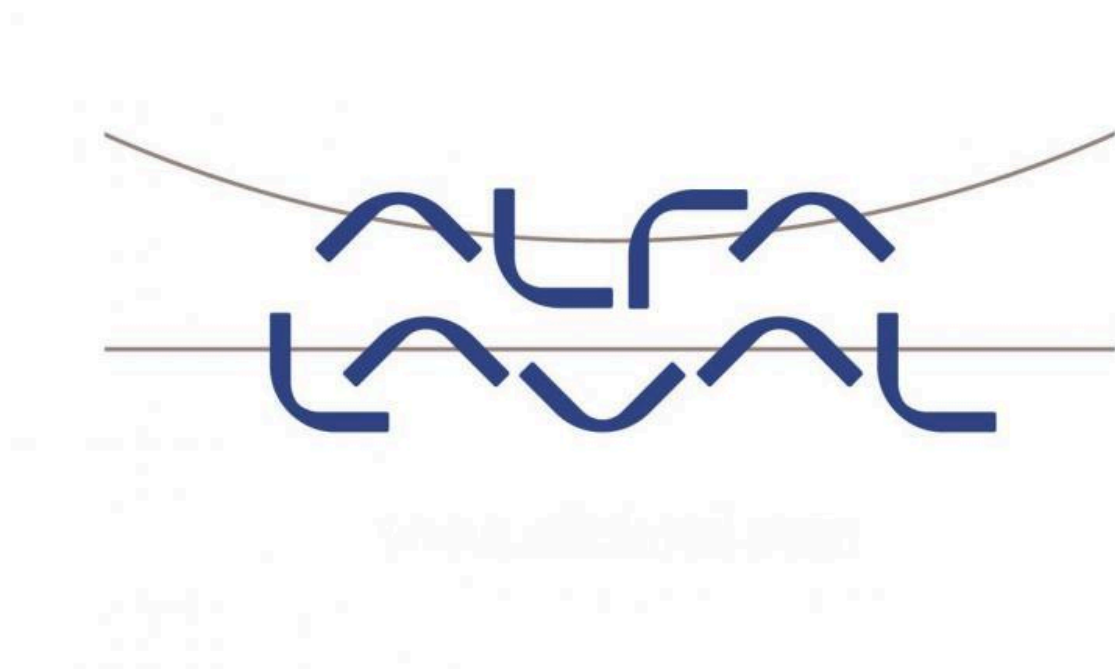


# Ferieplanlægningssystem til Alfa Laval Kolding



Eksamensprojekt skrevet af Gruppe 6:

Jesper Helstrup, Jeppe Kjeldgaard Lynge Hansen, Rafed Ali Beier, Jakob Baggesgaard  
Seeberg, Jonathan Adam Ricketts Lind

Antal anslag: 47579

<b>Indledning inkl. Problemstilling.....</b>	<b>3</b>
<b>Projektorganisering.....</b>	<b>4</b>
Team og samarbejde.....	4
Unified Process (UP).....	4
SCRUM.....	5
PlantUML.....	6
GitHub.....	6
Draw.io.....	7
<b>Design- og udviklingsprincipper.....</b>	<b>7</b>
<b>Artefakter.....</b>	<b>9</b>
Forretningsmodeller.....	9
Business Process Model Notation.....	9
Use-cases.....	10
Objekt- og domænemodeller.....	12
SSD.....	14
OC.....	15
Wireframes.....	16
DCD.....	17
SD.....	20
Databasemodel.....	21
<b>Implementering.....</b>	<b>23</b>
Designmodel og designmønster.....	23
Database.....	24
Business Logic (bedre forslag til overskrift?).....	27
Øvrig fremhævet kode (hvis nødvendig).....	27
Unit Testing.....	27
Brugergrænseflade.....	29
Algoritme.....	31
<b>Løsning som indeholder artefakter.....</b>	<b>39</b>
<b>Konklusion (refleksion og perspektivering).....</b>	<b>41</b>
<b>Bilag.....</b>	<b>44</b>

# Indledning inkl. Problemstilling

Alfa Laval er en førende global leverandør af førsteklassesprodukter inden for varmeoverførsel, separation og væskehåndtering. Alfa Laval har mere end 100 servicecentre med kapacitet til at levere service i over 160 lande. I forbindelse med dette projekt har vi samarbejdet med Alfa Laval Kolding afdelingen, mere specifikt deres produktionsafdeling med over 60 ansatte.

Som global virksomhed med fokus på effektivisering, benytter Alfa Laval en outdated approach til at håndtere ferieplanlægningen af deres medarbejdere.

Vi har derfor været i dialog med dem om at effektivisere deres ferieregistrerings- og planlægningsmodel til fordel for en tidsbesparende og økonomisk fordelagtig digital version. De har tidligere brugt op imod 4 ugers forberedelse til ferieplanlægningen, hvilket koster deres ledere dyrebare timer. Det har været vores hovedfokus at afhjælpe denne problematik.

Alfa Laval Kolding ønsker at effektivisere ferieplanlægningen for deres 65 lageransatte, da deres nuværende manuelle system er tidskrævende og indebærer risiko for fejl og bekymring om forskelsbehandling. Som processen ser ud lige nu, afleverer medarbejderne ferieønsker på papir, hvor de prioriterer op til tre perioder, som ledelsen derefter indtaster i Excel. På baggrund af disse, forsøger lederne at planlægge en ferieplan med hensyntagen til kompetencekrav og medarbejdernes tidligere ferieønsker.

Den ønskede digitale løsning skal reducere tidsforbruget, sikre retfærdighed og bevare de nødvendige kompetencer og bemanding på lageret under ferieperioder. Systemet skal bestå af et brugervenligt interface og baseret på en algoritme, der tager højde for medarbejdernes tidligere og nuværende ferieønsker samt virksomhedens kompetence- og bemandingskrav for ferieperioden, bidrage med forslag til en samlet ferieplan og håndtere detaljer som enkelte feriedage.

# Projektorganisering

## Team og samarbejde

Vores projekt-team består af fem personer, som alle har bidraget aktivt til både udviklingen af løsningen og udarbejdelsen af de nødvendige artefakter. Vi valgte en fleksibel rollefordeling, hvor alle deltog i forskellige aspekter af projektet. Denne tilgang sikrede, at vi udnyttede hinandens styrker og kompetencer bedst muligt.

Kommunikation var en vigtig del af vores samarbejde, og vi brugte primært Microsoft Teams som platform. Hvis vi stødte på problemer uden for møderne, benyttede vi os af den fælles gruppechat eller kontaktede hinanden direkte. Denne tilgang sikrede hurtige løsninger og understøttede fremdrift i projektet.

Samarbejdet har været præget af åbenhed og ansvarlighed. Vores fleksible tilgang til roller gjorde det nemt for os at hjælpe hinanden og lære af hinandens arbejde. Samtidig sikrede vores faste møder og strukturerede kommunikation, at vi alle arbejdede mod det samme mål og kunne tilpasse os hurtigt, hvis der opstod ændringer undervejs.

## Unified Process (UP)

Vi valgte at følge Unified Process i vores projekt, hvilket betyder, at vi arbejdede gennem forskellige faser for at sikre en struktureret tilgang til vores arbejde.

**Forberedelsesfase:** I samarbejde med Alfa Laval Kolding udarbejdede vi en målbeskrivelse, hvor vi samlede vores idéer og fastsatte projektets omfang og mål. Dette gav os en klar retning og forståelse af, hvad slutresultatet skulle være, samtidig med at vi sikrede, at projektet opfyldte Alfa Lavals behov.

**Etableringsfase:** Vi arbejdede med at samle krav ved at udarbejde use cases, som var tæt knyttet til de udfordringer, Alfa Laval Kolding stod overfor. Disse use cases hjalp os med at identificere og prioritere de funktionelle krav til systemet, så vi sikrede, at vi havde fokus på de mest relevante behov.

**Konstruktionsfase:** I konstruktionsfasen arbejdede vi intensivt med systemets arkitektur og design. Vi brugte modelleringsværktøjerne fra UML til at skabe objektmodeller, domænemodeller, systemsekvensdiagrammer (SSD), operationskontrakter (OC), wireframes, design-class diagrammer (DCD) og sekvensdiagrammer (SD). Disse modeller hjalp os med at visualisere systemets struktur og sikre, at de enkelte dele af systemet var godt definerede og integrerede.

**Fordele og udfordringer:** Brugen af Unified Process har givet os en struktureret ramme for at planlægge og gennemføre projektet systematisk. Dette har hjulpet os med at holde styr på projektets fremdrift og sikre, at vi ikke overså vigtige detaljer. En udfordring har været at finde balancen mellem struktur og fleksibilitet, men ved løbende at evaluere vores fremskridt har vi kunnet justere vores tilgang og sikre effektivitet.

## SCRUM

Vi brugte SCRUM som agil projektstyringsmetode, hvilket gav os en struktureret og fleksibel tilgang til at håndtere opgaver. Hver uge startede vi med et planlægningsmøde, hvor vi diskuterede og besluttede, hvilke opgaver der skulle løses i den kommende uge. Disse opgaver blev udledt af vores use cases og prioriteret på baggrund af deres relevans og kompleksitet. Vores sprints varede typisk én uge, fra tirsdag til tirsdag, men vi var også åbne for at forlænge dem, hvis opgaverne viste sig at være mere omfattende eller krævede yderligere afklaring.

En vigtig del af vores SCRUM-proces var de daglige stand-up møder, hvor vi hver især delte, hvad vi havde arbejdet på dagen før, hvad vi planlagde at gøre i dag, og om der var nogen forhindringer, vi havde brug for hjælp til at overvinde. Disse møder hjalp med at sikre, at vi hele tiden var på samme side og hurtigt kunne adressere eventuelle problemer.

Efter hver sprint holdt vi sprint reviews, hvor vi præsenterede det arbejde, vi havde færdiggjort, for hinanden og eventuelt også for interessenter. Vi fulgte op med retrospektive møder, hvor vi reflekterede over sprintets forløb og diskuterede, hvad der fungerede godt, samt hvordan vi kunne forbedre processen i de kommende sprints.

Vi udpegede ikke formelt specifikke roller som Scrum Master eller Product Owner. I stedet valgte vi en mere fleksibel tilgang, hvor alle var en del af udviklingsteamet og bidrog til planlægningen og styringen af opgaverne. I vores gruppe var vi gode til at coache hinanden og sikre, at vi fokuserede på de vigtigste opgaver. Den fælles tilgang styrkede både vores samarbejde og engagement, hvilket gjorde det lettere at fastholde fokus på vores overordnede mål. Derudover tog vi fælles teten under vores bi-weekly møder med Alfa Laval Kolding, hvilket sikrede en god kommunikation med dem og gjorde det nemmere at holde dem opdateret om vores fremskridt. Et fælles samarbejde i disse møder bidrog til at styrke vores samarbejde med Alfa Laval Kolding og sikre, at vi kunne tilpasse vores arbejde til deres behov.

SCRUM hjalp os med at holde fokus og strukturere vores arbejde gennem **korte, iterative sprints**. Det fremmede samarbejde og ansvarlighed i gruppen, da alle var involveret i planlægningen og gennemførelsen af sprintopgaverne.

## PlantUML

Som hjælp til at illustrere (og dokumentere) vores modeller har vi brugt PlantUML, som er en simpel tekst til model generator som anvender UML struktur. PlantUML gjorde det muligt for os at skrive diagrammer direkte i tekstform ved hjælp af et letforståeligt syntaks, hvilket sikrede en hurtig og fleksibel proces, når vi skulle lave eller ændre vores modeller.

Vi anvendte PlantUML til at skabe flere forskellige typer diagrammer. Primært DCD'er, men også databasemodeller, SD'er og SSD'er. En af de store fordele ved PlantUML var, at vi kunne anvende de genererede diagrammer både som det færdige output og som en inspirationskilde til at forbedre vores modeller.

## GitHub

GitHub har spillet en central rolle i vores projektorganisering. Vi brugte GitHub Projects til at holde styr på vores opgaver ved at oprette et projektboard med fire kolonner: *To do*, *In Progress*, *Pending Review* og *Done*. Hver opgave startede i *To do*, hvorefter vi diskuterede, hvem der havde lyst til at tage opgaven, og oprettede en branch til denne specifikke opgave. Når arbejdet var påbegyndt, blev opgaven flyttet til *In Progress*. Når opgaven var færdig, rykkede vi den til *Pending Review*, hvor vi i fællesskab gennemgik ændringerne. Dette betød at alle fik mulighed for at se hvilke opdateringer der blev lavet og at vi kunne snakke om udførelsen inden det blev pushet til main branch. Hvis alt var i orden, blev opgaven flyttet til *Done*.

Vi anvendte også GitHub issues til at strukturere vores opgaver. Når vi havde oprettet en task, blev denne konverteret til et issue, hvilket gjorde det nemt at knytte opgaven til en branch og senere lave pull requests. Issues blev desuden forsynet med mærkater som *prerequisite* og *follower* for at vise deres indbyrdes afhængighed og vigtighed. Dette system gjorde det nemt at prioritere opgaver og sikre, at intet blev overset.

Vores branching-strategi var baseret på feature branching. Vi havde en stabil *master*-branch, der kun blev ændret, når koden fra de enkelte feature-branches var blevet grundigt gennemgået og godkendt. For hver opgave oprettede vi en ny branch, som blev dedikeret til den specifikke opgave. Når arbejdet på branchen var færdigt, blev der oprettet en pull request, som vi gennemgik i fællesskab under vores daglige møder. Dette gjorde det muligt for hver person at forklare deres kode og sikre en fælles forståelse af løsningerne. Når pull requesten blev godkendt, blev den merged til *master*, hvilket gjorde det nemt at holde denne branch stabil og konfliktfri.

Vi havde også en klar struktur for commit-beskeder, som gjorde det nemt at forstå ændringerne i projektet. Hvis vi tilføjede en ny funktion, startede commit-beskeden altid med "feat". Hvis ændringen var relateret til designforbedringer, brugte vi "beautify". Denne standardiserede tilgang gjorde det nemt at skelne mellem nye funktioner og designændringer, hvilket forbedrede overblikket over projektets udvikling.

Endelig gav GitHub os en effektiv og struktureret måde at samarbejde på. Gennem GitHub Project kunne vi holde styr på opgavernes status, hjælpe hinanden, når nogen stødte på udfordringer, og dele viden gennem kodegennemgang. Dette styrkede ikke blot vores teamwork, men forbedrede også vores individuelle færdigheder.

