

*"I alle led af værdikæden hos Danish Crown er det mad, der er fokus på. Det er mad, vi producerer, og det er mad, vi sælger til kunder og forbrugere verden over."*

Danish Crown

## **Indholdsfortegnelse**

Intern struktur i Danish Crown.....	5
Organisationsformer.....	6
Globalt Overblik .....	8
Mission og Vision.....	9
Vision: .....	9
Mission:.....	10
Værdikæde Danish Crown Horsens .....	10
Indkøb: .....	10
Produktion: .....	10
Markedsføring og salg:.....	11
Service:.....	11
Porters 5 forces .....	11
Truslen fra potentielle indtrængere: .....	11
Truslen fra substituerende produkter: .....	11
Købernes forhandlingsmagt: .....	12
Leverandørernes forhandlingsmagt: .....	12
Konkurrencesituationen i branchen: .....	12
SWOT .....	12
Produktion.....	13
Logistik.....	14
Rige billeder .....	15
Manuel pakning af kasser: .....	15
Manuel pakning af juletræer: .....	16
Udlæsning:.....	16
Business Case .....	17
1. Løsningsbeskrivelse: .....	17
1.1 Forretningsmæssigt omfang .....	17
1.1.1 Kort beskrivelse af løsningen.....	17
1.1.2 Løsningens formål og succeskriterier .....	17
1.1.3 Forretningsmæssig problemstilling.....	17
1.1.4 Berørte brugere.....	17
1.1.5 Eventuelle lovgivningsmæssige forhold.....	18
1.2 IT omfang.....	18

1.3 Interessenter.....	18
1.4 Konsekvens af løsning.....	18
1.4.1 Cost/benefit .....	18
1.5 Risici.....	19
1.6 Uddannelsesplan.....	19
Delkonklusion .....	19
Systemanalyse .....	20
Resultat af inceptionsfasen:.....	20
1. Systemvision .....	20
2. Kravliste.....	20
3. Usecases.....	21
3.1. Usecase diagrammet.....	21
3.2. Aktørbeskrivelser .....	21
3.3. Use case beskrivelser .....	22
3.4. Sporbarhedsmatrix .....	23
4. Ordbog .....	24
7. Systemets betydning .....	24
Resultat af Elaboration.....	24
1. Krav.....	24
2. Use case model.....	24
3. Analysemodel.....	25
3.2. Klassediagram .....	25
3.4. Interaktionsdiagrammer .....	26
4. Designmodel .....	27
4.1. Arkitektur .....	27
4.3. Klassebeskrivelser .....	29
4.4. Detaljerede sekvensdiagrammer .....	34
4. Brugergrænseflade.....	36
Resultat af Construction .....	36
6. Testrapport.....	36
7. Status.....	37
2. Refleksioner 2.1. Unified Process .....	38
2.2. Udviklingsværktøjer .....	38
2.3. Øvrige forhold.....	38

Klassemodel:.....	38
Order:.....	38
PartialOrder:.....	39
Truck:.....	39
PackageType:.....	41
Ramp:.....	42
Loading: .....	44
TruckRegister: .....	45
Collection typer .....	47
Anvendelse af polymorfi: .....	48
Design Patterns:.....	50
Strategy Pattern .....	50
Singleton Pattern .....	50
Observer Pattern.....	51
Arkitektur .....	51
Fejlhåndtering.....	53
Test:.....	53
Specielt interessant kode .....	54
Guided Tour:.....	57
Register Arrival: .....	57
Register Departure: .....	59
Konklusion.....	61
Kildeliste.....	62

Danish Crown er en virksomhed i rivende udvikling, som verdens største eksportør af svinekød, står de i spidsen som ambassadør både inden for logistik, produktion og eksportering af varer ikke bare i Danmark, men over hele verdenen.

Vi vil igennem opgaven belyse, hvordan Danish Crown er opbygget som virksomhed startende fra det globale perspektiv helt ned til slagteriet i Horsens, hvor vores primære fokus vil være at se nærmere på logistikken og produktionen af 20.000 slagtesvin om dagen, og hvordan det kan optimeres.

## **Intern struktur i Danish Crown**

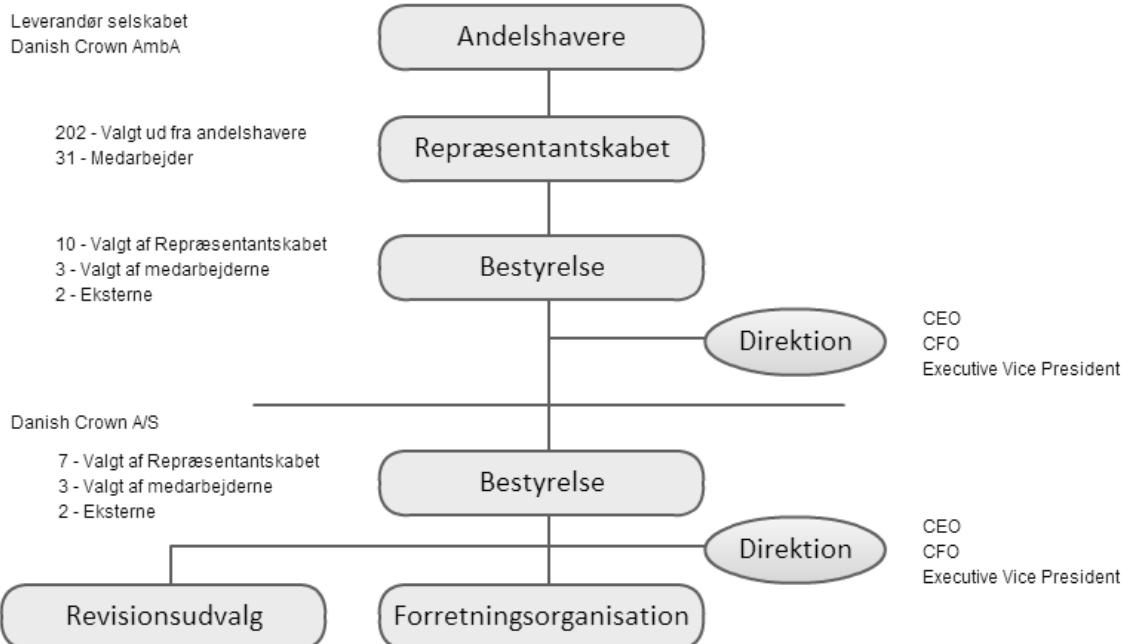
Danish Crown er dannet i 1960'erne, hvor de enkelte danske andelsslagterier begyndte at fusionere sammen for at skabe en fælles linje for produktudvikling og markedsføring. Dvs. at Danish Crown ejes af andelshavere, som er de svine og kreatur producenter, nærmere bestemt de landmænd som leverer til Danish Crown og opfylder kravet om at levere 5/6 af deres produktion til Danish Crown.

Danish Crown er delt i to; en andelsvirksomhed og et aktieselskab. Andelsvirksomheden består af andelshavere, repræsentantskabet og en bestyrelse. Bestyrelsen står med det primære ansvar for udtænkning af strategien og udstikker rammerne for den daglige ledelse af aktieselskabet Danish Crown - dog skal større beslutninger såsom radikale strategiændringer eller lignende godkendes i repræsentantskabet. Selve udførelsen af strategien foregår gennem direktionen, som består af toplederne i aktieselskabet Danish Crown.

Bestyrelsen i Danish Crown består af 3 medarbejdere, 10 andelshavere og 2 eksterne medlemmer som er valgt af repræsentantskabet. Repræsentantskabet består af 202 andelshavere, hvor 10 udvælges til bestyrelsen, derudover sidder der 31 medarbejdervalgte, hvoraf 3 udpeges til bestyrelsen.

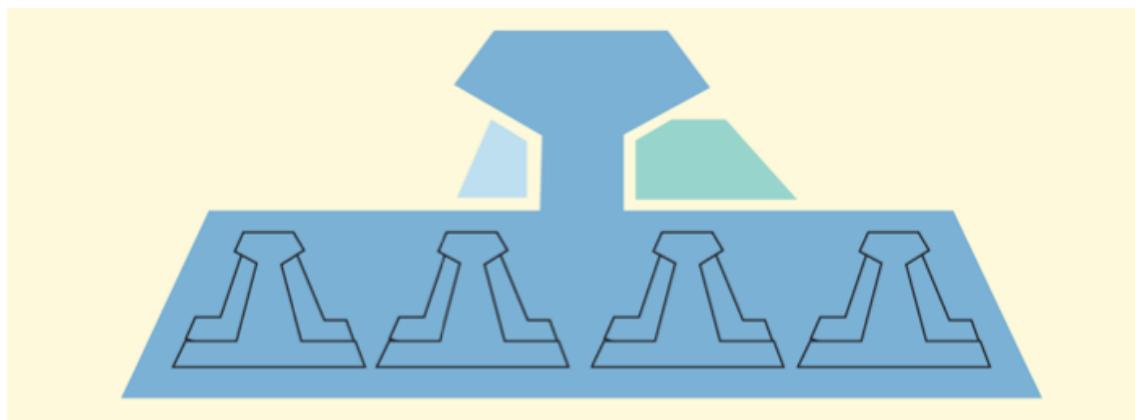
Alle 233 medlemmer er med til, at vælge de 2 eksterne som udpeges til bestyrelsen. Hovedansvaret for ansættelse af den administrerende direktør, koncernøkonomidirektør og koncerndirektør, som udgør direktionen hviler hos bestyrelsen.

Den anden del er aktieselskabet, hvor andelsvirksomheden Danish Crown er hovedaktionær og eneste aktionær. Den daglige ledelse foregår med grundlag i strategien gennem den valgte direktion, nærmere bestemt koordineringen af 10.500 medarbejdere, som er ansat i moderselskabet Danish Crown, bliver dirigeret fra hovedsædet i Randers ligesom mange af underafdelingerne, som tilhører Danish Crown også bliver dirigeret der.



Strukturen er her vist i et organisationsdiagram, hvor der dog skal tilføjes at siden 2011/12 har bestyrelsen i Danish Crown AmbA og Danish Crown A/S været ens, hvilket også gør sig gældende for direktionen.

### Organisationsformer

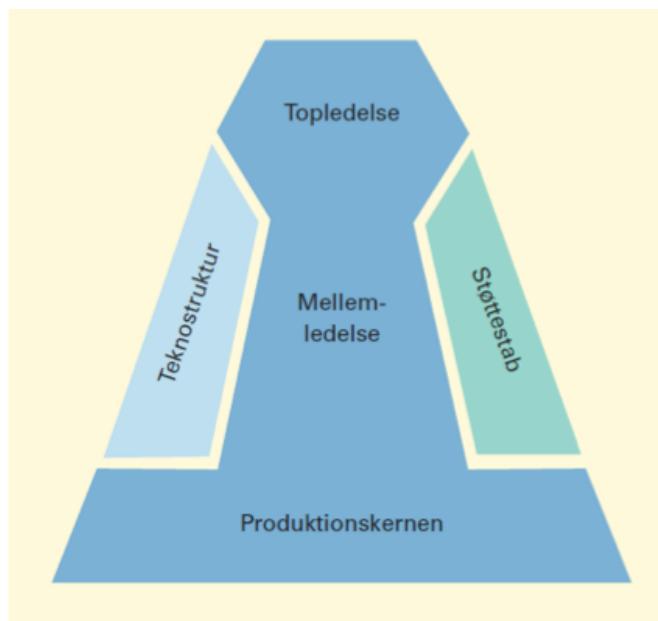


Hvis vi overordnet ser på hele organisation Danish Crown ud fra Mintzberg organisationsformer, arbejder vi med "Den divisionaliserede form", hvor repræsentantskabet sammen med bestyrelsen har primær fokus på at skabe det strategiske grundlag for hele koncernen, nærmere bestemt udstikke overordnede retningslinjer for koordinering, arbejdsdeling, personalepolitik og ressourceallokering på overordnet koncernniveau. Denne opdeling af ressourcer foregår ud fra nøgletal, som samles fra forskellige afdelinger i virksomheden. Dette kan f.eks. være salgsprognoser, produktions antal for hvert enkelt slagteri osv. De grundlæggende fælles værdier og fælles mål skabes her samt dannelsen/udbygningen af vision og mission.

Teknostrukturen i den overordnet koncern opdeling, vil være den logistiske problemstilling i at modtage information omkring afhentning af svin og dermed beslutte hvilket slagteri, svinene skal transporteret til. Alt dette foregår fra hovedkontoret i Randers. Så grundlæggende foregår der en tracking af svinene gennem hele processen, fra afhentning hos den enkelte landmand, til slagningen er fuldført og sendes videre til køber.

En ganske essentiel støttestab er direktionen, som består af primære driftsledere fra hele koncernen. Deres opgave er at støtte bestyrelsen, både med information om de aktuelle problemstillinger på enkelt koncern niveau, samtidig med at udføre den konkrete strategi. Dette gøres for at skabe et overblik og holde hvert enkelt slagteri som en enhed for sig selv, der så kan tilpasses i forhold til egen drifts og funktionsevne. Denne opsætning giver mulighed for at de enkelte stabsfunktioner, såvel som ledelsesfunktioner, kan udvikle sig uafhængigt af de enkelte slagterier og hinanden. Nederst i hierarkiet er placeret hvert enkelt slagteri, som en enhed for sig, med sin egen organisationsform. I denne opgave har vi valgt at se nærmere på Horsens slagteri ud fra samme model.

Modellen, som danner grundlag for Horsens slagteri, er maskinbureaucratiet - her skal dog tilføjes at selve opdelingen i virksomheden er baseret ud fra en antagelse, da vi ikke har information om den konkrete struktur. Topledelsen består af en fabrikschef og nogle daglige ledere, som står for at igangsætte de primære opgaver og kommunikere med hovedkontoret i Randers. Mellemledelsen består af gulvfolk, dvs. ledere som står med ansvaret for de enkelte afdelinger i produktionen og de dertilhørende produktionsarbejdere. Produktionskernen er så de mennesker som arbejder med produktionen af svinekød. Teknostrukturen er et ERP system defineret fra koncernen, hvor der sker samlet beregninger imellem alle slagterier, så tal og informationer er fuldt opdateret. Systemet er sammenkoblet gennem produktionsovervågning, som sørger for at holde styr på de enkelte svin gennem hele produktionsprocessen. Støttestaben er et centralt lønningskontor, kantiner og en mindre logistikafdeling, til intern logistik, for at kunne overskue de 20.000 svin.



Maskinbureauratiet gør brug af, at standardisere de enkelte arbejdsstationer og gøre dem rutinemæssige, så man opnår en specialisering af produktionsarbejdere, her skal dog nævnes, at produktionsarbejderne roterer mellem forskellige stationer for at opnå en bred specialisering. Udenfor dette skaber modellen en klar struktur gennem planlægning, regler og klare systemer, så hele produktion følger en klar linje, hvilket gør, at optimeringsprocessen bliver sat i fokus. Dette kendetegner generel masseproduktion, da standardisering af produkter, giver mulighed for øget fokus på optimering og effektivitet. Produktionsopgaven anses for enkel, hvor optimering betragtes som kompleks.

Gennem mintzberg modellerne gør det sig gældende, at det er et linje-stabsprincip, idet at specifikke hjælpe funktioner bliver brugt som en stabsfunktion, der kan kommunikere ud til de enkelte delledere, som så kan opnå aflastning, men kan også skabe en uklar ansvarsplacering. Her forsøger Danish Crown dog, at skabe et overblik ved at lade mellemlederen være specialist inden for det specifikke område, som han/hun leder. Deraf holder strukturen sig stadig rimelig overskuelig, så man undgår kommunikationsproblemer.

## Globalt Overblik

(mio. kr.)	Resultatopgørelse				
	2007/08	2008/09	2009/10	2010/11	2011/12
<b>Nettoomsætning</b>	46.972	44.757	45.211	51.754	56.462
<b>Bruttoresultat</b>	7.005	6.591	6.598	7.182	7.337
<b>Årets resultat</b>	1.020	1.164	1.648	1.762	1.732

Ved at kigge på et lille udkast af resultatopgørelsen kan man se, at Danish Crown virker til at have en stabil rentabilitet. Når man kigger på tabellen, kan man se, at årets resultat har en stor stigning fra 08/09 til 09/10. Dette skete hovedsageligt på et par områder.

Indtægter af kapitalandele i associerede virksomheder steg med 189 mio. kr., og finansielle omkostninger faldt med 180 mio. kr. Den resterende besparelse var at finde under distributionsomkostninger, hvor man her kunne se et fald i udgifterne på 137 mio. kr.<sup>1</sup> Nettoomsætningens stigning fra 45.211 mio. kr. til 56.462 mio. kr. (ca 25%) var en imponerende vækst. Dette skyldes specielt, at det har været muligt at øge aktiviteterne i udlandet hurtigere end reduktionen i det danske råvaregrundlag. Dette har specielt været at se hos ESS-FOOD, som er et datterselskab, hvor de har haft stor fremgang på det kinesiske marked<sup>2</sup>.

Danish Crown er en stor spiller på det internationale svinekødsmarked med en eksport omsætning på 24 mia. om året, hvilket svarer til omkring 4 %<sup>3</sup> af den samlede danske eksport. Danish Crown er Europas største svinekødsproducent, da de årligt slagter omkring 22 mio. svin.<sup>4</sup> På verdensplan er de kun overgået af Smithfield, som årligt slagter omkring 28 mio. svin.<sup>5</sup> Dette gør Smithfield til Danish Crowns største konkurrent på svinekødsmarkedet. Omsætningen hos Smithfield er på ca. 13 mia. US\$<sup>6</sup>(ca. 74 mia. kr) i sammenligning med Danish Crowns omsætning i regnskabsåret 11/12 var på 56 mia. kr<sup>7</sup>. 90 %<sup>8</sup> af Danish Crowns omsætning er eksport, hvorimod Smithfield kun eksporterer 24 %<sup>9</sup>. På det danske marked er den største konkurrent Tican, som har en omsætning på 4,7 mia. kr<sup>10</sup>, hvor deres produkter består af alt inden for svinekød. Dette er som sådan den eneste konkurrent på det danske marked, da Danish Crown er så store, at de ikke skaber meget råde plads til andre virksomheder.

