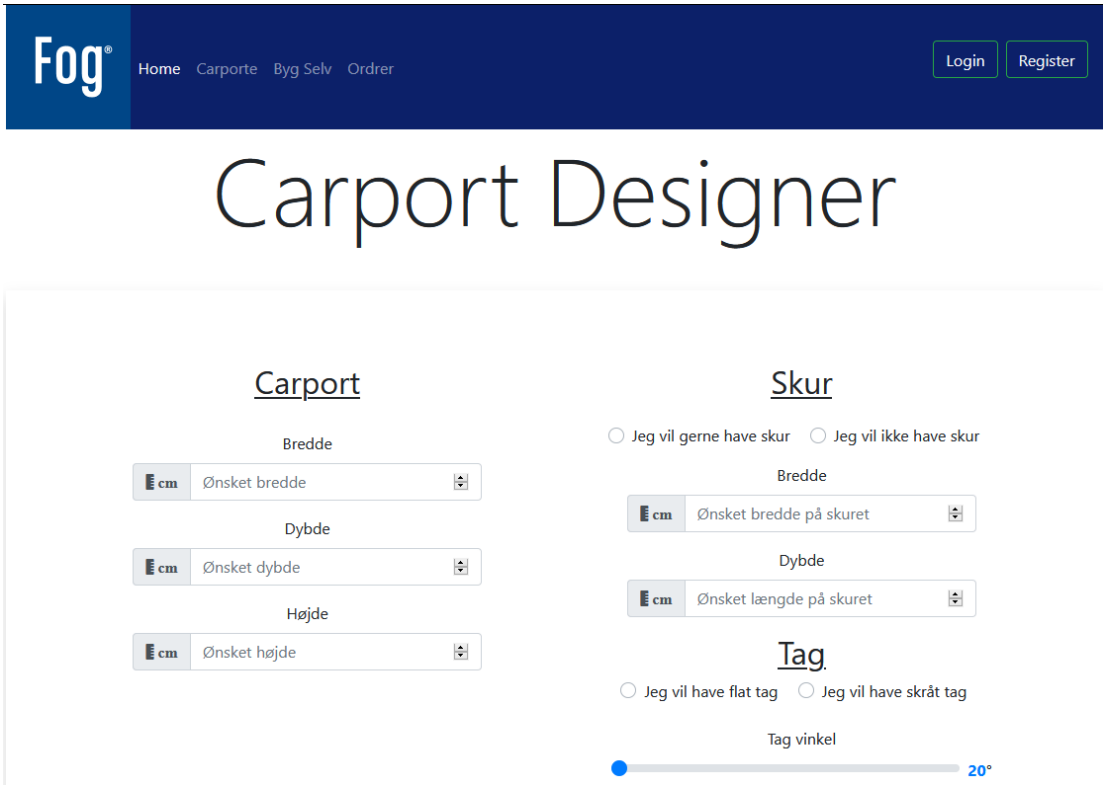
Fog Trælast & Byggecenter: Carport Designer

Datamatikeruddannelsen ved Cph Business Academy Lyngby
2. Semester Projekt



Klasse E, Gruppe π (pi / 3.14)

Navn	**Email**	**Github**
Simon Jørgensen	cph-sj414@cphbusiness.dk	Rodedrengen
Frederik Keis Dinsen	cph-fd77@cphbusiness.dk	Fdinsen
Oliver Vang	cph-ov111@cphbusiness.dk	Olvang

Link til GitHub Repository: https://github.com/olvang/Fogbyggemarked

Link til Java Docs: https://olvang.github.io/Fogbyggemarked/index.html

Link til Demo Video https://drive.google.com/open?id=1QXfhEzXxtd6l4Jyg-TSt2XOPWAs0QsqW

Link til kørende projekt (Digital Ocean): http://droplet.fdinsen.com/Fogbyggemarked-1.0/

Link til Taiga Product Backlog: https://tree.taiga.io/project/olvang-fog-byggemarked/backlog

Begyndt: 15. April, 2020
Afsluttet: 29. Maj, 2020

Indholdsfortegnelse

Indledning 4
Baggrund 4
Teknologivalg 4
Krav 5
User Stories 5
Overordnet beskrivelse af virksomheden 6
Arbejdsgange der skal IT-støttes 6
ER diagram 7
Orders 7
Sheds 8
Bill 8
Category 8
Materials_to_category 8
Materials 8
Variant 9
Normalisering 9
Navigationsdiagram 10
Særlige forhold 11
Udvalgte kodeeksempler 12
Component systemet 12
Front-end håndtering af validering 14
BillCalculator 16
Status på implementation 17
Test 18
Component: 18
DataMappers: 19
BillGenerator: 19
Process 21
Arbejdsprocessen faktuelt 21
1. Sprint 21
2. Sprint 21
3. Sprint 22
4. Sprint 22

5. Overskydende Stories 23
Arbejdsprocessen reflekteret 25
GitHub 26
Taiga 26
Konklusion 26
Bilag 28

Indledning

Opgaven er lavet ud fra de emner vi har gennemgået i løbet af 2. semester, samt ved brug af JavaScript. Vi er samtidigt blevet introduceret til nye værktøjer og arbejdsmetoder, som både var spændende og anderledes at arbejde med.

Som udviklingsmetode brugte vi for første gang Scrum, som har været meget anderledes end hvad vi tidligere har prøvet. Derudover har vi taget Git Branches i brug, hvilket har ladet os arbejde mere frit, uden bekymring om konflikter.

Resultatet af overstående er et projekt, der trods stadig at mangle noget af den planlagte funktionalitet, et vi alle er glade for at have bygget og er stolte over at kunne vise frem.

Baggrund

Johannes Fog Trælast & Byggecenter er en kæde af butikker som fører materialer og værktøj til byggeprojekter samt rådgivning dertil. En af disse byggeprojekter de tilbyder service til, er carporte.

Opgaven er lavet ud fra at Fog ønsker et nyt og bedre it-system for kunder der ønsker aflægge en forespørgsel på en Carport.

Fog ønsker at det nye system skulle erstatte et gammelt it-system, ved både at have flere af de samme funktioner, samt tilføjelse af flere nye.

Hovedfunktionaliteten af systemet består i at en kunde skal kunne lave en forespørgsel på en carport med eller uden skur samt fladt tag eller skråt tag. Herefter skal deres forespørgsel gemmes, således at Fogs medarbejdere kan gå ind og se de individuelle ordrer. Yderligere skal system kunne beregne en stykliste af de materialer som skal bruges til at bygge den givne carport, samt kunne vise en tegning af den som passer til de mål brugeren har givet.

Teknologivalg

Vi gjorde brug af en MySQL (V. 8.0.15) til opbevaring af forespørgsler samt de forskellige materialer tilgængelige. Den forbindes til Java applikationen (V. 1.8.0_251) ved hjælp af JDBC (Java Database Connectiviy) (V. 8.0.19). Programmet er skrevet i IntelliJ IDEA Ultimate (V. 2019.3), og kørt på localhost med en Tomcat (V. 8.5.511) server. Unit-tests er skrevet med Junit (V. 4.13) og Hamcrest (V. 1.3)

Til at forbinde web-siden til java-systemet gøres der brug af JSP og JSTL (V.1.2). Selve projektet er opbygget med Maven (V.4.0.0). Derudover er projektet opbygget omkring et Command-pattern skrevet af Kasper, en lærer på CPHBusiness. Tegningerne af carporten bliver genereret som et SVG-element.

Siden er bygget med en række Bootstrap (V. 4.4.1) elementer, og Datatables (V. 1.10.20) til at lave en tabel-oversigt over ordrer samt at vise styklisten. Derudover er der brugt javascript til at gøre siden mere responsiv, samt selvfølgelig HTML og CSS.

Alle UML-diagrammer er tegnet ved hjælp af PlantUML (V. 1.2020.10), og mockups er bygget i Adobe XD (V. 29.0.32). Vores docs er genereret med JavaDoc.

Krav

Kravene blev beskrevet af Johannes Fog i en video, der beskriver deres nuværende system samt hvilke forbedringer/nye funktioner de ønsker i et nyt system.
Udover det, er det Product Owner (en lærer) der også har været med til at definere kravene.

Ud fra video kunne vi finde frem til følgende krav:

- En nutidig webbaseret løsning
- Bestilling af carport er integreret med generering af stykliste og tegning
- Kunden skal kunne vælge alle dimensioner selv
- Kunden skal kunne vælge alle materialer selv. Tag, beklædning og evt. gulv
- Kunden skal kunne vælge hvilken type tag de ønsker, fladt eller rejst
- Kunden skal kunne vælge skur til eller fra
- Fog vil gerne kunne rette materialer og priser
- Fog vil gerne kunne tilføje nye materialer
- Sikring af deres styk liste så kunden først kan se den når de har betalt
- Stadig menneskelig kontakt mellem kunden og sælger
- Indkøbsprisen skal udregnes og et forslag til salgsprisen skal laves
- Salgsprisen skal manuelt kunne rettes
- Varianter af samme materiale skal kunne håndteres

User Stories

Kravene fra overstående liste, har vi omformuleret til User Stories, yderligere har vi også selv kommet på User Stories som vi mener kunne være gode at have i systemet.

Efter vi har skrevet alle User Stories færdige, har vi sammen med Product Owner gennemgået dem og blevet enige om hvorvidt der var nogle der skulle ændres / fjernes / tilføjes.
Herefter udvalgte Product Owner de højst prioriteret User Stories til første sprint.
Til at starte med var fokusset på at få et grundlæggende system op at køre, hvor der til sidst blev valgt nogle User Stories der ikke var ligeså nødvendige, men mere nogle funktioner der var 'Nice to have'.

Et eksempel på en User Story, er denne fra det første sprint:

Customer - Send selections to Fog salesperson
As a customer, I want to be able to send the selections I make to a salesperson at Fog, so I can make an order.

Acceptance requirements:
When I can send my order to a salesperson, and the order appears in the database.

Efter en User Story er blevet valgt af Product Owner, har vi snakket om hvilke opgaver der rent faktisk skal til, for at få implementeret den User Story. De vil så blive listet som 'SubTasks' under den User Story.

En komplet liste over alle User Stories, kan findes i afsnittet om Arbejdsprocessen faktuelt eller på Taiga.

Overordnet beskrivelse af virksomheden

Fog er både et byggemarked, men også et rådgivningsfirma. Når brugeren kommer til dem med et ønske om at købe en carport, er det i både Fogs og brugerens bedste interesse at komme i kontakt med en sælger, i stedet for bare at få en stykliste og tegning spyttet ud. Derfor ønsker Fog at når en bruger sender sin forespørgsel, bliver den sendt til en Fog medarbejder som så kan kontakte brugeren.

Fog vil gerne have at systemet skal facilitere denne kontakt mellem brugeren og sælgeren. En sælger skal kunne kontakte brugeren og komme med forslag og input til deres carport. Derfor skal sælgeren kunne ændre i brugerens ordre, i tilfælde af at et ønske de har er usmart. Derudover ønsker Fog at sælgeren skal kunne se en anslået pris, udregnet ved hjælp af indkøbsprisen af materialerne med et kommissionsgebyr lagt oven i. Den pris skal sælgeren så kunne lave om til et tilbud, som sendes til kunden. Hvis kunden accepterer tilbuddet og betaler for bestillingen, skal de første der få udleveret styklisten.

