

CDIO Final

Projektopgave forår 2017

Projektnavn: CDIO Final

Gruppe: 23

Afleveringsfrist: Fredag d. 16. juni 2017

Denne rapport er afleveret via Campusnet (der skrives ikke under).

Denne rapport indeholder 46 sider ekskl. forside og bilag.



Christian Niemann, s165220



Mads Pedersen, s165204



Frederik Værnegård, s165234



Viktor Poulsen, s113403

1 Abstract

In this paper you will get an overview of the process that takes place, from the point where a vision of a software system to handle the operations of a pharmaceutical company, is received, from a customer, to the moment where you stand with a fully functional software system.

You will be taken from requirements specification through analysis of the project by creation of a domain model and use cases. Followed by design with UML Sequence Diagrams and UML Class Diagrams. Finalized with implementation and testing of the software, with unit tests and black box testing, showing that the program meets the requirements. This paper is considered part of an iterative development process, going from CDIO part 1 to CDIO final, with this being the final iteration.

2 Timeregnskab

Deltager	Timer
Christian Niemann	74
Frederik Værnegaard	63.5
Mads Pedersen	41
Viktor Poulsen	62
I alt for alle deltagere	240.5

Tabel 1: Timeregnskab pr. deltager

Opgave	Timer pr. opgave
Design	14.5
Impl.	131
Test	32
Dok.	55.5
Andet	7.5
Ialt	240.5

Tabel 2: Timeregnskab pr. opgave

Se Bilag 11.1 for en detaljeret oversigt

3 Brugervejledning

For at initiere og opstarte vores program skal projektet *23_CDIOFinal.zip* først og fremmest importeres i Eclipse. Projektet består af to dele, et Maven Projekt, *23cdio_final*, og et Java Projekt, *23cdio_final_weightsimulator*. Yderligere information om konfiguration af programkomplekset er beskrevet i Afsnit 9.1.

Webinterface

Vores hjemmeside er deployet på en webserver og kan derfor tilgås direkte fra browseren ved at indtaste følgende URL: http://46.101.131.115:8080/23cdio_final-0.0.1-SNAPSHOT/

	opr_id	opr_name	ini	cpr	password	admin	role
1	1	Angelo A	AA	0707707003	lKjE4fa	0	Foreman
2	2	Antonella B	AB	0808802236	aToJ2lv	0	Pharmacist
3	3	Luigi C	LC	0909909007	jEfm5aQ	0	Operator
4	4	Super User	SU	0109162407	root	1	Pharmacist
5	5	Admin	ADM	0109162407	root	1	None
6	6	Pharmacist	PHA	0109162407	root	0	Pharmacist
7	7	Foreman	FM	0109162407	root	0	Foreman
8	8	Operator	OPR	0109162407	root	0	Operator
9	9	None	NONE	0109162407	root	0	None

Figur 1: Brugere i databasen

De forskellige brugere kan ses på Figur 1, hvortil der kan logges ind med brugeren »Super Admin«, hvis alle tilgængelige rettigheder i systemet ønskes. For at tilgå »Super Admin« logges der ind med **Operator ID: 4** og **Password: root**

Ønskes det alligevel at hoste tilsvarende lokalt kan det gøres ved at udføre følgende trin.

Al kode der forholder sig til webinterfacet er indeholdt i mappen WebContent i Maven Projektet, *23cdio_final*. For at hoste projektet selv gøres følgende:

- Højreklik på selve projekt mappen *23cdio_final*
- Hold over og udvid »Run As« menuen
- Vælg den øverste mulighed »1 Run on Server«
- Er serveren allerede defineret i Eclipse vælges blot »Tomcat v8.5 Server at localhost« og trykkes »Finish«. Webinterfacet burde nu poppe op af sig selv når serveren er oppe at køre. Resterende trin kan da springes over
- Er serveren ikke allerede defineret i Eclipse vælges den Server Type, der ønskes at køre på. I vores tilfælde »Tomcat v8.5 Server at localhost«
- Efterfølgende trykkes der »Next«, hvorefter Tomcat installationsmappen defineres
- Til sidst trykkes der »Finish«

Reset Data: Der er implementeret en gemt menu, som kan tilgås ved at klikke på DTU-logoet nede i bunden af siden. Her har man mulighed for at nulstille systemets data, hvis man ønsker at lege med data og evt. starte forfra undervejs.

Vægtsimulator

Vægtsimulatoren lokaliseres i Java Projektet *23cdio_final_weightsimulator*. Åbnes source-mappen i *23cdio_final_weightsimulator* og dernæst pakken *controller* vil det være muligt at finde klassen »Main.java«. Denne højreklikkes der på, hvorefter »Run As« udvides og »2 Java Application« vælges. Vægtsimulatoren skulle nu være startet op.

Afvejningsprocess

Når enten vægten eller vægtsimulatoren er sat op og tilsluttet kan afvejningsprocessen sættes i gang. Dette gøres ved at finde Maven Projektet, *23cdio_final*, hvori pakken *main.java.dk.dtu* indeholder klassen »MainWeighProcess.java«. Åbnes indholdet i klassen kan IP og Portnummer, til den vægt man har tilsluttet, ændres.

Som projektet importeres er den sat til at tilslutte vægtsimulatoren (localhost:8000). Dette kan selvfølgelig ændres til en af de fysiske vægte (169.254.2.3:8000 eller 169.254.2.2:8000). Når dette er valgt og indstillet korrekt vil man kunne højreklikke på »MainWeighProcess.java«, finde »Run As« og vælge »2 Java Application«. Afvejningsprocessen skulle nu være sat i gang og vise sig på vægtens display.

Indhold

1	Abstract	1
2	Timeregnskab	1
3	Brugervejledning	2
4	Indledning	6
5	Problemformulering	7
6	Analyse	8
6.1	Kravsspecifikation	8
6.2	Domænemodel	11
6.3	Use-case Diagram	12
6.4	Use Case Scenarier	13
6.4.1	Use-Case 01: Bruger-administration	13
6.5	Requirement Tracing	15
7	Design	16
7.1	Design Klassediagram	16
7.1.1	Afvejningsprocessen	16
7.1.2	Webinterface	17
7.2	Design Sekvensdiagram	19
7.2.1	Afvejningsprocessen	19
7.2.2	Webinterface	23
8	Implementering	26
8.1	Kildekode	26
8.1.1	Afvejningsprocessen	26
8.1.2	Webinterface	28

8.1.3	API	30
8.1.4	Data Access Layer	32
8.2	Argumentation for test	33
8.3	Test	34
8.3.1	Unit Test	34
8.3.2	Vægtsimulator	35
8.3.3	Afvejningsprocessen	36
8.3.4	Postman	39
8.3.5	Webinterface	40
8.3.6	W3 Validering	43
8.4	Test Tracing	43
9	Operation	44
9.1	Konfiguration	44
9.1.1	Afvejningsprocessen	44
9.1.2	Webinterface	44
10	Konklusion	46
11	Bilag	1
11.1	Tidsregistrering	1
11.2	Kildekode	2
11.3	Database Dokumentation	3
11.3.1	ER-Diagram	3
11.3.2	Relationsskema	4
11.3.3	Database Test	4
11.4	Vægt Simulator Dokumentation	14
11.4.1	Design Klassediagram	14
11.4.2	Kildekode	15

4 Indledning

Denne opgave er det sidste led i en række af opgaver, som vi har lavet gennem dette semester. Vi har fra starten lagt vægt på, at hver af CDIO delopgaverne, skal være robuste, så de er nemme at sammensætte til et større projekt. Vi har derfor genbrugt mange elementer fra de tidligere CDIO opgaver, lavet modifikationer, tilføjet funktionalitet og implementeret et webinterface, som samlet udgør det endelige produkt.

I denne rapport bliver det gennemgået hvordan en software udviklingsproces forløber, fra en vision modtages fra kunden, til afslutningen af det endelige softwaresystem.

I Analyse afsnittet, afsnit 6, kan man se den kravsspecifikation, der er blevet lavet ud fra kundens vision, efterfulgt af use case modellering, afsluttet med sporing, for at se om krav og use cases matcher hinanden.

I Design afsnittet, afsnit 7, kan man se de sekvensdiagrammer der er blevet fremstillet for at beskrive systemets flow, samt reverse engineered design klassediagrammer, der viser hvad de forskellige klasser indeholder og deres sammenhæng.

I Implementerings afsnittet, afsnit 8, beskriver vi ved hjælp af kodestumper, hvordan de forskellige dele af programmet er kodet, efterfulgt af argumentation for hvordan vi tester vores system, og beskrivelser af disse tests.

I Operations afsnittet, afsnit 9, bliver det gennemgået hvordan man sætter systemet op på en windows maskine, når det er gjort kan man i Afsnit 3 læse hvordan man bruger systemet.

Forud for denne opgave, har vi i CDIO1 implementeret et bruger-administrations modul, ved anvendelse af interfaces med implementering af 3-lags modellen. Det design, er det vi har bygget videre på for at lave bruger-administrationsdelen i det endelig system, samt implementeret administration af recepter, råvarer og produkter. CDIO2 omhandlede afvejningsprocessen, samt at konstruere en vægt simulator, som kunne være stand-in for den rigtige vægt. I denne opgave har vi udbygget den primitive afvejningsproces fra CDIO2 og finpudset vægt simulatoren, så den fungerer præcis som den rigtige vægt, dokumentation for vægt simulatoren fra CDIO2 kan findes i Bilag 11.4. I CDIO3, blev der lavet et web interface med et bruger-administrations-modul. Dette har vi taget i brug i denne sidste opgave, da kunden ønsker et webinterface, hvor man kan administrere brugere, produkter, råvarer og recepter.

Udover dette har vi også taget det statiske non-persistente data access lag der blev brugt i CDIO1 og CDIO3, og skiftet det ud med den database, vi har arbejdet på i kurset ”02327 Indledende databaser og database programmering”. Relevant dokumentation for databasen, taget fra rapporten der blev lavet til database kurset, kan findes i Bilag 11.3.

Der blev stillet en række krav, som krævede at vi bearbejdede de ældre komponenter en del. Men vi har haft stor nytte af, at have fokus på det endelig projekt fra starten, da det har gjort det let at samle det hele til dette sidste projekt.

5 Problemformulering

I sammenhæng med 13-ugers-perioden og forlængelse med de tre delprojekter, CDIO del 1, 2 og 3 for faget 02324 Videregående Programmering, er vi nu blevet bedt om at udvikle et endeligt softwaresystem til en medicinalvirksomhed. Softwaresystemet skal bruges til håndtering af receptafvejninger, dokumentation af afvejning og råvarebatch forbrug. For systemets brugere skal det være muligt at optræde i forskellige roller, henholdsvis laborant, værkfører, farmaceut og administrator.

I systemet skal man kunne oprette råvarer, der senere hen bestilles hjem fra en bestemt leverandør i en vis mængde af den pågældende råvare, kaldet råvarebatches. Råvarerne optræder i forskellige recepter - eller omvendt - en recept i systemet indeholder en bestemt mængde af nogle råvare. I systemet skal man derfor også kunne oprette de forskellige recepter, samtidig med at systemet vedligeholder et overblik - når råvarebatches bliver anvendt - over den aktuelle mængde for hver råvarebatch. Når en recept produceres af virksomheden, oprettes denne som en produktbatch ud fra den specifikke recept. De afvejede produktbatches indeholder yderligere information om dennes anvendte råvarebatches, samt den bruger i systemet, der har afvejet disse.

For systemet skal der optræde fire forskellige delkomponenter. En database, der vedligeholder al nødvendig information for virksomheden, hvori ny information kan lagres. Et webinterface, som brugervenligt viser det modul, der anvendes til administrering af de forskellige råvarer, recepter, batches og brugere i systemet, der skærpes alt afhængig af de rettigheder, som den bruger, der er logget ind har, ift. den rolle brugeren er givet. En række vejeterminaler med vejeplade, tastatur og display, der alle er udstyret med et Ethernet interface. En afvejnings-styringsenhed, der opretholder afvejningsprocessen for enhver vægt.

Derudover skal udviklingen af systemet dokumenteres, i form af en rapport med nødvendige rapportafsnit, herunder analyse, design, implementering og test - indeholdende relevante diagrammer for systemet.

6 Analyse

6.1 Kravsspecifikation

Punkter fra FURPS+ uden krav er udeladt.

Functional

Krav - Systemet

R.1 Systemet skal indeholde følgende funktioner:

- R.1.1 råvare - skal være muligt at oprette råvarer i systemet. (defineres ved: et brugervalgt og entydigt råvareNr, navn og leverandør)
- R.1.2 råvarebatch - given mængde af råvarer. (defineres ved: et brugervalgt og entydigt råvarebatchNr, råvareNr og mængde) Systemet skal ydermere holde styr på den aktuelle mængde af råvarebatches.
- R.1.3 recept - liste af råvarer, hver med bestemt kvantum. (defineres ved: et brugervalgt og entydigt receptNr, navn og sekvens af receptkomponenter)
- R.1.4 receptKomponent - består af en råvare, en mængde og en tolerance.
- R.1.5 produktbatches - indeholder information om hvilken recepter en produktbatch er oprettet ud fra, samt hvilke råvarebatches de anvendte råvarer er udtaget fra. (defineres ved: et brugervalgt og entydigt produktbatchNr, Nr på recept. Status for batch håndteres af database trigger)
- R.1.6 produktbatchkomponent - Afvejningsresultatet for receptkomponenter gemmes som produktbatchkomponent i produktbatchen i takt med produktet fremstilles.
- R.1.7 operatør - en liste af brugere, som anvender systemet. (defineres ved: et brugervalgt og entydigt oprNr, navn, initialer, password, rolle)

R.2 En bruger skal kunne tildeles en rolle:

- R.2.1 Administrator - Administrerer brugerne i systemet. Repræsenteret ved en true/false værdi
- R.2.2 Farmaceut - Varetager administrationen af råvarer og recepter i systemet og har tillige værktøjerens rettigheder.
- R.2.3 Værktøjereren - Varetager administrationen af råvarebatches og produktbatches og har tillige laborantens rettigheder.
- R.2.4 Laborant - Foretager selve afvejningen
- R.2.5 None - Brugerrolle til at deaktivere adgang

R.3 Administrator skal kunne følgende:

- R.3.1 Oprette, rette, fjerne og vise brugerne i systemet
- R.3.2 Vælge nyt kodeord til brugere.
- R.3.3 Deaktivere brugere i systemet.
- R.3.4 Opdatere egen brugerprofil

R.4 Farmaceut skal kunne følgende:

- R.4.1 Oprette, rette og vise råvarer i systemet.
- R.4.2 Oprette og vise recepter i systemet.
- R.4.3 Få vist enkelte productbatches med tilhørende komponenter.
- R.4.4 Have samme rettigheder som værkfører
- R.5 Værkfører skal kunne følgende:
 - R.5.1 Oprette og vise råvarebatches i systemet.
 - R.5.2 Oprette og vise produktbatches i systemet.
 - R.5.3 Have samme rettigheder som laborant
- R.6 Laborant skal kunne følgende:
 - R.6.1 Afveje på vejeterminalen.
 - R.6.2 Opdatere egen brugerprofil
- R.7 Afvejningssystemet skal kunne følgende:
 - R.7.1 Registrere laborant ID og hente navn på laborant.
 - R.7.2 Validere pågældende laborants password.
 - R.7.3 Bede om den produkt batch, der skal afvejes.
 - R.7.4 Bede om den råvare batch, der er anvendt for hver råvare.
 - R.7.5 På displayet vise hvad laboranten skal gøre og indtaste.
 - R.7.6 Afvejningssystemet skal kunne foretage afvejning med bruttokontrol.
 - R.7.7 Når afvejningsresultat er accepteret dvs. ligger inden for tolerance og har klaret brutto-kontrol, gemmes den i produktbatchkomponenten.
- R.8 Under afvejningsprocessen skal status ændres på produktbatchen efterhånden som processen skrider frem.
- R.9 Der skal eksistere en bruger i systemet med navn "admin" og password "root"
- R.10 Ved opstart af systemet skal admin og root være forvalgt så man let, kan logge på som admin.
- R.11 Bruger skal oprettes med følgende information:
 - R.11.1 Et unikt user-Id mellem 11-99
 - R.11.2 Et unikt brugernavn mellem 2-20 tegn
 - R.11.3 Et unikt initial mellem 2-4 tegn
 - R.11.4 Et gyldigt cpr nr.
 - R.11.5 Et ukrypteret password der overholder DTU's retningslinjer <https://password.dtu.dk/>
 - R.11.6 En rolle
- R.12 Systemet skal indeholde en række delkomponenter:
 - R.12.1 En database med diverse oplysninger. (se R.13)
 - R.12.2 Et webinterface (se Krav - Webapplikation, R.22-R.26) En række vejeterminaler udstyret med vejeplade, tastatur og display, alternativt kan
 - R.12.3 En række vejeterminaler udstyret med vejeplade, tastatur og display, alternativt kan det være en vægtsimulator (se Krav - Vægtsimulator, R.14-R.21)

R.12.4 ASE, som er et program der styrer vejeterminaler og gemmer data fra disse i databasen.

R.12.5 Datalag i applikation, implementeret som MySql database.

R.13 Databasen skal indeholde:

R.13.1 Oplysninger om: brugere, råvarer, råvarebatches, recepter, samt planlagte og færdigproducerede produktbatches.

R.13.2 Data-acces lag, som adskiller databasen fra de andre komponenter.

R.13.3 Fejl i data-acces laget, skal kaste exception der beskriver fejlen, til de øvre lag.

Krav - Vægt Simulator

R.14 Simulatoren skal simulere vægtens opførsel.

R.15 Simulatoren skal kunne modtage og sende kommandoer over TCP.

R.16 Simulatoren skal vise input fra tastatur på display.

R.17 Input skal kunne foregå samtidig.

R.18 Det skal være muligt at sendte følgende 9 kommandoer:

R.18.1 S: Send stabil afvejning.

R.18.2 T: Tarér vægten.

R.18.3 D: Skriv i vægtens display.

R.18.4 DW: Slet vægtens display.

R.18.5 P111: Skriv max. 30 tegn i sekundært display.

R.18.6 RM20 8: Skriv i display, afvent indtastning.

R.18.7 K - Skifter vægtens knap-tilstand.

R.18.8 B - Sæt ny bruttovægt.

R.18.9 Q - Afslut simuleringen.

R.19 Følgende kommandoer skal kunne afvikles fra den simulerede brugergrænseflade på vægten:

R.19.1 Tarér vægten.

R.19.2 Sæt ny bruttovægt.

R.19.3 Afslut simuleringen.

R.20 Simulatoren skal lytte på port 8000.

R.21 Simulatoren skal kunne virke som 'stand in' for den fysiske vægt.

Krav - Webapplikation

R.22 Webapplikationen skal implementere bruger-, råvare-, recept-, råvarebatch og produktbatch-administraton.

R.23 Der skal foretages autentificering af brugerne i en login-session.

R.24 Baseret på brugerens rolle, skal der vises sider der er tilpasset brugerens rettigheder.

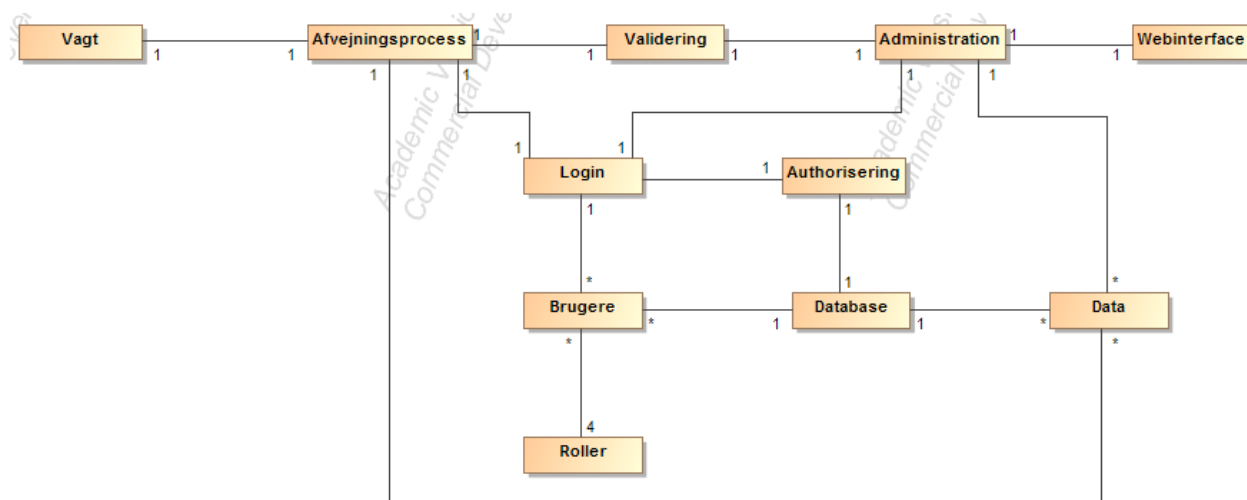
- R.25 Alle form for input felter skal valideres iht. gyldige områder og der skal vises passende fejlmeddelelser ved fejl.
- R.26 Der skal være implementeret tokenbaseret sikkerhed med JSON Web Tokens på webapplikationen.