## **Draw.io**

Ud over GitHub brugte vi Draw.io til at skabe vores diagrammer og artefakter. Draw.io var et væsentligt værktøj for os, da det gjorde det nemt at visualisere komplekse systemstrukturer og interaktioner på en klar og overskuelig måde.

Ved hjælp af Draw.io kunne vi udarbejde use cases, som hjalp os med at beskrive de forskellige funktioner, systemet skulle have, samt hvordan brugerne ville interagere med det. Wireframes blev brugt til at designe og visualisere brugergrænsefladen, hvilket sikrede, at vi havde en intuitiv og brugervenlig applikation. Wireframes gav os også mulighed for at præsentere designforslag for Alfa Laval Kolding og indhente feedback, hvilket styrkede samarbejdet og sikrede, at produktet ville kunne leve op til deres forventninger.

Draw.io gjorde det muligt for os hurtigt at ændre og tilpasse vores diagrammer og modeller, når projektets behov ændrede sig, eller nye krav blev identificeret. Dette gjorde det muligt for os at holde vores dokumentation opdateret og sikre, at alle gruppemedlemmer havde en fælles forståelse af systemets design og funktionalitet. Vi brugte også Draw.io som et fælles referencemateriale under møder og sprintplanlægning, hvilket hjalp med at skabe en fælles visuel ramme, som alle kunne forholde sig til.

## **Design- og udviklingsprincipper**

FURBS+ og SMART har været centrale principper i udviklingen af vores løsning. Vi har gennem hele projektet brugt FURBS+ til at sikre, at vores software er funktionel, brugervenlig og pålidelig. Brugervenlighed har særligt været et fokusområde, da det var et eksplicit ønske fra Alfa Laval. Med udgangspunkt i wireframes designede vi således en overskuelig grænseflade, der undgik at have for meget information på skærmen på én gang. Erfaringer fra tidligere projekter hjalp os med at holde vinduer overskuelige og intuitive.

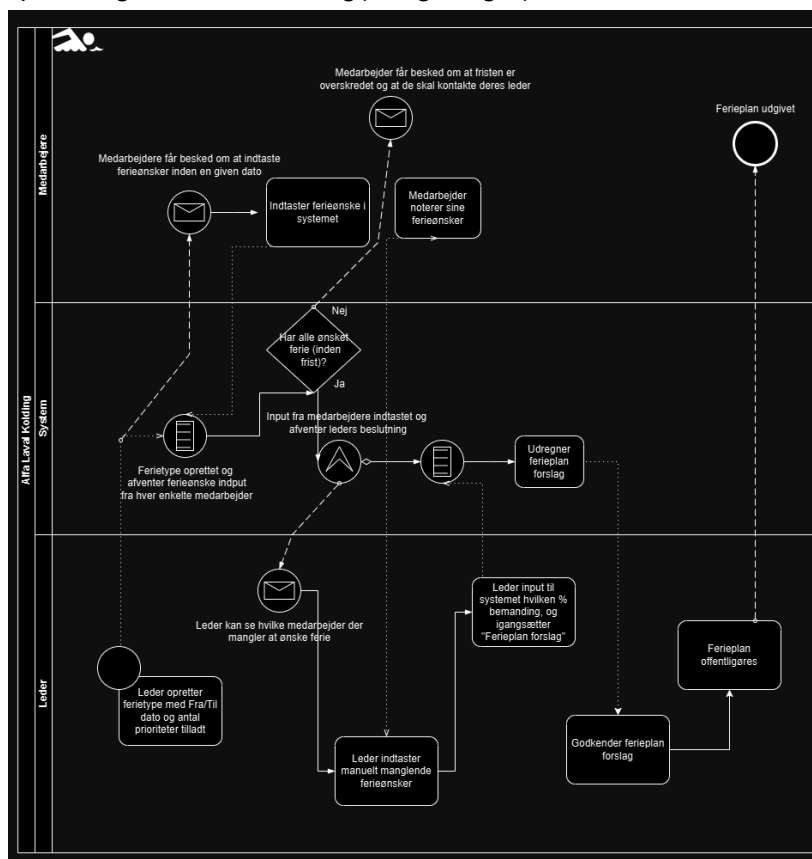
SMART-principperne hjalp os med at styre projektet ved at opstille målbare og klare mål, eksempelvis ved implementeringen af repositories, som blev færdiggjort indenfor en fastlagt tidsramme. Gennem denne tilgang har vi fra start til slut haft fokus på både Alfa Lavals krav og en brugervenlig og holdbar løsning.

# Artefakter

## Forretningsmodeller

### Business Process Model Notation

Vi arbejdede med Business Process Model Notation, for at få en bedre forståelse og overblik af Alfa Laval's forretningsproces ifm. ferieplanlægning. Da Alfa Laval på nuværende tidspunkt har meget manuelt arbejde i form af nedskrivning på fysiske sedler, indskrivning af ferieønsker i excel og sammenligningen med tidligere ferieønsker fra forrige års excel ark, mente vi at BPMN kunne skabe en forståelse af den nuværende proces men også illustrere vores forventede ændring i forretningsprocessen. Tidsoptimeringen var den største ændring i vores løsningsforslag sammenlignet med deres nuværende system. Dette var vanskeligt at repræsentere i vores BPMN, men tilføjelsen af systemet som swimlane, hjalp med at vise hvordan det tidligere tidskrævende manuelle arbejde bliver overladt til systemet. Ved hjælp af en BPMN for den nuværende (Se Bilag 5) og foreslåede forretningsproces (se nedenstående figur), fik vi et bedre overblik over subprocesser. Med overblik og forståelse for hvilke forretningsprocesser der igangsatte andre subprocesser, kunne vi arbejde imod at eliminere de mest tidskrævende processer fremadrettet i udviklingen af vores system. Model i større opløsning kan findes i bilag (Bilag 5 og 6).





## Use-cases

Da det stod klart, at et digitalt ferieplanlægningssystem ville kunne afhjælpe betydelige udfordringer på Alfa Lavals lager i Kolding, holdt vi et indledende møde med tre relevante aktører fra virksomheden: En warehouse-manager, som ville være potentiel førstehandsbruger af systemet, en it-kyndig projektleder, som ville fungere som sparringspartner ift. den tekniske udformning af systemet, og en team-manager, som ultimativt var Alfa Lavals beslutningstager på projektet. At have adgang til forskellige aktører, der hver især havde deres egne separate fokusområder og interesser, gav os enormt gode forudsætninger for hurtigt at danne os et overblik over de forskellige aspekter, der var på spil for Alfa Laval. På baggrund af dette møde havde vi således de vigtigste krav for systemet på plads tidligt i processen. Uden at gå for meget i dybden med disse, blev det for eksempel gennem den direkte samtale med warehouse-manageren klart, at et simpelt og brugervenligt UI skulle prioriteres højt, ligesom hun kunne beskrive den nuværende arbejdsproces for os og herunder hvilke frustrationer denne medførte. Alt i alt gav dette møde os et godt udgangspunkt for næste punkt i vores backlog: Udarbejdelse af use-cases.

Da vi på baggrund af ovenstående møde havde fået beskrevet forskellige scenarier i ferieplanlægningsprocessen, besluttede vi os for ikke at udarbejde scenarier skriftligt, men i stedet springe direkte til use-cases (selvom vi senere i processen gjorde brug af objektmodeller og i den forbindelse mundtligt diskuterede forskellige konkrete scenarier).

En oversigt over alle use-cases kan ses i bilag 1, men som eksempel fremhæves:

### **Use-case 2: Organisér medarbejderdata som forberedelse til ferieplanlægningen**

En leder på lageret organiserer virksomhedens information om lagermedarbejderne, herunder deres kompetencer og ferieønsker, på en overskuelig måde. Hvis nogle informationer ikke er tidssvarende eller på anden vis fejlbehæftede, tilpasses disse.

Denne use-case eksemplificerer vores generelle tilgang til use-cases: En brief use-case, der beskriver en central proces, som vores system skal kunne håndtere. Ifm. ferieplanlægningssystemet er den samlede proces forholdsvis klar fra start til slut, og der er således ikke forgrenede beslutningsprocesser eller mange alternative løsninger, der skal overvejes, og brief use-cases virkede derfor hensigtsmæssige som værktøj til internt i projektgruppen at fastlægge hvilke centrale processer, der var vigtigst at gå i dybden med.

Udover use-casenes funktion som beskrivende for vores system, brugte vi også vores use-cases som udgangspunkt for vores sprints, og i den sammenhæng viste use-case 2 sig særligt at være problematisk, i og med at de tre navneord i use-case 2, lagermedarbejdere, kompetencer og ferieønsker, udgør tre forskellige konceptuelle klasser i vores domæne og i øvrigt udgør størstedelen af modellaget i vores program. Denne use-case endte derfor med at strække sig over mange uger, og i bagklogskabens lys er det tydeligt for os, at det havde været fordelagtigt at bryde use-casen ned i tre separate sprints: én for hver af de tre konceptuelle klasser.

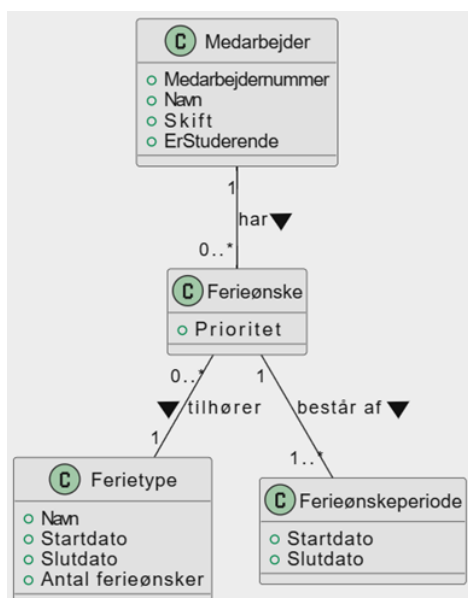
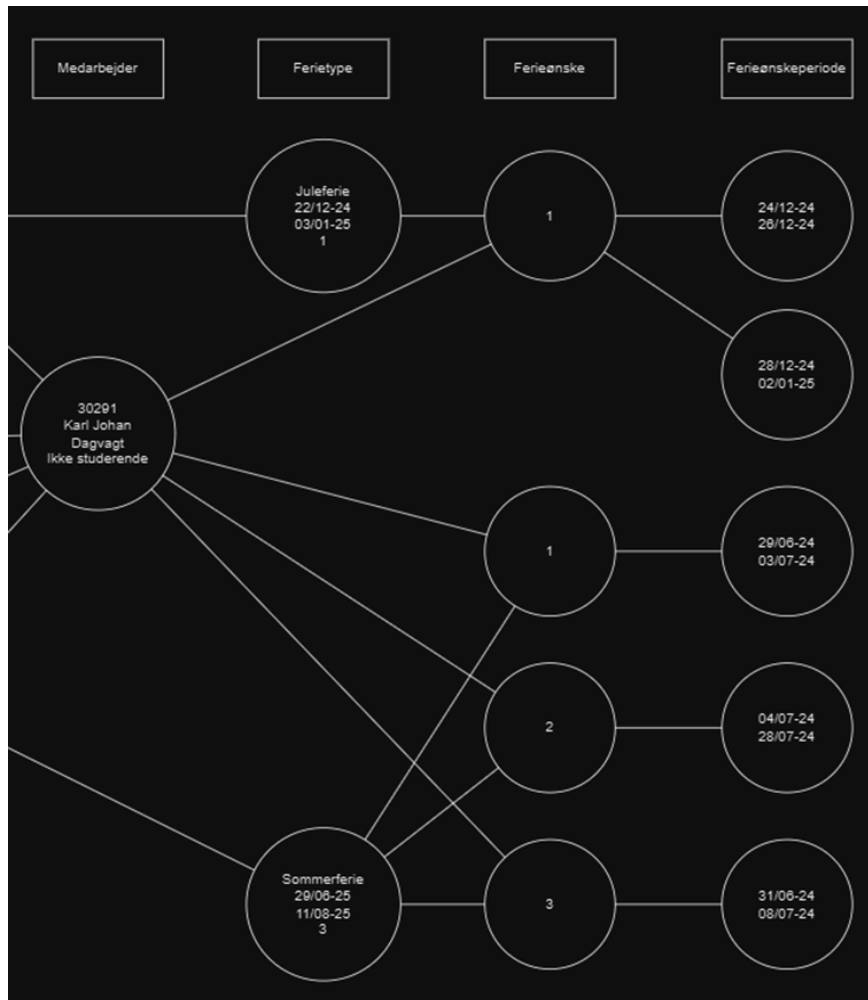
Ligeledes vil vi i resten af rapporten, når vi refererer til use-case 2, kun fokusere på vores proces relateret til en enkelt af de tre konceptuelle klasser. Dette gør vi, da beskrivelsen af arbejdsprocessen med hver af de tre klasser vil ligne hinanden til forveksling, og for at undgå unødigt gentagelse fokuserer vi således herfra på **ferieønske**-klassen. Denne tænkte use-case kan således læses som:

### **Use-case 2: Organiser medarbejdernes ferieønsker som forberedelse til ferieplanlægningen**

En leder på lageret organiserer virksomhedens information om lagermedarbejdernes ferieønsker på en overskuelig måde. Hvis nogle informationer ikke er tidssvarende eller på anden vis fejlbehæftede, tilpasses disse.

Som en sidste note til use cases kan det kort nævnes, at hvis man sammenligner vores samlede use cases i bilag 1 med den fulde funktionalitet i vores program, vil man se, at ikke alle funktioner dækkes i vores use cases. Dette er et resultat af, at ønskerne fra Alfa Laval relativt sent i vores arbejdsproces ændrede sig ift. hvordan de forestillede sig at integrere vores system på arbejdspladsen. De forhørte sig eksempelvis om, hvorvidt det var muligt for os at tilføje en side dedikeret til indtastning af ferieønsker, hvor lagermedarbejderne selv ville agere primær aktør, frem for at indsamlingen af disse hørte under lederens ansvarsområde. Vi gik med til at inkludere dette under forudsætning af at det gav mening for os inden for vores tidsramme. **En sen tilføjelse som denne er således ikke repræsenteret i hverken use-cases eller modeller – både pga. tidspres, men også fordi det ikke ændrede på den centrale stamdata internt i systemet.**

## Objekt- og domænemodeller



Begge modeller er skåret til, så det kun er de konceptuelle klasser, der har en relation til ferieønske-klassen, der er taget med. De komplette modeller kan findes i bilag 2 og 3. Modellerne her er vist på samme side for at illustrere sporbarheden modellerne imellem.

