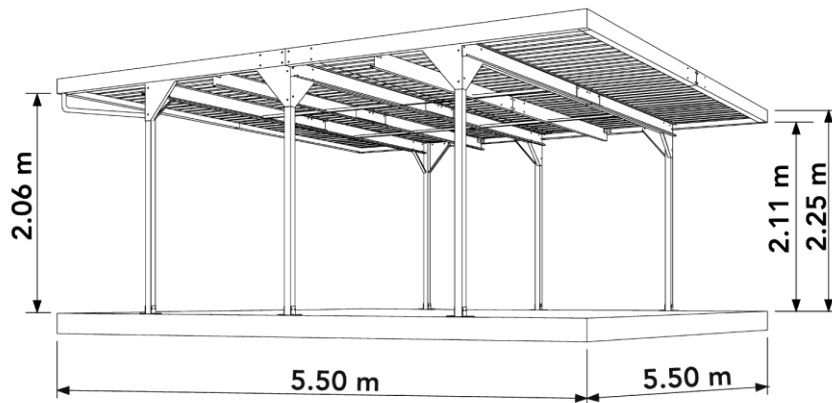


# Fog®



Af Betül Iskender,  
Ye Long He,  
Morten Bendeke  
og  
Denis Altin Pedersen

<https://github.com/coerth/carport>

<https://youtu.be/5VB7cyXHrKs>

<http://159.223.18.9:8080/carport/index.jsp>

# Indhold

Indledning.....	2
Baggrund .....	3
Virksomheden/Forretningsforståelse .....	4
User-stories og kravspecifikation .....	7
Mock-up.....	9
Bootstrap & CSS.....	10
EER-diagram .....	10
Domænemodel .....	14
Klassediagram.....	16
Aktivitetsdiagram .....	18
As-is .....	18
To-be .....	19
Is-now.....	20
Navigationsdiagram .....	22
Sekvensdiagram .....	23
Valg af arkitektur.....	25
Udvalgte kodeeksempler .....	26
Særlige forhold.....	27
Valg af scopes .....	27
Sikkerhed.....	28
Fejlhåndtering, logning og exceptions .....	28
SVG .....	28
Kodestandarder .....	29
Status på implementation .....	29
CRUD-operationer .....	31
Arbejdsprocessen.....	32
Gruppearbejde.....	32
Logbog.....	32
Github projects .....	33
Git .....	34
Discord .....	35
Hvad kan vi tage med til næste projekt? .....	38
Bilag.....	38
Sekvensdiagram for <i>login</i> :.....	40
Tænke-højt-test .....	42
Logbog, regneregler m.m.:.....	42

# Indledning

Denne opgave tager udgangspunkt i afslutningsprojektet på Datamatikeruddannelsens 2. semester.

Formålet med opgaven er at demonstrere forskellige værktøjer der er blevet præsenteret i løbet af semestret og hvordan disse værktøjer benyttes.

De værktøjer vi blandt andet har arbejdet med er *front-controller*, *database-styring* via *MySQL* samt brug af *HTML (Hyper Text Markup Language)*, *Java* og *JSP(Java Server Page)*. Derudover fik vi også en introduktion til *SVG (Scalable Vector Graphics)* der skulle demonstrere det grafiske element i projektet. Vi har også arbejdet med virksomheds- og forretningsforståelse, herunder SWOT-analyser af Fog, som kom os til gavn under hele projektets forløb. Det gjorde at vi ikke kun kiggede på opgaven med ”kode-øjne”, men også kiggede på opgaven med en mere kundeorienteret vinkel.

Selve projektet, består i en bestillingsside for Fogs Trælasthandel, hvorfra deres kunder kan bestille enten en standard carport med faste mål, eller en carport ud fra egne valgte mål. Siden skal være funktionel i forhold til at kunne beregne en styklister til kunden når bestillingen er godkendt, kunne vise en grafisk SVG-tegning der viser carporten, og beregne en eventuel pris.

Vi gik til opgaven med et ønske om at opfylde MVP (*Minimum Viable Product*). At kunne lave et produkt der viser de minimum funktionelle krav fra kundens (Fogs) side. Fokus herunder var først og fremmest på *backend-delen* og at sørge for at de funktionelle krav var opfyldt, inden vi gik videre og kiggede på *frontend-delen*.

# Baggrund

Firmaet Johannes Fog blev grundlagt i 1920 og består af et Bolig og Designhus samt forskellige Trælast- og byggecentre på Sjælland. Fogs Trælasthandel specialiserer sig på forskellige områder, men specielt deres rådgivning samt skræddersyede løsninger på store og små byggeprojekter er deres varemærke<sup>1</sup>.

Vi er blevet stillet til opgave, at udarbejde et nyt bestillingssystem til carporte ud fra kundens egne mål. Det system Fog har på nuværende tidspunkt er et forældet program som er afhængigt af, at medarbejderne hos Fog enten er i telefonisk eller personlig kontakt med kunden under hele bestillingsprocessen. Som det fungerer nu er bestillingen af carporte ud fra kundens egne ønskede mål besværliggjort både for kunden og for medarbejderne selv. Kunden kan bestille via hjemmesiden ud fra ønskede mål, men herefter skal en medarbejder fra Fog selv manuelt ind og kigge den enkelte ordre igennem for eventuelle fejl eller mangler. Når ordren er gennemgået og godkendt, vil kunden herefter modtage et tilbud med pris på den forespurgte carport. Før ordren kan gennemføres, skal kunden se tilbuddet igennem og betale. Herefter vil en bekræftelse af forespørgslen og en tegning af carporten, blive sendt til kunden. Fog vil så udarbejde selve styklisten for carporten via deres forældede program M3-Build, som ikke længere kan håndtere opdateringer. Det betyder også at nye varer ikke kan tilføjes og gamle varer heller ikke kan fjernes. Oveni det skal Fog i mange tilfælde lave en-til-en sammenligning med en anden stykliste beregner som de har liggende i papirform. Fog har også på nuværende tidspunkt besværliggjort deres Byg-Selv mulighed ved at gemme funktionen godt væk under en masse undermenuer på hjemmesiden.

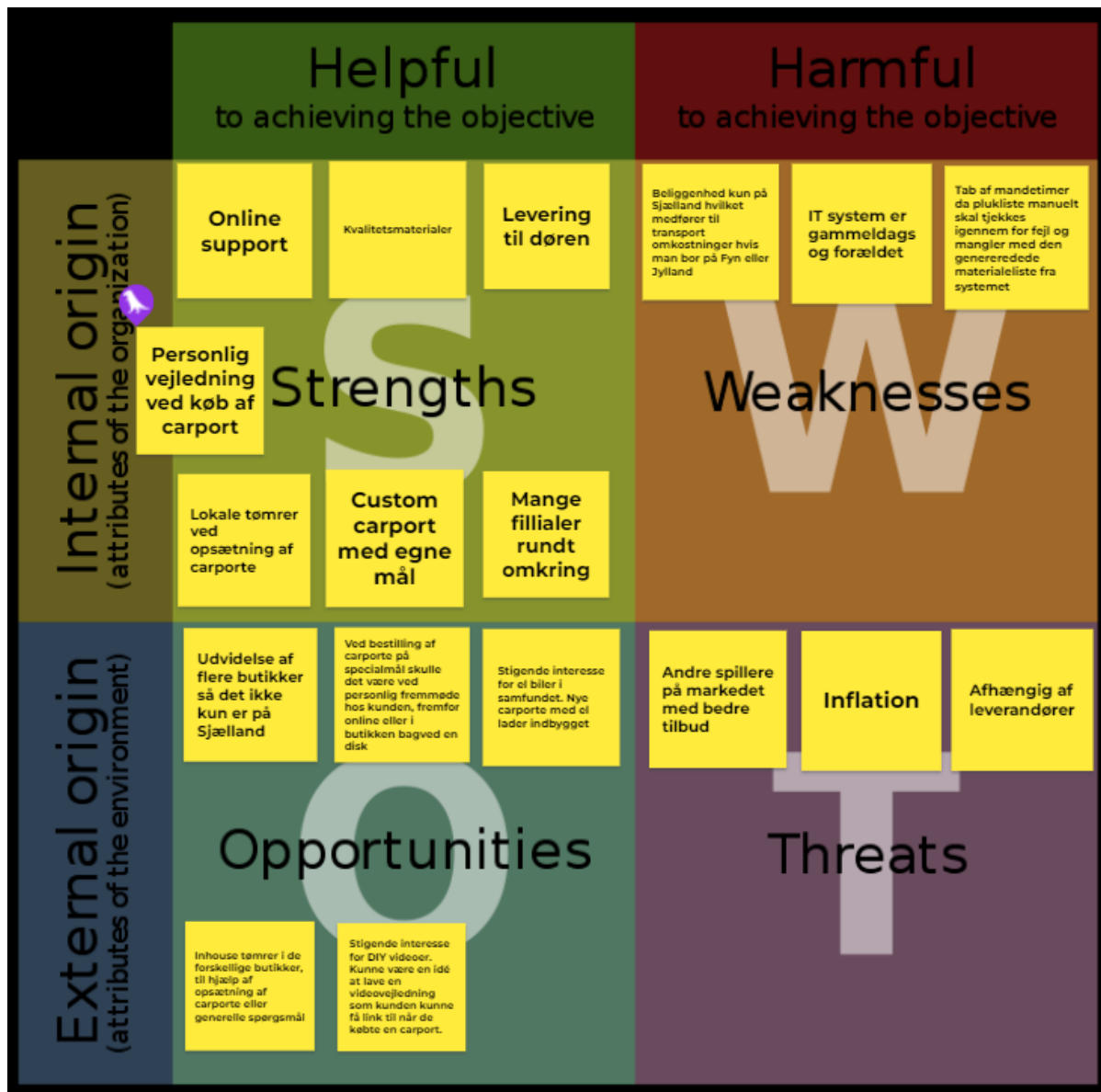
Fog ønsker i stedet, at processen bliver gjort mere strømlinet og nemmere. Kunden skal kunne bestille sin carport ud fra indtastede mål på hjemmesiden, og selvom medarbejderen stadigvæk skal se forespørgslen igennem førend ordren gennemføres, skal kunden allerede på dette tidspunkt kunne se en ca. pris.

Udregningerne af det nødvendige materiale skal også automatiseres, så Fog ikke selv skal ind og finde informationerne og rette til på M3-Build. Ydermere ønsker Fog også, at de fremadrettet via hjemmesiden kan se og acceptere forespørgslerne, som så sender en ordrebekræftelse ud til kunden med de relevante oplysninger. Der kan herefter betales og når betalingen er gennemført, får kunden automatisk en stykliste og Fog kan herefter se og pakke de relevante materialer og få dem leveret til kunden.

---

<sup>1</sup> <https://www.johannesfog.dk/byggecenter/om-fog/>

# Virksomheden/Forretningsforståelse



Inden projektet startede, lavede vi en SWOT-analyse af Fog, for at få en forståelse af virksomhedens styrker og svagheder. Dette gjorde vi, for bedre at kunne tilrettelægge et produkt, der opfylder deres krav.

Vi følte det var nødvendigt at finde ud af hvad Fog var dygtige og knap så dygtige til og hvilke muligheder de havde for at optimere virksomheden. Ud fra SWOT-analysen kan vi se at Fog skiller sig ud fra andre spillere på markedet, i form af deres ekspertise på området som gør dem i stand til at tilrettelægge carporte ud fra kundens behov og samtidig give kompetent vejledning heraf. SWOT-analysen fremhæver udover Fogs stærke sider, også de svagheder som gør dem mindre attraktive på markedet. Ud fra det har vi taget

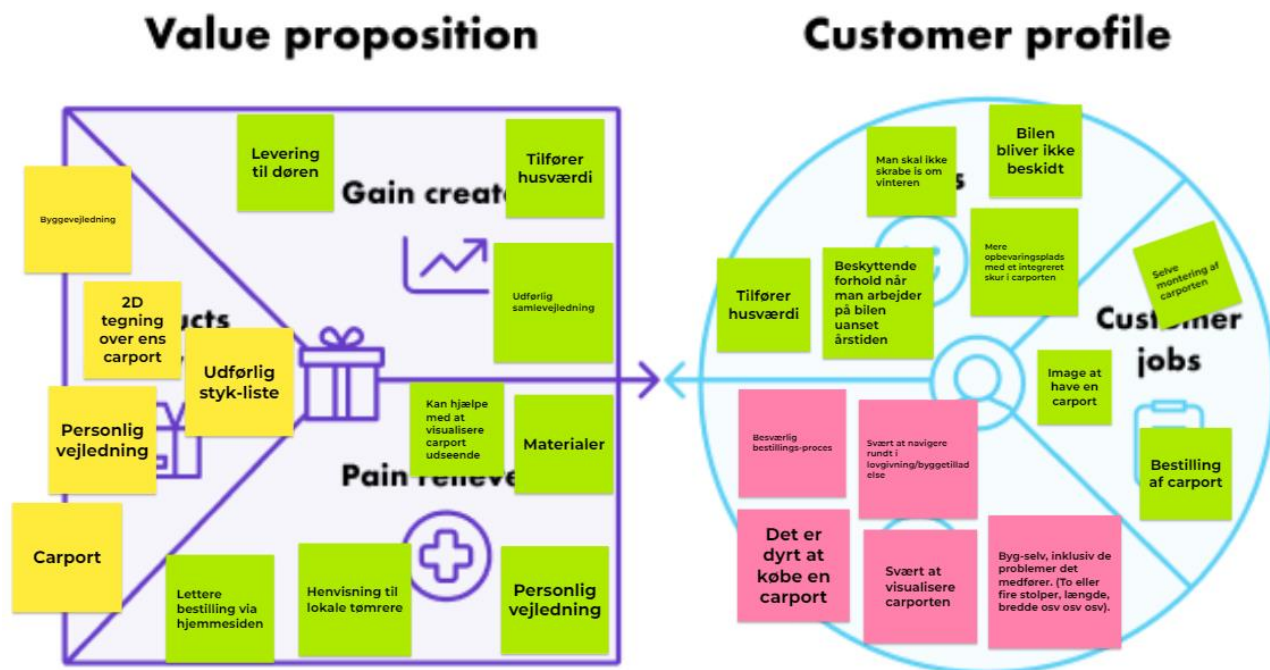
Betül Iskender  
[cph-bi32@cph-business.dk](mailto:cph-bi32@cph-business.dk)  
dat-0921b

Ye Long He  
[cph-yh24@cph-business.dk](mailto:cph-yh24@cph-business.dk)  
dat-0921b

Morten Bendeke  
[cph-mb809@cphbusiness.dk](mailto:cph-mb809@cphbusiness.dk)  
dat-0921b

Denis Pedersen  
[cph-dp@cphbusiness.dk](mailto:cph-dp@cphbusiness.dk)  
dat-0921b

udgangspunkt i en af deres største svagheder som er deres forældede IT-system. Denne håndterer alle kunders forespørgsler på carporte, samt deres forældede system til plukning af materialer til stykliste, hvilket medfører til tab af arbejdstimer, da medarbejderne hele tiden skal dobbelttjekke den genererede stykliste med deres egen plukliste.



