Aflevering 2 It 2

Andreas Bach Berg Nielsen s205869
Naveed Imam Shah s205491
Shiv Gopal S205490
Troels Engsted Kiib s205492
Gruppe nr.5
Github: https://github.com/Troels21/JavaFX.0.2

February 2021

Abstract

This IT assignment at DTU is a continuation of our previous IT assignment. We have been assigned to develop our software further by substituting our file-saving system with a SQL database. Furthermore, our updated software had some requirements including being a 'Minimum viable product'. The system should also be able to collect data, save data in a SQL database and lastly present the data. The simulation shall be implemented with an observer-pattern. The process and our thoughts should also be documented which we did by explaining out GUI design, database design and design patterns. Because the software is a continuation the code has obviously changed which results in new UML diagrams which we will account for. The diagrams will be used for showing the program structure and chronology. There will also be added and extra diagram which is the SQL database diagram. Overall we have fullfilled the previously mentioned new requirements and the results as well as the work behind it are described thoroughly in this journal.

Indholdsfortegnelse

1 Introduktion

I denne delaflevering 2 af vores projekt, vil vi bygge videre på det projekt vi startede i delaflevering 1. Vi vil hovedsageligt kigge på implementering af en SQL database i dette program. Vi vil lave en database til vores program, så der kan læses og skrives data fra den. Dernæst har vi tænkt os, at lave en datasamling i vores database med login informationer på vores brugere.

2 Analyse

Her vil vi analysere vores brug af SQL.

2.1 SQL

Vi skal have lavet en database, hvor der bliver gemt og opbevaret en masse data. Denne data indebærer brugerlogin og patientdata. Dette er der behov for, da man i sundhedssektoren arbejder i sygehuse, så man ofte arbejder i teams. Dette er vigtigt at have i baghovedet, da lægen og sygeplejersken, vil tilgå forskellige computere, for at opnå den samme data. Derfor skal der laves en database. En database der kan læses og skrives til. Tidligere har patient arkivet, læst data fra en fil. Den skal nu være i stand til at læse fra databasen. Der skal også justeres hvordan data gemmes. Dette skal gøres da man skal have muligheden, for hvor dataen skal gemmes. Hvis der tale om en specifik data der skal gemmes, kan dette gøres med en knap, der gemmer et billede af det specifikke interval. Dette gøres med fokus i at skrive journaler, der man gerne vil fange et nøje interval, der opstår interessant. Når denne database laves, betyder det for lægerne at de nu kan tilgå deres patienters data, fra deres kontorer. Dette vil give muligheden for nemmere behandling, da de ikke skal fra computer til computer, men rettere har alle patienterne samlet et sted. Dernæst giver det også muligheden for sygeplejersken at gemme data lokalt, hvis det er passende. Hvilket vil give den frihed, at der kan arbejdes i et lokale, altså tæt, men også fjernt, gennem databasen. Dette er der behov for, da læger og sygeplejersker ofte ikke har samme arbejdsopgaver, og er derfor fordelt ud, på et sygehus. For patientens vedkommende, vil dette betyde, at han uafhængigt kan tilgå sin data, ved blot at tilgå dem via databasen. Dette vil i hensigt, give patienterne mulighed for at tilgå deres data hjemmefra.

3 Design

3.1 Generelt

For at kunne visualisere hvordan vores program skal se ud, har vi lavet et Analyse Klasse Diagram over hele problemstillingen, samt et kollaborationsdiagram til, at kunne se kommunikationen imellem de forskellige stages. Vi har yderligere beskrevet dem i forrige delaflevering. Diagrammerne kan ses på figurerne 1 og 2.

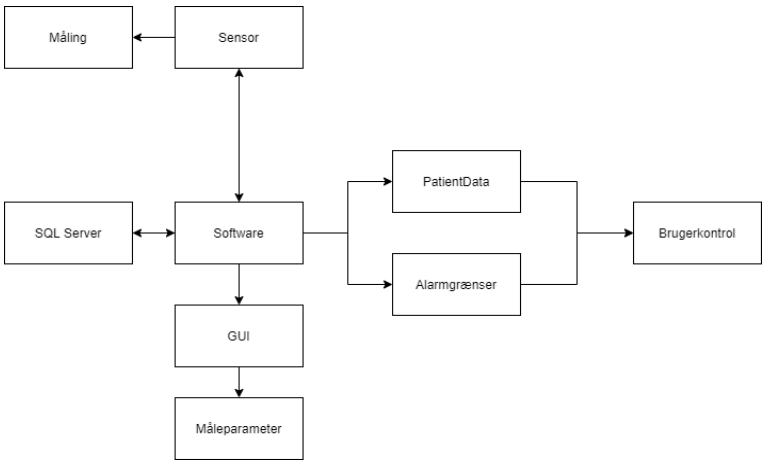

Figure 1: Analyse klasse Diagram

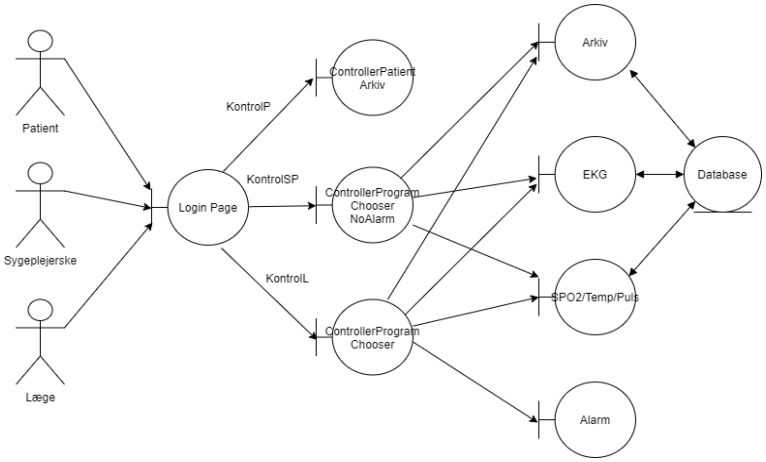

Figure 2: Kollaborations Diagram

3.2 SQL

Nu vil vi formidle visuelt hvordan vi har benyttet tabellerne i databasen, samt hvilke interne forbindelser disse tabeller indgår i. Nedenfor kan der ses et ERD/SQL-diagram:

På ovenstående billede fremgår der, at der er 4 tabeller som der benyttes.
Disse redegøres der nu for hvert for sig:

• loginInfo:
Denne tabel benyttes til at opbevare de relevante parametre, som der skal anvendes for at logge ind. Dvs, et brugernavn (username) og en kode (password). Begge er af datatypen "varchar(255)", dvs. at brugernavnet og koden maks. må være 255 tegn i alt. Udover disse er der også to andre kolonner ved navn "id" og "doctor". Begge er af datatypen INT, så de skal være bestå af hele tal. "id" har "auto increment" slået til, som er en constraint, der gør at der automatisk tælles op, hvorimod "doctor" udelukkende har constrainten "not null" tændt, som sørger for at værdien for doctor ikke kan være 0. Både brugernavnet og koden har derudover "not null" som constraint, for at sørge for at de ikke er tomme. Udover dette har brugernavn også "unique" som constraint, der gør at det samme brugernavn ikke kan benyttes af flere forskellige brugere.

