Bachelor projekt

Analyse af framework til multiobjektiv optimeringsproblemer

Forfattere:
Martin Dissing-Hansen
Morten Olsen
Peter Nellemann

Vejleder:
Lone Borgersen

30. maj 2012

Resumé

Som følge af den danske regerings 2020 plan, vil det danske energinet skulle gennemgå store reformer. Et af disse er SmartGrid, der vil gøre energipriserne fluktuerende. I danske dambrug er der et stort forbrug af energi, hvorfor reformerne vil have stor indflydelse på deres økonomi. I dette projekt er det forsøgt vist, hvordan multi-objektiv optimering ved hjælp af genetiske algoritmer kan hjælpe de danske dambrug. Der er udviklet et program ved hjælp af Unified Process til at vise, hvordan multi-objektiv optimering kan hjælpe dambrug. Det implementerede program har benyttet et framework, kaldet Controleum, til at løse optimeringen i et fiktivt dambrug. Ud fra simplificerede optimerings objektiver har programmet vist sig brugbart og lægger op til, at en fuld implementation sandsynligvis ikke blot ville kunne hjælpe dambrug at arbejde med SmartGrid, men også kunne optimere og hjælpe med automatisering af den daglige drift. Yderligere vil resultater og sensorinfo brugt til optimeringen også kunne benyttes til bedre overvågning af dambrugets drift og vedligeholdelse, samt have potentiale for at implementere prognoser for sygdomme og deslige. Denne rapport beskriver grundlaget for effektivisering og optimering af dambrug, samt giver forslag til hvordan dette kan videreudvikles.

Kildekode

Kildekode og program kan findes på følgende link:

```
http://www.madis.dk/bachelor/
```

Til projektet er der benyttet NetBeans IDE 7.1.1, sammen med Java SDK 1.7 på Windows 7 64-bit. For at køre programmet skal filerne fundet på hjemmesiden ovenfor åbnes i NetBeans. Hvis der er fejl i libraries skal disse fjernes og tilføjes igen. Filerne ligger i /lib mappen. For at køre programmet, er det PTestGUI.java, beliggende i Presentation-pakken, der skal køres.

Acknowledgements

Dette projekt har ikke kunnet lade sig gøre uden hjælp fra en række personer. Vi vil gerne takke Lone Borgersen for hendes tid og store arbejde som vores vejleder, for gennemlæsning af halvfærdige kapitler og gennemgang af de mange revisioner og designdokumenter. Vi vil gerne takke Bo Nørregaard Jørgensen og holdet bag Controleum-frameworket for deres arbejde på et spændende projekt og deres hjælp til at implementere dette i vores projekt. Vi vil gerne takke Morten Priess fra AquaPri, samt Svend Christensen og Lars Bjerregaard fra Lykkegaard A/S for deres tid til interviews og fremvisning af henholdsvis Lerkenfeld dambrug og Lykkegaard A/S faciliteterne. Også en tak til Mogens Brabech fra TEK Momentum for hans hjælp med at arrangere disse interviews, og hans tid til at fragte os frem og tilbage til selvsamme.

Indhold

1 Introduktion

Med stigende el-priser, som følge af verdens faldende reserver af fossile ressourcer, har den danske regering anno 2011 fremsat en 2020 plan. Planen involverer en 100% konvertering fra fossile brændstoffer til vedvarende energi og biobrændsel, herunder især vind-, sol- og bølge-energi. En bevægelse væk fra det nuværende, mere konstante niveau af elforsyning, der har basis i kulkraftværker samt energi-import fra nabolande, vil resultere i store udslag i energipriserne i dagens og ugens tidsintervaller.

Danmark er samtidigt et af verdens førende lande indenfor produktion af især fisk fra dambrug. Dambrugene er meget energiafhængige, idet de kunstige damme benytter avancerede pumpesystemer til filtrering og afgasning af vand, iltning, fodring, med mere. Med højere elpriser er disse virksomheder i risikozonen, idet deres forretning har udsigt til potentielt ikke længere at være rentable. På længere sigt vil de fluktuerende priser på el skabe endnu større problemer, idet mange processer har begrænsede tidsintervaller, hvori de kan køre, men samtidig er de livsvigtige for det levende produkt. Da strømpriserne kan ændre sig meget hen over dagen, kan det give øgede udgifter, hvis de strøm-intensive processer bliver kørt mens strømmen er dyr.

2 Formål

Dette projekt har til formål at undersøge de problemstillinger, der er skabt for danske dambrug som følge af højere elpriser og udsigt til meget fluktuerende priser på samme. Der vil gives et løsningsforslag inden for dette specifikke domæne. Hypotese er, et multiobjektivt optimeringssystem vil kunne finde besparelser for dambrug, der ligner det, hvori der tages udgangspunkt. Dette gøres ved at implementere et system, der bygger på Controleum, et multi-objektivt optimeringsframework til væksthuse, lavet under ledelse af Bo Nørregaard Jørgensen, og overføres til styring et dambrug og optimering af rentabiliteten ved at forudse priser på el og udviklingen af miljøet i dambruget. Med udsving i dagspriserne baseret på vejrudsigter samt forbrug hen over døgnet, skal systemet lave kvalificerede forudsigelser på status frem i tiden givet forskellige parametre. Systemet skal ud fra dette komme med kvalificerede bud på styring af et dambrugsanlæg til minimering af el forbruget og udgifterne på brugt el. Systemet skal også kunne udvides med flere agenter, så det vil kunne tage stilling til flere parametre, såsom vækst-hastigheder, udgifter til foder, filtre, mm.

