Sommer 2016

02324 – Videregående programmering

Gruppe nr.: 19

Afleveringsfrist: Lørdag den 17/06-2016 kl. 12:00

[CDIO FINAL]

Silas El-Azm Stryhn
s143599

Ramyar Hassani
s143242

Mikkel Hansen
s143603

Frank Thomsen
s124206

Martin Rødgaard
s143043

Denne rapport er afleveret via Campusnet

Denne rapport indeholder 57 sider inkl. denne side.

Timeregnskab

Timeregnskab **(***hele antal timer***)**					
	Silas	**Martin**	**Mikkel**	**Frank**	**Ramyar**
Analyse	5	10	2	11	10
Design	13	3	4	0	8
Implementering	41	50	69	44	42
Test	0	0	8	0	3
Dokumentation	16	19	6	3	1
I alt	**75**	**82**	**90**	**72**	**64**

Dette er et vejledende timeregnskab, der viser summen af det antal timer, hver deltager har brugt på hvert afnit i rapporten. Det fulde timeregnskab vil være at finde under bilag og kan tilgås ved at følge dette link: Timeregnskab

Initialisering og opstart

Dette er en vejledning for hvordan man henter projektet og databasen til vores projekt:

1. Projektet: For at få fat i vores projekt, kan man importere det direkte til Eclipse vha. linket https://github.com/Chaaaaap/19_final.git. Dette gøres ved at trykke File → Import → Git → Projects som Git → Clone URI → Indtaster linket i URI-feltet → Next → Finish. Når dette er gjort henter Eclipse selv hele projektet ned. Dette forudsætter at computeren har forudinstalleret GWT SDK V2.6.1. Projektet vil højst sandsynligt melde om fejl, på grund af en manglende test mappe. Programmet kan dog sagtens køre med denne fejl. Når først man har hentet projektet ned til Eclipse, henviser vi til opsætning af databasen i punkt 2, før man fortsætter. Når databasen er korrekt opsat, skal man blot højreklikke på projektet → Run As → Web Application(GWT Super Dev Mode). Når dette er gjort, compiler (behandler) Eclipse projektet, hvorefter der vises et link under fanen "Development Mode". Dette link kopieres og sættes ind i din internet browser.

2. Databasen: Vi har sat vores projekt op således, at hvis du har en mysql connector, som virker, så burde vores database fungere, da den ligger online. Vi har dog taget backup af databasen, og placeret under vores war mappe i projektet. Der er to filer, DDL grp19 og DML grp19 som skal executes. DDL filen opretter databasen og derefter alle tabellerne, hvor DML filen udfylder tabellerne med den info vi har lavet, så der kan foretages test på den. Til sidst skal der ændres i vores Connector klasse som ligger i pakken code.Connector. Det skal se ud som på dette billede herunder, dog med ændret password, hvis du benytter et.

```
public class Connector {
    private final String HOST     = "localhost";
    private final int    PORT     = 3306;
    private final String DATABASE = "19_CDIOFinal";
    private final String USERNAME = "root";
    private final String PASSWORD = "";
    private Connection connection;
```

3. Vores system har altid en Sys-Admin, som er vores administrator:

   Opr_id : 1

   Password : Test1234

4. For at teste vores system, kan man navigere rundt på vores hjemmeside efter at være logget ind som Sys-Admin, se de forskellige råvare eller oprette nogle nye. Det samme kan gøres med recepter, råvarebatches, brugere og produktbatches.
5. Afvej kan kun testes hvis det er vægt 1 der er tilsluttet, ellers skal der laves om i koden så IP'en bliver ændret til den korrekte.

Dette er et udtag fra vores DML fil, der opretter de forskellige komponenter i databasen. Her kan man se, hvilke komponenter databasen indeholder når den er blevet oprettet.

```
use 19_CDIOFinal;

INSERT INTO operatoer(opr_id, opr_navn, ini, cpr, password, aktiv, type) VALUES
(1, 'Sys-Admin', 'SA', '0000000000', 'Test1234', '1', 'administrator'),
(2, 'Hans Hansen', 'HH', '1234564321', 'Test1234', '1','farmaceut');
INSERT INTO raavare(raavare_id, raavare_navn, leverandoer) VALUES
(1, 'Vand', 'Vand A/S'),
(2, 'Salt', 'Salt og Peber A/S'),
(3, 'Citron', 'Sydens Sol');
INSERT INTO raavarebatch(rb_id, raavare_id, maengde) VALUES
(1, 1, 1000),
(2, 2, 750),
(3, 3, 500);
INSERT INTO recept(recept_id, recept_navn) VALUES
(1, 'Saltvand med Citron');
INSERT INTO receptkomponent(recept_id, raavare_id, nom_netto, tolerance) VALUES
(1, 1, 5.500, 0.3),
(1, 2, 0.330, 2.4),
(1, 3, 0.100, 1.8);
INSERT INTO produktbatch(pb_id, recept_id, status, dato) VALUES
(1, 1, 0, '2016-06-17');
```

Indhold

Resumé

Vores rapport indeholder et analyseafsnit, hvor vi analyserer kravene for opgaven og laver nogle diagrammer for forståelsen af hvordan vi har tænkt os at lave dette system. Vi viser hvordan vores system fungerer i virkeligheden med en domænemodel. I dette afsnit bliver alle vores Use Cases fra use case diagrammet også beskrevet. I vores Design afsnit viser vi hvordan vores klasser hænger sammen i projektet i et klassediagram, og viser hvordan man kan bruge systemet i et Design-sekvensdiagram. I Database afsnittet, bliver alt omkring vores database analyseret og vist ved hjælp af diagrammer, såsom E/R og Skema diagrammer, som viser hvordan vores database er stykket sammen. Vi har også lavet lidt ændringer i databasen og kommenterer også dette i afsnittet. Til sidst i dette afsnit beskriver vi de forskellige DTO, DAO og Service klasser, som laver databasekald, eller bliver brugt i forbindelse med dem.
Derefter kommer vores implementerings afsnit, hvor vi kommer ind på hvordan vores brugergrænseflade og ASE er lavet. Vi kommer her også ind på, hvad vi mangler i vores implementering, og skriver kort om hvorfor vi ikke har nået det. Så kommer vores test afsnit, som tester alle vores DAO klasser med JUnit test.

Indledning

En medicinalvirksomhed ønsker at få implementeret et softwaresystem, som skal bruges til afvejning. I systemet skal det være muligt at definere råvarer, som hjemtages fra forskellige leverandører til firmaets lager i såkaldte råvarebatches. Hver enkelt råvarebatch består af en mængde af den pågældende råvare.
Det skal også være muligt at oprette recepter. En recept består af en liste af råvarer hver med et bestemt kvantum.
Hver recept skal bruges til produktion af et vilkårligt antal produktbatches (konkrete produkter). De enkelte produktbatches indeholder information om hvilken recept som produktbatchen er oprettet ud fra, samt hvilke råvarebatches de anvendte råvarer er udtaget fra.
I takt med at dele af råvarebatches bruges, holder systemet styr på den aktuelle mængde i alle råvarebatches (lagerstyring).
Systemet skal bestå af 4 slags aktører:

- En Administrator, som er superbrugeres og har alle rettigheder, som de efterfølgende aktører har. Herudover kan han også oprette, redigere og deaktivere andre aktører i systemet.
- En Farmaceut, som kan administrere råvarer og recepter og som også har en værkførers rettigheder.
- En Værkfører, som kan administrere råvarerbatches og produktbatches og har også rettighederne som en operatør.
- En Operatør, som kan foretage en afvejning.

Rapportens basale formål er at give en grundlæggende forståelse for tankegangen igennem forløbet, hvilke overvejelser der er gjort og hvordan vores implementation er stykket sammen. Rapporten gennemgår således en analyse af kravene og en udarbejdelse af diverse diagrammer, som skal give indblik i systemets opbygning.

Dokumentation af projektplan

I vores 3-ugers forløb med projektet, har vi lavet en projektplan, så vi hver især, har haft ansvaret for et område af projektet. Vi har delt det op i nogle hovedområder, som vi så har fordelt og haft ansvaret for. Disse hovedområder udgør:
Analysering af projektet
Database (MySql)
Database Access lag (DAO, DTO og Service klasser)
ASE
Brugergrænseflade (GWT)
Diagrammer
Rapport
Andet
Disse hovedområder har vi prøvet at uddele så ligeligt som muligt, så alle gruppemedlemmer både får ansvar for at analysere opgaven, lave en form for implementering af en vis slags, hjælp med diagrammer og så rapport. Vi tænkte, at hvis alle har et hovedområde inden for implementeringen, så ville alle have lavet lidt af det, og derfor nemmere kunne forstå det andet kode, da det meste af systemet er lavet på samme måde. Tanken med at alle skulle være med til at analysere, var at alle fik en grundlæggende forståelse for, hvad opgavens omfang er, og hvilke dele som vi har valgt at nedprioritere frem for andre. Diagrammer fordelte vi ligeligt imellem os, så alle fik lidt at lave, så alle igen fik en vis forståelse for, hvad vores system kan og hvordan det ser ud. Til sidst er alle jo med til at skrive rapporten, da det er ting som dette der giver en god forståelse for hvad de andre i gruppen har implementeret og lavet, da man bliver tvunget til at analysere de andres arbejde for selv at kunne dokumentere det i rapporten.

Hovedafsnit

Analyse

I dette afsnit har vi analyseret opgaven, inden vi går i gang med implementeringen, for at få et overblik over hvilke krav der er til opgaven og vise vores tanker og overvejelser. Vi laver her en kravspecifikation, hvor vi viser hvor meget vægt vi lægger på diverse krav. I dette afsnit laver vi også et Use Case diagram, samt dertil hørende Use Case beskrivelser. Vi har også lavet en domænemodel, som viser hvordan vi har tænkt os at vores system skal fungere og snakke sammen. Et andet diagram i dette afsnit er vores BCE model, som viser hvilke af vores klasser der er Boundaries, Controllere og Entities. Til sidst i vores analyseafsnit har vi lavet et lille system sekvens diagram som viser kommunikationen mellem en bruger og systemet når en råvare skal oprettes.

Kravspecifikation

Vi skal i dette projekt udvikle et softwaresystem til en medicinalvirksomhed. System skal bruges til afvejning samt dokumentation af råvare afvejning og lagerstyring.
Ud fra opgaveformuleringen har vi udarbejdet en kravspecifikation. Derefter har vi inddelt hver enkelt krav i funktionelle og ikke funktionelle krav for derefter at bruge MoSCoW metoden på dem og til sidst kommentere vores valg.

Applikationen skal være brugervenlig.	Ikke funktionelt	Should have	Vi synes dette krav er vigtigt fordi hvis kunden ikke kan finde ud af at benytte applikationen er funktionaliteten irrelevant.
Web applikationen skal udvikles vha. GWT.	Ikke funktionelt	Should have	Vi synes dette krav er vigtigt, da det er noget som kunden ønsker. Dog har det ingen indflydelse på funktionaliteten.
Adgang til applikationen skal foregå gennem et password.	Ikke funktionelt	Should have	Dette krav er vigtigt, fordi så kan kunden let administrere hvem som skal have adgang.
Der skal være 4 aktører(administrator, farmaceut, værkfører og operatør).	Ikke funktionelt	Should have	De 4 forskellige typer aktører er vigtige at få med så kunden kan styre brugernes permission level.
Administratoren skal kunne oprette, rette og vise brugere.	Funktionelt	Must have	Dette krav er vigtigt fordi det tillader kunden efterfølgende at administrere brugerne.
En bruger må ikke kunne blive slettet.	Ikke funktionelt	Could have	Det er generelt en god ide ikke at slette ting fra en database da det kan påvirke andre ting i databasen. Vi vil i stedet

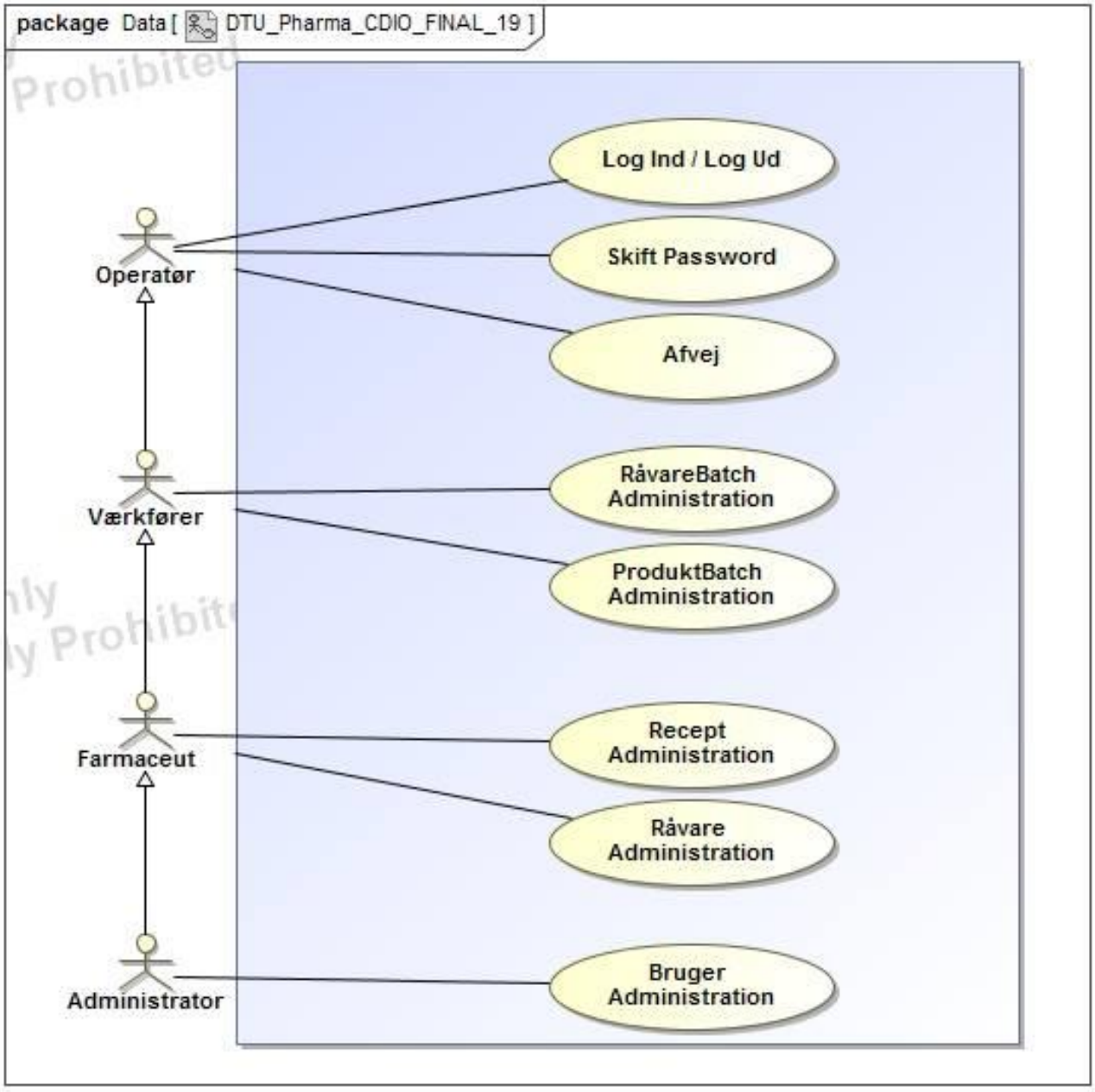
			implementere en aktiver/ deaktiver funktion på brugeren.
En bruger skal bestå af et unikt bruger ID, navn, initialer, password og brugerens rolle.	Funktionelt	Must have	Det som er vigtig her er at vi har nogle brugere. Det at de har et navn og initialer er måske ikke så vigtig men det kan være med til at gøre det mere brugervenligt.
Farmaceuten skal kunne oprette, rette og vise råvarer i systemet.	Funktionelt	Must have	Det er vigtigt at der kan blive oprettet råvarer, da det er en del af det som applikationen skal bruges til.
En råvarer skal bestå af et unikt råvare nummer, navn og leverandør.	Ikke funktionelt	Should have	Vi synes dette er fine og vigtige parametre på en råvarer og derfor synes vi det er vigtigt at få det hele med.
Farmaceuten skal kunne oprette og vise recepter i systemet.	Funktionelt	Must have	Det er vigtigt at der kan blive oprettet recepter, da det er en del af det som applikationen skal bruges til.
En recept defineres ved et unikt recept nummer, navn og en sekvens af recept komponenter.	Ikke funktionelt	Should have	Vi synes dette er fine og vigtige parametre på en recept og derfor synes vi det er vigtigt at få det hele med.
En recept komponent består af en råvare, en mængde samt tolerance.	Ikke funktionelt	Should have	Vi synes dette er fine variabler til recept komponenten.
Værkføreren skal kunne oprette og vise råvare batches i systemet.	Funktionelt	Must have	Raavare batches er som sådan ikke funktionelle da det kun holder styr på hvor meget vi har af de forskellige råvarer. Men dette synes vi er vigtigt at kunden kan og har derfor besluttet at for deres funktionalitet er det vigtigt.
En råvare batch defineres ved et råvare batchnummer raavare id og en mængde.	Ikke funktionelt	Should have	vi synes at råvare batches består af få, men vigtige variabler.
Værkføreren skal kunne oprette og vise produkt batches i systemet.	Funktionelt	Must have	Da produkt batches er en slags to-do synes vi det er vigtigt at brugeren kan oprette disse.
En produkt batch defineres ved et produkt batchnummer, nummeret på den recept produkt batch skal produceres ud fra, dato for oprettelse, samt status om batchen(oprettet, under produktion, afsluttet).	Ikke funktionelt	Could have	Vi synes dette er gode parameter til en produkt batch.
Afvejnings resultater for de enkelte recept komponenter skal gemmes som en produkt batch komponent i produkt batchen i takt med at produktet fremstilles.	Ikke funktionelt	Must have	Vi synes dette er en af de vigtigste ting at have med. Da vi antager at afvejningerne er vigtige for kunden.
Når værkføreren har oprettet en ny produkt batch skal denne kunne printes.	Ikke funktionelt	Wont have	Dette synes vi er så lille en ting at skulle have en funktion til og vil derfor