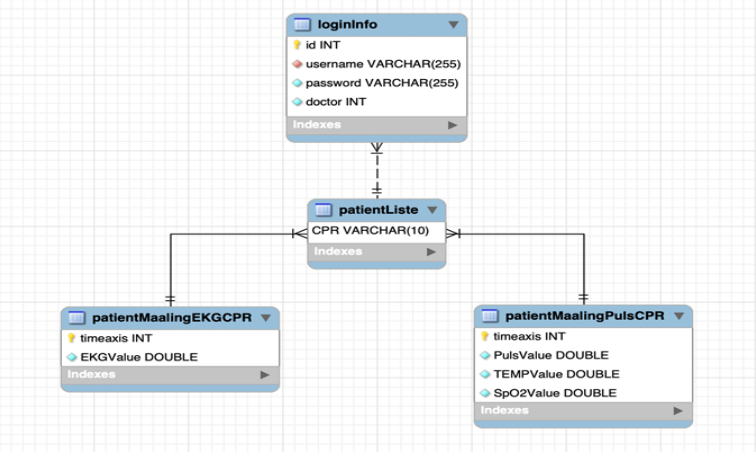

Figure 3: Sql diagram

• patientListe:
Denne tabel bruges til opbevaring af patienternes CPR numre, som også kan ses i den eneste kolonne der indgår, nemlig "CPR". "CPR" er af datatypen "varchar(10)", som betyder at værdien højst kan bestå af 10 tegn. Dette er naturligvis pga. et CPR nummer består af 10 tal. Derudover gøres der brug af følgende constraints: "primary key", "not null" og " unique". En patients CPR nummer kan ikke være null eller ens, dette forklare brugen af de constraints.

• patientMaalingEKGCPR:
Denne tabel benyttes til opbevaring af EKG-værdier over en hvis tid, for de individuelle patienter. Dette kan ses i tabelnavnet, hvori at patientens CPR-nummer også indgår. Tabellen indeholder følgende kolonner: "timeaxis" og "EKGValue". "timeaxis" indeholder de tilhørende INT-tidsværdier til EKG-værdierne, og derfor indgår følgende constraints: "primary key" og "not null". "EKGValue" har også "not null" som constraint, og dette er naturligvis fordi, at der ikke kan bruges en tom måling. Derudover er den af datatypen double, idet EKG-værdien angives som decimaltal.

• patientMaalingPulsCPR:
Denne tabel bruges på samme måde som "patientMaalingEKGCPR", bare hvor at der i stedet for kolonnen "EKGValue", så er der nogle andre kolonner "måleværdier" her: "PulsValue", "TEMPValue" og "SPO2Value". Grunden til at EKG værdierne står i en tabel for sig selv, er pga. at EKG værdierne kommer meget hurtigere end de resterende målinger, som vil resultere i noget synkroniseringsbesvær hvis det hele skulle stå i samme tabel.

Der er derudover nogle interne forbindelser mellem tabellerne. Disse er repræsenteret vha. hhv. den stiplede linje og de fulde linjer. Den stiplede linje er brugt mellem tabellen "loginInfo" og tabellen "patientListe". Den stiplede linje repræsentere et såkaldt "non-identifying relationship", dvs. at tabellerne har noget med hinanden at gøre, men at de ikke behøver at afhænge af hinanden. Dvs. at de kan benyttes selvstændigt. Dette er tilfældet idet at hvis sundhedspersonalet benytter programmet, så indtaster de deres legitimationsoplysninger osv. hvorefter, de så kan indtaste patientens CPR-nummer for enten at se eller foretage målinger. Så her afhænger tabellerne ikke af hinanden, men hvis det var en patient som loggede ind med sit CPR nummer, så afhænger loginInfo af patientListe idet at patienten jo ikke kan logge ind og se sin data medmindre det indtastede CPR-nummer i username kolonnen i loginInfo tabellen,

matcher et CPR-nummer i CPR kolonnen i "patientListe" tabellen.

Udover dette er der også to fulde linjer fra tabellen "patientListe" til hhv. tabellerne: "patientMaalingPulsCPR" og "patientMaalingEKGCPR". Denne fulde linje repræsentere et såkaldt "identifying relationship", som i princippet bare er det omvendte af et "non-identifying relationship", som blevet beskrevet tidligere. Dvs. at tabellerne: "patientMaalingPulsCPR" og "patientMaalingEKGCPR", afhænger af tabellen "patientListe". Dette gør de, idet at patientens CPR-nummer indgår direkte i deres tabelnavne, så for at tabellen kan indeholde en bestemt persons data, så skal tabellen jo kunne identificeres til den specifikke patient. Og dette gøres netop vha. CPR-nummeret, som jo fås fra kolonnen "CPR" i tabellen "patientListe", så derfor er der identifying relationships mellem dem. Normalt kaldes det også for "parent-child relationship", idet den ene (child) afhænger af den anden (parent).

4 SQLJava - Implementation

Nu har vi gennemgået hvordan vores database-struktur er konstrueret, samt hvilke tanker vi har gjort os bag det. Så nu vil vi fortsætte dette ved at forklare sammenkoblingen mellem SQL databasen og Java-koden. Nu bliver koden gennemgået helt konkret.