3 Problemstilling

Hvordan kan optimeringsteknikker hjælpe dambrug med at optimere deres systemer og samtidig minimere deres udgifter til energiforbrug?

1. Hvilke relevante domæne-specifikke elementer kan bruges til at udvikle et system, der skal styre et dambrug?

2. Hvordan kan et system bedst udvikles, der både er nemt at udvide og optimalt til at løse multioptimeringsproblemer?

4 Mål

1. Opnå forståelse for løsning af multiobjektive optimeringsproblemer ved hjælp af multiagent-systemer og genetiske algoritmer.
2. At få kendskab til dambrug og pumpesystemer til udformning af agenter for domænespecifikke interesser.
3. Udvikle et proof-of-concept system for dambrug, som bruger multioptimeringsframeworket, for at vise at det også kan anvendes i dette domæne.
4. Analyse af hypotesen om multiobjektive optimeringsteknikker som hjælp til at optimere energiforbrug i et dambrug. Herunder også analyse af alternativer til udvalgte områder af frameworkets implementering.

4.1 Uddybning af mål

Ad 1: Det er nødvendigt at få et gennemgående kendskab til multiagent-systemer, genetiske algoritmer og multiobjektiv optimering for at kunne sammenligne det med alternative metoder for løsning af problemet.

Ad 2: Da proof-of-concept systemet er baseret i et helt nyt domæne, dambrug, vil det være nødvendigt at indhente ny viden om de mere tekniske aspekter i domænet. Dette involverer, men er ikke begrænset til, pumpesystemer, vandkvalitet, fiskeadfærd samt fodring.

Ad 3: Her vil der blive udviklet agenter til et nyt domæne og få dette til at virke med en simpel brugerflade. Dette system vil ikke kunne tage imod live-opdateringer, men derimod kun benytte konstruerede data.

Ad 4: Målet med dette er at finde ud af, hvor godt det givne framework løser problemet, og opnå forståelse for de valg der er blevet taget undervejs i implementeringen.

5 Forventede resultater

- Et proof-of-concept system for et dambrug, som bruger multioptimeringsframeworket.
- En dokumenteret bekræftelse på at multi-agent systemer i samarbejde med genetiske algoritmer egner sig til løsning af multiobjektive optimeringsproblemer i dambrug.
- Et grundlag til viderebygning af et fuldbyrdet Control System for dambrug.

6 Metode

6.1 Unified Process

Unified Process (UP) er en software udviklings process, der definerer en fremgangsmåde til at udvikle software på en organiseret og effektiv måde. UP er

ifølge [Jim Arlow, 2008, Wikipedia, 2012c] baseret på de følgende tre grundsætninger: UP er krav og risiko drevet, UP har fokus på arkitektur, og UP er en iterativ og inkrementel process.

At UP er krav og risiko drevet betyder, at udviklings arbejdet i UP er baseret på en krav-indfangning, der laves i samarbejde med, eller baseret på, kundens behov. I denne sammenhæng er krav de krav, som det færdige system skal opfylde eller kunne udføre. Disse krav bruges i det videre udviklingsarbejde, hvor de bliver brugt til at skabe brugsmønstre for systemet. Disse brugsmønstre bruges herefter i det videre udviklingsarbejde. At UP er risiko drevet betyder, at der skal lægges vægt på analyse af risici ved forskellige dele af software udviklingen, således at de mest kritiske risici kan blive behandlet tidligt i projektets forløb.

UP har fokus på arkitektur, hvilket betyder, at der skal beskrives, hvordan et system skal nedbrydes i komponenter, samt hvordan disse skal arbejde sammen og placeres i hardware.

Endelig siges det at UP er en iterativ og inkrementel process. UP består af fire faser, Inception, Elaboration, Construction og Transition. Hver af disse er så delt op i et antal tidsbegrænsede iterationer. Hver af disse iterationer består af arbejde i hver af de seks discipliner: forretnings modellering, krav indfangning, analyse og design, implementering, test og deployering. I hver af disse iterationer vælges et sæt af brugsmønstre, der behandles fra krav til implementering, test og design.

I dette projekt er der brugt UP til at styre software udviklingen. Da dette projekt er af en relativ lille størrelse og kun har et begrænset antal brugsmønstre, har der ikke været så mange iterationer, som der ville have været i et større projekt, men der har været en række mindre iterationer til at skabe et godt produkt. En fordel ved denne måde at arbejde med software udvikling er, at i løbet af iterationer vil der ofte dukke nye detaljer op, der ikke er blevet afdækket i første omgang. Disse kan så behandles i følgende iterationer, så de kan blive udbedret.

Krav-indfangning i projektet er baseret på interviews med henholdsvis AquaPRI og Lykkegaard Pumper. Baseret på disse interviews er der udarbejdet et antal brugsmønstre, der ligger til grund for det videre arbejde indenfor UP. Dette arbejde udmunder i analyse modeller, sekvensdiagrammer, design model og avancerede sekvensdiagrammer. På basis af de førnævnte øvelser kan er der lavet en implementering, og efterfølgende test af denne implementering.

6.2 Interviews

Projektet udnyttede to interviews for at få udvidet information om dambrugsdomænet. Disse interviews var henholdsvis med Morten Priess fra AquaPri, samt Lars Bjerregaard og Svend Christensen fra Lykkegaard Pumper. Begge interviews blev foretaget med diverse Knowledge Management teknikker. Der blev afsendt en række spørgsmål til begge parter inden hvert interview, der havde til formål at vise, hvilke emner der var interessante for projektet, samt spørgsmål med tekniske svar, der sandsynligvis ville tage tid at finde svarene på. Resultaterne af interviewet med AquaPri ligger til grund for domæne afsnittet om generel viden om dambrug. Resultaterne af interviewet med Lykkegaard A/S er udmundet i definitioner af programmets komponenter, som styrer optimeringen i Controleum, dette vil blive forklaret i et senere afsnit.