Arbejdsgange der skal IT-støttes

Fog bruger på nuværende tidspunkt to separate IT-systemer til at håndtere bestillinger af carporte: Et som brugeren bruger til at afgive sine ønsker om carporten, og et som en Fog-salgsperson bruger til rent faktisk at lave ordren. I dette system får en Fog-salgsperson en besked når en forespørgsel fra en kunde kommer ind. Derfra er det op til salgspersonen at: 1. Tjekke om de indtastede værdier giver mening (fx skuret er større end carporten.) 2. Kontakte den mulige kunde. 3. Manuelt at indtaste de informationer som kunden indtastede i IT-system 1, i IT-system 2, hvilket på det tidspunkt gør forespørgslen til en ordre.

Alt i alt er Fog glade for det nuværende system, ikke fordi det er effektivt (for det er det ikke), men fordi det faciliterer og kræver en kontakt mellem en salgsperson og kunden. Det betyder at ethvert system som skal udskifte det nuværende skal ikke bare være mere effektivt, men også bibeholde den kontakt som det nuværende system skaber.

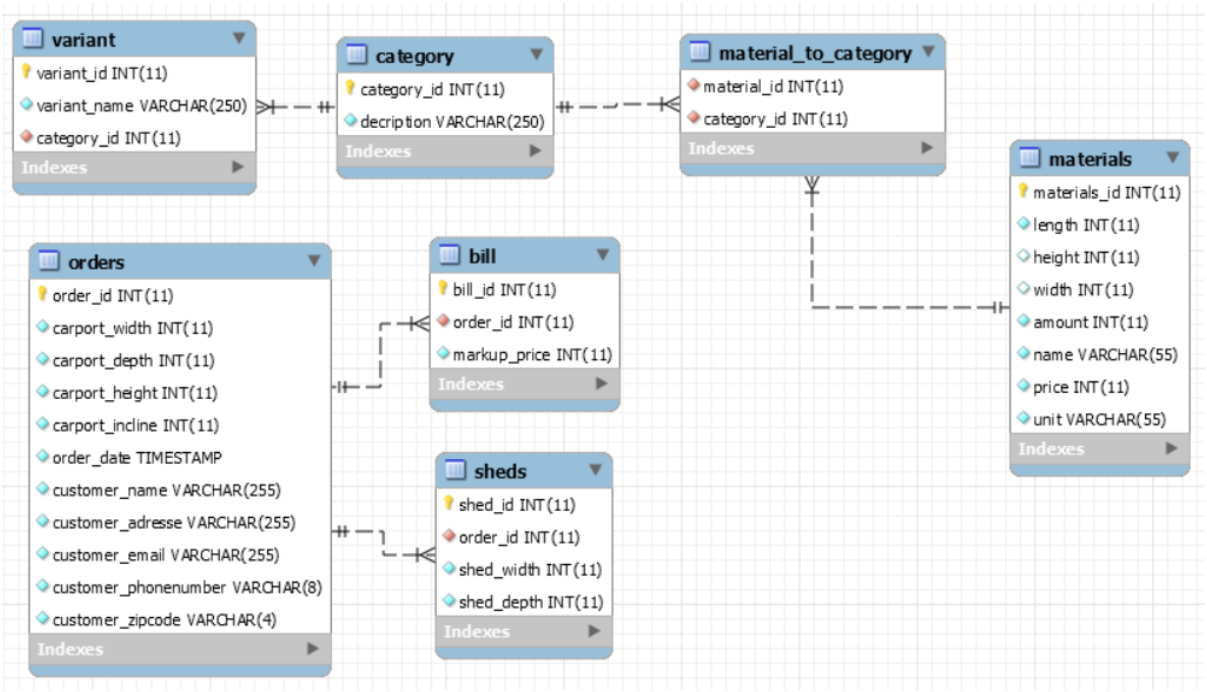
Det kan gøres ved at gøre det af med salgspersonens task 1 og 3, samt indbygge kommunikationen i designet af systemet.

1. Lad det være op til systemet at tjekke om carportens dimensioner giver mening.
2. Gør det af med de adskilte systemer. Lad kunden indtaste en forespørgsel, som først bliver til en ordre når Fog har været i kontakt.
3. Først da sendes styklisten og tegningen til kunden.

Disse planer gør det af med unødigt busy-work som spilder en masse af både Fogs og kundens tid, samtidig med at det kræver at der er kommunikation mellem salgspersonen og kunden.

ER diagram

Det fulde:



Figur 1. ER-Diagram over databasen

Orders

Er tabellen over selve ordren og den data der hører med der. Alle længder bliver gemt i cm for at sikre den samme enhed alle steder. Der bruges en timestamp til at gemme tidspunktet ordren er blevet oprettet.

CARPORT_INCLINE bruges til at afgøre om taget på ordren er rejst eller ej. Hvis **CARPORT_INCLINE** er 0 vil taget være fladt. Dette er i stedet for at bruge en boolean værdi også have incline stående for sig selv.

Alt information om kunden er også på ordren. Der er flere overvejelser om dette. Et, en kunde vil sjældent, tænker vi, bestille to carporte til samme adresse. To, hvis det var sat ude i sin egen tabel, selvom det ville give et mere sigende navn, ville det også øge kompleksiteten af hele databasen og derved resten af programmet.

Hvis vi havde nået login-systemet, så en bruger også havde kunne følge sin ordre, ville det helt klart være nødvendigt at flytte kunden ud af **ORDERS** tabellen.

ORDERS har en 1-1 relation til **SHEDS** da der kun kan tilknyttes et skur til en ordre. Samtidig med at det ikke nødvendigvis er alle ordre der har et tilknyttet skur. Så hvis **SHEDS** var en del af **ORDERS** ville det ske at null kom til at stå flere gange i den tabel.

ORDERS har desuden også en 1-1 relation til **BILL**. Dette skyldes at der er forskel på en ordre og en regning. Selvom regningen selvfølgelig stadig hører sammen med en ordre.

Sheds

Hvis orden har et tilhørende skur, findes der i **SHEDS** en række med et ORDERID der passer til ordren. Alle længder bliver gemt i cm for at sikre den samme enhed alle steder.

SHEDS er sat op med en foreign key constraint til ORDER_ID i **ORDERS**. Da der ikke kan eksistere et skur uden en tilhørende ordre. Desuden er *on delete cascade* valgt til den constraint. Dette betyder hvis en ordre bliver slettet vil SQL automatisk også slette alle rækker i **SHEDS** hvor foreign key'en matcher.

Bill

Til en hver række i *orders* findes der er en række i **BILL**. Bill består pt kun af MARKUP_PRICE og ORDER_ID som de eneste attributter. MARKUP_PRICE er indkøbsprisen for Fogs Byggemarked for hele ordren. Den er gemt i øre for at mindste risikoen for konversions fejl. Denne er også en 1-1 relation til **ORDERS**, men da der er så stor på hvilke data man har valgte vi at lave to tabeller.

BILL er derudover sat op med foreign key constraint til ORDER_ID i *orders.* Da der ikke kan eksistere en **BILL** uden en ordre og ordren altid vil blive lavet først. Der er også *on delete cascade* på denne constraint.

Category

Vi kunne se ud fra de PDF'er vi havde fået (PDF - fladt tag , PDF – rejst tag) et materiale kunne optræde brugt i og som flere forskellige ting. Fx kunne et bræt der var 25x225 godt både være brugt som oversternsbræt og understernsbræt. For ikke at have samme materiale stående mange i en tabel med forskellige beskrivelser lavede vi en **CATEGORY** tabel. Denne er så fyldt med alle kategorier der skal bruge og et materiale kan godt optræde i flere af dem. Dette vil blive uddybet i *Udvalgte kodeeksempler*.

Materials_to_category

MATERIALS_TO_CATEGORY er den oversættelses tabel der skal bruges for at finde ud af hvilke kategorier de forskellige materialer hører til i. Her kan samme MATERIAL_ID godt optræde flere gange. Det samme er sandt for CATEGORY_ID.

Den primære nøgle er sammensætning af de koloner. Dette er gjort fordi tabellen kun indeholder de to koloner og sammensætningen altid vil være unik. Desuden er den sat op med to *foreign key constraints.* En til MATERIAL_ID i **MATERIALS** og en til CATEGORY_ID i **CATEGORY**. Dette er gjort fordi der ikke skal kunne eksistere et link hvis der ikke også er både en dertilhørende kategori og et materiale.

Materials

MATERIALS beskriver alle de materialer bruges af systemet. Længder, højder og bredder er gemt i mm. I modsætning til **ORDERS** da højder og bredder er på materialer altid er opgivet i mm. Og samme enhed er at foretrække.

AMOUNT og UNIT hører til dels sammen. Da AMOUNT beskriver hvor mange af materialet der er i en UNIT. Fx: plasto bundskruer kommer i pakker af 200stk. Mens et bræt der er 360cm langt kommer som enkelt bræt.

materials_id	length	height	width	amount	name	price	unit
16	0	0	0	200	plastmo bundskruer 200 stk.	81	Pakke
1	3600	25	200	1	trykimp. Brædt	20	Stk

Figur 2. Uddrag fra Materials tabel

Kolonnen PRICE er pt. Fyldt med test data da vi ikke kender de reelle priser for nogen af tingene.

Variant

Der var et ønske om at kunne håndtere forskellige varianter af samme produkt fra Fog. Derfor lavede vi **VARIANT**. I systemet nuværende version bliver den ikke brugt. Det var dog tanken at den skulle indeholde kosmetiske varianter af de materialer vi allerede havde. Det kunne fx have været hvidmalede brædder eller brædder af en anden træsort.

Variant er sat op med *foreign key constraint* til CATEGORY_ID i *category*. Da der ikke kan eksistere en variant hvis der ikke er en tilhørende kategori.

Normalisering

Når det gælder den indledende opsætning af den database, er det vigtigt at man holder fokus på at den bliver normaliseret. Normalisering er en teknik til at holde en database effektiv og konsistent, blandt andet ved at udrydde redundans. I praksis udføres normalisering ved at følge en række af principper kaldet normalformer. I dette systems tilfælde har vi særligt forsøgt at følge de første 3, dog kan det i nogle tilfælde diskuteres om det lykkedes:

1. Normalform
    a. Hver tabel skal have en entydig nøgle til at identificere hver enkel række i tabellen.
    b. De enkelte felter må kun indeholde én værdi.
    c. Databasen må ikke have kolonner der gentages.

Til 1.a må den entydige nøgle gerne være opbygget af flere felter tilsammen, så længe kombinationen af de to ikke kan gentages. Et eksempel på dette er vores **MATERIALS_TO_CATEGORY** oversættelsestabel. Den indeholder ikke en M_TO_C_ID kolonne, da kombinationen af MATERIAL_ID og CATEGORY_ID er unik, selvom hver enkel af dem godt kan, og kommer til at opstå flere gange.