*Value Proposition Canvas* (kaldt *VPC* fremadrettet) er et værktøj opfundet af Alexander Osterwalder, som bruges hyppigt af firmaer, for at se om der er en relation mellem det valgte produkt og markedet. Når man snakker om *VPC* snakker man om henholdsvis to ting: kundesegmentet og value-proposition. Tager vi udgangspunkt i Fog kan det inddeles således:

*Customer profile* (kundeprofil) bliver inddelt i tre kategorier henholdsvis *Gains*, *Pains* og *Customer jobs*.

*Gains*: Hvad kunden får ud af at købe en carport gennem Fog.

*Pains*: Hvilke problematikker kan der opstå ved køb af carport gennem Fog.

*Customer jobs*: Hvad kræver det af kunden hvis kunden gerne vil have en carport hos Fog.

Ligeledes kan Value Proposition inddeles i tre kategorier:

*Gain creators*: Hvordan produktet fra Fog kan tilføre værdi til kunden.

*Pain relievers*: Hvordan produktet kan afhjælpe kundens problemer.

*Product and Services*: Hvilke produkter og services Fog har, som kan afhjælpe/løse kundens "jobs".

Vi brugte vores *SWOT*-analyse til udarbejdelsen af vores *VPC* af Fog, hvor vi har taget udgangspunkt i nogle problematikker vi finder interessante. Vi kunne konstatere hvordan personlig vejledning ville løse kundens udfordringer ved bestilling af en ny carport. En styrke som Fog giver til kunden er blandt andet en udførlig samlevejledning og stykliste til kunden. Dette valgte vi at have fokus på samt det at bestille carporte gennem Fogs hjemmeside skulle gøres mere strømlinet og mere enkel for kunden.

På baggrund af analysen lavede vi et pitch for Fog: [https://www.youtube.com/watch?v=h\\_NAbU8pr-E](https://www.youtube.com/watch?v=h_NAbU8pr-E)

## Teknologivalg

Under processen har vi benyttet følgende programmer:

- *Intellij IDEA 2021.3.2* → Til arbejdet med Java, HTML, JSP, CSS, JDBC
- *JAVA SDK 11.0.14*
- *Bootstrap 5.0* → Til arbejdet med CSS og styling
- *MySQL Workbench 8.0 CE* → Til databasen
- *Github Projects* → Som kanban under gruppearbejdet
- *Google Docs* → Til logbog og noter
- *Git Bash* → Til arbejdet på projektet
- *Tomcat Apache version 9.0.601* → Til at håndtere vores program/hjemmeside lokalt
- *Draw.io* → Til mockups
- *Plant UML* → Til diagrammer
- *JUnit* → Til testing
- *DigitalOcean* → En digital service til at køre vores program/hjemmeside på en server

# User-stories og kravspecifikation

Ud fra interviewet med Fogs medarbejder (Martin) samt de opfølgende vejledende samtaler med deres udsendte (Jörg, Jon og Nikolaj), lavede vi følgende *user-stories*:

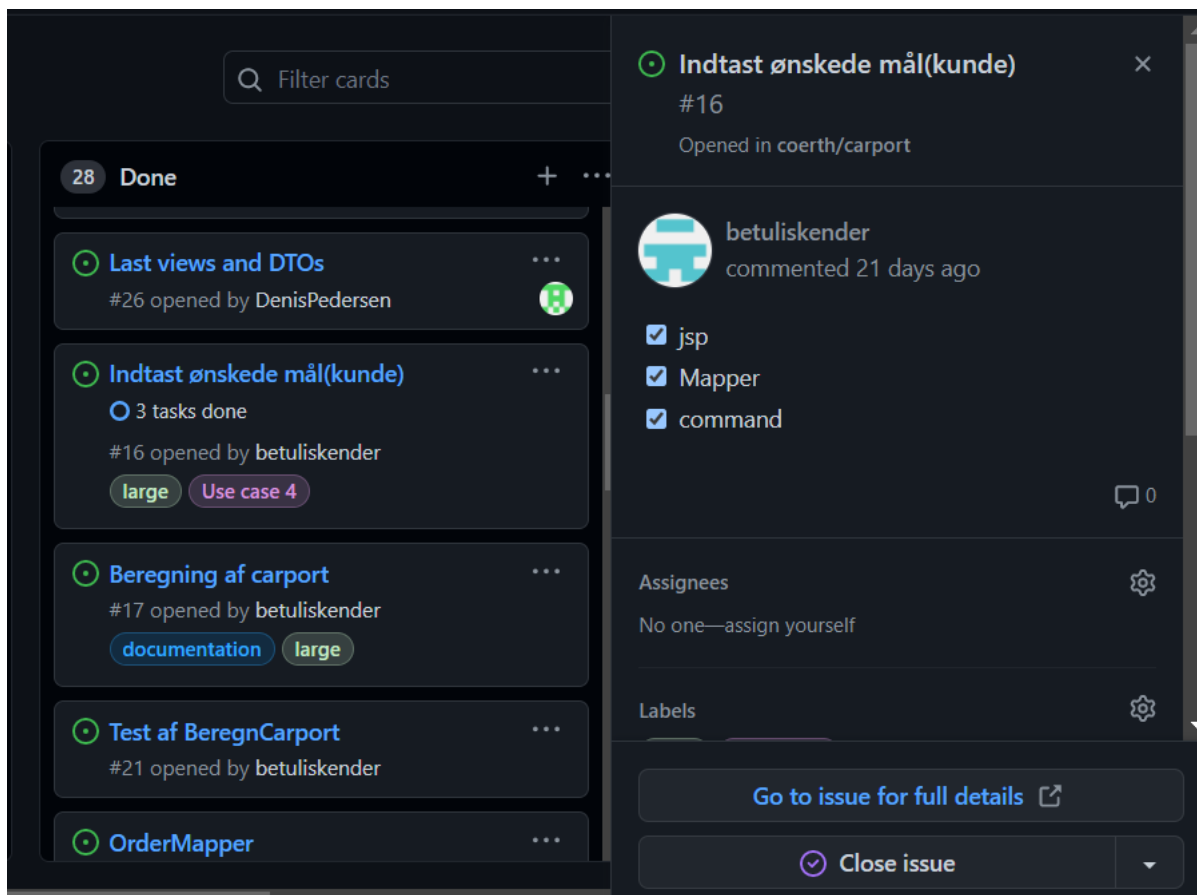
- Kunden kan indtaste ønskede mål på carport i en form.
- Admin kan ændre priser og opdatere varelager.
- Admin kan se hvilke mål kunden har indtastet til sin carport og tjekke for eventuelle fejl.
- Admin kan godkende ordre og sende styklister og tegning til kunden.

Ovenfor disse *user-stories*, fik vi tilføjet yderligere krav, som vi håbede ville give værdi til både kunden og medarbejderen hos Fog.

- Kunden skal kunne oprette en bruger.
- Kunden kan, efter at have oprettet en bruger, logge på med sin indtastede e-mail og kodeord.
- Kunden kan se egen ordrehistorik, forespørgsler og status på disse.
- Kunden kan ændre på sine oplysninger.
- Admin kan se alle ordrer og deres status.

Da nogle af kravene har været mere krævende end andre, har vi i *Github Projects* inddelt vores *user-stories* i forskellige kategorier som igen er opdelt efter opgavens størrelse (*small*, *medium* og *large*). De tre størrelser er også et tidsestimat for hvor længe man ca. skal bruge pr. opgave. Her har vi inddelt således, at opgaver markeret med *small* er ca. 1-2 timer, mens *medium* er mellem 3-6 timer og *large* er en til flere dage.





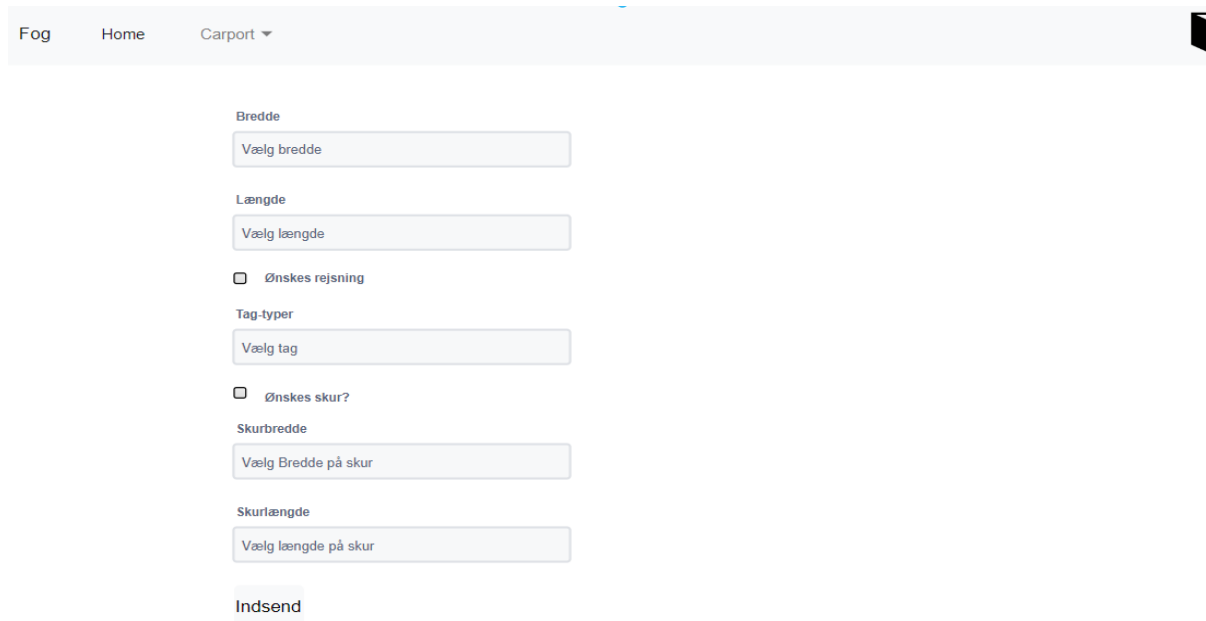
Vi har *acceptkriterier*<sup>2</sup> på vores *user-stories* som hjælper med at se hvorvidt det forventede resultat er opnået eller ej. Vores acceptkriterier er forsøgt lavet så kunden (Fog) forstår hvorvidt vi har opfyldt de stillede krav i den pågældende user-story. Acceptkriteriet til “Admin kan se hvilke mål kunden har indtastet til sin carport og tjekke for eventuelle fejl.” ser således ud:

*Hvis jeg er logget på som admin, forventer jeg, efter at have valgt “vis alle forespørgsler”, og derefter klikke på “vis” ud fra den valgte forespørgsel, at kunne se kundens indtastede mål.*

<sup>2</sup> Se [bilag](#) for fuld liste over acceptkriterier.

# Mock-up

Efter interviewet med Fog-medarbejderen, Martin, gik vi i gang med at lave et simpelt mock-up. Vores tanke var at lave et så simpelt produkt som overhovedet muligt. Kunden skulle vælge sin carport ud fra en form med nogle opstillede kriterier, som f.eks. længde, bredde, tagtype og om man vil have med- eller uden skur. Vores endelige produkt adskilte sig fra nogle af vores tidligere ideer (bl.a. lavede vi en *kundeprofil-side*), men det grundlæggende design-valg vi lagde ud med, holdt vand hele vejen.



The screenshot shows a web interface for selecting a carport. At the top, there is a navigation bar with links for 'Fog', 'Home', and 'Carport'. Below this, the form is organized into several sections: 'Bredde' (Width) with a dropdown menu 'Vælg bredde'; 'Længde' (Length) with a dropdown menu 'Vælg længde'; a checkbox for 'Ønskes rejsning' (Desired ramp); 'Tag-typer' (Roof types) with a dropdown menu 'Vælg tag'; another checkbox for 'Ønskes skur?' (Desired shed); 'Skurbredde' (Shed width) with a dropdown menu 'Vælg Bredde på skur'; 'Skurlængde' (Shed length) with a dropdown menu 'Vælg længde på skur'; and finally an 'Indsend' (Submit) button.

Billedet viser en af vores tidlige mock-up-designs for en forespørgselsform på en carport. Tidligt i forløbet havde vi snakket om hvilke kriterier formen skulle have. Hvis man f.eks. vil have et tag med rejsning skulle det være en mulighed, og hvis man valgte den mulighed, skulle man også kunne vælge taghældning og forskellige typer tagmateriale. Vi valgte dog i stedet at gøre det simpelt, så man kun kan vælge med fladt tag og kun én type tagmateriale (Fog har kun én type fladt tag). Ydermere gjorde vi det simple for kunden ved kun at vise skur-muligheder, hvis man har tilvalgt det.

## Byg selv carport

Vælg bredde (min 240 cm og max 600 cm:)

Vælg bredde ▾

Vælg længde (min 240 cm og max 780 cm:)

Vælg længde ▾

Vælg tag-type:

Vælg tag-type ▾

Ønskes skur? Klik her ☐

Vælg

Bootstrap & CSS:

Til styling af vores hjemmeside har vi gjort brug af Bootstrap og en smule CSS. Bootstrap bruger vi, da man via Bootstrap kan skalere hjemmeside-vinduet til at passe forskellige typer skærme (mobil, tablet etc.) Da backend-delen var første prioritet valgte vi derfor CSS fra, da det er mere tidskrævende i forhold til Bootstrap. Bootstrap har været nemmere for os at gå til, da der allerede findes færdig udformede templates man kan rette til.

## EER-diagram

Vores database-design endte i tre versioner. Vores første udkast, var meget simpelt. Vi valgte ikke at bruge 3. normalform på adresseoplysningerne i *customer*, da vi ikke mente det ville gavne funktionaliteten i programmet og det betød også, at vi skulle *hardcode* en stor del af postnumrene, hvilket ville tage for lang tid i forhold til projektets størrelse.

På vores EER-diagram kan man se, at vi har brugt 1-1 relation mellem *account* og *customer*. Dette gør vi da vi siger, at en *customer* kun kan have én *account*. Vi valgte, at separere *account* og *customer*, da vi så et behov for, at Fogs admin-konti bør kunne eksistere uden tilknytning til *customer*.

Tredje normalform anvendes på *description*, da én *description* bliver brugt flere steder. Det samme er gældende for *material* og *material-type*. Som det kan ses på billedet var relationerne ikke lavet korrekt, da *orderline* har en *description* men en *description* kan være i flere *orderlines* og det samme med *material* og *material-type*.