6.2.1 Den Naive Interviewer

Begge interviews blev foretaget under konceptet "Den Naive Interviewer" fra [Irma Beceraa-Fernandez, 2010]. Denne teknik er baseret på idéen, at en ekspert er glad for sit felt og ved hvilke ting inden for feltet, der er de vigtige. Her laver man som interviewer et grundigt forarbejde mht. domænet. På basis af dette stilles eksperten et generelt spørgsmål om domænet, og man lader herefter denne fortsætte med de ting, denne finder vigtigt. Der kan spørges efter uddybende svar og nye spørgsmål kan stilles, hvis eksperten går i stå, men generelt opfører intervieweren sig passivt og lader eksperten føre samtalen. Denne teknik bruger også ofte en anden, lignende teknik, der hedder "Storytelling". Denne teknik benytter igen en ekspert til at fortælle relevante historier omkring sit domæne, hvorfra detaljer og informationer uddrages [Irma Beceraa-Fernandez, 2010]. Efter afsluttet interview blev optagelser og noter efterbehandlet, hvor relevant information blev renskrevet og gemt, så det var nemt tilgængeligt under færdiggørelsen af projektet.

6.2.2 Interview med Morten Priess

Interviewet med Morten Priess fra AquaPri, et dansk Dambrugsfirma, foregik på Lerkenfeld dambruget i Himmerland. Dette var delt i to, hvor første del foregik sammen med Morten Priess i dambrugets stuebygning og blev optaget via telefon med diktafon funktion. Herefter fremviste han anlægget udenfor, hvor der blev taget noter, idet vinden gjorde brug af diktafon ubrugeligt. Interviewet gav en del generel viden om et moderne dambrugs opbygning samt forskellige metoder til iltning, fodring mm. Det største udbytte fra dette interview var dog et indblik i et dambrugs økonomi og en større indsigt i hvilke steder, der bruges flest penge. Interviewet gav herudover et indtryk af, at ikke alle indstillinger var helt så justerbare som forventet, men gav til gengæld større forståelse for, hvordan ting som fodring påvirker andre komponenter og statusser, heriblandt iltning og vandrensning.

6.2.3 Interview med Svend Christensen og Lars Bjerregaard

Første del af interviewet med Svend Christensen og Lars Bjerregaard fra Lykkegaard A/S, en virksomhed på Fyn der bygger avancerede pumper og tester disse på deres eget anlæg, foregik i virksomhedens mødelokaler. Her fortalte de to eksperter om deres viden indenfor pumper, dambrug og gik videre til at dække over fodring, temperatur og iltsikring. Alt dette blev ligeledes optaget via diktafon og behandlet senere. Anden del foregik på pumpevirksomhedens areal, hvor test tanke og konstruktions faciliteter blev fremvist. Dette interview gav meget store resultater i forhold til indstillinger af komponenter, hvilke komponenter, der skulle fokuseres på og dambrug generelt. Interviewet gav også en større indsigt i, hvordan forskellige aspekter af et dambrug påvirker andre, der ikke altid er lige tydelige, når man ser på dem fra en udenforståendes synspunkt.

6.3 Case Study

En case study er ifølge [Dawson, 2009] en dybdegående undersøgelse af en situation. Dette involverer udforskning af en bestemt situation, et bestemt problem eller en gruppe af firmaer. Strategien for en case study er at udvælge et specifikt

område, og kunne uddrage generel viden fra dette. Derfor er det for det første vigtigt at vælge et passende område for et case study, og det er for det andet vigtigt at forholde sig kritisk til resultater fra et case study, da det kan være problematisk at forsøge at drage konklusioner på basis af dette.

Den valgte case study i projektet fokuserer på situationen omkring stigende energipriser, for at finde ud af hvad der ligger til grund for dette og se hvilken effekt den har på et specifikt domæne. Det valgte domæne er dambrug, hvor hypotesen er at en optimering vil kunne minimere energiforbruget.

7 Domæne

7.1 SmartGrid

I forbindelse med regeringens 2020 plan skal det danske elnet ændres markant. Planen går i al sin simpelhed ud på, at Danmark skal modtage store dele af sin energi fra vedvarende energikilder [Regeringen, 2010]. I forbindelse med dette skal energinettet omstruktureres, således det bliver optimeret til en mere variabel forsyning af strøm. Dette skyldes, at energiformer som bølgekraft, vindkraft og strøm fra solpaneler fluktuerer afhængigt af vejr, vind og tid på dagen. Idet der stadig ikke findes rentable metoder til opbevaring af overskydende kraft fra disse, må elnettet derfor skulle indfinde sig med den førnævnte variable strøm. Samtidig vil elbiler og udskiftning af varmesystemer hos borgerne ændre og øge behovet for mere energi. Et forslag til løsningen af dette problem er den såkaldte SmartGrid, som er udarbejdet i samarbejde mellem EnergiNet og Dansk Energi. Denne løsning tager også højde for variationer i energibehovet fremfor de forudsete, samt antager, at al energien skal komme fra ikke-fossile brændstoffer [og Dansk Energi, 2010].