## Mission og Vision

Mission og vision er eksistensgrundlaget for en virksomhed, dvs. det er måden virksomheden definere hvilke mål, den skal opfylde og måden hvorpå, den har tænkt at opnå de mål.

*Vision:*

"Danish Crown skal være det førende europæiske selskab inden for kødområdet og skal ved en konkurrencedygtig international afsætning sikre danske svine- og oksekødsproducenter en afgørende og lønsom position på verdensmarkedet".

Danish Crowns langsigtede plan er at sikre, at andelshavere står som et afgørende element i det europæiske marked for kød, samtidig med at den position skal gøre dem godt lønnet. Dette vil blive opfyldt når Danish Crown står som det førende europæiske selskab.

---

<sup>1</sup> Danis Crown årsrapport 9/10

<sup>2</sup> Danis Crown årsrapport 11/12

<sup>3</sup> danishcrown.dk

<sup>4</sup> Danis Crown årsrapport 11/12

<sup>5</sup> Smithfield Årsrapport

<sup>6</sup> Smithfield Årsrapport

<sup>7</sup> Danis Crown årsrapport 11/12 & danishcrown.dk

<sup>8</sup> danishcrown.dk

<sup>9</sup> Smithfield årsrapport

<sup>10</sup> tican.dk

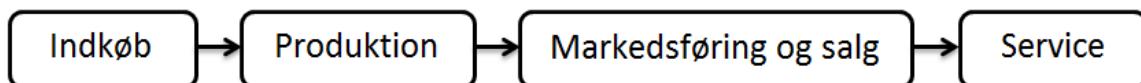
### *Mission:*

"At levere produkter, der opfylder forbrugernes krav til differentieret kvalitet, og dermed sikre vore ejere den højest mulige pris for deres råvarer".

Mission uddyber måden på hvordan Danish Crown skal blive det førende selskab i europa, dette sker ved at levere differentieret kvalitet. Altså at ramme markederne så bredt så muligt inden for kvalitet. Her skal nævnes at virksomheden selvfølgelig bestræber sig på at levere god kvalitet, men at de også levere produkter i mindre god kvalitet for at ramme markeder, der sætter pris foran kvalitet.

Danish Crowns vision og mission bærer tydeligt præg af, at det er en andelsvirksomhed, da ejerne er det vigtigste for virksomheden. De fleste virksomheder eksisterer for at generere penge til ejerne, hvilket både nævnes i visionen og missionen, som umiddelbart kan få andelshaverne til at virke usympatiske og griske. Derudover virker det også svært for den enkelte medarbejder at bruge missionen og visionen til noget konstruktivt, da de begge er meget flydende.

### **Værdikæde Danish Crown Horsens**



Kigger vi på værdikæden for Danish Crowns slagteri i Horsens, er den opbygget som vist på figuren ovenfor.

### *Indkøb:*

På slagteriet i Horsens ankommer hver dag 100 lastbiler med omkring 200 svin i hver. Svinene bliver leveret fra Danish Crowns leverandører inden for en transporttid på 2 timer af slagteriet i Horsens. Landmændene skal overholde Code of Practise, som er fastsat af Danish Crown, og da de er andelsejere, har de stor interesse i, at få det bedste produkt igennem. Danish Crown er forpligtet til at tage imod alle dyr som landmændene kunne ønske at komme af med uanset dyreart. Dette er en af fordelene, som landmændene har ved, at være en del af Danish Crowns leverandører.

### *Produktion:*

Mange af produkterne som bliver forarbejdet på slagteriet i Horsens, er solgt allerede før grisene ankommer til slagteriet, hvilket betyder at kunden selv har bestemt på hvilken måde kødet skal udskæres. I alle afdelingerne på slagteriet foregår arbejdsgangen i hold, hvor man efter en kort periode skifter arbejdsstation. Der bliver skiftet arbejdsstation i de enkelte afdelinger af den årsag at man vil forhindre de ansatte i at blive for hurtigt nedslidte. Denne rotation er påkrævet fra arbejdstilsynet.

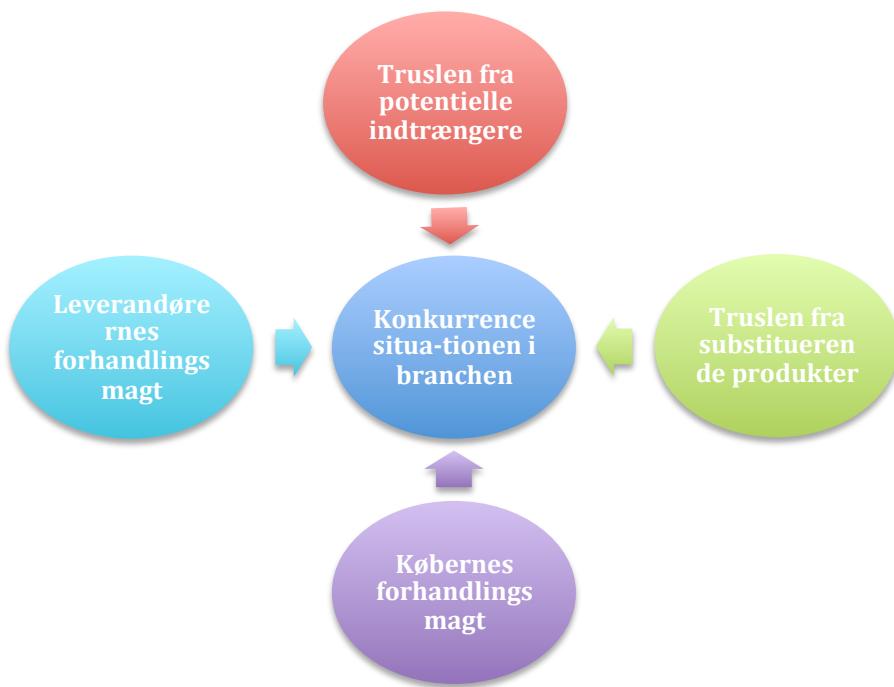
## *Markedsføring og salg:*

Danish Crowns slagteri i Horsens bliver ikke markedsført, ligesom koncernen ikke har noget markedsføringsbudget. Det er blevet besluttet i koncernen, at de satser på markedsføring gennem offentlige rundvisninger samt et samarbejde med en række danske kokke og restauranter, som kan få leveret kød inden for 24 timer uanset hvor de befinder sig i verden. Størstedelen af produktion i Horsens er solgt på forhånd, hvilket også er grunden at markedsføringen ikke er et fokus.

## *Service:*

Danish Crowns slagteri i Horsens forventer at en vare kan være klar til levering, indenfor 1-2 dage efter bestillingen er kommet til Danish Crown. Det er muligt, at vælge en individuel udskæring. Danish Crown har faste leveringstider til hver lokationszone.

## **Porters 5 forces**



### *Truslen fra potentielle indtrængere:*

Der er ikke den store sandsynlighed for at der kommer nye konkurrenter til Danish Crown, da de har mange af de danske svineproducenter som leverandører. De svineproducenter som leverer til Danish Crown skal leve 5/6 af deres produktion til Danish Crown. Dette bevirker, at de vil have svært ved at vælge et andet slagteri.

### *Truslen fra substituerende produkter:*

Et produkt som potentielt vil kunne konkurrere med Danish Crowns svinekød på enkelte markeder, er kunstigt kød, som forskere fra USA, Holland og Australien er i gang med at forske i.

*Købernes forhandlingsmagt:*

Dele af Danish Crowns kunder har høje krav til deres produkter, eks. Japan, som har en kontrollør i Danmark, som rejser rundt mellem de forskellige svinekødsslakterier i Danmark. England har også høje krav til den bacon, som skal leveres til dem. Dog er det ikke alle Danish Crowns kunder, som har den rette størrelse til at kunne stille krav ved forhandlinger.

*Leverandørernes forhandlingsmagt:*

De leverandøre som leverer svin til Danish Crown er samtidig også ejere, hvilket bevirker at leverandørene også ønsker at virksomheden har den størst mulige indkomst.

*Konkurrencesituationen i branchen:*

Danish Crowns eksport står stærkt på markedet. Størstedelen af eksporten består af svin, hvilket er en mangelvare nogle steder i verden, blandt andet i Asien, hvor de ser en stor fremgang hos datterselskabet ESS-FOODS.

## SWOT

Styrker	Svagheder
Leverandørerne ejer virksomheden Stor markedsandel nationalt og internationalt. Standardisering af produkter Effektiv produktion	Ændring af optimeringsprocessen Høje krav fra udenlandske købere Grundlønnen er høj i Danmark
Muligheder	Trusler
Store muligheder på det asiatiske marked	Faldende pris på svinekød Svinesygdom

SWOT analysen giver et overblik over, hvor stærkt Danish Crown står på det danske og europæiske marked. Deres store fokus er på optimeringsprocessen i forhold til slagningen af dyr, hvilket på sin vis kan ses både som en styrke og en svaghed, da denne process kan ændring sig fra den ene dag til den anden. Den eneste reelle trussel, som vi ser vigtig, vil være prisen på svinekød, da alt produktion afhænger, hvor stor en fortjeneste de har for hvert enkelt produkt.

Afspejlingen af mission og vision versus SWOT analysen viser, at vi næsten direkte har at gøre med de styrker som er primær fokus i analysen, samt at der bliver taget højde for alle trusler og svagheder.

## Produktion

Ved ankomst til Danish Crown i Horsens bliver dyrene tjekket af en ansat samt en ekstern dyrlæge fra fødevareregionen. Dette ser Danish Crown som et plus, da man her sikrer en høj kvalitet. Dyrlægen tjekker hvert svin for stress og i tilfælde af, at et svin er stresset, bliver det placeret i en bås sammen med 2 andre svin den kender. Der er stor fokus på at få så rolige svin som muligt, da det giver det bedste slutresultat. Alle godkendte svin, som ikke er stressede, bliver placeret i båse med plads til 12-16 svin. Dette gøres fordi svin er flokdyr og bliver derfor mere rolige jo tættere på hinanden de bliver placeret. Alle svin bliver placeret i stalden i 1-2 timer, hvorefter de bliver sendt til den sorte slagtegang.

På den sorte slagtegang ankommer svinene i grupper af 12-16 og bliver her delt i 2 mindre grupper. Transporten fra stalden til den sorte slagtegang foregår i en gang, som går opad, hvilket er noget der gør svinene mere rolige. Hver gruppe svin bliver herefter sänket et stykke under jorden og bedøvet med kuldioxid i 3 minutter. Herefter bliver svinene hævet igen, og slagterne har 3 minutter til, at tømme svinene for blod og derefter sende igennem en skoldingsproces. Derefter bliver svinene registreret, med information omkring gård, og gennem en maskine som kaldes en autofom måles kvaliteten af kødet, som registreres sammen med vægten. Svinene skæres op og indvoldene tages ud.

Hos Danish Crown findes der et apparat, som kaldes en leverudtrækker. Dette apparat kan, som navnet siger, trække leveren fra de andre organer. Efter en kort testperiode blev det konstateret, at leverudtrækkeren skabte mere spildprodukt end ved manuelt arbejde, og den blev derefter koblet af systemet.

Efter svinene har været igennem den sorte og den rene slagtegang, bliver de sendt til afkøling ved 5° op til 16 timer, hvorefter svinene bliver skåret ud til mindre dele. Disse produkter kan pakkes på 3 måder; juletrær, plastkasser eller papkasser. Juletrær sendest typisk til kunder, som skal bruge det som råvare til videre produktion, plastkasser er som oftest til Danish Crown eller Tulip og papkasser er et færdigt produkt.

Lean produktion er ifølge Danish Crown opfundet af dem. De var det første firma, som havde deres ansatte til at rotøre mellem arbejdsstationer. I produktionen arbejdes der i 67 minutter før man holder en pause, men i løbet af denne tid skifter man arbejdsstation. I hele produktionsanlægget arbejdes der på akkord, men det er forskelligt hvor meget man kan tjene afhængigt af hvad man skal lave. Man roterer pladser i anlægget for at mindske slid på produktionsarbejderne. Udover dette anvender Danish Crown "Just In Time" princippet da svinene bliver nødt til at slagtes, efter de kommer ind ad døren, både for at undgå spild, men også for at kunne varetage 20.000 slagtesvin om dagen. "Just In Time" principippet er baseret på pull-principippet, hvilket betyder, at det kunden venter på at produktet er færdigt frem for at få det leveret direkte fra lageret.

Danish Crown anvender flow produktion med en blanding af masseproduktion og specialisering. Dette sikrer mulighed for levering til mange forskellige målgrupper. Kundens specielle ønsker bruges som viden til, hvordan man bedst muligt kan skabe en høj kvalitet og produktion.

## **Logistik**

Der er hos Danish Crown stor fokus på optimering. Dette gør, at der er stort pres på logistikafdelingen. Disse folk skal sørge for, at Danish Crown får det ønskede antal vogne og tjekke deres gps position for at have en præcis ankomst. Der kan kun være 4-5 lastbiler ad gangen ved indlevering, så det hele skal times, så man ikke får en flaskehals. Selve salgsordrerne sker fra hovedkontoret i Randers, men logistikafdelingen i Horsens står selv med koordinationen af den interne produktion, og i tilfælde af et stop, skal der her arbejdes hurtigt for ikke at løbe tør for plads.

Alt produktion som ender i en hvid kasse kan spores. Alle kasser hos Danish Crown har et tracking system som går 7 dage tilbage, hvor man præcist kan se hvad kassen indeholder på et ønsket tidspunkt.

Hos Danish Crown har man derfor brug for nogle midlertidige opbevaringspladser. Disse kaldes bufferlageret og kaoslageret. Bufferlageret samler ordren, som så primært bliver pakket på paller af et robotsystem. Kaoslageret er inddelt i 3 zoner, hvor man på skift rydder op i hver zone. Et produkt kan maksimalt være i kaoslageret i 7 dage inden det skal være solgt eller sendt til destruktion. Man har været nødsaget til at have et interval, hvor man ryddede dele af lageret, da man ellers ville ende i problemer, hvis en kasse mistede dens tracker. Alt produktion som gemmes på kaos lageret er i risiko for at være et tab af fortjeneste. Den typiske vare som bliver placeret på kaoslageret er varer der ikke er solgt. Det er logistikafdelingen, som skal sørge for, at disse varer hurtigst muligt blive solgt videre og skaber den ønskede indtægt.

Koncernens salg er baseret på kontrakt om mængder og leveringstidspunkter. Danish Crown har stor fokus på overholdelse af aftaler. Produktionen er sårbar overfor vand og el afbrydelser, samt nedbrud i it-systemer. De fleste produkter Danish Crown har, kan produceres på mere end ét anlæg, hvilket er noget som tages i brug, hvis der skulle være et nedbrud. I tilfælde af problemer hos Horsens, har man også mulighed for, at sende noget af produktionen videre til et andet slagteri. Dette er med til at skabe en produktionssikkerhed, og derved gøre det muligt at leve op til deres leveringsaftaler.

For at blive leverandør hos Danish Crown skal man følge en række kriterier:

1. Respektere og understøtte UN Global Compact
2. Har implementeret et fødevaresikkerhedssystem
3. Løbende reducerer negativ miljøpåvirkning
4. Fastholder et vedvarende højt kvalitetsniveau
5. Opretholder en vedvarende høj leveringsevne

Pålæsning af lastbiler sker manuelt. Når en lastbil ankommer til Danish Crown afleveres lastbilens trailer, hvorefter den bliver kørt til rampen af Danish Crowns personale når de har tid. Det er vigtigt, at lastbilen som skal transportere varen er 100% rengjort. Skulle der ved læsning af f. eks. juletræer ske det, at et stykke kød falder af og lander i selve bilen, vil dette ikke være et problem, da bilen er ren og godkendt. Hver vogn har en indvejning og en udvejning, så man kan se, hvor meget der er kommet i lastbilen og

derved kan opfylde ordrerne korrekt. Nogle gange vil der være en ordre, som ikke fylder en hel vogn. Her planlægges så en lastbil får mere end 1 ordre for, at spare penge på distributionsomkostninger. Denne lastbil vil typisk blive kørt til et lokalt sted i nærheden af leveringsadresserne, hvor lastbilen her vil blive omlæsset.

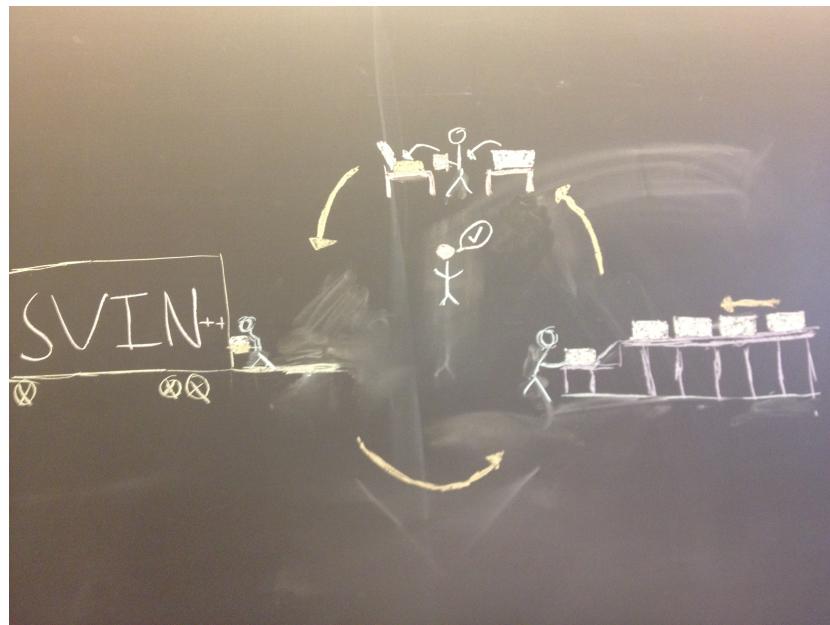
Et af de større planlægningsproblemer Danish Crown har er, at lastbilerne ikke ankommer på de tidspunkter der er blevet oplyst. Dette skaber en masse løbende planlægningsbeslutninger.