			blive prioriteret meget lavt.
Operatøren skal have adgang til veje terminalen.	Funktionelt	Must have	Da operatørens arbejde foregår ved veje terminalen er dette selvfølgelig meget vigtigt.
Administratoren have adgang til alle de andre aktørers funktioner.	Ikke funktionelt	Should have	Generelt er det meget almindeligt at administratorer har adgang til alt og derfor vil dette også blive tilføjet.
Farmaceuten skal have adgang til værkførerens og operatørens funktioner.	ikke funktionelt	Should have	Dette krav er ikke et funktionelt krav. Men vi vil tilføje det da det er noget kunden ønsker.
Værkføreren skal have adgang til operatørens funktioner.	ikke funktionelt	Should have	Dette krav er ikke et funktionelt krav. Men vi vil tilføje det da det er noget kunden ønsker.
Operatøren skal identificere sig overfor veje terminalen før brug. Ved at indtaste sit operatør nummer.	Ikke funktionelt	Should have	Dette krav er vigtigt at have med så kunden kan holde styr på hvem som laver hvilke afvejelser.
Brugerens navn skal vises på veje terminalens display.	Ikke funktionelt	Could have	Kravet er kun en lille visuel ting som tilføjer
Brugeren skal kunne indtaste den ønskede recept komponent og herefter skal han kunne afveje de forskellige råvarer.	Ikke funktionelt	Should have	Dette er vigtigt så kunden kan registrere hvilke recept komponenter som bliver afvejet.
Undervejs under afvejningen skal statussen ændres fra oprettet til under produktion og til sidst afsluttet.	Ikke funktionelt	Should have	Vi synes ikke dette krav er vigtigt, men det giver mulighed for andre at følge med i de forskellige opgaver.
Når en produkt batch komponent bliver afvejet skal den afvejede mængde trækkes fra i databasen.	Ikke funktionelt	Should have	Kravet gør at kunden kan følge med i hvor meget der bliver brugt af de forskellige råvare og det synes vi er vigtigt.

Use Case Diagram

Vores Use case diagram giver os et overblik over hvad vores tanker er omkring designet af vores system. Diagrammet viser os overordnet hvad systemet indeholder og hvilke muligheder der er. Som man kan se, indeholder vores system 4 aktører.
En "Operatør", som kan lave en afvejning og skifte sit password.

En "Værkfører", som har de samme rettigheder som en operatør, men som også kan oprette og vise råvare batches og produkt batches.
En "Farmaceut", som har de samme rettigheder som en værkfører, men som også har rettigheder til at oprette og vise recepter, samt oprette, redigere og vise råvarer.
En "Administrator", som har de samme rettigheder som en farmaceut, men som også kan oprette, redigere, deaktivere/aktivere og vise brugere. I vores diagram har vi vist at en administrator kan bruge alle use cases ved at han arver fra en farmaceut. Og det samme med en farmaceut, der viser vi at denne aktør har alle de rettigheder, som en værkfører har, ved igen at lave en arvepil op til værkføreren. Det samme sker fra værkføreren op til en operatør, som er den eneste aktør, som ikke arver rettigheder fra nogen andre aktører.


package Data [DTU_Pharma_CDIO_FINAL_19]
Log Ind / Log Ud
Skift Password
Afvej
RåvareBatch Administration
ProduktBatch Administration
Recept Administration
Råvare Administration
Bruger Administration
Operatør
Værkfører
Farmaceut
Administrator

Use-case beskrivelser

Herunder har vi lavet vores Use Case beskrivelser. Som man kan se, har vi fra vores use case diagram udeladt use casen "Log ind/Log ud", da den ikke gør andet end at logge en aktør ind eller ud. Så den blev ikke spændende nok til, at der kunne laves en Use Case beskrivelse omkring den. Alle andre er blevet beskrevet. Jeg har taget vores Afvej med her, selvom den beskrivelse her, er meget kort. Det er den fordi der kun kan trykkes på afvej, så den starter en connection op med en veje terminal. Når dette er gjort, har vi implementeret en ASE, som står for at sende kommandoer til vægten, men det er de samme kommandoer, da vi har programmeret den til at være efter en rækkefølge. Rækkefølgen kan ses under vores ASE afsnit i implementation.

Use Case Name:	Råvare Administration
Use Case ID:	2
Actors:	Administrator Farmaceut
Description:	Aktørerne kan oprette, redigere og vise råvarer
Preconditions:	Man er logget ind som en af aktørerne fra ID 2
Postconditions:	Råvare er blevet oprettet, redigeret eller vist

Main flow:	1. Opret Råvare:
	    1. Indtast råvare nr.
	    2. Indtast råvare navn
	    3. Angiv leverandør
	    4. OK
	2. Vælg råvare:
	    1. Rediger råvare
	        1. Gem/Annuller
	3. Vis råvarer:
	    1. Råvareliste bliver vist

Alternative Flow:	1. Log ud

Use Case Name:	Bruger Administration
Use Case ID:	1
Actors:	Administrator
Description:	En administrator kan administrere andre brugere i systemet
Preconditions:	Man er logget ind som Administrator
Postconditions:	Aktør/Aktører er blevet oprettet, redigeret eller vist

Main flow:	1. Opret bruger:
	    1. Indtast operatør ID
	    2. Indtast brugernavn
	    3. Indtast initialer (valgfri)
	    4. Indtast CPR-nummer
	    5. Indtast password
	    6. Gentag password
	    7. Vælg rolle
	        1. Operatør/Værkfører/Farmaceut/Administrator
	    8. OK
	2. Rediger bruger:
	    1. Vælg bruger
	        1. Rediger bruger
	            1. Gem bruger/Annuller
	3. Deaktiver bruger:
	    1. Vælg bruger
	    2. Tryk Deaktiver
	        1. OK/Annuller

Alternative Flow:	1. Log ud

Use Case Name:	Recept Administration
Use Case ID:	3
Actors:	Administrator Farmaceut
Description:	Aktørerne kan oprette og vise recepter
Preconditions:	Man er logget ind som en af aktørerne fra ID 3
Postconditions:	Recept er blevet oprettet eller vist.

Main flow:	1. Opret recept:
	    1. Indtast recept nr.
	    2. Indtast recept navn
	    3. Vælg råvare “X” (Fra 1-5 råvarer påkrævet)
	        1. Indtast mængde
	        2. Indtast tolerance
	2. Vis recept:
	    1. Recept liste bliver vist

Alternative Flow:	1. Log ud

Use Case Name:	Råvarebatch Administration
Use Case ID:	4
Actors:	Administrator Farmaceut Værkfører
Description:	Aktørerne kan oprette og vise råvarer batches
Preconditions:	Man er logget ind som en af aktørerne fra ID 4
Postconditions:	Råvarebatch er blevet oprettet eller vist

Main flow:	1. Opret Råvarebatch:
	    1. Indtast råvarebatch nr.
	    2. Indtast råvare nr.
	    3. Angiv mængde
	    4. OK
	2. Vis råvarer:
	    1. Råvarebatch liste bliver vist

Alternative Flow:	1. Log ud

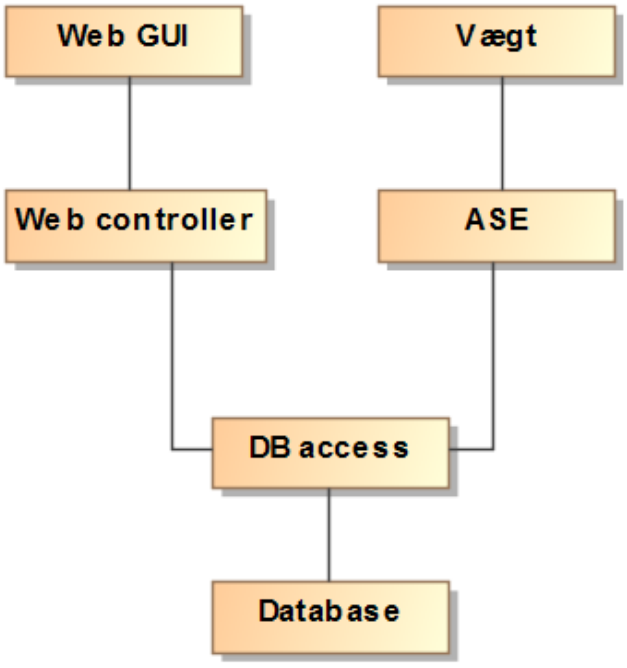
Use Case Name:	Produktbatch Administration
Use Case ID:	5
Actors:	Administrator Farmaceut Værkfører
Description:	Aktørerne kan oprette og vise produkt batches
Preconditions:	Man er logget ind som en af aktørerne fra ID 5
Postconditions:	Produktbatch er blevet oprettet eller vist
Main flow:	3. Opret produktbatch: 1. Indtast produktbatch ID 2. Indtast recept ID 3. Indtast dato (Er indtastet som dags dato) 4. OK 4. Vis produktbatches: 1. Produktbatch liste bliver vist
Alternative Flow:	1. Log ud

Use Case Name:	Afvej
Use Case ID:	6
Actors:	Administrator Farmaceut Værkfører Operatør
Description:	Lav en afvejning
Preconditions:	Man er logget ind som en af aktørerne fra ID 6
Postconditions:	Der er foretaget en afvejning
Main flow:	1. En aktør starter en afvejning
Alternative Flow:	1. Log ud

Use Case Name:	Skift Password
Use Case ID:	7
Actors:	Administrator Farmaceut Værkfører Operatør
Description:	En aktør skifter password
Preconditions:	Man er logget ind som en af aktørerne fra ID 7
Postconditions:	Password for aktøren er ændret
Main flow:	1. Indtast gamle password 2. Indtast nye password 3. Gentag nye password
Alternative Flow:	1. Log ud

Domænemodel og BCE model

Dette er vores domæne model, som viser hvordan vi har tænkt os at vores system skal fungere. Som man kan se, skal vi have implementeret en database, som vores system kan hente, redigere og gemme informationer i. Der skal også være et lag (DB access), som henter informationerne fra databasen, da det er mere hensigtsmæssigt, end at vi henter informationerne direkte fra databasen. Herfra har vi så to stier, da den ene skal kunne opsætte en hjemmeside, ved hjælp af GWT som omdanner java kode til et javascript, som så kan bruges som brugergrænseflade. Den anden sti laver vi en ASE(Afvejnings Styrings Enhed), som står for kommunikationen med vægten, som enten skal laves som en simulator ellers skal der benyttes en fysisk vægt. Vores ASE kommer så til at stå for at sende kommandoer til vores vægt og læse svarene fra denne, samt lave databasekald ud fra hvad svarene fra vægten er. Alle de databasekald, som vores ASE laver igennem DB access laget, bliver så gemt i databasen og derefter hentet op i vores GUI brugergrænseflade.



Dette er vores BCE model, som viser et udklip af vores program. Diagrammet viser vores boundaries, controllere og entities på henholdsvis server siden og client siden.

Boundary:

En boundary er et objekt, som snakker med andre system aktører, såsom GUI, vægt og database i vores tilfælde. I vores tilfælde på billedet er Login, MainMenu, OpretBruger og Connector Boundary objekter, da de alle sammen snakker med andre aktører i vores system. Hertil skal det nævnes, at de stiplede kasser også repræsentere boundries i den forstand, at de opfører sig som boundries mellem client og server.

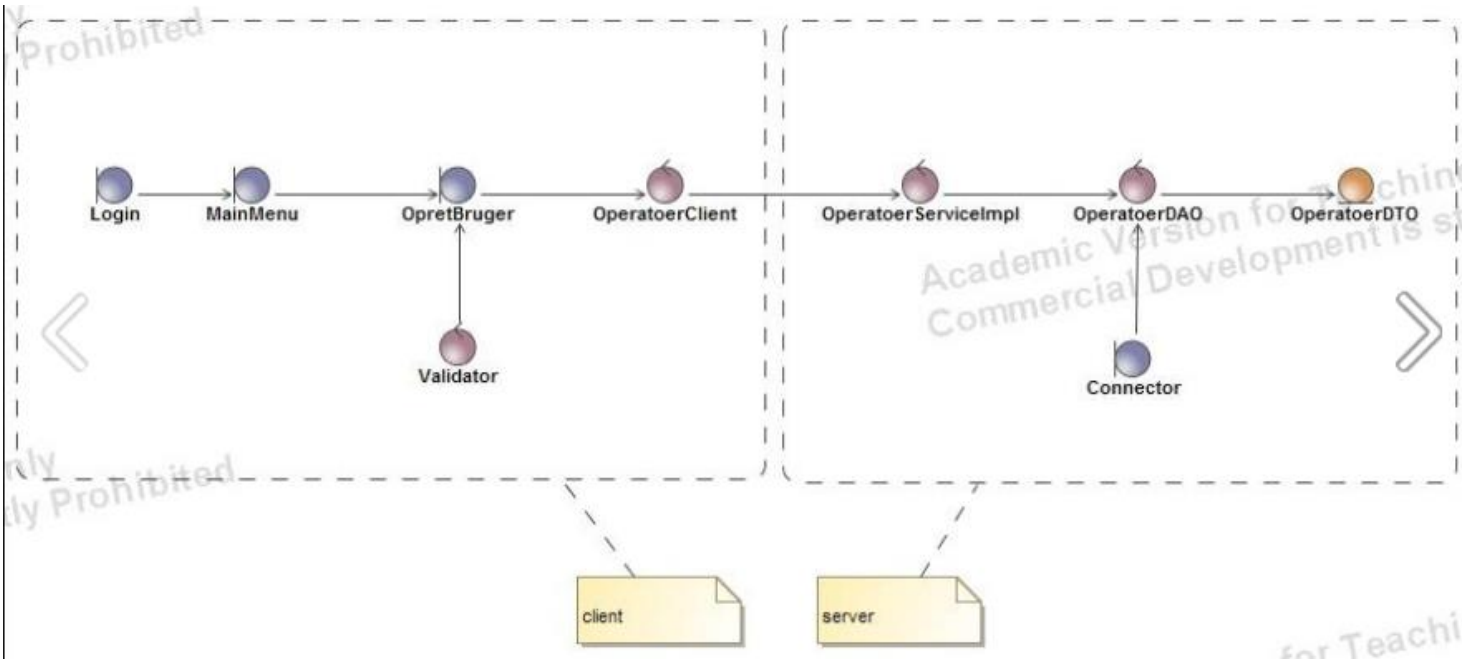
Controller:

En Controller er et objekt, som får entities til at kommunikere med Boundaries. De styrer hvilke metoder der skal køres, med de info fra entities, i Boundarierne. I vores eksempel er OperatoerClient, Validator, OperatoerServiceImpl og OperatoerDAO vores Controllere, da de styrer flowet mellem vores entity og boundaries.