Som tidligere nævnt formulerede vi, på baggrund af vores noter fra samtalen med Alfa Laval, mundtligt nogle scenarier, som vi brugte som udgangspunkt for vores objekt- og domænemodeller. Ovenstående modeller viser således fire forskellige konceptuelle klasser, der repræsenterer stamdata i vores system.

Objektmodellen blev brugt som et værktøj internt i gruppen med henblik på at udforske relationer mellem vores forskellige konceptuelle klasser gennem konkrete eksempler (eller scenarier). Værdien i dette bestod i at forsøge at blottlægge potentielle fejlagtige antagelser, vi havde gjort os, og specifikt også for at sikre os, at vi ikke havde mange-til-mange relationer klasserne imellem, der kunne skabe problemer for os senere i arbejdsprocessen. Ovenstående modeller har vi refaktoreret sidst i projektperioden, for at sikre sporbarhed og for så vidt muligt at leve op til de formelle krav, der nu engang er til de to modeller. Det er i den forbindelse værd at nævne, at vi i løbet af projektperioden i højere grad brugte modellerne som en form for brainstorm eller værktøj til at sikre, at vi internt i gruppen var på bølgelængde ift. den konceptuelle forståelse af domænet.

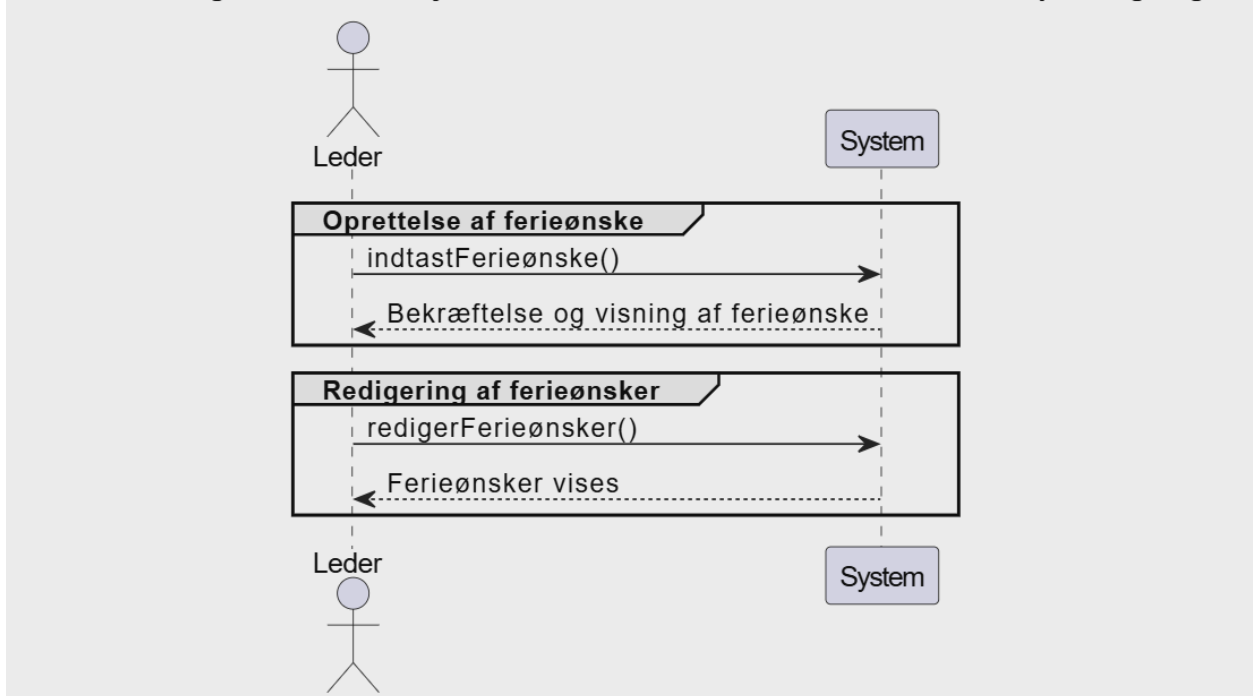
Af denne grund var der ikke et stringent fokus på, at modellerne løbende skulle stå 100% skarpt og altid følge alle best-practise regler, men var i stedet lidt mere flydende størrelser, vi kunne vende tilbage til, hvis der opstod tvivl om projektdomænet og relationerne mellem klasserne (en tidligere version af objektmodellen kan evt. findes i bilag 4, som viser en objektmodel, som ikke står helt skarpt, men stadig havde værdi for os, da vi udviklede vores domæneforståelse).

Ser vi på ovenstående modeller, er der primært to ting, som er værd at nævne. Konkret ift. use-case 2, var det igennem modellerne, vi blev opmærksomme på fordelene ved at have en konceptuel ferieønskeperiode-klasse, da vi havde brug for at kunne have et varierende antal separate perioder af dage under hvert ferieønske (som illustreret ved første ferieønske i objektmodellen).

For det andet er der hverken på ovenstående modeller eller på de komplette modeller i bilag 2 og 3 nogle brugere af systemet inkluderet. Dette var et aktivt valg, da systemet i første omgang kun var relateret til en leders arbejde, og der ville således ikke være forskellige brugere knyttet til forskellige dele af programmet. Derudover findes lederen heller ikke som konceptuel klasse internt i vores system, ligesom lagermedarbejdere gør, og der var derfor umiddelbart ingen værdi i at introducere lederen i disse modeller.

## SSD

### Use-case 2: Organiser medarbejdernes ferieønsker som forberedelse til ferieplanlægningen



Dette SSD repræsenterer detaljer fra Use Case 2, som specificerer ledernes handlinger.

Diagrammet fokuserer på interaktionen mellem lederen og systemet for håndtering af ferieønsker. Systemets funktionalitet bliver skitseret, uden at gå i detaljer om, hvordan data internt behandles.

Leder repræsenterer den eksterne aktør, der opretter, redigerer og viser ferieønsker.

Systemet behandler ferieønsker og returnerer information til lederen.

Diagrammet viser tre forskellige scenarier:

#### Oprettelse af ferieønske

Leder starter oprettelsen af et ferieønske ved at sende en forespørgsel til systemet. Systemet bekræfter oprettelsen og returnerer det indtastede ferieønske til lederen.

#### Visning af ferieønsker

Leder anmoder om visning af eksisterende ferieønsker. Systemet henter og returnerer de oprettede ferieønsker til lederen.

#### Redigering af ferieønsker


Lederen vælger et ferieønske at redigere, og ændrer de relevante data. Systemet registrerer ændringerne, opdaterer og viser ferieønskerne.

SSD'en viser tydeligt sammenhængen mellem lederen og systems handlinger, uden at gå i detaljer. Dette gør diagrammet nyttigt, til at visualisere de vigtigste hændelser, og flowet mellem brugeren og systemet.

Diagrammet fungerer desuden som grundlag for at udarbejde operationskontrakter, som er nødvendige for at specificere systemets funktionalitet.

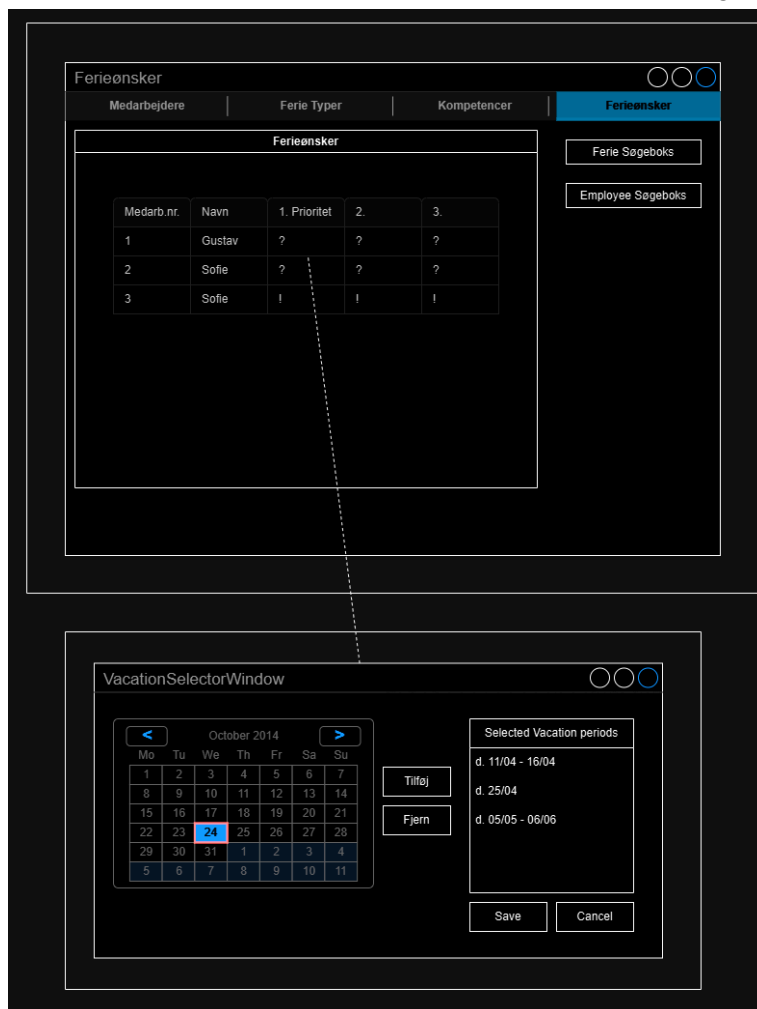
## Operationskontrakter

Vores arbejde med operationskontrakter (OC) tog udgangspunkt i vores objekt- og domænemodellen. Vi udformede OC'er for at tydeliggøre de tilstandsændringer, der sker som resultat af operationer. Arbejdet med OC'er hjalp os til at arbejde videre med en bedre før- og efter-forståelse af tilstandsændringerne. Dog så vi ikke udarbejdelsen af OC'er som altafgørende i vores forståelse af systemets operationer eller videreudvikling af modeller fra highlevel til lowlevel design. Vi lavede kun OC'er for enkelte systemoperationer.

	Operation: importérFerieønsker(nuværende ferieønsker:string)
Cross Reference: Importering af Data	
Preconditions: Ferieønske-fil findes	
Postconditions: Medarbejder-instanser blev oprettet Ferieønske-instanser blev oprettet Ferieønske-instanser blev associeret til ferietype-instanser Ferieønske-instanser blev associeret til medarbejder-instanser	

## Wireframes

Vores arbejde med wireframes endte med at spille en fundamental rolle i udviklingen af vores koncept, da det gav os mulighed for at kommunikere både internt og eksternt ud fra en visuel præsentation af struktur, koncepter og idéer. Vores diskussioner og refleksioner omkring funktionalitet ledte os ofte tilbage til wireframes, som endte med at være bindingspunktet ift. en fælles forståelse. Vi havde fokus på at optimere brugeroplevelsen ved at skabe simpel visning af ansatte og holde antallet af nødvendige klik til så få som muligt ift. at oprette medarbejderes prioriteter ved hjælp af visuel kalender. Med wireframes kunne vi individuelt arbejde imod at skabe den bedste brugeroplevelse med en fælles forståelse for programmets struktur og layout, lige meget om vi arbejdede i UI eller backend. De medium-fi wireframes, vi lavede i Use Case 2, blev brugt kontinuerligt gennem hele projektet til intern kommunikation, men også proof-of-concept-møder eksternt med Alfa Laval Kolding.



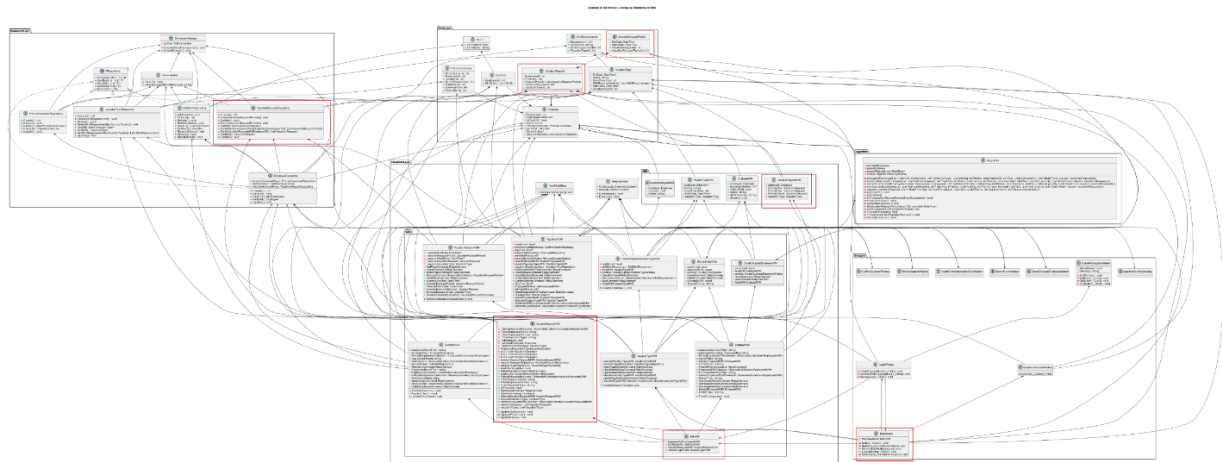
## DCD

For at skabe et overblik over systemdesignet, udviklede vi et design class diagram (DCD). Diagrammet hjalp os med at visualisere relationerne mellem programmets klasser, definere attributter og metoder for hver klasse og fastlægge deres access modifiers. DCD'et fungerede som en bro imellem vores domænemodel og den egentlige implementering.