Implementering

- R.27 Programmet skal kunne køres på Windows maskinerne i databarerne på DTU.

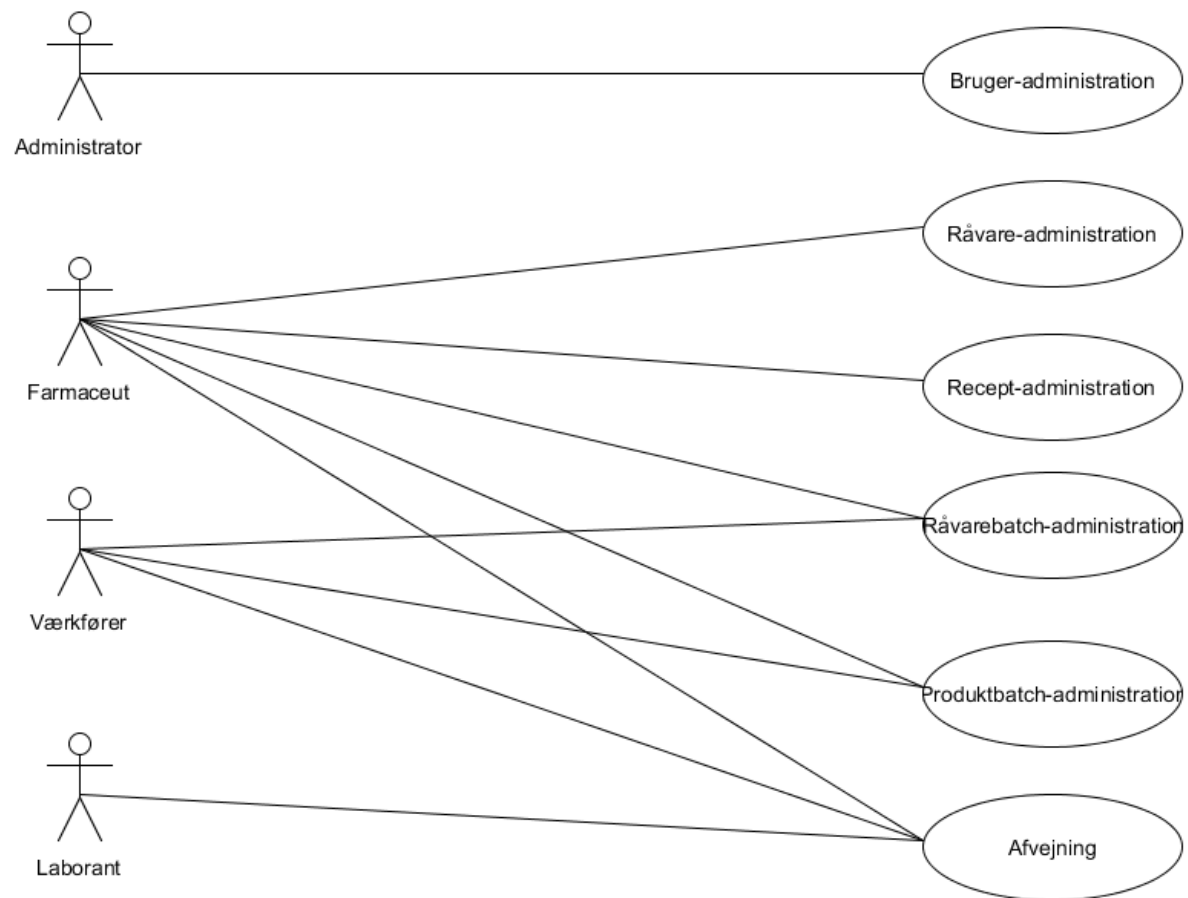
6.2 Domænemodel

Der er udviklet en domænemodel for at illustrere gruppens forståelse af det foreliggende problem. Som det fremgår af figur 2, har vi først og fremmest en afvejningsproces. Denne proces vil have en vægt, samt data som er det der bliver afvejet. Samtidig bliver afvejningsprocessen valideret, og det er krævet at man logger ind for at kunne benytte vægten. I systemet skal det være muligt, at administrere vha. af webinterfacet, den data der er i systemet. Her bliver også valideret og man skal være logget ind. Login kræver man er autoriseret for at tilgå databasen, som indeholder det data og brugerne, som har en række roller. Relationerne mellem de forskellige komponenter i systemet, er vist på diagrammet. Nedenstående domænemodel viser således et overblik over problemet, som forsøges at blive løst og hjælper os til at definere systemets klasser i det senere design.



Figur 2: Domænemodel

6.3 Use-case Diagram



Figur 3: Use case diagram

Som det ses af ovenstående use-case diagram, består det af 6 forskellige use-cases og 4 aktører. De 4 aktører i vores system, er de roller en person kan have i systemet, her har vi:

- Administrator
- Farmaceut
- Værkfører
- Laborant

De i alt 6 use-cases, er hovedkomponenterne i vores system, altså det systemet består af. Vi har først administratoren, som har ansvar for bruger administrationen i systemet. Laboranten kan kun foretage afvejninger. Værkfører har ansvar for råvarebatch- og produktbatch-administration, samt har laborantens rettigheder. Farmaceuten har ansvar for råvare- og produktbatch-administration, samt har værkførerens rettigheder. De i alt 6 use-cases lyder således:

- Bruger-administration
- Råvare-administration
- Recept-administration
- Råvarebatch-administration
- Produktbatch-administration
- Afvejning

Ud fra dette use-case diagram, kan vi altså let se hvilke rettigheder, de forskellige brugere i systemet har, samt se hvilke komponenter vores system skal have.

6.4 Use Case Scenarier

6.4.1 Use-Case 01: Bruger-administration

Use Case Name: Bruger-administration

ID: 1

Brief description: Administratoren ønsker at erstatte en eksisterende farmaceut med en nyansat farmaceut. Data vil ligge på en database, som bliver brugt til administrationen af dette.

Primary Actor Administrator.

Secondary actors Ingen.

Preconditions

- Webapplikationen kører.

Main Flow

- Administratoren åbner webapplikationen.
- Administratoren logger ind.
- Får nu vist en liste af alle medarbejdere, herefter finder administratoren den farmaceut, der ønskes slettet og trykker på "slet".
- Trykker nu "tilføj bruger" for at tilføje en ny medarbejder.
- Udfylder informationen, der er nødvendig for indsættelse, her kan rolle vælges.
- Den tidligere medarbejder er nu slettet, og den nye er indsat. Webapplikationen lukkes.

Postconditions

- Tidligere medarbejder slettet i systemet.
- Ny medarbejder tilføjet i systemet.

Alternative flow

- Administratoren vil også have mulighed for at rette information om eksisterende bruger, hertil sin egen profil.
 - Administratoren kan vælge et nyt password til brugere, hvis det nuværende er glemt.
 - Administratoren kan også tilgå de andre faner, såfremt han har rettighederne til det.
-
- **ID: 2 Råvare-administration:** Administrationen af råvarer i systemet foretages af farmaceut aktøren. Denne skal kunne oprette, rette samt vise råvarer i systemet. En råvare defineres ved et råvareNr (entydigt), navn samt leverandør. Data vil ligge på en database, som bliver brugt til administrationen af dette.

- **ID: 3 Recept-administration:** Administrationen af recepter foretages af farmaceut aktøren. Denne skal kunne oprette samt vise recepter i systemet. En recept defineres ved et receptNr (entydigt), navn samt en sekvens af receptkomponenter. En receptkomponent består af en råvare (type), en mængde samt en tolerance. Data vil ligge på en database , som bliver brugt til administrationen af dette.
- **ID: 4 Råvarebatch-administration:** Råvarebatch-administration. Administrationen af råvarebatches i systemet foretages af værkføreren. Denne skal kunne oprette samt vise råvarebatches i systemet. En råvarebatch defineres ved et råvarebatchNr (entydigt) samt mængde. Data vil ligge på en database , som bliver brugt til administrationen af dette.
- **ID: 5 Produktbatch-administration:** Administrationen af produktbatches i systemet foretages af værkføreren. Denne skal kunne oprette samt vise produktbatches i systemet. En produktbatch defineres ved et produktbatchNr (entydigt), nummeret på den recept produktbatchen skal produceres udfra, dato for oprettelse, samt oplysning om status for batchen. Status kan være: oprettet / under produktion / afsluttet. Afvejningsresultater for de enkelte receptkomponenter gemmes som en produktbatchkomponent i produktbatchen i takt med at produktet fremstilles. Når værkføreren har oprettet et nyt produktbatch udprintes denne og uddeles til en udvalgt laborant. laboranten har herefter ansvaret for produktionen af batchet. Data vil ligge på en database , som bliver brugt til administrationen af dette.
- **ID: 6 Afvejning:** Afvejning vil hovedsageligt blive udført af en laborant, som arbejder ved vejeterminalen. Laboranten har til ansvar, at afveje alle råvare i en recept. Laboranten identificerer sig overfor vægten, så man kan se hvem, der har afvejet en given batch. Når han har identificeret sig, vejer han først beholderen og derefter en given råvare, som er vist på displayet. Når han har vejet råvaren, skal laboranten angive hvilke batchNr den aktuelle råvarebatch har. Det er vigtigt, at afvejningen ligger inde for den givet tolerance og der er udført brutto-kontrol. Når afvejning af en råvare er færdig, vil status blive ændret fra "oprettet" til "under produktion". Det er muligt at bruge en simulator i stedet og få samme output.

6.5 Requirement Tracing

	Use cases	1	2	3	4	5	6
Requirements							
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							
20							
21							
22							
23							
24							
25							
26							
27							

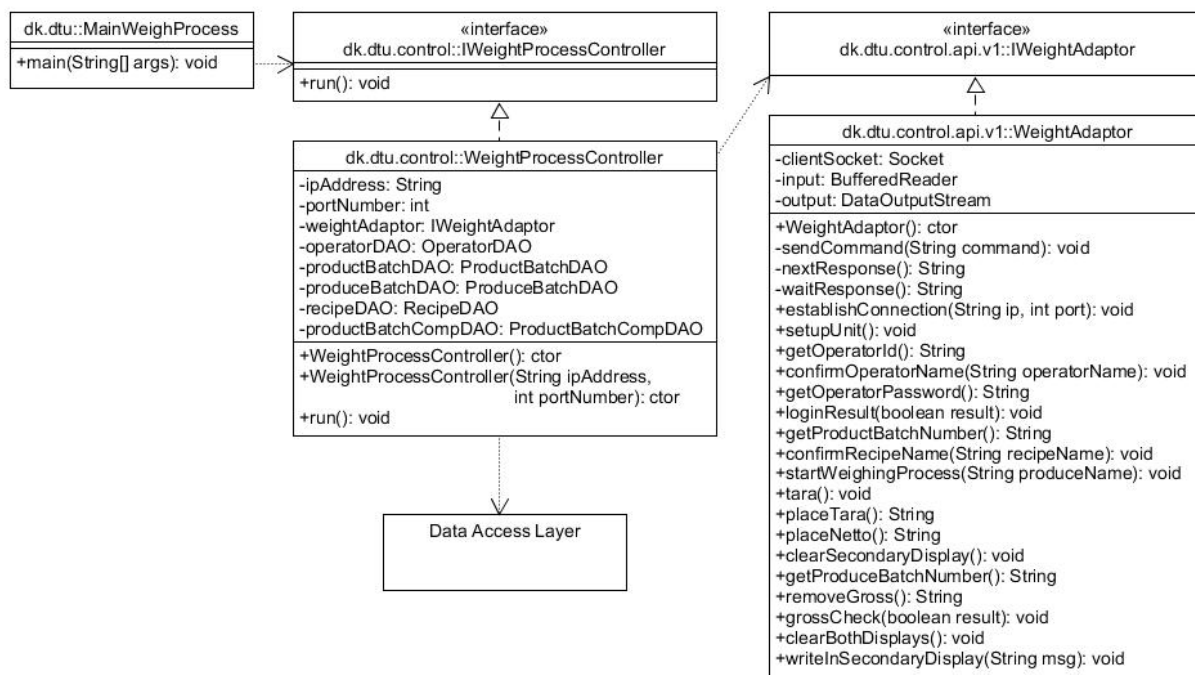
Figur 4: Requirement Tracing

Vores Requirements Traceability tabel viser, hvorledes de funktionelle krav og use-cases dækker hinanden. Som vi kan se i vores tabel, er ethvert funktionelt krav blevet dækket af mindst en use-case - og hver af vores use-cases dækker mindst et krav. Det er ikke muligt at dække krav R.27, da det ikke kan testes af en use-case.

7 Design

7.1 Design Klassediagram

7.1.1 Afvejningsprocessen



Figur 5: Klassediagram af afvejningsprocessen

Figur 5 viser et klassediagram for systemet til afvejningsprocessen. Dette er opdelt i tre lag, henholdsvis en adaptor, en controller og et Data Access Layer. På Figuren er metoderne fra interfacet `IWeightAdaptor` udeladt, for overskuelighedens skyld, da den implementerende klasses public-metoder alle ligger i interfacet.

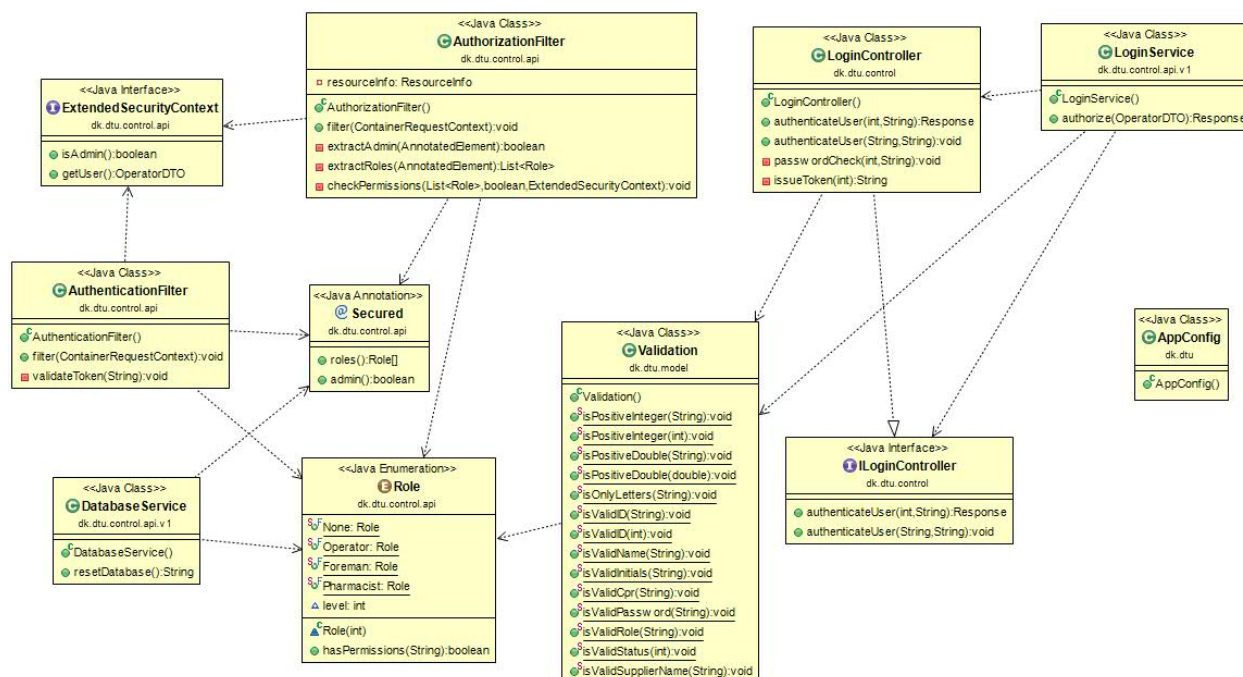
Klassen `MainWeighProcess` indeholder blot `main`-metoden for systemet, der igangsætter `run()` for den instans, som implementerer `IWeightProcessController` interfacet. Her kan IP og Port for den vægt man vil tilgå ændres.

Klassen `WeightProcessController` er den implementerende klasse for `IWeightProcessController` og gør både brug af klassen `WeightAdaptor` og Data Access Laget. Denne klasse styrer det overordnede flow for afvejningsprocessen, der indeholdes i den ene metode `run()`. Klassen har adgang til databasen gennem de forskellige DAO-klasser (Data Access Layer) og bruger `WeightAdaptor`-klassen til at kommunikere igennem til vægten.

`WeightAdaptor` er den implementerende klasse for `IWeightAdaptor`. Denne ses for at agere som API, mellem vægten og controlleren. Her pakkes de forskellige sekvenser for afvejningen ind i metoder, der gør det lettere at overskue i controlleren samt forøger robustheden af vores program, hvis vægten en dag skulle blive udskiftet med en anden vægt. Da kan vi blot udskifte vores adaptor, i stedet

for store mængder af vores kode.

7.1.2 Webinterface

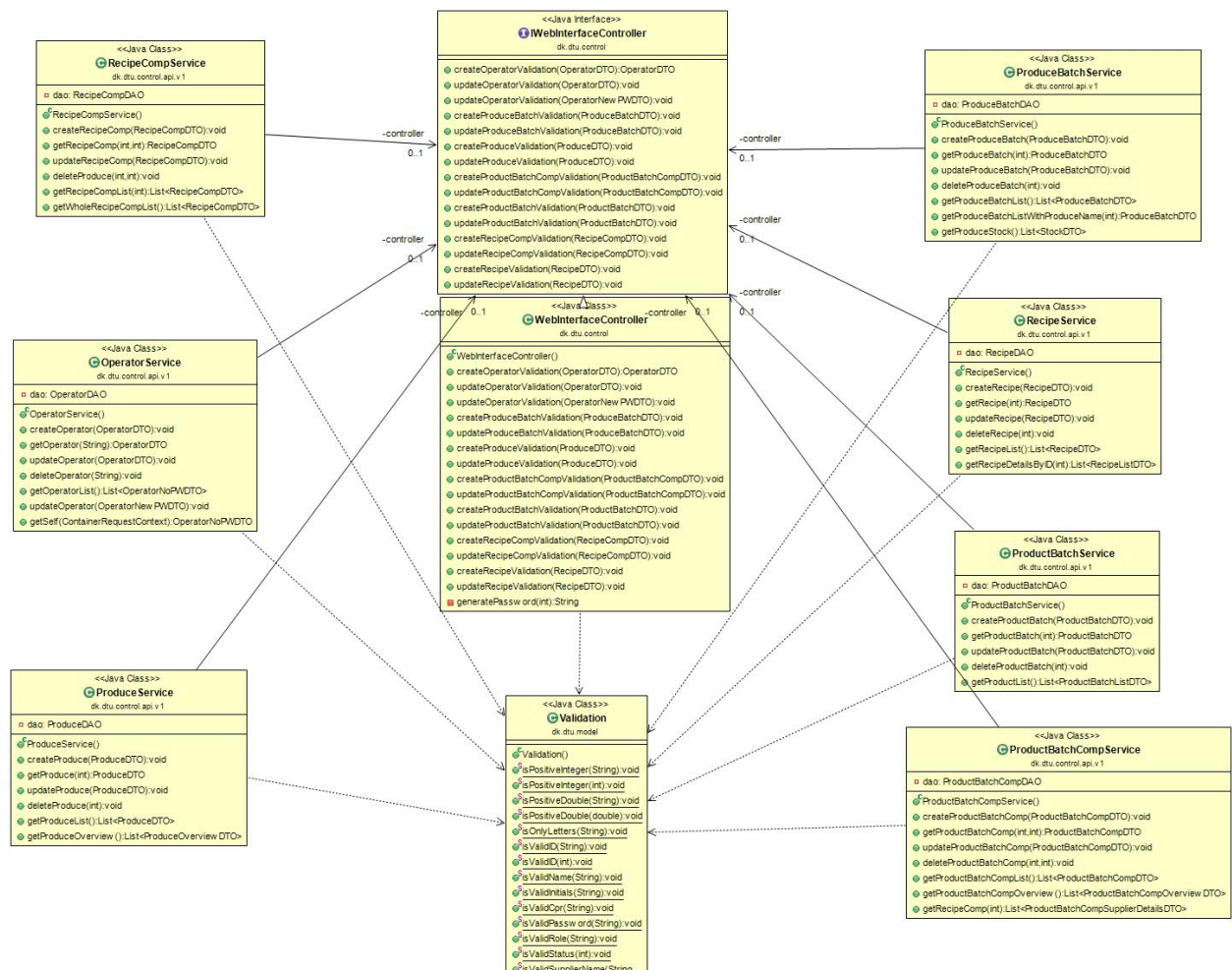


Figur 6: Klassediagram af webinterfacet - del 1

Figur 6 viser et klasse diagram over webinterfacets DAL niveau. Dette er det første af to klassediagrammet til webinterfacet.

