CPHBusiness

Johannes Fog Semester Projekt

Dat16EA, gruppe A1, 29. maj 2017

David Carl

Kristian Krog

Tjalfe Møller

Kasper Breindal

Indhold

[Business Case 3](#_Toc483813597)

[Vision 3](#_Toc483813598)

[Activity Diagram 4](#_Toc483813599)

[Domain Model 6](#_Toc483813600)

[Arkitektur og Design Mønstre 7](#_Toc483813601)

[Design lag 7](#_Toc483813602)

[Fordele ved 3 lags arkitekturen 8](#_Toc483813603)

[Design mønstre 8](#_Toc483813604)

## Business Case

Af David

Fog ønskede sig et nyt IT-system som skulle erstatte deres 20 år gamle system som de brugte. En af begrundelserne for at vi skulle bygge det nye til dem var fordi de havde mistet source-koden til programmet så de ikke kunne opdatere det mere, og deres program begyndte at blive langsommere og mere besværligt for deres medarbejdere at bruge.

Lige nu får Fog en e-mail når en kunde bestiller et tilbud som de selv designer, hvorefter at en medarbejder selv skal skrive værdierne ind i deres program for derefter at få pris og materialeliste. Der er desværre ingen tegning på hvordan carporten kommer til at se ud. Efter betaling af den ønskede carport får kunden udleveret en byggevejledning som PDF som en medarbejder har skulle sidde og lave manuelt med de ting som ville være relevant for kundens carport.

De havde nogle krav til det her IT-system.

Mulighed for at få kontakt med kunden, hvordan havde de ikke specificeret til os.

Fog ønskede en 2D tegning der gjorde det muligt at få en idé om hvordan carporten kom til at se ud, men det måtte ikke være for detaljeret så kunden kunne lade være med at købe carporten af dem, men købe materialerne hos en konkurrent, for derefter at bygge den selv.

De sagde samtidig også at hvis man kunne lave 3D model ville det være en ting at foretrække og vi meget gerne måtte gøre det.

Man skulle ikke kunne købe carporten direkte da fog gerne ville havde kontakt til kunden inden de begyndte at sætte sig ind i det store forløb, for at sikre sig at kunden vidste hvad de gik ind i, og så fog kunne komme med forslag og sikre sig at alt var i fineste orden.

### Vision

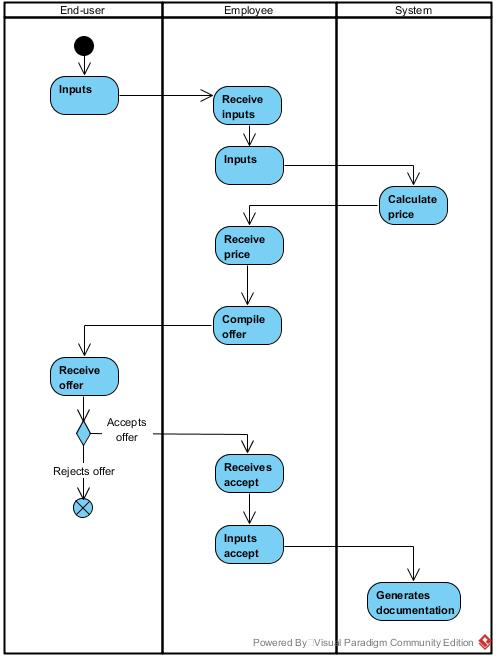
Af David

Idéen med programmet er at gøre det enklere for medarbejderne, at håndtere de ’ønsker’ kunderne kommer med. Det skal samtidig være en bedre oplevelse for kunden bestille en ikke standard carport fra fog. Dette har vi planer om skulle ske ved at vi laver en visuel repræsentation af carporten undervejs, så kunden nemt kan lege rundt med forskellige mål og på den måde kan se hvordan den ville komme til at se ud. Dette kunne vi godt tænke os at gøre med noget 3D og 2D tegning. 3D for kunden så de kunne se den fra alle vinkler, og 2D for at få det godkendt hos kommunen.