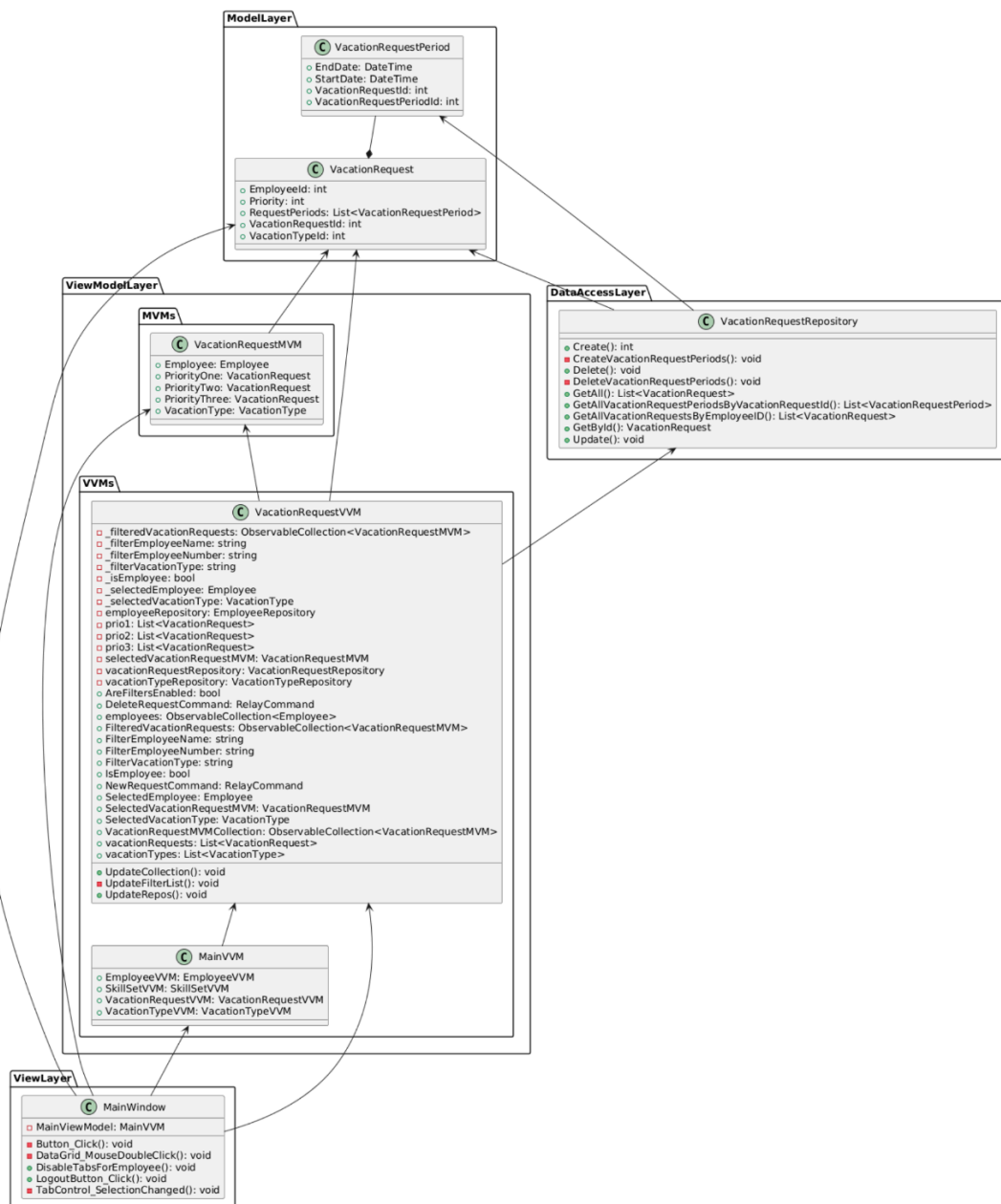
Vi arbejdede iterativt med DCD'et, hvilket betød, at nye klasser og relationer blev tilføjet undervejs, når vores forståelse af systemets krav udviklede sig. I projektets begyndelse var DCD'et et vigtigt kommunikationsværktøj, men efterhånden som projektet skred frem, blev behovet mindre, fordi teamet opnåede et fælles overblik. Samtidig blev diagrammet så komplekst, at det mistede meget af sin læselighed.

Derfor valgte vi at udskyde arbejdet med diagrammet, og i stedet lave en færdig model når programmet var færdigt. På næste side ses et eksempel på vores DCD, og specifikt hvordan vores forbindelser til VacationRequest modellen, som er low-level integreringen af vores use case 2, endte med at se ud. For læselighedens skyld er forbindelser til objekter uden for modellen fjernet, men en større version af vores fulde model kan findes i bilag 8 og en mere læselig version kan findes som ekstra materiale på wiseflow.

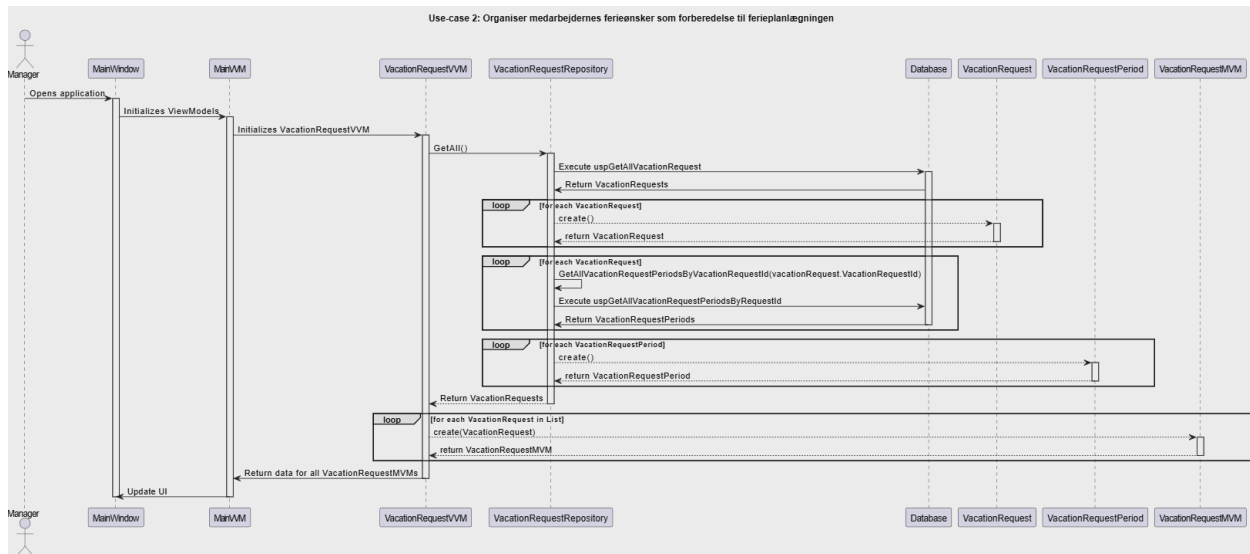




Usecase 2: Opretelse, visning og tilpasning af data (Spotlight på Ferieønsker)



# SD



Sekvensdiagrammer er nyttige til at vise, hvordan forskellige komponenter i systemet kommunikerer med hinanden over tid, hvilket gør det lettere at forstå flowet af information og processer.

Vores sekvensdiagram giver en visuel repræsentation af, hvordan data relateret til ferieønsker håndteres i systemet. SD'er beskriver hvordan brugeren interagerer med applikationen, fra åbning af systemet til indhentning og visning af ferieønsker.

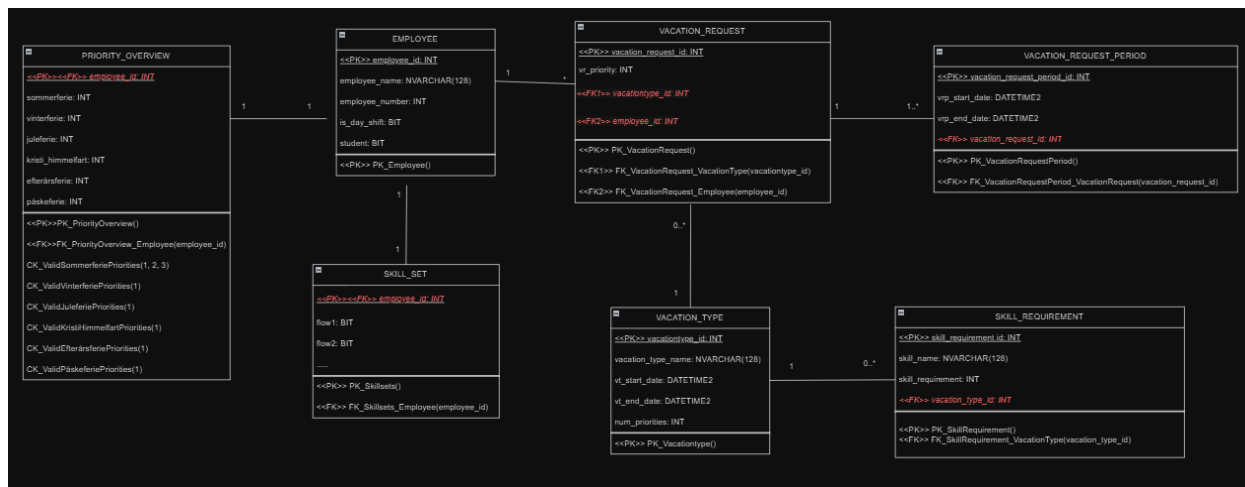
Diagrammet viser en lineær sekvens af handlinger, hvor systemet håndterer forskellige trin i forbindelse med indhentning af ferieønsker. Lederen åbner først applikationen, hvilket aktiverer systemets forskellige lag, der initierer og henter data fra systemets database.

Der fremhæves, hvordan forskellige komponenter som MainWindow, MainVVM og VacationRequestVVM arbejder sammen for at levere data til lederen.

SD'et bruges til at dynamisk visualisere nogle af de komplekse processer, involveret i håndtering af ferieønsker. Diagrammet viser tydeligt rækkefølgen af operationer, og hvordan data bevæger sig mellem de forskellige lag i systemet i realtid.

Dette hjælper os som udviklere med at forstå systemets arbejdsproces og de forskellige trin, der udføres for at sikre korrekt behandling af data.

# Databasemodel



Databasemodellen hjælper os med struktureret at organisere data og har dannet grundlaget for implementeringen af vores Relational Database management system- vores SQL database. Relationerne og kardinalitet mellem tabellerne vises gennem associationer i modellen. Den er desuden blevet normaliseret efter de 3 første normalformer, for at sikre dataintegritet og nøjagtighed.

Databasemodellen illustrerer strukturen af de data, der skal persisteres i systemets database. Hver tabel i modellen repræsenterer en entitet, der indeholder attributter, som oversættes til SQL-kolonner med tilhørende datatyper. Eksempelvis indeholder EMPLOYEE-entiteten kolonner som employee\_id (primærnøgle), employee\_name, employee\_number, is\_day\_shift og student.

I modellen er tabellen SKILL\_SET designet til at understøtte dynamisk opdatering af kolonner. Dette er vigtigt, fordi kravene til medarbejderenes kompetencer kan ændre sig over tid, og systemet skal derfor kunne tilpasse sig. Eksempelvis, hvis der kommer et behov for en ny kompetence som "Førstehjælp", kan kolonnen oprettes via. Programmet. Dynamikken i SKILL\_SET er illusretet ved hjælp af tre prikker (...) under attributlisten. Når systemet oprettes, starter tabellen med en enkelt kolonne, employee\_id, som er den sammensatte primærnøgle, med en fremmednøgle-relation. Yderligere kolonner kan derefter tilføjes dynamisk under kørsel eller ved hjælp af scripts, der kører efter deployment.

Vi har brugt databasemodellen som et vigtigt værktøj i kommunikationen med Alfa Laval. Modellen gjorde det muligt for os at præsentere, hvordan vi håndterer og strukturerer dataene, og det har hjulpet os med at afstemme forventninger og krav. Det gav en visuel forståelse af systemets dataopbygning og en mulighed for feedback inden fuld implementering af databasen.

Undervejs i udviklingsprocessen blev det klart, at Alfa Laval ønskede muligheden for dynamisk at kunne tilføje kompetencer i SKILL\_SET-tabellen. Denne besked kom sent i processen, hvilket førte til en nødvendig omlægning af den oprindelige databasearkitektur.

I stedet for at designe en statisk model med foruddefinerede kolonner til kompetencer, valgte vi en løsning, der tillader dynamiske opdateringer af kolonner.

Hvis dette krav var blevet kommunikeret tidligere i processen, kunne vi have overvejet alternative løsninger, såsom at modellere kompetencer i en separat tabel. Dette ville have reduceret afhængigheden af dynamiske kolonneændringer.

# Implementering

## Designmodel og designmønster

I vores projekt valgte vi at anvende en MVVM-struktur. Dette har vi gjort på baggrund af de fordele, som MVVM tilbyder i en WPF-baseret applikation.

ViewModel-laget gør det nemt at data- og command-binde brugergrænsefladens elementer (Views) direkte til Properties og kommandoer i ViewModels. Dette reducerer behovet for code-behind i Views og resulterer i en mere fleksibel og vedligeholdelsesvenlig kodebase.

Vi bruger altså færre event-handlinger i code-behind, hvilket også gør koden mere overskuelig. MVVM skaber en opdeling mellem UI (View) og forretningslogik (ViewModel). Dette gør det nemmere at teste forretningslogikken, fordi det kan testes uafhængigt af UI.

### Hvorfor ikke MVC?

Valget af MVVM frem for MVC blev truffet med fokus på at levere et fleksibelt, testbart og vedligeholdelsesvenligt program. I MVC er controller-laget ofte tættere knyttet til View-laget, hvilket kan gøre testning og udskiftning af lag mere kompliceret.

I en WPF-kontekst er MVVM mere naturligt integreret med frameworkets funktioner, som bindings.

```
3 public interface IRepository<T> where T : class
4 {
5     14 references
6     public int Create(T entity);
7     9 references
8     public void Update(T entity);
9     15 references
10    public void Delete(int id);
11    11 references
12    public T GetById(int id);
13    22 references
14    public List<T> GetAll();
15 }
```

For at gøre vores klasser mere ensartede og genbrugelige, brugte vi **interfacet** `IRepository<T>`. Interfacet definerer de grundlæggende metoder, som alle vores repositories skulle have. Disse var:

- Create: Til oprettelsen af en ny entitet i databasen
- Update: Til at opdatere eksisterende data
- Delete: Til at slette data fra databasen
- GetById: Til at hente en specifik entitet ud fra dens ID
- GetAll: Til at hente en liste over alle entiteter.

Vores repository pattern bruger repository laget til at stå for kommunikationen til vores datalager og til at holde CRUD metoderne adskilt fra forretningslogikken, men tilgængeligt for at vores VVM klasser kan bruge metoderne og opretholde høj persistens. I billedet under, highlighter vi vores `VacationRequestRepository`, der indeholder alle CRUD metoderne.

