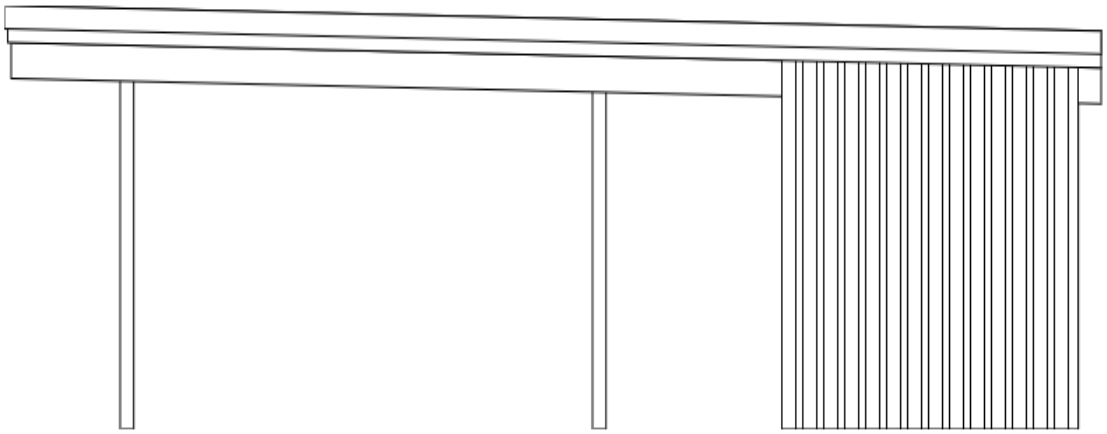
Fog Carport



2. semester eksamensprojekt
21/4/2020 – 29/5/2020
Af Bornit, bestående af

Benjamin Skovgaard – cph-bs190@cphbusiness.dk
Frederik Janum – cph-fj108@cphbusiness.dk
Mari Rødseth Haugen – cph-mh823@cphbusiness.dk
Matt Thomsen – cph-mt312@cphbusiness.dk
Pelle Rasmussen – cph-pr128@cphbusiness.dk

GitHub repository: https://github.com/benskov95/Fog_Carport_Bornit
Javadoc: https://benskov95.github.io/Fog_Carport_Bornit/
Link til hjemmeside: http://64.225.101.254:8080/Fog_Carport-1.0-SNAPSHOT/
Demo video: https://www.youtube.com/watch?v=17q49p9-hQ8

Login til admin – ID: *1* og password: *admin*
Login til lagermedarbejder – ID: *2* og password: *lager*

Indholdsfortegnelse

Indledning

Vi har som eksamensprojekt på 2. semester fået til opgave at lave en hjemmeside til firmaet Fog, der blandt andet sælger carporte. Formålet med hjemmesiden er, at der kan beregnes materialer til carporte samt laves todimensionelle tegninger ud fra angivne mål, og at der kan foretages bestillinger af carporte.

Det primære mål var at automatisere beregningsprocessen af carportbestillingen, således at hele processen er mere strømlinet end før, hvor der manuelt skulle kigges og beregnes materialer til hver ordre af en af Fogs ansatte (Martin).

Til det formål har vi brugt en given projektskabelon, og denne skabelon kører med en servlet (FrontController) og klassen Command, der styrer navigationsdelen af websiden. Det er gennem Command klassen, at diverse jsp-sider får instantieret deres respektive java-kodede funktioner, og gennem FrontController, at man sørger for, alle requests bliver håndteret korrekt – dvs. sørger for, at brugeren kommer til den rette side for eksempel.

Vi har kørt og testet websiden gennem en lokal TomCat server i IntelliJ og har senere smidt projektet op på vores DigitalOcean droplet, som også har en TomCat server installeret. På den måde har vi rykket projektet fra at blive kørt lokalt til at køre online.

Om leverandøren (Bornit)

Vi (Bornit) er et lille udviklerteam på 5 studerende, som har fået til opgave at levere et produkt til firmaet Fog. Før udviklingsprocessen begyndte, lavede vi en SWOT-analyse på os selv for at identificere vores styrker og svagheder mm., så vi var bevidste om dem, før vi gik i gang med projektet, og kunne prøve på bedste vis at arbejde effektivt i forhold til dem. Det følgende er uddybende konklusioner fra vores SWOT-analyse.

Vores vision

Som team har vores vision med projektet været at afhjælpe Fog med deres forældede IT-system ved at levere et produkt, der lever op til kundernes såvel som Fogs ansattes forventninger og som også er af en kvalitetsstandard, vi kan være stolte af som team.

Styrker

Vi synes selv, at vi som team er meget engagerede og passionerede omkring programmering, og det er jo en klar styrke, da det betyder, vi har gjort meget for at levere et tilfredsstillende produkt, og at vi også ville se det som et personligt nederlag, hvis vi ikke gjorde vores bedste.

Derudover kommer vi alle fra forskellige baggrunde, hvilket betyder, vi ser og forstår problemer på forskellige måder, og det leder til bedre løsninger, eftersom flere perspektiver giver et bedre og mere velovervejet resultat. Det hjælper desuden også generelt på problemløsning, da løsningen på ét problem måske står meget klarere for nogle end andre.

Der er heller ingen af os, der er teknologisk udfordret, så vi har ikke haft større tekniske problemer undervejs med den software, vi har brugt i udviklingsprocessen. Det nævnes kun, fordi vi i første semester havde enkelte, der havde lidt sværere ved at bruge computere.

Sidst men ikke mindst har vi haft meget fleksibilitet ift. vores arbejdstider, hvilket har gjort, at vores indsats har været svingende gennem hele forløbet. Dét mener vi er en god ting, da det har betydet, at vi visse dage har kunnet lægge alle kræfter i og kørt fuld skrue, men samtidig også haft dage, hvor vi tog et pusterum for at klare tankerne og følgende vendt tilbage med fornyet gåpåmod.

Svagheder

Da vi alle er uerfarne programmører, har vi naturligvis en del svagheder. Den manglende erfaring betyder, at vi kan (og har) løbet ind i en del problemer undervejs, som andre, mere erfarne teams muligvis ikke ville have gjort. Det har yderligere betydet, at vi har brugt mere tid på at løse bestemte problemer, hvor andre teams måske ville have haft den ekstra tid til at forbedre andre steder i produktet.

Derudover er der også kompetenceforskelle mellem de forskellige teammedlemmer, som har betydet, at nogle medlemmer har spillet en større rolle i udviklingsprocessen end andre. Selvfølgelig har nogle været skarpere på enkelte områder end andre, hvilket har betydet, at vi har kunnet komplementere hinanden af og til.

Vores manglende viden indenfor tømrerfaget har i dette projekt betydet, at vi har været nødsaget til at tilegne os viden om tømrerterminologi[1], så vi kunne diskutere på lige fod og blive enige om forskellige beslutninger i relation til vores beregninger og tegninger af carporte.

COVID-19 pandemien har også påvirket vores arbejdskvalitet, da diskussioner og fællesarbejde af og til har været vanskeligere, for eksempel i situationer hvor man kan have svært ved at visualisere hinandens tanker.

Muligheder

Naturligvis er vores omdømme som team på spil, da projektets succes ikke er garanteret. Hvis det er en kæmpe fiasko, kan det såre vores fremtidige arbejdsmuligheder og skabe karriereproblemer for os som udviklere. Omvendt kan det sikre større og flere karrieremuligheder hvis det er en stor succes.

Uanset udfaldet af projektet har vi i hvert fald fået væsentligt mere erfaring både med skabelonen, vi har arbejdet med som fundamentet for vores projekt, men også indenfor tømrerfaget (ift. terminologi) hvis vi nogensinde skulle arbejde på et tømrerrelateret projekt igen i fremtiden.

Trusler

Der er selvfølgelig også andre udviklerteams, som arbejder for andre, lignende virksomheder, der måske ender med at lave et bedre produkt end os, hvilket både ville påvirke Fog og os selv, da det andet firma måske kunne få en bedre plads på markedet end Fog grundet succesen, og det andet team ville være en mere attraktive mulighed end os hvis andre havde brug for at få udviklet et lignende IT-system.

Uvisheden, som COVID-19 pandemien har skabt, har betydet, at vores evne til at arbejde og om vores produkt overhovedet ville blive brugt (hvis nu Fog gik konkurs som følge af den manglende forretning i tiden) har været en klar trussel fra dag 1. Både *muligheder* og *trusler* er selvfølgelig mere hypotetiske i dette tilfælde – det er nærmere en forestilling om, hvilke konsekvenser der kunne være hvis vi var et reelt udviklerteam.

[1] Se Ordliste under Yderligere Links

Teknologivalg

- IntelliJ 2020 1.1
- MySQL Workbench 8.0.18
- TomCat 8.5.511
- JDBC Driver (mysql-connector-java 8.0.18)
- JDK 1.8.0 *og* 11.0.6
- Git version 2.24.1.windows.2
- DigitalOcean droplet (Linux server – Ubuntu 18.04.4)
- Graphviz (så PlantUML plugin virker i IntelliJ)

Krav

Firmaets vision

Formålet med det opdaterede bestillingssystem, hvor kunden selv kan vælge dimensionerne på sin carport, er at give Fogs kunder bedre mulighed og frihed, når de skal bestille en carport. Det vil give kunderne en bedre oplevelse, som så vil resultere i et bedre omdømme og på lang sigt bedre forretning for Fog.

Fordelene er heller ikke udelukkende kunderelaterede, da Fogs nuværende beregningssystem til carport materialer mm. er begrænset, da man ikke kan indsætte nye materialer, opdatere længder eller andre værdier på enkelte materialer etc.

Med det nye system, vi har udviklet, er der mere fleksibilitet og bedre fremtidssikring, som med god sandsynlighed kan gøre hele arbejdsprocessen ift. carportbestilling og vedligeholdelse af databasen nemmere for Fogs ansatte.

Overordnet beskrivelse af virksomheden

Johannes Fog A/S er en butikskæde, som har forskellige afdelinger, blandt andet Bolig & Designhus og Trælast & Byggecenter – den sidstnævnte er den afdeling, som vi har arbejdet for i dette projekt. I den afdeling sælges der bl.a. carporte, eller i hvert fald alle de nødvendige materialer til en carport, som så kan bygges af eksternt hyrede tømrere (dog har Fog selv tømrerkontakter, de kan henvise kunderne til).

Vi har i forbindelse med forberedelsen til projektet lavet en interessentanalyse for bedre at kunne forstå forholdene mellem de involverede personer og også vores egen rolle i projektet i forhold til kunden (Fog).

Interessentanalyse

Fog er selvfølgelig heller ikke det eneste byggecenter i Danmark, og det betyder naturligvis, at de har noget forretningskonkurrence. Derfor er det vigtigt at holde sig på forkant og udvikle sig som firma, hvis man vil sikre sig en god plads i markedet – og det kan det opdaterede bestillingssystem muligvis hjælpe på.

Fogs ansatte, der kommer i kontakt med det nye system, bliver også påvirket af dette projekt. Hvis det er en succes, kan det resultere i udvidet forretning for Fog eller måske lønforhøjelser for de enkelte ansatte – og hvis ikke, kan det i værste fald resultere i fyringer og skade Fogs omdømme og omsætning. Der er altså mange mennesker, der afhænger af projektet, og hvis fremtid kan påvirkes af dets succes.

Vi har sorteret alle involverede personer/grupper i en lille tabel som led i vores interessentanalyse. Som man kan se i tabellen[2], er alle involverede opdelt i 4 kategorier: *ressourcepersoner, gidsler, grå eminencer* og *eksterne interessenter.*

Vi synes, at placeringen af de fleste interessenter giver sig selv, taget i betragtning af uddybningerne ovenfor, men her er nogle eksempler på, hvorfor vi har valgt at sætte bestemte interessenter ind i visse kategorier:

Både Fog selv og os (Bornit) er ressourcepersoner, eftersom vi har stor indflydelse på og også bliver påvirket af projektet. Fog kan komme med forslag og måske bede om ændringer til enkelte funktioner, og vi har magten til at levere. Vi er også begge påvirkede af projektet, da der som sagt både er omdømme og omsætning på spil.

Derimod er de tømrere, som Fog har kontakt til, hverken påvirkede eller i stand til at påvirke projektets udvikling. Man kunne selvfølgelig sige, at hvis det skulle gå helt galt med projektet og påvirke Fogs omdømme meget negativt, kunne det resultere i færre arbejdsmuligheder for tømrerne igennem Fog.

[2] Se bilag 1

Vores undervisere har også været vigtige at kunne konsultere under projektet, hvis vi har siddet fast nogen steder. Derfor har de haft stor evne til at påvirke vores projekts udvikling, men de er ikke ligefrem påvirkede af det, da de ikke er ansvarlige for, hvad vi gør med den information, de giver os.

Arbejdsgange – før & efter

Det nuværende system på Fogs hjemmeside til carportbestilling er ikke væsentligt anderledes end på vores. Vi har valgt at følge nogenlunde det samme visuelle design med vores produkt, da tanken var, at det nemt skulle kunne integreres på hjemmesiden.

