CDIO del 2

Group 12

16. Marts 2018

Mathias
Fager
s175182

Sebastian
Stokkebro
s170423

Niklaes
Jacobsen
s160198

Alan
Jafi
s122417

Burhan
Shafiq
s175446

Indhold

Tidsforbrug

Herunder ses vores tidsforbrug på projektet, opdelt pr. pers. angivet i hele timeantal

Navn	**Analyse**	**Design**	**Implementering**	**Projektplan**	**Totalt**
Alan	4	5	4	0	13
Burhan	4	4	6	0	14
Mathias	4	5	11	0	20
Niklaes	3	5	13	5	26
Sebastian	4	7	12	0	23

1 Introduktion

I dette projekt vil vi beskæftige os med udviklingen af et afvejnings system. Systemet vil blive baseret på kundens vision om et system "som kan foretage afvejning med bruttokontrol". Systemet har et krav på at registrere brugerID og batchID. Løsningsforslaget, som vi skal arbejde med, kræver en Mettler vægt, hvor fra systemet vil hente operatør data samt styre afvejningerne på. Vi vil i denne rapport uddybbe de tanker vi har haft under de forskellige faser, og hvilke ændringer/overvejelser vi har foretaget os, jo længere i udviklings processen vi kom. Til sidst vil vi afslutte med en konklusion, hvor vi vil opsummere alt, hvad vi har lavet under dette projekt og, hvilke forbedringer/ændringer der kunne laves til det næste projekt.

2 Analyse

2.1 Kravspecifikation

I sammenhæng med analysen, har vi gennemgået de krav, som skal opfyldes, for at have et funktionsdygtigt program.

- Programmet skal kunne forbindes til vægten.
- En bruger skal have et brugernummer i området 11 - 99.
- Der skal hardcodes følgende i stedet for database:
  - en bruger: nr. 12 "Anders And"
  - et batch: nr. 1234 "Salt"
- Man skal kunne køre et "afvejningsflow."