Interfacet viser desuden, hvordan vi har brugt abstrakte datatyper i vores projekt. Ved at definere et generisk repository-mønster, skaber vi en fleksibel og genanvendelig abstraktion, der kan arbejde med enhver Model-klasse, som opfylder kravene. For eksempel kan T repræsentere en medarbejder, en ferieanmodning, eller en kompetence.

```

19  / references
20  public class VacationRequestRepository : IRepository<VacationRequest>
21  {
22      5 references
23      public int Create(VacationRequest vacationRequest) {...}
24      70
25      1 reference
26      public void Update(VacationRequest vacationRequest) {...}
27      117
28      6 references
29      public void Delete(int vacationRequestId) {...}
30      161
31      1 reference
32      public VacationRequest GetById(int vacationRequestId) {...}
33      239
34      2 references
35      public List<VacationRequest> GetAll() {...}
36      240
37      297
38      3 references
39      public List<VacationRequest> GetAllVacationRequestsByEmployeeId(int employeeId) {...}
40      360
41      2 references
42      public List<VacationRequestPeriod> GetAllVacationRequestPeriodsByVacationRequestId(int vacationRequestId) {...}
43      404
44      2 references
45      private void CreateVacationRequestPeriods(VacationRequest vacationRequest) {...}
46      405
47      417
48      2 references
49      private void DeleteVacationRequestPeriods(int vacationRequestId) {...}
50      418
51      425
52      426

```

Via connectionstring til vores SQL server kan vi tilgå vores tilhørende stored procedures som i dette tilfælde er `uspCreateVacationRequest` (se nedenstående snippet)

```

1  CREATE PROC [dbo].[uspCreateVacationRequest] @vrPriority INT
2  ,@vacationTypeId INT
3  ,@employeeId INT
4  AS
5  BEGIN
6      IF NOT EXISTS (
7          SELECT 1
8          FROM EMPLOYEE
9          WHERE [employee_id] = @employeeId
10         )
11     BEGIN
12         DECLARE @callingProcedure NVARCHAR(128) = OBJECT_NAME(@@PROCID);
13
14         EXEC [dbo].[uspLogAndThrowError] 50001
15             ,@callingProcedure;
16     END;
17
18     BEGIN TRY
19         INSERT INTO VACATION_REQUEST (
20             [vr_priority]
21             ,[vacation_type_id]
22             ,[employee_id]
23         )
24         VALUES (
25             @vrPriority
26             ,@vacationTypeId
27             ,@employeeId
28         );
29
30         DECLARE @vacationRequestId INT;
31
32         SET @vacationRequestId = SCOPE_IDENTITY();
33
34         SELECT @vacationRequestId;
35
36         PRINT 'Successfully created Vacation Request. Operation completed';
37     END TRY
38
39     BEGIN CATCH
40         EXEC [dbo].[uspLogAndThrowError] 50101;
41     END CATCH;
42 END;

```

Stored procedures spillede en vigtig rolle i vores projekt, da de blev brugt til at håndtere alle direkte operationer mod databasen. Vi brugte stored procedures til at udføre CRUD-operationer. Eksempelvis brugte vi stored procedures til at udføre alle databaseoperationer relateret til VacationRequest. Ved at en medarbejder oprettede en ferieansøgning i vores system, ville der ske flere ting.

## Database

For at vise hvordan vi har anvendt SQL, har vi taget nogle eksempler fra vores kode. Dette er et eksempel på to af vores tabeller, VACATIONREQUEST og EMPLOYEE. Vi anvender constraints, not null, og default, da vi gerne vil have muligheden for at oprette entries med standardværdier.

```
CREATE TABLE [dbo].[VACATION_REQUEST]
(
    [vacation_request_id] INT NOT NULL IDENTITY,
    [vr_priority] INT NOT NULL,
    [vacation_type_id] INT NOT NULL,
    [employee_id] INT NOT NULL,
    CONSTRAINT [PK_VacationRequest] PRIMARY KEY (vacation_request_id),
    CONSTRAINT [FK_VacationRequest_vacationType] FOREIGN KEY (vacation_type_id) REFERENCES VACATION_TYPE(vacation_type_id),
    CONSTRAINT [FK_VacationRequest_Employee] FOREIGN KEY (employee_id) REFERENCES EMPLOYEE(employee_id)
)
```

```
CREATE TABLE [dbo].[EMPLOYEE]
(
    [employee_id] INT IDENTITY,
    [employee_name] NVARCHAR (128) NOT NULL,
    [employee_number] INT,
    [is_day_shift] BIT DEFAULT 1,
    [student] BIT DEFAULT 0,
    CONSTRAINT [PK_Employee] PRIMARY KEY (employee_id)
);
```

Det næste er et eksempel på, hvordan vi bruger en stored procedure til at indsætte nye vacation requests på vores tabel. Vi anvender først en IF NOT EXISTS blok til at tjekke, om der eksisterer en employee på EMPLOYEE tabellen, da en af vores kolonner på vacation request er en fremmednøgle til den tabel og derfor skal have en reference. Hvis den finder én, indsætter den en ny entry med de parametre, der er givet, og hvis ikke, giver den fejl.

```
CREATE PROC [dbo].[uspCreateVacationRequest] @vrPriority INT
, @vacationTypeId INT
, @employeeId INT
AS
BEGIN
    IF NOT EXISTS (
        SELECT 1
        FROM EMPLOYEE
        WHERE [employee_id] = @employeeId
    )
    BEGIN
        DECLARE @callingProcedure NVARCHAR(128) = OBJECT_NAME(@@PROCID);
        EXEC [dbo].[uspLogAndThrowError] 50001
        , @callingProcedure;
    END;

    BEGIN TRY
        INSERT INTO VACATION_REQUEST (
            [vr_priority]
            , [vacation_type_id]
            , [employee_id]
        )
        VALUES (
            @vrPriority
            , @vacationTypeId
            , @employeeId
        );

        DECLARE @vacationRequestId INT;

        SET @vacationRequestId = SCOPE_IDENTITY();

        SELECT @vacationRequestId;

        PRINT 'Successfully created Vacation Request. Operation completed';
    END TRY

    BEGIN CATCH
        EXEC [dbo].[uspLogAndThrowError] 50101;
    END CATCH;
END;
```



Som sidste eksempel viser vi hvordan `uspCreateVacationRequest` bliver kaldt fra systemet.

```
public int Create(VacationRequest vacationRequest)
{
    bool isOuterTransaction = System.Transactions.Transaction.Current == null;

    try
    {
        ArgumentNullException.ThrowIfNull(vacationRequest, "The Vacation Request cannot be null.");

        if (vacationRequest.RequestPeriods.Count < 1)
        {
            MessageBox.Show("Please add at least one vacation period to your request.", "Invalid Request", MessageBoxButtons.OK, MessageBoxIcon.Warning);
            return -1;
        }

        using (TransactionScope transaction = isOuterTransaction ? new() : new(TransactionScopeOption.Required))
        {
            ConnectionManager.OpenIfClosed();

            SqlCommand createCommand = new("uspCreateVacationRequest", ConnectionManager.sqlCon);
            createCommand.CommandType = CommandType.StoredProcedure;
            createCommand.Parameters.Add("@vPriority", SqlDbType.Int).Value = vacationRequest.Priority;
            createCommand.Parameters.Add("@vVacationTypeId", SqlDbType.Int).Value = vacationRequest.VacationTypeId;
            createCommand.Parameters.Add("@vEmployeeId", SqlDbType.Int).Value = vacationRequest.EmployeeId;
            vacationRequest.VacationRequestId = (int)createCommand.ExecuteScalar();

            CreateVacationRequestPeriods(vacationRequest);

            transaction.Complete();
        }

        if (isOuterTransaction)
        {
            ConnectionManager.sqlCon.Close();
        }

        return vacationRequest.VacationRequestId;
    }
    catch (Exception ex)
    {
        if (isOuterTransaction)
        {
            ConnectionManager.sqlCon.Close();
        }

        ErrorHandler.Catch(ex);
        return -1;
    }
}
```

## Connection Manager

I samtlige repositories i vores kode kalder vi stored procedures, som vi gennemgående har brugt til at strukturere vores CRUD-queries til SQL-databasen. Nogle af vores repository-metoder kalder dog mere end en enkelt stored procedure eller kalder hjælpe-metoder, som også kalder stored procedures. Derfor har det været nødvendigt for os omhyggeligt at strukturere både vores database-forbindelseslogik og sørge for at benytte transactions på en gennemtænkt måde. For at håndtere forbindelsen til databasen, har vi således oprettet en **statisk** Connection Manager klasse, som sørger for, at hver gang en metode har brug for en forbindelse, kan den tjekke her, om der allerede er oprettet forbindelse til serveren – eksempelvis fordi den bliver kaldt som en del af en transaction – og i tilfælde af, at der allerede er forbindelse, laver den ikke en ny forespørgsel på en forbindelse til SQL-databasen; den kan blot tilslutte sig den eksisterende. Og ligeledes, når en forbindelse potentielt skal lukkes efter en stored procedure er blevet kaldt, tjekker vores metoder hos Connection Manager klassen, om der er en transaction i gang. Hvis der ikke er, lukker forbindelsen, og lader den blot stå til.

På denne måde sørger vi for aldrig at oprette indlejrede forbindelser og holder os til altid kun at have én forbindelse til databasen.

```

99+ references | Jakob Seeberg, 13 days ago | 1 author, 1 change
public static class ConnectionManager
{
    public static SqlConnection sqlCon = new(ConfigurationManager.ConnectionStrings["YourLocalConnection"].ConnectionString);

    32 references | Jakob Seeberg, 13 days ago | 1 author, 1 change
    public static void OpenIfClosed()
    {
        if (sqlCon.State == System.Data.ConnectionState.Closed)
        {
            sqlCon.Open();
        }
    }

    16 references | Jakob Seeberg, 13 days ago | 1 author, 1 change
    public static void CloseWithoutTransaction()
    {
        if (System.Transactions.Transaction.Current == null && sqlCon.State == System.Data.ConnectionState.Open)
        {
            sqlCon.Close();
        }
    }
}

```

## Transactions

Hver gang en af vores repository-metoder kalder flere stored procedures eller kalder andre metoder, der kalder stored procedures, er der også integreret transactions i vores kode. Ved at bruge transactionScopes i vores kode sørger vi for, at alt, der bliver kaldt inden for det samme transactionScope bliver samlet i en enkelt transaction og derfor enten alt sammen bliver gennemført eller at der bliver kaldet rollback på det hele. På den måde undgår vi uoverensstemmelser på vores database, hvis der opstår et problem halvvejs igennem en transaction. For ikke at igangsætte flere transactions på en gang, kalder vi eksplicit den transactionScope constructor, som kan tage TransactionScopeOption.Required som parameter. Dette benytter vi når en repository-metode, der benytter et transactionScope, kalder en anden repository-metode, der også benytter et transactionScope. På denne måde sikrer vi, at den kaldte metode benytter sig af den eksisterende transaction som den kaldende metode har startet. Bool-variabelen 'isOuterTransaction', som kan ses øverst i eksemplet med Update-metoden hjælper med at holde styr på om metoden bliver kaldt af en anden metode, der benytter transaction, og starter kun en ny transaction, og lukker ligeledes kun forbindelsen til databasen, hvis der ikke findes en eksisterende transaction.

```

1 reference | Jeppe Lynge Hansen, 4 days ago | 3 authors, 16 changes
public void Update(VacationRequest vacationRequest)
{
    bool isOuterTransaction = System.Transactions.Transaction.Current == null;

    try
    {
        ArgumentNullException.ThrowIfNull(vacationRequest, "The Vacation Request cannot be null.");

        if (vacationRequest.RequestPeriods.Count < 1)
        {
            MessageBox.Show("Please add at least one vacation period to your request.", "Invalid Request", MessageBoxButton.OK, MessageBoxImage.Warning);
            return;
        }

        using (TransactionScope transaction = isOuterTransaction ? new() : new(TransactionScopeOption.Required))
        {
            ConnectionManager.OpenIfClosed();

            SqlCommand updateCommand = new("uspUpdateVacationRequest", ConnectionManager.sqlCon);
            updateCommand.CommandType = CommandType.StoredProcedure;
            updateCommand.Parameters.Add("@vacationRequestId", SqlDbType.Int).Value = vacationRequest.VacationRequestId;
            updateCommand.Parameters.Add("@vrPriority", SqlDbType.Int).Value = vacationRequest.Priority;
            updateCommand.ExecuteNonQuery();

            DeleteVacationRequestPeriods(vacationRequest.VacationRequestId);
            CreateVacationRequestPeriods(vacationRequest);

            transaction.Complete();
        }

        if (isOuterTransaction)
        {
            ConnectionManager.sqlCon.Close();
        }
    }
    catch (Exception ex)
    {
        if (isOuterTransaction)
        {
            ConnectionManager.sqlCon.Close();
        }

        ErrorHandler.Catch(ex);
    }
}

```

## Øvrig fremhævet kode

### Kommentarer i koden

Her er et udsnit fra commanden AddColumnCommand i klassen SkillSetVVM. Vi har forsøgt at bruge kommentarer strategisk gennem hele projektet for at forbedre kodelæsarheden og sikre kvalitet. Ved kun at forklare "hvad" der sker fremfor "hvordan", holder vi kommentarerne korte og målrettede, hvilket gør koden lettere at vedligeholde og forstå for både os selv og andre udviklere.

```

public RelayCommand AddColumnCommand => new(execute =>
{
    // Check if a column with the same header already exists
    if (this.SkillColumns.Any(column => column.Header.ToString() == NewColumnName))
    {
        MessageBox.Show($"Kompetencen '{NewColumnName}' findes allerede.", "Duplicate Kolonne", MessageBoxButton.OK, MessageBoxImage.Warning);
        NewColumnName = string.Empty;
        return; // Exit if the column already exists
    }
}

```

# Unit Testing

Vi har generelt ikke anvendt tests igennem projektet, bortset fra ét specifikt tilfælde, som vi vil forklare senere. Vi valgte at undlade tests ifm. repositories fordi vi havde svært ved at se, hvordan de kunne implementeres effektivt i forbindelse med vores SQL-database. Unit Tests er i vores forståelse ikke designet til at fungere direkte med en database. De forsøg, vi havde, ændrede dataen på databasens tabeller og betød ofte, at testen resulterede i fejl, fordi den var designet til at redigere eller slette specifikke employees. Da det employees blev slettet sidste gang testen kørte fejlede testen selvfølgelig, og selv hvis vi prøvede at tilføje den i en anden test så kører tests i MSTest ikke i rækkefølge af hvordan de er skrevet, hvilket betød at testen kunne prøve at slette en employee der ikke var oprettet endnu.