Betül Iskender

[cph-bi32@cph-business.dk](mailto:cph-bi32@cph-business.dk)  
dat-0921b

Ye Long He

[cph-yh24@cph-business.dk](mailto:cph-yh24@cph-business.dk)  
dat-0921b

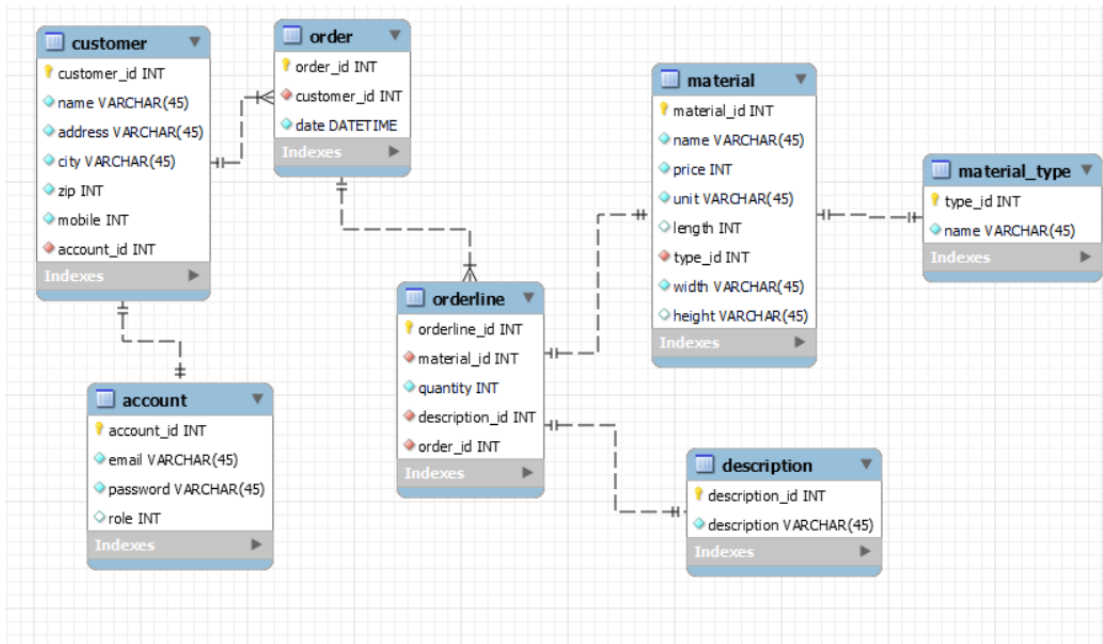
Morten Bendeke

[cph-mb809@cphbusiness.dk](mailto:cph-mb809@cphbusiness.dk)  
dat-0921b

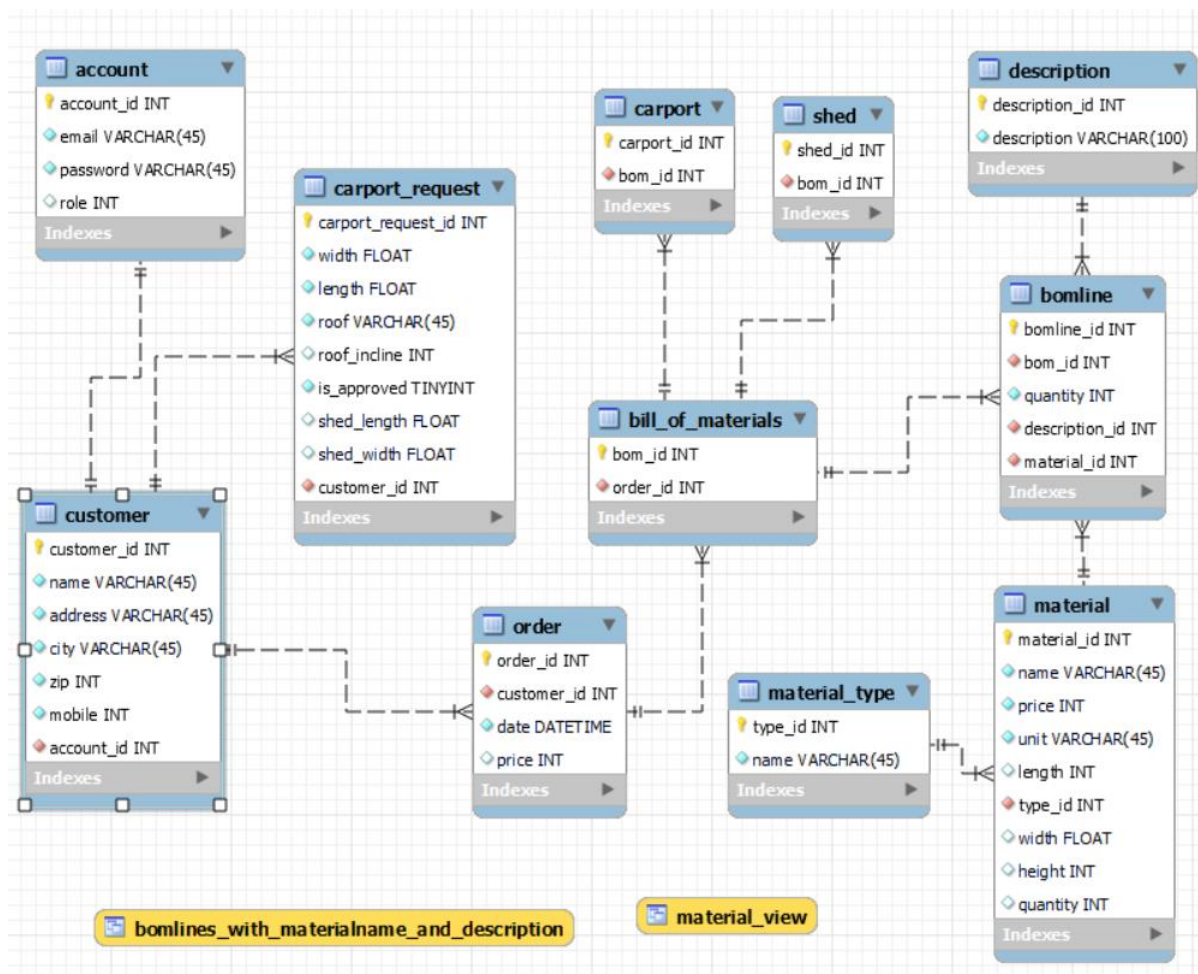
Denis Pedersen

[cph-dp@cphbusiness.dk](mailto:cph-dp@cphbusiness.dk)  
dat-0921b

I *orderline* får vi fremmednøgler ind fra *order*, *material* og *description*, det sker så vi kan få id'et med ind fra de tabeller, da de skal bruges til at få informationerne fra de pågældende steder. Da *order* indeholder et *customer-id* som fremmednøgle, kan vi trække kundeoplysninger direkte fra *order*, hvilket betyder vi kan se hvilken ordre der tilhører hvilken kunde.



Efter feedback fra Fogs repræsentant (Jörg), valgte vi at gå tilbage til database design-fasen og udtænke en større database, med fokus på 3. normalform. På dette stadie i vores design besluttede vi, at en carport principielt var en styklister, ligesom et skur også bestod af en række materialer. En ordre bestod så af 1-2 styklister som indeholdt materialerne til en carport og eventuelt et skur. Vi ville forbinde de styklister til enten et *carport-id* eller *skur-id* for at differentiere mellem de forskellige styklister. De forskellige styklister bestod så af linjer som vi skulle sammensætte i *bomlines* (tidligere *orderlines*). *Bomlines* ville bestå af et *description-id* og et *material-id* da de begge skulle anvendes på flere linjer.

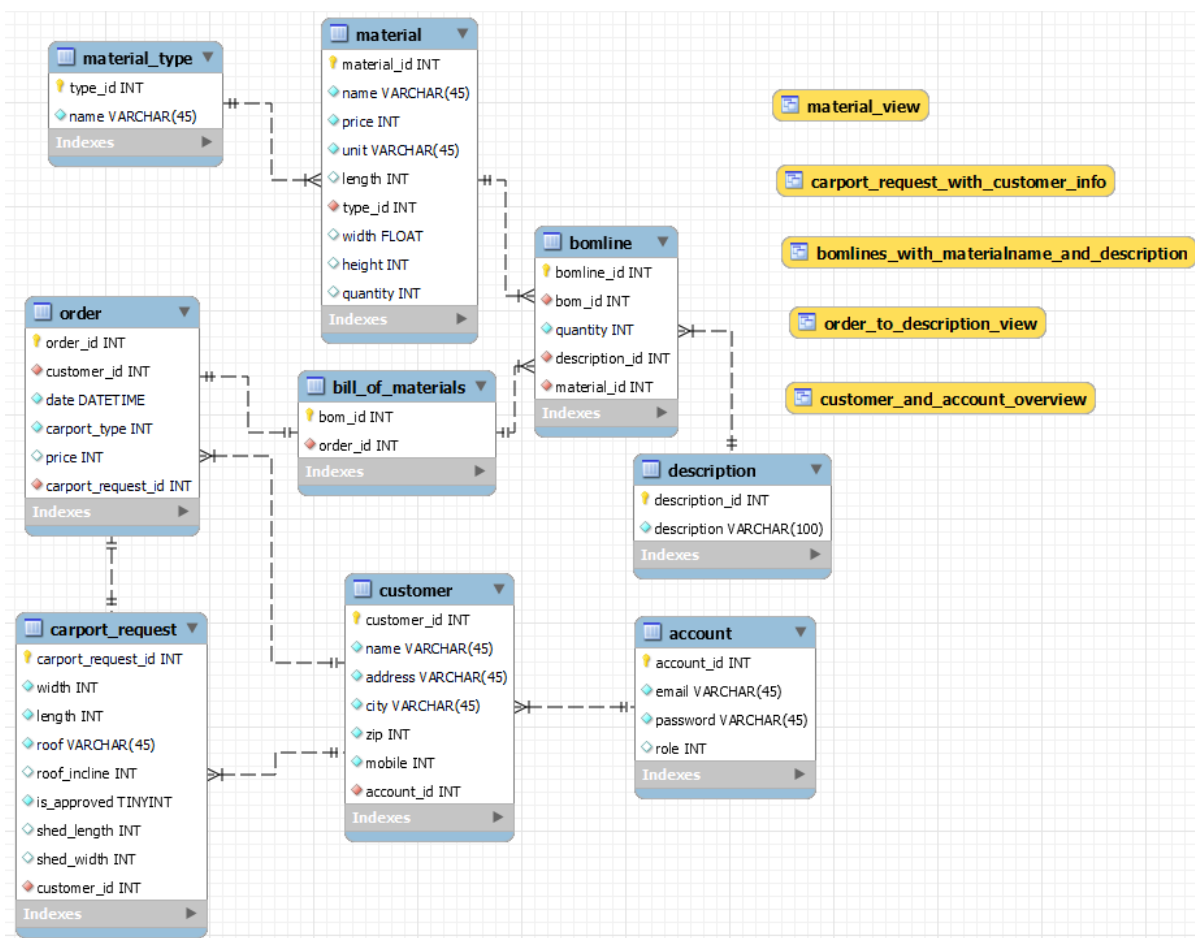


Undervejs i kodningen blev vi ved med at vende tilbage til EER-diagrammet og endte derfor med at ændre på den en sidste gang. I vores endelige udgave valgte vi at skære carport- og skur-tabellerne fra, da vi ikke så nogen merværdi ved at benytte 3. normalform her. En stykliste kan derfor indeholde en carport, eller en carport *med* skur.

Vi benyttede os i høj grad af *views*, for at få samlet relevante tabeller sammen. Dette gjorde vi for bedre at få overblik og for at gøre vores *datamappere* mere overskuelige ved ikke at bruge alt for mange *joins*.

Ud fra billedet ses det at vores *account* og *customer* stadig har 1-1 relation, derudover har vi også fået en *order* og *Bill Of Materials (BOM)* som har 1-1 relation, da én *order* indeholder én *BOM* og én *BOM* tilhører én *order* og det samme med *carport\_request* og *order*. Det vi siger er, at man ikke kan have en *order*, uden at have en *carport\_request*, men en *carport\_request* bliver ikke nødvendigvis til en ordre, hvis en kunde f.eks. ikke vælger at gennemføre sit køb.

De fleste tabeller indeholder fremmednøgler, da de skal bruge informationerne fra hinanden.



# Domænemodel

Billedet<sup>3</sup> viser vores første udkast af en domænemodel som er baseret på første udkast af vores EER-diagram. Modellen var på daværende tidspunkt meget simpel og enkel i sin form, da vi endnu var tidligt i forløbet og ikke havde diskuteret hvilke entiteter vi ville have med.

Vi vendte hvorvidt en *account* havde en *customer/admin* eller den anden vej rundt. Vi endte med at arbejde ud fra idéen om, at en *account* havde enten admin-rolle eller customer-rolle i sig. Således er *customer* en forlængelse af *account*. Tager man udgangspunkt i *customer*, så siger vi ligeledes at en *customer* har en ordre og ordren har ordrelinjer som indeholder materialer.

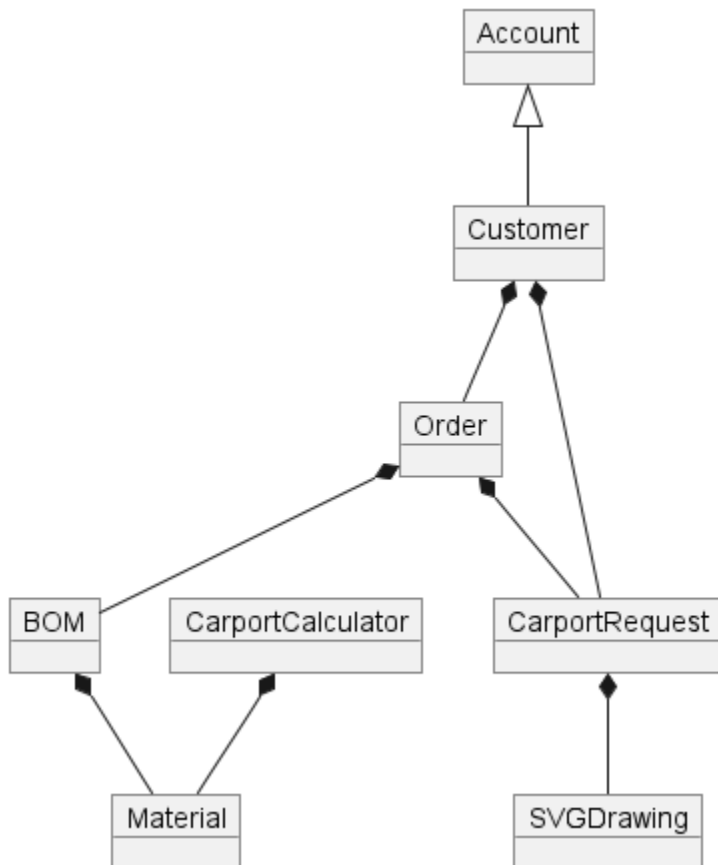
Undervejs i udviklingen opdagede vi behovet for flere entiteter, hvilket betød at vi implementerede *CarportCalculator* (som indeholder alle klasserne hvor udregningerne er), *CarportRequest* (forespørgsel), *SVG* (vores grafiske tegning) og *BOM* ( “*Bill of Material*” - den udregnede stykliste).

