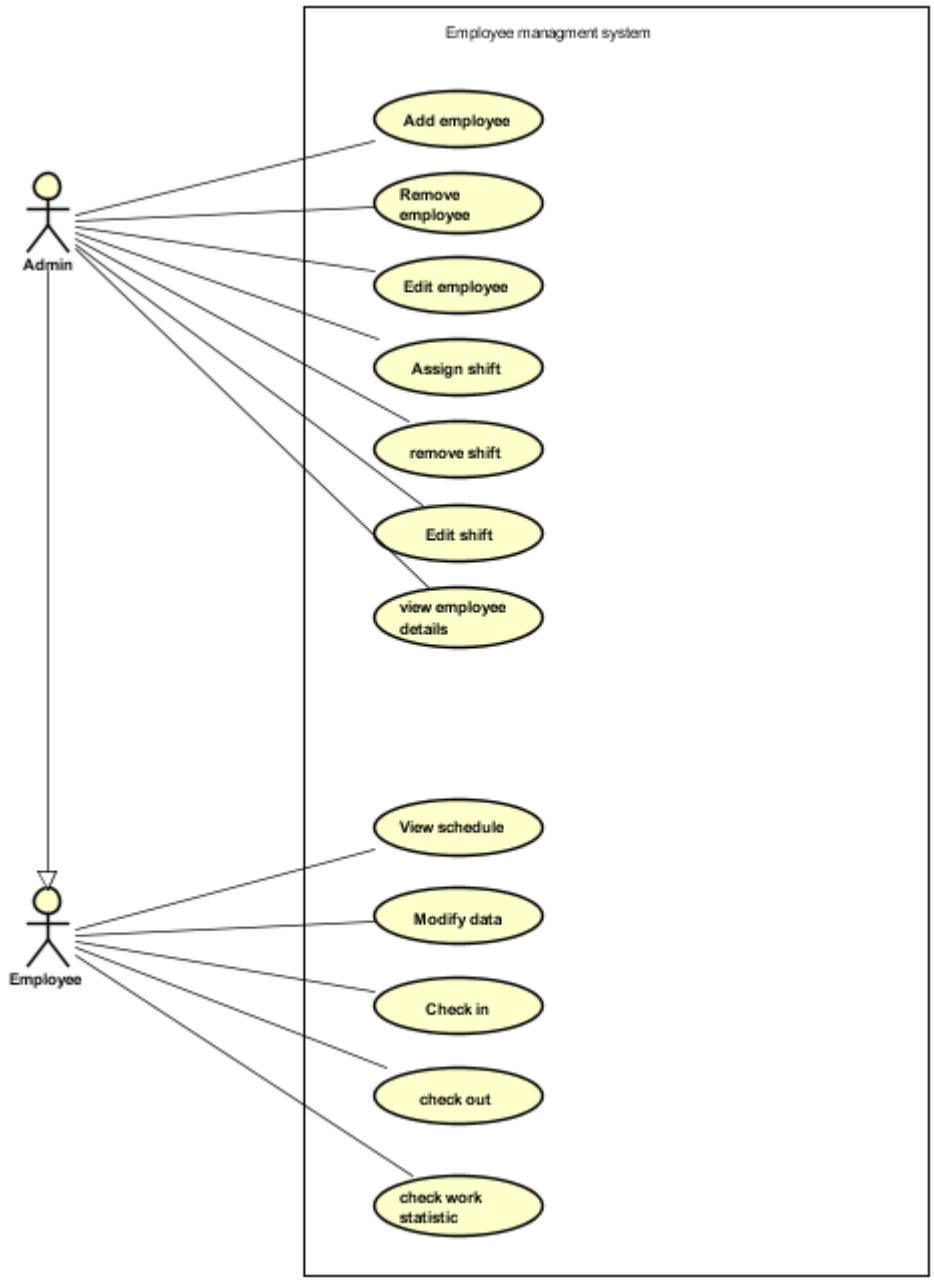
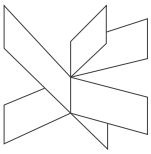


Figure 1use case diagram



The use cases methodology has been used to classify and organize the system requirements. As it is shown in the diagram there are two different actors each has it is own set of actions that needs to be performed as a part of their daily activities.

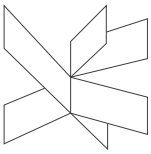
3.3 use case description

Based on the use case diagram shown above, several use case descriptions were made for this subchapter. Only one-use case description will be shown as an example while the rest can be seen in Appendix 2.

The use case description is a collection of values, preconditions, postconditions and base sequences which form a detailed view of the actor's actions while interacting with the system.

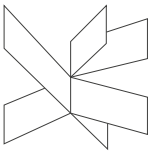
ITEM	VALUE
UseCase	Add employee
Summary	Adding a new user to the system
Actor	Admin
Precondition	The admin is already logged in
Postcondition	A new user is added
Base Sequence	1. The admin selects manage employees tab. 2. System displays a list of the employees. 3. The admin presses "Add employee" button 6. Add employee window opens up. 7. The admin inputs employee data into "name, email, . 8. The admin presses save button 9. System stores the data in the database. 10. Add employee window closes.
Branch Sequence	
Exception Sequence	1. Admin can cancel at any point. 3.1 The user already exists. 3.2 System displays an error "User already exists"
Sub UseCase	

Figure 2use case description for adding employee



The diagram above shows, what steps the admin will take for the system to successfully complete the requested action and what steps does the system make.

The precondition of a use case means, that some steps must be achieved for the use case to be able to reach the end of the branch sequence. In this case, the admin must be logged in to continue with the use case. The actor (admin) will fill out the form with the employee info. In case one of the existing employees has the same email address the system will reject the newly added employee and display an error, otherwise, the system will save the added employee in the database.



3.4 Activity diagram

The activity diagram was created during the inception face to get a clear idea of the certain actions that need to be executed by the actor to reach his goal.