1.b og 1.c er begge vigtige når databasens størrelse udvider sig. Har man et enkelt Excel ark med 5 ordrer er det ikke et problem hvis man lægger hele adressen (Vej, postnr., by) i samme felt, da man ved et hurtigt overblik kan se det hele. Men så snart du når op i bare mellemstore datamængder bliver det lige pludselig stærkt begrænsende. Ved at sprede dem ud kan man lige pludselig søge på de enkelte felter og finde alle fra samme by. Og gentager man kolonner betyder det at man har flere områder samlet i en tabel. I det tilfælde ville det være bedre at flytte den kolonne som blev gentaget ud i sin egen tabel, og forbinde med et id.

2. Normalform
    a. Alle krav for første normalform skal opfyldes.
    b. Ingen attributter, som ikke selv er en del af nøglen, må afhænge/være en del af kun en del af nøglen.

2.b er ved første øjekast lidt svær at forstå. Det den refererer til er at når en nøgle er bestemt ved en kombination af 2 felter (se 1.a), må de resterende felter i tabellen ikke være relateret til *kun en* af de felter som udgør nøglen. Et eksempel på dette ville være hvis vi i stedet for at have beskrivelsen af

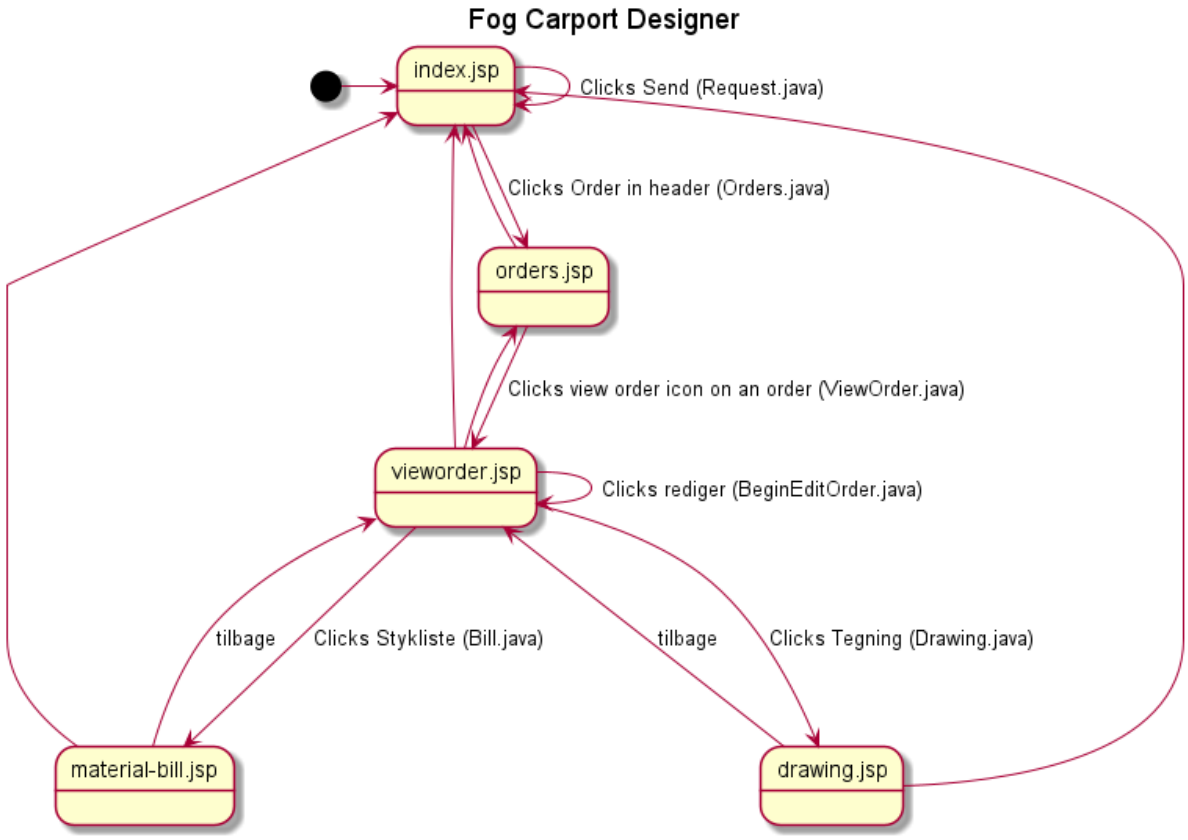
en kategori liggende på CATEGORY, havde den på MATERIALS_TO_CATEGORY, så ville beskrivelsen ikke have nogen direkte sammenhæng med materialet, men kun kategorien. Tabellen ville da blande data om to forskellige områder, og ville derfor bryde anden normalform.

3. Normalform
    a. Alle krav til anden normalform skal opfyldes.
    b. Ingen attributter må afhænge af andre attributter, som ikke selv er nøgler.

3.b beskriver at vi i en tabel ikke må have data som relaterer til andre kolonner som ikke er nøgler. Hvis man har det, betyder det at man har blandet to områder sammen. Denne normalform kan diskuteres hvorvidt vi følger. For på den ene side har vi i ORDERS tabellen både informationer om ordren, men også omkring kunden. For at følge 3. normalform fuldstændig, ville den skulle være delt op i en ORDERS og en CUSTOMERS tabel. Som beskrevet under *ER diagram – Orders* afsnittet var det en bevidst beslutning vi tog, da vi endnu ikke har nogen funktionalitet til at en kunde kan sende flere forespørgsler. Det ville da tilføje unødig kompleksitet til SQL-sætningerne hvis de også skulle til at joine med users-tabellen.

Det kan dog argumenteres at det havde været smartere at følge 3. normalform fra starten, da den alligevel skal følges så snart et bruger-system bliver implementeret. Og er Fog begyndt at bruge systemet inden da, skal vi som udviklerne til at adskille den brugerdata ad mens vi opbygger den nye tabel. Dette er præcis den hovedpine som det at følge normalformerne hjælper en med at undgå. Designer man databasen normaliseret fra starten, undgår man er stort pillearbejde når systemet skal om-designes senere.

Navigationsdiagram



Figur 3. Navigationsdiagram over systemet. Fuld størrelse kan findes på GitHub under: other -> diagrams -> NavigationFull.png

Brugeren starter på *index.jsp*. Alle siderne har en fælles header, hvorfra *index.jsp* (Home) og *orders.jsp* (Ordrer) er tilgængelige. Klikker man på Ordrer linket kommer man til *orders.jsp*, som med det samme henter alle ordrer ned (*Orders.java*, kaldt direkte fra siden).

Man kan så klikke sig ind på en individuel ordre ved at klikke på det lille øje-ikon ud for en af dem. Det sender en til *vieworder.jsp*, som med det samme henter ordren ned (*ViewOrder.java*, kaldt direkte fra siden).

Herfra er der flere muligheder:

1. Man kan klikke rediger, som så gennem *BeginEditOrder.java* opdaterer siden og udskifter informationsfelterne med tekstfelter. Derfra kan man klikke gem, og det opdaterer så siden igen med de nye informationer.
2. Man kan klikke Stykliste, hvilket sender en til *material-bill.jsp*, som ud fra de givne dimensioner udregner styklisten på stedet (*Bill.java*, kaldt gennem *FrontController*), og viser den.
3. Man kan klikke Tegning, som sender en til *drawing.jsp*, hvor *Drawing.java* (Kaldt gennem *FrontController*) genererer tegningen ud fra ordrens givne dimensioner.

Alle brugere har adgang til alle sider på nuværende tidspunkt.

Særlige forhold

Vi har valgt ikke at lave noget front-end validering. Dette var en beslutning som stammede fra den tid vi havde til at arbejde på systemet. Da back-end validering er sikrest valgte vi at begynde med det, og hvis product-owner så mente det vigtigt at have front-end validering også, måtte vi så implementere det senere. Det system vi endte med at bruge, er beskrevet dybere under *Udvalgte Kodeeksempler*.

Systemet vi endte med at bruge til validering, er et Component system, hvor vi i stedet for at have en metode til at validere som man skal huske at kalde alle steder dataen ændres, gemmes dataen i et objekt som internt validerer sig selv hver gang der sker en ændring. Dette system er beskrevet dybere under *Udvalgte Kodeeksempler*.

Vi har valgt have en lille mængde af exceptions, som siger noget om hvilket område i koden de er blevet kastet fra. Det var med den bagtanke at vi med en mindre samling af exceptions som kastes, nemmere kan håndtere hver af dem på sin egen måde. For eksempel bliver ValidationFailedException håndteret ved at tilføje fejl-beskeden under det givne tekstfelt. Havde vi gået med brugen af alle Javas indbyggede exceptions ville det ikke have været lige så realistisk at håndtere dem alle på en god måde.

Vores stykliste-generator-system er designet med fremtidig fleksibilitet i fokus. Vores mål med den var at man senere hen kunne tilføje flere materialer til databasen (altså forskellige dimensioner af planker, forskellige pakke størrelser til skruerne osv.), og så ville systemet bare håndtere at bruge dem også. Dette kræver **MATERIALS_TO_CATEGORY** oversættelsestabellen i databasen, som fortæller hvilke materialer skal bruges i hvilke områder på carporten. Da vi ikke er nået til at implementere siden hvor man kan tilføje nye materialer, er dette ikke gennemtestet. Men designet understøtter den fleksibilitet, så det er ikke et kæmpe omlægningsarbejde når den funktion bygges.

Vi har ikke brugt konkret kodestandard, dog har vi nogen fælles regler vi prøver at overholde så godt so muligt når vi koder. Vi prøver at undgå trainwrecks såsom, [Object].[GetObject].[GetLength], men i stedet bruge noget i stil med [Object].[GetObjectLength].

Klasser stater med stort forbogstav, variabler og metoder bruger kamel case. Alle metodekald skal gå igennem logicFaceden, så man ikke oppe fra .jsp siderne kan kalde direkte ned til mapperne.

Det command pattern projektet er opbygget efter, er også en slags kode standard. Den har vi bare fået udleveret og fået at vide vi skal bruge. Desuden prøver vi så vidt muligt at bruge klar og tydelig indrykning så man nemt at skille loops, if statements og switches fra hinanden.

Styklisten bliver på nuværende tidspunkt ikke gemt. Den bliver derimod genereret on-demand, når en bruger af systemet ønsker det. Dette dykker ned i den evige diskussion om hvor vidt det er det værd at ofre noget hastighed til fordel for hard-disk plads på serveren. I dette tilfælde tager udregningen et split-sekundt og det kan ikke mærkes at den genereres live. Det ville dog være nødvendigt at gemme den når funktionalitet til at redigere den bliver implementeret, men der er vi ikke nået til endnu. Et sekvensdiagram over hvordan styklisten genereres kan findes i *bilag*, som bilag 2.