3 Design

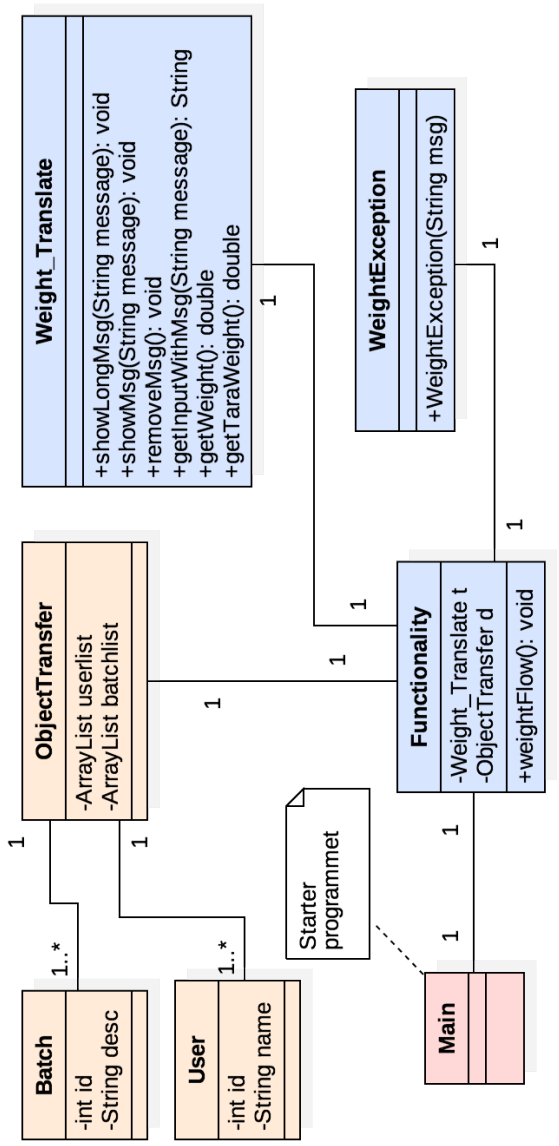
3.1 Design Overvejelser

Vores design overvejelser har først og fremmest ligget i hvordan programmets arkitektur skal se ud. Vi bruger 3-lags-modellen, som skal fungere mellem brugergrænsefladelaget, funktionalitetslaget og datalaget. Brugergrænsefladen er vægtsimulatoren, funktionalitetslaget er det lag, som håndterer kommunikationsflowet og "oversættelsen"af I/O mellem datalaget og bruger. Samtidig har vi sørget for at der er en `WeightException`, som tager sig af I/OExceptions fra vægten. Så i stedet for blot at modtage et `"L"` som fejlbesked, kommer der en reel besked, som brugeren forstår. Datalaget består af en `User` og en `Batch` klasse som er opskriften til den datainformation vi skal bruge. `ObjectTransfer`-klassen indeholder to `ArrayLists` med users og batches, da det er denne klasse der håndterer kommunikationen mellem selve dataen og funktionaliteten.

Vi har udover disse overvejelser lavet et design klassediagram, som skal vise forbindelserne mellem de forskellige klasser. Vi har også lavet et pakkediagram der viser hvordan klasserne er opdelt i de forskellige pakker der indgår i systemet.

3.2 Design klassediagram

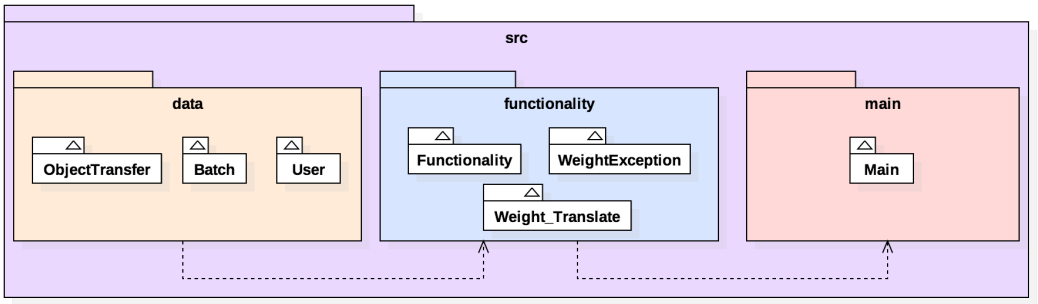
Herunder ses det designklassediagram vi har implementeret i vores projekt.



Vores designklassediagram viser her forbindelserne mellem klasserne. Vi kan bl.a. se at klasserne `User` og `Batch` vil der fremkomme flere instanser af i `ObjectTransfer`. De her instanser, har `Functionality` mulighed for at interagere med og hermed vil vi opnå det flow, som er en del af vores krav-specifikationer.

3.3 Pakkediagram

Herunder ses pakkediagrammet over programmet. Der vises ikke oversigt over vores test-lag



Vi kan ud fra vores pakkediagram se opdelingen af klasserne i systemet. `Data`-pakken indeholder alle de klasser der håndterer direkte data. Dvs. de klasser der definerer og giver direkte adgang til dataen, ligger i dette lag.

`Functionality`-pakken indeholder de funktionalitetsklasser, der er i systemet. Altså de klasser der håndeterer og "oversætter"den data der indhentes fra datapakken.

`Main`-pakken er blot den pakke der indeholder vores `Main`-klasse, som starter programmet op, og skaber forbindelse til vægtsimulatoren.

4 Implementering

Vores implementeringsovervejelser har ligget i hvordan det ville være muligt at skabe forbindelse til den virtuelle vægt. Vi har valgt at bruge en socket, der åbner en telnet adgang til vægten via en lokal IP-adresse og den tilpassende port. Gennem nogle objekt-streams, er det muligt at skrive og læse til og fra vægten. Her er en kort gennemgang af hvordan koden ser ud.