7.1.1 Antagelser

SmartGrid bygger på en række antagelser, hvoraf de vigtigste er:

- At vindmølle kapaciteten udbygges til at kunne dække 50% af energi behovet
- At der samlet set er 600.000 el- eller hybrid biler
- At der er 300.000 individuelle varmepumper i de danske hjem

Derforuden antages en stor ombygning af det gamle energinet til et nyt net, der ikke længere forholder sig relativt passivt til behovet. Fremfor at udbygge produktion og transport af strøm, skal SmartGrid nettet fungere ved intelligente systemer, der sikrer en balanceret fordeling af strømmen, både fra elproducenternes side, men også ved hjælp af systemer placeret hos de enkelte forbrugere [og Dansk Energi, 2010].

7.1.2 Resultater

SmartGrid bliver foreslået ikke kun fordi et anderledes net kræver anderledes teknologi, men også fordi det vil skaffe den største økonomiske og miljømæssige gevinst. En udbygning af det traditionelle net antages at koste 7,7 milliarder kroner for at kunne række til det estimerede behov uden nogen økonomisk gevinst,

mens en udbygning til SmartGrid vil koste ca. 9,8 milliarder, men resultere i en gevinst på knap 8,2 milliarder kroner i form af besparelser [og Dansk Energi, 2010]. SmartGrid vil samtidig ændre prisstrukturen fra den nuværende, faste struktur til en variabel struktur med prislister, der justerer strømmens pris i forhold til produktion og overordnet forbrug.

7.1.3 Relevans i forhold til dambrug

Dambrug har store udgifter til strøm til pumper og filtre, dette gælder især de mere kompakte dambrug som det fremviste Lerkenfeld dambrug, hvor knap 50% af udgifterne går til strøm [Morten Priess, 2012]. I den forbindelse er det naturligt at ville undersøge, hvorledes man kan optimere på et dambrugs systemer, så det bruger færre penge på strøm. Idet SmartGrids pris struktur snart implementeres nogle steder i erhvervslivet, blandt andet Lerkenfeld dambrug, er priser på strøm pludseligt variable, hvormed en intelligent strømjustering potentielt kan spare firmaet en del penge. I den forbindelse vil et optimeringssystem for et dambrug kunne optimere på mere end bare driften af dambruget. Et intelligent system til optimering vil kunne se på en liste over dagspriser og bruge denne til at tage beslutninger om blandt andet fodring med viden om, at dette vil øge energibehovet. Hermed vil systemet kunne tage energitunge beslutninger i perioder med lave elpriser og potentielt skabe besparelser i forhold til blot at køre systemet, som man gør det nu.

7.2 Dambrug

Der findes mange forskellige typer dambrug. Der er to overordnede typer: saltvand og ferskvands dambrug. Saltvands dambrug består af indelukker i havet, hvor en eller flere arter af fisk, muslinger og/eller tang opdrættes. Ferskvands dambrug er indelukker på land i form af, oftest, kunstige søer af den ene eller anden slags, hvori ens produkter opdrættes. Ferskvands dambrug har, i modsætning til saltvandsdambrug, ikke naturen til at hjælpe med gennemstrømning og vandrensning i et stort omfang. På nær hobby dambrug, der foregår i en egentlig sø, er deciderede profit-orienterede dambrug langt mere intense, og kræver derfor maskinel hjælp til iltning, vandfiltrering og - rensning med mere [Svend Christensen, 2012]. Gammeldags dambrug benytter afdæmning af en vandåre, hvor vandet føres ind i et anlæg af søer, ofte opsat som en mængde aflange søer, også kaldet "raceway"anlæg. Spildevandet føres derefter ud i vandåren igen. Moderne anlæg er også ofte opsat som raceway anlæg, men med genbrug af vandet der renses og filtreres, inden det genindføres i bassinerne. Andre anlæg er opsat som store, runde bassiner, hvor vandet pumpes igennem et renseanlæg, ned i runde bassiner og ud igennem bunden. Dette sikrer en kontinuerlig bevægelse af vandet, samtidigt med at det kan cirkuleres effektivt på mindre plads. Disse anlæg er meget afhængige af teknik for at sikre iltning og rensning til de store mængder biomasse, det vil sige den totale masse af produkt i tankene. Anlægget hvori der tages udgangspunkt i er et af Aquapri's anlæg i Nordjylland, der hedder Lerkenfeld. Anlægget er bygget op af gylletanke, idet dette har vist sig at være billigt og effektivt. Anlægget er delt op i underanlæg, hver indeholdende fire tanke til fisk samt en filter tank og en biofilter tank. Det ene anlæg har desuden på forsøgsbasis et ekstra bassin til dyse-iltning [Morten Priess, 2012].

7.2.1 Fisketanke

Fisketankene består af 4 meter dybe tanke med ca. 75 cm over jorden. De er hver ca. 6 meter i diameter. Vand kommer ind fra et tilløb i toppen og sikrer en cirkulation i tanken. Der er et afløb i bunden samt et i toppen, hvor vandet bliver sendt videre til en tank med et mekanisk filter. Afløbet i toppen sikrer, at større genstande, såsom grene og døde fisk, der flyder til tops, vil blive frasorteret hurtigt, således at disse ikke rådner i bassinet og afgiver giftstoffer til vandet. Fisketankene har også et flydende net af rør, der pumper atmosfærisk luft ind i den øverste halve meter vand. Dette er til afgasning og iltning, hvor vandets sammensætning af atmosfæriske stoffer normaliseres, det vil sige blandt andet kuldioxid (CO_2) kommer ud og ilt (O_2) kommer ind. På længere sigt skal der også installeres keramiske iltsten til dedikeret iltning af tankene [Morten Priess, 2012].

