VIA University College

Software Engineering
Via University College
SoftwareEngineering
Banegårdsgade 4
8700 Horsens

Titel:
1. Semester projekt: Kløverly

ECTS:
10 ECTS

Semester:
1. Semester

Projekt periode:
10/09/2025 - 19/12/2025

Projekt gruppe:
SEP1 Gruppe 3

Studie Nummer	Navn	Studieretning
362108	Devin Paul Jaworek	SW
361830	Victor Høggaard	SW
362962	Kasra Hojjatifar	SW

Vejleder:
Mona Wendel Andersen, Steffen Vissing Andersen

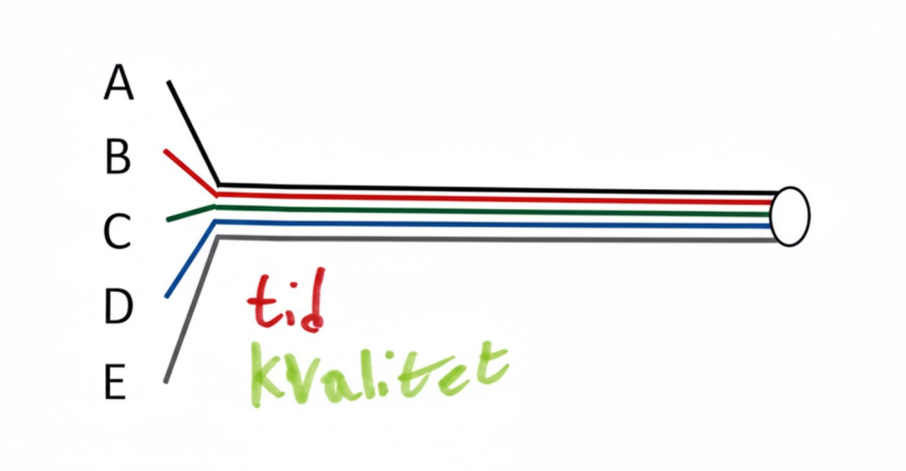
Sidetal: 16
Antal bilag: 30

1 Indledning

Der er lavet et projekt for Kløverlys administrator, Bob, som har brug for et bedre system til at holde styr på alle de opgaver, der bliver udført, og de grønne tiltag, som ydes i det grønne samfund. Vi har fået et interview med Bob som et værktøj til at kunne opstille krav til projektet og starte en designproces, efterfulgt af implementering og test. Bobs slutprodukt består af en GUI til Bob med intern databehandling og en offentlig hjemmeside, som kan tilgås af alle beboere.

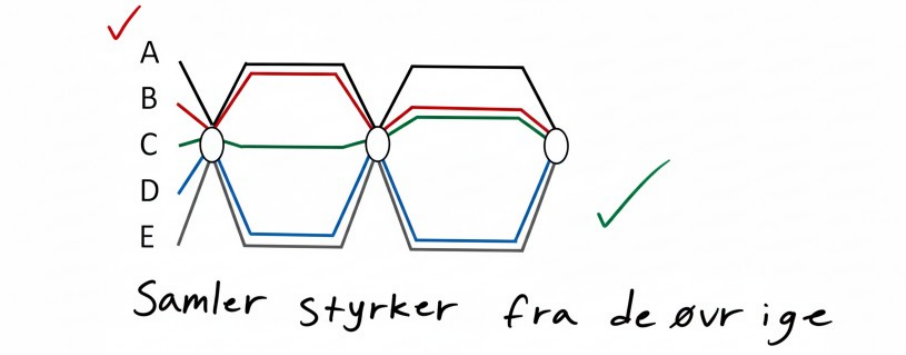
Vores motivation til det her projekt og semesterets læringsmiljø har været præget af intrinsiske mål bestående af, at vi alle er vilde med projektets omfang og tekniske udfordringer. I gruppen har vi alle arbejdet sammen om at fremme hinanden og vores tre psykologiske behov. Dette blev gjort ved både at lægge vægt på hinandens autonomi, kompetence og samhørigheden i gruppen, hvilket blev gjort på følgende måder: ved at lytte til hinanden med respekt, anerkende færdigheder, hjælpe hinanden og styrke relationer i gruppen.

Projektperioden var inddelt i to faser. Den indledende fase fokuserede på læring og planlægning, som forløb synkront med undervisningen. Gennem ugentlige møder og PBL-undervisning blev vi introduceret til foranalyse, analyse og design. I denne fase lavede vi foranalysen, analysen og designet til projektet.



Figur 1: Sammarbejdsmetode fælles

Ovenstående figur 1 illustrerer vores valgte samarbejdsmetode for Fase 1. Her valgte vi, at alle gruppemedlemmer skulle deltage i udarbejdelsen af foranalyse, analyse og design. Dette udspillede sig som en slags forventningsafstemning, der sikrede en fælles forståelse af projektet, hvilket både indebar, hvilke krav der skulle overholdes, samt hvilket mål der var for projektet. Hertil førte det til et meget veludført klassediagram, som kunne anvendes til den anden fase. Her lå fokus på implementering af kode og produktion af leverancer, såsom projektrapporten og denne procesrapport. Denne fase krævede en anden samarbejdsmetode for at spare tid (VIA University College, u.d.c).



Figur 2: Sammarbejdsmetode adaptiv

Ovenstående figur 2 beskriver samarbejdsmetoden, som er brugt i fase 2. Her har hele gruppen arbejdet med kode på samme tidspunkt, hvor der er nogle opgaver, som er blevet skrevet sammen, og nogle, som er blevet håndteret alene. Under udviklingsprocessen har man selvfølgelig også fundet nogle fejl, som har krævet, at man også fikser klassediagrammet og sekvensdiagrammet for Kløverly. Denne samarbejdsmetode har den fordel, at den er mere dynamisk, hvilket hjælper os med at kunne arbejde mere iterativt i programmeringsdelen af projektet. Der har også været fælles gennemgang af koden og test af programmet for at kunne sikre, at produktet opfylder kundens krav (VIA University College, u.d.c).

2 Gruppearbejde

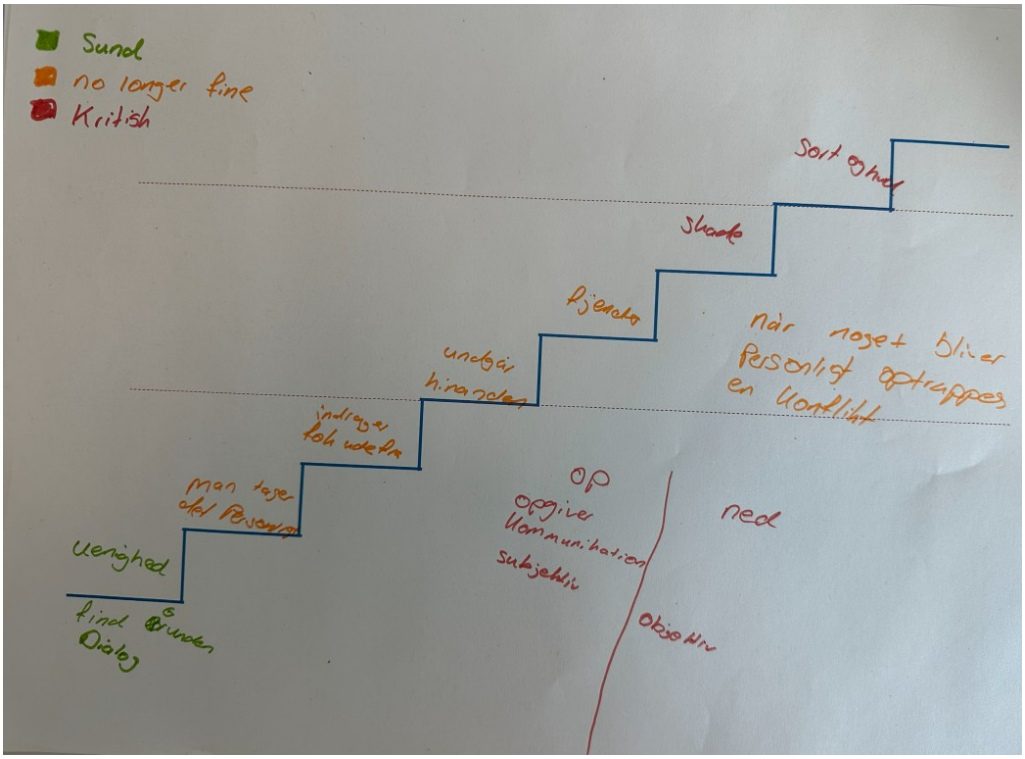
Vi har arbejdet ud fra en professionel tilgang, hvor vi har forsøgt at balancere det venskabelige med faglig effektivitet. Vores samarbejde har ændret karakter i takt med, at vi skiftede fra den indledende fase til den anden fase. Vi har bl.a. mistet et gruppemedlem i slutningen af den indledende fase. Dette resulterede i en forståelse af, hvor hårdt vi skulle arbejde for at opnå de mål, der blev sat som fire gruppemedlemmer i starten af projektet.

Gruppens dynamik har fulgt Tuckmans teamdynamik-model nogenlunde. Dog ikke til punkt og prikke.

Figur 3: Tuckman Teamdynamik

Figur 3 er et billede af vores fortolkning af Tuckmans teamdynamik-model, lavet i den indledende fase af projektet. Her var vi overbevisst om, at projektets gang ville komme til at ligne denne. Dog viste det sig at være usandt. Forming-fasen var ganske rigtig en fase, hvor vi lærte hinanden at kende og kom frem til, hvordan vi ville arbejde, og hvilke mål vi havde. Dog har vi sprunget Storming-fasen over, da der ikke er opstået konflikt i gruppen. Sommetider har der været små uenigheder, men de er ikke direkte kommet efter Forming-fasen. Disse er blevet behandlet ved konfliktstile, som kommenteres på senere. Resten af modellen er fulgt til punkt og prikke (MIT Human Resources, u.d.).

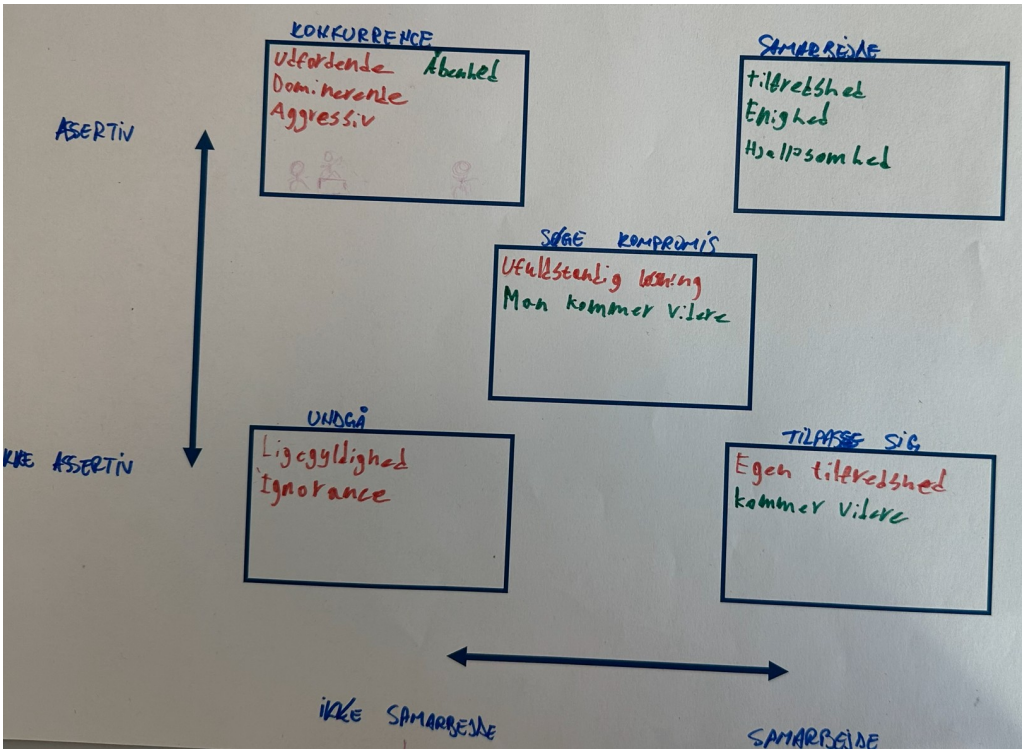
Konfliktløsning i gruppen er håndteret ved hjælp af vores forarbejde med konflikttrappen og konfliktstile. Hertil har gruppekontrakten (Bilag[19]) sørget for, at alle opretholder den rigtige indstilling til projektet, specielt i løbet af anden fase.



Figur 4: Glasl konflikttrappe

Figur 4 er et billede af vores forarbejdede version af Glasls konflikttrappe. Det viser et syn på konflikthåndtering via objektivitet og det at forblive i det upersonlige, når det angår konflikter. Dette har været en måde, hvorpå vi er gået ned ad konflikttrappen.(VIA University College, u.d.a).

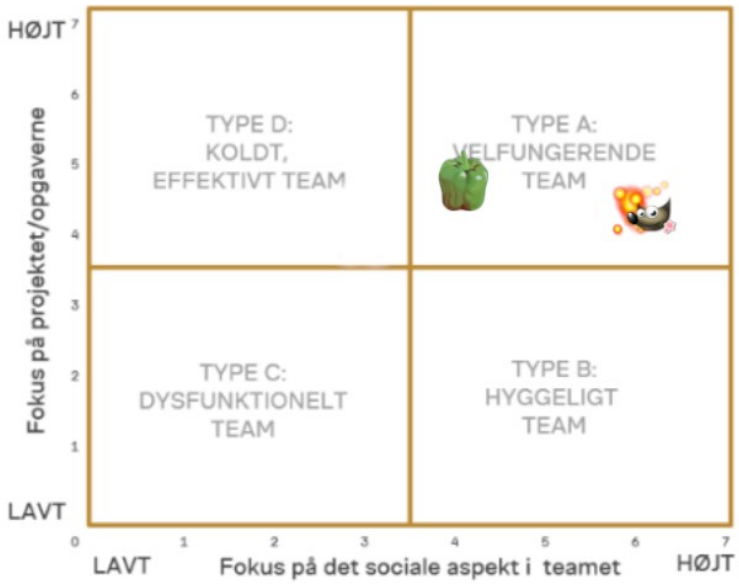
Derudover har vi brugt arbejdet med konfliktstile (Dual Concern Theory) til konflikthåndtering.



Figur 5: Konfliktstile (Dual Concern Theory)

Figur 5 viser vores fortolkning af teorien om konfliktstile. Her ses det, at vi vurderer samarbejde som værende en blanding af assertivitet og samarbejdsvilje. Det betyder, at det er utrolig vigtigt, at folk har mulighed for at komme med input i gruppen, som bliver modtaget med respekt. Det er også med denne indstilling, vi har forsøgt at håndtere vores konflikter (Lederweb, u.d.).

Alt i alt benytter vi os af åbenhed, respekt og objektivitet til at behandle mulige konflikter. Dog har vi været så heldige, at de eneste konflikter vi har oplevet, har været små uenigheder.



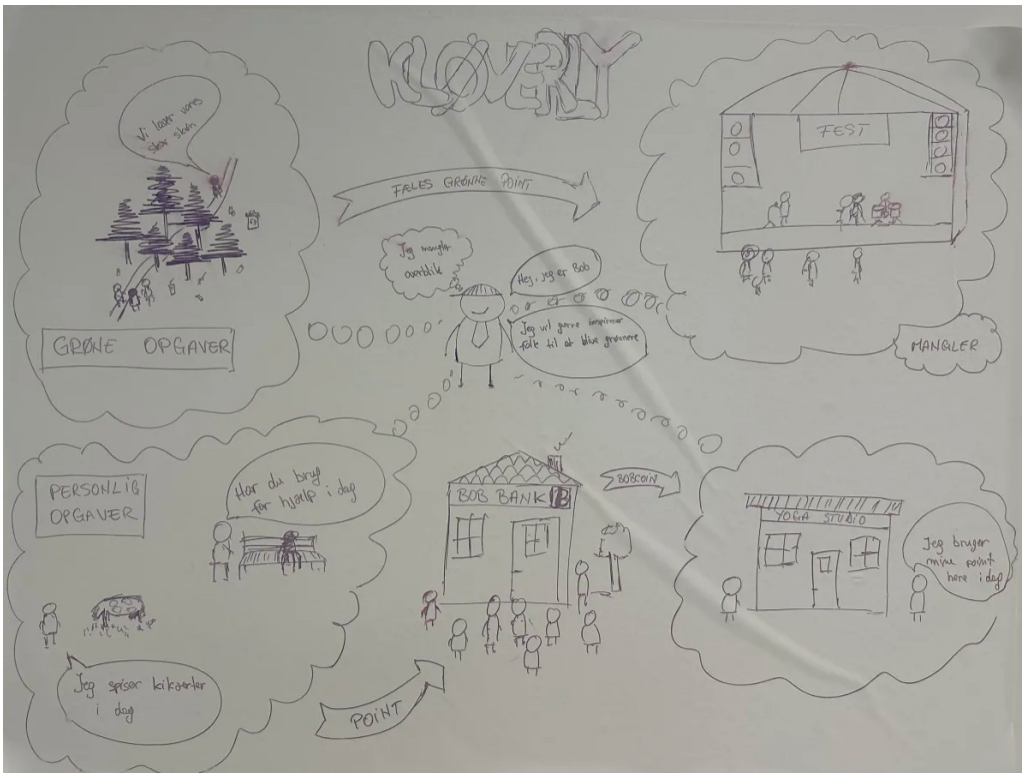
Figur 6: Teamtyper

Figur 6 viser alle gruppemedlemmers vurdering af teamtypen ud fra en række spørgsmål. Alle gruppemedlemmer vurderede gruppen som værende velfungerende, som vist på figuren.
Dette kan bl.a. skyldes, at der under ingen omstændigheder er opstået social loafing i vores gruppe. Ingen har trukket sig tilbage eller arbejdet mindre end muligt (VIA University College, u.d.b).

Gruppen har arbejdet i et ligeværdigt samarbejde, hvor alle har fået indført deres ønskede input. Alle har haft de samme ambitioner og mål, og derfor har vi alle bidraget med nogenlunde samme arbejdsindsats. Dette har resulteret i et samarbejde, der har fungeret godt med relativt få uenigheder og et godt læringsmiljø.

3 Projektinitiering

Projektinitiering, som i tidligere afsnit er blevet kaldt en indledende fase, indeholder til dels projektopstartsperioden og foranalysen. Her startede projektet ud med, at vi har fået et interview med Grønne Bob, hvor han beskriver sine ønsker til et administrationssystem. Udfra dette interview lavede vi et rigt billede til projektcasen.



Figur 7: Rigt billede

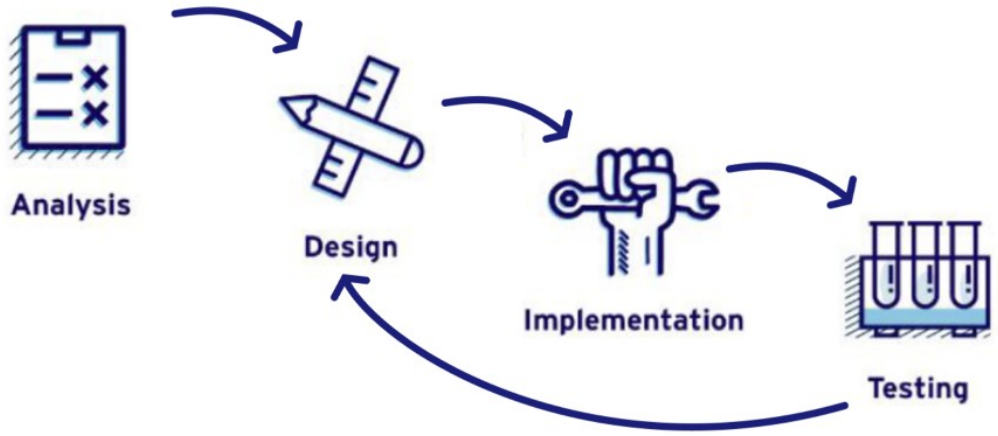
Ovenstående figur 7 viser det rige billede af projektet. Her fokuserede vi på Bobs hovedbehov, hvilket for os var, at han gerne ville kunne skelne imellem forskellige typer af opgaver og belønne beboerne i det grønne samfund til at yde arbejde for fællesskabet. Her identificerede vi i starten af projektet opgaverne lidt anderledes, end vi gjorde til sidst i projektet. Det kan ses i nederste venstre hjørne af det rige billede, at en beboer vil spise kikærter under personlige opgaver i stedet for grønne opgaver.

Det rige billede har her virket som et forståelsesværktøj for casen, der har gjort det nemmere for os at identificere Bobs krav. Foranalysen er baseret på interviewet med Bob og det rige billede. Foranalysen har været et vigtigt værktøj for os for at stille krav op, som resten af projektet er baseret på. Der kan argumenteres for, at projektinitieringsfasen har været den vigtigste fase for projektet. Her har initiering af gruppearbejde og tidshåndtering selvfølgelig også spillet en vigtig rolle, som beskrives i tidligere afsnit om gruppearbejde.

Det rige billede er, som tidligere beskrevet, et godt værktøj til den indledende fase af projektet. Vi har dog kunnet konkludere, at vores rige billede har været mangelfuldt, fordi vi efter projektinitieringsfasen blev nødt til at gentænke vores tolkning af problemstillingen. Dette skyldes blandt andet manglende opdateringer af det rige billede i takt med, at vi blev klogere på opgaven. Der er nemlig ikke blevet ændret på det rige billede efter første iteration.

4 Projektudførelse

Vi har samarbejdet med en professionel tilgang. Dette understøttes f.eks. af metoderne benyttet til projektstyringen. Det var nemlig essentielt med en god projektstyring, for at finde frem til, hvad er det næste på agendaen og, hvornår det skal være færdigt, samt specificere og opdeler det i delopgaver. Metode- og proces-artefakter blev udarbejdet under for-analysen og er dokumenteret i projektbeskrivelsen Bilag[3]projektets organisationsstyringen er der anvendt en tilpasset vandfaldsmodel. Modellen som anvendes kan ses herunder.