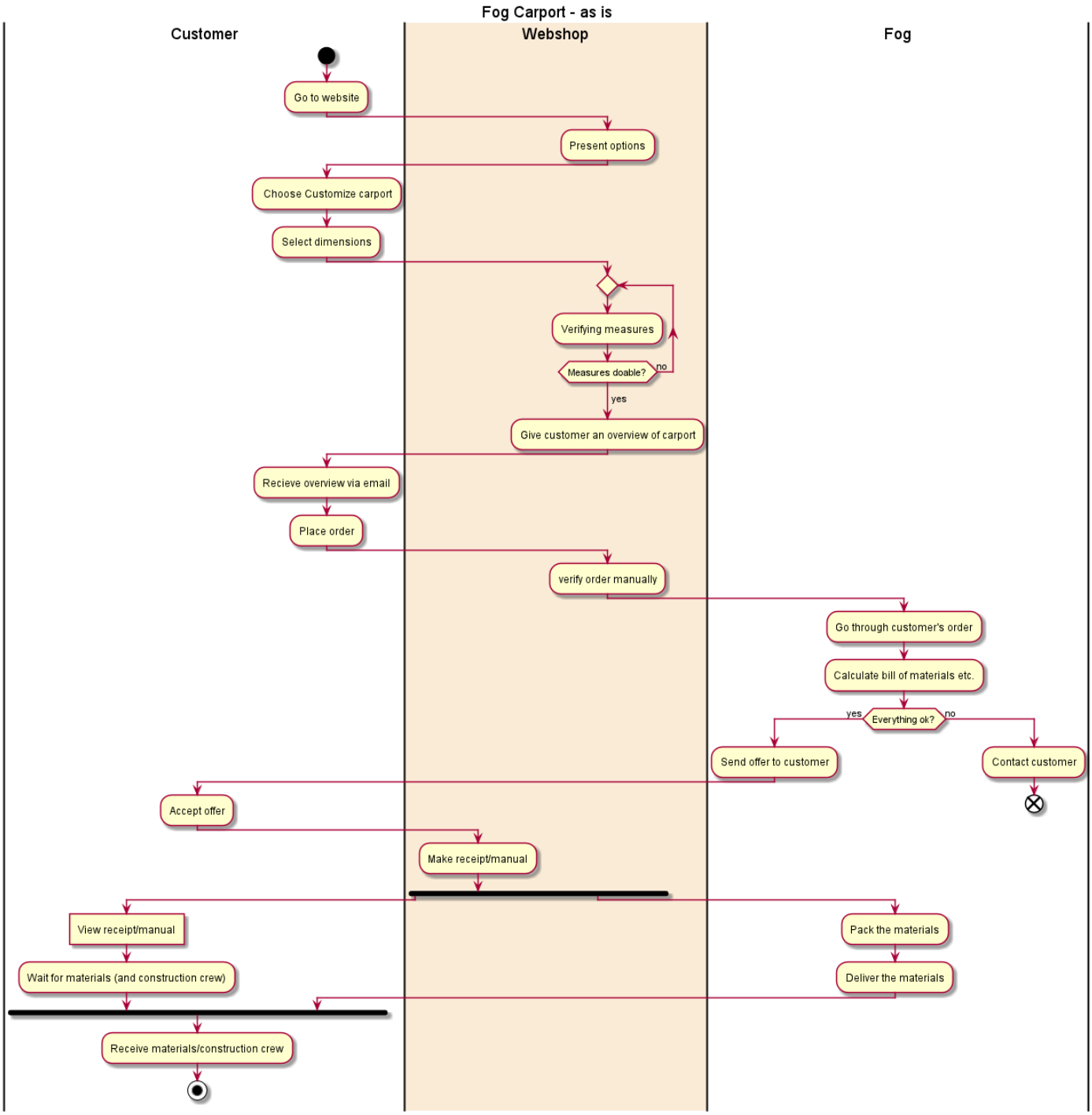
Den primære forskel er, at beregningsprocessen, som Martin skulle gå igennem, er blevet automatiseret, og også at kunden ikke behøver at kontaktes via email, da kunden bare kan logge ind og kigge på sin forespørgsel når som helst.

Nedenfor ses et aktivitetsdiagram over den nuværende sekvens af handlinger, der foregår, når en forespørgsel laves på Fogs hjemmeside:

Som man kan se nedenfor på figur 1, starter kunden med at gå ind på hjemmesiden og vælge, hvilken type carport de vil have – der er også standardcarporte på hjemmesiden, men her snakker vi udelukkende om carporte med selvvalgte mål. Herfra vælges dimensionerne på carporten og eventuelt på redskabsrum, og så laves der en forespørgsel, såfremt målene lever op til visse regler (eksempelvis må skuret ikke være større end selve carporten).

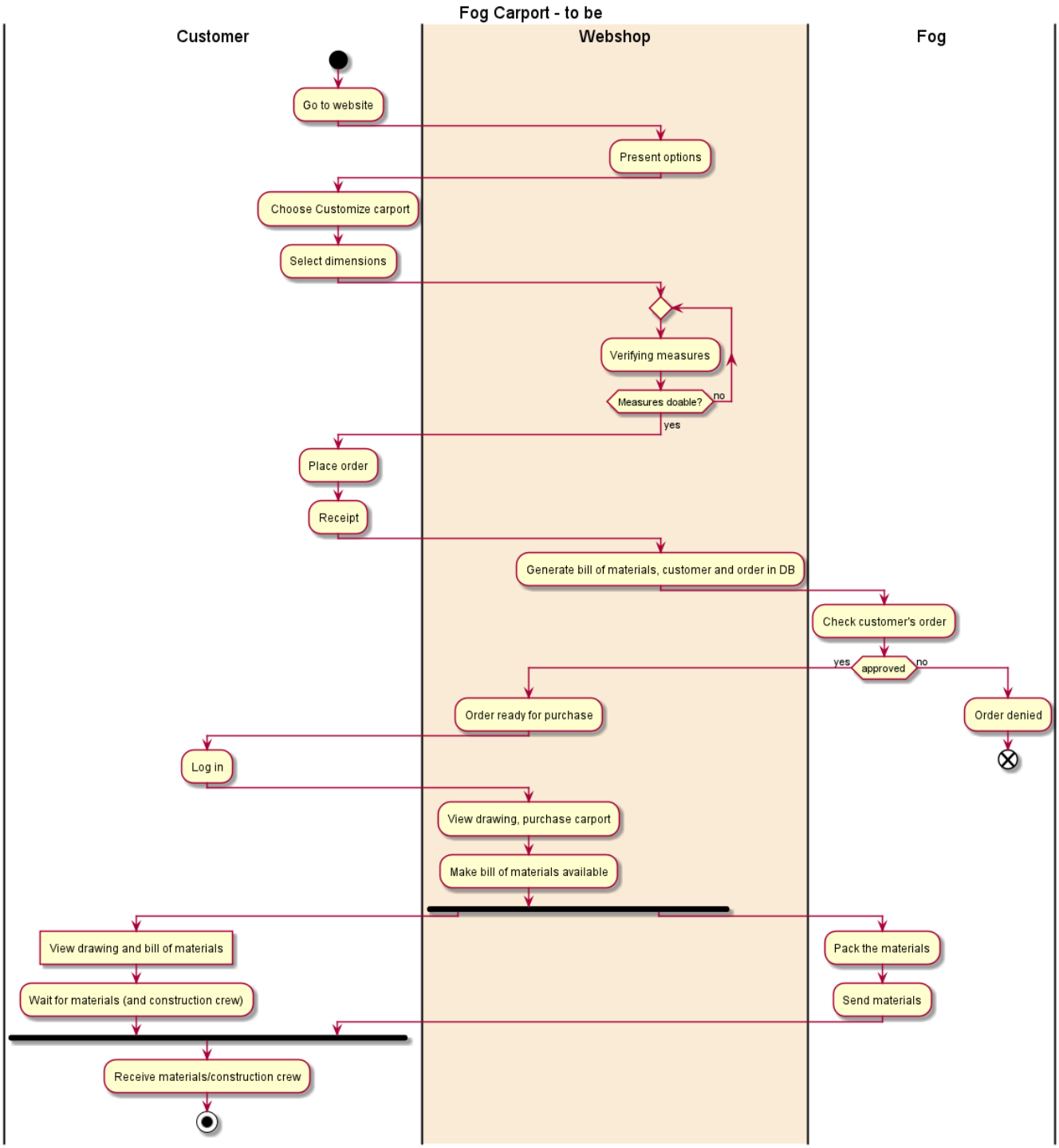
Når forespørgslen er lavet, sendes en email til Martin (en af Fogs ansatte). Han tager så målene fra kundens forespørgsel og kører dem selv gennem et regneprogram, hvor han vælger bl.a. tagpladetype og andet og til sidst kan få lavet en tegning og bestemme prisen, som han så kan sende til kunden. Han kan også i visse tilfælde kontakte kunden, hvis han har nogle forslag til ændringer i kundens forespørgsel.

Når alt så er beregnet og sendt til kunden, kan de acceptere tilbuddet. Hvis kunden accepterer tilbuddet, sendes en stykliste også men aldrig før betaling. Herfra pakkes og sendes materialerne så, og så venter kunden på at modtage materialerne og eventuelt hyrede tømrere, som kunden kan have fået kontakt til gennem Fog.



Figur 1: as-is aktivitetsdiagram

Her er så aktivitetsdiagrammet for vores hjemmeside, ”to be” versionen:



Figur 2: to-be aktivitetsdiagram

Som nævnt er figur 2 ikke væsentligt anderledes end den forrige figur. Starter man i venstre hjørne ligesom med det forrige diagram, kan man se, at kunden bevæger sig igennem bestillingsprocessen på samme vis som før. Den første markante ændring er, at når kunden har lavet sin bestilling, genereres der stykliste (bill of materials), kunde og ordre i vores database med kundens informationer og valgte mål på carport (og skur).

Da alle disse er generede allerede fra bestillingen, dvs. før et køb overhovedet er foretaget, kan kunden nu logge ind med sit ordre ID og telefonnummer og se sin bestilling og dens nuværende status samt de genererede tegninger af carporten. I mellemtiden kan Martin og lagermedarbejdere logge ind og kigge på alle ordrer, der er relevante for dem. Martin ser alle afventende ordrer, mens lagermedarbejdere ser bestilte ordrer, så de ved hvilke ordrer, de skal pakke materialer til.

Når Martin vælger at godkende en forespørgsel, får kunden mulighed for at købe carporten når de logger ind igen. Hvis han vælger at afvise en forespørgsel, slettes den genererede stykliste fra databasen, og så kan kunden logge ind og se, at ordren er blevet afvist og eventuelt vælge at lave en ny ordre derfra. Efter købet er foretaget, bliver styklisten tilgængelig for kunden, som de så kan se på.

Herfra kigger lagermedarbejderne på deres side, pakker materialerne til den enkelte ordre og afsender den. Kunden kan så se på sin side, at materialerne nu er afsendt, og så venter de på, at de ankommer.

SCRUM user stories

I starten af projektforløbet fik vi lavet vores user stories, som er kundens ønsker for produktet. Vi har brugt hjemmesiden taiga.io til at organisere og holde styr på vores arbejde og dets fordeling. Vi er nået igennem langt de fleste user stories, og her er et par eksempler:

US-2 (#2 på taiga.io): Som kunde ønsker jeg at kunne bestille en carport med fladt tag, angive højde og bredde og vælge tagtype for at kunne afgive en bestilling.

Vi brugte taiga.io's pointsystem til at afsætte tid til hver user story, og vi betragtede hvert point som 1 time. Til denne user story afsatte vi 8 point, da vi i de tidlige stadier (første sprint) ville være sikre på, at vi afsatte rigeligt med tid til at komme igennem den. Vi vidste på forhånd, at der skulle laves noget backend og frontend, bare ikke hvor længe det ville tage.

Med det in mente, splittede vi den op i tre tasks i vores første sprint. De tasks var:

- Lav en mockup i Adobe XD
- Backend bestillingsside
- HTML Bootstrap bestillingsside

Målet var, at vi indenfor den afsatte tid kunne levere på det, som user story'en specificerede, og vise det med en demo, hvor vi skulle kunne:

- Vise forside, angive højde + bredde, trykke bestil og få en kvittering.

Vi lavede først mockup'en sammen for at blive enige om udgangspunktet for vores design. Når vi var tilfredse med den og havde en generel struktur sat op for vores navigation gennem de forskellige sider, splittede vi os i to mindre grupper og begyndte på backend og frontend i IntelliJ.

Frontend involverede mest at få sat den generelle struktur op for vores jsp-sider ved at lave header og footer og derefter begynde på selve jsp-siderne til denne user story – flatorder.jsp og receipt.jsp.

På dette tidspunkt var databasen ikke opsat, så vi hardkodede værdierne i drop-down boksene og på kvitteringssiden, da vores mål bare var at få strukturen sat op, som vi så senere kunne gøre dynamisk ved at sætte relevante værdier på applicationScopet, blandt andet ved at få databasen forbundet.

Derfor var backend delen heller ikke omfattende. Der blev sat en klasse op, FlatOrder, som kunne modtage brugerens indtastede værdier, vi så senere kunne bruge til både beregning af carportmaterialer og indsætning af kunder, ordrer samt styklister i databasen.

Vi nåede i mål med det hele indenfor den afsatte tid og kunne i demoen vise det, vi havde planlagt. Her er så et andet eksempel:

US-10 (#15 på taiga.io): Som kunde ønsker jeg at modtage en simpel (og fuld) stykliste for carport m. fladt tag for at kunne se, hvad den består af.

