Fog Bestillingsservice - Rapport

af

Navn	Mail	Github Username	Klasse
Emil Jógvan Bruun	Cph-eb122	Svensmark	A
Henning Wiberg	Cph-hw98	Mutestock	A
Lukas Bjørnvad	Cph-lb269	Lukas-bjornvad	A
Simon Asholt Norup	Cph-sn233	wormple12	A

29-05/2019

https://github.com/Mutestock/Fog

http://165.227.148.141:8080/FogProject/

Indholdsfortegnelse

Indledning

Dette projekt omhandler udviklingen af en webapplikation, der fungerer som en bestillingsservice af en carport fra Johannes Fog. Gennem denne applikation kan man online se, hvad en carport med forskellige mål og specifikationer kommer til at koste, og hvordan den vil se ud (en genereret tegning baseret på disse mål og specifikationer). Man har også muligheden for som kunde at sende en forespørgsel til Fog om at få tilbud på den givne carport.

Gennem webapplikationen kan man som administrator gøre brug af et loginsystem, hvor man får adgang til generelle administrative funktionaliteter. Admins kan se en liste over forespørgsler fra kunder, informationer om specifikke bestillinger, bl.a. dato på bestilling og alle specifikationer angående den bestilte carport. Derudover kan man som admin også svare på en forespørgsel med et tilbud, hvor svaret vil blive sendt direkte til kundens angivne e-mailadresse. Som admin kan også man se en stykliste af delene der skal til for at lave den angivne carport, delt op i hhv. trægruppen, taggruppen, og beslag og skruer.

Der redegøres i denne rapport for, hvordan en løsning til problemstillingen og udviklingsprocessen til en løsning kunne se ud. Der vil blive diskuteret de udfordringer, der blev stødt på under konstruktionen, samt hvordan de blev løst og, hvis det ikke kunne lade sig gøre, hvad man potentielt kunne have gjort for at komme videre.

Baggrund

Virksomheden der ville have dette program udviklet, er Johannes Fog Trælast og Byggecenter. De ønskede en online service, der skulle kunne forenkle bestillingen af en carport og gøre det lettere for både kunde og virksomhed at tilfredsstille begge parter.

Virksomhedens krav til programmet var, at man som kunde skulle kunne indsætte mål til en carport og derefter kunne se en skitsetegning baseret på disse mål, samt have muligheden for at sende en forespørgsel til Fog om denne carport. Der skulle også være en mulighed for ansatte, så de kunne se bestillinger og håndtere dem på sådan en måde, at de kunne besvares tilbage til kunden. Endnu et af opgavens krav, var at programmet, ud fra kundens specificeret mål på en carport, kunne generere en skitsetegning af carporten fra både ovenfra og fra siden af.

Teknologivalg

- Netbeans 8.2 // m. Java 1.8.0 (inkl. SE Runtime Environment)
- MySQL 8.0 (inkl. Workbench)
- Ubuntu 18.10 // m. OpenJDK 11.0.1 - på DigitalOcean droplet
- JDBC driver (mysql connector java 5.1.47)

Krav

Firmaets vision

Fog som virksomhed har højt fokus på kvalitet og omdømmet heraf. Programmets funktion er, at man som bruger kan benytte sig af firmaets hjemmeside til at kunne komme i kontakt med firmaet. Det anses som en selvfølge at have en webshop til større firmaer inden for denne branche. Dette betyder, at firmaet for at kunne følge med skal have en selv. Hjemmesiden skal være simpel og nemt tilgængelig.

Givet, at hjemmesiden kan generere en skitse af en carport via selvvalgte specifikationer, kan firmaet danne udstråling af professionalisme og fleksibilitet, hvilket gør, at kunden kan føle sig tryk og mere sikker på, at Fog kan præstere et godt stykke arbejde.

Overordnede krav

Programmet skulle kunne udregne mængden og længden på de brædder og tagplader, såvel som antal af beslag og skruer, som skulle bruges til at bygge en given carport. Dette var angivet som den vigtigste funktion, der skulle opnås. Kunden gav adgang til 2 eksempler på, hvordan en stykliste kunne se ud; én for en carport med fladt tag samt én for en carport, hvor taget havde en rejsning på over 15 grader. Udover styklisterne var der også en forklaring, på hvordan man samlede carporten, samt en skitsetegning af den givne carport. I disse eksempler blev der ikke opgivet mere information, angående hvordan udregningerne på mængderne blev udført. Dette betyder, at der skulle antages en masse relationer mellem carportens mål og materialer baseret på skitserne. Baseret på mængder, længder og delpriser på alle disse dele til carporten, så skulle der udregnes en estimeret total pris for alle dele.

Programmet skulle på baggrund af en given carport også kunne lave to skitsetegninger, der kunne repræsentere, hvordan carporten skulle se ud. Størrelsesforholdene og afstande mellem de interne dele skulle være de samme, så det kan aflæses som en skaleret og præcis skitsetegning. Dette skulle både gøres fra oven og fra siden af.

Product-owner ville gerne have et log-in system for ansatte, hvilket giver mulighed for admin-rettigheder til relevante funktioner. Som administrator ville PO gerne kunne se en oversigt over alle forespørgsler og sende tilbud til kunderne. Det skulle være muligt at se alle detaljerne på individuelle forespørgsler, inkl. specifikationer, kundeinformationer, stykliste, prisestimat og skitsetegninger for carporten. Der var også en forventning, til at den ansatte ville have mulighed for at se den estimerede pris baseret på materiale omkostninger og derefter selv svare med en given pris.

User Stories

Ifm. fremgangsmetode og workflow har vi benyttet Scrum med de følgende sprints og user stories:

- **Sprint 1**
  - Customer – Carport specifications
    - Man skal kunne vælge carportens specifikationer
  - Customer – Personal Info
    - Kunden skal kunne indtaste basisinformation f.eks. navn, telefon, etc.
  - Everyone - Navigation Bar
    - Måde for en hvilken som helst bruger at kunne navigere rundt på vigtige sider fra et hvilket som helst udgangspunkt.
- **Sprint 2**
  - Employee – Receive Order
    - Mulighed, så admin kan se kundeoplysninger og detaljer om forespørgsel, efter en carport er blevet bestilt. Skal også kunne se genereret prisestimat.
  - Customer – Submit Request
    - Kunder skal kunne afsende en forespørgsel med kundeinformation og carportspecifikationer til Fog.
  - Employee – Send Offer
    - Mulighed for admins for at kunne afsende det endelige tilbud til kunden via email. Prisen for carport og fragtomkostninger skal kunne omdefineres af admin. Tilbuddet skal kunne afsendes simpelt og nemt.
- **Sprint 3**
  - Employee – Wood Parts
    - Algoritme til udregning af stykliste for trægruppen ift. carportens dimensioner.
  - Employee – Fittings and Screws Parts
    - Algoritme til udregning af stykliste for beslag og skruer ift. carportens dimensioner.
  - Customer – Sketch Drawing (Above)
    - Første del af muligheden for, at kunden kan se en visuel repræsentation af den genererede carports dimensioner fra toppen.
  - Customer – Sketch Drawing (Side)
    - Anden del af muligheden for, at kunden kan se en visuel repræsentation af den genererede carports dimensioner fra siden.
  - Employee – Roof Parts
    - Algoritme til udregning af stykliste for taggruppen ift. carportens dimensioner, inkl. dele fladt tag og tag med rejsning.
  - Employee – Admin Login
    - Mulighed for at admin kan logge ind, og dermed se informationer og bruge funktioner, som eksklusivt kan behandles af admins.
  - Split Carport Details
    - Anmodet ændring af kunden. Splittelse af to sider, så kunden kan se informationerne omkring carporten, før at det er nødvendigt at indtaste personlige informationer.

- **Sprint 4**
  - Fremtidssikring – Dokumentation
    - Dokumentation af programmeringsdelen af projektet. Gør det lettere tilgængeligt af andre programmører, som senere skal håndtere projektet og mere forståeligt at ændre på.
  - Fremtidssikring – Stabilitet
    - Yderligere og mere detaljerede test af programmet. Produktets fremtidige programmører skal kunne få et funktionelt program at arbejde videre på.
  - Customer – Transaktions-bekræftelses side
    - Kunden skal kunne få en klar besked fra hjemmesiden, om at forsøget på at sende en forespørgsel er lykkedes.
  - Fejlhåndtering
    - Yderligere fejlhåndtering. Sikrer bl.a., at brugere ikke forsøger at gå til sider hvor det ikke giver mening at være. F.eks. hvis kunden prøver at gå direkte til siden om carportens dimensioner og tegning, uden at have gået igennem den forrige side, om udfyldelse af carportens detaljer.
  - Employee – Control of Carport Possibilities
    - Mulighed for, at admin kan ændre på Carportens muligheder direkte fra siden, f.eks. hvis skurets sider skal kunne være konstrueret af andre materialer, end hvad der ellers var tilgængeligt.

Yderligere information om udøvelsen af Scrum i opgaven kan findes i Proces-afsnittet af rapporten.

Proces

Oprettelsen af et effektivt workflow er en afgørende faktor for produktionen af et kvalitetsprodukt. Siden de større projekter bliver lavet i teams af varierende størrelse, er det vigtigt, at der er kommunikation mellem alle medlemmer, så der bliver oprettet gensidig forståelse. Dette er for at undgå, at de samme opgaver bliver lavet flere gange på forskellige måder eller med forskellige programmer, frameworks, konfigurationer, osv. Som en forberedelse på de fremgangsmetoder, som mange arbejdspladser udnytter, var det en del af scenariets opgaver, at de skulle benyttes. Dette afsnit handler om gruppens workflow opsætning ifm. med dette.

Scrum

Kort om scrum

Scrum ligger under kategorien "agile metoder", dvs. at der løbende i forløbet dannes gensidig forståelse med kunden. Ved brug af denne metode opstår der et dynamisk workflow, som er en fordel, hvis kunden beslutter sig for at tage projektet i en anden retning, eller blot ændrer mening angående småting. Scrum er opdelt i forskellige roller, alt efter hvad der er nødvendigt inden for projektet. Hovedsageligt er der en scrum master som styrer scrum-møderne, et team som er opdelt i roller alt efter projektets opsætning og emne, samt en product owner, som repræsenterer kunden. Sammen med product owner, bliver der defineret user stories; overordnede emner/sektioner, som er nødvendige for et succesfuldt resultat, og teamet definerer derefter tasks, som er under-opgaver for hver user story, der per definition skal kunne klares på én dag. User stories og tasks bliver opdelt efter relevans og tidsforbrug.

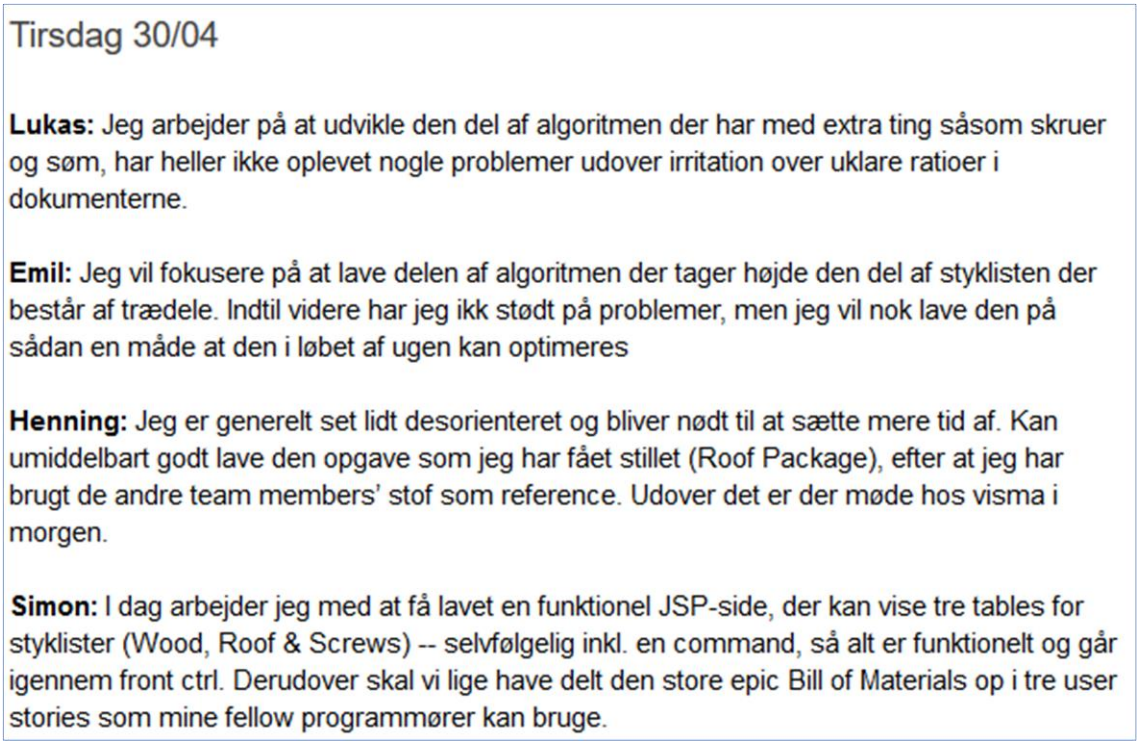
Det skal forstås, at mange arbejdspladser følger Scrum pragmatisk i den forstand, at der adopteres aspekter alt efter behov. Dette projekt udnytter de fundamentale fremgangsmetoder, det grundlæggende som nævnt tidligere.

Daily scrum

Daily scrum møder er korte opsummeringer, som typisk bliver diskuteret over 5-10 minutter. De handler hovedsageligt om status på de nuværende opgaver hos hver af gruppens medlemmer, problemer, der er opstået, og lignende.

