Optimering af hjemmeplejen

Aalborg Universitet 2014

Dat/SW 3 - P3

AALBORG UNIVERSITET
STUDENTERRAPPORT

3. semester
Software / Datalogi
Cassiopeia - Selma Lagerløfsvej 300
9220 Aalborg Øst

Titel:
P3 - Optimering af hjemmeplejen

Tema:
Udvikling af applikationer – fra brugere til data, algoritmer og test – og tilbage igen

P3 projektperiode:
1. september 2014 - 19. december 2014

Projektgruppe:
ds305e14

Gruppemedlemmer:
Benjamin Ahm
Emil Riis Hansen
Kasper Kohsel Terndrup
Kristian Hauge Jensen
Martin Hammer Thygesen
Morten Korsholm Terndrup

Vejleder:
Peter Axel Nielsen

Oplagstal: 8

Sideantal: 119

Synopsis

Denne rapport omhandler et system der løser et planlægningsmæssigt problem i hjemmeplejen i Aalborg. Systemet støtter planlægningsopgaver, med funktionalitet til automatisk planlægning af ruter. Rapporten giver desuden indsigt i tanker og vervejelser angående designet og udarbejdelsen af programmet, samt en gennemgang af de forskellige aktiviteter i objektorienteret analyse og design der er brugt under systemudviklingen. Programmet er skrevet under det objektorienterede paradigme, og er implementeret som et C# program med henblik på at skulle kunne køre på en computer med Windows styresystem. WPF er benyttet til at udvikle systemets brugergrænseflade. En prototypetestning af systemet, samt en aftestning af det systemet i den endelige tilstand, er blevet foretaget. Aftestningen gav mulighed for en evaluering af programmets, samt anledning til overvejelser og forbedringer af en viderearbejdelse af systemet.

Forord

Denne rapport er skrevet som en del af et semesterprojekt, for tredje semester på studieretningen datalogi/software på Aalborg universitet, og har det primære formål at opfylde de krav, der er opstillet i semesterbeskrivelsen og de tilhørende læringsmål[1].

Læsevejledning

Rapporten lægger ud med en gennemgang af de metoder vi har anvendt under projektet, og begrundelsen for valget af disse. Herefter bliver en gennemgang af hjemmeplejens situation præsenteret. Dette danner grundlag for problemformuleringen i kapitel 4, der er udgangspunktet for problemløsningen.

Med udgangspunkt i problemformuleringen, vil resultatet af en dybere analyse blive præsenteret, som danner grobund for de efterfølgende designbeslutninger. Herefter vil implementeringsspecifikke valg for problemløsningen blive forklaret, ved både abstraktion og konkrete eksempler.

Efter implementering vil kvalitetssikringen af løsningen blive gennemgået og vurderet. Til slut vil der være en evaluering af projektet, i form af en diskussion og konklusion.

Det anbefales at rapporten læses kronologisk, da der af hensyn til minimering af gentagelser kun vil blive refereret til forudstående begrundelser, hvor der er eksplicit behov for det.

Indhold

1 Indledning

Hjemmeplejen er ofte et debatteret emne i dansk politik, og tages ofte op af medierne. Der bliver flere ældre i Danmark, hvilket sætter større økonomiske krav til velfærdssamfundet. I daglig tale benævnes det ofte som ældrebyrden. Selvom antallet af ældre over 65 er steget med 115.000 i tidsperioden 2008-2013, er der 14% færre hjemmehjælpsmodtagere. Det kommer af at kommunerne har nedjusteret mængden af pleje der udføres, og at det er sværere at kvalificere sig til den[2].

Denne udvikling tyder på en voksende samfundsmæssig konflikt, mellem ønsker om en offentlig plejeordning der er både effektiv, billig og personlig. Disse tre ønsker er svære at forene, uden at foretage nedskæringer i andre aspekter, som eksempelvis lønninger af plejepersonale eller ved at øge arbejdsbyrden. Hvis ikke der ønskes en forværring af hjemmeplejen, skal der foretages ændringer og muligvis en omlægning af hjemmeplejens arbejdsgange.

Med dette samfundsmæssige problem i tankerne var motivationen for dette projekt at undersøge nærmere, hvorledes et system kan udvikles, som kan afhjælpe problemstillinger i hjemmeplejen.

Vi har i den sammenhæng været i dialog med mennesker indenfor hjemmeplejen for at opnå forståelse for deres situation og arbejdsgange, samt for at identificere mulige problemstillinger. Ligeledes har vi gennem interview og afprøvning af prototyper kunnet evaluere vores forståelse af kravene til systemet. Denne menneskecentrede tilgang til projektet var oplagt, da vi ønskede at inddrage de menneskelige aspekter i modsætning til blot at opfylde en given kravspecifikation.

Problemstillingen, systemet har til formål at løse, består i, at ineffektiv planlægning fører til ruter, hvor der bruges mere tid på transport end nødvendigt. Med bedre ruter ville der frigøres tidsresourcer, der med fordel kunne anvendes til at forbedre andre dele af hjemmeplejen. Problemet ligger bl.a. i, at deres nuværende planlægningsværktøj ikke tager højde for transporttid, og at det er uoverkommeligt manuelt at lave en effektiv fordeling af besøg, der minimerer transporttiden.

Udviklingsprocessen er blevet foretaget på baggrund af aktiviteter, der står beskrevet i bogen *Objektorienteret Analyse & Design*[3]. Brugergrænsefladen er i høj grad baseret på teori og værktøjer fra bogen *Designing Interactive Systems*[4].

Vi har analyseret og designet for det komplette system, men har været nødt til at afgrænse os til kun at implementere en del af det. Dette skyldes den begrænsede tid vi har haft til projektet. Vi har imidlertid haft hele designet i tankerne under implementeringen, således at den kan videreudvikles i fremtiden.

2 Metode

Dette afsnit redegører for valg af metode samt overvejelser og præmisser for analyse, design, implementering og test. Afsnittet redegører desuden for, hvorledes vi har identificeret problemer i hjemmeplejen samt krav til disse problemer.

2.1 Objektorienteret Analyse & Design

Til systemudviklingen har vi anvendt den objektorienteret analyse og design metode, der fremgår i bogen *Objektorienteret Analyse & Design*[3]. Formålet med metoden er at modellere alle væsentlige dele af udviklingsprocessen gennem en systematisk gennemgang af systemets problemområde, ansvarsområde, arkitektur og komponenter.

Dette gøres så forståelse for krav og vilkår til systemet, fastlægges på baggrund af omgivelserne, således at systemudviklingen bibeholder sin relation til virkeligheden igennem hele processen. *OOA&D* metoden modellerer virkeligheden vha. en analyse af problemområdet samt ansvarsområdet.[3]

På baggrund af denne analyse fastlægges et design for systemets komponentarkitektur. Denne arkitektur danner rammerne for komponentdesignet, der definerer hvordan komponenterne tilser systemets krav. Når disse emner er blevet gennemgået, vil man have et design, der lægger sig til implementering med et objektorienteret programmeringssprog.

2.2 Iterativ udviklingsmodel

Den iterative arbejdsform har været den primær dominerende under systemudviklingen. Denne er blevet valgt på baggrund af en vurdering af kompleksiteten og usikkerheden. Med kompleksitet menes der mængden og diversiteten af relevant information, der er påkrævet for at løse problemet[5].

Vurderingen af kompleksiteten og usikkerheden, blev foretaget på baggrund af undersøgelsen af hjemmeplejens situation, som er dokumenteret i afsnit 3. Denne undersøgelse skulle identi-

ficere det konkrete problem i hjemmeplejen vi arbejdede med. Problemet viste sig at have høj usikkerhed, da det var uklart hvordan det kunne løses. Vi anvendte derfor den iterative metode, til gradvist at opnå større forståelse af problemet, indtil det var klart hvilke muligheder der var for at løse problemet. Den mulighed vi valgte til at løse problemet, var af høj kompleksitet, hvilket betød at vi skulle inddrage mange informationer for at opfylde kravene til løsningen.

Iterativ arbejdsform indebærer at man gentagende gange gennemgår udviklingsaktiviteter, med det formål at inddrage viden, der er blevet afdækket løbende. Dette sikrer kvaliteten af aktiviteternes resultaterne, og derved kvaliteten af det udviklede system. Ved iteration er det ofte nødvendigt at genoverveje tidligere beslutninger, hvilket gør det svært at afgøre hvor langt projektet er i sit udviklingsforløb. Dette kan ofte betyde at der bliver gået på kompromis med avanceret funktionalitet, da systemudvikling tit foretages op imod en deadline.

Udviklingsprocessen for dette projekt, står beskrevet i kapitel 9.

2.3 Informationsindsamling

Vi startede vores informationsindsamling med at gå i uformel dialog med to personer, som har eller har haft tilknytning til hjemmeplejen. Vi ønskede ud fra denne dialog at få et initierende indblik i hjemmeplejen, som vi fik i form af *stories* jvf. *Designing Interactive Systems*[4]. Disse stories gav os en afklaring af hvilke mulige problemer der kunne være i hjemmeplejen.

Det var det muligt ud fra disse *stories* at få et indblik i hvordan arbejdsgangen for en hjemmehjælper ser ud. Det gennemgående udtryk fra hjemmehjælperne var, at de synes, de har for lidt tid med de ældre, når de er ude på besøg, samt at de følte ruterne kunne være mere effektive. Derfor blev dette vores udgangspunkt i forhold den nærmere undersøgelse af situationen.

Vi valgte dernæst at planlægge interviews med hjemmeplejen i Aalborg, for at kunne få et mere præcist indblik i planlægningen og udførelsen af hjemmepleje, men også for at identificere yderligere problemstillinger.

Alle interviews blev udført som semistrukturerede[4], da vi havde et udgangspunkt fra vores *stories*, men stadig ville udforske yderligere problemstillinger. Vi lavede en liste med åbne spørgsmål, der støttede under udførelsen af interviews. Spørgsmålene kan ses i bilag 12.3. Vi valgte at tage en lydoptagelse af alle interviews, for at indfange flest mulige informationer fra de tre møder.

Vi udarbejdede senere en interaktiv prototype, ud fra de krav som de første interviews afdækkede. Prototypen blev testet af en af de ovennævnte planlæggere, ud fra en række *use-cases*, vi havde formuleret. En mere dybdegående forklaring af aftestningen kan læses i afsnit 8.1. Informationen samlet ved denne prototype test, brugte vi efterfølgende til at validere kravene prototypen baseret på, samt at identificere nye krav til systemet.

3 Hjemmeplejens situation

Aalborg Kommunes hjemmehjælp er opdelt i forskellige områder, hvor hver af disse områder bliver styret fra lokale afdelinger. Vi afholdte interviews med Aalborg Kommunes hjemmepleje, samt den private hjemmepleje-virksomhed Attendo, der har fået udliciteret en del af Aalborg Kommunes område. Hos Aalborg Kommune foretog vi et interview med en planlægger, og et interview med en SOSU-medarbejder. Hos Attendo interviewede vi en planlægger. Dette kapitel indeholder en beskrivelse af hjemmehjælpens situation som vi forstår den, efter interviews med afdelinger af hjemmehjælpen.

3.1 Nuværende system

Alle afdelinger i Aalborg Kommune er pålagt at bruge IT-systemet Care. Formålet med Care er, at sundhedssektorens institutioner har et fælles kommunikationssystem til informationsdeling.

Care indeholder et internt beskedsystem, kaldet *Advis*, der gør det muligt at kontakte forskellige faggrupper, f.eks. læger, visitator eller hjemmeplejen. Visitationer, fagterminologi for den officielle vurdering af den hjemmehjælp en borger har behov for, sendes af visitator over Care. Man kan tilknytte informationer til individuelle borgere. Når information er tilknyttet, skal en faggruppe, som skal have disse oplysninger, vælges. Hermed sendes informationerne, til den del af faggruppe med tilknytning til den pågældende borger.

Hjemmeplejen anvender Care til at foretage planlægning. I Care ligger blandt andet ruter med adresser, der skal besøges. Ruterne fungerer således, at hvis man har en rute bestående udelukkende af f.eks. Klarup-adresser, og en borger der bor i Klarup bliver udskrevet af sygehuset, tilføjes borgeren manuelt af planlæggeren til "Klarup-ruten", hvis dette er muligt.

Når en borger bliver udskrevet fra hospitalet, får den tilknyttede hjemmepleje information via Care. Hvis borgeren har været på hospitalet i under 24 timer, bibeholdes retten til at modtage hjemmehjælp samme dag. Hvis borgeren derimod har været indlagt i over 24 timer, skal hjemmeplejen vide det mindst et døgn i forvejen. Det bliver ofte varslet en uge før, hvis borgeren skal have "fuld hjælp", f.eks. hvis borgeren er lam. I interviewet blev der udtrykt, at kommunikationen mellem sygehusene og hjemmeplejen var ofte var forvirrende, da vigtig

information ofte var pakket ind i vedhæftede filer, der indeholdte meget irelevant information om borgerne.

Care har brugergrænseflader til visning af ruterne, de enkelte medarbejderes ruter og persondata på hver enkelt borger.

Care findes både til PC og iPad. Hjælperne har kun adgang til de data i Care, der er relevante for dem. iPad versionen gør det muligt for hjælpere at medbringe ruter, lægejournaler og anden information om borgerne. Planlæggeren har lov til at bringe deres iPads med hjem.

Hvis en hjælper kommer ud for uforudsete hændelser såsom et problem hos en borger, der resulterer i, at det tager længere tid end forventet, kan hjælperen selv rette i sin rute ved hjælp af Care iPad systemet.

3.2 Aalborg Kommunes afdeling

I Aalborg Kommunes afdeling i Aalborg Øst foregik ingen dagvagter, så planlægningen blev udført om dagen, mens vagterne blev udført om aftenen. Denne afdeling dækker over ældreområde øst, som bl.a. består af Klarup, Gistrup og Vejgaard. Typisk vil der være en til to planlæggere i afdelingen. De benytter sig af Care systemet som et planlægningsredskab til at ændre ruterne.

Dagen starter med at tilrettelægge planen i forhold til sygemeldinger. Dette kan ske ved at indkalde hjælpere som har ønske om ekstra vagter eller finde vikarer fra et internt vikarbureau. Hvis dette ikke kan lade sig gøre, må et eksternt vikarbureau tilkaldes. De foretrækker at holde sig til egne rækker, da et eksternt bureau er dyrt at benytte, og de har et budget, der skal overholdes.

Planlægger vil tidligt hente en kurv som hjælperne kan lægge sedler i. Disse sedler bliver brugt til at notere:

- Hvis hjælperen har udført et nødkald, og dermed brugt ekstra tid.
- Generel, men uhensigtsmæssig, dokumentering af aftenvagterne, fordi:
  - Det er hurtigere at skrive på papir fremfor systemet.

  - Der er behov for notering af andet end hvad systemet tillader (f.eks. at dokumentere hvad der er sket ved nødkald til alarmcentralen),
  - Netforbindelsen er dårlig.

- Navn og CPR-nummer på en person, hvor en ydelse skal stoppes.
- Hvis hjælperen har brug for mere tid end hvad der er visiteret til hos en borger.
- Vikarernes indberetninger, da de ikke har kode til Care.
- Hvis planlæggeren har spørgsmål til hjælperne, som de så kan besvare via sedler.

Fordi det er en aftenvagt, bliver hjælperne nødt til at ringe til hinanden og håndtere det indbyrdes, hvis der sker uforudsete hændelser. Afdelingen har 15 hjælpere på 15 ruter hver aften, hvoraf tre møder i Storvorde, da det er tættere på deres ruter. En vagt er på otte timer.

Nye ruter laves hvis der kommer mange nye hjemmehjælpsmodtagere. Ruterne er lavet således at der tages højde for følgende:

- På gåruter (de steder hvor der er ældreboliger) kan man have mere på sin rute da køretiden er kortere. Gåruter har typisk en ATA-tid 6,5 timer.[1]
- Der tages højde for pauser.
- Hjælpere der skal ud til Mou har høj køretid og dermed kun typisk en ATA-tid på 4,5 timer.
- Afhænger af hvilken opgave der er visiteret til, og hvornår den opgave forekommer.
- Om der skal to hjælpere til visitationen.
- Demente hjemmehjælpsmodtagere da disse gerne skal være tilknyttet de samme hjælpere.

Opgaver, der skal ske i bestemte tidsintervaller, kan være aftensmad eller når de skal lægges i seng. Borgere der skal have medicin, skal have det på bestemte tidspunkter.

Alt hvad der registreres elektronisk af hjælperen skal planlægger ind at kigge på. Når en rute laves, bruges rutevejledningsservicen „Krak.dk“ til at finde en fornuftig rækkefølge for besøg hos borgerene. Hvis der skal tilføjes en borger, som f.eks. er visiteret til en time, kigger

[1] ATA-tid er ansigt-til-ansigt tid som betegner den samlede tid en hjemmehjælper er ude hos borgerne kontra transport og pause tid.

planlægger på ruter, hvor der er opstået huller, og tilføjer besøget der. Det hænder at et besøg placeres på en rute, som ligger langt væk fra de andre besøg på ruten.

Der findes situationer hvor to hjælpere er visiteret til et besøg (f.eks. hjælp til alt sengeliggende). I nogle situationer er der behov for to hjælpere til ét besøg, selv om der kun er visiteret én hjælper til besøget, f.eks. pludseligt at skulle forflytte[2] en borger.

Planlægger tildeler typisk en rute til to hjælpere (dog kun en hjælper per vagt), således at det er samme hjælpere, der er ude hos borgeren, med formålet at skabe tryghed hos borgeren. Afdelingen har p.t. to ruter som kun en hjælper er sat på; resten er ruter med to hjælpere.

3.3 Attendo Aalborg SV

I Attendo A/S, som er en privat hjemmepleje udbyder, foregik planlægning samt besøg ved borgerne i dagtimerne. Denne afdeling har af Aalborg kommune fået udliciteret ældreområdet sydvest samt Skalborg. Der er kun én planlægger i afdelingen, som har ansvaret for at planlægge ruterne i Care systemet.

Dagen for planlæggeren starter med at vedkommende samt størstedelen af de ansatte møder ind klokken syv om morgenen. Her får alle de ansatte deres ruter. Disse kan enten blive givet på papir eller iPad, men som oftest vil iPads være at foretrække. Herefter fortsætter planlæggers dag med at planlægge de følgende dage, dette kan indebære:

- Finde afløsere ved sygemelding, med højst prioritering i egne rækker, dernæst i eksternt vikarbureauet.
- Modtage advis fra visitatorer og foretage ændringer i Care systemet.
- Ændre ruter i forhold til borgeres tilbagemeldinger/fraværsmeldinger

Hvis ikke det er muligt at finde en afløser fra afdelingen selv ved sygemelding, har Attendo også en afdeling i Svenstrup som kontaktes før vikarbureauet. Derudover står planlægger også til rådighed i løbet af dagen, således at hjemmehjælperne kan ringe ind hvis de står i situationer, hvor de er i tvivl. Alle social- og sundhedshjælpere har under deres uddannelse fået en kort oplæring i brugen af Care. Herudover er planlæggeren sidemandsoplært af en planlægger fra Kolding, samt af en IT-ekspert fra Aalborg Kommune.

[2] *At forflytte en borger* er fagterm for *at løfte en borger*.

3.3.1 Hjælpe-personale

Hver afdeling i Aalborg Kommune har en gruppe hjælpere til rådighed, hvoriblandt én er iPad nøgleperson. Dette betyder at afdelingen har en person med specielt meget viden indenfor iPad systemet. Der findes social- og sundhedshjælper, kaldet SSH, og social- og sundhedsassistenter, kaldet SSA eller assistenter. Hertil findes der også enkelte med en gammel uddannelse, kaldet hjemmehjælpere. SSA har viden indenfor medicinering, og har derfor også mulighed for at give borgerne ekstra ydelser, som ellers kun måtte udføres af en sygeplejerske, såsom at dosere medicin til borgere.

Der er ingen forskel på alder eller erfaring indenfor hjemmeplejen, hvilket vil sige at en erfaren hjemmehjælper og en nyuddannet kan blive udsat for de samme opgaver. Hjælperne har selv en indflydelse på hvor lang tid, der skal sættes af til den enkelte borger. Hvis hjælperen vurderer, at det ikke kan lade sig gøre at udføre opgaven på den visiterede tid, kan dette indberettes, hvorefter visitatoren besøger borgeren igen, og tidsafgørelsen revurderes.

En hjælper får i teorien en ny rute hver dag, dog oftest med de borgere, de er vant til at besøge. Da der kan ske uforudsete hændelser, såsom en indlæggelse af en borger, ændres planen nogle gange. Pga. dette og at systemet ikke sker i realtid, bliver hjælperen nødt til at opdatere systemet for at se de nyeste ændringer.

Normalt er der kun påsat én hjælper pr. rute. Hvis to hjælpere skal bruges til en opgave, overlapper deres rute ved de borgere, hvor der skal bruges mere end en.

4 Problemformulering

Dette kapitel omhandler vores valg af problemformulering samt systemdefinitionen, vi har lavet, der skal løse dette problem. Beslutningerne er taget på baggrund af de problemstillinger, der blev fundet ved undersøgelse af hjemmeplejens situation, der beskrives i kapitel 3.

Her er en kort opsummering af de mest interessante problemstillinger fundet i undersøgelsen af hjemmeplejens situation i kapitel 3. Disse blev fundet fra processeringen af de tre interviews samt udarbejdelsen af *rige billeder* jvf. *Objektorienteret Analyse & Design*[3].

- Anvendte planlægningssystemer har en forvirrende brugergrænseflade, hvilket hindrer effektiviteten af brug af systemet.
- Der er et skiftende antal borgere tilmeldt hjemmeplejen samt skift i omfanget af ydelser visse borgere har behov for. Dette resulterer ofte i ændringer af ruteplanerne, i form af "huller" eller borgere, der skal findes plads til i de eksisterende planer.
- Ekstern kommunikation det via nuværende system er ofte uklar.
- Intern kommunikation det via nuværende system blev omgået af nogle brugere.

Planlægningsredskaberne, som benyttes af både Aalborg Kommune og Attendo, er en del af et ældre system, hvilket resulterer i en brugergrænseflade, der ikke støtter nye planlæggere i deres arbejde grundet dårlig *affordance*[4] i form af uklarhed omkring, hvordan systemet anvendes.

Det meste af ruteplanlægningen foregår manuelt, hvilket også giver anledning til forbedringer. Effektiviseringer afhænger udelukkende af planlæggerens erfaring og overblik, og der revurderes meget sjældent på eksisterende ruter.

Vi fandt desuden et problem i kommunikationen mellem hjemmepleje-institutionerne og hospitalerne. Kommunikationen foregår via et beskedsystem kaldet *Advis*. Der er dog problemer i denne kommunikationsform, der leder til uklarhed.

Dertil er det interne kommunikationssystem ikke altid den foretrukne kommunikationsform, hvilket betyder at information ikke altid kan forventes at være samlet ét sted.

Vi valgte at arbejde videre med problemet ved effektivisering af planlægningen. Dette valg tog

vi, da vi følte at dette problem er gældende på tværs af afdelingerne, samt der er flere forskellige tilgange til at løse dette problem. Dette problem er formuleret i følgende problemformulering:

Problemformulering

Det er et problem at ruteplanlægningen udarbejdes, uden sikkerhed for at de planlagte ruter, er de mest effektive i forhold til tidsspild. Tilføjelser af borgere til ruterne sker ligeledes uden revurdering af den hidtidige plan, hvilket på længere sigt kan medfører til ineffektivt planlagte ruter i hjemmeplejen. Et IT-system der forbedrer hjemmeplejens muligheder for at planlægge arbejde, samt støtter de ansatte i udførelse af planlægningen, vil være med til at effektivisere hjemmeplejens planlægning og mindske tidspild.

Ovenstående problemformulering beskriver problemet vi løser i resten af rapporten. Med et system som løsning af problemformuleringen i tankerne, opstår følgende spørgsmål:

- Løser systemet problemformuleringen?
- Hvilke styrker og svagheder er der ved løsningen?

Vi vil besvare disse spørgsmål i konklusionen i kapitel 11 af rapporten.

4.1 Systemdefinition

Der er mange forskellige tilgange til at løse problemformuleringen. Vi har evalueret tre mulige tilgange til at løse dette problem. En mulighed er at udvikle et system, der støtter den nuværende arbejdsgang i hjemmeplejens planlægning ved at forbedre de nuværende værktøjer til planlægningsarbejdet. Vi kalder dette system det *manuelle system*.

Et andet fremgangsmåde er være at udvikle et system, der indeholder værktøjer til at foreslå gode planlægningsbeslutninger, såsom forslag til planlægning af ruter. Vi kalder dette for det *semi-automatiske* system.

Den sidste mulighed vi overvejede var at udvikle et *fuldautomatisk* system, der automatiserer alle aspekter af planlægningsarbejdet. Da fuldautomatisering er vanskeligt at realisere, og ville kræve omfattende ændringer i hjemmeplejens nuværende arbejdsgang, har vi valgt ikke at følge denne tilgang.

Dog mener vi, at der er større forbedringer at hente ved automatisering af dele af planlægningsarbejdet end ved blot at forbedre hjemmeplejens nuværende, manuelle arbejdsgang for planlægning. Af disse grunde har vi valgt at udvikle det semi-automatiske system, som er mere præcist defineret i følgende systemdefinition:

Systemdefinition

Et IT-system der kan bruges som planlægningsværktøj til støtte ved planlægning og opdatering af ruter. Systemet skal assistere planlæggerens administration af ruter ved at give forslag til fordeling af borgere og deres behov til og fra ruter. Dertil skal systemet assistere hjælpere på lignende vis ved akuttilfælde. Forslag til ruter bliver lavet ved hjælp af computere eller tablets, samt databaser og eksterne vejfindingssystemer, og baseres på informationer om ruter, hjælpere, borgere og transporttider.

Denne systemdefinition er lavet ud fra *BATOFF-kriterierne* jvf. *Objektorienteret Analyse & Design*[3]. Systemdefinitonens overholdelse af disse kriterier ses herunder:

Betingelser: Skal betjenes af en person med kendskab til de etiske problematikker og de gængse arbejdsgange indenfor hjemmeplejen.

Andvendelsesområde: Planlægger og hjælperes administration af ruter.

Teknologi: Computere, database, kortservices, netværksteknologi og tablets.

Objekter: Ruter, hjælpere, borgere og transporttider.

Funktioner: Planlægning af ruter. Forslag til tilføjelse og fjernelse af borgere på kørselsplaner. Forslag til uddelegering af borgere ved akuttilfælde.

Filosofi: Semiautomatisk værktøj som støtter planlægning og opdatering af ruter ved f.eks. at give komplette forslag til effektivisering af ruter.

Denne systemdefinition repræsenterer det system, vi vil udvikle som løsning på problemformuleringen. Følgende kapitler omhandler analyse, design og implementering af dette system.

5 Analyse

Dette kapitel omhandler analysen af systemet, herunder analysen af problem-og anvendelsesområdet. Til slut vil en beskrivelse og uddybelse af brugergrænsefladen blive fremlagt samt en kort beskrivelse af den tekniske platform systemet udvikles til.

5.1 Problemområde

Dette afsnit vil redegøre for vores resultater af analysen af problemområdet. Der vil indgå en beskrivelse af problemområdets klasser og hændelser. Desuden beskrives de forskellige klassers adfærd. Resultaterne af analysen af problemområdet er repræsenteret i henholdvist klassediagram, hændelsestabel og adfærdsdiagrammer over problemområdet. Klasser skrives med **fed skrift**, hændelser repræsenteres med *kursiv*, mens attributter skrives med `monospace`.

5.1.1 Klasser

Dette afsnit indeholder beskrivelser af klasserne fra problemområdet og kan ses i figur 5.1. Klasserne er fundet på baggrund af en brainstorm af problemområdet, og derefter udvælgelse af de klasser der opfylder kriterierne jvf. *Objektorienteret Analyse & Design*[3].

Borger

Klassen repræsenterer systemets hjælpsmodtagere, og er valgt da systemet skal have informationer om disse, for at holde styr på mængden af **borgere** i systemet, og for at kunne planlægge hvordan den individuelle borger skal modtage hjælp. Denne inkluderer information omkring `Navn`, `Adresse` `CPR` samt `Køn`.

Behov

Behov klassen afspejler en visiteret ydelse for én bestemt **borger**, og attributter herpå stammer fra en visitation. Vi modellerer at en **borger** får sit **behov** opfyldt, hvis de tilhørende **opgaver**

er planlagte på en **rute**.

Klassen indeholder en `Startdato` og en `Slutdato`. Disse udgør den periode hvori den tilknyttede **borger**, skal have opfyldt sit **behov**. Nogle **behov** er ikke visiteret til at skulle udføres på et præcist tidspunkt på døgnet, men i stedet over et tidsrum. Dette tidsrum er repræsenteret ved attributten `Tidsrum`. `Varighed` repræsenterer det tidforbrug et **behov** er visiteret til.

Attributten `Beskrivelse` beskriver hvad det givne **behov** indebærer. `Udførelsestid` er en speciel sammensat attribut, der beskriver hvor ofte et **behov** skal opfyldes. Specifikt kan attributten opdeles i to underdele. Den ene beskriver hvor lang tid der mindst skal gå, fra et **behov** er blevet opfyldt, til det igen skal opfyldes. Den anden del beskriver hvor lang tid der maksimalt må gå før **behovet** skal være opfyldt igen.

Et problem med denne repræsentation af `Udførelsestid` er, at to **borgere** med de næsten samme **behov** i teorien kan have et vidt forskelligt antal **opgaver** fordelt over den samme antal uger. Vi har valgt ikke at tage hånd om dette, og overlader det til fremtidig udvikling.