På figur 6 har vi de interfaces klasser, enumartors og annotations som sørger for at systemet er i stand til at verificere en brugers adgangsniveau, samt login. Vi skal både kunne levere tokens og verificere dem i denne del af systemet, samt yderligere verificere brugerens adgange med roller og admin-boolean. Som vi kan se på klassediagrammet har vi logincontroller og loginservice. Login-controlleren står for at autorisere brugerne mens LoginService er det API-endpoint vi eksponerer til login. Samtidig har vi authorization, som er den del af systemet, der tjekker rettigheder for brugerne på webinterfacet.

På begge klassediagrammer, har vi Validation klassen, da den er brugt af mange klasser i webinterfacet. I dette klassediagram, bliver det blandt andet, brugt af loginservice, logincontroller og Role. Role er en enumerator der kan tage en af de 4 forudbestemte roller og som yderligere har en sammenlignelig "vægt"(int). Derfor kan vi sammenligne to roller med simpel boolks aritmetik.



Figur 7: Klassediagram af webinterfacet - del 2

Figur 7 viser et klasse diagram over systemets webinterface over DAL niveau. Dette er det andet af to klassediagrammer til webinterfacet.

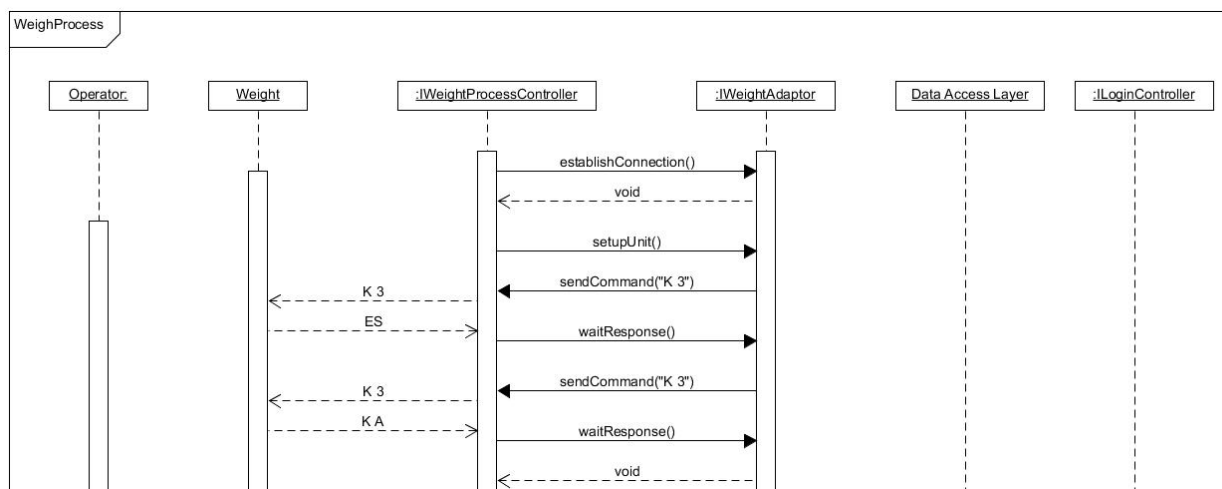
På ovenstående klasse diagram, kan vi se en række services, som udgør REST API'et i webapplikationen forbundet med en controller. I denne del af systemet defineres de API services vi eksponerer, samt en controller til at håndtere data-processeringen.

Vi har anvendt Java Jersey til at definere vores paths og methods med annotations, således at vi kan udføre CRUD-operationer med http-metoderne POST (create), GET (read), PUT (update), DELETE (delete)

Igen har vi validering med på klassediagrammet, da vi naturligvis ønsker at validere, det input der bliver skrevet ind på webapplikationen.

7.2 Design Sekvensdiagram

7.2.1 Afvejningsprocessen

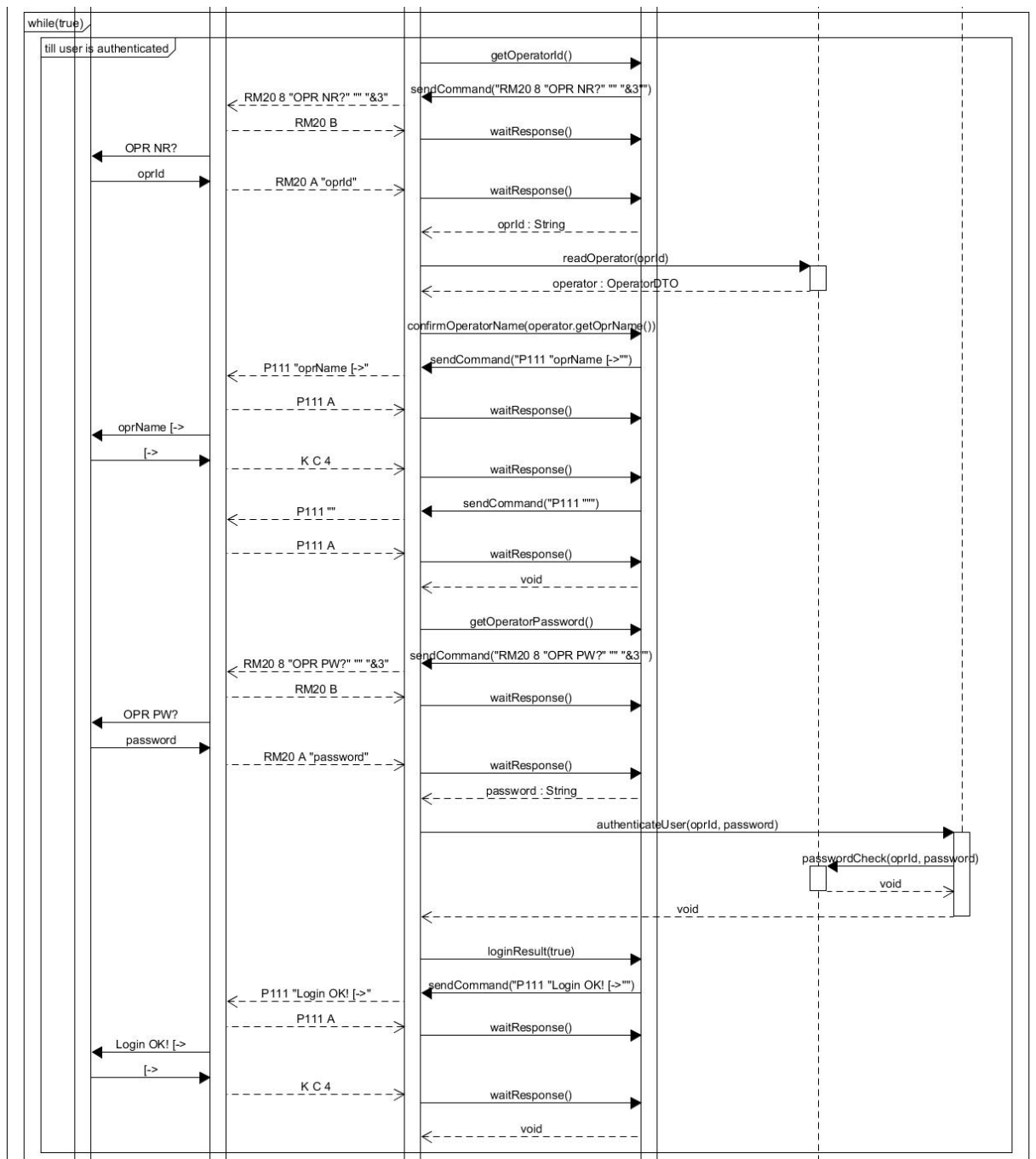


Figur 8: Sekvensdiagram af afvejningsprocessen. Initiering.

På Figur 8, 9, 10, 11 og 12 vises sekvensen for afvejningsprocessen. Figurene kan afvige fra det oprindelige program, idet der kan forekomme ændringer undervejs tilmed at figuren har til formål at vise og give et overblik over det overordnede flow, og af samme årsag er nogle ting udeladt.

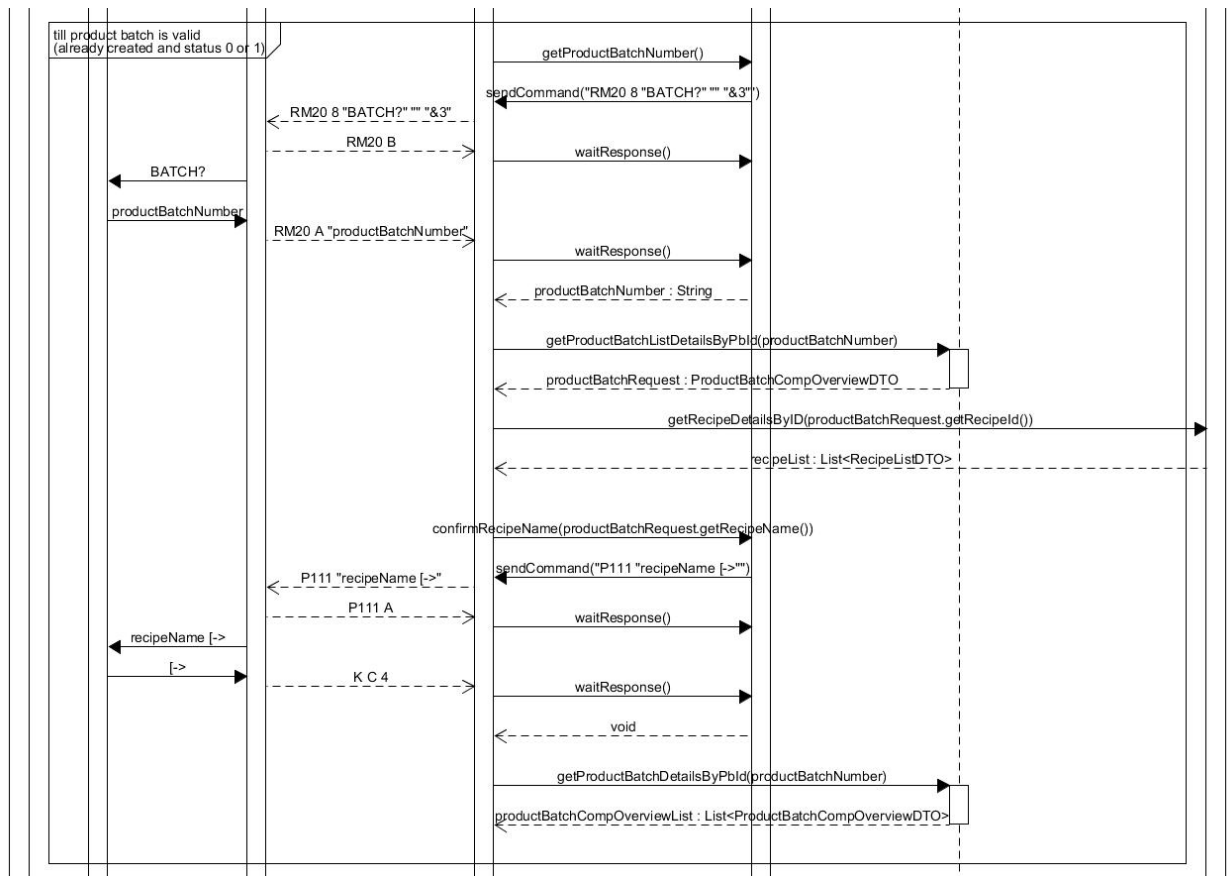
De inkluderede klasser (i sekvensdiagrammet) er følgende: `IWeightProcessController`, `IWeightAdaptor`, `ILoginController` og `Data Access Layer`, hvoraf sidstnævnte er en sammensmeltning af de forskellige DAO-klasser og henviser derfor til disse. `Operator` og `Weight` referere - som de hedder - til den pågældende operatør og den pågældende vægt. Vi forudsætter at vægten er tændt og at vi har adgang til databasen.

Når programmet igangsættes forsøger vi at oprette en forbindelse fra controlleren til vægten via metoden `establishConnection()`, der vises på Figur 8. Dernæst sender vi to ens beskeder, »K 3«, til vægten. Dette skyldes at den allerførste besked, der sendes til vægten bliver ignoreret. Vi ændrer altså vægtens Keystate til Keystate 3.



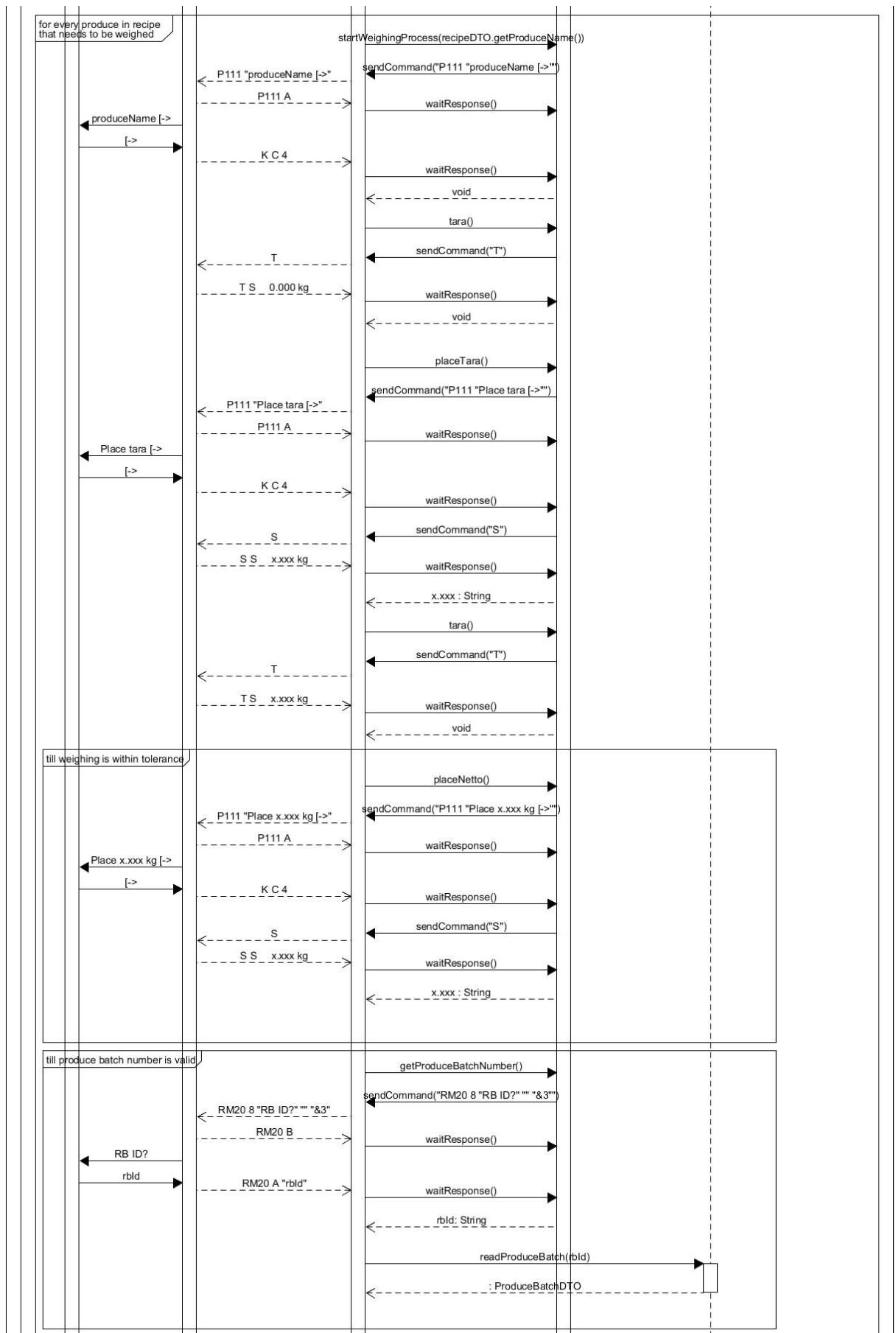
Figur 9: Sekvensdiagram af afvejningsprocessen. Login.

Vægten er nu klar til at modtage et operatørens identifikationsnummer. Denne indtastes på vægten, hvortil der tilgås databasen med det identifikationsnummer, som er blevet oplyst. Findes operatøren ikke i systemet/databasen startes der forfra. Findes han dog i systemet, vil operatørens navn blive hentet op fra databasen og fremvist på vægtens sekundære display. Efterfølgende vil operatøren skulle oplyse sit password, der gennem LoginController'en vil blive tjekket. Er ID og password korrekt vil vi fortsætte videre gennem processen. I modsatte tilfælde vil vi starte forfra igen. Denne sekvens er vist på ovenstående sekvensdiagram, Figur 9.



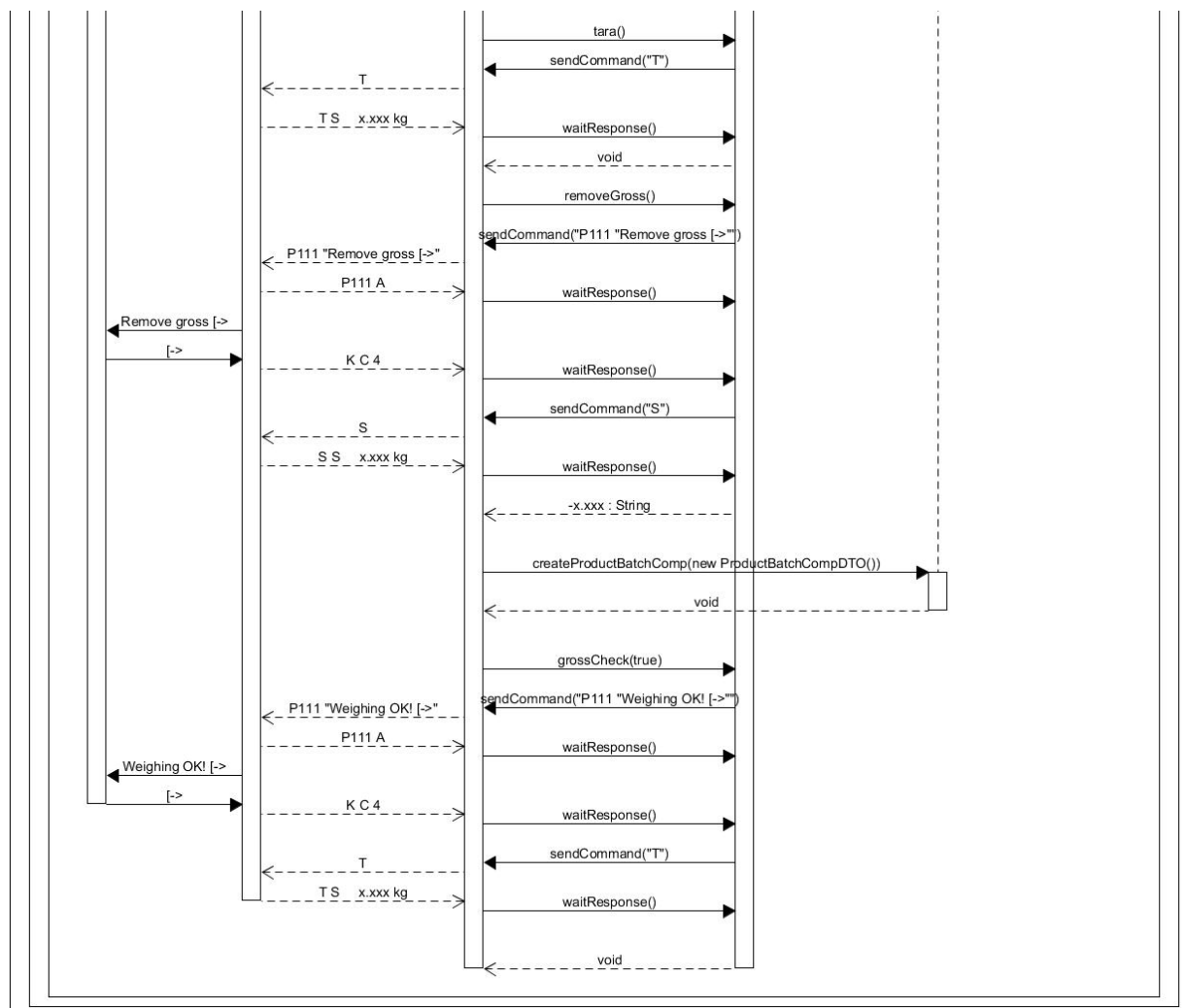
Figur 10: Sekvensdiagram af afvejningsprocessen. Produkt batch nummer.