I dette scenarie var det et krav, at alle scrum-møder skal opsummeres skriftligt. Dette var hovedsageligt for at kunne demonstrere relevansen af møderne. Derfor blev det et krav, at der blev sat 15 minutter af i stedet for 5-10 minutter.

Skriftlig notering har i dette tilfælde givet overblik over de nuværende og daværende problemer blandt alle teamets medlemmer. Dette har været brugbart for at kunne fortsætte med de næste trin i arbejdsprocessen. Det er en anden del af agile metoder, at teamet kan omfordele dets ressourcer, hovedsageligt i mandetimer, og at kunne optimere den brugte tid via råd, hvis der opstår komplikationer.


Tirsdag 30/04

Lukas: Jeg arbejder på at udvikle den del af algoritmen der har med extra ting såsom skruer og søm, har heller ikke oplevet nogle problemer udover irritation over uklare ratioer i dokumenterne.

Emil: Jeg vil fokusere på at lave delen af algoritmen der tager højde den del af styklisten der består af trædele. Indtil videre har jeg ikk stødt på problemer, men jeg vil nok lave den på sådan en måde at den i løbet af ugen kan optimeres

Henning: Jeg er generelt set lidt desorienteret og bliver nødt til at sætte mere tid af. Kan umiddelbart godt lave den opgave som jeg har fået stillet (Roof Package), efter at jeg har brugt de andre team members' stof som reference. Udover det er der møde hos visma i morgen.

Simon: I dag arbejder jeg med at få lavet en funktionel JSP-side, der kan vise tre tables for styklister (Wood, Roof & Screws) -- selvfølgelig inkl. en command, så alt er funktionelt og går igennem front ctrl. Derudover skal vi lige have delt den store epic Bill of Materials op i tre user stories som mine fellow programmører kan bruge.

Figur 1: Eksempel på daily-scrum noter. Omhandler den enkelte persons status, herunder nuværende situation, dagens opgaver og spørgsmål/hjælp.

Sprint Planning

Dette omhandler kontakt med product owner. En af hovedpointerne med Scrum er den dynamiske kontakt og gensidige forståelse. Formålet er ofte at kunne få indsigt i den nuværende situation; hvor langt er projektet ift. de originale mål; er der noget, der skal ændres; ekstra/skærpelse af tasks og user stories. Product owner har som regel ikke meget indblik i den tekniske del af produktet, og der bliver ofte fokuseret på demonstrationer af frontend-delen.

I dette scenarie var der sprint planning i midten af hvert sprint, som i dette tilfælde er defineret som én gang om ugen, hvilket er typisk i virksomheder.

Brugbarheden af sprint planning-møderne var essentielt for projektets gennemførsel, hvilket er ideelt, siden kunden bør kunne lave en grov klargørelse, af hvordan det endelige projekt skal fungere; teknisk såvel som visuelt.

Sprint Retrospective

Det retrospektive mødes funktion er at tydeliggøre projektets status over et længere spektrum af tid. Der er specielt fokus, på hvad der kan gøres bedre i det næste sprint og resten af projektet baseret på de tidligere sprint. I stedet for at der blev diskuteret i starten af dagen som i daily scrum, bliver retrospektet afholdt som den sidste aktivitet i sprintet. Der blev som regel også sat mere tid af til det. Dets funktion blev aldrig gennemgået i de næste sprints, men dets overordnede budskaber blev adopteret. Dvs. at der f.eks. blev highlightet, hvor nødvendigt det var, at vi på individuel basis lagde mere tid i brug af scrum-boardet, som i dette scenarie blev lavet på https://taiga.io.

Vi kan evaluere, at dette aspekt af scrum-udøvelsen der gav mindst værdi i forhold til arbejdsforløbet. De eventuelle større ændringer, der ville kunne implementeres i projektet, blev ofte gennemgået i løbet af ugen og ikke i retrospektet. I den forstand blev retrospektets budskaber også til en selvfølge, hvilket beskadigede dets seriøsitet. Retrospektet synes umiddelbart ud fra denne erfaring at være mere effektivt og mere relevant i større grupper.

Yderligere Scrum-metoder

Der kunne være blevet brugt flere metoder inden for Scrum for at opnå det bedste resultat. Dette er tankerne, om hvorfor de blev frasorteret.

- **Velocity**

  Velocity er relateret til tidsestimering. Efter estimering af user stories og deres story points, som bliver givet via afstemning fra holdet ifm. størrelse og tidsestimering af opgaven, bliver der udarbejdet ét spring efter at kunne afgøre velocity. Tanken er, at man udregner den egentlige mængde af story points som blev opfyldt versus det estimat, man havde defineret. Alt efter mængden af ekstra eller manglende udfyldte story points, bliver velocity højere.
  Det virkede som for meget arbejde at begynde at udregne så specifikke målinger i så småt et projekt (i forhold til egentlige virksomheders projekter). Dette blev vurderet som et unødig brug er tid og ressourcer.

- **Burn-down chart**

  Et burn-down chart er et diagram, der bliver brugt til at estimere resterende tid i forhold til opgaver, der er tilbage, og deres tidsestimat. På grund af dette er denne metode tæt relateret til velocity.
  Det ville kræve endnu mere arbejde at visualisere projektet vha. en burn-down chart, og det blev af samme årsag som velocity nedprioriteret.

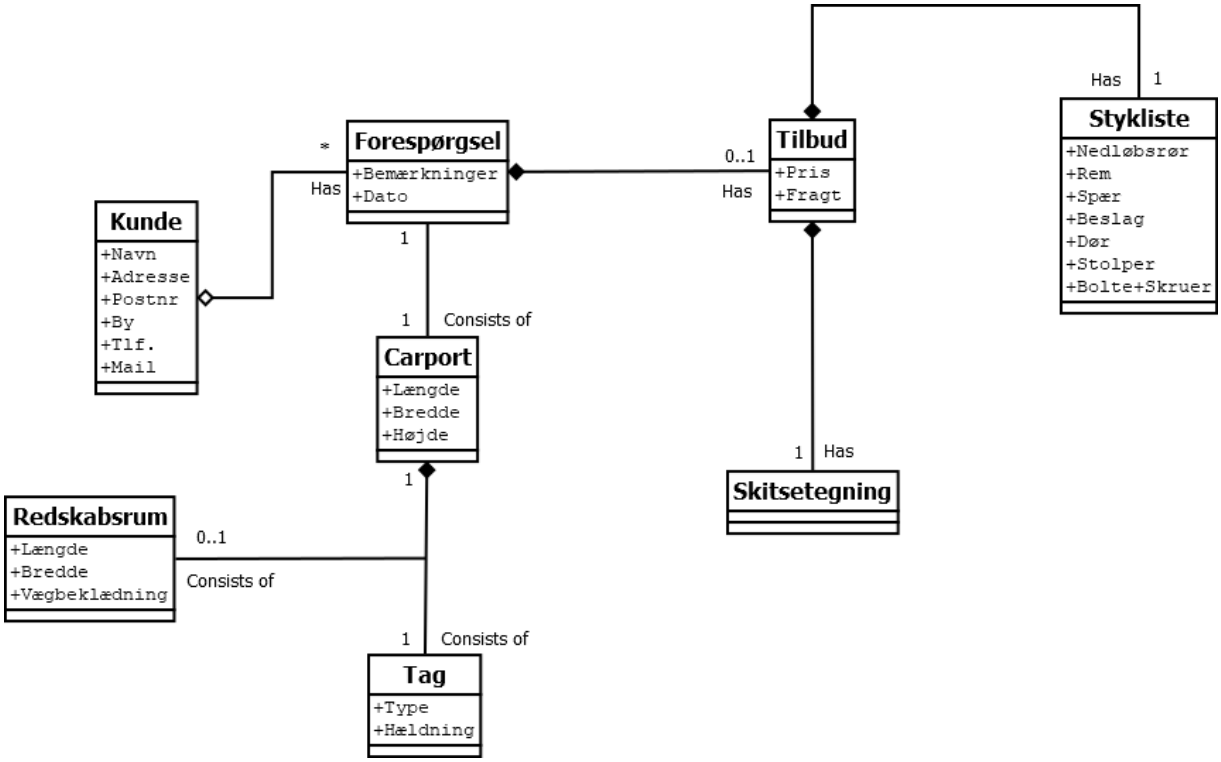
Generelt var tidsestimering et stort problem i Scrum-forløbet. Opgavernes tidskrav, relevans, samt deres egentlige tidsforbrug kunne være bedre dokumenteret og i forlængelse blevet reflekteret på senere i retrospektet. Dette var også den hovedsagelige årsag til retrospektets ineffektivitet.

Github

Som en del af øvelsen blev det stillet som opgave, at der skulle udbygges på den forståelse af github, som eleverne havde i forvejen. Dette betød hovedsageligt, at der skulle gøres brug af branches; separate opgaveversioner, udgivet over forskellige tidsintervaller med forskellige formål. I gruppens tilfælde blev der sat et arbejdsmiljø op, som delte versionerne op i hhv. en ugentlig branch før sprint planning kaldet "Master", en daglig branch med navnet "Develop", samt individuelle feature branches, der blev navngivet efter den task eller user story, som den givne feature var løsningen til. Denne fremgangsmetode betød, at "Master"-branch altid var funktionel og kunne bruges som demo til kundemøderne.

I et evaluerende perspektiv var dette succesfuldt i forhold til forventningerne, selvom det tog noget tilvænning, før det blev effektivt. Gamle vaner og mindre effektive metoder fra de tidligere projekter opstod i ny og næ i starten. Effektiviteten af den nye fremgangsmetode blev dog hurtigt åbenlys og det blev mere forståeligt hvorfor mange firmaer udnytter denne fremgangsmetode.

Domænemodel



Figur 2 Domæne Model

Her ses repræsentationen, af hvilke domænespecifikke dele der var planer om at implementere i det funktionelle program. Webapplikationen er blevet baseret på denne model, selvom der løbende er lavet ændringer til dette fundament baseret på kundens ønsker.

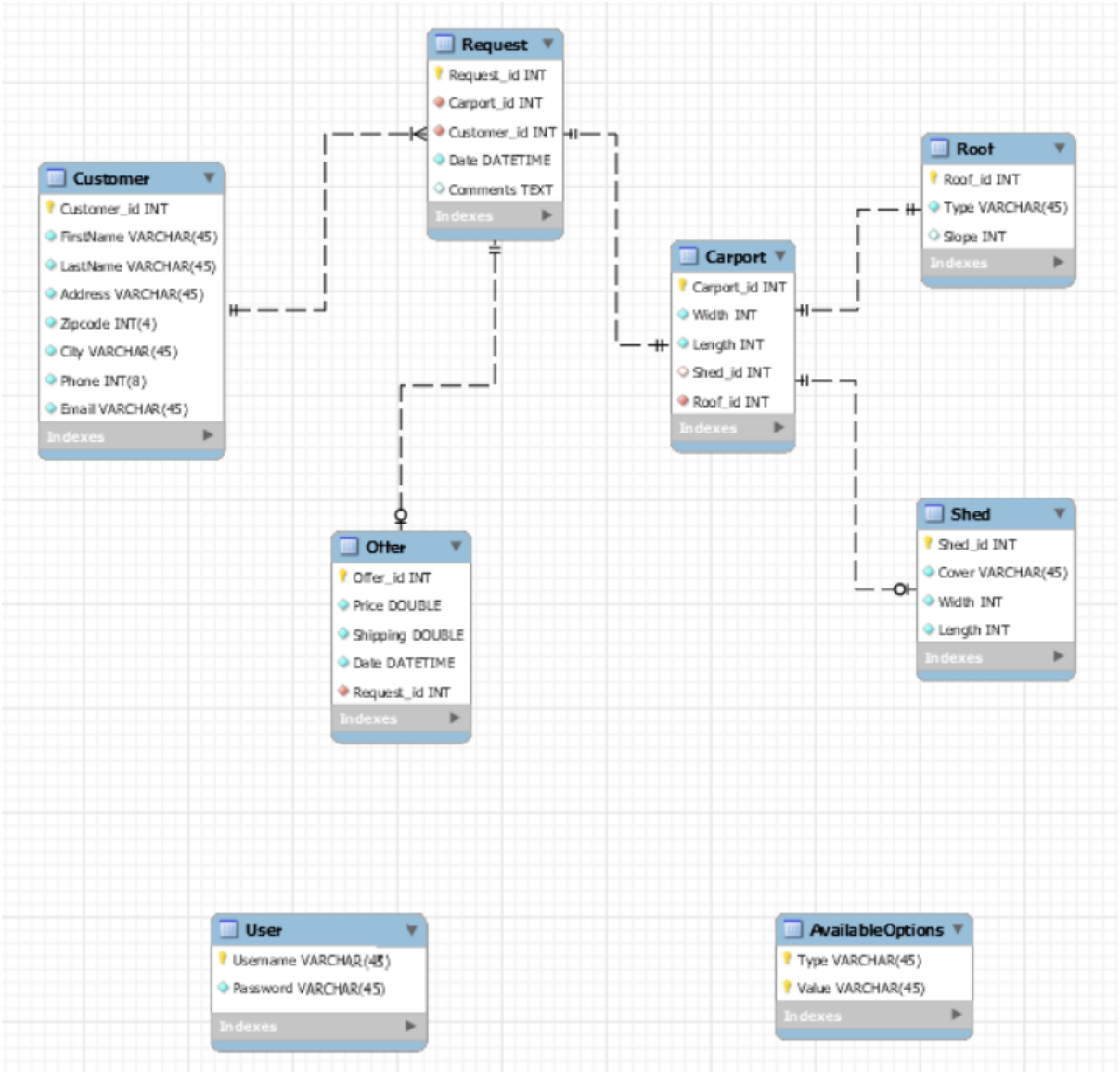
En kunde kan have mange forespørgsler tilknyttet, og her ses derfor den eneste 1-til-mange relation i programmet. En forespørgsel består derimod af én specifik carport, som består af et tag og eventuelt et redskabsrum - så disse er alle 1-til-1 relationer.