### Activity Diagram

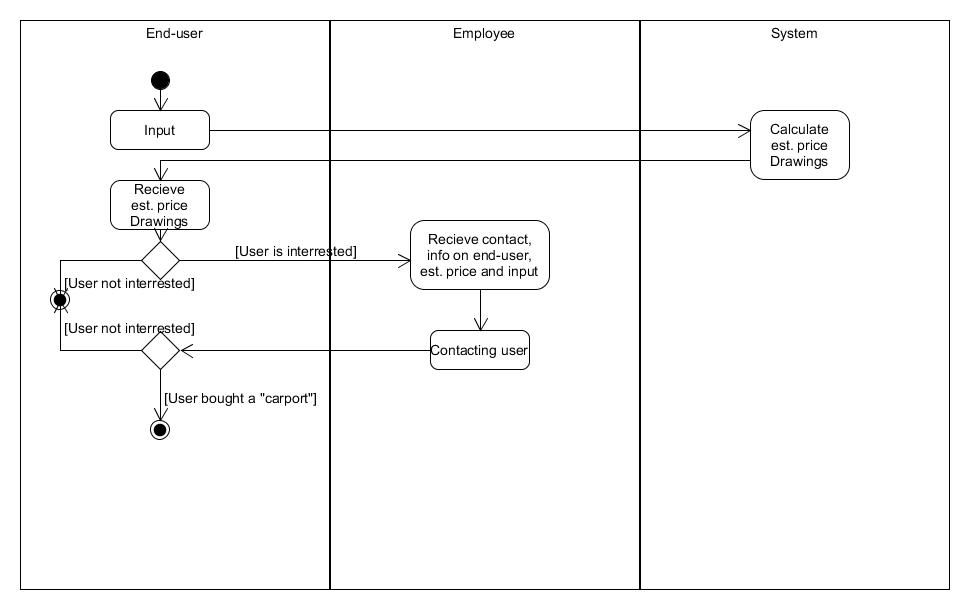
Af Kristian

**As is**



As is diagrammet viser den nuværende arbejdsgang med det nuværende system.

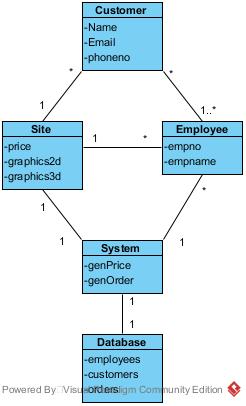
**To be**



To be diagrammet viser hvordan vores program automatisk håndterer en del ting som Fog-medarbejderen tidligere skulle håndtere selv. Det vil effektivisere arbejdsgangen og give mere tid til customer maintenance ved hver salg. Det vil også skære behandlingstiden ned som helhed, da kunden kan se en estimeret pris før kunden kontakter Fog omkring endelig pris og køb.

### Domain Model

Af Kristian



Vores domænemodel illustrerer simpelt hvordan vi har tænkt løsningen skulle opbygges. Kunder vil aldrig komme i kontakt med andet en en frontend, der vil vise dem en ca pris og et par autogenererede modeller af deres ønskede carport, og en af Fogs sælgere. Siden har adgang til systemet primært for at kunne generere en ca pris, men også for at kunne gemme kundens info (sker kun hvis kunden ønsker det). Fogs ansatte har også adgang til systemet, både for at kunne se den eksakte pris, og for at kunne se en liste over materialer. Al kontakt til databasen sker via vores system og aldrig direkte.

## Arkitektur og Design Mønstre

Af David

Når man skal bygge en applikation som skal kunne vedligeholdes og skrives på af mange forskellige programøre, er det en god ting at følge en anerkendt godkendt arkitektur. Dette gør det nemmere at vedligeholde i form af at en programmør ikke skal bruge længere tid på at lære programmet at kende end at vedligeholde det. Det gør det også nemmere at arbejde på sammen flere ad gangen da det giver nogle faster rammer til hvordan det skal se ud.

### Lagdesign

Vi har valgt at bruge 3 lags arkitekturen. Dette er en velkendt teknik blandt IT-folk så derfor gør det den nem at arbejde med. De 3 lag er følgende:

* Præsentations laget
* Logik laget
* Data laget

**Præsentations-laget**

Dette er hvad brugeren af vores applikation kommer til at ende med at se. Det er her at brugeren kommer med input og får de output vores program giver dem. I vores tilfælde er det gennem en webbrowser, da det vi har lavet er en hjemmeside. Alt det brugeren bliver serveret sker igennem JSP filer, men bruger aktivt logik laget for at finde ud af hvad den skal vise til brugeren. Dette sker igennem servlets.

**Logik-laget**

Det er her den meste logik foregår, det foregår igennem servlets som bruger metoder fra andre Java classer vi har lavet. Her foregår det som brugeren ikke skal se, som i vores tilfælde for eksempel er udregning af pris og dermed også en stykliste over de ting som kunden skal bruge til at bygge deres carport de designer. Her foregår alt kommunikation også til vores data lag, også selvom det er vores præsentation lag som skal bruge det.

**Data-laget**

Data lageret bliver kun brugt til kommunikation til vores MySQL database som under udvikling har ligget på vores Digital Ocean Droplet (Dette er en virtual linux server), dette sker igennem SQL calls. Dette er det eneste sted hvor der faktisk bliver snakket med databasen.

**Lille eksempel**

Når vores præsentations lag skal vise for eksempel noget med ikke færdige ordrer, så sender den en efterspørgsel til vores logik lag. Vores logik lag finder så ud af præcist hvad det er vores præsentations lag er interesseret i og spørger efter det i vores data lag. Vores data lag henter så de data ud af vores database og sender det tilbage til vores logik lag som så gør det klar til at vores præsentations lag ved hvordan det skal præsenteres til medarbejderen som har efterspurgt det og på den måde fungere de 3 lag sammen.