Udvalgte kodeeksempler

Component systemet

For at garantere at den data vi får ind, er brugbar hele vejen gennem systemet, har vi gjort brug af et component system. I stedet for bare at gemme et navn som en string, og så validere den når den kommer ind i systemet, gemmer vi den som et *NameComponent* objekt som ikke holder andet end den string. Alle vores *Component* objekter arver så fra *Component* interfacet, og derved skal de implementere *validate()* metoden. Den metode kaldes så fra konstruktøren og set-metoden.

Fordelen ved den opsætning er at i stedet for at validere dataen én gang på vej ind i systemet, bliver det uden ekstra arbejde fra programmøren valideret ved hver ændring gennem hele systemet. På den måde er der garanteret data-integritet på alle tidspunkter i systemet.

Hvis en validering fejler, så kastes en *ValidationFailedException* som skal gribes hvor *Component*'en opsættes. På den måde er man som programmør tvunget til at tænke over hvordan en fejlet validering håndteres.

Et eksempel på en *Component* er *PhoneComponent*. Følgende kode-uddrag er fra *PhoneComponent* i pakken Components:

```java
public class PhoneComponent implements Component {
    String phone;

    public PhoneComponent(String phone) throws ValidationFailedException {
        this.phone = phone;
        validate();
    }
    public String getPhone() {
        return phone;
    }
    public void setPhone(String phone) throws ValidationFailedException {
        String old = this.phone;
        this.phone = phone;
        try {
            validate();
        } catch (ValidationFailedException e) {
            this.phone = old;
            throw new ValidationFailedException(e.getMessage());
        }
    }
    @Override
    public boolean validate() throws ValidationFailedException {
        if(phone == null || phone == "") {
            throw new ValidationFailedException("Telefon nummer må ikke være tom.");
        }
        String test = "([1-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9])";
        boolean matches = Pattern.matches(test, phone);
        if(matches) {
            return true;
        }
        throw new ValidationFailedException("Telefon nummer skal indeholde 8 tal.");
    }
}
```

Figur 4. Kode-uddrag fra klassen PhoneComponent

Som beskrevet tidligere, så kaldes *validate()* fra både konstruktøren og set-metoden. Vi gemmer telefonnummeret som en string, da der ikke vindes noget på at have det som en int. Vi skal jo ikke lave beregninger på det.

Først tjekker *validate()* om input er tomt. Derefter sammenlignes det med en RegularExpression. I dette tilfælde, tjekkes der om input er et tal mellem 1-9, fulgt at 7 tal mellem 0-9. Danske numre må ikke starte med 0, så vi tjekker at det første ciffer ikke er et 0.

Vi kunne her have gjort det mere fleksibelt og ladet brugeren indtaste andre tal længder, måske endda have land-koder med, men det virkede som en lavere prioritet ting. Fordelen ved denne *Component* arbejdsmåde er også at hvis product-owner senere beslutter at det skal laves om, så skal det kun ændres ét sted og så fungerer det fra front-end til back-end.

Front-end håndtering af validering

Grunden til at valideringen bare kan ændres ét sted er at vi slet ikke bruger front-end validering. Det var en beslutning vi tog ret tidligt, da det er sikrere at bruge back-end validering, siden der ikke kan snydes ved at ændre i det http-request som sendes. Vi kom frem til at det ville i dette projekt være højere prioriteret bare at få noget godt back-end validering på, og så kan man altid senere udvide det for en bedre brugeroplevelse til noget front-end også.

Problemet med at køre back-end validering er dog at brugerens input først kan tjekkes når de har sendt det afsted og opdateret siden. Normalt betyder det at al brugerens input går tabt, og de skal til at indtaste det igen. Det er en rigtig dårlig brugeroplevelse, så det gjorde vi meget ud af at undgå.

Følgende kode-uddrag er fra *Request.java* i PresentationLayer. Store dele er blevet undladt for at gøre eksemplet mere læsbart.

```
public class Request extends Command {

String execute( HttpServletRequest request, HttpServletResponse response) throws
CommandException {
    […]
    //Phone component
    String phoneString = request.getParameter("phone");
    try {
        phone = new PhoneComponent(phoneString);
    } catch (ValidationFailedException e) {
        request.setAttribute("phoneError",e.getMessage());
        errorsFound = true;
    }
    […]

    if(!errorsFound){
        […]
    }else{
        //Errors found
        errorHandling(request,[…], phoneString);
    }
}

private void errorHandling(request,[…], phoneString ) {
    //Set all input fields to their input if not empty

    //Customer
    request.setAttribute("name",nameString);
    request.setAttribute("address",addressString);
    request.setAttribute("email",emailString);
    request.setAttribute("phone",phoneString);
    request.setAttribute("zipcode",zipString);
}
```

Figur 5. Kode-uddrag fra klassen Request

Denne *Command* kaldes når brugeren har klikket på send. I den gås alle input-felterne på siden igennem, og deres indhold forsøges at placeres i en *Component*. I dette tilfælde er det feltet med id'et *"phone"* som hentes ind, og der laves en *PhoneComponent* ud fra den. Hvis den så ikke består valideringen, fanges den *ValidationFailedException* som bliver kastet, og den *span* som sidder under feltet fyldes ud med fejl-beskeden.

Derefter bliver *errorsFound* sat til true. Dette gør at vi springer al koden omkring at gemme ordren over, og i stedet kalder *errorHandling()* metoden. Den metode tager hver eneste input brugeren lavede som parametre, og går så igennem hvert felt og placerer det i dem.

På den måde ser det for brugeren ikke ud som om siden opdaterede, da alt hvad de havde skrevet ind, stadig står der.

Ved første øjekast virker det dog lidt som spild at vi ikke så snart vi finder en fejl springer ned i *errorHandling()* og sender fejlbeskeden til brugeren, men at vi i stedet fortsætter igennem alle felterne. Det er der dog en grund til. Hvis brugeren havde indtastet et ulovligt input i to eller flere felter, ville der kun vises én fejl ved det første af felterne. Ved at fortsætte igennem alle felterne kan vi vise en fejlbesked ved alle de givne felter på en gang, og brugeren har så mulighed for at fikse dem alle i en omgang. Det var hvad vi kom frem til, gav den bedste brugeroplevelse, ved en meget lille omkostning af responstid.

BillCalculator

Når styklisten skal genereres, sendes ordren til *BillCalculator*, som ud fra informationerne på den begynder udregningen. Følgende uddrag er fra den klasse.

```
public ArrayList<BillLine> calculateBillFromOrder(Order order) throws GeneratorException,
DatabaseException {
        int[] categoriesNeeded = null;
        int orderType;
        //Check which type of order it is and sets the order type
        //Flat roof, no shed = 1
        //Flat roof, with shed = 2
        //Inclined roof, no shed = 3
        //Inclined roof, with shed = 4
        if(order.getIncline() > 0){
            //Inclined roof
            orderType = 3;
        }else{
            //Flat roof
            orderType = 1;
        }
        //Plus one if order has shed
        if(order.isWithShed()){
            orderType++;
        }
        //Select which categories are needed for the selected order type
        switch (orderType){
            case 1: //Flat roof, no shed
                categoriesNeeded = new int[]{1,2,3,4,8,10,11,13,14,15,16,17,18,19,20,21,22,23};
                break;
            […]
        }
        //Get each category from database
        ArrayList<Category> categoriesAvailable =
LogicFacade.getTheseCategories(categoriesNeeded);
            […]
        for(int category : categoriesNeeded){
            switch(category){
                case 1: //understernbrædder til for & bag ende
                    //The material categories needed in the generator method
                    categoryIdsUsedInGenerator = new int[]{1};

                    //Gets a list with only the categories needed
                    categoriesUsedInGenerator =
getCategoriesUsedInGenerator(categoryIdsUsedInGenerator, categoriesAvailable);

                    //Calls the generator and returns the BillLine
                    billLine =
CarportGenerator.underSternsBredderFrontAndBack(categoriesUsedInGenerator,order.getWidthComponent(
));
                    break;
```

Figur 6. Kode-uddrag fra klassen BillCalculator

Først skal typen af carport bestemmes. Dette er nødvendigt da materialerne som skal bruges afhænger af carport typen. Det gøres ved først at tjekke tag-type: Er den flad eller rejst? Der sættes et tal baseret på svaret. Derefter tjekkes der om ordren har et skur. Hvis den har det lægges der 1 til tallet. Det betyder at vi nu har et tal fra 1 til 4 som bestemmer hvilken type carport vi har med at gøre.

Som nævnt tidligere bruger vi et kategorisystem til at bestemme hvad der skal udregnes. Det fungerer ved at hver situation noget skal bruges i er en kategori. De er bestemt ud fra den beskrivelse som står skrevet på styklisten. For eksempel er *understernbrædder til for & bag ende* en kategori. Fordelen ved denne løsning er at der er flere materialer som kan bruges i flere situationer. Et eksempel er at de brædder som blev brugt til den ovennævnte kategori også bruges i *understernbrædder til siderne*. Derved behøver vi kun gemme materialerne en gang.

Det system bliver brugt her ved at vi ud fra carport-typen ved hvilke kategorier bruges. De er i dette tilfælde hard-coded ind. Dette var en beslutning vi tog da vi kom frem til at det er oftere at *materialerne* som bruges til at bygge en carport bliver udskiftet end det er for selve *måden* den bliver bygget. Og skulle måden den bygges på ændre sig ville det jo alligevel kræve en omskrivning af algoritmen.

Derefter hentes alle kategorierne ind fra databasen, og holdes i en række *Category* objekter. Disse objekter holder hver alle de *Materials* som er forbundet til dem gennem **MATERIALS_TO_CATEGORY** oversættelsestabellen. Det er de mulige materialer at bruge til det formål den individuelle kategori repræsenterer. Derfra går vi igennem hver kategori som skal bruges (det er de hard-codede tal), og ud fra tallet køres den rigtige metode fra listen.

Grunden til at vi sætter *categoryIdsUsedInGenerator* individuelt for hvert case er at nogle af dem kræver information fra andre kategorier. For eksempel skal skruerne til vandbrættet vide hvor lange vandbrædderne er, før det kan udregnes hvor mange der skal bruges.

Vi er godt bevidste om at *BillCalculator* i nuværende stadie af systemet er en over-designet løsning. Vi besluttede os for at designe det på den måde fordi vi havde fremtidig fleksibilitet i fokus. Det vi vinder ved den ovenstående løsning, er at beregneren ikke kender de materialer de får ind, før de hentes fra databasen. Det betyder at hvis man som lager-ansvarlig hos Fog køber en anden type planker ind, enten nogle af anden længde eller nogle af anderledes tykkelse, så kan de bare tilføjes til databasen, forbindes til de kategorier de skal bruges i (**MATERIALS_TO_CATEGORY**), og så finder beregneren selv ud af at inkludere dem i udregningen.

Status på implementation

Vi har nået det meste af kerne-funktionaliteten, men vi har en række af ting som ikke er implementeret.

Der er steder i de metoder som *BillGenerator* bruger hvor vi gentager kode. Et eksempel på dette er når vi looper igennem en liste af mulige materialer for at finde den hvis længde passer bedst. Det gør vi flere steder, og steder som det burde være refaktoreret ud i sine egne genbrugelige metoder.