På samme måde kan en forespørgsel (i programmets nuværende form i det mindste) kun besvares med ét tilbud, og kun have ét sæt skitsetegninger og én stykliste. Der er "0..1"-relationer, fordi en carport ikke *behøver* at have et redskabsrum, og en forespørgsel starter ubesvaret og derfor uden tilbud tilknyttet.

Først og fremmest var ideen, at en stykliste og skitsetegning blev tilkoblet til en carport, som var blevet sendt til Fog som en forespørgsel. Derefter kunne medarbejderne logge ind og se både stykliste og skitsetegning. Det var til et kunde-review, at ønsket blev ytret, om at dette skulle laves om, til at man som kunde kunne se en skitsetegning, *før* man bestiller den. Det vil sige, at den domænemodel, der blev lavet før forløbet, ikke er helt korrekt, når det kommer til forholdene mellem skitsetegning og resten. I stedet for at forbinde skitsetegning til et tilbud, ville det derfor give mere mening at forbinde den til selve carporten; skitsetegningerne eksisterer, før både forespørgslen og tilbuddet skabes.

Derefter skal det også nævnes, at styklisten ses som værende forbundet til et tilbud. Dette er heller ikke helt korrekt, da man som medarbejder kan logge ind og se en stykliste, før et tilbud bliver lavet til forespørgslen. Denne ændring skyldes ikke et kunde-ønske, men blot at det giver mere mening i programmet, at styklisten er tilgængelig på denne måde. Det vil sige, at det derfor ville give mere mening at forbinde styklisten til enten forespørgslen eller carporten selv.

ER Diagram



Figur 3 ER Diagram

Her ses ER-Diagrammet for databasen. Det blev forsøgt at være konsistente ift. domænemodellen, hvilket betyder, at alle relationer i databasen derfor er de samme som de angivne relationer i domænemodellen. De fleste tabeller har en ID-attribut som primary key, der kan refereres af andre tabeller. Disse ID'er har derfor alle automatisk to constraints; NOT NULL og UNIQUE. Derudover er alle ID'er auto-incrementing, så man ikke behøver at tænke over værdierne af disse attributter i praksis - det er blot referencepunkter for systemet.

De fleste attributter i databasen har en NOT NULL constraint for at undgå datasæt, som mangler vigtige værdier - dog er der undtagelser. For at overholde "0..1"-relationen kan "Shed_id" i Carport-tabellen godt være NULL, og det samme gælder "Slope" i Roof-tabellen, da der findes flade tage. "Comments" i Request-tabellen kan også være NULL, da det er valgfrit at sende kommentarer med sin forespørgsel.

Det blev overvejet derudover at gøre "Email" til Customer-tabellens primary key i stedet for ID, da denne attribut også har en UNIQUE constraint, men det blev besluttet, at det var et pænere resultat med en ID som fremmednøgle i Request-tabellen.

For at opretholde konsistens ved funktionelle afhængigheder i databasen bruges fremmednøgler mellem tabeller. For eksempel er Request-tabellen forbundet til Customer-tabellen, for at en forespørgsels "Customer_id"-attribut også altid eksisterer som Customer i databasen. På samme måde er Carport-tabellen forbundet med Roof-tabellen og Shed-tabellen, så en Carport ikke kan oprettes med id'er på tage og skure, der ikke findes.

Der er to mindre tabeller, User-tabellen og AvailableOptions-tabellen, der ikke findes i domænemodellen. Dette er ikke en tilstræbt inkonsistens, men er blot fordi domænemodellen er bagud ift. funktionalitet, idet disse to tabeller først blev skabt indenfor sidste sprint. De to tabeller har hverken relationer eller ID'er, da der blev besluttet, at det ikke var nødvendigt i de givne tilfælde (selvom man kan argumentere, for at attributter som "length", "width", "slope" og "cover" kunne være fremmednøgler, der relaterede sig til AvailableOptions-tabellen), og fordi det blev forsøgt at holde disse ekstra-tabellers design så simpelt og overskueligt som muligt. I stedet bruger User-tabellen attributten "Username" som primary key, og AvailableOptions-tabellen bruger en sammensat nøgle, da der aldrig bør forekomme den samme kombination af kategori og værdi.

Det er derudover relevant at nævne, at der er flere dele af domænemodellen, hvor det er blevet valgt, at det ikke skulle gemmes i databasen, navnlig styklisten og skitsetegningerne. Disse to dele af programmet udregnes i stedet fleksibelt, når det skal bruges. I stedet for at gemme denne mængde data i databasen, udregnes skitsetegningerne øjeblikkeligt, hver gang de relevante websider indlæses. Styklisten udregnes for at undgå gentagende resursetungt arbejde kun én gang, når en carport vælges, og kan derefter hentes fra sessionen (se afsnit om Sessions). Så selvom der ganske vist er grund til at gemme styklisten i kortere tidsrum ad gangen, behøves den aldrig at kunne hives frem igen efter dette tidsrum, og gemmes derfor ikke i databasen.

Normalisering

Det er blevet forsøgt at holde databasestrukturen normaliseret, således at redundant data minimeres. Tabellerne overholder kravene for 1. normalform, siden der ikke er flere værdier i samme felt. Det er gjort sådan, at alle dele af en række er af den samme datatype, alle koloner har unikke navne i deres tabeller og der er ikke problemer, med hvilken rækkefølge dataene bliver gemt i tabellerne. Alle værdier er altså atomare eller udelelige. Et eksempel på at der er opnået 1. normalform i en tabel er, at i stedet for at have en "Request_id"-fremmednøgle i Customer-tabellen, som ville resultere i enten én kolonne med et uvist antal værdier eller i et fastlagt og udynamisk antal ensartede kolonner, er der i stedet en "Customer_id" i Request-tabellen. Dermed håndhæves én-til-mange relationen fra Customer-tabellen til Request-tabellen.

2. normalform er opnået, ved at databasen har opnået 1. normalform, samt at der ikke er nogen partielle afhængigheder, da der i denne database kun er én tabel med en sammensat primærnøgle (AvailableOptions-tabellen). Denne tabel har ingen andre attributter end selve primærnøglen, der kan være afhængige af noget.

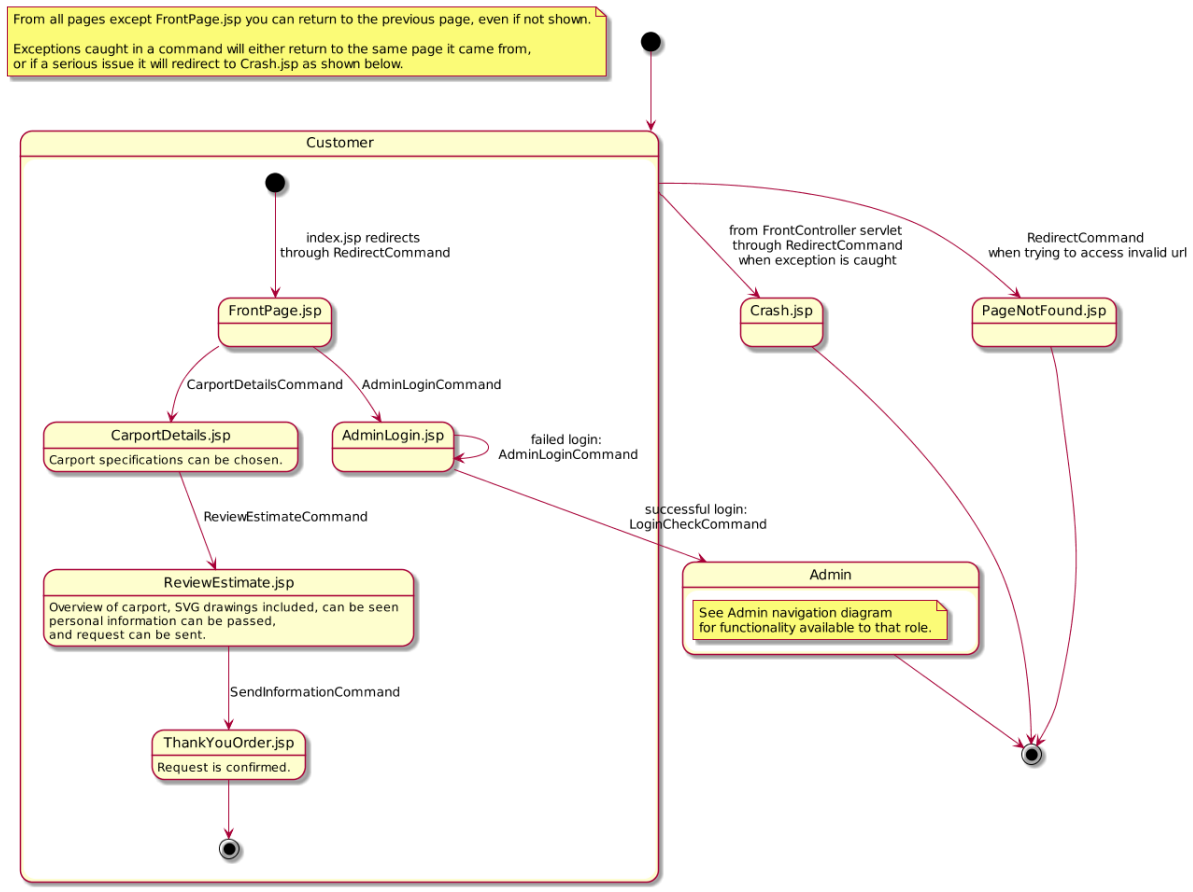
Det kan ses, at 3. normalform i de fleste tilfælde er opnået, ved at tabellerne ikke har nogle transitive afhængigheder, altså at der ikke er nogle værdier der er afhængige af ikke-primære attributter, samt at de opfylder kravene for 1. normalform og 2. normalform. De mange 1-til-1 relationer i databasen er bl.a. skabt med afsæt i dette normaliseringsprincip. For eksempel kunne alle attributter i Shed-tabellen ligge i Carport-tabellen i stedet, og på grund af 1-til-1 relationen være afhængig af "Carport_id"-nøglen som alle andre af tabellens attributter. Dog ville skurets længde, bredde og beklædning være indbyrdes afhængige på den måde, at hvis én af attributterne var NULL, skulle de to andre også være NULL.

Derfor er disse værdier lagt i en ny tabel. Selvom det samme ikke kan siges om andre 1-til-1 relationer som f.eks. Roof-tabellen, så gav det god mening at anvende samme designprincip og på den måde sikre, at dette data forblev normaliseret – selv hvis programmet pludselig blev ændret, så man godt kunne bestille carporte uden tag. Samtidig bliver det meget mere overskueligt, når dataet på denne måde er splittet op imellem flere tabeller. Havde man alle kolonner fra tabellerne Request, Carport og Roof liggende i samme tabel, ville det hurtigt blive for mange forskellige emner man kiggede på samtidig – og man ville desuden have afveget kraftigt fra domænemodellen.

Customer-tabellen er dog ikke i 3. normalform og bør deles op, idet attributterne "Zipcode" og "City" er indbyrdes afhængige af hinanden. Der er en direkte forbindelse mellem hvilket postnummer man har, og hvilken by/kommune dette postnummer repræsenterer. Derfor kunne man lægge disse to over i en tabel for sig selv og blot lade Customer henvise til denne nye tabel med en fremmednøgle.

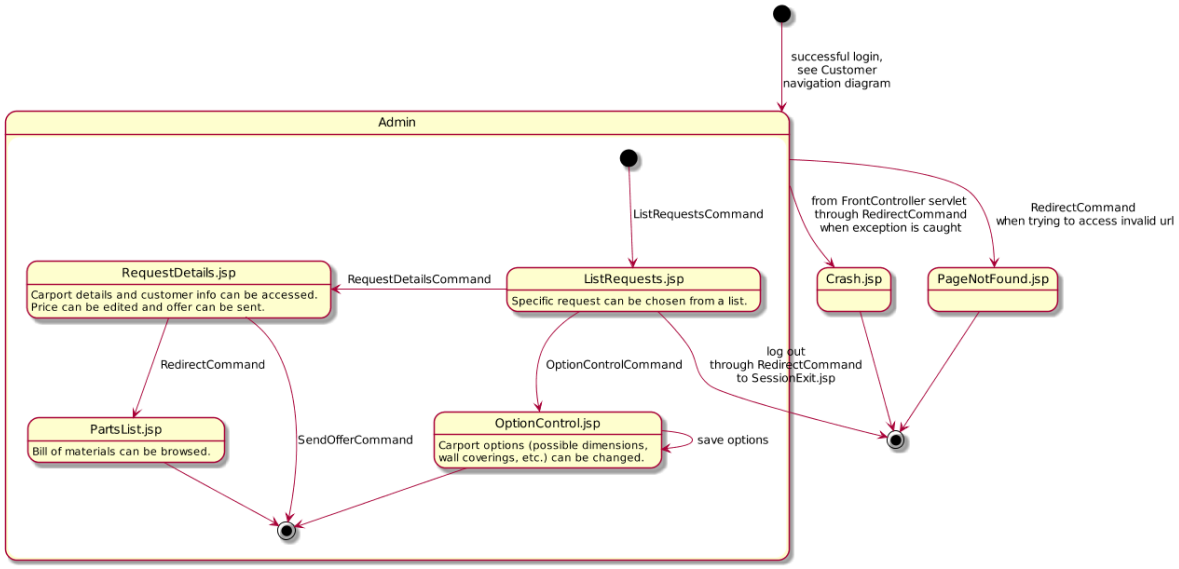
Navigationsdiagram

Diagrammet I figur 4 beskriver forholdet mellem alle JSP-sider og de dertil hørende servlet-Commands. Det giver et overblik over, hvordan man i rollerne som hhv. kunde og administrator kan navigere rundt. Der er heri benyttet UMLs "composite-states" for at vise hvilke roller, der har adgang til hvilke funktionaliteter.