For at blive leverandør til Danish Crown skal man have en høj vedvarende leveringsevne, hvilket gør, at vi kan antage, at Danish Crown har en god leveringsservice. Virksomheden har faste rammer for deres leveringstider. Leveringstiden, som først beregnes efter produktion, til England og det sydlige Europa er 36 timer, mens resten af Europa skal kunne klares på 24 timer. Slagteriet i Horsens har en formustring leveringsfleksibilitet, da det er muligt, at få hjælp fra andre slagterier.

Danish Crown bruger manuelt arbejde til at læsse lastbiler. Der er stor fokus på optimering i firmaet, hvilket har gjort, at der er blevet indsat et robotsystem til at pakke pallerne, men det har endnu ikke været muligt at finde et forbedret alternativ til det manuelle arbejde.

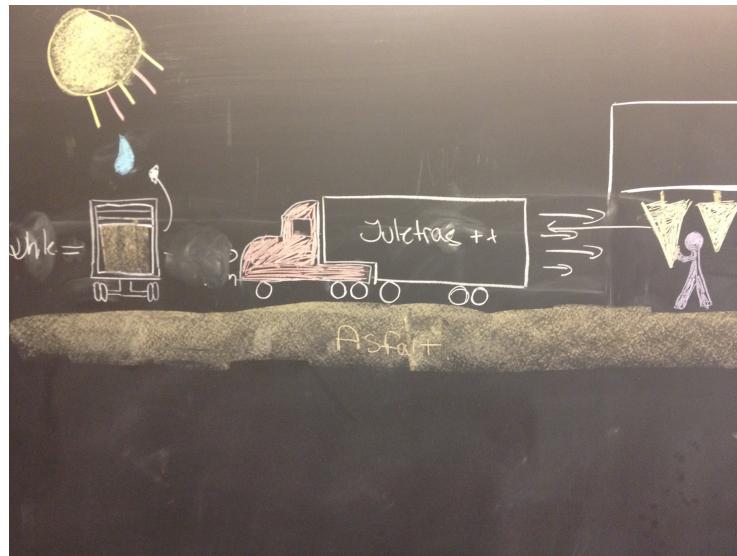
## Rige billeder

*Manuel pakning af kasser:*



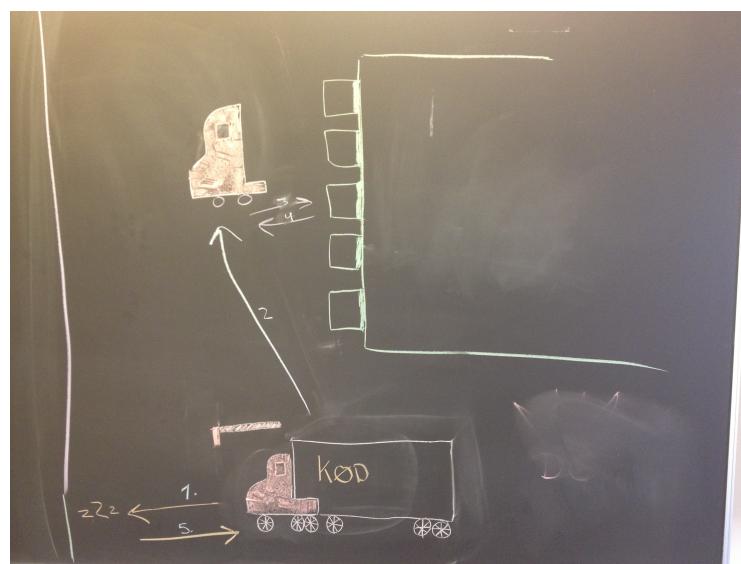
Enkelte kunder kræver, at produkterne bliver pakket i specialfremstillede kasser, hvor bl.a. Japan har krav til, hvordan kødet bliver placeret. Da vi arbejder med specialfremstillede kasser, er det svært at tilføje et robotsystem til pakning.

*Manuel pakning af juletræer:*



Juletræer hænger på en skinne, og bliver skubbet langs skinnen ind i lastbilen. Inden dette sker skal lastbilen være rengjort og godkendt, hvilket også gør, at selvom et stykke skulle falde på af, vil kødet stadig være godkendt, da lastbilen er ren.

*Udlæsning:*



Ved ankomst afleverer lastbilchaufføren traileren til Danish Crown og kører til et hvileområde, hvor de her kan slappe af indtil vognen er læsset. Her er der logistisk planlagt, hvornår hver vogn skal læsses, da det er praktisk, at lastbilchauffører, som skal langt, holder deres hvilepause inden de kører. Herefter overtager Danish Crown flytning af trailer til og fra rampen, så de bliver læsset i den ønskede rækkefølge. Selve logistikken bag flytning af trailere, er overvejelser omkring hvor den enkelte vogn skal hen, dvs. at en vogn kan indeholde ordre fra flere kunder hvis de er placeret i samme

område.

Vi vil gennem resten af opgaven forsøge at finde en løsning på de logistiske problemstillinger og sammenfatte det i en business case model.

## **Business Case**

### *1. Løsningsbeskrivelse:*

#### *1.1 Forretningsmæssigt omfang*

##### *1.1.1 Kort beskrivelse af løsningen*

At varetage læsning af trailere med forbehold for chauffør hviletider, varesortering, trailer håndtering og tid. Dette foregår f.eks. ved at melde til den enkelte chauffør, når den trailer han skal kører med er klar.

##### *1.1.2 Løsningens formål og succeskriterier*

Forbedre servicekvalitet (f.eks. hurtigere sagsbehandlingstid)	X
Effektivisere forretningsprocesser eller reducere driftsomkostninger (f.eks. automatisering af processer)	X
Effektivisere it-applikationer (f.eks. udskiftning af gammel applikationslogik eller integrere applikationer)	
Effektivisere it-infrastruktur (f.eks. konsolidering af servere, reducering af antal platforme)	
Forbedre fleksibilitet i forretningsarkitektur (f.eks. understøttelse af standardiserede processer)	X
Forbedre fleksibilitet i it-arkitektur (f.eks. indføring af åbne standarder eller implementering af SOA)	
Opfylde lovgivning (hvis projektet sikrer opfyldelse af lovgivning (national eller international))	

##### *1.1.3 Forretningsmæssig problemstilling*

Den nuværende process mangler en stor del logistisk sammenhæng, da varetagelsen af trailere mellem forskellige udlæsnings ramper såvel som det logistiske med at chaufførens hviletid skal overholdes skal tages i betragtning.

##### *1.1.4 Berørte brugere*

Chauffør og læssefolk.

### *1.1.5 Eventuelle lovgivningsmæssige forhold*

Der skal overholdes regler for hvileperioder for chauffører.

### *1.2 IT omfang*

Vi programmerer systemet i Java, hvilket er platforms uafhængigt så det kan køre på alle platforme. Samtidig skal systemet køre separat, så der ikke er nogen kobling mellem dette og SAP.

### *1.3 Interessenter*

#	Identificerede interesser (benyt én række per interessent)	Påvirket af løsning			Indflydelse på løsning			Involvering i dialog omkring løsningen (sæt ét kryds)		Beskrivelse af involvering (kun hvis der har været en involvering)
		Lav	Middel	Høj	Lav	Middel	Høj	Ja	Nej	
1	Logistikpersonale			X			X	X		Personale skal være med til test af løsning samt have muligt for at komme med forslag til ændringer
2	Lastbilchauffører	X			X				X	
3	Læsningspersonale		X			X		X		Personale vil være i løbende dialog omkring udarbejdelse af løsning

### *1.4 Konsekvens af løsning*

#### *1.4.1 Cost/benefit*

Da vi ikke besidder nøgletal fra Danish Crown har vi på baggrund af antagelse forsøgt at komme med en vurdering på hvilke steder der kan spares. Da systemet vil optimere processen betragteligt, ved f.eks at sørge for at der hele tiden er taget højde for de enkelte hviletider, og hvilken rampe en specifik trailer skal rykkes til, vil der være en stor tidsbesparelse på udlæsningen, deraf vil flere svin kunne læsseg hurtigere.

## 1.5 Risici

Risikoområde (organisation; teknisk løsning; leverandører; interessenter)	Beskrivelse af identificeret risiko	Sandsynlig- hed			Konsekvens			Håndtering af identificeret risiko
		Lav	Middel	Høj	Lav	Middel	Høj	
Teknisk løsning	Danish Crown når ikke at udvikle erfaring til brug af systemet samt vedligehold		X		X			Sende personale på kursus  Som nødløsning kan vi tilbyde at vedligeholde produktet
Leverandør	Ingen identificerede risici							
Organisation	Ingen identificerede risici							
Interessenter	Ingen identificerede risici							

## 1.6 Uddannelsesplan

Chauffører og logistik ansatte vil skulle uddannes i det nye system, for at kunne bruge systemet, der vil desuden skulle bruges tid på at få inkorporeret i den daglige rutine.

## Delkonklusion

Studering af individuelle processer i logistik og produktion skaber stor værdi for optimeringen af Danish Crown. Vi vil ud fra vores virksomhedsanalyse viderebygge Danish Crowns logistiksystem med primær fokusering på vækst og forbedret logistik og deraf forsøge at optimere Danish Crowns læsning og koordinering af lastbiler i Horsens.

## Systemanalyse

Gennem opgaven har vi nu dannet et overblik omkring, hvordan et logistik system vil kunne optimere flowet i Danish Crown omkring læsning af lastbiler.

### Resultat af inceptionsfasen:

#### 1. Systemvision

Systemets skal koordinere læsningen af lastbiler ved Danish Crowns læsseramper, på den mest effektive måde. Systemet skal reducere noget af det manuelle arbejde ved ankomst og afgang fra Danish Crown, som eks. advisering af chaufføren.

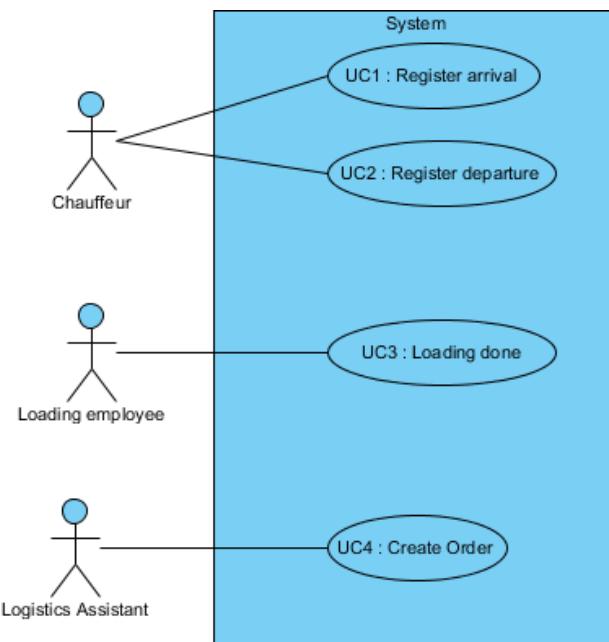
#### 2. Kravliste

ID	Beskrivelse	Prioritet
K1	Med systemet skal lastbilchaufføren kunne registrere sin ankomst.	M
K2	Med systemet skal lastbilchaufføren kunne registrere sin afgang.	M
K3	Med systemet skal man kunne knytte en eller flere delordrer til en trailer.	M
K4	Med systemet skal man kunne ændre en ordre på en trailer	S
K5	Med systemet skal man kunne avisere chaufføren, når traileren er færdiglæsset	M
K5	Med systemet skal lastbilchaufføren kunne registrere tidspunkt hvor han tidligst kan aviseres.	M
K7	Systemet skal kunne foretage en kontrolvejning ved ankomst og afgang	M
K8	Systemet skal kunne beregne starttidspunkt for læsning	M
K9	Systemet skal kunne beregne sluttidspunkt for læsning	M
K10	Systemet skal kunne vise en oversigt over hvilke typer en trailer kan indeholde	M
K11	Systemet skal kunne varetage en omlæsning	S
K12	Systemet skal kunne varetage hvilken rampe traileren skal flyttes til	M
K13	Systemet skal kunne lave en oversigt over ramperne	S
K14	Systemet skal kunne lave en oversigt over trailere og hvad de kan indeholde af delordrer	S
K15	Systemet skal kunne varetage at der er flere trailere pr. ordre	M
K16	Systemet skal kunne vise en statistik, der beregner hvor godt læsningen for en specifik rampe er gået (Om tiden er overholdt i forhold til de læsninger som har været)	W

MoSCoW modellen bruges til at vurdere hvilke krav der er de vigtigste i systemet, heraf vurdering af hvilke krav, der er Must Have for at systemet virker.

### 3. Usecases

#### 3.1. Usecase diagrammet.



Det grundlæggende i systemet er, at en chauffør kan registrere sin ankomst og afgang, hvilket ses i usecase diagrammet ovenfor. Læssemedarbejderen kan melde til systemet når en læsning er færdig og en logistikassistent kan oprette en ordre.

#### 3.2. Aktørbeskrivelser

De aktører, som interagerer med systemet, er chauffører, læssemedarbejdere og logistikassistenter.

##### *Chauffører:*

Chauffører ankommer med lastbilerne. Med hensyn til deres IT-baggrund forventes der ikke det helt store, dog skal de kunne anvende en telefon, samt registrere nogle data for at identificere, hvilke ordrer som skal med lastbilen.

##### *Læssemedarbejder:*

De ansatte ved Danish Crown i Horsens læsser ordrerne i traileren, når de ankommer til ramperne. Deres IT-baggrund skal være mere dybdegående end chaufførernes, da de skal kunne lave ændringer med systemet.

*Logistikassistent:*

Det er logistikafdelingens ansvar, at lastbilerne bliver bestilt og ankommer til de ønskede tider. Deres IT-baggrund skal sikre, at de har kompetencer til at udføre mere advancerede ændringer.

### 3.3. Use case beskrivelser

<b>Use Case Name:</b>	Register Arrival <b>ID:</b> UC1		
<b>Scenario:</b>	Chaufføren ankommer for at afhente en ordre		
<b>Triggering Event:</b>	En chauffør kommer for at hente en ordre hos Danish Crown		
<b>Brief Description:</b>	Chaufføren skal registrere sin ankomst til Danish Crown, hvor chaufføren logger alle informationer ind omkring den ordre, som skal hentes. Chaufføren vejer traileren og påfører information til ordenen.		
<b>Actors:</b>	Chauffør		
<b>Related use cases:</b>			
<b>Stakeholders:</b>			
<b>Precondition:</b>	PartialOrder og Truck eksisterer i systemet		
<b>Postcondition:</b>	Loading ordre oprettet		
<b>Flow of events:</b>	1. Chaufføren registrerer registreringsnummer, navn, telefonnummer, pakketyper, hvilketid og delordrenumre 2. Chaufføren vejer traileren og inputter lastbilens vægt	1.1 Systemet tillader adgang 2.1 Systemet gemmer lastbilens vægt i arrivalWeight 2.2 Systemet beregner forventet starttidspunkt 2.3 Systemet beregner den forventede læssetid 2.4 Systemet beregner det forventede sluttidspunkt 2.5 Systemet finder den optimale rampe baseret på den samme type 2.6 Systemet opretter en ny loading ordre.	
<b>Exceptional Flows:</b>	1.1 Hvis vægten delordre ikke ligger mellem 0 og lastbilens kapacitet, prøv igen 1.2 Hvis hviletid er under 0, prøv igen 2.1 Hvis lastbilens vægt er under 0, prøv igen		

Her har vi lavet en usecase beskrivelse, hvor vi viser forløbet ved en ankomst.

<b>Use Case Name:</b>	Register departure <b>ID:</b> UC2														
<b>Scenario:</b>															
<b>Triggering Event:</b>	Loading færdig														
<b>Brief Description:</b>	Chaufføren vejer trailer og melder afgang														
<b>Actors:</b>	Chauffør														
<b>Related use cases:</b>															
<b>Stakeholders:</b>	Danish Crown, Transportselskabet														
<b>Precondition:</b>	Trailer er læsset og chauffør har modtaget advisering														
<b>Postcondition:</b>															
<b>Flow of events:</b>	3. Chaufføren registrerer registreringsnummer, navn, telefonnummer, pakketyper og delordrenumre samt lastbilens nye vægt						3.1 Systemet godkender og sætter loadings state til done.								
<b>Exceptional Flows:</b>	1.1 Hvis vejningen ikke passer, sender systemet trailer til omlæsning, hvor der oprettes en ny loading med prioritet, og den sorteres ind på først mulige rampe														

Usecase beskrivelserne bruges til at skabe overblik omkring de to mest centrale usecases og skabe forståelse for interaktionen, som foregår med de enkelte aktører og systemet.

### 3.4. Sporbarhedsmatrix

ID	K 1	K 2	K 3	K 4	K 5	K 6	K 7	K 8	K 9	K 0	K1 1	K1 2	K1 3	K1 4	K1 5	K1 6
U 1	x		x			x	x	x	x	x		x		x		
U 2		x					x				x					
U 3					x							x				
U 4														x		

Gennem sporbarhedsmatrixen får man et godt overblik over hvor vidt man har klaret sine krav. Vi kan ved udfyldelsen se, at vi mangler krav 4 og 16. Krav 4 er, at man skal kunne

ændre en ordre på en trailer. Dette ville være fornuftigt at have, men det nåede vi ikke at løse. Krav 16 er et statistik krav, som først skal løses til version 1.2. Når vi tjekker vores kravliste kan vi med glæde se, at vi har opfyldt alle krav, som var et must. Ting som var burde eller kunne have, kan man tilføje efterfølgende.