Denne user story var en del af vores andet sprint og var langt mere omfattende end den forrige, og det vidste vi på forhånd. Der skulle både være en funktionel beregnerklasse med alle de nødvendige regnemetoder, de nødvendige mappers og selvfølgelig backend og frontend til at kunne få styklisten vist for den enkelte kunde. Derfor afsatte vi 25 point til den og splittede den op i 6 tasks:

- Færdiggør backend til type 1 carport stykliste
- Lav frontend + backend til stykliste (så kunden kan se sin stykliste på myorder.jsp siden)
- General exceptionhandling (på flatorder + evt. resten af websitet) + lav fejlbeskeder så kunden forstår problemet
- Lav adminside (frontend + backend) hvor man kan ændre status på en ordre
- Lav login til forhandler/medarbejdere
- Lav mapper til carport_parts + tilføj type 1 materialer i carport_parts i databasen

Vores mål for denne user story var, at vi i demoen kunne:

- Logge ind som kunde og se en hardkodet stykliste (og senere en dynamisk stykliste, der er specifik for den enkelte ordre)

Som man kan se, har alle vores tasks ikke været lige relevante for denne user story. Vi valgte at sætte nogle flere, ikke direkte forbundne tasks på den, fordi vi gerne ville fordele arbejde rundt mellem teammedlemmerne og ikke mente, at det var effektivt at have alle 5 på opbygningen af backend og frontend til styklisten. Der var for eksempel noget fejlhåndtering i bestillingsprocessen, hvor man på tidspunktet kunne trykke ”bestil” uden at udfylde registreringsfelterne, uden at vælge carportlængde og ~bredde og uden at vælge skurlængde og ~bredde – hvilket naturligvis ville få websitet til at gå ned, da tomme felter ikke ville kunne blive parset i backenden.

Adminside og login til admin/medarbejdere valgte vi også at lave i denne forbindelse, da vi skulle have sat det op, for at vores ordresystem ville fungere korrekt. Vi kører med et statussystem, hvor kunden får adgang til mere information og flere muligheder afhængigt af ordrens status, som bliver ændret af bl.a. admin.

Vi valgte også at lave et separat loginsystem til medarbejdere, da vi med kundens loginsystem valgte at bruge telefonnummer og ordrenummer som loginoplysninger. Derudover fik vi også lavet en mapper til carport_parts tabellen i databasen og fik sat relevante materialer derind, da mapperen skulle bruges til de primære beregnermetoder (til at sætte beskrivelser på styklistens materialer), der beregner al nødvendig data til en stykliste.

Derefter fokuserede vi på at få sat jsp-siden til styklisten op og valgte at følge samme opsætning som styklisten på PDF'en, vi havde fået til opgaven, så vi havde to kategorier til materialerne: "Træ og tagplader" og "Beslag og skruer".

Vi hardkodede bare nogle materialer ind for at vise, hvordan det skulle se ud, og fordi vores beregner ikke var færdig på tidspunktet. Til demoen kunne vi vise, hvad vi havde regnet med – nemlig at kunden kunne logge ind og se en stykliste, der senere skulle gøres dynamisk og være specifik for den enkelte ordre.

Tabel over alle user stories

De to eksempler, der er givet, er bare et par udpluk, der er blevet brugt til at beskrive vores arbejdsproces i større detalje. Vi har lavet en tabel over alle vores user stories[3] – både dem vi nåede, dem der blev løst indirekte som led i en anden user story, og dem vi ikke nåede. Vi har lavet dem i et Google spreadsheet[4], og brugt dem hovedsageligt på taiga.io.

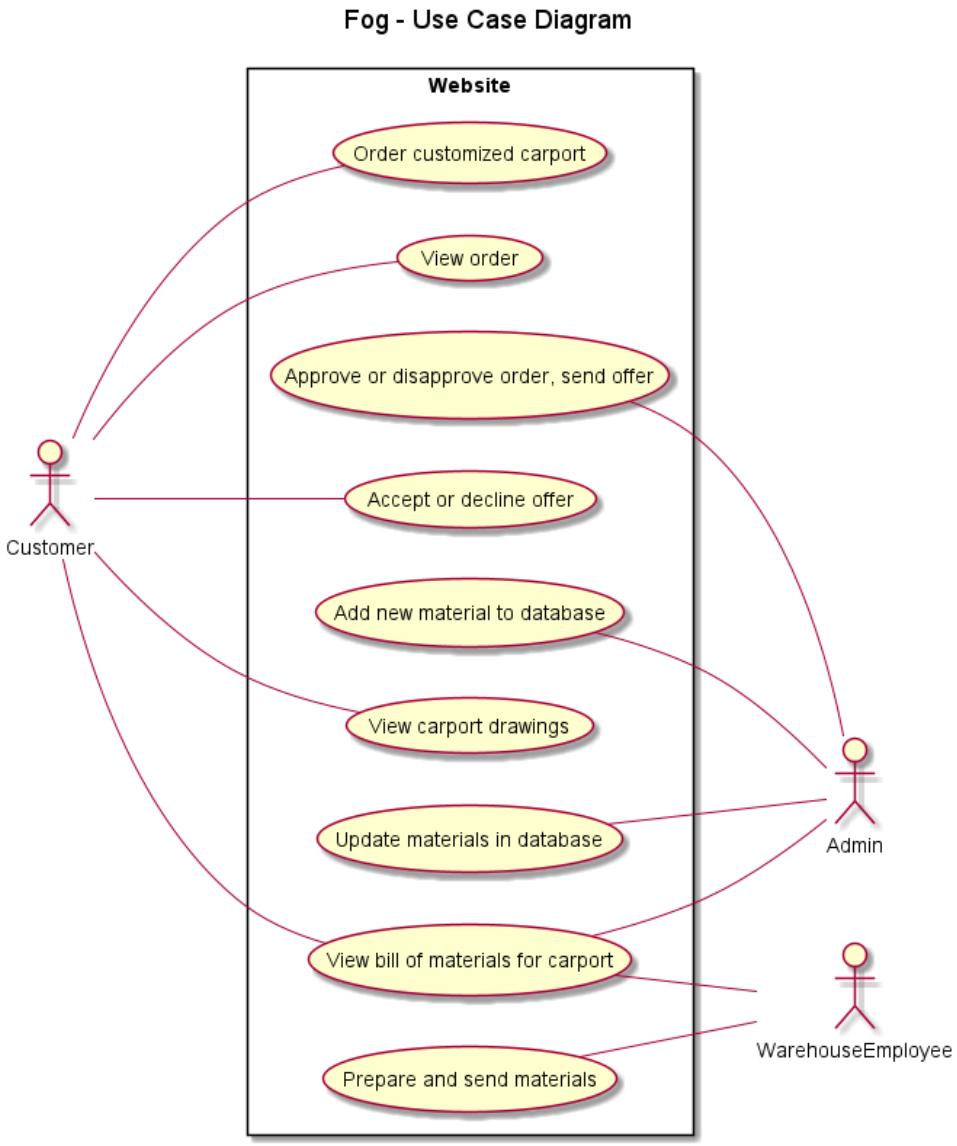
Som man kan se i tabellen, har vi nået de fleste user stories, og der er også enkelte user stories på taiga.io, som ikke er på vores spreadsheet, og de indgår derfor ikke her. Det er også vigtigt at bemærke, at vi ikke har arbejdet igennem vores user stories kronologisk. Vi har hoppet lidt rundt fra den ene til den anden, og tabellen er derfor kun opsat som den er for en ordens skyld.

Der er også demoer til de user stories, som vi ikke nåede – de er lavet under rapportskrivningen og er bare en forestilling om, hvad man ville have gjort for at vise de relevante funktioner, hvis man havde lavet dem.

[3] Se bilag 2
[4] Se User Stories i Yderligere Links

Use Case diagram

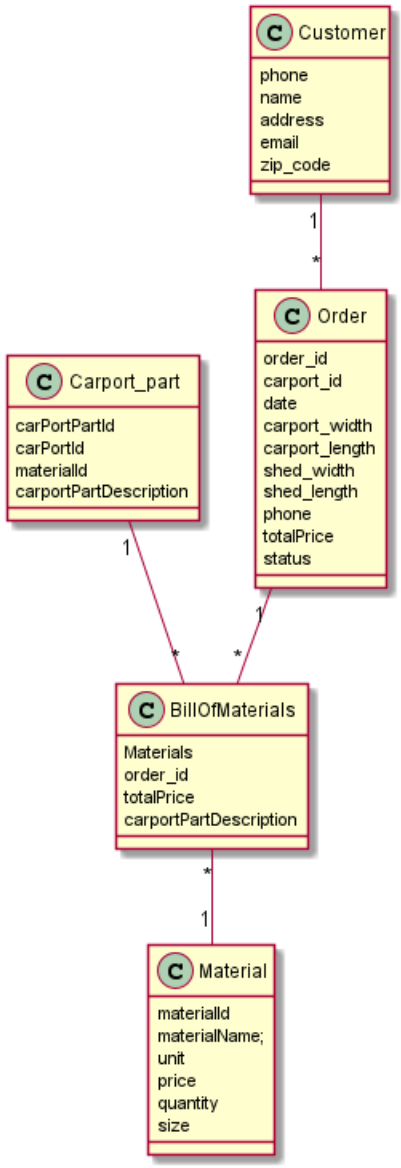
Under projektforberedelsen udarbejdede vi et use case diagram for bedre at kunne forstå de involverede brugeres forhold til de forskellige funktioner, vi havde planer om at implementere. Det ses nedenfor:



Figur 3: Use case diagram

Som man kan se ovenfor i figur 3, er der tre aktører, dvs. tre forskellige typer brugere, der skal bruge websiden. Kunden skal kunne bestille en carport med selvvalgte mål, følge sin bestilling og enten acceptere eller afslå det givne tilbud. Derudover skal man som kunde kunne se en tegning samt en stykliste over den bestilte carport.

Admin skal kunne godkende eller afvise forespørgsler og indsætte nye materialer eller opdatere de eksisterende materialer i databasen. Admin skal desuden kunne se den specifikke stykliste for hver ordre i forbindelse med godkendelse. Endelig skal lagermedarbejdere kunne se stykliste for hver ordre for at kunne pakke de rigtige materialer til en ordre og senere hen afsende, når dette er fuldført.



Figur 4: Domænemodel

Domæne model

Til venstre ses en domænemodel, figur 4, som blev udarbejdet, før nogen egentlig kodning fandt sted. Formålet med dette er at få dannet et overblik over de konceptuelle klasser.

Customer har en 1 til mange relation til Order, da én kunde kan have mange ordrer.
En kunde udfylder personlige oplysninger som telefonnummer, navn og adresse som identificerer den enkelte kunde.
Kunden kan lave en ordre, der indeholder specifikationer for carporten såsom længde og bredde, og hvilken type carport det skal være.

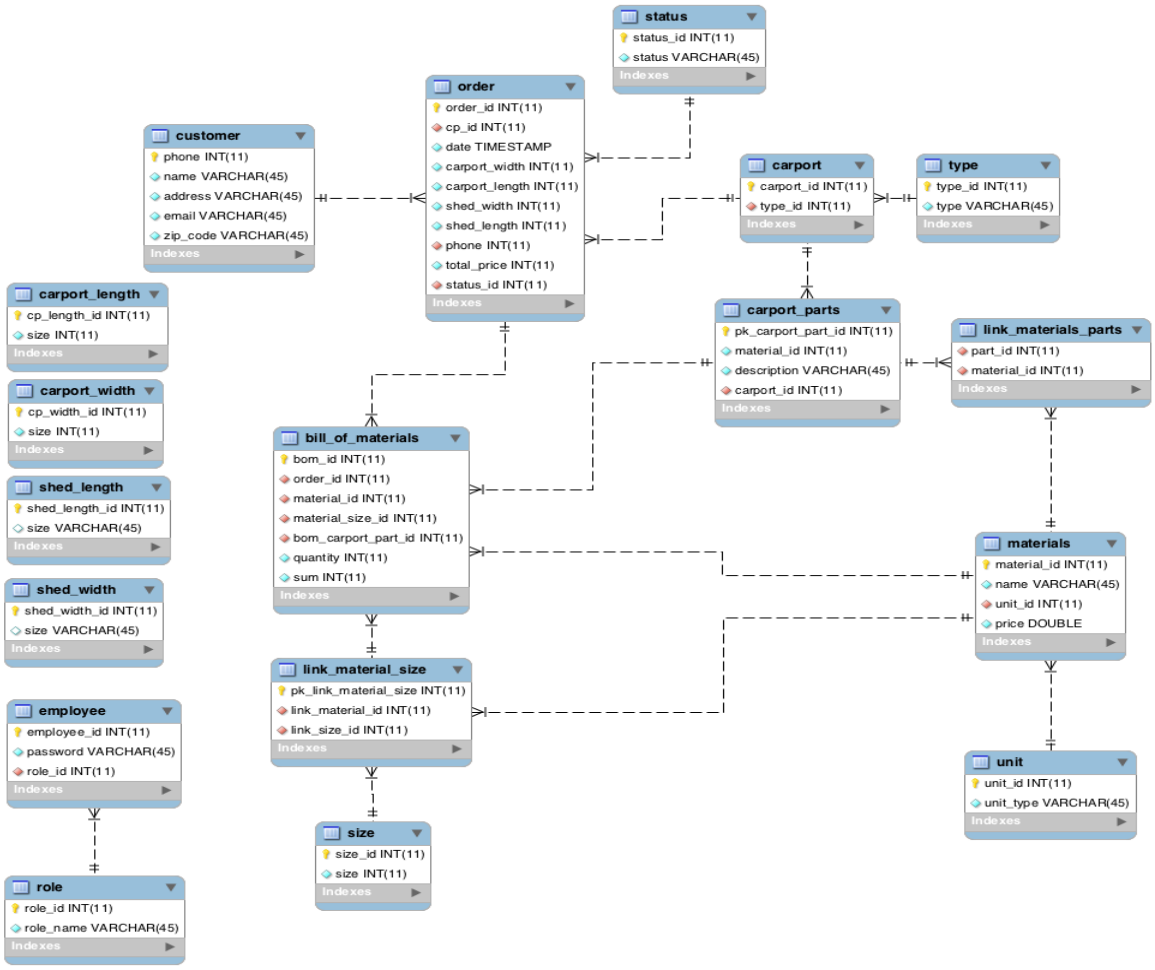
1 ordre består af mange materialer og carportdele.
BillOfMaterials kan opfattes som ordrelinjer, hvor hvert materiale i arraylisten ”materials” udgør en række i bill_of_materials tabellen i databasen.