Derudover var vi i snak om at det som vi kalder *Order* i koden og **ORDERS** i databasen burde hedde request. Ordet order er nemlig ikke sigende nok, da det jo på nuværende tidspunkt ikke er en ordre kunden lægger, men derimod en anmodning om konsultation på baggrund af en ønsket carport størrelse.

Vi har heller ikke implementeret den planlagte login- og registrer-funktion. Det betyder at på nuværende tidspunkt har alle brugere adgang til alle sider, og en kunde kan derfor gå ind og se sin egen ordres stykliste, hvilket var noget Fog specifikt bad om ikke kunne lade sig gøre. Et login-system burde derfor prioriteres hvis der arbejdes videre på projektet i fremtiden.

Et problem vi er bevidst om er at hvis siden har stået inaktiv i en længere periode vil forbindelsen til databasen timoute. Det betyder at den første bruger som anvender systemet vil få en fejlet forbindelse ved første forsøg. De er da nødt til at refreshe siden og prøve igen, hvorfra forbindelsen så er genetableret.

Der er ikke lavet CRUD til alle tabeller i databasen. Da vi ikke er noget til et sted i projektet hvor User Stories har dækket de funktionaliteter.

Vi har pt. 4 uløste issues pr. vores GitHub:

- *SaveEditOrder.java* og *Request.java*, er opbygget af nær identisk kode. De burde være refaktoreret ud i en fælles metode så man undgår kode duplikation.
- Logging af de mest gængse fejl er implementeret, men fungerer ikke i runtime. Det kører dog fint gennem unit-tests.
- Når carporten kommer op over en vis størrelse, skal der en ekstra række stolper i midten. Hvis der så også er skur på, bliver tegningen af carporten ikke rigtigt da stolperne i den midterste række bliver placeret forkert.
- Lige nu bruger *ShedGenerator* den samme metode til at placerer stolper som *CarportGenerator*. Det skal også fixes.

Vores implementation sigter som tidligere beskrevet meget efter at være fleksibel da Fog havde et ønske om at kunne tilføje og rette materialer. Derfor er mange af metoderne til at genere styklisten ret omfattende da de sigter at finde det bedst mulige materiale. Der er dog pt meget få kategorier med mere end en mulighed. Det burde dog være muligt at tilføje nye materialer og metoderne så selv vælger hvilke der passer bedst. Vi er desværre ikke nået til at teste det.

Derudover havde vi, for at leve op til Fogs ønske om kommunikation mellem kunde og sælger, planlagt et system som ville lade sælgere markere en ordre og derved tage ansvar for den. Planen var at sælgere ville kunne sortere alle ordrer efter hvorvidt de er blevet tildelt en sælger eller kun vise dem som er tildelt den sælger som er logget ind. Fordelen ved dette system vil være at det vil introducere struktur til sælger-kunde kommunikationen, og at kunden på et hvert tidspunkt kan se status for deres forespørgsel.

Test

Vi har primært brugt unit-tests til at teste vores system. Vi har forsøgt at teste så mange af de ting som kan med rimelighed testes gennem unit-tests. Alle coverage procenter er på linje-basis.

Component:

Klasse	Metode navne	Coverage
AddressComponent	*Validate()*, *setAddress()*	100%
CityComponent	*Validate()*, *setCity()*	100%
DepthComponent	*Validate()*, *setDepth()*	100%
EmailComponent	*Validate()*, *setEmail()*	100%

HeightComponent	*Validate()*, *setHeight()*	100%
InclineComponent	*Validate()*, *setIncline()*	100%
MaterialHeightComponent	*Validate()*, *setHeight()*	100%
MaterialLengthComponent	*Validate()*, *setLength()*	96%
MaterialWidthComponent	*Validate()*, *setWidth()*	100%
NameComponent	*Validate()*, *setName()*	100%
PhoneComponent	*Validate()*, *setPhone()*	100%
ShedDepthComponent	*Validate()*, *setDepth()*	91%
ShedWidthComponent	*Validate()*, *setWidth()*	91%
WidthComponent	*Validate()*, *setWidth()*	100%
ZipCodeComponetn	*Validate()*, *setZip()*	100%

Vores *Components* er ret gennemtestede, med både happy-path og unhappy-path. Det er kun *Validate()* og set metoderne som har kode der skal testes.

DataMappers:

Klasse	Metode navne	Coverage
MaterialsMapper	*getAllCategories()*, *getTheseCategories()*	72%
OrderMapper	*createOrder()*, *updateOrder()*, *getOrder()*, *getAllOrders()*	70%

Vores datamappers er testet på happy-path kun. De kunne gavne af at blive udviddet til at teste unhappy-path også. Disse test-klasser arver fra klassen *TestDataSetup* som indeholder en @before metode der kører vores test-data script. Dette sørger for at hver metode har den samme data, og hvis en af dem piller ved databasen, påvirker det ikke den næste.

BillGenerator:

Klasse	Metode navne	Coverage
CarportGenerator	*underSternBredderFrontAndBack()*, *sternBredderSides()*, *overSternBredderFront()*, *remInSidesCarport()*, *sperOnRem()*, *posts()*, *perforatedBand()*, *universalBeslagRight()*, *universalBeslagLeft()*, *screwsForSternAndWaterBoard()*, *screwsForUniversalBeslagAndPerforatedBand()*, *boltsForRemOnPost()*, *skiverForRemOnPost()*,	66%
FlatRoofGenerator	*waterBoardOnSternSides()*, *waterBoardOnSternFront()*, *roofPanels()*, *screwsForRoofPanels()*	100%
GeneratorUtilities	*sortMaterialsByLength()*, *sortMaterialsByWidth()*, *searchForAmountInCategoryFromBillLines()*, *calculateRoofLength()*, *calculateRoofHeight()*	96%

InclinedRoofGenerator	*topRoofLath()*, *topRoofLathHolder()*, *rygstenBracket()*, *roofTileBinders()*, *screwsForRoofLaths()*, *roofTiles()*	87%
ShedGenerator	*zOnBackOfDoor()*, *losholterGabled()*, *remInSidesShed()*, *losholterSides()*, *boardForShed()*, *stalddorsgreb()*, *hingeForDoor()*, *vinkelBeslag()*, *screwsForOuter()*, *screwsForInner()*,	79%

Klasserne som genererer styklisten, var ret svære at teste. De krævede at vi første lavede udregningerne for den givne størrelse carport/skur, så vi vidste hvad vi skulle forvente. Det store arbejde betød at vi ikke var særlig grundige omkring det da vi var under tidspres. Havde vi haft mere tid sat af til at fokusere på at gennemteste systemet havde vi nok fokuseret på at komme godt omkring *BillGenerator*.

Det første skridt ville nok være at dele de store metoder op i mindre dele så man kan teste hvert stykke, i stedet for kun at teste på helheden.

Forneden ses

Spørgsmål	**Teststrategi**
Virker vores beregner?	Hver metode har én unit-test som tjekker om beregningen passer med den værdi vi på forhånd har udregnet os til at resultatet skal være.
Passer antallet af stolper?	Én unit test, plus visuelt tjek af tegning som baserer sig på styklistens resultat.
Passer antallet af spær?	Én unit test, plus visuelt tjek af tegning som baserer sig på styklistens resultat.
Andre beregninger?	Samme som foroven.
SVG-motor	Manuel test af en så bred som muligt række af dimensioner
Kan jeg tegne et rektangel med en vis størrelse? Virker vores metode?	Manuel test
Kan jeg tegne en pil som f.eks. er 300px lang?	Manuel test
Datamappere	Happy-path unit-tests af hver metode
Er der overhovedet hul igennem til databasen?	Unit-tests
Kan jeg indsætte en ny bruger?	N/A
Kan jeg hente et produkt / liste af produkter?	N/A
Bliver en ordres status gemt i databasen når man opdaterer den på jsp siden?	Manuel test
Hvad med vores brugergrænseflade?	
Kan en kunde finde ud af at bestille en carport? Hvordan afgør vi det?	Q/A test af familie/venner
Kan de ansatte hos Fog finde ud af at bruge systemet?	Ikke testet, men manuel test.
Hvordan håndterer vi input validering?	Component-system tjekker værdier internt.
Generelt	
Er vores kode skrevet, så vi automatisk kan teste den?	Dele af den er, andre dele burde deles op i mindre metoder så de individuelle dele kan testes
Hvordan sikrer vi os en ensartet kodekvalitet?	Man aftaler på forhånd en række regler for hvordan kode opbygges og skrives. Dette kan være

	mere generelt, som ”Undgå tranwrecks i metodekald”, men det kan også være helt specifikt som ”Brug curly-braces hvor det er valgfrit”.
Hvordan sikrer vi os at vi ikke tjekker fejlagtig kode ind på vores master- eller developerbranch i GitHub?	Idealet ville være at vi ved hvert pull-request havde nogen til at teste manuelt, samt køre unit-tests, men det gjorde vi ikke altid.

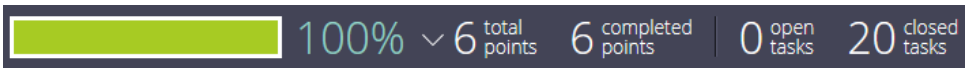
Process

Arbejdsprocessen faktuelt

Vi valgte ikke at bruge en SCRUM-Master. Dette uddybes i *Arbejdsprocessen reflekteret*. Vores lille gruppe gjorde også SCRUM morning meetings lidt redundante da vi ofte sad og snakke over Zoom det meste af dagen.

Retrospective meetings blev afholdt den dag sprintet sluttede. De fungerede til at få et overblik over hvor langt vi var i hele processen med hensyn til hvor mange userstories vi nåede. Desuden brugte vi dem som værktøj til at finde et bedre tidsestimat til næste sprints userstories.

1. Sprint



Figur 7. Sprint 1. header

1.1. Customer - Carport dimensions

1.1.1. As a customer, I want to be able to enter the dimensions of the carport I want, so that I can order the exact size I need.

1.2. Customer - Specify the dimensions of the shed

1.2.1. As a customer, I want to be able to specify the dimensions of the shed, so that I can get best size of space I need.

1.3. Customer - Select shed or not

1.3.1. As a customer, I want to be able to select whether I want a shed or not, so that I can get the kind of storage space that I want.

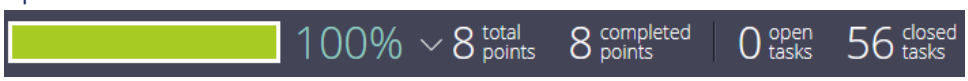
1.4. Customer - Select flat or a slanted roof

1.4.1. As a customer, I want to be able to select whether I want a flat or a slanted roof, so that the roof fits the aesthetics I want.

1.5. Customer - Send selections to Fog salesperson

1.5.1. As a customer, I want to be able to send the selections I make to a salesperson at Fog, so that I can make an order.

2. Sprint



Figur 8. Sprint 2. header

1.1. Employee - Create bill of materials for carport

1.1.1. As an Employee, I want to be able to create a bill of the required materials for an ordered carport's main body, so that I can immediately give the customer an exact price.