Først ser vi de globale variabler og klassens metode(makeConnectionSQL) til at skabe en database connection, metoden(removeConnectionSQL) til at slukke connectionen og de tre String variabler skal bruges til at danne forbindelse mellem Java-koden og SQL-databasen. Dernæst kan vi se deklarationen af to objekter, Connection og Statement objekterne, disse bliver initialiseret i makeConnectionSQL metoden. Connection objektet bliver initialiseret med getConnection metoden, som tager de tre Strings som argumenter, og på den måde oprettes der forbindelse til SQL databasen. Vi bruger så connection objektet til at initialisere statement objektet med createStatement metoden. Sidst kan removeConnectionSQL metoden ses, som indholder to kontrolstrukturer, et try/catch og et if statement. if statementet kontrollere om forbindelsen til negationen af isClosed metoden er sand, hvis den er kalder den close metoden og forbindelsen til SQL databasen lukkes.

```
package sample;

import java.sql.*;

public class SQL {

    static String url = "jdbc:mysql://localhost:3306/login";
    static String user = "root";
    static String password = "1234";
    static Connection myConn;
    static Statement myStatement;

    public void makeConnectionSQL(){
        try {
            myConn = DriverManager.getConnection(url, user, password);
            myStatement = myConn.createStatement();
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
    }

    public void removeConnectionSQL(){
        try {
            if (!myConn.isClosed()){
                myConn.close();
            }
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
    }
```

Figure 4: sql constructor

Nu burde forbindelsen hermed bare dannes. Nu kan der gøres brug af den. Dette gøres ved en række metoder. Nedenstående metode "createNewPatient", har til formål at skabe en ny patient i databasen. Dette gøres ikke i selve metoden, men derimod så gøres der brug af 3 andre metoder hvor at String værdien af CPR så benyttes. Disse 3 metoder forklares nu.

```
31
32      //metode til der samler oprettelse af tabeller og opdatering af patientliste.
33      public void createNewPatient(String CPR) {
34          makeConnectionSQL();
35
36          writePatientListe(CPR);
37          createTableCPREKG(CPR);
38          createTableCPRPuls(CPR);
39
40          removeConnectionSQL();
41      }
42
```

Figure 5: createNewPatient metode

Den første af de 3 er "writePatientListe()". Først angives en String-værdi, hvor der på SQL står værdien som skal indsættes i CPR kolonnen i patientListe tabellen. Dette bruges så i sammenhæng med en PreparedStatement i en try/catch. Her gøres der brug af ovenstående, hvorefter det executes. Hvis ikke det kan lade sig gøre, kommer der en exception.

```
43      // Write metoden, som skriver CPR ind i patientlisten
44      public void writePatientListe(String CPR) {
45          String write_to_database1 = "INSERT INTO patientListe " + "(CPR) values(?);";
46          PreparedStatement PP1;
47          try {
48              PP1 = myConn.prepareStatement(write_to_database1);
49              PP1.setString( parameterIndex: 1, CPR);
50              PP1.execute();
51          } catch (SQLException throwables) {
52              System.out.println("CPR eksisterer allerede i systemet.");
53          }
54      }
55
```

Figure 6: writePatientListe metode

Den anden af de 3 er "createTableCPRpuls()". Her dannes der en ny tabel, som så skal indeholde tid, puls, kropstemperatur og iltmætning for den pågældende patient. Dette kan også ses, idet tabellens navn også kommer til at indeholde patientens CPR-nummer. Dette kan ses i den String-værdi som bruges, hvori følgende tabel med tidligere nævnte kolonner skal oprettes hvis ikke en allerede findes. Her bruges der igen en try/catch hvor at der så prøves af executes vha. myStatement. Hvis ikke det virker, fremkommer der så en exception.

```
55
56      // create metode som laver en tabel som indeholder tid, puls, temp og spo2.
57      public void createTableCPRPuls(String CPR) {
58          String sql_CreateTable = "CREATE TABLE IF NOT EXISTS patientMaalingPuls" + CPR + "(\n"
59                  + "timeaxis INT PRIMARY KEY AUTO_INCREMENT,\n"
60                  + "PulsValue DOUBLE,\n"
61                  + "TEMPValue DOUBLE,\n"
62                  + "SpO2Value DOUBLE);";
63          try {
64              myStatement.execute(sql_CreateTable);
65
66          } catch (SQLException e) {
67              e.printStackTrace();
68          }
69      }
```

Figure 7: createTableCPRPuls metode

Den sidste af de 3 er "createTableCPREKG()". Metoden er bygget op akkurat på samme måde som den forrige. Den eneste forskel er, at i stedet for puls, kropstemperatur og iltmætning er der så EKG. Grunden til EKG-værdierne har sin egen tabel er pga. synkroniseringsårsager idet EKG-værdierne kommer markant hurtigere end de resterende.

```
71      // create metode som laver en tabel som indeholder tid og ekg.
72      public void createTableCPREKG(String CPR) {
73          String sql_CreateTable = "CREATE TABLE IF NOT EXISTS patientMaalingEKG" + CPR + "(\n"
74                  + "timeaxis INT PRIMARY KEY AUTO_INCREMENT,\n"
75                  + "EKGValue DOUBLE);";
76          try {
77              myStatement.execute(sql_CreateTable);
78          } catch (SQLException e) {
79              e.printStackTrace();
80          }
81      }
```

Figure 8: createTableCPREKG metode

Nu er CPR-nummeret for den pågældende patient indsat i patientListe tabellen, og der er oprettet de to tabeller til den specifikke patient, så nu kan der skrives i dem. Dette gøres bla. i nedenstående metode: "writeToPatientMaalingPuls()". Metoden har samme struktur som de tidligere forklarede metoder. Idet først laves der en String hvor det som der skal, står på SQL, altså at puls, kropstemperatur og iltmætning indsættes i de respektive kolonner i den pågældende patients tabel til de værdier. Dette udeføres også vha. en try/catch, hvor der så også bruges en PreparedStatement samt Connection. Hvorefter der så bruges "setDouble()" på værdierne til de pågældende pladser (parameterindex).

```
83      // write metode som skriver puls temp og spo2 ind i en tabel, som den identificere med CPR.
84      public void writeToPatientMaalingPuls(String CPR, double Puls, double Temp, double SpO2) {
85          try {
86              makeConnectionSQL();
87              String write_to_database2 = "insert into patientMaalingPuls" + CPR + "(PulsValue, TEMPValue,SpO2Value) values(?, ?, ?)";
88              PreparedStatement PP2 = myConn.prepareStatement(write_to_database2);
89
90              PP2.setDouble( parameterIndex: 1, Puls);
91              PP2.setDouble( parameterIndex: 2, Temp);
92              PP2.setDouble( parameterIndex: 3, SpO2);
93
94              PP2.execute();
95              removeConnectionSQL();
96          } catch (SQLException e) {
97              e.printStackTrace();
98              removeConnectionSQL();
99          }
100     }
```

Figure 9: writeToPatientMaalingpuls

I metoden "writeToPatientMaalingEKG()", sker der præcis det samme som den forrige metode, bare hvor puls, kropstemperatur og iltmætning er udskiftet med EKG.

```java
//write metode som skriver ekg ind i en tabel, som den identificere med EKG.
public void writeToPatientMaalingEKG(String CPR, double EKG) {
    try {
        makeConnectionSQL();
        String write_to_database2 = "insert into patientMaalingEKG" + CPR + "(EKGValue) values(?)";
        PreparedStatement PP2 = myConn.prepareStatement(write_to_database2);

        PP2.setDouble( parameterIndex: 1, EKG);

        PP2.execute();
        removeConnectionSQL();
    } catch (SQLException e) {
        e.printStackTrace();
        removeConnectionSQL();
    }
}
```