Opgave

Denne klasse er valgt som en abstraktion over et **behovs** udførelse, for at kunne registrere de konkrete **opgaver** der skal udføres for at opfylde et **behov**. En **opgave** har attributten `Note`. Denne benyttes hvis der skal tillægges ekstra information til den specifikke **opgave**. En **opgave** oprettes af det **behov** den bliver specificeret ud fra.

Hjælper

Systemet indeholder **hjælpere**, hvilket er de ansatte som udfører **opgaver**. Denne klasse er valgt fordi **ruter** skal kunne tillægges en **hjælper** og en **hjælper** skal kunne se hvilken **rute** han eller hun er påsat. Klassen skal derfor bruges til at indfange informationer omkring hjælpsudførende personale i systemet. Dette inkluderer `Navn`, `Køn`, `Kvalifikationer` og `Kørekort`. Systemet skal vide noget om en **hjælpers** `Køn`, da nogle hjælpsmodtagere kan have behov for at deres ydelser bliver udført af samme køn. Ligeledes skal der vides noget om `Kørekørt`, da nogle **ruter** i hjemmeplejen udføres kørende i bil. `Kvalifikationer` skal systemet vide noget om, da ikke alle typer af **hjælpere** må f.eks. give medicin.

Præference

Denne klasse er valgt fordi systemet har brug for en indikator til at udtrykke forholdet mellem en **borger** og en **hjælper**, da nogle **borgere** har brug for at der bliver taget hensyn til dette. F.eks. har demente **borgere** brug for, at deres **opgaver** bliver løst af en **hjælper** de kender godt. `Grad` er en attribut på **præference**, der skal illustrere niveauet for en **borgers** tilknytning til en **hjælper**.

Rute

Rute er en klasse, som består af en samling **tidsperioder**. Denne klasse er valgt, fordi systemet skal holde styr på rækkefølgen af udførelsen **arbejde** der skal udføres og hvilken **hjælper** der skal udføre dem.

Tidsperiode

Tidsperiode er valgt fordi systemet skal kunne definere bestemte tidsrum for en **rute**. **Tidsperiode** er abstrakt og er en generalisering af fire andre klasser, der alle har til fælles, at de repræsenterer et tidsrum (fra klokkeslæt til et andet klokkeslæt) på en **rute**. Klassen indeholder attributterne *Starttidspunkt* og `Sluttidspunkt`. `Starttidspunkt` og `Sluttidspunkt` angiver henholdsvis starttidspunktet og sluttidspunktet for en **tidsperiode**.

Arbejde

Denne klasse er valgt da systemet skal kunne repræsentere at der bliver udført en **opgave**, i en **tidsperiode**. **Arbejde** er en af de fire specialiseringer af **tidsperiode**.

Pause

Denne klasse er valgt da systemet skal kunne repræsentere at **hjælpere** skal have en pause ude på deres **rute**. **Pause** er en af de fire specialiseringer af **tidsperiode**.

Ledig

Denne klasse er valgt da systemet skal kunne repræsentere et tomt tidsrum på **ruten**. **Ledig** er en af de fire specialiseringer af **tidsperiode**.

Transport

Denne klasse illustrerer en **tidsperiode**, der er dedikeret til transport mellem to adresser på en **rute**. Dette er nødvendigt for at kunne planlægge **ruterne**. **Transport** er en af de fire specialiseringer af **tidsperiode**.

Forbindelsestid

Denne klasse holder styr på transporttiden mellem to **borgere**. **Forbindelsestid** bruges af **transport**, når der ønskes at definere den tidslige afstand imellem **borgere**. Begrundelsen for denne klasse ligger i stor grad i en ansvarsdeling, for at begrænse **transports** ansvar til repræsentation overfor **rute**. Bemærk derfor, at **forbindelsestid** repræsenterer en tidslig varighed, hvorimod **transport** repræsenterer et tidsrum.

5.1.2 Struktur

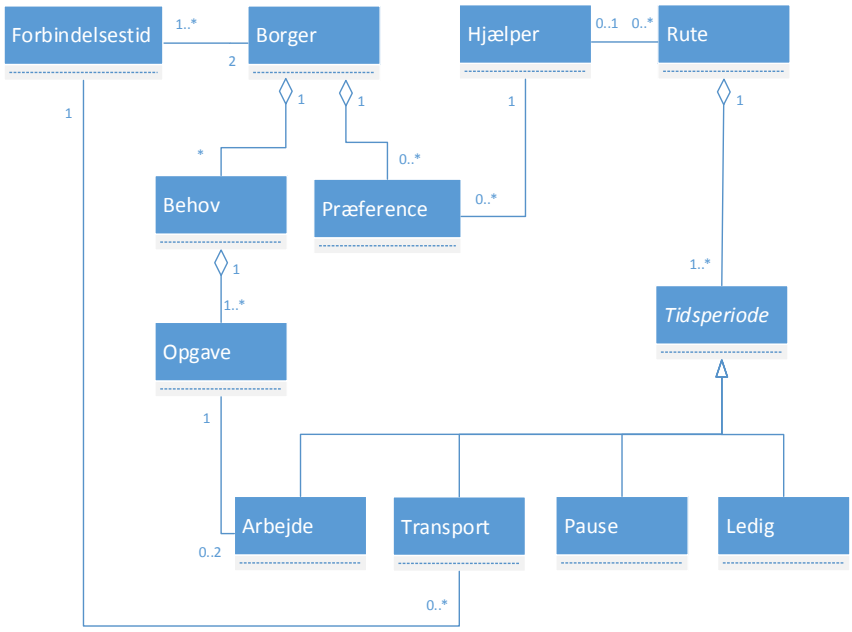
Herunder beskrives interessante strukturer og multipliciteter, som findes i klassediagrammet fra figur 5.1.

Objekter af **hjælper** er tilknyttet til flere **rute** objekter, via en associeringsstruktur da en **hjælpere** skal udføre en eller flere **ruter**. Multipliciteten er nul til én **hjælper** for flere **ruter**, da systemet vil kunne have en **rute** uden en påsat **hjælper**, og at **ruten** kun tages af én **hjælper**.

Tidsperiode er en generaliseringstruktur over specialiseringerne: **arbejde**, **transport**, **pause** og **ledig**. Denne struktur er valgt for at at få et overblik over de former for tidsforbrug der skal planlægges på en **rute**. En **rute** indeholder dermed flere **tidsperioder**. En **rute** vil altid have mindst én **tidsperiode**, da der selv ved ingen **arbejde** vil være én stor **ledig**.

Borger er dekomponeret til **behov** da en **borger** kan have mange **behov**. En **borger** har et antal **behov** alt efter hvad han er blevet visiteret til. Ved oprettelse vil en **borger** ikke have nogle **behov**, men vil få tildelt et kort efter, da der ikke registreres **borgere** uden at der eksisterer et **behov**. Efter det første **behov** er tildelt, vil en **borger** altid have mindst ét.

Dertil er **Behov** dekomponeret til flere objekter af **opgave**, og er i et relateringsmønster med **arbejde**. Dette er valgt for ved en modellering af, at der, for hvert et **behov**, findes et antal **opgaver**. Planlægningen opfylder **behovet** hvis alle tilknyttede **opgaver** er placeret på en **rute**. Da der netop er flere **Opgave** objekter med samme beskrivelse, er **opgave** og **behov** et



Figur 5.1: Klassediagram over problemområdet

genstand-beskrivelsesmønster, hvor **opgaverne** er genstande for ét **behov** der skal opfyldes.

For at modellere at der kun er én **opgave** for hver **arbejde** påsat en **rute**, og at en **opgave** kan være forbundet til maksimalt to forskellige **arbejde** objekter (hvis den skal løses af to **hjælpere**), er multipliciteterne sat efter dette på figur 5.1.

Relationen mellem en **borger** og flere **hjælpere**, modelleres i et relateringsmønster mellem disse klasser. Dette er valgt for at modellere det krav at nogle **borgere** har brug for, at der ikke sker for meget udskift i **hjælperen** der besøger dem. Én **borger** kan have mange **præferencer**, da en **borger** kan besøges af flere **hjælpere**. En **præference** kender til én **hjælper**, men **hjælperen** kender ikke til **præferencer**. På denne måde modelleres, at en **Borger** har 0 til mange **præferencer** for 0 til mange **hjælpere**, men at flere **borgere** kan have **præferencer** for de samme **hjælpere**.

To **borgere** er associeret med flere **forbindelsestider**. Dette er for at modellere tiden det tager

af rejse imellem to **borgere**. Der er et antal **forbindelsestider** alt efter hvor mange **borgere** der findes, da der skal kunne findes en **forbindelsestid** mellem alle **borgere**. Hvis én **rute** har **arbejde** med **opgaver** tilknyttet forskellige **borgere**, skal der mellem dem være en **transport**. Denne **transport** skal derfor have en association til den **forbindelsestid** der matcher de pågældende **borgere**. Da disse **arbejde** kan laves i samme rækkefølge på en senere **rute**, har **forbindelsestiden** mulighed for at associeres til flere **transport**.

5.1.3 Hændelser

Dette afsnit beskriver hændelserne fra problemområdet, herunder hvorfor systemet skal tage højde for dem. Hændelserne er fundet ud fra en brainstorm og systematisk udvælgelse af de hændelser der opfylder kriterierne jvf. *Objektorienteret Analyse og Design*[3]. Hændelserne er opstillet i en hændelsetabel i figur 5.2. I hændelsestabellen ses de klasser vi har valgt i øverste række, og til venstre kan hændelserne ses. Et '+' beskriver at en hændelse kan forekomme en gang på en klasse (sekvens), mens en '*' beskriver at en hændelse kan forekomme flere gange på en klasse (iteration) jvf. *Objektorienteret Analyse & Design*[3].

Fravær meldt

Forekommer når en **borger** midlertidigt ikke længere skal modtage hjemmehjælp, eksempelvis ved hospitalsindlæggelse. *Fravær meldt* skal registreres af systemet, da det skal kende til **tidsperioder** hvor der ikke skal løses **opgaver** længere.

Adresse tilføjet

Denne hændelse forekommer når en **borger** ændrer `adresse`. Hændelsen skal administreres af systemet, da den skaber nye **forbindelsestid** objekter ud fra den **borger** der ændrer `adresse`.

Behov godkendt

Sker når en **borger** får visiteret et nyt **behov**. Hændelsen skaber et nyt **behov** objekt, som tildeles den **borger** som **behovet** er blevet godkendt til.

Behov frakendt

Sker når en **borger** vurderes til ikke længere, at have brug for at modtage en bestemt ydelse. Denne hændelse skal administreres af systemet, da hændelsen skal fjerne alle overskydende **opgaver** af det bestemte **behov**, fra **ruter**.

Opgave oprettet

Denne hændelse skaber **Opgave** objekter, ud fra deres respektive **behov**. Systemet skal overvåge hændelsen, da det er nødvendigt at vide hvilke **opgaver** der skal planlægges.

Tilføjet til rute

Forekommer når en **opgave** bliver tilføjet til en **rute**. Hændelsen skal registreres for at holde styr på om udførelsen af **opgaver** er blevet planlagt.

Fjernet fra rute

Denne hændelse sker når en **opgave** fjernes fra en **rute**. Dette skal overvåges, for at holde styr på hvor der er **ledigt** på en **rute**, samt om en **opgave** er planlagt som **arbejde** på en **rute**.

Planlægning påbegyndt

Forekommer når en ny **ruter** oprettes. Systemet skal overvåge denne hændelse, for at holde styr på hvilke **ruter** der er blevet oprettet.

Rute planlagt

Denne hændelse sker når en planlægger vælger at acceptere en forslået **rute**. Hverved registreres at **ruten** kommer i tilstanden planlagt.

Rute nedlagt

En **rute** kan i en hvilken som helst tilstand blive nedlagt. Dette vil medføre at objektet dør. Hændelsen skal registreres af systemet, da alle **opgaver**, der var associeret med **arbejde** objekter fra den pågældende **rute**, nu igen skal medtages i planlægningen.

Rute initieret

Når en planlagt **rutes** udførelsesdato nåes, bliver denne hændelse udført. Dette skal registreres af systemet, for at kontrollere at planlagte **ruter** kommer i tilstanden aktuel.

Rute udført

Når alle **opgaver** på en **rute** er udført, dør **rute** objektet ved hændelsen *rute udført*. Denne hændelse skal registreres af systemet, for at kunne fastslå at **rutens arbejde** blev udført, og at **præferencer** skal justeres.

Visse klasser havde brug for en hændelse, der markerer hvornår et **arbejde** udføres. Da vi ikke havde mulighed for at overvåge en sådan hændelse, blev det besluttet at *rute udført* skulle markerer dette i stedet. Dette har forårsaget at klasser der har fælles hændelser, uden at have associationer. Vi føler dog at dette stadig en bedre repræsentation, da det fjerner unødig kompleksitet, der ikke havde bidraget til en øget forståelse.

Ansat

Når en ny **hjælper** skal tilføjes til hjemmeplejen, sker hændelsen *ansat*. Systemet skal registrere denne hændelse fordi nye **hjælpere** skal kunne sættes på en **rute**.

Fratrådt

En **hjælper** skal kunne fjernes fra systemet, hvis denne **hjælper** ikke længere skal kunne sættes på **ruter**.

Hjælper påsat rute

Hændelsen skal registreres af systemet, for at overvåge hvilken **hjælper** der er sat på en **rute**.

Hjælper fjernet fra rute

Når en **rute** er i tilstanden foreslået, kan man sætte en **hjælper** på en **rute**. Derfor er det nødvendigt at systemet overvåger hændelsen *hjælper fjernet fra rute*, så system ved om en **rute** er dækket af en **hjælper**.

Aktuel arbejde fjernet

Denne hændelse fjerner et **arbejde** fra en **rute**. Når en **rute** er i tilstanden aktuel rute, kan det igen ske at der skal fjernes og tilføjes **tidsperioder**. Dette afspejler sig i form af hændelserne *aktuel arbejde fjernet* og *aktuelt arbejde tilføjet*. Disse hændelser skal overvåges af systemet, da alle ændringer af **ruter** skal registreres.

Aktuel arbejde tilføjet

Dette er hændelsen der tilføjer **arbejde** til en **rute** der er i tilstanden aktuel.

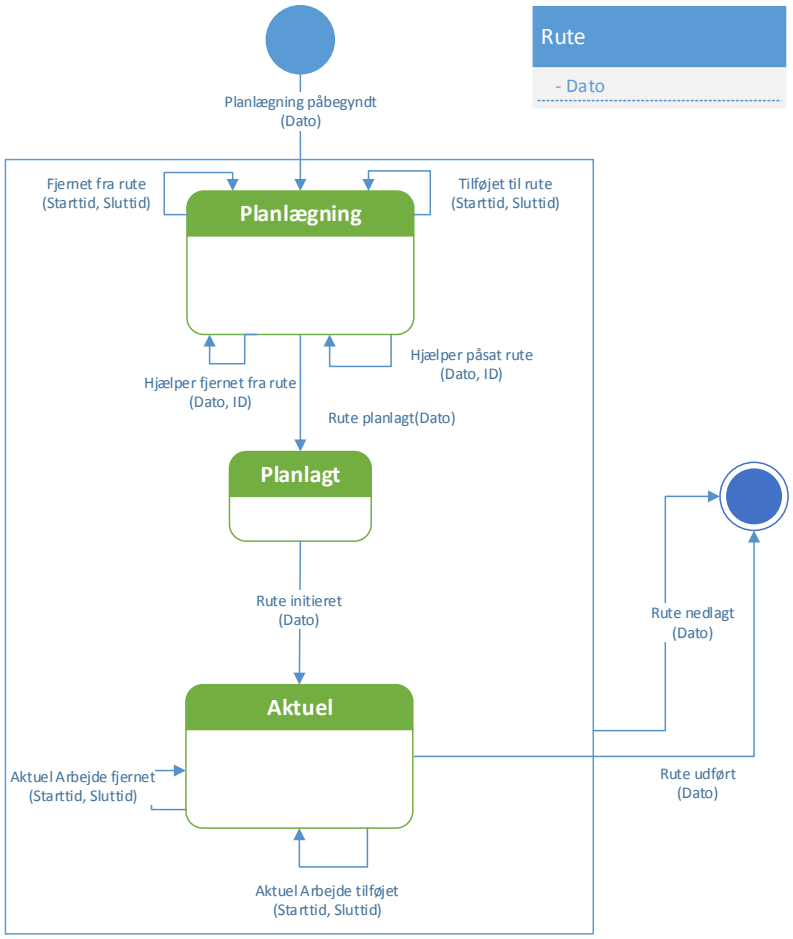
Hændelser	Borger	Præference	Behov	Opgave	Forbindelsestid	Hjælper	Rute	Tidsperiode	Arbejde	Ledig	Pause	Transport
Borger registreret	+				+							
Behov godkendt	*		+									
Behov frakendt	*		+		+							
Adresse tilføjet	*				*							+
Fravær meldt	*			+								
Opgave oprettet			*	+								
Planlægning påbegyndt							+					
Rute planlagt				*			+					
Rute nedlagt				*			+	+	+	+	+	+
Rute initieret							+					
Rute udført		*		+		*	+	+	+	+	+	+
Tilføjet til rute				*			*	+	+	+	+	+
Fjernet fra rute				*			*	+	+	+	+	+
Ansat						+						
Hjælper påsat rute						*	*					
Hjælper fjernet fra rute						*	*					
Fratrådt		+				+						
Aktuelt Arbejde tilføjet							*					
Aktuelt Arbejde fjernet							*					

Figur 5.2: Hændelsestabel over problemområdet

5.1.4 Adfærd

Ud fra hændelserne i afsnit 5.1.3 samt klassediagrammet over problemområdet har vi udarbejdet tilstandsdiagrammer for alle klasser i klassediagrammet. I dette afsnit beskrives de centrale tilstandsdiagrammer.

Rute



Figur 5.3: Tilstandsdiagram over adfærden for **ruter**

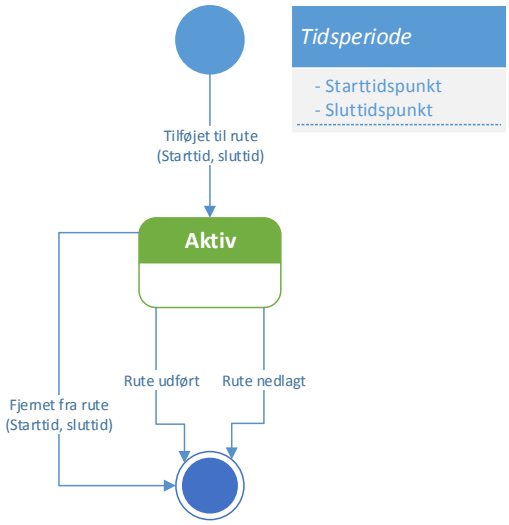
Tilstandsdiagrammet for **rute** kan ses i figur 5.3. **Rute** fødes ved *planlægning påbegyndt*, og dør ved *rute nedlagt*, eller *rute udført* alt efter hvorvidt **ruten** er blevet erstattet eller gennemført.

I tilstanden 'planlægning' er det muligt at mængden af **tidsperioder** ændres, ved *tilføjet til rute* eller *fjernet fra rute*. Hvilken **hjælper** der er påsat **rute** ændres også i denne tilstand, ved hændelserne *hjælper påsat rute* og *hjælper fjernet fra rute*. Når *rute planlagt* sker skifter tilstanden til 'planlagt', og *rute nedlagt* sker for en eventuel tidligere udgave af **ruten**.

I tilstanden 'planlagt' er det ikke muligt at manipulere med **rutens** indhold. Hvis man ønsker at ændre den, skal den erstattes med en ny. Dette er valgt for at simplificere forståelsen af planlægningsprocessen. Fra denne tilstand vil **ruten** skifte til tilstanden 'aktuel', når hændelsen *rute initieret* sker.

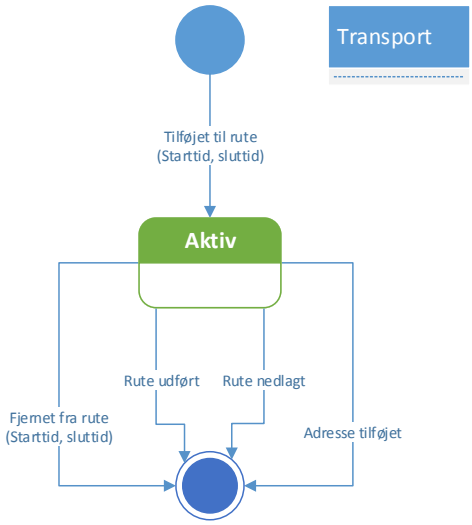
I tilstanden 'aktuel' er det igen muligt at **ruten** manipuleres. Dette sker ved *aktuelt arbejde tilføjet* eller *aktuelt arbejde fjernet*. Dette er for at vise at systemet skal tillade akut planlægning fra plejepersonalets side imens **ruten** udføres.

Tidsperiode



Figur 5.4: Tilstandsdiagram over adfærden for **tidsperiode**

En **tidsperiode** fødes, når den bliver sat på en **rute**. Den er i tilstanden 'aktiv' indtil den dør, hvilket kan ske på tre måder som vist i figur 5.4.

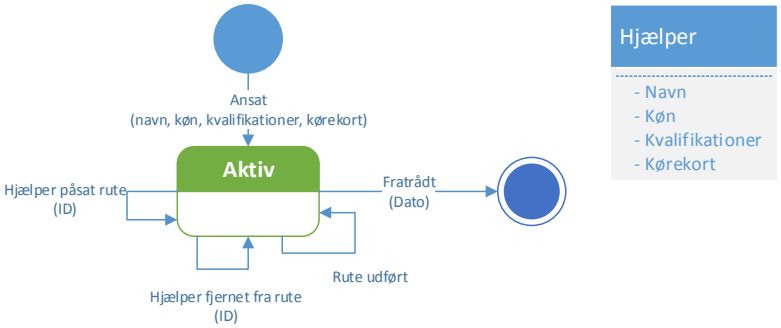


Figur 5.5: Tilstandsdiagram over adfærden for **transport**

Transport

Transport er en specialisering af **tidsperiode** med den ene tilføjelse, at den også dør, når den tilhørende **forbindelsestid** ændres ved *adresse tilføjet* (se figur 5.5).

Hjælper

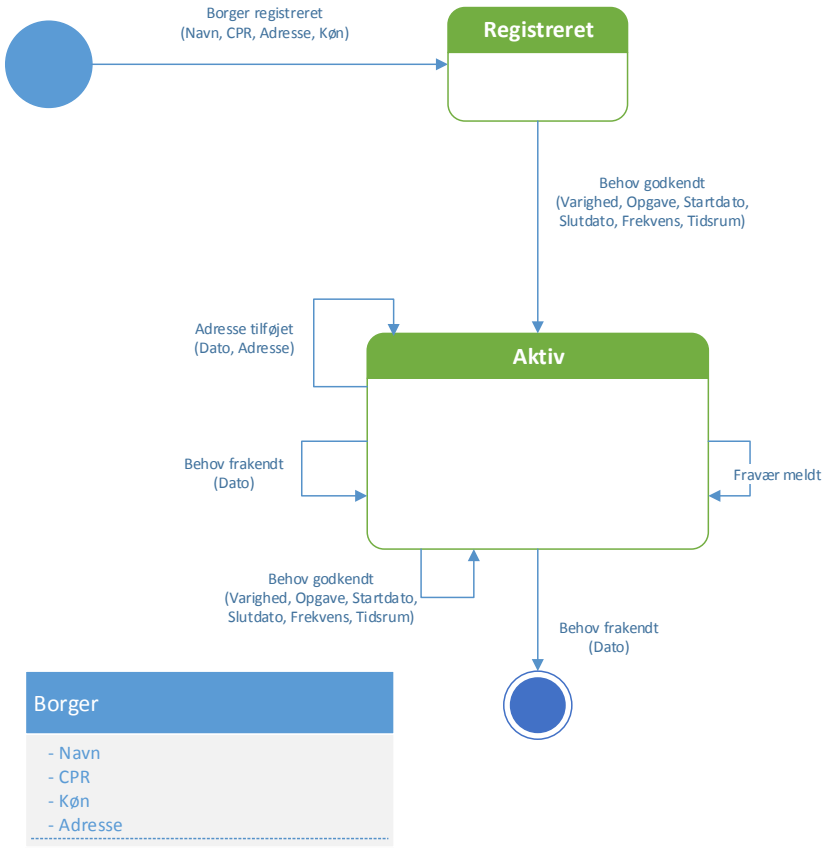


Figur 5.6: Tilstandsdiagram over adfærden for **Hjælper**

Hjælper kan indgå i hændelser som systemet skal håndtere i perioden imellem ansættelse og fratrædelse, hvilket modelleres med hændelserne *Ansat* og *Fratrådt*. Som vist på figur 5.6 så skal systemet registrere hvilke **ruter** som **hjælpere** er sat på. Dette er valgt, fordi det er et brugerkrav at der vides hvilke **hjælpere** er på hvilke **ruter**.

Borger

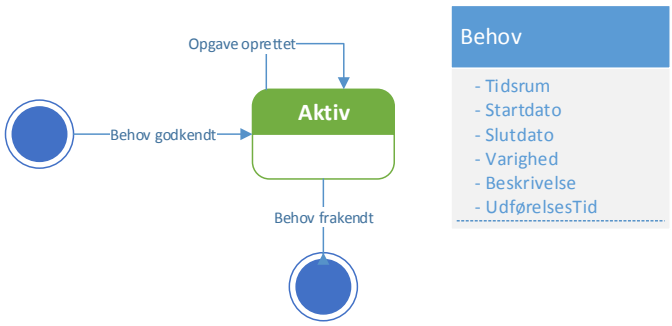
En **borger** fødes ind i tilstanden 'registreret', hvor den bliver indtil det først **behov** godkendes. Det forventes at **borgeren** ikke forbliver i denne tilstand længe. En **borger** dør, når det sidste **behov** frakendes. Der findes et specialtilfælde, hvor den virkelige person skal modtage hjemmehjælp igen efter en periode helt uden tilknytning til sundhedssektoren. Dettes udelades da det er unødig kompleksitet, og ikke bryder imod det nuværende tilstandsdiagram, set i figur 5.7.



Figur 5.7: Tilstandsdiagram over adfærden for **borger**

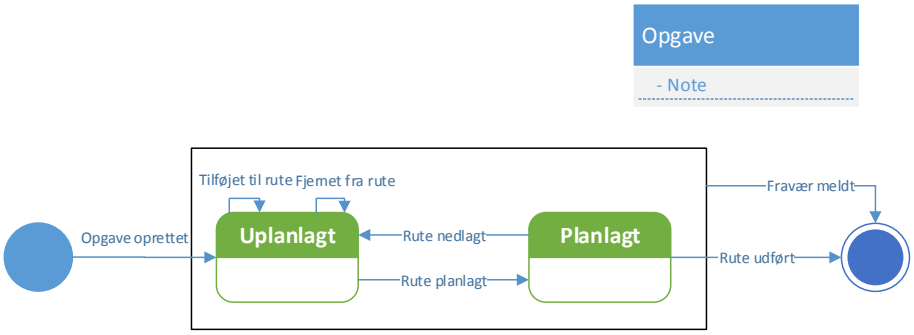
Behov

Som vist i figur 5.8 bliver **behov** født og dør ved henholdvis *behov godkendt* og *behov frakendt*. Det eneste der kan ske i mellemtiden er, at *opgave oprettet* hændelsen tilføjer **opgaver** til **behovet**.



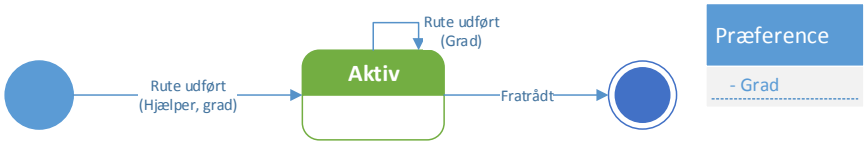
Figur 5.8: Tilstandsdiagram over adfærden for **behov**

Opgave



Figur 5.9: Tilstandsdiagram over adfærden for **opgave**

Når en **opgave** fødes er den i tilstanden 'uplanlagt' som vist i figur 5.9. Den forbliver i denne tilstand uanset eventuelle associationer til **arbejde** som ændres ved *tilføjet til rute* og *fjernet fra rute*. Først når *rute planlagt* sker for den **opgave** der er associeret til, skifter tilstanden. Det bør specificeres at *Fravær meldt* ikke fjerner et eventuelt associeret **arbejde** fra en **rute**.



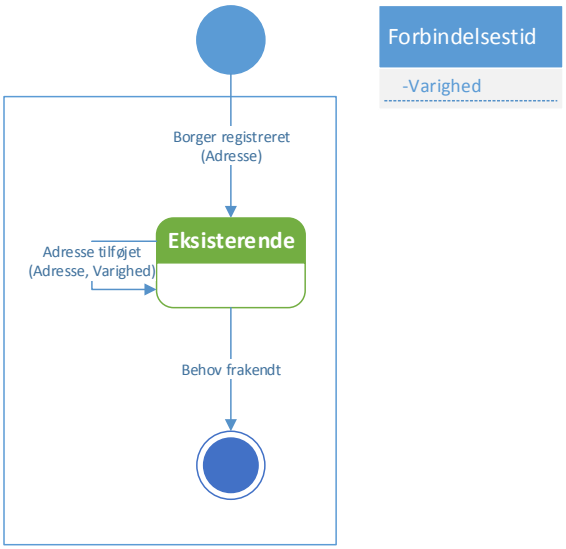
Figur 5.10: Tilstandsdiagram over adfærden for **præference**

Præference

Præference, som er illustreret i figur 5.10, fødes når *rute udført* sker for en **rute**, hvor en **hjælper** løser en **opgave**, for en **borger**, som den samme **hjælper** aldrig før har løst en **opgave** for. Efterfølgende udførelse af **ruter** med **arbejde** der løser **borgerens behovs opgaver** vil øge **præferencens** `grad`. Det forventes at `graden` skal forfalde, men da dette ikke eksisterer i problemområdet har vi ikke modelleret det her.

Forbindelsestid

Når en *borger registreres*, bliver der oprettet **forbindelsestider** (se figur 5.11) mellem **borgeren** og alle nuværende **borgere**. Når **borgeren** *ændrer adresse*, opdateres `varigheden` af alle dens **forbindelsestider**. **Forbindelsestider** dør først, når en tilknyttet **borger** sidste *behov fraken-des*.



Figur 5.11: Tilstandsdiagram over adfærden for **forbindelsestid**

5.2 Anvendelsesområde

Formålet med dette kapitel er at identificere aktører i anvendelsesområdet og deres brugsmønstre. Disse brugsmønstre bruges herefter til at beskrive basale funktioner og krav til systemet. Derudover vil der redegøres for brugergrænsefladen og den tekniske platform.

5.2.1 Brug

Brugsmønstre bruges til at fastlægge hvorledes aktørerene interagerer med systemet. De er abstraktioner over egentlige arbejdsopgaver som aktørerne udfører.

Aktører

Vi kan ud fra vores systemdefinition samt vores interviews med personerne i anvendelsesområdet fastslå, at der er to aktører i vores overordnede system. Det drejer sig om Planlægger og Hjælper. Disse aktøreres formål i systemet, samt en kort karakteristik af deres roller er begge kort beskrevet i tabel 5.1 og tabel 5.2.

Planlægger
Formål: En person som er ansvarlig for planlægningen af ruter og besøg.
Karakteristik: Systemet er associeret med 1-2, planlæggere per afdeling. Alle med forskellige erfaringer og behov.
Eksempler: Planlægger A har mange års erfaring med systemet, og bruger nødigt musen til at interagere med. A ønsker hellere at der er genvejstaster i systemet som kan bruges i så stort omfang at brug af musen bliver overflødig.
Planlægger B har knap så mange års erfaring indenfor systemet, men kommer derimod med en rimelig forståelse for IT, og har derved nemt ved at sætte sig ind i systemet. B læner sig mere op af, at der er en intuitiv brugergrænseflade, hvor kun det nødvendige er præsenteret, og hvor ikoner og tekst giver mening.

Tabel 5.1: Aktørspecifikation for "Planlægger"

Hjælper
Formål: En person der modtager planlagte ruter, og har ansvar for at udføre pågældende ruter.
Karakteristik: Systemets brugere omfatter mange hjælpere med forskellige erfaringer.

Tabel 5.2: Aktørspecifikation for "Hjælper"

Arbejdsopgaver

Planlæggers arbejdsopgaver består hovedsageligt af oprettelse og nedlæggelse af **ruter**. De håndterer også ændring af daglige **ruter** i forhold til indberetninger fra **borgere**, og sygemeldinger fra **hjælpere**. De har derudover også mulighed for at tilføje `noter` til en given **opgave**, hvis der er ting som hjælperen skal tage højde for.

Hjælperens arbejdsopgaver består af at modtage **ruter** og indberette eventuelle `noter` angående deres daglige besøg ved **borgerne**. Dette kan være alt fra at indberette hvis en **opgave** ikke kan udføres indenfor den visiterede tid, til hvis der har været eventuelle problematikker ved et besøg. Derudover har hjælperen også til opgave at indberette forsinkelser, hvis sådanne opstår.

Brugsmønstre

Vi har i dette afsnit udarbejdet en række brugsmønstre, baseret på de netop beskrevne arbejdsopgaver. Vi har samlet disse brugsmønstre i tabel 5.3 og derefter beskrevet dem hver for sig.

Brugsmønster \ Aktører	**Planlægger**	**Hjælper**
Se notifikationer	x	
Se data for tidsperiode	x	
Se data for opgave	x	
Godkend foreslag	x	
Afvis foreslag	x	
Fjern hjælper fra rute	x	
Se rute	x	
Etablering ny rute	x	
Slet rute	x	
Tilføj arbejde til rute	x	
Fjern arbejde fra rute	x	
Ændring af tidsrum for arbejde	x	
Tildel hjælper til rute	x	
Lad systemet foreslå hvordan arbejdsopgaver tilføjes en rute	x	
Optimering af flere ruter	x	
Information om ændring i en aktuel rute til hjælpere		x
Indberetning af forsinkelse		x
Se aktuel rute		x

Tabel 5.3: Aktørtabel

Se notifikationer

Foretages af planlægger, når denne ønsker at kende til relevante informationer der kommer fra et eksternt system.

1. Planlæggeren navigerer til notifikationsfanen.
2. Planlæggeren kan nu se nyeste notifikationer

Grunden til at planlægger skal have muligheden for at kunne se notifikationer der kommer fra et eksternt system, er for at se hvilke nye arbejdsopgaver der er opstået, udover dem planlægger

får over telefon fra **hjælpere** og **borgere** selv.

Etablering af ny rute

Igangsættes af planlægger, når planlæggeren vurderer, at der skal være en ny **rute**.

1. Planlæggeren navigerer til planlægningsfanen.
2. Herfra kan der klikkes på "Ny rute", hvorved et nyt vindue vil fremkomme.
3. Der indtastes navn på den nye **rute**, samt dato for **rutens** udførelse.
4. Planlæggeren bekræfter den nye **rute** ved klik på 'Opret rute'.

Planlægger skal have muligheden for at kunne lave nye **ruter**, da vedkommende har brug for dette når der registreres flere **opgaver** end hvad der er plads til på de nuværende **ruter**. I vores system ønsker vi at planlægger skal planlægge **ruterne** for hver dag, hvilket er derfor at planlægger skal navigere til planlægningsfanen for at kunne danne en ny **rute**.

Slet rute

Igangsættes af planlægger, når planlæggeren vurderer, at der skal slettes en **rute**.

1. Planlæggeren navigerer til planlægningsfanen.
2. Planlægger vælger **ruten** der skal fjernes, under en bestemt dato.
3. Herfra trykkes: "slet rute", som fjerner den valgte **rute**.

Planlægger skal have muligheden for at kunne slette **ruter**, da det er varierende hvor mange **borgere** er tilmeldt hjemmeplejen, og det derfor kan hænde at der kommer overflødige **ruter**.

Tilføj arbejde til rute

Tilføjelse af **arbejde** til en **rute** udføres af en planlægger.

1. Planlæggeren navigerer til planlægningsfanen.
2. Planlægger kan i planlægningsfanen se alle **opgave**, der endnu ikke er sat på en **rute**. **Opgaver** er fremvist i form af en liste.

3. Planlægger vælger **ruten** og dato **opgaven** skal placeres i.
4. Planlægger trækker **opgaven** til den pågældende **rute** og på den ønskede **ledig tidsperiode**.
5. Planlæggeren gemmer **ruten**.

Planlægger skal have muligheden for at kunne tilføje **opgaver** til **ruten** via **arbejde**, da det repræsentere **hjælperens** besøg ude hos **borgeren**, og som er det der skal planlægges. Forklaringen af træk-og-slip sker i afsnit 5.3.

Fjern arbejde fra rute

Fjernelse af **arbejde** fra en **rute** udføres af planlæggeren. Et **arbejde** fjernes enten, når **borgeren** afmeldes eller ved omlægning af eksisterende **ruter**, hvor et **arbejde** skal overføres fra en **rute** til en anden.

1. Planlæggeren navigerer til planlægningsfanen.
2. Planlæggeren finder dato og **ruten** hvor **arbejdet** skal fjernet
3. Planlægger vælger **arbejdet** der skal fjernes.
4. Planlæggeren fjerner **arbejdet** ved at trække det ud i en liste med løse **opgaver**.
5. Planlæggeren gemmer **ruten**.

Grunden til at planlægger skal have muligheden for at kunne fjerne **arbejde** fra **ruten** ved at placere det på en liste med løse **opgaver**, er fordi at disse **opgaver** nu ikke længere er med til at opfylde en **borgers behov**, og skal derfor tilses på et tidspunkt.

Ændring af tidsrum for arbejde

Ændring af tidsrum for **arbejde** foretages af planlæggeren, som er når der opstår **behov** for at ændre i hvilket tidsrum **arbejdet** skal placeres i på **ruten**.

1. Planlæggeren navigerer til planlægningsfanen.
2. Planlæggeren finder dato og **ruten** hvor tidsrummet for et **arbejde** skal ændres.
3. Et **arbejde** trækkes hen til den **tidsperiode** den ønskes placeret ved.

4. Et nyt tidspunkt for for **arbejdets** udførelse vælges.
5. Planlæggeren gemmer **ruten**.

Det at planlægger skal have muligheden for at kunne flytte **arbejde** på **ruten**, via træk-og-slip argumenteres der for i afsnit 5.3.

Tildel hjælper til rute

Tildeling foretages af planlæggeren. Tildeling sker når en **rute** mangler en **hjælper**.

1. Planlæggeren navigerer til planlægningsfanen.
2. Planlæggeren finder dato og **ruten** hvor **hjælperen** skal på.
3. Pågældende **rute** højreklikkes.
4. 'Vælg hjælper' klikkes, og et vindue præsenteres for planlæggeren med en liste af **hjælpere**.
5. Planlægger vælger ønskede **hjælper** og klikker på: "sæt på rute".
6. Planlæggeren gemmer **ruten**.

Grunden til at dette brugsmønster er valgt, er fordi at det er planlæggeren der bestemmer hvilke **hjælpere** der placeres på **ruter**. Grunden til at planlæggeren tilgår et nyt vindue, beskrives i afsnit 5.3.

Fjern hjælper fra rute

Handlingen foretages af planlæggeren. Fjernelsen sker når en **rute** skal have en anden **hjælper**.

1. Planlæggeren navigerer til planlægningsfanen.
2. Planlæggeren finder dato og **ruten** hvor **hjælperen** skal fjernes fra.
3. Pågældende **rute** højreklikkes.
4. 'Vælg hjælper' klikkes, og et vindue præsenteres for planlæggeren med en liste af **hjælpere**.
5. Planlægger vælger ønskede **hjælper** og klikker på: "fjern hjælper".

6. Planlæggeren gemmer **ruten**.

Grunden til at dette brugsmønster er valgt er fordi at det er planlæggeren, der bestemmer hvilke hjælpere der placeres på **ruter**.

Lad systemet foreslå hvordan arbejdsopgaver tilføjes en rute

At lade systemet foreslå hvordan **opgaver** tilføjes en **rute**, initieres af planlæggeren, hvis planlægger vurdere at systemet er bedre i stand til at placere **pgaverne** på **ruten**, end planlæggeren selv.

1. Planlæggeren navigerer til planlægningsfanen.
2. Planlæggeren finder dato og **ruten** hvor **opgaverne** skal på.
3. ctrl-tasten holdes nede og der klikke på de pågældende **opgaver**, der skal på **ruten**.
4. "Foreslå placering"trykkes, hvorefter systemet selv placerer de markerede **opgaver** på **ruten**.
5. Planlæggeren gemmer **ruten**.

Grunden til at man skal kunne lade systemet tilføje **opgaver** i planlægningsfanen, er for at støtte planlæggeren i planlægningsprocessen, i stedet for at planlæggeren skal ind på en korttjeneste for at indsætte en **arbejde**. Ctrl-tasten er valgt, da mange systemer gør brug af den, til at vælge flere enheder.

Optimering af flere ruter

Hvis planlæggeren vurderer, at der kan spares kørselstid ved at omrokerer **arbejde** på flere **ruter**.

1. Planlæggeren navigerer til optimeringsfanen.
2. Fjern afkrysninger ud fra de ændringer som ikke må realiseres.
3. "Udfør optimering"klikkes, for at realisere de ændringer man har tilladt.

Grunden til at optimeringen, hvor flere **ruter** revurderes får sin helt egen tab, er for at give plads til at give en overskuelighed over disse optimeringer i brugergrænsefladen. Se afsnit 5.3.

Der kan være optimeringer som planlægger ikke ønsker realiseret, som f.eks. hvis systemet vil flytte **arbejde**, som planlægger ved, skal ske i det nuværende tidsrum. Derfor er der givet mulighed for, at planlægger kan påvirke dette under optimeringen.

Information om ændring i en aktuel rute til hjælpere

Når der foretages en ændring af en **rute**, informeres en **hjælper** herom, via en besked på sin tablet.

1. Hjælper navigerer til notifikationsfanen for at se listen af ændringer.
2. Hjælper trykker på den nyeste ændring for at se ændringens detaljer.

Hjælperen skal have information om ændring i sin **rute** når de er ude på den. Der skal navigeres hen på notifikationssiden, da det er startssiden når tabletten åbnes, så hjælperen kan se ændringen som det første.

Indberetning af forsinkelse

Indberetningen af en forsinkelse sker via hjælpers tablet, når **hjælpere** er forsinket på en **rute**.

1. Hjælper navigerer til notifikationsfanen for at se listen af ændringer.
2. Hjælper vælger feltet over ”indberet forsinkelse” knappen.
3. Hjælper indtaster hvor langt **hjælper** er bagud med sin **rute** (I form af minutter).
4. Knappen ”indberet forsinkelse” trykkes.

Hjælperen skal kunne indberette forsinkelse, for at systemet kan enten kan give planlægger notifikation om at der er en **borger** der mangler en **opgave** udført, eller kunne signalere en anden hjælper, og forespørge om de kan varetage **opgaven**.

Se aktuel rute

En hjælper har mulighed for at se sin daglige **rute**.

1. Hjælper navigere til "se rute"fanen.

2. Nu fremvises **hjælperes rute** for den pågældende dag.

Grunden til at den aktuelle **rute** er under en fane for sig selv, er for at give plads til repræsentationen i tablettens brugergrænseflade.

Se rute

'Se rute' initieres når planlægger vil se en **rute** for en bestemt dato.

1. Planlæggeren navigerer til planlægningsfanen.
2. Planlæggeren finder og klikker på pågældende dato
3. Den ønskede **rute** vælges.

Muligheden for at se alle **ruterne** sker under planlægningsfanen da planlægger ofte kigger på ruterne i forbindelse med planlægningsprocessen.

Afvis forslag

Igangsættes af planlægger, når planlæggeren ønsker at annullere de ændringer, der er lavet på en **rute**. Brugsmønsteret starter i planlægningsfanen med en redigeret **rute**:

1. Planlægger navigerer til en vilkårlig fane, væk fra planlægningsfanen.
2. Et vindue vises, der fortæller at den foreslåede **rute** endnu ikke er gemt, og at ændringerne vil gå tabt, hvis man vælger at gå videre uden at accepterer den foreslåede **rute**.
3. Planlægger vælger at gå videre til et andet vindue.

Grunden til at planlægger skal have muligheden for at kunne afvise et forslag er, at planlæggeren kan fortryde de ændringer der er påført **ruten**, og dermed slipper for at ændre det tilbage manuelt.

Godkend forslag

Igangsættes af planlægger, når planlæggeren vil accepterer et foreslag.

1. Planlægger laver ændringer på en **rute**.
2. Planlægger trykker nu på knappen "Gem Ændringer".

Da planlægger har mulighed for afvisning af **rute** ved at skifte fane, skal planlægger også have muligheden for at gemme ændringer på en **rute**. Dette skifter **rutes** tilstand til 'planlagt'.

Se data for opgaver

Igangsættes af planlægger, når denne skal se informationer om en **opgave** der endnu ikke er tillagt en **rute**.

1. Planlæggeren navigerer til planlægningsfanen.
2. Planlægger højreklikker på **opgaven**.

Planlægger skal have muligheden for at se data om en **opgave** der endnu ikke er lagt på en **rute**. Denne data er vigtig, når planlæggeren skal placerer de uallokerede **opgaver** på **ruter**.

Se data for tidsperioder

Igangsættes af planlægger, når denne skal se informationer om en **tidsperiode** på en **rute**.

1. Planlæggeren navigerer til planlægningsfanen.
2. Planlægger højreklikker på **tidsperioden**.

Planlægger skal have muligheden for at se data om en **tidsperiode** der er lagt på en **rute**. Denne information er relevant når det skal vurderes om den skal fjernes eller flyttes, eller hvis planlægger skal kende **opgaven** på et bestemt **arbejde**.

5.2.2 Funktioner

Dette afsnit omhandler funktioner fra anvendelsesområdet. Disse beskriver, hvad systemet indholdsmæssigt tilbyder brugerne af programmet. Funktioner beskriver hvad systemet skal kunne gøre, hvorimod brugsmønstre beskrev hvordan systemet skal bruges. En liste med alle funktioner kan ses i tabel 5.4, hvoraf de komplekse funktioner bliver udspecificeret via dekomponering.

Funktioner	**Type**	**Kompleksitet**
Opret rute	Opdatering	Simpel
Fjern rute	Opdatering	Simpel
Tilføj hjælper til rute	Opdatering	Simpel
Fjern hjælper fra rute	Opdatering	Simpel
Tilføj tidsperiode til rute	Opdatering	Simpel
Fjern tidsperiode fra rute	Opdatering	Simpel
Accepter forslag	Opdatering	Simpel
Afslå forslag	Opdatering	Simpel
Indberet forsinkelse	Opdatering	Kompleks
• Tilføj tidsperiode til aktuel rute	Opdatering	Simpel
• Vælg aktuel arbejde ud fra vurderingskriterier	Beregning	Middel
• Forespørg hjælper om aktuel indsættelse	Beregning	Simpel
• Fjern tidsperiode fra aktuel rute	Opdatering	Simpel
Notificer planlægger om aktuel ændring	Signalering	Simpel
Se rute	Aflæsning	Simpel
Se opgave	Aflæsning	Simpel
Se planlægger notifikationer	Aflæsning	Simpel
Se hjælper notifikationer	Aflæsning	Simpel
Se hjælper	Aflæsning	Simpel
Se tidsperiode	Aflæsning	Simpel
Aflæs forslagsbegrænsninger	Aflæsning	Kompleks
• Aflæs ruter der ikke må ændres	Aflæsning	Simpel
• Aflæs opgaver der ikke må fjernes	Aflæsning	Simpel
• Aflæs hjælpere der ikke må fjernes	Aflæsning	Simpel
Beregn forbindelsestid	Beregning	Kompleks
• Find forbindelsestid	Beregning	Middel
• Opdater forbindelsestid	Opdatering	Simpel
Lav forslag	Beregning	Kompleks
• Fjern tidsperioder fra ruter ud fra forslagsbegrænsninger	Opdatering	Simpel
• Indsæt tidsperioder efter algoritme	Beregning	Kompleks
Foreslå arbejdsopgaver på rute	Beregning	Kompleks

Tabel 5.4: Funktionsliste

Opdateringsfunktioner

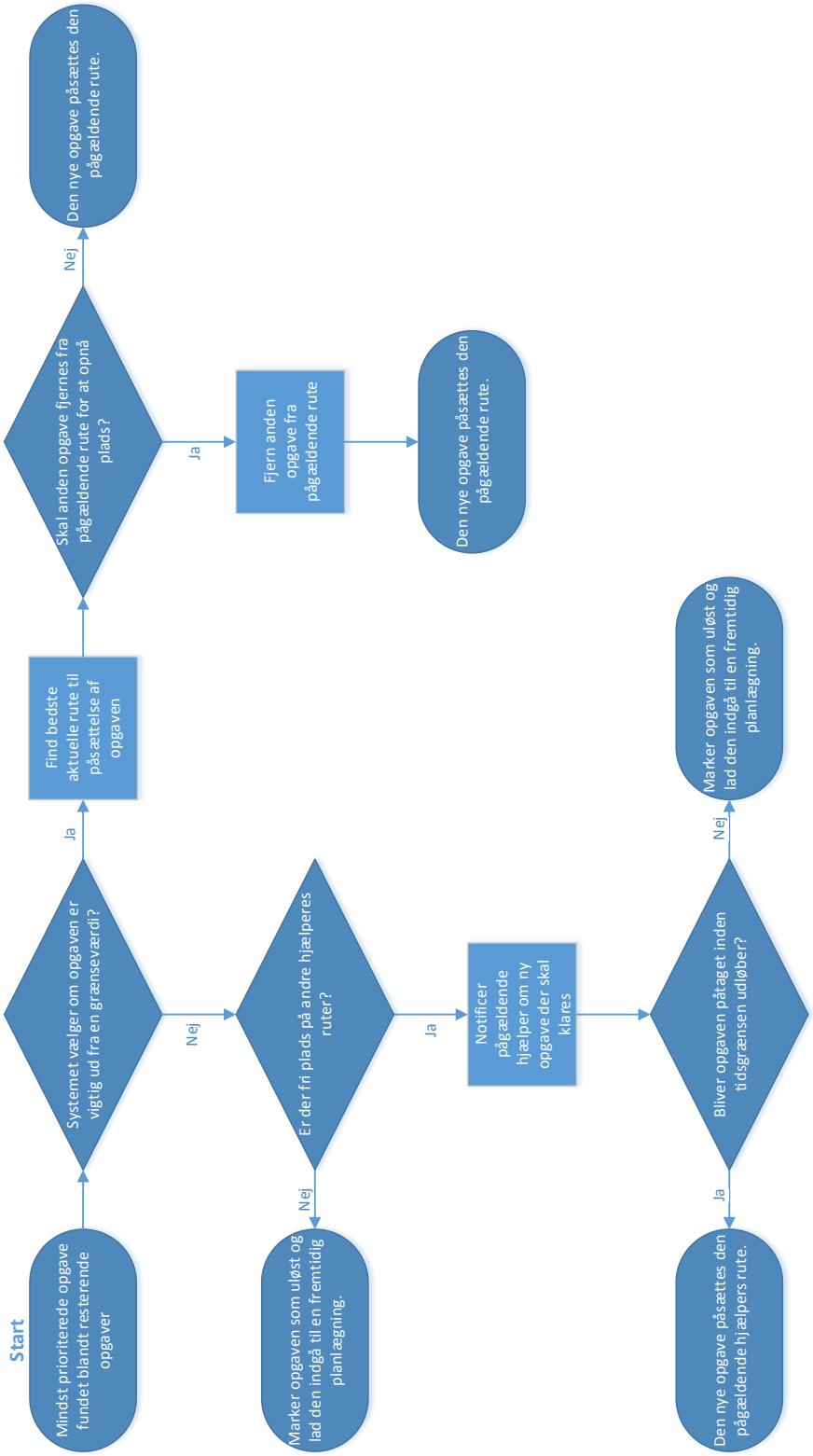
Systemet skal kunne administrere flere opdateringsfunktioner. Disse beskrives i det følgende:

- Opret rute: Opretter en ny **rute**. Funktionen er knyttet til hændelsen: *Planlægning påbegyndt*, der føder et **rute**-objekt. Denne funktion er valgt for at understøtte brugsmønsteret: 'Etablering af ny rute'
- Fjern rute: Fjerner en **rute**. Funktionen er knyttet til hændelsen: *Rute nedlagt*, der dræber et **rute**-objekt. Denne funktion er valgt for at understøtte brugsmønsteret: 'Slet rute'.
- Tilføj hjælper til rute: Tilføjer en **hjælper** til en specifik **rute**. Funktionen er knyttet til hændelsen: *Hjælper påsat rute*, da den påvirker **rute**. Denne funktion er valgt for at understøtte brugsmønsteret: 'Tilføj Arbejde til rute'.
- Fjern hjælper fra rute: Fjerner en **hjælper** fra en specifik **rute**. Funktionen er knyttet til hændelsen: *Hjælper fjernet fra rute*, da den påvirker **rute**. Denne funktion er valgt for at understøtte brugsmønsteret af samme navn: 'Fjern Arbejde fra rute'.
- Tilføj tidsperiode til rute: Tilføjer en **tidsperiode** til en specifik **rute**. Funktionen er knyttet til hændelsen: *Tilføjet til rute*. Denne funktion er valgt for at understøtte brugsmønsteret: 'Tildel hjælper til rute'.
- Fjern tidsperiode fra rute: Fjerner en **tidsperiode** fra en specifik **rute**. Funktionen er knyttet til hændelsen: *Fjernet fra rute*, da den påvirker **rute**. Denne funktion er valgt for at understøtte brugsmønsteret af samme navn: 'Fjern hjælper fra rute'.
- Accepter forslag: Ændrer en **rutes** tilstand fra foreslået til planlagt. Funktionen er knyttet til hændelsen: *Rute planlagt*, da den påvirker **rute**. Denne funktion er valgt for at understøtte brugsmønsteret: 'Godkend forslag'.
- Afslå forslag: Nedlægger en **rute** i tilstanden foreslået **rute**. Denne funktion er også tilknyttet *Rute nedlagt*, dog fjerner denne funktionalitet ikke **ruten** fra systemet, til forskel fra funktionen 'Fjern rute'. Denne funktion er valgt for at understøtte brugsmønsteret: 'Afvis forslag'.
- Indberet forsinkelse: Indberetter forsinkelse til systemet, for at lade systemet håndtere hvordan **ruterne** nu skal se ud. Funktionen er valgt for at tage hånd om aktuel planlægning, med hensyn til uforudsete hændelser, der medfører forsinkelser. Funktionen er knyttet til hændelserne: *Aktuelt arbejde tilføjet* og *Aktuelt arbejde fjernet*. Denne funktion skal bruges for at kunne udføre brugsmønsteret: 'Indberetning af forsinkelse'.

Funktionen: 'Indberet forsinkelse' kan være svær at forstå, da der her kan udspille sig mange forskellige scenarier. Flowchartet på figur 5.12 viser disse scenarier. Det centrale i funktionen er at den starter med at lede alle **opgaverne** på en **hjælpers rute** igennem for at finde den lavest prioriterede **opgave** ud fra en vurdering af hvilke **opgaver** der er mulige at udskyde. Der skal være sat en grænseværdi vurderet med aktørerne, for at afgøre hvad der skal gøres med den mindst prioriterede. Denne fremgang er valgt, fordi at hvis en **opgave** er vurderet vigtig, skal den reageres på, fremfor **opgaver** der kan udskydes.

Hvis **opgaven** bliver markeret som værende ikke vigtig vil serveren prøve at finde en **hjælper** med plads på sin **rute** for at kunne overtage **opgaven**. Dette er valgt sådan at, hvis ikke der findes en **hjælper** eller hvis der findes en **hjælper** der ikke når at respondere på en notifikation om hvorvidt vedkommende kan overtage **opgaven**, så vil **opgaven** blive markeret som uløst og indgå i fremtidig planlægning. Hvis hjælper accepterer notifikationen vil **opgaven** blive flyttet til vedkommendes **rute**.

Hvis **opgaven** bliver markeret som vigtig, vil serveren lede efter den **rute** der bedst kan overtage **opgaven**, enten i form af der er nok **ledig** tid på **ruten** eller om at der er mindre prioriterede **opgaver** der kan fjernes i fordel for den der skal på en **rute**.



Figur 5.12: Flowchart over "Indberet forsinkelse"

Signaleringsfunktioner

Systemet skal kunne administrere en enkelt signaleringsfunktionn. Denne beskrives i det følgende:

- Notificer planlægger om aktuel ændring: Notificerer planlæggeren om en ændring af aktuelle **ruter**. Den kritiske tilstand der giver anledning til denne funktion er når et **rute**-objekt i aktuel tilstand, får tilføjet eller fjernet **tidsperioder**.

Aflæsningsfunktioner

Systemet skal kunne administrere flere aflæsningfunktioner. Disse beskrives i det følgende:

- Se rute: Fremviser data om en specifik **rute**. Denne er valgt da både planlægger og hjælper skal bruge informationer om **tidsperioderne** og hvilken **hjælper** er tildelt en specifik **rute**. Denne funktion er valgt for at understøtte brugsmønsteret: 'Se rute'.
- Se Opgave: Fremviser data om en specifik **opgave**. Denne er valgt da både planlægger og hjælper skal bruge informationer om de **opgaver** som skal placeres som **tidsperioder** på **ruter**. Denne funktion er valgt for at understøtte brugsmønsteret: 'Se data for opgaver'.
- Se planlægger notifikationer: Fremviser notifikationer om ændringer der er sket i modellen. Denne skal bruges af planlægger til at se ændringer der er sket i modellen, f.eks. når et eksternt system tilføjer et nyt **behov**, eller når en **tidsperiode** bliver fjernet fra en aktuel **rute**. Denne funktion understøtter brugsmønsteret: 'Se notifikationer'.
- Se hjælper notifikationer: Denne skal bruges af planlægger til at se ændringer der er sket i modellen som er specifik for denne **hjælpers** aktuelle **rute**. Denne funktion understøtter brugsmønsteret: 'Information om ændring i en aktuel rute til hjælpere'.
- Se hjælper: Fremviser data om en specifik **hjælper**. Denne skal planlægger bruge, for at se hvilken **hjælper** der er placeret på, eller skal placeres på, en **rute**. Denne funktion understøtter brugsmønsteret: 'Tildel hjælper til rute', da planlægger skal brug **hjælper** information til tildelingen af **ruter**.
- Se Tidsperiode: Aflæser data om en specifik **tidsperiode**. Denne funktion skal bruges af både planlægger og hjælper, da de skal fremvises på repræsentationen af en **rute**. Denne funktion er valgt for at understøtte brugsmønsteret: 'Se data for tidsperioder'.

- Aflæs forslagsbegrænsninger: Aflæser de begrænsninger planlægger har sat, som begrænser et optimeringsforslag. Begrænsinger er **ruter**, **opgaver** og **hjælpere** som ikke må flyttes rundt. Denne funktion skal bruges af planlægger, for at se de forslagsbegrænsninger, som planlægger har sat. Denne funktion understøtter brugsmønsteret: 'Optimering af flere ruter', da de begrænsninger der allerede er sat, skal kunne ses af planlægger.

Beregningsfunktioner

Systemet skal kunne administrere flere beregningsfunktioner. Disse beskrives i det følgende:

- Beregn forbindelsestid: Beregner tiden det tager at køre imellem to **borgere** og opdaterer **forbindelsestid** herefter. 'Find forbindelsestid' er sat til middel, da vi kan finde denne med en underliggende korttjeneste. Grundlaget for denne beregning kommer fra modellen, da klassen: **forbindelsestid** kræver en beregning. Både planlægger og hjælper har brug for denne beregning til de **transport** der skal repræsenteres.
- Lav forslag: Laver optimeringsforslag for alle **ruter** ud fra forslagsbegrænsninger. Grundlaget for denne beregning stammer fra planlægger. I stedet for at kigge på dele af den samlede plan, kan planlægger med denne beregning optimere **ruten** ud fra en helhed der vurderer hele den hidtige plan. Denne funktion understøtter brugsmønsteret: 'Optimering af flere ruter'.
- Foreslå arbejdsopgaver på rute: Laver et forslag hvor et antal løse **opgaver** er indsat på en enkelt **rute**. Grundlaget for denne beregning stammer fra planlægger. Beregningen er valgt, da den støtter planlægger under tildeling af **opgaver** til en enkelt **rute**. Denne funktion understøtter brugsmønsteret: 'Lad systemet foreslå hvordan arbejdsopgaver tilføjes en rute'.

5.3 Brugergrænseflade

Dette afsnit omhandler systemets brugergrænseflade. Afsnittets opbygning er inspireret af *OOA&D*s kapitel om brugergrænseflade[3]. Selve indholdet af afsnittet lægger vægt på anbefalinger fra *Designing Interactive Systems*[4], samt kapitlet om brugergrænseflader i *Objektorienteret Analyse & Design*[3].

Afsnittet redegører for valg af dialogform. Derefter vil brugergrænsefladens layout og de bagvedliggende beslutninger blive beskrevet, efterfulgt af en oversigt over brugergrænsefladen i form af et navigeringsdiagram. Til sidst gennemgås eksempler af brugergrænsefladens bestanddele med dertilhørende elementer.

Dialogform

Vi valgte at bruge mønsteret *direkte manipulation* til både navigering og manipulering af objekter i brugergrænsefladen, samt mønsteret *skemaudfyldelse* til indtastning af data. Årsagen til disse valg var, at der var en klar forventning fra brugeren om, at træk-og-slip manipulation var muligt (se afsnit 8.1). Desuden er *direkte manipulation* let at huske[3], hvilket øger planlæggerens effektivitet i brug af systemet.

Det er desuden muligt at undgå mange fejl ved at anvende *direkte manipulation*[3], f.eks. ved at flytte **opgaver** ind ved hjælp af træk-og-slip, i modsætning til at skulle, via *skemaudfyldelse*, indtaste tidspunkter hvor hver **tidsperiode** der påsættes **ruten**. Træk-og-slip opfordrer til at planlæggeren nemmere kan udforske hvorledes **ruterne** kan planlægges. F.eks. kan dette gøres ved, at der kan flyttes rundt på de forskellige **opgaver**, for at undersøge forskellige muligheder for ruteplanlægningen.

Vi valgte *skemaudfyldelse* til indtastning af data, da mængden af data der skal udfyldes er minimal i forhold til funktionerne *etablering af nye ruter* og *ændring af borgers adresse*. *Skemaudfyldelse* har en tendens til at fylde store dele af skærmbilledet[3], men vi vurderer at det er acceptabelt i forhold til disse funktioner.

For at markere besøg som aflyste, valgte vi at bruge et mønster for menu. Menuen er forbeholdt de få funktioner, der ikke kan udformes som *direkte manipulation* på en ligefrem måde.

Layout

Valget af *direkte manipulation* som det hovedsagelige mønster for interaktion med systemet, sætter vilkårene for designet. Derfor er det vigtigt at have et velovervejet layout for systemet. Der er lagt vægt på ikoner, farvevalg og strukturering af elementerne i vinduerne, til at fastlægge layoutet for brugergrænsefladen.

Der er lagt vægt på at gruppere informationer og funktionaliteter således, at relaterede elementer samles så vidt muligt. På den måde kædes relaterede elementer logisk sammen i brugergrænsefladen. F.eks. er knapper til at manipulere med en **rutes tidsperioder** placeret

tæt på information om en **rute**. Denne form for sammenkædning af relaterede elementer er gennemgående i brugergrænsefladen. Evalueringen af brugergrænsefladen beskrevet i dette afsnit er dokumenteret i afsnit 8.2.

Ikoner

Vi har valgt at repræsentere relevante objekter samt navigationen ved ikoner. Ikonerne kan være med til at fremme forståelse af systemet, og identifikation af objekter i brugergrænsefladen. En fordel er ligeledes at markere **tidsperioderne** på **ruter** med ikoner, således planlægningen kan foretages af farveblinde planlæggere. Udskiftning af ikoner ved **transport** alt efter hvilken transportform der bruges på en given **rute**, er ligeledes en mulighed. F.eks. kan der fremkomme et bil-ikon når der skal køres i bil, kontra en cykel-ikon der skal cykles på **ruten**. I figur 5.13 ses anvendte ikoner, som er hentet fra flaticons.net, og hvad de repræsenterer i brugergrænsefladen.

Figur 5.13: Anvendte ikoner. Anvendelsen af ikonerne er illustreret i eksemplet på figur 5.16

Da planlæggere i hjemmeplejen ikke har en decideret uddannelse indenfor kontorarbejde, kan vi ikke forvente nær så meget af planlæggerens skrive-og læsefærdigheder. En mulighed er derfor at udskifte så meget tekst som muligt med ikoner. Vi har imidlertid valgt ikke at gøre dette, da andre af planlæggerens opgaver ofte kræver læsning og skrivning. Dog er det værd at overveje flere ikoner til brugergrænsefladen til tablet-versionen, da hjælperne hurtigt skal kunne overskue deres **rute**, uden at få for mange detaljer med.

Farvevalg

Farvevalget til brugergrænsefladen er baseret på vestlige farvekonventioner[4]. Et gennemgående farvetema med blå, grå og hvid er valgt for at fastholde en simpel, lys brugergrænseflade, der symboliserer rolighed, stabilitet og neutralitet[4], hvilket mindsker distraktioner for brugeren. Ligeledes er det farver, der symboliserer at brugeren kan stole på systemet, hvilket er relevant da planlæggeren skal kunne stole på at planlægningen udføres korrekt. Ligeledes er der kun

anvendt få farver i overensstemmelse med Aaron Marcus' regler for brugergrænseflader[4].

Til fremhævning af valgte elementer anvendes komplementærfarver for at vise, dynamik i brugergrænsefladen. F.eks. til navigation, er brugerens position i systemet symboliseret ved, at det valgte faneblad er en anden farve end den normale blå (se figur 5.16). Ved at fremhæve hvor brugeren er i systemet, fastholder bruger følelsen af kontrol.

Visualisering af ændringer

Når planlæggeren foretager ændringer på f.eks. **ruter** kan være svært at overskue, hvad der er blevet ændret, hvis der bliver foretaget mange ændringer i en planlægningssession. Derfor vil vi sørge for at ændringer vises visuelt med farver, således planlæggeren kan overskue ændringerne. Dette er især essentielt ved udførelse af automatisk planlægningsfunktion, hvor planlæggeren ikke vil kunne skelne mellem det nye foreslag og den gamle **rute**, hvilket hindrer planlæggeren i at se om den nye **rute** er acceptabel.

Chunking Under planlægningen er det nødvendigt at brugeren skal huske forskellig information omkring **borgere**, **opgaver** og **ruter**. Ligeledes foregår planlægningen i et miljø, hvor telefoner ringer og der tales med andre ansatte, hvilket resulterer i afbrydelser af planlægningsprocessen. Derfor er det vigtigt at planlæggeren kan afbryde og fortsætte planlægningsarbejdet uden at miste overblikket.

En måde at gøre det nemmere for brugeren at arbejde med information, er at udvikle brugergrænsefladen med såkaldt *chunking*[6]. Chunking går ud på at mindske og præsentere information, således at det er nemt at huske for brugeren. Ved specifikt at begrænse information der vises, til kun det mest relevante, mindsker man mængden af information der skal huskes af brugeren, og øger derfor brugbarheden af systemet, da man ikke skal skifte frem og tilbage mellem vinduer, for at tilgå den information der er nødvendig.

Chunking er især vigtigt i brugergrænsefladen til hjælpernes tablets, da hjælpere hurtigt skal danne sig et overblik over deres **rute**, og derfor ikke har tid til at nærlæse detaljer. Ligeledes er det vigtigt at hjælper skal huske hvilke **opgaver** der skal udføres ved en **borger**, samt specielle forhold der skal tages hensyn til.

Vi har valgt at anvende principperne bag chunking til kun at vise det mest nødvendige information om objekterne i brugergrænsefladen. Hvis yderligere information ønskes, indkapsles dette i et Tool Tip, der kan fremkaldes ved dobbeltklik på det givne objekt. På den måde sikres, at planlægningen kan udføres uden for meget "informations-støj".

Feedback og affordance For at brugeren kan vide hvad deres interaktion med systemet gør, vil vi have *feedback* på kontrolbestanddele i systemet. *Feedback* er princippet bag, at sikre at brugeren ved at handlinger i systemet faktisk gør eller ændrer noget. F.eks. vil gem-knappen i planlægningsvinduet vise en besked når gem-instruktionen er udført. Således er der ikke tvivl om, at den ønskede handling blev udført. Ligeledes vil vi have *feedback* fra funktioner, der ikke udføres øjeblikkeligt. F.eks. vil der ved automatisk planlægning fremkomme en progress-bar, der viser hvor langt systemet er med at foretage handlingen.

På samme tid vil vi sikre *affordance*[4] i systemet. *Affordance* er et princip, der går ud på at sikre brugerens opfattelse af hvordan man anvender systemet er korrekt. Vi vil sikre *affordance*[4] ved at inkorporere hjælpebeskeder i tilfælde af at brugeren forsøger at udføre invalide handlinger via træk og slip. F.eks. hvis planlæggeren prøver at trække en for **opgave** med for lang varighed ind i en **rute**, hvor der ikke er nok **ledig** tid tilbage.

Genvejstaster I det nuværende system i hjemmeplejen er genvejstaster en hurtig måde for planlæggeren at navigere systemet. Ligeledes vil vi derfor lave genvejstaster til mange af de mest gængse funktioner. Hvilke taster der skal anvendes er ikke fastlagt på dette tidspunkt i systemudviklingen.

Oversigt

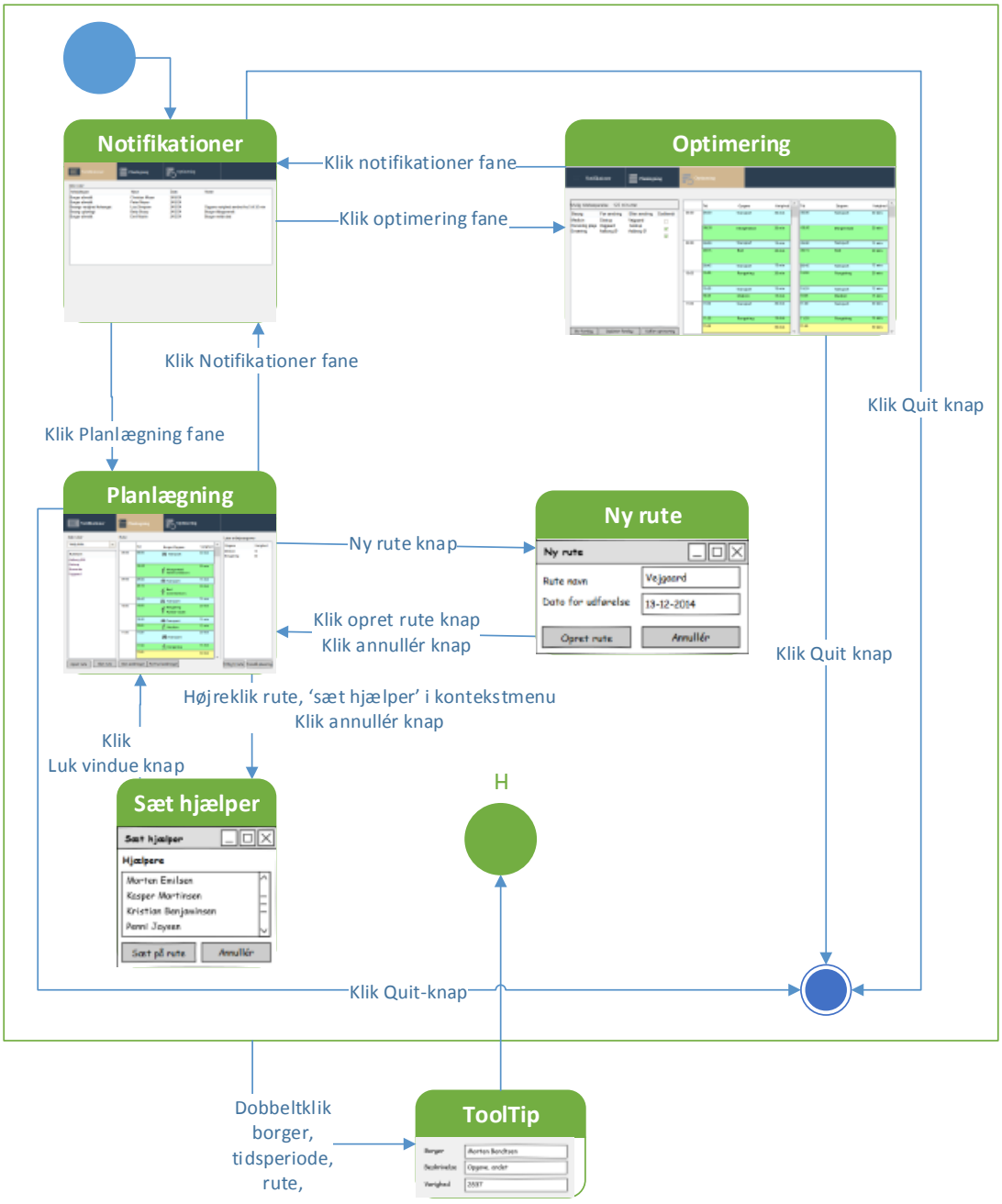
I figur 5.14 ses et navigeringsdiagram for brugergrænsefladen. Det fremgår hvordan man kan navigere mellem de forskellige dele af brugergrænsefladen, samt hvilke funktioner der findes i hver del. Frem for at repræsentere klasser i vinduer har vi valgt at gøre det med faneblade. Man kan se konceptet bag vores brug af faneblade i figurerne i afsnit 5.3.

Faneblade gør det muligt at indfange vinduer i et "hovedvindue", hvor man skifter mellem visning af de forskellige faneblade, ved navigere mellem faneblade i toppen af programmet. Vi har valgt denne form for visning, for at undgå uoverskuelighed i programmet, hvilket vinduer i store mængder kan være årsag til. Dog giver vinduer frihed til brugeren til at placere vinduer efter ønske, hvilket især er en fordel, når flere skærme er i brug. F.eks. ville brugeren da kunne placere et vindue, der viser en **rute** på en skærm, og en anden **rute** på en anden skærm under planlægningen.

Da systemet skal anvendes sammen med andre systemer i hjemmeplejen, er det ufordelagtigt at have mange vinduer til at optage plads i skærmbilledet[3], hvilket ville fjerne overblikket når brugeren skal skifte mellem de forskellige programmer.

Idéen er at lade fanebladende fungere som "beholdere" for vinduer, og lade det være muligt, via træk-og-slip, at trække vinduer ud af hovedvinduet efter behov. På den måde fastholder man overblikket, ved at gøre det muligt at indfange vinduer, men på samme tid begrænser man ikke brugerens frihed til at flytte vinduer rundt efter behov. Vi har valgt at repræsentere funktionalitet baseret på typen af arbejdsopgaver. F.eks. er planlægningsrelateret funktionalitet placeret under fanebladet planlægning.

På figur 5.14 ses notifikations-, planlægnings- og optimeringsfanebladene. Systemet starter i notifikationsfanebladet.



Figur 5.14: Navigeringsdiagram af brugergrænsefladen: Systemet startes i notifikationsvinduet. Fra hvert fanebladsvindue kan man navigere til hvert af de andre. Nuværende position i systemet er markeret med fremhævning af valgte faneblad. ToolTip vinduet kan tilgås fra alle vinduer i systemet, og ved luk af ToolTip, returneres man til det tidligere vindue, markeret med H.

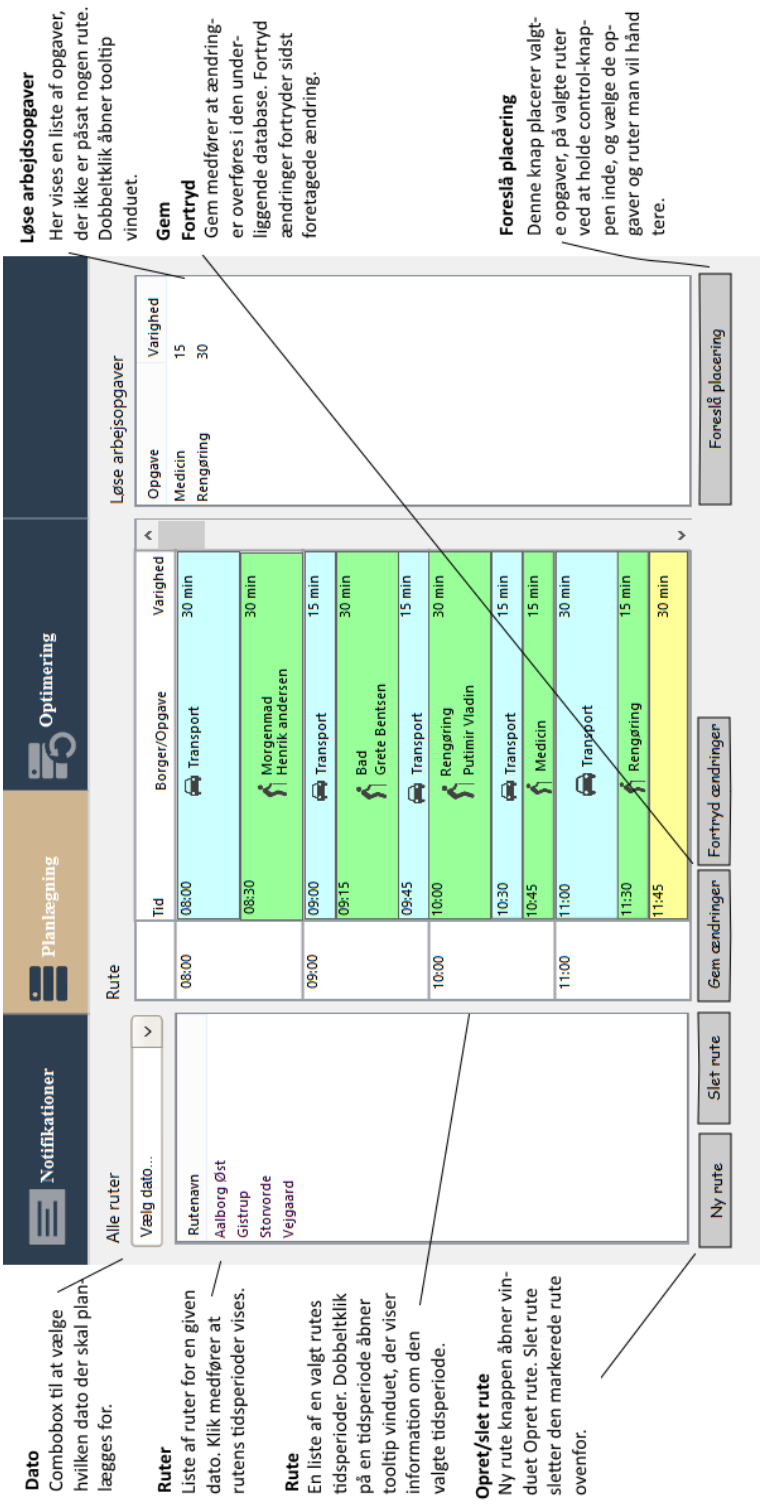
Eksempler

Systemet har 6 vinduer:

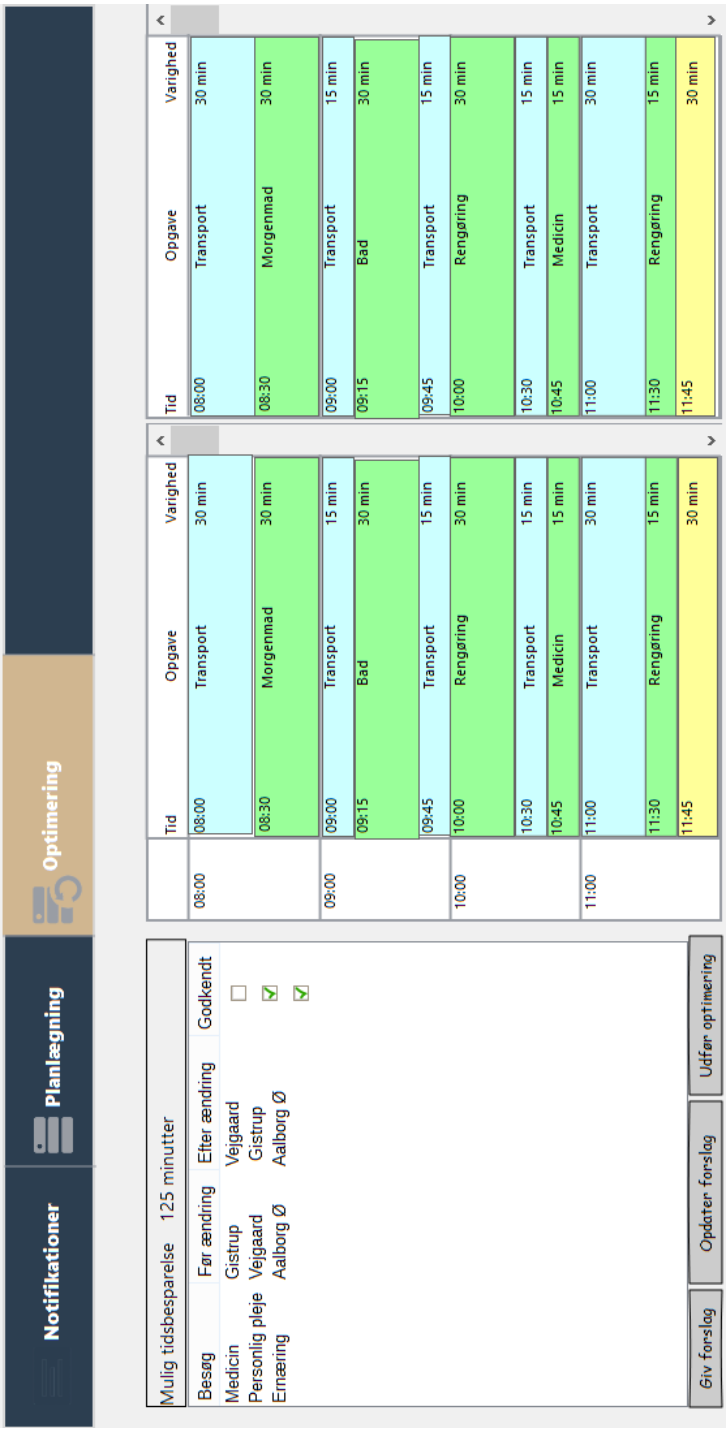
- Notifikationsvinduet i figur 5.15
- Planlægningsvinduet i figur 5.16
- Optimeringsvinduet i figur 5.17
- Tooltip vinduet
- Ny rute vinduet figur 5.19
- Sæt hjælper vinduet figur 5.18

De første konkrete prototyper af brugergrænsefladens vinduer blev udviklet med Blend SketchFlow, og efterfølgende blev layoutet udført i mockup-værktøjet Pencil. Implementeringen af brugergrænsefladen blev udført i Windows Presentation Foundation (WPF), hvoraf detaljer blev tilpasset dennes standard. Windows konvention for placering af knapper er anvendt på vinduerne.

Figur 5.15: Planlægningsvindue



Figur 5.16: Planlægningsvindue



Figur 5.17: Optimeringsvindue

Figur 5.18: Ny rute vindue

Figur 5.19: Sæt hjælper vindue

5.3.1 Den tekniske platform

Systemet udvikles til en PC og til tablet. Da systemet anvendes på flere platforme, er et objekt-orienteret programmeringssprog til flere platforme oplagt at bruge. Vi vælger at programmere systemet i C#, da dette er programmeringssproget vi har mest erfaring med, samt at vi ikke har erfaring med nogen programmeringssprog henvendt til flere platforme. Ligeledes vil det hindre hvor meget vi kan nå at implementere, hvis vi skal sætte os ind i et nyt programmeringssprog.

For at have persistent data, skal data opbevares i en database. Denne database skal være relational for samtidigt at bevare forholdene mellem klasser i systemet. Brugergrænsefladen skal være baseret på en kombination af faneblade og vinduer og udvikles med Windows Presentation Foundation (WPF). Systemet til PC skal betjenes med mus og tastatur, og til tablet via berøring af skærmen.

6 Design

I dette afsnit forklares de designmæssige valg, vi har foretaget, med henblik på arkitekturen af komponenter og deres indbyrdes kobling. Vi gennemgår overordnede kriterier for systemet, og forklarer derefter valg af arkitektur og komponenter.

6.1 Kriterier

For at gøre arbejdet med design og implementering mere overkommeligt, kan det være en fordel at vælge en række kriterier, ud fra hvilke områder, der skal være mest fokus på under disse processer. Dette kan også efterfølgende bruges til at evaluere på kvaliteten af systemet, da man efter udarbejdelsen kan undersøge hvorvidt om systemet stemmer overens med de udvalgte kriterier.

Vi har valgt at prioritere en række klassiske kriterier, fundet i *OOA&D* bogen[3], som vi, med udgangspunkt i vores system, har fortolket. Valget bygger på i hvilken grad, vi vil stræbe efter at opfylde de pågældende kriterier. Vores prioritering af disse kriterier kan ses i tabel 6.1 og en forklaring af vores fortolkning af hvert kriterie vil efterfølgende blive beskrevet.

	1	2	3	4	5	Irrelevant
Brugbart		x				
Sikkert					x	
Effektivt			x			
Korrekt	x					
Pålideligt		x				
Vedligeholdbart				x		
Testbart				x		
Fleksibelt			x			
Forståeligt			x			
Genbrugeligt						x
Flytbart						x
Integrerbart					x	

Tabel 6.1: Prioritering af designkriterier

Brugbart er hvorvidt om brugerne har let ved at forstå og gøre brug af brugergrænsefladen.

Grunden til at dette er blevet vægtet højt er, at vores system skal bruges hver dag til planlægning og derfor er det meget vigtigt at det er nemt at interagere med.

Sikkert er hvor meget, der bliver gjort for at sikre data og adgang til systemet. Udover at adskilde adgangsniveauet i de forskellige klienter for henholdsvis hjælpere og planlæggere, vil vi ikke beskæftige os med sikkerhedsmæssige aspekter.

Effektivt beskriver i hvor høj grad, vi vil stræbe efter et hurtigt program. Den er prioritereret middelt, da det ikke er essentielt, at de komplekse dele af programmet er så hurtige som muligt. Det er f.eks. ikke noget problem, hvis optimering af **ruter** tager fem minutter. Der ønskes dog en rimelig begrænsning på tidsforbruget.

Korrekt har vi fortolket som værende hvorvidt om kravene fra modellen og funktionaliteten bliver opfyldt. Vi har valgt, at dette er det vigtigste kriterium, som vi vil stræbe efter at opfylde, da det vil være den vigtigste opgave, systemet skal varetage. Hvis kravene fra modellen og funktionerne ikke opfyldes, vil planlægningen ikke kunne udføres korrekt, hvilket er meget vigtigt.

Pålideligt er i dette tilfælde hvorvidt om brugerne stoler på at programmet gør som det skal når der gøres brug af dets funktionalitet. Dette skyldes at systemet har fået tiltænkt en del automatisering, og da brugerne ikke har nogen kontrol over hvad der sker når automatiseret funktionalitet bruges, er det derfor vigtigt at brugerne kan se ændringerne af disse.

Vedligeholdbart beskriver hvor mange kræfter der bliver brugt på at lave et design som kan svare sig at finde og rette fejl i. Dette har vi valgt at prioritere forholdsvis lavt, det skal dog være muligt for os selv at finde rund i det og ikke bruge for lang tid på at lave ændringer da vi har andvendt iterativ metode.

Testbart beskriver i hvor høj grad, systemet designes med henblik på at nemt at kunne teste det. Vi har vurderet, at automatiseringsfunktionaliteten bør testes for at sikre, at vi den opfylder kravet fra funktionaliteten i analysen.

Fleksibelt er vigtigt uanset størrelsen af et projekt, især når udviklingsprocessen er iterativ, da kravene hurtigt kan ændre sig. Systemet er desuden afhængige af Google Maps, der er en service tilbudt af en tredjepart, hvilket betyder, at programmet bør kræve minimale ændringer, hvis servicen bliver ændret, lukket eller lignende.

Forståeligt er hvorvidt om designet af systemet skal være overkommelig at sætte sig ind i.

Genbrugbart er ikke noget vi har tiltænkt designet, da komponenterne af systemet er udviklet med speciel henblik på hjemmeplejens planlægningssituation. Derfor har vi ment at dette kriterium var irrelevant at stræbe efter i dette system.

Flytbart er hvorvidt om der er blevet tiltænkt at systemet skal kunne flyttes over på andre tekniske platforme. Vi har valgt dette som værende irrelevant, da vi i dette projekt har udviklet i C#, som beskrevet i den tekniske platform i afsnit 5.3.1.

Integrerbart er prioriteret lavt, da vi ikke har planer om at integrere systemet i hjemmeplejens nuværende systemer.

6.2 Arkitektur

Denne sektion omhandler systemets komponent arkitektur. Den overordnede struktur vil blive beskrevet, herunder koblingen mellem komponenter, samt måden MVVM mønstret er implementeret i vores arkitektur design. I dette afsnit vil komponenter blive skrevet med *kursiv tekst*.

6.2.1 Model-View-ViewModel

Model-View-ViewModel (MVVM) er en variation af MVC mønstret, der er et designmønster til at udvikle lagdelt arkitektur. MVVM er udviklet med C# og WPF i tankerne, og det er derfor oplagt at MVVM anvende i kombination med dem. MVVM adskiller et system i *View*, *ViewModel* og *Model*, og definerer retningslinjer afhængigheder og koblinger mellem disse lag.

Model Model indkapsler data systemet behandler. Modellen har ikke adfærd der manipulerer eller på anden måde beregner på data.

View Viewet indkapsler præsentationen af data til brugeren igennem brugergrænsefladen. Ansvaret for den visuelle repræsentation af programmet, samt håndtering af input fra brugeren ligger i View. View koples på ViewModellen igennem *DataBinding*.

ViewModel ViewModel er en abstraktion af Viewet, og indkapsler funktionalitet til at vise og manipulere modellen. ViewModel udstiller disse metoder og funktionalitet til View, der kan lave DataBinding til en brugerkontrol såsom en knap i brugergrænsefladen. På den måde

kan man aktivere funktionalitet i ViewModellen ved at interagere med brugergrænseflade, og derved i sidste ende manipulere med data i modellen.

Et *View* kender til en *ViewModel*, som udstiller dele af en *Model*. På den måde skabes der lav kobling mellem lagene, da model ikke afhænger af viewmodel og viewmodel ikke er afhængig af view. Denne form for arkitektur gør det derfor muligt at udskifte dele, uden det påvirker resten af arkitekturen. F.eks. kan view udskiftes uden at påvirke viewmodel og model. Dette er nyttigt, da vores system fungerer på både PC og tablet, hvor brugergrænseflade er forskellig mellem disse to platforme. Ved at have MVVM designmønsteret, opnår vi derfor muligheden for at have udvikle brugergrænseflader til forskellige formål, med den samme underliggende arkitektur og model.

6.2.2 Komponentarkitektur

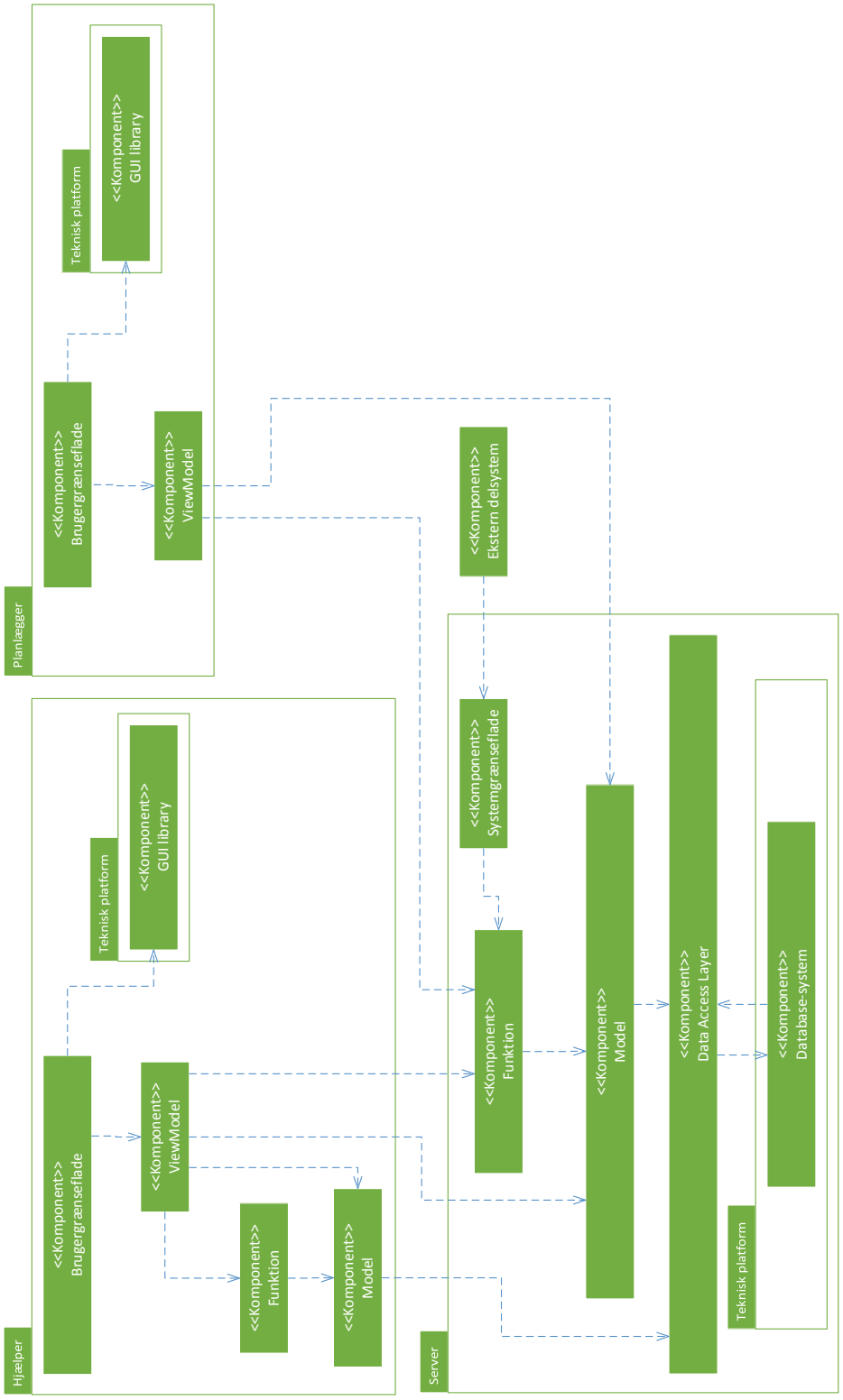
Vi har valgt at gøre brug af en klient-server arkitektur som set på figur 6.1. Vi har valgt at lave én server, samt 2 klienter der hver har relationer til serveren. De to klienter er henholdsvis planlægger klienten og hjælper klienten. Serveren er ment som værende en maskine med meget regnekraft, som hver af de to klienter kan tilgå for at indhente informationer, samt tilgå for at få udføre regnetunge funktioner.

Da vi har valgt at benytte MVVM mønstret til designet af arkitekturen, har hver klient en *ViewModel* komponent, samt en *Brugergrænseflads* komponent. For mere information om MVVM mønstret kan afsnit 6.2.1 læses. Mønstret er valgt for at få modularitet i programmet, for nemmere at kunne implementerer systemet på flere platforme, samt for at skabe lav kobling mellem klasser og komponenter. MVVM gør desuden at ViewModel komponenterne vil indeholde designet af alle aflæsningsfunktioner. Designet af signaleringsfunktionen vil gå igennem planlæggerklientens viewmodel, med et observatørmønster. Derfor indeholder funktionskomponenterne kun opdatering og beregningsfunktioner. Vi har valgt en distribueret funktionalitet arkitektur, fordi ViewModellerne skal designes efter de specifikke brugergrænseflader.

Serveren indeholder en *funktion* komponent, hvilken varetager funktioner som ikke kan bruges offline. Hjælperklientens og planlæggerklientens er *viewmodel* er afhængig af denne funktionskomponent. Serveren indeholder desuden en *model* komponent, hvilken repræsenterer en model af problemområdet. Både hjælper og planlægger har adgang til servers model komponent, adgangen er dog ikke direkte da MVVM mønstret så vidt muligt ønskes overholdt. Data Access Layer er det lag der kommunikere med database systemet, som er indsat for at vise

at databasen ikke kender modellen, men er afhængig af et lag imellem den og modellen. Systemet har desuden komponenten *systemgrænseflade*. Denne udstilles som et interface til andre eksterne systemer, som derigennem kan kommunikerer med vores system (f.eks. visitator). Disse kan indberette informationer om nye **borgere** der skal tilføjes til databasesystemet.

Et problem ved denne uddeling af funktionalitet til *server* komponenten, er spørgsmålet om at være tilsluttet serveren på alle tidspunkter. Forbindelsen skulle eksisterer i form af en trådløs internet opkobling. Et problem kunne opstå, hvis en hjælper kommer ud i et område hvor der ikke kan oprettes net forbindelse, hvorved hjælperen ikke vil have adgang til serverens funktionalitet. For at undgå denne problematik har vi valgt at hjælperens klient skal være afhængigt af intern funktionskomponent og modelkomponent som vist på figur 6.1, til at kunne arbejde på hvis der ingen forbindelse er til serveren. Hvis der bliver foretaget ændringer offline vil disse ændringer blive propageret til serveren når klienten igen får forbindelse. Funktionen: 'Indberet forsinkelse' forklaret i figur 5.2.2 er den funktionalitet der skal kunne give den respektive hjælper en opdateret rute, og sende ændringen ud som påvirker planlægger og andre hjælpere, så snart der er forbindelse igen.



Figur 6.1: Design af komponent arkitektur

6.3 Komponenter

Vi vil i dette afsnit uddybe nogle af de essentielle komponenter fra system arkitekturen, som er beskrevet i afsnit 6.2. Dette indebærer modelkomponenten, planlæggerens View og Viewmodel, samt serverens funktionskomponent. Klasserne **subjekt** og **observatør**, der er en del af observatørmønstret, er ikke vist i diagrammet i figur 6.2 for at undgå unødig kompleksitet. Vores brug af observatørmønstret er beskrevet i afsnit 6.3.3.

6.3.1 Modelkomponent

Dette afsnit omhandler designet af modelkomponenten, der er baseret på modellen fra analysen. Klasserne og hændelserne fra analysen af problemområdet i afsnit 5.1 realiseres og repræsenteres i modelkomponenten.

Hændelsen *Fravær meldt* er en iteration på **borger** og en sekvens på **opgave**. Dette modellerer vi i modelkomponenten som klassen **fravær** på **borger**, med attributterne fra *Fravær meldt*.

Hændelserne *ansat* og *fratrådt* er sekvenser på **hjælper**, og derfor har vi valgt at modellere det i designet som attributten `ansat` på **hjælper**.

Under designprocessen opstod der en nødvendighed for, at **forbindelsestider** kunne modellere ikke blot forbindelser imellem **borgere**, men også til det sted, hvor **hjælpere** befinder sig før det første **arbejde** på deres **ruter**. Dette blev til klassen **Afdeling**. Da **afdeling** og **borger** klasserne har flere attributter og associationer til fælles, har vi valgt at generalisere dem til klassen **AdresseIndehaver**.

Hændelsen *Adresse ændret* var en iteration på **borger** klassen og er til fælles med **afdeling**. Derfor har vi valgt at modellere den som klassen **adresse**. Denne klasse indeholder en `adresse` tekst og en `dato` for hvornår **adressen** den beboede. En **AdresseIndehaver** kan have én til mange **adresser**, hvor vi kigger på attributten `dato`, hvis vi vil vide hvilken der er den nuværende. Der er således kun én gældende **adresse** ad gangen.

Hændelsen *rute udført* var en iteration på **præference** og **hjælper**, samt en sekvens på **opgave**, **rute** og **tidsperiode**. Vi valgte at **rute** skulle have en attribut der markerede denne hændelse, da **hjælpere** allerede var forbundet med en mængde **ruter**. Ifølge **rutes** adfærd, sker *rute udført* som grundregel efter noget tid, hvis ikke *rute nedlagt* sker inden da. Vi lader derfor `dato`

vise hvorvidt en **rute** er udført. *Rute udført* er også nødvendig at repræsentere på **præference**, da vi ikke valgte at forbinde den med **rute**, trods fælles hændelse. Da det er en iteration, repræsenterer vi den ved klassen **grad** der findes én til mange gange på en **præference**.

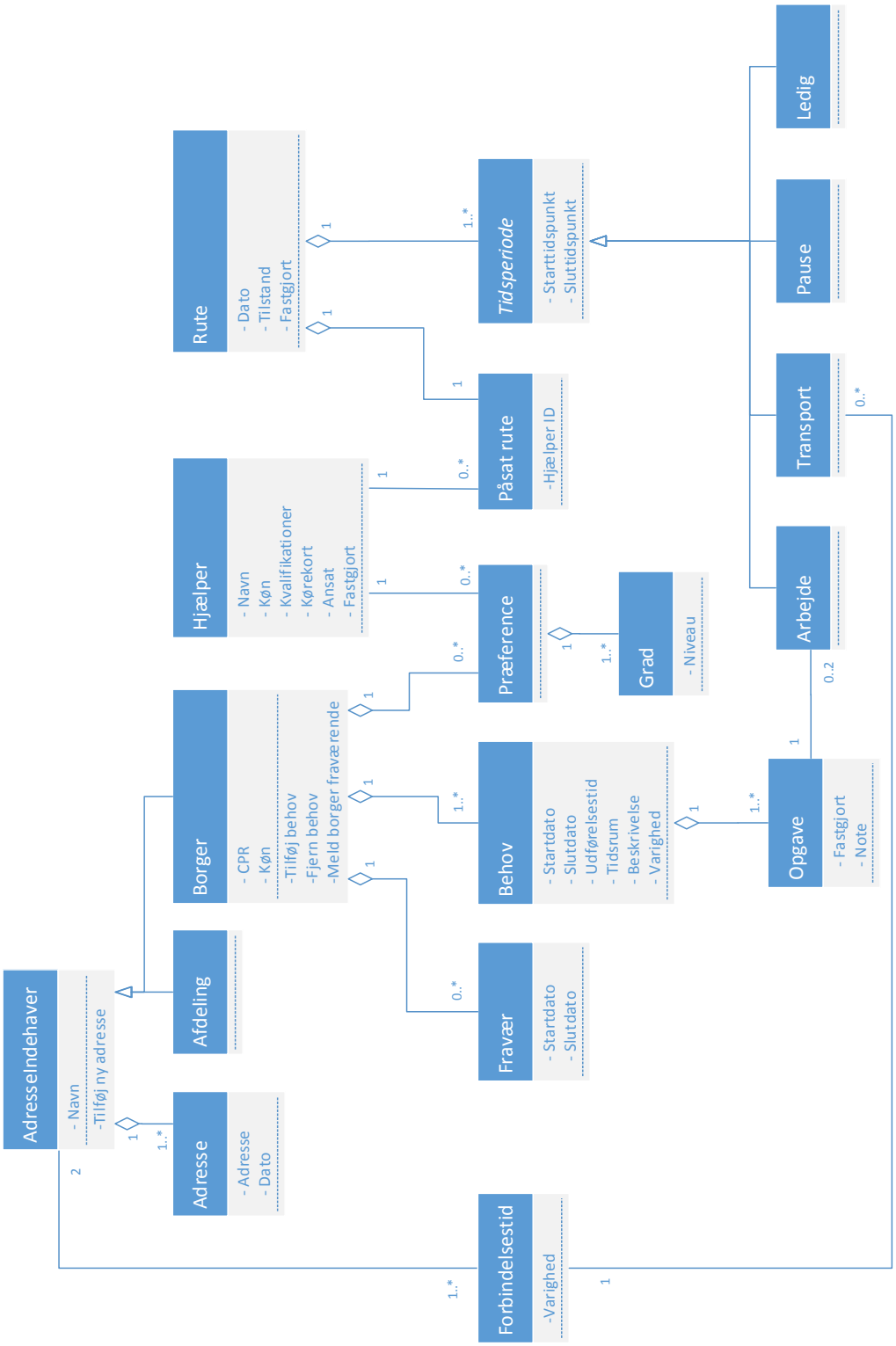
Hændelserne *Hjælper påsat rute* og *Hjælper fjernet fra rute* sker som iteration på **rute**, og derfor bruges en klasse **påsat rute**, til at modellere dette. **Påsat rute** har et `hjælper ID` der identificerer hvilken **hjælper** der er påsat **ruten**. Da en **hjælper** kan påsættes flere **ruter**, laves et relateringsmønster med **hjælper**, som repræsenterer dette.

På **rute** har vi hændelserne *planlægning påbegyndt*, *rute planlagt* og *rute initieret* som sker i sekvens og skifter **rutes** tilstand. Dette er modelleret via en attributten `tilstand` på **rute**, som kan have de tre værdier, 'Planlægning', 'Planlagt', 'Aktuel'.

Rute har også hændelserne *tilføjet til rute* og *fjernet fra rute* som iterationer. Disse hændelser refererer til når **tidsperioder** bliver tilføjet eller fjernet. Ifølge *OOA&D*[3] bør man her tilføje en ny klasse, for at kunne identificere den enkelte hændelse. Dette krav optræder dog allerede fra problemområdet i form af klassen **tidsperiode**.

Operationer der skal dække opdateringer fra det eksterne system, påvirker hver kun objekter fra en klasse, (eller en enkelt samling af aggregerede objekter). Disse er derfor placeret på klasser i modelkomponenten baseret den idé at de hører samhørighed med de andre elementer i klassen. Operationerne tilføjet i modelkomponenten er: 'Tilføj Behov', 'Fjern Behov', 'Meld borger fraværende' på **Borger**. Opdateringsoperationerne: 'Tilføj Behov' og 'Fjern Behov' repræsenterer hændelserne *behov godkendt* og *behov frakendt*. De påvirker samlingen af aggregerede **behov** på **borger**, ved at gøre **behovet** henholdsvist aktivt eller ikke-aktivt. Det er sådan at man kan stoppe **behovet** fra at generere **opgaver**. Opdateringsoperationen: 'Meld borger fraværende' repræsentere hændelsen *fravær meldt* og tilføjer et **fravær** på en **borger**.

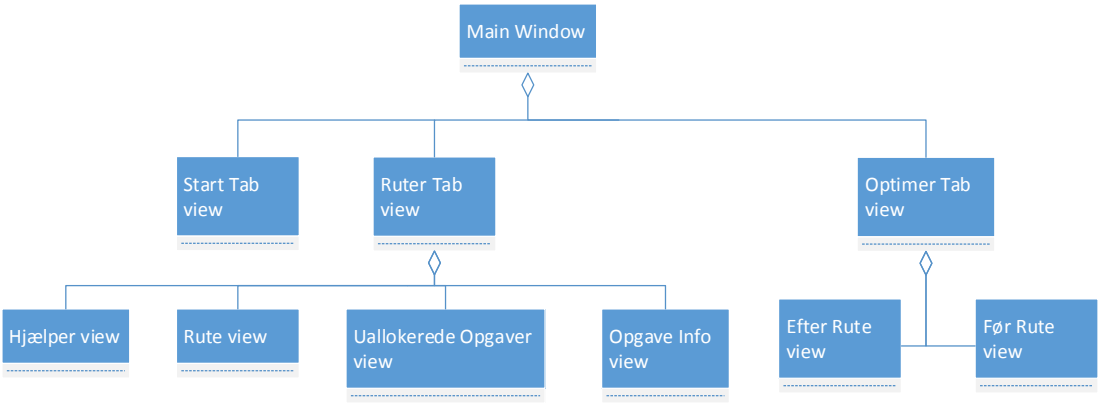
'Tilføj ny adresse' er også en information vi har brug for i det eksterne system, for at repræsentere opdateringsoperationen for hændelsen: *adresse ændret*. Denne er knyttet til **adresseindehaver**, da det den kun skaber **adresse** objekter.



Figur 6.2: Klassediagram for modelkomponenten

6.3.2 Planlæggers View

Planlægger klientens View komponent stemmer overens med de tabs defineret i afsnit 5.3. Hver tab er aggregeret ind i hovedvinduet, som er hvor alt navigation foregår. Se figur 6.3.



Figur 6.3: Planlægger Brugergrænseflade

Hvad der ikke er vist i figur 6.3 er at, hver af disse views observerer deres respektive viewmodeller via observatørmønsteret. Da vi mener dette vil komplicere designet unødigt, viser vi konceptet bag hvordan dette fungerer i afsnit 6.3.3.

6.3.3 Planlæggers ViewModel

Vi vil her give et indblik i vores brug af MVVM, ved at vise sammenhængen for én af de essentielle viewmodeller fra vores system, RuterTabViewModel som vist i figur 6.4.

Viewmodellen RuterTabViewModel er her den overordnede viewmodel som indeholder, en til flere af de fire viewmodeller, TidsperiodeViewModel, OpgaveViewModel, RuteViewModel og HjælperViewModel. Disse fire viewmodeller er lavet for at udstille dele af deres tilsvarende modeller til viewet. Grunden til dette er, at viewet ikke må kende til modellen, og at viewmodellerne beskriver den information, der skal bruges, når en planlægger skal arbejde med **ruter**.

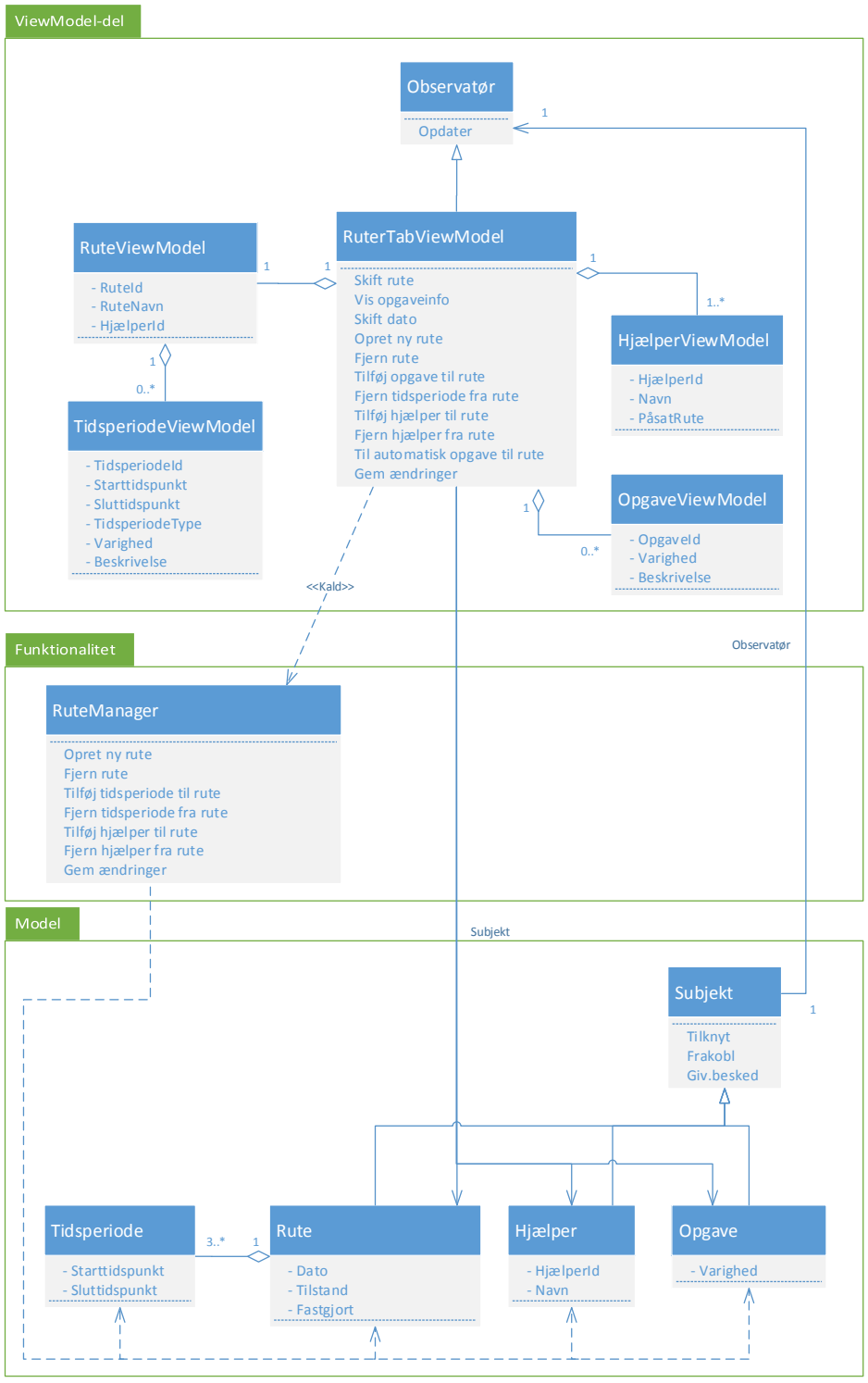
Funktionalitetslaget indeholder her funktioner til at lave ændringer i modellen som den sørger for at udstille til viewmodellen.

Ideen med modelkomponenten i figur 6.4 er, at den skal indeholde ændringer, der endnu ikke er blevet gemt. Dette er hovedsageligt nyttigt i forhold til at ændre planlagte **ruter**, som endnu ikke er udført. Det skyldes, at det vil være muligt at oprette en ny **rute**, som er en kopi af en pågældende **rute** i databasen, og så markere denne kopi som værende et forslag. Derved bliver det muligt kun at lave operationer på denne foreslåede **rute**, i stedet for at ændre på den i databasen. Hvorved det bliver muligt ikke at bryde med adfærden for **rute**, som specificerer, at en **rute** ikke kan gå fra at være planlagt til foreslået. Hvis planlægger accepterer forslaget, bliver det gemt ved at overskrive den tilsvarende planlagte **rute** i databasen.

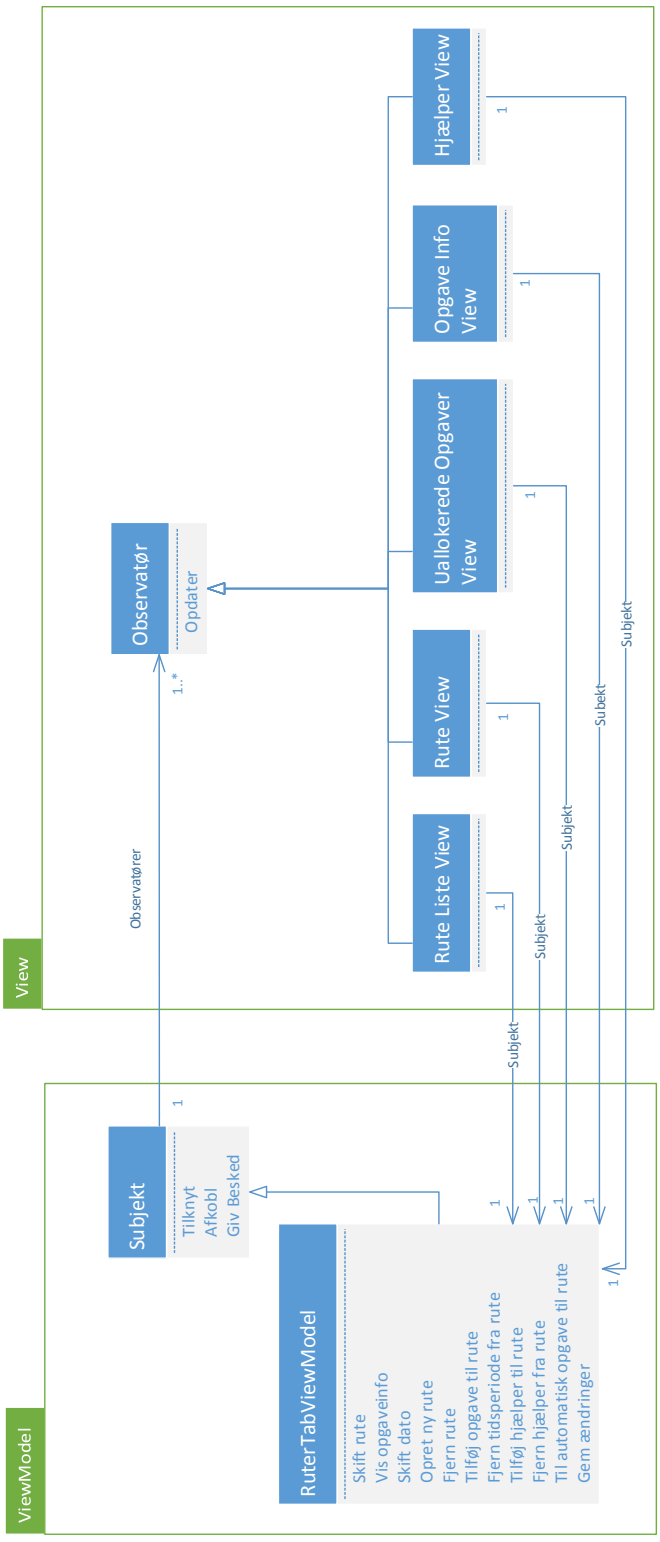
Denne modelkomponent skal desuden kunne indeholde **opgaver**, der er blevet fjernet fra en foreslået **rute**, således at disse kan markeres som værende ikke-planlagt, hvis ændringerne gemmes. Det samme gælder for **påsat rute**; hvis en **hjælper** tilføjes eller fjernes fra en **rute**, skal **påsat rute** ændres, hvis ændringerne gemmes. Vi har dog i figur 6.4 valgt ikke at inkludere **påsat rute**, men det vil dog være denne relation mellem **rute** og **hjælper**, der bliver fjernet eller oprettet, når funktionerne 'Tilføj hjælper til rute' og 'Fjern hjælper fra rute'.