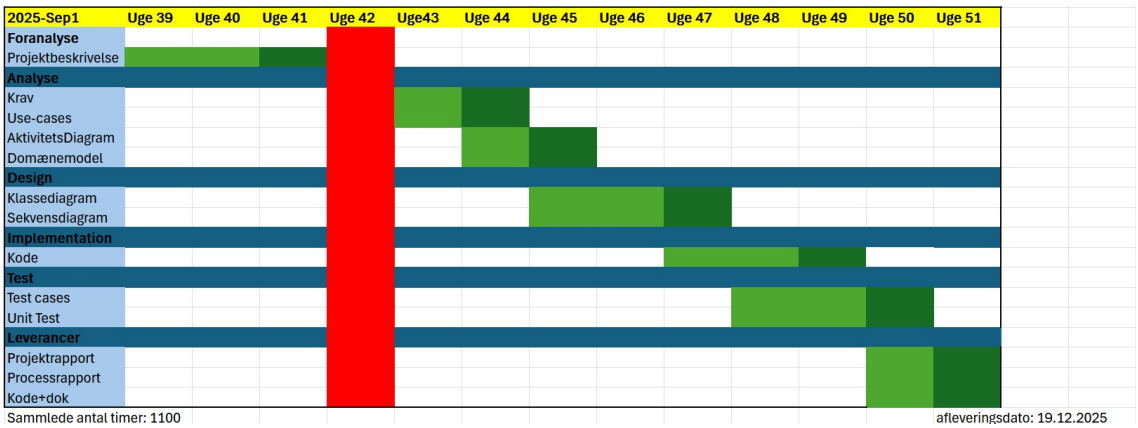
Figur 8: Projektets organisationsmodel

Ovenstående Figur 8 viser den tilpassede vandfaldsmodel som er blevet anvendt til projektarbejdet. Her er der valgt ikke at arbejde med standardmodellen som inkluderer 2 ekstra steps deployment og testing, da de vurderes at være uden for projektets omfang. Det er dog ikke den eneste forskell fra den klassiske vandfaldsmodel, hvor der gennerelt ikke arbejdes iterativt imellem faser men sekventielt.

Hertil vises analysefasen som et step, hvor den i realiteten har været opdelt i 2 faser værende for-analysen og hoved-analysen også refferet til som analysen.
Begge sub-faser har været kritiske for projekt-processen og har udformet 2 forskellige analyse-artefakter – en projektbeskrivelse Bilag[3] og et analyse artfakt Bilag[7].

Det er dog blevet valgt at gøre Design, implementationen og test fasen iterativ for at gøre det muligt at forbedre designfejl og implementationsfejl som kan findes under test fasen.
Den tilpassede vandfaldsmodellen fungerer som et stærkt velfungerende projektstyringsværktøj for at komme fra Interviewet med Bob (bob, 2025) til et fungerende produkt.

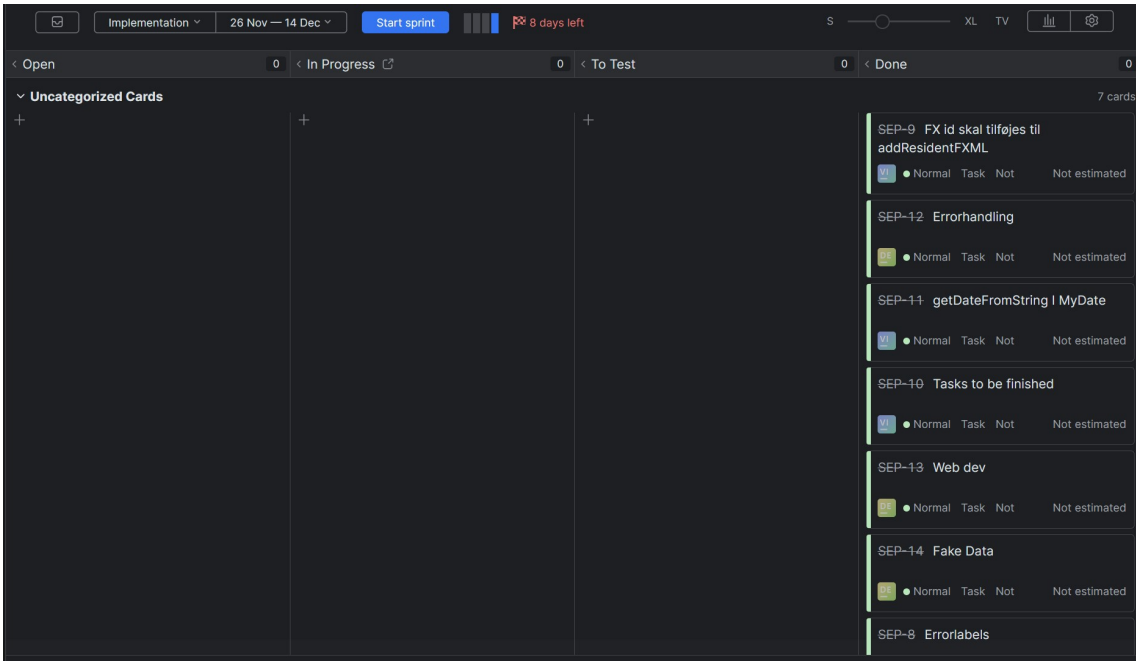
Håndtering af tidsstyringen i projektet er sket ved at anvende en tidsplan. Tidsplanen er baseret på undervisningslektionerne og tidsfrister som er givet af undervisere og tidsfrister som er besluttet af projektgruppen.



Figur 9: Projektets tidsorganisations værktøj

Ovenstående figur 9 viser tidsplanen for projektperioden, denne tidsplan er blevet anvendt til tidsstyringen af projektet og holde styr på arbejde som skal være færdigjort før at det er muligt at gå videre til en anden fase af projektet. På Figur 9 er det muligt at aflæse, hvornår en fase begynder og hvornår den skal være afsluttet ved at bruge 2 forskellige grøn-toner. Det er dog forekommet at der har skullet være ændringer i tidsplanen, hvilket betyder at der findes forskellige versioner af tidsplanen, disse kan findes i Bilag[26]. Tidsplanen som er en del af projektbeskrivelsen i Bilag[3] er dog ikke den nyeste version af tidsplanen siden den efterfølgende er blevet opdateret som en seperat fil.

Som tidligere nævnt er design, implementation og test en del af en iterativ process, hvilket har ført til at der har været anvendt et iterativt tidstyring værktøj for at holde styr på opgaver i projektet. Her har det været valgt at anvende YouTrack som et tidstyringsværktøj. Projekfaserne er indelt efter tidsplanen som vises på Figur 9 og sepperate opgaver er tilføjet i YouTrack.



Figur 10: Youtrack over implementationsfasen

Ovenstående figur 10 viser arbejdsopgaver i YouTrack, som er færdiggjort efter en fuldført implementationsfase. Resterende billeder af YouTrack kan findes i Bilag [28]. Her har YouTrack fungeret som et værktøj til både at holde styr på udestående arbejdsopgaver og som et redskab til at prioritere opgaverne. YouTrack har været et yderst effektivt værktøj til at fastholde de ændringer, der løbende skulle tilføjes projektet i den iterative proces, når der har været rettelser i eksempelvis implementeringen eller klassediagrammerne.

5 Personlige Reflektioner

I dette afsnit vil hvert gruppemedlem komme med en refleksion over, hvordan læringsudbyttet fra projektet har været. Dertil vil vi reflektere over samarbejdet og vores individuelle bidrag. Derudover vil vi nævne, hvad vi vil ændre eller bevare næste gang vi laver projekt.

5.1 Victor

Jeg mener at gruppen har håndteret projektet på en rigtig god måde især ift. at det er et første semester projekt. Blandt andet har det gode forarbejde i den indledende fase hjulpet enormt meget på at implementere i en overskuelig og hurtig proces. Her tænker jeg konkret på klassediagrammet. Samarbejdsmetoden i den indledende fase var glimrende (nævnt i indledningsafsnittet), for at spare tid på den anden fase, fordi alle havde samme tanker om, hvordan implementation skulle laves. Fremadrettet vil jeg igen anbefale mine fremtidige gruppemedlemmer, at specielt design laves i fællesskab, ligesom vi har gjort her i projektet. Jeg synes også, at logs på GitHub-aktivitet virker godt, så hele gruppen notificeres, når en ny ting er lavet. På den måde undgår man komplicerede branches eller merge konflikter. Derfor vil jeg helt klart bruge det igen. YouTrack var et virkelig godt redskab til at skabe overblik i, hvad der skulle laves og hvornår. Blandt andet kunne man til sidst på hver arbejdsdag gå igennem, hvad der skulle laves næste gang og skrive ind. Dermed vidste alle, hvad de skulle lave næste gang vi mødtes. Derudover kunne man konkretiserer tvivlsomhed, ved at snakke om et bestemt punkt på YouTrack, så man nemmere kunne hjælpe hinanden. Af de tidligere nævnte grunde kunne jeg godt tænke mig, at bruge et lignende værktøj til at danne overblik igen. Mit eget bidrag finder jeg tilfredstillende, da jeg har gjort mit bedste, for at yde mest muligt. Hertil har jeg sørget, for at hjælpe gruppemedlemmer, der havde brug for det. Jeg har også sørget for at forklare kode, som jeg har skrevet, hvis de andre ikke forstod det. Derfor føler jeg mig inkluderende og som et "godt"gruppemedlem.

5.2 Devin

Jeg synes, at projektet har været meget vellykket og uden de helt store problemer i form af konflikter i gruppen eller uenighed. I starten af projektet blev vi hurtigt enige om, hvad vi kunne forvente af hinanden. Her brugte vi gruppekontrakten til at opsætte regler for acceptabel adfærd over for hinanden og for rollen som gruppemedlem. Det kan godt være, at selve gruppekontrakten ikke havde stærkt definerede konsekvenser, men den hjalp alle i gruppen med at være på bølgelængde i forhold til forventninger og indsats i løbet af semesterets første og anden fase.
Vi startede også projektet ud med at lette samarbejdet ved at oprette en fælles Discord-gruppe, et YouTrack-board og et GitHub-repository samt en sammenkobling mellem GitHub og Discord. Det gjorde det en del nemmere at arbejde sammen, da vi alle havde adgang til de samme filer. Vi havde en god kommunikation, som betød, at vi ikke behøvede at være bange for at overskrive hinandens ændringer. Dette skyldtes både beskeder på Discord, men også en fælles forståelse for, hvor vigtig kommunikation er i et gruppearbejde.
Fremadrettet vil jeg fortsat have fokus på at starte projekter stærkt ud med fælles kommunikationsplatforme, GitHub-repositories og en god forventningsafstemning i starten af semesteret for at afklare alles mål. Jeg vil også i fremtidige grupper forsøge at bibeholde den måde, vi implementerede projektet på; her fordelte vi arbejdsbyrden, så alle deltog i implementeringen, i stedet for at lade én person stå for det hele eller dele gruppen op, så halvdelen kun programmerer og den anden halvdel kun skriver leverancer. Jeg synes personligt, at jeg har gjort mit bedste i dette projekt, har arbejdet engageret og har haft en positiv indflydelse på forløbet.

5.3 Kasra

Jeg ser tilbage på projektforløbet med stor tilfredshed, især fordi vores samarbejde har fungeret over al forventning. For mig har det været afgørende, at gruppekontrakten ikke blot var et dokument, vi skrev under på, men et værktøj vi aktivt overholdt. Da alle gruppemedlemmer mødte op med en ansvarlig, ambitiøs og punktlig indstilling, skabte det en tryghed i processen, som gjorde, at vi kunne fokusere fuldt ud på det faglige indhold.

En væsentlig årsag til gruppens succes har været vores tilgang til problemløsning og konflikthåndtering. Vi har i høj grad praktiseret demokrati og søgt enighed i fællesskab, hvilket har betydet, at vi helt har undgået konflikter. I stedet for uenigheder opstod der en kultur, hvor vi hjalp hinanden; hvis en i gruppen havde svært ved at forstå en opgave, trådte andre til med forklaringer. Dette var især tydeligt under implementeringsfasen, som jeg personligt er meget tilfreds med resultatet af.
I fremtiden vil jeg tage denne erfaring med mig som et bevis på, hvor vigtigt det sociale fundament og forventningsafstemning er for et teknisk projekt. Jeg vil i kommende projekter igen insistere på en stærk overholdelse af aftaler og en demokratisk arbejdsform, da jeg kan se, at det direkte minimerer spildtid og øger kvaliteten af det endelige produkt, som vi så det i vores implementering.

6 Refleksioner over vejledning

I dette projekt har det varieret, hvordan vejlederne er blevet brugt i de to faser, som projektet har gennemgået. I den indledende fase har vejledningen været fokuseret direkte på foranalysen, analysen og designet, hvilket har gjort udviklingsprocessen mere strømlinet. Dette har hjulpet med at fange fejl tidligt i processen og har skabt større klarhed over projektets rammer og formål. Vi har drøftet alle feedbackpunkter fra vejlederne i gruppen og taget de fleste til os. Da dette er vores første semesterprojekt, har vejledningen været særdeles brugbar til at fastlægge, hvordan de forskellige områder skulle udformes og udarbejdes..

I den anden del af projektet er vejledningen foregået via møder. Dette foregik ved at booke en tid, hvorefter der var mulighed for at stille spørgsmål til specifikke emner. Vi har i alt haft to møder – begge med Mona – da de omhandlede tvivl i forbindelse med projektrapporten. Vi var altid forberedte til disse møder og havde på forhånd nedskrevet de spørgsmål, vi ønskede at stille. Dette fungerede rigtig godt, da vi fik klare svar på vores tvivlspørgsmål, fordi de var veldefinerede på forhånd. Undervejs noterede vi altid de vigtigste pointer for at fastholde vejlederens input. Vi vil også i fremtidige projekter benytte metoden med at forberede spørgsmål både før og under vejledningen, da vi føler, at man får det største udbytte på denne måde.
Til gengæld har vi ikke afholdt møder om implementeringen, hvilket skyldes, at vores tvivl på dette område har været kortvarig. Da det ikke var muligt at få et øjeblikkeligt møde angående implementering, har vi ikke benyttet os af det, da vi løbende selv har løst udfordringerne. Vi havde naturligvis booket et møde, hvis der havde været behov for det, hvilket vi også vil gøre i fremtidige projekter, hvor kompleksiteten og dermed tvivlen sandsynligvis vil være større.

7 Konklusion

Indledningsvist har vores rige billede været mangelfuldt. Hertil blev der nævnt, at der var brug for at gennemtænke problemstillingen flere gange af denne grund. Derfor anbefaler vi at bruge meget tid på, at forstå problemstillingen og lave det rige billede.

I den indledende fase er det en god idé at benytte sig af en samarbejdsmetode, hvor alle kender til alt, der bliver lavet på projektet, ligesom den vist på figur 1. På den måde er alle på lige fod, når det angår, hvad projektet handler om og hvilken løsning, der er tiltænkt.

Bruget af værktøjer som YouTrack, GitHub og Discord har fungeret virkelig godt til projektstyring. Her har de førnævnte været specielt vigtige for overblikket af projektet. Fremdrift i projektet er blevet fremskynnet af disse værktøjer.

Afslutningsvist har vi en række anbefalinger til andre projektgrupper:

1. Det kan give mening at ændre sammarbejdsmetode alt efter, hvor i projektets fase man er.
2. Det er vigtigt at bruge noget tid i starten af semesteret på, at skrive en god gruppekontrakt. Ligeledes er det vigtigt at alle gruppemedlemmer overholder den.
3. Respekter alle gruppemedlemmer og deres mulige spørgsmål eller tvivlsomheder. Skab et godt læringsmiljø, hvor der er plads til fejl, tvivl og diskussioner. Det vigtige er, hvordan det håndteres.
4. Det er vigtigt at skabe et fælles kommunikations-netværk i starten af semesteret og fælles stedet at opbevare og dele dokumenter på.
5. Det er en god ide at anvende et værktøj som YouTrack til at holde styr på opgaver som ikke er fuldførte endnu.
6. Det er en god idé at booke vejledermøder, hvis der skulle opstå et behov.

Litteratur

bob, G. (2025). *Kløverly interview med bob sw-sep1-case, efterår 2025.* (VIA University College, Software Engineering case-materiale)

Lederweb. (u.d.). *Fem måder at håndtere konflikter.* Retrieved 2025-12-16, from https://www.lederweb.dk/fem-maader-at-haandtere-konflikter/ (Hentet December 2025 fra Lederweb)

MIT Human Resources. (u.d.). *Stages of development.* Retrieved 2025-12-16, from https://hr.mit.edu/learning-topics/teams/articles/stages-development (Hentet December 2025 fra MIT Human Resources)

VIA University College. (u.d.a). *Konflikthåndtering i gruppearbejde.* Retrieved 2025-12-16, from https://mit.via.dk/media/mitvia/studievaerktoejer/projekt-guidelines/faelles-ingenioer/gruppearbejde/konflikthaandtering (Hentet December 2025 fra VIA University College)

VIA University College. (u.d.b). *Teamtyper og roller i gruppen.* Retrieved 2025-12-16, from https://mit.via.dk/media/mitvia/studievaerktoejer/projekt-guidelines/faelles-ingenioer/gruppearbejde/teamtype (Hentet December 2025 fra VIA University College)

VIA University College. (u.d.c). *Teamudvikling og gruppedynamik.* Retrieved 2025-12-16, from https://via.itslearning.com/plans/courses/37274/plan/1170201/element/4275967 (Hentet December 2025 fra VIA University College)

8 Bilag

Bilag 1 - Projektbeskrivelse V1
Bilag 2 - Projektbeskrivelse V2
Bilag 3 - Projektbeskrivelse V3
Bilag 4 - Grønne bob analyse V1
Bilag 5 - Grønne bob analyse V2
Bilag 6 - Grønne bob analyse V3
Bilag 7 - Grønne bob analyse V4
Bilag 8 - Administrer beboer
Bilag 9 - Bytteopgaver
Bilag 10 - fællesopgaver
Bilag 11 - GrønneOpgaver
Bilag 12 - Brugermanual
Bilag 13 - Klassediagram
Bilag 14 - SekvensDiagramBytteopgaver
Bilag 15 - Wireframe af hjemmeside
Bilag 16 - Wireframe af kontaktside side
Bilag 17 - Wireframe af omside
Bilag 18 - Wireframe af hjemmeside på mobil
Bilag 19 - Gruppekontrakt
Bilag 20 - teamdynamik
Bilag 21 - konfliktstile
Bilag 22 - konflikttrappe
Bilag 23 - fælles
Bilag 24 - adaptiv
Bilag 25 - teamtyper
Bilag 26 - Tidsplan
Bilag 27 - tillpassetVandfaldsmodel
Bilag 28 - YouTrack
Bilag 29 - YouTrackVsTidsplane
Bilag 30 - rigtBillede
Bilag 29 - YouTrackVsTidsplane
Bilag 30 - rigtBillede
Bilag 29 - YouTrackVsTidsplane
Bilag 30 - rigtBillede

VIA University College

Software Engineering
Via University College
SoftwareEngineering
Banegårdsgade 4
8700 Horsens

Titel:
1. Semester projekt: Kløverly

ECTS:
10 ECTS

Semester:
1. Semester

Projektperiode:
10/09/2025 - 19/12/2025

Projektgruppe:
SEP1 Gruppe 3

Studienummer	Navn	Studieretning
362108	Devin Paul Jaworek	SW
361830	Victor Høggaard	SW
362962	Kasra Hojjatifar	SW

Vejleder:
Mona Wendel Andersen, Steffen Vissing Andersen

Sidetal: 45
Antal bilag: 30

1. Semester Projekt: Kløverly

Resume

Dette semesterprojekt omhandler udviklingen af et softwaresystem til økosamfundet Kløverly. Projektets baggrund er et behov, for at optimere den administrative byrde for samfundets administrator, Grønne Bob, som hidtil har håndteret opgaver og pointsystemer manuelt via en notesbog. Formålet med projektet er at optimere denne proces, for at skabe bedre overblik over ressourcer. Gennem en udviklingsproces bestående af analyse, design, implementering og test er der udviklet en todelt løsning. Løsningen består af en administrativ GUI-applikation udviklet i Java, som gør det muligt for administratoren at styre beboere, tildele point og administrere henholdsvis fælles-, bytte- og grønne opgaver. Derudover indeholder systemet en offentlig hjemmeside, der visualiserer samfundets fælles pointkonto, grønne tiltag og tilgængelige bytteopgaver for beboerne. Projektet konkluderer, at det udviklede system opfylder alle opstillede funktionelle og ikke-funktionelle krav. Systemet effektiviserer administrationen ved at automatisere pointlogikken. Hertil bidrager systemet til en eksponering af idéen om et økosamfund ved, at gøre indsatsen synlig for omverdenen.

Indhold

1 Introduktion

1.1 Problemfelt

Økosamfund er en bæredygtig bosætning, der eksperimenterer i en omstrukturering af samfundet, så miljøet bliver mindre belastet. Der er omkring 30 økosamfund i Danmark, og der kommer flere til. Derfor bliver det, med tiden, en mere væsentlig del af det danske samfund. I økosamfund er samarbejde mellem familier og husstande en stor del af værdierne, hvor alt fra arbejdsplads til fritid er integreret. Det er muligt at kunne opdele et økosamfund i fire centrale værdier; økologien, verdensbilledet, økonomien og det sociale (Landsforeningen for Økosamfund, u.d.; Nissen, 2016).

Den første centrale værdi økologien understøttes af, at det økologiske bofællesskab er en samling af bosteder, hvor folk deles om faciliteter og arbejder sammen om hverdagens opgaver (Den Danske Ordbog, u.d.). I det økologiske bofællesskab er der dog et større fokus på økologien. Her kan dette gøres ved lokal fødevareforsyning, grønt byggeri og diverse andre grønne tiltag (Himmerlandsbyen, u.d.).

Dernæst kommer den centrale værdi om økonomien til udtryk, fordi økosamfundet ønsker at være uafhængige af den globale økonomi og i stedet fokusere på bæredygtig udvikling. De vil gerne have lokale banker og valutaer som kunne være handelsvarer produceret i bofællesskabet. Derudover vil de gerne opnå en enkel og grøn levestil. Det kan godt skabe problematik, når de ikke ønsker at afhænge af den nationale valuta og dermed kun handle på lokal plan, fordi det kræver at samfundet har mulighed, for at dække alle de mange mulige vanskeligheder, der kunne opstå, som f.eks. at fikse et landbrugsværktøj (Himmerlandsbyen, u.d.; Landsforeningen for Økosamfund, u.d.).

Til sidst kommer det sociale aspekt til udtryk ved den fællesskabssøgende adfærd som økosamfundet har. De forsøger at skabe et kernefællesskab i økosamfundet. Det gør de på mange måder. F.eks. ved fejring af livet, ritualer og kreativitet. Hertil har de også uddannelser og andre sociale netværk. (Magasinet Natur&Miljø, 2025; Landsforeningen for Økosamfund, u.d.). .

Disse 4 centrale værdier resulterer i økosamfundet og dets fordele og ulemper. Fordele som at beboerne i samfundet arbejder sammen for at løse fælles opgaver, og hjælper hinanden ved at dele deres ressourcer og færdigheder. Her udveksles ydelser og økologiske ressourcer som betaling. Her er det kritisk at alle medlemmer i et økosamfund bidrager til det, hvilket tilføjer noget kompleksitet til organisationen af et økosamfund (Den Danske Ordbog, u.d.).

For bedre at forstå ulemperne, kigges der på Kløverly, hvilket er et lille økosamfund, der netop handler internt med bæredygtige varer og ydelser. Det kan dog være udfordrende at holde overblik i den slags betaling, og på hvad der mangler at blive lavet i lokalsamfundet for at sikre dens fremadrettede drift (bob, 2025).

I Kløverly er der en række af opgaver som udføres henholdsvis alene og i fællesskab. Opgaverne er klassificeret som grønne, individuelle, fællesopgaver og bytteopgaver. Disse opgaver belønnes ved Kløverlys eget pointsystem. Dette system administreres af en person i økosamfundet ved anvendelse af en notesbog. Organisationsværktøjet er dog ikke længere tilstrækkeligt for at holde styr på alle opgaver, som udføres i Kløverly (bob, 2025).