Entity:

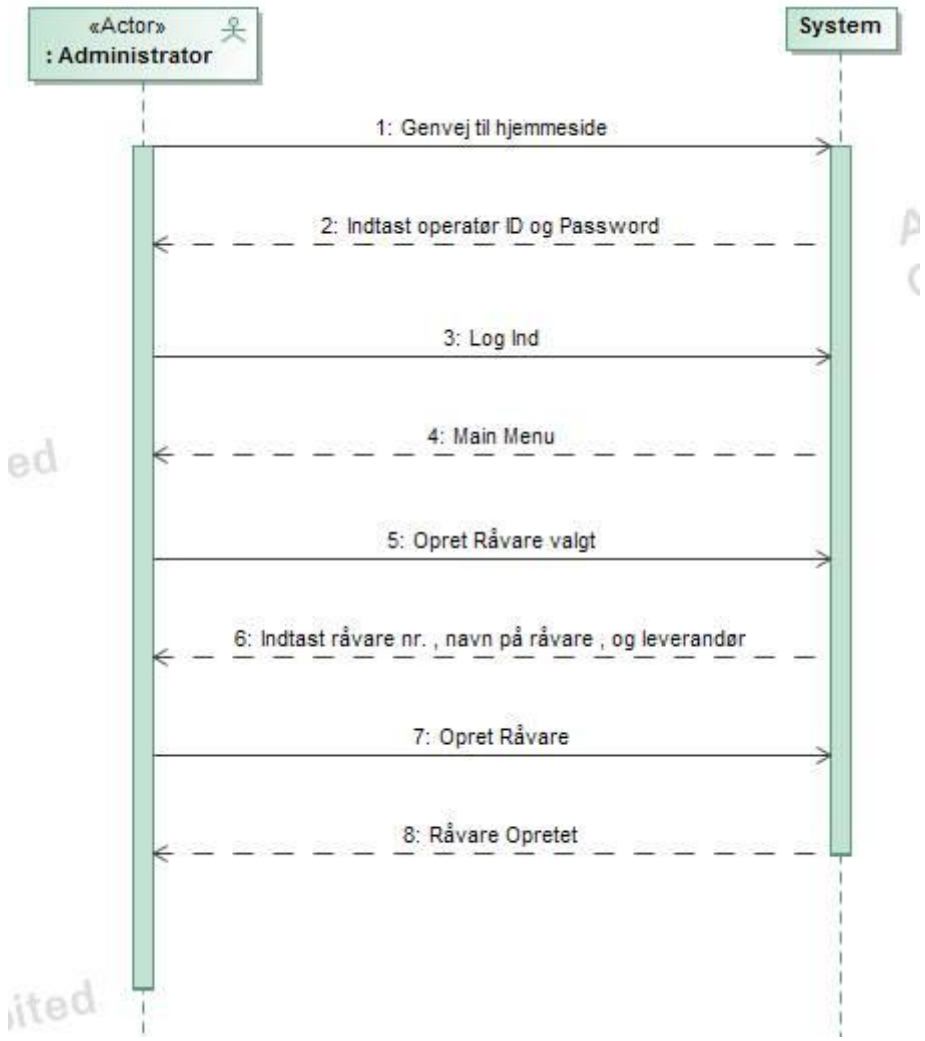
En Entity er et objekt, som repræsentere data. I vores eksempel er OperatoerDTO en entity da det er klassen som indeholder alt information om en operatør og ikke indeholder noget logik, men blot getters.

Vi har valgt kun at lave dette eksempel, selvom vi har mange flere klasser, fordi de andre klasser hænger sammen på nogenlunde samme måde mellem server side og client side, og derfor bare ville være det samme diagram bare med andre navne.



System-sekvensdiagram

Dette diagram har vi lavet, for at vise hvordan vi har tænkt os at interaktionen mellem systemet og en aktør, som i dette tilfælde er en administrator, skal fungere i vores system. Det der sker, er at systemet genererer et link til hjemmesiden, og når denne er åbnet i en browser bliver aktøren bedt om at indtaste et operatør id og password, hvor aktøren så gør det, og bliver logget ind, hvis kodeordet stemmer overens med operatør id'et. Så viser systemet main menu, som har en masse forskellige knapper og drop down menuer, som kan meget af det samme, så i dette tilfælde har vi bare valgt at vise "Opret råvare". Man kan se at vi vælger "Opret råvare" og så svarer systemet tilbage, at der skal indtastes råvare nr, råvare navn og leverandør. Det gør aktøren, og så er råvaren oprettet, og det svarer systemet så tilbage med at den er.
Som sagt har vi bare valgt at lave et diagram til at vise hvordan vores system fungere, da hele vores brugergrænseflade kommer til at fungere på samme måde som dette, og langt hen af vejen er implementeret på samme måde, kun med små ændringer.

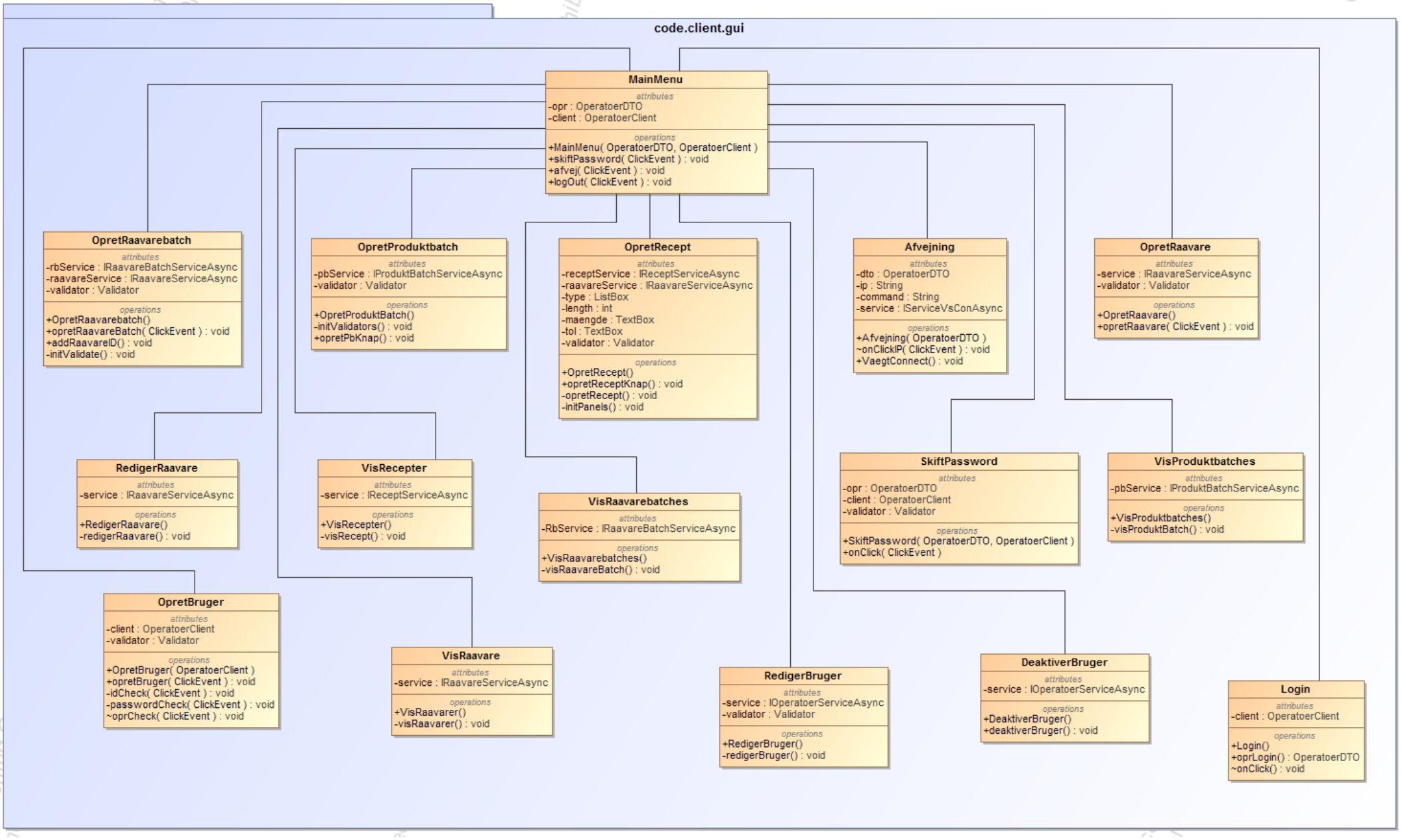


Design

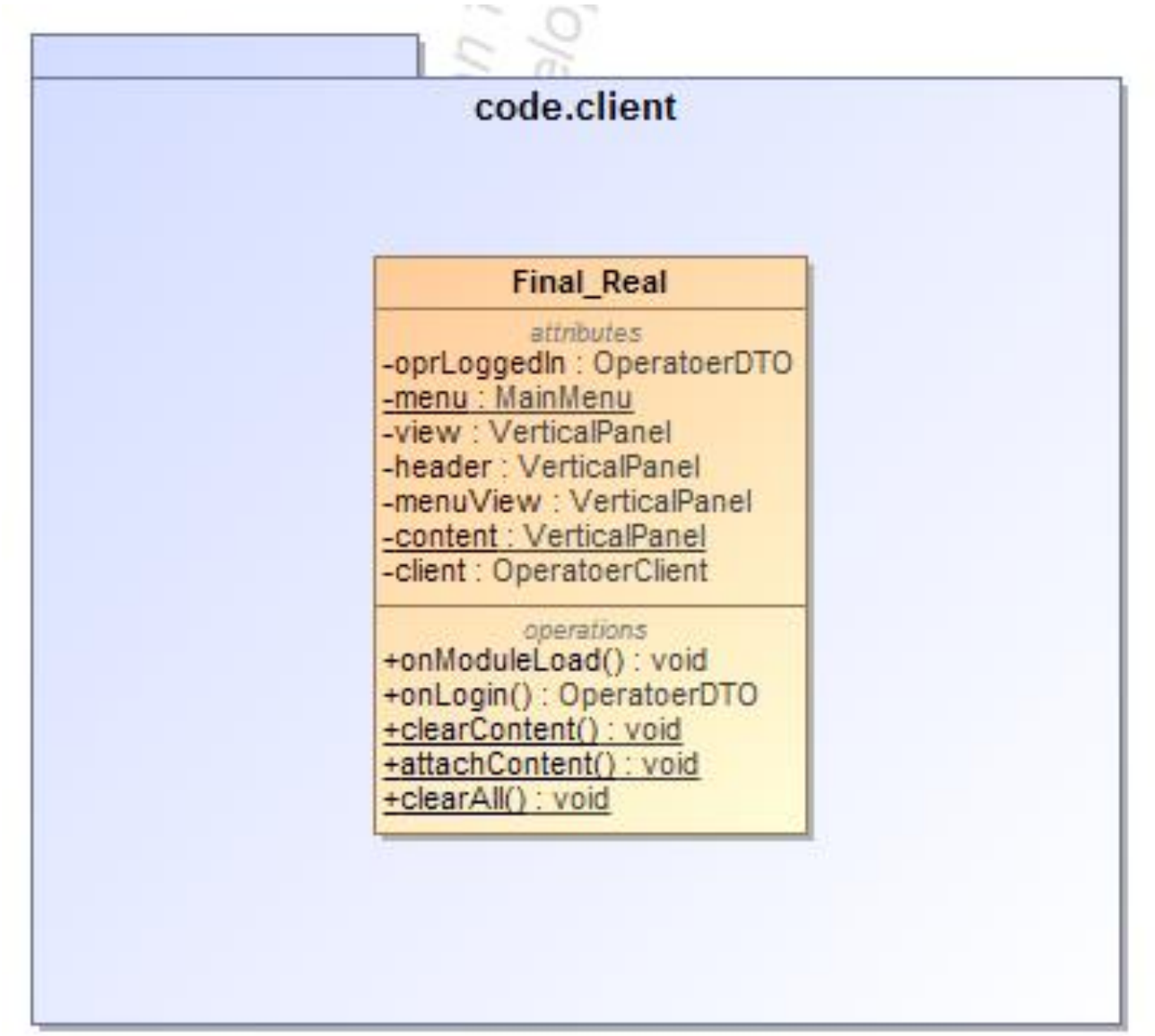
I dette afsnit viser vi hvordan designet for vores system ser ud ved brug af klassediagrammer og design sekvensdiagrammer. Meningen ved at have dette afsnit er at man kan se hvordan vores projekt hænger sammen med hensyn til klasserne. Til at vise hvordan vores system virker, så har vi lavet et design sekvensdiagram, som viser hvordan der bliver kommunikeret i mellem vores klasser, og hvilke metoder der bliver brugt. Klassediagrammerne står dernæst for visualiseringen af systemets opbygning med alle klasser og dertilhørende metoder og attributter.