4.1 Kode gennemgang

Først deklereres navne på objekterne vi skal bruge. Derefter initialiseres disse i konstruktøren `Weight_Translate`. Det er altså her vi skal bruge IP-addressen og porten, som skal benyttes til adgang af vægten - IP'en fås fra `Main` som argument til `Weight_Translate`-klassens kontruktør. Den virtuelle vægt bruger den lokale IP-addresse, hvor den fysiske vægt bruger en

anden specifik IP-addresse. Herefter initialiseres `writer` og `reader` med hhv. `socket` output stream og input stream.

```
public class Weight_Translate {

    // declare socket to open connection to TCP/Telnet
    // declare Writer and reader for I/O
    private Socket socket;
    private PrintWriter write;
    private BufferedReader read;

    private final int WEIGHT_PORT = 8000;

    public Weight_Translate(String ip) {

      try {

         // create socket connection with ip and port, delivered
             from Main
         socket = new Socket(ip, WEIGHT_PORT);

         // initialize the writer and the reader with the socket
             output and input stream
         write = new PrintWriter(socket.getOutputStream(), true);
         read = new BufferedReader(new
             InputStreamReader(socket.getInputStream()));

         // catch of exceptions

      } catch (IOException e) {

         e.printStackTrace();
         return;
      }
   }
}
```

Vi har nu fået adgang til vægten via telnet, og vi kan nu oprette metoderne, der giver vægten de ønskede kommandoer fra brugeren. Et eksempel kunne være `getWeight()`-metoden, som simpelt nok modtager de kilogram vægten lastes med, og giver dette som output i konsollen. Metoden ser ud som følger:

```
public double getWeight() throws WeightException {

    try {
       // S command retrieves weight
       System.out.println("Running function getWeight()");
       write.println("S");
       System.out.println("getWeight successfully run");

       System.out.println("awaits response");
       String response = read.readLine();
       System.out.println(response + " successfully stored into
           response string");

       // extracts only the numbers from response to a string
       System.out.println("Cuts the response into useful string");
       String weightString = response.substring(9,
           (response.length() - 2));
       System.out.println("String successfully cut into: " +
           weightString);

       // convert from string to double.
       System.out.println("Converts " + weightString + " into
           double");
       double weight = Double.parseDouble(weightString);
       System.out.println("Convertion successful" + weightString
           + " is now double " + weight + "\n \n");

       return weight;

    } catch(IOException e) {

       throw new WeightException("Error showing weight");
    }
 }
```

I denne metode modtager brugeren, som sagt de kilogram vægten lastes med. Når vægten kommer med outputtet, foregår det i form af en streng. Metoden skulle gerne returnere en double og derfor er det nødvendigt at udtrække tallene der forekommer i streng-resultatet og konvertere dem til en double. Dette gøres med hhv. `substring()` og `parseDouble()` metoden. Substring metoden udtrækker de tal der er i streng-resultatet fra index 9 til index `response.lenght()-2` i den givne streng. Metoden kan bruges på denne måde da streng-resultatet altid vil indholde tal fra index 9 til længden af strengen minus 2. parseDouble konverterer blot disse tal fra streng til double. Slutvis kastes en `WeightException` videre op i laget, hvis der opstår en fejl under visningen af kilogram.

4.2 Overvejelser af klasseopbygning

Vores kode for dette projekt er sat op efter 3-lags modellen, ved at dele klasserne op iblandt 2 pakker, der repræsenterer funktionalitets- og data laget. Vi har ikke lavet en brugergrænseflade pakke, da vægten, som vores program skal interagere med, kommer til at være brugergrænsefladen.

I vores data lag har vi vores `Batch` og `User` klasser. Disse to klasser står for at give en "opskrift"for opbyggelsen af en bruger og en batch, samt relevante metoder for at kunne evt, redigere de forskellige informationer disse objekter består af. Vi har også en `database` klasse i vores datalag. Denne klasse fungerer som en midlertidig simulation af, hvordan vi regner med en SQL database ville fungere. Vi har besluttet os for at kode den til at indeholde en bruger og et batch, med nogle metoder til at kunne hente de forskellige informationer om disse to objekter. Årsagen bag, hvorfor vi har valgt at sætte vores database op på denne måde, istedet for at hardcode det hele, er at vi følte at det ville være en mere akurat repræsentation af, hvordan en rigtig SQL database fungere. Til sidst har vi så vores `DatabaseTransfer` klasse. Denne klasse står for at hente alt information om brugerne og batchesne fra databasen og indsætte dem i en User og Batch arrayliste. Årsagen til at denne klasse skal hente alle informationerne fra databasen er, så alt koden fra vores funktionalitets klasser kan interagere med disse informationer.