Da en carport består af mange dele, vil der være en 1 til mange relation til BillOfMaterials.
Carportdelene bliver valgt ud fra hvilken carport type, der er bestilt.

Material har en 1 til mange relation til BillOfMaterials, da der bruges ét materiale til mange forskellige dele af carporten.

EER-Diagram

Et *enhanced entity-relationship* (**EER**) diagram er en udvidet version af et *entity-relationship* (**ER**) diagram. Et ER-diagram viser relationer mellem forskellige enheder. I denne sammenhæng vil man sige, at et ER-diagram viser relationerne mellem de forskellige tabeller, vi har i databasen. Disse diagrammer er gode hjælpemidler i udformningen og forståelsen af en database, blandt andet i forbindelse med normalisering af databasen. *Normalisering vil sige, at man så vidt som muligt fjerner alle former for redundans med det formål at gøre databasen effektiv, konsistent og let at vedligeholde*[5].



Figur 5: EER diagram

[5] Se PDF i Yderligere Links

Ovenfor ses EER-diagrammet (figur 5) over vores database til Fog og for at kunne forklare dette diagram på den mest muligt overskuelige måde, har vi valgt at gennemgå det tabel for tabel. Under hver tabel giver vi et overblik over, hvilke relationer den har til andre tabeller.

Customer-tabel:

Hvis vi starter fra customer-tabellen, har vi en 1 til mange relation til order, da én kunde kan have mange ordrer. *Phone*, som er kundens telefonnummer, er vores primary key; denne er ikke auto-incremented og bliver brugt som en *foreign key* i order-tabellen. *Phone* bliver på den måde linket mellem kunde og ordre, som sikrer, at kundens ordre altid hænger direkte sammen med selve kunden gennem kundens telefonnummer.
Customer-tabellen lever ikke helt op til 3. normalform, da vi burde have lavet en tabel for sig selv til *postnummer* og *by*. På nuværende tidspunkt indskriver kunden by og postnummer som en tekststreng i samme felt. Det blev vi enige om at lade ligge, da vi ikke følte, det var relevant for opgaven.

Order-tabel:

Order-tabellens *primary key* er order_id. Dette nummer er unikt for hver ordre og bliver automatisk genereret ved oprettelse af en ordre. Udover de praktiske informationer, som carportens mål, pris og dato for bestilling, indeholder order-tabellen tre fremmednøgler. Vi har tidligere nævnt "phone". Den anden fremmednøgle er *status_id*. Order-tabellen har en mange til 1 relation til status-tabellen.
Den tredje og sidste fremmednøgle i order-tabellen er *cp_id*. Denne nøgle linker ordren til carport-tabellen, som bruges til at afgøre carport type. Her finder vi også en mange til 1 relation.

Status-tabel:

Da vi har valgt, at kunden skal kunne følge status på sin ordre, har vi lavet en status-tabel, som har en primary key og en status streng. Denne primary key bruges som en foreign key i order-tabellen.

Carport-, Type- og Carport_parts-tabel:

Carport-tabellen består af en primary key, *carport_id*, og har en fremmednøgle, som er *type_id*. I type-tabellen finder man to forskellige typer: Type 1 og type 2. Typen henviser til, om carporten er med skur (type 1) eller uden skur (type 2). Dette gør sig videre gældende, når man linker til

carport_parts-tabellen. I carport_parts finder man “opskriften” på en carport. Denne er forskellig, alt efter om det er en carport af type 1 eller type 2.

Bill_of_materials-tabel:

Bill_of_materials-tabellen er den tabel, som indeholder alle materialer, der skal bruges til den enkelte carport. Order_id er vores foreign key fra order-tabellen. Fra order-tabellen til bill_of_materials-tabellen er der en 1 til mange relation, da én ordre består af mange bill_of_materials.

Material_id er vores foreign key fra material-tabellen, som refererer til de bestemte materialer, der skal bruges til at lave den enkelte carport. Den har en mange til 1 relation, da ét materiale bruges til mange bill of materials.

Bom_carport_part_id er vores foreign key fra carport_parts-tabellen, som bruges til at finde beskrivelsen til hvert materiale, der bruges til at afgøre formålet for hvert materiale til en carport. Derudover består tabellen af *quantity*, hvilket er antallet af det enkelte materiale. Sum er materialets pris ganget med antal for hver bill of materials.

Materials-tabel:

Materials-tabellen indeholder alle materialer, vi har på lageret. Til at starte med havde vi i denne tabel alle længder på de forskellige materialer men for at holde 3. normalform, lavede vi en tabel til længder/størrelser med en link tabel mellem materials og size, eftersom der er en mange til mange relation mellem de to tabeller, da ét materiale fås i forskellige længder. Ydermere har vi en unit-tabel, som indeholder de forskellige måleenheder, f.eks. “stk.” og “pk.”.

Carport_length, Carport_width, Shed_length, Shed_width:

De enkeltstående tabeller, carport_length, carport_width, shed_length og shed_width, har vi oprettet i databasen for at kunne lave dynamiske drop-down lister til valg af bredder og længder på carport og skur. Ved at gøre det på denne måde, i stedet for at hardkode listerne, er det nemmere at justere valgene senere hen.

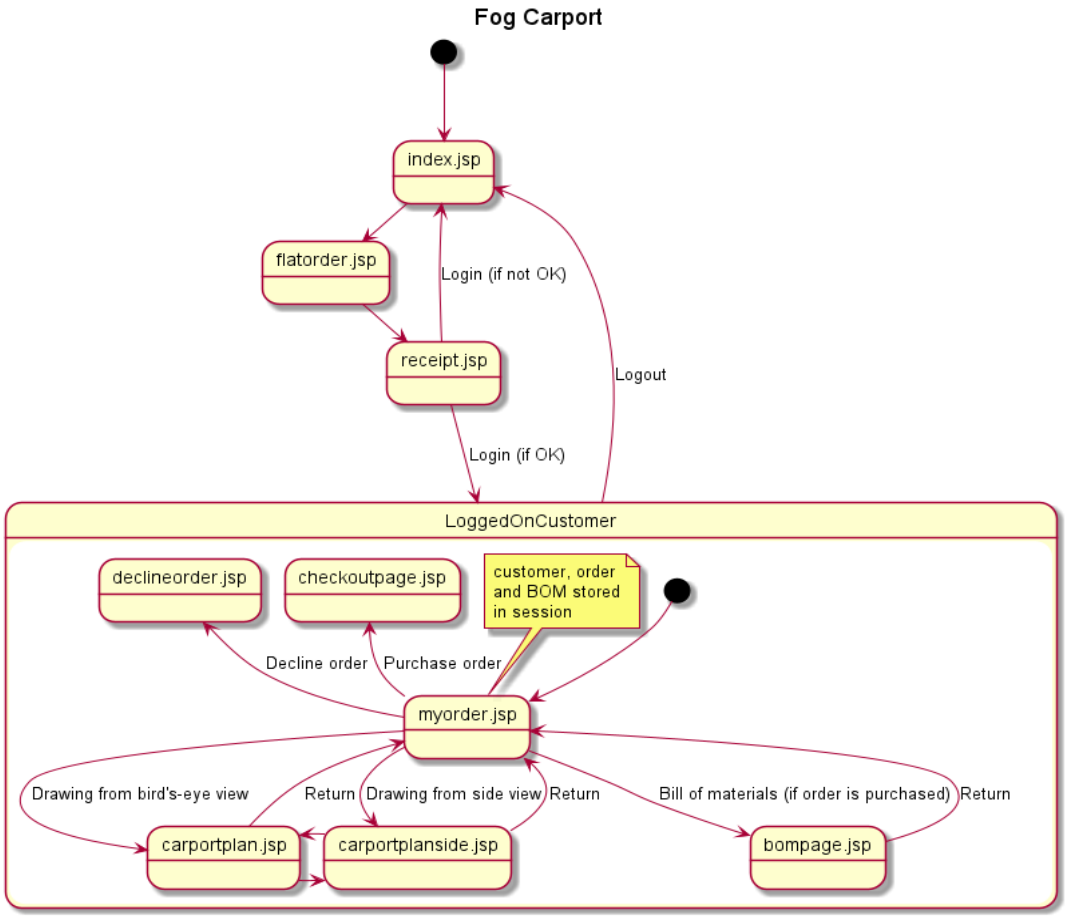
Employee- og Role-tabel:

Employee-tabellen indeholder employee_id, password og role. Role er en foreign key og linker tabellen sammen med role-tabellen. Her finder vi en mange til 1 relation. En ansat kan have en af følgende roller: Admin eller warehouse.

Disse to tabeller gør det muligt for enten administrator eller en lageransat at logge ind på siden. Hvis det er en admin som logger ind, vil han/hun kunne se alle ordre med status_id 1. Det vil sige en ordre, en kunde netop har placeret. Tanken er, at admin så gennemgår denne ordre, giver et pristilbud og trykker godkend. Lageret vil så kunne logge ind og se ordre med status_id 3. Dette er ordrer både admin har godkendt, og kunden har betalt for. Lageret kan på denne måde se hvilke ordrer, der skal pakkes og sendes og videregive besked til kunden, når ordren er afsendt.

Navigationsdiagram

Formålet med et navigationsdiagram er at illustrere, hvordan man navigerer rundt på for eksempel en hjemmeside. Boksene repræsenterer de individuelle sider. Det er lavet sidst i arbejdsprocessen for at danne et visuelt overblik over de forskellige muligheder, et kundelogin giver adgang til.



Figur 6: Navigationsdiagram

Navigationsdiagram (figur 6) lavet specifikt med udgangspunkt i customer, eftersom det indeholder flest muligheder mht. omdirigering. På adminsiden er der ingen redirect til andre sider på nær logout, som blot fører til forsiden (index.jsp). Admin kan blot ændre i prisen og godkende eller slette en ordre, men visuelt sker der ikke noget.
Fra siden for lagermedarbejdere er det dog muligt at blive videreført til bompage.jsp, men der sker intet usædvanligt, som ikke allerede er illustreret på det ovenstående diagram. De kan desuden afsende en ordre, men igen er det bare en ændring, der sker i databasen.

Da det ikke er synligt på figur 5, skal det nævnes, at hele hjemmesiden benytter sig af en fælles navigationsbar. Den er altså ens for alle, uanset om man er logget på eller ej samt uafhængig af rolle (kunde/admin/lager). Dens elementer er statiske, forstået på den måde, at de altid er tilgængelige. Følgende funktioner er til stede:

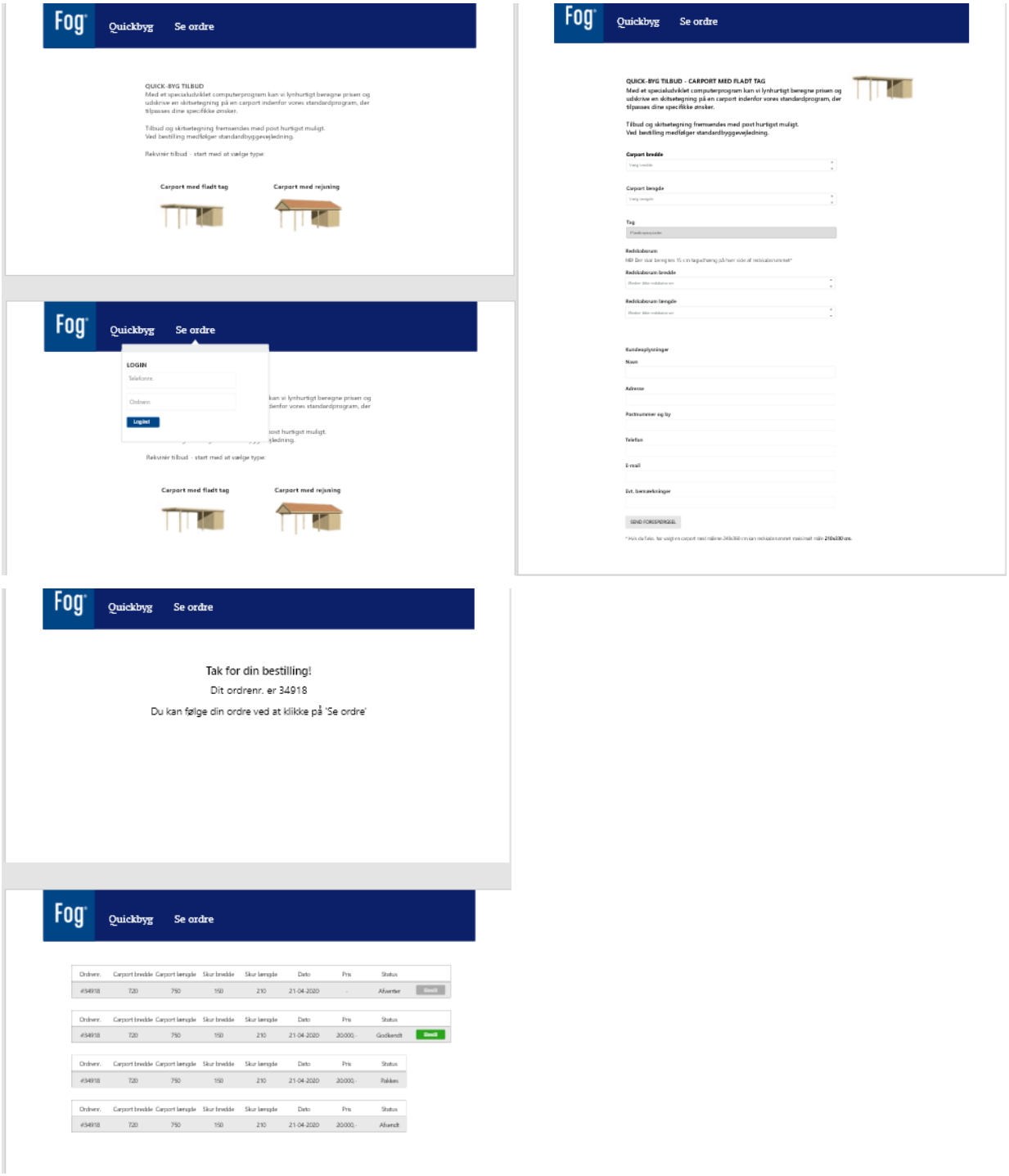
1. **QuickByg:** Returnerer brugeren til forsiden/index.
2. **Se ordre:** Her kan man logge ind som kunde, hvorefter man omdirigeres til myorder.jsp
3. **Forhandler login:** Som navnet lyder, er dette et login særligt til forhandleren herunder admin og lagermedarbejdere. Afhængigt af employee ID føres man til enten adminpage.jsp eller warehousepage.jsp.