Figur 4 Navigationsdiagram Customer

Man starter på en forside, hvorfra man via en navbar kan fortsætte til enten bestillingssiden for carporte eller til login-siden for admins. På bestillingssiden vælges specifikationer til carporten, hvorefter man kan få et overblik over det man har valgt, en skitsetegning af carporten fra oven såvel som fra siden, og et automatisk udregnet prisestimat. Er man tilfreds med resultatet, indtastes ens personlige kundeoplysninger, og man kan sende en forespørgsel til Fog. Derefter bliver man sendt til en side, der bekræfter ordren og informerer kunden, om at den er blevet sendt.



Figur 5 Admin

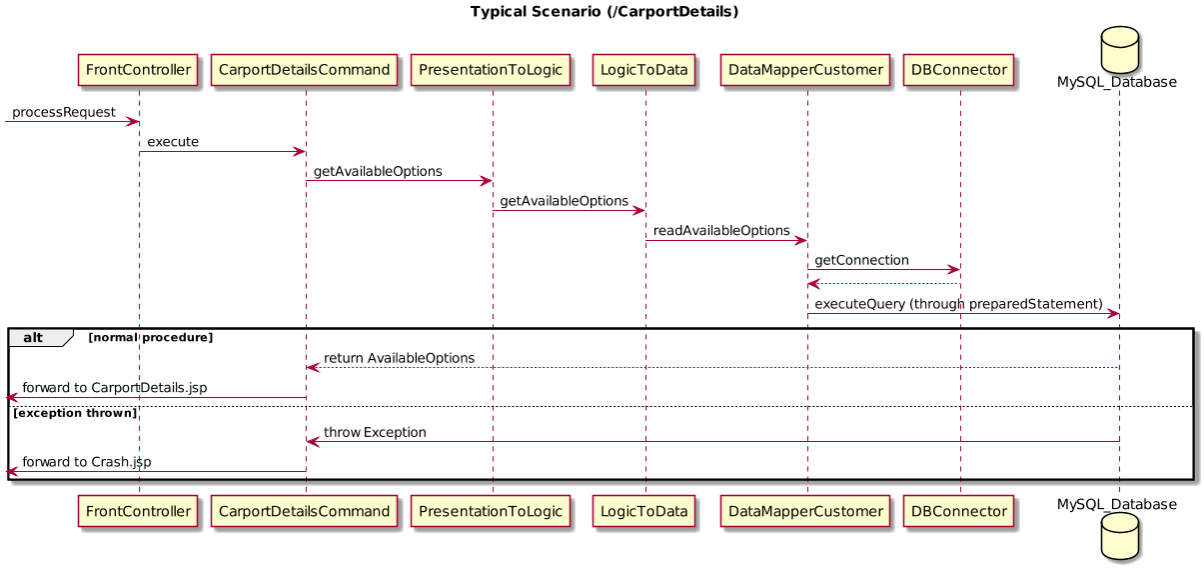
Vælger man derimod at logge ind som admin, bliver flere funktionaliteter tilgængelige, hvis loginforsøget virker. Først og fremmest vises en liste over forespørgsler i databasen. Vælger man en forespørgsel fra listen, sendes man til en oversigt over ordren, hvorfra man kan se carportens specifikationer, kundeinformationer, skitsetegninger, en stykliste delt op i tre grupper (træ, tag og skruer) samt et prisestimat. Prisen kan reguleres, hvis ønsket, og et tilbud kan derefter sendes til kunden via e-mail. Admins har derudover mulighed for at tilgå listerne over tilladte specifikationer for carporte. Herfra kan man f.eks. ændre, på hvilke længder og bredder, der kan vælges til carport og skur, og hvilke tagtyper, et tag med rejsning kan have.

Som vist på Customer-diagrammet, kan man fra de fleste sider selvfølgelig vende tilbage til den forrige side. Derudover er der gjort brug af en navigation bar øverst på alle sider, der linker til jsp-siderne CarportDetails.jsp og hhv. AdminLogin.jsp, når man ikke er logget ind endnu, eller ListRequest.jsp, hvis man allerede er logget ind som admin. Som Admin kan man også altid logge ud fra navbar'en.

Forsøger man at tilgå en side, der ikke eksisterer, navigeres man til PageNotFound.jsp, samt hvis der opstår en fejl i programmet, sendes man til Crash.jsp. Dog er der mange undtagelser til dette, hvor programmet er struktureret således, at exceptions i stedet sender brugeren tilbage til samme side eller en tidligere side med en relevant fejlbesked. Dette er f.eks. tilfældet i SendInformationCommand, hvis man indtaster et postnummer, der ikke leveres til; eller på login-siden, hvor siden genindlæses med en fejlbesked, hvis brugernavn eller password ikke er korrekt. Dette gælder også i OptionControl, hvor man også få en besked på samme side, hvis man indtaster invalide specifikationer – f.eks. bogstaver i kategorien taghældning.

Sekvensdiagram

Diagrammet I figur 6 viser, hvordan et typisk scenarie forløber, og hvordan forskellige dele af programmet kalder hinanden, med udgangspunkt i en af de mest almindelige handlinger i programmet; at tilgå path'en /FogProjekt/c/CarportDetails, der kan blive kaldt fra f.eks. nav-bar'en.



Figur 6 Sekvensdiagram CarportDetails

Der brugt et Front Controller/Command pattern, hvor front controller'en reelt er den eneste servlet i web-projektet. Når der sendes en HTTP request fra en client browser med en path i stil med "/c/...", fanges denne af servletten, som matcher den givne path med et HashMap af Command-klasser for at finde den rette Command til formålet.

Servletten uddelegerer derefter arbejdet til den fundne Command og kalder execute på den, uanset hvilken klasse det endte ud med at være, ved hjælp af polymorfi. Imens en Command udfører al relevant kode og sender brugeren til den rette JSP-side, er servletten på den måde hurtigt åben for at modtage flere HTTP requests.

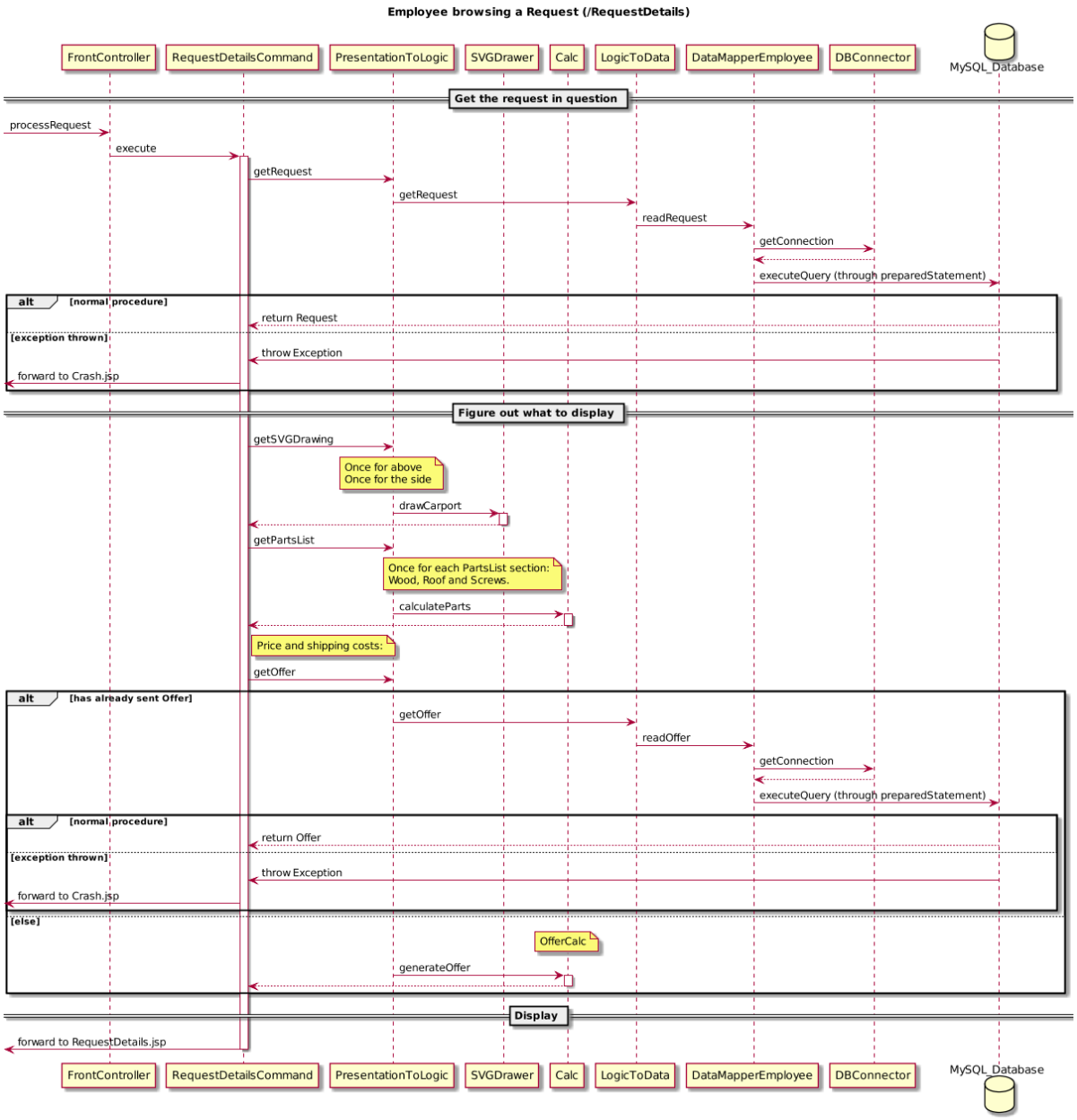
En Command udfører typisk al arbejde igennem facaden PresentationToLogic, som får fat i relevante resultater af udregninger fra logik-laget, og kalder facaden LogicToData, hvis hentning eller manipulation af data påkræves. Igen i det typiske forløb kræves der ingen udregninger i logiklaget, og den ene facade kalder derfor bare den anden facade uden mellemled eller andre samtidige handlinger, og man kan derfor synes at én facade ville have været tilstrækkeligt.

Men til tilfælde som eksemplet I figur 7 (/c/RequestDetails), hvor algoritmerne for SVG-generering og styklister kommer i spil, er det vigtigt for overblikket og for at overholde trelagsarkitekturens principper, at der er en ekstra facade ned til datalaget.

I datalaget bruges en data mapper, der indeholder alle metoder til at hente og manipulere data fra eksterne kilder – i dette tilfælde en MySQL database. Data mapperens metodenavne forholder sig til SQL's CRUD-standarder, så Create, Read, Update eller Delete indgår i navnene for en hurtig forståelse af deres interaktion med databasen.

Når data mapperen bliver kaldt, forsøges allerførst at opnås en forbindelse til databasen fra en DBConnector klasse. Hvis dette lykkes, sendes en SQL-query til databasen, og et ResultSet returneres med informationer, som med det samme omformes til relevante entitetsklasser.

Det færdige resultat – i dette tilfælde en LinkedList med Strings – returneres hele vejen tilbage til Command, og brugeren bliver sendt til den rette JSP-side, eller i få tilfælde, som i SendOfferCommand sendes man blot tilbage til servletten med en ny path. Opstår der en fejl i forbindelse med getConnection eller CRUD-metoderne i data mapper'en, smides en exception hele vejen tilbage gennem systemet til Command'en, som fanger denne DataAccessException og sender brugeren til Crash.jsp i stedet. Til at vise dette i sekvensdiagrammerne bruges såkaldte alt-blokke, som repræsenterer conditional forgreninger i programmet. Det viser, at hvis intet går galt, returneres resultatet fra databasen og processen fortsætter, men det viser også alternativet (thrown exception og Crash.jsp).



Figur 7 Sekvensdiagram RequestDetails

I eksemplet I figur 7, hvor RequestDetailsCommand skal have fat i et Offer for at kunne vise en pris, er der endnu en forgrening og derfor endnu en alt-blok. Denne viser, at hvis der ikke allerede er et sendt et tilbud til kunden for denne forespørgsel, så skal et prisestimat udregnes og returneres fra logiklaget. Hvis der derimod allerede er tilsendt et tilbud, så skal dette være til at finde i databasen, og PresentationToLogic kalder derfor i stedet LogicToData og derigennem data mapperen. Desuden forekommer SQL-forgreningen fra før igen for at vise muligheden for, at et problem opstår i databasen.

Særlige forhold

Entitetsklasser

Der bliver brugt mange entitetsklasser i dette projekt, som bruges til at sende samlede pakker af data rundt mellem forskellige dele af programmet i forståelige domænespecifikke betegnelser. Mange af klasserne bruges som en slags kopi af en tabel i databasen, hvilket gør at klassen kan bruges som en nem måde at skrive den nødvendige information til databasen. "Shed" er et godt eksempel, hvor man kan se, at dens attributter er 1-til-1 mellem databaseversionen og klasseversionen; klassen tager imod "id", "length", "width", og "wall_coverings" og de 4 kolonner i "Shed"-tabellen bruger præcis samme navngivning. Mellem java og databasen sker dog en ofte konvertering. "Offer" har f.eks. Boolean-datatypen som ikke findes i MySQL, men dette bliver der taget højde for fra JDBC's side, ved at de konverterer boolean værdier, der bliver modtaget. Denne konvertering går begge veje. På samme måde der har også været nødt til at konvertere javas LocalDateTime til SQL's Date-datatype og tilbage igen.