#### 4. Ordbog

Udtryk	Definition
Truck	En lastbil bestående af et styrehus, samt en trailer
Hviletid/Rest needed	Hviletid en chauffør har brug for i minutter
Packagetype	Beskriver en pakketype
CardboardBox	Papkasse
PlasticBox	Plastikkasse
XmasTree	Juletræ

#### 7. Systemets betydning

De største konsekvenser af det samlede system er placeret i nye arbejdsrutiner er, at læssemedarbejderne selv skal angive hvornår en trailer er færdiglæsset, så systemet kan notificere en chauffør. Da systemet kommer til at varetage koordineringen af hver enkelt læsning til en specifik rampe. Læssemedarbejderen vil kunne holde fokus på den enkelte læsning, som deraf vil give en mere optimeret arbejdsroutine. Systemet vil først indføres i Horsens og dermed kunne testes på en af Danish Crowns primære slagterier og deraf vil man kunne lave en analyse opsætning omkring indførsel af det systemet på andre slagterier.

### Resultat af Elaboration

#### 1. Krav

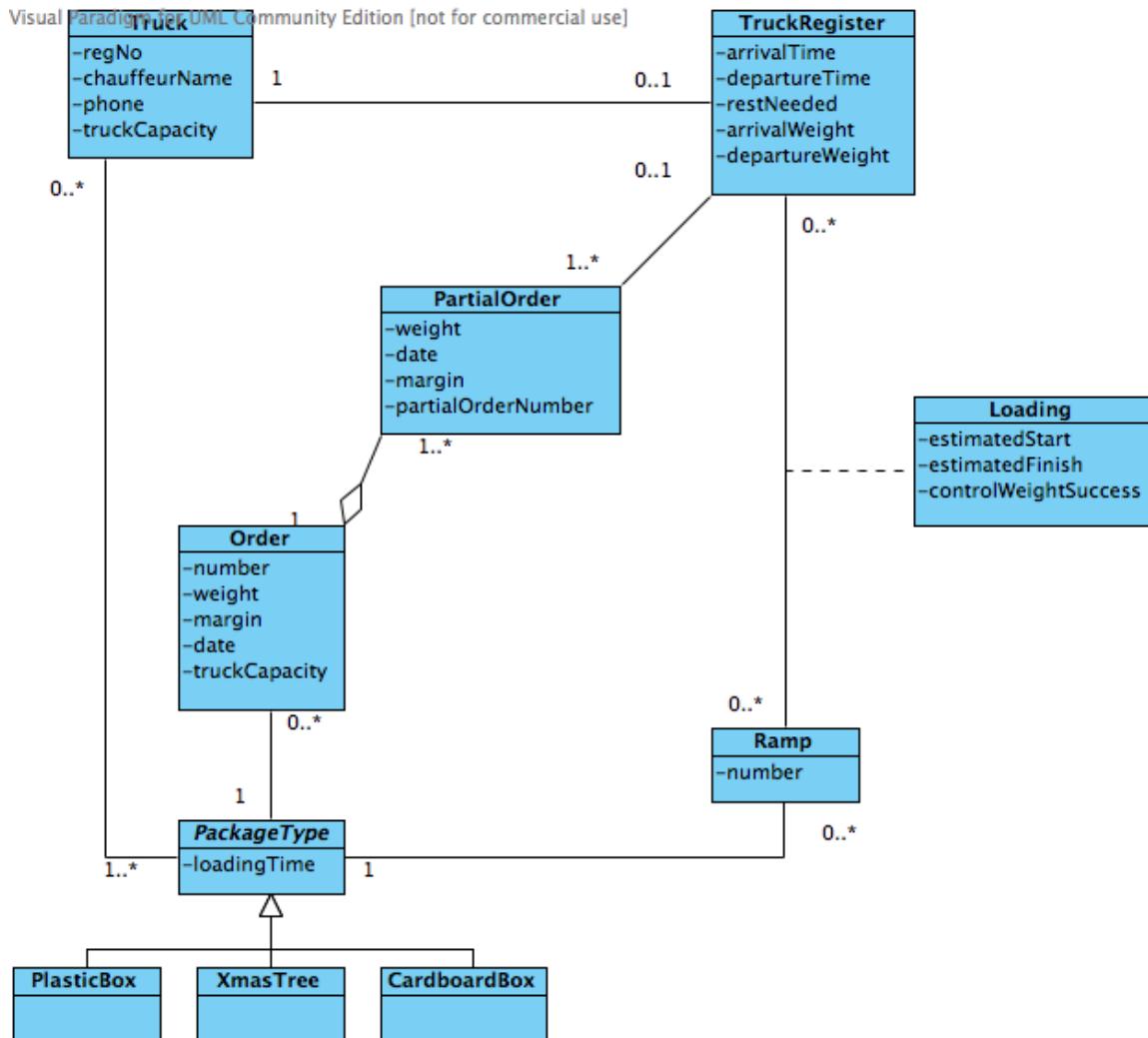
Samme som i inception

#### 2. Use case model

Samme som i inception

### 3. Analysemodel

#### 3.2. Klassediagram

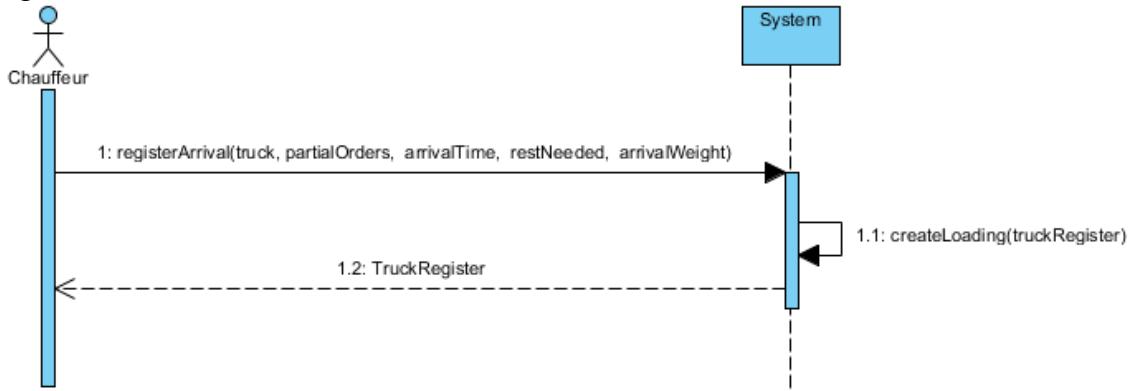


Vi brugte mange(!) timer på, at finde den mest optimale måde at designe vores system på. Vi valgte, at have **PackageType** som en abstrakt klasse, hvor vores konkrete pakagetypes kunne nedarve fra. Dette gør det muligt, at tilføje flere pakagetypes uden at skulle ændre i flere klasser. **Loading** er blevet oprettet som en assosiationsklasse mellem **Ramp** og **TruckRegister**, som viser de løsninger et **TruckRegister** har til fælles med en rampe. **Order** har en aggregering til **PartialOrder**, som gør det muligt at dele store ordre til mindre delordre. I den perfekte verden ville vi gerne have relationen mellem **Order** og **PartialOrder** til at være en komposition, men da **TruckRegister** har brug for linkattribut til **PartialOrder**, blev vi nødsaget til at vælge aggregering.

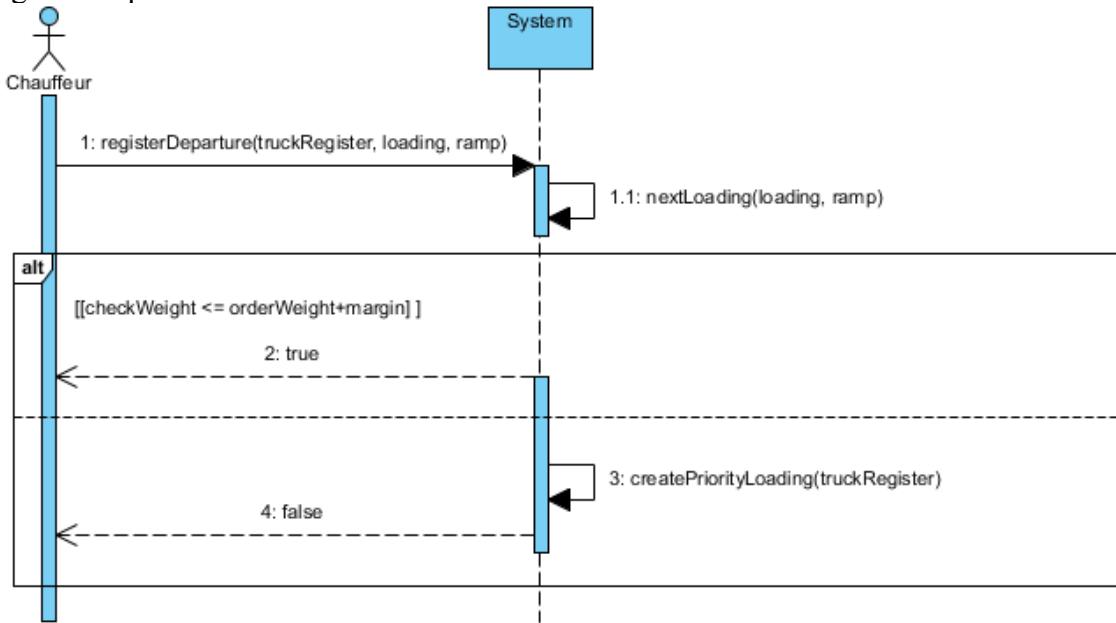
### 3.4. Interaktionsdiagrammer

Vi har valgt at holde fokus på de to centrale usecases i vores system og lave system sekvens diagrammer til at skabe et overblik over de system kald der bliver lavet.

Register Arrival:



Register Departure:

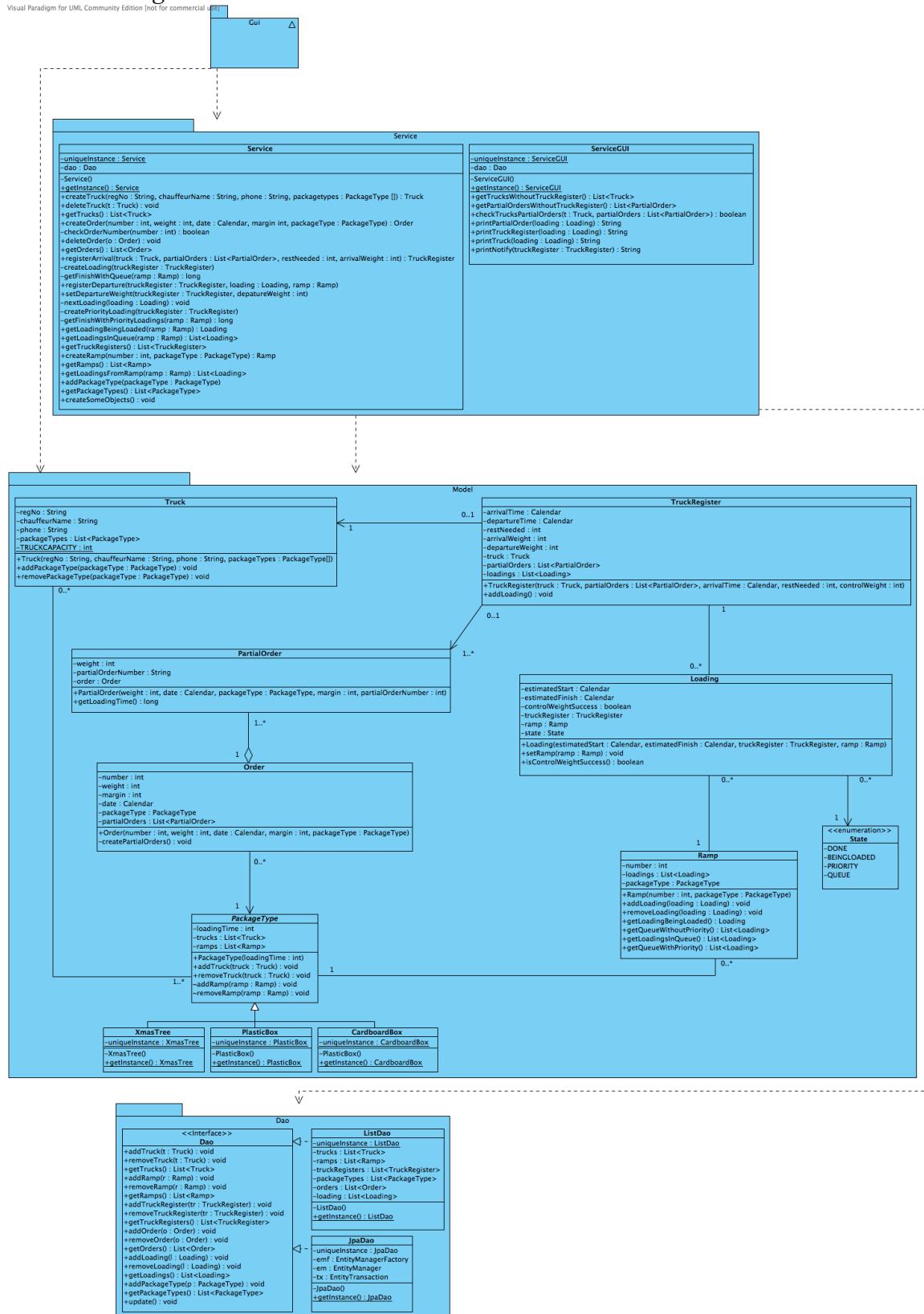


## *4. Designmodel*

### *4.1. Arkitektur*

Vi besluttede os for at benytte 4-lags arkitektur. Lagene er GUI, Service, Dao og Model. Ved en 4-lags arkitektur kan GUI kun kalde metoder i Service klassen, og Service så kan kalde metoder i Dao og Model klasserne, men det er ikke tilladt at gå den modsatte retning. Så snart JPA tilføjes, er det kun Dao som må kalde til databasen. En af fordelene ved denne arkitektur er, at man mindsker sandsynligheden for, at modelklasser ender som såkaldte gudeklasser, altså klasser som kan alt. Det sikrer samtidig en høj samhørighed og lav kobling, hvor vi beholder alle komplekse metoder i Service klassen. Men det vigtigste element i arkitekturen er, at vi kan udskifte de enkelte lag uden, at der skal ske ændringer i nogle af de andre lag. Et eksempel kunne være at man vil indsætte en anden grafiks brugerflade, hvilket vil foregår simpelt fordi at man ikke behøver at ændre i andre lag længere nede.

## 4.2. Klassendiagramm



### 4.3. Klassebeskrivelser

Navn	Order
Formål	Denne klasse fungerer som et hovedelement, da der ikke bliver nogle læsninger uden ordrer.
Attributter	<ul style="list-style-type: none"> <li>• number : int <ul style="list-style-type: none"> <li>◦ Denne variabel bruges til at kende ordrer fra hinanden og på den måde er unikt</li> </ul> </li> <li>• weight : int <ul style="list-style-type: none"> <li>◦ Her holdes ordrens samlede vægt i kg, hvilket bruges til oprettelse af PartialOrder(s)</li> </ul> </li> <li>• date : Calendar <ul style="list-style-type: none"> <li>◦ Datoen ordren forventes afhentet. Vi valgte at bruge Calendar, da Date klassen er forældet. Calendar er en abstrakt klasse, og gennem polymorfi kan vi bruge klassen GregorianCalendar</li> </ul> </li> <li>• margin : int <ul style="list-style-type: none"> <li>◦ Ved ordreoprettelse angives der en fejlmargin i kg. Dette betyder, at systemet skal acceptere læsninger med vægt <math>\pm</math> margin</li> </ul> </li> <li>• packageType : PackageType <ul style="list-style-type: none"> <li>◦ Dette er en link-attribut, som skal angive ordrens pakketype (juletræ, plastkasser eller papkasser)</li> </ul> </li> <li>• partialOrders : List&lt;PartialOrder&gt; <ul style="list-style-type: none"> <li>◦ En liste over ordrens delordre</li> </ul> </li> </ul>
Metoder	<ul style="list-style-type: none"> <li>• createPartialOrders() : void <ul style="list-style-type: none"> <li>◦ Metoden opretter et PartialOrder objekt for hver gang restmængden i ordren er mindre eller lig med Truck.TRUCKCAPACITY og fortsætter til hele ordren er delt ud.</li> </ul> </li> </ul>

Navn	PartialOrder
Formål	Denne klasse fungerer som et hovedelement, da vi ligesom med Order, ikke kan have nogle læsninger uden PartialOrder
Attributter	<ul style="list-style-type: none"> <li>• weight : int <ul style="list-style-type: none"> <li>◦ Her holdes vægten på en PartialOrder, hvilket bruges til at tjekke om læsningen vejer det forventede</li> </ul> </li> </ul>
Metoder	

Navn	Truck
Formål	Denne klasse modellerer en lastbil. En lastbil er både styrehus og trailer, hvilket gør, at vores klasse indeholder informationer om både trailer og chauffør.
Attributter	<ul style="list-style-type: none"> <li>• regNo : String <ul style="list-style-type: none"> <li>◦ Lastbilens registreringsnummer</li> </ul> </li> <li>• chauffeurName : String <ul style="list-style-type: none"> <li>◦ Chaufførens navn</li> </ul> </li> <li>• phone : String <ul style="list-style-type: none"> <li>◦ Chaufførens telefonnummer, hvilket skal bruges til advisering</li> </ul> </li> <li>• packageTypes : List&lt;PackageType&gt; <ul style="list-style-type: none"> <li>◦ En liste af typer som traileren kan indeholde. Her har vi lavet det, så man ikke kan fjerne alle typer fra lastbilen. Der skal altid være mindst en type</li> </ul> </li> <li>• <u>TRUCKCAPACITY</u> : int <ul style="list-style-type: none"> <li>◦ Her gemmes max mængde én lastbil kan indeholde, hvilket bruges til oprettelse af PartialOrders</li> </ul> </li> </ul>
Metoder	<ul style="list-style-type: none"> <li>• addPackageType(packageType : PackageType) : void <ul style="list-style-type: none"> <li>◦ Tilføjelse af packagetype til vores Truck, hvor vi samtidig sikrer en tilføjelse af vores Truck under PackageType.</li> </ul> </li> <li>• removePackageType(packageType : PackageType) : void <ul style="list-style-type: none"> <li>◦ Her fjerner vi en packagetype fra vores truck, men sikrer samtidig at overholde associeringen fra Truck til PackageType. Man skal mindst have en packagetype.</li> </ul> </li> </ul>