7.2.2 Filtertank

To pumper pumper vandet fra filter bassinet op i et biofilter bassin. Vandet bliver herved trukket igennem et finmasket mekanisk filter, der med ca. 10 sekunders intervaller drejer og renser sig selv ved brug af jetdyser. Slammet herfra bliver pumpet over i et sump bassin.

7.2.3 Biofiltertank

Det filtrerede vand bliver herefter cirkuleret rundt i biofilter bassinet sammen med tusindvis af biokugler; små plastik kugler med en bakterie belægning, der nedbryder nitrat, nitrit og ammonium i vandet.

7.2.4 Dysebassin

Endelig pumpes det rensede vand tilbage til fisketanken. I bassinerne med jetdyser sker dette via et par mindre bassiner, hvor vandet føres under jetdyser, der tvinger vandet sammen med ilt. Dyserne sikrer et højere tryk, hvorved mere ilt optages i samme mængde vand.

7.2.5 Pumper

For at holde gennemstrømningen af vand i gang i et anlæg, skal der nogle store pumper til. Lerkenfelds anlæg har to pumper til hvert anlæg med fire tanke. Hver pumpe er i stand til at pumpe 150 L vand i sekundet, og er enten tændte eller slukkede. Disse pumper står for at sikre en cirkulation af vandet fra tankene og igennem mikrofilteret samt biofilteret. Udover at flytte vandet betyder disse pumper også, at der skabes en lille mængde iltning, lidt varme grundet bevægelse og friktion imod rør og vægge, samt en bevægelse, der simulerer strømmen i en ørreds naturlige miljø. Pumperne er en af dambrugets store udgifter til strøm, men er samtidig en nødvendighed, da de for det meste kører på fuld kraft døgnet rundt.

7.2.6 Vandrensning

Vandrensningen foregår via tre stadier: *afgasning*, *mikrofiltrering* samt *biorensning*. Afgasningen foregår ved, at store mængder luft pumpes igennem hullede

rør under vandskorpen. Dette hjælper til med at ilte vandet, men får samtidig ændret på vandets balance i form af gasser. Mikrofiltrering er trin to og forgår ved hjælp af et stort mikrofilter. Her presses vandet igennem et filter med 60 μ masker. Dette frasorterer hoveddelen af vandets partikler. Disse frasorteres og ledes hen til et slamopsamlings anlæg. Det sidste stadie er bioanlægget. Det er her, vandet fra pumperne ledes hen i. Pumper i anlægget sikrer en kontinuerlig cirkulation i et stort kar fyldt med biokugler. Disse kugler er opsamlingssteder for forskellige mikroorganismer, der hjælper til med at nedbryde nitrat (NO_2), nitrit (NO_3) mm. i vandet. Herefter bruges tyngdekraften til at sende det rensede vand tilbage i bassinerne.

7.2.7 Iltning

Iltningen forgår i tre stadier: *afgasning*, *pumpning* samt *dedikeret iltning*. Afgasningen er, hvor atmosfærisk luft pumpes ind i bassinerne via hullede rør under vandets overfalde i den øverste meter. Udover at afgasse vandet ilter dette også betydeligt. Imidlertid er det under lavt tryk, hvilket ikke er det meste effektive. Pumpningen foregår ved mikrofilteret, hvor vandet tvinges igennem og op til biorensningstanken, hvor det pumpede vand ledes ned i tanken via et fald. Stor overflade-til-vand ratio her skaber en betydelig iltnings effekt samtidig med at dette også bruges til at føre vandet igennem de nødvendige stadier. Dette stadie negeres dog af biofilterets iltforbrug. Endelig er der dedikerede iltningsmetoder. I nogle af bassinerne bliver der brugt tryk-injektion. Her ledes vandet igennem et mindre bassin med trykjets, der presser vandet ind med et stort tryk og injicerer ren ilt. På grund af det store tryk bliver der optaget mere ilt, men prisen er høj, idet der bruges ilt fra tank samt strøm til at skabe trykket. Et alternativ der er ved at blive indført på Lerkenfeld er iltsten. Til dette lægges måtter med finthullede sten ud i bunden af bassinet. Grundet fire meters dybde er der her større tryk end ved overfladen, og ved at presse ren ilt igennem stenene, skabes der meget fine iltbobler. Tryk samt stor overflade-til-vand ratio betyder en stor iltoptagelse. Imidlertid er det usikkert om dette system vil være nok, eller om jet-systemet er nødvendigt til de mere iltkrævende tanke.

7.2.8 Foder

I det besøgte dambrug benyttes et ganske simpelt foder-system. Systemet består af en tank fyldt op af foder, og en pumpe, der kan blæse foder ud af et rør. Der bliver så blæst en hvis mængde foder ud et passende antal gange i løbet af dagen.

Foderet består af en blanding af vegetabilske produkter, olier og animalske produkter fra fisk. Foderet er lavet i pille størrelse, og størrelsen på disse piller varieres i takt med, at fiskene bliver større. Dette er for at sikre, at fiskene kan nå at spise nok. Hvis pillerne er for små for fiskene, kan disse ikke nå at spise en stor nok mængde.

Der er her to hensyn:

- Der skal fodres så meget som muligt, således at fiskenes vækst bliver maksimeret.
- Der skal undgås spild af foder, hvilket vil sige, at der ikke skal fodres mere, end fiskene kan nå at spise.