Enums

Der er gjort brug af en enum-klasse én gang i projektet. En "Location"-enum er lavet til udregning af fragtomkostninger i OfferCalc, hvor en switchcase bestemmer hvilken pris der er den rette ift. postnummerets oprindelse - Sjælland, Fyn eller Jylland. På samme måde bruges denne Location-enum i Customer-klassen til at tjekke, hvorvidt et givent postnummer overhovedet er validt og kan leveres til. Ift. informationer fundet på Fogs egen hjemmeside, leverer de ikke til Grønland og Færøerne, så postnumre mellem 3700 og 4000 er derfor ikke tilladt og resulterer i en IllegalArgumentException. Man kunne også have valgt blot at bruge en String eller en Integer, der f.eks havde Sjælland som 0 og Fyn som 1, men enums giver et meget bedre designmæssigt overblik over betydningen af værdier, og brugen af enums er derfor god praksis i mange tilfælde.

Arkitektur

Der er blevet brugt en trelagsarkitektur hvor java klasserne er blevet delt op i data, logic og presentation. I data laget er datamappers, dbconnectoren, de "custom exceptions" der er blevet udviklet til programmet og java klasserne for objekter (så som Parts, Carport, Roof, Shed, etc.) der bliver brugt til nemmer håndtering af dataet. Logic laget indeholder to facader, en der giver adgang til data delen fra logic (LogicToData) og en der giver adgang til logic fra presentation (PresentationToData), facaderne er med til at gøre det mere overskueligt, når der bliver brugt dele af kode fra forskellige lag, det opnås ved at samle alt adgang mellem de forskellige lag i facaderne. I logic laget er der også de forskellige udregningsalgoritmer (WoodCalc, RoofCalc, OfferCalc, etc) og svg tegningernes algoritmer (SVGDrawerFromAbove og -FromSide) der bliver brugt til at producere en visuel repræsentation af de carporte der bliver bestilt. Presentation laget indeholder projektets frontcontroller og commands. Presentations laget er det lag der forbinder alt det visuelle, ved at fungere tæt sammen med de forskellige "web pages", de modtager information fra jsp sider som så bliver behandlet, og derefter enten sendt videre til andre jsp sider (CarportDetailsCommand) eller gemt i databasen (SendInformationCommand).

Web Pages er delt op i 4 forskellige mapper der bliver brugt: "CSS", "WEB-INF", "Images" og "inclusions". "CSS" indeholder alle css filerne der bliver brugt til at style jsp siderne. "WEB-INF" indeholder alle de jsp sider der er forbundet til Commands i præsentationslaget, de har to slags funktioner, nogle er der for at tage imod information, som så kan sendes videre (CarportDetails) eller tjekkes (AdminLogin), den anden slags er dem der informere kunden/brugeren om noget, ved enten at vise en besked (Crash, PageNotFound) til brugeren eller vise information fra databasen udregnet fra logiklaget(ReviewEstimate, RequestDetails, PartsList). "Images" er en mappe lavet til at indeholde billeder der skal bruges på jsp siderne, der er to billeder i mappen (fog, fogCenter), et billede til at vise Fogs logo, der bliver brugt på forsiden og et som bliver brugt som en visuel repræsentation af en fysisk butik. "inclusions" mappen indeholder de jsp sider der har funktionalitet der bruges på flere jsp sider, der er en jsp side i denne mappe (NavBar). Denne NavBar bliver inkluderet på de fleste JSP-sider ved brugen af jsp:include tags, siden den indeholder links til de forskellige relevante hovedsider.

Entitetsklasser og custom exceptions er placeret i hver deres mappe i datalaget, da dette var den anbefalede placering fra flere lektorer. Dog kunne man også have valgt at placere disse mapper helt uden for trelagsarkitekturen, så man i stedet havde dem liggende direkte under Source Packages.

For at sørge for at opretholde arkitekturen i programmet, for at være sikker på at de overordnede mål og funktioner for programmet, såvel som for at gøre programmet fremtidssikret mod nye implementationer af samme funktionalitet, bruger en del interfaces i programmet. Mange af de større klasser benytter interfaces som deres interne kontrakter. For alle disse interfaces bruges polymorfi. Der er interfaces som bliver brugt til enkelte klasser som "DataMapper" (eksempelvis DataMapperUserInterface, som instantieres med polymorfi ved "user_dao = DataMapperUser();" i *LogicToDataImpl*) og interfaces der bliver brugt som facader til adgang mellem de forskellige data lag (eksempelvis PresentationToLogic, som kan ses instantieret med polymorfi ved f.eks. "final PresentationToLogic PRES_TO_LOGIC = new PresentationToLogicImpl();" i de fleste Commands).

I dette program bliver der også brugt abstrakte klasser i form af "Command"-klassen, som bliver brugt til at sætte "FrontControlleren" op, og bliver extended i 10 forskellige implementationer under "commands"-mappen der alle instantsieres i "Command"-superklassen. FrontController får fat i den rette Command-extension, og kalder derefter execute på den ved hjælp af polymorfi.

PartsList-algoritmer

Et af kravene til projektet var, at man baseret på givne carport mål, kunne få udregnet en stykliste af de materialer, der bliver brugt til at bygge denne carport. Den måde, det er håndteret på i denne opgave, er ved at dele listen op i tre forskellige klasser, som hver især udregner en del af carporten. Dette er delt op i træ-materialer, tag-materialer (materialer brugt til tagdelen af carporten) samt resten (skruer og lign. diverse dele). Denne opdeling er baseret på de to eksempler af carporte, der blev udleveret ved opgavens start.

Den overordnede struktur og løsning til dette problem blev til at udregne hver enkelt del for sig. Dette er blevet gjort af flere årsager, blandt andet og hovedsageligt fordi denne struktur gør det utroligt nemt senere at ændre på eventuelle forhold, der pludselig skulle ændres - for eksempel, hvis der skal flere træstolper til et mindre tag. Strukturen er baseret, på at alle de givne klasser hver især har en metode, der tager et carportobjekt som parameter, altså den carport som styklisten skal udregnes efter, og derefter gennemgår alle nødvendige hjælpemetoder brugt til at kunne udregne styklisten. Det vil sige, at alle metoder til alle givne muligheder til en carport skal være taget højde for; en carport med taghældning og en uden skal nemlig udregnes forskelligt, i forhold til hvilke materialer der bliver brugt.

Som eksempel kan metoden til udregning af dele af spærene for et tag uden rejsning ses på nedenstående billede. Dette er basis, for hvordan alle disse hjælpemetoder ser ud. Det kan ses, at der findes en meterpris på materialet, hvorefter prisen bliver udregnet baseret på længden i meter, da "length" variablen har enheden centimeter. Derefter bliver et part-objekt returneret med disse værdier.

```
/**
 * Help method used to calculate some of the parts needed for the carport (wooden parts).
 * @param carport predefined carport object with length and width.
 * @return the amount of rafter boards based on the width of the carport. (Spær i danish)
 */
private static Part calcRafterBoardNorm1(Carport carport) {
    int length = carport.getWidth();
    double meter_price = 33;
    double price = meter_price * (length / 100);
    return new Part("45x195mm. spærtræ ubh. spær", length,
            (int) Math.ceil(Math.ceil(carport.getLength() / 100) * 2), "stk", "Spær monteres på rem", price);
}
```

Dette ses især i WoodCalc klassen (algoritmen for træ-materialerne), da der bruges vidt forskellige materialer til en carport med tag hældning og en med fladt tag. Dog ved denne struktur gøres det meget nemmere at håndtere, præcis hvilke forhold der skal bruges til materialer, altså hvor mange og hvor lange de skal være, og hvornår disse materialer skal bruges.

```
//Parts for sloped roof
if (carport.getRoof().getRaised()) {
    parts.add(calcRafterBoardSlope(carport));
    parts.add(calcFasciaBoardsSlope1(carport));
    if (carport.getShed() != null) {
        parts.add(calcFasciaBoardsSlope2(carport));}
    parts.add(calcWaterBoardSlope(carport));
    parts.add(calcBarge(carport));
    parts.add(calcRoofLaths1(carport));
    parts.add(calcRoofLaths2(carport));
} //Parts for nonsloped roof
else {
    parts.add(calcRafterBoardNorm1(carport));
    parts.add(calcFasciaBoards1(carport));
    parts.add(calcFasciaBoards2(carport));
    parts.add(calcFasciaBoards3(carport));
    parts.add(calcFasciaBoards4(carport));
    parts.add(calcWaterBoard1(carport));
    parts.add(calcWaterBoard2(carport));
}
```

Ovenstående billede viser dette eksempel. Alle metoder, der tager højde for om carporttaget har hældning, bliver kaldt, hvis carport-objektets tag har en hældning, og de andre metoder blive kaldt, hvis det ikke har. Dette er med til at skabe overblik og meget nemmere at ændre senere. I klassen RoofCalc (algoritmen for tag-materialerne) er det opbygget med samme struktur, dog er der tilføjet konstante værdier i fields til alle de værdier og forhold, som brugeren senere kunne have brug for at ændre. Dette er næsten ideelt, i forhold til hvordan WoodCalc ser ud, da der i WoodCalc ikke er tilføjet lignende konstante fields til alle de brugte værdier og forhold.

RoofCalc benytter sig af standard pythagoras og cosinus til at udregne tagets ukendte dimensioner, og i forlængelse deraf udregne hvor mange dele, der skal bruges, f.eks. tagsten.

I FittingsAndScrewsCalc (algoritmen for alle beslag og skruer) bliver der brugt en LinkedList af Parts hentet fra "WoodCalc", som indeholder værdier, der skal bruges til udregninger. Der bliver brugt både Part-listen og et Carport-objekt til dens udregninger, hvilket gjorde, at der skulle vurderes, hvornår de skulle sendes videre ned i udregningen alene, eller om der var brug for dem begge to. Et eksempel på dette fra algoritmen er "getScrewsBeams"-metoden, hvor informationen fra Carport relateret til hvorvidt der er et "shed" eller ej, bliver brugt til at vælge, hvor mange informationer, der skal hentes fra "WoodCalc" listen. Dette bliver gjort i en stor mængde af metoderne, fordi tilstedeværelsen af et Shed kan ændre drastisk, på hvor mange af den del, der skal bruges.

Egentlige Roof-forhold

Som en del af projektet skal man kunne indtaste værdierne for en Carport med et fladt tag. Med scenariet fulgte der to eksempeltegninger med pre-definerede værdier med.[1] Med dette i tankerne, blev der ikke fastsat definitioner på alle værdier af elementernes dimensioner. F.eks. det flade tags tagplader. Derfor kunne man kun estimere sig frem til hvor store tagpladernes bredde var, hvis man ville opnå det korrekte resultat.

Det viser sig dog, at alle tagplader af typen "Plastmo Ecolite Blåtonet" kun varierer i længden. Tagpladerne er et egentligt produkt og alle pladevariationerne har samme bredde. Det viser sig også, at man gennem plastmos hjemmeside kan generere sit eget tag med de angivne dimensioner[2] og at dimensionerne ikke er de samme som i eksempeltegningerne. Dette er enten fordi at:

1. Den definerede bredte på tegningen er upræcis
2. Den definerede bølgelængde på tegningen er upræcis
3. Fejl I den bruget af bølgelængderne af pladerne.

Der tages derfor forbehold for, at de default-dimensioner som er defineret i projektet, muligvis ikke er de korrekte, siden de er defineret ud fra scenariets tegninger og ikke virkelighedens generering. Hvis klienten undrer sig over det, kan der nemt ændres i værdierne i RoofCalc-klassens konstanter, men man bør være opmærksom på, at 'problemet' eksisterer. Se evt. Bilag 13.

SVG

SVG-kode, eller Scalable Vector Graphics, er en måde at implementere vektorgrafik i sin html kode. Det er på denne måde, at kravet om at kunne vise skitsetegninger af en carport, både oppefra og fra siden, blev løst. Baseret på de værdier, som er givet i de carporte objekter, som skal skitseres, såsom længde, vil disse tegninger blive opbygget af rektangler og enkelte linjer med matchende værdier. Dette betyder at den endelige carport og carport-skitsen vil få et rigtigt størrelsesforhold til hinanden.

I modsætning til blot at have html koden i de forskellige JSP-sider, så er der blevet gjort brug af dynamiske metoder i SVGDRawerFromAbove og SVGDrawerFromSide der returnerer og genererer denne kode. Til dette formål har vi, på samme måde som med algoritmeklasserne (Calc), angivet adskillige konstanter og variable øverst i klasserne med initialisering i deres constructors, som derefter bruges løbende igennem hele SVG-genereringen. Dette gælder f.eks. "startX" og "startY", som angiver tegningens øverste venstre hjørne, "yEaves", som angiver tagets udhæng langs carportens længde (altså hvor langt det går ud over stolperne på y-aksen set oppefra), og "maxDistanceBetweenPoles", som angiver hvor lang en carport må være, før en tredje stolpe kræves imellem de to andre.