Navn	PackageType
Formål	Klassen er defineret som en abstract super klasse, hvilket betyder at vi ikke kan oprette objekter af den. Men i stedet bruges den som en klasse der definerer nogle fælles metoder og attributer som alle dens subklasser nedarver. Formålet er at vi laver subklasser af PackageType som er konkrete implementationer af en PackageType hos Danish Crown, her er det XmasTree, PlasticBox og CardBoardBox, som definerer de forskellige typer af pakkemateriel der kan fragtes og læsses.
Attributter	<ul style="list-style-type: none"> <li>• loadingTime : int <ul style="list-style-type: none"> <li>◦ En variabel som viser læssetid for den enkelte type. Dette er pr. 300 kg.</li> </ul> </li> </ul>
Metoder	<ul style="list-style-type: none"> <li>• addTruck(truck : Truck) : void <ul style="list-style-type: none"> <li>◦ Her tilføjer vi en lastbil til en bestemt pakketype samt sikrer en dobbeltrettet assosiering mellem de to klasser</li> </ul> </li> <li>• removeTruck(truck : Truck) : void <ul style="list-style-type: none"> <li>◦ Her fjerner vi en lastbil fra en bestemt pakketype samt</li> </ul> </li> </ul>

	<p>sikrer at pakketypen bliver fjernet hos lastbilen</p> <ul style="list-style-type: none"> <li>• addRamp(ramp : Ramp) : void           <ul style="list-style-type: none"> <li>◦ Tilføjelse af en ramp med sikring af assosiering mellem rampe og pakketype. Denne metode er gjort package private for at sikre, at Service ikke har adgang til den.</li> </ul> </li> <li>• removeRamp(ramp : Ramp) : void           <ul style="list-style-type: none"> <li>◦ Dette er til at fjerne en rampe. Som udgangspunkt er det noget man ikke har brug for, men vi beholdte den, da det så gav en mulighed for omrøkering af ramper, altså skift af en rampe fra f.eks. XmasTree til PlasticBox. Denne metode er også package private.</li> </ul> </li> </ul>
--	---

Navn	TruckRegister
Formål	Klassen er lavet så den indeholder en lastbil, og den ordre som lastbilen skal afhente, sammen med de tilhørende loadings. Klassen skal indeholde information omkring hvornår lastbilen er ankommet, og hvornår den er taget af sted igen, samt det tidspunkt hvorpå chaufføren vil være klar til at tage af sted ifølge hans kørehviletidsbestemmelser.
Attributter	<ul style="list-style-type: none"> <li>• arrivalTime : Calendar           <ul style="list-style-type: none"> <li>◦ Det tidspunkt hvor lastbilen ankommer til Danish Crown</li> </ul> </li> <li>• departureTime : Calendar           <ul style="list-style-type: none"> <li>◦ Det tidspunkt hvor lastbilen forlader Danish Crown.</li> </ul> </li> <li>• restNeeded : int           <ul style="list-style-type: none"> <li>◦ Ved oprettelse af TruckRegister skal der også indtastes hvormange minutter chaufføren behøver at hvile før han kan tage afsæd igen</li> </ul> </li> <li>• arrivalWeight : int           <ul style="list-style-type: none"> <li>◦ Vægten på lastbilen i kg før ordren bliver læsset</li> </ul> </li> <li>• departureWeight : int           <ul style="list-style-type: none"> <li>◦ Vægten på lastbilen i kg efter læsning</li> </ul> </li> <li>• truck : Truck           <ul style="list-style-type: none"> <li>◦ Link attribut til lastbilen</li> </ul> </li> <li>• order : Order           <ul style="list-style-type: none"> <li>◦ Link attribut til ordren som skal afhentes</li> </ul> </li> <li>• loadings : List&lt;Loading&gt;           <ul style="list-style-type: none"> <li>◦ List til indeholdelse af en læsning og alle de eventuelle læsninger som er foretaget på den enkelte TruckRegister</li> </ul> </li> </ul>
Metoder	<ul style="list-style-type: none"> <li>• addLoading/loading : Loading) : void           <ul style="list-style-type: none"> <li>◦ Metoden tilføjer en loading hvis den ikke allerede er tilknyttet.</li> </ul> </li> </ul>

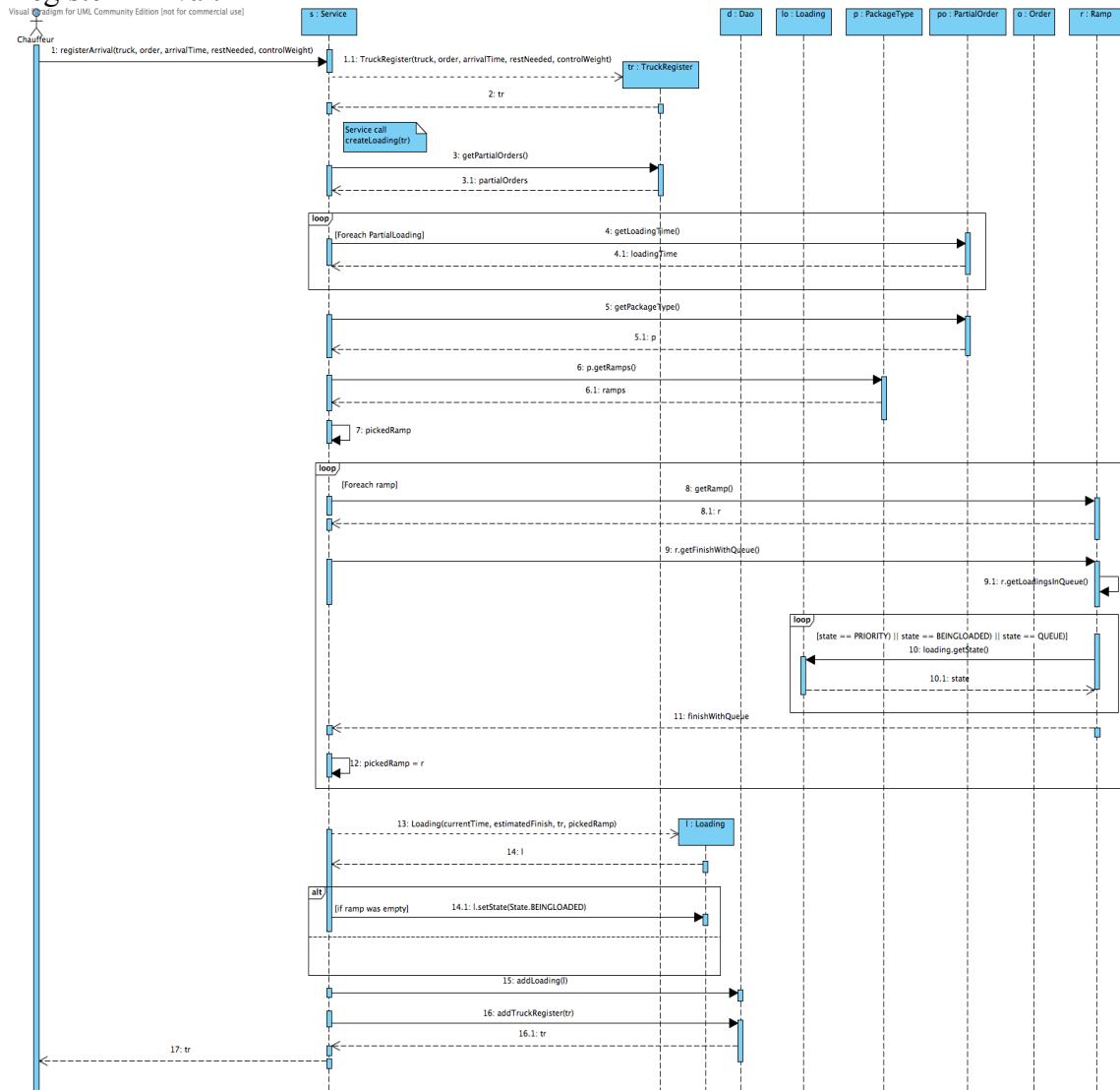
Navn	Loading
Formål	Klassen er blevet lavet som en associerings klasse mellem TruckRegister og Ramp da det skal være muligt at se om en lastbil er blevet forsøgt læsset flere gange samt hvilke tidspunkter det forventes, at de enkelte loadings blev påbegyndt og afsluttet.
Attributter	<ul style="list-style-type: none"> <li>• estimatedStart : Calendar <ul style="list-style-type: none"> <li>◦ Det tidspunkt hvor læsningen af en ordre forventes at starte.</li> </ul> </li> <li>• estimatedFinish : Calendar <ul style="list-style-type: none"> <li>◦ Det tidspunkt hvor læsningen af en ordre forventes at være afsluttet.</li> </ul> </li> <li>• controlWeightSuccess : boolean <ul style="list-style-type: none"> <li>◦ En boolean variabel som indeholder svaret på om den Loading er godkendt.</li> </ul> </li> <li>• truckRegister : TruckRegister <ul style="list-style-type: none"> <li>◦ En link attribut som indeholder den tilhørende TruckRegister.</li> </ul> </li> <li>• ramp : Ramp <ul style="list-style-type: none"> <li>◦ En link attribut som indeholder den rampe som en Loading skal foregå på.</li> </ul> </li> <li>• state : State <ul style="list-style-type: none"> <li>◦ state en en Enumeration som fortæller hvilken tilstand som den enkelte Loading er i.</li> </ul> </li> </ul>
Metoder	<ul style="list-style-type: none"> <li>• setRamp(ramp : Ramp) : void <ul style="list-style-type: none"> <li>◦ Metoden sætter den tilknyttede Ramp og hvis der i forvejen var en Ramp tilknyttet fjernes den og denne Loading fjernes fra den før tilknyttede Ramp, hvorefter den nye Ramp bliver tilknyttet og denne Loading bliver tilknyttet den nye Ramp.</li> </ul> </li> <li>• isControlWeightSuccess() : boolean <ul style="list-style-type: none"> <li>◦ Metoden returnere en boolean værdi for om denne Loading er lykkedes eller fejlet.</li> </ul> </li> </ul>

Navn	Ramp
Formål	Klassen Ramp skal symbolisere de ramper som slagteriet har, med en perfekt packageType som den kan udlæsse.
Attributter	<ul style="list-style-type: none"> <li>• number : int <ul style="list-style-type: none"> <li>◦ Det nummer som rampen har.</li> </ul> </li> <li>• loadings : List&lt;Loading&gt; <ul style="list-style-type: none"> <li>◦ En liste som indeholder alle de loadings som venter på at blive læsset på den pågældende rampe</li> </ul> </li> <li>• packageType : PackageType <ul style="list-style-type: none"> <li>◦ Den type vare som rampen kan håndtere.</li> </ul> </li> </ul>
Metoder	<ul style="list-style-type: none"> <li>• addLoading/loading : Loading) : void <ul style="list-style-type: none"> <li>◦ Metoden tilføjer en Loading til en liste. Loadingen bliver tilføjet til listen ved hjælp af en insertionsort, som sætter de nye loading ind foran en anden, hvis det forventede starttidspunkt er før det på den anden Loading.</li> </ul> </li> <li>• removeLoading/loading : Loading) : void <ul style="list-style-type: none"> <li>◦ Metoden fjerner en Loading</li> </ul> </li> </ul>

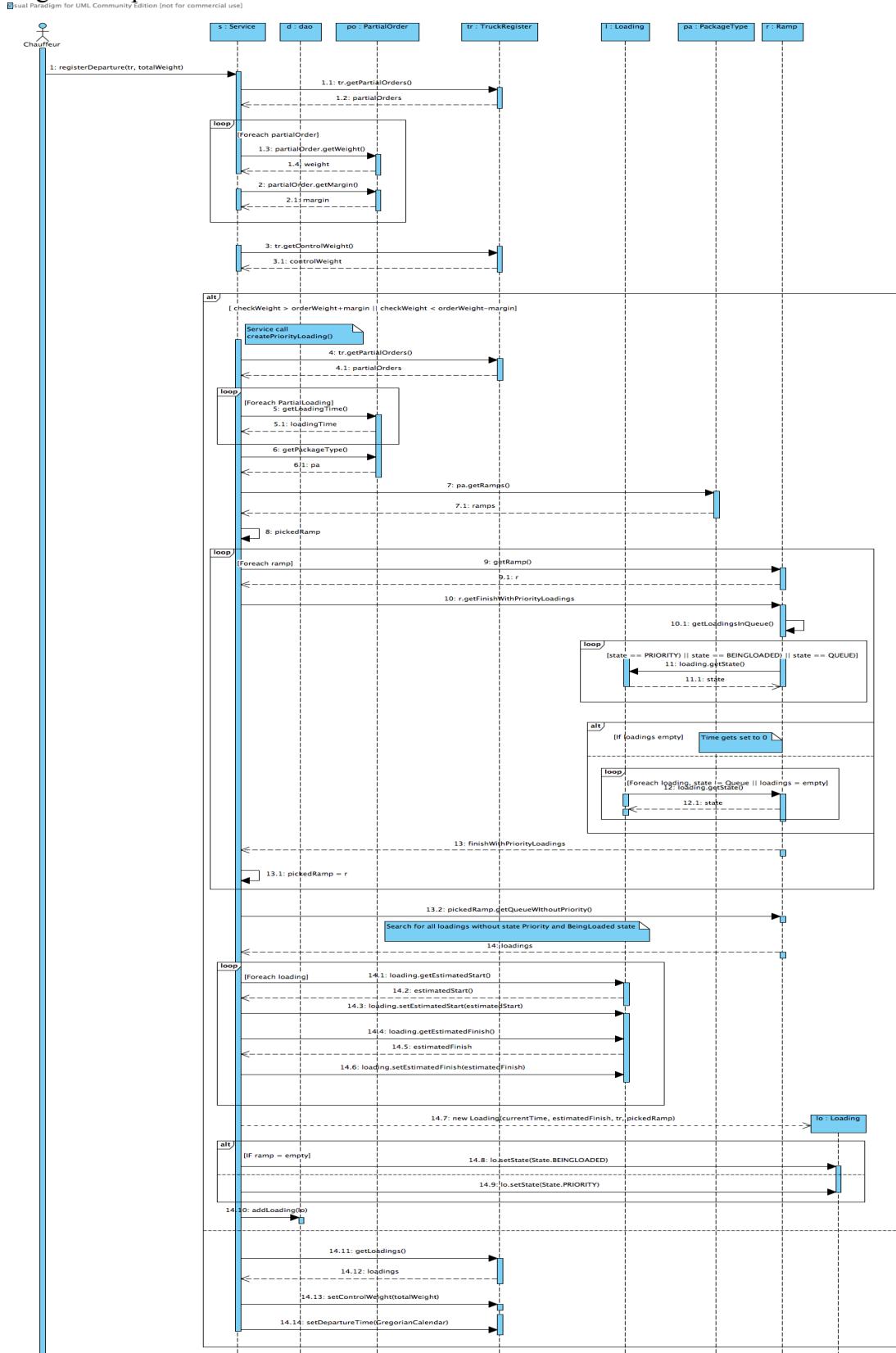
#### 4.4. Detaljerede sekvensdiagrammer

Vi har videre udviklet interaktions diagrammer til design sekvens diagrammer, for at skabe det samlede overblik over alle system kald, der bliver lavet gennem systemet.

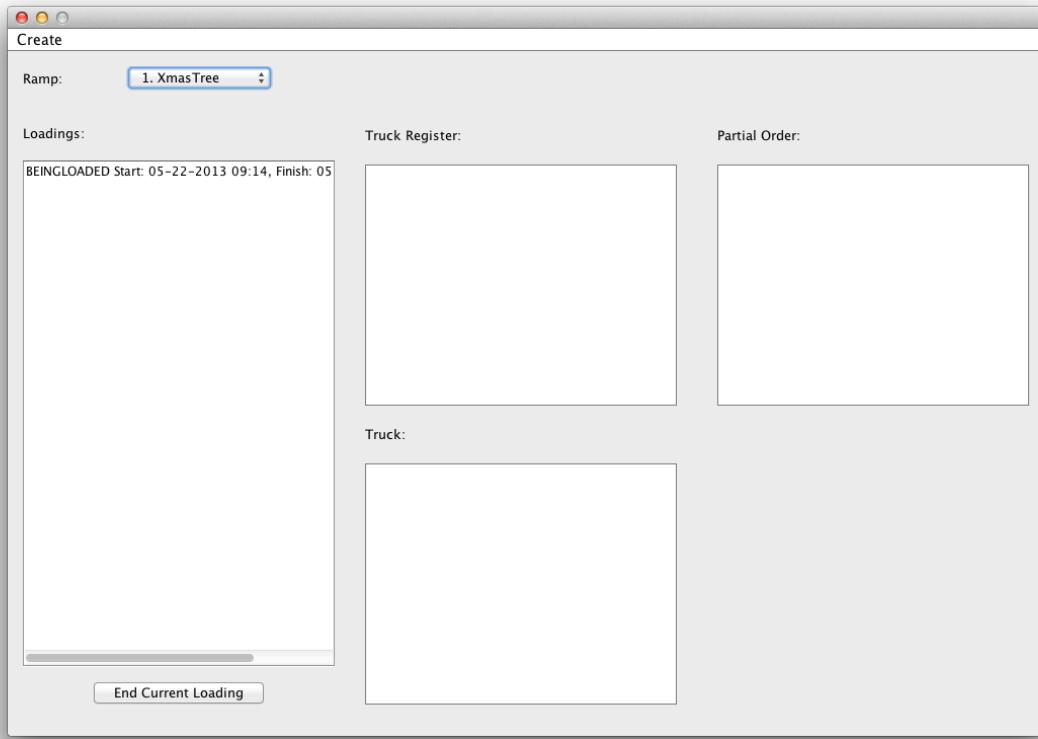
##### Register Arrival:



## Register Departure:



#### 4. Brugergrænseflade



Hovedvinduet hvor det er muligt at se, hvilke Loadings som er på en Ramp, samt information omkring den enkelte Loading. Der er også en mulighed for at afslutte en Loading. Det er samtidig muligt at tilgå vinduer hvor man kan oprette objekter af Order og Truck samt registrere en ankomst.