FrontController fungerer som vores servlet, der, i samarbejde med Command klassen, håndterer al navigation mellem de forskellige sider.

Mockups

Før nogen egentlig kodning fandt sted, lavede vi en mockup i Adobe XD, så vi nemmere ville kunne visualisere hjemmesidens layout, når frontend skulle skrives. Det sikrer, at hele gruppen har et fælles, ens billede i hovedet af det færdige produkt.
Der er naturligvis visse afvigelser fra disse skabeloner og den færdiglavede hjemmeside, og vi havde endnu ikke en fuldstændig klar vision om præcis hvilke jsp-sider, den ville komme til at bestå af.
Vi valgte at holde os så tæt på den originale hjemmeside, https://www.johannesfog.dk/, som muligt med et clean, minimalistisk design. Selvfølgelig er der nogle udeblivende funktioner, som

ikke var væsentlige i vores projekt, og vi endte desuden med at tilføje et forhandlerlogin for at adskille kunder og ansatte.



Figur 7: Mockups

Arkitektur

For at løse opgaven blev vi opfordret til at bruge Kaspers skabelon. Denne skabelon er bygget op af en såkaldt trelags-arkitektur. Det vil sige, at vi har et præsentationslag *(PresentationLayer),* et logiklag (*FunctionLayer)* og et datalag (*DBAccess).*
Hvis man sammenligner trelags-arkitektur med klient-server arkitektur kan man sige, at præsentationslaget tilsvarer klientdelen , hvor data- og logiklaget er en opdeling af serverdelen[6].

Hvert af disse lag har forskellige funktioner.
I *præsentationslaget* ligger de java klasser, der holder funktionaliteten til vores JSP-sider. Det er også her alt input fra brugeren først kommer, og information fra og til brugeren behandles og præsenteres. Inputtet og informationen bliver så sendt til og fra *logiklaget*, hvis der anvendes mapper metoder.

I logiklaget er der blandt andet klassen LogicFacade, som sørger for, at der ikke er nogen direkte kontakt mellem præsentationslaget og datalaget. På den måde er vores database mindre sårbar. Logiklaget modtager som sagt information fra præsentationslaget, og herfra kan den hente information fra databasen gennem datalaget, evt. bearbejde informationen og sende den tilbage til præsentationslaget.

I *datalaget* ligger den kode, der kommunikerer direkte med vores database gennem mapper klasser, der indeholder specielle metoder, der kan interagere med databasen. Disse metoder sender hver én eller flere SQL sætninger op til databasen, som bruges til at foretage CRUD operationer, såsom nedhentning, indsættelse og opdatering af data mm. Metoderne kaldes via logiklaget, hvor man kan foretage beregninger og andre manipulationer på data, og sendes så videre til præsentationslaget, og evt. tilbage til datalaget (eksempelvis hvis der skal opdateres eller indsættes data).

Singleton pattern og Command pattern er to andre centrale dele af skabelonen. Singletonen findes i Connector klassen i DBAccess (datalaget), og sørger for, at der kun oprettes én forbindelse til databasen ad gangen ved, at Connector klassen kun instantieres én gang.
Command klassen findes, som nævnt i indledningen, i præsentationslaget, og sørger for, at

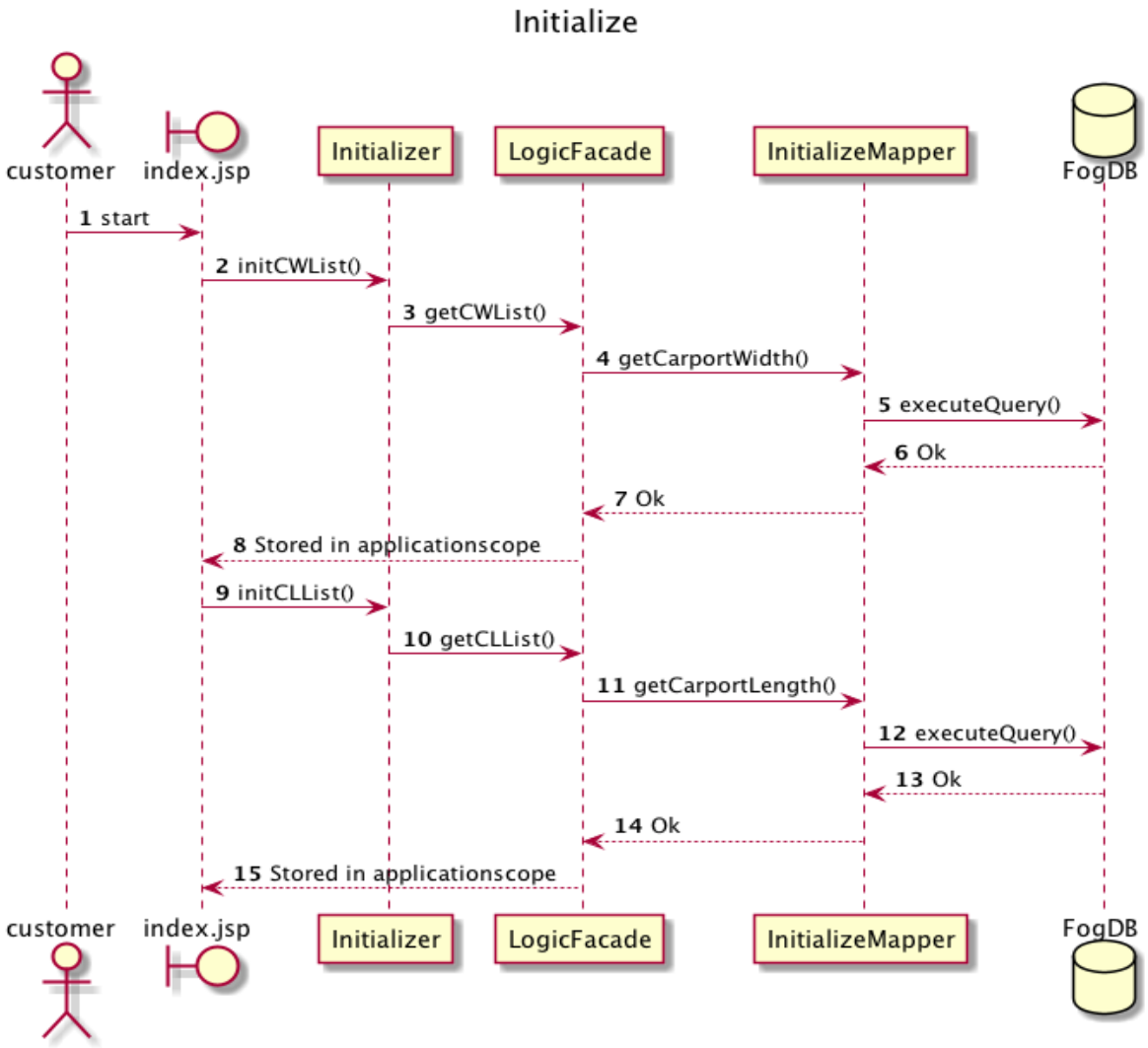
[6] Se Kilde for afsnit om arkitektur i Yderligere Links

requests bliver behandlet korrekt i FrontController klassen, dvs. at brugeren sendes til de rigtige jsp sider, og at disse sider får den funktionalitet, der hører til dem.

Sekvens Diagrammer

I dette afsnit har vi vedlagt tre sekvens diagrammer. De tre scenarier, vi har valgt, er Initialize, Ordre og Login. Sekvensdiagrammerne er alle lavet sidst i arbejdsprocessen.

Initialize



Figur 8: Sekvensdiagram over Initialize

Ovenfor vises sekvensdiagrammet (figur 8) over indlæsning af carportlængder og ~bredder til drop-down menuerne.

(**1**) Når index-siden indlæses, kaldes initCWList().

(**2**) Hvis applicationScope for "CWList" er tomt, kaldes initCWList() i Initializer klassen.

(**3**) Initializer kalder getCWList() fra LogicFacade,

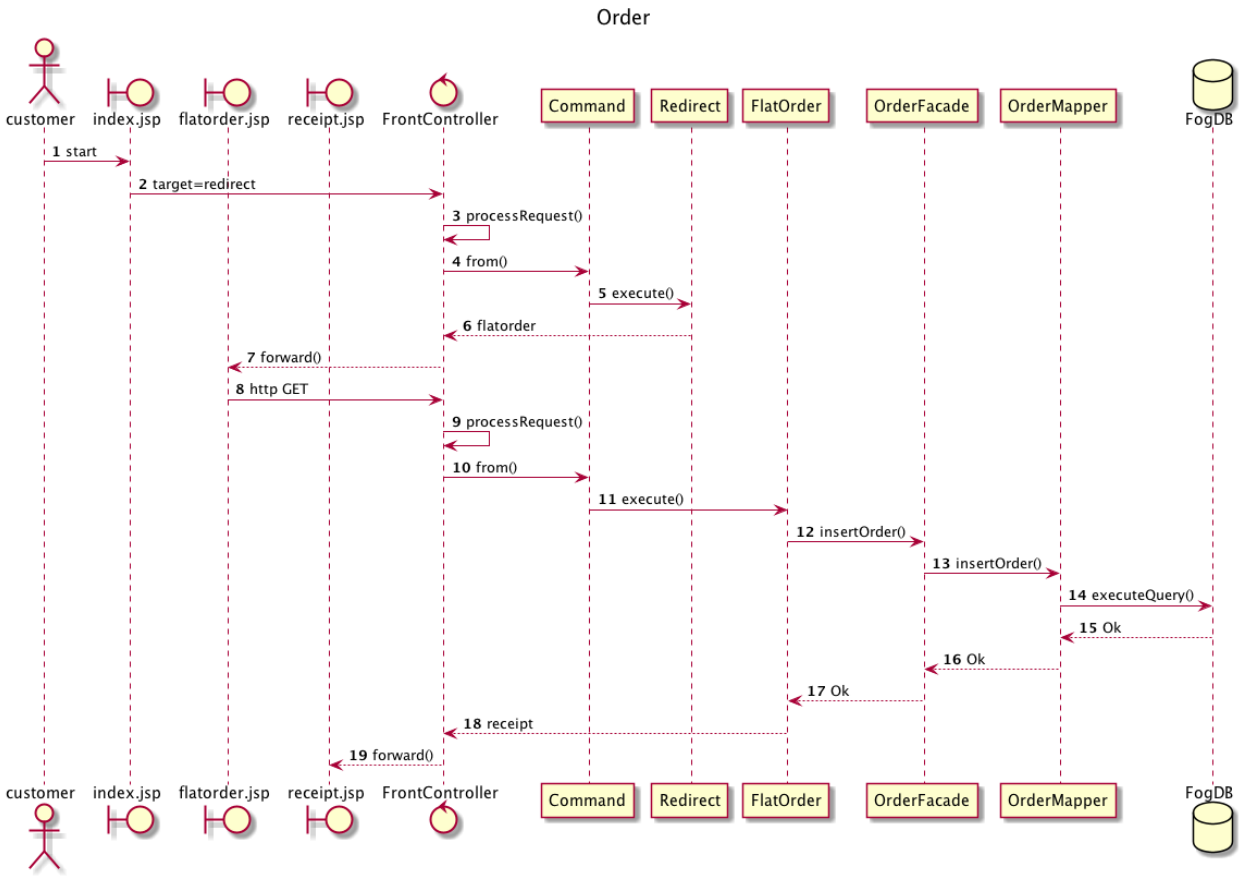
(**4**) Som kalder getCarportWidth() i InitializeMapper.

(**5**) Herefter eksekveres SQL-kommandoen vha. executeQuery().

(**6,7,8**) Hvis alt går, som det skal, bliver CWList gemt i applicationscope.

(**9-15**) Det er stort set den samme process, der sker i disse trin. Forskellen er blot, at i stedet for at hente CWList (carportbredder) ned, hentes CLList (carportlængder).