Design-klassediagram

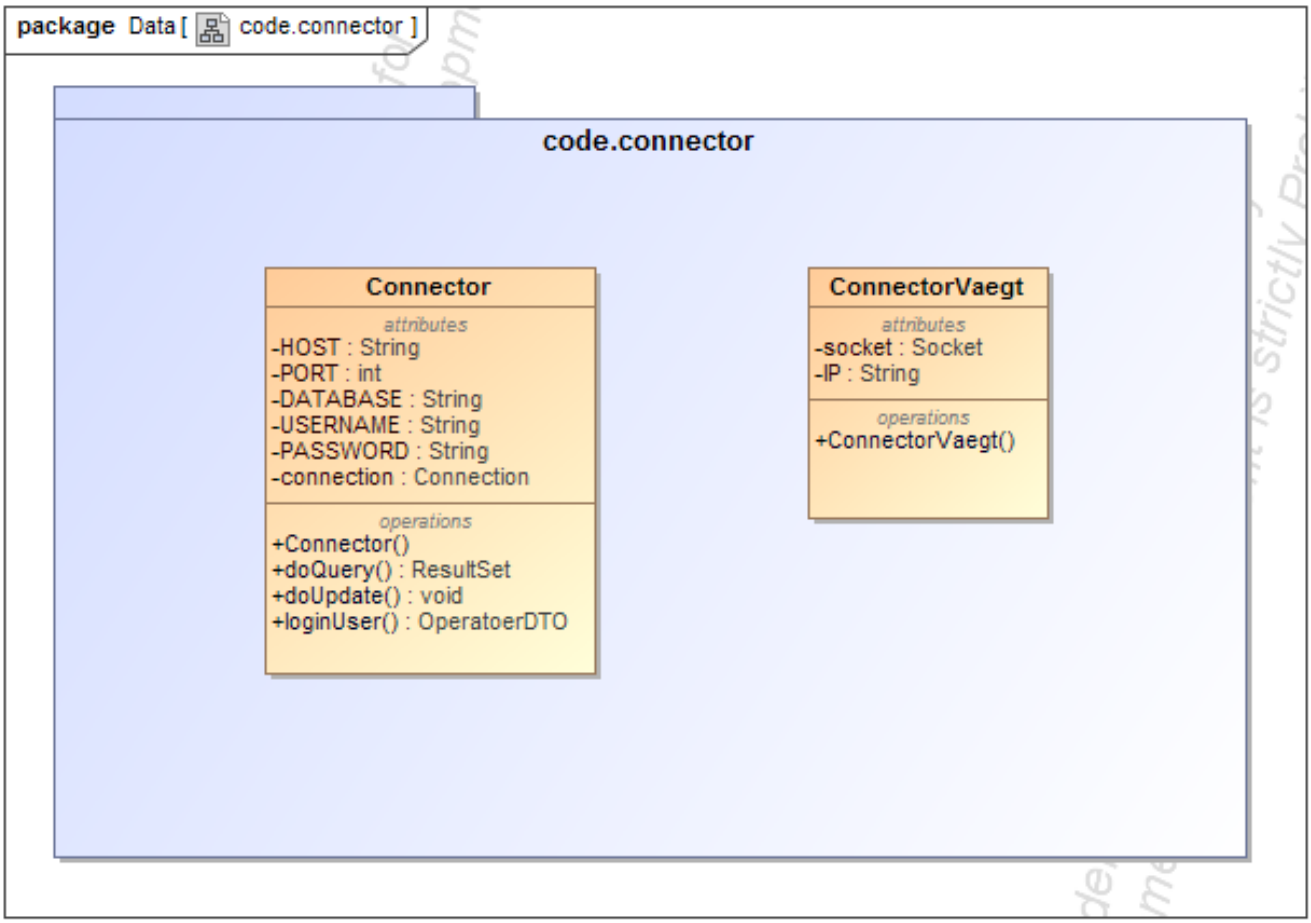
Det vil fremgå af de forskellige pakker og klasser, hvilke andre klasser der er forbindelse til i de følgende diagrammer. Dvs klasser fra andre pakker, som der ikke umiddelbart kan vises en visuel forbindelse til, grundet at de ligger i andre pakke. . Disse er angivet som en attribut af den type, som vi har forbindelse til. Dette vil gøre sig gældende for alle klasse-diagrammerne i denne rapport.
Vores klassediagram er delt op i de pakker vi har i projektet. Dette er gjort med henblik på en mere overskuelig fremvisning af sammenhængen mellem klasserne og for at gøre det lettere at se, hvordan vi har valgt at dele systemet op. Pakkerne er blevet tildelt relevante navne, der er med til at give en god indikation om, hvor de forskellige klasser høre ind under. Forklaring af diagrammerne fortsætter nedenfor.
Nedenstående diagram viser de forskellige klasser som udgør vores GUI, dvs selve bruger interfacet. Denne pakke står for alt de visuelle, vi ser i vores web-applikation. Klasserne er blevet tildelt relevante navne, som stort set hver især står for de forskellige menuer på hjemmesiden. I klasserne ser vi, hvilke metoder de indeholder, der giver en indikation af, hvad der sker i dem. Som det fremgår af diagrammet, er det MainMenu der står for initialiseringen af de forskellige menuer, da denne har forbindelse til dem alle.


code.client.gui
MainMenu
attributes
-opr : OperatoerDTO
-client : OperatoerClient
operations
+MainMenu(OperatoerDTO, OperatoerClient)
+skiftPassword(ClickEvent) : void
+afvej(ClickEvent) : void
+logOut(ClickEvent) : void
OpretRaavarebatch
attributes
-rbService : IRaavareBatchServiceAsync
-raavareService : IRaavareServiceAsync
-validator : Validator
operations
+OpretRaavarebatch()
+opretRaavareBatch(ClickEvent) : void
+addRaavareID() : void
-initValidate() : void
OpretProduktbatch
attributes
-pbService : IProduktBatchServiceAsync
-validator : Validator
operations
+OpretProduktBatch()
-initValidators() : void
+opretPbKnap() : void
OpretRecept
attributes
-receptService : IReceptServiceAsync
-raavareService : IRaavareServiceAsync
-type : ListBox
-length : int
-maengde : TextBox
-tol : TextBox
-validator : Validator
operations
+OpretRecept()
+opretReceptKnap() : void
-opretRecept() : void
-initPanels() : void
Afvejning
attributes
-dto : OperatoerDTO
-ip : String
-command : String
-service : IServiceVsConAsync
operations
+Afvejning(OperatoerDTO)
~onClickIP(ClickEvent) : void
+VaegtConnect() : void
OpretRaavare
attributes
-service : IRaavareServiceAsync
-validator : Validator
operations
+OpretRaavare()
+opretRaavare(ClickEvent) : void
RedigerRaavare
attributes
-service : IRaavareServiceAsync
operations
+RedigerRaavare()
-redigerRaavare() : void
VisRecepter
attributes
-service : IReceptServiceAsync
operations
+VisRecepter()
-visRecept() : void
VisRaavarebatches
attributes
-RbService : IRaavareBatchServiceAsync
operations
+VisRaavarebatches()
-visRaavareBatch() : void
SkiftPassword
attributes
-opr : OperatoerDTO
-client : OperatoerClient
-validator : Validator
operations
+SkiftPassword(OperatoerDTO, OperatoerClient)
+onClick(ClickEvent)
VisProduktbatches
attributes
-pbService : IProduktBatchServiceAsync
operations
+VisProduktbatches()
-visProduktBatch() : void
OpretBruger
attributes
-client : OperatoerClient
-validator : Validator
operations
+OpretBruger(OperatoerClient)
+opretBruger(ClickEvent) : void
-idCheck(ClickEvent) : void
-passwordCheck(ClickEvent) : void
~oprCheck(ClickEvent) : void
VisRaavare
attributes
-service : IRaavareServiceAsync
operations
+VisRaavarer()
-visRaavarer() : void
RedigerBruger
attributes
-service : IOperatoerServiceAsync
-validator : Validator
operations
+RedigerBruger()
-redigerBruger() : void
DeaktiverBruger
attributes
-service : IOperatoerServiceAsync
operations
+DeaktiverBruger()
+deaktiverBruger() : void
Login
attributes
-client : OperatoerClient
operations
+Login()
+oprLogin() : OperatoerDTO
~onClick() : void

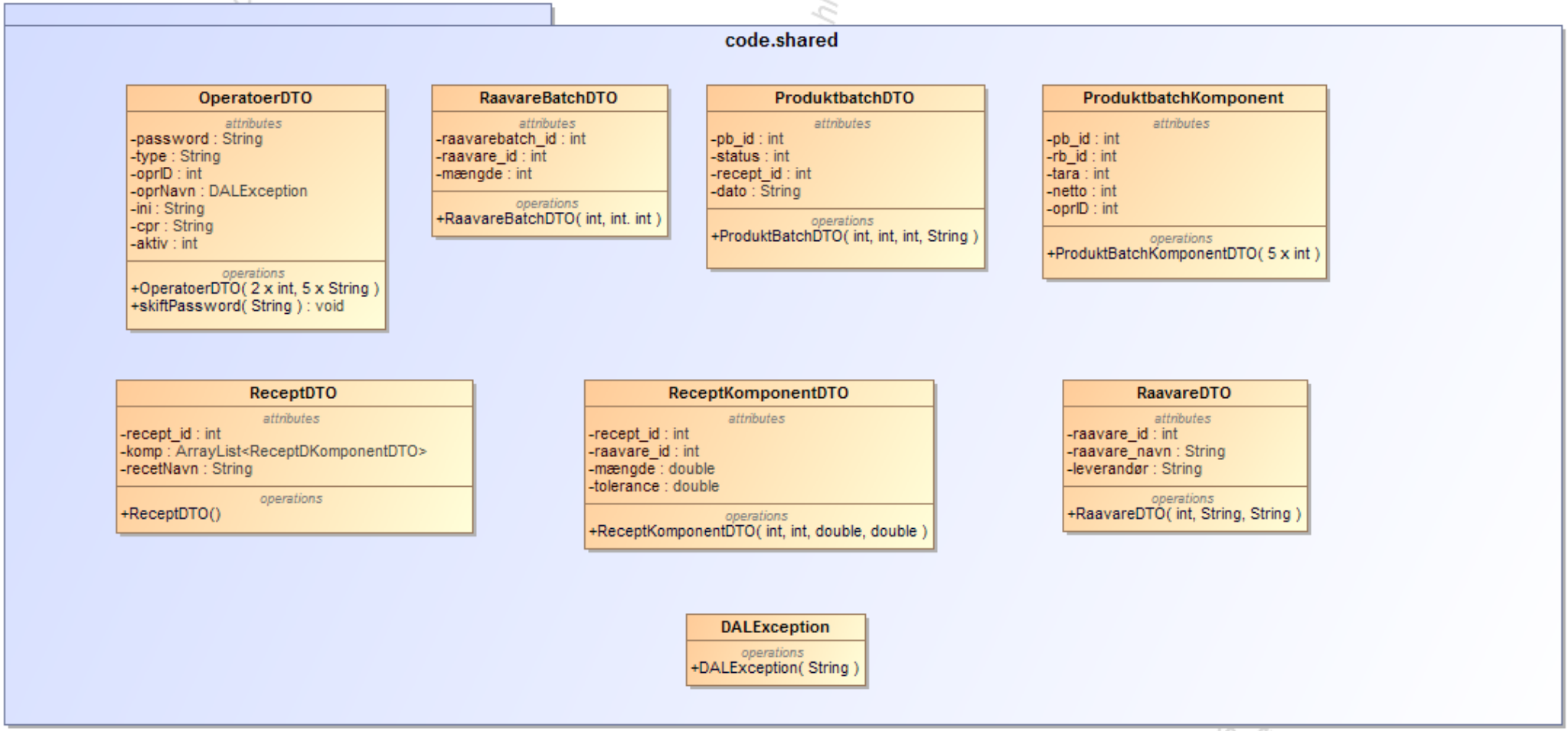
I denne pakke har vi kun en enkelt klasse. Denne klasse opretter vores paneler og login vindue, og står ligeledes for logo'et til web-siden.



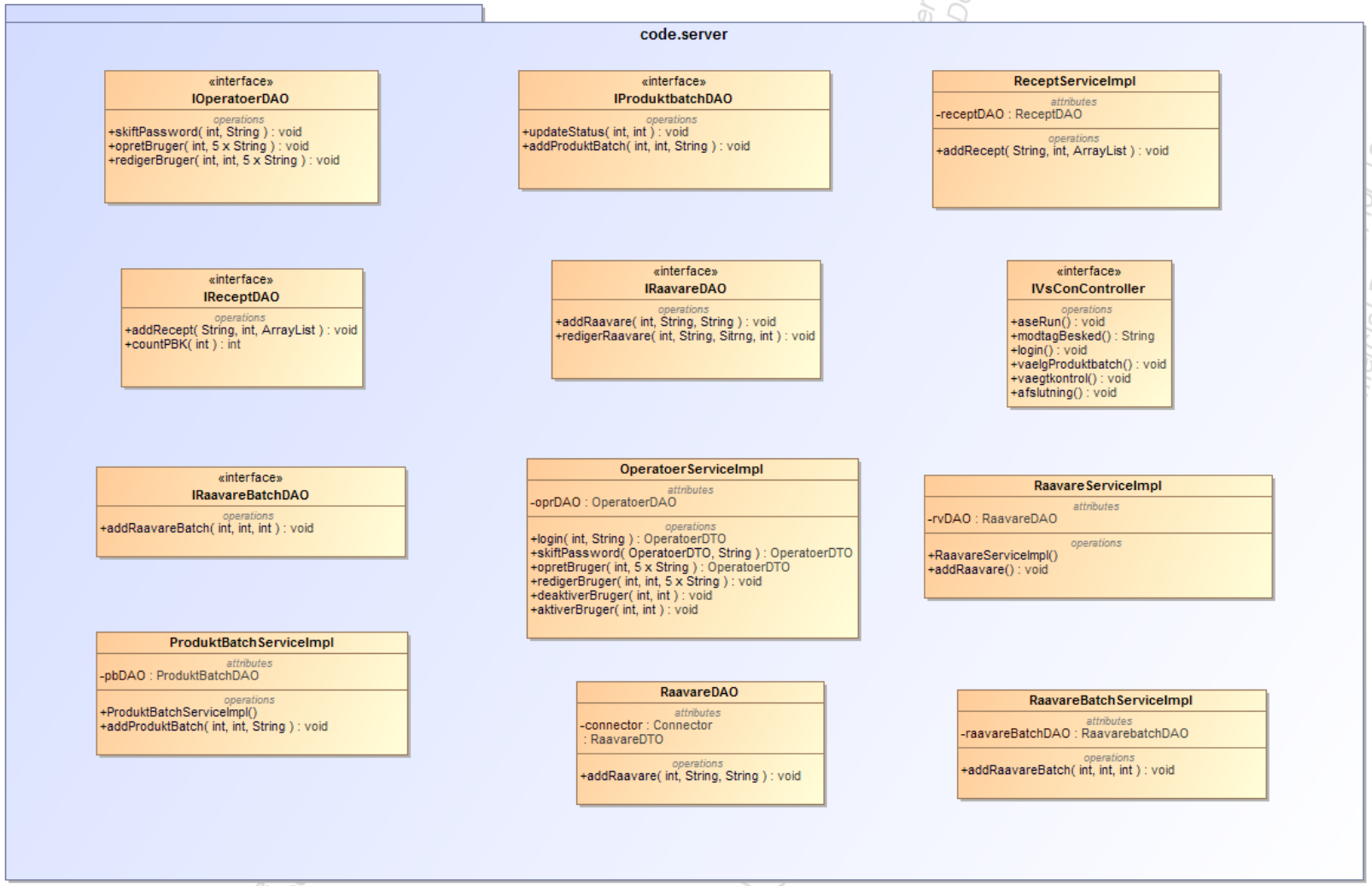
Nedenfor ses vores connector pakke. Denne indeholder to klasser, hhv Connector og ConnectorVaegt. Connector klassen har selvsagt ansvar for at oprette forbindelse til vores database. Som det fremgår af klassen indeholder denne de fornødne informationer til at oprette en forbindelse til vores databaseserver, som bliver beskrevet i et senere afsnit. Ifølge opgaveoplægget skulle denne web-applikation kunne tilkobles en fysisk vægt, der kunne bruges til at afveje og gemme data. ConnectorVaegt klassen står for forbindelse til vægten, som sættes til via ethernet kabel og en tilsvarende IP-adresse, der vil være at finde på den fysiske vægt. Connector klassen står ligedes for håndtering af de databasekald vi opretter i vores DAO klasser.