### Resultat af Construction

Gennem construction fasen har vi ikke lavet nogle ændringer og vi har fortsat arbejdet ud fra dokumentationen i elaboration.

#### 6. Testrapport

Vi har valgt at teste klassen Order, da den indeholder en kompleks metode til oprettelse af PartialOrderer, baseret på MAXCAPACITY i Truck.

## Test af createPartialOrders

Testcase	Input	Forventet resultat	Aktuelt resultat	Status
TC1 getPartialOrders().size	Vægt = 1	1	1	Ok
TC2 getPartialOrders().size	Vægt = 20000	1	1	Ok
TC3 getPartialOrders().size	Vægt = 40000	2	2	Ok
TC4 getPartialOrders().size	Vægt = 0	0	0	Ok
TC5 getPartialOrders().size	Vægt = -500	0	0	Ok

Her testes antallet af PartialOrders, der bliver skabt ved oprettelse af en Order. Vores input for vægt kan inddeltes i ækvivalensmængder, og det er det vi tester med. En ækvivalensklasse indeholder data som er forskellige, men giver det samme output. Vi kan se på vores status at testen er gået godt. Black box testing er typisk gjort af folk, som ikke har kendskab til koden. Ved brug af white box testing har man kendskab til koden, og vores test er et mix af de 2, som hedder grey box testing.

## Test use case Register Arrival (krav: Order og Truck er oprettet)

Test case ID	Test kilde	Start system tilstand	Input	Forventet resultat	Aktuelt resultat
U1	Use case #1	Start test fra main frame	(a) Tryk på menuen Create og vælg TruckRegister  (b) Vælg Truck, sæt hviletid = 30, lastbilens vægt = 3000 ved ankomst og vælg en PartialOrder. Tryk derefter på ok	(a) Et vindue åbner, hvor man kan oprette et TruckRegister  (b) En ny loading vil blive tilføjet til den rampe, som først er ledig	Ok  Ok

Gennemgang af den formelle usecase test Register Arrival. Ved at køre denne usecase i vores program, kunne vi teste om programmet reagerede på den måde vi forventede, og som vi kan se i aktuelt resultat, så gik det godt.

## 7. Status

Vi har udviklet et program, som kan håndtere logistikken ved ankomst og afgang af lastbiler hos Danish Crown. I vores program melder chaufføren sin ankomst, og vil automatisk blive registreret hos den rampe, som først bliver ledig. Vi har ikke nået at tilføje en feature som gjorde det muligt for loading employee at skifte en trucks order. En

anden fejl i systemet er vores fejmargen. Vi havde opfattet det som en absolut værdi, men opdagede først til sidst, at der var tale om en procentmæssig marge, hvilket vi ikke havde tid til at rette.

## Dokumentation af processen

### 1. ResUME AF FORLOBET

Vi brugte hele den første uge på at finde krav til systemet, lave usecase diagram, lave usecase beskrivelser og påbegynde et analysediagram. Ugen efter skrev vi aktørbeskrivelser, ordbog og usecasebeskrivelser, hvorefter vi afsluttede inception fasen med et brugbart analysediagram, og begyndte på elaboration. Gennem elaboration revurderede vi vores use cases og påbegyndte vores designdiagram, samt udfylde klassebeskrivelser. Først lavede vi system sekvens diagrammer af vores usecases, hvorefter vi lavede design sekvens diagrammer. I construction sluttede vi med en testrapport, som kan ses i rapporten.

### 2. Refleksioner

#### 2.1. Unified Process

En ulempe ved unified process er, at hvis man laver en ændring skaber det en kaskade effekt mod diagrammer med mere man fik udarbejdet i tidligere iterationer. Hele idéen om at oprette og tilpasse diagrammer for hver iteration skaber hurtigt kaos.

#### 2.2. Udviklingsværktøjer

Diagrammer er oprettet ved brug af visual paradigm. Udarbejdelse af kode i projektet har vi skrevet i eclipse, og samtidig benyttet os af egit, der er et versionsstyringssystem, hvor det er nemt at tjekke alle ændringer samt gå tilbage til en tidligere version, hvis der skulle opstå en fejl.

#### 2.3. Øvrige forhold

Vi startede ud med at dele alle opgaver ud, men som forløbet gik, endte vi alle med at have arbejdet i alle klasser. Gennem hele projektperioden har vi siddet ved siden af hinanden og arbejdet hele tiden, så vi var sikret en gensidig indflydelse i alle dele af opgaven. Af den årsag har vi valgt at notere alle navne på alle klasser, da vi enten havde lavet klassen eller rettet i den løbende.

## Klassemødel:

### Order:

```
public class Order {  
    ...  
    private List<PartialOrder> partialOrders;  
    ...  
  
    public Order(int number, int weight, int margin, Calendar date, PackageType  
    packageType) {  
        ...  
    }  
}
```

```

        //Making the call for creating partialOrders
        createPartialOrders();
    }

    private void createPartialOrders() {
        int weight = this.weight;
        int index = 1;
        while(weight > 0) {
            if(weight <= Truck.TRUCKCAPACITY) {
                PartialOrder p = new PartialOrder(this, weight, number+"-"+index);
                partialOrders.add(p);
                weight = 0;
            } else if( weight > Truck.TRUCKCAPACITY) {
                PartialOrder p = new PartialOrder(this, Truck.TRUCKCAPACITY, number+"-"+index);
                partialOrders.add(p);
                weight -= Truck.TRUCKCAPACITY;
            }
            index++;
        }
    }
}

```

Order kan indeholde 1 til mange PartialOrder, og er blevet oprettet som en aggregering. Dette gør, at vi må oprette PartialOrder fra vores Order klasse samtidig med, at andre klasser godt må have referencer til PartialOrder. Realiseringen sker ved at kalde en private hjælpemetode som opretter et vist antal PartialOrder objekter baseret på maks kapaciteten på en Truck.

```

public class Order {
    ...
    private PackageType packageType;
    ...

    public Order(int number, int weight, int margin, Calendar date, PackageType
    packageType) {
        ...
        this.packageType = packageType;
        ...
    }
}

```

Order er oprettet med en tvungen assosiering, hvor vi skal have en PackageType og kan ikke få flere PackageTypes tilføjet, selve associationen er enkeltrettet fra Order til PackageType.

*PartialOrder:*

```

public class PartialOrder {

    ...
    private Order order;
}

}

```

PartialOrder oprettes gennem aggregeringen i Order, og er dobbeltrettet.

*Truck:*

```

public class Truck {
    private List<PackageType> packageTypes;
    ...

    public Truck(String regNo, String chauffeurName, String phone, PackageType[]
    packagetypes) {

```

```

    packageTypes = new ArrayList<>();

    //adding the package types to the truck
    for (PackageType packageType : packageTypes) {
        addPackageType(packageType);
    }
}
}

```

En Truck indeholder 1 til mange PackageTypes, hvor Truck og PackageType begge kan se hinanden.

```

public class Truck {
    ...
    private List<PackageType> packageTypes;
    ...
    public void addPackageType(PackageType packageType) {
        if (!packageTypes.contains(packageType)) {
            packageTypes.add(packageType);
            packageType.addTruck(this);
        }
    }
}

```

Hvor en packageType tilføjes, hvis den ikke allerede eksisterer i listen

```

Public abstract class PackageType {
    ...
    public void addTruck(Truck t){
        if(!trucks.contains(t)){
            trucks.add(t);
            t.addPackageType(this);
        }
    }
}

```

Når der tilføjes en Packagetype eller Truck sørger vi for at opretteholde associationen i fra både Truck og PackageType.

```

public class Truck {
    ...
    private List<PackageType> packageTypes;
    ...

    public void removePackageType(PackageType packageType) {
        if (packageTypes.size() > 1) {
            if (packageTypes.contains(packageType)) {
                packageTypes.remove(packageType);
                packageType.removeTruck(this);
            }
        }
    }
}

```

Hvor en packageType fjernes, hvis listens størrelse er større end 1 og listen indeholder den packageType som er angivet i parameteren.

```

Public abstract class PackageType {
    ...
    public void removeTruck(Truck t){
        if(trucks.contains(t)){
            trucks.add(t);
            t.removePackageType(this);
        }
    }
}

```

```
}
```

Når der fjernes en Packagetype eller Truck sørger vi for at opretteholde associationen i fra både Truck og PackageType.

*PackageType:*

```
public abstract class PackageType {  
  
    ...  
    private List<Truck> trucks;  
    private List<Ramp> ramps;  
}  
  
En packagetype har linkattributter til Truck og Ramp. De er dobbeltrettet, så der er add og remove metoder i PackageType, som opretholder truck associationen i begge retninger.  
public abstract class PackageType {  
    /**  
     * Method for adding a truck  
     * @param t - truck to be added  
     */  
    public void addTruck(Truck t){  
        if(!trucks.contains(t)){  
            trucks.add(t);  
            t.addPackageType(this);  
        }  
    }  
  
    /**  
     * Method for removing a truck  
     * @param t - truck to be removed  
     */  
    public void removeTruck(Truck t){  
        if(trucks.contains(t)){  
            trucks.remove(t);  
            t.removePackageType(this);  
        }  
    }  
  
    /**  
     * Method for getting all the trucks  
     * @return List<Truck> with all trucks  
     */  
    public List<Truck> getTrucks(){  
        return new ArrayList<>(trucks);  
    }  
  
    /**  
     * Method for getting all the ramps  
     * @return List<Ramp> with all the ramps  
     */  
    public List<Ramp> getRamps(){  
        return new ArrayList<>(ramps);  
    }  
}  
  
public abstract class PackageType {  
    /**  
     * Method for adding a ramp  
     * @param r - ramp to be added  
     */  
    void addRamp(Ramp r){  
        if(!ramps.contains(r)){  
            ramps.add(r);  
        }  
    }  
}
```

```

    /**
     * Method for removing a ramp
     * @param r - ramp to be removed
     */
    void removeRamp(Ramp r) {
        if(ramps.contains(r)) {
            ramps.remove(r);
        }
    }
}

```

Dog bliver ramp associationen kun oprethold fra ramp, hvor metoder i PackageType er packageprivate

*Ramp:*

Mellem Ramp og PackageType er der en, mange til en dobletrettet associering ,hvor Ramp er tvunget til at have en PackageType, som ikke skal kunne ændres på et senere tidspunkt.

```

public class Ramp {
    ...
    private PackageType packageType;

    public Ramp(int number, PackageType packageType) {
        this.number = number;
        loadings = new ArrayList<>();
        this.packageType = packageType;
        packageType.addRamp(this);
    }
}

```

Hvor PackageType bliver sendt med ind som parameter.

```

public class Ramp {
    ...
    void addRamp(Ramp r) {
        if(!ramps.contains(r)) {
            ramps.add(r);
        }
    }
}

```

Når PackageType bliver sat i constructoren, sørger vi for at opretholde associationen fra PackageType, metoden er package protected hvilket gør at vi sørger for at når vi opretter en rampe, så bliver den automatisk tilføjet til den PackageType som vi angiver i constructoren.

```

public class Ramp {
    ...
    private PackageType packageType;

    public PackageType getPackageType() {
        return packageType;
    }
}

```

Hvor man kan få den tilknyttede PackageType.

Der er ingen set metode, da en Ramp ikke skal kunne ændre den PakageType som den kan læsse.

```

public class Ramp {
    ...
    private List<Loading> loadings;
    ...

    public List<Loading> getLoadings() {
        return new ArrayList<>(loadings);
    }
}

```

Mellem Ramp og Loading er der en, en til mange dobbeltrettet associering, da en Ramp kan indeholde flere Loading. Hvor man kan få alle de Loadings som er tilknyttet denne Ramp

```

public class Loading implements Comparable<Loading> {
    ...
    private Ramp ramp;
    ...

    public Loading(Calendar estimatedStart, Calendar estimatedFinish, TruckRegister
truckRegister, Ramp ramp) {
        ...
        setRamp(ramp);
    }

    public void setRamp(Ramp ramp) {
        if(this.ramp != ramp) {
            if(this.ramp != null) {
                this.ramp.removeLoading(this);
            }
            this.ramp = ramp;
            if(ramp != null) {
                ramp.addLoading(this);
            }
        }
    }
}

```

Set fra Loading, igen varetager vi associationen fra begge sider, så når en Loading oprettes bliver der sørget for at dens rampe bliver sat med det samme.

```

public class Ramp {
    ...
    private List<Loading> loadings;
    ...

    public void addLoading(Loading loading) {
        if(!loadings.contains(loading)) {
            int index = loadings.size();
            while(index > 0 && loadings.get(index-1).compareTo(loading) > 0) {
                index--;
            }
            loadings.add(index, loading);
        }
        loading.setRamp(this);
    }
}

```

Hvor en Loading bliver tilføjet ved hjælp af en insertionsort.

```

public class Loading implements Comparable<Loading> {
    ...
    public void setRamp(Ramp ramp) {
        if(this.ramp != ramp) {
            if(this.ramp != null) {
                this.ramp.removeLoading(this);
            }
            this.ramp = ramp;
            if(ramp != null) {

```

```

        ramp.addLoading(this);
    }
}
}
```

Associationen varetages igen fra begge sider.

```

public class Ramp {
    ...
    private List<Loading> loadings;
    ...

    void removeLoading(Loading loading) {
        loadings.remove(loading);
        loading.setRamp(null);
    }
}
```

Hvor man kan fjerne en Loading og sætte dennes Ramp til null.

*Loading:*

Mellem Loading og Ramp er der en dobbeltrettet mange til en associering da en Loading kun kan være tilknyttet en Ramp

```

public class Loading implements Comparable<Loading> {
    ...
    private Ramp ramp;
    ...

    public Loading(Calendar estimatedStart, Calendar estimatedFinish, TruckRegister
truckRegister, Ramp ramp) {
        ...
        setRamp(ramp);
    }
}
```

Hvor der i constructeren bliver kaldt en setRamp metode.

```

public class Loading implements Comparable<Loading> {
    ...
    private Ramp ramp;
    ...

    public void setRamp(Ramp ramp) {
        if(this.ramp != ramp) {
            if(this.ramp != null) {
                this.ramp.removeLoading(this);
            }
            this.ramp = ramp;
            if(ramp != null) {
                ramp.addLoading(this);
            }
        }
    }
}
```

Hvor der bliver sat en eventuel ny Ramp, hvis den ikke er identisk med den allerede tilføjede eller null.

```

public class Loading implements Comparable<Loading> {
    ...
    private Ramp ramp;
    ...

    public Ramp getRamp() {
        return ramp;
    }
}
```

```
}
```

Hvor man kan få den tilknyttede Ramp.

Mellem Loading og TruckRegister er der en dobbeltrettet mange til en associering da en Loading kun kan være tilknyttet en TruckRegister, som ikke kan ændres efter oprettelse.

```
public class Loading implements Comparable<Loading> {
    ...
    private TruckRegister truckRegister;
    ...
    public Loading(Calendar estimatedStart, Calendar estimatedFinish, TruckRegister
truckRegister, Ramp ramp) {
        ...
        this.truckRegister = truckRegister;
        ...
    }
}
```

Hvor der i constructeren bliver sat en TruckRegister, og Loading bliver tilknyttet denne TruckRegister.

```
public class Loading implements Comparable<Loading> {
    public TruckRegister getTruckRegister(){
        return truckRegister;
    }
}
```

Hvor den tilknyttede TruckRegister kan hentes.

TruckRegister bliver sat i constructeren da det ikke er muligt at ændre den tilhørende TruckRegister, derfor er der heller ingen setTruckRegister

*TruckRegister:*

Mellem TruckRegister og Truck er der en enkeltrettet 0-1 til 1 associering da en TruckRegister altid vil have en Truck registeret, og der ikke er nogen grund til at en Truck kan se hvilke TruckRegister som den er tilknyttet.

```
public class TruckRegister {
    ...
    private Truck truck;
    ...

    public TruckRegister(Truck truck, List<PartialOrder> partialOrders, Calendar
arrivalTime, int restNeeded, int arrivalWeight){
        ...
        this.truck = truck;
        ...
    }
}
```

TruckRegister har en Truck med som parameter i constructeren.

```
public class TruckRegister {
    ...
    private Truck truck;
    ...

    public Truck getTruck(){
        return truck;
    }
}
```

Hvor den tilhørende Truck kan hentes.

Vi har valgt ikke at have en set metode senere da det ikke skal være muligt at ændre den tilknyttede Truck.

Mellem Truckregister og PartialOrder er der en enkeltrettet 0-1 til mange associering da en TruckRegister kan indeholde mange PartialOrder, og en PartialOrder ikke har nogen grund til at vide hvilke TruckRegister som indeholder den.

```
public class TruckRegister {  
    ...  
    private List<PartialOrder> partialOrders;  
    ...  
  
    public TruckRegister(Truck truck, List<PartialOrder> partialOrders, Calendar  
        arrivalTime, int restNeeded, int arrivalWeight){  
        ...  
        this.partialOrders = new ArrayList<>(partialOrders);  
        ...  
    }  
}
```

Hvor der er en liste af PartialOrders med som parameter i konstrueren, hver PartialOrder i den liste bliver tilføjet til en liste som bliver oprettet i konstrueren.

```
public class TruckRegister {  
    ...  
    private List<PartialOrder> partialOrders;  
    ...  
  
    public List<PartialOrder> getPartialOrders(){  
        return new ArrayList<>(partialOrders);  
    }  
}
```

Hvor de tilhørende PartailOrders kan hentes.