Figure 10: writeToPatientMaalingEKG

I "rowCounter()" metoden bruges der omtrent samme struktur som før, med en String og en try/catch osv, men forskellen er at her bruges next() til at tage den næste linje. Vha. metoden får man så talt antallet af rækker.

```java
//row tæller
public int rowCounter(String Table, String CPR) {
    String sql_Count = "SELECT COUNT(*) FROM " + Table + CPR;
    ResultSet rs;
    try {
        makeConnectionSQL();
        rs = myStatement.executeQuery(sql_Count);
        rs.next();
        int buffer = rs.getInt( columnIndex: 1);
        removeConnectionSQL();
        return buffer;
    } catch (SQLException throwables) {
        throwables.printStackTrace();
        removeConnectionSQL();
    }
    return 0;
}
```

Figure 11: rowCounter

Nu skal værdierne så kunne aflæses. Dette gøres bland andet med metoden "readDataEKG()", hvor der i String-variablen, står at alle kolonner skal hentes fra tabellen der indeholder EKG-værdier for patienten. Hvorefter der så vha. en while-løkke, indsættes værdier i hhv. en tids- og ekg-array.

```
137     // Read metode som læser data fra ekg tabel
138     public void readDataEKG(String CPR, int[] tid_array, double[] ekg_array) throws SQLException {
139         makeConnectionSQL();
140         String sql_SelectFrom = "SELECT * FROM login.patientMaalingEKG" + CPR;
141         ResultSet rs = myStatement.executeQuery(sql_SelectFrom);
142         int i = 0;
143         while (rs.next()) {
144             tid_array[i] = rs.getInt( columnLabel: "timeaxis");
145             ekg_array[i] = rs.getDouble( columnLabel: "EKGValue");
146             i++;
147         }
148         removeConnectionSQL();
149     }
```

Figure 12: readDataEKG

Dettes samme gøres i metoden: "readDataPuls()", bare med puls, kropstemperatur og iltmætning i stedet for EKG.

```
151     // read metode som læser data fra Puls tabel
152     public void readDataPuls(String CPR, int[] tid_array, double[] puls_array, double[] temp_array, double[] SpO2_array) throws SQLException {
153         makeConnectionSQL();
154         String sql_SelectFrom = "SELECT * FROM login.patientMaalingPuls" + CPR;
155         ResultSet rs = myStatement.executeQuery(sql_SelectFrom);
156         int i = 0;
157         while (rs.next()) {
158             tid_array[i] = rs.getInt( columnLabel: "timeaxis");
159             puls_array[i] = rs.getDouble( columnLabel: "PulsValue");
160             temp_array[i] = rs.getDouble( columnLabel: "TEMPValue");
161             SpO2_array[i] = rs.getDouble( columnLabel: "SpO2Value");
162             i++;
163         }
164         removeConnectionSQL();
165     }
```

Figure 13: readDataPuls

"Readdatalogin()" metoden bruges på samme måde som de andre med en try/catch. Her aflæses logininfo om den pågældende bruger, hvilket kan ses ved der hvor der står "WHERE username =...". Metoden forsøger så at returnere dataen fra logininfo.

```
167     // read metode som læser data fra logininfo
168     public String ReadDataLogininfo(String username) {
169         makeConnectionSQL();
170         try {
171             String sql_SelectFrom = "SELECT *\n" +
172                     "From login.logininfo\n" +
173                     "WHERE username ='" + username + "' ;";
174             ResultSet rs = myStatement.executeQuery(sql_SelectFrom);
175             rs.next();
176             String buffer = rs.getString( columnIndex: 2) + "," + rs.getString( columnIndex: 3) + "," + rs.getString( columnIndex: 4);
177             removeConnectionSQL();
178             return buffer;
179         } catch (SQLException throwables) {
180             removeConnectionSQL();
181             return "null";
182
183         }
184     }
```

Figure 14: ReadDataLogininfo

Den sidste metode i klassen: "doesPatientExsist()", bruges til at prøve at se om et pågældende CPR nummer findes i databasen. Dette gøres ved en try/catch hvor der, ses om patienten findes i patientListe tabellen. Det er en boolsk kontrol, så enten bliver den true eller false. Altså om patienten findes, i vores database, eller ej. Hvilket kan bruges til bekræftelse når CPR-nummer indtastes i Arkiv, eller når patienten skal logge ind.

```java
    // boolsk kontrol af om cpr eksistere i database
    public boolean doesPatientExsist(String CPR) {
        makeConnectionSQL();
        String findPatient = "SELECT CPR FROM patientliste WHERE CPR =" + CPR + ";";
        ResultSet rs;
        try {
            rs = myStatement.executeQuery(findPatient);
            rs.next();
            boolean buffer = rs.getBoolean( columnIndex: 1);
            removeConnectionSQL();
            return buffer;

        } catch (SQLException throwables) {
            removeConnectionSQL();
            return false;
        }
    }
}

```

Figure 15: doesPatientExsist

5 Afprøvning

I dette afsnit vil vi afprøve vores kode. Vi vil teste simuleringsklassen og SQL klassen. I simuleringsklassen vil vi afprøve, om simulationerne kan udskrive forskellige tal, indenfor et ordentligt interval, og om vi gentagende kan skrive til evt. konsollen, for at simulere hvordan vi skriver til linecharts. Ved SQL vil vi afprøve, om vi kan skrive til og læse fra vores database, samt undersøge om vi kan søge efter data i vores database. Vi vil gøre brug af Junit5 til dette.