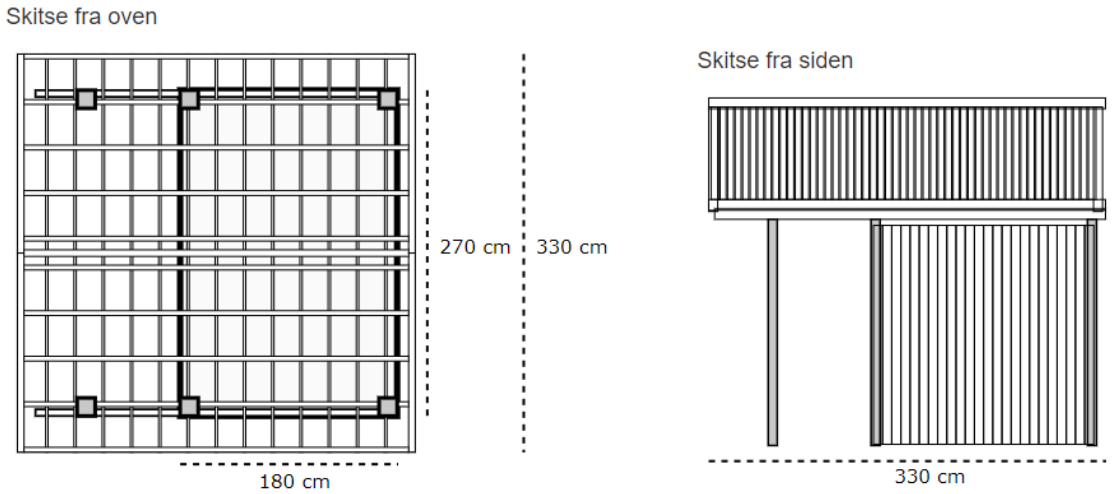
Der bliver brugt tre forskellige typer SVG-elementer til tegningen af carporten: rektangel ("rect"), linje (line) og tekst. Hver af disse elementer har deres egen private metode, som tager imod dimensioner og detaljer (fx tykkelse af streger og farve) og derefter skaber SVG-koden for den lille bid af det totale billede. Disse tre metoder bruger alle metoden "cmToDrawUnits" (SVGDrawerFromAbove, linje 418), som tager ethvert reelt mål for carporten og konverterer det, således at 1 cm i virkeligheden altid vil blive til præcis 0,25 mm på hjemmesiden.

[1] https://datsoftlyngby.github.io/dat2sem2019Spring/Modul4/Fog/CP01_DUR.pdf

[2] https://www.plastmo.dk/beregnere/tagberegner/trapeztag/ecolite/blaatonet.aspx

Ved hjælp af ovenstående hjælpemetoder, gennemføres alle udregningerne, og delene tegnes del for del; lag oven på lag, indtil resultatet har samlet en String, der indeholder alle rektangler, linjer og mål (tekst) i præcist måleforhold 1:40. Denne returneres til og tegnes i frontend (se RequestDetails og ReviewEstimate jsp-sider og commands).

Ekstra virkemidler og SVG-attributter, som der bliver benyttet sig af for at gøre tegningerne mindst muligt kaotiske og mest muligt forståelige, indebærer "stroke-dasharray" på linjerne, for at gøre dem stiplede, "stroke-width" på rektangler for fx at gøre grænserne på redskabsskuret set oppefra tydelige i virvaret af andre streger, og "fill" på rektangler for fx at udfylde stolperne med en grålig farve som får også *dem* til at skille sig ud.



Sessions

I dette program bliver sessions brugt til at gemme information, der senere skal bruges på flere forskellige sider, så man undgår at generere den samme information flere gange end nødvendigt (så som når et Offer bliver gemt som "estimate" på sessionen i ReviewEstimateCommand). Sessions bruges også til at holde data, der skal være konstant tilgængeligt. Dataet er gemt på denne måde indtil der gøres noget for aktivt at fjerne det, eller indtil sessionen løber ud. Et eksempel er, når en admin logger ind, og deres information bliver gemt på sessionen, så de ikke behøver logge ind hver gang de udfører en admin handling. Der bliver derudover brugt session.invalidate() metoden, som er til at fjerne alt fra session igen, når det påkræves, f.eks. ved log out, eller når et nyt Request vælges fra ListRequests.jsp, og alle dets attributter, som omhandler et tidligere valgt Request, skal fjernes fra session igen.

I situationer hvor sessions ikke bruges, skyldes det, at de attributter som sættes i Command'en, kun er relevante i netop dette tilfælde, hvor "forward" videresender det. Dette er f.eks. tilfældet i CarportDetailsCommand, hvor alle options findes i databasen og derefter gemmes som attributter. Her skal de kun bruges i CarportDetails.jsp, til at udfylde diverse dropdown lists. De er dog ikke længere relevante, så snart brugeren vil have et prisestimatet for carporten - så den gemmer i stedet de valgte specifikationer som et Carport objekt under Offer-attributten på session. Derved kan de hurtigt findes frem igen, når en forespørgsel afsendes. Et andet eksempel, hvor session ikke er nødvendigt at bruge, er fejlbeskeder, hvor brug af session ville betyde, at en fejlbesked bliver ved med at blive vist, selv når årsagen til fejlen er rettet.

Exceptionhandling

Der er blevet benyttet custom exceptions til at specificere de enkelte bugs som opstår under programmet. De fleste omhandler som extensions af InvalidArgumentException forkert brug af tegn, dvs. f.eks. forkerte symboler ved navngivning af kunden, adresse, postnummer, og så videre.

En custom exception er også blevet brugt til at gøre interfaces mere generelle og til at fremtidssikre programmet ift. nye implementationer. DataAccessException er en overordnet exception-type til brug, når noget går galt i forsøget på at tilgå data i datalaget. Uanset om det er en SQLConnectionException, en SQLException eller en exception der har med en hel anden type database eller filhåndtering at gøre, så kastes blot en DataAccessException ned gennem systemet for at fortælle, at noget gik galt i den proces. Dette betyder, at metodedeklarationerne i interfaces til data mappers såvel som til facader blot kan smide en universel exception, der kan betyde fejl for en hvilken som helst datakommunikation, i stedet for at de smider SQLExceptions og dermed bliver afhængige af at bruge MySQL for evigt.

Der kastes exceptions under forhold, som er anset for at være ulovlige. Disse exceptions bliver derefter oftest fanget i en try-catch i en Command, og her bliver der skrevet til loggeren, som noterer problemerne i .log filen under den angivne path og til konsollen. Herefter sætter den en errormessage og sender den til en .jsp side, som er lavet til at reagere ud fra den angivne errormessage. For eksempel, hvis der bliver indtastet et forkert username eller password, bliver den fanget af en WrongCredentialsException, definerer errormessage som en besked om forkert brugernavn/password, og sender den tilbage til loginsiden, som registrerer, at en errormessage eksisterer. Dette kan ses i LoginCheckCommand l. 57, 58, 68-71 og i Adminlogin.jsp l. 28-42 under error.equals("WrongCredentials").

```
if (user == null || !user.getPassword().equals(password)) {
    throw new WrongCredentialsException("Wrong credentials");
```

```
} catch (WrongCredentialsException ex) {
    request.setAttribute("errormessage", "WrongCredentials");
    logger.log(Level.SEVERE, ex.toString(), ex);
    loadJSP(request, response);
```

```
<%
    String error = (String) request.getAttribute("errormessage");
    String errormessage = "";

    if (error == null) {
        errormessage = "";
    } else if (error.equals("WrongCredentials")) {
        errormessage = "<p style=\"color:red\">Det indtastede brugernavn eller password er forkert.</p>";
    } else if (error.equals("EmptySession")) {
        errormessage = "<p style=\"color:red\">Vær venlig at logge ind.</p>";
    } else {
        errormessage = "";
    }
    out.println(errormessage);
%>
```

Et forsøg på at gå til en .jsp side uden den nødvendige information, smider brugeren over til den side, hvor der mangler input. F.eks. hvis brugeren forsøger at gå ind på en side, som kun adminstratoren kan åbne, uden at have logget ind forinden, vil brugeren blive smidt tilbage til login. Her vil brugeren få en besked om at logge ind før de næste sider bliver tilgængelige. Dette kan f.eks. ses på ListRequestsCommand l. 53-55, 61-65 og igen AdminLogin.jsp l.28 - 42 under error.equals("EmptySession").

```
if (request.getSession().getAttribute("user") == null) {
    throw new EmptySessionException("Attempt at admin access in listRequests without admin on session");
}
```

```
} catch (EmptySessionException ex) {
    ex.printStackTrace();
    request.setAttribute("errormessage", "EmptySession");
    logger.log(Level.SEVERE, ex.toString(), ex);
    request.getRequestDispatcher("EmpLogin").forward(request, response);
}
```

Hvis der opstår fejl i datalaget, bliver der via DataAccessException sendt dispatch til crash.jsp

```
} catch (DataAccessException ex) {
    ex.getCause().printStackTrace();
    logger.log(Level.SEVERE, ex.toString(), ex);
    request.getRequestDispatcher("Crash").forward(request, response);
```

Regex

Som en udvidelse af webprojektets fejlhåndtering er der implementeret brugen af regular expressions både i frontend og i backend, så man både på client-side (JSP) og server-side (exception handling) totalsikrer programmet mod indtastningen af invalide oplysninger. Funktionaliteten, der bliver brugt regex i, er når kunden indtaster sine personlige informationer på ReviewEstimate.jsp. Her bruges for det første HTML-attributten "required" på de essentielle input-tags, for at være sikker på, at alle nødvendige informationer bliver sendt. Dernæst bruges "pattern"-attributten for at sætte regex-conditions på. Følger telefonnummeret f.eks. ikke regex "\d{8}" (otte cifre), vil kunden ved forsøget på at sende en forespørgsel øjeblikkeligt få en besked om at feltet ikke følger reglerne for hvordan et telefonnummer skal se ud, i stedet for at der bliver sendt en unødig request til serveren.

Hvis en fremtidig opdatering af programmet skulle ændre på denne HTML-kode, bruges regular expressions dog også i entitetsklassen Customer i backend. Her tjekkes på samme måde som i HTML på JSP-siden, om et parameter er null (dvs. et felt, som indeholder vigtige data, er tomt), og om alle de forskellige Strings følger de angivne regex-patterns. Følger telefonnummeret f.eks. ikke samme regex som nævnt før, så kastes en IllegalArgumentException, som fanges af SendInformationCommand, og kunden får en fejlbesked som beskrevet i afsnittet herover.

Logging

Som en del af errorhandling af et program blev det foreslået, at man kunne benytte sig af logging; en måde at kunne danne overblik over hvilke fejl, der er opstået hvor og hvornår, dvs. at kunne specificere de bugs, som opstår på en måde, så det er nemmere for teamet selv at forstå hvor problemerne er opstået. Logging filerne bliver gemt under projektets egen mappe i en .log fil. Der bliver skrevet til konsollen under både tests og kørsel af selve web-siden.

Logger-funktionen har sin egen klasse: LoggerSetup. Hvis der ikke findes en log-fil i forvejen i den designerede path, bliver der lavet en. Log-filen er i .gitignore og vil blive genereret ved start af programmet. Logger niveauet er på Level.ALL i den forstand, at alle logs uanset relevans eller type bliver logget (type med henblik på f.eks. Level.info).
Log-filens path er placeret i rapporten, siden den meget hurtigt bliver stor og for det meste ikke har relevans over for de andre programmører. Log filen ligger på nuværende tidspunkt I programmets hovedmappe og bliver genereret under clean and build.

SQL-Injection

Som en forhindring til brugere, som forsøger at udnytte programmets input fields ifm. SQL-Injections, er der blevet benyttet Prepared Statements i alle data mapper-metoder. Datamapperne er dermed sikret mod SQL-injections, og queries vil være pre-compiled og hurtigere end hvis der var blevet benyttet normale Statements. Ved brug af prepared statements betyder det, at brugeren kun har mulighed for at angive parametre til forudbestemte queries, og man undgår ændringer af selve query'ens formål og struktur. Det er dog anerkendt, at prepared-statements ikke normalt vil være tilstrækkeligt til at gøre programmet fuldstændigt sikkert. Derfor skal det understreges, at der i dette projekt ikke er taget højde for sikkerhed i udpræget grad (se afsnittet om sikkerhed længere nede).

E-mail

Efter kunden accepterer carportens detaljer efter at have specificeret, hvilke dimensioner der er påkrævet, skal kunden kunne udfylde sine personlige informationer. Der er brugt standard SMTP-transmission til Google's adresse "smtp.gmail.com" for at kunne afsende en bekræftelses-e-mail til kunden. Denne bekræftelses-e-mail bliver afsendt af administratoren, efter evaluering af ordren. Bekræftelses-e-mailen indeholder det meste af den information, som er blevet indtastet af brugeren.

E-mail-systemet udnytter en klasse kaldet EmailUtility, som ligger i .gitignore og derfor ikke kan ses af brugeren. Dette er pga. username og password, som er synligt i klassen, og at den slags information ikke bør være tilgængeligt for alle github-brugere.
Der kan nemt udskiftes EmailUtility username og passwords, så alle kan afsende e-mails, så længe afsenderen benytter sig af en Gmail adresse.

Git Ignore

Git ignore er en tekstfil, der følger med, når man bruger github, hvor meningen er, at man kan vælge filer fra, som man ikke vil ikke have pushet til github. Dette er blevet gjort på de filer, der indeholder bl.a. brugernavn og password til administrative rettigheder – dvs. DBConnector, hvis informationer kan give CRUD-rettigheder til databasen, og EmailUtility, hvis informationer giver adgang til firmaets e-mail-adresse. Hvis det var tilfældet, at DBConnector ikke var blevet taget højde for i git ignore filen, ville alle med adgang til projektet på github også have CRUD-rettigheder til databasen. Derudover ligger /target-mappen i git ignore, da denne ikke er relevant at få uploadet. Projektet skal alligevel gennemgå en Clean and Build, hver gang der laves en ændring i programmet, og .war-filen er tilgængelig via Ubuntu-servletten.