Derudover indgår modellen og viewmodellen i et observer mønster, som gør det muligt for modellen at notificere viewmodellen omkring ændringer, der er foretaget i modellen. Ved ændringer kan viewmodellen reagere ved at hente eventuel ny data fra modellen og efterfølgende opdatere sig selv. Derved sikres det at viewmodellen afspejler ændringer i modellen. Viewmodellen RuterTabViewModel er her den konkrete observatør, og de konkrete subjekter er **rute**, **opgave** og **hjælper**.

Viewet, RuteTabView, der indeholder de views som er vist i figur 6.5, der understøtter den viewmodel i figur 6.4 består også af en række mindre specialiserede views, som hver især står for at visualisere den underliggende viewmodels tilgængelige data. Disse mindre views binder til viewmodellen vha. databinding, som er beskrevet i afsnit 7.5.2. I denne databinding sker der en observatør knytning til de viewmodeller, der bindes til. Dette muliggør at viewet altid bliver underrettet, når de underliggende viewmodels ændres. Gunden til at vi har lavet flere mindre views er for nemt at kunne ændre placeringen af disse, uden at skulle ændre en masse i forhold til viewet.



Figur 6.4: Viewmodel sammenhæng



Figur 6.5: Sammenhæng mellem view og viewmodel

6.3.4 Servers funktionskomponent

Dette afsnit beskriver beregnings-og opdateringsoperationerne realiseret i servers funktionskomponent. Beskrivelsen af aflæsning-og singaleringsoperationerne findes i afsnit 6.2.2 om komponentarkitekturen.

Opdateringsfunktioner

De opdateringsfunktioner i afsnit 5.2.2, som ikke er placeret på klasser i modelkomponenten er her designet til funktionskomponenten. Vi laver en klasse: **RuteManager** til at opbevare alle sammenhørende operationer der håndterer **ruten**. Funktionerne 'Opret rute', 'fjern rute', 'tilføj tidsperiode til rute', 'fjern tidsperiode fra rute', 'tilføj hjælper til rute' og 'fjern hjælper fra rute' er repræsenteret i **RuteManager** med operationer af samme navn. Funktionerne 'accepter forslag' og 'afslå forslag' er modelleret som operationen 'gem rute' i **RuteManager**. Denne operation gemmer **ruten** som står beskrevet i afsnit 6.3.3. Denne klasse er placeret i funktionskomponenten, da nogle af operationerne ikke klart kan placeres i modelkomponenten.

I forbindelse med design af operationer opdagede vi at analysen manglede hændelser og opdateringsfunktioner. Dette står beskrevet i afsnit 9.5.4. Dette har ledt til operationerne: 'Markér **ruter** der ikke må ændres', 'Markér hjælpere der ikke må ændres' og 'Markér opgaver der ikke må ændres'. Disse opdateringsoperationer er nødvendige for at planlæggeren kan lave forslag over den hidtige plan, uden at de **ruter**, **hjælper** eller **opgaver** som planlægger ikke ønskes ændret, bliver ændret. Pga. operationers sammenhørige natur samles de i klassen: **Forslagsadministrator**

For at designe for funktionen: 'indberet forsinkelse' er det nødvendigt at finde operationerne for dens dekomponeringer, samt udlede operationer ud fra figur 5.12 i afsnit 5.2.2. Funktionerne: 'Tilføj tidsperiode til aktuel rute' og 'fjern tidsperiode fra aktuel rute' designes som operationer af samme navn på klassen: **Aktuel RuteManager**. Funktionen: 'Forespørg hjælper om aktuel indsættelse' placeres også som operation på **Aktuel RuteManager**, fordi den er sammenhørende med 'tilføj tidsperiode til aktuel rute'. Den er samhørende da operationen: 'forespørg hjælper om aktuel indsættelse' også indsætter tidsperiode (dog efter forespørgsel).

Der skal være en 'indsæt opgave på bedste rute' operation for at kunne designe funktionaliteten på figur 5.12 i afsnit 5.2.2, hvor valgte **opgave** vurderes vigtig af systemet. Hverken hvad der konstituere den bedste **rute** til **opgave**. indsættelse, og hvilke vurderingskriterier der bruges

vides ikke på dette tidspunkt i udviklingen.

En del af 'indberet forsinkelse' befinder sig i hjælper-klientens funktionskomponent, hvor den efter at have udført funktionaliteten 'vælg aktuel arbejde ud fra vurderingskriterier' og et antal 'fjern tidsperiode fra aktuel rute' venter med at kalde operationer på serveren, indtil der er forbindelse.

Beregningsoperationer

Beregningsfunktionerne i afsnit 5.2.2 er her designet til funktionskomponenten.

Funktionen 'beregn forbindelsestid' repræsenteres som en operation i en klasse: **find forbindelsestid strategi**. Denne klasse designer vi i et strategimønster, sådan at vi kan specialisere til klasser alt efter hvilken strategi vi vælger. Dette er valgt for at tage højde for fleksibiliteten i systemet sådan at vi kan udskifte hvilken korttjeneste eller anden strategi vi bruger til at udregne **forbindelsestid**.

Vi har også brug for beregningsoperationer for forskellige rute-optimeringsformer. Vi skal have muligheden for at kunne smide en **opgave** ind i en allerede eksisterende **rute**. Vi skal også have muligheden for at kaste alle **opgaver** op i luften og danne nye **ruter**. Disse er funktionerne: 'Lav forslag' og 'foreslå arbejdsopgaver på rute'. Operationen der repræsenterer 'lav forslag' kalder vi for 'lav ruter' og placerer i klassen **OptimerTabStrategi**. Operationen der repræsenterer 'foreslå arbejdsopgaver på rute' kalder vi for 'Tildel opgaver til ruter' og placeres i klassen **TravellingSalesmanStrategi**. Disse kan ses på figur 6.6

Vi har valgt at bruge A-stjerne, algoritmen til at finde korte **ruter**. A-stjerne er en udgave af Dijkstras algoritme til rutefinding, der tilføjer en heuristik[7]. A-stjerne blev valgt, da vi identificerede problemet som et rutefindingsproblem af typen 'travelling Salesman'. A-stjerne kræver en heuristik, kanternes vægte og målsætning og systemet er designet sådan at man kan gøre disse dele af A-stjerne udskiftelig. I figur 6.6 kan det ses at vi har lavet en fabrik: **AStjerneFabrik** til at kunne kalde fra, alt efter hvilken slags kombination af A-stjerne man ønsker. Dermed kan et **OptimerTabStrategi** objekt (der bedes om at udføre strategien, hvor alle opgaver bliver kastet op i luften) eller et **TravellingSalesmanStrategi** objekt (der bedes om at udføre strategien, hvor alle opgaver besøges i en graf) bede om den **AStjerne** som de skal bruge. De stiplede pile betyder her afhængigheder, altså at klasserne har operationer der har instanser af de klasser pilene peger på.

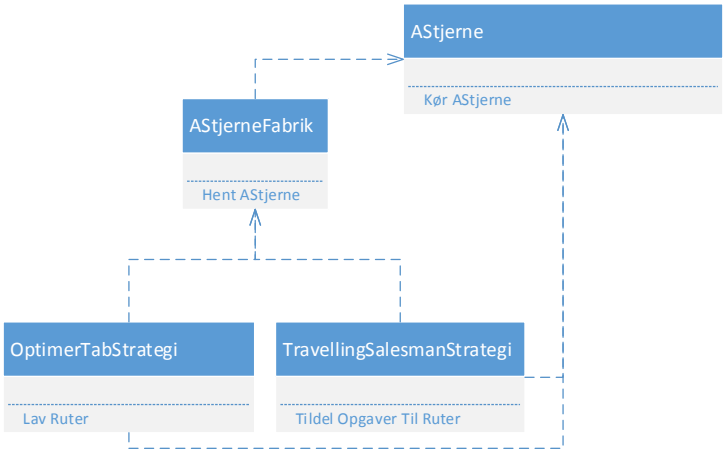
figur 6.7 viser hvordan **AStjerneFabrik** er designet. **AStjerne** har en **Konfiguration** der indsam-

ler sammensætninger, for at påvirke algoritmens opførsel. Sammensætningen **TSPMålsætter** og **TSPHeuristikUdregner**, implementerer en løsning på et travelling salesman problem, hvor alle knuder i den indsatte graf skal besøges. Det er **TravellingSalesmanStrategi** fra figur 6.6 der beder **AStjerneFabrik** om denne version.

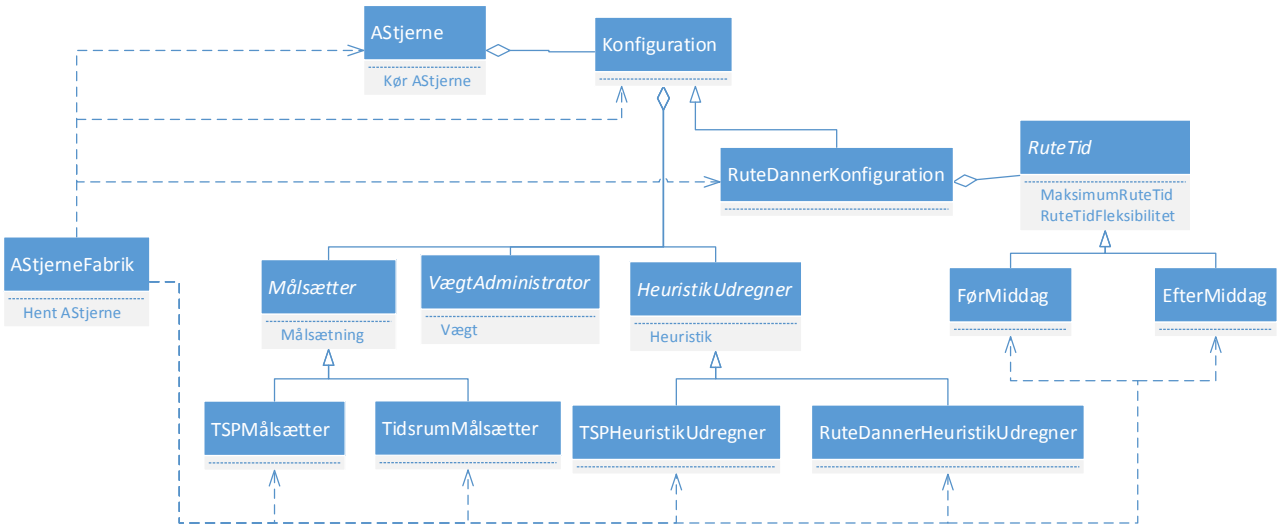
Sammensætningen **TidsrumMålsætter**, **RuteDannerHeuristikUdregner**, samt en RuteTid implementerer en løsning på at finde en kort **rute**, der er indenfor et bestemt tidsrum. Det er **OptimerTabStrategi** fra figur 6.6 der beder **AStjerneFabrik** om denne version.

Hvis man ville tilføje en ny indstilling uden en **Konfiguration** klasse ville man skulle ind og ændre i **AStjerne**. Nu kræver det bare at nedarve fra Konfiguration, og så sammensætte den nye konfiguration i AStjerneFabrik. Målsætter, VægtAdministrator og HeuristikUdregner skal med i alle AStjerne konfigurationer, og derfor er de i Konfiguration superklassen.

Hver indstilling (**TSPMålsætter**, **TidsrumMålsætter**, **TSPHeuristikUdregner**, **RuteDannerHeuristikUdregner** osv.) er indsat i en strategimønster, sådan at f.eks. et **Konfiguration** objekts Målsætter kan have forskellige **Målsætter** strategier. Dette er valgt, da det er mere modtageligt for ændringer. Ved en ny **AStjerne** konfiguration, vil der altid være en **Målsætter**, **VægtAdministrator** og **HeuristikUdregner**. Derfor skal de nye indstillinger nedarve derfra, og i **AStjerneFabrik** skal konfigurationen sammensættes.



Figur 6.6: OptimerTabStrategi og **TravellingSalesmanStrategi** brug af **AStjerneFabrik** til at få et **AStjerne**-objekt



Figur 6.7: Designet af AStjerneFabrik

TSPMålsætter & TSPHeuristikUdregner

TSPMålsætters Målsætning returnerer *true*, når alle knuderne i grafen er besøgt, udover startKnuden som er hjemmeplejens Afdeling, som man både skal starte ved og ende ved.

TSPHeuristikUdregner estimerer den resterende tid det tager at rejse de ubesøgte knuder (herunder også Afdeling). Dette gør den ved at udregne vægten af et minimum udspændingstræ, samt vægten fra sti.Last() til den nærmeste naboknude. Dette er en admissible heuristik, da estimatet aldrig vil overstige vægten af den korteste sti, hvilket vil sige at i dette tilfælde vil A-stjerne altid finde en optimal løsning.

TidsrumMålsætter & RuteDannerHeuristikUdregner

TidsrumMålsætters Målsætning returnere *true*, når stiens samlede sum (inklusiv opgavernes varighed) tager et bestemt tidsrum. Dette tidsrum er specificeret i RuteTid attributterne: MaksimumRuteTid og RuteTidFleksibilitet. Dette er valgt fordi at det skal være muligt at danne **ruter** over alle adresserne, som ikke må overskride arbejdstiden.

I RuteDannerHeuristikUdregner havde vi problemer med at finde et godt heuristik estimat. Det vi havde mest succes med var at summen af antal ubesøgte kanter med distancen til den nærmeste ubesøgte knude. Dette vil i langt de fleste tilfælde give et estimat der ligger under den reelle rutedistance, men det er en heuristik der øger algoritmens kørselstid mere end hvis den ikke var der.

7 Implementering

Dette kapitel omhandler hvorledes vi har implementeret dele af systemet. Der vil først redegøres for afgrænsinger vi har foretaget fra design til implementeringen herunder argumenter for disse. Dernæst fremgår overvejelser omkring implementering af A-stjerne algoritmen i forhold til effektivisering af **ruterne**. Derefter beskrives anvendelsen af Google Maps vejfindingsservicen samt vores anvendelse af Object relational mapping til at realisere persistent data i systemet, herunder en kort forklaring af vores brug af Entity Framework. Til sidst redegøres der for realiseringen af brugergrænsefladen, herunder markup-sprog og databinding i forhold til Windows Presentation Foundation og MVVM.

7.1 Afgrænsninger

Med det objektorienterede analyse & design på plads, vil vi implementere systemet. Vi har afgrænset os fra flere ting fra både modellen beskrevet i analysen og arkitekturen beskrevet i designet. Årsagen til disse afgrænsninger har hovedsageligt været den tidshorisont, der har været til dette projekt.

Vi valgte at implementere den funktionalitet, der dækker de mest nødvendige krav identificeret i analysen, og afgrænse os fra resten. En detaljeret liste af afgrænsninger og argumenter for dem findes i dette afsnit. Ligeledes er mange af de afgrænsninger, vi har foretaget, baseret på vores gradvise udvikling af A-stjerne algoritmen til effektivisering af **ruter**. På denne måde blev dele af systemet implementeret i takt med, at vi udvidede effektiviseringsalgoritmen til at kunne håndtere flere variabler.

Målet er at alle dele af system skal implementeres, og der er taget hensyn til dette, men den egentlige opfyldelse af dette mål, overlades til fremtidig implementering systemet som beskrevet i afsnit 9.5.4.

7.1.1 Komponenter

Vi har valgt at afgrænse os fra hele hjælper klienten. Dette vil sige at vi ikke har implementeret hjælper-klientens komponenter. Afgrænsningen sker på baggrund af vores valg om, at det ville være meget tidskrævende at implementere akutplanlægningsdelen.

Vi har ligeledes afgrænset os fra serverens systemgrænseflade, der skulle bruges til at kommunikerer med eksterne systemer. Den skulle for eksempel benyttes til at hente informationer fra Care systemet, vedrørende informationer leveret fra visitator. Systemgrænsefladen er ikke implementeret i den nuværende tilstand af systemet.

Planlæggers ViewModel komponent skulle oprindeligt have adgang til serverens *Data Access Layer*, da *Brugergrænsefladen* har fanebladet notifikationer. Denne skulle kunne vise notifikationer på ændringer af modellen. Fanebladet er implementeret, men kan ikke reagere på ændringer sket i modellen. Planlæggers ViewModel er altså ikke fuldt ud implementeret.

Vi har også afgrænset os fra at lave funktionaliteten til Optimér fanebladet. Denne skulle have haft funktionaliteten til at kunne kaste alle **ruter** og **opgaver** op i luften for en bestemt dag, for derefter at skabe de mest optimale **ruter**, mht. parametre som køretid. Denne funktionalitet kunne også have bidraget til løsning på problemformuleringen. Det blev dog vurderet at denne funktionalitet ville være for tidskrævende at implementere.

7.1.2 Klasser

Følgende klasser har vi ikke implementeret i den nuværende tilstand af systemet:

- Hjælper
- Præference
- Grad
- Påsat rute
- Adresse
- Fravær

Vi implementerede ikke **hjælper** det nuværende system. Årsagen var, at vi først afklarede den mere komplekse planlægningsfunktionalitet med **ruter**, **opgaver** og **tidsperioder**. Herefter ville det være mindre komplekst at introducere **hjælpere** i implementeringen. Vi har afgrænset os fra **præference** klassen, da denne ikke ville give mening uden **hjælper** klassen. En **præference** skulle være implementeret som en klasse der aggregerer op i **borger**, da hændelsen

præferencegrad ændret er en iterativ hændelse på **borger**.

Aktuelle **ruter** er ikke implementeret i udarbejdelsen programmet, da hele hjælperkomponenten blev afgrænset fra.

7.1.3 Funktioner

Følgende funktioner har vi ikke implementeret i det nuværende system:

- Opret rute
- Fjern rute
- Tilføj hjælper til rute
- Fjern hjælper fra rute
- Indberet forsinkelse
- Notificer planlægger om aktuel ændring
- Se planlægger notifikationer
- Se hjælper notifikationer
- Se hjælper
- Aflæs forslagsbegrænsninger
- Lav forslag

De nævnte komponenter, klasser og funktioner er ikke implementeret, på grund af tidshorisonten for projektet. Som en del af designet, er det dog stadig meningen at de skal implementeres i fremtidig udvikling af systemet.

7.1.4 Forskelle fra design til implementering

Her beskrives den adfærd fra afsnit 5.1.4, der i programmet afviger væsentligt fra tilstandsdiagrammerne.

figur 5.3 viser tilstandsdiagrammet for **rute**, som har tre tilstande: 'planlægning', 'planlagt' og 'aktuel'. Begrænsningen med at en planlagt **rute** ikke må ændres, samt at **ruter** kun kan gå fra 'planlægning' til 'planlagt' til 'aktuel' og ikke omvendt, er implementeret. **Ruter** ændres dog aldrig fra 'planlægning' tilstanden, da vi har afgrænset os fra aktuelle ruter i den nuværende implementering.

Ved implementationen af modellen har vi taget nogle valg, med hensyn til den tekniske platform. Der er blevet taget udgangspunkt i klassediagrammet fra designet. **Borger**, **behov** og

opgave er blevet opbygget med en forholdsvis høj indbyrdes kobling op igennem aggregeringerne, da det ikke var ønsket, at en klasse kunne oprettes uden en reference til en klasse den aggregerede ind i. Dette har resulteret i en tovejs binding imellem **borger**-**behov** og **behov**-**opgave**.

En lignende opbygning var tiltænkt til **tidsperioder** i forhold til **rute**, således at **tidsperioder** ikke kan eksistere udenfor en **rute**. Implementeringen af dette blev nedprioriteret pga. tidsmæssige begrænsninger. Specialiseringen *arbejde* er blevet implementeret således at den er påkrævet en reference til en **opgave**. **Opgaven** har ikke nogen viden om hvilken **tidsperiode** den er knyttet til, men indeholder en boolsk attribut der udtrykker hvorvidt den er placeret, og står til udførelse.

Grundet valget om ruteudregning på baggrund af **opgaver** og ikke **borgere**, var det fordelagtigt at have en generalisering der var fælles for **afdeling** og **opgave**. Dette blev til *IKnude* interfacet, som definerer reglerne for at kunne vurdere besøgets vægtning. Ved udvidelse af planlægningspræcisionen, vil dette interface blive den centrale kobling.

Der blev i afsnit 6.3.3 beskrevet, hvordan en opdeling af viewmodel, funktionalitet og model kunne være udarbejdet for den viewmodel der har med ruteplanlægningen at gøre. I vores implementering har vi dog brudt med dette, da vi her valgte at foreslåede **ruter** ikke skulle være en del af modellen, men derimod være en pågældende **rute** der var under ændring i viewmodellen.

7.2 A-stjerne

Herpå følger implementeringen af vores løsning til den automatiske rutegenerering. For fastholde en generisk tilgang i algoritmen, har vi valgt at modellere **rute** som en 'sti' heri.

En sti beskrives i algoritmen i listing 7.1 med en sekvensnotation: $< p_0, p_1, ..., p_n >$. Enhver sti har metoderne First() og Last(), som hhv. giver det første element i stien og det sidste element i stien. Enhver sti har også attributten: samletVægt, der er 0 ved instantiering, men assignes til summen af vægten imellem alle knuderne i stien, under algoritmens forløb.

A-stjerne gør brug af en min-prioritetskø, hvor alle elementer skal have attributten 'key'. Prioritetskøen har metoden: ExtractMin() som fjerner og returnere det element med lavest key.

Linje 2-3 initierer algoritmen, hvilket inkluderer at 'frontier' tildeles startKnude. Det forventes at være en **afdeling**, men dette er ikke en algoritmisk nøvendighed. StartKnude markerer blot hvor man ønsker at starte og derfor slutte sin sti, hvilket i hjemmeplejens situation vil være **afdelingen**.

linje 4 undersøger hvorvidt frontier er tom. Hvis dette sker, returneres NULL i linje 16, da ingen permutation af 'sti' opfylder målsætningen.

Linje 6-7 tjekker om målsætningen mødes. Denne målsætningen er defineret i argumentet 'konfiguration'. Hvis målsætningen mødes af en sti, returneres denne sti.

Linje 8-15 udvider frontier med en ny sti, for hver nabo til den sidste knude i den valgte sti, som ikke allerede er inkluderet i stien. Dette gøres ved at der i linje 10-12 laves en ny sti, der udvider den gamle med den pågældende nabo, og i linje 13-14 finder dens key, udfra konfigurationens heuristik, kombineret med de inkluderede kanters vægt. I linje 15 tilføjes denne nye sti, til frontier, så den kan indgå i fremtidige gennemløb af while-løkken i linje 4.

```
1  AStjerne(graf, startKnude, konfiguration)
2      Lad frontier være en ny min-prioritetskø
3      frontier.Add(<startknude>)
4      while (frontier != ∅)
5          sti = frontier.ExtractMin()
6          if(konfiguration.Målsætter.Målsætning(graf, sti,
               konfiguration))
7              return sti
8          for hver naboKnude til sti.Last()
9              if(sti.contains(naboKnude) == false OR naboKnude
                   == startKnude)
10                 lad nySti være en ny sti
11                 nySti.AddRange(sti)
12                 nySti.Add(naboKnude)
13                 nySti.samletVægt = sti.samletVægt +
                       konfiguration.vægtObj.Vægt(sti.Last(),
                       naboKnude, konfiguration)
14                 nySti.key = nySti.samletVægt + konfiguration.
                       heuristikObj.Heuristik(naboKnude,
                       startKnude, graf, nySti, konfiguration)
```

```
15                  frontier.Insert(nySti)
16      return NULL
```

Listing 7.1: AStjerne algoritme

7.3 Google maps API

For at finde transporttiden mellem adresser, valgte vi at anvende Google Maps, da denne service dækker en stor mængde af adresser i Danmark, og udstiller en API til at gøre brug af. Desværre udstilles denne API *ikke* direkte til .NET og derved C#. Der er dog en uofficiel API[8] skrevet af en uafhængig udvikler. Denne API indfanger Googles egen API og adapter den således .NET udviklere kan tilgå den via et C# library.

Google fastholder dog et sæt af begrænsninger vedrørende brug af deres service[9]. Disse begrænsninger har konsekvenser for antallet af adresser man kan forespørge på. Således kan der ikke forespørges frit efter behov, hvilket resulterer i problemer f.eks. når der er mange **borgere** i hjemmeplejen.

Dette er en af grundene til at implementere programmet med persistent datalagring, da det ikke vil være muligt at forespørge API'en efterhånden som antallet af **borgere** stiger. I mod betaling er det dog muligt for at få adgang til den udvidede version af Google Maps API, hvorledes begrænsningerne formindskes markant. Vi så dog ikke årsag til dette og valgte i stedet at fastholde et mindre dataset for overblikkets skyld.

7.4 Relational database

Dette afsnit omhandler vores brug af relational database til lagring af persistent data, herunder vores brug af Entity Framework (EF). Der vil indgå en overordnet redegørelse for principperne bag EF og funktionalitet vi har gjort brug af. Derudover beskrives processerne for hvordan der interageres mellem programmet og databasen. Der vil ikke blive gået i dybden med de underliggende tekniske detaljer, da dette er uden for projektets rammer.

EF er .NET's implementation af Object-relational mapping (ORM) teknikken, der fungerer

som bindeled mellem objektorienterede sprog og relationelle databaser. EF fungerer med en række forskellige databaseteknologier, herunder Structured Query Language (SQL), som er anvendt i dette projekt.

En relationel database indeholder tabeller med rækker og kolonner. De grundlæggende principper bag ORM indfanger klasser i tabeller i databasen, således at én række har relation til ét objekt, og klassernes attributter findes som kolonner i tabellerne. På den måde kan der konverteres mellem lagret data og objekter i hukommelsen.

Der kan derfor drages mange paralleller mellem objekt- og databaseorienterede strukturer. Der er dog stadig forskelle, som er værd at bemærke. I objektorientering opbygges forhold og identitet ved reference - i relationelle databaser gøres dette ved primærnøgler.

Unikke primærnøgler sikrer at der ikke findes to ens indgange i databasen. Typisk er primærnøgler integreret, hvilket også er tilfældet i dette projekt. I projektet ses primærnøglerne, der anvendes af EF, direkte i klasserne som forskellige attributter med 'Id' i navnet. Ligeledes findes der *fremmednøgler* i klasserne og tabellerne. En fremmednøgle refererer en primærnøgle i en anden tabel, hvor der på den måde skabes et forhold mellem to objekter. Dette betyder, at associeringer mellem klasser i objektorientering, repræsenteres i databasen ved, at en række indeholder en fremmednøgle, der er associerede rækkes primærnøgle.

Derudover kan simple datatyper, som f.eks. integers, ikke som standard være *null* (i C#), men det kan de i relationelle databaser. Af denne grund vil *nullable* operatoren '?' optræde i visse klasser i programmet, hvor der har været behov for at en datatype skulle kunne være null.

7.4.1 Konfiguration af associationer i Entity Framework

Forbindelse mellem program og database kan foregå hvorend, der er behov for det. Dette skaber dog høj kobling mellem databasen og alle de stedet i programmet, hvor der forbindes til den. EF indkapsler derfor koblingen til databasen i en klasse, der fungerer som et datatilgangs-lag; en såkaldt databasekontekst.

Databasekonteksten er en objektorienteret abstraktion af databasen. Den udstiller kollektioner, der repræsenterer tabeller i databasen. At tilgå disse kollektioner svarer derfor til at tilgå tabellerne. Når kollektionerne tilgås, instantieres de med objekter, der laves ud fra rækkerne i databasen der er konfigureret til konteksten.

Databasekonteksten står ligeledes for at konfigurere associationerne mellem objekternes data repræsenteret i tabellerne. Dette gøres i EF Fluent API, hvor forholdende indstilles via C# lambda udtryk.

```
1 modelBuilder.Entity<Rute>()
2               .HasKey(r => r.RuteId)
3               .HasMany(r => r.Tidsperioder)
4               .WithRequired(t => t.Rute);
```

Listing 7.2: Konfiguration af et-til-mange forhold

I lambda udtrykket listing 7.2 ses et eksempel på konfiguration af et binært et-til-mange forhold mellem **rute** og **tidsperiode**. Forholdet skal læses således:

Linje 2 betyder at en '**rute**' har en primærnøgle 'RuteId'.

Linje 3 betyder at én '**rute**' har mange '**tidsperioder**'.

Linje 4 betyder at '**tidsperioderne**' på **ruten** er påkrævet at være associeret med én **Rute**.

På denne måde, konfigurereres der associationer mellem alle klasserne i modellen ved et lamdba udtryk for hver association.

7.5 Brugergrænseflade

Da vi i dette projekt arbejdede med C# og .NET's framework, havde vi valgmulighederne for at udarbejde brugergrænsefladen i enten *Windows Forms* eller *Windows Presentation Foundation* (WPF).

Vi ville forsøge at skabe en løst koblet brugergrænseflade, så vi endte med at vælge WPF, da det understøtter databinding. Vi har realiseret dette ved at bruge MVVM designmønstret, som er beskrevet i afsnit 6.2.1, da det er lavet specifikt til at bruge databinding og adskille brugergrænsefladen fra det bagvedliggende data[10]. Dette gør det muligt at udskifte brugergrænsefladen, uden at ændre i andre dele af programmet. Derved er mønstret ideelt at benytte, for systemer der skal køre flere forskellige platforme, da det herved kun er brugergrænsefladen der skal ændres fra platform til platform.