Order

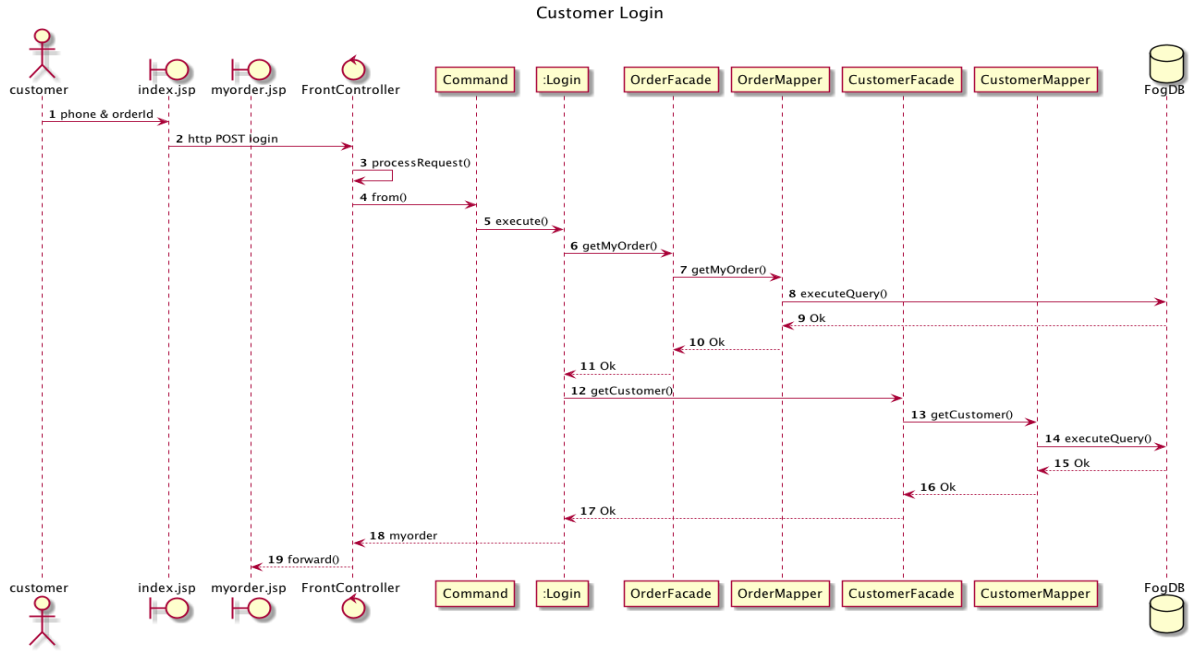


Figur 9: Sekvensdiagram over Order

Ovenfor vises et overordnet sekvensdiagram (figur 9) for en ordre af en carport med fladt tag.

(**1**) Kunden starter med at ramme index.jsp, når han eller hun besøger Fogs hjemmeside.
(**2**) Ved at klikke på billedet af den flade carport, kaldes FrontController vha. href (destinationslink).
(**3**) processRequest() tager sig af både doPost() og doGet() i FrontControlleren.
(**4**) FrontControlleren kalder Command klassen med from().
(**5**) Command klassen kalder execute() metoden i Redirect,
(**6**) der returnerer den korrekte destination (String) til FrontControlleren.
(**7**) FrontControlleren videredirigerer flatorder.jsp vha. metoden forward().
(**8**) Kundens indtastede data sendes via en HTTP GET til FrontControlleren, hvor (**9, 10, 11**) gentages på samme manér som før.
(**12**) Når kundens indtastede data er blevet behandlet i FlatOrder klassen, kaldes insertOrder() først igennem OrderFacade og efterfølgende (**13**) OrderMapper.
(**14**) insertOrder() eksekveres vha. executeQuery(), der fyrer SQL-sætningen af.
(**15, 16, 17**) Hvis alt går, som det skal, indsættes kunde, ordre og stykliste i databasen.
(**18**) FlatOrder returnerer til sidst strengen ”receipt”.
(**19**) FrontControlleren videredirigerer receipt.jsp vha. metoden forward().

Customer Login



Figur 10: Sekvensdiagram over Customer Login

Ovenfor vises sekvensdiagrammet for kundelogin (figur 10). I diagrammet kan man se, hvordan kunden logger ind for at se sin ordre. Kunden kan i realiteten logge på fra alle sider, men i diagrammet har vi valgt at tage udgangspunkt i index.jsp.

(**1**) Kunden indtaster loginoplysninger, telefonnummer og ordre ID.
(**2**) Det sendes videre til FrontController med en HTTP POST - doPost().
(**3**) processRequest() tager sig af doPost()-metoden (også doGet()),
(**4**) inden den vha. from() kalder Command klassen.
(**5**) Command kalder execute() i Login klassen,
(**6, 7**) der først kalder getMyOrder() i OrderFacade og derefter OrderMapper.
(**8**) Metoden executeQuery() udfører SQL-kommandoen, som indlæser ordredata fra databasen
(**9, 10, 11**) Hvis ikke der opstår nogen fejl, ryger det tilbage til Login, hvornæst
(**12, 13, 14**) det samme forløb skydes i gang med getCustomer(), gennem CustomerFacade og CustomerMapper, eksekveres vha. executeQuery(),
(**15, 16, 17**) og hvis OK, sendes dataen tilbage til Login og sættes på sessionScopet.
(**18, 19**) En destinations String bliver sendt til FrontControlleren, som sender brugeren til myorder.jsp.

Særlige forhold

Scopes

Alt, der hentes fra databasen gemmes i sessionScope - med undtagelse af bredde- og længdelisterne til carport og skur på bestillingssiden, der i stedet gemmes i applicationScope. RequestScope bruges hovedsageligt til fejlbeskeder.

Exception og logger håndtering

Vi bruger forskellige exceptions i projektet. De fire, der er værd at nævne, er SQLException, LoginSampleException, ClassNotFoundException og OrderException.
Alle exceptions bliver kastet til FrontControlleren, hvor der bliver logget og genereret fejlmeddelelser til brugeren eller os som udviklere.

SQLException

Vi bruger SQLException i alle mappers til at fange de SQL-fejl, der nu måtte opstå, og i FrontControlleren sættes der en fejlbesked på requestScope, som vises på de relevante jsp-sider, når en fejl finder sted.
Derudover logger vi fejlmeddelelsen som en “severe” fejl, da al kommunikation til databasen ses som en alvorlig fejl.

LoginSampleException

I projekt skabelonen, vi bruger som udgangspunkt, er der lavet en LoginSampleExeception, som vi valgte at bruge til kunde- og medarbejderlogin. Hvis en medarbejder eksempelvis forsøger at logge ind med et forkert employee ID eller adgangskode, kastes LoginSampleException. Dette sker i login() metoden i EmployeeMapper. Den bliver kastet tilbage til FrontController, der sætter en fejlbesked på requestScopet – f.eks. “Forkert ID eller Password”. Derefter bliver brugeren dirigeret tilbage til index-siden, hvor fejlmeddelelsen vises øverst på siden. Derudover bliver fejlen logget som en “info” fejl.

ClassNotFoundException

ClassNotFoundException bruges i Connector klassen og kastes hvis JDBC driveren, vi bruger, ikke kan findes når Class.forName() metoden kaldes i connection() metoden fra Connector klassen.

OrderException

OrderException er en custom exception, vi selv har lavet, som tager sig af brugerinput fejl, når der laves en bestilling.
Denne exception bliver kastet hvis der er fejl i brugerens input på bestillingssiden, f.eks. hvis brugeren har glemt at udfylde et felt. Vi har besluttet, at alle input felterne på bestillingssiden skal udfyldes, medmindre man trykker "Jeg er allerede registreret", og lavet nogle if statements, som eksempelvis sørger for, at kunden ikke vælger et skur, der er større end carportens mål.
OrderException bliver kastet til FrontController, som sætter fejlmeddelelsen på et requestScope og dirigerer kunden tilbage til ordresiden, hvor fejlen bliver vist. Fejlen bliver logget som en "info" fejl.

Logger

Næsten alle exceptions bliver fanget i FrontControlleren og logges. I loggeren har vi valgt at operere med to niveauer: "Info" og "severe".
Info bliver primært brugt til brugerinput fejl, eller hvis der sker en LoginSampleExeception.
Severe bliver brugt til de alvorlige fejl såsom SQLExceptions.
Enkelte steder har det været nødvendigt at lægge logger inde i selve klassen, f.eks. i Initializer klassen, som går indover FrontController. Hvis målene på carporte og skur ikke kan hentes fra databasen, bliver der logget en "severe" fejl.

Validering af brugerinput

Størstedelen af dette er håndteret ved hjælp af try-catch blokke, der i nogle tilfælde er kombineret med if-sætninger. Det benyttes hovedsageligt på bestillingssiden, da det er her, brugeren indtaster alle de nødvendige oplysninger. Som eksempel er der to forskellige fejl, der kan opstå ved input af telefonnummer: 1) Hvis der I stedet for tal bruges bogstaver, andre symboler eller ingenting, og 2) hvis ikke det er præcist 8 cifre langt. Begge håndteres af OrderException.
Valgmuligheder til mål på carport og evt. tilhørende skur er lagt i drop-down menuer, hvor de ligger på applicationScope, for at undgå at en kunde kan lægge ordre på en carport med urealistiske dimensioner.

Sikkerhed ifb. med login

Eftersom kundelogin kræver både et unikt ordre ID samt et telefonnummer, der altid vil være koblet på en specifik kunde, sikrer vi os, at der ikke opstår duplikater. Kunden vil derfor altid logges ind på den korrekte ordre, såfremt de har indtastet de rette legitimationsoplysninger.

Ellers mødes de af en fejlbesked. Det er dog muligt at have flere ordre på samme telefonnummer, men det er stadigvæk en unik kombination af et telefonnummer og et ordre ID.

Brugertyper

Brugere er opdelt i to tabeller i databasen. En employee-tabel, med en tilhørende rolle-tabel, og en customer-tabel. Vi har valgt at gøre det således fremfor at lave en enkelt tabel, der indeholder alle brugere, siden vi opererer med to forskellige slags login, som kræver hver deres form for legitimationsoplysninger.

Udvalgte kodeeksempler

insertOrder()

```
37    public static int insertOrder(Customer customer, Order order) throws OrderException, SQLException, ClassNotFoundException {
38
39        int generatedId = 0;
40        String sqlCustomer = "INSERT INTO customer (phone, name, address, email, zip_code) VALUES (?,?,?,?,?)";
41        String sqlOrder = "INSERT INTO `order` (cp_id, carport_width, carport_length, shed_width, shed_length, phone, total_price, status_id ) VALUES (?,?,?,?,?,?,?,?)";
42        Connection con = Connector.connection();
43        try {
44            try (PreparedStatement ps = con.prepareStatement(sqlCustomer, Statement.RETURN_GENERATED_KEYS)) {
45                con.setAutoCommit(false);
46                ps.setInt( parameterIndex: 1, customer.getPhone());
47                ps.setString( parameterIndex: 2, customer.getName());
48                ps.setString( parameterIndex: 3, customer.getAddress());
49                ps.setString( parameterIndex: 4, customer.getEmail());
50                ps.setString( parameterIndex: 5, customer.getZip_code());
51                ps.executeUpdate();
52                try (PreparedStatement ps1 = con.prepareStatement(sqlOrder, Statement.RETURN_GENERATED_KEYS)) {
53                    ps1.setInt( parameterIndex: 1, order.getCarport_id());
54                    ps1.setInt( parameterIndex: 2, order.getCarport_width());
55                    ps1.setInt( parameterIndex: 3, order.getCarport_length());
56                    ps1.setInt( parameterIndex: 4, order.getShed_width());
57                    ps1.setInt( parameterIndex: 5, order.getShed_length());
58                    ps1.setInt( parameterIndex: 6, order.getPhone());
59                    ps1.setInt( parameterIndex: 7, order.getTotalPrice());
60                    ps1.setInt( parameterIndex: 8, order.getStatus_id());
61                    ps1.executeUpdate();
62
63                    ResultSet idResultset = ps1.getGeneratedKeys();
64                    if (idResultset.next()){
65                        generatedId = idResultset.getInt( columnIndex: 1);
66                    }
67                } catch (SQLException e) {
68                        Log.severe( description: "insertOrder - rollback");
69                        e.printStackTrace();
70                        con.rollback();
71                }
72                con.commit();
73                con.setAutoCommit(true);
74            }
75        } catch (Exception e) {
76            Log.severe( description: "insertOrder "+ e.getMessage());
77            throw new OrderException("Der skete en uventet fejl i din ordre.");
78        }
79        return generatedId;
80    }
```

Eksempel 1

For at indsætte vores kunde- og ordreoplysninger i databasen, bruger vi insertOrder() i OrderMapper. Metoden har to parametre, som kræver, at den modtager et Customer objekt og et Order objekt.
Linje 40 og 41 er vores SQL-sætninger, som bruges som argument i PreparedStatement metoden. Det første der sker i metoden, er at vi laver en connection til databasen. Derefter sætter vi AutoCommit til false (Linje 45), i modsætning til vores andre mappermetoder, hvor AutoCommit er sat til true (som standard). Når AutoCommit er true, bliver hver enkelt SQL-sætning eksekveret i deres egen SQL-transaktion. Dette resulterer i, at vi ikke kan lave et rollback(), hvis der skulle ske en fejl under en transaktion. For at undgå at vi kunne risikere at få indsat en ”halv” ordre, sætter vi AutoCommit til false, som gør at hver SQL-sætning bliver til én transaktion, der først bliver eksekveret, når vi kalder commit() (Linje 72). På den måde sikrer vi os, at alle customer data og ordre data bliver lagret i databasen.
Skulle der ske fejl, vil den blive fanget af vores SQLException, hvor der vil blive kaldt et rollback(), og derudover bliver det også logget i vores log fil.
Efter transaktionen er fuldført sætter vi AutoCommit til true igen. (Linje 73) Til sidst bliver der returneret et ordre id, der genereres i ordre-tabellen. (Linje 79)

Login.java - execute()

```
45          HttpSession session = request.getSession();
46          int phoneNumber;
47          int orderId;
48          Order order;
49          Customer customer;
50          BillOfMaterials bom;
51          String destination = "index";
52
53          if (session.getAttribute( s: "order") != null) {
54              session.setAttribute( s: "order",  o: null);
55              session.setAttribute( s: "employee",  o: null);
56          }
57
58          try {
59              phoneNumber = Integer.parseInt(request.getParameter( s: "phoneNumber"));
60              orderId = Integer.parseInt(request.getParameter( s: "orderId"));
61          } catch (Exception e) {
62              throw new LoginSampleException("Der gik noget galt. Sørg for, at begge felter er udfyldt og at der ikke bruges bogstaver.");
63
64          }
65
66          if (phoneNumber != 0 && orderId != 0) {
67              order = OrderFacade.getMyOrder(orderId, phoneNumber);
68              customer = CustomerMapper.getCustomer(phoneNumber);
69              bom = BomFacade.getBillOfMaterials(orderId);
70
71              if (order == null || customer == null) {
72                  throw new LoginSampleException("FEJL: Der blev ikke fundet nogen ordre med dette telefonnummer og ordrenummer i databasen. Prøv igen.");
73              } else {
74                  session.setAttribute( s: "bom", bom);
75                  session.setAttribute( s: "customer", customer);
76                  session.setAttribute( s: "order", order);
77
78                  String carportType = OrderFacade.getCarportType(order.getCarport_id());
79                  session.setAttribute( s: "carportType", carportType);
80                  session.setAttribute( s: "statusId", order.getStatus_id());
81
82                  destination = "myorder";
83              }
84          }
85
86          return destination;
```

Eksempel 2

Metoden execute() er nedarvet fra Command klassen. I dette eksempel ser vi specifikt på brugerloginet. Som det er nævnt tidligere, har vi implementeret to forskellige slags login. Det, der er særligt ved dette, er, at i stedet for den typiske brug af email og kodeord, logger man som bruger ind med ens telefonnummer og ordre ID. Et telefonnummer kan være knyttet til flere ordre ID'er og muliggør det derfor, som kunde, at placere mere end blot én ordre.
Den første if-statement håndterer et problem med, at hvis der ikke logges ud mellem visning af forskellige ordre, så nulstilles session for både ordre og medarbejder. Idéen er, at ordren vil være sat til noget, hvis lagermedarbejderen har været inde og kigge på en stykliste. Med det if-statement logger man essentielt medarbejderen ud, hvis de er logget ind når man logger ind som kunde. Der laves dernæst en try-catch på phoneNumber og orderId, som tjekker om den indtastede data er tal og ikke talord, samt at begge felter er udfyldt.