Som en del af udviklingen af SkillSetRepository skulle vi implementere, at databasetabellen SKILL\_SET kunne tilføje og fjerne kolonner dynamisk, og i den sammenhæng gav vi tests, et forsøg igen og fandt en måde hvorpå vi kunne køre test methods i den rækkefølge, som vi gerne ville.

Herunder ses TestInitialize, som er kode, der kører, inden testen udføres og som klargøre alt til, at testene kan køre, som de skal.

```
public class SkillSetRepositoryTests
{
    private string connectionString;
    private bool _testAddMethod;
    private bool _testUpdateMethod;
    private bool _testUpdateSkillMethod;
    private bool _testDeleteMethod;
    private bool _testGetByIdMethod;
    private bool _testGetAllMethod;

    private int testEmployeeId;
    private int testEmployeeNumber;

    [TestInitialize]
    public void TestInitialize()
    {
        // Dynamically set the config file path
        AppDomain.CurrentDomain.SetData("APP_CONFIG_FILE", AppDomain.CurrentDomain.BaseDirectory + "TestForProject.dll.config");
        // (Optional) Set the current directory to the bin folder
        Directory.SetCurrentDirectory(AppDomain.CurrentDomain.BaseDirectory);

        var employeeRepository = new EmployeeRepository();

        Random rnd = new Random();

        testEmployeeNumber = rnd.Next(10, 99999999);

        Employee testemployee = new Employee(
            testEmployeeNumber,
            "Lars Johan",
            true,
            false
        );

        testemployee.EmployeeId = employeeRepository.Create(testemployee);
        testEmployeeId = testemployee.EmployeeId;
    }
}
```

Derefter køres CombinedTest, som kalder andre metoder hvori vores "egentlige" tests ligger.

```

[TestMethod]
public void CombinedTest()
{
    TestAddMethod();
    Assert.IsTrue(_testAddMethod, "TestAddMethod did not execute successfully.");

    TestUpdateMethod();
    Assert.IsTrue(_testUpdateMethod, "TestUpdateMethod did not execute successfully.");

    TestUpdateSkillMethod();
    Assert.IsTrue(_testUpdateSkillMethod, "TestUpdateSkillMethod did not execute successfully.");

    TestGetByIdMethod();
    Assert.IsTrue(_testGetByIdMethod, "TestGetByIdMethod did not execute successfully.");

    TestGetAllMethod();
    Assert.IsTrue(_testGetAllMethod, "TestGetAllMethod did not execute successfully.");

    TestDeleteMethod();
    Assert.IsTrue(_testDeleteMethod, "TestDeleteMethod did not execute successfully.");
}

```

Og til sidst viser vi TestUpdateMethod, som er en af de metoder der blev kaldt i CombinedTest.

```

public void TestUpdateMethod()
{
    var ssRepo = new SkillSetRepository();

    Skill dans = new Skill("danse", true);
    Skill bagning = new Skill("bagning", true);
    Skill programmering = new Skill("programmering", true);

    SkillSet testss = new(testEmployeeId);
    testss.SkillList.Add(dans);
    testss.SkillList.Add(bagning);
    testss.SkillList.Add(programmering);

    ssRepo.Update(testss);

    _testUpdateMethod = true;
}

```

## Brugergrænseflade

Koden herunder viser, hvordan vi har brugt data- og command-binding.

ItemsSource="{Binding FilteredEmployeeMVMCollection}" sørger for, at DataGrid dynamisk opdateres med data fra ViewModel'en.

Kommandoerne (OpenCreateWindowCommand, OpenUpdateWindowCommand, OpenDeleteWindowCommand) er bundet til knapper, hvilket giver mulighed for at udføre handlinger som oprettelse, opdatering og sletning af medarbejderdata direkte fra brugergrænsefladen.

```

<DataGrid Grid.Column="1" Grid.Row="1"
    ItemsSource="{Binding FilteredEmployeeMVMCollection}"
    SelectedItem="{Binding SelectedEmployeeMVM}"
    AutoGenerateColumns="False"
    IsReadOnly="True"
    AlternationCount="2">
    <DataGrid.RowStyle>
        <Style TargetType="DataGridRow">
            <Setter Property="Background" Value="White" />
            <Style.Triggers>
                <Trigger Property="AlternationIndex" Value="1">
                    <Setter Property="Background" Value="#BEBEBE" />
                </Trigger>
            </Style.Triggers>
        </Style>
    </DataGrid.RowStyle>
    <DataGrid.Columns>
        <DataGridTextColumn Header="Medarbejder" Binding="{Binding Name}" Width="*" />
        <DataGridTextColumn Header="Medarbejdernummer" Binding="{Binding EmployeeNumber}" Width="*" />
        <DataGridTextColumn Header="Skift" Binding="{Binding ShiftDisplay}" Width="*" />
    </DataGrid.Columns>
</DataGrid>

<StackPanel Grid.Row="2" Grid.Column="1" Orientation="Horizontal" HorizontalAlignment="Center">
    <Button Content="Ny" Command="{Binding OpenCreateWindowCommand}" Width="120" Margin="5,0,20,0" Height="30"/>
    <Button Content="Opdater medarbejder" Command="{Binding OpenUpdateWindowCommand}" Width="120" Margin="5,0,20,0" Height="30"/>
    <Button Content="Slet" Command="{Binding OpenDeleteWindowCommand}" Width="120" Margin="5,0,20,0" Height="30"/>
</StackPanel>
</Grid>

```

# Algoritme

Vores program kulminerer med en algoritme, der på ryggen af det solide fundament, alle de andre dele af systemet udgør, producerer et output, som i csv-format giver et udkast til en ferieplan.

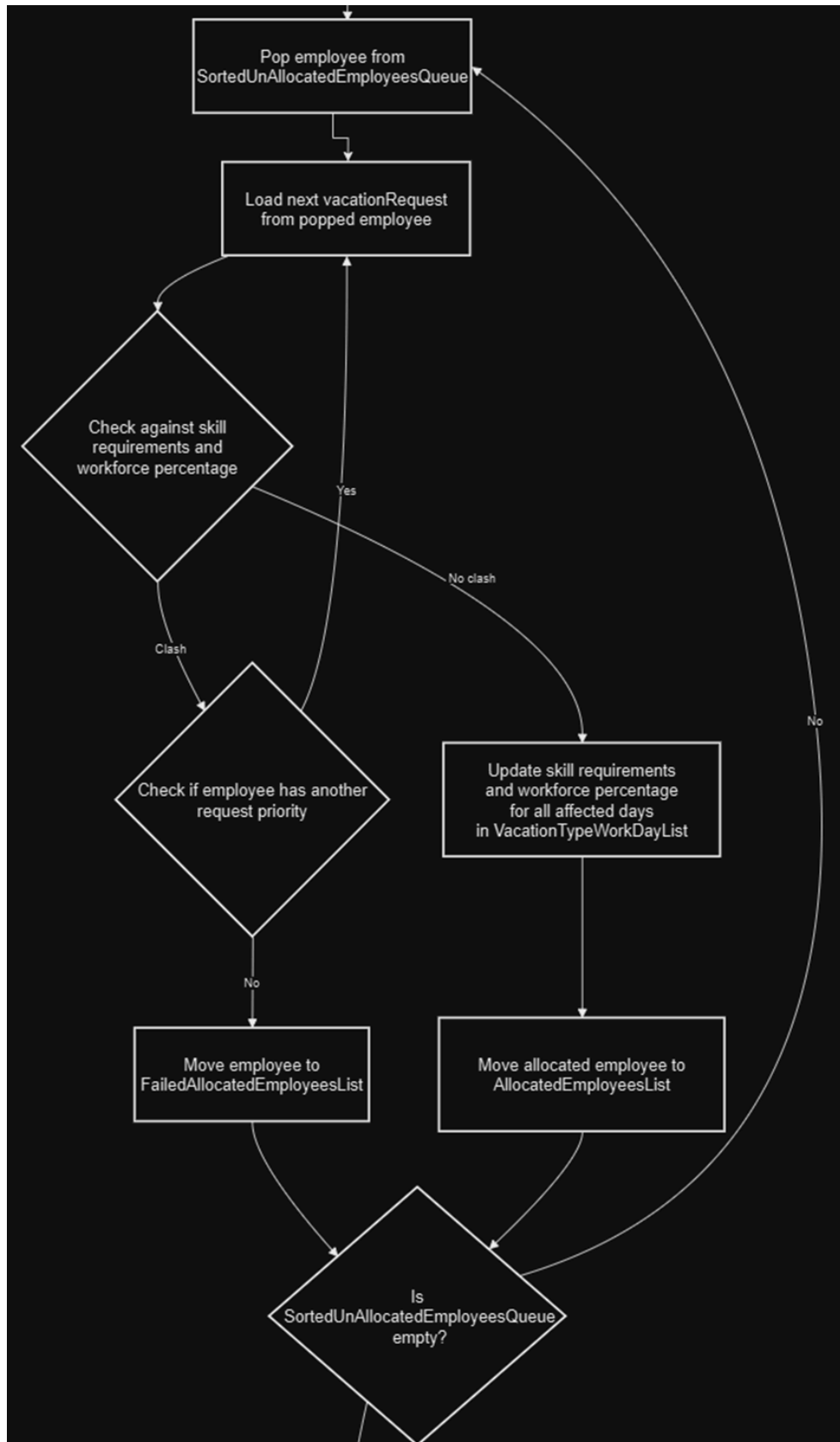
Inputtet til algoritmen består af

- Den ferie det drejer sig om.
- Om det er aften- eller dagsholdet, der laves ferie for.
- Medarbejdernes prioriterede ferieønsker.
- Medarbejdernes opnåede prioritet for samme ferie sidste år. Dette input har til hensigt at sikre en rimelig fordeling af ferier på lang sigt.
- Et minimumsbemandingskrav for ferien angivet i procent.
- Et minimumskrav for hver kompetence for ferien, angivet som det mindste, acceptable antal medarbejdere med en given kompetence.

At designe og programmere en algoritme med så relativt mange input vakte en smule ærefrygt i os, givet at vi ikke havde den store erfaring med algoritmedesign. Vi gik derfor ydmygt til opgaven, og vores udgangspunkt var således at formulere den simplest mulige algoritme, der i en eller anden grad forholdt sig til samtlige input. Idéen var at sortere medarbejderne ud fra sidste års opnåede ferieprioritet, så dem, der ikke fik sit ønske opfyldt sidste år, forhåbentlig kunne få det i denne ferieplan – og så blot forsøge at fordele medarbejderne fra en ende af. Så længe en medarbejder kan fordeles uden at overskride minimumskravene, bliver de fordelt, og minimumskravene opdateres, da der efter allokering er én medarbejder, med dennes specifikke kompetencer, færre til rådighed for de dage, der hører under det allokerede ferieønske. Hvis en allokering derimod ville overskride et krav, forsøges allokering af 2. og 3. prioritet, og ellers allokeres der ikke ferie til denne medarbejder, og de tilføjes i stedet til en liste af uallokerede medarbejdere.

Den primære datatype der benyttes under algoritmens afvikling, er lister af tupler. Denne datatype blev valgt for ikke at oprette en masse klasser, der kun ville blive brugt i algoritmen. Med brug af tupler, kunne vi i stedet repræsentere og manipulere elementerne internt i algoritmeklassen.

Til at designe algoritmen benyttede vi os af et flow-chart, og et udsnit af dette er inddraget her for at vise den mest essentielle logik for algoritmen. Udarbejdelsen af flow-chartet hjalp os til at gøre algoritmen modulær og bryde den ned i tilpas små bidder, til at den blev overskuelig at kode.





For at lave et enkelt nedslag i algoritmekoden, er der en interessant kvalitet ved sorteringen af medarbejdere, der er værd at notere sig. Det var et udtalt ønske fra Alfa Laval, at ferieallokeringsalgoritmen ikke indeholdt nogen bias – at den var retfærdig - da deres nuværende metode med manuel allokering af medarbejdere, selv hvis ferien var retfærdigt fordelt, stadig kunne give medarbejdere en følelse af forskelsbehandling, hvis de uheldigvis ikke fik deres ferie. For at imødekomme dette, fordeler algoritmen først ferie til de tilbageværende medarbejdere, der fik den laveste prioritet sidste år. Men for at undgå en algoritme-bias efter f.eks. alfabetisk rækkefølge af navne, randomiseres rækkefølgen af hver gruppe. Så hver gang algoritmen kører, vil alle dem der f.eks. fik deres førsteprioritet sidste år, blive allokeret i en ny tilfældig rækkefølge. Dette medfører, at algoritmen kan producere forskellige resultater ud fra identiske input. Der kan både argumenteres for at dette er en “bug” og en feature, men en fordel er, at man med forholdsvis små justeringer af algoritmen, kan sætte algoritmen til at køre 5, 10 eller 100 gange og så tage det bedste resultat den finder.