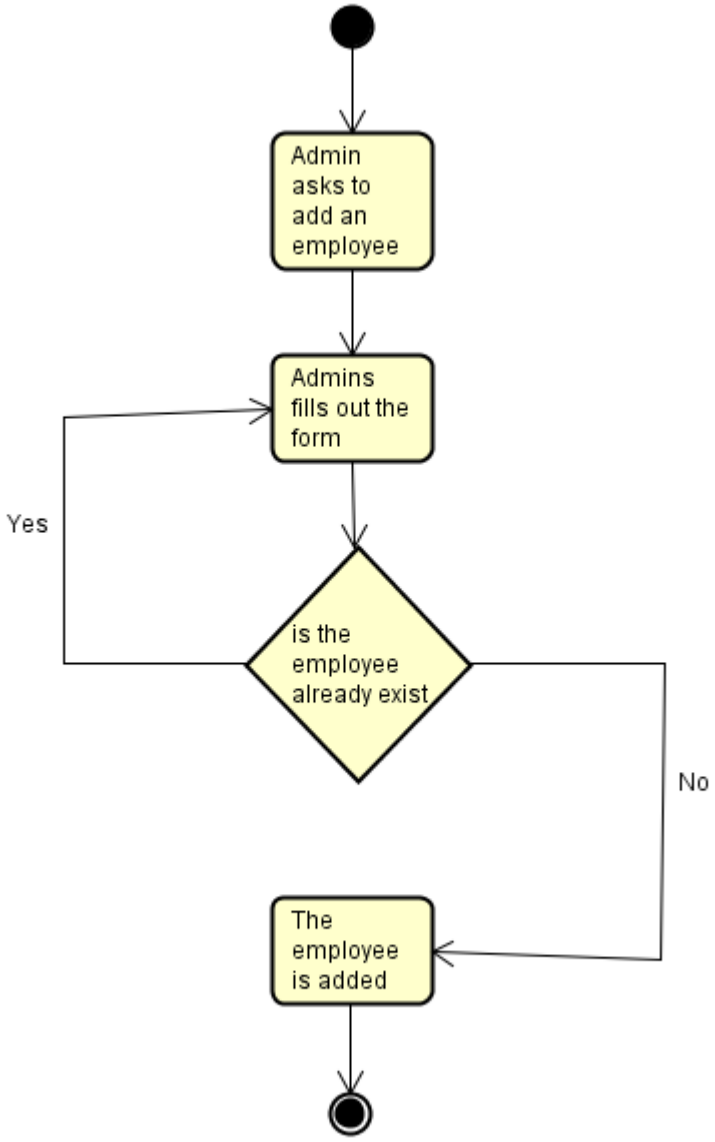
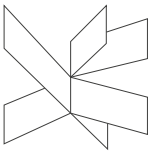


Figure 3Add Employee activity diagram



One of the Admin task is to add an Employee the activity diagram shows the steps needs to be taken to save the employee in the database.

3.5 Conceptual diagram

Conceptual Diagram demonstrates how all of the classes in their packages interact with each other. This is important since it provides a graphical representation on how the system will be created and how each component will interact with another to make it functional.

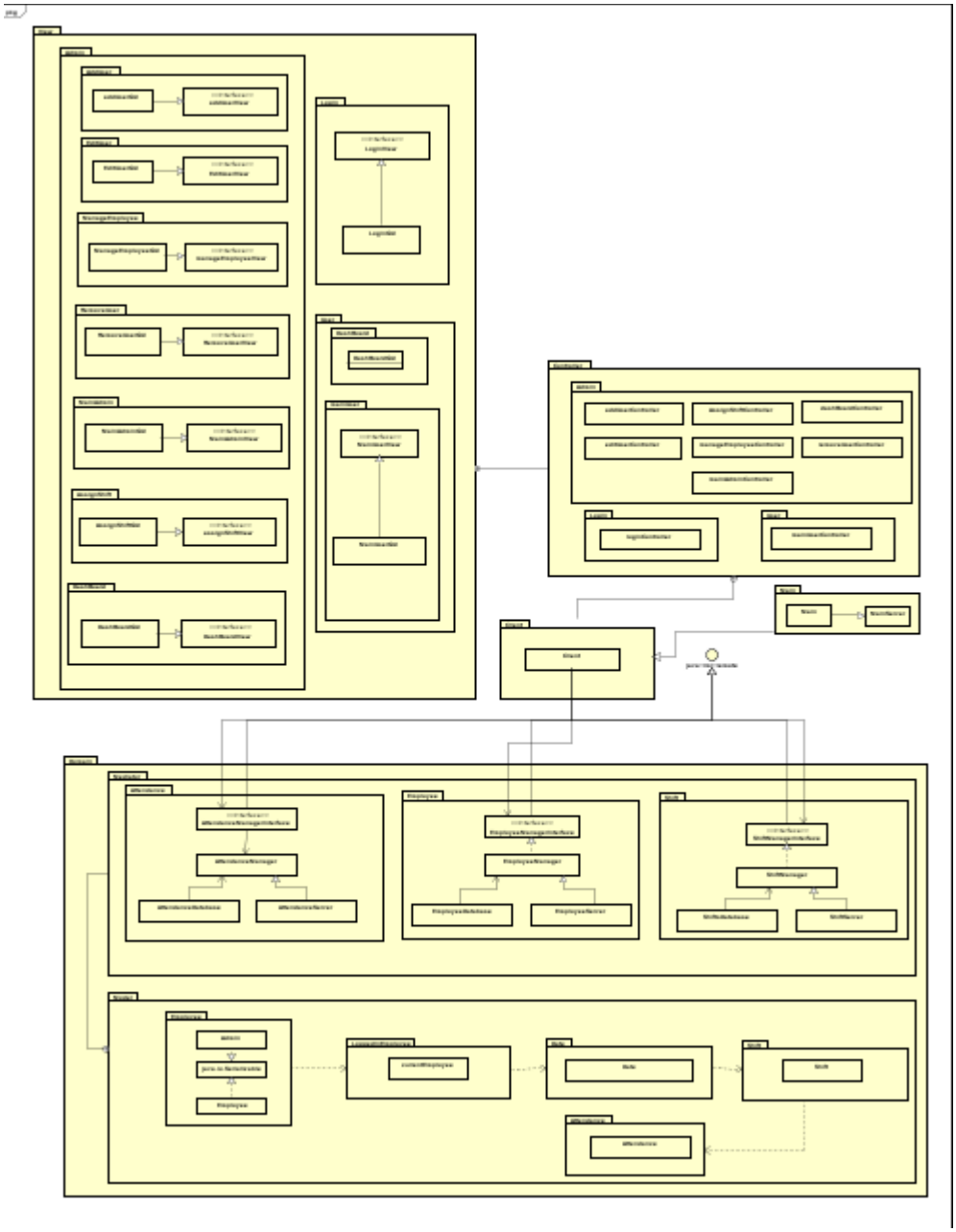
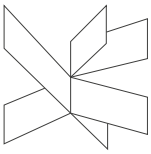
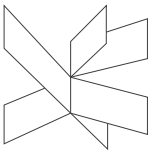