Da foderet til disse fisk repræsenterer en betydelig udgift, er det vigtigt at begge disse bliver opfyldt. Af denne grund er det også problematisk at lave et fuld-automatisk foder system, hvis dette system ikke samtidig kan sikre sig, at det foder, der bliver givet til fiskene, rent faktisk også bliver spist. Derfor er det stadig vigtigt at have en opmærksom fiskemester, selvom der bruges et fuld-automatisk foder system.

7.2.9 pH-Værdi

Styring af pH-værdi i bassinerne foregår ved at tilsætte saltsyre hvis pH-værdien er for høj, og kalk hvis pH-værdien er for lav. I opstarts perioden for det besøgte dambrug blev der tilsat saltsyre over en periode for at give den korrekte pH-værdi. Efter dette havde stabiliseret sig, er der ikke blevet tilføjet noget for styring af pH-værdien særlig ofte. I stedet bliver pH-værdien målt med mellemrum, for at sikre at pH-værdien forbliver på et acceptabelt niveau.

Grunden til at pH-værdien ændrer sig i et system som det besøgte, er at når biologiske filtre bruges til at fjerne ammoniak fra vandet, giver det et fald i pH-værdien. Der er derfor behov for at rette dette, hvis optimale forhold skal opretholdes. Problemer med pH-værdien kan både forekomme ved for høje og for lave værdier. For akvakultur faciliteter anbefales det at have pH-værdier imellem 6,5 og 9 [Lekang, 2007].

7.2.10 Temperaturer

I det konkrete besøgte dambrug, er der en begrænset styring af temperaturen. Dette er der to grunde til, for det første er temperaturen i forvejen ganske stabil, grundet den store mængde vand i de dybe bassiner, der sikrer en stabil temperatur grundet jordvarme. For det andet, og måske mere vigtigt, ville det være økonomisk set uoverskueligt at styre temperaturen direkte i et anlæg som det besøgte. Der er imidlertid problemer med dette, hvis temperaturen når til ekstreme yderpunkter. Hvis temperaturen for eksempel kan svinge imellem 0 og 20 grader, er der problemer i begge yderpunkter, da der her er for koldt eller for varmt til det er muligt at fodre fiskene.

Dambruget har derfor planlagt at anskaffe en række skygge-net, der skal dække bassinerne. Med disse skygge-net vil det være muligt at holde temperaturen inden for et mere stabilt område, og dermed gøre det muligt at fodre alle årets måneder.

I andre typer af dambrug er der mere direkte styring af temperaturen. Et eksempel her på er dambrug hvor der skabes yngel. Disse dambrug er anderledes opbygget, og der er dermed mulighed for at styre temperaturen. Målet for denne styring af temperaturen er at skabe kortere årstider, således at fiskene yngler oftere.

7.2.11 Udledning i naturen

Traditionelt har tilladelser til dambrug været givet med et loft på fodermængden. Dette betyder, at det har været i dambrugenes interesse at få mest fisk per foder enhed og ignorere udledning i naturen af rest stofferne, hvilket igen har betydet, at dambrug traditionelt har været store syndere angående forurening. Med ny lovgivning er dette ved at ændre sig, idet dambrug skal flyttes til en ny

ordning, hvor der i stedet er kvoter for udledning af diverse stoffer, samt forbrug af vand, herunder grundvand, postevand og andre former for indvundet vand. Dette betyder, at dambrug kan placeres uden frygt for ødelæggelse af det lokale vandmiljø, men også opfordrer til effektivisering af det individuelle dambrug. Dambrugene må nu have så stor produktion, de ønsker, så længe de blot holder sig indenfor deres kvoter.

7.2.12 Økonomi for energiforbrug i besøgt dambrug

Anlægget bruger store mængder strøm. Energi-forbruget i firmaet AquaPri [Morten Priess, 2012] er på omkring 10% af omsætningen. De bruger omkring 1.000.000 kr. per år bare på energi til et dambrug. Der blev lavet en energiaftale i 2009, så stigende el-priser har ikke betydet noget endnu, da deres aftale ikke er udløbet. Men det kommer de til, når deres aftale udløber. Da hele virksomheden bruger cirka 10.000.000 kW om året, vil en stigning på bare 1 øre betyde en ekstra udgift på 100.000 kr. om året.

Da dambruget har behov at fortsætte med et stort energiforbrug for at holde systemerne kørende, vil de nye energiregulering regler betyde nye problemer for dambruget. Når dambrugets eksisterende aftale med faste priser udløber, vil de komme over på en ny model, hvor priserne ændrer sig fra time til time. Dette kan give potentielle udsving i prisen på energiforbruget, især hvis energikrævende processer bliver placeret i perioder med høje priser på elektricitet.

Til sammenligning brugte Lerkenfeld under deres gamle lagune dambrug kun cirka 3000 kW om måneden og var derfor ikke så afhængig af energi. De dele som bruger allermest energi i dambruget er pumperne, der løfter vandet til et andet bassin, dernæst kommer luftpumperne, som står for størstedelen af det resterende energiforbrug.