Koden for sortering og gruppevis randomisering er inkluderet herunder, og udnytter LINQ-bibliotekets forskellige muligheder for manipulation af kollektioner. Først grupperes den samlede liste af medarbejdere efter sidste års opnåede ferieprioritet, dernæst randomiseres rækkefølgen inden for hver gruppe ved at give hver medarbejder et tilfældigt genereret ID og sorterer efter det, og til sidst samles medarbejderne igen i en til en samlet liste. Dette sker alt sammen ved brug af en samlet sekvens af LINQ-metoder og er et kraftfuldt eksempel på, hvor koncis kode, man kan skrive med LINQ.

```
// Liste af uallokerede lagermedarbejdere
var groupedAndOrderedEmployees = unallocatedEmployeeList

    // Gruppér medarbejdere efter hvilken prioritet de fik sidste år (null-værdier sættes til 1)
    .GroupBy(employee => employee.lastYearsPriority ?? 1)

    // Randomiser rækkefølgen inden for hver gruppe
    .Select(group => new
    {
        Priority = group.Key, // (1, 2, 3, 4)
        Employees = group.OrderBy(employee => Guid.NewGuid()).ToList() // Randomize order
    })

    // Sorter grupperne i faldende rækkefølge efter sidste års prioritet (4, 3, 2, 1)
    .OrderByDescending(group => group.Priority)
    .SelectMany(group => group.Employees) // Saml grupperne til én samling igen.
    .ToList();
```

Til sidst har vi et eksempel på et udsnit af et output, algoritmen har produceret for en juleferie hvor alle medarbejdere har ønsket to feriedage. Øverst har vi datoer. Til venstre har vi medarbejdernavne og -numre og den ferieønskeprioritet de har fået tildelt. Ved siden af ses et skema over tildelt ferie for ferieperioden. Tallene for minimumsbemanding viser hvor langt fra minimumsbemandingskravet, lageret samlet set er for hver dag: Så hvis der eksempelvis står 20, viser det at der stadig kan allokeres 20 medarbejdere mere for den givne dag, uden at overskride minimumsbemandingskravet. Logikken er den samme for kompetencerne nedenunder.

			lørdag	søndag	mandag	tirsdag	onsdag	torsdag	fredag	lørdag	søndag	mandag	tirsdag	onsdag	torsdag
Navn	Nummer	Prioritet	21/12/2024	22/12/2024	23/12/2024	24/12/2024	25/12/2024	26/12/2024	27/12/2024	28/12/2024	29/12/2024	30/12/2024	31/12/2024	01/01/2025	02/01/2025
Kevin	12	1		FERIE	FERIE										
Umar	22	1		FERIE	FERIE										
Oscar	16	1						FERIE	FERIE						
Ian	10	1										FERIE	FERIE		
Zachary	27	1							FERIE	FERIE					
Quentin	18	1								FERIE	FERIE				
Jane	11	1											FERIE	FERIE	
Aaron	28	1								FERIE	FERIE				
Paula	17	1							FERIE	FERIE					
Yvonne	26	1						FERIE	FERIE						
Status på minimumsbemanding			20	18	17	17	17	16	15	15	15	15	17	20	21
Status på kompetencekrav															
amm			3	2	2	2	1	1	2	2	1	1	1	2	3
dispatch			9	7	7	9	9	7	7	7	6	6	7	9	10
flow_1			3	2	1	2	3	3	2	0	0	1	1	2	3
flow_2			12	11	10	9	10	9	8	9	9	9	11	13	13
flow_3			12	11	10	9	10	11	9	8	9	9	10	12	13
flow_4_maskine			8	8	8	8	6	3	4	5	4	4	5	7	8
flow_4_pak			8	6	5	7	6	4	6	6	4	4	6	8	8
flow_5			9	8	7	7	9	10	9	7	5	4	6	9	10

# Konklusion (refleksion og perspektivering)

Arbejdet med denne rapport har givet os en dybere indsigt i systemudvikling, projektorganisering, og anvendelsen af moderne teknologier og metoder. Vi har fokuseret på at opbygge en datamodel, der imødekommer både teknologiske krav samt brugernes behov, samtidig med at vi har implementeret robuste løsninger, der sikrer fleksibilitet og fremtidig skalerbarhed. Vores valg af MVVM som designmodel, samt brugen af databasepraksisser, såsom transactions og repository mønstre, har bidraget til at skabe en velstruktureret og effektiv softwareløsning.

## Threads og processer

Vi har valgt at undlade brugen af tråde og processer i vores program, da det primært skal køres på én pc med én bruger ad gangen. Brug af tråde kunne optimere processer ved at udføre opgaver simultant, men da vores nuværende algoritme kun tager omkring et halvt sekund, har det ikke været bydende nødvendigt.

Dog kan brugen af tråde eller processer blive relevant i fremtiden, hvis programmet skal håndtere flere samtidige brugere, eller hvis opgaverne kræver tungere databehandling, der kan udnytte flere ressourcer parallelt. På den måde vil tråde og processer bidrage til at øge programmets ydeevne og responsivitet.

## Transactions vs. triggers

Vi har valgt at bruge transactions frem for triggers, fordi transactions giver os mere kontrol over, hvordan logikken håndteres i applikationen. Triggers gemmer logik i selve datasen, hvilket kan gøre det sværere at gennemskue, hvor og hvornår bestemte handlinger udføres. Dette kan gøre debugging mere komplekst og kan føre til u hensigtsmæssige kædeoperationer, når triggers trigger andre triggers.

## Commands inline eller i metoder

Vi valgte primært at implementere Commands inline for at holde koden kompakt. Denne tilgang gør logikken for den enkelte Command synlig direkte i konteksten og reducerer antallet af små fragmenterede metoder i ViewModelen.

I DCDét noterede vi Commands som metoder, da det bedre repræsenterer deres funktionelle karakter, end som properties. Selvom opsplитning i metoder kan gøre logikken mere genbrugelig og let at isolere, vurderede vi, at inline Commands var tilstrækkelige, da de var knyttet til specifikke funktioner.

At arbejde med et projekt af denne størrelse har givet os mulighed for at reflektere over vigtigheden af at vælge de rigtige teknologier og metoder til forskellige problemstillinger. I fremtidige projekter kunne vi med fordel fortsætte med at udforske anvendelsen af tråde og processer, især hvis systemet skal kunne håndtere øget belastning eller parallelisering af opgaver.

Gennem projektet har vi haft et tæt samarbejde med Alfa Laval Kolding for at optimere deres ferieplanlægningsproces. Ved at udvikle en digital løsning baseret på en brugervenlig grænseflade og avancerede algoritmer, har vi løst de udfordringer, som det manuelle system tidligere medførte – nemlig tidskrævende arbejde, fejlrisiko og problemer med forskelsbehandling. Vores løsning sikrer en mere effektiv ferieplanlægning, samtidig med at der tages højde for både medarbejdernes ønsker og virksomhedens kompetence- og bemandingskrav. På den måde er vi lykkedes med at imødekomme Alfa Lavals behov for en mere moderne og intelligent tilgang til ferieadministration.

På baggrund af vores arbejde med at udvikle et digitalt ferieplanlægningssystem til Alfa Laval Kolding kan vi konkludere, at projektet har givet os både praktisk og teoretisk forståelse af systemudvikling, projektorganisering og anvendelsen af teknologi. Hvor den manuelle planlægning tidligere var præget af tidskrævende koordination, fejlrisici og bekymringer om fairness, vil vores løsning blive omsat til et koncept, der bedre imødekommer lederledernes behov og krav. Arbejdet er sket i tæt dialog med Alfa Laval Kolding, og har sikret, at de identificerede behov er forstået og omsat til en funktionel model, som i praksis kan lette arbejdsbyrden for lederne og samtidig tage hensyn til medarbejdernes individuelle ønsker. Alt i alt har vores arbejde med projektet givet os en bedre forståelse af, hvordan vi kan udvikle digitale løsninger, der rent faktisk passer til en virksomheds behov. Selvom systemet endnu ikke er implementeret, giver vores model et godt udgangspunkt for at gøre ferieplanlægningen mere effektiv, gennemskuelig og retfærdig.

# Bilag

## Bilag 1;

### **Usecase 1: Organiser feriedata som forberedelse til ferieplanlægningen**

En leder på lageret fastlægger start- og slutdato for en given ferie. Hvis der opstår behov for ændringer, udføres tilpasninger.

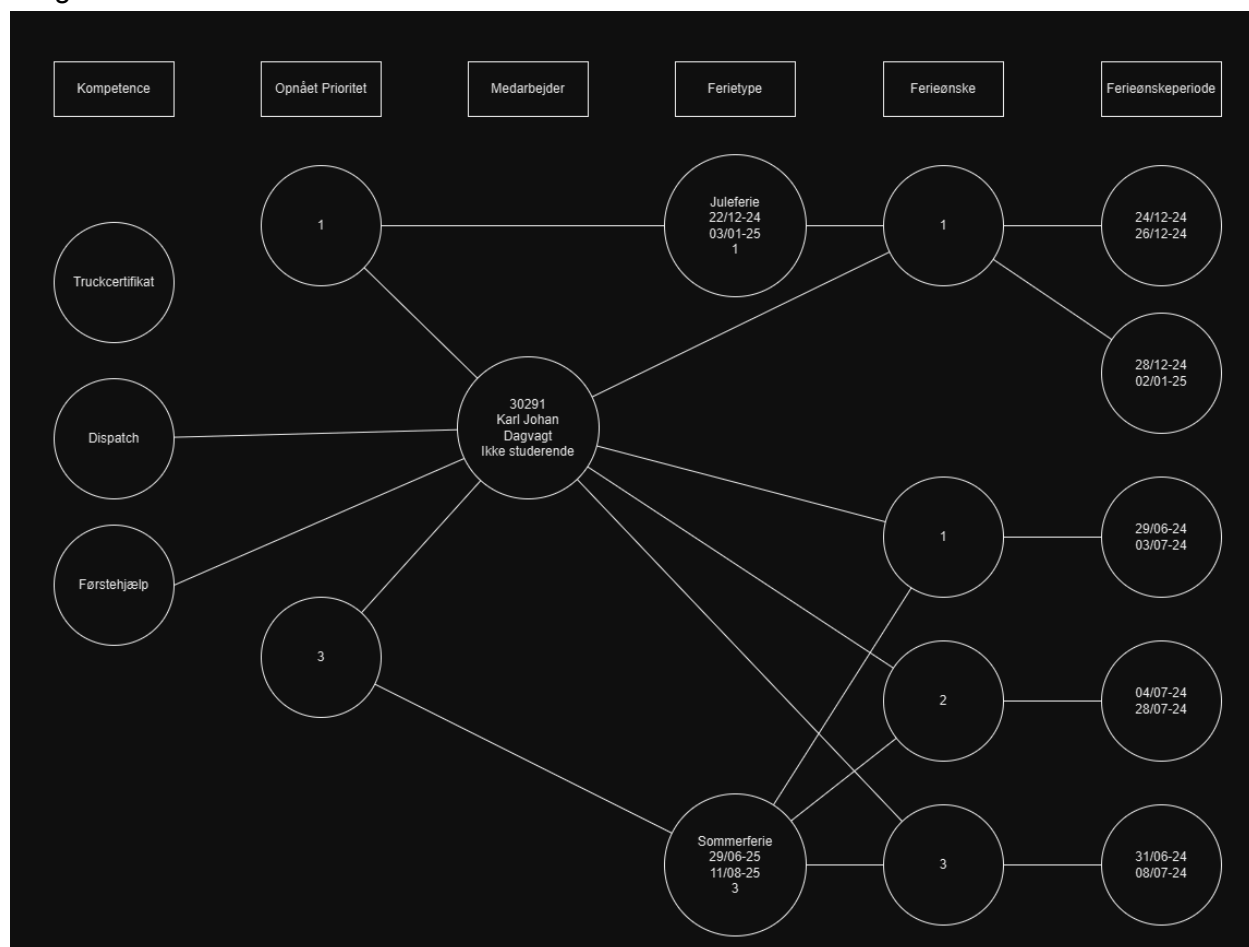
### **Use-case 2: Organiser medarbejderdata som forberedelse til ferieplanlægningen**

En leder på lageret organiserer virksomhedens information om lagermedarbejderne, herunder deres kompetencer og ferieønsker, på en overskuelig måde. Hvis nogle informationer ikke er tidssvarende eller på anden vis fejlbehæftede, tilpasses disse.

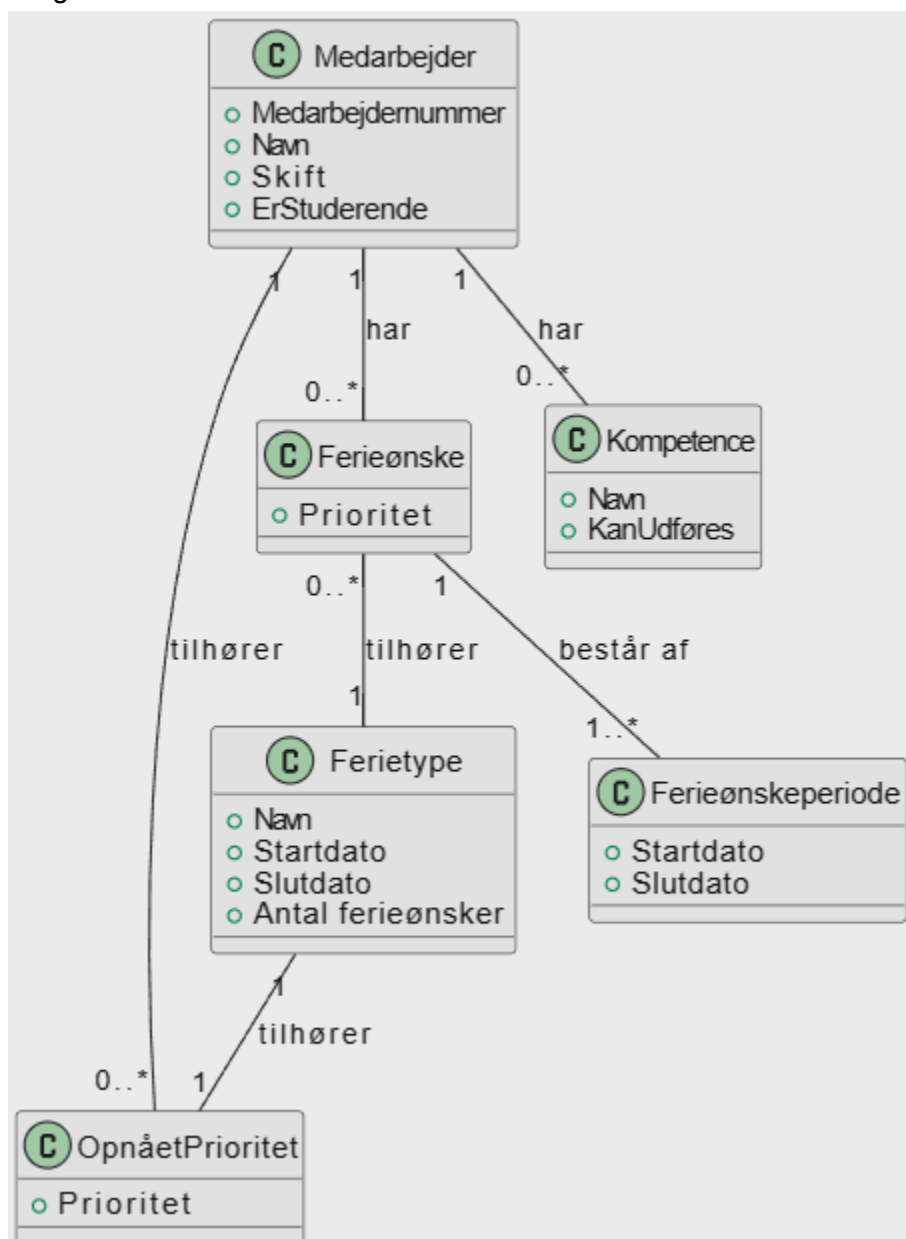
### **Usecase 3: Udformning af ferieplan for lagermedarbejdere**

En leder på lageret udformer på baggrund af alle relevante information, herunder medarbejdernes ferieønsker, deres opnåede ferieprioritet for sidste år, lagerets nødvendige minimumsbemanding og -kompetencekrav, en samlet ferieplan for den givne ferie.