Figure 4conceptual diagram

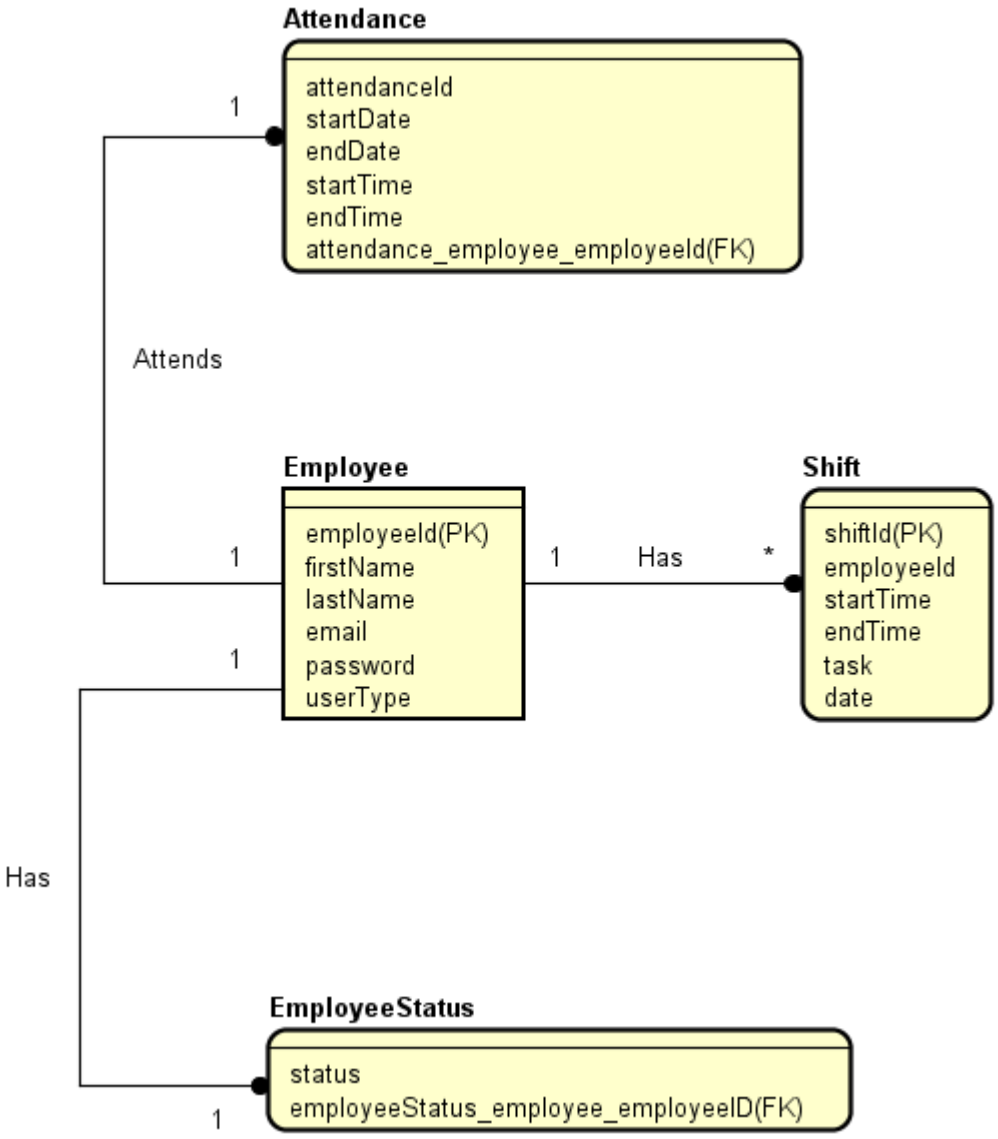
Following the SOLID principles, the conceptual diagram is divided in three main packages Domain, Controller and View. In addition, all main packages have their sub packages. Domain carries Mediator and Model.



In Model Employees, shifts.., and most of the classes with their lists. In Mediator package there are model managers and databases for shifts, attendance and employees…. Controller and View have Add, Remove and Edit for Shifts and employee.



3.6 Database

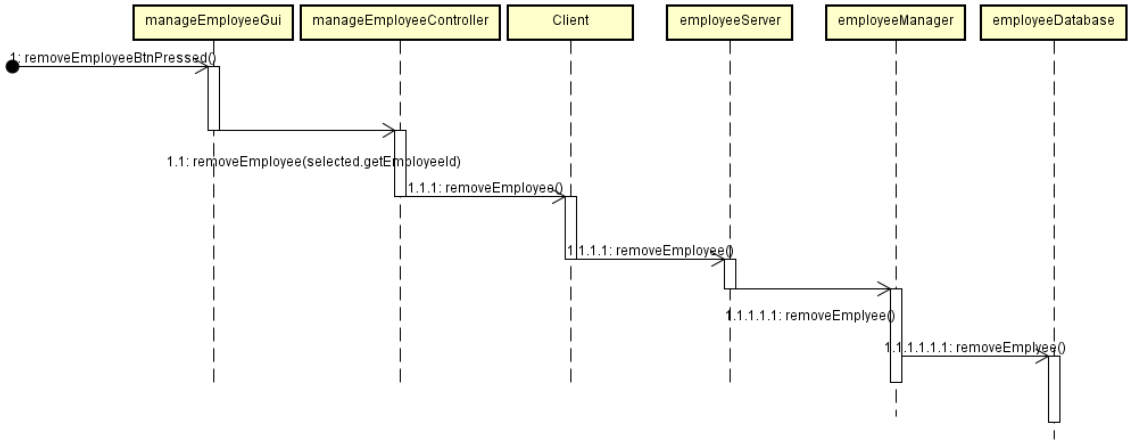

Attendance
attendanceId
startDate
endDate
startTime
endTime
attendance_employee_employeeId(FK)
1
Attends
Employee
employeeId(PK)
firstName
lastName
email
password
userType
1
1
Has
*
Shift
shiftId(PK)
employeeId
startTime
endTime
task
date
1
Has
EmployeeStatus
status
employeeStatus_employee_employeeID(FK)
1

4 Design

4.1 sequence diagram

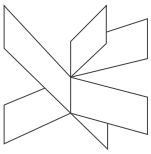
To get a better understanding on how the system behaves while certain commands are being given a sequence diagram is created. He sequences diagram will provide a visual representation on what steps the system will take, and which methods will be executed to fulfil a certain task.

The following figure will present an example of a sequence diagram for removing employee.