Vi har valgt ikke at have en add metode til at tilføje en PartialOrder, da alle de PartialOrders som ønskes registeret med den givne TruckRegister skal tilføjes gennem konstrueren.

Mellem TruckRegister Og Loading er der en 1 til mange dobbeltrettet associering da en TruckRegister kan have flere Loadings, det vil ske hvis en Loading fejler.

```
public class TruckRegister {  
    ...  
    private List<Loading> loadings;  
  
    public TruckRegister(Truck truck, List<PartialOrder> partialOrders, Calendar  
        arrivalTime, int restNeeded, int arrivalWeight){  
        ...  
        this.loadings = new ArrayList<>();  
    }  
}
```

Der bliver i konstrueren lavet en liste som kan indeholde alle de tilknyttede Loadings.

```
public class TruckRegister {  
    ...  
    private List<Loading> loadings;  
  
    public void addLoading(Loading loading){  
        if(!loadings.contains(loading)){  
            loadings.add(loading);  
        }  
    }  
}
```

```

        }
    }
}
```

Hvor en Loading bliver tilføjet til hvis den ikke i forvejen er i listen. Denne metode er lavet som public da den skal kunne kaldes fra service og ikke fjernes senere, derfor er der heller ingen remove metode.

```

public class TruckRegister {
    public List<PartialOrder> getLoadings(){
        return new ArrayList<>(loadings);
    }
}
```

Hvor alle de tilknyttede Loadings kan hentes.

## Collection typer

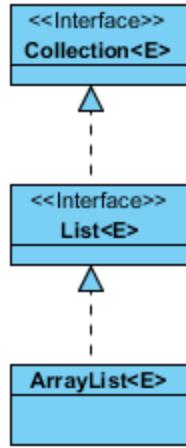
Vi anvender gennem projektet Collection typen List som den statiske type og ArrayList som den dynamiske konkrete type.

Vi har valgt ArrayList af den årsag at vi vil have mulighed for at kunne sortere på flere forskellige måder gennem systemet. Vi anvender InsertionSort til at indsætte elementer i Listen i Ramp. InsertionSorts tidskompleksitet vil i værste tilfælde altid være den samme, som er Big(O)=n<sup>2</sup>

```

public class Ramp {
    ...
    /**
     * Method for adding a new loading and sorting it
     * with insertion sort, for keeping the whole list sorted
     * @param loading to be sorted in
     */
    public void addLoading>Loading loading) {
        if(!loadings.contains(loading)){
            int index = loadings.size();
            while(index > 0 && loadings.get(index-1).compareTo(loading) >
            0){
                index--;
            }
            loadings.add(index, loading);
        }
        loading.setRamp(this);
    }
    ...
}
```

Da ArrayList implementerer Interfacet List, og List implementerer interfacet Collection, arver det samtidig alle metoder, hvilket gør at det er muligt for os at kalde Collections.reverse().



I en ArrayList har de enkelte objekter en index placering, hvorfra det er muligt at få objektet.

I teorien ville det være muligt at anvende Set i stedet for List de fleste steder, dog ikke i Ramp da det ikke er muligt at sortere et Set.

### Anvendelse af polymorfi:

PackageType er oprettet som en abstract klasse, indeholdende metoder til at opretholde associationer mellem Truck og Ramp. Udover dette indeholder den også en bestemt loadingTime, som er en tiden i minutter for hver 300 kg.

Da klassen er abstract, defineres den som at vi ikke kan oprette objekter af den, men gennem "Composition over inheritance" udnytte, at hvert enkelt specific PackageType, dvs. XmasTree, CardBoardBox, PlasticBox, nedarver fra PackageType, så de hver har deres egen liste over trucks og ramps, som har den samme konkrete packageType.

Eksempelt er baseret på klassen Truck, men gør sig gældende hele vejen igennem systemet.

```

public class Truck {
    ...
    private List<PackageType> packageTypes;
}
    
```

Her er der defineret en Collection indeholdende PackageType, selvom vi ikke kan oprette objekter af PackageType, definerer polymorfi at denne Collection kan indeholde alle subtyper af PackageType.

```

public class Truck {
    ...
    /**
     * Constructor for the class Truck
     * @param regNo - registration number for the truck
     * @param chauffeurName - name of the chauffeur
     * @param phone - phone number of the chauffeur
     * @param packagetypes != null - package types which the truck can contain
     */
    public Truck(String regNo, String chauffeurName, String phone, PackageType[] packagetypes) {
        ...
        //adding the package types to the truck
        for (PackageType packageType : packagetypes) {
            addPackageType(packageType);
        }
    }
}
    
```

```

        }

    }

    /**
     * Method for adding a packagetype
     * @param packageType != null - packagetype to be added
     */
    public void addPackageType(PackageType packageType) {
        if (!packageTypes.contains(packageType)) {
            packageTypes.add(packageType);
            packageType.addTruck(this);
        }
    }
}

```

Når vi konstruerer en metode som tager en PackageType med som parameter, vil vi på runtime angive en konkret subtype til PackageType, og den konkrete type er også det som vil blive tjekket for i contains checket. Så hvis vi tilføjer XmasTree som PackageType, vil contains tjekke Collection igennem for om der allerede eksisterer et XmasTree objekt.

Det smarte i polymorfien er at når vi kalder addTruck i addPackageType metoden, vil javas JVM selv gå ind og finde den dynamiske type på runtime og tilføje det truck objekt vi ser på nu til den specifikke packageType, her xmasTree.

Hvis vi udnytter at JVM finder den dynamiske type på runtime, vil vi kunne lave en metode som returnerer de ramper som har den samme konkrete packageType, som den vi tager med i parameteren:

```

public abstract class PackageType {
    public List<Ramp> getRamps(PackageType packageType) {
        return packageType.getRamps();
    }
}

```

Denne metode vil være polymofisk idet at vi kan parse hvilken som helst subklasse af PackageType ind som parameter og på runtime vil JVM, finde den aktuelle/dynamiske type og kalde dens getRamps() metode, som kun vil returnerer de ramps som har samme packageType.

Ideen i dette kommer fra nedarvningen, da PackageType klassen definerer alle de metoder og ikke private attributer som er fælles for subklasserne, dvs. At attributer enten kan bruges direkte eller kaldes gennem metoder. Private attributer nedarves ikke, men kan tilgås gennem metoder, hvis disse er oprettet. Derudover kan man i subklasserne overskrive de enkelte metoder som er defineret af superklassen, ved at bruge samme metode signature i subklassen. Superklassen vil også kunne definerer abstract metoder, som skal konkret implementeres i subklasserne eller superklassen kan være et interface, som kræver at specifikke metoder implementeres.

Så mulighederne for individuelle konkrete implementationer af de forskellige nedarvet/implementeret metoder er næsten endeløse.

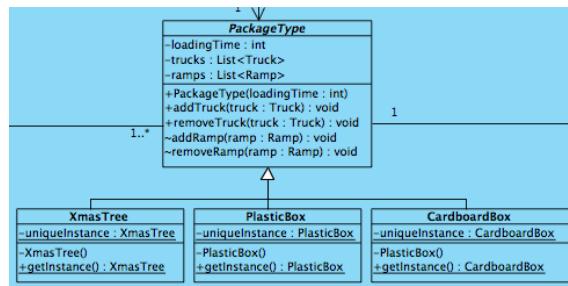
Helt konkret giver polymorfi os mulighed for at ændre opførsel på runtime, så hvis et Truck objekt har et XmasTree object som den peger på, vil vi på runtime kunne ændre det object til en CardBoardBox.

## Design Patterns:

### Strategy Pattern

Strategy Pattern er defineret som:

*"Define a family of algorithms, **encapsulate** each one, and make them interchangeable. [The] Strategy [pattern] lets the algorithm vary independently from clients that use it." – The Gang of Four Design Patterns.*



PackageType er defineret som en abstract klasse, der indeholder fælles metoder, som subklasser nedarver. Dette bliver udnyttet ved at definere konkrete typer såsom XmasTree, CardBoardBox og PlasticBox, som kan bruge metoder som defineret i PackageType eller overskrive metoderne. Dette gøres, hvis man kræver specielle implementationer af dem. Strategy pattern giver mulighed for at skifte opførsel på runtime. Vi kan ændre PackageType på de objekter, som har referencer til PackageType.

### Singleton Pattern

*"Ensure a class only has one instance and provide a global point of access to it" – The Gang of Four Design Patterns*

```
public class Service {
    private static Service uniqueInstance;
    ...
    /**
     * Static method for getting a Service object
     * and creating one if it doesn't exist
     * @return Service object to be used
     */
    public static Service getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Service();
        }
        return uniqueInstance;
    }
}
```

Service, ServiceGUI, ListDao, JpaDao såvel som alle subklasser af PackageType er lavet som singletons. I tilfældet med PackageType subklasser bruger vi singleton for at være sikker på, at alle aspekter af programmet arbejder med den samme liste indeholdende trucks og ramps. Dette gør sig som sådan også gældende for ListDao, da vi vil være sikker på at Service og ServiceGUI arbejder med de samme elementer. Singleton giver mulighed for, at vi først optager plads i JVM når singleton klassens statiske metode kaldes for at create/returnere et singleton objekt.

## Observer Pattern

*"Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically"* – The Gang of Four Design Pattern

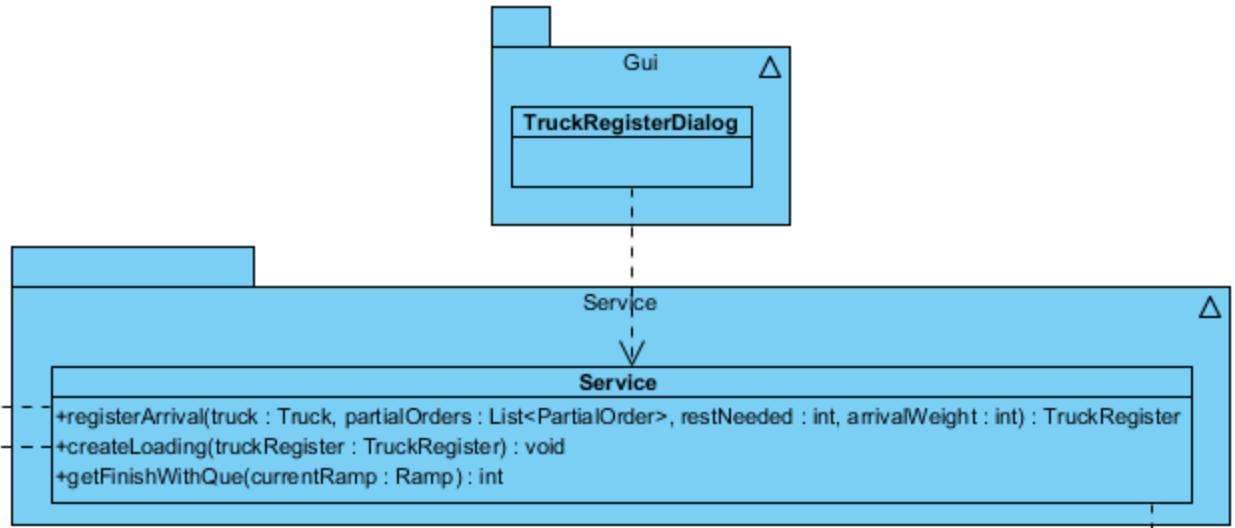
```
public MainFrame() {  
    ...  
    btnDone = new JButton("End Current Loading");  
    ...  
    private class Controller implements ActionListener, ListSelectionListener {  
  
        @Override  
        public void actionPerformed(ActionEvent ae) {  
  
            } else if (ae.getSource().equals(btnDone)) {  
                Ramp ramp = (Ramp) cbbRamps.getSelectedItem();  
                Loading loading = service.getLoadingBeingLoaded(ramp);  
                if(loading != null) {  
                    TruckRegister truckRegister = loading.getTruckRegister();  
                    NotifyChauffeurDialog ncd = new NotifyChauffeurDialog(truckRegister);  
                    ncd.setVisible(true);  
                    if(ncd.isModalResult()){  
                        if (!service.registerDeparture(truckRegister, loading, ramp)){  
                            JOptionPane.showMessageDialog(null, "Truck was sent to get repacked",  
                                "Error", JOptionPane.INFORMATION_MESSAGE);  
                        }  
                        clearTextArea();  
                    }  
                    updateLoadings();  
                }  
            }  
        }  
    }  
}
```

Observer bruges til alle vores listeners i gui delen. Hver gang vi trykker på en knap i et vindue, vil der blive smidt en actionEvent, som systemet fanger og beder om at opdatere, baseret på hvad knappen er programmeret til at gøre.

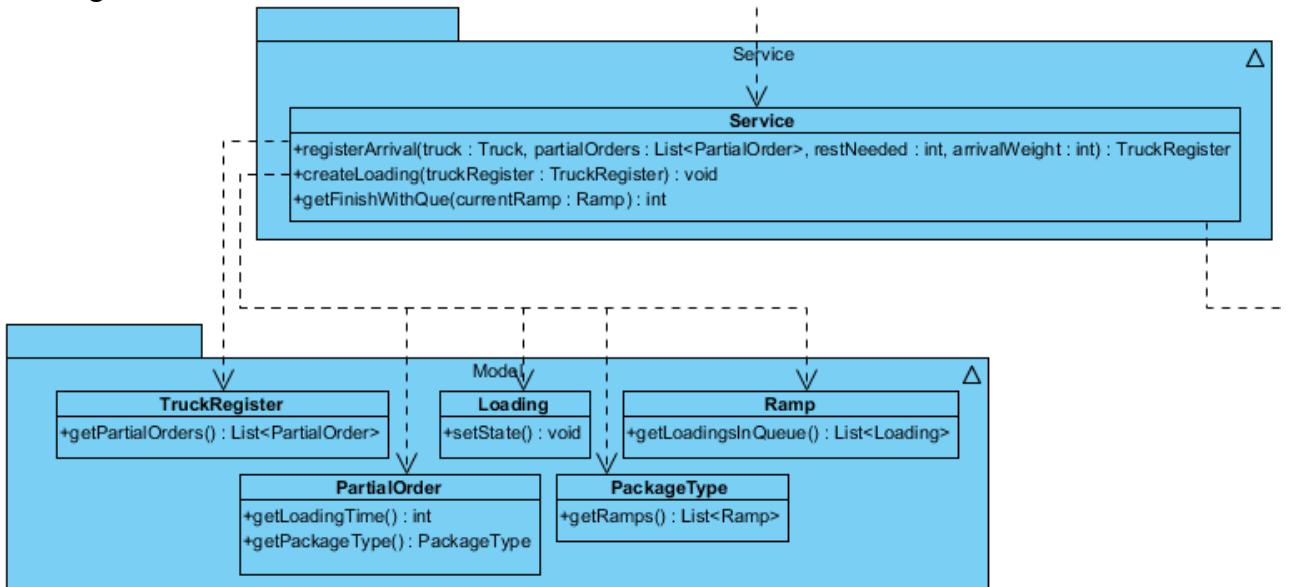
## Arkitektur

Til at beskrive arkitekturen anvendes usecasen Register arrival(UC1). For at kunne registrere en ankomst, skal der være oprettet en Truck samt en Ordre, som har en eller flere PartialOrdres.

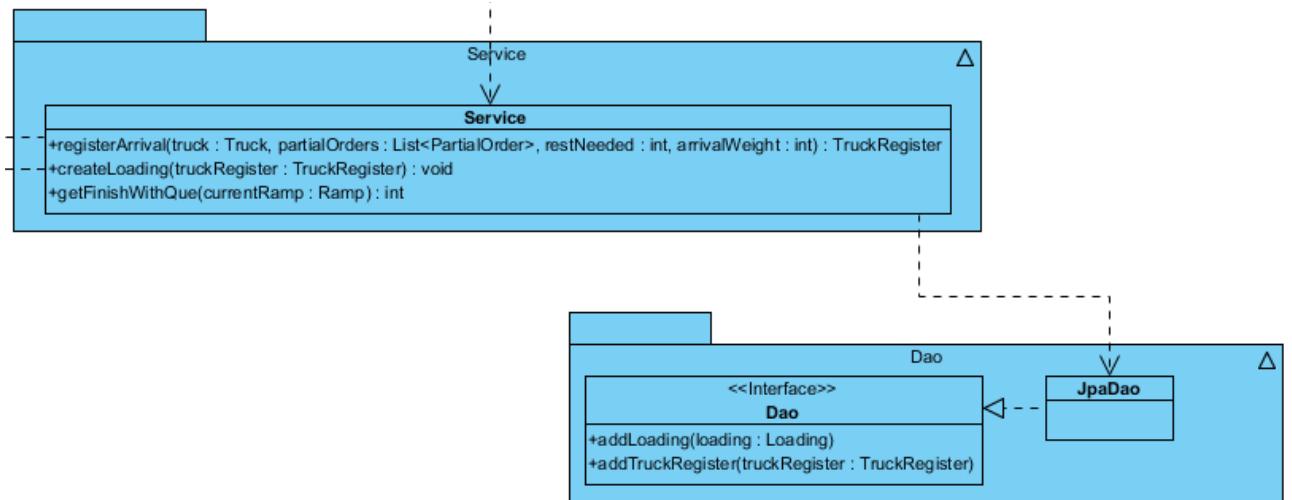
Arkitektur laget Gui er lavet sådan, at TruckRegisterDialogen ikke opretter en TruckRegister, men derimod kalder Service metoden registerArrival() med de nødvendige attributter.



Service håndterer oprettelse og håndtering af objekterne TruckRegister og Loading samt hvilken Ramp den nyoprettede Loading skal placeres på, og hvornår den forventes at starte og slutte.



Service håndterer samtidig de nødvendige kald til Dao efter oprettelse eller redigering af et objekt.



Dao tager sig af alt kommunikation til og fra databasen med objekter som skal hentes fra eller persistes til databasen.