Vi vil i det følgende kigge på XAML og Databinding som relevante teknologier inden for WPF.

7.5.1 XAML

Brugergrænseflader bliver i WPF opbygget ved at bruge opmærkningssproget XAML, Extensible Application Markup Language. Dette simplificere måden hvorpå brugergrænsefladen defineres. XAML er en simpel træstruktureret måde hvorpå man kan opbygge GUI elementer.

XAML er en abstraktion over underliggende C# kode. Alt XAML kan altså beskrives med C#, men kan med fordel skrives via XAML i stedet. Vi har benyttet XAML i forbindelse med databinding og MVVM, da dette danner grobund for at skabe en løst koblet brugergrænseflade.

7.5.2 Databinding

Databinding giver en nem og konsistent måde for brugergrænsefladen at præsentere og interagere med data. Elementer kan bindes til data i form af mange forskellige data bindings.

Databinding i WPF har mange fordele over traditionelle modeller. WPF databinding har for eksempel in bred vifte af properties der kan anvendes ved databinding, samt at det giver en klar separation af GUI og den underliggende logik.

For at XAML skal kunne databinde til en property, skal klassen, der indeholder propertien, implementere INotifyPropertyChanged interfacet, og derudover specificere et PropertyChanged event. Dette event er hvad der skal affyres, når den pågældende property ændres.

```
1  public class TidsperiodeViewModel : IVarighed,
       INotifyPropertyChanged
2  {
3      ...
4      public int Varighed
5      {
6          get { return Tidsperiode.Varighed; }
7          set {
8                  if (Tidsperiode is Ledig)
9                  {
10                     Tidsperiode.Varighed = value;
```

```
11                      NotifyPropertyChanged();
12                  }
13              }
14      }
15      ...
16      public event PropertyChangedEventHandler PropertyChanged;
17
18      private void NotifyPropertyChanged([CallerMemberName]
            String propertyName = "")
19      {
20          if (PropertyChanged != null)
21          {
22              PropertyChanged(this, new
                    PropertyChangedEventArgs(propertyName));
23          }
24      }
25  }
```

Listing 7.3: Tidsperiodes ViewModel

I koden i listing 7.3, bruges en metode, NotifyPropertyChanged, til at affyre eventet PropertyChanged for de properties, der skal bindes til. Attributten [CallerMemberName] sætter parameterstrengen til navnet på den property, som kaldte den, hvilket gør at metoden bare kan kaldes som NotifyPropertyChanged() i propertien.

I View komponenten vil man via XAML derefter binde til `Varighed` vha. syntaksen i listing 7.4.

```
1           <Grid>
2               ...
3               <TextBlock Margin="2" Text="{Binding Varighed}" Grid.
                    Column="2"/>
4           </Grid>
```

Listing 7.4: Eksempel på binding

For at denne binding kan ske, skal **TextBlock**s *DataContext* sættes til et objekt af **TidsperiodeViewModel**, som indeholder en property kaldet *Varighed*. DataContext arves automatisk

fra det ydre element, så **TextBox** arver altså i listing 7.4 *DataContext* fra **Grid**, som igen arver DataContext fra det element dén er indeholdt i.

8 Kvalitetssikring

I dette kapitel beskriver vi, hvad vi har gjort for at sikre, at funktionalitet og brugergrænseflade lever op til de krav vi har sat. Vi lavede tidligt i forløbet en prototype, for at undersøge interessenternes krav og forventninger til systemet. Under udviklingen af programmet begyndte vi at lave unit tests, for at sikre at funktionaliteten virkede som forventet. Til sidst udførte vi en usability test for at evaluere brugergrænsefladen.

8.1 Evaluering af prototype

I slutningen af 1. iteration blev der aftalt et 30 minutters møde med Attendo A/S, hvor deres planlægger skulle afprøve en interaktiv prototype, som vi havde udviklet. Prototypen kan ses i bilag 12.1. Denne prototype blev udviklet ud fra vores forestilling om, hvad planlægger havde af arbejdsopgaver, og hvilke funktioner der kunne understøtte disse.

Formålet med prototypen var at undersøge yderligere krav til systemet, samt at evaluere de krav vi havde i forvejen. Denne test blev desuden brugt til at undersøge, hvorvidt den valgte brugergrænseflade ville være forståelig for testpersonen.

8.1.1 Metode

For at sikre at testpersonen gennemgik de aspekter af systemet som var ønsket, var der blevet udarbejdet en række testcases, disse kan findes i bilag 12.2. Da prototypen var et stykke interaktivt software var det derfor oplagt at dokumentere test sessionen, dette blev gjort ved at der var installeret software til at optage hvad der skete på skærmen, og derudover også optage testpersonen via webcam i computeren. Ved at gøre brug af dette software var det derved muligt at gennemse testen efterfølgende, for at gennemføre en dybere analyse af testen.

For at kunne dokumentere testen var 3 af gruppens medlemmer med ude til mødet, disse havde forskellige opgaver i forhold til at dokumentere og styre testen. Disse tre opgaver var følgende:

- Test monitor

- Observatør
- Data logger

Test monitorens opgaver var at sørge for at testpersonen forstod de testcases der blevet givet, samt også sikre sig at testpersonen så vidt muligt tænkte højt under udførelsen af disse. Observatørens opgave var at observere testpersonens interaktion med prototypen, i form af at spotte eventuelle steder hvor testpersonen havde besværligheder. Data loggerens opgave var at notere de ting som testpersonen tænkte højt i forhold til besværligheder ved systemet.

8.1.2 Resultater

Efter at have analyseret videoen samt noterne til testen, blev forskellige styrker og svagheder ved prototypen identificeret. Vi kunne ud fra videoen se at der var enkelte punkter, hvor testpersonens mentale model ikke stemte overens med vores forestilling af systemet. Dette kunne blandt andet ses, ved at testpersonen forsøgte at trække et uallokeret besøg over på en **rute**, da vedkommende blev bedt om at tilføje den til **ruten**. Vi har derfor undersøgt muligheden for at implementere træk-og-slip, og kunne konkludere at en realisering af træk-og-slip var mulig, men kun på væsentlig bekostning af andre dele af systemet. Derfor valgte vi at undlade at implementere træk-og-slip i brugergrænsefladen, så vi kunne fokusere på udvikling af de andre funktioner i systemet.

Da testpersonen skulle fjerne et besøg fra en **rute**, forsøgte vedkommende at trykke på det pågældende besøg, for derefter at trykke på knappen "Fjern Borger", selvom det kun var muligt i omvendt rækkefølge. Det blev også kommenteret at titlen "Uallokerede besøg"muligvis burde ændres til noget mere sigende. Det kunne også ses, ved at testpersonen ikke gjorde brug af den støttende funktionalitet, som vi havde lavet i *Optimér ruter*.

Evalueringen af prototypen påviste, at prototypen var anvendelig af brugeren. Dog havde brugeren problemer med at anvende brugergrænsefladen for optimering. Brugeren var især i tvivl om rækkefølgen af instruktioner der skulle udføres, for at få det ønskede resultat. Dette pegede på, at optimeringsgrænsefladen enten var for avanceret eller forvirrende designet. Dette medførte at denne del af brugergrænsefladen blev tænkt om for at gøres det mere forståeligt for brugeren at anvende.

8.2 Usability test

Denne sektion vil beskrive en evaluering af usability i vores brugergrænseflade vi har lavet med en planlægger i hjemmeplejen.

8.2.1 Usability

Usability beskriver kvaliteten af et interaktionsdesign. Der findes mange definitioner på usability. Rubin & Chisnell definere et program med høj usability som: Brugbart, nemt at lære, nem at bruge, effektivt at bruge, tilfredsstillende at bruge og har funktionalitet der er ønsket af brugerne [11]. Det er denne definition vi benytter os af. Et usability problem er når en bruger laver fejl, føler sig frustreret eller bliver forsinket eller forhindret i at udføre en opgave.

8.2.2 Brugerbaseret evaluering

Vi har valgt en brugerbaseret evaluering til at evaluere systenets usability. Dette er valgt, fordi det identificere problemerne mere præcist til forskel fra heuristisk inspektion. I en brugerbaseret evaluering er det repræsentative brugere som interagere med systemet, med en test monitor der observere. Brugeren bliver bedt om at tænke højt imens de udføre opgaver, som tager udgangspunkt i det arbejde som er tilsigtet at de skal kunne udføre i systemet. Resultatet af en brugerbaseret evaluering er en rankeret liste af usability problemer.

8.2.3 Kategorisering af usability problemer

Usability problemer kan rankeres efter hvor alvorlige de er. Denne rankering har niveauerne: Kosmetiske problemer, seriøse problemer og kritiske problemer. Disse kan findes i en brugerbaseret evaluering, ved at give brugeren et sæt af opgaver. Et kosmetisk problem giver typisk en forsinkelse på under 1 minut eller der er en lav irritation. Et seriøst problem tager mere end et minut, der kan være middel irritation eller en signifikant forskel på forventning og hvad der faktisk sker. Et kritisk problem er hvor brugeren slet ikke løser opgaven, er stærkt irriteret eller der er en kritisk forskel imellem forventning og hvad der faktisk sker.

Ud fra de opgaver som vi gav planlægger er her en rankering af de usability problemer vi fandt:

Problem 1 - Seriøs

Planlægger har svært ved at identificere den borger der skal placeres på en rute. Dette tager ca. halvandet minut, efter en smule hjælp.

Problem 2 - Kosmetisk

Planlægger har en forventning om træk-og-slip funktionalitet fra løse opgaver over til ruten.

Problem 3 - Kosmetisk

Planlægger udtrykker mild irritation over at skulle trykke på en opgave, for at se information om en borger

Problem 4 - Kosmetisk

Planlægger trykker flere gange på Gem Rute tasten. Der er en forventning om at systemet skulle respondere når der gemmes.

Problem 5 - Kosmetisk

Planlægger klikker igennem flere opgaver på en rute for at identificere en opgave til er bestemt borger.

Problem 6 - Kosmetisk

Planlægger trykker på 'Ny rute', uden at have skrevet et rutenavn. Dette er fordi at planlægger forventer et nyt vindue hvor dette sker.

Problem 7 - Seriøs

Planlægger tager over 1 minut om at sætte en opgave på om eftermiddagen, da planlægger har svært ved at se at der skal scrolles ned for at finde 'pause' tabben hvor opgaven skal sættes efter.

Problem 8 - Seriøs

Planlægger har svært ved at se hvilke opgaver der er placeret automatisk på ruten.

8.2.4 Overvejelser over usability-problemer

Det viste det sig at vores idé om kun at vise det mest nødvendige information under ruteplanlægningen, ikke var nok til at planlæggeren kunne foretage planlægningen. Brugeren efterspurgte yderligere information, som var gemt væk i ToolTip vinduet (se afsnit 5.3). Dette ledte til, at det var nødvendigt at have mere information tilgængelig, i brugergrænsefladen, omkring **borgere** og hvilke **opgaver** der skulle løses.

Under brugerens afprøvning af den automatiske planlægning, blev det hurtigt klart at brugeren ingen mulighed havde, for at se hvilke ændringer den automatiske planlægningsfunktion have foretaget. Dette betød at planlæggeren ingen mulighed havde, for at kontrollere om den foreslåede **rute** var acceptabel. Derfor valgte vi i brugergrænsefladen, at symbolisere ændringer i **ruterne** visuelt.

Brugeren havde også en ide om at træk-og-slip funktionalitet eksisterede i programmet. Dette blev allerede fundet under evalueringen af første prototype, men vi valgte at afgrænse os fra funktionaliteten, da denne ville tage for mange resourcer at implementerer i programmet. Flere reflektioner og informationer om brugergrænsefladens understøttlese af træk-og-slip funktionalitet, kan findes i afsnit 9.5.3.

Testen viste at planlægger havde problemer med at identificere scroll funktionen på en **rute**, hvilket resulterede i en seriøs systemfejl. Denne fejl vil vi dog ikke tage hånd om, i dette projekt, men i stedet vente med til en anden iteration.

8.3 Unit test

Vi har foretaget en række unit tests ved hjælp af det indbyggede test framework i Visual Studio Ultimate 2013. Hver metode der skal testes har en eller flere testmetoder, som er navngivet på formatet "*MetodeNavn*_When*Input*_Should*ForventetOutput*".

"Arrange-Act-Assert"mønstret er blevet anvendt for at skabe letlæselige og overskuelige tests. Det går ud på at opdele unit tests i tre dele jvf. *Beginning Windows 8 Application Development: XAML Edition*[12], som opsummeres her:

- I 'Arrange' delen opstilles de objekter, der er nødvendige i testen, i den ønskede tilstand.

- I 'Act' delen igangsættes den funktionalitet vi vil teste.
- I 'Assert' delen undersøges hvorvidt funktionaliteten påvirkede systemet som forventet.

Testene har til formål at sikre korrektheden af programmets kernefunktionalitet. *AStjerne*-klassen indeholder den mest komplekse funktionalitet i programmet (se afsnit 5.2.2). Vi har på baggrund af det, og projektets tidsramme, valgt kun at teste denne klasse.

8.3.1 A-stjerne

AStjerne-klassen indeholder én public metode *KørAStjerne*, som er blevet testet med input og forventede output som vist i tabel 8.1.

Input	**Forventet output**
En graf der er tom	AfdelingIkkeFundetException
En graf der har opgaver men ingen afdeling	AfdelingIkkeFundetException
En graf der kun har en afdeling	SamletVægt = 0
En graf der er struktureret som et træ	Null
En graf der har en passende mængde knuder, men ingen kanter mellem dem	Null
En graf der er opdelt i to med en passende mængde knuder	Null
En graf der er en klike med tre knuder	SamletVægt stemmer overens med den korteste rutes vægt
En graf der er en klike med fem knuder	SamletVægt stemmer overens med den korteste rutes vægt

Tabel 8.1: Testinput og forventede output til KørAStjerne

Testmetoderne bruger attributterne [TestClass], [TestMethod] og [ExpectedException(...)] samt *Assert* klassen fra Visual Studio test frameworket. En af testene, *En graf der er tom*, samt den anvendte opbygning af unit tests er vist i figur 8.1.

Vi har desuden sikret, at alt koden bliver dækket af testene, som det fremgår af skærmbilledet i figur 8.2. Dette blev gjort også gjort vha. test frameworket i Visual Studio.

Testene er blevet kørt løbende, og blev i sidste ende gennemført uden fejl.

```
9  namespace HjemmeplejenTests
10 {
11     [TestClass]
       0 references
12     public class AStjerneTest
13     {
14         private AStjerneFabrik _fabrik = new AStjerneFabrik();
15         private TestData _testData = new TestData();
16
17         [TestMethod]
18         [ExpectedException(typeof(AfdelingIkkeFundetException), "En tom graf blev fejlagtigt godtaget.")]
           0 references
19         public void KørAStjerne_WhenEmptyGraf_ShouldThrowException()
20         {
21             // Arrange
22             AStjerne tspStrategi = _fabrik.HentAStjerne(AstjerneMode.TravelingSalesmanMode);
23             UndirectedGraph<IKnude, Kant<IKnude>> graf = new UndirectedGraph<IKnude, Kant<IKnude>>();
24
25             // Act
26             Sti<IKnude> actual = tspStrategi.KørAStjerne(graf);
27
28             // Assert
29             // Handled in ExpectedException
30         }
```

Figur 8.1: Opbygning af unit test

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
AStjerne	0	0,00 %	63	100,00 %
AStar(Hjemmeplej...	0	0,00 %	47	100,00 %
FindStartknude(Qu...	0	0,00 %	12	100,00 %
KørAStjerne(Quick...	0	0,00 %	4	100,00 %

Figur 8.2: Kodedækning

9 Udviklingsproces

Dette kapitel omhandler vores udviklingsprocess, med hensyn til vores anvendelse af den iterative metode, vi har beskrevet i afsnit 2.2. Vi vil beskrive forløbet af hver iteration, herunder iterationens målsætninger og indhold. Herefter vil vi evaluerer på vores anvendelse af den iterative arbejdsform, i kombination med den menneskecentrede tilgang vi anvendte i projektet (se kapitel 1). I denne evaluering vil vi reflekterer over brugen disse metoder, og konsekvenserne af dem.

9.1 Iteration 1

Iteration 1 blev planlagt med henblik på at forstå problemområdet og anvendelsesområdet. For at forstå hjemmeplejens situation, arrangerede vi interviews med tre interessenter. Disse interviews etablerede grundlaget for at vi kunne formulere et problem, identificere systemer, og modellere problem- og anvendelsesområdet.

Det blev klart, at et væsentligt problem var planlægning af ruter. Derfor blev problemformuleringen udformet med fokus på effektivisering af ruteplanlægningen i hjemmeplejen. Til valg af system brugte vi forbilleder, metaforer, prototyper og rige billeder over hjemmeplejens situation. Vi fandt et muligt kommercielt forbillede *VSC VITAE*, men det var desværre ikke muligt at få dybdegående information om dette system.

Som metaforer fandt vi et eksempel på en frisørsalon, som kunne anvendes til at generere ideer til en initiel systemdefinition. Der blev lavet forskellige prototyper for at eksperimentere med bud på, hvordan en løsning kunne se ud. Vi valgte en systemdefinitionen, der fokuserede på semi-automatisk planlægning frem for manuel/fuldautomatisk planlægning.

For at analysere problemområdet itererede vi imellem klasseaktiviteten, strukturaktiviteten og adfærdsaktiviteten[3], da der var mange forskellige ideer til disse. Ideer, der ikke blev valgt i denne iteration, blev dokumenteret til overvejelser i senere iterationer. Første udkast til en hændelsestabel, klassediagram og tilstandsdiagrammer over adfærd blev derved udformet som basis for efterfølgende iterationer.

I analysen af anvendelsesområdet fandt vi i brugsaktiviteten, at planlæggeren var den hoved-

saglige aktør i systemet. For at få en bedre forståelse af hvordan aktørerne kunne interagere med vores idéer til et system, udformede vi en grov prototype, lavet i Blend Sketchflow, der repræsenterede vores opfattelse af planlæggerens arbejde, og som planlæggeren skulle forsøge at navigere i igennem test cases. Samtidig var prototypen også med til at evaluere om vores forståelse af problemområdet stemte overens med planlæggerens forståelse. Interviewet gav en bekræftelse i vores forståelse af problemområdet, dog med få vigtige mangler, samt en evne til at navigere i prototypen efter hensigten.

Som en del af iteration 1 ønskede vi også at finde ud af hvilke algoritmiske muligheder, der kunne være relevante i dette projekt. Pga. systemdefinition blev det ligeledes deduceret, at en stor del af løsningen kunne komme til at omhandle grafteori i form af shortest path og Travelling Salesmen problemer.

9.2 Iteration 2

Iteration 2 blev planlagt med henblik på at danne en bredere forståelse af systemets objekt orienterede design, samt at lære og benytte **Model View View Model** (MVVM) mønstret via **Windows Presentation Foundation** (WPF). Ud fra dette valg opstod et implicit behov for at lære XAML, da dette er grundstenen i at skrive MVVM Views. En stor del af 2. iteration gik derfor med at få et overblik over XAML, samt at gå i dybden med dette.

Igennem iteration 1 var der også kommet en bredere forståelse af problemområdet, som gjorde at vi ønskede at gennemgå analysefasen igen. Iterationen udmundede i et første udkast til programmets brugergrænseflade. Samtidigt blev en del af Programmets ViewModel udarbejdet.

Der blev fastlagt kriterier for vores systemdesign samt arkitektur, design af modelkomponent, og design af funktionskomponent. Arkitekturen blev en lille variation af grundarkitekturen, med en ViewModel komponent imellem brugergrænsefladen og modellen.

Efter iteration 1's prototypetestning var det tydeligt, at træk-og-slip var et krav til systemet. En del timer blev derfor lagt i at udforske denne funktionalitet, men det viste sig dog at være urealistisk at implementere, i forhold til tidshorisonten for projektet. Funktionaliteten endte derfor med ikke at blive færdig til iterationens afslutning. Vi vurderede at træk-og-slip bestemt var et krav og skulle med i systemet, men at implementationen var for stor en mundfuld i

projektets nuværende tidsramme. Vi valgte derfor at afgrænse os fra implementeringen af træk-og-slip og lade vende tilbage til det i fremtidig udvikling af system.

Fra iteration 1 manglede vi informationer om Attendo's systemer og arbejdsgange. Derfor planlagde vi i iteration 2 at få svar på nogle af spørgsmålene gennem en mailkorrespondace med Attendo. I afslutningen af iteration 2 havde vi dog endnu ikke fået svar fra Attendo's afdeling, hvilket resulterede i at vi måtte træffe nogle systemvalg på baggrund af intuition og logisk tænkning

I denne iteration ville vi ligeledes undersøge algoritmiske muligheder og begrænsninger for projektet. Det blev fundet, at en Traveling Salesman Problem (TSP) løsning, med fordel vil kunne benyttes i forbindelse med ruteplanlægning. Vi undersøgte mulighederne for at bruge A-stjerne algoritmen med et minimum udspændende træ som heuristik.

I forbindelse med udarbejdelsen af brugergrænsefladen, at forsøge vi at skabe et mere brugervenligt arbejdsmiljø, da det var efterspurgt i vores interview med Attendos planlægger. Resultatet af dette arbejde kan ses af afsnit 5.3. Gennem iteration 2 fandt vi ligeledes, at en form for database til lagring af statisk data ville blive nødvendigt for programmets fremtidige udvikling. Dette resulterede i en implementation Microsoft Windows **Entity Framework**. Implementationen blev ikke komplet i iteration 2, men grundet nødtvendigheden for at kunne lagre statisk data, har teamet valgt at gennemføre implementationen i starten af iteration 3.

9.3 Iteration 3

Iteration 3 blev planlagt med henblik på at udarbejde de sidste essentielle dele af vores system, samt at udarbejde dokumentation for det aktuelle og akutte planlægningsmæssige perspektiv af vores systemdefinition. Grunden til vi valgte at arbejde med persistent datalagring, var grundet at dette kunne ses som et indirekte krav til systemet.

Fra iteration 2 fandt vi et behov for at lagre persistent data, hvilket derfor blev implementeret i denne iteration. Der blev lagt mange ressourcer i at få denne funktionalitet til at virke, hvilket resulterede i en formindsket implementationer af andre store funktionaliteter. Vi valgte at benytte Microsoft Windows Entity Framework til datalagring, samt hentning af data. En essentiel del af planlægningssystemet var også, at det skulle være muligt at tilføje og slette opgaver fra ruter, hvilket derfor også blev implementeret i iteration 3.

Under 2. iteration fandt vi, at der var behov for en algoritme til løsningen af et TSP, da dette kunne benyttes i forbindelse med ruteplanlægningen. Vi prøvede i 3. iteration at udvikle en sådan algoritme. Under udviklingen af algoritmen, blev forsøgt med A-stjerne algoritmen med et minimum udspændingstræ som heuristik. Denne blev valgt, da vi fra tidligere semestre allerede havde samlet viden om minimum udspændende træer, samt Dijkstras algoritme. Desuden blev der arbejdet på visningen og sortering af ruter på tværs af dage, da dette også var en vigtig faktor for et funktionelt kørende program afsnit 6.3.4.

Iteration 3 blev på ligevis brugt til at inkludere aktuel planlægning i analysen og designet af systemet. Vi afgrænsede os dog fra at implementere aktuel planlægning, og overlade dette til fremtidig udvikling af systemet. Da projektet har en tidsgrænse, samt en endelig mængde ressourcer, bliver vi nødt til at fravælge bestanddele af implementeringen. Disse afgrænsninger er redegjort for i afsnit 7.1.

Denne iteration skulle også bruges til at hente de sidste manglende informationer fra planlægger, så der kunne komme styr på de sidste systemkrav. Dette viste sig dog at være en større udfordring end ventet. Den afdeling vi havde ventet at kunne hente informationer fra, havde haft en travl periode, og kunne derfor ikke holde en samtale kørende med os i perioden vi afholdte 3 iteration. Med dette faktum, blev vi derfor nødsaget til at tage designbeslutninger uden at inddrage slutbruger. Optimalt havde vi kunne holde respons med systembruger gennem hele forløbet.

Med lidt under en måned til semestrets afslutning, begyndte vi desuden at afsætte ressourcer til rapportskrivning, hvilket vil blive en stadig større del af hverdagen den kommende periode, frem mod projektets afslutning.

9.4 Afsluttende iteration

Den afsluttende iteration var præget af væsentlig forskellighed fra normale iterationer. Dette betød at vi ikke længere foretog intentionel informationssøgning, med det formål at afrunde og udjævne. Denne afrunding og udjævning blev foretaget hvor iterationer tidligere havde udvidet forståelsen af ét område, uden at have inkluderet ændringerne i forbundne dele. Disse forbundne dele blev derfor bragt op på et niveau, uden at gennemgå en egentlig iteration. Dette betød blandt andet, at brugergrænsefladen gennemgik en grafisk opdatering, for at stemme overens med valg der var blevet taget i afsnit 5.3.

Iterationen bar tydeligt præg af en øget fokus på dokumentering. Dette medføre en gradvist fald i udvikling af systemet, hvilket kulminerede i en *feature freeze*, hvorefter vi så vidt muligt arbejdede imod at færdiggjorde påbegyndt funktionalitet, og ikke implementerede yderligere dele af designet. Færdiggørelsen indebar i stor grad håndtering af logiske fejl og køretidsfejl.

Denne fejlhåndtering blev også motiveret af forberedelser til udførelsen af en usability test. Disse forberedelser indebar blandt andet opbygning af et datasæt, der kunne repræsentere situationen til brug i testen. Testen blev udført, med formålet at finde fejl og mangler i brugergrænsefladen. Resultaterne af testen kan ses i afsnit 8.2. Resultaterne af testen blev inkorporeret i systemets design, men undladt i den tekniske del af systemet. Dette var grundet den tidsmæssige horisont for projektets aflevering.

Vi benyttede også denne iteration til at udføre unittests afsnit 8.3, for at sikre os at den komplekse funktionalitet i vores implementation af A-stjerne algoritmen var korrekt.

9.5 Evaluering af udviklingsprocess

Dette afsnit omhandler evalueringen af vores iterationer. Herunder beskrives hvilke fordele og ulemper ved anvendelse af iterativ metode vi oplevede, samt hvorledes kombinationen af iterativ metode og menneskecentreret tilgang har bidraget til udviklingsprocessen. Desuden omhandles konsekvenserne af vores anvendelse af iterativ metode for vores system.

9.5.1 Informationsindsamling

Den viden som ligger til grund for dette projekt, er blevet indsamlet fra to kilder: Aalborg Kommune og Attendo. Det blev besluttet at fokusere på én af disse kontakter grundet den begrænsede tidshorisont for projektet. Vi ville desuden gerne undgå konflikterende informationer om arbejdsgange, da de to afdelinger har forskellig interne strukturer. Dette blev gjort med viden om et tab af alsidighed. Af disse to kilder til information, blev Attendo valgt, da de var mere samarbejdsvillige.

Den iterative arbejdsmetode i kombination med vores menneske-centrerede tilgang, gav anledning til en dialog med brugerne fra attendo, der spændte over hele udviklingsforløbet. At vi blev ved med at opnå yderligere viden hver gang vi var i diaolog med brugeren, er et tegn

på at projektets usikkerhed har været høj, som beskrevet i afsnit 2.2, men at vi i takt med at have opnået ny information om kravene til systemet, også har sænket usikkerheden omkring hvordan problemformuleringen kan løses.

9.5.2 Relationsdatabase

Under udviklingen af systemet, fandt vi ud af at et krav til systemet var persistent data. Valget faldt en relational database vha. Entity Framework, da det understøttes af .NET, men implementationen af dette, viste sig at være tidskrævende, da vi ingen erfaring med dette framework havde. Selvom denne type database ikke er et krav til projektet, føler vi at det have givet os en større forståelse af arkitekturen, da vi var tvunget til at tænkte dette med i designet af systemet. Vi vurderer at arbejdet med databasen, indirekte har øget kvaliteten af systemet som helhed, og at potentiel videreudvikling af systemet vil være støttes grundet dette.

9.5.3 Elementer fra brugergrænsefladen

Der eksisterede allerede tidligt i forløbet et ønske fra brugerens side om en modernisering af grænsefladen. Dette indebar især brugen af træk-og-slip funktionalitet. Der blev forsøgt med forskellige implementationer af sådan en funktionalitet, men det var klart at dette krævede et avanceret kendskab til WPF, hvis vi samtidigt ønskede at overholde MVVM mønstret. Resultatet blev at analysen og designet af brugergrænsefladen afspejler ønsket om træk-og-slip, men selve implementering var undladt i den nuværende tilstand af implementeringen af systemet.

9.5.4 Iterativ udvikling og forbedringsforslag

Da dette var først gang vi arbejdede med med den iterative metode, medførte dette i starten af projektet til uklarhed som arbejdsformen. Dette betød at vi i tidlige dele af processen brugte meget lang tid på at iterere på tidligere aktiviteter. Vi undlod også at beskæftige os med implementationen i de første iterationer, da vi ønskede at lægge vægt på analysen og designet af systemet. Vi vurderer at dette gavnede os, da den objektorienterede analyse og design var med til at danne et solidt grundlag for implementeringen. Ligeledes har vores analyse og design af systemet givet os et grundlag for videre udvikling af systemet i fremtidige iterationer.

Som afsnit 7.1 viser, er der mange afgrænsninger i implementationen i forhold til analysen og designet af systemet. Ligeledes er der flere forskelligheder mellem hvordan vi har implementeret systemet og hvordan designet er. Årsagen til dette har blandt andet været, at vi ønskede at udvikle bredtdækkende funktionalitet i til at dække mange af hjemmeplejens krav, men på samme tid var ønsket også at udvikle en avanceret funktionalitet i både brugergrænsefladen og effektivisering af ruterne.