For this example, there are 5 life cycles consisting of the manageEmployeeGUI, the controller, the client, employee server, employeemanager, emplyeedatabase.

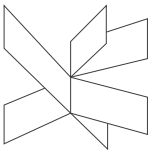
The first step should the admin take is selecting the employee they want to remove in the table, then clicks the remove employee button.
When the remove employee button is clicked, the selected employee is obtained by calling the tbData.getSelectionModel().getSelectedItem();
method in the manageEmployeeGUI class which returns the currently selected item.
The removeEmployee method from the manageEmployeeController class is then called,
using the value obtained above as the parameter.



Within the manageEmployeeController class the removeEmployee method is called from the client class.

removeEmployee method inside client class then through the Server class calls method removeEmplopyee
inside EmployeeManagerInterface which implements EmployeeManager class which calls removeEmployee
method inside EmployeeDatabase class and removeEmployee method inside EmployeeDatabase class sends
query to the database with employeeid of the employee to remove.

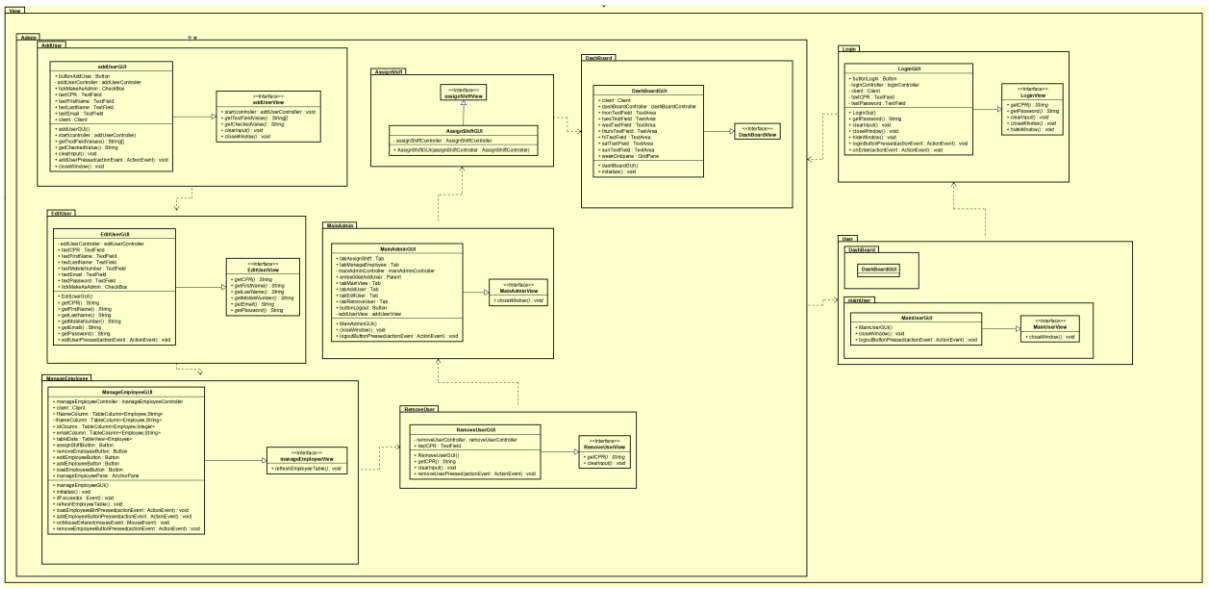
Back in the manageEmployeeGUI class the table is then refreshed using the refreshEmployeeTable(); method.



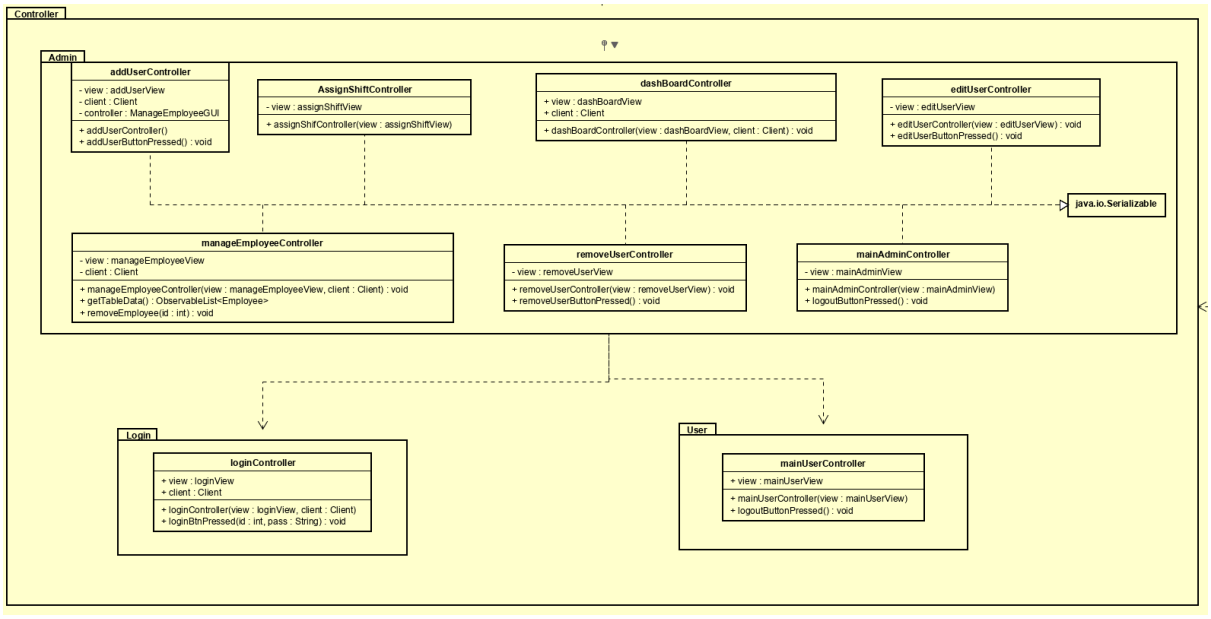
4.2 Class diagram and design pattern.

4.2.1 MVC Design pattern

The MVC design pattern is the most prevalent pattern throughout this project as it is how the GUI and its many windows are structured. This structure consists of the model, the view (actual interface the user sees) and the controller which dictates how the model and the view interact. A prime example of the implementation of the MVC can be seen in the class diagram below.