Der bliver nu bedt om at indtaste et produkt batch identifikationsnummer, for den gældende batch man skal afveje. For at komme videre skal vi derfor indtaste et identifikationsnummer for en eksisterende produkt batch, der har status 0 eller 1, idet vi selvfølgelig kun ønsker at afveje batches der ikke er færdige. Alt andet vil blive afvist og vi vil blive bedt om at indtaste et ID igen. Når produkt batch nummeret er blevet tilkendegivet af operatøren vil der blive hentet en masse information op fra databasen, der skal bruges til selve afvejningen. Og vi er da klar til at afveje de manglende råvarer til produkt batchen. Vist på Figur 10.



Figur 11: Sekvensdiagram af afvejningsprocessen. Afvejning.

Afvejningen starter da ved at operatøren får oplyst den aktuelle råvare, hvorefter han bliver bedt om at placere taraen, hvoraf dennes vægt registreres i systemet. Når dette er gjort bliver der bedt om at placere netto. Netto indregistreres først indtil afvejningen forholder sig indenfor den nominelle vægt i et interval bestemt af dets tolerance. Når afvejningen er lykkedes bedes der om et råvare batch identifikationsnummer, der specificerer hvilken råvare man har brugt. Vist på Figur 11.



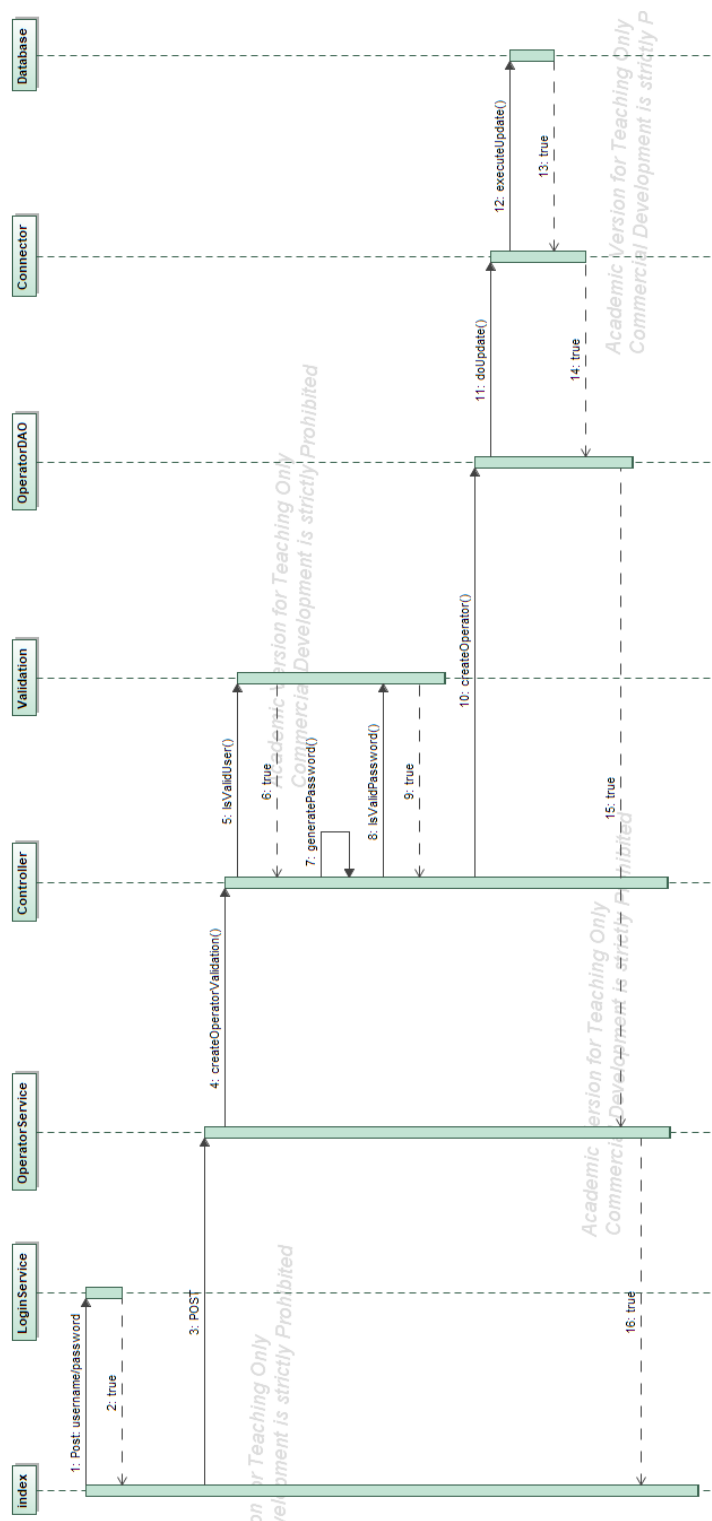
Figur 12: Sekvensdiagram af afvejningsprocessen. Fjern brutto.

Sidst fjernes brutto, der ses på Figur 12, hvortil der laves et bruttotjek for afvejningen. Er bruttotjekket ikke i orden vil afvejningen af samme råvare gentage sig. Er bruttotjekket derimod i orden vil afvejningen blive oprettet i databasen, og operatøren vil få oplyst den næste råvare som skal afvejes til den pågældende produkt batch. Dette gøres indtil alle råvarer er afvejet. Da vil vi igen blive bedt om et operator ID - og vi vil starte forfra.

7.2.2 Webinterface

På figur 13 som er et sekvens diagram over webinterfacet, er tilføjelse af bruger processen vist. Som det fremgår af sekvensdiagrammet, starter processen i webinterfacet hvor man logger ind som admin og ønsker at tilføje en bruger. Processen fortsætter så ned gennem controller og validering,

som sikrer at brugeren, der bliver tilføjet, har valid brugerinformation. Når det er blevet valideret, bliver informationen sendt videre til vores DAO, som sidst sender det til databasen hvor det bliver lagret, og man vil få antallet af ændringer lavet, i retur.



Figur 13: Web interface sequence diagram

Diagrammet er simplificeret en smule for overskuelighedens skyld, men det skal nævnes at con-

trolleren kører flere kald mod Validation-metoderne til at verificere hele brugerobjektet, inden det fortsætter ned i databasen.

8 Implementering

8.1 Kildekode

8.1.1 Afvejningsprocessen

Afvejningsprocessen består hovedsageligt af to klasser, WeightAdaptor og WeightProcessController. WeightAdaptor-klassen er mellemløbet mellem vægten og controlleren (WeightProcessController-klassen), der pakker de forskellige del-sekvenser ind i metoder, så det er lettere at håndtere i controlleren. De fleste metoder i adaptor-klassen består derfor af *sendCommand(command)* og *waitResponse()*, der henholdsvis sender en kommando til vægten eller læser/venter på en respons fra vægten.

```
private void sendCommand(String command) throws IOException {
    output.writeBytes(command + "\r\n");
    output.flush();
    System.out.println("Bruger sendte kommandoen: " + command);
}

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

Et eksempel på en metode, der benytter sig af de to ovenstående metoder, *sendCommand(command)* og *waitResponse()* kunne være *getOperatorId()*, der skal modtage et identifikationsnummer fra vægten. I følgende stykke kode ses det at vi sender en RM20-kommando til vægten, hvortil vi forventer et respons om at vægten har modtaget beskeden, samt en responsbesked fra brugeren når denne har indtastet sit ID, hvilket vi i metoden returnerer. Skulle der i dette tilfælde blive kastet en Exception fanger vi denne og kaster en ny selvskreven AdaptorException videre til controlleren. Denne metode viser den generelle måde hvorpå vi har skrevet resten af metoderne, idet vi blot sender nogle andre beskeder til vægten. Resten af metoderne i WeightAdaptor ligner derfor nedenstående metode.

```
@Override
public String getOperatorId() throws AdaptorException {
    try {
        sendCommand("RM20 8 \"OPR NR?\" \"\" \"&3\"");
        waitResponse();
        return waitResponse().split("\\\")[1];
    } catch (Exception e) {
        throw new AdaptorException(e);
    }
}
```

WeightProcessController-klassen styrer det overordnede flow og logikken for vores afvejningsstyringsenhed. Denne klasse benytter adaptor-klassen, tilgår databasen og håndterer de forskellige Exceptions, der kastes. Et udsnit af klassen kan ses i nedenstående stykke kode, der netop bruger *getOperatorId()* fra WeightAdaptor-klassen.

Denne del af processen er bundet af et while-true-loop, der kun breaker, idet vi gennemfører alle metoderne uden at der bliver kastet nogle exceptions. I tilfælde af at der skulle blive kastet en Exception vil vi sende en besked ud til brugeren og starte forfra (*continue oprIdLoop;*). Vi forsøger i dette udsnit først at hente operatørens identifikationsnummer, validere dette og fra databasen hente brugerens informationer op. Og til sidst vise operatørens navn på vægtens display. Lykkedes dette vil vi bryde loopet og ryge videre i processen.

```
oprIdLoop: while(true) {
    try {
        oprId = weightAdaptor.getOperatorId();
        Validation.isPositiveInteger(oprId);
        operatorDTO =
            operatorDAO.readOperator(Integer.parseInt(oprId));
        weightAdaptor.confirmOperatorName(
            operatorDTO.getOprName());
        break oprIdLoop;
    } catch (AdaptorException e) {
        errorMessageInSecondaryDisplay("Problem trying to send
            or recieve message from weight!");
        e.printStackTrace();
        continue oprIdLoop;
    } catch (NotFoundException e) {
        errorMessageInSecondaryDisplay("Operator was not
            found!");
        e.printStackTrace();
        continue oprIdLoop;
    } catch (PositiveIntegerValidationException e) {
        errorMessageInSecondaryDisplay("The ID was not
            valid!");
        e.printStackTrace();
        continue oprIdLoop;
    } catch (DALException e) {
        errorMessageInSecondaryDisplay("Problem trying to read
            operator from database!");
        e.printStackTrace();
        continue oprIdLoop;
    }
}
```

Ovenstående stykke kode viser, hvordan vi generelt har struktureret koden i WeightProcessController-klassen, der selvfølgelig med undtagelser ligner. Det meste af koden er indkapslet i loops, der kun breaker hvis alt går som det skal, hvortil vi ved fejl eller mangler kaster og styrer efter de forskellige Exceptions der fanges.

8.1.2 Webinterface

Til afvikling af frontend er der anvendt et par javascript biblioteker, som hjælper med bla. at præsentere elementerne overskueligt, lade os genbruge opsætninger ved templates og mere. Nedenfor ses en liste over de biblioteker vi har importeret og hvad de bruges til:

Jquery 3 DOM manipulation og event-binding

Mustache Template rendering

js.cookie Cookie CRUD

Materialize Framework til layout samt visningsskift og mere

Dertil kommer det kode som vi har skrevet i:

index.html Markup samt mustache templates

style.css Ændringer til præsentation af forskellige elementer

script.js Javascript funktioner, event-listeners og initialisering af materialize-funktionalitet (faner der skifter o.lign)

HTML'en er bygget op med en række templates, som er skrevet i dokumentets <head> sektion. De er hver lavet i et <script> tag og loades på deres unikke ID'er. I systemet er der følgende templates:

ProductBatchAdministrationTemplate Den overordnede template der renderer fanen "Product Administration"

productBatchEditTemplate Den template der renderer redigerboksen, som åbnes ved klik på redigér ud for et productbatch

productBatchInsertTemplate Den template der renderer indsættelsessboksen, som åbnes ved klik på tilføj knappen for productbatch

productBatchComponentsTemplate Den template der renderer tabellen af komponenter når en productbatch åbnes

productBatchComponentInsertTemplate Den template der renderer indsættelsesboksen når der trykkes på tilføj knappen for productbatchcomponent

ProduceAdministrationTemplate Template til oversigt over alle råvare i systemet

produceEditTemplate Template til redigering af en råvare

produceInsertTemplate Template til indstættelse af en råvare

ProduceBatchAdministrationTemplate Template til oversigt over alle råvarebatches i systemet

produceBatchEditTemplate Template til redigering af et productbatch

produceBatchInsertTemplate Template til oprettelse af et nyt råvarebatch

ProduceBatchStockAdministrationTemplate Template til visning af råvarelagerbeholdning
(anvender specielt designet view i databasen)

RecipeAdministrationTemplate Template til fanen Recipe Administration

recipeInsertTemplate Template til indsættelse af en ny opskrift

RecipeComponentsTemplate Template til visning af en opskriffs komponenter

recipeComponentEditTemplate Template til redigering af en opskriffs enkelte komponent

recipeComponentInsertTemplate Template til indsættelse af en ny komponent til en opskrift

UserAdministrationTemplate Template til administration af alle systemets brugere

userEditTemplate Template til redigering af en enkelt bruger

userEditProfileTemplate Template til redigering af egen profil

userInsertTemplate Template til oprettelse af en ny bruger

I webinterfacets markup er der netop én popup-boks til indsættelse og redigering af elementer, som bliver født med renderede templates vha. mustache.js. Dvs. vi kan indsætte flere forskellige html `<form>`-elementer afhængig af hvilken information vi ønsker brugeren skal udfylde. Vi kan samtidigt pre-udfylde indholdet ved redigering, ud fra et enkelt kald i API'et efter det objekt vi ønsker at redigere.

For at muliggøre det template-baserede system har vi generaliseret én javascript-metode til asynkront at kalde api'et efter data og renderer templates. Denne funktion ses nedenfor:

```
function doAjax(method, url, data, notice, template, dom_target,
  callback, contextTab){
  var contentType = "application/json";
  if(typeof data !== "string"){
    data = JSON.stringify(data);
  }
  $.ajax({
    type: method,
    context: contextTab,
    contentType: contentType,
    processData: false,
    crossDomain: true,
    url: url,
    data: data,
    headers : {
      Authorization: Cookies.get("auth")
    },
    statusCode:{
      401 : function (response) {
        callback(response);
      },
      403 : function (response) {
```

```

        callback(response);
    }
},
success: function(response) {
    if(template !== null){
        Mustache.parse(template);    // optional, speeds up
        future uses
        var rendered = Mustache.render(template, response);
        dom_target.html(rendered).promise().done(function () {
            callback(response);
        });
    }else{
        callback(response);
    }
},
error: function ( msg ) {
    ajaxErrorHandler(msg, notice, contextTab);
}
})
}

```

Det centrale i metoden er værdierne `template`, `dom_target`, `callback` og `contextTab`. Template er den urenderede mustache template hvor `dom_target` er der vi ønsker at rendere template til. Vi har så en callback funktion, som vi kalder når vores template er færdig med at blive indsat i DOM'en. Dette gør at vi kan få den præcise timing som vi ønsker, når vi fremadrettet skal kalde flere funktioner efter API-kaldet.

Der er samtidigt lavet nogle specielle funktioner omkring statuskoderne 401 og 403, hvilket gør at vi kan fortsætte med at eksekvere callback-funktioner når vi forventer fejl. `ContextTab` er den context, som brugeren tilføjer ved at trykke på et element. I nogle tilfælde ønsker vi at deaktivere det brugeren har trykket på, når/hvis han f.eks. får svarkodern 401 eller 403.

Hver gang en brugeren interagerer med et element indlæses og overskrives det relevante data. Dette gør at det aldrig er nødvendigt at refresh siden for at se det nyeste indhold.

Ved fejl i ajax-kaldene er der skrevet en `ajaxErrorHandler`-funktion. Denne funktion sørger for at deaktivere faner hvis vi får fejlkode 401 eller 403. Det er også denne funktion som fremkalder en notits til brugeren når noget går galt - denne notits undertrykkes ved forudindlæsning af data, så vi kun notificerer brugeren ved interaktioner.

8.1.3 API

Frameworket Java Jersey er anvendt til at lave et REST api til eksponering og modificering af data. API'et består af 9 services/endpoints hvor der på de forskellige DTO-objekter er CRUD-metoder implementeret. Et eksempel på et API-endpoint ses nedenfor:

```

@Path("/v1/produce")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class ProduceService {

```

```

// This class implements the methods from MySQLProduceDAO
private ProduceDAO dao = new MySQLProduceDAO();
private IWebInterfaceController controller = new
    WebInterfaceController();

@POST
@Secured( roles = { Role.Pharmacist })
public void createProduce(ProduceDTO produce) throws
    ValidationException, DAException {
    controller.createProduceValidation(produce);
}

@GET
@Path("/{id}")
@Secured( roles = { Role.Pharmacist })
public ProduceDTO getProduce(@PathParam("id") int produceId)
    throws ValidationException, DAException {
    Validation.isPositiveInteger(produceId);
    return dao.readProduce(produceId);
}

@PUT
@Secured( roles = { Role.Pharmacist })
public void updateProduce(ProduceDTO produce) throws
    ValidationException, DAException {
    controller.updateProduceValidation(produce);
}

@DELETE
@Path("/{id}")
@Secured( roles = { Role.Pharmacist })
public void deleteProduce(@PathParam("id") int produceId)
    throws ValidationException, DAException {
    Validation.isPositiveInteger(produceId);
    dao.deleteProduce(produceId);
}

@GET
@Secured( roles = { Role.Pharmacist })
public List<ProduceDTO> getProduceList() throws DAException {
    return dao.getProduceList();
}

@GET
@Path("/overview")
@Secured( roles = { Role.Pharmacist })
public List<ProduceOverviewDTO> getProduceOverview() throws
    DAException {
    return dao.getProduceOverview();
}

```



```

    }
}

```

Vha. filtre og annotations er der implementeret adgangskontrol på de forskellige endpoints således at kun brugere, som er logget ind med rollen "Pharmacist" kan kalder klassens metoder. Deruover har vi specificeret de generelle HTTP-methods GET, POST, PUT og DELETE til hhv. at hente, indsætte, redigere og slette data. Disse fire http-methods er altid brugt på denne måde i oversættelsen af CRUD-operationer til API'et. Med @Path annotateringen kan vi yderligere differentiere mellem flere GET-metoder, så man f.eks. ovenfor både kan bede om alle råverer ved en GET-request på /v1/produce eller en specifik råvare ved at tilføje et ID på sin path (eksempeltvis GET /v1/produce/1 for at få råvaren med ID 1).

Dertil er der udviklet en Validation klasse som bruges til at validere hvorvidt det data der gives i API-kaldene er som vi forventer.

8.1.4 Data Access Layer

Til denne CDIO opgave, oplevede vi nogle concurrency problemer med vores gamle Data Access Layer, fordi vores REST API åbnede en tråd for hvert kald, men vores Data Access Layer kun kunne håndtere en anmodning ad gangen. Derfor har vi lavet om på vores Data Access Layer, ved at erstatte vores Connector klasse med en DataSource klasse, som er en singleton, hvor vi opretter og vedligeholder en Connection Pool, der kan give en mange forbindelser ud.