7.3 Domænemodel

Domænet, hvori der arbejdes, er dambrug. For at modellere domænet på en letforståelig måde, er det nødvendigt at dele det op i undergrupper med en letforståelig opdeling. Generelt set består et dambrug af tanke, vand, fisk, pumper, filtre og iltningsmetoder. Dette blev delt op i følgende undergrupper:

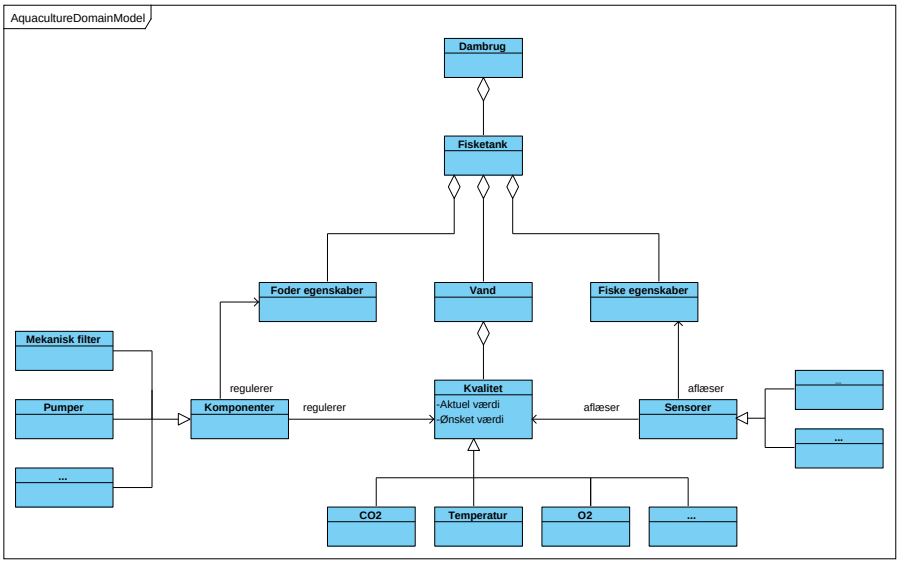
- Sensorer
- Produkt
- Vand
- Komponenter

Sensorerne består af en lang række måleapparater, der kan se på status af systemet. Disse inkluderer, men er ikke begrænset til, O_2 niveau, CO_2 niveau, temperatur og ammonium-koncentration. Sensorer er alle passive dele af systemet, disse kan afgive informationer om systemet, men kan ikke agere på det. På baggrund af disse kan vandkvaliteten beregnes og beslutninger tages angående komponenter.

Komponenter kan, i modsætning til sensorerne, påvirke systemet. Komponenter inkluderer blandt andet pumper, filtre, iltnings metoder og pH regulatorer. Vand kan tilføjes eller fjernes via komponenterne, og kvaliteten på dette

beregnes ud fra den information, sensorerne afgiver. Kvaliteten kan herefter påvirkes via forskellige komponenter.

Produktet lever i vandet. Sensorinformation og historik benyttes til at beregne kvaliteten på produktet og dettes helbred. Forskellige komponenter sikrer at vandkvaliteten er optimal for produktet, imens andre sikrer fodring og administrering af produktet.



Figur 1: Domæne model, stor udgave kan ses på Figur 20, s. 65.

8 Teori

8.1 Teori generelt

Indenfor dette projekt benyttes en række principper, der vil blive forklaret i det følgende teori afsnit. Det drejer sig om områder indenfor frameworks, genetiske algoritmer, multi-objektiv optimering og multi-agent systemer. Hver af de følgende sektioner er baseret på en eller flere artikler, der er blevet læst og analyseret, og vil blive brugt i det videre arbejde i projektet.

Udover en gennemgang af hvert emne, der er beskrevet i artiklerne, vil der blive draget en sammenhæng med projektet, og givet en kort beskrivelse af hvordan emnerne vil blive brugt i projektet.

8.2 Frameworks

Inden for programmering er det meget populært at bruge frameworks, det værende nye eller gennemtestede generelle frameworks. Et framework er en

abstraktion, hvori fælles kode kan give generisk funktionalitet, som så kan blive overskrevet eller specialiseret ud fra specifikke krav til en specifik situation [Wikipedia, 2012a].

Hvis man ikke bruger et framework, kan man risikere at adskillige problemer optstår, heriblandt at have kode som ikke er svær at modificere eller videreudvikle. Dette skyldes, at hvis en del af koden ændres, ved man ikke hvad det har indflydelse på, og derfor kan andre dele af programmet, der benytter den ændrede del, få deres funktionalitet ødelagt. Dette gør det sværere at udvikle på, da man hver gang skal være sikker på hvilke dele, der har indflydelse på den ændrede del eller benytter sig af denne. For at undgå dette kunne man bruge et framework, som løser lige præcis dette problem, for eksempel ved hjælp af interfaces, så alle kald på tværs af klasser sker gennem disse.

8.2.1 Hvorfor Frameworks?

Frameworks kan betragtes som værende en fordel i de fleste felter indenfor programmering, men spørgsmålet er; hvorfor man egentlig skal bruge frameworks. Det korte svar er ifølge [Michael Mattson, 1998] "genbrugelig kode, oftest nemt udvideligt og nemt at fange bugs idet disse er isoleret til ens egen kode". Det længere svar, derimod, kræver en større indsigt i frameworks. Denne viser, at der både er positive og negative sider ved brug af frameworks, men at de positive oftest vejer tungest, og dermed gør frameworks til tid- og pengebesparende metoder. Følgende er et indblik i framework-udvikling og -brug, hvor der kigges på problemer i de forskellige faser samt løsninger på disse.

8.2.2 Faser i Framework-udvikling

Indenfor framework udvikling er der tre overordnede faser: *Framework Development Fasen*, *Framework Instantiation Fasen* og *Framework Evolution and Maintenance Fasen* [Michael Mattson, 1998].

- *Framework Development* fasen består af skabelsen og dokumentationen af et framework.
- *Framework Instantiation* fasen består af udvidelsen af et framework med domæne specifik kode. Her benyttes et frameworks abstrakte klasser til at skabe et fuldgyldigt program til slutbrugerens specifikationer.
- *Framework Evolution and Maintenance* fasen er vedligeholdelse af det færdige program, efter levering. Her udvides, skabes eller ændres alt efter kundens ønske samt eventuelle bugs rettes.