Controller Portion:



4.2.2 DAO Design pattern

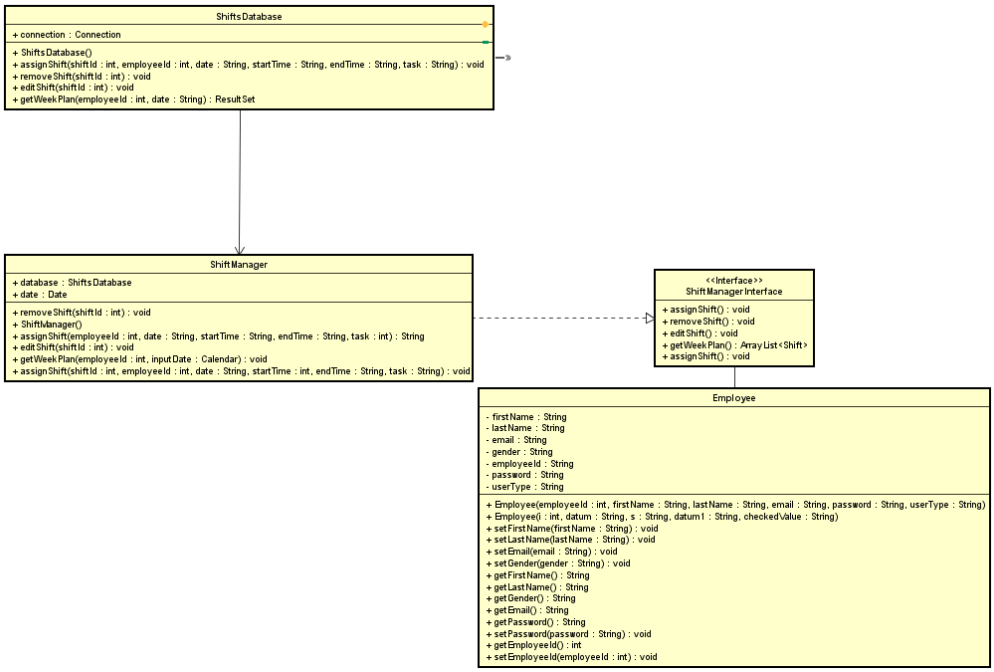


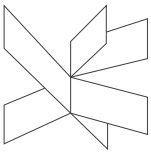
Figure 5 class diagram of DAO pattern implementation in the system

DAO pattern is used for separating lower level accessing API from higher level services. DAO pattern includes 3 participants.

Data Access Object concrete class – This is an implementation class for the interface explained below. Its responsibility is to get data from database or other data storage method.

Data Access Object Interface – The interface defines operations that will be performed on a model.

Model Object – This is an object containing get and set methods. It's used to store data received from any data storage mechanism.



This design pattern suits the needs because there is constant communication between database and model. The pattern is advantageous for the project for a few reasons:
- Because model and service layer are disconnected both of them can be updated independently, without disrupting the functionality of any of them, which also encapsulates the first SOLID principle.
-The pattern includes interface, so it also helps with sticking to interface segregation principle in SOLID.

4.3 GUI

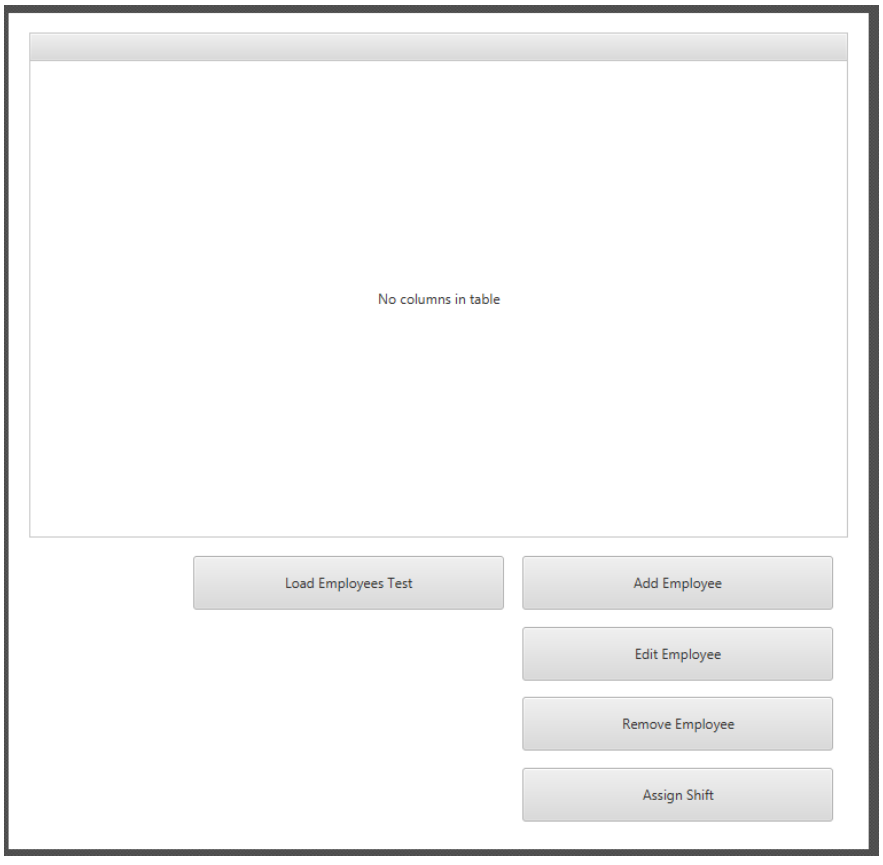
This section will elucidate a portion of the GUI's design as it is the sole medium through which the user interacts with the software.

The figure below depicts the Manage Employee window as it is an apt example:

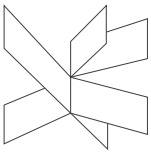
The FXML Code:

```
<AnchorPane fx:id="manageEmployeePane" onMouseEntered="#onMouseEntered" prefHeight="730.0" prefWidth="750.0" xmlns="http://javafx.com/javafx/11.0.1" xmlns:fx="http://javafx.com/fxml/1" fx:controller="view.admin.manageEmployee.manageEmployeesGUI">
   <children>
      <TableView fx:id="tbData" layoutX="18.0" layoutY="17.0" prefHeight="441.0" prefWidth="714.0">
      </TableView>
      <Button fx:id="assignShiftBtn" layoutX="448.0" layoutY="659.0" mnemonicParsing="false" onAction="#assignShiftBtnPressed" prefHeight="47.0" prefWidth="271.0" text="Assign Shift" />
      <Button fx:id="removeEmployeeBtn" layoutX="448.0" layoutY="597.0" mnemonicParsing="false" onAction="#removeEmployeeBtnPressed" prefHeight="47.0" prefWidth="271.0" text="Remove Employee" />
      <Button fx:id="editEmployeeBtn" layoutX="448.0" layoutY="536.0" mnemonicParsing="false" prefHeight="47.0" prefWidth="271.0" text="Edit Employee" />
      <Button fx:id="addEmployeeBtn" layoutX="448.0" layoutY="474.0" mnemonicParsing="false" onAction="#addEmployeeBtnPressed" prefHeight="47.0" prefWidth="271.0" text="Add Employee" />
      <Button fx:id="loadEmployeesBtn" layoutX="161.0" layoutY="474.0" mnemonicParsing="false" onAction="#loadEmployeesBtnPressed" prefHeight="47.0" prefWidth="271.0" text="Load Employees Test" />
   </children>
</AnchorPane>
```