Hvis phoneNumber og orderId ikke er 0, hentes ordre- og kundeoplysninger plus stykliste fra databasen - ellers smides en LoginSampleException. Alle disse oplysninger lagres i session. Efter alt er hentet ind, returneres en destination String, der i dette tilfælde sættes til at være ”myorder”.

Uddrag af FlatOrder.java – execute()

```
85          try {
86              registeredTelephone = Integer.parseInt(request.getParameter( s: "registeredTelephone"));
87          } catch (Exception e) {
88              throw new OrderException("Telefonnummeret må kun bestå af tal.");
89          }
90
91          customer = CustomerFacade.getCustomer(registeredTelephone);
92
93          if (customer == null && telephone != 12345678) {
94              name = request.getParameter( s: "name");
95              address = request.getParameter( s: "address");
96              postalCodeCity = request.getParameter( s: "postalcode");
97              email = request.getParameter( s: "email");
98
99              if (name.length() == 0 || address.length() == 0 || postalCodeCity.length() == 0 || email.length() == 0) {
100                 throw new OrderException("Du mangler at udfylde et felt.");
101             }
102
103             customer = new Customer(telephone, name, address, email, postalCodeCity);
104             orderId = OrderFacade.insertOrder(customer, order);
105
106         } else if (customer != null) {
107             order.setPhone(customer.getPhone());
108             orderId = OrderFacade.insertOrderForExistingCustomer(order);
109         } else {
110             throw new OrderException("Du er ikke registreret, udfyld venligst alle felter for at bestille");
111         }
112
113
114         bom.setOrderId(orderId);
115         BomFacade.insertBillOfMaterials(bom);
116
117         request.setAttribute( s: "orderId", orderId);
118
119         return "receipt";
120     }
121 }
```

Eksempel 3

Her ses en del af execute() metodens indhold i FlatOrder klassen, som arver metoden fra Command klassen. I FlatOrder tager metoden imod diverse variable i forbindelse med bestilling af carporte, såsom carport mål eller kundeinformation, og de bruges til at indsætte en ordre og evt. kunde i databasen og til sidst styklisten for den enkelte ordre.

Før man når til starten af udklippet i FlatOrder's execute() metode, tages der imod forskellige variable til selve carporten, som parses (dvs. konverteres til at passe med den datatype man har angivet, at variablen skal have når den modtages – bruges mest til tal) hvis altså de kan – ellers kastes en OrderException, en af vores egne fejl, som stopper bestillingsprocessen og giver kunden besked om, at de har gjort noget forkert.

Der køres også et par if-statements, der sørger for, at kundens input ikke bryder nogen af de etablerede regler – fx må skuret ikke være større end carporten (se linje 62 i FlatOrder), og det håndteres så af et if-statement, hvis det er tilfældet. Der bruges også et if-statement til at afgøre, hvilken type carport der skal laves stykliste til, og hvilken beregnermetode der skal bruges ved at se på, om skurets dimensioner er sat til noget eller ej (se linje 77).

Hvis alt er gået godt indtil da, når man til starten af udklippet på linje 85. Her forsøges det at parse variablen "registeredTelephone", som er et felt på flatorder.jsp der tastes i, hvis en kunde trykker på "Jeg er allerede registreret" knappen. Herefter søges der på linje 91 med getCustomer() metoden fra CustomerFacade i databasen efter en kunde med et telefonnummer, der er identisk med det indtastede.

En vigtig detalje at bemærke er, at både "registeredTelephone" og "telephone" er sat til "12345678" som standard værdi. Det er for at undgå, at der opstår parsing problemer når der laves en bestilling – hvis de ikke er sat til noget som udgangspunkt, vil mindst én af felterne ikke kunne parses, da en af dem altid vil være tomme; afhængigt af om man er en allerede registreret kunde eller en ny.

Fra linje 93 og ned foretages der så en af to operationer; hvis getCustomer() metoden returnerede null, dvs. ikke fandt en kunde med "registeredTelephone" variablen, og "telephone" variablen ikke er sat til sin standard værdi, dvs. der er blevet tastet et telefonnummer ind i registreringsformen (som man ville gøre ved førstegangsbestilling), så hentes værdierne fra alle registreringsparametre ned som variable af typen String, som ikke behøver at blive parset.

Hvis nogen af disse variable er tomme, eller med andre ord har en længde på 0, kastes der igen en OrderException, fordi det betyder, at en af felterne ikke er blevet udfyldt. Ellers laves der et Customer objekt med variablene, og en ordre indsættes sammen med kunden i insertOrder() metoden fra OrderFacade i linje 104.

Den anden operation foretages i linje 106, hvis getCustomer() metoden fandt en kunde med ”registeredTelephone” variablen. Hvis en kunde blev fundet, bruges insertOrderForExistingCustomer() metoden i stedet for insertOrder(), hvis eneste forskel er, at der bare indsættes en ordre i databasen med det angivne telefonnummer og ikke både ordre og kunde som i insertOrder().

Hvis man forsøger at bestille gennem ”Jeg er allerede registreret” knappen uden at være registreret, kastes der en OrderException som fortæller kunden, at de må udfylde alle felter, før de kan bestille. I dette tilfælde ville ”customer” variablen være null, da der ikke kunne findes en kunde med det indtastede telefonnummer, og ”telephone” variablen ville stadig være på sin standard værdi, ”12345678”, så derfor ville fejlen fanges i det sidste else-statement i linje 109.

Endelig sættes ordre ID’et på det tidligere skabte Bill Of Materials objekt (laves i linje 77), som blev genereret og returneret af enten insertOrder() eller insertOrderForExistingCustomer() metoderne, og så indsættes styklisten i databasen. Så sættes ID’et på requestScopet, og der returneres tekststrengen ”receipt”, som tager kunden videre til en kvitteringsside gennem FrontController og Command klasserne, hvor ordre ID’et vises sammen med en lille kvitteringsbesked.

Uddrag fra DrawingSide.java

```
76          //skur
77          if (shedWidth != 0) {
78
79              int numberOfCladdingPlanks = calcNumberOfCladdingPlanks(shedLength);
80              int underPlank = (int) Math.ceil( numberOfCladdingPlanks/2);
81              int overPlank = (int) Math.floor(numberOfCladdingPlanks/2);
82              double firstUnderPlankX = finalPostsSpace+offset;
83              double firstOverPlankX = finalPostsSpace- 2.5;
84              double firstPlankY = underFasciaHeight;
85              double plankWidth = 10;
86              double plankHeight = carportHeight-underFasciaHeight;
87
88               for (int i = 0; i < underPlank ; i++) {
89
90              svg.addRect(firstUnderPlankX,firstPlankY,plankHeight,plankWidth );
91              firstUnderPlankX -=15;
92               }
93
94               for (int i = 0; i < overPlank ; i++) {
95
96                  svg.addRect(firstOverPlankX,firstPlankY,plankHeight,plankWidth );
97                  firstOverPlankX -=15;
98              }
```

Eksempel 4

Denne kodestump hører til i DrawingSide klassen i vores projekt, linje 77 til 98. Denne klasse indeholder metoden execute(), som den arver fra Command klassen. Når execute() metoden kaldes fra DrawingSide klassen, bliver der lavet en SVG-tegning af carporten set fra siden. Denne tegning er baseret på de mål, kunden har tastet ind under sin bestilling.

Før vi når til dette punkt i koden, bliver der kaldt en del andre metoder; forskellige stringbuildere, som lægger elementer til SVG-tegningen, og elementerne danner den bærende konstruktion af carporten. Hvis kunden har valgt skurmål, vil koden i eksemplet blive løbet igennem. Vi har lavet fejlhåndtering i vores kode, der sikrer, at en kunde, i bestillingsformularen, enten vælger mål på *både* skurbredde og skurlængde eller *ingen* af dem. Derfor er det nok kun at sætte shedWidth til forskellig fra null, for at den skal starte med at tegne skuret.

Det første koden gør, er at hente antallet af beklædningsbrædder man behøver til væggen af skuret. Dette sker ved hjælp af metoden CalcNumberOfCladdingPlanks(). Metoden tager en parameter, som svarer til det område, der skal beklædes. I SVG-tegningen vil den eneste del, der faktisk kan ses, være shedLength. Beklædningen bliver lagt i to lag. Derfor deler vi antallet af beklædningsbrædder med to. Skulle udregningen komme frem til et ulige antal brædder, har vi valgt at runde udregningen for beklædningsbrædder, der ligger underst (underplanker), *op* til nærmeste heltal. Dette sker ved hjælp af Math.ceil() – en metode fra Math klassen, som er en klasse fra Javas eget bibliotek. Omvendt har vi valgt at runde udregningen af beklædningsbrædderne, der skal ligge øverst (overplanker), *ned* til nærmeste heltal ved hjælp af Math.floor().

Efterfølgende skal vi finde ud af, hvor tegningen af selve beklædningen starter. Vi har valgt at starte i bagenden, ved den bagerste stolpe på carporten, hvor underplankerne sættes på som det første. For at få disse planker på, løber vi gennem underplankerne og for hver gang, vi støder på en, tilføjes en rektangel i SVG-tegning. Dette gør vi ved hjælp af metoden addRect(). Denne metode ligger i Svg.java. Metoden tager fire parametre: Ét parameter for hvor på x-aksen rektanglet skal starte, *firstUnderPlankX*, ét parameter for hvor på y-aksen det skal starte, *firstPlankY*, ét for højde, *plankHeight*, og ét for bredde, *plankWidth*. For hver gang man har tegnet en planke, trækker man 15 fra *firstUnderPlankX.* 15 svarer til en plankebredde på 10 cm plus 5 cm mellemrum.

Når alle underplankerne er blevet sat på, tilføjes overplankerne. Processen er næsten identisk med undtagelse af, hvor på tegningens x-akse den første overplanke starter – her bruger vi *firstOverPlankX.* Forskellen mellem *firstUnderPlankX* og *firstOverPlankX* er en lille forskydning, som gør at overplankerne overlapper de 5 cm mellemrum.

Status på implementering

I forhold til status på implementeringen vil den afhænge af, hvilket udgangspunkt man tager. Hvis vi går ud fra den originale opgavetekst, mangler vi, at kunden skal kunne bestille og se en tegning af en carport med rejsning.

Tager man udgangspunkt i vores egne user stories, har der været nogle idéer til funktioner, der kun er blevet delvist implementeret. For eksempel kan Martin opdatere materialerne i databasen og indsætte nye materialer, men skal de nye materialer bruges, må beregneren også opdateres. Derfor ville man blive nødt til at få lavet nye beregnermetoder hvis nye materialer skulle inkluderes i beregneren. Dette var ikke en del af opgaveteksten, men noget Martin selv nævnte i videoen som introducerede opgaven.

Undervejs har både vi og vores undervisere (Nikolaj og Jon) løbende vurderet, hvad der i realiteten ville kunne færdiggøres. Vi valgte at nedprioritere carport med rejsning for i stedet at kunne fokusere på optimering af bestilling af carport med fladt tag samt kvalitetssikring i form af blandt andet fejlhåndtering.

Ud fra de nye vurderinger kan vi sige, at helt generelt er den overordnede status på implementeringen, at vi er nået i mål med det, vi havde sat os i hovedet.
Det skal dog nævnes, at der er visse mangler rent designmæssigt om end ganske få. Layout er ikke helt så responsivt som ønsket, når man nedskalerer det. Det skyldes, at forhandlerloginet er fikseret i øverste højre hjørne, hvilket gjorde det nær umuligt at lægge de overskydende elementer i en burgermenu.

Test

Vi har lavet en del tests på forskellige metoder fra forskellige klasser i DBAccess og FunctionLayer mapperne, men tests har ikke været fundamentet for selve kodningsprocessen, som de ville være i TDD (Test-driven development).