```

public void createOperator(OperatorDTO opr) throws
    ConnectivityException, IntegrityConstraintViolationException {
    Connection conn = null;
    PreparedStatement stm = null;
    try {
        conn =
            DataSource.getInstance().getConnection();
        stm = conn.prepareStatement("CALL
            create_operator(?,?,?,?,?,?,?)");
        stm.setInt(1, opr.getOprId());
        stm.setString(2, opr.getOprName());
        stm.setString(3, opr.getIni());
        stm.setString(4, opr.getCpr());
        stm.setString(5, opr.getPassword());
        stm.setBoolean(6, opr.isAdmin());
        stm.setString(7, opr.getRole());
        stm.executeUpdate();
    } catch (SQLException e) {
        throw new
            IntegrityConstraintViolationException(e);
    } catch (SQLException e) {
        throw new ConnectivityException(e);
    } finally {
        try { if (stm != null) stm.close(); } catch
            (SQLException e) {};
        try { if (conn != null) conn.close(); } catch
            (SQLException e) {};
    }
}

```

```
    }  
}
```

Som det ses på ovenstående kildekode, har vi også implementeret PreparedStatements, som har en fordel over Statement, da det får givet en SQL statement, når den bliver oprettet og den vil så blive sendt direkte til databasen. Sammenlignet med Statement, bliver PreparedStatement pre-kompileret, hvor statement først skal kompileres når den bliver sendt. Sidst i ovenstående kildekode, lukker vi forbindelserne til Statement og Connection, forbindelsen bliver så returneret til vores Connection Pool. Vi lukker statement og Connection, da det befrier en række ressourcer og i større systemer, hvor man vil opleve problemer, hvis man har mange forbindelser og Statements der ikke bliver lukket.

8.2 Argumentation for test

Til at teste softwaresystemet til håndtering af receptafvejninger, som vi har produceret, har vi gjort brug af flere forskellige former for tests, som alle kan findes i afsnit 8.3.

Til vores Data Access Layer, lavede vi meget tidligt i forløbet, et komplet sæt unit tests til hver Data Access Object klasse. Vi kunne således køre testdreven udvikling, ved at man hver gang man har lavet noget nyt, kan køre den tilhørende unit test klasse og se om det virker. Dette gjorde vi meget brug af da vi lavede total omstrukturering af vores data Access Layer, pga. concurrency problemer. Data Access Object test klasserne kan også bruges til at udelukke fejl i Data Access Layer, hvis man har problemer med det komplette system.

Et andet område hvor det også var oplagt at bruge unit tests til testdreven udvikling, var vores valideringsklasse, hvor vi også anvendte den tilgang.

Til at teste vores vægtsimulator og afvejningsproces, har vi anvendt Black Box testing, hvor der opstilles nogle test cases, der skal afdække så mange af kravene, der er for det testede som muligt, for at sikre at de bliver overholdt.

Til at teste vores REST API, har vi gjort brug af webapplikationen Postman, der er en rigtig god måde at teste om ens API end points fungerer korrekt. Postman kan også bruges til at finde ud af, om et eventuelt problem ligger i front end eller back end, ved at forbigå front end koden.

For at sikre at det komplette system virker helt op til web interfacet, har vi også udført Black Box testing på webinterfacets funktioner. Her har vi vha. test cases forsøget at afdække om kravene til webinterfacet er opfyldte. Da formelle test cases er ret omfangsrige i en rapport, og vi er begrænset i forhold til omfang, viser vi kun et udsnit af vores Black Box testing på webinterfacet. Vi har selvfølgelig testet at alle krav i kravsspecifikationen i afsnit 6.1 er opfyldt.

Slutteligt har vi brugt W3C Markup Validation Service, til at se om vores HTML overholder de syntaktiske regler, og dermed virker i flest mulige browsere.

8.3 Test

8.3.1 Unit Test

Data Access Layer

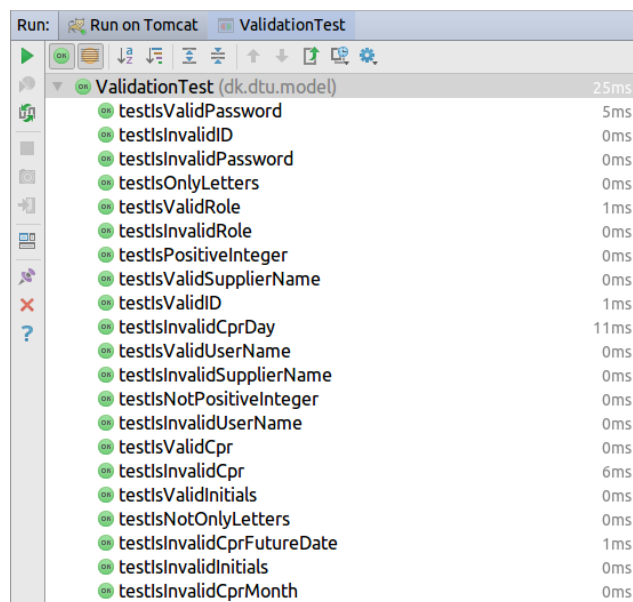
Til udvikling af vores Data Access Layer har vi kørt testdreven udvikling, vha. et sæt unit tests til hver Data Access Object. Testklasserne blev allerede lavet da vi arbejdede med vores database i faget ”Indlende databaser og database programmering”, og blev grundigt beskrevet i vores afsluttende rapport til det fag, hvorfor vi vedlægger dokumentation for disse testklasser, fra den anden rapport, som bilag.

Testklasserne har været guld værd, da vi lavede en total omstrukturering af vores Data Access Objects pga. concurrency problemer. Vi gik fra at have en statisk ”Connector”-klasse, der lånte en enkelt forbindelse ud, og aldrig lukkede Connections, Statements eller ResultSets. Til at have en Singleton ”DataSource”-klasse, med en Connection Pool, der giver Connections ud, som Data Access Objects selv lukker når de er færdige med. Gennem den proces kunne man løbende blot køre testklasserne, uden at ændre noget i dem, for at se om alt var som det skulle være.

Dokumentationen for testklasserne findes i Bilag 11.3.3.

Validation

Java klassen Validation testes ved unit-tests for at sikre at den er i stand til at validere korrektheden af forskellige inputs. Dette sikrer også at videreudvikling på klassen foregår indenfor de forudbestemte rammer for hvordan valideringen skal fungere. Nedenfor ses et screenshot af alle de tests vi foretager på klassen.



Run: Run on Tomcat ValidationTest	
ValidationTest (dk.dtu.model)	25ms
testIsValidPassword	5ms
testIsInvalidID	0ms
testIsInvalidPassword	0ms
testIsValidOnlyLetters	0ms
testIsValidRole	1ms
testIsInvalidRole	0ms
testIsValidPositiveInteger	0ms
testIsValidSupplierName	0ms
testIsValidID	1ms
testIsInvalidCprDay	11ms
testIsValidUserName	0ms
testIsInvalidSupplierName	0ms
testIsValidNotPositiveInteger	0ms
testIsInvalidUserName	0ms
testIsValidCpr	0ms
testIsInvalidCpr	6ms
testIsValidInitials	0ms
testIsValidNotOnlyLetters	0ms
testIsInvalidCprFutureDate	1ms
testIsInvalidInitials	0ms
testIsInvalidCprMonth	0ms

Figur 14: ValidationTest eksekvering

8.3.2 Vægtsimulator

Fra CDIO del 2 har vi taget vægtsimulatoren og tilpasset denne, så beskeder der sendes og modtages fra denne matcher den oprindelige vægt. Dette blev gjort i starten af 3-ugers perioden, så vi ikke var afhængige af den fysiske vægt. På Figur 15 ses to vinduer af PuTTY, hvor der er oprettet en Telnet forbindelse til vægtsimulatoren (localhost), med ip 127.0.0.1 og port 8000, til venstre og til højre en Telnet forbindelse til den fysiske vægt, med ip 169.254.2.3 og port 8000. Her er det vist, hvordan det samme flow er kørt gennem både simulatoren og den fysiske vægt, der til sammenligning agerer - i en vis udstrækning - ens. Figuren viser ikke det komplette flow for afvejningsprocessen og er samtidig af ældre dato (starten af 3-ugers), hvilket betyder, at få mindre ændringer er foretaget ved vægtsimulatoren. Da afviger Figur 15 fra den endelige vægtsimulator, der nu virker identisk med den fysiske vægt.

127.0.0.1 - PuTTY	169.254.2.3 - PuTTY
K 3	K 3
ES	ES
K 3	K 3
K A	K A
RM20 8 "OPR NR [-> " "" "&3"	RM20 8 "OPR NR [-> " "" "&3"
RM20 B	RM20 B
RM20 A "12"	RM20 A "12"
P111 "Anders And [-> "	P111 "Anders And [-> "
P111 A	P111 A
K C 4	K C 4
RM20 8 "BATCH NR [-> " "" "&3"	RM20 8 "BATCH NR [-> " "" "&3"
RM20 B	RM20 B
RM20 A "1234"	RM20 A "1234"
P111 "Salt [-> "	P111 "Salt [-> "
P111 A	P111 A
K C 4	K C 4
P111 ""	P111 ""
P111 A	P111 A
T	T
T S 0 kg	K A 1
P111 "PLACER TARA [-> "	T S 0.000 kg
P111 A	P111 "PLACER TARA [-> "
K C 4	P111 A
S	K C 4
S S 0,2 kg	S
T	S S 0.008 kg
T S 0,2 kg	T
P111 "PLACER NETTO [-> "	T S 0.010 kg
P111 A	P111 "PLACER NETTO [-> "
K C 4	P111 A
S	K C 4
S S 1,3 kg	S
T	S S 0.014 kg
T S 1,5 kg	T
P111 "FJERN [-> "	T S 0.022 kg
P111 A	P111 "FJERN [-> "
K C 4	P111 A
S	K C 4
S S 0 kg	S
S	S S -0.022 kg
S S -1,5 kg	P111 "AFV OK [-> "
P111 "AFV OK [-> "	P111 A
P111 A	K C 4
K C 4	P111 ""
P111 ""	P111 A
P111 A	T
T	T S 0.000 kg
T S 0 kg	

Figur 15: Screenshot af beskeder med både vægt og vægtsimulator

8.3.3 Afvejningsprocessen

Til at teste vores afvejningsprocess, laves der en blackbox test, i dette tilfælde, vil der blive opsat en testcase, hvorefter den testes og dokumenteres. Testcasen dækker en afvejning, som ville den blive brugt i praksis og vi tester derved størstedelen af afvejnings-komponenten.

Formelle Test Cases

ID: TC01

Beskrivelse: Denne testcase tester afvejnings processen for en afvejning i praksis. Vægt simulatoren bliver taget i brug til testen. Kunden har til systemet lavet, et oplæg for hvordan en afvejnings process skal fremløbe, derfor bliver denne process testet og derved kundens krav opfyldt. Vi sætter den samtidig op mod, de opstillede krav til systemet og tjekker at alle krav der omfatter afvejning bliver testet.

Krav:

R.7 Afvejningssystemet skal kunne følgende:

- R.7.1 Registrere laborant ID og hente navn på laborant.
- R.7.2 Validere pågældende laborants password.
- R.7.3 Bede om den produkt batch, der skal afvejes.
- R.7.4 Bede om den råvare batch, der er anvendt for hver råvare.
- R.7.5 På displayet vise hvad laboranten skal gøre og indtaste.
- R.7.6 Afvejningssystemet skal kunne foretage afvejning med bruttokontrol.
- R.7.7 Når afvejningsresultat er accepteret dvs. ligger inden for tolerance og har klaret bruttokontrol, gemmes den i produktbatchkomponenten.

R.8 Under afvejningsprocessen skal status ændres på produktbatchen efterhånden som processenskrider frem.

- R.12.3 En række vejeterminaler udstyret med vejeplade, tastatur og display, alternativt kan det være en vægtsimulator (se Krav - Vægtsimulator, R.14-R.21)
- R.12.4 ASE, som er et program der styrer vejeterminaler og gemmer data fra disse i databasen.

R.14 Simulatoren skal simulere vægtens opførsel.

R.15 Simulatoren skal kunne modtage og sende kommandoer over TCP.

R.16 Simulatoren skal vise input fra tastatur på display.

R.18 Det skal være muligt at sendte følgende 9 kommandoer:

- R.18.1 S: Send stabil afvejning.
- R.18.2 T: Tarér vægten.
- R.18.3 D: Skriv i vægtens display.
- R.18.4 DW: Slet vægtens display.
- R.18.5 P111: Skriv max. 30 tegn i sekundært display.
- R.18.6 RM20 8: Skriv i display, afvent indtastning.
- R.18.7 K - Skifter vægtens knap-tilstand.
- R.18.8 B - Sæt ny bruttovægt.
- R.18.9 Q - Afslut simuleringen.

R.19 Følgende kommandoer skal kunne afvikles fra den simulerede brugergrænseflade på vægten:

R.19.1 Tarér vægten.

R.19.2 Sæt ny bruttovægt.

R.19.3 Afslut simuleringen.

R.20 Simulatoren skal lytte på port 8000.

R.21 Simulatoren skal kunne virke som ‘stand in’ for den fysiske vægt.

Forudsætning:

Man har vægt eller vægtsimulator til rådighed.

Man er minimum laborant og har en bruger, der kan logge ind på vægten.

Efterfølgende betingelser:

Man har lavet en afvejning og det er indsat i databasen.

Status for det produktbatch for det afvejede, vil blive ændret til afsluttet.

Test procedure:

1. Laver en typisk afvejnings process, som vist på opgavebeskrivelsens bilag 4.

Test data:

1. Opr Nr: 1
2. Opr PW: lKje4fa
3. Batch: 4
4. Place tara: 0.5 kg
5. RB ID: 3
6. Place netto: 1.5 kg

Forventede resultater: Vi forventer at processen forløber ligesom beskrevet i opgavebeskrivelsen på bilag 4. Det vil sige at systemet, svarer tilbage med beskeder om hvad brugeren skal gøre, samt give beskeder om fejl og om det der er skrevet er korrekt. Der henvises også til afsnit 7.2.1 for en nærmere gennemgang af afvejnings sekvensen.

Faktiske resultater:

1. Opr Nr?
 - Angelo A
2. Opr PW?
 - Login OK!
3. Batch?
 - capricciosa
4. tomat
5. Place tara
6. Place 1.5kg
7. RB ID?
8. Remove Gross
9. Weighting OK!

Status: Godkendt.

- Man skal identificerer sig overfor systemet ved laborant ID og hente navnet på laboranten.
- Displayet viser hvad man skal gøre
- Gemmes i productbatchkomponenten ved korrekt afvejning og bruger bruttokontrol.
- Status ændres ved afsluttet afvejningsprocess.

- Simulator afspejler den rigtige vægt.
- Kommunikerer over TCP
- Back-end benytter sig af kommandoerne i krav R.18.
- Simulatoren kan Tårer, sætte ny bruttovægt og afslutte en simulation.
- Lytter på port 8000

Testet af: Mads Pedersen

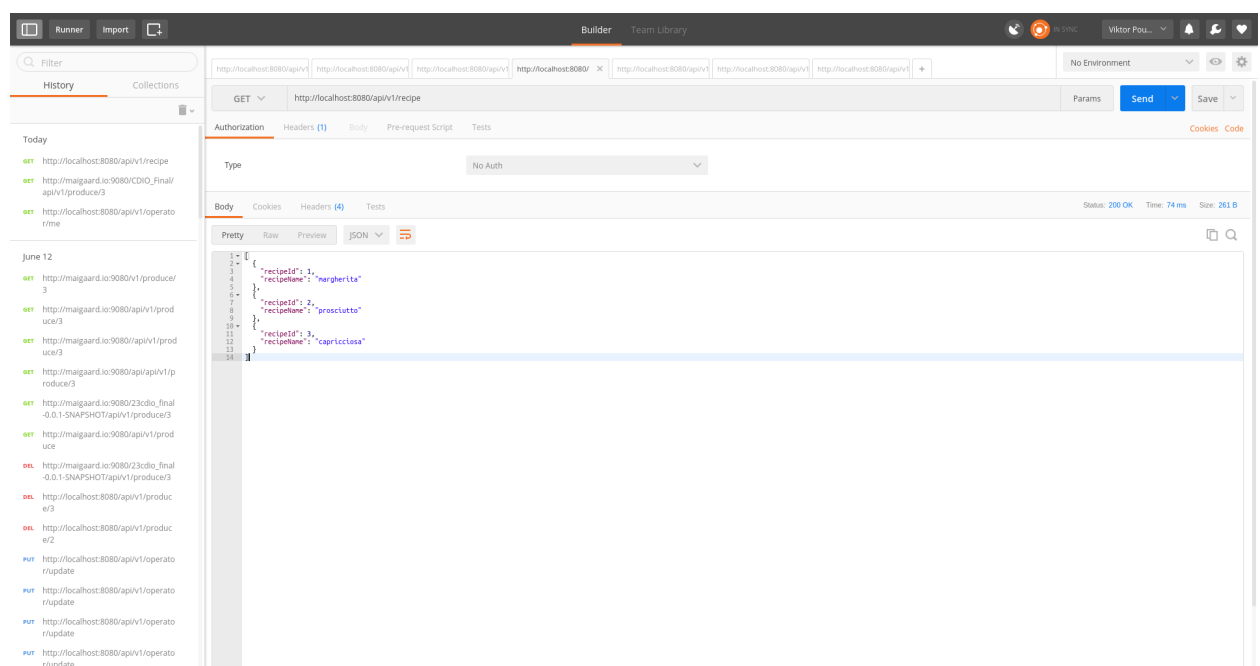
Dato: 14.06.2017

Testmiljø: Windows 10 Home 64-bit.

Eclipse Java EE IDE for Web Developers. Version: Neon.1a Release (4.6.1).

8.3.4 Postman

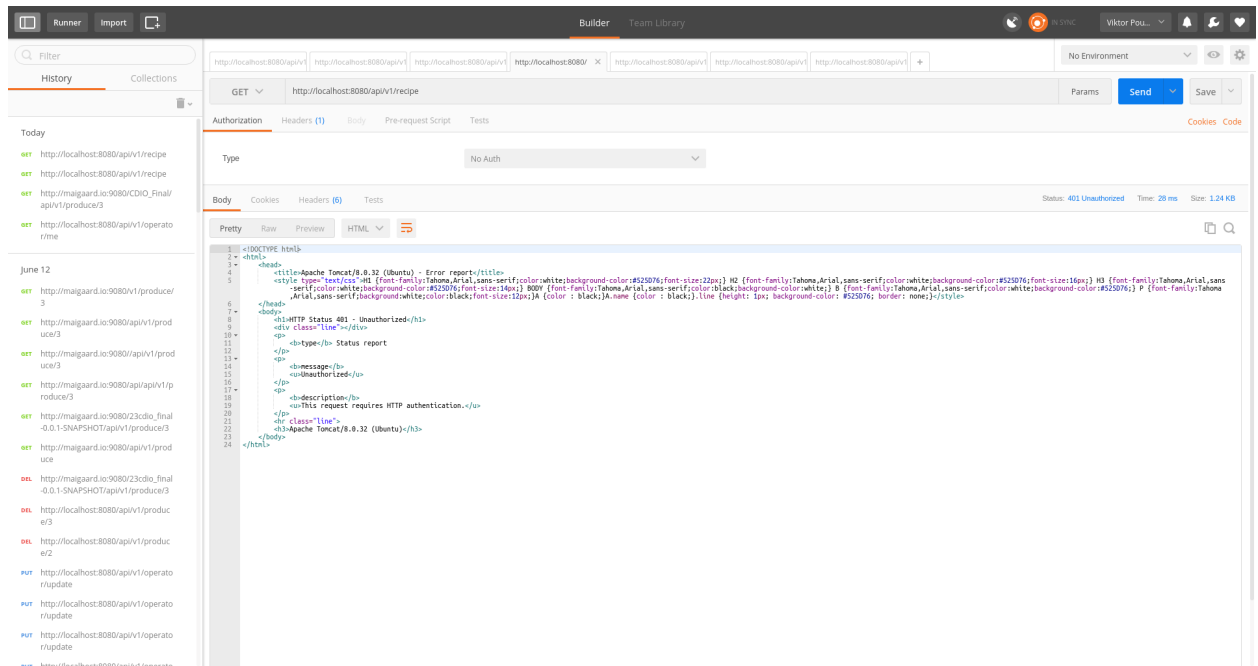
Under implementering af REST API'et har vi anvendt postman til løbende at verificere de forskellige endpoints. Nedenfor ses et kald `/api/v1/recipe`, hvilket returnerer en liste af alle systemets opskrifter.



Figur 16: Screenshot af postman request med valid token

Den returnerede data er på formatet JSON, hvilket gør det meget nemt at sende videre til vores mustache templates.

Det samme kald ses herunder, dog uden en valid Json Web Token. Uden denne kan API'et ikke vide om brugeren har logget korrekt ind og svarer derfor med fejlkoden 401 Unauthorized

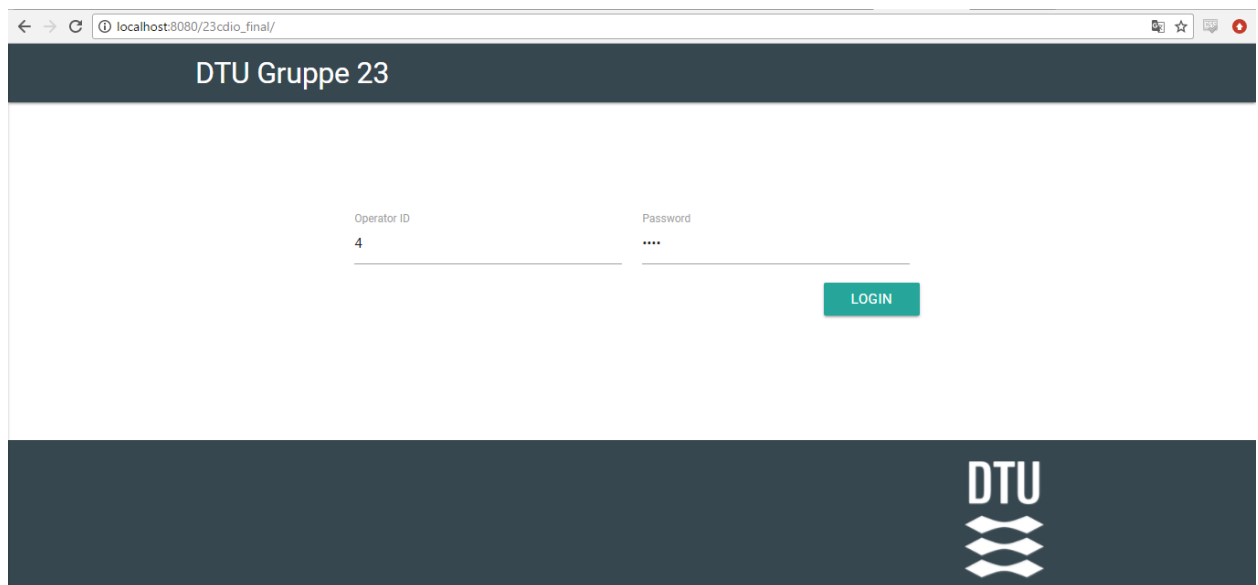


Figur 17: Screenshot af postman request uden token

Bruger man en token som ikke kan verificeres eller har brugeren for lav adgangsrolle-niveau svares der med 403 - Forbiden.

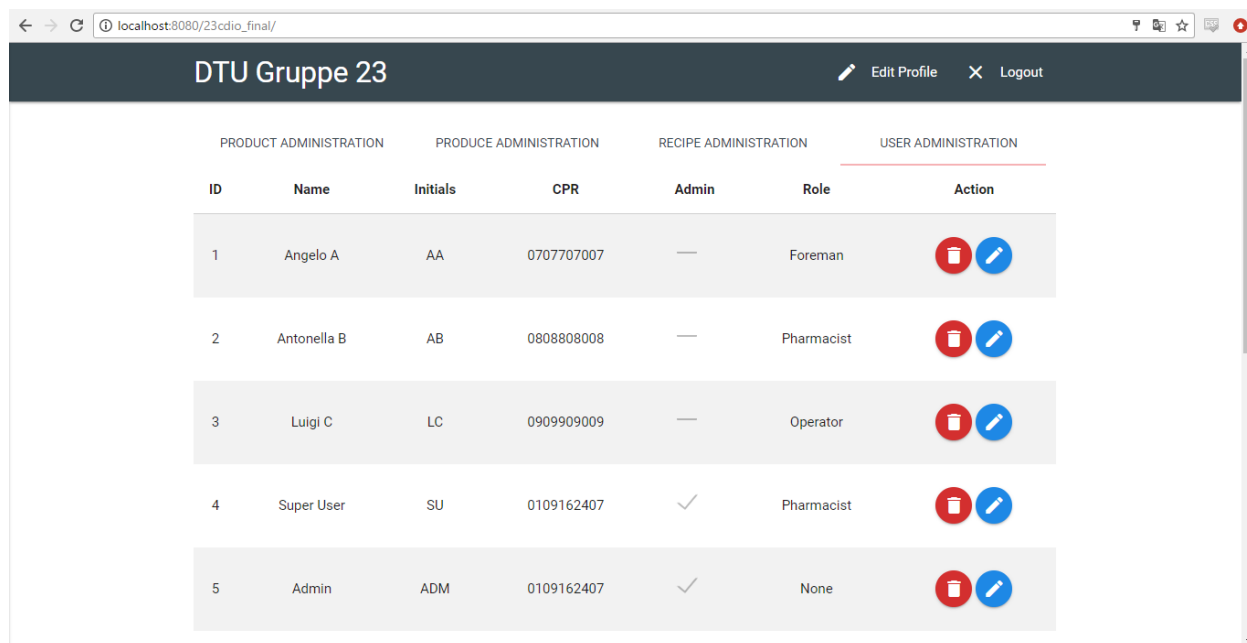
8.3.5 Webinterface











Til test af webinterfacet, som er en del af det færdige produkt, benytter vi os af Black Box testing. Her afprøver vi alle funktionerne på websitet og om alle komponenterne virker.



Figur 18: Screenshot af webinterface login side

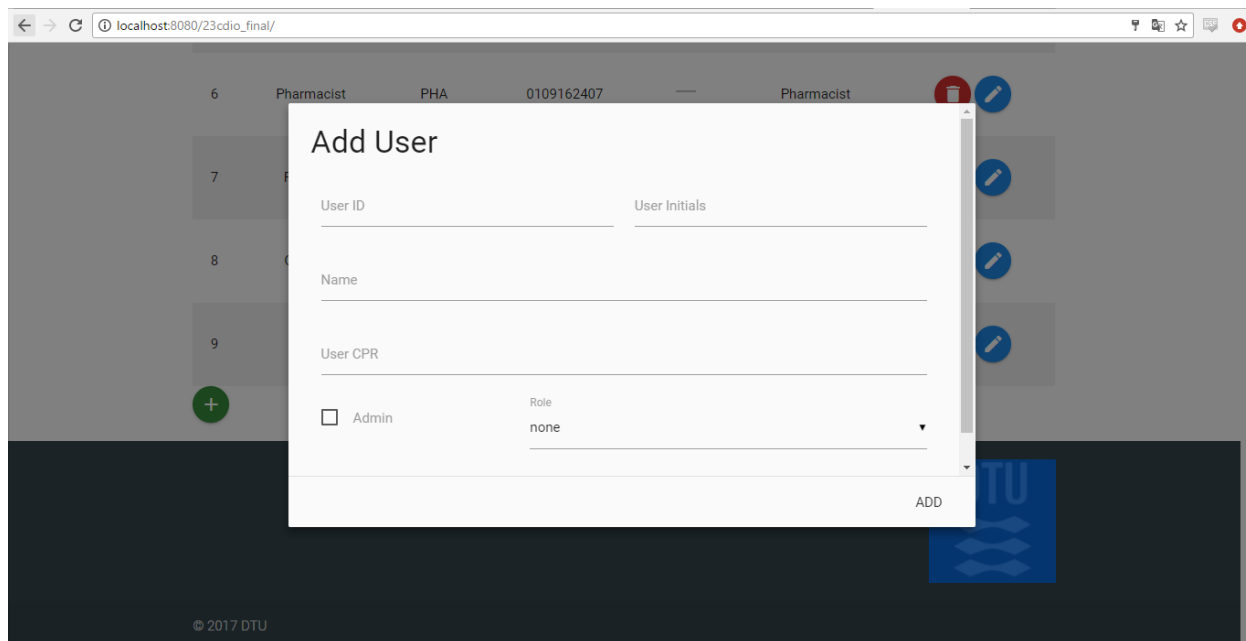
På figur 18 ser vi den side, man bliver mødt af når man først åbner siden. Her har man mulighed for at logge ind. Vi tester så hvorledes login validering virker korrekt, altså at man kun kan logge ind hvis man har en konto på siden. Dette virker, man vil ikke have mulighed for at logge ind hvis man ikke har en konto. Hvis man har en konto og logger ind vil man blive mødt af en ny side.



DTU Gruppe 23						
PRODUCT ADMINISTRATION				PRODUCE ADMINISTRATION		
RECIPE ADMINISTRATION				USER ADMINISTRATION		
ID	Name	Initials	CPR	Admin	Role	Action
1	Angelo A	AA	0707707007	—	Foreman	 
2	Antonella B	AB	0808808008	—	Pharmacist	 
3	Luigi C	LC	0909909009	—	Operator	 
4	Super User	SU	0109162407	✓	Pharmacist	 
5	Admin	ADM	0109162407	✓	None	 

Figur 19: Screenshot af webinterface administrations side

På figur 19 ses siden man bliver henvist til hvis man har logget ind. Her har man en række muligheder. Man kan redigere og slette brugere, vist med en blå og rød knap. I toppen kan man redigere sin egen profil eller logge ud. I bunden, som man ikke kan se på dette screenshot, ses en grøn knap, som bruges til at tilføje brugere i systemet. Først afprøves redigering profilen, dvs ens egen brugerinformation, ved at trykke edit profil, i toppen af siden. Her er så et popup, hvor man har mulighed for at redigere profilen, samme popup som bruges til at redigere eller tilføje på siden, hvis man har rettighed til det. Dette ses også på figur 20. Når man har redigeret ens egen profil eller en anden profil, vil informationen blive ændret, dvs det virker som ønsket, samt trykkes der på en røde knap som er slet, vil den bruger blive slettet fra systemet. Vi kan altså bekræfte at alle CRUD (create, read, update, delete) funktioner, virker som ønsket.



Figur 20: Screenshot af webinterface redigering af profil

Product Batch ID	Recipe ID	Recipe Name	Status
1	1	margherita	2

Produce Batch ID	Produce Name	Supplier	Nom. Netto	Opr. ID
1	dej	Wawelka	10.05	1
2	tomat	Knoor	2.03	1
4	ost	Ost og Skinke A/S	1.98	1

2	1	margherita	2
3	2	prosciutto	2
4	3	capricciosa	1
5	3	capricciosa	0

Figur 21: Screenshot af webinterface

På figur 21, ses administrationssiden for productbatch-administration. Her har man mulighed for at trykke på de forskellige productbatches og man vil så få givet listen productbatchcomponenter. Vi har altså mulighed for let, at læse informationer om hvert productbatch.

Vi kan altså konkludere, ved at have prøvet alle funktionerne på hjemmesiden, at de virker som de skal. Der er testet både med korrekt og ikke-korrekt information, for samtidig at teste sikkerheden og valideringen på hjemmesiden. Kravene for web interfacet er derfor opfyldt, da det overholder de krav som kunden har stillet.

8.3.6 W3 Validering

Der har ikke været krav til supportering af ældre browser-versioner, hvorfor vores html markup er valideret til HTML4.1 standard med W3 Validator. W3 fandt ingen fejl i vores HTML og derfor burde visningen fungere på tværs af alle moderne browsere. Nedenfor ses resultatet af W3Validatoren:

The screenshot shows the W3 HTML Checker interface. At the top, there's a blue header with the text "Nu Html Checker". Below it, a small note states: "This tool is an ongoing experiment in better HTML checking, and its behavior remains subject to change". The main section is titled "Showing results for contents of text-input area". Under "Checker Input", there are tabs for "source" (selected), "outline", and "image report", along with an "Options..." button. A "Check by" dropdown menu is set to "text input". The main text area contains the following HTML code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <meta charset="UTF-8">
  <title>CDIO 2.3</title>
  <!-- Materialize CSS & Icons -->
  <link rel="stylesheet" href="css/materialize.min.css">
  <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">

  <!-- Mustache JS -->
  <script src="js/mustache.js" type="text/javascript"></script>
</-->
```