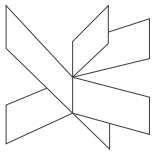
The View the User Sees:



This window was chosen as it encompasses a multitude of actions that can be taken with the system and as such contains a reasonable amount of complexity.

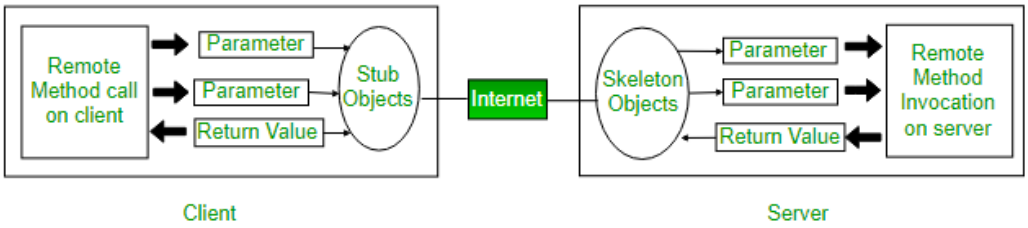


Upon selecting an item from the list which is populated with current employees in the database, various actions can be performed on them through the use of the buttons below the table. This tab is only visible to admins, this is done during the login as two different main windows exist; one for employees and one for users.



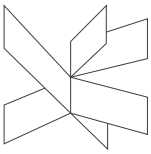
4.4 Client/Server (RMI)

In order to achieve client/server communication RMI API had been used. It allows one part of the software to remotely access methods that exist on different address space, for example another machine. The communication is handled using stub object on a client side and skeleton object on the server side. The stub object creates an information block that consists of method name of the object which we want to access and parameters for accesses method. The skeleton object role is to call and forward parameters to the method from stub object.



Working of RMI

RMI is widely used because of its reliability and built in multithreading. The implementation of RMI is very clear and because of that it's easy to stick to a certain code architecture.



5 Implementation

5.1 GUI Implementation

The figure below outlines one of the more interesting examples of the code with regards to the GUI as it deals with complicated tableviews and client/database requests in the controller.
The beauty of the MVC design pattern is exemplified within this example as it links the complex and fragile nature of tableViews with the dynamic nature of the database and client.

The manageEmployee controller adheres to the SOLID design principle of single responsibility as its sole purpose is to be the bridge between the view and the client.

The controller's purpose is to obtain information from the database and pass it to the view where it is then displayed within the TableView. Its connection to the view will be further expounded upon below as it is pertinent to the implementation of the MVC in this example.

manageEmployee Controller:

```
public class manageEmployeeController {
    private manageEmployeeView view;
    private Client client;

    public manageEmployeeController(manageEmployeeView view, Client client) {
        this.view = view;
        this.client = client;
    }


    public ObservableList<Employee> getTableData() throws Exception {
        ObservableList<Employee> employeeList = FXCollections.observableArrayList(client.getEmployees());
        for (Employee employee:employeeList) {
            System.out.println(employee.getFirstName());
        }

        return employeeList;
    }


    public void removeEmployee(int id) throws Exception {
        client.removeEmployee(id);
    }



}
```

manageEmployeeGUI:

```
public class manageEmployeeGUI implements manageEmployeeView {
    public manageEmployeeController manageEmployeeController;
    public Client client;
    @FXML
    public TableColumn<Employee, String> fNameColumn;
    @FXML
    public TableColumn<Employee, String>lNameColumn;
    @FXML
    public TableColumn<Employee, Integer> idColumn;
    @FXML
    public TableColumn<Employee, String> emailColumn;
    @FXML
    public TableView<Employee> tbData;
    public Button assignShiftBtn;
    public Button removeEmployeeBtn;
    public Button editEmployeeBtn;
    public Button addEmployeeBtn;
    public Button loadEmployeesBtn;
    public AnchorPane manageEmployeePane;



    public manageEmployeeGUI() throws Exception {
        client = new Client();
        this.manageEmployeeController = new manageEmployeeController( view: this, client);
        System.out.println("Employee management started");
        tbData = new TableView<>();
        tbData.getColumns().clear();
        fNameColumn = new TableColumn<>( text: "First Name");
        fNameColumn.setCellValueFactory(new PropertyValueFactory<>("FirstName"));
        lNameColumn = new TableColumn<>( text: "Last Name");
        lNameColumn.setCellValueFactory(new PropertyValueFactory<>("LastName"));
        idColumn = new TableColumn<>( text: "ID");
        idColumn.setCellValueFactory(new PropertyValueFactory<>("EmployeeId"));
        emailColumn = new TableColumn<>( text: "Email");
        emailColumn.setCellValueFactory(new PropertyValueFactory<>("Email"));

    }

    @FXML
    public void initialize() throws Exception {
        ObservableList<Employee> employees = manageEmployeeController.getTableData();
        tbData.setItems(employees);
        tbData.getColumns().clear();
        for (Employee employee:employees) {
            tbData.getColumns().clear();
            tbData.getColumns().addAll(fNameColumn,lNameColumn,idColumn,emailColumn);

        }
    }
```