Vores funktionalitet lag har ansvaret for at håndtere alt information der kommer fra datalaget og brugergrænselflade laget. Derfor har den en `Weight_Translate` klasse, der står for at oprette en forbindelse mellem vægten og programmet. Det er også i denne klasse at vi får sat metoder op til at kunne interagere med vægten. Vi har også en `WeightException` klasse, som vi bruger til at kunne identificere fejl i vores program/ metoder, hvis der noget der er noget der ikke fungere. Vi har også lavet en funktionalitets interfrace. Årsagen bag dette er at vi skal have vores program til at kunne

køre både på den fysiske og den virtuelle vægt. Ud fra vores interface har vi lavet functionality physical og functionality virtual, begge disse klasser står for at få printet de forskellige instrukser til brugeren på vægten. Det er også her at alt brugerens input bliver modtaget og behandlet.

5 Dokumentation

Vi bruger javadoc for at dokumentere koden. Det er afgørende for at hjælpe andre med at forstå vores program og endda at minde os selv om, hvordan vores egne metoder fungerer. Vi har skrevet javadoc som kommentarer i selve koden hvor de let kan opdateres med eventuelle ændringer.

6 Konfiguration

Vi valgte at bruge Github som versionskontrolværktøj, da det er designet netop til dette formål, samt gør det overordnet gruppearbejde meget nemmere, ved at alle har den nyeste version af programmet m.m. Yderligere hvis der skulle opstå fejl i programmet eller andre ting der gør så programmet ikke længere fungere er det super nemt at gå tilbage til en ældre version som vi ved fungere.
Derudover har vi brugt Maven, så alle kørte på den samme version af JDK, samt asana som en platform til at styre projektplanlægning.

7 Test

For test af vores program, har vi benyttet os af brugertest, hvor vi har kørt vores program på både den fysiske og virtuelle vægt. Under vores brugertest fandt vi ud, af at der var nogle småfejl med kommunkationen mellem den fysiske vægt og programmet. Dette problem skyldes at vi ikke havde implementeret et par metoder til, at kunne kommunikere optimalt med den fysiske vægt, da de fungerede helt fint med den virtuelle vægt. Vi løste dette problem ved at lave to forskellige funktionalitetsklasser, som er bygget op på et funktionalitetsinterface. En der står for kommunikationen mellem programmet og den virtuelle vægt og en der står for kommunikationen mellem programmet og den fysiske vægt. Derefter kørte vi endnu en brugertest af programmet, hvor vi nu testede kommunikationen med de to typer af vægte med de tilsvarende functionality klasser. Under denne test har vi fundet et problem i programmet med at gemme tara vægten. Efter lidt undersøgelse af vores kode har vi fundet frem til at problemet ligger i vores `Functionality_Physical`, da vi havde kommet til at smide `setVirtualWeight` inde i denne klasse. Vi har nu fjernet denne metodekald fra klassen på baggrund af denne sidste test.

8 Projektplanlægning

I de foregående uger for projektets afleveringsfrist, var det vigitgt for os at have et struktureret skema over hvad der skulle laves i projektet. Vi valgte derfor, at bruge Asana til opdeling af arbejdsopgaver for projektet. På den måde havde vi deadlines for hver del af projektet, som gjorde at vi to dage for afleveringsfrist havde projektet færdigt, og kunne bruge de resterende dage på ekstra finpudsning.

9 Konklusion

Vi kan konkludere at vi har et program som fungerer super godt, til både den fysiske og den virtuelle vægt, dette skyldes blandt andet at hver vægt har sin helt egen klasse til at kommunikere igennem. Derudover har vi haft nogle successfulde brugertests på den fysiske vægt, hvor vi ikke kunne fejl nogle fejl med vores system. Det flow som kunden bad om fungerer uden problemer og vores system er, på nær mangel af funktionel database, så er programmet klar til at blive sendt ud til opsætning hos kunden.