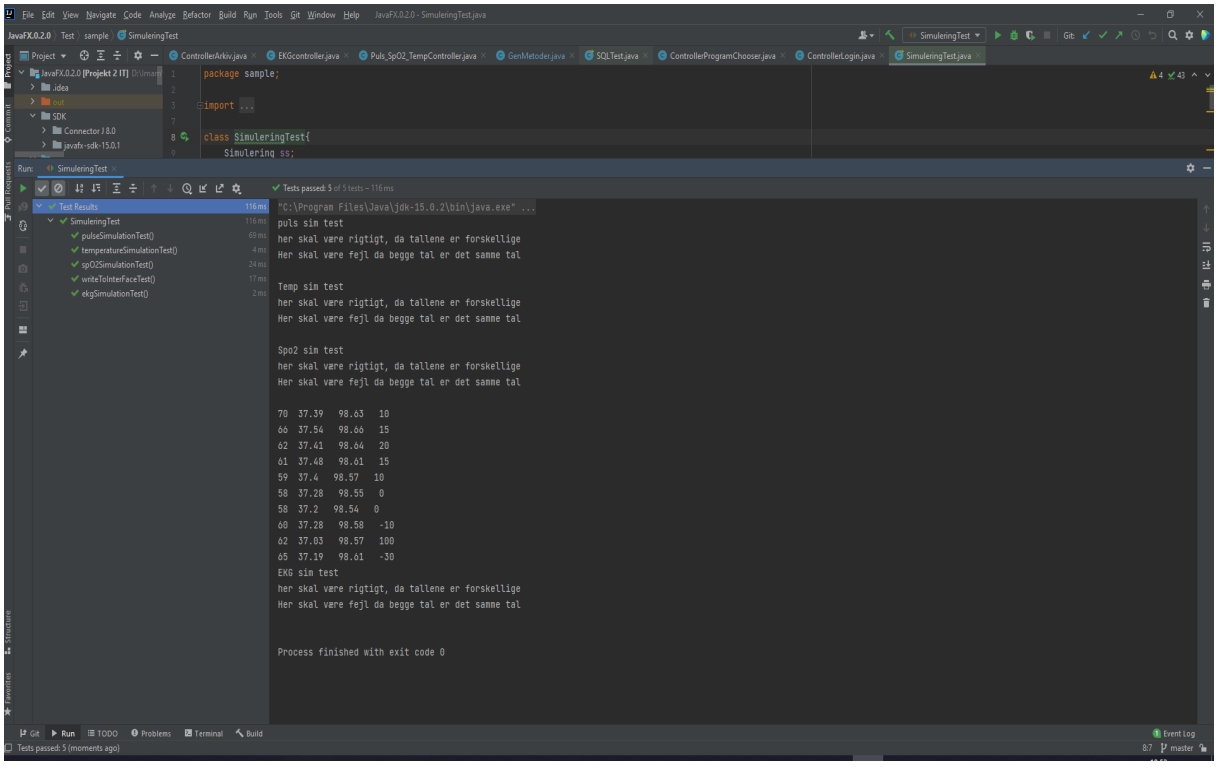

Figure 16: Test af Simulering

Figur 16 viser afprøvning af vores simulerings metoder. Hvor vi afprøver, om de simulerede værdier lagres korrekt i databasen, og om vores arrays opdateres på en ordentlig måde.

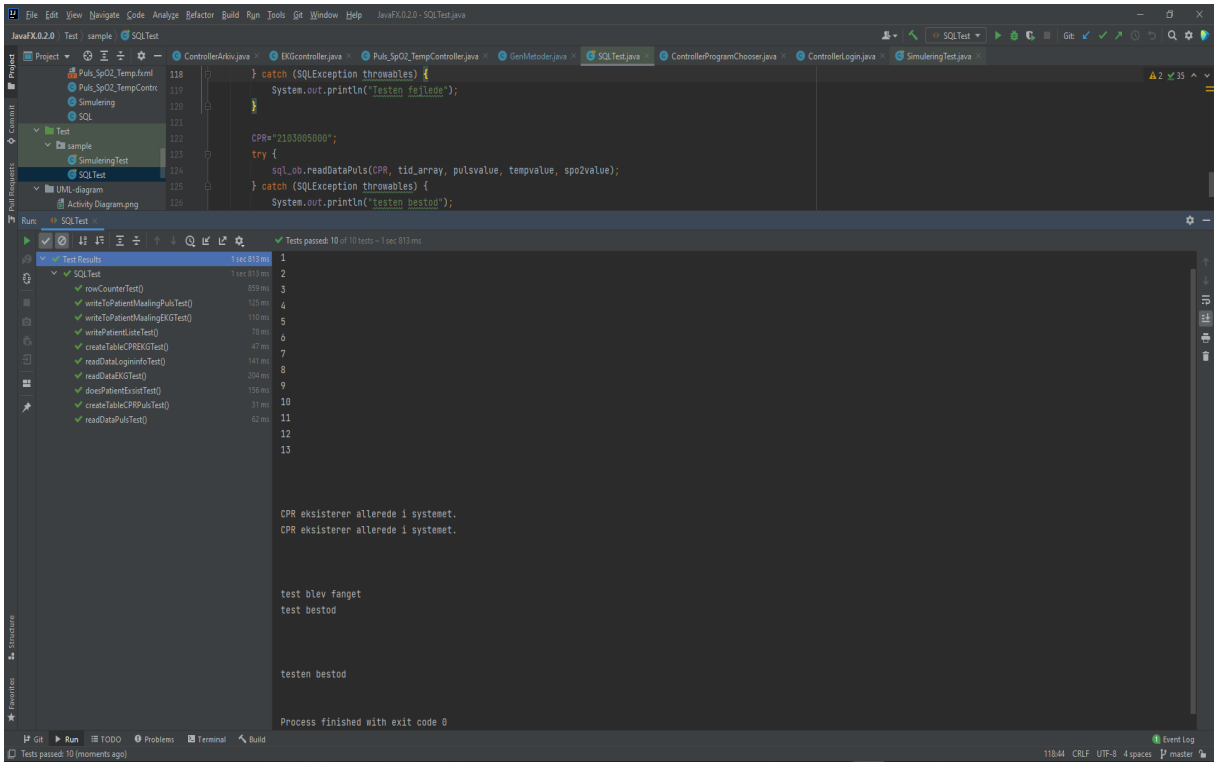

Figure 17: Test af SQL og dets tilhørende metoder

Figur 17 viser hvordan vi afprøver vores SQL database og dets metoder. Dette indebære afprøvning af Read/Write metoder, hvor vi prøver at læse og skrive værdier til databasen. Dernæst så afprøver vi Create metoderne, hvor vi opretter patienten i databasen, samt de tilhørende tabeller. Til sidst afprøver vi Rowcount metoden, hvor vi prøver, at tabellen stemmer overens med den fyldte Array.

Det kan ses, at vores Test klasser kører uden fejl, hvilket er ens betydende med, at vores metoder er funktionelle.

6 diskussion

6.1 Ting der er blevet ændret siden sidst

Vi har under revurdering af vores program, ment at vi gerne ville forsimple det en smule, samt holde DRY bedre. Derfor har vi lavet en klasse til simulering, en klasse til generelle metoder, sørget for at de hænger sammen og forsimplet lidt. Ud fra disse ændring lavede vi et nyt implementations klasse diagram 18.