## Fejlhåndtering

Gennem programmet har vi håndteret fejl ved brug af JOptionPane, hvor man får en besked frem om, hvilke steder der har krav til input. Inden vi smider en JOptionPane frem, har vi et tjek på input. Dette gøres blandt andet med regulære udtryk. Regulære udtryk tjekker om det indtastede passer i overensstemmelse med det mønster vi har angivet. Andre steder i programmet valgte i regulære udtryk fra, da vi havde en try-catch med en parseInt i, og hvis den fejler, venter vi bare på en rettelse. Vi vil kort nævne en liste over tegn brugt i mønstret og deres betydning.

- ^ angiver starten på strengen
- \* betyder nul eller flere forekomster af det forgående
- + betyder en eller flere forekomster af det forgående
- \$ angiver slutningen på strengen
- \d betyder et tal
- [a-z] betyder alle tegn fra a-z er gyldige

## Test:

### Modultest

I følgende test tester vi Order klassens metode createPartialOrders(), gennem White-Box testing, derved kan vi se om metoden opretter det rigtige antal PartialOrders.

Først laver vi en række Orders:

Order	Number	Vægt	Margin	Dato	PackageType
o1	1	0 kg	0 kg	05-20-2013	CardboardBox
o2	2	1 kg	0 kg	05-19-2013	XmasTree
o3	3	20000 kg	10 kg	25-05-2013	PlasticBox
o4	4	40000 kg	20 kg	15-11-2013	PlasticBox
o5	5	83117 kg	40 kg	15-05-2013	CardboardBox

Efter oprettelsen af de forskellige Orders, testes der om det rigtige antal PartialOrders er blevet oprettet som vist i en tabel nedenfor:

Order	Metoden som kaldes	Forventet Output	Aktuelt output
o1	getPartialOrders().size()	0	0
o2	getPartialOrders().size()	1	1
o3	getPartialOrders().size()	1	1
o4	getPartialOrders().size()	2	2
o5	getPartialOrders().size()	5	5

.size() kan kaldes på metoden getPartialOrders() fordi denne metode returnere en liste med de tilknyttede PartialOrders.

Selve JUnit testen kan findes på tilhørende cd og i den udprintede kildekode.

## Specielt interessant kode

Metoden registerArrival bruges til at oprette den truckRegister, som indeholder information om hvilke partialOrders der skal hentes af en bestemt truck, og vægten for trucken, samt hviletid i minutter.

```
public class Service {
    /**
     * Method for registering an arrival of a truck
     * and creating a truckRegister with the information and
     * thereby creating a loading from that specific truckRegister
     * @param truck - truck
     * @param partialOrders - List with all the partialOrders the truck should contain
     * @param restNeeded - rest needed of the chauffeur
     * @param arrivalWeight - weight of the truck
     * @return TruckRegister with the information
     */
    public TruckRegister registerArrival(Truck truck, List<PartialOrder> partialOrders, int
restNeeded, int arrivalWeight) {
        TruckRegister tr = new TruckRegister(truck, partialOrders, new GregorianCalendar(),
restNeeded, arrivalWeight);
        dao.addTruckRegister(tr);
        createLoading(tr);
        return tr;
    }
}
```

Efter truckregisteren er oprettet og tilføjet til databasen gennem daos metode .addTruckRegister, bliver den private metode createLoading kaldt. Her bliver der oprettet og skabt association mellem en truckRegister og den mest optimale rampe, samtidig med at den estimerede start og estimerede slut tid bliver beregnet på den specifikke rampe.

```
public class Service {
```

```

    /**
     * Method for searching for the best ramp for a loading without priority
     * @param truckRegister - For getting the information about the time and packagetype
     */
    private void createLoading(TruckRegister truckRegister){
        List<PartialOrder> currentOrders = truckRegister.getPartialOrders();

        int loadingTime = 0;

        for(PartialOrder po : currentOrders){
            loadingTime += po.getLoadingTime();
        }
    }
}

```

Først sammenregner vi hele loadingtiden for alle de partialOrders som truckRegisteren indeholder.

```

    PackageType currentPackageType = currentOrders.get(0).getPackageType();

```

Derefter ved vi en truckRegister kun kan indeholde en bestemt PackageType. Denne gemmes så i en lokal variabel af typen PackageType, som gennem polymorfi kan indeholde alle de forskellige subtyper af PackageType.

```

        List<Ramp> ramps = currentPackageType.getRamps();

```

Når vi så har packageType kan vi uden problemer få fat i alle de tilhørende ramper, da navigationen mellem Ramp og PackageType er dobbeltrettet.

```

        Ramp pickedRamp = null;

```

Den rampe som vi ender med at oprette loadingen med

```

        int bestTimeInMin = Integer.MAX_VALUE;

```

En lokal variabel til at sammenligne sluttiden for hver enkelt rampe

```

        boolean foundEmptySlot = false;
        int index = 0;

        while(index < ramps.size() && !foundEmptySlot){

```

De ramper skal så bruges til gennemsøgning for at finde det bedste spot til den loading som skal oprettes og indsættes. Denne søgning er linær og vil umiddelbart fortsætte indtil der findes en tom rampe eller at der ikke er flere ramper at gennemsøge

```

            Ramp currentRamp = ramps.get(index);

            int queueDoneInMin = 0;

            queueDoneInMin += getFinishWithQueue(currentRamp);

```

En anden service metode kaldes for at udregne den samlede tid for hvornår en bestemt rampe er færdig med sin kø.

```

                if(queueDoneInMin == 0){
                    bestTimeInMin = queueDoneInMin;
                    foundEmptySlot = true;
                    pickedRamp = currentRamp;

```

Hvis tiden er nul, må rampen nødvendigvis være tom, deraf har vi fundet en tom plads og vi stopper søgningen, samtidig med at gemme den bedste tid, som selvfølgelig er nul og den rampe vi ser på nu.

```

                }else if(queueDoneInMin < bestTimeInMin){
                    bestTimeInMin = queueDoneInMin;
                    pickedRamp = currentRamp;

```

Hvis tiden er større end nul og bedre end den bedste tid vi har gemt i forvejen, overskriver vi den bedste tid og gemmer den nye rampe.

```

            }
            index++;

```

```

    }

    if(pickedRamp != null){
        Calendar currentTime= new GregorianCalendar();
        currentTime.add(Calendar.MINUTE, bestTimeInMin);

```

Starttiden må være systemets aktuelle tid, adderet med den bedste tid som vi fandt i vores søgning.

```

        Calendar estimatedFinish = (GregorianCalendar) currentTime.clone();
        estimatedFinish.add(Calendar.MINUTE, loadingTime);

```

Sluttiden er så starttiden adderet med den loadingtid, som vi har beregnet med gennemløb mellem alle partialOrders.

```

        Loading l = new Loading(currentTime, estimatedFinish, truckRegister, pickedRamp);

```

Derefter oprettes der en loading, indeholdende de informationer vi har fundet og regnet os frem til, hvor loading så automatisk bliver sat ind i rampens List af loadings.

```

        if(foundEmptySlot){
            l.setState(State.BEINGLOADED);
        }
        dao.addLoading(l);
    }else {
        throw new RuntimeException("The specific loading cant not be created");
    }
}
}

```

Hvis den pågældende rampe var tom, vil statussen blive sat til BEINGLOADED, istedet for dens default init som er QUEUE.

Metoden getFinishWithQueue(Ramp ramp) er en private hjælpemetode, der bruges til beregning af tiden, det tager for en specifik rampes kø at blive færdig, returneret i en long værdi.

```

public class Service {
    /**
     * Method for getting the time, for a specific ramp
     * to be done with its queue
     * @return long, minutes till done with whole queue
     */
    private long getFinishWithQueue(Ramp ramp){
        long time = -1;
        Calendar currentTime = new GregorianCalendar();
        List<Loading> loadings = ramp.getLoadingsInQueue();
    }
}

```

På en rampe har vi en linær søgning som gennemgår alle loadings på en rampe som har en status der er enten BEINGLOADED, PRIORITY eller QUEUE.

```

    if(loadings.isEmpty()){
        time = 0;
    }else {
        Loading l = loadings.get(loadings.size()-1);

        long start = TimeUnit.MILLISECONDS.toMinutes(currentTime.getTimeInMillis());
        long finish = TimeUnit.MILLISECONDS.toMinutes(l.getEstimatedFinish().getTimeInMillis());

        time = finish-start;
    }
}

```

Ellers er tiden baseret på afstanden mellem den aktuelle system tid og den sidste loading som er placeret i listen, det udregner vi så i minutter, ved hjælp af TimeUnit fra javas standard biliotek.

```

    }
    return time;
}

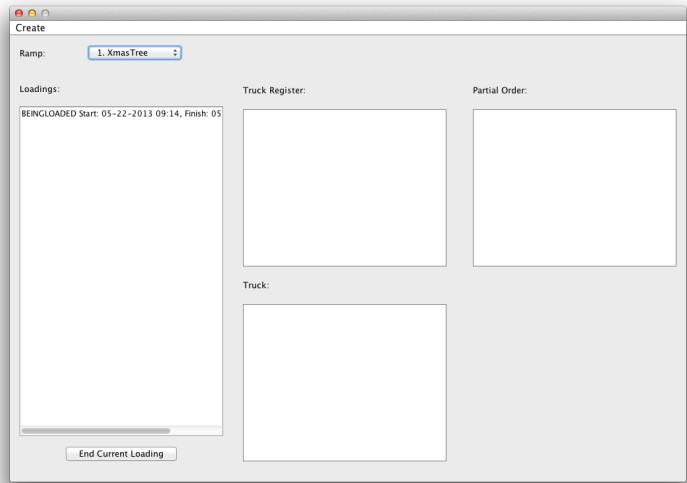
```

}

## Guided Tour:

### Register Arrival:

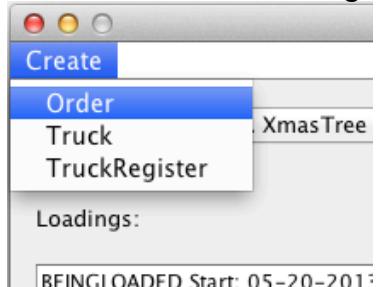
Vi har valgt at lave en tour over registrering af en ankomst, hvor vi kommer ind på at lave en ordre og lastbil. Når programmet er startet ser man følgende billede.



Billedet viser en menubar, hvori man kan navigere mellem de forskellige vinduer, som skal anvendes til at oprette en ordre eller en lastbil. Det er også muligt at registrere en ankomst, hvilket vi modellerer gennem TruckRegister.

I vinduet kan man vælge en ramp og se alle tilknyttede løsninger med deres forventede start- og sluttidspunkt. Hvis brugeren vælger en løsning i listen vil der blive vist information omkring lastbilen, hvornår den ankom og hvilke delordre som den skal have med.

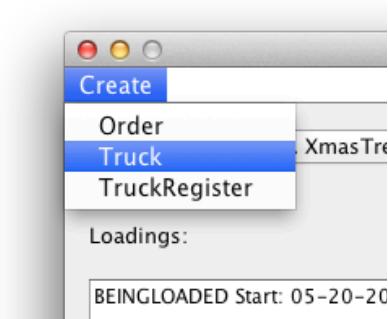
For at oprette en ordre skal man åbne menuen "Create" og vælge "Order".



Derefter vises vinduet "Create Order".



I dette vindue skal man indtaste den ønskede information omkring en ordre, eks. Hvad den skal veje, hvornår den ønskes afhentet og af hvilken type ordren skal være. Efter oprettelse af en ordre, vises det første vindue igen. Det er også muligt at oprette en lastbil. Dette sker ved at åbne menuen "Create" og vælge "Truck".

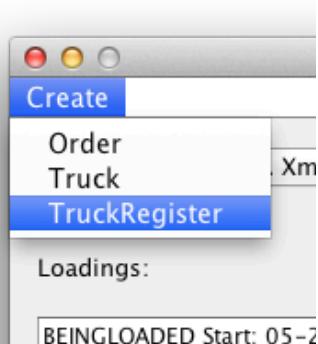


Derefter vises vinduet "Create Truck".

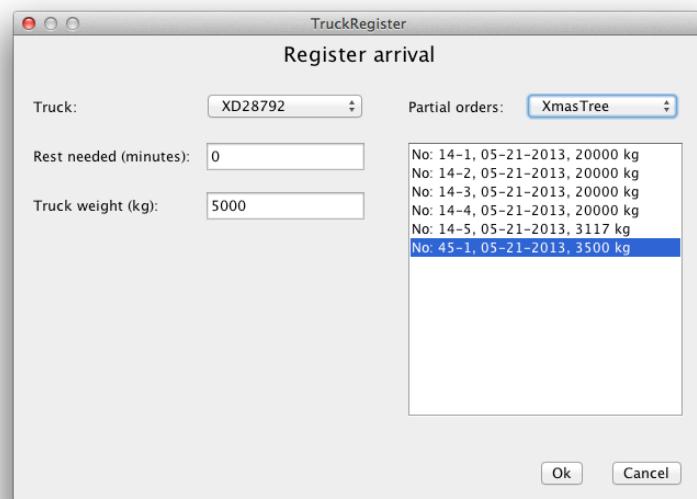


I dette vindue skal man indtaste den ønskede information omkring en lastbil, eks. Registrerings nummer, chaufførens navn og telefon nummer samt vælge hvilke typer ordre som den skal kunne indeholde.

Når der både er oprettet en ordre og en lastbil kan lastbilen registreres ankommet og få tilknyttet en eller flere delordrer. Dette gøres ved at vælge menuen "Create" og derefter "TruckRegister"



Derefter vises vinduet "TruckRegister".

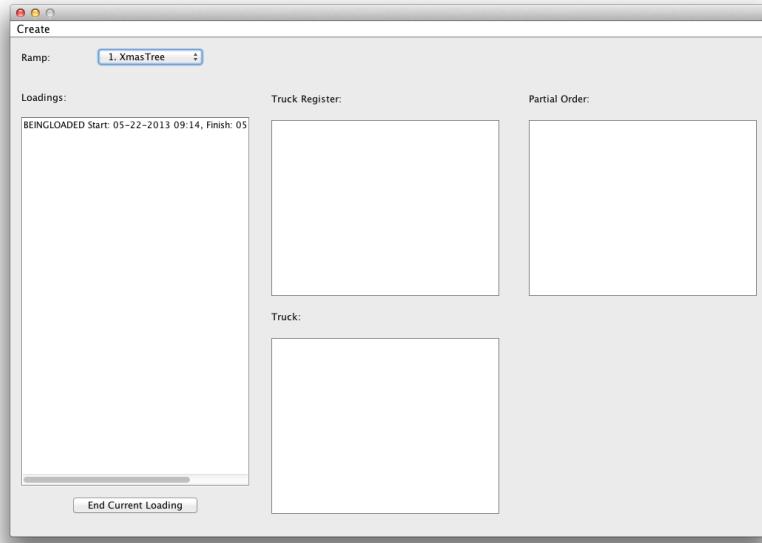


I dette vindue skal man vælge registrerings nummeret på den ønskede lastbil, indtaste den behøvende hvile før man må køre igen, det tidspunkt hvorpå man ankom, samt vægten på lastbilen. Der skal også vælges hvilke delordre som lastbilen skal medtage.

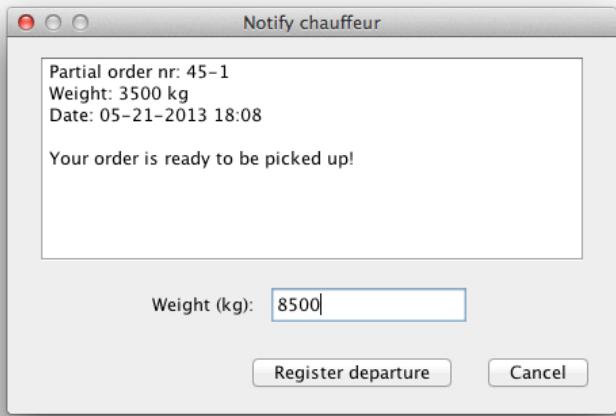
Efter at ankomsten af en lastbil er blevet registreret, har programmet lavet en læsning som er blevet tilføjet til den rampe hvor den vil være færdig hurtigst.

#### *Register Departure:*

Vi har valgt at lave en tour over registrering af en afgang. Dette starter i samme vindue som registrering af ankomst.



Når en læsning er færdig trykkes der på knappen ”End current loading”, som åbner vinduet, der skal symbolisere en besked til chaufføren.



Dette vindue viser en besked som chaufføren skal få når han kan hente hans lastbil, den viser en liste over de delordre som er blevet læsset i lastbilen. Vægten på lastbilen inklusiv ordre skal registreres sådan at den forbliver inden for den påsatte marge. Når vægten er indtastet er det muligt at trykke ”Register departure”.

## **Konklusion**

Igennem vores virksomhedsanalyse har vi udviklet et system som vil optimere flowet med læsning af lastbiler. Vi føler os sikre på, at efter en kort testperiode hos Danish Crown i Horsens vil vores logistiksystem kunne udvides til andre slagterier. Systemet kommer til at være behjælpeligt, da Danish Crown vil slippe for manuelt arbejde ved udvalg af rampe samt beregning af start og sluttid af en læsning.

## Kildeliste

1. Danish Crown Årsrapport 9/10
2. Danish Crown årsrapport 11/12
3. danishcrown.dk
  - <http://www.danishcrown.dk/Forbruger/Fakta-eller-poelsesnak/Fakta.aspx>
4. Danish Crown årsrapport 11/12
5. Smithfield årsrapport
6. Smithfield årsrapport
7. Danish Crown årsrapport 11/12 & danishcrown.dk
  - <http://www.danishcrown.dk/Forbruger/Fakta-eller-poelsesnak/Fakta.aspx>
  - <http://aarsrapport2012.danishcrown.dk/#/feb/historier/dc-er-lokomotiv-for-dansk-eksport/>
8. danishcrown.dk
  - <http://www.danishcrown.dk/Forbruger/Fakta-eller-poelsesnak/Fakta.aspx>
9. Smithfield årsrapport
10. tican.dk
  - <http://www.tican.dk/om-tican>