De tidligere nævnte grønne og individuelle opgaver er opgaver, hvor man selv gør en indsats for at fremme en bæredygtig livsstil. Det er opgaver som ændring af kost med et lavere klimaaftryk og ændring af transportmidler, som er bedre for miljøet. Fællesopgaver er de opgaver som udføres for fællesskabet og for at vedligeholde driften i Kløverly. Opgaverne kan være vedligeholdelse af fællesarealer og service til fællesarrangementer. Disse opgaver giver fællespoint til en fælles pulje. Fællespoint kan bruges til at købe ting til fællesskabet som nye værktøjer. Bytteopgaverne er de opgaver som udføres ved handel af individuelle points som generelt bruges til handel mellem point og ydelser. Lige nu bliver opgaverne ikke registreret og dermed bliver man ikke opmuntret eller inspireret til at fortsætte (bob, 2025).

1.2 Projektets formål

Dette projekt er baseret på et interview med grønne bob som har et ønske om at gøre de grønne initiativer og arbejdet i sit grønne samfund mere synlige og sin daglidag lidt nemmere. I dette projekt er det valgt at arbejde videre med bobs krav og ønsker for at give ham et produkt.

Grønne Bob har brug for et administrationssystem for at håndtere opgaverne i sit lille grønne økosamfund. Økosamfundet har 2 forskellige pointsystemer or 3 forskellige typer af opgaver bytteopgaver, fællesopgaver og grønne tiltag. Her bruges 2 forskellige point-systemer hvilke er individuelle-point og fællespoint. Fællespoint bruges som fællesskab til større projekter eller arrangementer og individuelle point bruges til handel af ydelser og ressourcer mellem hinanden.

Grønne Bob fungerer som administrator og er den eneste, der har adgang til at registrere og redigere data i systemet, mens beboerne får adgang til udvalgte oplysninger via en offentlig hjemmeside. Her kan de se kataloget over grønne tiltag, den samlede klimakonto samt aktuelle bytteopgaver.

Systemet understøtter både administrationen af beboere, point og opgaver, og samtidig formidler den landsbyens grønne initiativer udadtil på en enkel og energieffektiv måde i tråd med Kløverlys bæredygtige værdier. En mere dybdegående beskrivelse af problemfeltet kan findes i projektbeskrivelsen Bilag[3]

1.3 Problemformulering

Hvordan kan man optimere organiseringen og skabe bedre overblik over opgaver og ressourcer i økosamfundet Kløverly, og hvordan kan man inspirere indbyggere og andre i samfundet?

2 Analyse

2.1 Analyse Indledning

Der er lavet en analyse baseret på projektbeskrivelsen, som kan findes under Bilag[7], helt konkret problemfelt og problemformuleringen. Her er der lavet funktionelle krav og ikke funktionelle krav ud fra projektets øverste mål, som kan findes i afsnit 1.2. Baseret på kravene, er der lavet et use-case diagram, som er beskrevet i use-case-beskrivelser, hvilket kan findes i afsnit 2.3. Disse use-case-beskrivelser er blevet visualiseret som aktivitetsdiagrammer. De kan findes i afsnit 2.4. Domænemodellen kan findes i afsnit 2.5 sammen med ordlisten, der er lavet baseret på use-case-diagrammerne og -beskrivelserne.

2.2 Krav

Som beskrevet i Analyse Introduktion i afsnit 2.1, er der lavet krav baseret på indholdet i afsnit 1. Her er kravene opdelt i funktionelle og ikke funktionelle krav. Kravene er listet efter prioritet med ID. De funktionelle krav er sorteret efter prioritet det vil sige, at der er blevet prioriteret at gøre administratorens job nemmere ved individuel registrering af Point.

Krav 1-5 giver administratoren mulighed for at manipulere enkelte beboers data, opsøge beboer på en liste, oprette beboere og fjerne beboere. Det vurderes at være den vigtigste funktion for administratoren, da det skaber et overblik over alle beboer og medlemmer i økosamfundet.

Krav 6-9 giver administratoren mulighed for at administrere bytteopgaverne, her skal det være muligt for administratoren at kunne se bytteopgaverne, ændre i oplysninger og sætte købere på bytteopgaven. Det vurderes at være vigtigt, da det giver beboerne mulighed for at drive handel med hinanden.

Krav 10-13+24 giver administratoren mulighed for at administrere fællesopgaverne, her skal det være muligt for administratoren at se, finde, ændre i oplysninger på opgaven og færdiggøre opgaven. Det skal også være muligt at give en beboer ekstra point ved udførelse af en opgave, når administratoren ønsker at gøre dette. Det vurderes til at være vigtigt,da det giver motivation til inaktive beboere.

Krav 14-19 gør det muligt for administratoren at se, finde, ændre i oplysninger, færdiggøre og fjerne grønne opgaver. Det skal desuden være muligt for administratoren, at gøre de grønne opgaver og den samlede grønne kontopulje synlig for beboerne. Det vurderes at være vigtigt, da det gør det muligt for beboerne at få et nemt tilgængeligt overblik, over hvad de kan udrette.

Krav 20 gør det muligt for beboerne, at se tilgængelige bytteopgaver uden at skulle kontakte administratoren direkte.

Krav 26+23 er 2 ikke funktionelle krav som står for at give brugeren mulighed for at kunne se fællespoint kontoen, tilgængelige bytteopgaver og grønne tiltag synlige på en hjemmeside samt regler for, hvordan hjemmesiden skal opføre sig.

2.2.1 Funktionelle krav

1 Som administrator ønsker jeg at kunne hente oplysninger på beboere og kunne tilgå dem på en liste, for at kunne holde styr på, hvem der er i systemet.

2 Som administrator ønsker jeg at registrere beboere ved navn, fødselsdagsdato, husnummer og køn, for at jeg kan tildele dem point og opgaver.

3 Som administrator ønsker jeg at kunne søge i beboere efter navn, alder og køn, for at kunne administrere deres oplysninger.

4 Som administrator ønsker jeg at fjerne en beboer i tilfælde af udflytning ved at bruge navn alder og køn.

5 Som administrator ønsker jeg at kunne tildele og fjerne individuelle point til en bruger. For at kunne rette på den opsøgte brugers individuelle point.

6 Som administrator ønsker jeg at kunne hente oplysninger på bytteopgaver og kunne tilgå dem på en liste, for at kunne se status på nuværende bytteopgaver.

7 Som administrator ønsker jeg at kunne tilføje en bytteopgave for at kunne tildele opgaven til en eller flere brugere ved navn på opgaven, beskrivelsen, dato, antal point og den, der tilbyder ydelsen, for at kunne holde styr på opgaven og hvem der skal modtage point og hvem der skal bruge point.

8 Som administrator ønsker jeg at kunne søge på en bytteopgave ved brug af navn på opgave, navn på beboer eller dato af opgaven. For at kunne tilgå den specifikke opgave.

9 Som administrator ønsker jeg at kunne administrere bytteopgaverne for færdiggøre, slette dem og tilføje/fjerne købere af ydelsen.

10 Som administrator ønsker jeg at kunne hente oplysninger på fællesopgaver og kunne tilgå dem på en liste for at kunne se information over de nuværende fællesopgaver og hvilke fællesopgaver der er for at kunne have et overblik.

11 Som administrator ønsker jeg at kunne tilføje en fællesopgave ved navn, beskrivelse, dato og antal individuelle point, for at kunne tildele denne opgave til beboere.

12 Som administrator ønsker jeg at kunne søge på en fællesopgave ved brug af opgavetitel eller slutdato af opgaven. For at kunne tilgå den specifikke opgave.

13 Som administrator ønsker jeg at kunne administrere en opsøgt opgave for at kunne tilknytte eller fjerne beboere til fællesopgaven, ændre dato på opgaven, ændre point som opgaven giver, slette opgaven og for at færdiggøre opgaven.

24 Som administrator ønsker jeg at kunne tildele bonus, når beboere tilknyttes til en opgave, ved at give ekstra point for at motivere beboerne.

14 Som administrator ønsker jeg at kunne hente oplysninger på grønne opgaver og kunne tilgå dem på en liste for at kunne se dato, navn på opgaven og fællespoint for opgaverne.

15 Som administrator ønsker jeg at kunne tilføje en grøn opgave til opgavekataloget ved opgavetitel, beskrivelse og point. For at kunne administrere de grønne opgaver som bliver udført af beboerne samt at kunne vise disse opgaver til beboerne.

16 Som administrator ønsker jeg at kunne søge på en grøn opgave ved brug af navn eller dato af opgaven. For at kunne administrere den specifikke opgave.

17 Som administrator ønsker jeg at kunne administrere en opsøgt opgave, for at kunne færdiggøre opgaven og tildele points til den grønne fælles konto.

18 Som bruger ønsker jeg at kunne se kataloget over grønne tiltag for at danne mig et overblik over mulige tiltag.

19 Som bruger ønsker jeg at kunne se antal fællespoint i klimakontoen for at vide, hvor meget der er blevet optjent.

20 Som bruger ønsker jeg at kunne se tilgængelige bytteopgaver for at danne mig et overblik over mulige ydelser.

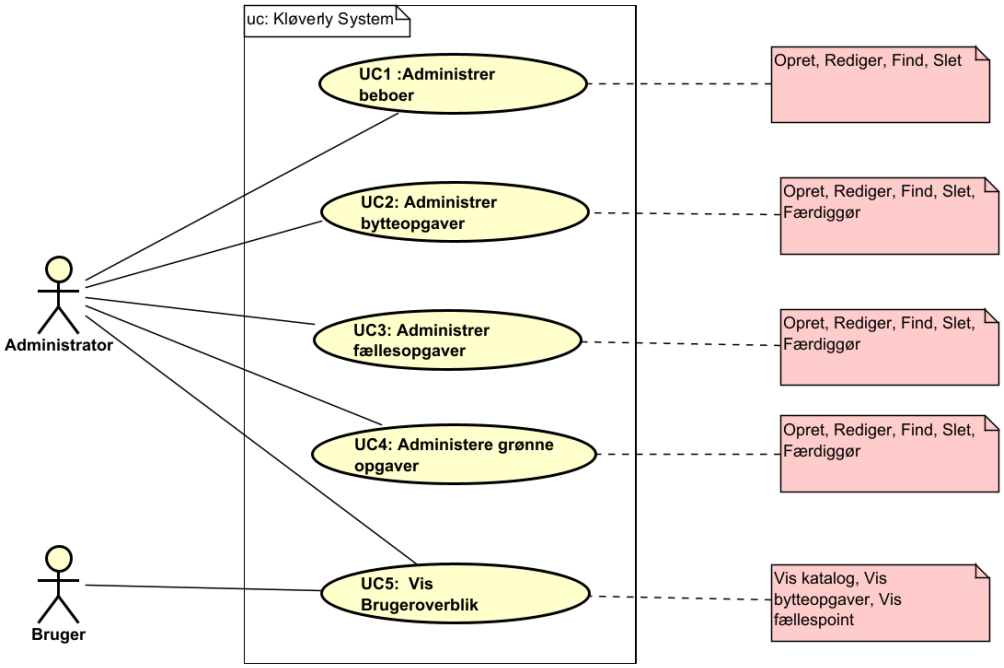
2.2.2 Ikke funktionelle krav

26 Der skal være en hjemmeside, hvor kataloget over grønne tiltag, fællespoint i klimakontoen og tilgængelige bytteopgaver bliver vist.

23 Hjemmesiden skal være responsiv og være tilpasset til brug på forskellige enheder

2.3 Use-cases

Der er lavet use-cases for Kløverly systemet, disse use-cases er baseret på de funktionelle krav som beskrevet i afsnit 2.2. De use-cases som er lavet, kan ses på nedenstående Figur 1 .



Figur 1: Use case diagram

Figur 1 viser use-case-diagrammet for kløverly her er der 5 use-cases og 2 aktører samt noter for hver use-case som beskriver dens vigtigste funktioner. Hver use-case har sin egen rolle og Administratoren har adgang til alle use-cases, hvor brugeren/beboerne kun har adgang til UC5: Vis Brugeroverblik, hvilket giver dem adgang til at kunne se bytteopgaverne, grønne tiltag og point-fælleskontoen. De andre 4 use-cases gør det muligt for Administratoren at oprette beboere, give dem opgaver, færdiggøre opgaver, redigere opgaver og slette opgaver. Det er her vigtigt at beboerne/brugerne slet ikke har adgang til de andre use-cases så de ikke selv skal kunne oprette eller færdiggøre opgaver.

Use-case	Dækkede krav
Administrer beboere	Denne use-case dækker kravene 1, 2, 3, 4, 5.
Administrer bytteopgaver	Denne use-case dækker kravene 6, 7, 8, 9.
Administrer fællesopgaver	Denne use-case dækker kravene 10, 11, 12, 13, 24.
Administrer grønne opgaver	Denne use-case dækker kravene 14, 15, 16, 17.
Vis brugeroverblik	Denne use-case dækker kravene 18, 19, 20.
	Ikke-fuktionelle krav (25, 26) er ikke direkte relaterede til use-cases og dækkes på anden vis af systemet.

Tabel 1: Dækkede use-cases

Ovenstående Tabel 1 viser hvordan kravene dækkes af alle use-cases. Her er det vigtigt at understrege, at alle krav er dækket af disse use-cases.

<table>
<tr><td>Use-case 2</td><td>Administrer bytteopgaver</td></tr>
<tr><td>Resumé</td><td>Opret, fjern, find eller rediger bytteopgaverne ved at tilknytte beboere til opgaven. Her kan redigeres point, datoen og beskrivelsen på opgaven samt sættes afsender og modtager af opgaven. Opgaven kan findes ved at søge på navn på beboer eller slutdato på opgaven. Opgaven kan også færdiggøres.</td></tr>
<tr><td>Aktører</td><td>Administrator</td></tr>
<tr><td>Forudsætninger</td><td></td></tr>
<tr><td>Post-betingelser</td><td>En bytteopgave er enten blevet fjernet, oprettet, redigeret eller færdiggjort. Opdateringer er både i systemet og bliver gemt.</td></tr>
<tr><td>Basis-sekvens</td><td>1. Systemet viser en liste over alle aktuelle bytteopgaver. Listen er sorteret alfabetisk. Her vises navn, dato og antal point på opgaven for, hver opgave. Hertil vises en mulighed for at slette bytteopgaven, redigere, færdiggøre opgaven. Samt en mulighed for at oprette en bytteopgave.
2. HVIS OPRET gå til scenario A trin A.1
HVIS REDIGER gå til scenario B trin B.1
HVIS SLET gå til scenario C trin C.1
HVIS SØG gå til scenario D trin D.1
HVIS FÆRDIGGØR gå til scenario E1
<table><tr><td>SCENARIO A: OPRET</td><td>A.1 System åbner nyt vindue, som viser administrator mulighed for at udfylde beskrivelse, navn, slut-dato, antal point. Systemet viser muligheder som kan søges i for at finde sælger af opgaven.
A.2 Administrator søger efter oplysninger for bytteopgaven.
A.3 System validerer navn, beskrivelse, dato og sælger af bytteopgaven.
A.4 System validerer information [ALT1]
A.5 Bytteopgave er tilføjet til bytteopgavelisten og vises sammen med alle andre bytteopgaver.</td></tr></table>
<table><tr><td>SCENARIO B: REDIGER</td><td>B.1. System giver mulighed for at tilføje/fjerne køber af ydelsen. System giver mulighed for at opdatere oplysninger på opgave.
B.3 Hvis tilføj/fjerne køber af ydelsen.</td></tr></table></td></tr>
</table>

	SCENARIO BB: tilføj fjerne køber	BB.1 System viser købere og viser mulighed for at tilføje og fjerne køber. BB.2 Administrator vælger at tilføje køber til bytteopgaveALT 3] BB.3 System viser beboere i Kløverly. BB.4 Administrator vælger beboer som tilføjes. BB.5 System viser beboer/e som er tilknyttet til bytteopgaven. B5. System viser data som kan redigeres. Slut-dato, navn, beskrivelse og antal point. B6. System validerer nye oplysninger. [ALT1] B7. System viser alle aktuelle bytteopgaver.
	SCENARIO C: SLET	C1. System validerer om administrator vil slette opgaven. C2. Administrator vælger Ja. [ALT3] C3. Opgave fjernes fra aktuelle bytteopgaver. C4. System viser aktuelle bytteopgaver
	SCENARIO D: SØG	D1. System giver mulighed for at søge. D2. Administrator taster navn på bruger/navn på opgave/ dato på opgave. D3. Så snart der er et nyt input opdateres listen ved søgeargumentet.
	Scenario E Færdiggør	E1. Systemet viser brugeren opgaven. E2. System giver bruger mulighed for at færdiggøre opgaven. E3. System validerer om brugeren gerne vil færdiggøre opgaven. E4. Opgaven fjernes fra oversigt over opgaver og beboere tiknyttet til opgaven får fjernet/tildelt point.
Alternative sekvenser	[ALT1] Validation ikke godkendt administrator system giver besked. [ALT2] Opgave fuldføres ikke administrator får vist alle aktuelle bytteopgaver. [ALT3]

	3.1 Administrator vælger at fjerne køber fra mulige købere. 3.2 System fjerner køber fra bytteopgaven og viser mulige tilbageværende købere. [ALT4] Bruger kunne ikke findes system viser "ingen bruger med denne information".
Bemærkninger	Denne use-case dækker kravene 6, 7, 8, 9.

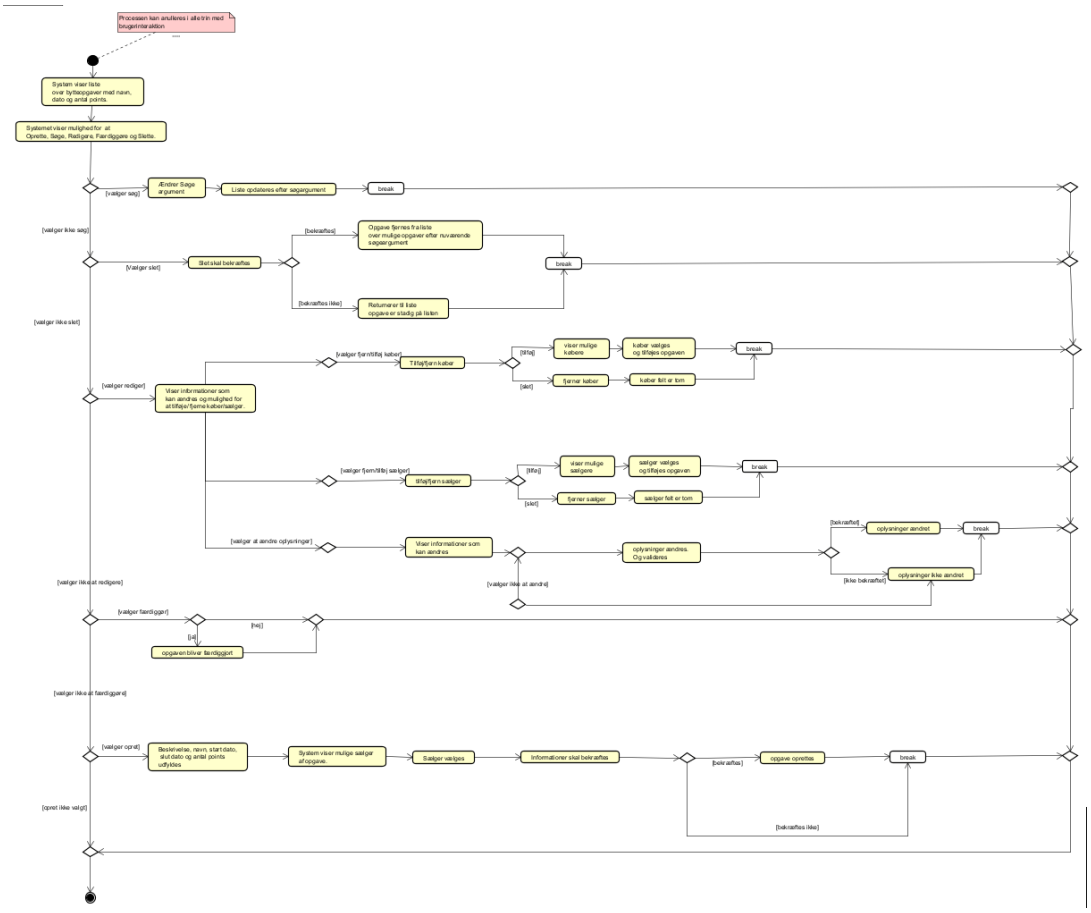
Figur 2: Use case beskrivelse for use-case 2

Figur 2 viser use-case-beskrivelsen for use-case 2. Den viser hvordan 2 beboere kan lave en handel med hinanden. Det er lavet med tanken om, at man skal kunne udføre alle scenarier på, hvilket som helst tidspunkt.

Use-case diagrammet er opbygget således at aktøren kan se alle bytteopgaver i systemet og har mulighed for at trykke OPRET, REDIGER, SLET, SØG og FÆRDIGGØR. Når administratoren er inde i REDIGER Scenariet er det kun der han skal kunne tilgå subscenariet BB.
For alle use-cases fra 2-4 er det vigtigt at postbetingelsen er sande sådan at man f.eks ikke kan færdiggøre en opgave flere gange og derved fjerne/tilføje for mange point til en beboer. De resterende use-cases kan findes under Bilag[7].

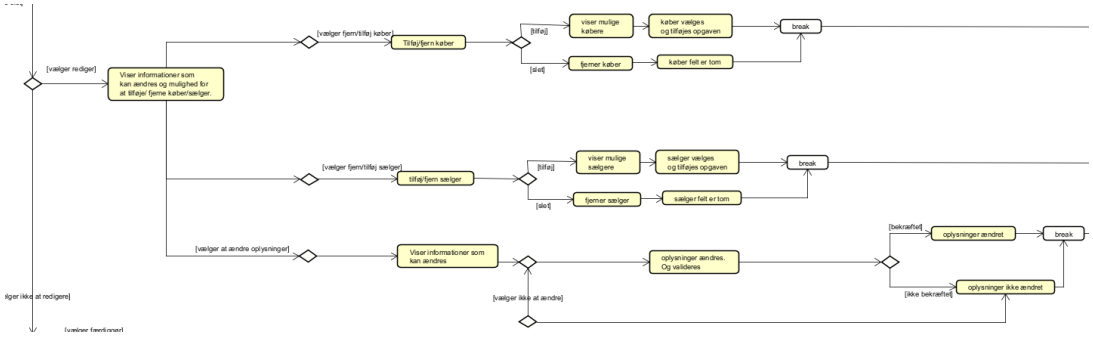
2.4 Aktivitetsdiagrammer

Der er lavet aktivitetsdiagrammer for Kløverly Systemet herunder vises diagrammet for bytteopgaver, de andre diagrammer kan findes under Bilag[8-11]. Diagrammerne er lavet med udgangspunkt i kravene, som er beskrevet i afsnit 2.2. Her har aktivitetsdiagrammer lignende funktion som use-case-beskrivelserne og beskriver, hvordan programmet skal opføre sig overfor brugeren.