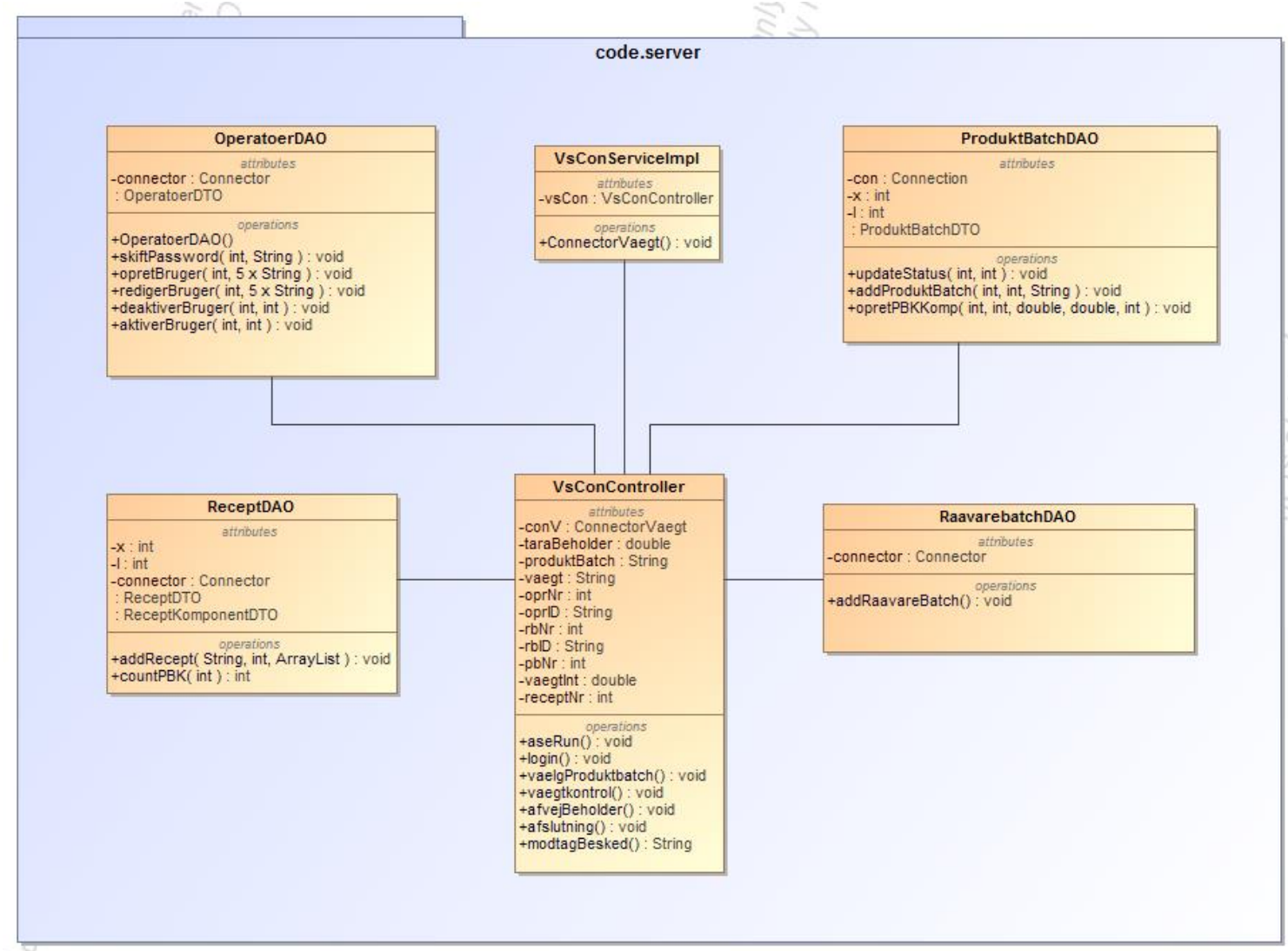
Pakken code.shared indeholder klasser, der indeholder vores getters- og setters-metoder. De inputs vi giver når vi fx opretter en ny bruger bliver gemt i disse attributter og de bruges selvsagt til igen at hente denne information. Dog med undtagelse af vores DALException klasse, som står for håndtering af de forskellige Exceptions der måtte opstå.



Nedenfor ses vores code.server pakke. Denne indeholder alle vores interfaces samt implementerings klasser for service delen af vores web-applikation. Vores interface klasser er lavet, for at vi kan være sikre på at de metoder vi ved skal være i systemet altid altid, hvis vi på et tidspunkt skulle ønske at skifte den klasse ud der implementere en af dem. Dette kan være fornuftigt, hvis det skulle ske, at en klasse skal ændres væsentligt, uden at ændre på de faste metoder, man har hevet med fra interfacet. Service implementerings er beskrevet i et afsnit senere i rapporten.

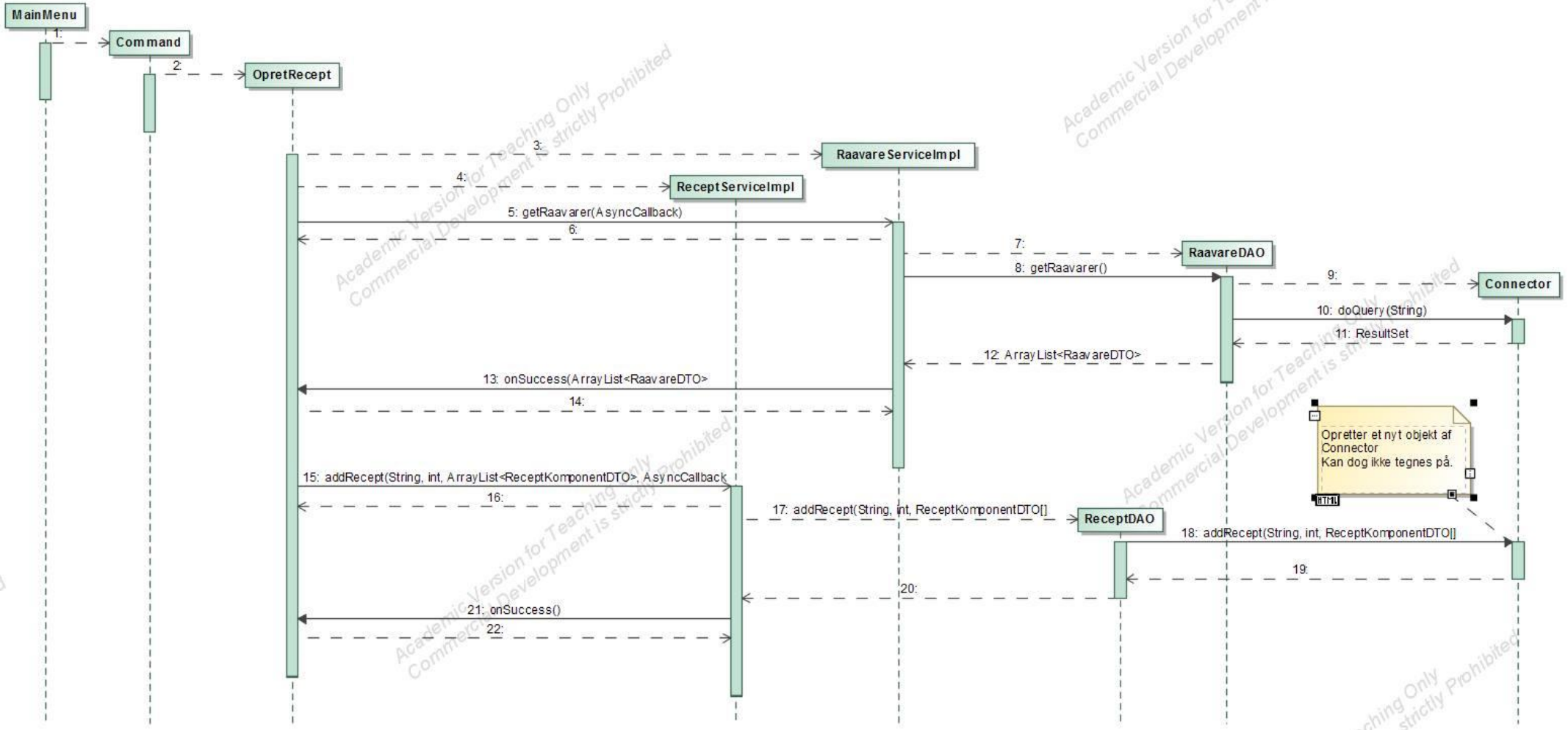


I code.server pakken, ser vi klasserne, som viser hvordan klasserne hænger sammen med simulatoren, og hvilke attributter denne skal tage imod. Det fremgår også at de forskellige klasser, hvilke andre klasser der er forbindelse til. Dvs klasser fra andre pakker. Disse er opgivet som en attribut af den type, som vi har forbindelse til.
Nedenstående DAO klasser er jo dem, der står for at gemme data i vores database, og disse data bruger vægtsimulatoren. Simulatoren er dog ikke fuldt funktionel, men dette er beskrevet i manglende implementations-afsnittet.



Design-sekvensdiagram

I nedenstående diagram er der taget udgangspunkt i funktionaliteten til at oprette en ny recept i systemet. Her ses det, at vores MainMenu opretter en Command, som derefter opretter "OpretRecept". I det, at OpretRecept bliver lavet, opretter den både et objekt af RaavareServiceImpl og af ReceptServiceImpl hvorefter der bliver lavet et asynkront metode kald, der henter alle vores råvarer i databasen og fylder disse ind i vores dropdown menu'er. Da metode kaldet er asynkront, kommer der først et void svar tilbage, så client siden kan køre videre mens den venter på, at server siden svarer. I det her tilfælde venter client siden dog på svaret før den kan gå i gang med det næste. På server siden er resten af kaldene synkrone. Her snakker RaavareServiceImpl med RaavareDAO, som opretter et objekt af vores connector der henter dataen i databasen og returnerer det. Når det kommer til RaavareServiceImpl bliver det herefter returneret til OpretRecept i form af et nyt metode kald onSuccess hvortil der kommer et void svar. Når først alle råvarer er blevet hentet, har brugeren mulighed for at udfylde alle felter og derefter trykke på submit knappen der ligeledes er lavet med et asynkront metode kald der fungerer på præcis samme måde som beskrevet ovenover, med undtagelsen af, at der ikke bliver hevet data ud af databasen, men derimod ligger vi en ny recept og dertilhørende receptkomponenter ind i databasen.
Ligesom med BCE-modellen har vi valgt blot at demonstrere et enkelt diagram her, da de andre i selve GWT-delen af projektet virker på præcis samme måde. Vi kunne have valgt også at lave et for vores ASE, men da den stort set kun består af en klasser der både indeholder data og alt logikken uden at "uddelegere" noget mente vi, at det var et ligegyldigt diagram at lav


MainMenu
Command
OpretRecept
RaavareServiceImpl
ReceptServiceImpl
RaavareDAO
Connector
ReceptDAO
1:
2:
3:
4:
5: getRaavarer(AsyncCallback)
6:
7:
8: getRaavarer()
9:
10: doQuery(String)
11: ResultSet
12: ArrayList<RaavareDTO>
13: onSuccess(ArrayList<RaavareDTO>
14:
Opretter et nyt objekt af Connector
Kan dog ikke tegnes på.
15: addRecept(String, int, ArrayList<ReceptKomponentDTO>, AsyncCallback
16:
17: addRecept(String, int, ReceptKomponentDTO[]
18: addRecept(String, int, ReceptKomponentDTO[]
19:
20:
21: onSuccess()
22:

Database

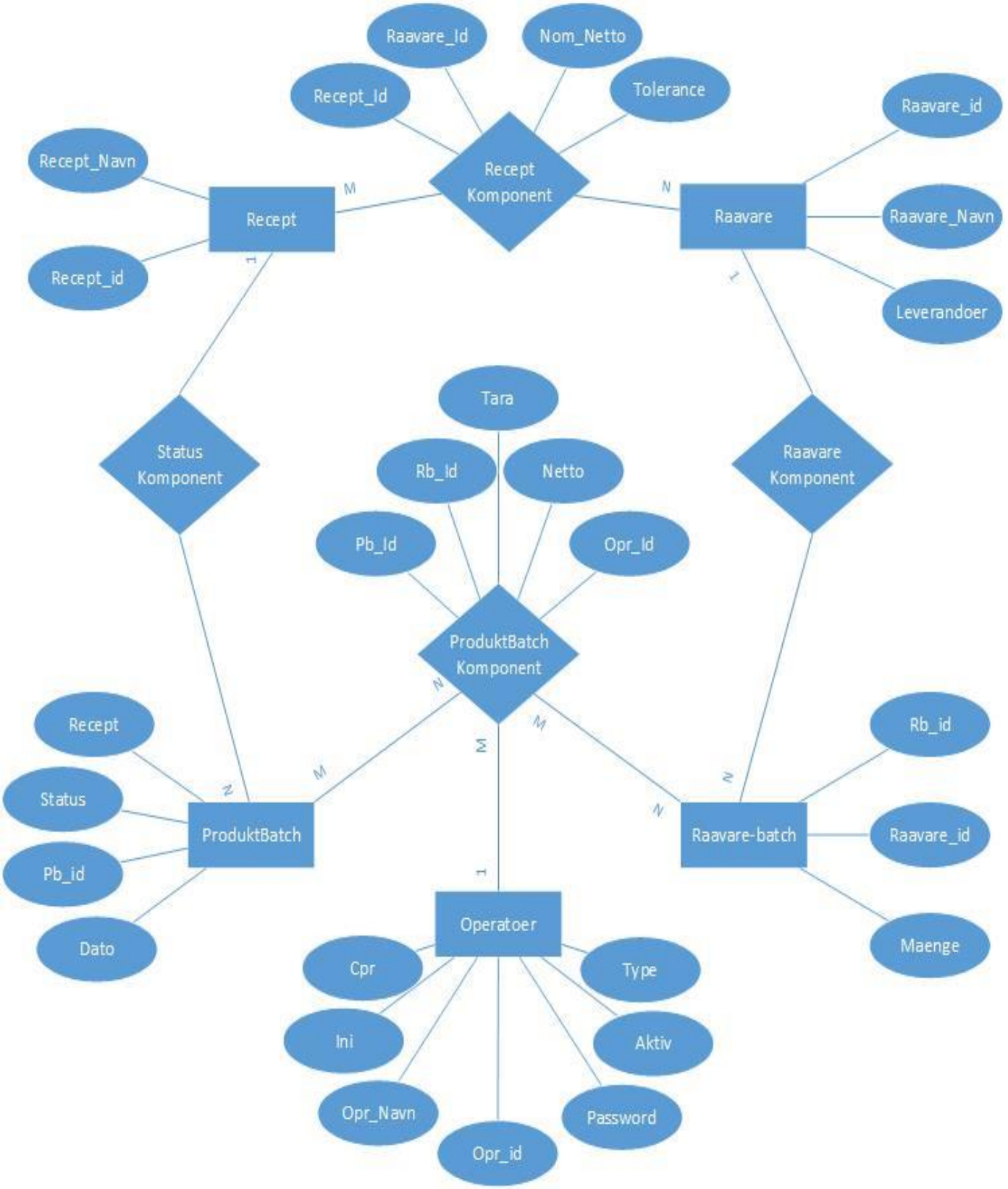
Normalisering

For at en database er godt opbygget skal man forsøge at undgå redundans. Redundans betyder at data er opbevaret flere steder og for at undgå dette benytter man normalisering.
Den database som vi benytter er på tredje normalform. Det betyder at vores tabeller overholder kravene på første, anden og tredje normalform.
For at en tabel er på første normalform kræver det at tabellen har en primær nøgle som unikt kan identificere en række i tabellen. Desuden må der maksimalt være en ting i hver celle.
Hvis vi for eksempel kigger på vores tabel operatoer hvor opr_id er primal key og de resterende kolonner er attributter som kan være null. Kan man se at vi overholder dette krav. Vi har nemlig en primal key og vi har ikke mere end en værdi i hver tabel.
Kravet for at vores tabel er på anden normalform kræver at vores attributter afhænger af hele vores primal key. Når jeg siger hele er det fordi en tabel godt kan have en primal key defineret ud for mere end en kolonne. Hvis man igen kigger på vores tabel kan man se at alle vores attributter afhænger af vores primary key.
Til sidst har vi så tredje normalform som er den form vores tabeller er på. Her er kravet som sagt at tabellen allerede overholder kravene på første og anden normalform og derudover at ingen ikke nøgle attributter er transitivt afhængige af primærnøglen. Dette vil for eksempel sige at vi ikke må have en attribut som afhænger af en anden attribut som afhænger af primal key'en. Dette overholder vi også og derfor må vi konkludere at vores tabel er på tredje normalform.

opr_id	opr_navn	ini	cpr	password	aktiv	type
1	Sys-Admin	SA	0000000000	Test1234	1	administrator
2	Martin Radgaard	MR	1301931234	Hej123	1	farmaceut
3	Frank Thomsen	FT	1806934321	Heej123	1	værkfører
4	Mikkel Hansen	MH	1906921234	Heeej123	1	operatør