### Fordele ved 3 lags arkitekturen

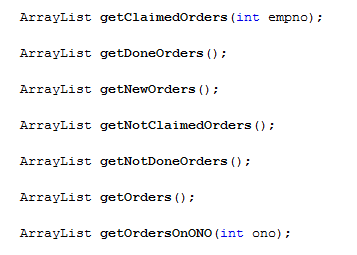
* Det er nemt at begynde at arbejde med, og forstå hvordan det allerede er sat op hvis man kender til 3 lags arkitekturen.
* Hvis man skal vedligeholde noget fra et lag behøver man nødvendigvis ikke pille hvad alt andet i de andre lag.
* Genbrugsmulighed ved de forskellige lag.
* Det er nemmere at arbejde flere på samme projekt ved 3 lags arkitekturen end ved at bare at lave det på 3 forskellige måder, da 3 lags arkitekturen har nogle faste rammer for hvordan det fungere.
* 3 lags arkitektur er også kendt for at være en arkitektur med lav kobling og høj samhørighed der som tidligere sagt gør det nemt at vedligeholde eller helt udskifte dele af programmet.

### Design-mønstre

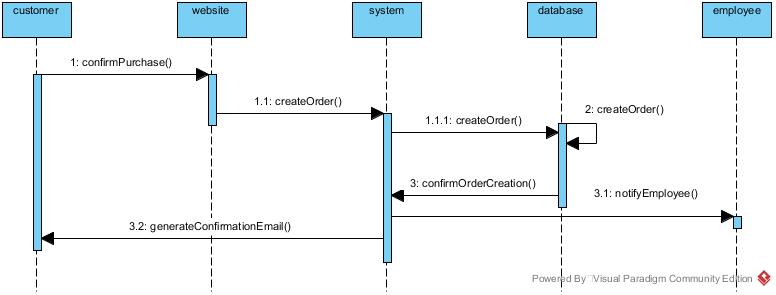
”The interface lets you take greater advantage of polymorphism in your designs, which in turn helps you make your software more flexible.” - Bill Venners, JavaWorld[[1]](#footnote-1)

Der er mange gode grunde til hvorfor man skal bruge noget som for eksempel interface, en af de grunde er at det bliver nemmere og skulle ind og få et overblik over hvad man allerede har af metoder. En anden ting er også det bliver nemmere at skifte motoren bagved eller bare ændre nogen småting, det er også lav kobling som vi snakkede om tidligere og som er en god ting at havde med.

I vores projekt har vi brugt interface til vores database del, dette er med til at gøre det nemmere hvis man pludselig finder ud af at man vil væk fra den database type vi bruger. Når vi bruger interfacet så ved de kommende udviklere lige præcis hvilke metoder der skal være der og hvad de skal returnere og hvad de skal hedde og skal havde af input. Det gør man ikke skal bruge den første halvdel af tiden på at forsøge at læse koden og forstå den, når den alligevel ikke skal bruges mere.

  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Det her er en sammenhængende del af vores kode, vi kan se her at de alle sammen returner en ArrayList, og så også se deres navne. Vi kan samtidig se at ’getOrdersOnONO(int ono)’ og i ’getClaimedOrders(int empno)’ (ono = order number, empno = employ number) metoderne skal havde en int ind. I den første ”getOrdersOnONO” skal vi havde havde en order tilbage specifikt på et order nummer. Hvor i ”getClaimedOrders” skal vi havde alle de ordrer som den givne medarbejder har ansvaret for. hvor i de andre skal vi bare havde alle ordrer ud af en specifik type. Denne type kender vi allerede så derfor skal vi ikke sende en variable med ind til metoderne. Dette er et godt eksempel på hvordan vi har brugt det.

### Sekvensdiagram



Vores simplificerede sekvens diagram illustrerer hvordan et ordreforløb vil foregå for kunden. Grunden til at vi har valgt at simplificere det er den rå mængde klasser der bliver kaldt i et realistisk forløb. Sekvens diagrammet forudsætter at kunden allerede er logget ind på en profil hos Fog. Når kunden bekræfter sit ønske om at købe en carport vil det først blive logget i den webapplikation hvor kunden har designet sin carport. De data vil derefter blive sendt videre til selve systemet hvor de relevante data vil blive omdannet til et ordre objekt der så bliver videresendt til databasen hvor ordren bliver oprettet og kædet sammen med den relevante kunde. Når systemet får bekræftet at ordren er oprettet vil det(via admin panel) informere en sælger om at der er en ny mulig køber, og vil efterfølgende informere kunden om at ordren er modtaget(NYI).

1. http://www.javaworld.com/article/2076841/core-java/designing-with-interfaces.html [↑](#footnote-ref-1)