Below the code area is a "Check" button. A note below the button says: "Use the Message Filtering button below to hide/show particular messages, and to see total counts of errors and warnings." Below this is a "Message Filtering" button. At the bottom, a green status bar displays the message: "Document checking completed. No errors or warnings to show."

Figur 22: W3 Validering af HTML

8.4 Test Tracing

9 Operation

9.1 Konfiguration

9.1.1 Afvejningsprocessen

For at køre afvejningsprocessen forventes, at projektet (23cdio_final¹) er importeret lokalt (f.eks. i Eclipse) og at en af vægtene er tilsluttet eller vægtsimulatoren (23cdio_final_weightsimulator²) er startet op lokalt på den enhed der anvendes. Ønskes det at benytte vægtsimulatoren skal projektet 23cdio_final_weightsimulator importeres, hvorefter Main.java, der findes i pakken *controller*, køres som en Java Applikation. Vægten eller vægtsimulatoren forventes at være tændt eller startet inden processen køres.

- Når projektet 23cdio_final er importeret lokaliseres pakken *dk.dtu*
- Klassen MainWeightProcess.java, der befinder sig i pakken *dk.dtu*, åbnes
- Her kan IP og Portnummer indstilles til at matche den pågældende vægt der anvendes
Vægtsimulator - IP: localhost, Port: 8000
Fysisk vægt - IP: 169.254.2.3 eller 169.254.2.2, Port: 8000
- Når IP og Port er indstillet kan klassen MainWeightProcess.java køres som en Java Applikation.
- Afvejningsprocessen er nu sat i gang, og burde vise sig i vægtens display

9.1.2 Webinterface

For at starte projektets API og frontend kræves følgende redskaber installeret:

- Java 1.8 eller nyere
- Tomcat8
- Apache Maven
- En internetforbindelse

Projektet clones eller downloades fra github (master branch). Herefter køres følgende Maven goals i rækkefølge:

1. clean
2. compile
3. war:war

¹https://github.com/DTU23/23cdio_final

²https://github.com/DTU23/23cdio_final_weightsimulator

Efter dette kan war-filen deploy'es fra et tomcat manager-vindue eller alternativt kan projektet køres i eclipse som et dynamisk webprojekt, hvor ovenstående Maven Goals kan ignoreres. Login-boksen er sat til automatisk at udfylde superbrugerens login, men hvis andre rettigheder ønskes afprøvet kan følgende logins bruges:

ID 5: Administrator uden rolle-adgang

ID 6: Pharmacist

ID 7: Foreman

ID 8: Operator

ID 9: None / deaktiveret bruger

Alle ovenstående logins har koden root.

Alternativt kan projektet testes på http://46.101.131.115:8080/23cdio_final-0.0.1-SNAPSHOT/, hvor det vil være kørende indtil eksamen er afsluttet.

10 Konklusion

Det er lykkedes at implementere et komplet webinterface til administration af alle de forskellige relationer fra vores database, samt en afvejnings-styrings-enhed, der begge lever op til alle kravene i vores kravsspecifikation. Hvis ”kunden” oplever at disse ikke stemmer 100% overens med opgavebeskrivelsen, skyldes det at vi har tilladt os lidt kunstnerisk frihed, for at gøre brugervenligheden af systemet lidt bedre. Der er lavet sikring af API-end points ved JWT, templating med mustache.js og en ekstern database-opkobling - elementer som bidrage til systemets overordnede robusthed og mulighed for videreudvikling.

I afsnit 8.3.2 ses det at vi har implementeret en identisk simulator til den fysiske vægt. Derved er det muligt for os at simulere hele vægtinteraktionen, og arbejde med vores controller, til at afvikle afvejningsprocessen for brugeren.

Testopsætningen har hjulpet projektet med at holde styr på funktionaliteten af diverse metoder i java-laget. Det har været vigtigt at vores Data Access Objekter og statiske valideringsmetoder fungerer upåklageligt og her har unit-tests sikret at der under hele udviklingsprocessen blev holdt styr på dette.

Forslag til forbedringer

I det oprindelige databasedesign var der lagt op til at flere af primærnøglerne (ID'er) kunne auto-generes under oprettelse, hvilket ”kunden” dog i sidste ende ikke ønskede at have implementeret. Derfor er det nu krævet at man selv vælger ID på oprettelse af nye elementer i systemet.

Vores afvejnings-styrings-enhed er ikke sat op til at håndtere flere vægte på samme tid. Den er heller ikke sat op til at køre hele tiden og selv søge efter om vægten/vægtene i dens kartotek er online. Disse er begge features der kunne være spændende at tilføje, som vi ikke har kunnet nå.

11 Bilag

11.1 Tidsregistrering

Dato	Deltager	Design	Impl.	Test	Dok.	Andet	I alt
01/06/2017	Christian Niemann	1	3	1		1	6.0
01/06/2017	Frederik Værnegaard	1	3	1		1	6.0
01/06/2017	Mads Pedersen	1			4	1	6.0
01/06/2017	Viktor Poulsen	1	4			1	6.0
02/06/2017	Christian Niemann		3				3.0
02/06/2017	Frederik Værnegaard		3				3.0
02/06/2017	Viktor Poulsen		2				2.0
04/06/2017	Frederik Værnegaard		2.5				2.5
04/06/2017	Mads Pedersen				1		1.0
05/06/2017	Christian Niemann		5	0.5			5.5
05/06/2017	Frederik Værnegaard		3.5				3.5
05/06/2017	Mads Pedersen		1		3		4.0
05/06/2017	Viktor Poulsen		2.5				2.5
06/06/2017	Christian Niemann	4	3.5			1	8.5
06/06/2017	Frederik Værnegaard		6.5	1		1	8.5
06/06/2017	Mads Pedersen		5		1		6.0
06/06/2017	Viktor Poulsen	4	2				6.0
07/06/2017	Christian Niemann		2.5				2.5
07/06/2017	Frederik Værnegaard		3	1			4.0
07/06/2017	Viktor Poulsen		4				4.0
08/06/2017	Christian Niemann		4	1		0.5	5.5
08/06/2017	Frederik Værnegaard		2.5				2.5
08/06/2017	Mads Pedersen		3.5				3.5
08/06/2017	Viktor Poulsen		0.5				0.5
09/06/2017	Christian Niemann		5	5			10.0
09/06/2017	Frederik Værnegaard		6	2			8.0
09/06/2017	Viktor Poulsen		6	4			10.0
10/06/2017	Christian Niemann		0.5	0.5	0.5		1.5
10/06/2017	Mads Pedersen				2		2.0
10/06/2017	Viktor Poulsen		2				2.0
11/06/2017	Frederik Værnegaard				5		5.0
11/06/2017	Mads Pedersen				5		5.0
11/06/2017	Viktor Poulsen				0.5		0.5
12/06/2017	Christian Niemann		7	1	0.5	1	9.5
12/06/2017	Frederik Værnegaard		4	0.5			4.5
12/06/2017	Mads Pedersen		1		2.5		3.5
12/06/2017	Viktor Poulsen		10				10.0

Tabel 3: Detaljeret Timeregnskab del 1

Dato	Deltager	Design	Impl.	Test	Dok.	Andet	I alt
13/06/2017	Christian Niemann	1	4	3			8.0
13/06/2017	Frederik Værnegaard		3	1.5	2		6.5
13/06/2017	Viktor Poulsen	1.5	4		1		6.5
14/06/2017	Christian Niemann			1.5	3.5		5.0
14/06/2017	Frederik Værnegaard		1	1	4.5		6.5
14/06/2017	Mads Pedersen		3	4			7.0
14/06/2017	Viktor Poulsen		1		3		4.0
15/06/2017	Christian Niemann			2	6		8.0
15/06/2017	Frederik Værnegaard				3		3.0
15/06/2017	Viktor Poulsen				1		1.0
15/06/2017	Viktor Poulsen		3		2		5.0
16/06/2017	Christian Niemann			0.5	0.5		1.0
16/06/2017	Mads Pedersen				3		3.0
16/06/2017	Viktor Poulsen		1		1		2.0

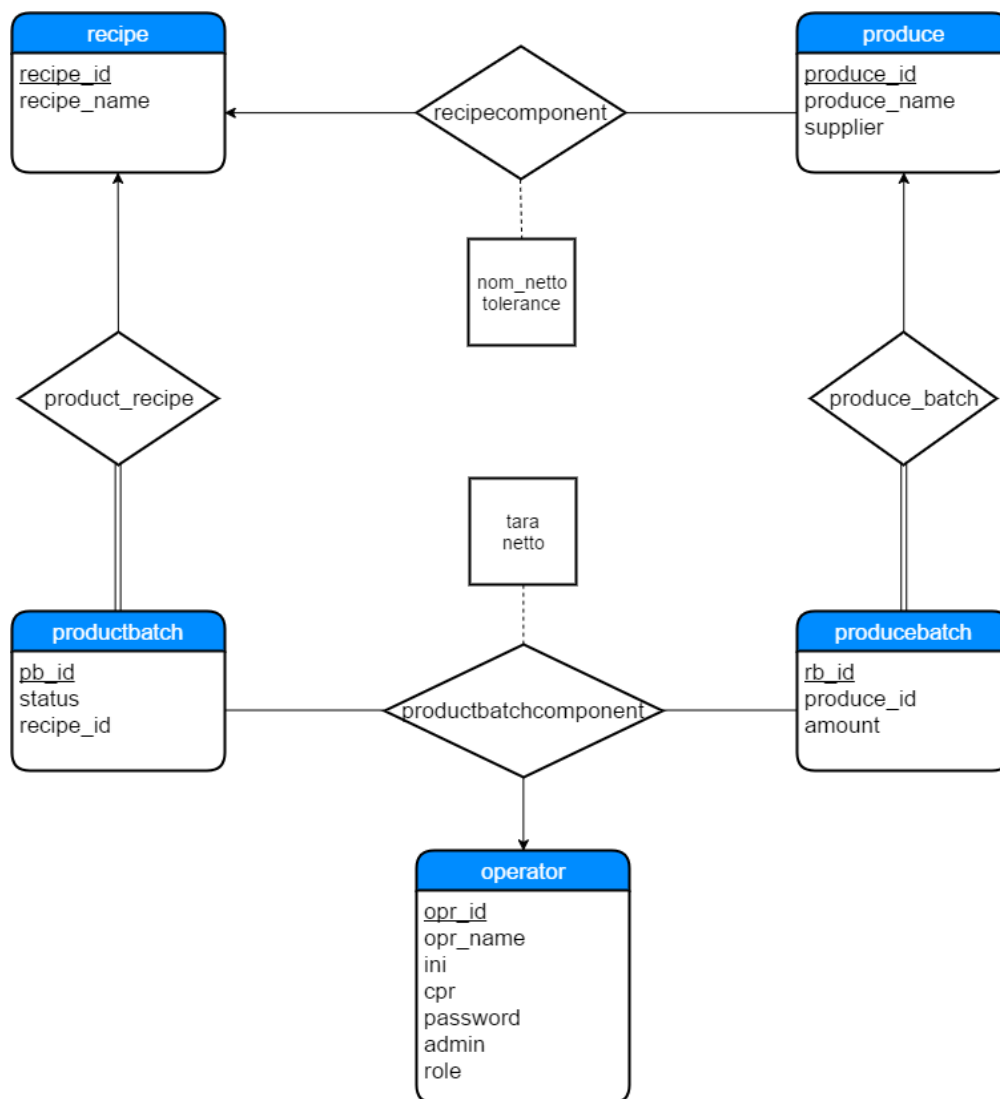
Tabel 4: Detaljeret Timeregnskab del 2

11.2 Kildekode

Al kildekode samt udviklingshistorik kan gennemgås på https://github.com/DTU23/23cdio_final

11.3 Database Dokumentation

11.3.1 ER-Diagram



Figur 23: Entity Relation Diagram

På Figur 23 ses et Entity Relationship Diagram af vores database, der beskriver forholdene mellem de forskellige relationer i databasen.