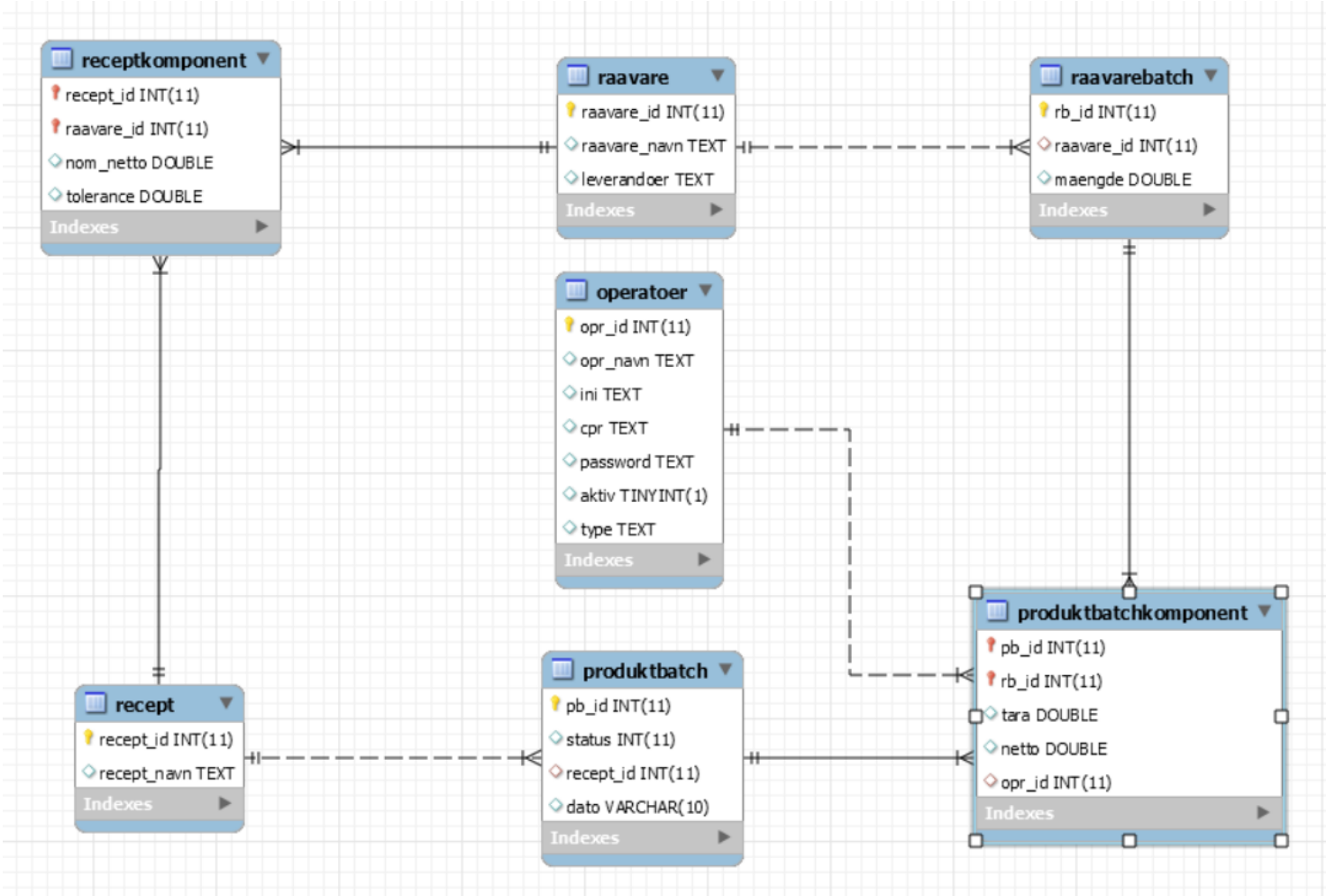
E/R Diagram

Vi har lavet et E/R diagram, som viser vores tanker bag, hvordan vores database skal bygges op. Vi har beskrevet de entiteter, som vi mener skal være med, samt deres attributter. Diagrammet viser så relationerne de forskellige entiteter imellem, og hvordan vores database skal struktureres. Relationerne, har forskellige navne, da en relation kan blive til en tabel senere i forløbet, og kan derfor også indeholde attributter. Som man kan se på vores diagram, har vi også beskrevet forholdet mellem entiteterne, som enten er beskrevet med et N, M eller 1, som betyder at det enten er et "mange til mange" forhold, som er vist med et M ved den ene og et N ved den anden entitet. Der kan også være et "en til mange" forhold som er vist ved et M og et 1 tal. Til sidst er der et "1 til 1" forhold, som er vist ved et 1 tal ved hver entitet.
Som man kan se på vores diagram herunder, er recept, raavare, raavarebatch, produktbatch og operatoer entities og raavare komponent, recept komponent, status komponent og produktbatch komponent er alle relationer i mellem entiteterne og resten er attributter til enten entiteterne eller relationerne.


Raavare_Id
Nom_Netto
Recept_Id
Tolerance
Recept Komponent
Recept_Navn
Recept
M
N
Raavare
Raavare_id
Raavare_Navn
Recept_id
1
1
Leverandoer
Tara
Status Komponent
Rb_Id
Netto
Raavare Komponent
Pb_Id
Opr_Id
ProduktBatch Komponent
N
M
M
Recept
Rb_id
Status
M
N
N
Raavare-batch
Raavare_id
ProduktBatch
Pb_id
1
Operatoer
Maenge
Dato
Cpr
Type
Ini
Aktiv
Opr_Navn
Password
Opr_id

Skema diagram

Vi benytter os af en database, som vi har implementeret. Databasens skema diagram kan ses herunder. Dette diagram, viser hvad de enkelte tabeller indeholder og hvordan deres primær nøgler og fremmednøgler hænger sammen. Diagrammet er lavet i MySql Workbench, som reverse engineering, da databasen var givet på forhånd, vi har dog lavet enkelte ændringer i den.
De gule nøgler er primærnøgler, og de røde er både primærnøgler og fremmednøgle.
Alle de blå tomme firkanter er attributter. Kardinaliteterne tabellerne imellem er symboliseret ved de pile, som er tegnet ind. Pilene betyder at der er mange til en forhold ved alle vores klasser, hvor de ender der går ud i flere grene, som fx ved receptkomponent, betyder en eller flere, og den ende ved raavare betyder en og kun en.



Ændringer i databasen

Vi har som sagt lavet enkelte ændringer i denne database i forhold til den database vi fik stillet til rådighed i kurset "02327 Indledende databaser og database programmering". Vi har i operatoer tabellen tilføjet to ekstra værdier i *aktiv* og *type*. Vi har lavet værdien *aktiv* som en int, og har valgt at 1 symbolisere at aktøren er aktiv og 0 hvis operatøren er deaktiveret. Værdien *type* har vi lavet som en TEXT, da vi godt ville kunne gemme om en aktør enten er operatør, værkfører, farmaceut eller administrator. Til sidst har vi i tabellen "produktbatch" tilføjet værdien *dato*, som er lavet som en VARCHAR, hvilket er gjort fordi vi bruger den til at gemme den dato, som produktbatchen bliver oprettet. Dato variablen er lavet som en tekst da vi skriver en dato således YYYY-MM-DD.

Transactions

Vi har i vores projekt også benyttet os af transactions, som er en funktion man bruger, hvis man vil køre flere statements uden at der bliver committed, hvilket er smart, hvis man laver flere ændringer i databasen på en gang og man så lige pludselig mister forbindelsen. Hvis man mister forbindelsen uden at bruge transactions, så vil det kunne ske, at kun dele af alle statement vil blive opdateret i databasen, og derfor vil komme fejl.
Vi har brugt transactions, når vi opretter en recept, da vi samtidig med det, opretter en receptkomponent. Man kan se vores eksempel herunder.

```
@Override
public void addRecept(String receptNavn, int recept_id, ArrayList<ReceptKomponentDTO> komp) throws DALException{
    Connection con = connector.getConnection();
    try {
        con.setAutoCommit(false);
        connector.doUpdate("INSERT INTO recept VALUES("+recept_id+", '"+receptNavn+"');");
        for (int i = 0; i < komp.size(); i++) {
            connector.doUpdate("INSERT INTO receptkomponent VALUES("+recept_id+", "
                    + ""+komp.get(i).getRaavare_id()+", "+komp.get(i).getMængde()+", "
                    + ""+komp.get(i).getTolerance()+");");
        }

    } catch(SQLException e) {
        try {
            con.rollback();
        } catch (SQLException e1) {
            e1.printStackTrace();
        }
        throw new DALException(e.getMessage());
    } finally {
        try {
            con.commit();
            con.setAutoCommit(true);
        } catch (SQLException e) {
            e.printStackTrace();
        }

    }

}
```

I vores implementation, som man kan se ovenover, starter vi med at sætte autocommit til false, da det gør, så de statements vi laver senere, ikke bliver committed, før vi beder om det. Herefter laver vi de to statements, som vi så senere committer. Hvis det ikke går godt, har vi så kaldt rollback metoden, som starter transactionen forfra. Når hele transactionen er committed sætter vi autocommit metoden til true igen, så fremtidige statements bliver kørt med det samme.

Implementering

Se bilag for vejledning til installation og systemkrav.

I dette afsnit vil vi komme ind på vores implementering, som indeholder en webbaseret brugergrænseflade, en database, et database access lag og en ASE som skal kunne kommunikere med en vægt. Vi kommer ind på, hvad vi har implementeret, hvordan det virker og hvilke ting vi ikke har nået at få med.

DTO klasser

Vores DTO klasser består af data. Disse klasser bliver brugt når vi henter information ud fra databasen, da vi gemmer de data i objekter af vores DTO klasser.
Dette bliver gjort i vores DAO klasser, hvor vi henter informationen fra databasen, og lægger disse informationer ind i vores DTO objekt. Når man snakker om DTO klasser skal man være opmærksom på, at disse typisk vil blive brugt på både client og server side. Derfor er det vigtigt, når man arbejder med GWT, at man sørger for at placere disse klasser i "shared"-pakken. Derudover skal de implementere interfacet Serializable samt have en tom konstruktør foruden den konstruktør man selv har tænkt sig at benytte. Såfremt man ikke overholder disse krav, vil men ikke kunne bruge sin DTO klasse på client siden, da GWT vil brokke sig over, at det ikke kan compile.

DALException

Vi har oprettet vores egen exception klasse, som extender Exception. Det er gjort med henblik på at være helt sikker på, at de exceptions der bliver smidt er Serializable (se DTO klasser). Et godt eksempel på hvorfor det har været nødvendigt er, at SQLExceptions ikke er serializable, og derfor har vi været nødt til at pakke beskeden fra sådanne exceptions ind i en ny exception for at kunne få en meningsfuld besked ud af det. DALExceptions er også blevet brugt når vi af en eller anden årsag blev nødt til at smide en exception, da vi jo vidste, at den ville kunne bruges på både client og server siden. DALException skal ligge i shared pakken ligesom DTO af samme årsager.

DAO klasser

I vores DAO klasser laver vi vores implementation, hvor vi laver vores databasekald. I disse klasser er der i alle oprettet forbindelse til vores database ved at oprette et objekt af vores Connector klasse. Når man har et objekt af Connector klassen kan vi derefter oprette en Connection som vi kan bruge

funktioner som “setautocommit”, “rollback” og “commit” til fx vores transactions. I vores DAO klasser kunne vi også have benyttet os af prepared statements (se afsnittet manglende implementering).

ServiceImpl klasser

ServiceImpl klasserne bliver oprettet som instanser af både det asynkrone og synkrone interface i forskellige klasser på client siden som skal snakke med server siden. Her har det asynkrone interface en RemoteServiceRelativePath attribut, som vi bruger i web.xml filen til at pege på den pågældende ServiceImpl klasse. Det betyder i praksis, at vi peger på vores ServiceImpl klasse som værende den klasse på server siden, clienten skal snakke med. I web.xml filen vil alle vores ServiceImpl være beskrevet som værende servlets med dertilhørende url-patterns, der bruges til at pege på hvor ServiceImpl kan findes henne.
I vores projekt bruger vi ServiceImpl som et bindeled mellem client siden og vores DAO klasser som laver alle vores database kald.

GIT

Vi har i vores arbejde med projektet benyttet os af Github, da vi herinde har kunnet lave et repository, så alle fra vores gruppe har kunnet arbejde på det samme projekt samtidig. Fordelen ved at arbejde på denne måde, er at alle kan følge med hvor langt vi er nået, og så hvis der skulle opstå en fejl, kan man nemt rulle tilbage til en tidligere udgave hvor fejlen endnu ikke var implementeret. Vores repository står der beskrevet om i bilag under GIT afsnittet.

Udviklingsmetode

I forbindelse med projektet har vi benyttet nogle forskellige udviklingsmetoder og arbejdsformer. Vi har benyttet os af reverse engineering. Det vil sige at vi har udarbejdet vores diagrammer ud fra vores implementering i stedet for at lave vores implementering ud fra diagrammer. Dette er oftest et dårligt valg, da det gør implementeringen meget lettere hvis man allerede på forhånd ved hvordan man vil opbygge programmet og hvad de forskellige variabler og metoder skal hedde.
Udfordringerne ved dette, som kaldes forward engineering, kan være at få skabt sig et overblik over projektet. Så kan det være lettere at bygge på efterhånden som man implementere, selvom dette kan give problemer. Problemet kan for eksempel være at man pludselig finder ud af at programmet er skruet sammen så en feature bliver svær at få implementeret.

En af de arbejdsmetoder vi har forsøgt os med er pair programming. Det som pair programming går ud på er at man sidder to om en computer og implementerer. Den ene person skriver koden, denne rolle er ofte kaldet driver. Imens den anden person sidder og tjekker om det som bliver skrevet er korrekt, denne rolle kaldes for observatør. Rollerne skiftede vi engang imellem i løbet af dagen. Det var første gang vi forsøgte os med dette og vi synes det gik utrolig godt. Fordelene ved at benytte denne metode er blandt andet at fejlene ofte blev fundet med det samme eller inden for kort tid. Desuden blev vores metoder implementeret på en god måde fra start, da man pludselig sad to om det og ud fra dette kunne vælge den bedste løsning. En anden fordel er at personerne som deltager, fra start af forstår hver eneste detalje i koden fra start af. Ulemperne ved denne arbejdsmetode kan være at implementeringen kan gå langsommere, da man sidder to om en pc i stedet for at have en hver. Dette følte vi dog ikke var tilfældet da tiden vi sparede på at finde fejl og finde på en god løsning gik meget hurtigere. En anden ulempe kan være at det er den ene programmør som laver det hele, men vi synes heller ikke dette var tilfældet da vi følte at vi supplerede hinanden godt samtidig med at der hvor partneren havde en bedre løsning lærte man selv noget. Så vi vil helt sikkert overveje dette igen i fremtiden.

Coding Standards

Indenfor software udvikling har man nogle såkaldte coding standards. Det som dette begreb betyder er at koden skal være let at læse for andre programmører, for eksempel i forbindelse med vedligeholdelse, men også i udviklingsfasen hvis man er flere programmører om et projekt. En af de ting man gør, er at kalde sine variabler, metoder og klasser det de rent faktisk skal gøre eller bliver brugt til. Hvis vi tager et eksempel for vores projekt har vi for eksempel en metode i vores ASE som vi bruger til at modtage retur beskederne fra vægten. Denne metode har vi valgt at kalde modtagBesked, det betyder at nu ved alle at denne metode har noget at gøre med at modtage en besked, havde det været helt optimalt skulle dette nok have været på engelsk.
En anden del af coding standards er at lave kommentarer i sin kode for yderligere at uddybe en del af sit program. Vi har dog nedprioriteret dette en smule i dette projekt grundet tidspres.

Brugergrænseflade

Til brugergrænseflade implementeringen af vores projekt har vi benyttet Google Web Toolkit (GWT), som omdanner java kode til javascripts der kan vises som en web side. Vi har i GWT benyttet os af UiBinder, som gør det lettere at style de forskellige komponenter. Vi har også benyttet os af Remote Procedure Calls (RPC) asynkrone callbacks til at skifte imellem de forskellige sider i vores GUI alt efter om et metodekald gik godt eller dårligt. Dette er gjort for at sikre, at man ikke blev ledt ind på en side hvor man enten ikke skulle have adgang eller sikre mod, at der senere i forløbet ville ske fejl på grund af manglende informationer. Vi har brugt RPC til at kommunikere fra client siden til server siden af vores applikation. Fordelen ved RPC i den forbindelse er, at vi kan benytte asynkrone callbacks, så hele vores web side ikke "fryser" mens, at clienten venter på, at serveren svarer. På server siden har vi koblet en database til, så alle ændringer der bliver lavet bliver gemt og kan hentes når det skal bruges. For at kunne snakke med denne database har vi en Connector klasse. Denne bliver omtalt nede i Database afsnittet i bilag. Vi har også på server siden koblet en vægt connector til, som kan kommunikere med en Mettler BBK vægt.