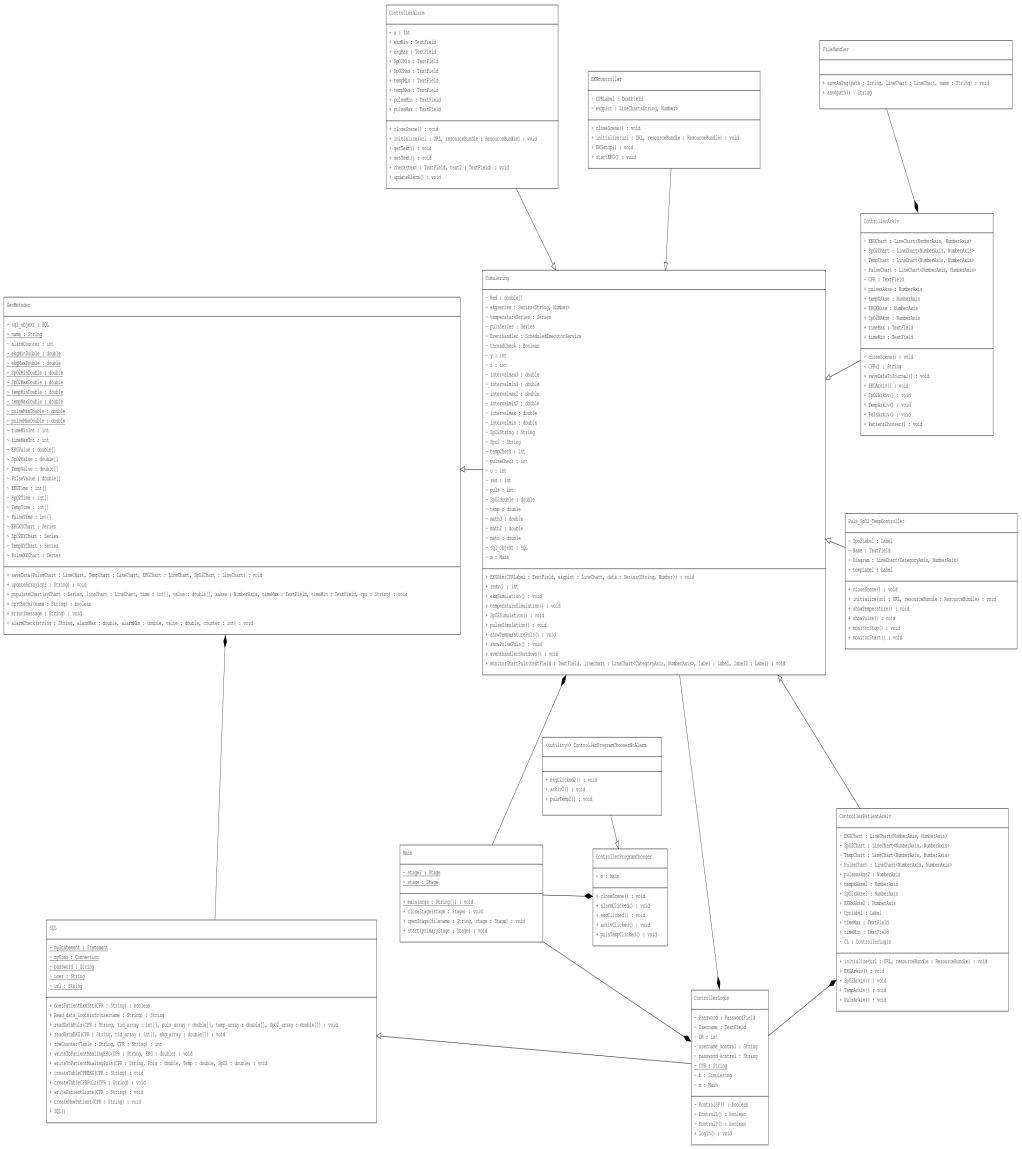

Figure 18: Implementations klasse diagram

6.2 dokumentering af ændringer

Vi lavede nogle få metoder meget om, da vi mente, at der var små fejl eller at de kunne skrives bedre. Her er en liste over hvad vi har ændret:

- SaveDataMetoden
- Metoderne der indsætter simulerede værdier på graferne i realtime
- Login metoderne
- Hvordan vi læser data til graferne (nu via. sql)

6.2.1 ControllerLogin

I controllerLogin klassen, ændrede vi på hvordan man loggede ind. Vi brugte de nye metoder i SQL-klassen, til at vurderer om username og password eksistererede, samt om patientens CPR eksisterede hvis den loggede ind. Fra patientens vinkel valgte vi, at man ikke skal skrive CPR da der optimalt skulle bruges nemID her i stedet, samtidig med, at vi mente at ens CPR er forholdsvist privat nok, til at det kan regnes som et password i sig selv. Ses på figuren 19.

```
    public void login() throws IOException {
        if (KontrolP()) {
            // patienter skal kun kunne tilgå deres arkiv.
            m.openStage( filename: "PatientArkiv.fxml", Main.stage);
        } else if (KontrolL()) {
            // læge skal kunne tilgå det hele
            m.openStage( filename: "ProgramChooser.fxml", Main.stage);
        } else if (KontrolSP()) {
            // sundhedspersonale skal kunne tilgå det meste, undtagen alarmgrænser.
            m.openStage( filename: "ProgramChooserNoAlarm.fxml", Main.stage);
        } else {
            b.error("Forkert adgangskode");
        }
    }

    private boolean KontrolP() {
        //Hvis dit CPR findes at PatientData folderen, kan du logge ind
        String U = Username.getText();
        if (doesPatientExsist(U)) {
            CPR = U;
            return true;
        } else {
            return false;
        }

    }

    private boolean KontrolL() {
        String U = Username.getText();
        String P = Password.getText();
        String s = Read_data_logininfo(U);

        if (s.equals("null")) {
            return false;
        } else {
            String[] data = s.split( regex: ",");
            username_kontrol = data[0];
            password_kontrol = data[1];
            DR = Integer.parseInt(data[2]);
            return U.equals(username_kontrol) && P.equals(password_kontrol) && DR == 1;
        }
    }

    private boolean KontrolSP() {
        String U = Username.getText();
        String P = Password.getText();
        String s = Read_data_logininfo(U);

        if (s.equals("null")) {
            return false;
        } else {
            String[] data = s.split( regex: ",");
            username_kontrol = data[0];
            password_kontrol = data[1];
            DR = Integer.parseInt(data[2]);
            return U.equals(username_kontrol) && P.equals(password_kontrol) && DR == 0;
        }
    }
}
```