JavaScript

Der er et enkelt tilfælde hvor javascript bliver benyttet i webprojektet, selvom dette ikke har indgået i pensum for semestret. Dette bruges i CarportDetails.jsp (se linje 41) i form af to javascript-metoder (switchRoof() og switchShed()), der kaldes fra "onchange"-events på flere HTML-elementer, og én metode, der kaldes internt i switchShed().

Den primære funktionalitet i de to førstnævnte metoder går ud på, at nogle dropdown-menuer på siden, hvor man vælger sin carports specifikationer. De skal være synlige mens der er nogle der skal fjernes, afhængig af hvorvidt de to HTML checkbox-inputs for hhv. tag med rejsning og redskabsskur er slået til eller fra. Fjernes en dropdown-menu, bliver den gemt væk og nulstillet til det øverste element i dets liste (menuens overskrift). Den interne metode, updateAvailableSheds(), kaldes fra switchShed(), hvis muligheden for redskabsskur er slået til, den tager også højde for de angivne dimensioner for carporten. Med disse informationer udregner den præcis hvilke længder og bredder for redskabsskure, der bør være tilgængelige i menuerne, og hvilke der ikke skal kunne vælges ved brug af HTML-attributten "disabled".

Med denne javascript-implementation sørger man for, at man får en langt mere dynamisk webside, der tilpasser sine indbyggede muligheder automatisk, efterhånden som brugeren interagerer med den. Via det undgås der unødig brug af klodsede fejlbeskeder og unødige http requests til serveren. I brugen af programmeringssprog har forskellene mellem java og javascript navnlig været, at variable ikke er typefaste; de kan ændre type dynamisk, og en variabels type (f.eks. boolean) ikke ses direkte i koden, men blot foregår i backend systemet); og de angives med keyword'et "var". Derudover er der fat i alle elementer og deres attributter, og kan ændre på disse uden Getters og Setters, ved hjælp af det overordnede objekt for hele HTML- eller JSP-siden (document), og gennem kaldet af getElementByID() på dette.

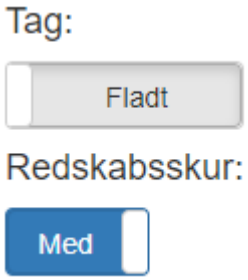
CSS

Den css der bliver brugt i dette projekt er tre forskellige cascading style sheets; "styleFrontPage.css", som bliver brugt til styling af forsiden (den har en meget unik styling i forhold til de andre sider), "ThankYoucss.css", som bliver brugt til at style den side en kunde ser, efter at have bestilt en Carport (unik , styling), og "main.css", som er den generelle css, der bliver brugt på største delen af jsp siderne.

Der er bestræbt efter, at få størstedelen af stylingen til at komme fra style-deklarationer fra de eksterne .css-filer. F.eks. er der en ".mainbody"-class på næsten alle JSP-sider, som blot lægger padding på alt undtagen navbaren, således at indholdet ikke bliver skrevet helt ude på skærmens grænser. Man kunne have skrevet denne styling-attribut på alle individuelle div-elementer på de forskellige JSP-sider, men ved brug af en class-attribut i stedet og ved at style alle disse elementer udefra, sikres der er en gensidighed blandt alle elementerne, uden at være nødt til at ændre en detalje mange forskellige steder, hver gang der skal prøves et nyt design af. Der er på samme måde valgt nogle fælles værdier af padding, border og alignment for alle tabeller på hjemmesiden ved hjælp af kategorierne "Table", "td", "th", og klassen ".paddedtable".

I få tilfælde er der dog alligevel valgt at beholde lokal HTML-styling, i stedet for ekstern CSS, f.eks. i RequestDetails.jsp, linje 96-99, der bliver brugt "display: inline-block;", til at placere inputfelterne for priser ved siden af deres overskrifter i stedet for under. I situationer som denne, har gruppen vurderet, at brugen af styling er så minimal og specifik for lige netop disse HTML-elementer, at flytningen af den ene linje til en ekstern CSS-fil blot ville gøre det mindre overskueligt nærmere end mere, og generelt ikke ville spare en eneste linje kode.

Udover den CSS der er blevet skrevet, er der også importeret Bootstrap-biblioteket til at gøre det overordnede visuelle design lidt pænere. På CarportDetails.jsp importeres derudover et ekstra Bootstrap-stylesheet, som definerer en stor mængde CSS for checkbox-inputs. Dette gør, at alle checkboxes for at slå tag med rejsning og redskabsskur til og fra, er blevet til reelle toggle-buttons i stedet, inkl. simple animationer og mulighed for at angive hvad der skal stå på knapperne, når de er hhv. slået til og fra.



Figur 8 on/off knap

Encoding

Det IDE som der er blevet brugt til opgaven, Netbeans, bruger et lettere atypisk default encoding. Når der bliver lavet en ny html / .jsp side, vil standard encoding og charset være sat til UTF-8, hvilket bl.a. burde inkludere danske tegn. I stedet bliver tegnene mødt af UTF-8's debugging tool og de danske "ÆØÅ" bogstaver bliver i stedet til "Ã¦ Ã¸ Ã¥". Som en løsning til det problem, bliver der i projektet brugt Windows-1252 i stedet for UTF-8, siden det er Netbeans's standard encoding. Dog blev det kun implementeret på sider, som er direkte afhængige af danske bogstaver, for ikke at skabe eventuelle unødvendige komplikationer.

```
<%@page contentType="text/html" pageEncoding="Windows-1252"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=Windows-1252">
```

Alternativt kunne man gå ind i Netbeans's config fil og ændre på default charsettet, men det ville have mulige katastrofale følger for tidligere projekter. Udover det ville alle andre, der skulle håndtere programmet også blive nødt til at ændre i den samme config fil.

Test

Der er blevet udviklet en række tests til de vigtigste dele af programmet, så det hurtigt kan ses, hvorvidt en ændring på programmet har gjort skade på de vigtigste deles funktionalitet. En tabel med de forskellige klasser og metoder, som er blevet testet, kan ses i tabellen nedenfor.

Class	Testede Metoder
Carport	>getLength >getWidth >getRoof >Carport (Illegal value)
Customer	>getName >Customer (Illegal value) >getAddress >getZipcode >getCity >getPhone >getEmail
Offer	>Offer (Illegal value) >getPrice >getID >getShippingCosts >getCustFirstNameThroughOffer >getCarportWidthThroughOffer
Request	>Request (Illegal value) >getComments >CustomerNameThroughRequest >getID >getHasRecievedOrder >getCarportHeigtThroughRequest
Roof	>getRaised >getID >getSlope >getType >getRoofHeight >Roof (Illegal Value, tre forskellige tests)
Shed	>getID >getLength >getWidth >getWallCoverings >Shed (Illegal value, 4 forskellige test værdier)

Dette er en kort gennemgang af de avancerede tests. Der er blevet oprettet en testdatabase til alle de klasser der tilføjer objekter, så testning af dem ikke skaber problemer. Der bliver testet følgende;

- createRequest er en funktion der tilføjer et objekt til en database. Den bliver testet ved, at den tilføjer et Request objekt til testdatabasen. En try-catch er sat op til at fange exceptions, som så registrer testen som fejlet.
- readAllRequest er en funktion, som læser alle Requests i en database. Den bliver testet ved brug af en "assertTrue" metode fra JUnit librariet, der tester objekternes størrelse
- createOffer fungerer som "createRequest" med et Offer objekt i stedet for et Request objekt. Den bliver testet på samme måde, ved brug af en try-catch som registrer det som en fejl hvis der bliver kastet en exception.
- addUser tilføjer en user til databasen. Den testes ved at en try-catch tjekke om det bliver kastet exceptions, når en user er tilføjet til testdatabasen. Hvis der gør, kører den et "fail statement".
- getUser henter en bruger fra databasen via et brugernavns variabel det sker ved, at databasen bliver tjekket for, at brugernavnet er det samme som variablen. Testen tjekker, hvorvidt de to brugeres brugernavne er ens, og om deres passwords er identiske med en "assertEquals" metode.

Class	Testede Metoder
DataMapper-Customer	>createRequest
DataMapper-Employee	>readAllRequest >createOffer
DataMapper-User	>addUser >getUser
Fittingsand-ScrewsCalc	>calculateParts (Negative values) >calculateParts (If Empty) >calculateParts (Correct list size)
RoofCalc	>calculateParts (Negative values) >calculateParts (Null values) >calculateParts (Not empty) >calculateParts (Correct list size) >calculateParts (Correct pricing) >calculateParts (Correct amount of tiles)
WoodCalc	>calculateParts (Negative values) >calculateParts (Not empty) >calculateParts (Correct list size)

Et problem opstod med code coverage biblioteket "JaCoCo" hvilket gjorde, at det ikke var muligt at se dækningsgraden for testene.

Status på implementering

CRUD

Indførsel af alle CRUD datamappers (CREATE, READ, UPDATE, DELETE), var en opgave, som blev overvejet. Den blev dog nedprioriteret og der blev kun lavet de datamappers, som var direkte nødvendige for gennemførslen af produktet. Det er anerkendt, at det er god skik at benytte sig af CRUD eller lign. for alle tabeller i databasen, så det er nemt at indføre nye datahentnings- eller manipulationsfunktioner til programmet.

Optimering af database

Som nævnt tidligere i rapporten, bør kolonnerne Zipcode og City tages ud af Customer-tabellen og lægges over i en tabel for sig selv. Customer kan derefter forbindes til dem via en fremmednøgle. Dette ville fjerne kolonnernes transitive afhængighed, og få tabellerne på 3. normalform.

Endnu en ting der er værd at kommentere på, er den måde som gruppen har valgt at håndtere carportens, tagets og redskabsrummets relationer. Som det er lavet nu, skabes der en ny Carport, Roof og Shed object for hver eneste Request. Man kunne dog overveje, at ligesom Customers med samme Email genbruges i databasen, når den samme kunde sender flere forespørgsler, så man f.eks. kunne genbruge Roof med samme kombination af type og hældning. Dette ville give en fordel, da man så ville kunne genbruge dataet, i stedet for at der kommer duplikeret data i databasen hver gang der bliver bestilt en ny carport. Udover det ville man undgå yderligere redundans. I et projekt af denne størrelse ville det ikke give en stor forskel, men hvis det var en offentlig hjemmeside, med virkelige kunder og transaktioner, ville det være en fordel og mere overskueligt.

Dog ville denne fremgangsmåde kræve mere arbejde programmeringsmæssigt, siden man ville blive nødt til at tjekke i databasen, hvorvidt der allerede eksisterer en Carport, et tag eller et skur med samme kombination af data, når man skal lave en Request. Denne arbejdsbyrde kunne potentielt skade websidens performance, i stedet or at gøre gavn mht. redundans i databasen.

Sikkerhed

Sikkerheden på denne website har ikke været i fokus mht. andet end brugen af gitignore, ifm. fjernelse af direkte adgang til E-mailen og databaser. F.eks. er det ikke en optimal løsning, rent sikkerhedsmæssigt, at administratorens brugerinformation gemmes både i SQL-databasen, og (endnu værre) direkte på sessionen i form af entitetsklassen User. Kun brugernavnet burde i princippet gemmes på session.

I stedet for at gemme admins kodeord direkte i databasen, kunne man gøre brug af en hashfunktion til ensidig kryptering af værdien. Herved ville man have gemt krypteringen af kodeordet, i stedet for kodeordet selv og dermed gøre databasen mere sikker. Man ville på samme måde, ved validering af passwords, kunne sammenligne de krypterede passwords i stedet for de oprindelige versioner.

Derudover bør man på en website som denne, hvor administratorrettigheder giver så meget magt ift. priser og carportsidens valgmuligheder, anvende en sikker HTTPS-protokol. Dette er ikke tilfældet i projektets nuværende stadie.

Logging

Logging skrev kun til log under tests. Der blev kun skrevet til konsollen, når der opstod logging under projektets kørsel og der burde også blive printet til .log filen. Der opstod også problemer i pathing til logging mappen. Det er meget muligt, at de to problemer var relaterede.

Forbedringer af tests

Der er nogle tests, der kunne blive forbedret ved at tilføje extra funktionalitet til datamapper klasser. Tilføjelse af funktionalitet er blevet gjort før, med tilføjelsen af muligheden til at skifte mellem testdatabasen og normale database der bliver brugt i programmet. Det blev gjort det muligt at teste datamappersrne siden de har funktionalitet forbundet til database brug. Der er nogle dele af de forskellige datamappers der kunne testes, hvis der blev lavet funktionalitet, der trækker specifik information ud af databasen (request_id) eller en funktion der bruger "DROP" på test tabellerne, hvor den derefter bruger "CREATE" til at lave dem igen. Grunden til den funktionalitet ikke blev tilføjet var, at det ville tage for meget af udviklingstiden, som ville kunne bruges bedre andre steder. Udover det, anså Product Owner det ikke for at være vigtigt nok.

Code Coverage

Som et løsningsforslag på opgaven, kunne man vælge at importere et library der kaldes "JaCoCO" og bruge det til en funktion i NetBeans, der hedder Code Coverage. Funktionen beskriver, som navnet antyder, at man kan få et bedre overblik over, hvilke dele af koden der er dækket af tests i procent. Gruppen satte noget tid af til det, men havde problemer med at få det til at virke. Det blev derfor nedprioriteret

PDF-version af Carport

Det kunne være brugbart, at kunden kunne downloade carportens specifikationer og skitsetegninger som pdf, via. en knap på siden. En lignende pdf kunne også sendes med i E-mailen ved administratorens bekræftelse og afsending af et tilbud. Denne funktionalitet var der et ønske om at få lavet, men det blev nedprioriteret.