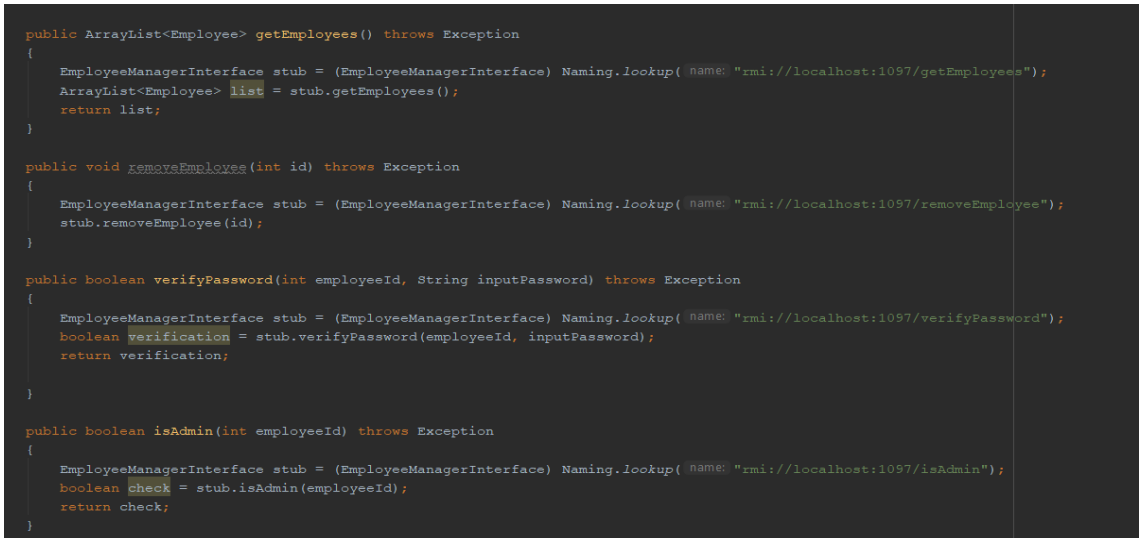
In order to obtain information and then insert it into the tableview of the GUI, the cellvaluefactorys had to be set for each column with the column's name pertaining to the get method for the respective column's intended data.
Upon initialization of the GUI, an observable list is retrieved through the manageEmployeeController where the items are then set to the table. The table is then cleared in case of any duplicates during initialization and a for loop inserts the respective data for each employee into the columns.

5.2 RMI implementation

The program implements 3 server classes, each responsible for managing one aspect of the software. In order to implement RMI we have to go through

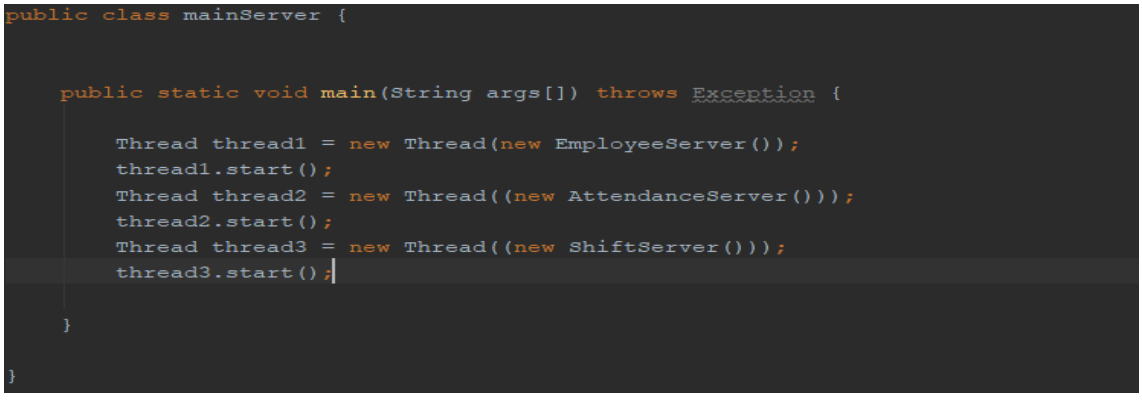
```
public class EmployeeServer extends EmployeeManager implements Runnable {

    @Override
    public void run() {

        try {
            EmployeeManager obj = new EmployeeManager();

            Registry registry= LocateRegistry.createRegistry( port: 1097);
            EmployeeManagerInterface stub = (EmployeeManagerInterface) UnicastRemoteObject.exportObject(obj, port: 1097);
            System.out.println("Registry started...");

            registry.bind( name: "addEmployee", stub);
            registry.bind( name: "removeEmployee", stub);
            registry.bind( name: "getEmployees", stub);
            registry.bind( name: "verifyPassword", stub);
            registry.bind( name: "isAdmin", stub);
            registry.bind( name: "getOneEmployee", stub);
            registry.bind( name: "editEmployee", stub);
            System.out.print("server rdy");

        } catch(Exception e)
        {
            System.out.print("not rdy " + e.toString());
            e.printStackTrace();

        }
```

On the picture above it can be seen that registry is created on given port and methods from implementation class are binded to the registry with registry.bind() method. Before that stub objects are created by exporting remote object that is casted into EmployeeManagerInterface class . Now the Client class can communicate with the server through stub object and access those methods inside implementation class.



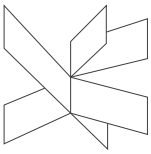
```
public ArrayList<Employee> getEmployees() throws Exception
{
    EmployeeManagerInterface stub = (EmployeeManagerInterface) Naming.lookup( name: "rmi://localhost:1097/getEmployees");
    ArrayList<Employee> list = stub.getEmployees();
    return list;
}

public void removeEmployee(int id) throws Exception
{
    EmployeeManagerInterface stub = (EmployeeManagerInterface) Naming.lookup( name: "rmi://localhost:1097/removeEmployee");
    stub.removeEmployee(id);
}

public boolean verifyPassword(int employeeId, String inputPassword) throws Exception
{
    EmployeeManagerInterface stub = (EmployeeManagerInterface) Naming.lookup( name: "rmi://localhost:1097/verifyPassword");
    boolean verification = stub.verifyPassword(employeeId, inputPassword);
    return verification;

}

public boolean isAdmin(int employeeId) throws Exception
{
    EmployeeManagerInterface stub = (EmployeeManagerInterface) Naming.lookup( name: "rmi://localhost:1097/isAdmin");
    boolean check = stub.isAdmin(employeeId);
    return check;
}
```

Above image represents a part of Client class, in here stub objects are created based on registry lookup results and methods can now be remotely accessed.



```
public class mainServer {


    public static void main(String args[]) throws Exception {

        Thread thread1 = new Thread(new EmployeeServer());
        thread1.start();
        Thread thread2 = new Thread((new AttendanceServer()));
        thread2.start();
        Thread thread3 = new Thread((new ShiftServer()));
        thread3.start();


    }

}
```

Lastly in order to launch all 3 servers multithreading is being used. 3 threads are created, and all the servers are running at the same time.