*CarportRequest* i sig selv er en entitet som ofte ender i en ordre men som nogle gange kun vil forblive en forespørgsel.

Her siger vi endnu engang, at en *customer* er en forlængelse af *account*, samt at *carportRequest* og *ordrer* har en kundetilknytning. Grunden til dette er, at vi ikke kan have en ordre, før der laves en forespørgsel, og når først en forespørgsel er lavet og godkendt, ryger den til en ordre. *CarportRequest* har en *SVGDrawing* i sig, da man ikke kan få en tegning, før man laver en forespørgsel. Vi siger herefter, at en ordre har *BOM* i sig, vores komplette stykliste, som indeholder materialer, og disse materialer ligeledes findes i *CarportCalculator*, som vi bruger til at udregne x antal materialer der skal bruges.

---

<sup>3</sup> Se [bilag](#), figur 1.

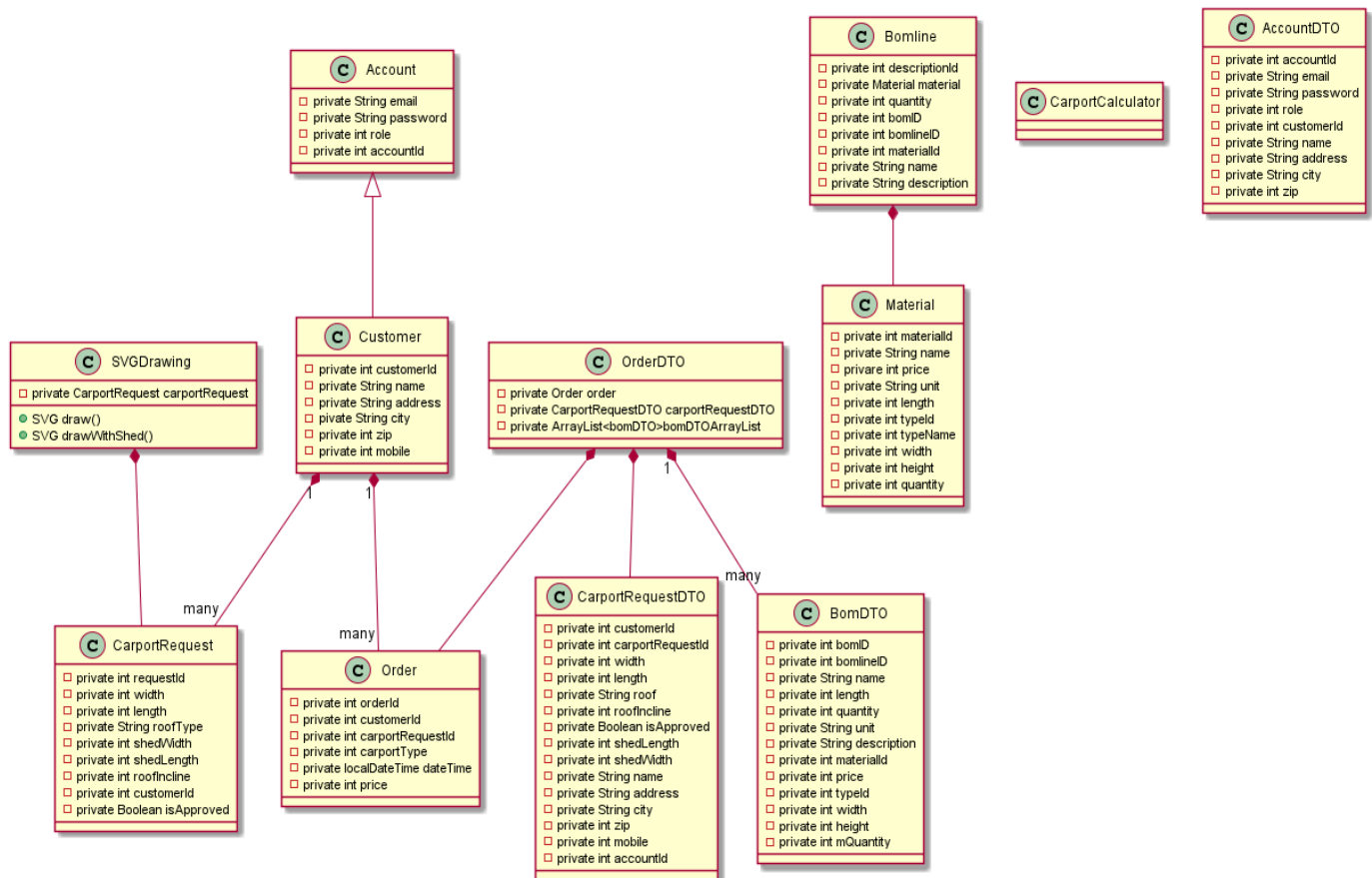




# Klassediagram

Vores første klasse-diagram<sup>4</sup> var, ligesom med vores domæne- og EER-diagram, meget simpelt i struktur. Man kan ud fra billedet se, de tidlige tanker vi gjorde os mht. hvilke attributter de forskellige klasser skulle indeholde. Ligesom med vores domæne- og EER-diagram, gennemgik vores klassediagram en tilsvarende udvikling. Vi oprettede en *CarportCalculator* klasse som håndterer overgangen fra forespørgsel til styklister.

Billedet viser version 2 af vores klassediagram. Som man kan se, er der kommet flere entiteter og nogle *DTO*'er. Vi benytter os af *DTO*'er når vi henter fra databasen. Det gør vi for at holde alle de relevante oplysninger, både til visning af ordre og hentning af styklister, da en stykliste består af en arrayliste af *bomlines*.

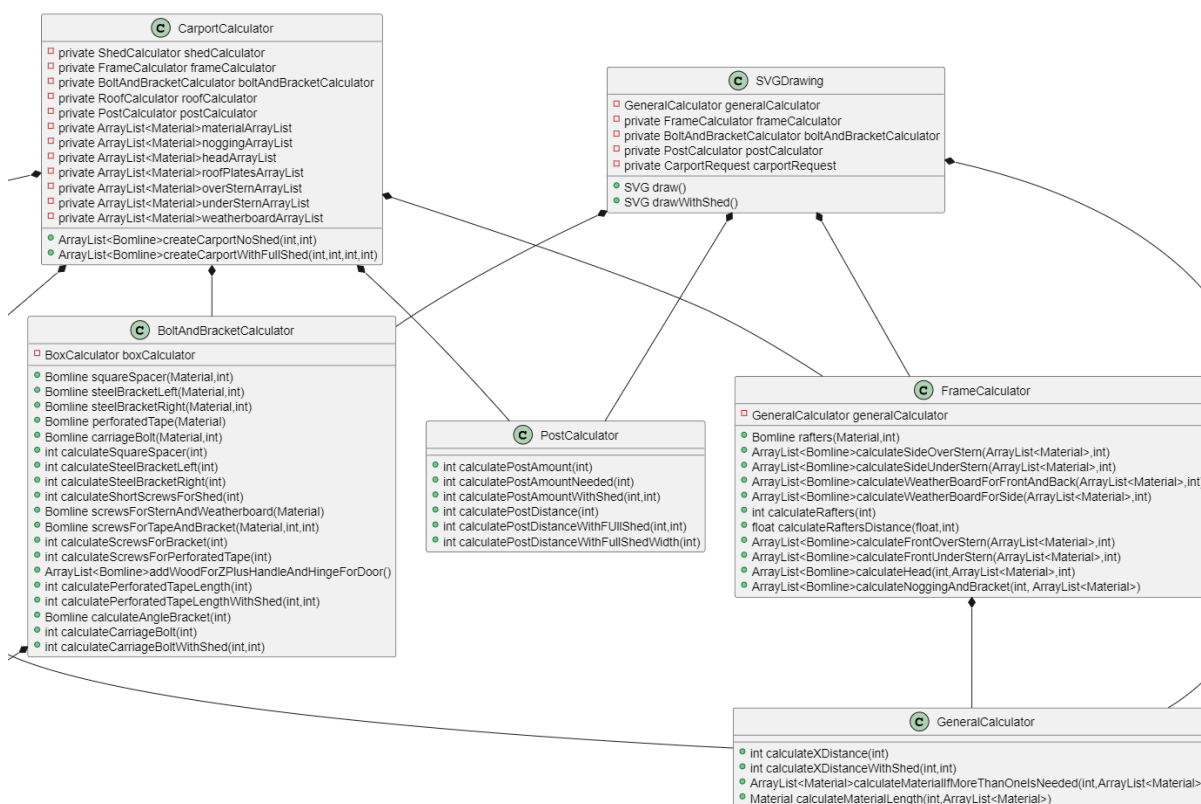


Efter mange overvejelser og diskussioner valgte vi at refaktorere vores klassediagram for sidste gang, da vi ikke følte at vores *CarportCalculator*-klasse ville stemme overens med SOLID-principperne. Vi valgte at dele vores *CarportCalculator* op i flere klasser, som hver især skulle have til opgave at beregne en specifik

<sup>4</sup> Se [bilag](#), figur 2.

del af carporten, som f.eks. carport-skelettet, taget eller stolper. Opdelingen i flere klasser, betyder også at vi fremadrettet kan lave ændringer eller tilføje mere til én del af koden, uden at forholde os til hvorvidt det vil påvirke andre dele af koden.

Billedet viser et udklip af vores endelige klassediagram, hvor man kan se at vi har refaktoreret *CarportCalculator*-klassen til mindre klasser. En af fordelene efter at have refaktoreret er, at vi ikke længere behøver at instantiere den store *CarportCalculator*-klasse i *SVGDrawing*-klassen, når vi skulle genere en tegning for kunden. Nu var det muligt blot at kalde de enkelte beregnere alt afhængig hvilken type carport der bliver indtastet i forespørgselsformen.

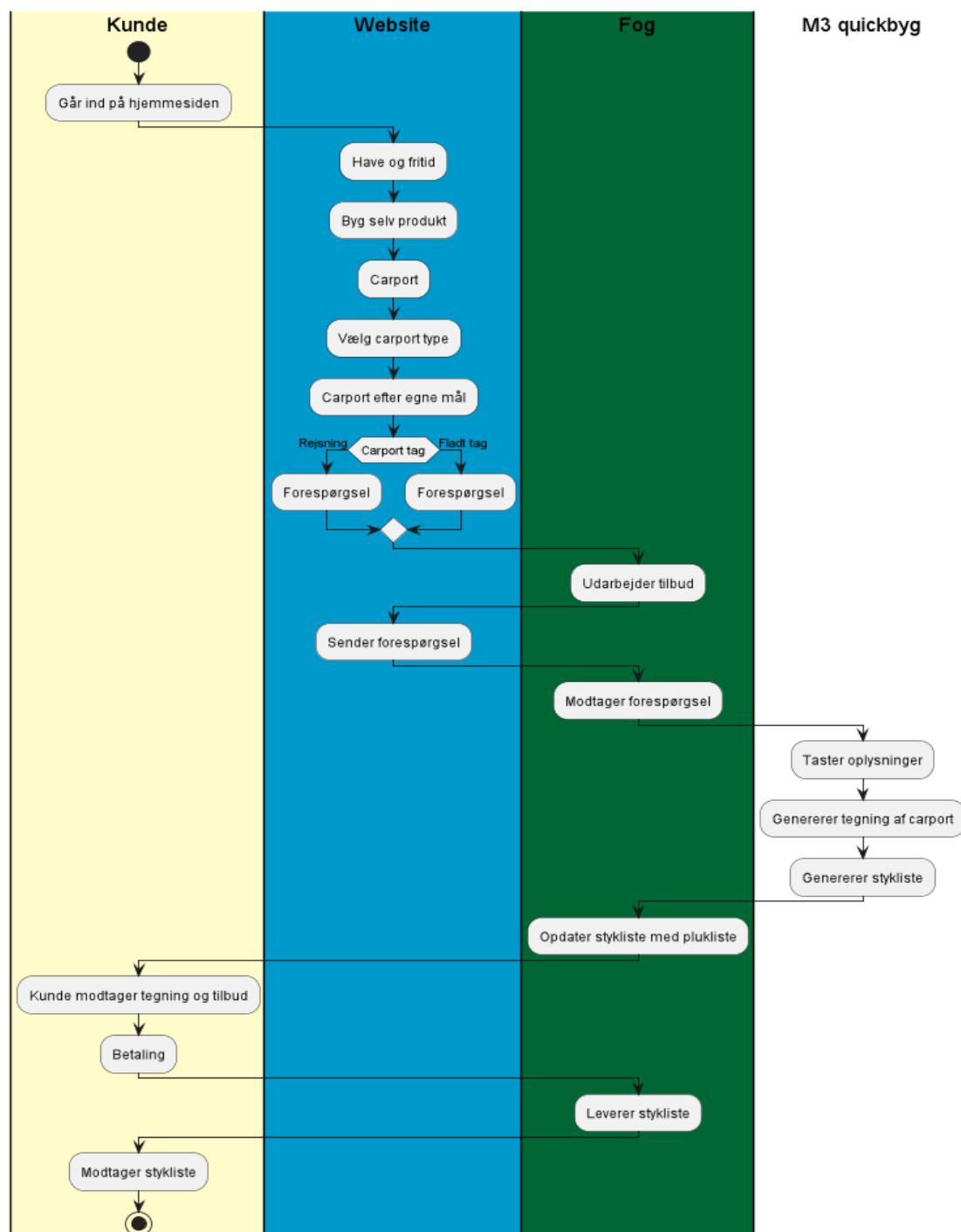


# Aktivitetsdiagram

Vi diskuterede hvorvidt det er nødvendigt med et, to, tre eller fire aktivitetsdiagrammer. Diagrammerne ville se meget forskellige ud alt efter hvilken rolle brugeren har på siden (kunde eller admin). Hvis man laver fire diagrammer kan der vises både et *as-is*-diagram for begge roller og et *to-be*-diagram. *As-is* er et udtryk for vores forståelse og indsigt i hvordan systemet p.t. fungerer, hvor *to-be* er vores visioner og idéer for systemets udvikling. Det diagram vi valgte tog udgangspunkt i det flow som foregår når kunden skal bestille en carport.