Kigger vi på relationen mellem recipe og produce (recipecomponent), vil én recept bestå af mange råvarer, hvorfor vi har en *one-to-many* relation. Her bør enhver recept indgå med nogle bestemt tilknyttede råvarer og mængden af disse. Relationen recipecomponent har yderligere to attributer tilknyttet, *nom_netto* og *tolerance*, der er vist i kvadratet, som er tilknyttet med en stiblet linje.

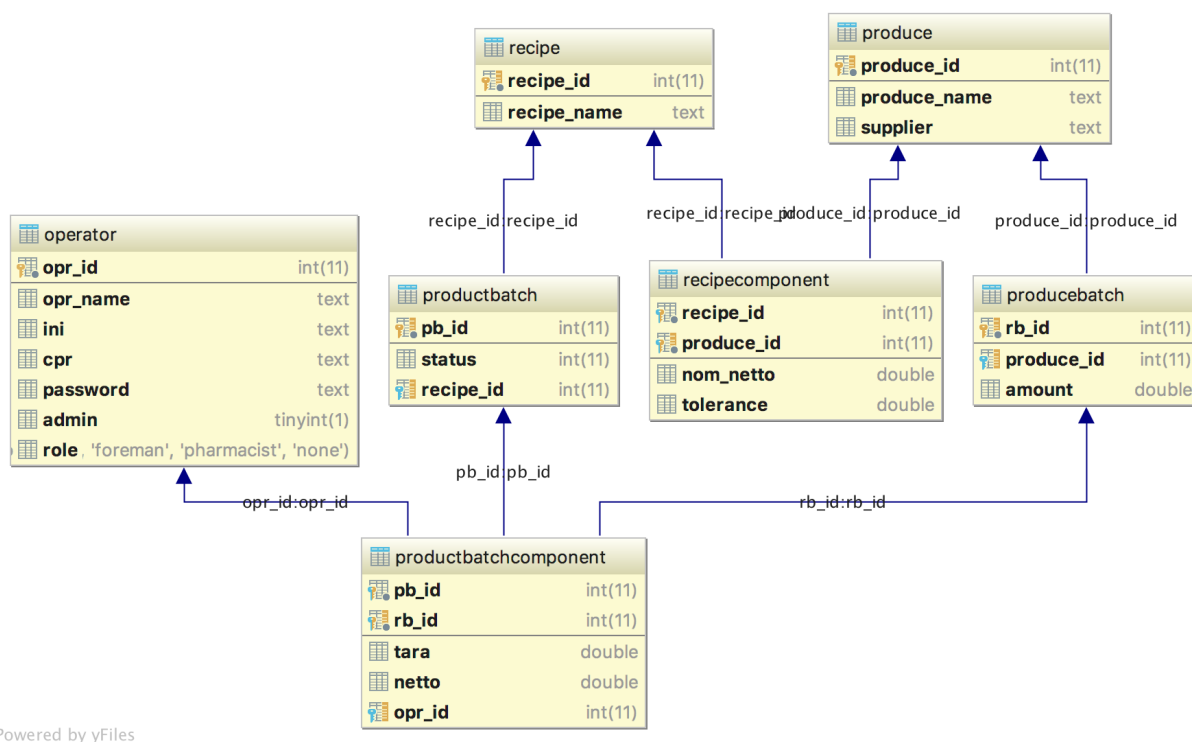
For relationen mellem recipe og productbatch (product_recipe) vil enhver produktbatch indeholde én recept. Omvendt kan en recept indgå i flere forskellige produktbatches, hvortil vi har en *one-to-many* relation. Det samme gør sig gældende for relationen mellem produce og producebatch (produce_batch). Her ser vi igen, at en instans af råvarebatches indeholder én råvare, hvorimod en

instans af en råvare kan indgå i flere forskellige råvarebatches.

Den sidste relation, productbatchcomponent, er en kombination af productbatch, producebatch og operator, samt to yderligere attributter, *tara* og *netto*, der vedrører afvejningerne. Relationen er en *many-to-many* relation for productbatch og producebatch, hvortil der tilknyttes en operatør. Relationen vil altså kunne indeholde flere forskellige produktbatches, samt flere forskellige tilknyttede råvarebatches. Derudover vil der for enhver afvejning af en produktbatch skulle tilknyttes én operatør - og samtidig kan én operatør være tilknyttet flere forskellige afvejninger, heraf *one-to-many* relationen for operator.

11.3.2 Relationsskema

På figur 24 ses relationsskemaet, der repræsenterer den endelige database. Af større ændringer har vi foretaget en oversættelse for at ensarte måden vi navngiver variable gennem hele projektet.



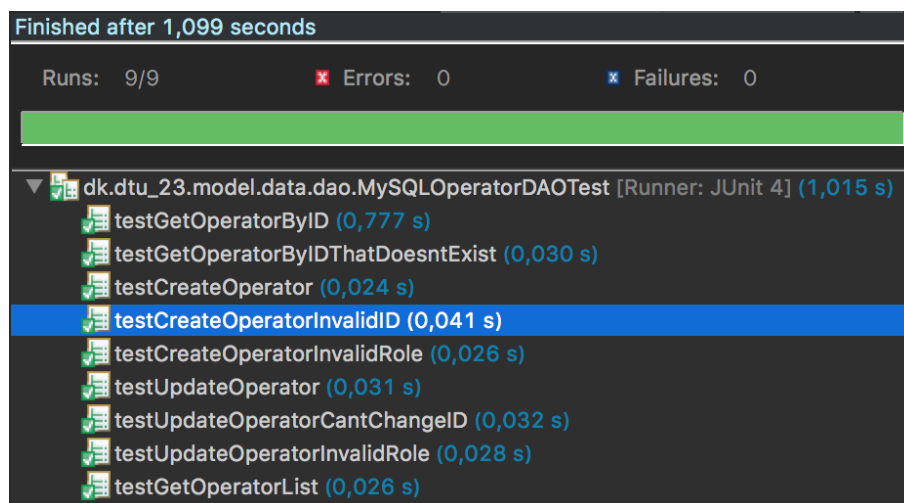
Figur 24: Relationsskema over den endelige database.

11.3.3 Database Test

MySQLOperatorDAOTest

MySQLOperatorDAOTest er en Junit test-klasse, der bruges til at teste om metoderne i JDBC-laget, der har med operator-relationen at gøre, samt relaterede views og stored procedures, virker efter hensigten. Klassen består af 9 test-cases, hvoraf de 4 er positive og de 5 er negative. På figur 25 ses et screenshot fra Eclipse, hvor test-klassen er kørt. Man kan se at der er et grønt flueben

ved alle test-cases, og hvilket betyder at alle metoder, views og stored procedures med relation til operator-relationen, opfører sig som forventet.



Figur 25: Screenshot fra Eclipse, der viser resultatet af test-klassemens kørsel.

Test cases

testGetOperatorByID Dette er en positiv test, der som navnet tydeligt angiver, prøver at hente en operatør ud fra et operatør ID, og ser om det data der kommer retur, stemmer overens med det data vi ved der bør ligge i databasen.

testGetOperatorByIDThatDoesntExist Dette er en negativ test, der prøver at hente en operatør, der ikke findes i databasen. Testen tjekker om der kommer en exception med en beskrivende fejlmeddelelse.

testCreateOperator Dette er en positiv test, der forsøger at oprette en operatør ved navn Don Juan med operatør ID 5 i databasen. Og bagefter hentes operatøren med ID 5 og tjekker om oplysningerne er korrekte.

testCreateOperatorInvalidID Dette er en negativ test, hvor der forsøges at oprette en bruger med ID 3, der allerede er brugt. Derefter tjekkes at det ikke er den nye bruger, der ligger i databasen med ID 3, og at der er kommet en fejlmeddelelse da man forsøgte at oprette ny bruger med et eksisterende ID.

testCreateOperatorInvalidRole Dette er en negativ test, der tester at man ikke kan få lov, til at oprette sig i systemet med andre roller end Operator, Foreman, Pharmacist eller None. Det tjekkes også at der kommer en fejlmeddelelse når man forsøger.

testUpdateOperator Dette er en positiv test, der tjekker om man kan ændre brugeren Don Juans initialer til DoJu i stedet for DJ.

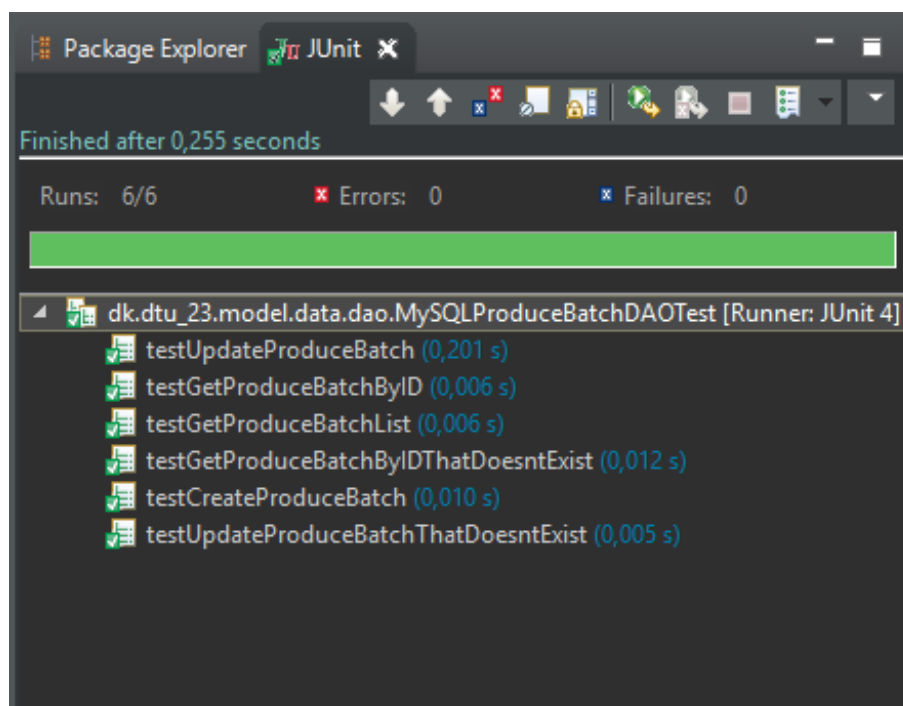
testUpdateOperatorCantChangeID Dette er en negativ test, der forsøger at ændre en operatørs ID, det tjekkes at den berørte tuppel forbliver som den var, at der ikke kommer nogen tuppel med det nye ID, og at der kommer en fejlmeddelelse.

testUpdateOperatorInvalidRole Dette er en negativ test, der forsøger at ændre en operatørs rolle til CEO, som er en ugyldig rolle i systemet. Det tjekkes at operatørens rolle forbliver som den var, samt at der kommer en fejlmeddelelse.

testGetOperatorList Dette er en positiv test, der tjekker at man via view'et operator_list, kan få en liste med alle operatørernes oplysninger, på nær deres kodeord.

MySQLProduceBatchDAOTest

MySQLProduceBatchDAOTest er en JUnit test-klasse, hvor vi tester metoderne fra vores MySQLProduceBatchDAO. Vi tester hvorvidt de forskellige metoder i DAO-klassen modtager/sender det korrekte data til/fra databasen. Klassen indeholder 6 forskellige tests som følger. Hvis testene fejler er det fordi, at testen er kørt mere end én gang, hvilket forårsager, at nye tupplernes identifikationsnummer bliver auto-incrementet mere end én gang - dette giver et forkert forventet id ift. testen.



Figur 26: Resultat af JUnit testene i MySQLProduceBatchDAOTest

JUnit Tests i MySQLProduceBatchDAOTest

testGetProduceBatchByID Dette er en positiv test, hvor vi tester om metoden getProduceBatch(rbId) i MySQLProduceBatchDAO returnerer et korrekt ProduceBatchDTO-objekt når vi giver et specifikt id som input.

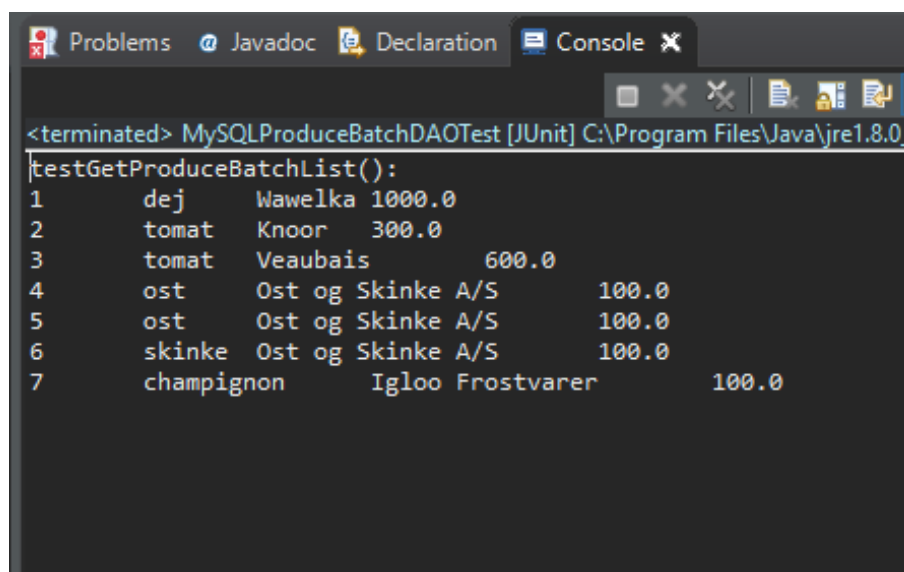
testGetProduceBatchByIDThatDoesntExist Dette er en negativ test, hvor vi tester metoden getProduceBatch(rbId) - denne gang med et input-id, som ikke findes i databasen. Vi tjekker om vi fanger en DALException med en tilhørende fejlbesked.

testGetProduceBatchList Dette er en positiv test, hvor vi forsøger at hente et helt view op til vores applikations-lag. Vi tester metoden getProduceBatchList(), og printer den returnerede liste ud.

testCreateProduceBatch Dette er en positiv test, hvor vi tester metoden createProduceBatch (ProduceDTO, amount). Vi forsøger at oprette en tilfældig råvarebatch i vores database, hvorefter vi henter denne ud igen med getProduceBatch(rbId), for at tjekke om den er blevet oprettet i vores database.

testUpdateProduceBatch Dette er en positiv test, hvor vi tester metoden updateProduceBatch(ProduceDTO, amount). Vi tjekker om vi kan opdatere vores data og at dataen er opdateret når vi henter objektet op på ny.

testUpdateProduceBatchThatDoesntExist Dette er en negativ test, som forsøger at opdatere en råvarebatch, der ikke findes.

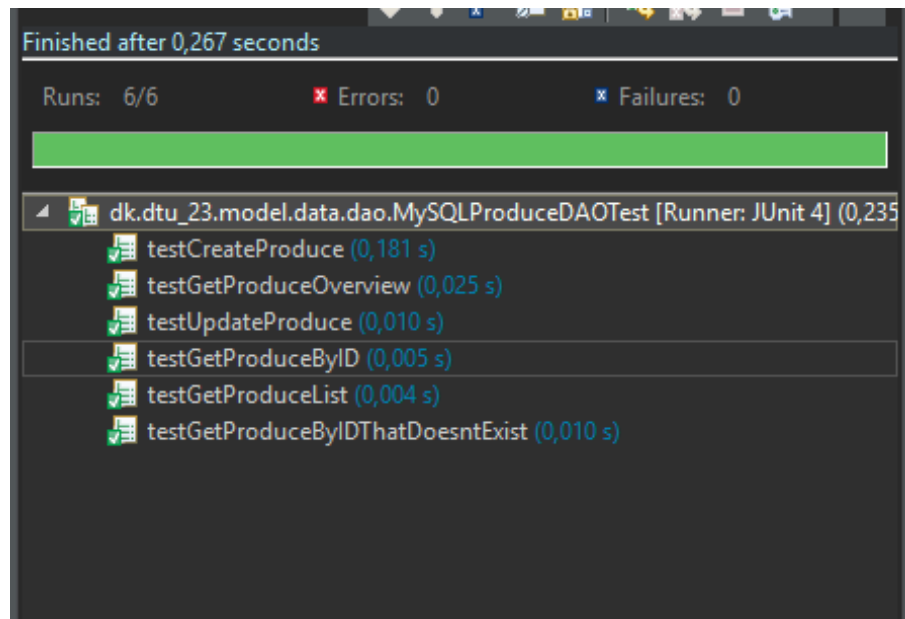


```
<terminated> MySQLProduceBatchDAOTest [JUnit] C:\Program Files\Java\jre1.8.0_101
testGetProduceBatchList():
1      dej      Wawelka 1000.0
2      tomat    Knor    300.0
3      tomat    Veaubais 600.0
4      ost      Ost og Skinke A/S 100.0
5      ost      Ost og Skinke A/S 100.0
6      skinke   Ost og Skinke A/S 100.0
7      champignon Igloo Frostvarer 100.0
```

Figur 27: Udprint fra testGetProduceBatchList()

MySQLProduceDAOTest

MySQLProduceDAOTest er en JUnit test-klasse, der tester metoderne fra vores Data Access Object, MySQLProduceDAO. I vores setUp() opretter vi et Connector-objekt, samt et MySQLProduceDAO objekt - og i vores tearDown() nulstiller vi vores MySQLProduceDAO objekt ved at sætte dette til null efter hver test. Klassen indeholder 6 forskellige tests som følger. Nogle af testene i denne klasse vil fejle, hvis testene køres mere end én gang, idet vi auto-incrementer identifikationsnummeret - og netop det forventede id i testene er hardcodede.



Figur 28: Resultat af JUnit testene i MySQLProduceDAOTest

JUnit Tests i MySQLProduceDAOTest

testGetProduceByID Dette er en positiv test, hvor vi tester metoden `getProduce(pId)` i `MySQLProduceDAO`. Her tjekkes der om vi får det korrekte Data-Transfer Object retur (`ProduceDTO`) via `getProduce(pId)`, hvor der indsættes et specifikt ID.