Implementationen af systemet er derfor blevet fanget i konflikten mellem disse to ønsker, hvilket har betydet at den nuværende tilstand af systemet ikke har avanceret funktionalitet i forhold til ønsket om at udvikle den automatiske udregning af ruter. Kontrasten mellem systemet beskrevet i designet og implementeringen indikerer, at vi har været for optimistiske omkring hvad der kunne udvikles i tidshorisonten for projektet, samt en mangel på afgrænsning tidligere i udviklingen af systemet.

Denne konflikt er en tydelig konsekvens af vores anvendelse af den iterative metode, som indirekte påkræver tid til at revurdere beslutningerne taget igennem hele projektets forløb. På trods af dette, ville vi få løst denne konflikt i fremtidige iterationer over systemets udvikling, da den grundlæggende arkitektur og implementering af systemet er fastlagt med dette for øje.

Yderligere idéer til systemet, der kan implementeres i fremtidige iterationer, er 'fortryd funktionalitet' og indkapsling af **opgaver** i **arbejde**. Det er p.t. ikke muligt at fortryde handlinger, men det er oplagt at bruge kommandomønsteret til at implementere den funktionalitet.

Ligeledes, når flere af den samme **borgers opgaver** ligger efter hinanden på en **rute**, bliver de repræsenteret med hver deres **arbejde**. Det ville gøre **ruten** mere overskuelig, hvis de blev samlet i én **arbejde**. Dette kan med fordel implementeres med samlingsmønsteret.

I designet af funktionskomponenten fandt vi opdateringsoperationerne *MarkerRuterDerikkeMåÆndres*, *MarkerHjælperDerIkkeMåÆndres*, *MarkerOpgaverDerIkkeMåÆndres*, samt klassen **ForslagsAdministrator** der indkapsler disse operationer. Disse operationer og klassen i arkitekturen påkræver dog at der findes en opdateringsfunktion og ligeledes en hændelse i analysen. Det er dog ikke afspejlet, da operationerne og klassen blev inkorporeret sent i sidste iteration under designfasen, hvilket betød at den blev skubbet foran udviklingen. Dette betyder at yderligere iteration over analysen er påkrævet, for at få inkorporeret den manglende hændelse og funktion i analysen, så analysen er korrekt afspejlet i forhold til designet.

A-stjerne algoritmen beskrevet i afsnit 6.3.4, tager kun højde for en tidsmæssig heuristik. For at opfylde krav til placering af **opgaver** på **ruter**, således at der eksempelvis ikke serveres

morgenmad om aftenen, vil en mulig løsning være at tilføje flere heuristikker, der håndterer andre krav end de tidslige.

10 Diskussion

Under udviklingen af dette projekt blev der taget nogle valg, der påvirkede den etiske retning projektet tog. I dette afsnit diskuterer vi konsekvenserne af disse valg og argumenterne for dem i forhold til problemformuleringen. Ligeledes vil der være en kort opsummering af kriterierne til designet af systemet, og hvorledes vi opfyldte dem.

10.1 Deterministisk vurdering af pleje

Hvis en optimering af hjemmeplejens ruteplanlægning skal opnås via automatisering, er det nødvendigt at kunne determinere, hvorvidt en plan er mere eller mindre optimal end et alternativ. Hjemmepleje er dog et nuanceret socialpolitisk emne, og sagens natur er, at kriterierne for en optimal rute er subjektive og ofte indbyrdes omvendt proportionelle. Denne flygtighed sætter begrænsninger for brugen af en statisk deterministisk tilgang til beslutning om, hvad god hjemmepleje er.

Denne vurdering vil have forskellige udfald, alt efter hvem der tilspørges, og hvornår de tilspørges. I praksis kunne dette specifikke eksempel muligvis blive løst ved at spørge beboeren, hvad der ønskes for det pågældende besøg, når hjælperen ankommer til husstanden. Men fra et planlægningsperspektiv skal denne vurdering tages objektivt forud for besøget.

Med dette i tankerne har vi dog stadig valgt give et forslag til en modellering, som vil muliggøre sammenligningsmetoder med et deterministisk udfald. For at dette skal kunne lade sig gøre, skal der bestemmes en skala, hvorpå værdien af ethvert valg om plejen kan sammenlignes med andre. At bestemme denne skala er en beslutning som bør overlades til brugerene af systemet og skal kunne kalibreres løbende, i takt med at det socialpolitiske landskab ændrer sig.

10.2 Varierende besøg

Den eksisterende planlægningsmetode beror på en ugentlig gentagelse af ruter, hvorpå der så blot tilføjes eller fjernes besøg fra eksempelvis "mandags-ruten". Vi har med denne løsning

forsøgt at ændre på strukturen, for at bedre give mulighed for optimeringer. Dette har givet anledning til nogle bekymringer.

10.2.1 Kvalitet for borgeren

I interviewene med hjemmeplejen blev der beskrevet, at en stor del hjemmehjælpsmodtagerne ønskede regelmæssig gentagelse af deres besøg. Mangel på rytme i besøgene kan muligvis give anledning til at borgerne glemmer at være tilstede til besøget eller føler en mangel på besøg. Udover at dette kan bidrage til uro hos borgeren, vil det sandsynligvis føre til flere henvendelser til hjemmeplejen fra borgerene.

Det vil også være nødvendigt at kunne informere borgere omkring fremtidige ruter og ændringer af allerede eksisterende ruter. Jo oftere man ændrer tidspunktet for besøget hos borgeren, desto oftere skal man informere borgeren om ændringerne. Dette problem eksisterer ikke i særlig høj grad i hjemmeplejens nuværende system, da de har faste ruter for hver dag i ugen. Ved vores system forværres dette problem markant, da der hyppigere vil foretages ændringer. Dette er et helt andet problem, der bør overvejes i videreudvikling af systemet.

Til gengæld vil vores system kunne give tidsmæssige besparelser for hjemmeplejen. Tidsmæssige besparelser vil i sidste ende ikke have en indvirkning på den tid, der bliver afsat til besøg hos borgerne. Denne bliver sat af visitator som beskrevet i afsnit 3.1.

10.2.2 Hjælperes effektivitet

For hjælpere kan hyppige ændringer af deres ruter medføre en lavere effektivitet, da det forhindrer dem i at opbygge en længere rytme, og derfor oftere ville være nødt til at sætte sig ind i ny information om borgere, ruter m.m. Tilgengæld kan variationen tænkes at være et friskt pust for hjælpere, der mangler afveksling i hverdagen. Dette skal dog undersøges nærmere, før en eventuel gevinst kan fastlægges.

10.3 Opfyldelse af kriterier

I afsnit 6.1 definerede vi en række kriterier for designet af systemet. I dette afsnit beskrives hvorledes vi opfyldte disse kriterier.

Vi har opfyldt *brugbarheden* igennem analysen og realiseringen af brugergrænsefladen. Vi anvendte teknikker og retningslinjer beskrevet i litteratur omkring brugergrænseflader.

Sikkerheden af systemet opfyldte vi ved at skelne mellem klienterne for planlægger og hjælper, der ikke har samme rettigheder i systemet.

Effektiviteten af designet har vi opfyldt ved at undersøge mulige algoritmer til planlægning af ruter, og ved den efterfølgende anvendelse af A-stjerne algoritmen beskrevet i afsnit 6.3.4.

Korrektheden af designet opfyldte vi ved at opfylde kravene til modellen og funktionaliteten. Realiseringen af disse er ikke fuldkommen, men de er taget hånd om i designet.

Pålideligheden af designet opfyldte vi ved at vise ændringer visuelt i brugergrænsefladen, så brugeren kan skabe overblik over ændringer foretaget af både den manuelle og den automatiske planlægning.

Forståeligheden opfyldes ved indkapsling og samhørighed af design i form af de forskellige komponenter. Ligeledes i realiseringen ved anvendelse af objektorienteret programmering.

Vedligeholdbarheden blev opfyldt ved objektorientering og anvendelsen af den lagdelte arkitektur sammen med anvendelsen af MVVM designmønsteret.

Fleksibiliteten har vi opfyldt med fokus på lav kobling og lagdelt arkitektur. Ligeledes har vores anvendelse af strategimønsteret i implementeringen skabt mere fleksibilitet i dele af systemet. Derudover har vores valg af Entity Framework været med til at forøge fleksibiliteten, da det giver mulighed for nemt at udskifte af den underliggende databaseteknologi.

Genbrugbarhed har vi ikke fokuseret på, så der er ikke foretaget tiltag for opfyldelse af dette kriterie.

Integrerbarheden er opfyldt ved designet af systemgrænsefladen, der udstiller et interface til at interagere med vores system.

Testbarheden er opfyldt gennem vores anvendelse af MVVM.

Hvis planlæggeren skal foretage en komplet vurdering af resultatet, mistes store dele af den tidsbesparelse automatiseringen ellers kunne have givet. Det er derfor til en vis grad en nødvendighed for systemets automatiske funktionalitet, at planlæggeren stoler på resultaterne, der genereres. Denne pålidelighed kan ikke forventes at være tilstede fra første gang programmet tages i brug, men bør opnås relativt hurtigt.

10.3.1 Korrekt kalibrering af udregninger

Før en pålidelighed kan eksistere i nogen form, er det nødvendigt, at resultatet matcher planlæggerens forventninger. Selv hvis kalibreringen er vurderet korrekt ifølge det samfundspolitiske landskab, vil det give problemer, hvis de ikke tilfredsstiller planlæggerens personlige vurdering. Hvis der opstår mistillid i tidlige versioner af systemet, kan denne potentielt blive længe efter rettelser er lavet. Det anbefales derfor, at kalibreringen foretages i tæt samarbejde med flere dele af hjemmeplejen, og specialiseres yderligere til de enkelte institutioner.

10.3.2 Gennemskuelighed via brugergrænsefladen

Hvis tilliden til automatiske resultater skal opbygges indenfor en begrænset tidsramme, er det nødvendigt at bidrage til dette via brugergrænsefladen. Dette bør gøres ved gennemskuelig fremvisning af ruteændringer. Den primære problematik er at formidle resultatet på en sådan måde, at der fra planlæggerens synspunkt ikke behøves forklaring af de komplekse matematiske udregninger, der foretages forud for fremvisningen af resultatet.

Det anbefales, at enhver bruger af systemet deltager i et kort introduktionskursus, og at der eventuelt gives et uddybende kursus til en administrativ nøgleperson, der kan løse forståelsesproblemer, når de opstår lokalt i afdelingerne.

11 Konklusion

Dette kapitel vil bestå af en besvarelse af problemformuleringen i kapitel 4, i form af besvarelse af spørgsmålene stillet i samme kapitel.

Kvalitetssikringen har påvist, at de krav vi har identificeret til modellen og funktionaliteten stemmer overens med hjemmeplejens ønsker. På samme tid har kvalitetssikringen også vist, at der stadig er krav, vi ikke har opfyldt fuldkomment endnu. Dette skyldes især de afgrænsninger, der blev foretaget fra modellen og funktionaliteten blev lavet, til systemet blev implementeret. Derfor konkluderer vi, at systemet i sin nuværende tilstand tager hånd om, men ikke fuldt ud tilfredsstiller kravene til modellen og funktionaliteten. Vi konkluderer også, at der er brug for yderligere iterering på projektet.

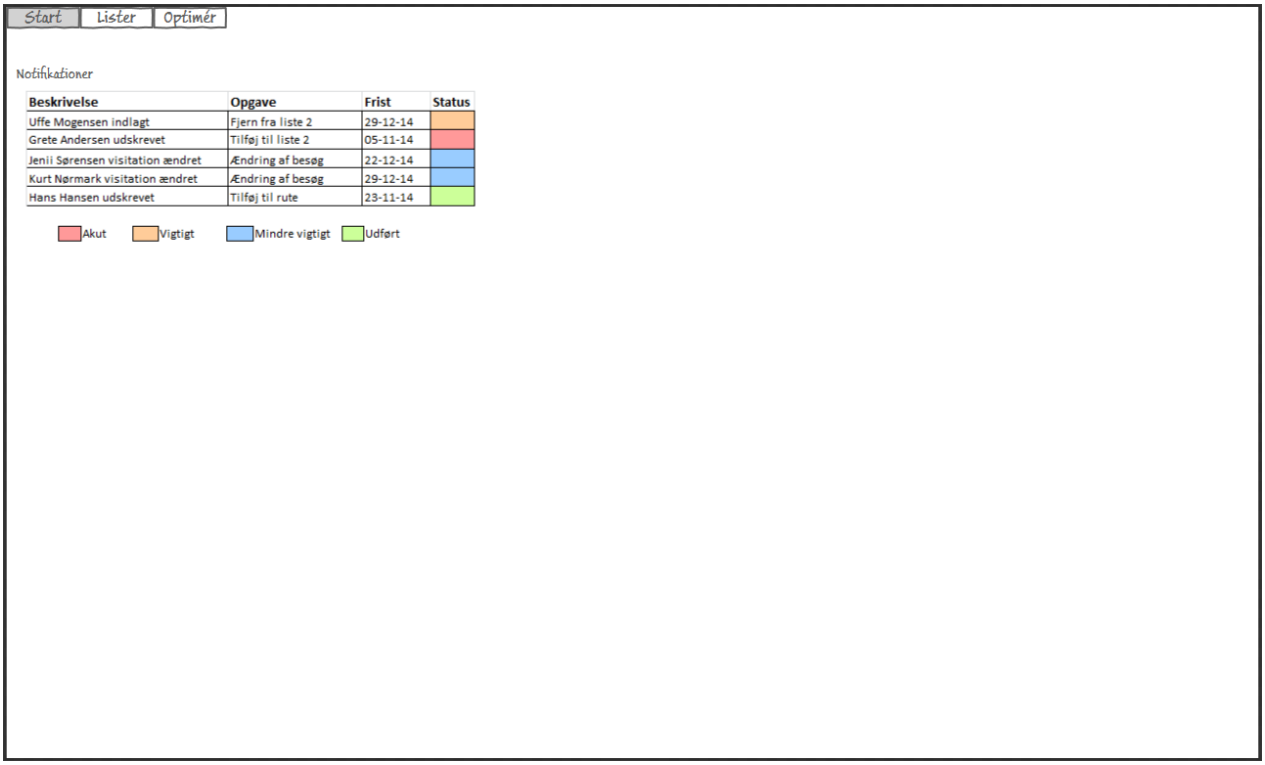
Da vi ikke har realiseret og testet den fulde udgave af systemets planlægning af ruterne, kan vi ikke konkluderer, hvorvidt den vil overholde de krav, der stilles til opbygningen af ruterne. Det kan ikke konkluderes med sikkerhed, hvorvidt den komplette implementering af den semi-automatiske funktionalitet ville kunne mindske tidsspildet.

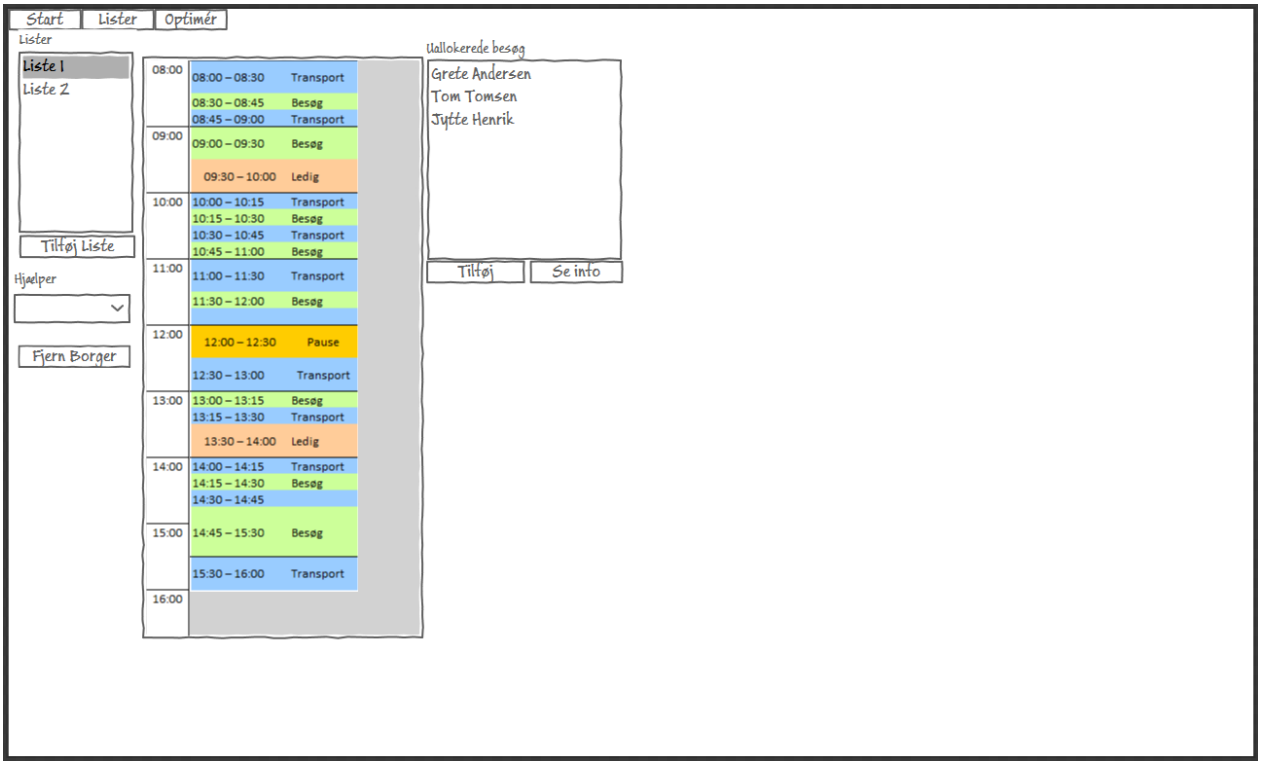
Ved udvikling og implementering af alt den afgrænsede funktionalitet, bliver det muligt at sammenligne planlagte ruter, lavet af hjemmeplejens medarbejdere, med ruter, som systemet ville foreslå. På den måde kan man teste om systemet kan mindske tidsspild, og derved foretage en mere præcis konklusion omkring effektiviteten af den automatiserede funktionalitet. Dog bidrager funktionaliteten til bedre muligheder for brugerens evne til at vurdere planlægningen af ruter. Vi konkluderer derfor, at systemet vil mindske tidsspild under planlægningen af hjemmeplejen.

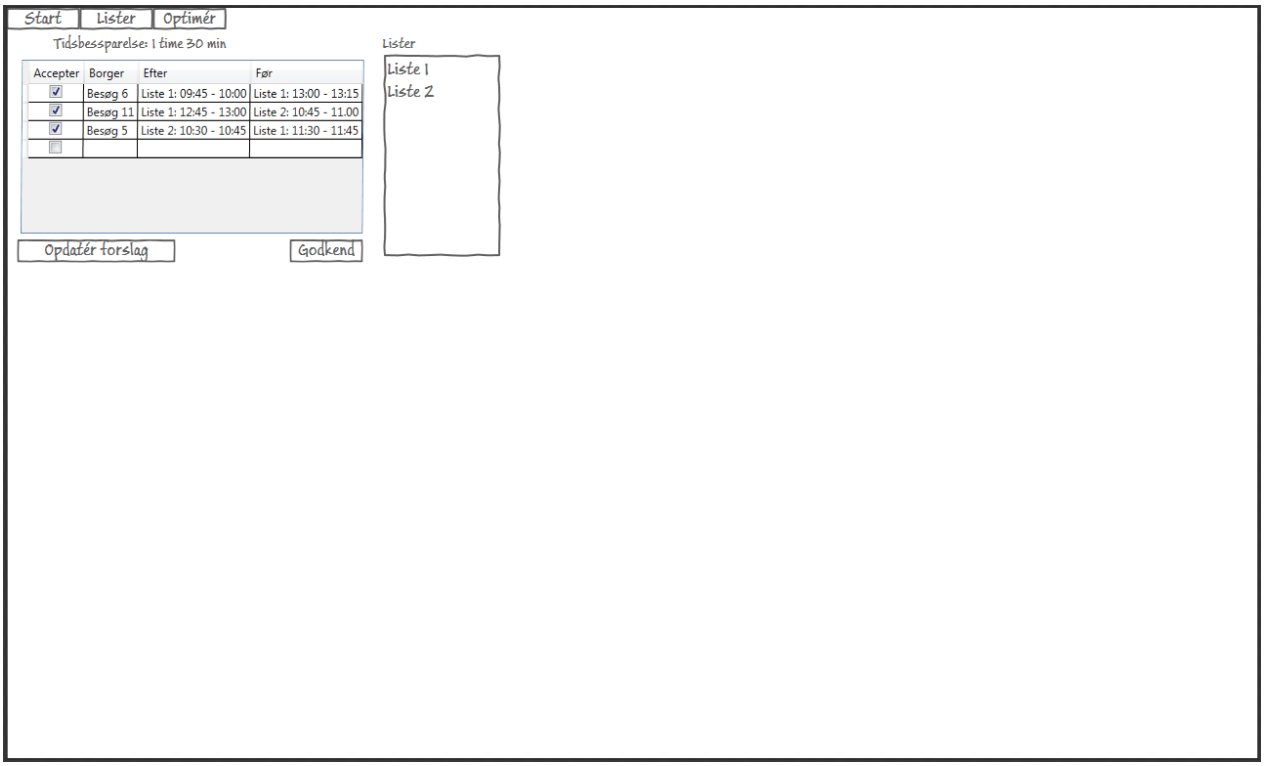
Vi vurderer at yderligere iteration, og videreudvikling af systemet, ville afhjælpe alle væsentlige svagheder af designet, samt tilfredsstille kravene til modellen og funktionaliteten. Denne vurdering baseres på kvalitetssikringen, der påviser, at den hidtidige udvikling har haft success med gradvist at forbedre forståelsen for hjemmeplejens krav, og hvordan de opfyldes.

12 Bilag

12.1 Design af prototype




Start
Lister
Optimér
Lister
Liste 1
Liste 2
Tiltøj Liste
Hjælper
Fjern Borger
08:00
09:00
10:00
11:00
12:00
13:00
14:00
15:00
16:00
08:00 – 08:30 Transport
08:30 – 08:45 Besøg
08:45 – 09:00 Transport
09:00 – 09:30 Besøg
09:30 – 10:00 Ledig
10:00 – 10:15 Transport
10:15 – 10:30 Besøg
10:30 – 10:45 Transport
10:45 – 11:00 Besøg
11:00 – 11:30 Transport
11:30 – 12:00 Besøg
12:00 – 12:30 Pause
12:30 – 13:00 Transport
13:00 – 13:15 Besøg
13:15 – 13:30 Transport
13:30 – 14:00 Ledig
14:00 – 14:15 Transport
14:15 – 14:30 Besøg
14:30 – 14:45
14:45 – 15:30 Besøg
15:30 – 16:00 Transport
Uallokerede besøg
Grete Andersen
Tom Tomsen
Jytte Henrik
Tiltøj
Se info


Start
Lister
Optimér
Tidsbesparelse: 1 time 30 min
Accepter
Borger
Efter
Før
Besøg 6
Liste 1: 09:45 - 10:00
Liste 1: 13:00 - 13:15
Besøg 11
Liste 1: 12:45 - 13:00
Liste 2: 10:45 - 11.00
Besøg 5
Liste 2: 10:30 - 10:45
Liste 1: 11:30 - 11:45
Opdatér forslag
Godkend
Lister
Liste 1
Liste 2

12.2 Prototype testcases

Opgave 1: Opret ny liste

Der er ikke længere plads på de nuværende lister. Derfor skal der oprettes en ny liste, ”Liste 3”, således at der bliver plads til flere besøg.

Opgave 2: Tilføj besøg til liste

Grete Andersen er lige blevet udskrevet fra sygehuset og skal igen have besøg. Tilføj derfor Grete Andersen til Liste 2 i tidsrummet 11:00 til 11:15.

Opgave 3: Fjern besøg på liste

Uffe Mogensen er på ferie med familien og skal derfor ikke længere have besøg. Fjern Uffe Mogensen (Besøget i tidsrummet 9:45 til 10:15) fra Liste 2.

Opgave 4: Tilføj hjælper til liste

Tilføj hjælperen Rasmus Hansen til liste 2 og flyt hjælperen Martin Jensen til liste 1.

Opgave 5: Borgerinformationer

Du skal have informationer om en borger for at se, om vedkommende kan flyttes til et andet tidsrum. Find informationer omkring borgeren, som er på Liste 2 i tidsrummet 14:30 – 14:45.

Opgave 6: Optimér

Du har fået et værktøj der kan generere forslag til effektive lister. Værktøjet findes under fanen Optimér. Ledelsen har bestemt at samtlige lister skal lægges om for at kunne effektivisere listerne. Anvend værktøjet til at generere forslag til lister. Besøg #5 må ikke flyttes til en anden liste, da borgeren er dement.

12.3 Interviewspørgsmål

Indledende eller opbyggende eller afsluttende

Kunne i være interesserede i at deltage i vores projekt, som en kontaktmulighed for os?

Vi har en interesse i at være observatør en normal dag, sammen med en hjemmehjælper. Kunne dette lade sig gøre?

Organisation (fokus på planlægning af hjemmeplejens opgaver, data) - Planlægning

Hvordan får planlæggeren informationer fra visitatoren?

Hvem foretager planlægningen, og hvad er dennes kvalifikationer?

Gør i brug af et IT-System til styring og planlægning af hjemmebesøg?

- Hvis nej, hvad benytter i jer så af?

Hvordan planlægger i hjemmehjælperens kørselsruter på nuværende tidspunkt?

Hvordan er fordelingen og samarbejdet mellem forskellige hjemmeplejer?

Hvordan er tidsfordelingen mht. et hjemmehjælpsbesøg, har i konkrete data på dette?

- Konkret. Hvis i står I en tidspresset situation, har I da prioriteter I forhold til arbejdsopgaver?

Hvilke arbejdsroller findes der i jeres organisation? (Planlægger, hjemmehjælpsmedarbejder, og andre roller?)

- Hvor mange besidder rollerne i jeres organisation?

Hvilke platforme gør I brug af (tablets, telefoner, computer, osv.)

- Må vi se teknologierne I anvendelse?

Har i nogen retningslinjer for uforudsete hændelser hos borgerne / hjemmehjælperne?

- Hvordan kontakter hjemmehjælperne jer I disse hændelser? (telefon, email osv..)

Hvor foretager planlæggeren dagens arbejdsopgaver? (kontor, hjemme, slet ingen)

Hvordan modtager de hjemmehjælpende dagens arbjedsopgaver?

Kunders forhold (kundernes indflydelse og bestemmelse)

Kan de ældre selv bestemme hvilken hjemmehjælper de vil have hjælp af, eller/og har de ansatte nogen rolle i denne fordeling?

Ansatte (Ansattes forhold og arbejdsopgaver)

Vil du give en beskrivelse af et normalt hjemmehjælperbesøg?

- Hvilke arbejdsopgaver varetager hjemmeplejen, her tænker vi I forhold til konkrete opgaver så som (indkøb, rengøring osv.)

Hvor mange borger indgår i jeres område?

Er der dage I ikke når alle arbejdsopgaver, som er planlagte for dagen?

- Hvis du kunne ændre på din arbejdsbyrde, for at gøre denne mere overskuelig, hvad ville du så have ændret?

I hvilke tilfælde må flere hjemmehjælpere være sammen om at udføre en given opgave?

Har i problemer med medarbejdere der går fysiske skavanker pga. Arbjedsbyrder?

- Hvilke typer skader er der tale om, og hvad fører til disse?

Hvilke krav eller forudsætninger har i til jeres ansatte?

- Sprog

Hvordan interagere hjemmehjælperne med systemet?

Hvilke transportmidler benytter de hjemmehjælperne sig af?

- Og hvorfor?

Bibliografi

[1] *Semester Beskrivelse*. 2009. URL: https://www.moodle.aau.dk/mod/page/view.php?id=264481 (sidst set 15.12.2014).

[2] Tine Rostgaard m.fl. „Send bare mere hjemmehjælp“. I: *Politiken* (2013), s. 7–7. ISSN: 0907-1814.

[3] Lars Mathiassen m.fl. *Objektorienteret analyse & design*. 3. udg. Marko, 2001. ISBN: 8777511530.

[4] David Benyon. *Designing Interactive Systems - A comprehensive guide to HCI, UX and interaction design*. APress, 2014.

[5] Bo Dahlbom og Lars Mathiassen. *Computers in context: The philosophy and practice of systems design*. Blackwell Publishers, Inc., 1993.

[6] *Chunking*. 2014. URL: https://www.interaction-design.org/encyclopedia/chunking.html (sidst set 01.12.2014).

[7] David Poole og Alan Mackworth. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, 2010.

[8] *Google Maps API for .NET*. 2013. URL: https://github.com/ericnewton76/gmaps-api-net (sidst set 01.12.2014).

[9] *Google Maps Distance Matrix begrænsninger*. 2014. URL: https://developers.google.com/maps/documentation/distancematrix/#Limits (sidst set 17.12.2014).

[10] *WPF Apps With The Model-View-ViewModel Design Pattern*. 2014. URL: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx (sidst set 17.12.2014).

[11] Dana Chisnell Jeffrey Rubin. *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*. John Wiley & Sons Inc., 2008.

[12] Kyle Burns. „Introducing MVVM“. I: *Beginning Windows 8 Application Development: XAML Edition*. Springer, 2012, s. 127–140.