As-is:

Før vi begyndte at kode lavede vi aktivitetsdiagram over Fogs nuværende system. Her kan man se, at der er en del led hvor kunden venter på medarbejderen, og det er absolut nødvendigt at kunde og medarbejder taler sammen før ordren bliver gennemført. Her er også en aktør i M3 Quickbyg som Fog meget gerne vil undgå i fremtiden. For at komme hen til byg-selv-carport-funktionen skal man igennem en del sider, og strukturen er ikke hensigtsmæssig for hverken kunde eller medarbejder. Funktionen er også gemt godt væk og det er ikke synderligt logisk, at man skal ind under standard-carport førend man kan vælge byg-selv.

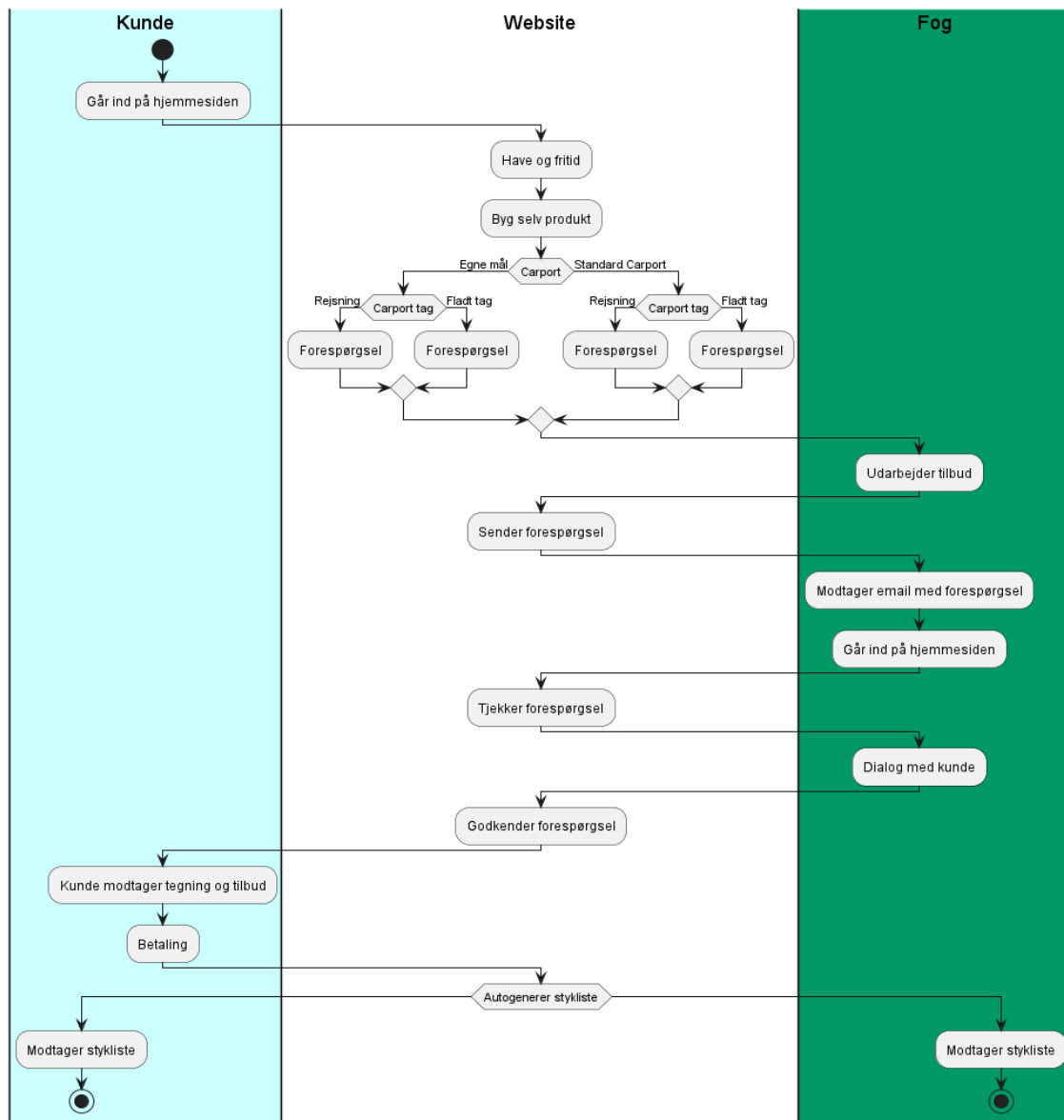


To-be:

Ud fra den stillede opgave indeholder vores *to-be* aktivitetsdiagram bl.a. flere undermenuer, da vi mente det kunne give kunden og medarbejderen et bedre flow undervejs. Tanken var, at kunden allerede efter at have klikket på carport kan vælge om de vil have egne mål eller standard. Herefter udregner systemet, ud fra de

indtastede mål, hvilke materialer der er nødvendige og prisen på disse. Admin kan godkende forespørgslen, som herefter sendes til dialog med kunden og blive til en gennemført ordre.

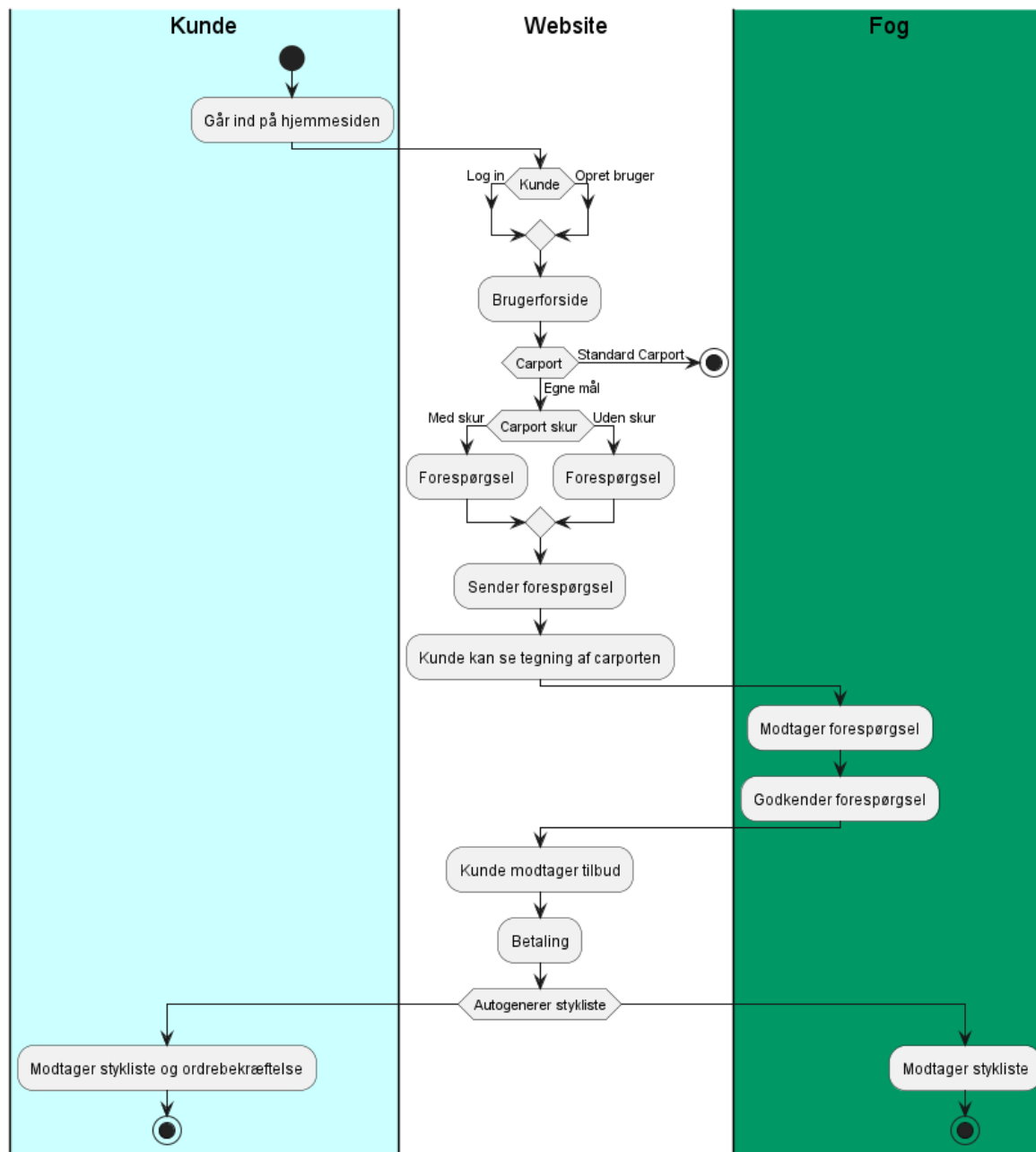
Vi var dog ikke helt tilfredse, da der stadigvæk var en del kommunikation mellem kunde og medarbejder.



Is-now:

Vi valgte at lave et sidste aktivitetsdiagram (*is-now*), hvor vi forsøgte at automatisere det meste. Kunden får, efter at have indtastet sine mål på carporten, automatisk en tegning af carporten og en foreløbig pris. Samtidig bliver der genereret en styklister som Fogs medarbejder kan kigge igennem for fejl. Kun hvis der er nogle fejl kan medarbejderen kontakte kunden og løse fejlen, ellers bliver forespørgslen godkendt og kunden

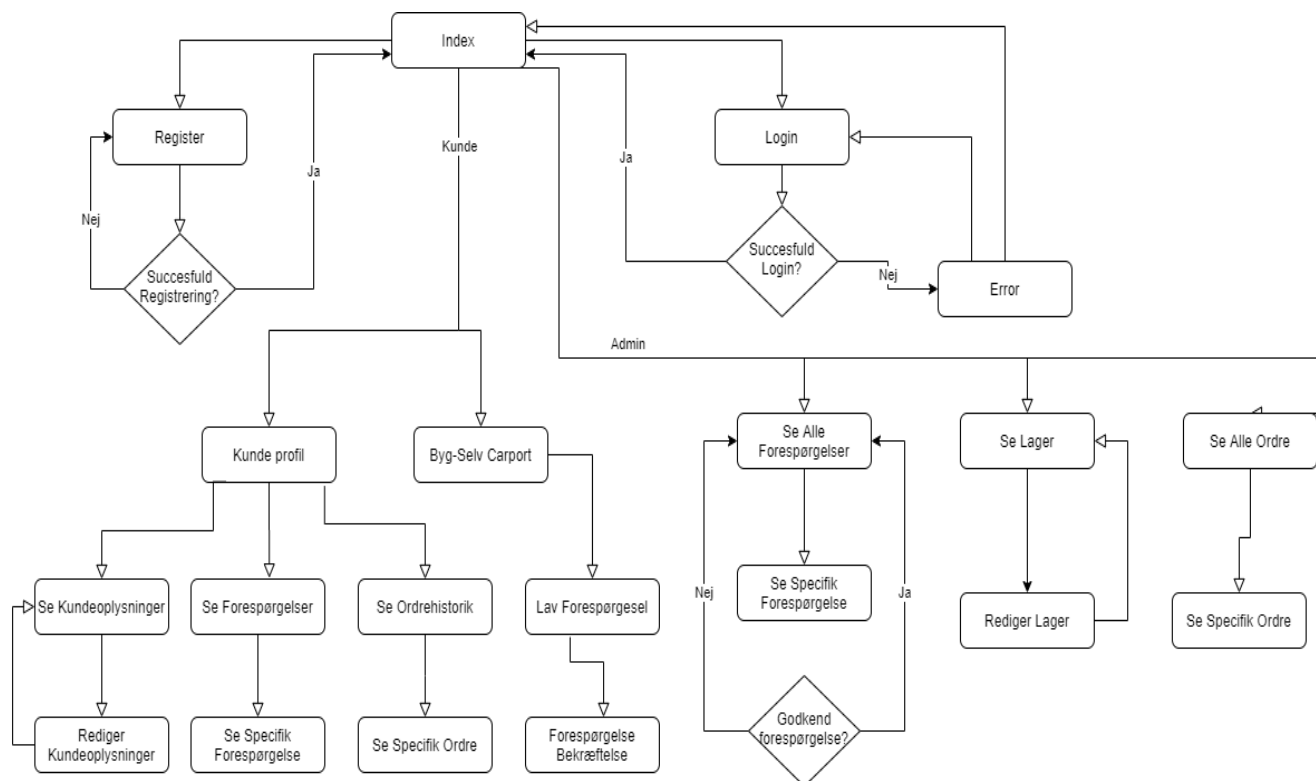
får den tilbage til betaling. Efter betaling fra kunden bliver der automatisk sendt styklister til både kunde og Fog.



# Navigationsdiagram

Vi har lavet et navigationsdiagram for at illustrere hvordan man som admin og bruger navigerer på hjemmesiden. Før der er logget ind er der ikke noget i navigationsbaren, men efter succesfuldt login vil der, for kunden være en navigationsbar og en anden for admin. Hvis brugeren ikke allerede har en profil på siden kan der via forsiden (index.jsp) registreres en bruger (dog kun med kunde-account-funktion, admins skal oprettes manuelt i databasen).

Herefter har hhv. kunden og admin adgang til de funktionaliteter som der er givet til rollen.