1.2. Employee - Create bill of materials for flat roof

1.2.1. As an Employee, I want to be able to create a bill of the required materials for an ordered carport's flat roof, so that I can immediately give the customer an exact price.

1.3. Employee - Create bill of materials for shed

1.3.1. As an Employee, I want to be able to create a bill of the required materials for an ordered carport's shed, so that I can immediately give the customer an exact price.

3. Sprint

Figur 9. Sprint 3. header

3.1. Employee - Create bill of materials for slanted roof

3.1.1. As an Employee, I want to be able to create a bill of the required materials for an ordered carport's slanted roof, so that I can immediately give the customer an exact price.

3.2. Employee - View bills of material

3.2.1. As an Employee, I want to be able to view the bill of materials that has been created, so that I can actually make use of them

3.3. Employee - View placed orders

3.3.1. As an Employee, I want to be able to view the orders placed, so that I can enter them into the system.

3.4. Employee - view the contents of a specific order

3.4.1. As an Employee, I want to be able to view the contents of a specific order, so that I can see what the customer wants.

4. Sprint

Figur 10. Sprint 4. header

4.1. Display carport without shed, flat roof

4.1.1. As a customer, I want to be able to view a preview of the carport I have selected, before sending it off, so that I can visualize what I'm ordering, as well as make sure it's correct.

4.2. Customer - Display carport with shed, flat roof

4.2.1. As a customer, I want to be able to view a preview of the carport I have selected, before sending it off, so that I can visualize what I'm ordering, as well as make sure it's correct.

4.3. Customer - add contact information to the order

4.3.1. As a customer, I want to be able to view a preview of the carport I have selected, before sending it off, so that I can visualize what I'm ordering, as well as make sure it's correct.

4.4. Employee - Edit customer selection

4.4.1. As an Employee, I want to be able to edit the selections which the customer has made, so that I can talk the customer through the process and give advice as a salesperson.

4.5. Customer - Display carport without shed, slanted roof

4.5.1. As a customer, I want to be able to view a preview of the carport I have selected, before sending it off, so that I can visualize what I'm ordering, as well as make sure it's correct.

4.6. Customer - Display carport with shed, slanted roof

4.6.1. As a customer, I want to be able to view a preview of the carport I have selected, before sending it off, so that I can visualize what I'm ordering, as well as make sure it's correct.

4.7. Sys-admin - Log files

4.7.1. As a systems admin, I want to have log files logging any errors that might come up, so that I can have any problems fixed.

5. Overskydende Stories

5.1. Customer - Download my order as a PDF

5.1.1. As a customer, I want to be able to download my order as a PDF, so that I can save it, or bring it to a salesperson in the store.

5.2. Employee - log in as an administrator

5.2.1. As an Employee, I want to be able to log in, so that I can access the admin only settings.

5.3. Customer - register as a user

5.3.1. As a customer, I want to be able to register as a user, so that I can have the site remember my information

5.4. Customer - change password

5.4.1. As a customer, I want to be able to edit my password, email and contact info, so that I can change these when I need to.

5.5. Customer - view my account information

5.5.1. As a customer, I want to be able to view my account information, so that I can see what information is saved on my account.

5.6. Customer - select variant of material for roof

5.6.1. As a customer, I want to be able to select the variant of material for my roof, so that I can make the carport fit the aesthetics I am going for.

5.7. Customer - select variant type of carport

5.7.1. As a customer, I want to be able to select the variant of material for my carport, so that I can get the exact build I am looking for.

5.8. Employee - delete, add or edit the available material sizes

5.8.1. As an Employee, I want to be able to delete, add or edit the available material sizes, so that the options which the user is presented with, match what we can deliver.

5.9. Employee - delete, add or edit the available material variants

5.9.1. As an Employee, I want to be able to delete, add or edit the available material variants, so that the options which the user is presented with, match what we can deliver.

5.10. Employee - Send offer

5.10.1. As an employee, I want to be able to send an offer for a price to a customer, so that I can give them the best possible price they are willing to pay.

5.11. Customer - see/accept offer made by the salesperson

5.11.1. As a customer, I want to be able to see/accept the offer made by the salesperson, so that I can decide whether I'm interested.

5.12. Customer - status of order / offers

5.12.1. As a customer, I want to be able to see the status of my order, and any offers made, so that I can keep track of where in the process my order is.

5.13. Customer - Pay for order

5.13.1. As a customer, I want to be able to pay for my order, so that I can actually complete the purchase.

5.14. Customer - view parts from past orders

5.14.1. As a customer, I want to be able to view the parts I've purchased, after paying for my order, so that I know what I'm receiving and can begin planning.

5.15. Employee - assign orders to individual salespeople

5.15.1. As an Employee, I want to be able to assign orders to individual salespeople, so that I can organize which salesperson is responsible for each order.

5.16. Employee - View orders with no assignees

5.16.1. As an Employee, I want to be able to view only the orders that haven't been assigned to any salespeople, so that I can select which I want to take care of.

5.17. Employee - view only the orders assigned to me

5.17.1. As an Employee, I want to be able to view only the orders assigned to me, so that I don't have to view orders that aren't relevant to me.

5.18. Employee - Default materials

5.18.1. As a salesperson, I want to have a range of default materials that cannot be deleted, so that if I delete all materials of a type, the program has something to fall back on.

5.19. Employee - Edit the Bill of an individual order

5.19.1. As an employee, I want to be able to edit the specific materials used in a bill, so that I can make sure the correct materials are used, when building the carport

5.20. Customer - Live preview of carport

5.20.1. As a customer, I would like to be able to see a live preview of my carport as I enter the dimensions. so that I know what I'm ordering

5.21. Customer - View 3D model of carport

5.21.1. As a customer, I want to be able to view a 3D model of the carport, so that I can visualize the finished product.

Vi gjorde stærkt brug af tasks til at opdele User Stories i mindre bidder samt til at holde øje med hvem der havde ansvar for hvad. Derudover brugte vi vores simple tidsestimering, som vil blive beskrevet i *Arbejdsprocessen reflekteret*, til at vægte størrelsen af de forskellige stories og derfra beslutte hvilke vi startede et sprint med og hvilke måtte vente. Bortset fra første sprint, hvor en masse opsætning af infrastruktur var nødvendig, prioriterede vi for det meste de større User Stories i et sprint.

Et eksempel på en fuldt beskrevet User Story er:

- Customer – Send selections to Fog salesperson
    - As a customer, I want to be able to send the selections I make to a salesperson at Fog, so that I can make an order.
    - Acceptance requirement:
        - This story is complete when an order that has been sent is visible in the database.
    - Tasks:
        - Setup Connector-class correctly
        - Create class for orders
        - Create table for orders

- Create mapper class for orders
- Create order through ordermapper when clicking send
- Test mapper class
- Create Components for each input-type
- Create ValidationFailedException
- Validate input for each component
- Handle exceptions when input validation fails
- Create tests for component validation
- Create equals() method in each component

Arbejdsprocessen reflekteret

Når vi ser tilbage på vores arbejdsproces, kan vi godt se at vi ikke holdt os tæt nok op imod Scrum-arbejdsprocessen: Som sagt havde vi ikke en Scrum-Master. Det var en utalt beslutning vi kom til, da vi fandt det underligt, at en i gruppen skulle bestemme mere end andre. Desuden fandt vi det ikke nødvendigt med en der havde mere styr på opgaven end andre da vi kun er 3 i gruppen, og projektet var relativt småt. Så alle har en god indsigt i hele projektet.

Derudover holdt vi, som nævnt før, heller ikke rigtig Daily-Scrum-Meetings hver dag. Vi forsøgte tre-fire gange, men da vi som sagt talte sammen live det meste af vores arbejdstid fik vi ikke rigtig noget ud af det. Vi er så lille en gruppe at vi hele tiden, uden meget arbejde, havde konstant overblik over hvad alle de andre lavede.

På mange måder føltes Scrum som en byrde mere end en hjælp. Det kom af at den struktur som det skulle give endte med at føles restriktivt. Vi havde rigtig svært ved at finde ud af hvornår ting som planlægning af design (både visuelt og kodemæssigt), tegning af diagrammer, ændring i gamle klasser mm. skulle laves. Ting som ikke giver direkte værdi til product-owner (eg. Planlægning af fleksibelt design, refaktorering af kode, mm.) var meget svært at få passet ind i rammerne for Scrum.

Vi havde også rigtig svært ved at vurdere de første user-stores i hver del af systemet, da der inkluderet i dem jo er en hel masse opsætning af infrastruktur som skal på plads før arbejdet overhovedet kan starte. De steder blev ofte til bottle-necks hvor nogen ikke kunne arbejde videre, før en anden havde færdiggjort opsætningen. I de situationer savnede vi friheden ved ikke at have sprints, så en af os kunne have lavet det ugen før, så det var klar når vi andre nåede dertil. Dette er dog nok mest et problem med vores manglende erfaring inden for denne arbejdsproces.

Noget vi må tage med fremover, er at vi skal arbejde på at holde vores User-Stories små og fokuserede. Vi endte, især med *BillGenerator* klasserne, at have ufatteligt mange Tasks i hver User-Story, selv efter vi havde delt dem op i 4 (Fladt-tag + skur, fladt-tag % skur, rejst tag + skur, rejst tag % skur).

Vores tidsestimering var også meget upræcis. Første sprint nåede vi 6 point, andet 8, tredje 8 igen, og fjerde 17. Da vi aldrig har tidsestimeret før holdt vi os til en simpel small/medium/large opdeling. Dette gjorde det hurtigt at estimere, men gjorde det også svært at være konsekvent. Det ville nok have hjulpet hvis vi havde defineret hvad vi mente med small, medium og large. For eksempel kunne vi sige small er under 2 arbejdstimer, medium er under 5 osv.