Optimering af E-mail

Systemet for at kunne afsende E-mails er meget simpelt og kunne optimeres:

- Andre eksterne (og mere komplekse) libraries. På nuværende tidspunkt er den bl.a. afhængigt af et gammelt sun SMTP transport library. Det allerede importerede javax.mail library indeholder selv en transport metode, men ville kræve omskrivning.
- Benyttelse at Session.getInstance() tager et authenticator object som attribut, som kan sikre programmet via username/password protection på sessions niveau i stedet for predefineret i E-mailUtility Handling. Mulighed for at sende e-mails alt efter hvem der er logget ind.
- Brug af et E-mail Relay, som kan tage e-mails fra andre end Gmail accounts. Man ville blive nødt til at finde et andet open SMTP relay, som ikke koster for meget.
- Opsætning af et fake SMTP server, for bedre at kunne teste E-mail systemet. Det ville kræve opsætning af et nyt UI. Det virker i ubelejligt at blive nødt til at bruge temporære E-mail-konti eller bruge en af sine egne for at kunne teste systemet.
- På nuværende tidspunkt, laver E-mailHandler klassen alt der har noget at gøre med E-mailen, i den forstand, at den henter information fra sessionen, konverterer den og sender dem ud i tekstformat. Det ville være bedst, hvis dokumentet blev autogenereret ud fra E-mailHandler klassen selv i et normalt tekstdokument. Der ville være mere plads til billeder og layout af beskeden.

GitIgnore udvidelse af DBConnector

Siden der ikke bør være direkte adgang til database login-informationerne, blev der besluttet at indsætte den ind under gitignore. Dog kunne man argumentere for at benytte sig af samme teknik, som var brugt i E-mailHandleren: At lave en separat klasse, udelukkende bestående af username og password til databasen. På den måde, ville det være muligt for brugeren at få netop den samme DBConnector indstilling, som er brugt i dette program, bare uden databasens login information. Derudover, kunne et MySQL script bruges til opsætning af samme database, som er blevet brugt i dette projekt.

Slet Request

I 4. sprints retrospektiv var der sankket om en feature, hvor man som administrator skulle kunne slette en forespørgsel fra ListRequests.jsp. Ud for hver række i listen af Requests, skulle der blot være en knap, som ved aktivering ville slette den relevante Request fra databasen. Det ville indebære tilhørende carport-, roof-, shed- og offer-information i databasen, og derefter genindlæse siden, så man ser listen uden den slettede forespørgsel. Dette ville kræve en ny Command, en ny data mapper-metode, og opdateret arkitektur til at understøtte funktionaliteten op igennem alle interfaces og facader. Denne feature er desværre ikke blevet gennemført.

Config-fil

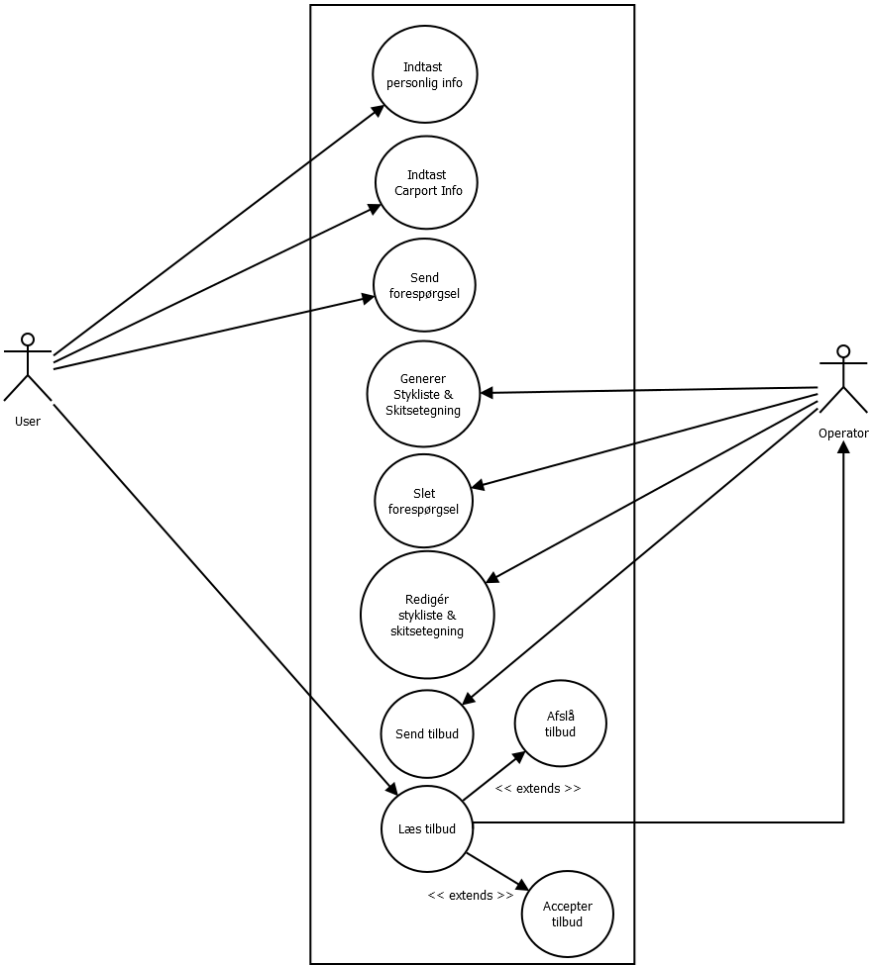
Mange algoritmer kan omskrives til at bruge konstanter fra eksterne datasources, der ville gøre det nemmere for brugeren at ændre på programmets egenskaber. Dette kunne f.eks. være priser, beskrivelser eller navne på individuelle Parts; eller det kunne være konstante værdier til brug i algoritmerne, fx “startX” eller “maxDistanceBetweenPoles”. Hvis dette kunne sættes op som en konfigurationsfil, der kunne læses via en Read-metode i datalaget, så ville det være nemmere tilgængeligt for brugeren at ændre på programmets egenskaber. Man ville kunne ændre på det, uden at have et IDE eller en mere avanceret texteditor åben.

Flere muligheder for dele

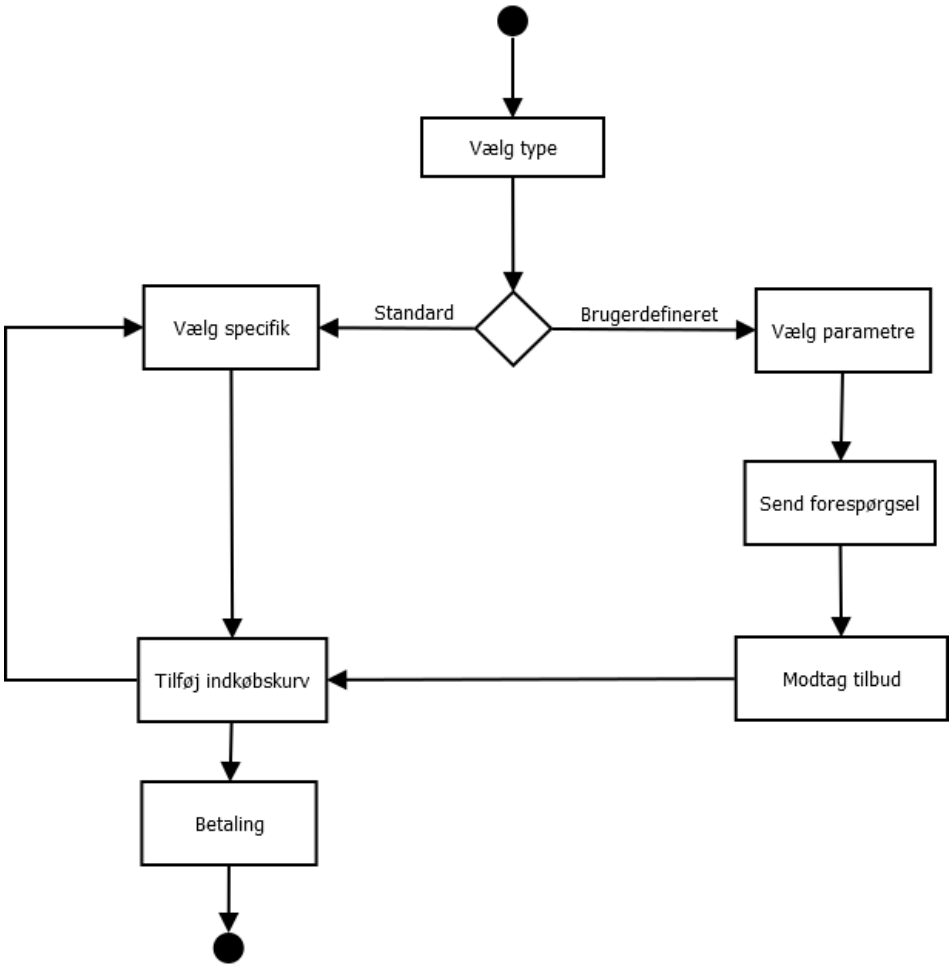
Plastmo, firmaet som står for produktionen af tagplader, har flere forskellige tagplader og bør have nemmere adgang til tilføjelse af nye. Dvs. At programmet f.eks. kunne omskrives til, at man ville kunne lave flere tagplader ved at duplikere elementer i en config fil. På den måde, kan man lettere indsætte nye plader, men nye dimensioner. På nuværende tidspunkt, er det besværligt at tilføje nye dele, pga. metodernes relationer til hinanden.

Dette er en mulighed, som man kan overveje igennem hele programmet. Gennemførslen af denne idé er dog kompleks og ville tage langt tid at indføre.

Bilag



Figur 9 Use Case Diagram



Figur 10 Aktivitetsdiagram

Objective = Profit

SWOT	Helpful *…to achieving the objective*	Harmful *…to achieving the objective*
Internal Origin *(Attributes of the organization)*	***STRENGTHS***	***WEAKNESSES***
	-Kvalitet -Etableret Brand -Stærk virksomheds identitet	-Dyrt -Oplærings periode -Ikke fleksibel
External Origin *(Attributes of the environment)*	***OPPORTUNITIES***	***THREATS***
	-Tilbyde håndværker hjælp -Arbejde med håndværks-Uddannelser -General stigning i "gdp per capita" i Danmark	-Nye billigere alternativ -Folk har mindre lyst til at bygge selv -Mulig stigning af import priser.

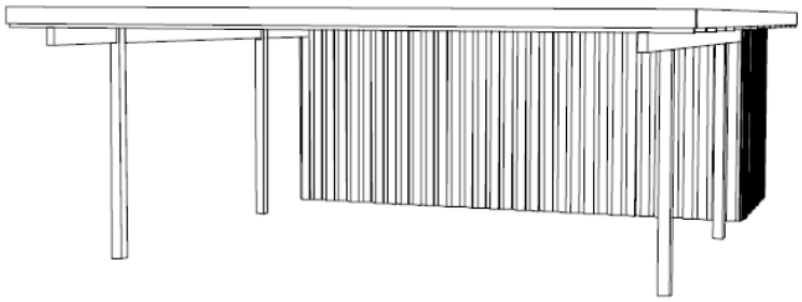
Figur 11 SWOT analyse profit

Objective = Functional computer program

SWOT	Helpful *…to achieving the objective*	Harmful *…to achieving the objective*
Internal Origin *(Attributes of the organization)*	***STRENGTHS***	***WEAKNESSES***
	-Hurtigere stykliste -Generelt mere brugervenlighed -Vare længe før at der skal laves nyt	-For kompliceret kode -Ikke effektiviserede stykker kode -En glitch kan ødelægge meget
External Origin *(Attributes of the environment)*	***OPPORTUNITIES***	***THREATS***
	-Give kunden mere information via styk-liste -Ny-moderne systemer kan tiltrække folk som søger nyt job	-Teknologisk inkompetence -Nye svagheder i systemet kan opstå, hvilket kan udnyttes af hackere

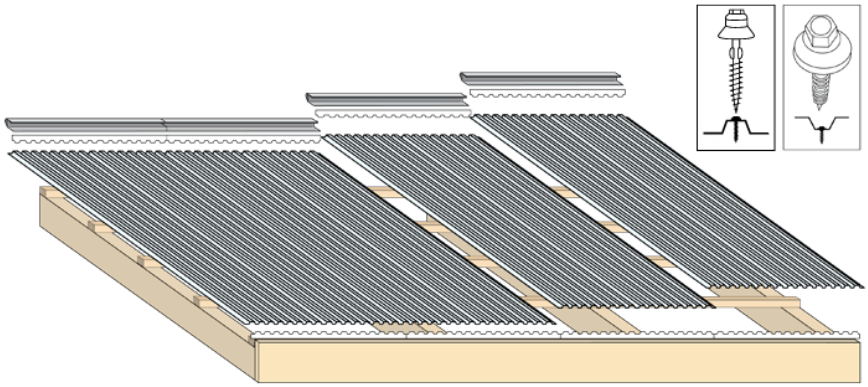
Figur 12 SWOT analyse program

CP01 DUR
******2016******
6,0 X 7,8 MTR.



Plastmo Ecolite blåtonet	600	6	Stk	tagplader monteres på spær
Plastmo Ecolite blåtonet	360	6	Stk	tagplader monteres på spær

Dine valg: bredde 6 m. / længde 7.8 m.



Antal		Varenavn	DB
14		EcoLite 109x420 cm Blåtonet	5191095

Figur 13 Egentlige roof forhold