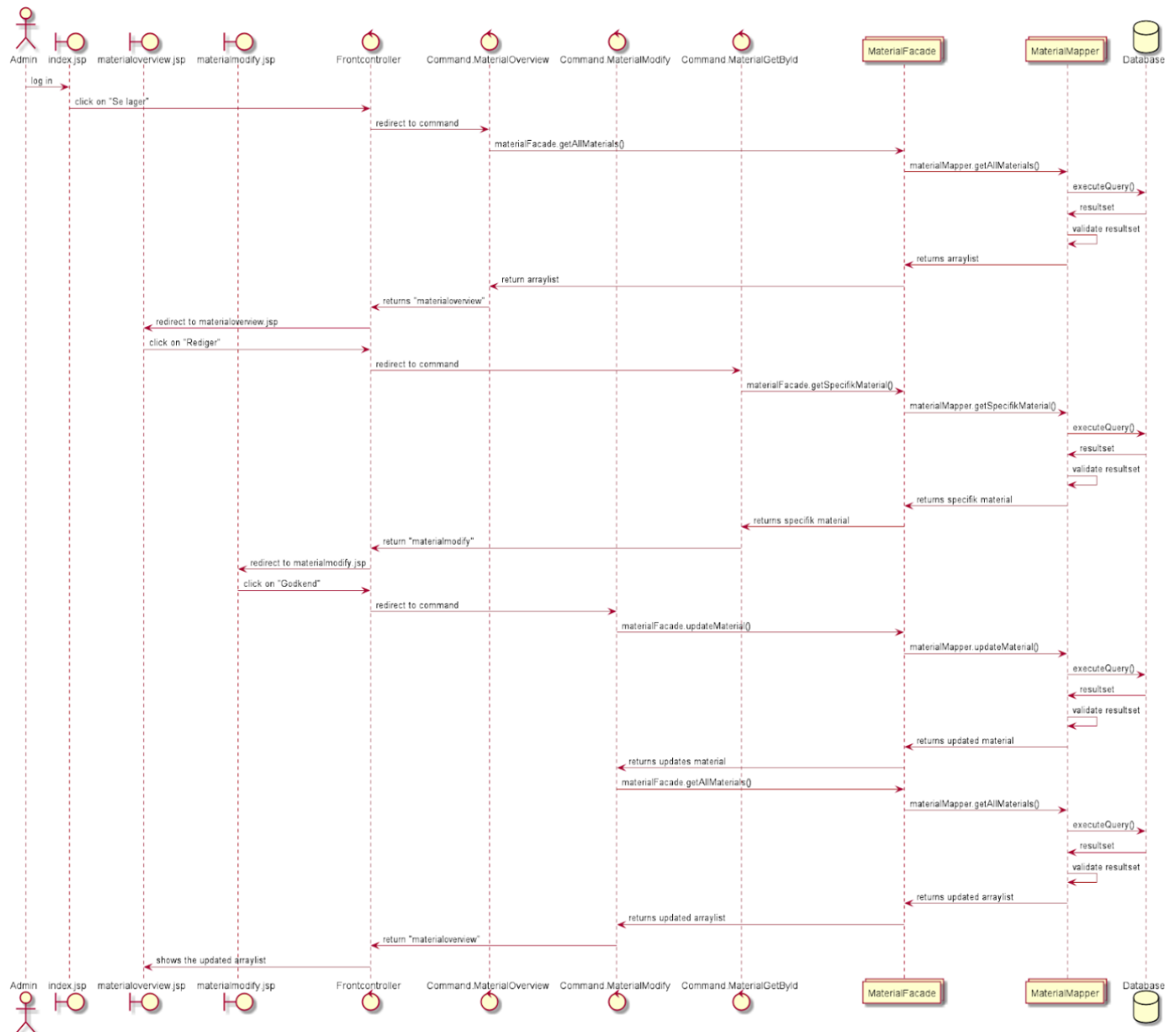
# Sekvensdiagram

Vi illustrerer her den sekvens som foregår når en admin skal redigere et materiales pris.

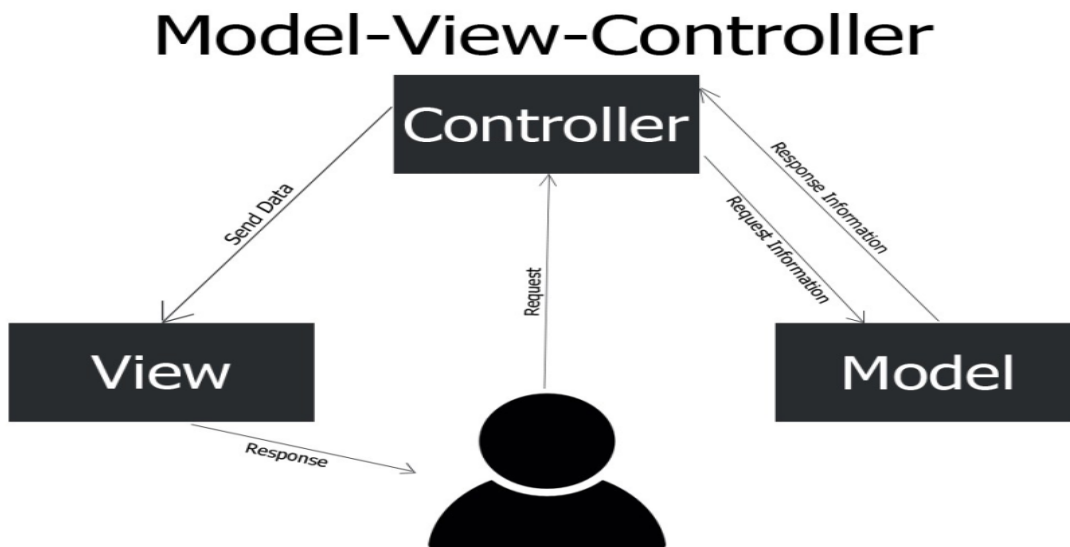
- Admin logger ind og lander på index.jsp.
- Admin klikker på “Se lager”.
- Frontcontrolleren kigger på request parameteren *command* og omdirigerer til Commanden: MaterialOverview.
- MaterialOverview kalder på MaterialFacade.getAllMaterials().
- MaterialFacade tager fat på MaterialMapper.getAllMaterials().
- GetAllMaterials()-funktionen tager fat i databasen og returnerer en arrayliste med materialer.
- Arraylisten bliver returneret tilbage til MaterialOverview og bliver sat på requestScopet.
- MaterialOverview returnerer en String (*materialoverview*) til Frontcontrolleren.
- Frontcontrolleren omdirigerer til materialoverview.jsp.
- Admin klikker på *rediger*-knappen på det pågældende materiale.
- Frontcontrolleren kigger på request parameteren *command* og omdirigerer til Commanden: MaterialGetByID.
- MaterialGetByID tager request parameteren materialID og kalder MaterialFacade.getSpecificMaterial().
- MaterialFacade tager fat på MaterialMapper.getSpecificMaterial().
- GetSpecificMaterial() finder og returnerer det pågældende materiale.
- Materialet bliver returneret til MaterialGetByID og sat på requestScopet.
- MaterialGetByID returnerer en String (*materialmodify*).
- Frontcontrolleren omdirigerer til materialmodify.jsp.
- Admin klikker på *godkend* efter rettelser.
- Frontcontrolleren kigger på request parameteren *command* og omdirigerer til Commanden: MaterialModify.
- MaterialModify tager alle request parametrene og kalder på MaterialFacade.updateMaterial().
- MaterialFacade tager fat på MaterialMapper.updateMaterial().
- UpdateMaterial() opdaterer materialet og returnerer en boolean om hvorvidt handlingen var succesfuld eller ej.
- Boolean bliver returneret til MaterialModify.
- Hvis boolean er true, kaldes MaterialFacade.getAllMaterials().



- MaterialFacade tager fat på MaterialMapper.getAllMaterials().
- GetAllMaterials()-funktionen tager fat i databasen og returnerer en arrayliste med materialer.
- Arraylisten sættes på requestScopet og MaterialModify returnerer en String *materialoverview*.
- Frontcontrolleren omdirigerer til materialoverview.jsp.



# Valg af arkitektur



Ved projektets start blev vi præsenteret for to forskellige arkitekturer, der henholdsvis bruger *servlets* eller *FrontController* og *command-pattern*. Valget af *FrontController* blev taget på baggrund af, at den tidligere udleverede startkode benyttede sig af *servlets*, som kunne medføre visse problematikker. Eksempelvis kan der forekomme mange *servlet*-klasser, hvilket kan gøre kode og navigation uoverskuelig. En af styrkerne ved *FrontController* med *command-pattern* er, som vi ser det, nedrivningen fra en fælles *command*-klasse. Det vil f.eks. lette en eventuel fremtidig implementering af sikkerhed. Dette ville kunne gøres ved hjælp af den overordnede *command*-klasse, som hver underklasse så nedarver fra.

Vores brug af MVC, har gjort det muligt for os, at inddele vores kode i tre kategorier som hver især håndterer deres respektive område. Tager vi udgangspunkt i vores projekt, fungerer MVC på den måde, at når en kunde laver en *request*, er det vores controller som modtager *requesten* samt bearbejder den forespurgte information og sender det videre. *Controlleren* er bindeleddet mellem *model* og *view*.

*Controlleren* kontrollerer hvad forespørgslen fra kunden handler om og sender det videre til *model*. Et eksempel kunne være, at kunden efterspørger en ordreoversigt. *Controlleren* tager denne *request* og sender videre til *model*. *Model* håndterer den efterspurgte data. Når *model* har processeret hvad det er kunden vil have fat i, sender *model* den validerede data tilbage til controlleren, som igen videresender den validerede data til *view*. *View* er her vores JSP-side, hvis eneste formål er at præsentere dataen på en overskuelig måde.

På *model*-siden benytter vi samtidig *Facade design-pattern*, dette gør vi for at simplificere vores brug af, samt overblik over *mappere*.

Vi benytter facade-design-pattern når vi benytter vores *mappere*. Et eksempel kunne være, at når vi henter *OrderDTO* igennem en facade, tager vi fat i den overordnede facade, som indeholder de forskellige mappere og de pågældende funktioner. Samtidig undgås der, at der er mapper-kald spredt forskellige steder i koden.

## Udvalgte kodeeksempler

Til udregning af det nødvendige materiale og mål på carporten ud fra kundens indtastede ønsker, har vi lavet en del *beregnere* i backend-delen. Vi har valgt at lave så små funktioner som muligt.

```
public int calculateQuantityOfBoxes(int amountNeeded, Material material) {  
  
    int amount = 1;  
  
    while (amountNeeded > material.getQuantity() * amount) {  
        amount++;  
    }  
    return amount;  
}
```

I *calculateQuantityOfBoxes* tager vi en integer, *amountNeeded*, samt et materiale. Vores algoritme tjekker derefter om det antal der er behov for er større end antallet af materiale gange antal (der som standard er sat til 1). Hvis ikke der er nok lægges endnu en til *amount*, indtil der er nok af denne og den returneres.

```
public Material calculateMaterialLength(int dimension, ArrayList<Material> listOfMaterials) {  
  
    for (Material material : listOfMaterials) {  
        if (material.getLength() > dimension) {  
            return material;  
        }  
    }  
    return null;  
}
```

I vores *calculateMaterialLength*-funktion tager vi en integer (*dimension*), samt en ArrayListe af materialer med i parametrene. Vores funktion tjekker om det materiale, som vi henter fra databasen, passer til den dimension det skal bruges til. Hvis vi skulle bruge en rem på x meter, tjekker vi om der er en rem som er lang nok, hvis vi har en rem der er passende i længde, returnerer vi det materiale. Hvis det fra databasen hentede materiale ikke er langt nok, så returneres der *null*.

```

public ArrayList <Bomline> calculateHead (int carportLength, ArrayList<Material> headArrayList, int shedLength) {
    ArrayList<Bomline> bomlineArrayList = new ArrayList<>();
    Material head = calculateMaterialLength( dimension: carportLength - shedLength, headArrayList);

    Material headForShed = null;

    for (Material m : headArrayList) {
        if (m.getLength() > shedLength) {
            headForShed = m;
            break;
        }
    }
    if (headForShed == null) {
        throw new ArithmeticException("Kunne ikke finde en rem som passer til dit skur");
    }

    bomlineArrayList.add(new Bomline( descriptionId: 8,head, quantity: 2));
    bomlineArrayList.add(new Bomline( descriptionId: 9,headForShed, quantity: 1));

    return bomlineArrayList;
}

```

I *calculateHead* (udregn rem) tages to *integers*, carportens længde og skurets længde, samt en liste over materialer. Vi benytter os herefter af en anden udregner, *calculateMaterialLength*, som beregner længden på første del af vores rem. Længden på anden del af remmen, den som skal på skuret, udregnes derefter ved at tage længden af det materiale som vi har i vores liste af remme og tjekke om længden af den er større end længden af skuret. Når der findes et materiale som er længere end skurets længde, sættes dette til vores rem-til-skur, som derefter kan tilføjes til vores bomline-liste sammen med den tidligere fundne længde af resten af remmen.

## Særlige forhold

Valg af scopes:

Vi bruger *SessionScope* når en bruger/kunde logger ind, det gør vi fordi *pagetemplate* og *index.jsp* tjekker hvilken type bruger der er logget ind, fordi der bliver vist forskellige ting afhængig af hvilken rolle man er logget ind som. Det er gældende så længe brugeren er logget ind, derfor mener vi, at det giver mere mening at gemme det på session.

*ApplicationScope* bliver ikke benyttet, da scopet kun bliver brugt til ting der er fælles for alle sessioner.

*RequestScope* er det primære *scope* vi bruger, det gør vi da vi har en masse informationer der skal gemmes og hentes fra databasen. Der bruges f.eks. *requestScope* hver gang vi henter materialer, eller når vi skal

genere en *SVG*-tegning. *RequestScopet* bliver også brugt fordi vi vil være sikre på, at den data vi henter fra databasen er opdateret. Et eksempel kunne være, at når vi vil redigere varelageret, så er vi interesseret i at hente den nyeste liste fra databasen og lave vores rettelser. På den måde sikrer vi, at vi ikke ligger inde med en forældet liste af materialer på *sessionScopet*.

Sikkerhed:

Der er ikke implementeret sikkerhedsforanstaltninger omkring kontooplysninger da det ikke har været en del af pensum.

Fejlhåndtering, logning og exceptions:

Fejlhåndtering for kundens bestillingsside er håndteret ved hjælp af drop-down menuer og påkrævede felter. Disse gør, at det kunden indtaster altid følger den logik som der forventes i backend.

Fejlhåndtering ved registrering af nye kunder sker ved, at vi laver et check om hvorvidt den indtastede email adresse allerede eksisterer i databasen. Hvis emailen findes, vil der komme en fejlbesked frem.

Vi benytter logning når der foretages databaseforespørgsler, alvorlige (*severe*) fejl logges også i tomcat loggen.

Vi bruger *database-exceptions* ved mapper, når vi tilgår databasen og skal hente/indlæse data.

Der mangler fejlhåndtering fra admin, i forbindelse med oprettelse af nyt materiale.

SVG:

Vi vurderede først at SVG ville være en 1-2 dages implementering, men fandt hurtigt ud af at vi fejlvurderede hvor krævende det ville være at implementere SVG-tegningen i projektet. Vi har prøvet at gøre det så dynamisk som overhovedet muligt.

Vi har bl.a. valgt i gruppen, at der ikke er SVG-tegning fra siden af carporten, ligeledes er der heller ikke retningsbestemte pile eller detaljeret tekst der beskriver afstanden mellem spær, eller carportens størrelse på tegningen.

Kodestandarder:

Vi blev i gruppen hurtigt enige om en kode-standard, som skulle gælde under hele projektets forløb. Bl.a. skulle alle funktioner i Java skrives med *camelcase* og alle navne i databasen skulle skrives med *underscore*. Alle tabeller i databasen samt alle funktioner i Java skulle også skrives på engelsk og navngivningen af funktionerne skulle være så præcise som overhovedet muligt, for at spare os selv for en overflod af kommentarer i koden. Dog har vi valgt at de materialer som ligger i databasen skulle være på dansk, da disse bliver sendt direkte fra databasen til stykliste.

## Status på implementation

Vi har benyttet *CRUD*-metoder hvor vi følte det gav mening. Bl.a. har vi ikke en funktion der sletter brugere, da vi ikke så denne funktion nødvendig. Til gengæld har vi lavet en *Create*-funktion for brugere, da kunden jo kan lave en ny profil på siden.

På nuværende tidspunkt har vi ikke nået at implementere PDF-genereringen af stykliste og instrukser til kunden. Vi har heller ikke nået at implementere en opdateret ordrepris som er baseret på styklisten.

Ydermere har vi ikke kunne nå at rette vores forespørgsel-form til. Lige pt. viser vores form at man kan vælge længde på skur der er for lang i forhold til carportens længde. Ligeledes mangler der korrekt formatering af tidsstempellet inde på vores ordreoversigt.

Tabellen nedenunder viser status på vores user-stories.

User-story	Implementeret	Ikke-implementeret
Kunden kan oprette en bruger på hjemmesiden.	✓	

Kunden kan, efter at have oprettet en bruger, logge på med indtastede e-mail og kodeord.	✓	
Kunden kan indtaste ønskede mål på carport i en form.	✓	
Kunden kan se sin email på alle sider efter login.		Vi implementerede ikke denne user-story, da vi i stedet valgte at lave en kundeprofil-side, hvorfra kunden kan se og redigere i sine oplysninger.
Kunden kan se sin egen ordrehistorik og forespørgsel.	✓	
Kunden kan ændre i sine oplysninger.	✓	
Admin kan ændre priser og varelager direkte i MySQL.	✓	
Admin kan se hvilke mål kunden har indtastet i sin carport og tjekke for eventuelle fejl.	✓	
Admin kan godkende ordrer og sende stykliste og tegning til kunden.		Delvist implementeret ved aflevering. Admin kan godkende ordrer og tegning genereres, men kunden modtager endnu ikke stykliste.

Admin kan redigere i priser og databasen direkte på hjemmesiden.	✓	
Admin kan se alle ordrer og deres status.	✓	
Admin kan se alle oprettede kunder i systemet.		Ikke implementeret. Det ville være nice-to-have, men vi måtte prioritere anderledes.

CRUD-operationer:

Mapper/DTO	Create	Read	Update	Delete
Account	✓	✓	✓	
Bom	✓	✓		
CarportRequest	✓	✓	✓	✓
Customer	✓	✓	✓	
Material	✓	✓	✓	✓
Order	✓	✓		
AccountDTO		✓		
BomDTO		✓		
CarportRequestDTO		✓		



# Arbejdsprocessen

## Gruppearbejde:

Inden vi startede på projektet, fastlagde vi nogle grundregler for hvordan gruppearbejdet skulle foregå. Vi valgte i gruppen, at arbejde ud fra princippet at alle skulle få kodet “lige meget”, så det ikke kun var én person der kodede det hele. På denne måde ville alle i gruppen have samme forståelse for hvert enkelte linje kode og have samme indsigt i kodens opbygning samt funktionalitet.

Inden dagen gik i gang gennemgik vi gårsdagens agenda, for at se hvad vi havde nået og ikke nået. Derefter delte vi os op i to grupper med to personer i hver, hvorefter der blev kigget på enten det som ikke blev løst fra gårsdagen, eller dagens program og opgaver. Hvis der opstod problemer med koden i grupperne eller hvis der skulle diskuteres funktionalitet i koden, blev disse løst i plenum. Grupperne blev roteret hver dag.

Det var vigtigt for os at alle var med i beslutningsprocessen. Det gjorde muligvis processen langsommere i starten, men efter at veje fordele og ulemper, så vi klare fordele for læringsprocessen med denne inddeling. Vi aftalte også at man ikke skulle arbejde videre på projektet derhjemme efter endt dag, da det var vigtigt at alle havde samme indsigt i koden og arbejdsbyrden var ligeligt fordelt, medmindre der internt blev aftalt at give lektier ud til hinanden som skulle laves til dagen efter.

## Logbog:

Under projektet har vi valgt at føre logbog for hver dag via *Google Docs*. Logbogen skulle bruges til at give os overblik over hvad vi havde nået i løbet af dagen, men lige så vigtigt hvad vi *ikke* havde nået på den pågældende dag. Logbogen var også ekstra vigtig for os, idet den ikke kun indeholdte vores daglige agenda, men også størstedelen af vores tanker og refleksioner for hvordan projektet skulle tackles og hvorfor vi har gjort som vi har gjort.

Et eksempel på en gennemsnitlig logbog-entry:

### **10-05-2022**

*“Kiggede på quickbuild.jsp igen. Forsøgte at sætte javascript på formen, men vi fik endnu ikke de “required”-fields til at blive vist korrekt.*

*Arbejdede videre på beregneren, se noter forneden.*

*Startede på at lave beregnings-funktioner i java.*

*Fik lavet udregning på stolper og bolte til stolper og lavet unit test af metoderne. Denis og Morten arbejder videre på beregning af areal.*

*Betül og Long arbejder videre med de andre beregninger”*

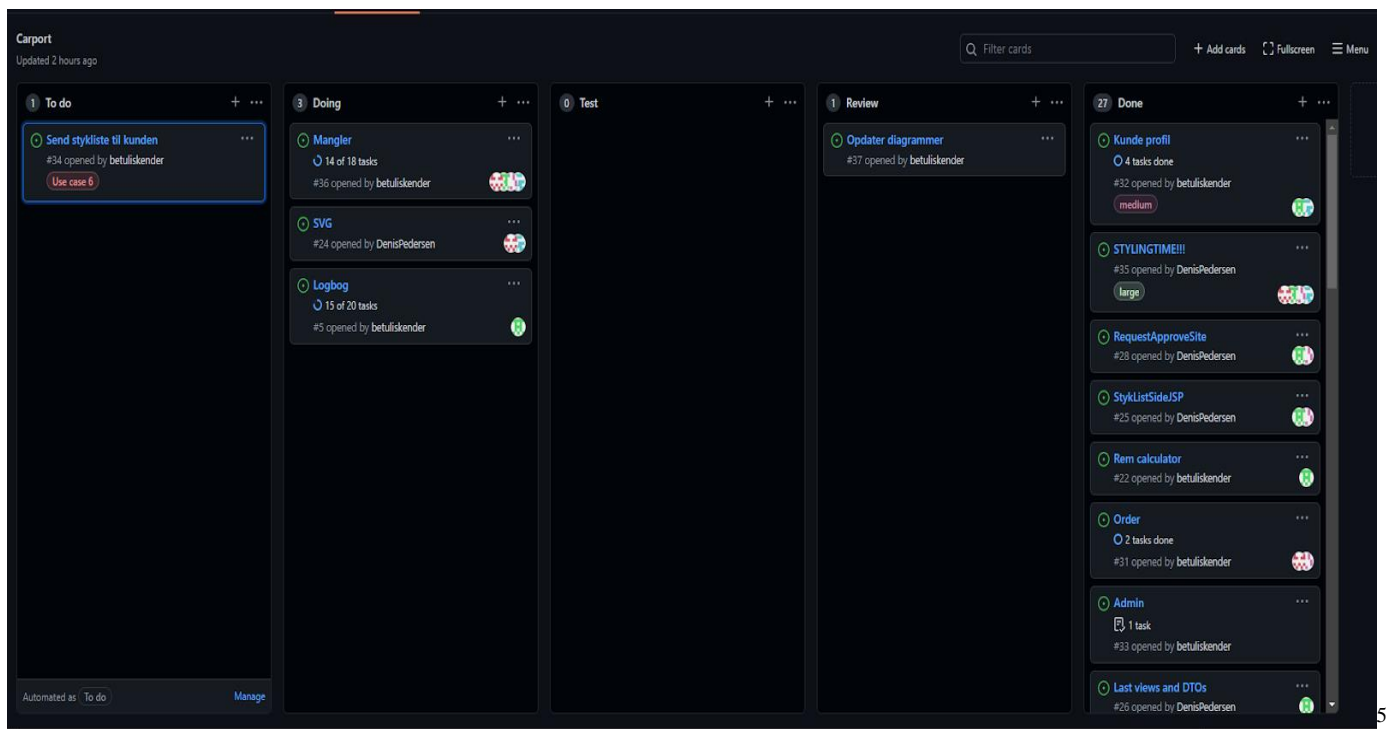
Github projects:

Inden vi startede kodningen brugte vi de første par dage til opsætning af den udleverede startkode, Github repository og lave et kanban board. Vi fik vist at man kunne lave kanban board i Github, hvilket vi valgte i stedet for de mere gængse kanban-boards. Det var også en god måde at holde styr på alle underopgaver.

Vi aftalte i gruppen hvem der havde ansvaret for at skrive logbog og hvem der havde ansvaret for kanban-boardet og få det opdateret løbende.

Før vi startede på kodningen, blev der lavet en række *Issues* (opgaver). Disse blev i starten påsat *To-Do* på kanban boardet som efterhånden blev rykket rundt alt afhængigt af hvor i processen vi var nået. Når man var i gang med en opgave ville en *Issue* blive rykket til *Doing* og når en opgave var færdig, rykkede man den videre til enten *Review*, hvor man gennemgik den færdige opgave og ellers til *Test* hvis funktionerne skulle testes inden det blev rykket til *Done* når man var helt færdig. *Review* blev foretaget internt i grupperne på dagen eller i plenum hvis det var større funktioner. Hvert *Issue* havde også en størrelsesforholdslabel (*small*, *medium*, *large*), så vi kunne se hvor stor den valgte opgave var, og hvor længe det ca. ville tage at kode.

Et eksempel på vores kanban board i Github der illustrerer de forskellige *Issues*.



Git:

Under projektet har vi valgt at bruge Git som vores primære værktøj til deling af kode. For hver arbejdsopgave oprettede vi en *branch* hvorfra vi kunne arbejde i grupperne. Når en opgave var færdig, testet og godkendt af alle i gruppen, blev *branchen* merged ind i *main* branchen.

Git har været et ekstremt vigtigt værktøj for os, på trods af små problemer undervejs. Bl.a. har det til tider slettet vores *project structure* i IntelliJ, når man lavede *rebase* med *main*, men i langt de fleste tilfælde har vi været glade for Git, specielt i situationer hvor koden ikke har virket, eller hvis vi har ændret i noget som vi ikke ville have ændret, at man så havde mulighed for at gå tilbage til tidligere *commits*. Git blev også brugt til at orientere sig med, da man kunne se hvor langt ens gruppemedlemmer var og hvad de arbejdede med. Herunder ses et udkast af nogle *commits* fra Git, hvor vi ca. var to uger inde i kodningen og midt i arbejdet af beregner-klassen.

<sup>5</sup> <https://github.com/coerth/carport/projects/1>

```

| | * | | 7bf4d7d udregning af stolper og udregning af bolte til stolper lavet
| | * | | c8de5da branch lavet
| * | | | c44bcc8 Mockup tilføjet
| * | | | 8bc1aed udregning af stolper og udregning af bolte til stolper lavet
| * | | | b4f452f branch lavet
* | | | | 449be34 Merge branch 'MortenogDenisCalcBranch'
| \ \ \ \ \
* | | | | | 76ccd77 idea balladen fikset
* | | | | | f1f7c62 panik over .idea mappe
| | | | | * aeef858 (origin/refactorbranch) branch lavet til refactoring
| | | | | /
| / | | | |
| | | | | * 021e976 (origin/MortenogDenisCalcBranch) small stuff in calc
| | | | | /
| / | | | |
| * | | | | 902bdb9 lavet tests på alle funktioner
| * | | | | 0501e2d Merge branch 'MortenogDenisCalcBranch' of https://github.com/coerth/carp
| | \ \ \ \ \
| | * | | | | 50e48d3 lavet test med array bomline
| * | | | | e953996 calcSterns fisket
| / / / / /
| * | | | | 86e42e0 fisket merge
| | \ \ \ \ \
| | * | | | | dec4a6e rettet fejl i carportcalculator og test
| | * | | | | 27d61a0 lavet tests for skrue og stjern funktioner
| | * | | | | 0dc91c4 fisket merge
| | | \ \ \ \ \

```

Discord:

Som kommunikationsplatform benyttede vi os af Discord. Her havde vi en fælles gruppe-kanal, hvor vi kunne udveksle idéer, filer og kunne skrive information om lokale- eller andre ændringer. Discord blev også brugt til online arbejde når skolen havde weekend-og helligdage lukket.

## Test

Til projektet har vi benyttet os af test, og vi har taget udgangspunkt i *v-modellen* og foretaget test på et low-level niveau i form af *unit-test* og high-level niveau i form af *integrationstest*. Grunden til vi laver disse test, gør det muligt for os at sikre, at alle vores individuelle under-funktioner samt mappere virker som de skal,

inden vi tager dem i brug og undervejs som projektet skrider frem. Derudover har vi også benyttet os af en *tænke-højt test*<sup>6</sup>, hvor vi har fået en repræsentant fra Fog(Jon) til at gennemgå de forskellige user stories.

I *CarportCalculator* klassen benytter vi os af *unit-testing*, dette gør vi da vi ikke har brug for en database forbindelse til at teste hvilke materialer vi skal bruge eller hvor mange. *Unit-testing* giver os også et fingerpeg om hvorvidt vores logik virker som det skal, inden vi fuldt implementerer funktionen. Et typisk eksempel på en unit test af en funktion i vores beregner klasse:

```
@Test
void calculatePerforatedTapeLengthWithShedTest(){
    int perforatedTapeLength = carportCalculator.calculatePerforatedTapeLengthWithShed( carportLength: 780, shedLength: 210);
    assertEquals( expected: 511, perforatedTapeLength);
}
```

Ovenover ses testen af vores udregning for hulbånd. Der tages en *integer* som sættes til det vores beregner returnerer efter vi har indtastet carportens mål. Herefter kaldes *assertEquals* ud fra det vi forventer at få tilbage. I dette tilfælde 511, da hulbånd skal have en længde på 511 cm. Et andet eksempel ses herunder, der testes beregningen af antallet af æsker ud fra samme princip som med hulbånd:

```
@Test
void calculateQuantityOfBoxesTest()
{
    Material material = new Material( materialId: 1, name: "Bundskruer", price: 5, unit: "Pakke", typeId: 2, quantity: 200);

    int result;

    result = carportCalculator.calculateQuantityOfBoxes( amountNeeded: 350, material);
    assertEquals( expected: 2, result);

    result = carportCalculator.calculateQuantityOfBoxes( amountNeeded: 200, material);
    assertEquals( expected: 1, result);

    result = carportCalculator.calculateQuantityOfBoxes( amountNeeded: 1, material);
    assertEquals( expected: 1, result);
}
```

Til vores *mappere* benytter vi *integrationstest*, og i modsætning til beregner-klasserne, har vi brug for databaseforbindelse, når vi f.eks. tester hvorvidt vi kan indsætte en ny bruger i databasen, eller hive forskellige informationer om brugeren fra databasen. Integrationstesten fortæller os også, hvorvidt vores *mappere* fungerer på den rigtige måde inden vi implementerer dem fuldt ud. Vores test-database er bygget efter samme princip og struktur som den database vi bruger i resten af projektet.