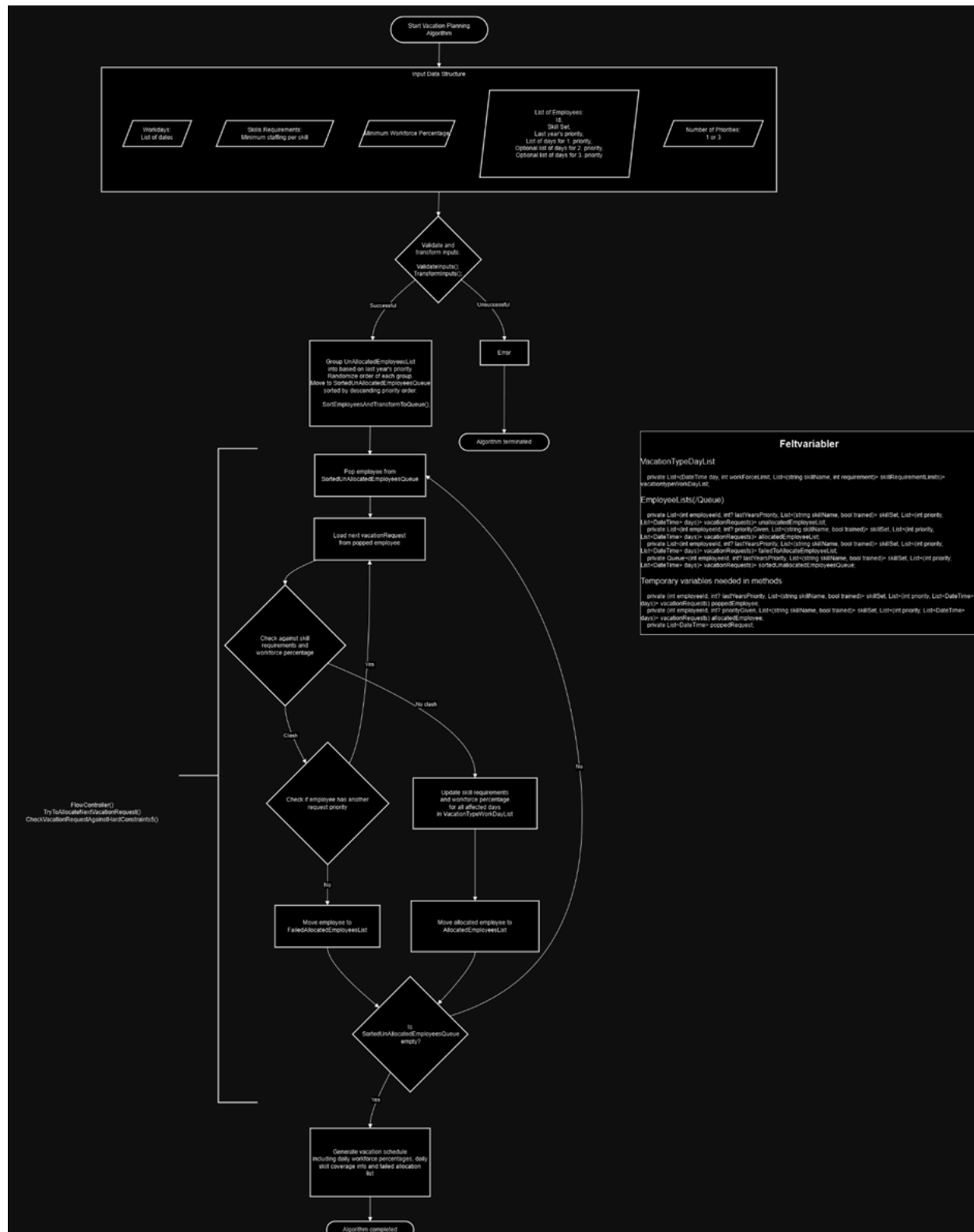
## Bilag 2:



Bilag 3:

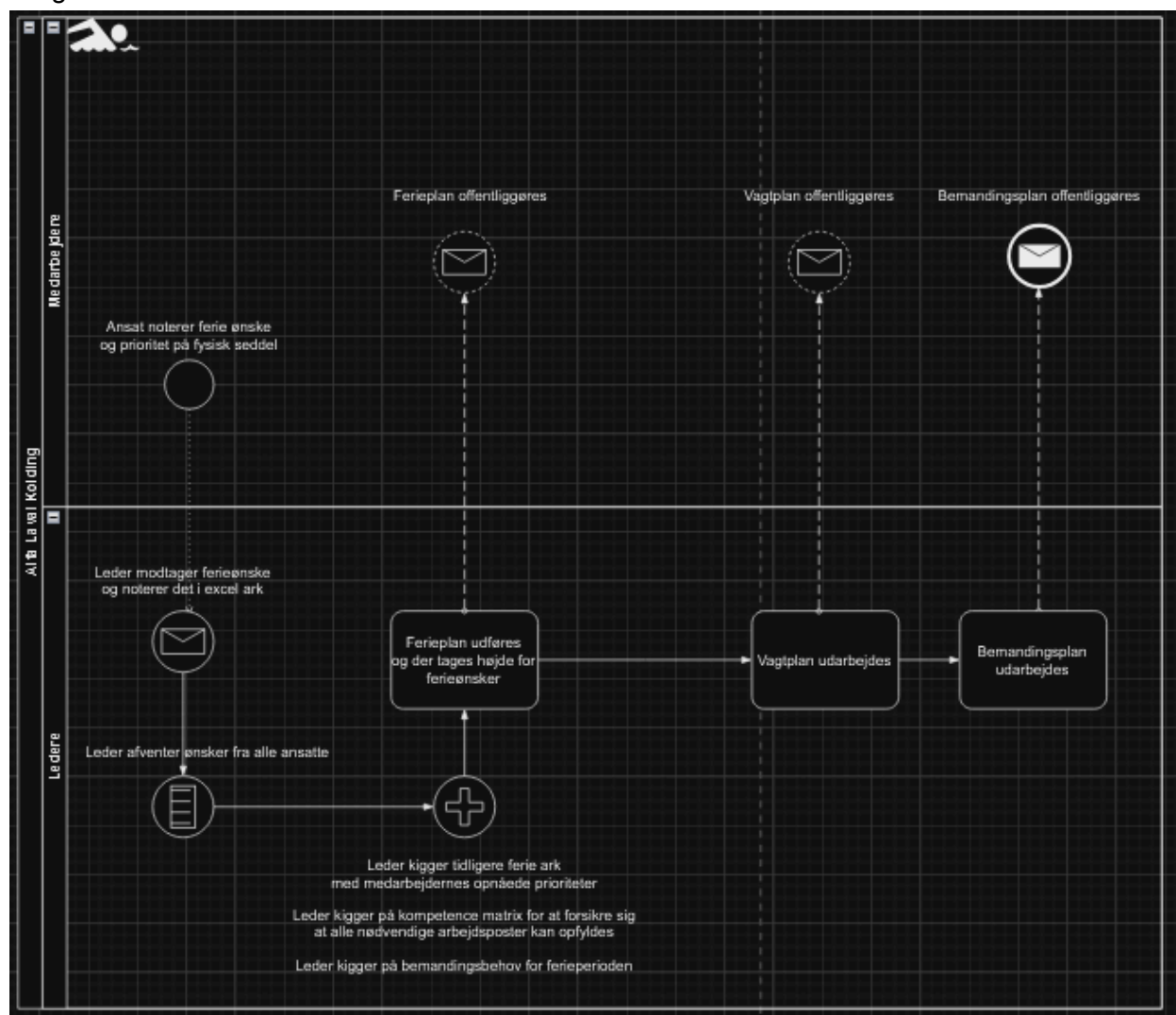


## Bilag 4:

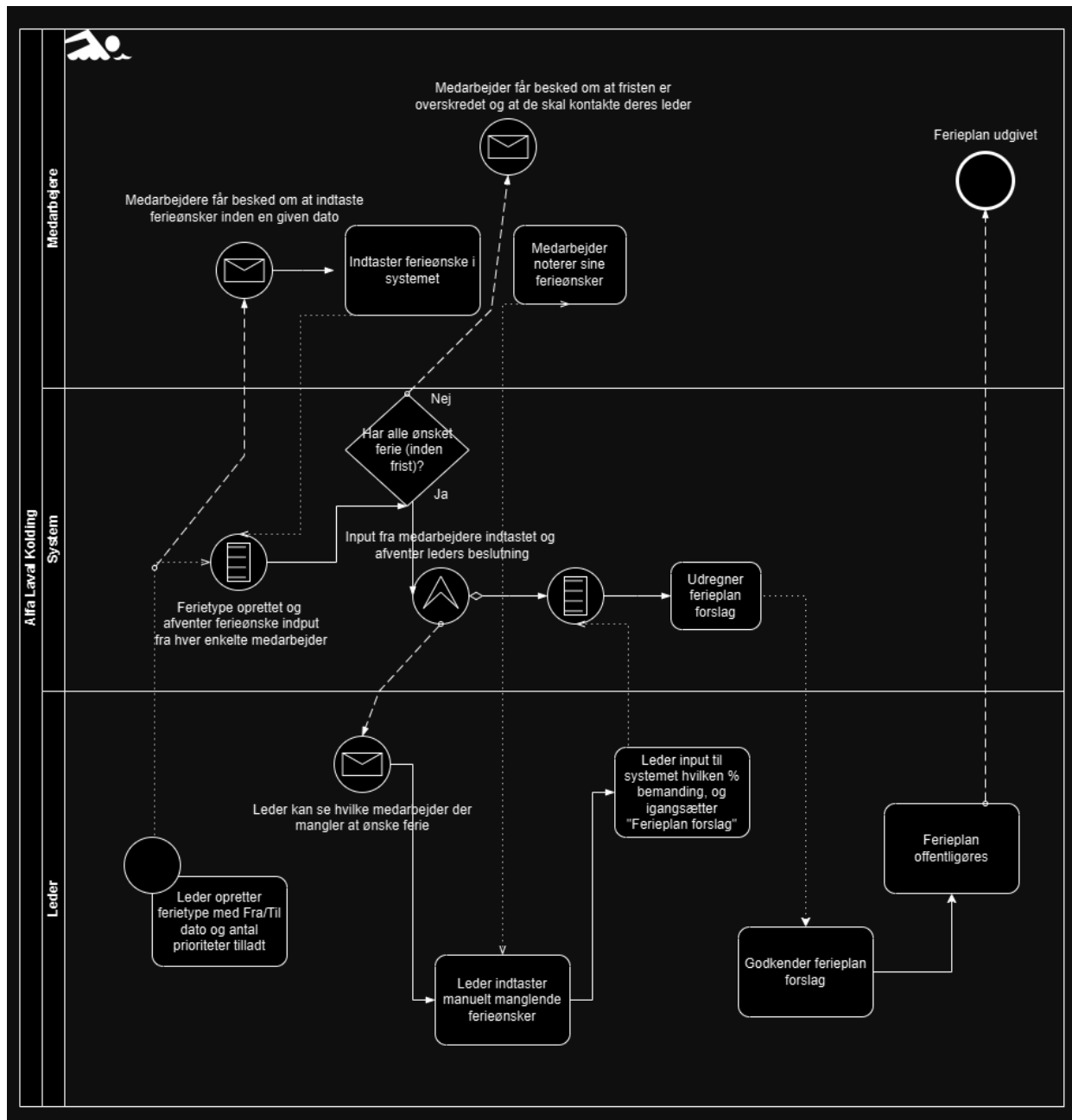




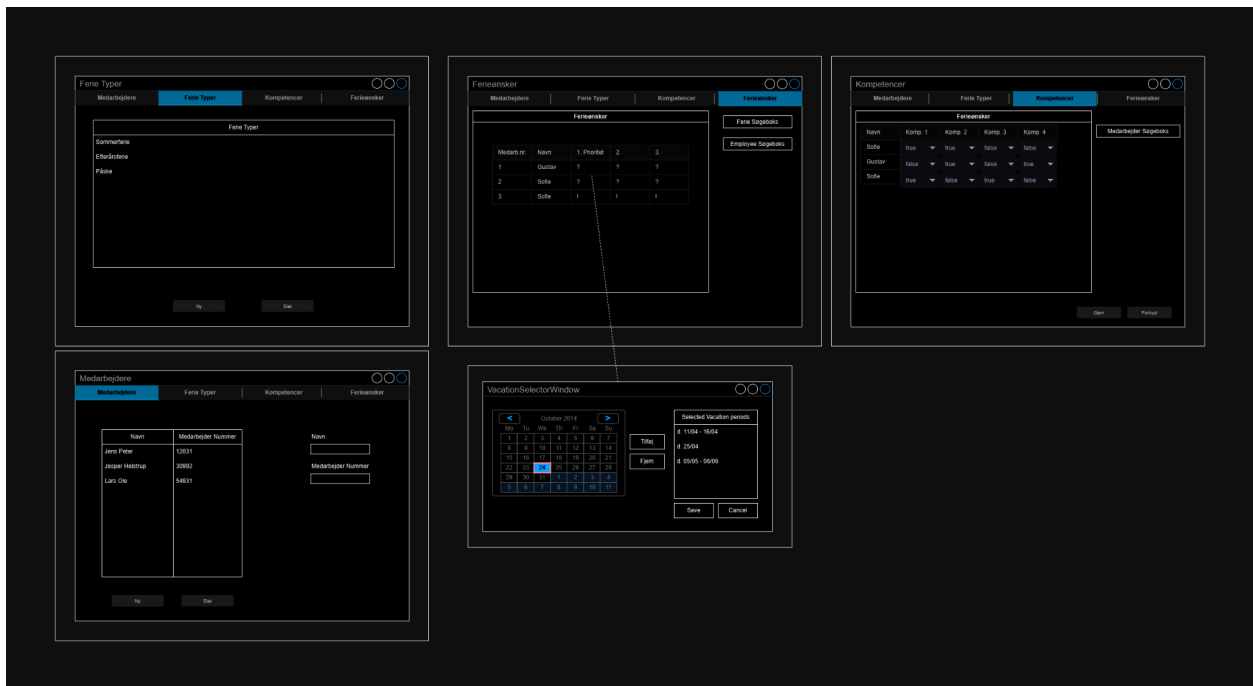
## Bilag 5:



# Bilag 6:



## Bilag 7:



## Bilag 8:

