CDIO del 1

Projektopgave forår 2017

Projektnavn: CDIO 2
Gruppe: 23
Afleveringsfrist: Lørdag d. 18. marts 2017

Denne rapport er afleveret via Campusnet (der skrives ikke under).
Denne rapport indeholder 16 sider ekskl. forside og bilag.


Christian Niemann, s165220


Mads Pedersen, s165204


Frederik Værnegaard, s165234


Viktor Poulsen, s113403

Danmarks Tekniske Universitet
DTU Compute
02324 Videregående programmering
Undervisere: Stig Høgh & Finn Gustafsson

1 Timeregnskab

I alt per deltager	
Christian Niemann	11
Frederik Værnegaard	11,5
Mads Pedersen	11,5
Viktor Poulsen	1
I alt for alle deltagere	35

Tabel 1: Timeregnskab pr. deltager

Opgave	I alt pr. opgave
Design	0
Impl.	21,5
Test	1,5
Dok.	12
Andet	0
Ialt	35,0

Tabel 2: Timeregnskab pr. opgave

Se Bilag 7.1 for en detaljeret oversigt

Indhold

2 Problemformulering

I dette CDIO delprojekt har vi fået stillet to opgaver. Den første er, at vi skal implementere funktionaliteten af en vægt GUI og et socket i et udleveret programskellet, hvor det overordnede design, et observer pattern, og selve det grafiske interface er lavet for os. Dette skal ende ud i, at hvis man starter GUI'en og åbner en forbindelse til socket, vil interaktionen fungere præcis som, hvis man arbejdede med den rigtige fysiske vægt.

Den anden opgave er at vi skal lave et program, der kan styre en afvejningsprocess på en Mettler vægt, hvor der først indhentes operatør data over vægten, og efterfølgende udføres en afvejning med bruttokontrol. I denne omgang skal vi forbinde programmet til vægtsimulatoren, men hvis man har lavet sin simulator godt nok, er det meningen at man senere kan forbinde til den rigtige vægt i stedet, og så fungerer det også der.

3 Analyse

3.1 Kravsspecifikation

Punkter fra FURPS+ uden krav er udeladt.

Functional

Krav - Vægt Simulator

R.1 Simulatoren skal simulere vægtens opførsel.

R.2 Simulatoren skal kunne modtage og sende kommandoer over TCP.

R.3 Simulatoren skal vise input fra tastatur på display.

R.4 Input skal kunne foregå samtidig.

R.5 Det skal være muligt at sendte følgende 9 kommandoer:

- S. Send stabil afvejning.
- T: Tarér vægten.
- D: Skriv i vægtens display.
- DW: Slet vægtens display.
- P111: Skriv max. 30 tegn i sekundært display.
- RM20 8: Skriv i display, afvent indtastning.
- K - Skifter vægtens knap-tilstand.
- B - Sæt ny bruttovægt.
- Q - Afslut simuleringen.

R.6 Følgende kommandoer skal kunne afvikles fra den simulerede brugergrænseflade på vægten:

- Tarér vægten.

- Sæt ny bruttovægt.
- Afslut simuleringen.

R.7 Simulatoren skal lytte på port 8000.

R.8 Programmet skal både afprøves via brugergrænsefladen og over netværket.

R.9 Simulatoren skal kunne virke som 'stand in' for den fysiske vægt.

Krav - Afvejningsprocessen

R.10 Afvejningsystemet skal kunne foretage afvejning med bruttokontrol.

R.11 Systemet skal registrere brugerid og batchid.

Implementering

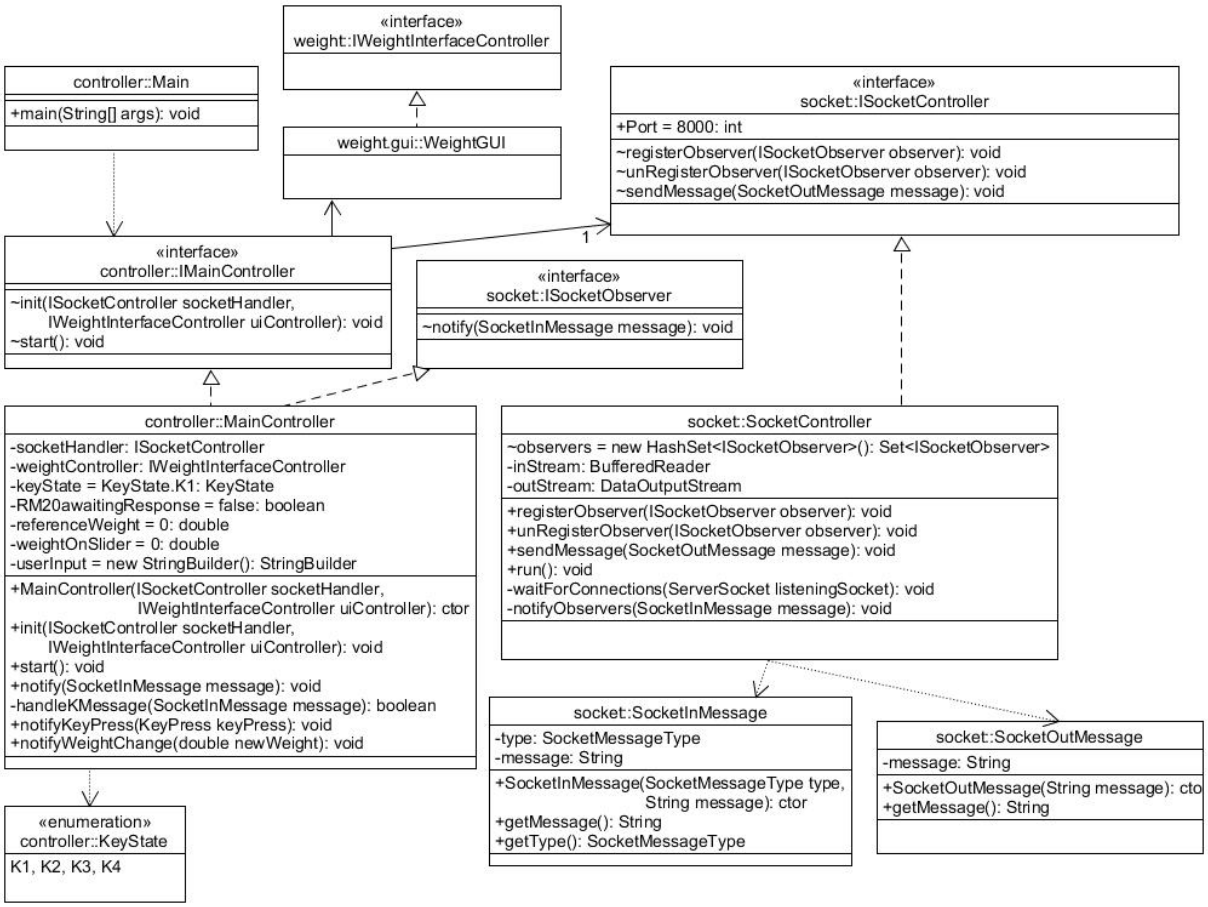
R.12 Programmet skal kunne køres på Windows maskinerne i databarerne på DTU.

4 Design

4.1 Design Klasse Diagram

De to følgende sektioner viser og beskriver de to design klasse diagram for henholdsvist den udleverede vægtsimulator - der er blevet implementeret - og den udarbejdede web server.

4.1.1 Vægt Simulator



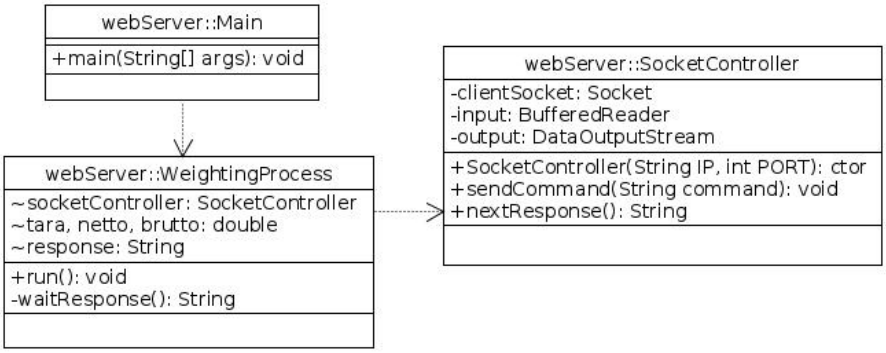
Figur 1: Design Klasse Diagram af Vægt Simulator

På Figur 1 ses design klasse diagrammet for vores vægt simulator. Selve klasse diagrammet er blevet konstrueret efterfølgende - og derfor "reverse engineered". Dette gør sig gældende for begge klasse diagrammer. Klasse diagrammet veksler dog ikke klasse-mæssigt, da de udleverede klasser i forvejen var oprettet - derfor er det kun attributer samt metoder der veksler fra det udleverede kode. Af Figur 1 kan det ses, hvordan klasserne er opdelt, hvilket også vises gennem pakkeopdelingen. Den samme opdeling kan tydligt ses, hvis man kigger på koden i main metoden. Her er det IMainController (fra controller-pakken), der modtager en IWeightInterfaceController (fra weight-pakken) og en ISocketController (fra socket-pakken), der vises i nedenstående stykke kode.

```
public static void main(String[] args) {
```

```
        ISocketController socketHandler = new SocketController();
        IWeightInterfaceController weightController = new WeightGUI();
        IMainController mainCtrl = new MainController(socketHandler, weightController);
        mainCtrl.start();
}
```

4.1.2 Web Server



Figur 2: Design Klasse Diagram af Web Server

Web serveren består af de tre klassser, Main, WeightingProcess og SocketController, der som ud fra deres navn til dels fortæller om deres ansvarsområde. Og det vises også tydeligt på billedet, hvordan forholdene mellem de tre klasser er.

5 Implementering

5.1 Kildekode

De følgende sektioner omhandler kun den implementerede del af koden.

5.1.1 Afvejningsprocess SocketController

SocketController har til ansvar, at skabe kommunikation med vægten. Som det ses i SocketController konstruktøren, skaber den en forbindelse til en adresse på en port. Vi skal både sende og modtage forbindelser, dette gør vi blandt andet, med BufferedReader og DataOutputStream, som vi initialisere i konstruktøren. Til at sende beskeder, bruger vi metoden sendCommand. sendCommand tager en String og sender den til vægten. Den sender output vha af writebytes metoden, hvorefter der bruges metoden flush, for at sikre, at alt buffered output bliver sendt. Vi printer så beskeden til brugeren, så man kan se hvilke kommandoer der bliver sendt. Denne metode bruger vi, når vi skal sende de kommandoer, der skal udføre noget på vægten. Til at modtage beskeder, bruger vi metoden nextResponse. Den læser det vi får tilbage fra vægten, dette kunne f.eks være, vægten af en given ting. Vi opretter en String nextLine, som sættes lig med det input vi får, vha af readLine. Vi tjekker så om nextLine ikke er lig null, og den vil så printe den respons vi får tilbage, hvis der er noget.

```
public SocketController(String IP, int PORT) throws IOException {
        clientSocket = new Socket(IP, PORT);
        input = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        output = new DataOutputStream(clientSocket.getOutputStream());
}
// Sends a command to the simulator
public void sendCommand(String command) throws IOException{
        output.writeBytes(command + "\r\n");
        output.flush();
        System.out.println("Bruger sendte kommandoen: " + command);
}
// Reads the response from the simulator
public String nextResponse() throws IOException {
        String nextLine = input.readLine();
        if(nextLine != null) {
                System.out.println(nextLine);
        }
        return nextLine;
}
```

5.1.2 waitResponse() i WeightingProcess

Der laves en metode, som har til formål, at vente et sekund på respons. Metoden har en string som vi kalder response, som læser respons vha af nextResponse metoden i SocketController. Vi bruger så en while løkke, til at tjekke, om der er noget respons, hvis ikke, venter den et sekund, hvorefter den igen læser respons fra vægten og sætter det lig respons. På denne måde sikrer vi, at vi modtager al respons fra vægten, så koden kan blive eksekveret korrekt.

```
private String waitResponse() throws IOException, InterruptedException {
        String response = socketController.nextResponse();
        while(response == null) {
                TimeUnit.MILLISECONDS.sleep(1);
                response = socketController.nextResponse();
        }
        return response;
}
```

5.1.3 run() i WeightingProcess

På nedenstående kode, ses vores run metode. Vi bruger først den SocketController konstruktør, vi lavede i SocketController klassen. Vi opretter her forbindelse til localhost, som opretter forbindelse til f.eks vægt simulatoren, som bliver kørt på samme computer. Det har været et krav, der bliver oprettet forbindelse på port 8000. Hvis det ikke lykkedes, at oprette forbindelse, vil der forekomme en IOException, som vi så catcher og udskriver, hvis det ikke lykkes at oprette forbindelse. Afvejningsprocessen fungerer på den måde at, vi sender en kommando vha af sendCommand() og vægten vil udføre den kommando der bliver sendt. Der vil herefter bliver ventet på input fra vægten.

```
public void run() {
        try {
                socketController = new SocketController("localhost", 8000);
        } catch (IOException e) {
                System.out.println("Connection couldn't be established!");
                e.printStackTrace();
                System.exit(1);
        }
```

Hvis vi tager udgangspunkt i processen, på nedenstående kode, som er en del af run metoden, ses det at vi først starter en while løkke. Dette gør, at når programmet slutter, vil processen blive kørt igen og man vil kunne lave flere afvejninger, i samme session. Herefter sendes kommandoen K 3. Dette vil sætte vægtens keystate, K 3 udfører ikke funktionen på vægten, men sender funktionskoden over netværket. Vi venter så på respons fra vægten, i dette tilfælde er det en acknowledgment der bliver sendt tilbage. Vi beder så vægten om at udskrive "Opr nr?" så det bliver vist på displayet. Her bliver det sendt vha af en RM20 som skriver i displayet, her må der kun skrives 7 karakter. Vi venter så på respons fra vægten. Der vil komme både Opr Nr og en acknowledgment tilbage, derfor er der to waitResponse() i træk. Vægten henter så navnet på personen med det Opr Nr der er indtastet og det bliver udskrevet på vægten. I dette tilfælde er det "Anders And [->". Dette bliver sendt vha af en P111, som udskriver på displayet med maks 30 karakter. Brugeren skal bekræfte dette, ved at trykke på [-> knappen på vægten og der vil blive sendt en kvittering, her modtager vi igen to respons, en fra kvitteringen og en acknowledgment. Vi gør nu det samme, når der skal vælges Batch. Herefter sendes en P111 kommando som resetter vægtens display.

```
try {
                        //Sends the desired commands and receives response
                        while(true){
                        socketController.sendCommand("K 3");
                        waitResponse();
                        socketController.sendCommand("RM20 8 \"Opr Nr?\" \"\" \"&3\"");
                        waitResponse();
                        waitResponse();
                        socketController.sendCommand("P111 \"Anders And [->\"");
                        waitResponse();
                        waitResponse();
                        socketController.sendCommand("RM20 8 \"Batch?\" \"\" \"&3\"");
                        waitResponse();
                        waitResponse();
                        socketController.sendCommand("P111 \"Salt [->\"");
                        waitResponse();
                        waitResponse();
                        socketController.sendCommand("P111 \"\"");
                        waitResponse();
```

Nu sendes sendes kommandoen T, som tararer vægten. Vi sender en P111 kommando, der skal vise "Placer tara" på displayet. Når tara er blevet placeret, sender vi kommandoen S som gemmer den vægt, som tara har. Problemet med dette er,at vi modtager f.eks "T S 1,4 kg" og vi ønsker at gemme det som en int, så vi kan validere til sidst om det er korrekt afvejet. Derfor bruges metoden split, til kun at gemme tallet 1,4. Et andet problem er, at i java kan skal decimaler skrives med punktum og ikke komma. Derfor bliver vi nødt til, at erstatte komma med punktum. Vi kan nu gemme talet, så vi kan validere senere. Vi beder nu brugeren placere netto og udfører samme process, som da vi udførte tara. Brugeren bliver nu bedt om, at fjerne brutto og denne værdi vil også blive gemt, her vil det være en negativ int, som er den samlede vægt er netto og tara. Vi laver så et tjek om det er korrekt afvejet, ved at tage tara, netto og brutto og lægge sammen, og hvis det giver 0 i alt, er det

afvejet ok. Dette vil blive vist på displayet. Under denne process, kan der forekomme IOException og InterruptedException, disse catcher og printer hvis afvejningsprocessen fejler.

```
                        socketController.sendCommand("T");
                        waitResponse();
                        socketController.sendCommand("P111 \"Placer tara [->\"");
                        waitResponse();
                        waitResponse();
                        socketController.sendCommand("S");
                        response = waitResponse();
                        tara = Double.parseDouble(response.split(" ")[7].replace(',', '.'));
                        socketController.sendCommand("T");
                        waitResponse();
                        socketController.sendCommand("P111 \"Placer netto [->\"");
                        waitResponse();
                        waitResponse();
                        socketController.sendCommand("S");
                        response = waitResponse();
                        netto = Double.parseDouble(response.split(" ")[7].replace(',', '.'));
                        socketController.sendCommand("T");
                        waitResponse();
                        socketController.sendCommand("P111 \"Fjern brutto [->\"");
                        waitResponse();
                        waitResponse();
                        socketController.sendCommand("S");
                        response = waitResponse();
                        brutto = Double.parseDouble(response.split(" ")[7].replace(',', '.'));
                        // Makes sure that it has been weighted correctly
                        if(tara + netto + brutto == 0) {
                                socketController.sendCommand("P111 \"Afvejning ok");
                                waitResponse();
                        } else {
                                socketController.sendCommand("P111 \"Afvejning ikke ok");
                                waitResponse();
                        }
                        }
                } catch (IOException | InterruptedException e) {
                        e.printStackTrace();
                        System.out.println("Afvejningsprocessen fejlede.");
                }
        }
```

5.1.4 Main i afvejningsprocessen

Vores Main klasse skaber en instans af vores WeightProcess klasse, hvorefter den kører metoden run().

```
public static void main(String[] args) {
                WeightingProcess process = new WeightingProcess();
                process.run();
        }
```

5.1.5 sendMessage() i SocketController

Når vi skal sende en besked til vores vægt gør vi hovedsageligt brug af denne metode (sendMessage()), der modtager en SocketOutMessage som inputparameter. Inputtet overføres og skrives til vores output stream vha. metoden writeBytes(), hvorefter output streamen flushes som ses i nedenstående stykke kode. Er der ikke oprettet nogen forbindelse til en socket, vil der i så fald fanges en IOException når vi forsøger at skrive til og flushe vores output stream. Da printer vi vores stack trace, samt sender en besked videre om at der ikke er tilsluttet en socket. Dette gøres vha. notifyObservers(), der modtager en SocketInMessage, hvortil vi ad hoc har oprettet en SocketMessageType, som vi kalder 'Error'.

```
public void sendMessage(SocketOutMessage message) {
        try {
                //TODO send something over the socket!
                outStream.writeBytes(message.getMessage() + "\r\n");
                outStream.flush();
        } catch (IOException e) {
                //TODO maybe tell someone that connection is closed?
                e.printStackTrace();
                notifyObservers(new SocketInMessage(SocketMessageType.Error, "No socket is connected!"));
        }
}
```

5.1.6 run() & waitForConnections() i SocketController

Da SocketController implementerer interfacet ISocketController som extender Runnable er vi nødt til at have en ***run()*** metode. Metoden var dog i forevejen implementeret, men hertil har vi tilføjet en fejlbesked tilsvarende beskrevet i sendMessage() metode, hvis vi fanger en IOException. Her gør vi igen brug af vores egen SocketMessageType kaldet 'Error' som det ses i nedenstående stykke kode.

```
public void run() {
        //TODO some logic for listening to a socket //(Using try with resources for auto-close of socket)
        try (ServerSocket listeningSocket = new ServerSocket(Port)){
                while (true){
                        waitForConnections(listeningSocket);
                }
        } catch (IOException e1) {
                notifyObservers(new SocketInMessage(SocketMessageType.Error, "Could not establish
                    connection to the cloud!"));
                e1.printStackTrace();
        }
}
```

waitForConnections() er en metode der håndterer input og output mellem den forbindelse der er oprettet til vores socket. Metoden bliver kørt af run() og baserer sig på de beskeder der kommer fra socketens input stream. Beskeder som kommer fra input streamen bliver differentieret udfra den første del af beskeden, der ender i en switch-case, som udgør de netop 8 forskellige kommandoer specificeret i opgavebeskrivelsen.

Modtager vi en besked, hvoraf første del af beskeden er 'RM20' vil dette blive oplyst til vores MainController og der vil blive svaret med en returbesked, 'RM20 B'. Returbeskeden sendes vha. sendMessage() som en SocketOutMessage til vores output stream. For at oplyse MainControlleren om at der er modtaget en besked gør vi brug af notifyObservers(), der sender en SocketInMessage videre, hvor vi oplyser SocketMessageType (RM208) samt det resterende interessante af beskeden - f.eks. ved inputtet »RM20 8 'OPR NR' " '&3'« er vi nu allerede klar over beskedtypen og sender derfor kun »'OPR NR' " '&3'« (vha. substring()) videre til MainControlleren.

```
case "RM20": // Display a message in the secondary display and wait for response
        notifyObservers(new SocketInMessage(SocketMessageType.RM208, inLine.substring(8)));
        sendMessage(new SocketOutMessage("RM20 B"));
        break;
```

Det samme gør sig også gældende for de andre cases - og er derfor helt generelt. Det er derfor ikke beskrevet for de andre cases, da det til dels er den samme overordnede måde vi håndterer inputs. I nogle tilfælde oplyser vi kun MainControlleren med notifyController() og udelader sendMessage(), da dette håndteres ovre i MainControlleren. Dette gøres hovedsageligt når vi skal udskrive afvejninger, da MainControlleren indeholder disse nødvendige variable til udregninger.

```
case "S": // Request the current load
        notifyObservers(new SocketInMessage(SocketMessageType.S, null));
        break;
```

Et problem ved at vi bruger substring() i stedet for split() er dog at vi forventer beskederne at være på en bestemt form. Dette vil kunne kaste en IndexOutOfBoundsException, hvis beskeden er for kort, eftersom substring(x) er forudbestemt. I disse tilfælde fanger vi blot den Exception og sender beskeden 'ES'. Eksempel på dette kan ses i nedenstående kode.

```
case "P111": //Show something in secondary display
        try {
                notifyObservers(new SocketInMessage(SocketMessageType.P111, inLine.substring(5)));
                sendMessage(new SocketOutMessage("P111 A"));
        } catch (IndexOutOfBoundsException e) {
                e.printStackTrace();
                sendMessage(new SocketOutMessage("ES"));
        }
        break;
```

Modtages der en besked der ikke svarer til nogle af de specifikke kommandoer vil der blot blive sendt en fejlbesked 'ES' retur.

```
default: //Something went wrong?
        sendMessage(new SocketOutMessage("ES"));
    break;
```

5.1.7 notify() i MainController

Metoden notify() har vi implementeret således at de forskellige beskeder, der sendes til vægten - fra en hvilken som helst tilknyttet process - udføres som beskrevet i både opgavebeskrivelsen samt kravspecifikationen. Metoden tager en besked af typen SocketInMessage som inputparameter, der ender i en switch-case, som afgør udførslen alt efter beskedtype.