---

<sup>6</sup> Se [bilag](#)

Inden hver test køres, kaldes en *setUp()*-funktion, som bl.a. starter med at fjerne alt fra test-databasen. Det sikrer at der startes fra en blank database, så alle tests kører fuldkommen uafhængigt af tidligere tests.

```
@BeforeEach
void setUp() {
    try (Connection testConnection = connectionPool.getConnection()) {
        try (Statement statement = testConnection.createStatement()) {
            statement.execute( sql: "delete from `carport_request`");
            statement.execute( sql: "alter table carport_request auto_increment=0");
            statement.execute( sql: "delete from `account`");
            statement.execute( sql: "alter table account auto_increment=0");
            statement.execute( sql: "delete from `customer`");
            statement.execute( sql: "alter table customer auto_increment=0");

            statement.execute( sql: "INSERT INTO `account`(`account_id`,`email`,`password`,`role`)VALUES(1,'bla@bla.dk',1234,2)");
            statement.execute( sql: "INSERT INTO `customer`(`customer_id`,`name`,`address`,`city`,`zip`,`mobile`,`account_id`) VALUES(1,'Poul','Poulvej 2','Aalborg',2100,12345678,2)");
            statement.execute( sql: "INSERT INTO `carport_request`(`carport_request_id`,`width`,`length`,`roof`,`roof_incline`,`is_approved`,`shed_length`,`shed_width`,`customer_id`)VALUES(1,600,78
```

Ovenover ses vores *setUp()*-funktion, som kaldes før hver integrationstest. Efter der er ryddet fra de forskellige tabeller (her *account*, *customer* og *carport\_request*), sættes en smule data i, og man kan nu kalde de forskellige funktioner der er i vores *mapper*, blandt andet:

```
@Test
void getSpecificRequest() {

    CarportRequest carportRequest = CarportRequestFacade.getSpecificCarportRequest( requestId: 1, connectionPool);
    assertEquals( expected: 1, carportRequest.getRequestId());
}
```

I *getSpecificRequest()*-testen kontrolleres om der kommer det forventede carport-request-id ud.

```
@Test
void deleteCarportRequest() {

    boolean carportRequest = CarportRequestFacade.deleteRequest( carportRequestId: 1, connectionPool);

    assertEquals( expected: true, carportRequest);
}
```

Og i *deleteCarportRequest* testes hvorvidt det lykkedes at slette den forventede carport-forespørgsel ud fra det indtastede id.

Alle test som er lavet virker når de køres enkeltvis. Dog viser vores test ikke 100% gennemførelse, når man tester dem alle sammen på én gang. Grundet brug af *auto-increment* fejler vores test, da den ikke starter på 1, men på 3. Ud fra testene har vi en dækningsgrad på 36% på vores klasser, 43% på metoder og 50% på kodelinjer.

Vi har primært lavet test på vores funktionaliteter og knap så meget på entiteter hvor der kun er *gettere/settere*. Vi har på nuværende tidspunkt ikke lavet test på vores *DTO-mappere*, da vi kun laver en *read*-funktion der henter data og ikke indsætter data.

## Hvad kan vi tage med til næste projekt?

Vi har haft stor glæde af vores arbejdsfordelingen i gruppen og har været godt tilfreds med hele arbejdsgangen vi har haft. Vi har løbende haft positive og konstruktive diskussioner om projektets indhold og dets funktionaliteter, og har haft god sparring med hinanden ved kodning i grupper eller i plenum. Vores princip om at alle skulle kode "lige meget" gav det ønskede resultat, da alle i gruppen fik et tilhørsforhold til koden og hjemmesiden, således at den følte som *vores egen*.

Til en anden gang skal vi huske at opdatere diagrammer løbende. På trods af gode intentioner om at gøre dette, fik vi kun lavet de gængse klassediagrammer og domænediagrammer før projektstart og til slut ved rapportskrivning. Kun vores EER-diagram har vi løbende opdateret hver gang efter vores forskellige vejledermøder.

Git har været et vigtigt og brugbart redskab, og når det har fungeret optimalt har det været en fornøjelse at arbejde med. Git Projects har også, især med kanban-boardet givet et fint overblik over hvad der manglede at blive lavet og hvor langt vi var i processen med de forskellige arbejdsopgaver. Vi skal dog huske til næste gang, at benytte Gits *squash*-funktion, da vores log godt kan blive lang uden.

Logbogen har også været et godt redskab til at huske præcis hvad der er foregået den pågældende dag. Efter en lang arbejdsdag kan det være svært lige at huske *hvorfor* man valgte at gøre som man gjorde dagen før, og her har det givet et fint overblik.

Til en anden gang skal vi huske at *reviewe* vores *process* oftere. Vi har løbende *reviewet* i plenum når vi har lavet f.eks. på udregner-klassen, eller på større *issues*, men generelt skal vi være bedre til at *reviewe* selv mindre ting, hvordan det er gået, hvordan vi har tacklet opgaven etc.

## Bilag

Vores *user-stories* og deres tilhørende acceptkriterier. Vi har valgt ikke at lave dem som en accept-video med Fogs udsendte, da videoen ville blive for lang.

User-story	Acceptkriterie
Kunden kan indtaste ønskede mål på carport i en form.	Givet jeg er logget på som kunde, forventer jeg, at kunne vælge mål til min carport i en form efter jeg har klikket på <i>“byg selv”</i> .
Admin kan ændre priser og opdatere varelager.	Givet jeg er logget på som admin, kan jeg klikke på <i>“lager”</i> og herfra vælge <i>“rediger”</i> ud fra det materiale jeg ønsker at ændre på, og herefter vil ændringerne fremgå af databasen og hjemmesiden.
Admin kan se hvilke mål kunden har indtastet til sin carport og tjekke for eventuelle fejl.	Hvis jeg er logget på som admin, forventer jeg, efter at have valgt <i>“vis alle forespørgsler”</i> , og derefter klikke på <i>“vis”</i> ud fra den valgte forespørgsel, at kunne se kundens indtastede mål.
Admin kan godkende ordre og sende stykliste og tegning til kunden.	Givet jeg er logget på som admin, kan jeg efter at have valgt <i>“ordrer”</i> klikke på <i>“vis”</i> ud for den ønskede ordre og derefter vælge <i>“godkend ordre”</i> hvilket vil sende stykliste og ordrebekræftelse til kunden.
Kunden skal kunne oprette en bruger.	Givet jeg endnu ikke har oprettet en bruger, kan jeg vælge <i>“opret bruger”</i> og derefter indtaste informationer i en form hvilket vil skabe en brugerprofil som jeg senere kan bruge til at logge ind på siden med.
Kunden kan, efter at have oprettet en bruger, logge på med sin indtastede email og password.	Givet jeg har oprettet en bruger på siden, kan jeg logge på ved at vælge <i>“log ind”</i> .
Kunden kan se egen ordrehistorik, forespørgsler og status på disse.	Givet jeg er logget på som kunde, kan jeg, efter at have valgt <i>“kundeprofil”</i> i navigationsbaren klikke på <i>“se forespørgsler”</i> eller <i>“se ordrer”</i> , hvilket vil give mig en oversigt over de forespørgsler/ordrer jeg har hos Fog, samt status på dem.



Kunden kan ændre på sine oplysninger.	Givet jeg er logget på som kunde, kan jeg, efter at have valgt " <i>kundeprofil</i> " i navigationsbaren klikke på " <i>se kundeoplysninger</i> " vælge punktet " <i>rediger oplysninger</i> ", hvilket vil sende mig til en form hvorpå jeg kan redigere disse oplysninger.
Admin kan se alle ordrer og deres status.	Givet jeg er logget ind som admin forventer jeg, efter at have klikket på hhv. " <i>se alle ordrer</i> " eller " <i>se alle forespørgsler</i> ", at se en liste over alle forespørgsler/ordrer samt deres status.

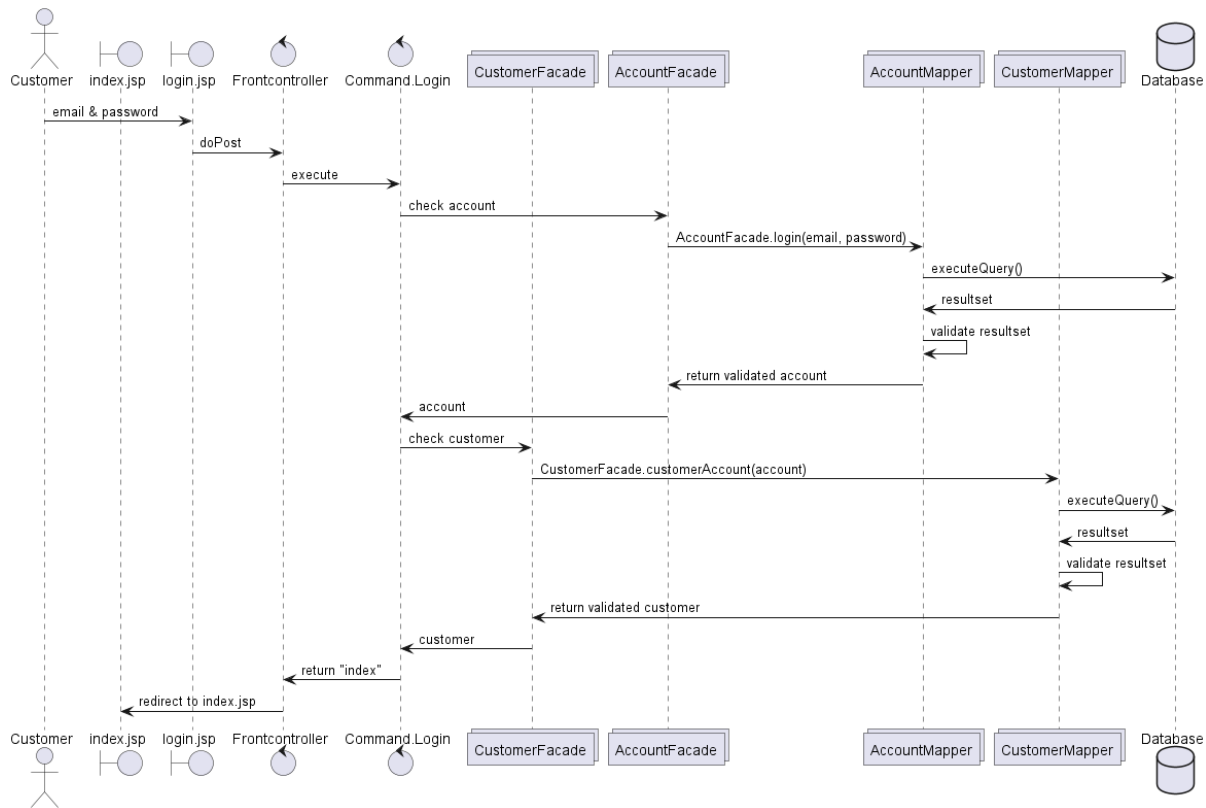
Sekvensdiagram for *login*:

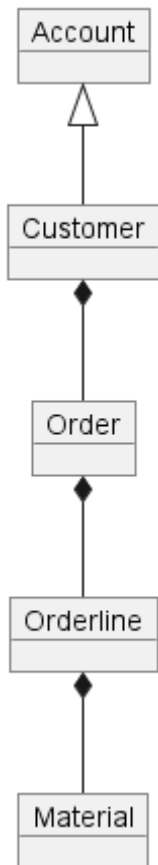
Først efter at have lavet login-funktionalitet lavede vi *DTO'er*. Hvis vi havde lavet *DTO'erne* tidligere, ville første del af vores sekvensdiagram sandsynligvis se noget anderledes ud.

Vi illustrerer her den sekvens som foregår når en kunde logger på siden.

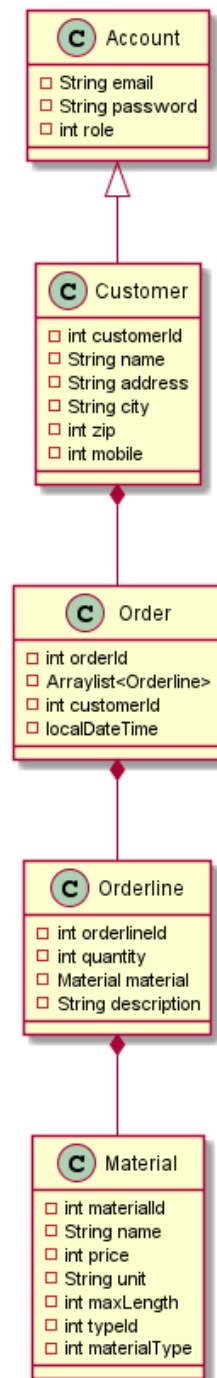
- Kunden indtaster email og password på login.jsp.
- Det kunden har indtastet i formen sendes videre til Frontcontroller med doPost.
- Frontcontroller udfører *post* og sender videre til Command (login.java).
- Login-klassen starter check-processen og sender info videre til AccountFacade.
- Der benyttes login-funktionen i AccountMapperen.
- Accountmapperen kigger i databasen og denne returnerer et resultset.
- Resultsettet valideres i AccountMapperen.
- AccountMapper sender den validerede account til AccountFacade.
- AccountFacade sender en account-entitet videre til login.java.
- Login.java starter check for customer.
- CustomerFacade checker med den tidligere modtagne account.
- CustomerMapperen kigger i databasen og denne returnerer et resultset.
- Resultsettet valideres i CustomerMapperen.
- CustomerMapperen sender den validerede customer til CustomerFacade.
- CustomerFacade sender en customer-entitet videre til login.java.
- Login.java returnerer index til Frontcontroller.
- Frontcontroller omdirigerer til index.jsp.

- Login-processen er fuldført.





Figur 1



Figur 2

Tænke-højt-test

<https://youtu.be/hQrJKiJmjPo>

Logbog, regneregler m.m.:

<https://docs.google.com/document/d/1TURKcCyGZ5irLmAah91D89C9ImQijX6V-7dQlcpzuic/edit#>