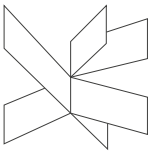


5.3 DAO pattern implementation

In the implementation Data Access Object concrete class is seperated into 2 classes. One of them EmployeeDatabase is responsible only for sending SQL statements into the database, and performs no other function. EmployeeManager class on the other hand performs operations on data coming from or to database. This way the first SOLID principle is being uphold.

This is part of EmployeeManager class, it either takes data from database using EmployeeDatabase class or forwards data into it. During the communication it performs various operations on the data as seen below and in the getEmployees method. Firstly it obtains ResultSet object with all the employee data from the database by calling database.getEmployees() method from EmployeeDatabase class. It starts with while loop going through all employees, creating new Employee class object and using it's set method to assign data based on column names. When all set methods are called object is added to an ArrayList by list.add(employee) and processes repeats itself as long as there are still not exported employees.

```
public ArrayList getEmployees() throws Exception {
    ArrayList<Employee> list = new ArrayList<>();
    ResultSet rs = database.getEmployees();
    while (rs.next()) {
        int employeeId = rs.getInt( columnLabel: "employeeid");
        String firstName = rs.getString( columnLabel: "firstname");
        String lastName = rs.getString( columnLabel: "lastname");
        String email = rs.getString(("email"));

        Employee employee = new Employee();
        employee.setFirstName(firstName);
        employee.setLastName(lastName);
        employee.setEamil(email);
        employee.setEmployeeId(employeeId);
        list.add(employee);
    }
    return list;
}

public boolean verifyPassword(int employeeId, String inputPassword) throws Exception
{
    ResultSet rs = database.getPassword(employeeId);
    rs.next();
    String password = rs.getString( columnLabel: "password");


    if(password.equals(inputPassword))
    {
        return true;
    }
    return false;
}
```



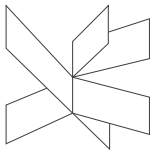
The DAO pattern is completed by having implemented the interface shown below.

```
public interface EmployeeManagerInterface extends Remote {

    void addEmployee(String firstName, String lastName, String email, String userType) throws Exception;
    void removeEmployee(int employeeId) throws Exception;
    ArrayList<Employee> getEmployees() throws Exception;
    boolean verifyPassword(int employeeId, String inputPassword) throws Exception;
    boolean isAdmin(int employeeId) throws Exception;
    Employee getOneEmployee(int employeeId) throws Exception;
    void editEmployee(int employeeId, String firstName, String lastName, String email, String password) throws Exception;

}
```

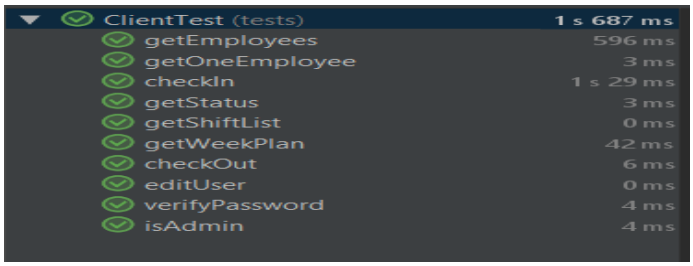
6 Testing

In order to test the software two methods of testing had been used - scenario and white-box testing. White-box testing was achieved with the use of JUnit test framework. With JUnit testing most of the methods inside Client class. Special user was created inside the database in order to allow database related methods to run the tests through it. In verifyPassword() method test, asserEquals method is used. First value is the value we are expecting to receive, assertEquals method will check if value received from client.verifyPassword() will be the same. The testing employee password is set to “test” so based on given argument we should receive Boolean true, if that will be the case the test will pass.

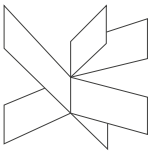
```
@Test
public void verifyPassword() throws Exception {
    Client client = new Client();
    assertEquals( expected: true, client.verifyPassword( employeeId: 0,  inputPassword: "test"));
}

@Test
public void isAdmin() throws Exception {
Client client = new Client();
assertEquals( expected: false, client.isAdmin( employeeId: 0));

}

@Test
public void getOneEmployee() throws Exception {

    Client client = new Client();
    assertEquals( expected: "test", client.getOneEmployee( employeeId: 0).getFirstName());

}


@Test
public void getWeekPlan() throws Exception {

    Client client = new Client();
    assertNotNull(client.getWeekPlan( employeeId: 1, Calendar.getInstance()));
}
```



This is an example of few methods tested with JUnit framework. Whenever changes to the code were made, all of the tests would be executed and hopefully testing class would return results shown on the picture below



In addition to JUnit tests scenario testing was also used, where GUI part was used to test various user activity.

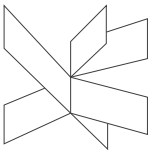


7 Conclusion

The project “Employee management system” has been completed, but due to lack of time, there are few delimitations.

It is required by the university that the project should follow the Client-Server and it also should have multiple design patterns. To follow these requirements, many user stories where drawn and it took a long time to make a correct class diagram that would be relevant to the user’s needs. Designing and Coding was challenging, and the reason was the summer holiday. It was not easy to get the whole team members focusing on the project

In conclusion, we did our best and we tried as much as it is possible to deliver a good product that can be possible for future development.

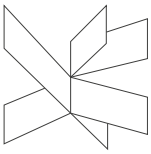


8 References

Ken Schwaber, Jeff Sutherland, 2011; The Scrum Guide, [Last accessed 10/04/2018] via link: https://studienet.via.dk/Class/IT-SWE1XS18/Session%20Material/Scrum_Guide.pdf

Project report, 2017 (Appendix 3) VIA Engineering Guidelines [Last accessed 10/04/2018] via link: https://studienet.via.dk/projects/Engineering__project_methodology/General/Guidelines/2017%20Project%20Report%20(Appendix%203)%20%20VIA%20Engineering%20Guidelines.pdf Project description, 2017 (Appendix 1) VIA Engineering Guidelines [Last accessed 27/02/2018] via link: https://studienet.via.dk/projects/Engineering__project_methodology/General/Guidelines/2017%20Project%20Description%20(Appendix%201)%20%20VIA%20Engineering%20Guidelines.pdf

Process report,2017 (Appendix 2) VIA Engineering Guidelines, [Last accessed 05/04/2018] via link: https://studienet.via.dk/projects/Engineering__project_methodology/General/Guidelines/2017%20Process%20Report%20(Appendix%202)%20%20VIA%20Engineering%20Guidelines.pdf



Appendices

Appendix1 use case diagram

Appendix2 Activity diagrams and sequence diagram

Appendix3 Scenarios

Appendix4 Project description

Appendix5 user guid