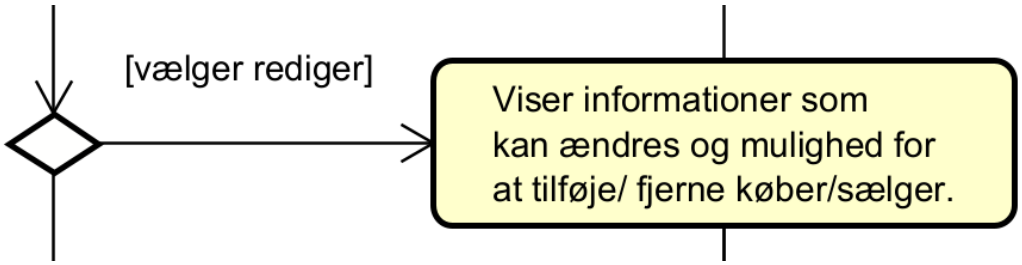
Figur 3: Aktivitetsdiagram

Figur 3 viser hele aktivitetsdiagrammet for bytteopgave use-casen. Her ses at der er 5 forskellige muligheder for administratoren at benytte use-casen. Her vælges dog at fokusere på det, der svarer til Scenarie B i afsnit 2.3 Figur REDIGER 2.



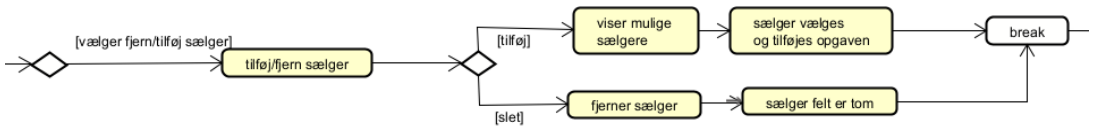
Figur 4: Aktivitetsdiagram udsnit rediger

På Figur 4 ses, hvad der sker, hvis administratoren vælger at trykke på Rediger. Administratoren har 3 forskellige muligheder inde i Rediger: Tilføj/fjern køber, Tilføj/fjern sælger og vælge at ændre oplysninger. Her er det interessant at kigge på Rediger siden, at den har disse 3 Scenarier som implementeres senere, i afsnit 4 Implementering.



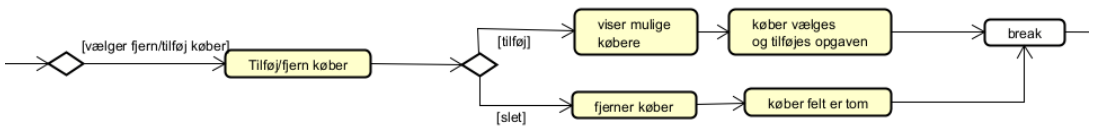
Figur 5: Aktivitetsdiagram udsnit som viser rediger

Som ses på ovenstående Figur 5 når administratoren vælger rediger funktionen, vises en mulighed for at ændre data vedrørende bytteopgaven, eller justere på sælger og køber ved enten at sætte sælgeren eller tilføje/fjerne købere.



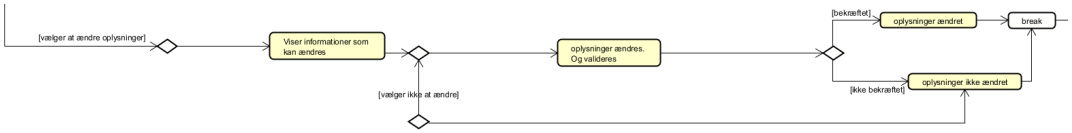
Figur 6: Aktivitetsdiagram udsnit tilføj/fjern sælger

Ovenstående aktivitetsdiagram på Figur 6 viser, hvad der sker hvis at administratoren vælger at gå videre med tilføj/fjern sælger, hvis administratoren vælger at tilføje en sælger på bytteopgaven vil sælgeren tilføjes til opgaven og administratoren vil returneres til starttilstanden, hvor administratoren igen kan vælge en af de 5 mulige startscenarier som kan ses på Figur 3.



Figur 7: Aktivitetsdiagram udsnit tilføj/fjern køber

Ovenstående udsnit af aktivitetsdiagrammet på Figur 7, viser scenariet for at tilføje/fjerne købere. Som i udsnittet af aktivitetsdiagram på Figur 6 er der 2 muligheder, enten tilføj eller fjern, hvor Administratoren herefter vil blive sendt tilbage til de 5 mulige startscenarier.

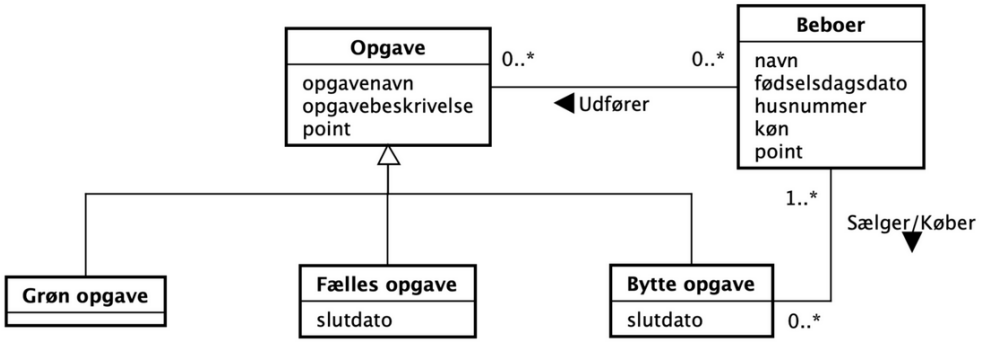


Figur 8: Aktivitetsdiagram rediger oplysninger på bytteopgave

Den sidste mulighed som Administratoren har, når rediger scenariet er valgt, er at redigere selve bytteopgaven. Som ses på ovenstående figur (8) vises de oplysninger som kan ændres, hvorefter der valideres om administratoren vil ændre bytteopgaven, hvis der vælges at fortyde ændringen vil administratoren blive sendt tilbage til starttilstanden, hvor systemet viser de oplysninger som kan manipuleres. Administratoren kan vælge at bekræfte oplysningerne i stedet for, hvis dette er tilfældet vil bytteopgaven opdateres og administratoren vil sendes tilbage til starten, hvor der kan vælges mellem de 5 hovedscenarier.

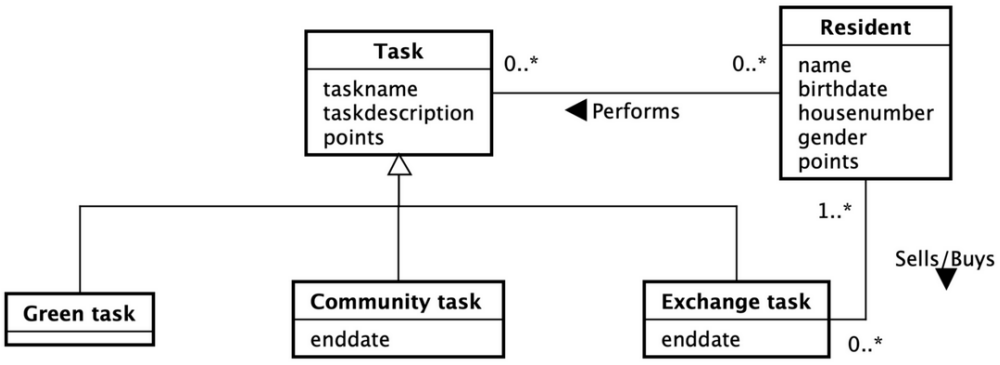
2.5 Domænemodel og Ordliste

Der er lavet en domænemodel for Kløverly, denne domænemodel er baseret på de krav som introduceres i afsnit 2.2 og de use-cases som introduceres i afsnit 2.3.



Figur 9: Domænemodel dansk

Den ovenstående Figur 9 viser domænemodellen for Kløverly, som tager udgangspunkt i kravene og use-cases og simplificerer dem ned til 5 klasser, her er det interessant at bemærke, at bytte-opgaver, fællesopgaver og grønne opgaver er en type af opgave. Hver af disse opgavetyper har nogle karakteristika som gør, at de ikke er ens, de deler dog allesammen, at de har en opgave-beskrivelse, opgavenavn og point. Ligeledes kan der ses, at der er en beboer, som kan sælge og købe en bytteopgave, og udføre en opgave.



Figur 10: Domænemodel engelsk

Ovenstående domænemodel er identisk til domænemodellen på figur 9 funktionsmæssigt, dog ikke i navn. Da den er oversat til engelsk. Her er der lavet en ordliste fra dansk til engelsk som anvendes.

Analyse termer (Dansk)	Design Termer (Engelsk)	Beskrivelse
Opgave	Task	Opgave er et blueprint til de andre opgaver.
Beboer	Resident	Beboer er de personer som udfører opgaver og handler med opgaver.
Fællesopgave	Community Task	Fællesopgaver er de opgaver som laves for at optjene points.
Grøn opgave	Eco Task	Grønne opgaver er tiltag som vises de har dog ikke indflydelse på andre dele af diagrammet.
Bytte Opgave	Exchange Task	Bytteopgaver kan forhandles

Figur 11: ordliste engelsk til dansk

Ovenstående Figur 11 viser ordlisten for domænemodellen fra dansk til engelsk, her findes beskrivelser til alle klasser i domænemodellen og deres funktion.

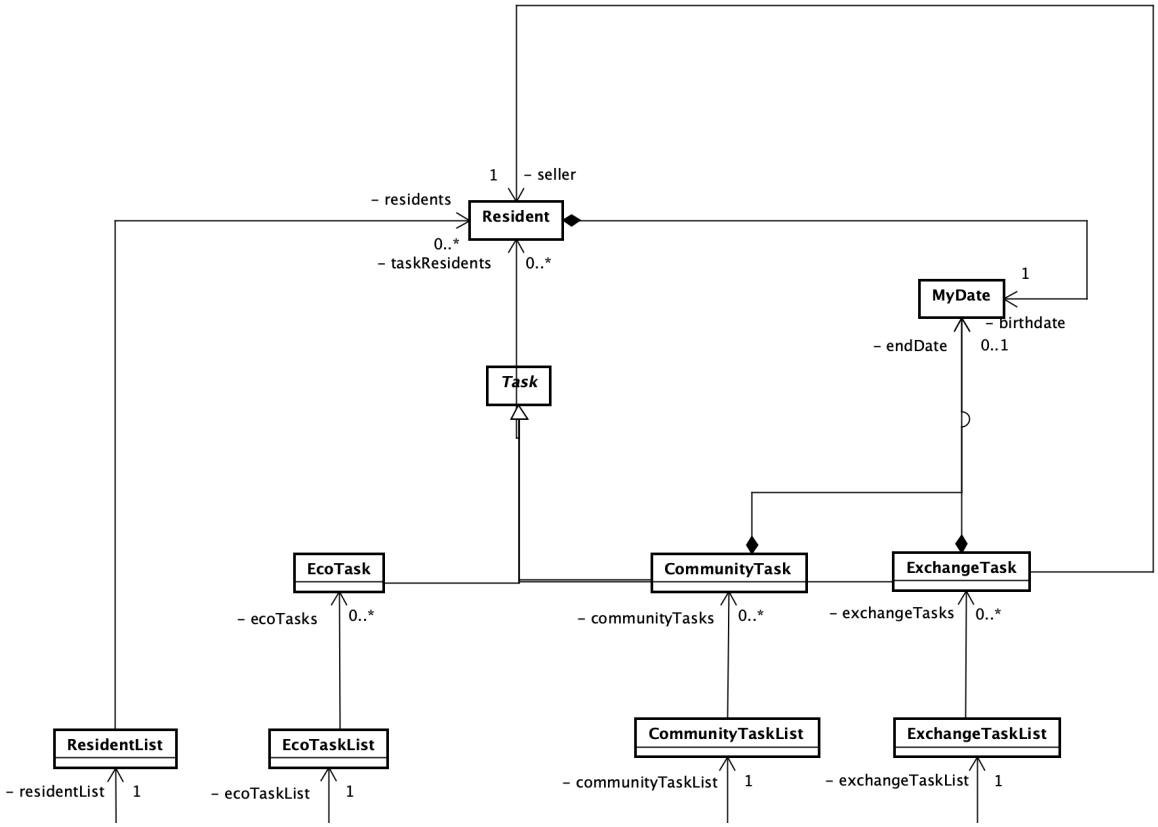
3 Design

3.1 Design introduktion

Baseret på analyse (afsnit 2) laves der design. Helt konkret tages der udgangspunkt i den engelske domænemodel, som blev en første version af klassediagrammet. Den engelske domænemodels klasser blev også de første klasser i klassediagrammet. Hertil opdateres den med attributterne fra domænemodellen. Dog tilføjes hertil alle metoder, der vil give mening at have på hver klasse, samt ekstra klasser til at skabe logikken bag programmet. Domænemodellen er en lille del af modellen, som er skåret helt ind til benet. Dette bliver også klart senere i afsnittet.

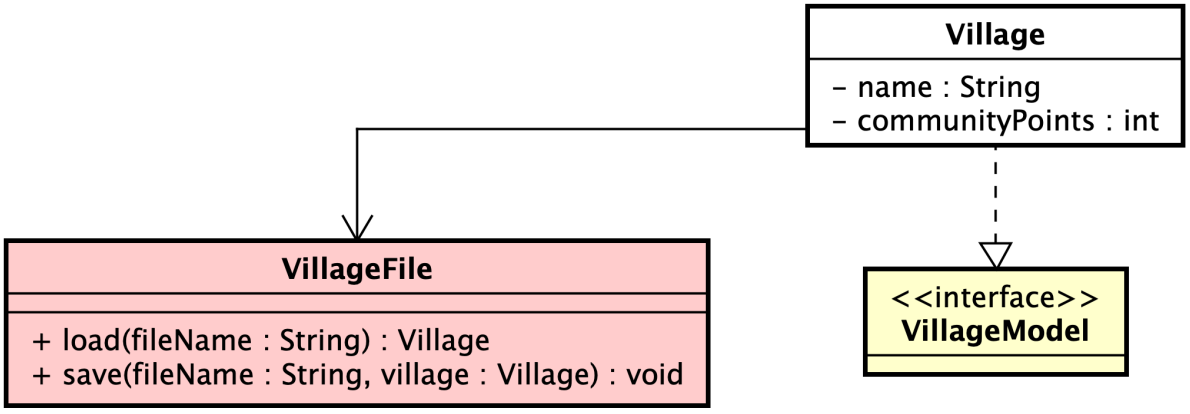
3.2 Design klassediagram

Til en start kigges der på klassediagrammet. Der tages udgangspunkt i den engelske domænemodel fra afsnit 2.5. Til domænemodellen tilføjes yderligere attributter, klasser og metoder. Der skelnes mellem generelle og specifikke klasser. F.eks. er en overordnet klasse til opgaver generel, hvorimod en specifik opgavetype som klasse er specifik. Dette kan bruges til, hvordan klasserne skal hænge sammen med associations. Dernæst bruges princippet om enkelt ansvar. Dette omhandler at én klasse skal have ét formål. På baggrund af det laves liste-klasser for beboer-klassen og alle typer af opgaver. Hertil tilføjes metoder, der evt. kan være anvendelige, dette inkludere setters, getters, equals og tostring.



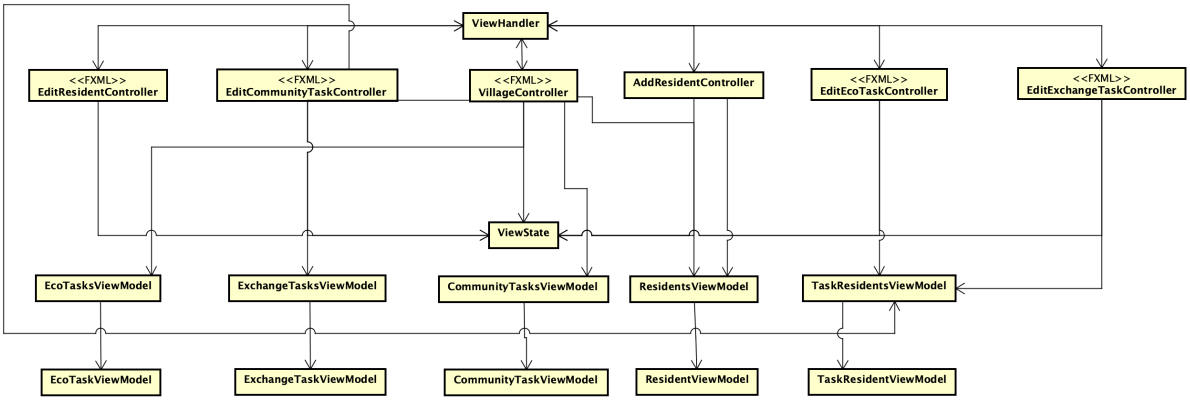
Figur 12: Klassediagram: Model-pakken

På den ovenstående figur (12) vises en oversigt over model-pakken i klassediagrammet. Model-pakken har til formål, at sørge for at alt logikken bag programmet virker. Her er alle metoder og instansvariabler gemt væk, for at der kan dannes et overblik over, hvordan model-pakken hænger sammen. Grundlæggende hænger det sammen som 4 forskellige lister af forskellige objekter. Hvilket består af beboere, grønne opgaver, fælles opgaver og bytteopgaver. Hvis det er en opgavetype, så nedarver det fra opgave-klassen. Opgaveklassen har en liste af beboere, der kan have fra 0 til mange beboere.



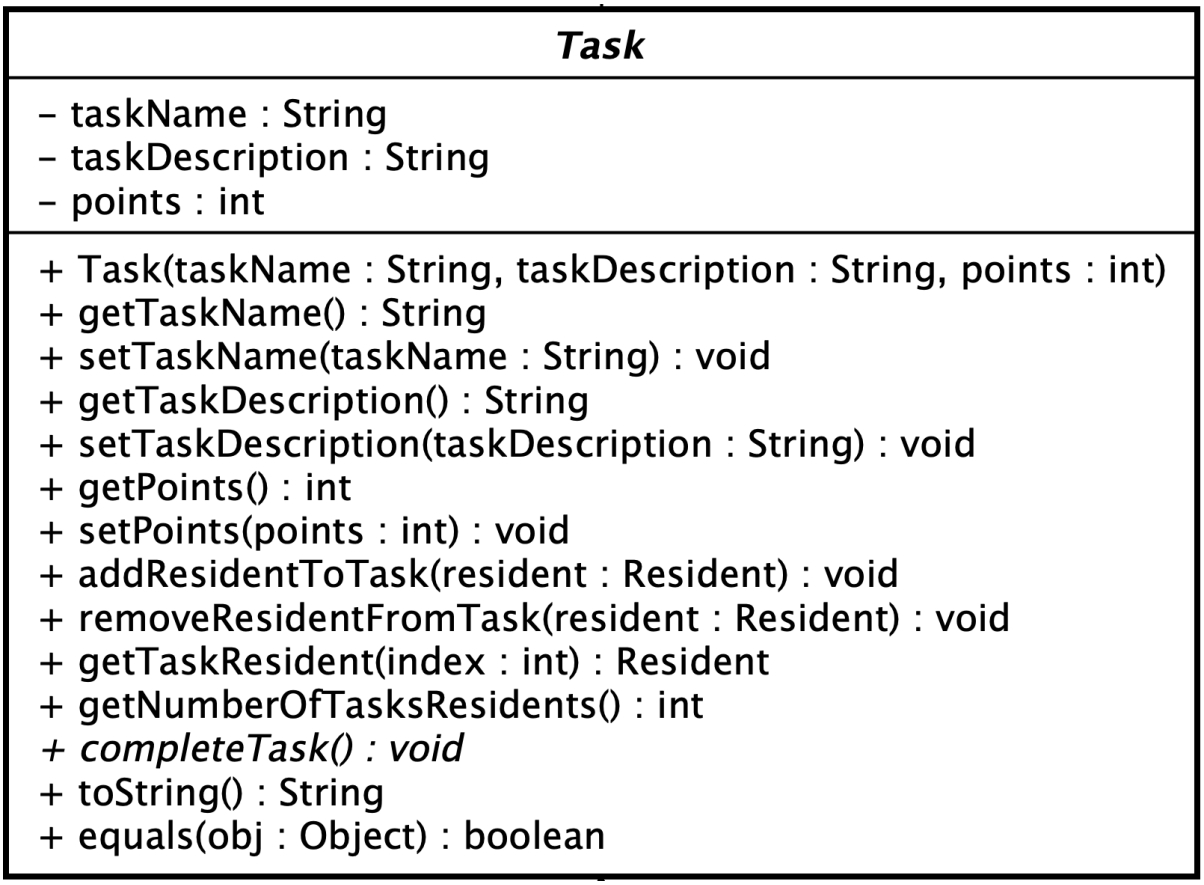
Figur 13: Klassediagram: Mediator-pakken

Figur 13 viser en oversigt over mediator-pakken i klassediagrammet. Her er metoder i Village og VillageModel gemt væk, for at skabe et let overblik over pakken. Mediator pakken skal ses som pakken, der styrer kommunikationen mellem model-pakken og view-pakken (bliver introduceret om lidt). Den kobler derfor de sidste led af model-pakken sammen, og skaber det som variabler og metoder, der kan bruges af view-pakken. Village og VillageFile har egenskaber som er pakket ind i VillageModel interfacet, for at skabe en tydelig definition af, hvad Village og VillageFile skal kunne. Hertil for at forhindre direkte afhængighed mellem lagene i mediator og view.



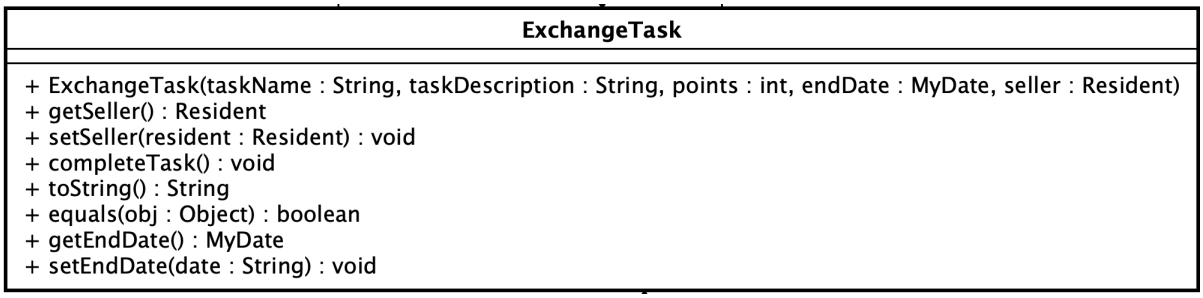
Figur 14: Klassediagram: View-pakken

På ovenstående figur (14) vises en oversigt over view-pakken. Formålet her er at vise brugeren af programmet noget som f.eks. data. Derudover at modtage input fra brugeren. Den benytter sig af mediator klassen, for at benytte modellen, som ved hvad den skal gøre ved forskellige handlinger af brugeren. Først og fremmest er der en ViewHandler klasse, formålet ved denne klasse er at den kan ændre på, hvad der bliver vist på skærmen. Dernæst haves en controller, for hver eneste FXML-fil (view). Hver gang der skal bruges en liste til et TableView i GUI, er der brugt to ViewModel klasser, en til at opbevare en ObservableList, og en til at opbevare de forskellige properties til listen. Klassen med ObservableList opbevarer en liste af objekter med de properties der er deklareret, derfor en instans af den anden klasse. Derudover er der forskellige basisoperationer i begge klasser, bl.a. til at tilføje, fjerne og hente objekter fra listen, samt at hente properties fra objektet. Dette gør at listerne kan opdateres med direkte påvirkning på, hvad der bliver vist. Til sidst haves en ViewState klasse, der skal holde styr på, hvilke variabler og objekter de forskellige controllers skal kunne hente fra hinanden, for at fungere. Dette gør at controllers ikke får adgang til unødvendigt meget fra hinanden, men kun det absolut nødvendige.