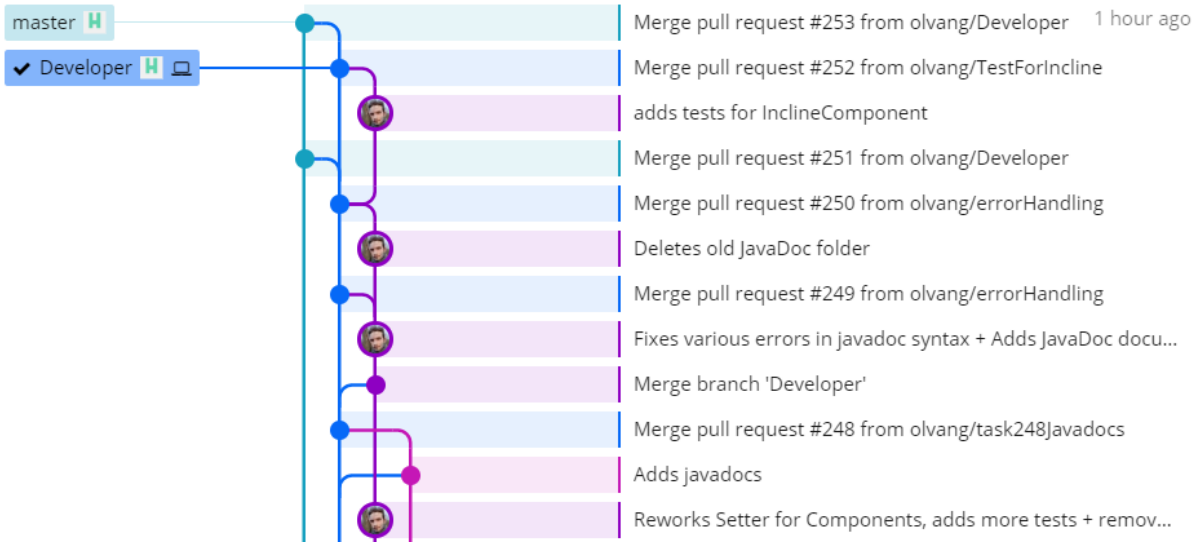
Fremadrettet vil vi nok forsøge at holde os mere til Scrum-arbejdsprocessen, da vi nok ikke fik det ud af det som vi burde have. Hvis vi senere bliver en større gruppe, eller vi begynder at arbejde mere asynkront vil det garanteret udvide vores udbytte af den struktur det giver.

GitHub

Vi har brugt GitHub til at sikre os versions-kontrol. Til det har vi en master og developer branch. Master branchen bruges til helt færdig kode der virker, og får developer merget ind i sig hver gang et sprint er færdigt. Så den nyeste færdige kørende kode ligger på den. Developer er den der bruges når der bliver færdig gjort en task.

Hver gang man assigner en task eller et issue opretter man en ny branch. Denne branch kalder man så enten task eller issue fulgt af tallet på den task eller issue, plus en kort beskrivelse af dette. Dette kunne være fx resulterer i **task236componentsForCustomer** for en task, eller **issue246DeletingUnused** for et issue.

Vi har brugt GitKraken til nemmere at visualisere de forskellige branches i forhold til hinanden. Det gør det også nemmere at oprette commits og pull requests.



Figur 11. Uddrag fra GitKraken

Det kunne fx se således. Her er developer næsten lige blevet merget ind i master da det er i slutningen af et sprint.

Taiga

(https://tree.taiga.io/project/olvang-fog-byggemarked/backlog)

Vi har brugt onlineværktøjet Taiga til at organisere vores sprints, User Stories, tasks og tidsestimeringer. Vi blev instrueret til at tidsestimeringen bare skulle bare small, medium, large og ekstra large som indikator for hvor lang tid vi tænkte ting ville tage.

Alle tasks der ikke er assignet til nogen ligger i *new* fanen. Herfra rykkes til de til *in progress* når der bliver assignet til en person. Når en task så er færdig, og den branch man arbejder på er klar til at blive merget med developer ligger man den over i *ready for test*. Til slut når man har løst alle merge conflicts og succesfuldt har merget ligges tasken i *closed*.

Konklusion

I dette projekt har vi, gennem vores første møde med Scrum processen, lavet fundamentet til et system som kan erstatte Fog Trælast og Byggemarkeds nuværende to IT-systemer. I det afleverede produkt er kernefunktionaliteten som Fog ønskede, men der er plads til videre arbejde.

Scrum arbejdsprocessen blev ikke fulgt til præcision, hvilket vi som gruppe vil tage med videre og prøve at forbedre fremover. Helt præcist vil vi gøre mere brug af Daily Meetings, udvælge en Scrum Master samt være mere specifikke i tidsestimering.

Derudover har vi kunne se at de få gange vi har taget en beslutning om hvordan koden skal opsættes, har været rigtig god for hvor *konsistent* vores kode har været. Fremover vil vi overveje at tage en rigtig kode-standard i brug for at skabe en mere ensartet kode.

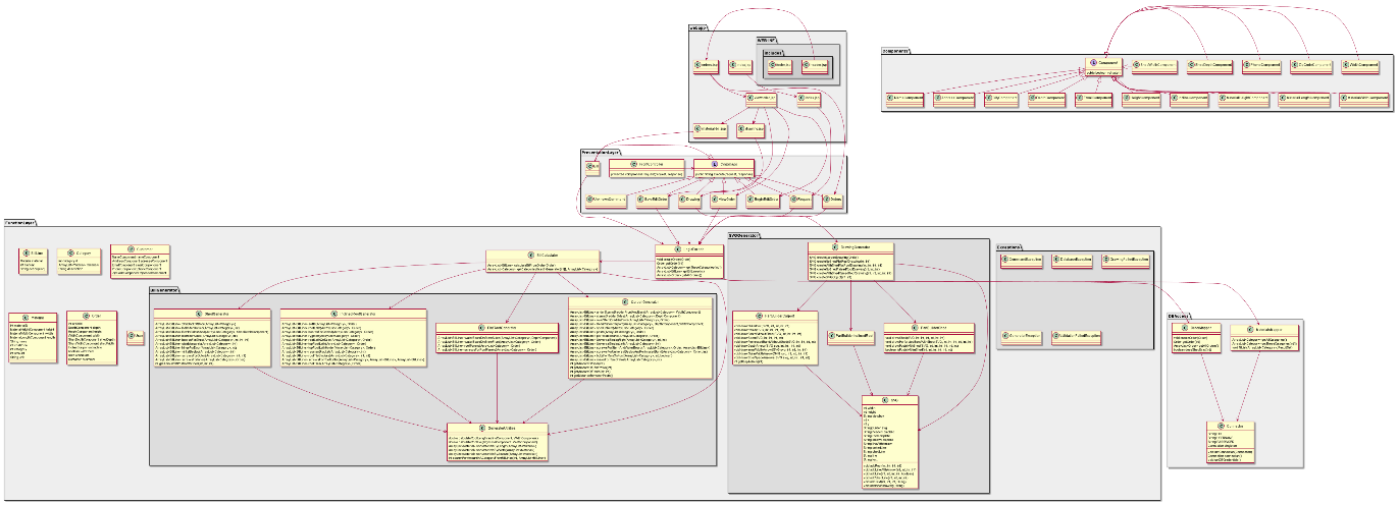
I samarbejde med Product-Owner identificerede vi de mest essentielle dele af systemet og fik dem lavet først. Af den grund er det endelige produkt et hvor de vigtigste funktioner er implementeret: Systemet bruger en database til at gemme alle ordrer, samt Fogs sortiment af materialer. Derudover har systemet en web-baseret front-end som lader både kunden og Fogs sælgere bruge systemet. Systemet kan ud fra en kundes ordre så udregne en stykliste af materialer som skal bruges til den givne carport, samt lave en tegning af den. Begge disse kan ses i web-interfacet.

Der er dog en række planlagte funktioner som ikke blev nået. Disse var mindre vigtige for systemet, og blev derfor i samarbejde med Product Owner prioriteret lavere. De resterende User Stories er sorteret efter den prioritering vi ville bruge, skulle vi arbejde videre på systemet.

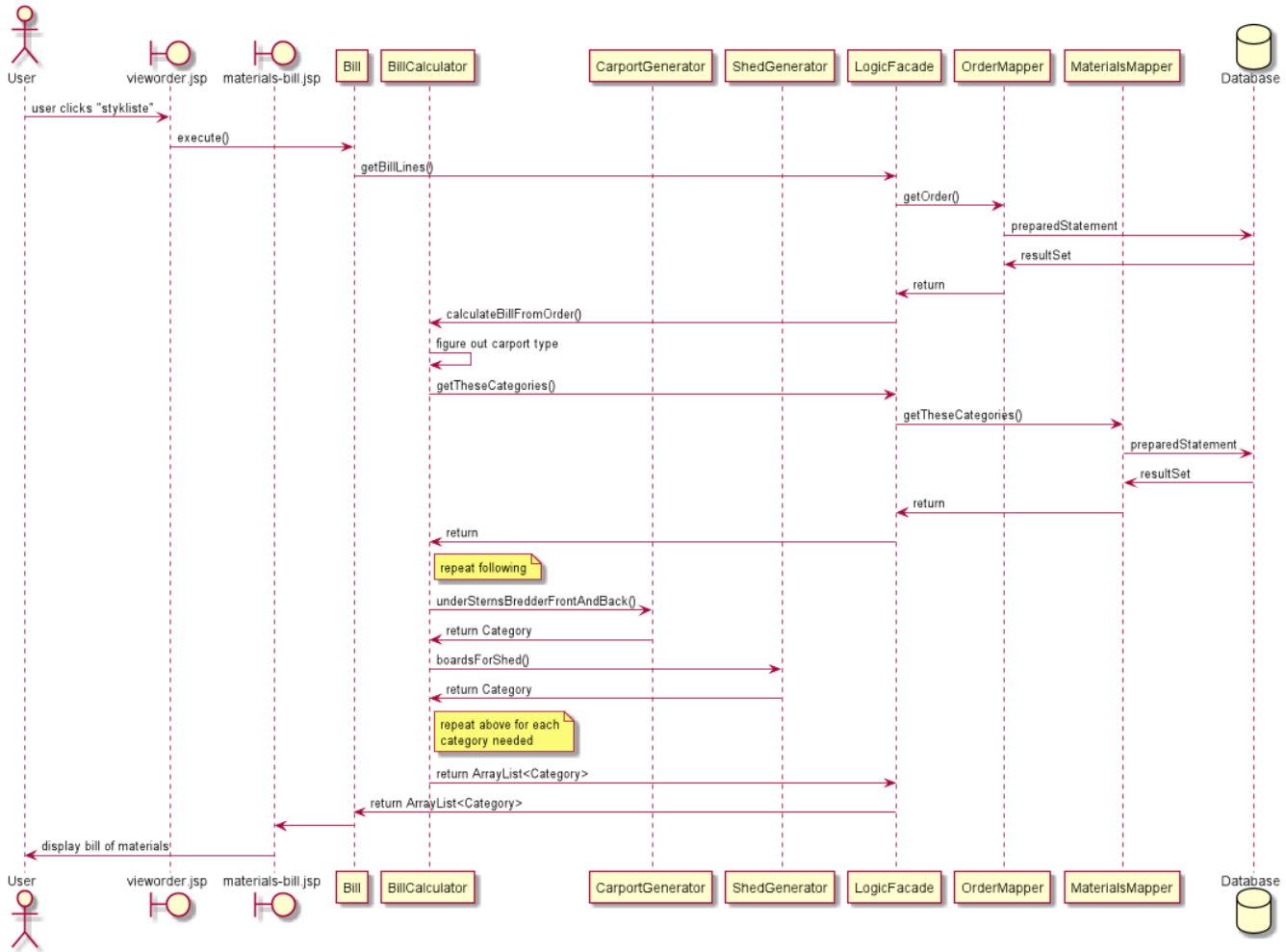
Alt i alt er vi, som gruppe, godt tilfredse med det endelige produkt og det tror vi også at Fog ville være.