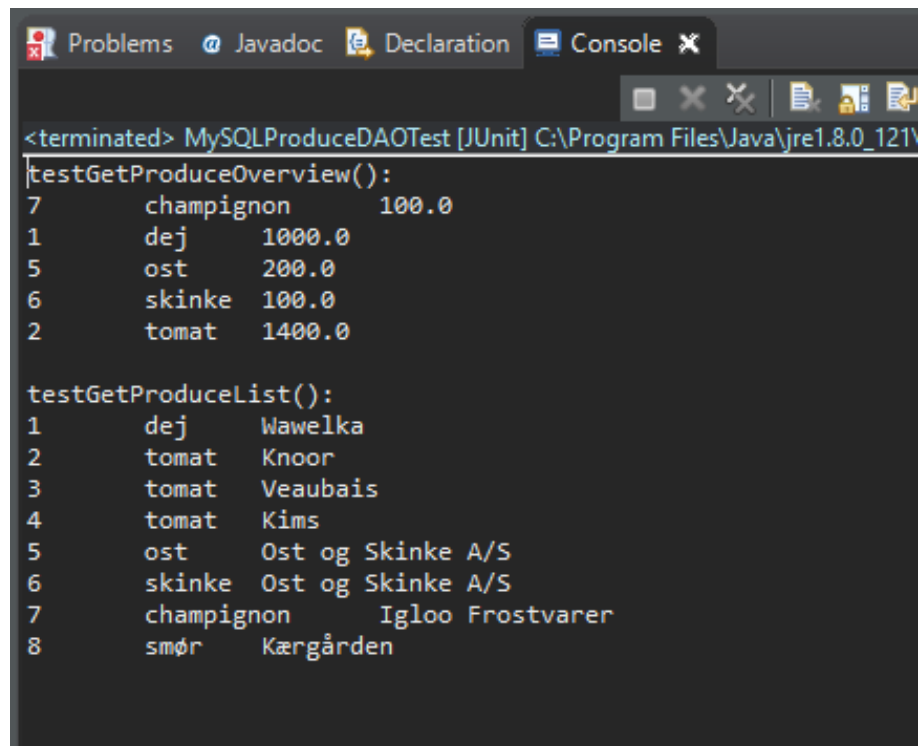
testGetProduceByIDThatDoesntExist Dette er en negativ test, hvor vi tester metoden `getProduce(pId)` i `MySQLProduceDAO` - denne gang med et input-id, der ikke findes i databasen. Vi tjekker om vi fanger en `DAException` med en tilhørende fejlbesked.

testGetProduceList Dette er en positiv test, hvor vi forsøger at hente hele råvare tabellen ud, som en liste med `ProduceDTO`. Vi tester metoden `getProduceList()` fra vores DAO, der gerne skulle returnere en `List<ProduceDTO>`, som stemmer overens med dataen fra vores database. Listen printes ud til konsollen.

testCreateProduce Dette er en positiv test, der tester metoden `createProduce(ProduceDTO)`. Vi forsøger at oprette en tilfældig råvare i vores database, hvorefter vi henter den ud igen, for at tjekke om den er blevet oprettet i vores database.

testUpdateProduce Dette er en positiv test, hvor vi tester metoden `updateProduce(ProduceDTO)`. Vi tjekker, efter opdatering, at råvaren er ændret når vi henter den på ny.

testGetProduceOverview Dette er en positiv test, som tjekker at vi kan tilgå al vores data i vores view, `produce_overview`, gennem metoden `getProduceOverview()`, som tilsvarende `testGetProduceList` gemmer al dataen i en liste, der printes ud.



```
<terminated> MySQLProduceDAOTest [JUnit] C:\Program Files\Java\jre1.8.0_121\
testGetProduceOverview():
7      champignon      100.0
1      dej      1000.0
5      ost      200.0
6      skinke      100.0
2      tomat      1400.0

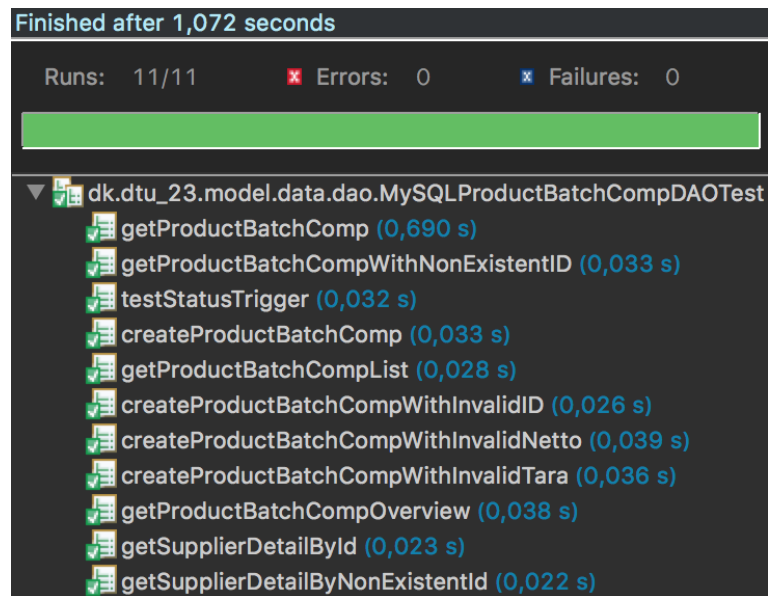
testGetProduceList():
1      dej      Wawelka
2      tomat      Knoor
3      tomat      Veaubais
4      tomat      Kims
5      ost      Ost og Skinke A/S
6      skinke      Ost og Skinke A/S
7      champignon      Igloo Frostvarer
8      smør      Kærgården
```

Figur 29: Udprint fra testGetProduceOverview() og testGetProduceList()

På Figur 29 vil der udprintet variere en smule fra den udleverede database, eftersom nogle af de tests, som køres før der bliver udprintet, opretter og opdaterer tupler i databasen. Vi får derfor ændringerne med i udprintet.

MySQLProductBatchCompDAOTest

I denne klasse tester vi de metoder der er implementeret til brug ved ProductBatchComponent-relationen. Der er skrevet 11 test cases; 6 positive & 5 negative. Vi tester 1 view, 2 procedures og 1 trigger hhv. product_batch_component_overview (view) samt create_product_batch_component & get_product_batch_component_supplier_details_by_pb_id (procedures) & after_component_insert (trigger). Derudover testes tabellen productbatchcomponent også.



Figur 30: Udprint fra MySQLProductBatchCompDAOTest

Test cases

getProductBatchComp Tester at vi kan modtage en liste af alle ProductBatchComponent'er udfra et givet ID. Vi tester med et ID vi ved eksisterer og forventer derfor at få en java List<> med indhold retur.

getProductBatchCompWithNonExistentID Tester at vi ikke kan hente ProductBatchComponent'er udfra et ikke-eksisterende ID og accepterer derudover at der smides en DALException (hvilket der gør).

createProductBatchComp Tester at vi kan opretter en ProductBatchComponent. Dette er en positiv test, så alt data vi tester med er korrekt her. Vi tjekker derudover at den tilhørende ProductBatch får opdateret sin status.

createProductBatchCompWithInvalidID Tester at der ikke kan oprettes ProductBatchComponent'er på et ProductBatch ID, der ikke eksisterer.

createProductBatchCompWithInvalidNetto Tester at der ikke kan oprettes ProductBatchComponent'er med ugyldig Netto.

testStatusTrigger Denne metode tester at status på productbatch bliver opdateret korrekt efter indsættelse af productbatchcomponent. Vi tester productbatch med id 4 og verificerer at den er i status 1 før indsættelse og status 2 efter.

createProductBatchCompWithInvalidTara Tester at der ikke kan oprettes ProductBatchComponent'er med ugyldig Tara.

getProductBatchCompList Tester at vi får en ikke-tom java List<> retur med ProductBatchCompDTO'er.

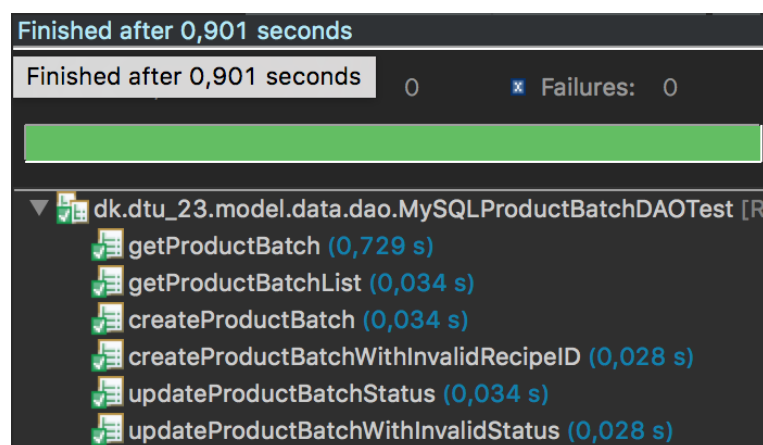
getProductBatchCompOverview Tester at vi får en ikke-tom java List<> retur med ProductBatchCompDTO'er. Dette er en positiv test.

getSupplierDetailById Tester at vi får java List med ProductBatchCompSupplierDetailsDTO retur fra datalaget. Dette er en positiv test så vi forventer også indhold i listen.

getSupplierDetailByNonExistentId Tester at vi får tom java List retur fra metoden, da vi føder den et ikke-eksisterende ID.

MySQLProductBatchDAOTest

I denne klasse testes de JDBC-metoder, der er tilknyttet ProductBatch-relationen. Der er skrevet 7 test-cases; 4 positive og 2 negative. I de negative tests testes der, at de forventede exceptions bliver smidt fra data-laget, samt at der ikke indsættes uønsket data i databasen. Vi tester 1 view og 1 procedure, hhv. product_batch_list (view) & create_product_batch_from_recipe_id (procedure). Derudover testes selvfølgelig selve tabellen productbatch. På figur 31 ses et screenshot af tests kørt umiddelbart inden aflevering af denne opgave.



Figur 31: Udprint fra MySQLProductBatchDAOTest

Test cases

getProductBatch Dette er en positiv test, der tester at der kan hentes en ProductBatch, som eksisterer i databasen.

getProductBatchList Denne positive test laver en liste og tester at der ikke returneres null fra DAO'en.

createProductBatch Denne metode forsøger at oprette en ProductBatch og verificerer efterfølgende at vi har netop én indtastning mere i databasen efterfølgende.

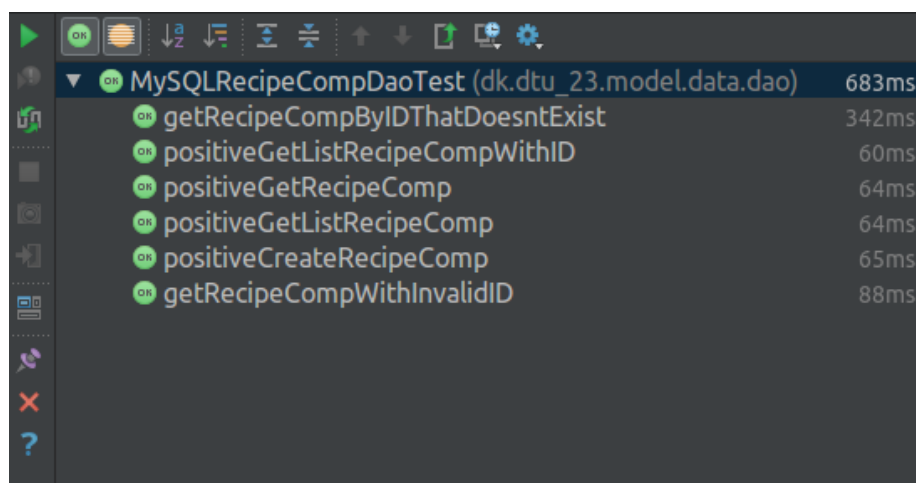
createProductBatchWithInvalidRecipeID Denne metode forsøger at oprette et ProductBatch med en ugyldig RecipeID (-1). Vi forventer en DALException og tjekker at databasen ikke har flere ProductBatches efterfølgende.

updateProductBatchStatus Denne test forsøger at opdatere status på ProductBatch med ID 1 fra 0 til 1. Metoden anvendes og der tjekkes at status i databasen er ændret.

updateProductBatchWithInvalidStatus Denne test forsøger at opdatere status ProductBatch med ID 1 fra 0 til -1 (ugyldig). Metoden anvendes og der tjekkes at status i databasen ikke er ændret.

MySQLRecipeCompDAOTest

Denne klasse tester de metoder, hvori vi opretter nye komponenter til opskrifter og henter fra databasen. Denne klasse indeholder i alt 6 test-cases. Vi har heri 4 positive og 2 negative test. Dette skyldes, at klassen indeholder 4 metoder, så vi ønsker at sikre de virker som de skal. To af metoderne fra klassen er næsten identiske, man kan enten hente RecipeCompList vha. af 'recipeID', da skal man få en liste med alle ingredienser i en enkelt recept, eller man få en liste med alle recepters ingredienser. I de negative tests, tester vi hvorledes, de forventede exceptions bliver smidt fra data-laget.



Figur 32: Screenshot fra IntelliJ, der viser resultatet af RecipeCompDAO test-klassemens kørsel.

Test cases

positiveGetRecipeComp Dette er en positive test, som tester at vores getRecipeComp metode, henter de informationer som man forventer.

getRecipeCompByIDThatDoesntExist Dette er en negative test, som tester at vi får den forventede exception, når vi prøver at hente med et ID der ikke findes i databasen.

positiveGetListRecipeWithID Denne positive test, tester hvorledes vores getRecipeCompList henter den ønskede information, denne metode skal ikke hente alt fra databasen, men kun opskrifter med det givne recipeID.

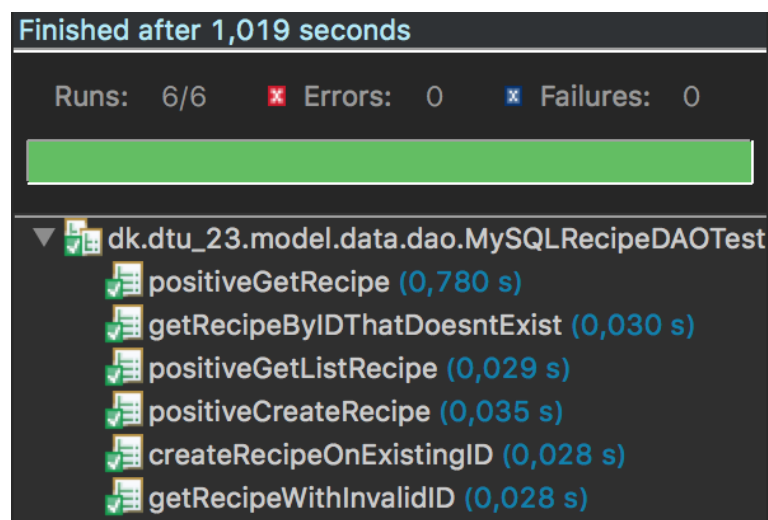
positiveGetListRecipe Denne positive test henter en RecipeCompList, men modsat den forrige metode, henter denne metode al information i fra relationen og ikke kun opskrifter med et givent ID.

positiveCreateRecipeComp Denne positive test, tester om vi succesfuldt kan indsætte en ny tuppel i relationen.

getRecipeCompWithInvalidID Denne negative test, tester om getRecipeComp metoden, smider den rigtige exception, når vi prøver at hente med et ugyldigt ID, altså et ID som aldrig ville kunne findes.

MySQLRecipeDAOTest

Denne klasse har til formål at teste metoderne i MySQLRecipeDAO. Der er ialt skrevet 6 tests, 3 positive og 3 negative. Vi har 3 metoder i klassen og vi ønsker, at de alle kører som de skal og at de ikke kan det vi ønsker de ikke skal. Vi tester 1 view og 1 procedure. Som man kan se på Figur 33 screenshot, fremgår det at alle test cases godkendes.



Figur 33: Screenshot fra Eclipse, der viser resultatet af RecipeDAO test-klassens kørsel.

Test cases

positiveGetRecipe Dette er en positive test, som tester at vores getRecipe metode, henter de informationer som man forventer.

getRecipeByIDThatDoesntExist Dette er en negative test, som tester at vi får den forventede exception, når vi prøver at hente med et ID der ikke findes i databasen.

positiveGetListRecipeWithID Denne positive test, tester hvorledes vores getRecipeList henter den ønskede information, den skal hente alle opskrifter fra tabellen.

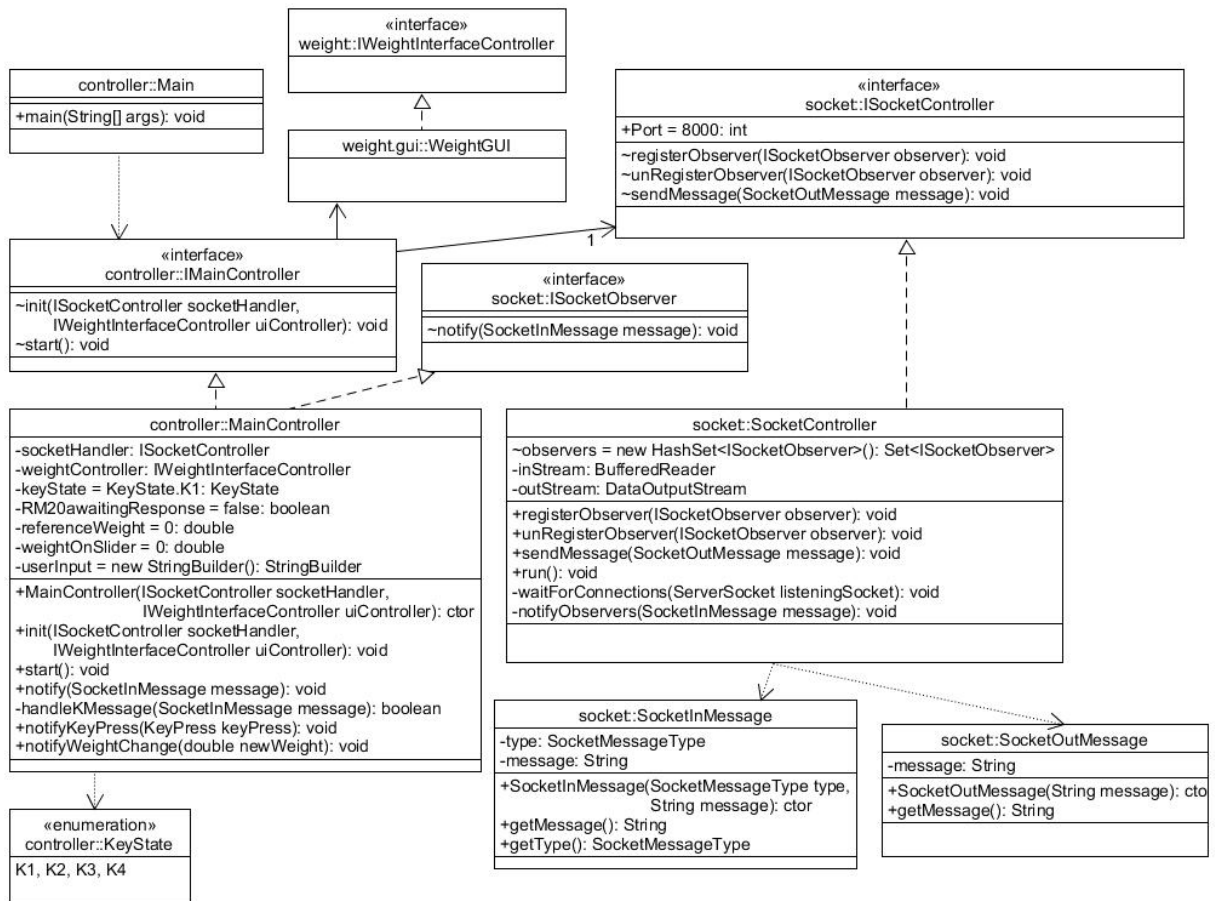
positiveCreateRecipe Denne positive test, tester hvorledes vores createRecipe metode indsætter et nyt object i viewet.

createRecipeOnExistingID Dette er en negative test til createRecipe metoden. Den prøver at oprette et nyt objekt på et ID der allerede er taget.

getRecipeWithInvalidID Denne negative test, tester om getRecipe metoden, smider den rigtige exception, når vi prøver at hente med et ugyldigt ID, altså et ID som aldrig ville kunne findes.

11.4 Vægt Simulator Dokumentation

11.4.1 Design Klassediagram



Figur 34: Design Klasse Diagram af Vægt Simulator

På Figur 34 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 34 kan det ses, hvordan klasserne er opdelt, hvilket også vises gennem pakkeopdelingen. Den samme opdeling kan tydeligt 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();
}
```

11.4.2 Kildekode

De følgende sektioner omhandler kun den implementerede del af koden.

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 flushes 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!
        outputStream.writeBytes(message.getMessage() + "\r\n");
        outputStream.flush();
    } catch (IOException e) {
        //TODO maybe tell someone that connection is closed?
        e.printStackTrace();
        notifyObservers(new SocketInMessage(SocketMessageType.Error, "No socket is connected!"));
    }
}
```

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 forvejen 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 ud fra 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 oven 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;
```

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 udførelsen 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 kommatall, 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;
```

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 `MainController` 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 display'et, bliver vægtens nuværende belastning minus `referencevægt`, 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 `MainController`. 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 resultere 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;
```

notifyWeightChange() i MainController

notifyWeightChange-metoden minder meget om notifyKeyPress-metoden, 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 blevet tareret forinden. Hvis der ikke er blevet tareret, 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;  
}
```