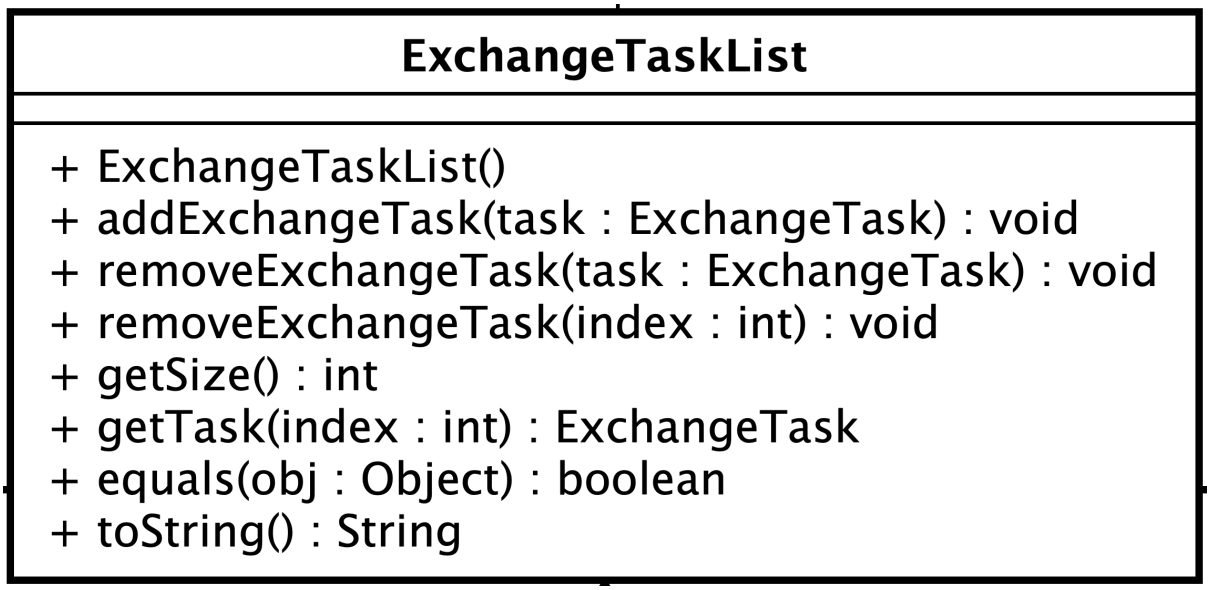
Figur 15: Task klassen

På den ovenstående figur (15) vises klassen "Task", som er vist med alle metoder og instansvariabler. Dette er en super-klasse til de forskellige typer af klasser, fordi de alle sammen har nogle ting til fælles, som kunne samles i denne klasse. Derudover er den abstrakt, fordi den ikke skal kunne oprettes. Der ønskes kun instanser af de forskellige typer af opgaver. Alle opgaver har som vist på figur 3 et navn, en beskrivelse og et antal point. Dog har disse antal point forskellige betydninger, for de forskellige typer af opgaver. F.eks. er det i EcoTask antal point, der skal tilføjes til fælleskontoen, efter udførelse, hvorimod det for ExchangeTask er, hvor meget det koster at købe ydelsen.
Hertil kan der findes de mange metoder, som alle opgaverne deler. Dette giver mulighed, for at kunne hente oplysninger om titlen, beskrivelsen og point, eller at sætte dem til noget nyt. Derudover er der også metoder til, at behandle beboere på opgaven, hvilket indebærer at tilføje, fjerne eller hente en beboer på opgaven. Til sidst er der en metode "completeTask", der skal færdiggøre opgaven. Den er abstrakt, da den skal gøre noget forskelligt i alle typer af opgaver. Ved EcoTask skal den nulstille listen af beboere på opgaven og give point til fælleskontoen, hvorimod den ved CommunityTask skal give individuelle point, som er en variabel på hver beboer.



Figur 16: ExchangeTask klassen

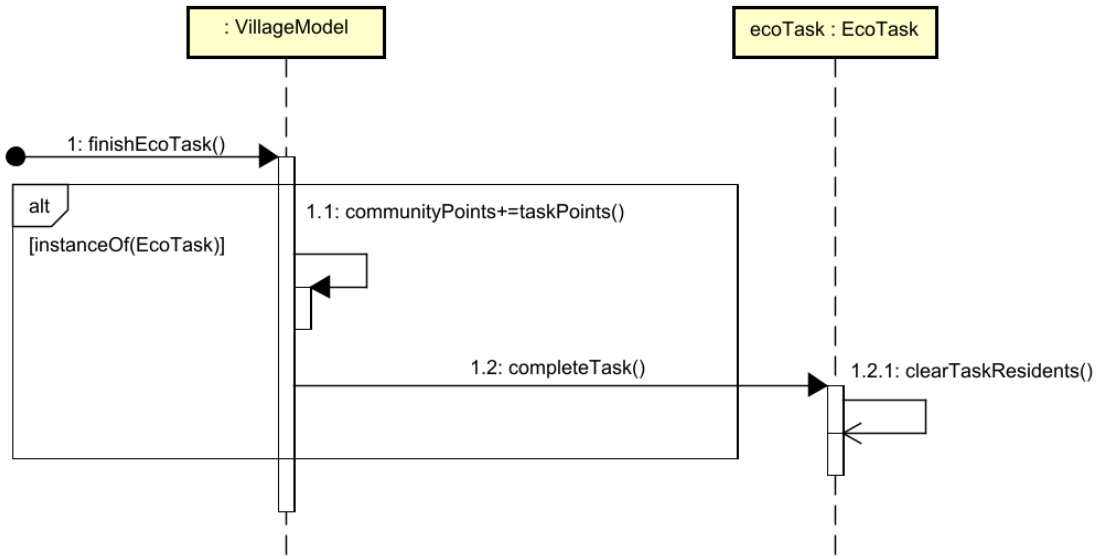
På den ovenstående figur (figur 16) vises klassen "ExchangeTask", med alle metoder og instansvariabler. Dette er en klasse der nedarver fra "Task-klassen. Den har en association til "Resident-klassen, for at have en sælger-variabel. Der kan kun være præcis 1 sælger. De forskellige variabler som er ekstraordinære fra super-klassen kan sættes og hentes fra setters og getters, som f.eks. "setSeller"og "getSeller"metoderne. Derudover implementerer den, den abstrakte metode "completeTask", der er deklareret i super-klassen. Her skal den kunne fjerne point fra købere og give dem til sælgeren.



Figur 17: ExchangeTaskList klassen

På den ovenstående figur (figur 17) vises klassen "ExchangeTaskList". Her er der inkluderet alle metoder og instansvariabler. Klassen har en association med 0 til mange, til "ExchangeTask-klassen. Dette betyder, at den skal have en liste over objekter bygget på "ExchangeTask-klassen. Dertil er der metoder til at tilføje, hente og fjerne bytteopgaver til listen. Herudover at få størrelsen af listen (hvor mange bytteopgaver der er).
De andre klasser af opgavertyper, samt beboer-klassen er bygget op på nogenlunde samme vis.

3.3 Design sekvensdiagram



Figur 18: Sekvensdiagram: Færdiggør grøn opgave

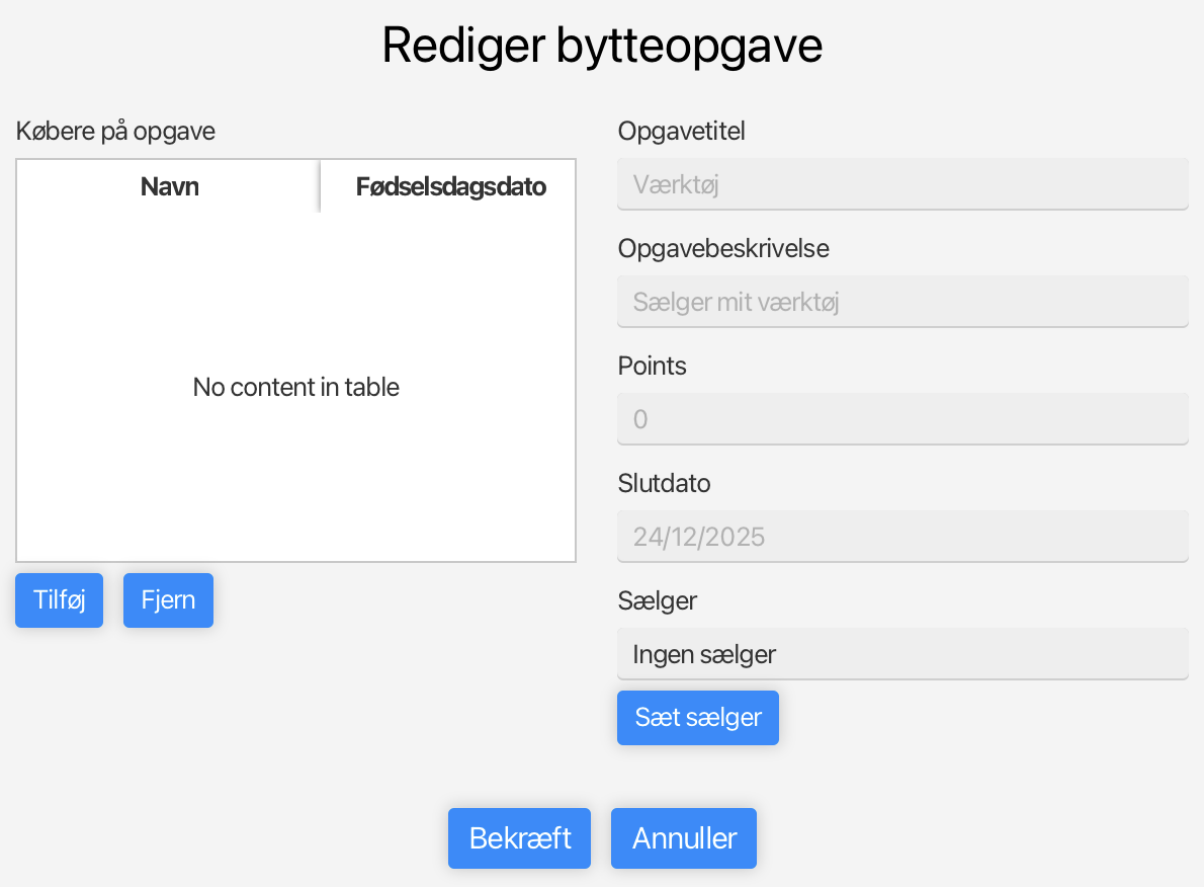
På den ovenstående figur (18) vises et sekvensdiagram for færdiggørrelse af en grøn opgave. Dette sekvensdiagram starter, når en grøn opgave bliver færdiggjort i view-pakken. Det starter i VillageController i metoden finishEcoTask(), som kalder metoden completeTask() i interfacet VillageModel. Den finder derefter ud af, hvilken type opgave det er ved hjælp af instance of EcoTask, hvis den er en EcoTask, så færdiggører den, den grønne opgave. Det sker ved først at tilføje point fra opgaven multipliceret med antallet af beboere på opgaven, til fælleskontoen. Derefter kalder den completeTask() metoden på EcoTask objektet som kalder metoden clearTaskResidents(), som fjerner alle beboere på opgaven.

3.4 Design GUI

Dette delafsnit vil tage udgangspunkt i de fleste FXML-filer, der er blevet lavet til GUI ud fra, hvad der er i view-pakken fra tidligere i afsnittet. Desuden er der lavet en skræddersyet CSS til dem, for at gøre det tydeligere at skelne mellem elementerne i designet.

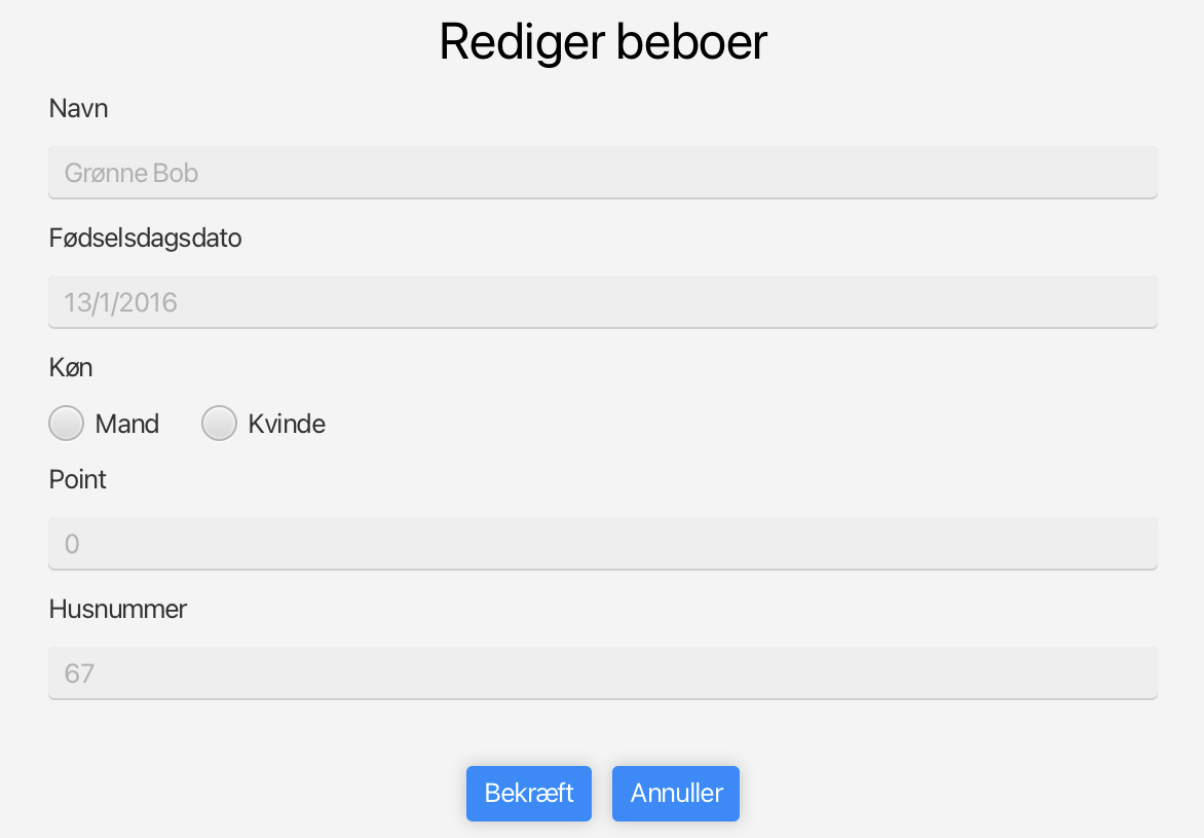
Figur 19: GUI: Hovedmenu

På figur 19 vises en FXML-fil åbnet i SceneBuilder for hovedmenuen. Hovedmenuen består af et TabPane med 4 forskellige Tabs som er beboere, bytteopgaver, fællesopgaver og grønne opgaver. Hver tab har et TableView med TableColumns der passer til de informationer, der skal oplyses om hver opgave i hovedmenuen. Derudover er der nederst i hver Tab et InputField, som skal bruges til at søge i listen. Hertil er der knapper til handlinger på listen. Disse handlinger består af opret, slet, rediger og for opgaver også færdiggør.



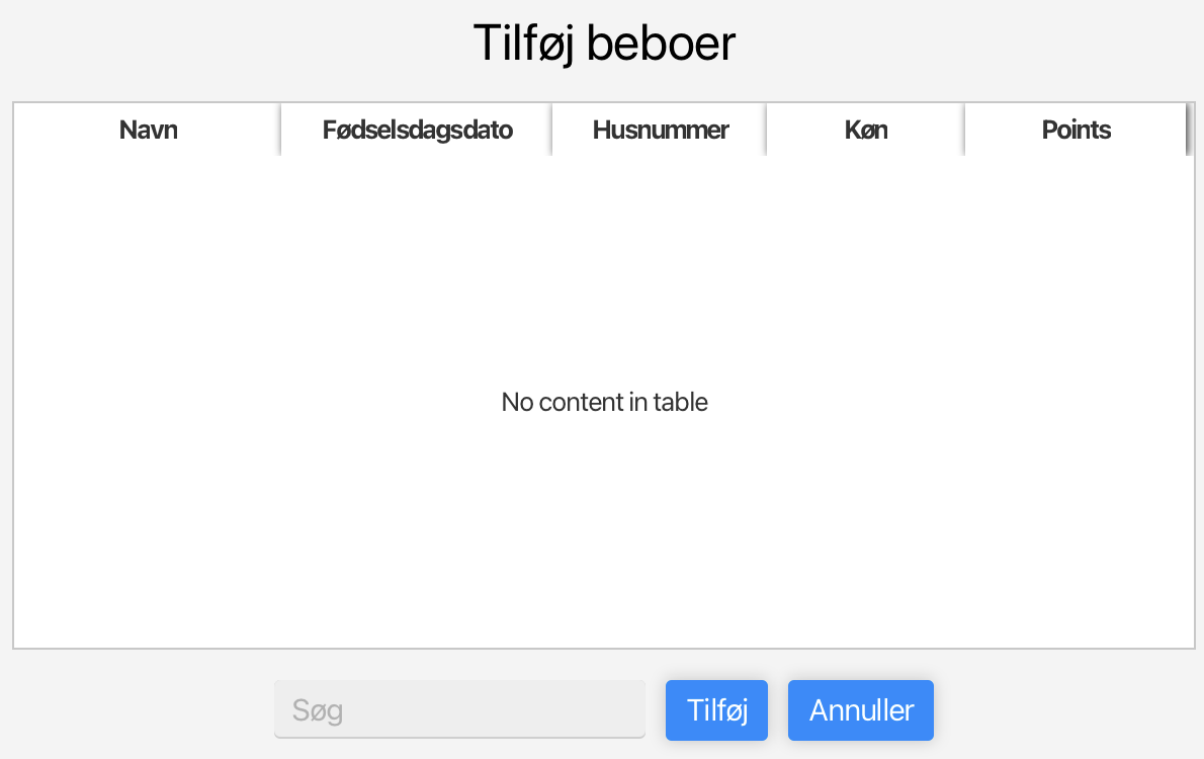
Figur 20: GUI: Bytteopgave

På figur 20 vises en FXML-fil åbnet i SceneBuilder, for at tilføje, redigere eller oprette en bytteopgave. GUI'en består af InputFields for hver instansvariabel til en bytteopgave. Hertil er der et TableView til at se, hvilke beboere der er på opgaven. Derudover er der knapper til at tilføje og fjerne beboere på opgaven, samt for at sætte sælgeren på opgaven. Slutningsvis er der i bunden to knapper til enten at bekræfte oprettelsen / ændringen af bytteopgaven, eller at annullere handlingen.
Oprettelsen af andre opgaver ligner rigtig meget denne GUI. Den eneste forskel er, hvilke InputFields der er, og at man ikke kan sætte en sælger.



Figur 21: GUI: Beboer

På figur 21 vises en FXML-fil åbnet i SceneBuilder, for at tilføje redigere eller oprette en beboer. GUI'en består af InputFields, for hver instansvariabel til en beboer. I bunden er der to knapper til enten at bekræfte oprettelsen / ændringen af bytteopgaven, eller at annullere handlingen.



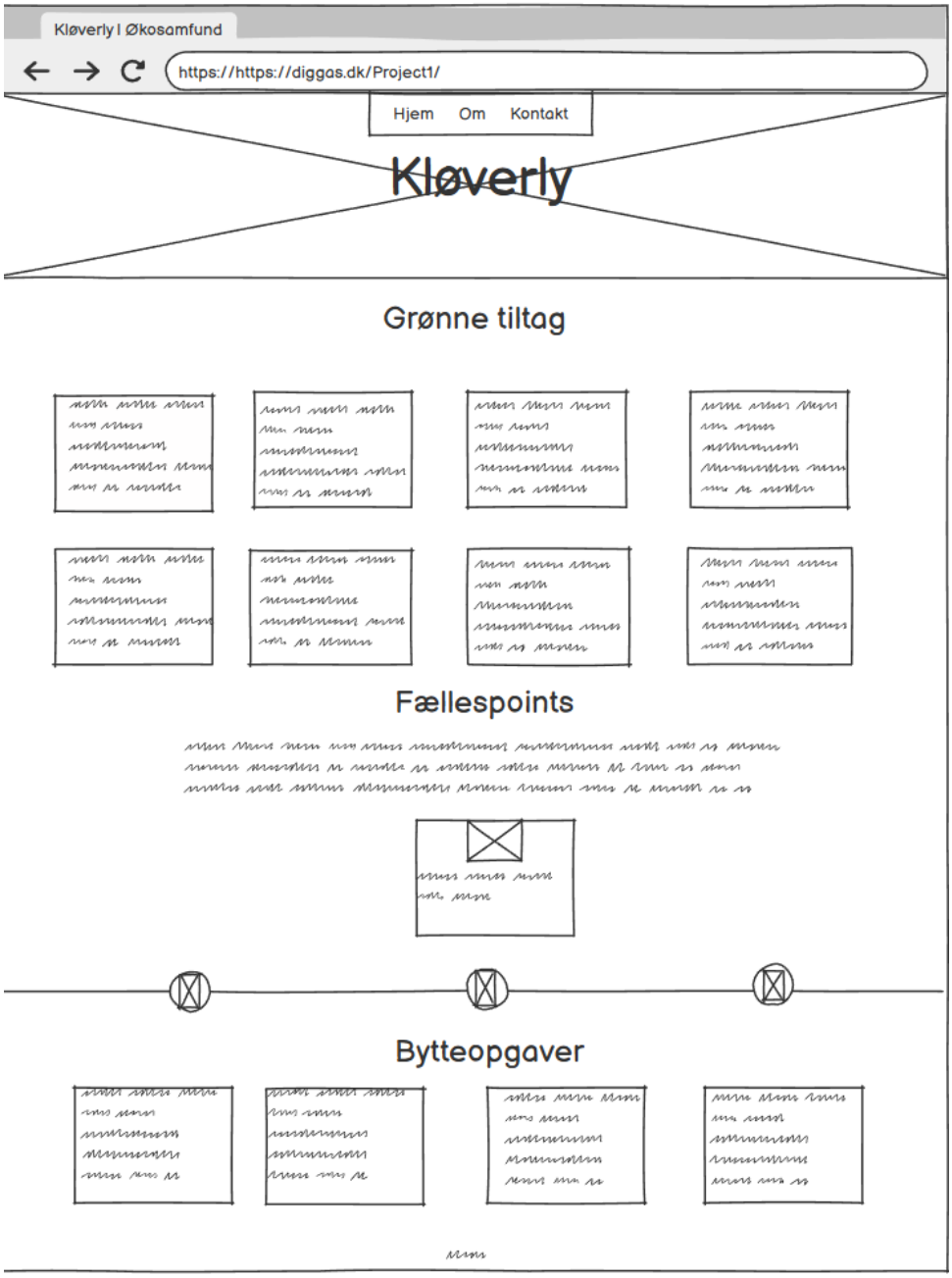
Figur 22: GUI: Tilføj beboer

På figur 22 vises en FXML-fil åbnet i SceneBuilder, for at tilføje beboere til en opgave. Dette bruges til at tilføje beboere på en opgave, men også for at tilføje en sælger til en bytteopgave. Dette består af et TableView, med samme information som beboer-listens TableView. Derfor alle de relevante oplysninger på beboere, så man kan skelne imellem dem. Dertil har GUI'en også et InputField i bunden, der bruges som søgefelt og en knap til at tilføje den valgte beboer eller annullere handlingen.

Nu er de forskellige visninger til GUI'en blevet vist og designet er forklaret. Hvordan de bruges er forklaret i brugermanualen, som kan findes i Bilag[12].

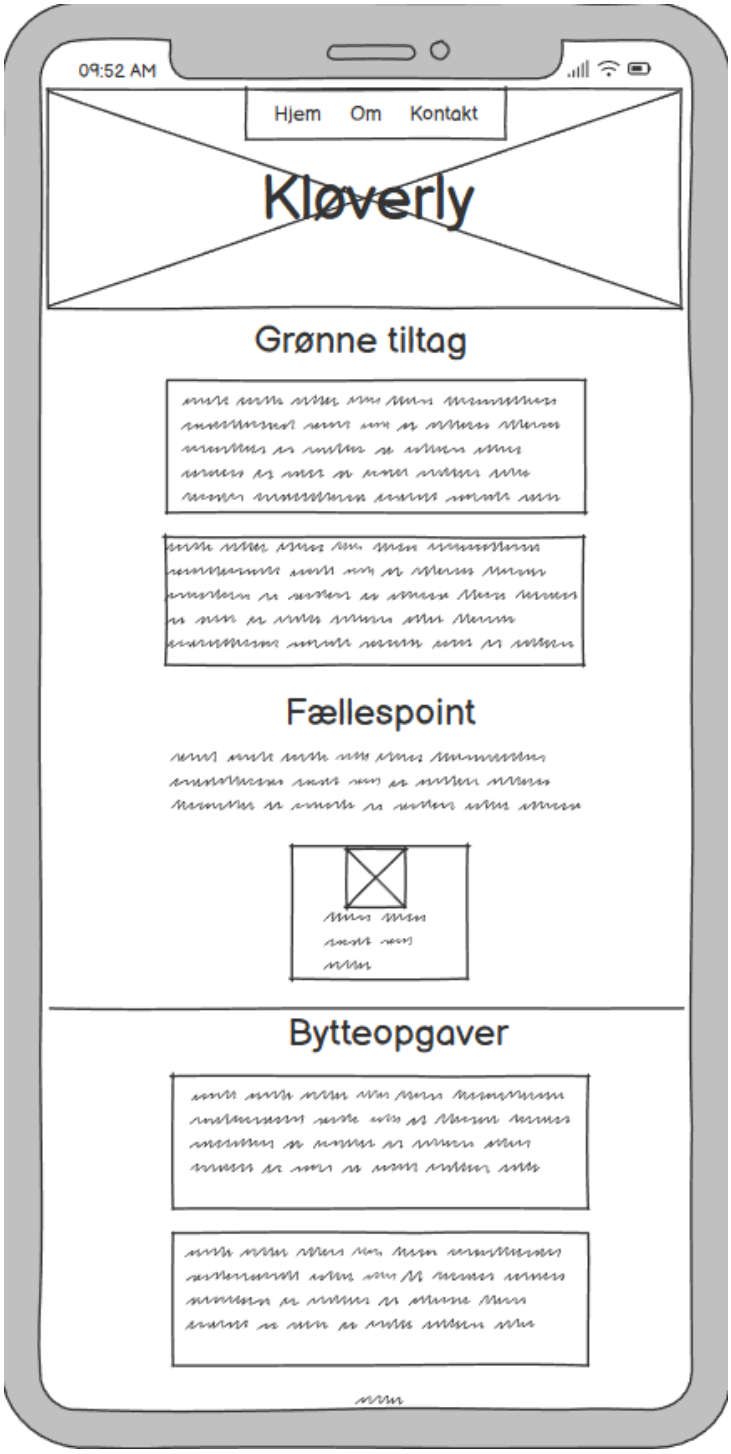
3.5 Design WEB

Der laves wireframe til hjemmesiden, for at der kan indskærpes en klar vision til designet og layoutet af hjemmesiden. Til denne opgave bruges programmet Balsamiq. Der vælges at lave 3 forskellige sider. En hjem-side, en om-side og en kontakt-side. Derudover vælges der, at lave wireframes for desktop-view og telefon, fordi det følger de ikke-funktionelle krav stillet i afsnit 2.2.



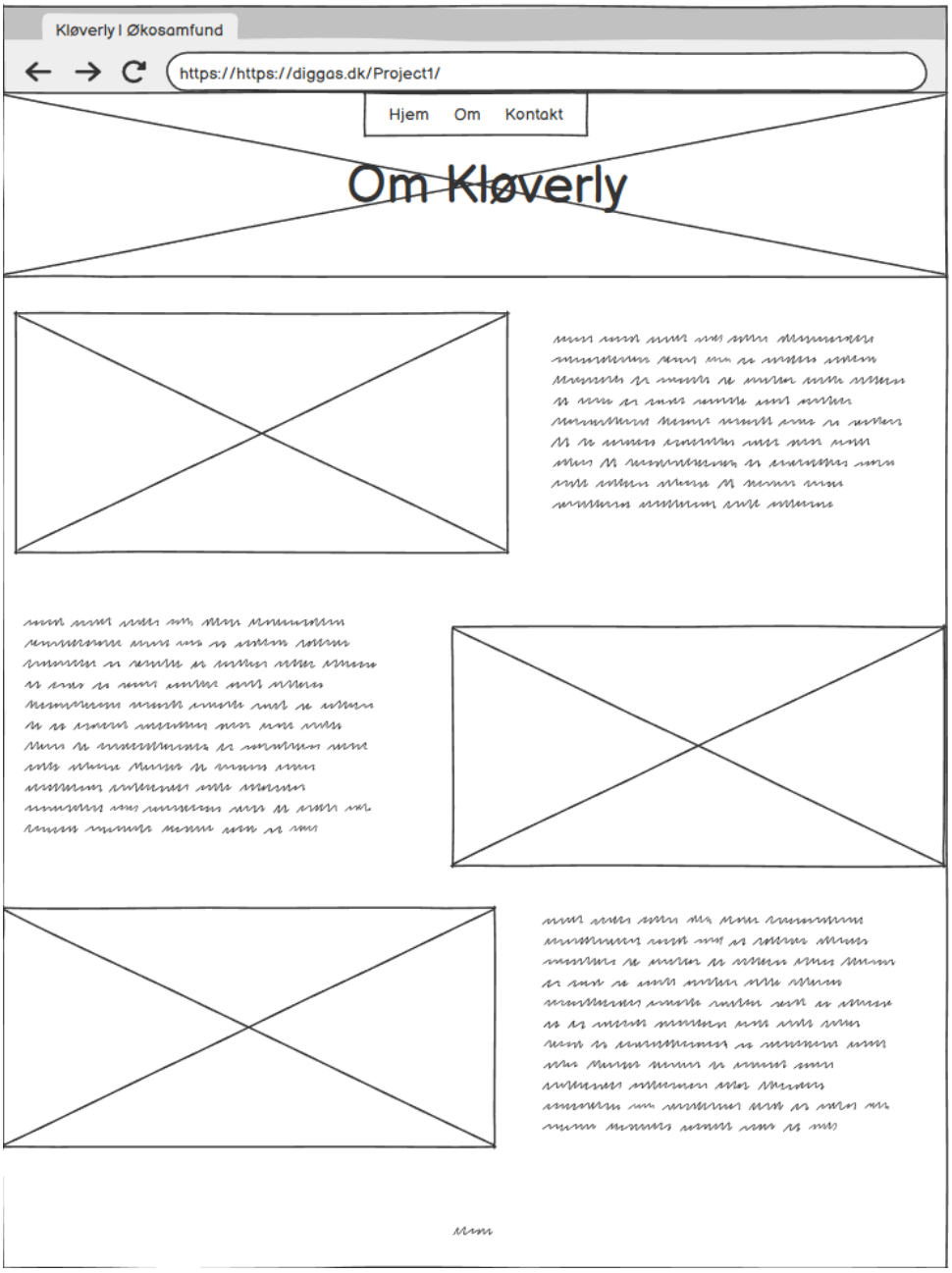
Figur 23: Wireframe: Laptop hjem-side

På figur 23 vises wireframe for laptop-view af home. Der er valgt et design med en navigationsbar øverst, for intuitivt at kunne navigere mellem de 3 forskellige sider. Dernæst vælges at have en baggrund bag det, der går lidt ned af siden med Kløverly stående i midten af. Dette er blot, for at vise Kløverlys økosamfund. Her kan man også se grønne tiltag, nemt adskilt af kasser til hvert tiltag. Efterfulgt af fællespoint og en beskrivelse af, hvad det er.
Dernæst kan man se en let oversigt over, hvilke mål man har for antallet af fællespoint på en "progress bar"med mål på.
Til sidst er der bytteopgaver opdelt på samme vis som grønne tiltag.
Dette design sætter brugervenlighed og tilgængelighed højt, fordi alt er tilgængeligt fra den første og primære side.



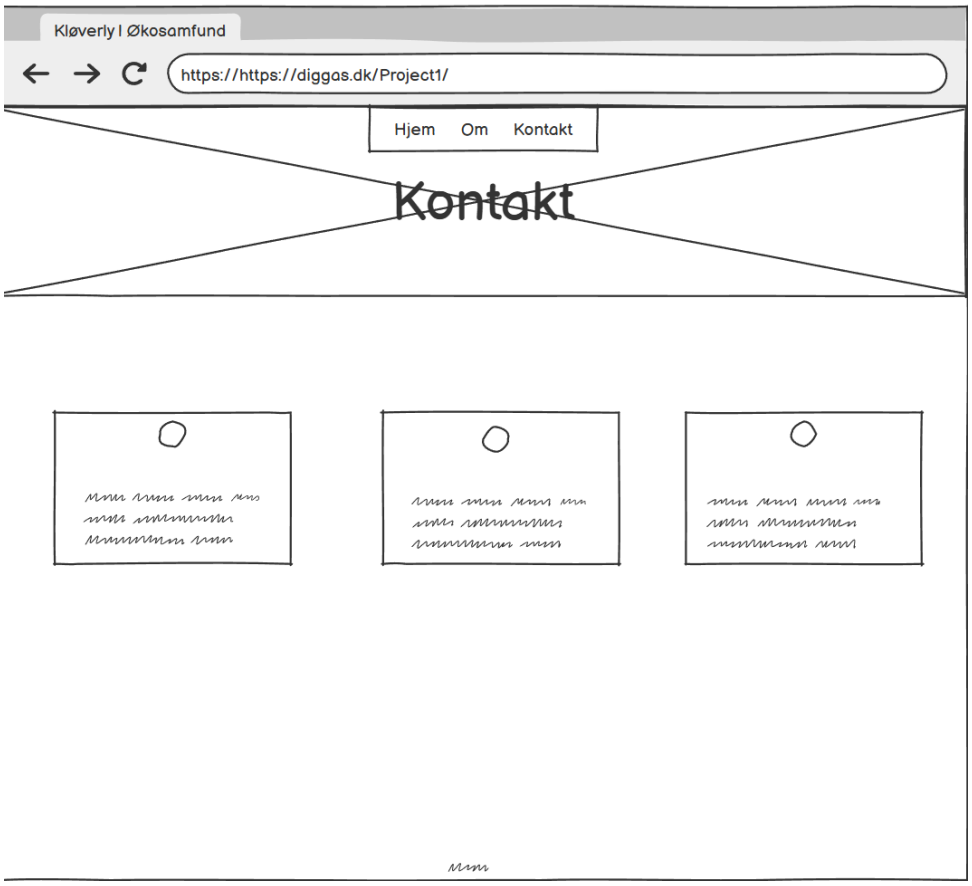
Figur 24: Wireframe: Telefon hjem-side

På figur 24 vises wireframe for telefon-view af home. Dette følger stort set designet fra laptop. Dog er der ændringer som, at opgaverne ikke bliver vist med 4 på hver række, men i stedet en, så det er læsbart. Hertil er der heller ikke mål på "progress bar", da det ikke er læsbart på en mobil med lav skærmbredde.



Figur 25: Wireframe: Laptop om-side

På figur 25 vises wireframe for laptop-view af om-siden. Den skal give indsigt i Kløverlys hverdagsliv og hvilke værdier de har. Derfor er der udvalgt billeder af deres aktiviteter, samt noget beskrivende tekst. Selvfølgelig følger det også navigationen og nogenlunde det samme layout som hjem-siden. Telefon-view har det hele på én kolonne, i stedet for 2 ved siden af hianandet, for at følge aspect-ratio på telefon.



Figur 26: Wireframe: Laptop kontakt-side

På figur 25 vises wireframe for laptop-view af kontakt-siden. Her er genbrugt navigationen og generelt toppen af layoutet fra hjem-siden. Derudover er der lavet 3 bokse, en til hver person bag hjemmesiden. Her er der plads til billede og kontaktoplysninger som f.eks. mail.
På telefon-view er det stort set det samme, udover at de 3 kolonner bliver til en, så der følger bredden af skærmen på en telefon og det er læsbart.

4 Implementation

4.1 Implementation Indledning

Baseret på design laves implementation. Der følges som udgangspunkt klassediagrammet fra afsnit 3.2 til at skrive koden i Java. Hertil bruges sekvensdiagrammer fra afsnit 3.3 til, hvad der skal ske i de forskellige metoder i klassediagrammet. Derudover bruges FXML-filerne fra afsnit 3.4 til de forskellige visninger i GUI'en.

4.2 Implementation

Der tages udgangspunkt i use-case-beskrivelsen fra afsnit 2.3 om administrer bytteopgaver. Derfor startes der med at implementerer en beboer, så der er nogen, som kan udføre opgaven.

```
1 public class Resident
2 {
3   private String name;
4   private int houseNumber;
5   private char gender;
6   private int points;
7   private MyDate birthdate;
```

```
8
9   public Resident(String name, MyDate birthdate, int houseNumber, char gender)
10  {
11    if(birthdate == null)
12    {
13      throw new IllegalArgumentException("Birthdate␣can't␣be␣null");
14    }
15    this.name = name;
16    this.birthdate = birthdate.copy();
17    this.houseNumber = houseNumber;
18    this.gender = gender;
19    this.points = 0;
20  }
21
22  public String getName()
23  {
24    return name;
25  }
26
27  public void setName(String name)
28  {
29    this.name = name;
30  }
31
32  @Override
33  public boolean equals(Object obj)
34  {
35    if (obj == null || getClass() != obj.getClass())
36      return false;
37    Resident other = (Resident) obj;
38
39    return name.equals(other.name) && houseNumber == other.houseNumber
40        && gender == other.gender && points == other.points && birthdate.equals(
41        other.birthdate);
42  }
43
44  @Override
45  public String toString()
46  {
47    return "name␣=␣" + name + "␣birthdate␣=␣" + birthdate + "␣houseNumber␣=␣"
48        + houseNumber + "␣gender␣=␣" + gender + "␣points␣=␣" + points;
49  }
50 }
```

Listing 1: Resident-klassen

På listing 1 ses et kodestykke af Resident-klassen. Her er der udeladt alle setters og getters, udover for instansvariablen name. Dog er der setters og getters for alle instansvariabler. Til at starte med deklareres alle variabler, der skal bruges i klassen, hvilket er alle der skal bruges til en beboer. En speciel variabel er her birthdate, da det er en MyDate type, derfor skabes en association til MyDate klassen.
Derefter skrives konstruktøren, der tager alle variabler, udover points og initialiserer alle instansvariabler. Hertil tjekker konstruktøren også om birthdate er null og kaster en IllegalArgumentException, hvis den er.
Efterfølgende er getters og setters skrevet. Simpelt returnerer getters den type som variablen er deklareret til og selvfølgelig med den værdi som variablen er. Setters sætter variablen til argumentet.
Til sidst er @Override metoderne skrevet. Først equals som sammenligner objektet, den bliver kaldt på med objektet i argumentet. Først tjekker metoden om det modtagede objekt er null eller

om det ikke er har samme klasse, hvis det er sandt returnerer den falsk. Bagefter returnerer den et udtryk af om alle variabler har samme værdier i objektet fra argumentet og i objektet, som den er blevet kaldt på.
Slutningsvis er toString metoden skrevet. Den returnerer en lang streng af variablenavnene og deres værdier.

Begge disse @Override metoder er skrevet i langt de fleste klasser. Både dem, getters og setters vil ikke blive vist på resten af kodestykkerne, da de er lavet på samme vis.
Efter beboeren er indført, skal den kunne have en opgave. Derfor kigges nu på opgave-klassen.

```
1  public abstract class Task
2  {
3    private String taskName;
4    private String taskDescription;
5    private int points;
6    private ArrayList<Resident> taskResidents;
7
8    public Task(String taskName, String taskDescription, int points)
9    {
10     // Udeladt alle initialiseringer for de andre variabler
11     this.taskResidents = new ArrayList<>();
12   }
13
14   public void addResidentToTask(Resident resident)
15   {
16     if(resident == null)
17     {
18       throw new IllegalArgumentException("Resident can't be null");
19     }
20     taskResidents.add(resident);
21   }
22
23   public void removeResidentFromTask(Resident resident)
24   {
25     if(resident == null)
26     {
27       throw new IllegalStateException("Resident can't be returned because it's null");
28     }
29     taskResidents.remove(resident);
30   }
31
32   public Resident getTaskResident(int index)
33   {
34     return taskResidents.get(index);
35   }
36
37   public int getNumberOfTaskResidents()
38   {
39     return taskResidents.size();
40   }
41
42   public abstract void completeTask();
43
44   public void clearTaskResidents()
45   {
46     taskResidents.clear();
47   }
48 }
```

Listing 2: Task-klassen

På listing 2 ses et kodestykke af Task-klassen. Der er udeladt simple getters, setters og konstruktøren, da det minder meget om det første kodestykke. Det første der er værd at bide mærke i er, at det er en abstrakt klasse, for at den ikke kan blive oprettet. Det næste er ArrayListen med typen Resident som variablen taskResidents. Dette skaber en association til Resident-klassen fra tidligere, med 0 til mange. Derfor bliver den også i konstruktøren initialiseret.
Efter dette er der vist alt håndtering af ArrayListen. Først addResidentToTask-metoden, som tager et Resident-objekt ind som argument og tilføjer det til listen, hvis objektet er null bliver der kastet en IllegalArgumentException i stedet. Derefter er metoden removeResidentFromTask deklareret, der også tager et resident objekt ind og fjerner det fra listen, hvis det ikke er null.
Dernæst deklareres metoden getTaskResident, som returnerer et Resident-objekt ved hjælp af det index, der er blevet givet som argument.
Herefter bliver getNumberOfTaskResidents deklareret. Den returnerer, hvor mange beboere, der er på opgaven ved hjælp af ArrayList-metoden size.
Næste metode, der bliver deklareret er den abstrakte metode completeTask, der bliver deklareret som abstrakt, fordi den først bliver implementeret af de klasser, der nedarver fra Task-klassen.
Den sidste metode der bliver deklareret er clearTaskResidents, som bruger ArrayList-metoden clear til, at fjerne alle beboere fra taskResidents-listen.
Lignende kode med håndtering af ArrayLister udelades fra resten af kodestykkerne.
Nu er den del af alle opgaver, som er ens implementeret, nu kigges der på bytteopgave-klassen, fordi der, som tidligere udtrykt, tages udgangspunkt i use-case-beskrivelsen for administrer bytteopgaver.

```
1  public class ExchangeTask extends Task
2  {
3    private MyDate endDate;
4    private Resident seller;
5
6    public ExchangeTask(String taskName, String taskDescription, int points,
7        MyDate endDate, Resident seller)
8    {
9      super(taskName, taskDescription, points);
10     // resten af konstruktren er udeladt
11   }
12
13   @Override
14   public void completeTask()
15   {
16     seller.addPoints(getNumberOfTaskResidents() * getPoints());
17
18     for (int i = 0; i < getNumberOfTaskResidents(); i++)
19     {
20       getTaskResident(i).addPoints(-getPoints());
21     }
22   }
23 }
```

Listing 3: ExchangeTask-klassen

På listing 3 ses et kodestykke af ExchangeTask-klassen. Der er udeladt getters, setters og det meste af konstruktøren. Det første væsentlige i denne klasse er, at den nedarver fra Task-klassen. Det vil sige, at den bruger Task-klassen som skabelon og bygger videre på den. Derfor skal den også kalde superklassens konstruktør i sin egen konstruktør, samt implementere alle abstrakte klasser i Task-klassen. Det er det, der er anderledes i denne klasse. På linje 9 ses, hvordan en superklasses konstruktør kaldes. I metoden completeTask, ses hvordan den abstrakte metode, deklareret i Task-klassen bliver implementeret ved, at give sælgeren point for antallet af beboere på opgaven (beboere, der køber ydelsen), multipliceret med prisen af ydelsen. Derefter loopes der igennem alle beboere på opgaven og der trækkes prisen af ydelsen fra hver beboers point-variabel, ved hjælp af metoden addPoints med den negative værdi af prisen.

Nu er selve bytteopgave-klassen indført, så der kan oprettes bytteopgaver. Dog kan en bru-

ger af systemet ikke ændre på noget, uden at noget view at kunne interagere med. Derfor kigges der nu på den væsentligste del af det, som er ViewHandleren.

```
public class ViewHandler
{
  private Scene currentScene;
  private Stage primaryStage;
  private VillageModel model;
  private EditExchangeTaskController editExchangeTaskController;
  private AddResidentController addResidentController;
  private VillageController villageController;
  private EditEcoTaskController editEcoTaskController;
  private EditCommunityTaskController editCommunityTaskController;
  private EditResidentController editResidentController;
  private ViewState viewState;

  public void start(Stage primaryStage)
  {
    this.primaryStage = primaryStage;
    openView("Village");
  }

  public void openView(String id)
  {
    Region root = null;
    switch (id)
    {
      case "Village":
        root = loadVillage(id +".fxml");
        break;
      case "EditExchangeTask":
        root = loadEditExchangeTask(id + ".fxml");
        break;
      case "EditEcoTask":
        root = loadEditEcoTask(id+".fxml");
        break;
      case "EditCommunityTask":
        root = loadEditCommunityTask(id+".fxml");
        break;
      case "EditResident":
        root = loadEditResident(id+".fxml");
        break;
      case "AddResident":
        root = loadAddResident(id+".fxml");
        break;
    }

    currentScene.setRoot(root);
    String title = "";
    if (root.getUserData() != null)
    {
      title += root.getUserData();
    }
    primaryStage.setTitle(title);
    primaryStage.setScene(currentScene);
    primaryStage.setWidth(root.getPrefWidth());
    primaryStage.setHeight(root.getPrefHeight());
    primaryStage.show();
  }
```

```
57
58    public void closeView()
59    {
60      primaryStage.close();
61    }
62
63    public Region loadVillage(String fxmlFile)
64    {
65      if (villageController == null)
66      {
67        try
68        {
69          FXMLLoader loader = new FXMLLoader();
70          loader.setLocation(getClass().getResource(fxmlFile));
71          Region root = loader.load();
72
73          villageController = loader.getController();
74          villageController.init(this, model, root, viewState);
75        }
76        catch (Exception e)
77        {
78          e.printStackTrace();
79        }
80      }
81      else
82      {
83        villageController.reset();
84      }
85      return villageController.getRoot();
86    }
87  }
```

Listing 4: ViewHandler-klassen

På listing 4 ses et kodestykke af ViewHandler-klassen. Der er udeladt metoder, der gør det samme, det indebærer alle load-metoder, udover for loadVillage. Først og fremmest er der start-metoden, der starter programmet og åbner det første view. Den tager et Stage-objekt ind som argument, som den sætter som primaryStage. Derefter kalder den metoden openView.
openView-metoden tager et String-objekt ind som argument. Derefter bruges en switch case til at gøre noget alt efter, hvad argumentstrengen er. Alt efter, hvilket view der skal åbnes er argumentet forskelligt, derfor passer det med at den kalder den metode, der passer til det view, der skal åbnes. Den kalder det views load-metode som skal åbnes og sætter det på variablen "root", som er deklareret til null udenfor switch-casen. Derefter bryder den ud af switch-casen og kalder scenens setRoot-metode, for at sætte roden til den region som load-metoden returnerer. Den region som bliver returneret af load-metoden indeholder alt, hvad der hører til visningen af det, der ønskes at vises. Det vil sige alt, hvad FXML filen indeholder. Derefter sættes den primære scenes attributter til de attributter, der er hentet, for at visningen opdateres til den nye.

Herefter deklareres closeView-metoden, der kalder close metoden på den primære scene, hvilket lukker visningen og dermed programmet.
Til sidst er loadVillage-metoden deklareret, hvilket er en af de load-metoder, der blev nævnt og brugt i openView-metoden. Dens job er at returnere et Region-objekt med alt den information, der skal bruges for at kunne åbne visningen af det ønskede view. Det bliver gjort ved at den tager et argument som er filnavnet på den FXML-fil, der er lavet til viewet, herefter tjekkes om controlleren er null, som den blev, da den blev initialiseret i konstruktøren. Hvis den er null, så loader den FXML-filen med navnet fra argumentet. Det gør den med et FXMLLoader-objekt og et Region-objekt. Bagefter sættes villageControlleren til FXML-filens controller og bliver initialiseret gennem dens init-metode. Dette bliver gjort, for at den kan blive interaktiv, som styres i controller-klassen.

Hvis villageController ikke er null, når loadVillage bliver kaldt, så kalder den reset-metoden i villageController.
Til sidst returnerer loadVillage-metoden roden, som indeholder alt information, sådan at viewHandleren kan åbne viewet med alle de attributter det skal have på scenen.

Nu haves en klasse, der kan påvirke, hvad der vises på skærmen af brugeren. Derfor skal vi blot have indført en controller til FXML-filen lavet i afsnit 3.4, så den kan blive interaktiv.

```
public class EditExchangeTaskController
{
  //Instansvariabler udeladt
  public void init(ViewHandler viewHandler, VillageModel model, Region root,
      ViewState viewState)
  {
    //initialisering af instansvariabler er udeladt
    residentNameColumn.setCellValueFactory(
        cellData -> cellData.getValue().getNameProperty());
    residentBirthdateColumn.setCellValueFactory(
        cellData -> cellData.getValue().getBirthdate());
    residentTable.getSelectionModel().selectedItemProperty()
        .addListener((obs, oldSelection, newSelection) -> {
          if (newSelection != null)
          {
            Resident currentResident = new Resident(
                newSelection.getNameProperty().get(),
                MyDate.getDateFromString(newSelection.getBirthdate().get()),
                newSelection.getHouseNumberProperty().get(),
                newSelection.getGender().get().charAt(0));
            currentResident.setPoints(newSelection.getPointsProperty().get());
            viewState.setResident(model.getResident(currentResident));

            removeTaskResidentButton.setDisable(false);
          }
          else
          {
            removeTaskResidentButton.setDisable(true);
          }
        });

    reset();
  }

  public void reset()
  {
    setTaskResidents(); //udeladt metode
    residentTable.setItems(viewModel.getList());
    if (viewState.getExchangeTask() == null && viewState.getSeller() == null)
    {
      viewState.clearTaskResidents();

      titleText.setText("Opret␣bytteopgave");
      titleField.setText("");
      descriptionField.setText("");
      endDateField.setText("");
      pointsField.setText("");
      sellerField.setText("");
    }
    else if (viewState.getExchangeTask() != null && !viewState.getTaskType().equals("
        addingTaskResidents"))
```

```
51      {
52        titleText.setText("Rediger␣bytteopgave");
53        titleField.setText(viewState.getExchangeTask().getTaskName());
54        descriptionField.setText(
55            viewState.getExchangeTask().getTaskDescription());
56        endDateField.setText(viewState.getExchangeTask().getEndDate().toString());
57        pointsField.setText(
58            Integer.toString(viewState.getExchangeTask().getPoints()));
59        sellerField.setText(viewState.getExchangeTask().getSeller().getName());
60      }
61      else
62      {
63        viewState.setTaskType("exchangeTask");
64      }
65      errorLabel.setText("");
66
67      sellerField.setDisable(true);
68      if (viewState.getSeller() != null)
69      {
70        sellerField.setText(viewState.getSeller().getName());
71      }
72      removeTaskResidentButton.setDisable(true);
73      if (viewState.getExchangeTask() == null)
74      {
75        addTaskResidentButton.setDisable(true);
76      }
77      else
78      {
79        addTaskResidentButton.setDisable(false);
80      }
81    }
82
83    @FXML private void submitButtonPressed()
84    {
85      if (!titleField.getText().isEmpty() && !descriptionField.getText().isEmpty()
86          && !pointsField.getText().isEmpty() && !sellerField.getText().isEmpty()
87          && !endDateField.getText().isEmpty())
88      {
89        try
90        {
91          if (viewState.getExchangeTask() != null)
92          {
93            Resident seller = viewState.getExchangeTask().getSeller();
94            if (viewState.getSeller() != null && !viewState.getExchangeTask()
95                .getSeller().getName().equals(viewState.getSeller().getName()))
96            {
97              seller = viewState.getSeller();
98            }
99            ExchangeTask newTask = new ExchangeTask(titleField.getText(),
100               descriptionField.getText(), Integer.parseInt(pointsField.getText()),
101               MyDate.getDateFromString(endDateField.getText()), seller);
102
103           for (int i = 0; i < viewState.getNumberOfTaskResidents(); i++)
104           {
105             newTask.addResidentToTask(viewState.getTaskResident(i));
106           }
107
108           model.removeExchangeTask(viewState.getExchangeTask());
```

```
109
110           model.addExchangeTask(newTask);
111         }
112         else
113         {
114           ExchangeTask exchangeTask = new ExchangeTask(titleField.getText(),
115               descriptionField.getText(), Integer.parseInt(pointsField.getText()),
116               MyDate.getDateFromString(endDateField.getText()),
117               viewState.getSeller());
118
119           model.addExchangeTask(exchangeTask);
120         }
121
122         viewState.setSeller(null);
123         viewState.setExchangeTask(null);
124
125         cancelButtonPressed();
126       }
127       catch (Exception e)
128       {
129         errorLabel.setText("Formatteringen er ikke korrekt");
130       }
131     }
132     else
133     {
134       errorLabel.setText("Du skal udfylde alle informationer");
135     }
136   }
137 }
```

Listing 5: ExchangeTaskController-klassen

På listing 5 ses et kodestykke af ExchangeTaskController-klassen. Her er der bl.a. vist de metoder, der blev kaldt af ViewHandleren fra forrige kodestykke. Først skal der lægges mærke til, at init metoden initialiserer alle instansvariabler i controlleren. Det vil sige, at det ikke sker i konstruktøren af controlleren, men først, når ViewHandleren har noget at initialisere. Efter initialisering af alle instansvariabler, behandler den residentTable, som er alle de beboere, der er købere på bytteopgaven. Først sætter den, hvad der skal være på hver kolonne af tabellen. Derefter oprettes der en listener på selektion af en række i tabellen. Dette opretter derefter et beboer objekt, der ligner det på tabellen og sætter viewStatens beboer til den beboer i modellen, der har samme værdier som i tabellen. Dette gør det muligt for de andre controllers, at bruge denne information.
Reset-metoden kaldes af ViewHandler-klassen, hver gang der skiftes view til EditExchangeTask, efter den er blevet initialiseret. Formålet her er at fjerne alt data fra alle InputFields osv.
Først sætter den alle købere på opgaven ved, at kalde setTaskResidents og sætte tabellen. Derefter tjekkes der om hvorvidt, der findes en exchangeTask og getSeller, hvis der gør er der nemlig blevet markeret en bytteopgave ude i hovedmenuen, og der skal ikke oprettes en ny, men redigeres. Dette betyder derfor at denne controller, både kan redigere og oprette, hvilket giver mindre controllers og FXML-filer i alt. Hvis der ikke er markeret en bytteopgave, skal der oprettes en ny. Derfor sættes alle InputFields til at være tomme (linje 44-48). Hvis der derimod er valgt en bytteopgave, sættes InputFields til de værdier som bytteopgaven har (linje 53-59). Her tjekkes også om der tilføjes beboere på opgaven, for ikke at slette alle de andre ændringer, hvis man derefter tilføjer en beboer.

Til sidst er der en @FXML-metode kaldet submitButtonPressed, som bliver kaldt, hver gang der trykkes på Bekræft-knappen i GUI'en. Først tjekker den om alle InputFields har værdier, hvis ikke viser den en fejl til brugeren. Derefter laves en try-catch block, der skal forsøge at oprette eller erstatte en bytteopgave. Der tjekkes nemlig først om der er valgt en bytteopgave, eller ikke. Dermed vides der hvorvidt brugeren forsøger at oprette eller redigere. Hvis der er en bytteopgave valgt, så redigeres der. Af den grund opretter den en ny bytteopgave og sætter alle de nye værdier

ind i den, bagefter slettes den gamle bytteopgave, der blev valgt ude på listen, fra modellen og der tilføjes den nye opgave. Hvis der skulle oprettes en bytteopgave i stedet, oprettes den blot og tilføjes til modellen. Derefter sættes sælger og bytteopgave til null i viewState, for ikke at komme til at redigere den samme bytteopgave igen, uden at have valgt den. Med alle disse klasser fra afsnittet, kan der nu administreres og tilføjes bytteopgaver.

På hjemmesiden læses en JSON-fil med data fra Java-programmet. Java-programmet gemmer hele VillageModel-objektet i JSON. På hjemmesiden læses derfor fra en JSON-fil, der ligner.

```
const exchangeTasksElement = document.querySelector(".exchangeTasks")
const pointCount = document.querySelector("#pointCount")
fetch("data.json")
    .then(response => {
        if (!response.ok) {
            throw new Error("not␣ok")
        }
        return response.json()
    })
    .then(result => {
        pointCount.innerText = result.communityPoints

        for (let i = 0; i < result.exchangeTaskList.exchangeTasks.length; i++) {
          const exchangeTaskDiv = document.createElement("div")
          exchangeTaskDiv.classList.add("exchangeTask")
          const titleTag = document.createElement("h2")
          titleTag.innerText = result.exchangeTaskList.exchangeTasks[i].taskName
          const descriptionTag = document.createElement("p")
          descriptionTag.innerText = result.exchangeTaskList.exchangeTasks[i].
              taskDescription
          const pointsTag = document.createElement("p")
          pointsTag.innerText = "Pris:␣" + result.exchangeTaskList.exchangeTasks[i].
              points

          const sellerTag = document.createElement("p")
          sellerTag.innerText = "Saelger:␣" + result.exchangeTaskList.exchangeTasks[i].
              seller.name
          const endDateTag = document.createElement("p")
          endDateTag.innerText = "Slutdato:␣" + result.exchangeTaskList.exchangeTasks[i
              ].endDate.day + "/" + result.exchangeTaskList.exchangeTasks[i].endDate.
              month + "/" + result.exchangeTaskList.exchangeTasks[i].endDate.year

          exchangeTaskDiv.append(titleTag)
          exchangeTaskDiv.append(descriptionTag)
          exchangeTaskDiv.append(pointsTag)
          exchangeTaskDiv.append(sellerTag)
          exchangeTaskDiv.append(endDateTag)

          exchangeTasksElement.append(exchangeTaskDiv)
        }
    })
    .catch(error => console.log(error))
```

Listing 6: Hjemmeside: Fetch

På listing 6 ses et kodestykke fra hjemmesiden som viser, hvordan der hentes data fra JSON-filen og oprettes elementer i HTML, for at vise dataen på hjemmesiden. Der bruges fetch-metoden med argumentet af JSON-filens navn. Derefter ventes på, at dataen er hentet fra filen, og så gøres noget. Derfor bruges .then. Først tjekkes om responsen er okay, hvis ikke kastes en fejl, som bliver håndteret ved .catch, blot ved at printe det i console. Derudover bliver der også kaldt funktionen .json() på response. Dette omdanner JSON-filen til et JavaScript-objekt, som nu kan håndte-

res i det næste .then. Her loopes over længden af resultatets bytteopgave liste. Hvorefter der oprettes et div-element og de forskellige tekstelementer, der skal bruges til opgaven som titel, beskrivelse, point, sælger og slutdato. Alle tekstelementer appendes til bytteopgave-elementet (div-elementet), hvorefter bytteopgave appendes til et element i HTML'en, der skal vise bytteopgaverne. På den måde bliver data fra JSON hentet i JavaScript og vist på hjemmesiden.

5 Test

5.1 Test introduktion

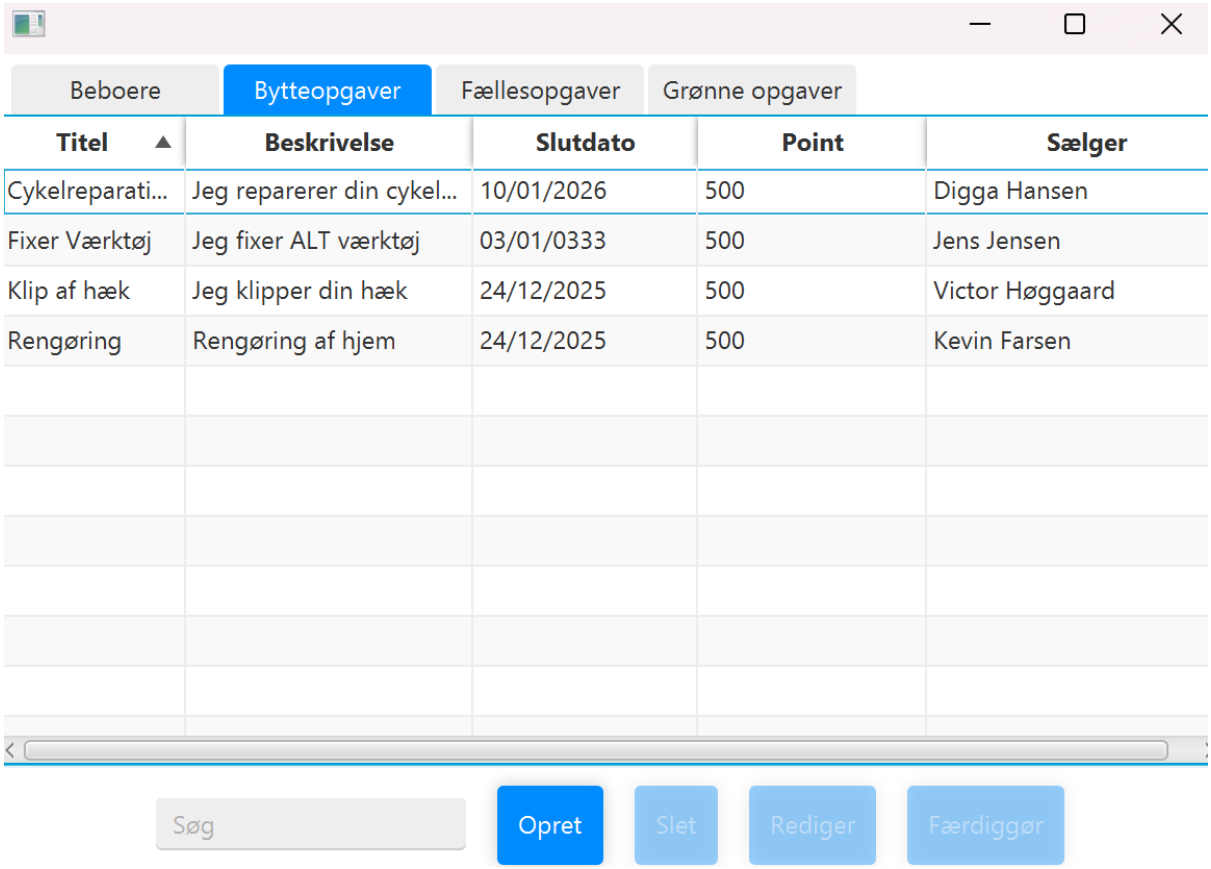
Test gennemgår udvalgte scenarier i use-cases fra afsnit 2.3. Scenarierne bliver gennemgået i programmet lavet fra implementation afsnit 4. Her kigges på om disse use-cases har samme resultat som implementationen. Dette opskrives i en tabel, som kan ses i næste delafsnit.

5.2 Test via usecases

Use Case / Trin	Forventet resultat	Faktisk resultat
Basis-sekvens: Vis Aktuelle Bytteopgaver	Systemet viser en alfabetisk liste over aktuelle bytteopgaver med navn,beskrivelse,sælger, slutdato og point.	Listen over bytteopgaver vises korrekt, sorteret alfabetisk med alle nødvendige detaljer.
Scenario A: Opret Ny Bytteopgave	Systemet åbner vindue, validerer alle påkrævede felter (navn, beskrivelse, dato, point, sælger) og tilføjer opgaven til listen.	Alle oplysninger blev indtastet, valideringen bestod, og den nye bytteopgave blev oprettet.
Scenario B: Rediger Opgave (Opdatér)	Systemet viser opgavens informationer, tillader redigering (dato, navn, beskrivelse, point,sælger), validerer ændringerne og gemmer dem.	Opgavedetaljer blev redigeret og gemt uden valideringsfejl.
Scenario BB: Tilføj/Fjern Køber	Systemet viser beboere i Kløverly, tillader valg og tilknytning/fjernelse af købere til bytteopgaven.	Beboere kunne tilknyttes som købere (og fjernes) uden problemer, og systemet opdaterede tilknytningen korrekt.
Scenario C: Slet Opgave	Systemet beder om bekræftelse, sletter opgaven permanent fra aktuelle bytteopgaver og opdaterer listen.	Bekræftelse blev valgt, opgaven blev slettet, og listen blev opdateret.
Scenario D: Søg efter Bytteopgave	Systemet opdaterer opgavelisten i realtid baseret på søgeargumentet (opgavenavn, beboernavn, eller dato).	Søgning efter opgavetitel og beboernavn filtrerede listen korrekt og straks.
Scenario E: Færdiggør Opgave	Systemet fjerner opgaven fra oversigten og tildeler/fjerner individuelle point til de involverede beboere.	Opgaven blev færdiggjort, fjernet fra listen, og de individuelle point blev justeret korrekt for sælger og købere.

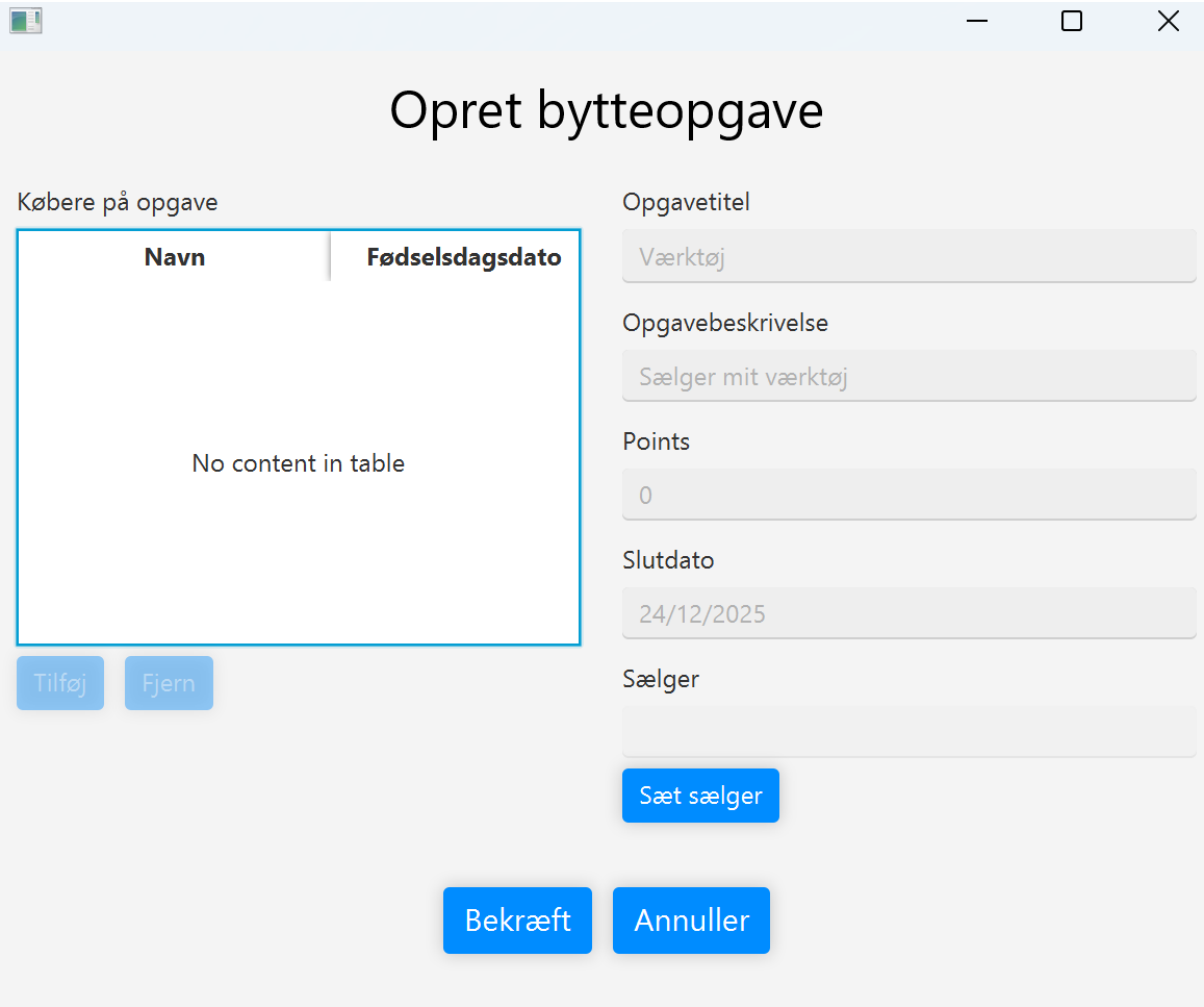
De nedenstående figure 27-33 viser dokumentation, for at det faktiske resultat vises i vores

program fra implementation. Det viser blot at det faktisk påvirker data, på den måde som beskrevet.