Sender vi f.eks. beskeden 'B 1.0' vil vi ramme nedenstående stykke kode. Her gør vi brug af notifyWeightChange(), der vil ændre bruttovægten til 1.0 kg. Beskeden der sendes vil omskrives til et kommatal, der bruges som inputparameter i notifyWeightChange() metoden. Lykkedes dette af en eller anden årsag ikke vil vi fange en Exception, der udskriver en StackTrace, samt en besked til processen, der sendte B-beskeden.

```
case B:
        try {
            notifyWeightChange(Double.parseDouble(message.getMessage()));
        } catch (Exception e) {
                e.printStackTrace();
            socketHandler.sendMessage(new SocketOutMessage("ES"));
        }
        break;
```

Beskeden 'D' beskrives ikke, da den i forevejen var implementeret. Den næste implementerede case i vores switch-case er beskeden Q, som vil afslutte programmet. Her gør vi blot brug af 'System.exit(0)', som netop terminerer den igangværende Java Virtual Machine. Tallet nul, der er inputparameter indikerer blot at programmet termineres under normale omstændigheder

```
case Q:
        System.exit(0);
        break;
```

Beskeden 'RM20' forekommer i to forskellige varianter. 'RM204' og 'RM208', hvor vi har valgt kun at implementere 'RM208', da det er den eneste af de to, som bliver omtalt i opgavebeskrivelsen - og derudover udfører de til dels det samme. I koden sender vi en besked ud til operatøren via showMessagePrimaryDisplay(), hvorefter vi sætter en boolsk værdi til 'true', der anvendes til at afvente en reaktion fra operatøren. Denne variabel har vi selv tilføjet.

```
case RM204:
        //TODO Not implemented yet
        break;
case RM208:
        weightController.showMessagePrimaryDisplay(message.getMessage());
        RM20awaitingResponse = true;
        break;
```

For at foretage en afvejning kan vi sende beskeden 'S' til vores vægt over netværket, der vil svare med en besked, hvor nettovægten vil indgå - altså det der står i displayet. Vi sender altså blot en besked via sendMessage(), der modtager en SocketOutMessage, som anvender to variable, der bruges til at udregne differensen (nettovægten). Dette vil returnerer beskeden 'S S X.XXX kg'. De to variable 'referenceWeight' og 'weightOnSlider' er tilføjet for at holde styr på den forrige belastning som referenceværdi og den nuværende belastning.

```
case S:
        socketHandler.sendMessage(new SocketOutMessage("S S       " + new
            DecimalFormat("#.###").format(weightOnSlider-referenceWeight) + " kg"));
        break;
```

På samme måde kan vi tarére vægten således at vi nulstiller vægtens belastning til 0.000 kg. For at tarére kan vi altså sende beskeden 'T' til vores vægt, som vil svare med en besked, hvori den nuværende belastning vil indgå. Her gemmer vi den nuværende belastning i en variabel, 'referenceWeight', hvorefter vi nulstiller displayet ved showMessagePrimaryDisplay("0.0 kg") og dernæst sender vi en besked retur med den nulstillede belastning via sendMessage(), der modtager en SocketOutMessage. Dette udskrives således 'T S X.XXX kg'.

```
case T:
        referenceWeight = weightOnSlider;
        weightController.showMessagePrimaryDisplay("0.0 kg");
        socketHandler.sendMessage(new SocketOutMessage("T S " + new
            DecimalFormat("#.###").format(referenceWeight) + " kg"));
        break;
```

Lignende at tarére vægten kan vi nulstille vægten, dog uden at gemme et nulpunkt (referencepunkt). Til dette kan vi sende beskeden 'DW', der ikke vil gemme den nulstillede belastning, men altså blot nulstille displayet og svare med beskeden 'DW A'. I nedenstående kode nulstiller vi blot displayet, da returbeskeden er implementeret i SocketController klassen.

```
case DW:
        weightController.showMessagePrimaryDisplay(referenceWeight + " kg");
        break;
```

Ønsker vi da at ændre vægtens knap-tilstand kan vi sende en af følgende fire beskeder 'K 1', 'K 2', 'K 3' eller 'K 4', som er yderligere beskrevet i opgavebeskrivelsen. Afhængig af tilstanden vil funktionstasterne på vægten udføre deres funktion eller ej, samt sende funktionskoden retur eller ej. Dette giver os de fire kombinationer. Vi har da ændret handleKMessage() til at returnere en boolsk værdi alt efter om tilstanden er blevet ændret eller ej. Lykkedes det at ændre tilstanden vil vi da returnere beskeden 'K A' - og i værst tilfælde returnere 'ES'.

```
case K:
        if(handleKMessage(message)) {
        socketHandler.sendMessage(new SocketOutMessage("K A"));
        } else {
                socketHandler.sendMessage(new SocketOutMessage("ES"));
        }
        break;
```

Den sidste case er 'P111', der udskriver en besked til operatøren i det sekundære display. Dette håndteres blot ved at bruge showMessageSecondaryDisplay() metoden. Vi kan da udskrive en hvilken som helst besked af den maksimale størrelse på 30 tegn.

```
case P111:
        weightController.showMessageSecondaryDisplay(message.getMessage());
        break;
```

5.1.8 notifyKeyPress() i MainController

notifyKeyPress er en metode i vægtsimulatorens MainController, den bliver kaldt af WeightGUI, når der bliver trykket på en knap på GUI'en, fordi MainControlleren er registreret som observer hos WeightGUI.

notifyKeyPress-metodens opgave er at håndtere hvad der skal ske, når der bliver trykket på de forskellige knapper.

KeyState er 4 forskellige tilstande vægten kan være i. Funktionsknapperne er aktive i K1 og K4, og der sendes funktionskoder over socket i K3 og K4. Mere vil der ikke blive snakket om KeyState i dette afsnit.

Metoden består af en switch case, der tjekker hvilken type keyPress der bliver sendt fra WeightGUI.

Vi vil herunder gennemgå hvad der sker ved tryk på de forskellige knapper:

Når der trykkes på funktionstasten TARA, gemmes vægtens nuværende belastning, i en variabel vi har kaldt referenceWeight, og tallet der bliver vist i diplay'et, bliver vægtens nuværende belastning minus referencevægten, altså 0 kg.

```
case TARA:
        if(keyState.equals(KeyState.K1) || keyState.equals(KeyState.K4)) {
                referenceWeight = weightOnSlider;
                weightController.showMessagePrimaryDisplay(weightOnSlider - referenceWeight + " kg");
        }
        break;
```

Når der trykkes på et tal eller et bogstav på vægtens tastatur, sendes der et keyPress af typen TEXT, som indeholder en karakter. Da alle karakterer på denne måde kommer en ad gangen, er vi nødt til at holde den samlede sekvens af karakterer i MainControlleren. Til dette har vi instantieret en string builder, som vi gemmer karaktererne i. Hver gang vi har tilføjet til string builderen, viser vi dens nuværende indhold i det sekundære display.

```
case TEXT:
        userInput.append(keyPress.getCharacter());
        weightController.showMessageSecondaryDisplay(userInput.toString());
        break;
```

Når der trykkes på funktionstasten ZERO, nulstilles referencevægten, og referencevægten(0 kg) vises i vægtens primære display. Så snart brugeren begynder at trække i slideren(belaste vægten), vil det være slider værdien der vises i display'et, al tarering vil altså være nulstillet.

```
case ZERO:
        if(keyState.equals(KeyState.K1) || keyState.equals(KeyState.K4)) {
                referenceWeight = 0;
                weightController.showMessagePrimaryDisplay(referenceWeight + " kg");
        }
        break;
```

Når der trykkes på C-tasten, har vi kodet funktionaliteten til at det fungere som et backspace, så det kun er sidste bogstav i string builderen der bliver fjernet hver gang man trykker. Vi ved ikke helt om det er sådan den rigtige vægts C-tast fungerer, men da det er så omstændigt at skrive bogstaver med en num pad, synes vi det er mest brugervenligt, at man ikke er nødt til at slette alt hvad man har skrevet, hvis man kun har skrevet et enkelt tegn forkert.

```
case C:
        userInput.deleteCharAt(userInput.length()-1);
        weightController.showMessageSecondaryDisplay(userInput.toString());
        break;
```

Når der trykkes på funktionstasten EXIT, har vi kodet programmet til bare at afslutte som et ikke fejlbehæftet forløb.

```
case EXIT:
        System.exit(0);
        break;
```

Når der trykkes på funktionstasten SEND, er det afhængigt af hvilken tilstand vægten er i, hvad der sker. Vi har vores KeyState, men også en tilstand vi har indført, der hedder RM20awaitingResponse, som har højere prioritet end KeyState. Hvis RM20awaitingResponse er sat til true(af en RM20-kommando), vil et tryk på SEND resulterer i, at indholdet af string builderen bliver sendt som svar over socket, string builderen og display bliver nulstillet, og RM20awaitingResponse bliver sat til false, da den nu har fået sit svar. Hvis man ikke er i RM20awaitingResponse tilstand, vil et tryk på SEND resultere i at en acknowledgement bliver sendt over socket, ud fra hvilken KeyState vægten er i. Vi er af den opfattelse, at det er sådan den rigtige vægt gør, men vi har ikke haft mulighed for at få det afprøvet, inden aflevering. Det kan nemt justeres senere, når vi får mulighed for at låne vægten igen, hvis det ikke er korrekt.

```
case SEND:
        if(RM20awaitingResponse) {
                socketHandler.sendMessage(new SocketOutMessage("RM20 A \"" + userInput.toString() +
                    "\""));
                userInput.setLength(0); display.
                weightController.showMessageSecondaryDisplay(userInput.toString());
                RM20awaitingResponse = false;
        } else if (keyState.equals(KeyState.K3)) {
                socketHandler.sendMessage(new SocketOutMessage("K A 3"));
        } else if(keyState.equals(KeyState.K4)) {
                socketHandler.sendMessage(new SocketOutMessage("K A 4"));
        }
        break;
```

5.1.9 notifyWeightChange() i MainController

notifyWeightChange-metoden minder meget om notifyKeyPress-metoden, som er bekrevet i afsnit 5.1.8, og bliver også kaldt af samme klasse. Forskellen er bare, at i stedet for ved et taste tryk, bliver denne metode kaldt hver gang der rykkes på slideren på GUI'en.

Måden vi har implementeret metoden på, er ved at hver gang metoden bliver kaldt(slideren bliver rykket), skriver vi i vægtens primære display, hvad slideren er rykket til minus referencevægten. Dette gør at den også viser den korrekte vægt i displayet, hvis der er bleveret tareret forinden. Hvis der ikke er blevet tararet, vil referencevægten være 0.0, og da vises bare hvad slideren står på.
I den sidste linje af metoden, gemmer vi hvad slideren står på, i en lokal variabel, da vi skal have adgang til det tal, hvis der over socket bliver bedt om at modtage en stabil afvejning(en S-kommando).

```
public void notifyWeightChange(double newWeight) {
        weightController.showMessagePrimaryDisplay(newWeight - referenceWeight + " kg");
        weightOnSlider = newWeight;
}
```

5.2 Argumentation for test

Grundet et stramt program med andre opgaver sideløbende med CDIO del 2, har vi måttet prioritere vores tid, og har ikke kunnet finde tid til at lave et formelt test afsnit. Så vores test af denne del af projektet, har bestået af uformelle test, sideløbende med udviklingen af vægt simulator og web server. Vi har under disse test konstateret, at vores overordnede design, implementering og flow fungerer, men at der blot er lidt finjusteringer der skal tilpasses, når vi kan få lov at låne vægtens igen, så vi kan få vores simulator til at give 100% samme output, som den rigtige vægt.

6 Konklusion

Vi har fået implementeret alt den ønskede funtionalitet i vægtsimulatoren, så den opfører sig ligesom den rigtige vægt. Vi mangler blot at låne den fysiske vægt, så vi kan finjustere kommandoerne vores simulator sender, så det er præcis de samme strings osv. Det bliver gjort snarest, så vi for fremtiden kan arbejde med vores simulator, og være sikre på, at hvad vi laver også vil fungere med den rigtige vægt.

Vi har også fået lavet et system, der kommunikerer med vores vægtsimulator over et socket, og korrekt styrer en afvejningsprocess ved at indhente operatør data og efterfølgende laver en afvejning med bruttokontrol. På nuværende tidspunkt er systemet ikke lavet med validering af om operatøren gør hvad han for besked på, men det kan let implementeres efterfølgende. Vores system agerer på nuværende tidspunkt på hvilket output vores simulator giver, hvis det output bliver justeret, skal programmet der styrer afvejningsprocessen justeres tilsvarende. Ligeledes er en del funktionalitet, som fx login, på nuværnede tidspunkt hard coded, så den process skal også implementeres senere. Men vi har lavet et overordnet framework, der kan håndtere kommunikationen med en vægt eller en vægtsimulator, vi anser derfor denne delopgave som løst til de nuværende specifikationer.

7 Bilag

7.1 Tidsregistrering

Dato	Deltager	Design	Impl.	Test	Dok.	Andet	I alt
11/03/2017	Christian Niemann		1				1,0
12/03/2017	Christian Niemann		2,5				2,5
13/03/2017	Christian Niemann		3				3,0
14/03/2017	Christian Niemann		1	0,5	2		3,5
16/03/2017	Christian Niemann				1		1,0
12/03/2017	Frederik Værnegaard		2,5				2,5
13/03/2017	Frederik Værnegaard		3				3,0
14/03/2017	Frederik Værnegaard		1	0,5			1,5
15/03/2017	Frederik Værnegaard				3		3,0
16/03/2017	Frederik Værnegaard				1,5		1,5
11/03/2017	Mads Pedersen		1,5		1		2,5
13/03/2017	Mads Pedersen		4				4,0
14/03/2017	Mads Pedersen		1	0,5			1,5
15/03/2017	Mads Pedersen				2,5		2,5
16/03/2017	Mads Pedersen				1		1,0
11/03/2017	Viktor Poulsen		1				1,0

Tabel 3: Detaljeret Timeregnskab

7.2 Kildekode

Al kildekode samt udviklingshistorik kan gennemgås på https://github.com/DTU23/CDIO2