Figure 19: ControllerLogin metoder opdateret

6.2.2 ControllerPatientArkiv

I patientArkivet, ændrede vi på hvordan data[] blev indlæst, dette gjorde vi, da den ikke længere fungere helt ligesom metoderne i Arkivet. Derfor kalder vi nu updateArray metoden inde i initialize. Ses på figuren 20.

```
59      @Override  //En patient skal kun kunne tilgå sine egne data, derfor bliver CPR automatisk overført
60      public void initialize(URL url, ResourceBundle resourceBundle) {
61          Cprlabel.setText(CL.CPR);
62          try {
63              updateArray(CL.CPR);
64          } catch (SQLException throwables) {
65              throwables.printStackTrace();
66          }
67          // populære charts fra start
68          try {
69              EKGArkiv();
70              PulsArkiv();
71              TempArkiv();
72              SpO2Arkiv();
73          } catch (FileNotFoundException | SQLException e) {
74              e.printStackTrace();
75          }
76      }
77  }
```

Figure 20: ControllerPatientArkiv metoder opdateret

6.2.3 FileHandler

I fileHandler klassen, slettede vi brugen at filer, pga. vi nu bruger SQL i stedet. Derefter overvejede vi brugervenligheden ved SaveAsPng metoden. Her vurderede vi at det ville være bedre, hvis man selv kunne vælge hvor filerne blev gemt, i stedet for at skulle finde hvor filerne blev gemt. Derfor lavede vi denne funktionalitet med JavaFX FileChooser klassen. Ses på figuren 21.

```
13  public class FileHandler {
14
15      public String savepath() {
16          FileChooser fc = new FileChooser();
17          File file1 = fc.showSaveDialog( window: null);
18          file1.mkdir();
19          return file1.getAbsolutePath();
20      }
21
22  @   public void saveAsPng(String path, LineChart lineChart, String name) { //Laver png billede
23          WritableImage image = lineChart.snapshot(new SnapshotParameters(), writableImage: null);
24          File file = new File( pathname: path + "/" + name);
25          try {
26              ImageIO.write(SwingFXUtils.fromFXImage(image, bufferedImage: null), formatName: "png", file);
27          } catch (IOException e) {
28              e.printStackTrace();
29          }
30      }
31  }
32
```

Figure 21: Filehandler metoder opdateret

6.2.4 GenMetoder

I genMetoder implementerede vi SQL i populateChart metoden, samt lavede den vitale updateArray, der sørger for at tilpasse længden på vores Array til den rette længde og indsætte værdierne. Ses på figurerne 22 og 23.

```
88      //metode til at finde data og indlæse det i grafer.
89      public void populateChart(XYChart.Series xyChart, LineChart lineChart,
90                                int[] time, double[] value, NumberAxis xakse,
91                                TextField timeMax, TextField timeMin, String cpr) throws SQLException {
92
93          if (sql_objekt.doesPatientExsist(cpr)) {
94              xyChart.getData().clear();
95              lineChart.getData().clear();
96
97              try {
98                  timeMaxInt = Integer.parseInt(timeMax.getText());
99                  timeMinInt = Integer.parseInt(timeMin.getText());
100             } catch (NumberFormatException e) {
101                 timeMaxInt = value.length;
102                 timeMax.setText(String.valueOf(timeMaxInt));
103                 timeMinInt = 0;
104                 timeMin.setText(String.valueOf(timeMinInt));
105             }
106
107             if (timeMaxInt > value.length) {
108                 timeMaxInt = value.length;
109                 timeMax.setText(String.valueOf(timeMaxInt));
110             }
111             for (int a = 0; a < timeMaxInt; a++) {
112                 xyChart.getData().add(new XYChart.Data(time[a], value[a]));
113
114             }
115             xakse.setUpperBound(timeMaxInt);
116             xakse.setLowerBound(timeMinInt);
117             lineChart.getData().add(xyChart);
118         } else {
119             error("Ugyldigt cpr");
120         }
121     }
```

Figure 22: populateChart metoden opdateret

```
123     public void updateArray(String cpr) throws SQLException {
124         EKGTime = new int[sql_objekt.rowCounter( Table: "patientMaalingEKG", cpr)];
125         EKGValue = new double[sql_objekt.rowCounter( Table: "patientMaalingEKG", cpr)];
126
127         PulseTime = new int[sql_objekt.rowCounter( Table: "patientMaalingPuls", cpr)];
128         TempTime = new int[sql_objekt.rowCounter( Table: "patientMaalingPuls", cpr)];
129         SpO2Time = new int[sql_objekt.rowCounter( Table: "patientMaalingPuls", cpr)];
130
131         PulseValue = new double[sql_objekt.rowCounter( Table: "patientMaalingPuls", cpr)];
132         TempValue = new double[sql_objekt.rowCounter( Table: "patientMaalingPuls", cpr)];
133         SpO2Value = new double[sql_objekt.rowCounter( Table: "patientMaalingPuls", cpr)];
134
135         sql_objekt.readDataEKG(cpr, EKGTime, this.EKGValue);
136         sql_objekt.readDataPuls(cpr, PulseTime, PulseValue, TempValue, SpO2Value);
137         TempTime = PulseTime;
138         SpO2Time = PulseTime;
139     }
```

Figure 23: updateArray metoden opdateret

6.2.5 Simulering

I simuleringsklassen rettede vi fejlen, af at man kunne starte flere tråde samtidigt når dataen skulle indføres på graferne. Dette gjorde vi bare med en boolean, der er true så længe tråden ikke kører. Dernæst indsatte vi selvfølgelig SQL funktionalitet. Dette ses på figurene 24, 25 26 og 27

```
public class Simulering extends GenMetoder {
    Main m = new Main();

    //Variabler til puls, temperatur,ekg simulation og fremvisning
    double math, math2, math3, temp, SpO2double;
    int puls, red, u, pulseCheck, tempCheck;
    String Spo2, SpO2String;
    double intervalmin = 70;
    double intervalmax = 70;
    double intervalmin2 = 98;
    double intervalmax2 = 100;
    double intervalmin3 = 36;
    double intervalmax3 = 38;
    int i = 0;
    int y = 0;

    Boolean threadCheck = true;
```