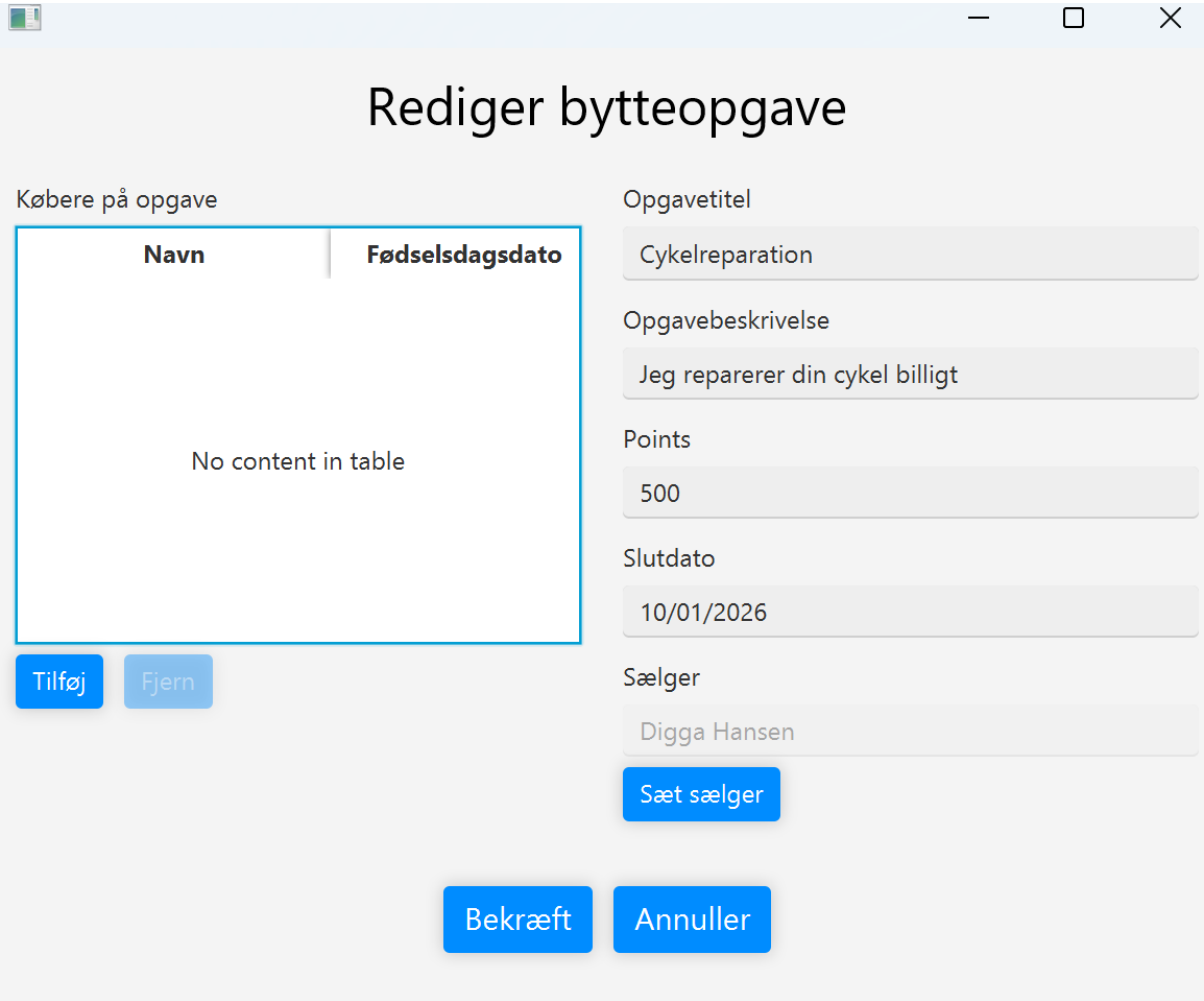
Figur 27: Liste over bytteopgaver

På figur 27 bekræftes, at systemet korrekt viser den alfabetisk sorterede liste over bytteopgaver med alle nødvendige detaljer som titel, beskrivelse, dato, point og sælger. Dette opfylder det forventede resultat for basis-sekvensen: Vis Aktuelle Bytteopgaver.



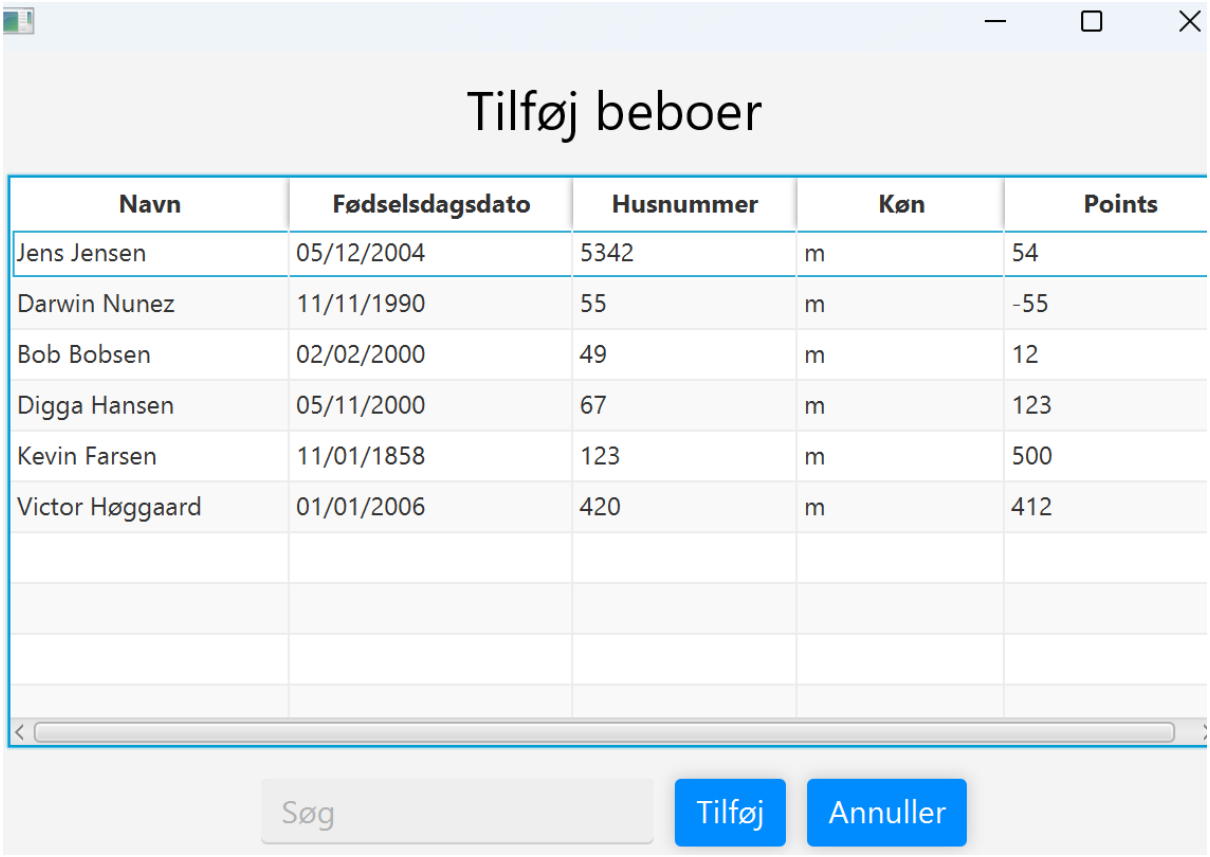
Figur 28: Opret bytteopgaver

På figur 28 vises oprettelsesvinduet, der muliggør indtastning af opgaveoplysninger og valg af sælger. Dette dokumenterer, at Scenario A: Opret Ny Bytteopgave kan udføres, og at validering af felter kan finde sted.



Figur 29: Rediger bytteopgaver

På figur 29 bekræftes, at opgaveinformationer (titel, beskrivelse, point, slutdato) kan redigeres, og en sælger er tilknyttet. Dette bekræfter, at Scenario B: Rediger Opgave (Opdatér) fungerer til at ændre og gemme opgavedetaljer.

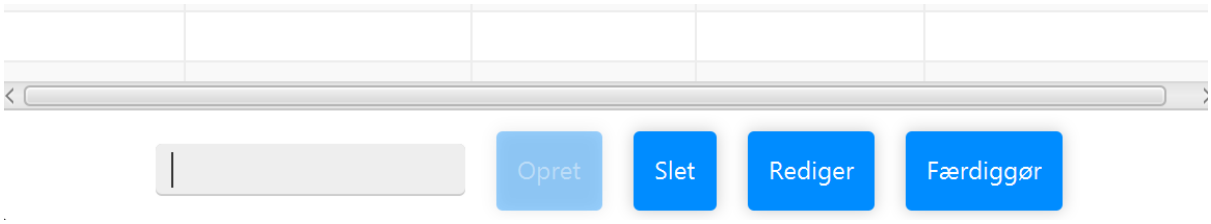


Figur 30: Tilføj/fjern køber

På figur 30 vises beboere, der kan vælges til at blive tilknyttet eller fjernet som købere af bytteopgaven. Dette dokumenterer, at Scenario BB: Tilføj/Fjern Køber fungerer korrekt for at administrere købertilknytningen.

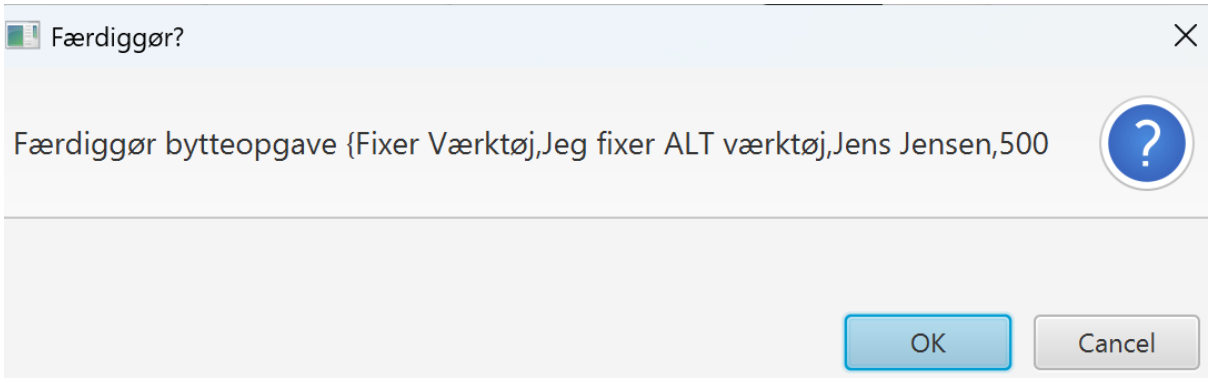
Figur 31: Slet bytteopgaver

På figur 31 sikrer bekræftelsesvinduet, at administratoren bekræfter sletningen af en bytteopgave permanent. Dette understøtter Scenario C: Slet Opgave og sikrer, at opgaven fjernes fra listen efter bekræftelse.



Figur 32: Søg efter bytteopgaver

På figur 32 ses søgefunktionen, der muliggør filtrering af opgavelisten baseret på indtastet søge-argument. Dette bekræfter, at Scenario D: Søg efter Bytteopgave fungerer ved at opdatere listen straks og korrekt.



Figur 33: færdiggøre bytteopgaver

På figur 33 sikrer bekræftelsesvinduet, at administratoren bekræfter færdiggøring af en bytte-opgave herefter fjerner opgaven fra listen og giver/fjerner individuelle point fra de involverede beboere. Dette dokumenterer, at Scenario E: Færdiggør Opgave kan bekræftes.

6 Resultater

Tidligere i afsnit 2.2 er der lavet en række funktionelle og ikke funktionelle krav for systemet. Her bliver der samlet op på, hvilke krav der er overholdt og, hvilke der mangler.

Krav	Opfyldt
Krav 1	Ja
Krav 2	Ja
Krav 3	Ja
Krav 4	Ja
Krav 5	Ja
Krav 6	Ja
Krav 7	Ja
Krav 8	Ja
Krav 9	Ja
Krav 10	Ja
Krav 11	Ja
Krav 12	Ja
Krav 13	Ja
Krav 14	Ja
Krav 15	Ja
Krav 16	Ja
Krav 17	Ja
Krav 18	Ja
Krav 19	Ja
Krav 20	Ja
Krav 21	Ja
Krav 22	Ja
Krav 23	Ja

Tabel 2: Oversigt over krav og om de er opfyldt

Ovenstående tabel 2 alle krav givet via ID om de er opfyldte eller ej. Her kan dog ses på den ovenstående tabel at alle krav er opfyldte. Det er ikke alle use-cases som er testet i afsnit 5 test. Alle use-cases implementation er dokumenteret i Brugermanuelen under Bilag[12].

Kravene er dog delt op i administrere beboere, administrere bytteopgaver, administrere fællesopgaver, administrere grønne opgaver og vis brugeroverblik. Fordeling af krav over disse usecases kan ses på tabel 1 i analyse afsnittet.

Administratoren har en GUI med adgang til alle beboerdata, beboere kan registreres med relevante oplysninger,søgning, tilføjning og fjernelse af beboere er understøttet i systemet hvilket opfylder kravene 1-5.

Administratoren har adgang til en GUI med alt data for bytteopgaver, her er det muligt at registrere nye opgaver, fjerne opgaver, tilføje sælger, fjerne sælger eller færdiggøre opgaver. Det er muligt for administratoren at opsøge på opgaver ved hjælp af et søgefelt. Her opfylder søgefeltet de søgekriterier som er sat i use-case-beskrivelserne hvilke kan findes i Bilag[7]. Systemet opfylder kravene 6-9+20.

Administratoren har adgang til en GUI med alt data for fællesopgaver, her vises alle opgaver på en liste, hvor det er muligt at søge på specfikke opgaver via et søgefelt, der kan tilføjes flere beboere på opgaven, en opgave kan redigeres og en opgave kan færdiggøres. Det er muligt for administratoren at gå ind på en beboer og give dem ekstra point for udførelsen af en opgave. Beboere får automatisk point ved færdiggørelse af en opgave, hvilket de kan bruge på bytteopgaver. Systemet opfylder kravene 10-13+24.

Administratoren har adgang til en GUI med alt data for grønne opgaver, her vises grønne opgaver på en liste, det er muligt for Administratoren at søge på listen, tilføje beboere, redigere opgaven, slette opgaven og færdiggøre opgaven. De grønne opgaver vises på en hjemmeside som beboerne kan tilgå på diggas.dk/Project1/. Når en grøn opgave udføres af en beboer og administratoren færdiggør opgaven for beboeren går point automatisk til kløverly fælleskontoen. Systemet opfylder kravene 14-19.

Hjemmesiden kan tilgåes offentligt på diggas.dk/Project/, hjemmesiden viser både grønne opgaver, bytteopgaver, fællesmål, fælleskonto af point, point-mål og har en om os side der deler kløverlys idealer med omverdenen med et responsivt layout. Systemet opfylder kravene 26+23.

Systemet opfylder alle krav som er stillet i analysen afsnit 2.2.

7 Perspektivering

I afsnit 6 kan det ses, at alle krav for projektet som stillet i afsnit 2.2 er implementeret. Kravene dækker dog ikke hele problemfeltet, da den er afgrænset i projektbeskrivelsen, som kan findes i Bilag[3]. Afgrænsingens fokus lå ikke på tekniske aspekter af projektet, som ikke ville være mulige, men sociale, økonomiske og politiske problemer, som ikke ville kunne løses med en applikation, som ønskes.

Derudover er der en række tekniske valg, som er blevet truffet under Design og Implementation af projektet, som ville kunne gøres anderledes for muligvis at give Bob et bedre produkt. Under design og implementation har der ikke været tænkt over, at det kunne være praktisk, at kunne sortere igennem opgaverne ved hjælp af en filterfunktion. Det kunne f.eks. være at sortere opgaver fra 300 point til 400 point, eller sortere opgaver efter slutdato mellem 2 datoer, som f.eks. alle opgaver, der slutter, mellem den 12. januar og 12. februar. Et andet design-valg som kunne være truffet anderledes ville være muligheden, for at give pointbonus, efter færdiggørelse af en fællesopgave. I det nuværende design skal der manuelt redigeres en beboers point, for at give dem en bonus. Yderligere kunne der have været et større fokus på grøn kode, både i implementation og design principper.

I designet blev der valgt, at implementere et TabPane, hvor hele hovedmenuen af GUI'en skrives. Det kan blive et skaleringsproblem, fordi det er svært at ændre småting og finde rundt i. I stedet kunne man have have lavet et view til hver tab i TabPane, for nemmere at kunne finde rundt i det, når der senere skal laves ændringer.

Der er nogle tekniske limitationer i projektet, som skyldes manglende kompetence. F.eks. kunne data lagres på anden vis end i en JSON-fil. Et eksempel på en bedre løsning havde været at lave en database, som også ville kunne opdatere opgaverne automatisk på hjemmesiden, i stedet for at der skulle importeres en JSON-fil. I den nuværende implementation gemmes hele VillageModel i en JSON-fil, der bruges til hjemmesiden, selvom alt dataen fra filen ikke bruges på hjemmesiden. Dette er ikke den mest grønne beslutning, da det kræver mere tid og drivkraft. Derfor kunne der have været gemt en mindre JSON-fil til hjemmesiden. Derudover kunne en anden forbedring til systemet være, at reducere administratorens arbejdsbyrde. Lige nu virker systemet ved, at beboerne direkte kontakter administratoren, således at der kan administreres opgaverne for beboerne. Man ville dog kunne fjerne en stor del af denne byrde, ved at give beboerne adgang til at kunne købe bytteopgaverne selv, uden at skulle kontakte administratoren. Her ville en mulighed være, at beboerne kunne gøre det gennem hjemmesiden. Der kunne også have været implementeret selvadministration for de andre opgavetyper, så beboere selv kunne sætte sig på opgaver, se deres point osv.

8 Konklusion

I dette projekt er der udviklet et administrationssystem, bestående af en GUI til en administrator og en offentlig hjemmeside til Kløverlys beboere. Systemet er designet til at tage udgangspunkt i et Interview med den kommende administrator af systemet (bob, 2025). Gennem en systematisk udviklingsproces fra analyse til test er alle 23 funktionelle krav og ikke funktionelle krav som fastsat i afsnit 2.2 blevet opfyldt, hvilket sikrer at systemet opfylder dens overordnede mål. Her er projektets formål den samme som i Problemformuleringen fra afsnit 1.3.

Problemformulering: Hvordan kan man optimere organiseringen og skabe bedre overblik over opgaver og ressourcer i økosamfundet Kløverly, og hvordan kan man inspirere indbyggere og andre i samfundet?

Systemet optimerer organiseringen og overblikket ved at centralisere administrationen af beboere, fællesopgaver, bytteopgaver og grønne tiltag i en GUI, hvilket gør det adminstrative arbejde nemmere. Der er implementeret en automatisk logik, som holder styr på Kløverly's grønne point-konto og de individuelle beboers point-kontoer.
Systemet inspirerer beboere og omverdenen, ved at gøre Kløverlys indsats mere synlig, ved hjælp af en offentlig tilgængelig hjemmeside. Her kan ses alle de grønne tiltag som Kløverly har udrettet og deres fællespoint-konto, samt noget information om det grønne samfund på Om-siden.

Systemet opfylder lige nu alle de krav, som er stillet til det. Der er en række mulige forbedringer, som kunne overvejes ved en mulig opdatering af implementationen. F.eks. bedre håndtering af data eller tilføjelse af filtreringsfunktioner. Disse mulige forbedringer kan læses i afsnit 7 perspektivering.

Litteratur

bob, G. (2025). *Kløverly interview med bob sw-sep1-case, efterår 2025.* (VIA University College, Software Engineering case-materiale)

Den Danske Ordbog. (u.d.). *Ordnet – bofællesskab.* Retrieved 2025-10-01, from https://ordnet.dk/ddo/ordbog?query=bof%C3%A6llesskab

Himmerlandsbyen. (u.d.). *Himmelhaven.* Retrieved 2025-10-01, from https://himmerlandsbyen.dk/omtanke-og-co2/himmelhaven/ (Hentet Oktober 2025 fra Himmerlandsbyen)

Landsforeningen for Økosamfund. (u.d.). *Home.* Retrieved 2025-10-01, from https://okosamfund.dk (Hentet Oktober 2025 fra Økosamfund)

Magasinet Natur&Miljø. (2025). Fra by til land - fællesskab og omsorg for naturen trækker. *Magasinet Natur & Miljø*, *1*.

Nissen, D. (2016, jan 12). *Hvad er et økosamfund?* Retrieved 2025-10-01, from https://levendelokalsamfund.dk/hvad-er-et-oekosamfund/ (Hentet Oktober 2025 fra Levende Lokalsamfund)

9 Bilag

Bilag 1 - Projektbeskrivelse V1
Bilag 2 - Projektbeskrivelse V2
Bilag 3 - Projektbeskrivelse V3
Bilag 4 - Grønne bob analyse V1
Bilag 5 - Grønne bob analyse V2
Bilag 6 - Grønne bob analyse V3
Bilag 7 - Grønne bob analyse V4
Bilag 8 - Administrer beboer
Bilag 9 - Bytteopgaver
Bilag 10 - fællesopgaver
Bilag 11 - GrønneOpgaver
Bilag 12 - Brugermanual
Bilag 13 - Klassediagram
Bilag 14 - SekvensDiagramBytteopgaver
Bilag 15 - Wireframe af hjemmeside
Bilag 16 - Wireframe af kontaktside side
Bilag 17 - Wireframe af omside
Bilag 18 - Wireframe af hjemmeside på mobil
Bilag 19 - Gruppekontrakt
Bilag 20 - teamdynamik
Bilag 21 - konfliktstile
Bilag 22 - konflikttrappe
Bilag 23 - fælles
Bilag 24 - adaptiv
Bilag 25 - teamtyper
Bilag 26 - Tidsplan
Bilag 27 - tillpassetVandfaldsmodel
Bilag 28 - YouTrack
Bilag 29 - YouTrackVsTidsplane
Bilag 30 - rigtBillede