Bilag

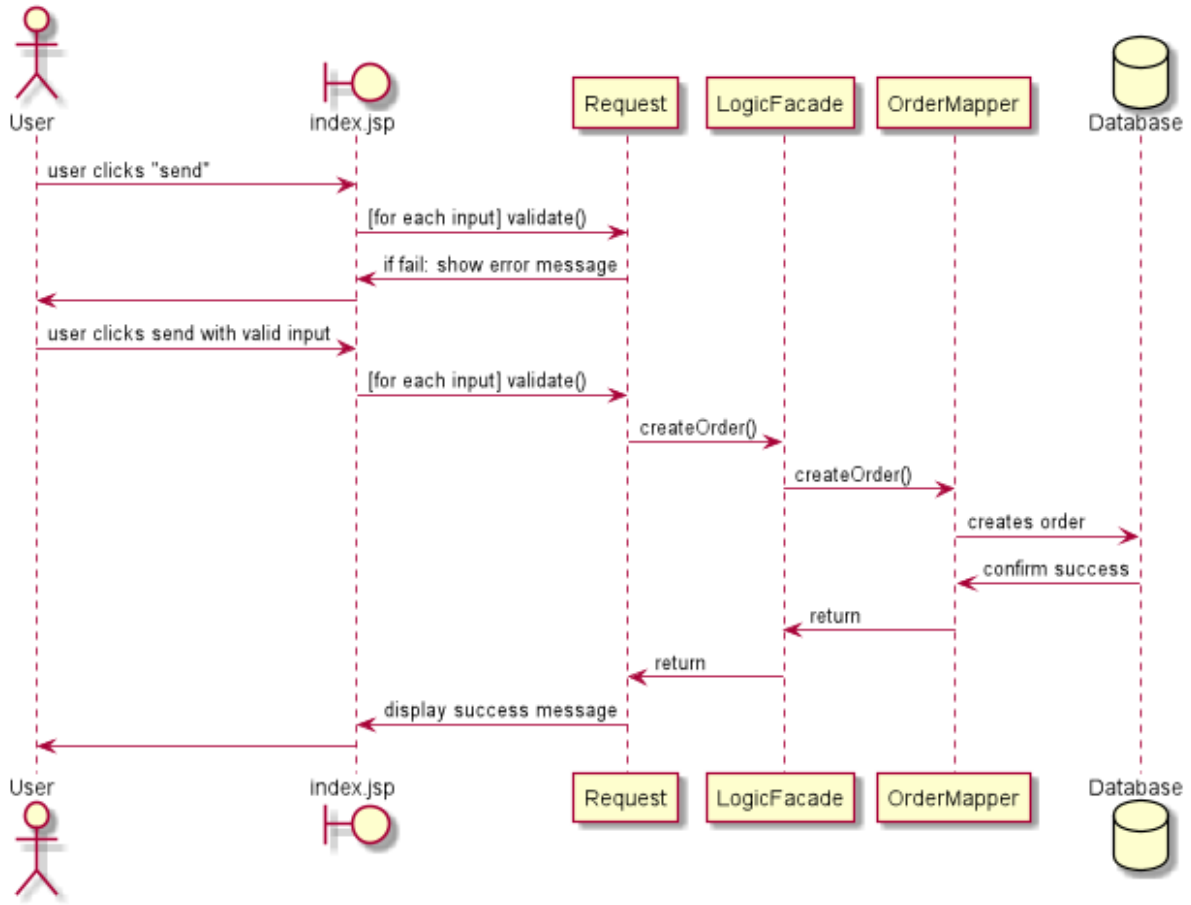
Bilag 1.



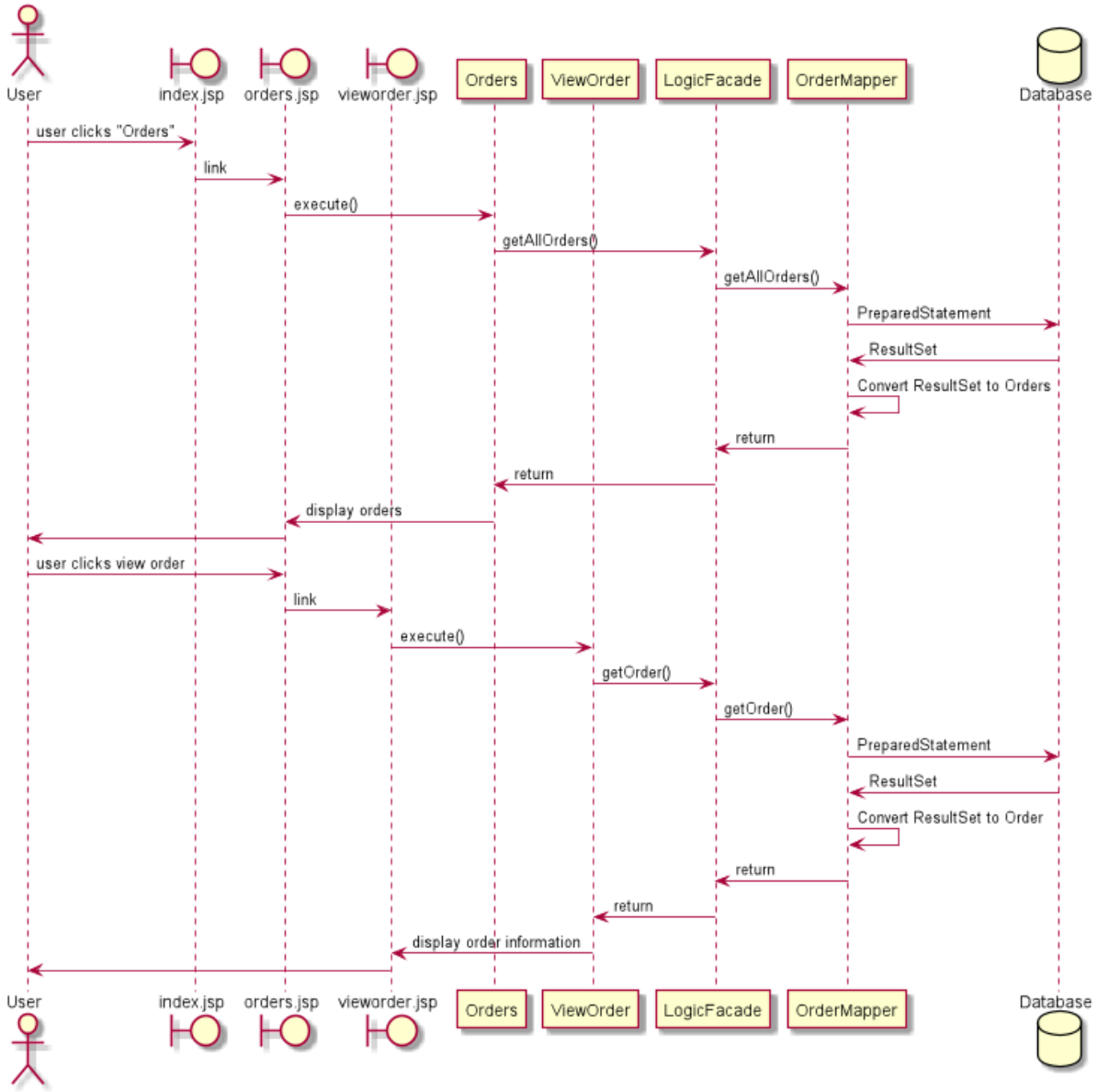
Bilag 1. Klasse diagram, fuld størrelse version kan findes på GitHub under: other -> diagrams -> classdiagram.png



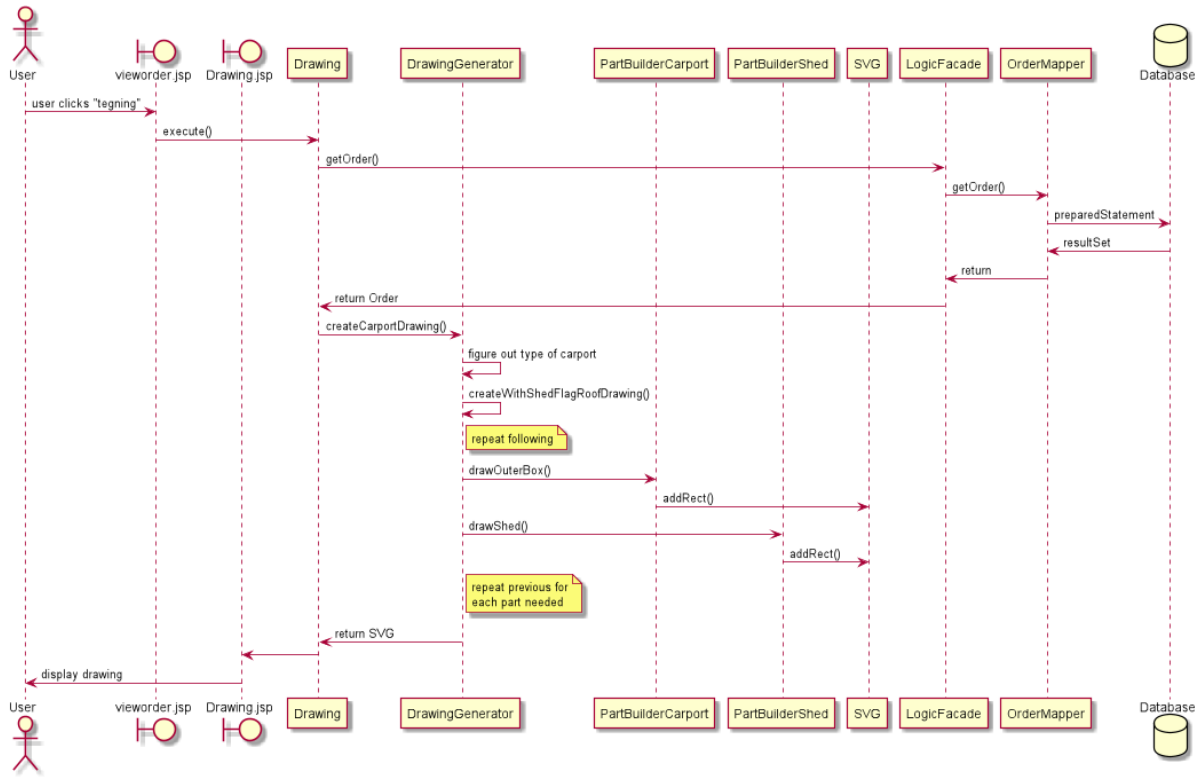
Bilag 2. Sekvensdiagram over generering af styklisten. Fuld størrelse kan findes på GitHub under: other -> diagrams -> SequenceDiagramBill.png



Bilag 3. Sekvensdiagram over oprettelse af en ordre. Fuld størrelse kan findes på GitHub under: other -> diagrams -> SequenceDiagramOrder.png



Bilag 4. Sekvensdiagram over processen i at se indholdet af en bestemt ordre. Fuld størrelse kan findes på GitHub under: other -> diagrams -> SequenceDiagramViewOrder.png



Bilag 5. Sekvensdiagram over genereringen og visningen af en tegning. Fuld størrelse kan findes på GitHub under: other -> diagrams -> SequenceDiagramDrawing.png