Den måde vi har lavet vores menu på nu, hvor vi først opretter objekter af de nye menuer, når vi logger ind på dem, er meget smartere end den måde vi startede med at gøre det på. Vi startede nemlig med at oprette samtlige sider ved login, hvilket godt kunne give en lang svartid inden man så var logget ind i systemet. Den måde vi så har lavet det på nu, der undgår vi den længere svartid ved login, dog mister vi lidt af hastigheden i responstiden når man vælger mellem de forskellige menupunkter. Derudover var der en stor ulempe ved vores første fremgangsmåde. Den sidst oprettede side ville altid blive sat på vores content panel (altså det panel vi bruger til at vise de forskellige GUI komponenter).

ASE

Vores Afvejning Styrings Enhed (ASE) fungere sådan at vi opretter en forbindelse ved at oprette en ny Socket, som lytter på port 8000 og som skal have indtastet IP nummeret fra den vægt, som vi skal bruge. Dette foregår i vores ConnectorVaegt klasse.

Så har vi lavet en masse metoder i vores VsConController klasse på server siden, som bliver kaldt i vores VsConServiceImpl klasse, som sender de asynkrone callbacks til vores client side i Afvejning klassen.

Når vi sender en besked bliver det gjort ved en DataOutputStream og når vi modtager en besked bruger vi en DataInputStream, som vi har indkapslet i metoden modtagBesked().

Brugervejledning

I vores ASE har vi slavisk programmeret hvad der skal sendes til vægten, og vi går kun videre hvis svarene fra vægten svarer overens med hvad vi forventer, ellers vil den kører metoden igen. Dette er gjort, så det ikke er muligt at få vægten til at lukke ned midt under en afvejning, så data kan gå tabt. Vi har lavet vores ASE's metoder ud fra disse punkter:

1. Operatøren har modtaget en produktions forskrift på papir fra værkføreren.
2. Operatøren vælger en afvejning terminal.
3. Operatøren indtaster operatør nr.
4. Vægten svarer tilbage med operatør navn som så godkendes.
5. Operatøren indtaster produktbatch nummer.
6. Vægten svarer tilbage med navn på recept der skal produceres (eks: saltvand med citron)
7. Operatøren kontrollerer at vægten er ubelastet og trykker 'ok'
8. Systemet sætter produktbatch nummerets status til "Under produktion".
9. Vægten tareres.
10. Vægten beder om første tara beholder.
11. Operatør placerer første tara beholder og trykker 'ok'
12. Vægten af tara beholder registreres
13. Vægten tareres.
14. Vægten beder om raavarebatch nummer på første råvare.
15. Operatøren afvejer op til den ønskede mængde og trykker 'ok'
16. Pkt. 7 – 14 gentages indtil alle råvarer er afvejet.
17. Systemet sætter produktbatch nummerets status til "Afsluttet".
18. Det kan herefter genoptages af en ny operatør.

Når dette er gået igennem har vi tilføjet at man skal fjerne tingene fra vægten, og så nulstilles den, så den er klar til næste afvejning.
Punkt 9 og 13 har vi lavet, så brugeren ikke selv tarere, men at det bliver gjort automatisk, efter der bliver trykket "OK" på vægten, som der bliver bedt om i beskederne 7 og 11.
Punkt 16 har vi lavet i et loop, så den gentager processen så mange gange, som der er recept komponenter til den pågældende recept, da det svarer overens med hvor mange råvarer der skal afvejes i til recepten.

Kommandoer

De forskellige metoder som vi skriver til vægten bliver kommenteret herunder:

RM20 8 “String” “String” “String” → Når denne kommando bliver skrevet til vægten , svarer vægten tilbage med RM20 B, og derefter venter den på brugerinput, samtidig med at den har overskrevet alt, som der står på vægtens display. Dette brugerinput bliver så sendt tilbage til clienten som RM20 A “*beskeden*”. Det er derfor at vi i vores Controller kalder to modtagBesked() metoder for hver RM20 8 kald, da den som sagt sender to beskeder tilbage.

P111 “String” → Når denne kommando bliver skrevet til vægten, viser den beskeden i bunden af displayet.

T → Denne kommando tarere vægten.

S → Denne kommando viser hvor meget vægt, som der er på vægten.

K 3 → Denne kommando gør, så man kan trykke på ON/OFF, ->0<-, ->T<- og [-> på vægten. Disse knapper har så forskellige svar, som vi så i vores kode venter på, så man kan nå at lave en afvejning på vægtens display. Vi bruger knappen [-> som trigger, og den svarer tilbage til clienten med beskeden K C 4, som vi så venter på.

K 1 → Denne kommando annullere K 3 ordren, hvilket vil sige at vi kan bruge knapperne normalt igen og at de ikke sender beskeder tilbage til clienten.

Herunder kan man se et eksempel på, hvordan vi i en “P111 “String”” skriver beskeden “Tryk [-> for afvej” i den nederste del af vægtens display og derefter gør så vi kan bruge vægtens knapper til at modtage beskeder, og derefter vente på den rigtige besked for til sidst at nulstille knapperne, så de bruges normalt igen.

```
os.writeBytes("P111 \"Tryk [-> for afvej\"\r\n");
modtagBesked();

os.writeBytes("K 3\r\n");

do {
    modtagBesked();
}while(!modtagBesked().contains("K C 4"));

os.writeBytes("K 1\r\n");
modtagBesked();
```

Det var alle de kommandoer som vægten kan tage imod, der er en del flere som vi ikke benytter os af, som D “String”. Denne kommando skriver en besked på vægtens display, og dækker samtidig for at man kan se, hvad der bliver afvejet, derfor har vi ikke benyttet denne.

Der er også DW kommandoen, som sletter en D “String” besked. Denne kommando benytter vi heller ikke, da vi ikke har benyttet os af D”String” kommandoen, og derfor ikke har nogen besked at fjerne.

Validation

Vi har lavet en Validator klasse, som vi bruger til at validere brugerinput på vores GUI. Metoderne i denne klasse kan se om inputtet er en int, en String, en dato, en double eller om et password er korrekt sat op, med et stort bogstav, et lille bogstav, et tal og at længden på det er minimum 6 karakterer lang. Den sidste metode, gør at vi kan validere om et cpr nummer er for langt samt, at det kun indeholder tal. Grundet måden valider cpr er lavet er der nu også et krav om at skrive cpr numre på formen ddmmyyxxxx.

Manglende implementation

Udprint

Vi fik ikke implementeret at når en værkfører opretter en produktbatch at den så bliver udprintet, da dette ikke var en prioritet vi lagde så meget vægt på. Det gjorde vi ikke da det ikke har noget med om systemet kan fungere at gøre, men kun kosmetisk, så man rent faktisk kan leve sig ind i rollen som en operatør, med en udprintet produktbatch, som skal følges under produktionen. Vi kigger bare i databasen, hvad der skal bruges til for at den produktbatch bliver produceret korrekt. Vi fik ligeledes heller ikke lavet en udprintning af en produktbatch under produktion og en afsluttet produktbatch, som der er vedhæftet som bilag 6 og 7 i opgavebeskrivelsen. Dette er igen nedprioriteret, da vi først og fremmest ville have et system, som fungere frem for at få en ekstra detalje med, da vi stadig sagtens kan følge med i produktbatchen og produktbatch komponenternes status både før, under og efter en produktion.

Validering af input

Vi har kun fået valideret input på client siden, så vi mangler at få valideret mulige input på server siden, men dette var igen en mindre prioritet, da vi jo som sagt allerede for de fleste input valideret på client siden, og derfor næsten er helt sikret, da det er fra vores client side at vi får brugerinput, som oftest er dem der fejler. Det validering, som der kunne være på server siden ville for eksempel være at teste på om et ID allerede var optaget inden vi prøver at smide det i databasen og få en "underlig" fejlmeddelelse, set med slutbrugerens øjne. Hvis vi havde valideret ID, ville vi have kunne give meningsfyldt fejlbesked som "Der findes allerede en bruger med det ID", eller vi ville have mulighed for at sørge for, at der ikke fandtes flere brugere med det samme cpr nummer. Lige netop cpr nummeret har vi dog ikke anset some værende synderligt vigtigt, da vores primary key i operatør tabellen er operatør ID, da vi godt kunne forestille os en situation hvor en person måske havde flere typer brugere. Det kunne for eksempel være i forbindelse med nogle test der skulle laves hos kunden.

ASE

I vores ASE mangler vi at tjekke om vægten stemmer overens med nom_netto, som der står skrevet i receptkomponent, samt at tjekke tolerancen. Det vil sige at når vi laver den endelige afvejning, kommer der ikke en fejlbesked, hvis mængden afviger mere end den tilladte tolerance tillader. Det kunne laves ved at lave en udregning, på hvor meget tolerancen er, både over og under den givne nom_netto vægt, så ville man kunne lave grænser, som man så tjekker at vægten holder sig indenfor, ellers sender man en fejlbesked tilbage, hvor man fx kunne skrive, at vægten er uden for den tilladte tolerance. Vi valgte at prioritere andet højere end dette, da det ikke har nogen indflydelse på om vores program kan fungere, men bare er en fed feature.

Vægt simulator

Vi valgte at fokusere på at få ASE'n til at fungere sammen med en fysisk vægt, og derfor nåede vi ikke at få lavet en vægt simulator som kunne fungere sammen med vores ASE, da simulatoren ikke er programmeret til at sende de samme svar, som den fysiske vægt gør, og da vi bruger svarene fra vægten til at hente info fra databasen, så får vi ikke det rigtige fra vægt simulatoren. Fra vores ASE sender vi også kommandoen "K 3" til vægten, og den kommando kan vægt simulatoren slet ikke læse, da den initialisere knapperne på den fysiske vægt. Så vi valgte at prioritere på at få den ene til at virke helt og det blev den fysiske vægt.

Connection

I vores program på vores GUI har vi en knap som starter en afvejning, denne afvejning er hardcoded til at være en bestemt vægt. Vi kunne have lavet det anderledes, ved at lave flere knapper, hvor hver knap i vores GUI kunnet have oprettet forbindelse til hver deres veje terminal. Dette valgte vi også at nedprioritere, da det også er en fed feature, men ikke en nødvendighed, for at programmet kan køre. Vi kan i vores system kun benytte veje terminal 1, da det er dens IP, vi har hardcoded ind i vores Connector klasse. Hvis vi skal benytte veje terminal 2 vil vi skulle ind i koden og ændre den IP som vi opretter vores Socket med, så den stemmer overens med veje terminal 2's IP, men det vil slutbrugeren formentlig ikke selv kunne gøre.

Prepared Statements

I vores database kald burde vi have brugt prepared statements, da det er en mere sikker måde at kommunikere med databasen. Et Prepared Statement bliver anvendt til at udføre databasekald med høj effektivitet. De bliver oftest anvendt i SQL forespørgsler og opdateringer, som passer meget godt med det vi har lavet, som er doQuery og doUpdate.

Forskellen på et Prepared Statement og et "normalt" statement er, at værdierne, som fx skal indsættes i tabellen er uspecificeret og symboliseret med et spørgsmålstegn (?). Det kunne se således ud:

```
INSERT INTO operatoer (opr_id, opr_navn, ini, cpr, password, aktiv, type)
VALUES(?,?,?,?,?,?,?)
```

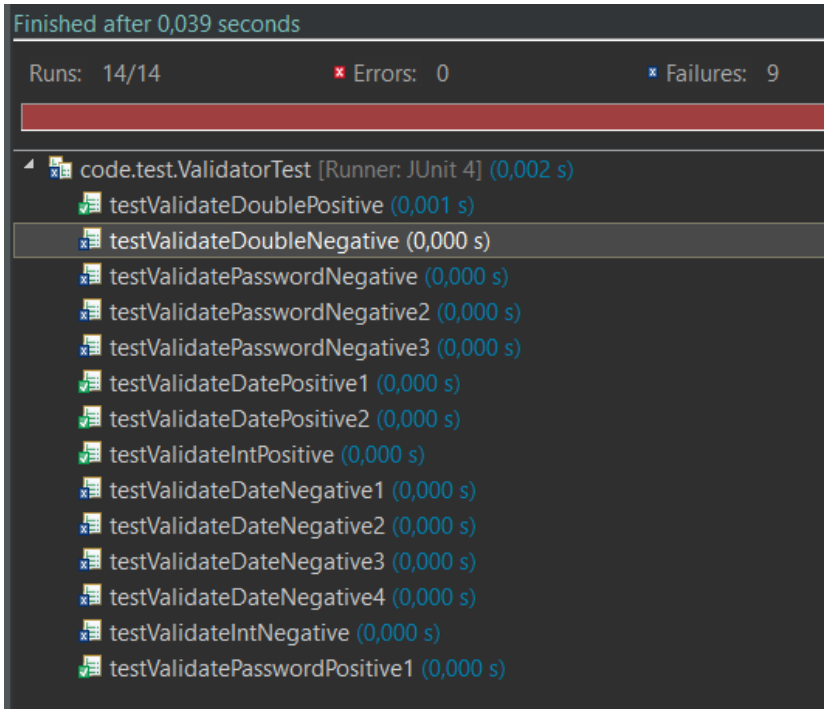
Værdierne som skal sættes ind i tabellen i ovenstående eksempel, bliver sat ind på et senere tidspunkt, men statementet bliver gemt her. Når værdierne nu skal indsættes, bliver det gjort en ad gangen, hvor man vælger hvilket spørgsmålstegn der skal indsættes på. Man tjekker først hvilken type værdi der skal indsættes på det gældende spørgsmålstegn og skriver dernæst *set"typen"("? nr.", "beskeden")*, hvor typen er hvilken type værdien har, som fx int eller String. "? nr." er hvilket spørgsmålstegn der er tale om i statementet ovenover, og "beskeden" er selvfølgelig, hvilken besked som der skal indsættes på spørgsmål tegnets plads. Denne besked skal passe overens med hvilken type, som det er der bliver sat. Når dette så er gjort for alle spørgsmålstegn, vil man lave en executeUpdate metode, som kører alle de statements der erstatter spørgsmålstegnene og til sidst bliver det hele committed.

Det der er smart ved at gøre det på denne måde, er at man beskytter sin database imod SQL injection, som er et angreb rettet mod databaselaget i en applikation, da man uden prepared statements kan indskyde fjendtlig SQL kode i et SQL kald og derved få adgang til følsomme informationer fra databasen, som fx en administrators brugernavn og password eller blot droppe tabeller, som ødelægger systemet for slutbrugeren.

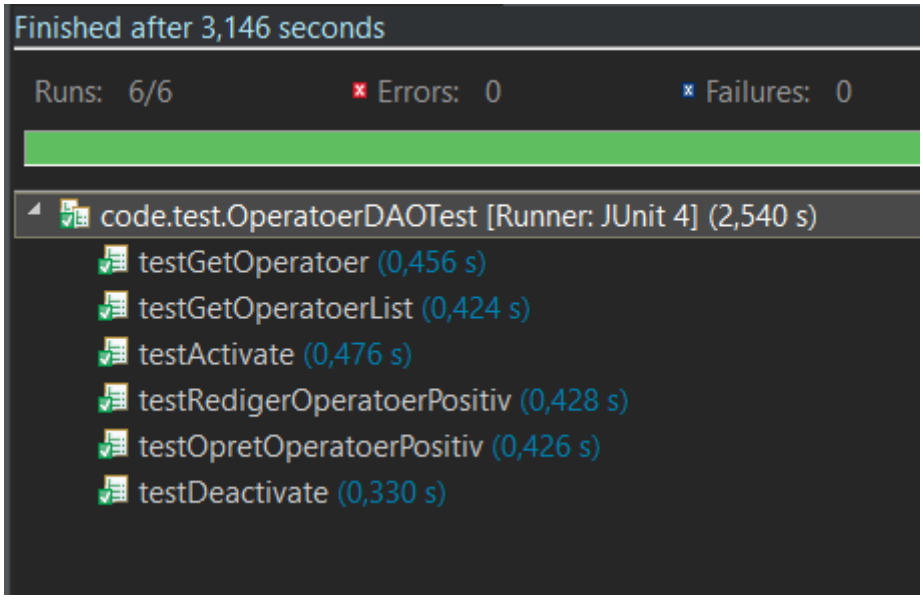
Test

Vi har skrevet JUnit test til alle vores DAO klasser og vores validering klasse. I DAO klasserne er der kun positive test mens der i validering testene er en god del negative tests. De negative tests er der blot for at dokumentere, at valideringen virker i den forstand, at den ikke "godkender", at vi prøver at sende forkert data ind i systemet.

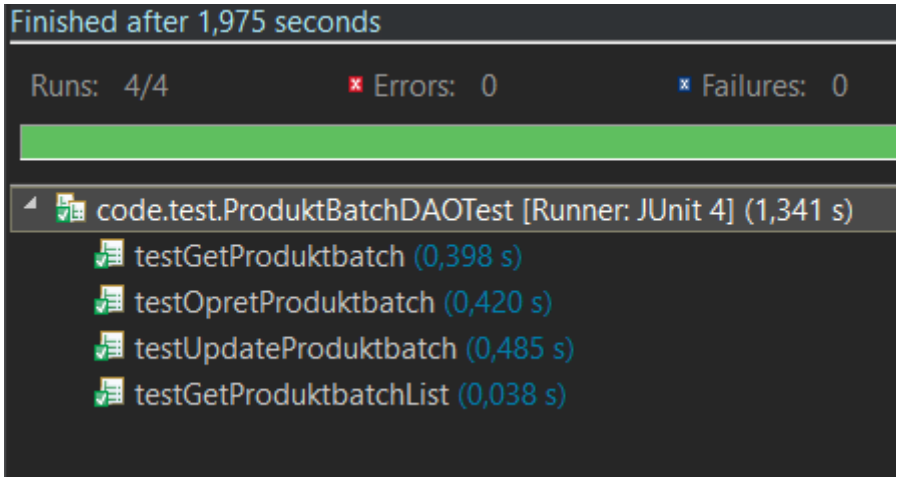
JUnit



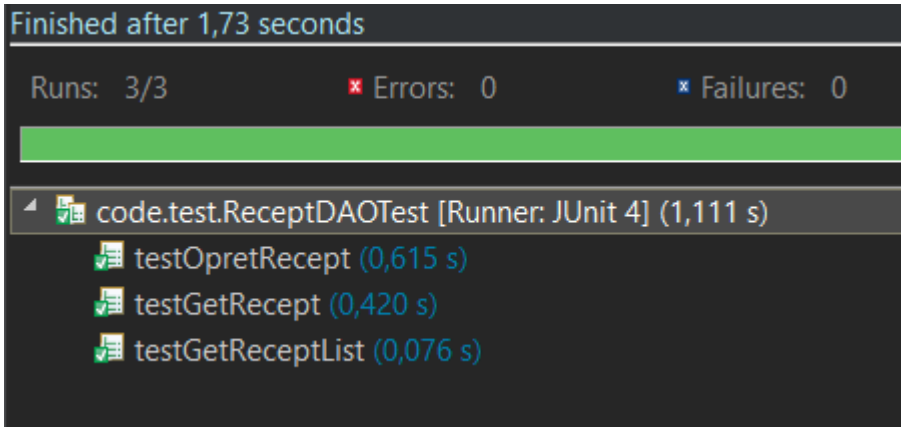
Her ses resultatet af vores validator test. De 9 failures er for hver af de negative test vi laver. En negativ test betyder blot, at det vi tester ikke burde lykkedes. Et eksempel fra denne test case kunne være, testValidateDoubleNegative hvor vi prøver at give den en string med bogstaver som input. Her bør den gerne throw en NumberFormatException, som bliver catchet og derefter returnerer metoden false. Alle de positive test, tester ting som validatoren meget gerne skulle kunne. De lykkedes også alle sammen. Password testene tester forskellige forskellige passwords hvor et enkelt lever op til minumums kravene og så har vi tre negative test hvor de ikke lever op til kravene. Det kan være at der ikke er nogen tal, ikke er nogen store bogstaver, ikke er nogen små bogstaver eller, at passwordet ikke er over 6 karakterer langt. I dato testene, bliver der testet på hvorvidt den indtastede string er en korrekt dato skrevet på formen yyyy-mm-dd. Her går de fleste af testene ud på om der bliver regnet korrekt hvad skudår angår og om man kan have enten dag 0 i en måned eller om måned 0 findes. De er begge negative.



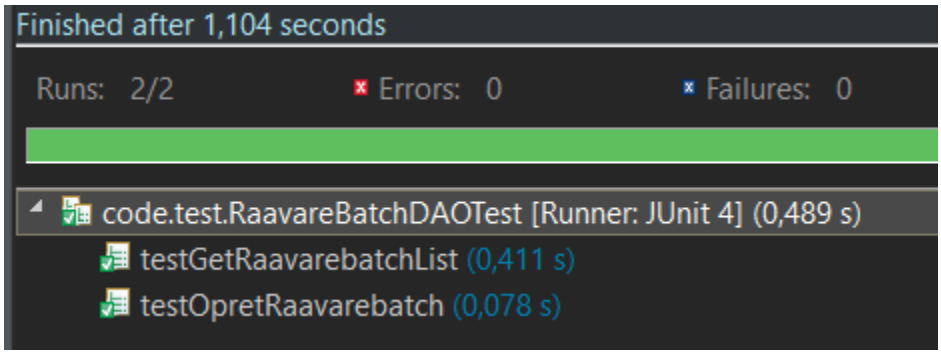
I vores OperatoerDAO test bliver der testet på samtlige af metoderne. Her er alle testene positive, altså de skulle gerne virke alle sammen. Som det ses på billedet var der hellere ingen errors eller failures. Her bliver der testet om vi kan hente en operatør og en arraylist med operatører fra databasen. Vi tester hvorvidt vi kan deaktivere og aktivere systemets brugere og vi tester om vi kan oprette og redigere systemets brugere.



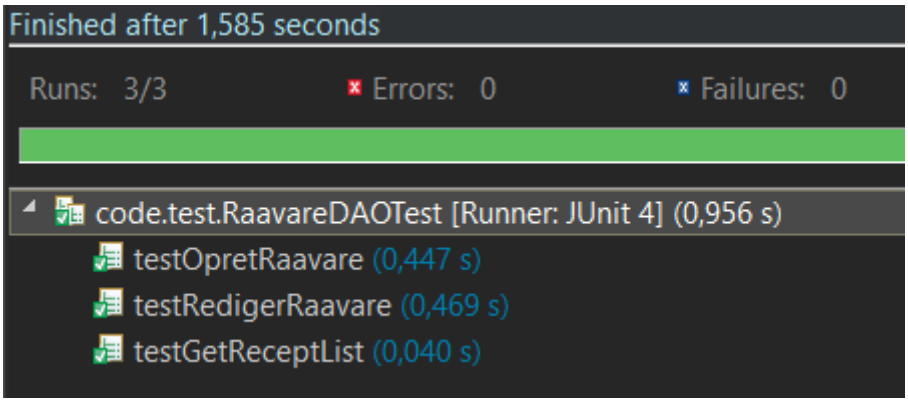
I ProduktBatchDAO testen bliver der ligeledes testet på hvorvidt vi kan hente både et produktbatch og en arraylist med produktbatches fra databasen samt om vi kan oprette et og om vi kan opdatere statussen. Statussen bliver brugt til at se hvor i processen vores produktbatches er, altså om de blot er oprettet, om de er igangsat eller om de er færdige. Her er alle testene igen positive.



I RecetDAO testen tester vi endnu engang om vi kan hente både en enkelt recept og en arrayliste med recepter og så tester vi om det er muligt at oprette en ny recept. Igen er alle testene positive, hvilket vil sige at disse virker.



I denne test, tester vi vores RaavareBatchDAO. Her bliver der blot testet om vi kan hente en arraylist med raavarebatches og om hvorvidt vi kan oprette en ny raavarebatch. Testene forløb igen som forventet, som kan ses da de alle er positive.



Den sidste test er hvor vi tester vores RaavareDAO. Her tester vi oprettelsen og redigeringen af råvarer samt om de kan hives ud som en arraylist. Dette gik også som det skulle, og alle testene blev positive igen.

Konklusion

Vi har i denne rapport fået lavet et softwaresystem, som kan bruges til afvejning, og som har fået en pæn brugergrænseflade. Som ønsket er der implementeret at man på brugergrænsefladen har forskellige muligheder, alt efter hvem man er logget ind som, og så har vi tilføjet at alle brugere også kan skifte deres password, hvis dette ønskes. Når en aktør skal foretage en afvejning, er der blevet lavet en ASE, som vores kommunikation med vægten foregår igennem. Der er også blevet implementeret en database, som gemmer på alt vores data. Denne database ligger online, så alle har de samme informationer i databasen uanset hvilken maskine programmet kører på.
Alt i alt har vi fået implementeret det meste, som var påkrævet i opgaven, dog med enkelte mangler, som der kan læses om i vores "manglende implementation" afsnit.
Vi har derudover også testet alle vores DAO klasser, som står for alle vores database kald, og sikret os at dette virker helt optimalt, så der ikke vil forekomme fejl, når der bliver navigeret rundt på vores hjemmeside.

Bilag

Git

Projektet: For at få fat i vores projekt, kan man importere det direkte til Eclipse vha. linket https://github.com/Chaaaaap/19_final.git. Dette gøres ved at trykke File → Import → Git → Projects som Git → Clone URI → Indtaster linket i URI-feltet → Next → Finish. Når dette er gjort henter Eclipse selv hele projektet ned. Når først man har hentet projektet ned til Eclipse, henviser vi til opsætning af databasen i punkt 2, før man fortsætter. Når databasen er korrekt opsat, skal man blot højreklikke på projektet → Run As → Web Application(GWT Super Dev Mode). Når dette er gjort, compiler (behandler) Eclipse projektet, hvorefter der vises et link under fanen "Development Mode". Dette link kopieres og sættes ind i din internet browser.

Databasen

For at kunne bruge vores system, kræver det at man benytter den rigtige database. Der er to databaser vi kan bruge til vores projekt, enten en online ellers en lokal.
Hvis vi skal bruge den online database, så skal i ændre i Connector klassen, så den ser ud som på billedet herunder: (sådan er den normalt, så hvis i har internetforbindelse, så har i forbindelse til databasen og så virker programmet.)

```
public class Connector {
    private final String HOST     = "ec2-52-30-89-247.eu-west-1.compute.amazonaws.com";
    private final int    PORT     = 3306;
    private final String DATABASE = "grp19";
    private final String USERNAME = "grp19";
    private final String PASSWORD = "hS#Vk94G";
    private Connection connection;
```

Hvis vi skal bruge en offline database, kræver det for det første at den er oprettet, og sat op som den skal. Vi har i vores projekt lavet lidt om i databasen end det script som vi fik udleveret fra "Pizza" databasen, og den har vi lagt op som to sql filer inde i mappen "war" i vores projekt (DDL grp19 og DML grp19).
For at kunne oprette den skal du gå ind under Database Development, og drag and drop database.sql filen og udfylde din Connection Profile. Derefter skal du execute all, ved at højreklikke på skærmen og vælge "Execute all", dernæst er databasen oprettet og sat op som ønsket.
Den sidste ting du skal være opmærksom på, før du kan connecte til databasen er, at gå ind i vores Connector.java klasse og ændre "password" til dit password, hvis du bruger et, for at connecte til mysql.

Så det skulle komme til at se ud som på billedet nedenunder, bare med dit password.
Derefter har du adgang til den lokale database, som også kan bruges i vores system.

```
public class Connector {
    private final String HOST     = "localhost";
    private final int    PORT     = 3306;
    private final String DATABASE = "19_CDIOFinal";
    private final String USERNAME = "root";
    private final String PASSWORD = "";
    private Connection connection;
```

Systemkrav

Vores web applikation kan køres på alle maskiner der som minimum kører med Windows XP. Det eneste systemkrav der er, er at man skal have installeret JAVA 1.8, for at kunne bruge det library som vi har anvendt (jre 1.8.0_60). Man kan dog godt bruge en nyere udgave, så skal man blot ind og ændre i, hvilke libraries der er tilknyttet projektet. Det er ikke noget større problem, men hvis man ikke føler sig sikker på hvordan man tilføjer / fjerner libraries vil vi ikke anbefale det.
Der skal til dette projekt også installeres GWT på computeren, såfremt du vil kører det igennem Eclipe, da det bliver brugt i vores implementering. Hvis den lokale database skal benyttes, skal der også sættes en connector op til denne, som kan gøres under Database development i Eclipse.

Logo til hjemmeside lånt fra dette billede:
http://exame1.abrilm.com.br/assets/images/2013/4/174161/size_590_brasilpharma.jpeg?136666695

Timeregnskab

Silas						
	Design	Implementering	Test	Dokumentation	Analyse	Total antal timer
03-06-2016	0	0	0	0	2	
06-06-2016	0	4	0	0	2	
07-06-2016	0	5	0	0	1	
08-06-2016	0	2	0	0	0	
09-06-2016	0	7	0	0	0	
10-06-2016	0	7	0	0	0	
11-06-2016	0	0	0	0	0	
12-06-2016	0	0	0	0	0	
13-06-2016	0	7	0	1	0	
14-06-2016	1	7	0	2	0	
15-06-2016	5	0	0	1	0	
16-06-2016	6	2	0	3	0	
17-06-2016	1	0	0	9	0	
i alt	13	41	0	16	5	75
Mikkel						
	Design	Implementering	Test	Dokumentation	Analyse	
03-06-2016	0	4	0	0	0	
06-06-2016	0	6	0	0	1	
07-06-2016	0	6	0	0	1	
08-06-2016	0	7	1	0	0	
09-06-2016	0	7	1	0	0	
10-06-2016	1	5	1	0	0	
11-06-2016	0	0	0	0	0	
12-06-2016	0	2	0	0	0	
13-06-2016	0	13	0	0	0	
14-06-2016	0	11	0	0	0	
15-06-2016	0	6	3	0	0	
16-06-2016	3	1	2	1	0	
17-06-2016	0	1	0	6	0	
i alt	4	69	8	7	2	90

Martin						
	Design	Implementering	Test	Dokumentation	Analyse	
03-06-2016	0	1	0	0	2	
06-06-2016	1	1	0	0	3	
07-06-2016	0	4	0	0	1	
08-06-2016	0	4	0	0	0	
09-06-2016	0	5	0	0	0	
10-06-2016	0	7	0	0	0	
11-06-2016	0	0	0	0	0	
12-06-2016	0	0	0	0	0	
13-06-2016	0	9	0	0	0	
14-06-2016	0	10	0	0	0	
15-06-2016	0	6	0	5	1	
16-06-2016	2	3	0	8	3	
17-06-2016	0	0	0	6	0	
i alt	3	50	0	19	10	82
Frank						
	Design	Implementering	Test	Dokumentation	Analyse	
03-06-2016	0	1	0	0	2	
06-06-2016	0	1	0	0	3	
07-06-2016	0	3	0	0	1	
08-06-2016	0	3	0	0	2	
09-06-2016	0	7	0	0	1	
10-06-2016	0	7	0	0	1	
11-06-2016	0	0	0	0	0	
12-06-2016	0	0	0	0	0	
13-06-2016	0	7	0	0	0	
14-06-2016	0	10	0	0	0	
15-06-2016	0	5	0	3	1	
16-06-2016	0	3	0	6	2	
17-06-2016	0	0	0	3	0	
i alt	0	47	0	12	13	72

Ramyar						
	Design	Implementering	Test	Dokumentation	Analyse	
03-06-2016	0	0	0	0	1	
06-06-2016	0	2	0	0	1	
07-06-2016	0	5	0	0	1	
08-06-2016	0	1	0	0	0	
09-06-2016	0	5	0	0	0	
10-06-2016	0	5	0	0	0	
11-06-2016	0	1	1	0	0	
12-06-2016	0	6	1	0	1	
13-06-2016	0	6	1	0	1	
14-06-2016	0	8	0	0	0	
15-06-2016	2	3	0	0	2	
16-06-2016	6	0	0	1	3	
17-06-2016	0	0	0	0	0	
i alt	8	42	3	1	10	64
Total antal timer brugt for alle deltagere						383