Derfor synes vi stadig, at de er gode at have for at bekræfte, at visse metoder fungerer som forventet – dog løb vi ind i nogle problemer med enkelte mappermetoder i visse klasser fra DBAccess, specifikt i BomMapperTest, hvor getBillOfMaterials() metoden bruger en lang SQL-sætning med mange joins, der afhænger af, at visse værdier er blevet sat, fx carport_part_id. Derfor måtte vi lave en forenklet metode, der henter stykliste elementerne ned kun med ordre ID'et og intet andet.

Vi har lavet en tabel[7], der viser hvilke tests vi har lavet, til hvilke klasser og metoder de er lavet, og hvilken dækningsgrad der er på dem.

NB: 7 ud af 8 klasser og 28 ud af 36 metoder anses som dækket i DBAccess, fordi Connector klassen og dens metode connection() bruges i alle mappertests til at etablere forbindelse til databasen. Den sidste, der mangler, er InitializeMapper, som kun bruges til visning af mulige længder og bredder til carporte og skur på flatorder.jsp. Dækningen er fundet gennem IntelliJ's "run with coverage" funktion.

7 ud af 21 klasser og 64 ud af 203 metoder anses som dækket i FunctionLayer, fordi visse klasser er blevet instantieret og deres metoder brugt i de testede klassers metoder, eller i selve testmetoderne (vi bruger et Bill Of Materials objekt i testType1Calc() metoden for eksempel).

Process

Arbejdsprocessen faktuelt

Som nævnt tidligere har vi brugt Taiga.io til at styre arbejdsprocessen. Vi havde i alt fire sprint, som blev fordelt på ca. fire uger. Det løb fra onsdag til onsdag, hvor vi både holdt PO-møder og fik forberedt sprint til den følgende uge.

[7] Se bilag 3

Vores daglige standup møder var kl. 9; dog blev de holdt online via Zoom, eftersom COVID-19 satte en stopper for at mødes ansigt til ansigt.

Da vi ingen tidligere erfaring havde med SCRUM, fik vi aldrig udpeget en decideret SCRUM master. Vi fik organiseret de forskellige user stories og delt dem op i tasks, så de nemt kunne fordeles til gruppens medlemmer. Da hele vores arbejdsproces, med undtagelse af PO-møder, var selvstyret, var SCRUM master rollen måske endda også overflødig.

Vi føler ikke, at vi med sikkerhed kan sige, om det ville have været bedre at have en SCRUM master eller ej, da vi ikke har meget erfaring med SCRUM i forvejen. Man kan sige, at det at have en person, der styrer eller dirigerer projektet eksternt kunne hjælpe med strukturering, da vi selv måske kunne få tunnelsyn og vælge at arbejde på ting, der ikke var strengt nødvendige i et givent sprint.

Arbejdsprocessen reflekteret

Alt i alt har vi arbejdet rimelig effektivt under det meste af projektforløbet, men strukturen på arbejdet har ikke været lige stærk hele tiden. Eksempelvis har vi haft visse user stories, som ikke var en del af vores sprints, der blev lavet i forbindelse med andre user stories, der lå nær dem i forhold til funktionalitet eller område.

Derfor har vi ikke sat dem ind som user stories i nogen sprints, og derfor heller ikke lavet tasks til dem. Visse ting er også blevet lavet, som slet ikke var en del af nogen af vores user stories, og mere var fordi vi selv ville have dem på. I den forstand har arbejdet med de user stories ikke været så struktureret, som det kunne have været.

Vi har kun haft et retrospektiv møde, hvor vi kiggede en ekstra gang på vores SWOT- og interessentanalyse omkring 2 uger efter projektstart, og vi havde ikke nogen væsentlige ændringer at lave der. Ligeledes har vi ikke lavet store ændringer på dem siden, da vi ikke mente, at vi havde opdaget nye eller anderledes forhold mellem interessenter eller nye ting om os selv ift. SWOT-analysen.

Vedrørende vores ugentlige PO-møder, og de sprints vi fik lavet i forbindelse med dem, var der ingen større problemer med at bryde de valgte user stories til hvert sprint ned i enkelte tasks. Dog har vi enkelte gange løbet ind i, at vi lavede flere tasks end nødvendigt til visse user stories, men vi har ikke som sådan haft problemer med at bryde dem ned i tasks generelt.

Vores estimeringer var ikke altid helt præcise, da vores primære formål bare var at få fyldt omkring 37 point på et sprint (svarende til 37 timer på en uge). For eksempel havde vi afsat 8 points til US-2 i første sprint, som bare krævede, at vi skulle kunne vise en simpel bestillingsproces for en carport med fladt tag (uden knytning til databasen), og den tog ikke 8 timer at få lavet.

Det skyldes nok også, at vi på tidspunktet lige var startet på projektet og ikke havde en god fornemmelse for, hvor længe vi ville bruge på de forskellige dele, så vi afsatte mere tid end nødvendigt for at være på den sikre side.

Vi havde ingen problemer med hverken vejledning eller PO-møder, men til gengæld var vores egen arbejdsgang ikke særlig effektiv i starten af projektforløbet. Vi sad mest sammen som hele gruppen og arbejdede på enkelte dele af projektet ad gangen, hvilket betød at hele processen i perioden var ret langsom og fuld af en masse snak og diskussion, der ofte ikke ledte til noget produktivt.

Et par uger inde splittede vi os så op og arbejdede adskilt på hver vores del. Nogle arbejdede solo, andre i mindre grupper på 2, og det ledte til større produktivitet. Dog havde denne arbejdsform også sine problemer, da nogle af os af og til havde svært ved at løse et problem og ikke altid havde mulighed for at snakke med de andre om det direkte.

Derfor kan man sige, at hver arbejdsform havde sine fordele og ulemper, men umiddelbart var rytmen og produktiviteten ved sit højeste efter vi begyndte at splitte os.

Konklusion

Projektet har vært udfordrende, men også vældigt lærerigt. Vi har jo den sidste tid levet under forhold, som ikke står til en normal hverdag. Det har ellers gået fint, da vi i forvejen har brugt bl.a. Zoom en hel del i vores undervisning. Udover nye elementer såsom SVG og lidt ekstraordinære elementer som Calculator, er koden bygget op af elementer, vi kender fra tidligere. Det vigtigste i læringsprocessen har været at arbejde sammen i SCRUM formatet, hvor man uddelegerer arbejde og faktisk må give slip på dele af processen. Alt i alt er vi fornøjede med slutresultatet på produktet, og vi mener, at det lever op til langt størstedelen af de krav, der er blevet stillet.

Yderligere links

Taiga.io:
https://tree.taiga.io/project/pelle-pr-bornit/backlog

User Stories spreadsheet:
https://docs.google.com/spreadsheets/d/16qe5ipDncZXU_DJwf_BXVvBgG2ON6v12JWci09pdN4s/edit

Interessantanalyse:
https://docs.google.com/document/d/1b2MUS-S3W2-h8prRetp-GdGypmjhLpEIWsTFgHpoHZE/edit

SWOT-analyse:
https://docs.google.com/document/d/1f7sRxWUq1ZOzSEomoKVrwxB6x4IGbUwGXzsicoUFsB0/edit

Uddrag af bog om databaser, kapitel om normalisering – specifikt side 24:
https://datsoftlyngby.github.io/dat2sem2020SpringBornholm/Modul2/Week2-Database/resources/DB_normalisering.pdf

Kilde for afsnit om arkitektur:
https://informatik.systime.dk/?id=1124

Ordliste:
https://docs.google.com/spreadsheets/d/1XDiDSWqBOJnpvD2w7KU3g01JIyRl8kQrwNAiI4nqJFY/edit#gid=0

Bilag

Bilag 1: Tabel over interessenter

	Stor indflydelse	**Lille indflydelse**
Påvirket af projektet	***"Ressourceperson"*** Fog (firma) Martin Bornit	***"Gidsler"*** Kunder Konkurrenter Medarbejdere
Ikke påvirket af projektet	***"Grå eminence"*** Jon og Nikolaj (undervisere)	***"Eksterne interessenter"*** Lagermedarbejder Eksterne tømrer

Bilag 2: Tabel over user stories

ID	**Taiga.io US #**	**Sprint #**	**Beskrivelse (forenklet)**	**How-to-demo**	**Estimat i timer**	**Status**
US-1	1	1	Kunde ønsker at se hvilke typer carporte der tilbydes.	Vis forside (index.jsp).	6	Færdig
US-2	2	1	Kunde ønsker at kunne bestille en carport m. fladt tag.	Angiv højde, bredde, tryk bestil og få kvittering.	8	Færdig
US-3	Ubestemt	Ubestemt	Kunde ønsker at kunne bestille carport m. tag med rejsning.	Angiv højde, bredde, vælg tagtype, tryk bestil og få kvittering.	Ubestemt	Ikke påbegyndt
US-4	1	1	Kunde ønsker at kunne bestille en carport m. fladt tag og skur.	Angiv højde, bredde, skurhøjde, skurbredde, tryk bestil og få kvittering.	8	Færdig NB: lavet i forbindelse med US-2.
US-5	35	3	Kunde ønsker at kunne se en simpel (og fuld) tegning af carport m. fladt tag.	Log ind som kunde, gå til tegningside og se tegning.	8	Færdig

US-6	Ubestemt	Ubestemt	Kunde ønsker at kunne se simpel (og fuld) tegning af carport m. tag med rejsning.	Log ind som kunde, gå til tegningside og se tegning.	Ubestemt	Ikke påbegyndt
US-7	Ubestemt	Ubestemt	Kunde ønsker at kunne bestille carport m. tag med rejsning og skur.	Angiv højde, bredde, skurhøjde, skurbredde, vælg tagtype, tryk bestil og få kvittering.	Ubestemt	Ikke påbegyndt
US-8	14	2	Kunde ønsker at kunne se samlet pris på carport.	Log ind som kunde, se ordre inkl. pris.	13	Færdig NB: Lavet i forbindelse med US-17.
US-9	36	3	Kunde ønsker at kunne acceptere eller annullere sin ordre.	Log ind som kunde, se ordre og tryk ”bestil” eller ”afslå”.	3	Færdig NB: Lavet i forbindelse med US-11
US-10	3	1	Kunde ønsker at kunne modtage simpel (og fuld) stykliste for carport m. fladt tag.	Gå ind på kundeside, gå til styklisteside og se hardkodet stykliste.	25	Færdig
US-11	36	3	Kunde ønsker at følge bestillingsstatus.	Log ind, se ordre, check status.	3	Færdig
US-12	40	3	Lager ønsker at kunne se accepteret tilbud.	Log ind via forhandlerlogin og se betalte ordrer.	4	Færdig
US-13	40	3	Lager ønsker at kunne se hver ordres stykliste.	Log ind via fohandlerlogin og se styklister på ordrer.	4	Færdig NB: Lavet i forbindelse med US-12
US-14	40	3	Lager ønsker at kunne ændre en ordres status til ”afsendt”.	Log ind via forhandlerlogin og tryk ”afsend” på ordre.	4	Færdig NB: Lavet i forbindelse med US-12

US-15	Ubestemt	Ubestemt	Martin ønsker at kunne opdatere materialer og priser.	Log ind via forhandlerlogin, gå ind på side hvor man kan ændre priser på og indsætte materialer osv.	Ubestemt	Færdig NB: Lavet efter sidste sprint sideløbende med rapporten.
US-16	37	3	Martin ønsker at kunne se og godkende ordrer.	Log ind via forhandlerlogin, tryk ”godkend” på ordre.	3	Færdig
US-17	14	2	Kunde ønsker at kunne logge ind og se sin bestilling.	Log ind som kunde, se ordre.	13	Færdig
US-18	38	3	Kunde ønsker at modtage fuld stykliste for carport m. fladt tag og skur	Log ind som kunde, gå til styklisteside, se fuld stykliste.	9	Færdig

Bilag 3: Tabel over fuldførte tests

Mappe	Klasse	Testede metoder	Dækningsgrad
DBAccess	Connector	Connection()	Metoder: 66% (2/3) Linjer: 25% (4/16)
DBAccess	BomMapper	insertBillOfMaterials() deleteBom() getBillOfMaterialsForTest() getNumberOfMaterials()	Metoder: 80% (4/5) Linjer: 59% (44/74)
DBAccess	CarportPartsMapper	getCarportParts() getCarportPartIds() getCarportPartDescriptions()	Metoder: 100% (3/3) Linjer: 77% (42/54)
DBAccess	CustomerMapper	getAllCustomers() getCustomer() deleteCustomer()	Metoder: 100% (3/3) Linjer: 75% (34/45)
DBAccess	EmployeeMapper	login() setEmployeeRole()	Metoder: 100% (2/2) Linjer: 80% (24/30)

DBAccess	MaterialMapper	getMaterialLengths() setMaterialValues() setMaterialSizeIds() setLinkMaterialSizeIds() setUnitTypes() getAllMaterials()	Metoder: 100% (6/6) Linjer: 80% (87/108)
DBAccess	OrderMapper	getAllOrdersByStatusId() getCarportType() deleteOrder() getMyOrder() getOrderStatus() updateStatus() updateTotalPrice() insertOrder()	Metoder 80% (8/10) Linjer: 59% (116/194)
FunctionLayer	Calculator	type1Calc() type2Calc() calcOptimalLengthOfMaterial() calcNumberOfCladdingPlanks() calcLengthOfTrapezPlates() calcSpaceBetweenRafters() calcNumberOfPosts()	Metoder: 100% (23/23) Linjer: 87% (320/366)
FunctionLayer	Svg	addRect() addDefs() addPerforatedBand() addTextRotated() addText() addRectDecline() addInnerDrawing()	Metoder: 40% (9/22) Linjer: 61% (34/55)

Mappe	Samlet dækning
DBAccess	Klasser: 87% (7/8) Metoder: 77% (28/36) Linjer: 61% (356/577)
FunctionLayer	Klasser: 33% (7/21) Metoder: 31% (64/203) Linjer: 55% (426/764)