Figure 24: Simuleringsklassen

```
    public void monitorStartPuls(TextField textField, LineChart<CategoryAxis, NumberAxis> linechart,
                                 Label label, Label label2) {
        name = textField.getText();
        if (!threadCheck) {
            return;
        }
        if (cprCheck2(name)) {
            sql_objekt.createNewPatient(name);
            threadCheck = false;
            pulsSeries.setName("puls");
            temperatureSeries.setName("Temperature");
            linechart.getData().clear();
            linechart.getData().addAll(pulsSeries, temperatureSeries);
            Eventhandler = Executors.newSingleThreadScheduledExecutor();
            Eventhandler.scheduleAtFixedRate(() ->
                    Platform.runLater(() -> {
                        String bogstav = String.valueOf(i);
                        SpO2Simulation();
                        SpO2String += bogstav + " " + Spo2 + " ";
                        pulseSimulation();
                        temperatureSimulation();
                        if (pulseCheck == 1) {
                            pulsSeries.getData().add(new XYChart.Data(bogstav, puls));
                            alarmCheck( string: "ALARM PULS ER FARLIG", pulseMaxDouble, pulseMinDouble, puls, i);
                        }
                        if (tempCheck == 1) {
                            label2.setText((temp + "°C"));
                            temperatureSeries.getData().add(new XYChart.Data(bogstav, temp));
                            alarmCheck( string: "ALARM TEMPERATUR ER FARLIG", tempMaxDouble, tempMinDouble, temp, i);
                        }

                        label.setText(Spo2);
                        alarmCheck( string: "SPO2 ER FARLIG", SpO2MaxDouble, SpO2MinDouble, SpO2double, i);

                        sql_objekt.writeToPatientMaalingPuls(name, puls, temp, SpO2double);
                        i++;
                    }),  initialDelay: 0,  period: 1, TimeUnit.SECONDS);
        } else {
            error("Invalid input, ugyldigt CPR");
        }
    }

    //Metode til at stoppe fremvisning i realtid af puls, temp og spo2
    public void eventhandlerShutdown() {
        if (!threadCheck) {
            Eventhandler.shutdown();
            threadCheck = true;
        }
    }
```

Figure 25: Puls/SpO2/Temp metoden opdateret

```
87      //Metode til at stoppe fremvisning i realtid af puls, temp og spo2
88      public void eventhandlerShutdown() {
89          if (!threadCheck) {
90              Eventhandler.shutdown();
91              threadCheck = true;
92          }
93      }
```

Figure 26: Shutdown metoden opdateret

```
152 @   public void EKGSim(TextField CPRLabel, LineChart ekgplot, XYChart.Series<String, Number> data) {
153         name = CPRLabel.getText();
154         if (threadCheck = false) {
155             {
156                 return;
157             }
158         }
159         if (cprCheck2(name)) {
160             sql_objekt.createNewPatient(name);
161             threadCheck = false;
162             Eventhandler = Executors.newSingleThreadScheduledExecutor();
163
164             Eventhandler.scheduleAtFixedRate(() ->
165                     Platform.runLater(() -> {
166                                 ekgplot.getData().clear();
167                                 ekgSimulation();
168                                 String n = String.valueOf(y);
169                                 int redval = redv();
170                                 data.getData().add((new XYChart.Data<>(n, redval)));
171                                 ekgplot.getData().add(data);
172                                 alarmCheck( string: "EKG ER FARLIG", ekgMaxDouble, ekgMinDouble, redval, y);
173                                 sql_objekt.writeToPatientMaalingEKG(name, redval);
174
175                                 y++;
176                             }
177                     ), initialDelay: 0, period: 100, TimeUnit.MILLISECONDS);
178         } else {
179             error("Invalid input, ugyldigt Cpr");
180         }
181     }
```

Figure 27: EKGSim metoden opdateret

6.3 SQL Diskussion

Vi har lavet vores SQL databasen således, at man får 2 tables til hver patient. Dette har vi for at gøre det nemmere at organisere. Vi løber ikke ind i et limit, da man via MYSQL kan gemme næsten uendelige tables per database, her er grænsen størrelsen af databasen. Man kunne lave 2 tabels, der gemte alt data fra puls og EKG, men vi valgte at gøre det anderledes, da vi mente at 2 tabels ville medfører 2 alt for store tabels.

Vi lavede også 2 tabels, et til puls og et til EKG, da vi sampler disse værdier med forskellige hastigheder, og derfor ville være nødt til at sortere i dem. Derfor mente vi at det var bedre at gemme dem forskellige steder, da det også ville afspejle vores program, hvor vi tager værdierne forskellige steder.

Vi har lavet en metode til at skabe en connection til databasen og en til at slutte den samme connection, da vi gerne vil undgå, at connectionen er tændt konstant, da nogle evt. ville kunne misbruge dette. Derfor husker vil altid at slukke connectionen efter at den har været brugt. Dette betyder, at hvis vi skal skrive én række værdier til databasen, tænder vi lige før vi skriver til databasen og slukker lige efter. Dette medfører jo så, at når vi skriver data til databasen, så tænder vi og slukker vi en gang pr. registreret måling.

6.4 ændring af afprøvning

Vi har prøvet at udfører en unit test. Vi løb dog ind i et problem under fasen hvor vi ville skrive til vores grænseflade i simulering, da vi ikke kunne bruge javaFX indenfor Junit5. Derfor valgte vi at konkludere, at konsollen også er en grænseflade, så hvis vi kan skrive til den, burde vi også kunne skrive til JavaFX Linechart.

6.5 Nye brugere

Vores program fungere således, at nye brugere får givet et brugernavn og password af en administrator. Dette har vi valgt, at gøre således så vi skaber en professionel tilgang til valg af brugernavn, samt tvinger brugerne til at have et godt password, da administratoren kan generere det. Udover kommer der også en yderligere kontrol af, hvilke privilegier man har adgang til, da administratoren skal kontrollere brugeren.

7 Konklusion

Vi må konkludere at vi har lavet et program der virker, SQL databasen kunne godt have været bedre normaliseret, men den vil virke som den fungere nu. Man ville evt. kunne løbe ind i problemer hvis dataen ender med at fylde alt for meget, men det ville give problemer over alt. Ellers virker vores program godt og udfylder alle kravene. Idet at programmet nu hermed er i stand til at kunne opsamle data fra et måle apparat (simulering), gemme daten og til sidst at kunne præsentere daten vha. listener observer. Derudover har vi primært sat fokus, på selve kriterierne og ikke andre elementer, idet vi sigter mod at udvikle et MVP (Minimum Viable Product), så det er netop ikke relevant at gå op i andre aspekter som ikke indgår i selve kravene.

Vi har derudover også redegjort for database designet og de anvendes design patterns. Dette er gjort vha. en række relevante afsnit som SQL-diagram, hvori selve SQL strukturen tydeligt fremgår og forklares.

8 Literaturliste

References

[1] Jacob Nordfalk. *Objektorienteret programmering i Java udgave 6. - dækker Java 11.* Forlaget Globe A/S.

[2] Kendall Scott. *UML Explained.* Addison Wesley.

[3] Lars Ingesman *Introduktion til SQL - databaser på nettet.* Nyt Teknisk Forlag

[4] Datanamic.com *I*dentifying and Non-Identifying Relationships.

9 Bilag