Framework Development fasen

Dette er den sværeste fase rent designmæssigt. Et framework skal ikke blot designes robust men også relativt alsidigt, og alle funktionaliteter der skal tilgås af domæne specifik kode, skal grundigt dokumenteres, og det skal samtidigt sikres, at de ikke kan benyttes uhensigtsmæssigt. Når et framework designes, skal man tage hensyn til, om man laver et "kaldet"eller et "kaldende" framework [Michael Mattson, 1998]. Forskellen ligger i måden, hvorpå et framework håndterer den domæne specifikke kode på: hvis et framework betragtes som et traditionelt bibliotek i kodning, så kaldes det af koden. Langt oftere vil et

framework dog være kaldende, hvor det er frameworket der laver arbejdet og kalder den ekstra kode, når denne behøves til specifikke opgaver. Dette kan give store problemer i et større projekt, hvor der benyttes to eller flere kaldende frameworks. Her skal man pludselig til at håndtere to frameworks, der begge vil bestemme. Løsningen kan være *concurrency*, *wrapping* eller *remove and rewrite*.

Concurrency betyder, at hvert framework køres i en separat tråd. Denne løsning kan kun bruges, når hvert framework ikke har behov for at kommunikere med andre frameworks eller få notifikationer om events fra andre framework.

Wrapping involverer en "wrapper" omkring hvert framework, der behandler alle beskeder sendt imellem frameworks. Denne "wrapper" kan så sende de relevante beskeder videre og bearbejde dem for at forhindre konflikter. Dette virker kun til events og beskeder, der sker udenfor hvert framework.

Endelig er der *remove and rewrite*, der kort og godt betyder, at relevante interne kommunikations funktioner og loops genskrives, så de sendes eksternt og kan findes af en wrapper eller bearbejdes på en hensigtsmæssig måde af andre framework.

Framework Instantiation fasen

I denne fase opstår de fleste problemer med frameworks. Disse kan deles op i tre kategorier: *legacy component problemer*, *framework gap problemer* samt *composition overlap problemer*. Hvert af disse problemer illustrerer tre forskellige områder, hvor frameworks kan løbe ind i en myriade af problemer afhængigt af den specifikke situation [Michael Mattson, 1998].

Legacy Component problemer

Legacy komponenter er gamle metoder, teknologier, computer systemer eller programmer, der stadig bruges, fordi de stadig opfylder brugerens behov, på trods eksistensen af nyere eller mere effektive løsninger. *Legacy component* problemer opstår, når et framework skal håndtere legacy komponenter med deres egne biblioteker og kald. Der kan her opstå konflikter med forskellige kald, der har den samme syntaks i henholdsvis frameworket og biblioteket til den udefrakommende komponent. Løsningen til et sådant problem er oftest enten et redesign af frameworket for at fjerne konflikterne, eller skabelsen af en adapter, der virker som mellem-mand og håndterer konflikterne. En adapter kan skabe interferens i form af forsinkelser imellem komponenter, men dette er oftest mere hensigtsmæssigt i tidsforbrug end et redesign af et framework, især når der arbejdes med mange legacy komponenter.

Framework Gap problemer

Framework gap problemer opstår, når der skal bruges to eller flere frameworks til et program, men disse ikke helt dækker det nødvendige. Disse problemer opstår oftest, når de brugte frameworks ikke dækker deres domæner godt nok, men der kan ligge mange andre grunde bag. Mulige løsninger hertil er *wrapping*, *mediating software* og *redesign and extend*. *Wrapping* involverer et wrapper omkring et framework, hvori der udvides med yderligere kode, så det fulde domæne dækkes. *Mediating Software* kan bruges til at kommunikere imellem de brugte frameworks, og kan samtidig give den manglende funktionalitet. Dette kan dog blive dyrt at vedligeholde, når de brugte frameworks udvides. *Redesign and extend* kan bruges, når man har adgang til kildekoden til et givent frame-

work. Her kan man så udvide frameworket med den ønskede funktionalitet, og bruge dette nye framework i stedet.

Composition Overlap problemer
Composition overlap problemer opstår, når der benyttes to frameworks, der hver især repræsenterer den samme entitet fra hvert deres perspektiv. Her sker der en overlapning, der ofte er meget kompleks, når et frameworks bearbejdning af en entitet medfører ændringer, der skal tages højde for i det andet framework og så fremdeles. Igen er der tre mulige løsninger: *multiple inheritance*, *aggregation* samt *subclassing and aggregation*.

Multiple inheritance virker som en ligefrem løsning, men kræver at egenskaberne af hver klasse er disjunkte og derfor ikke påvirker hinanden. Multiple inheritance betyder at en klasse kan nedarve fra mere end en overklasse. Der findes dog programmeringssprog hvor dette ikke er tilladt, blandt andet Java og C#.

Aggregation er et alternativ, hvor aggregate klasser skabes, hvor hvert framework har dele det repræsenterer. Dette kræver dog kildekoden til frameworks, idet disse skal ændres, for at være i stand til at arbejde med de nye aggregat klasser. Aggregation er en association, der repræsenterer forhold, hvor en klasse er en del af en helhed. En aggregation kan forekomme, når en klasse er en samling eller en beholder til andre klasser, men hvor de andre klasser ikke er afhængige af beholderen for at overleve. Med andre ord, i en aggregation, hvis beholderen er ødelagt, eksisterer de andre klasser, der var indeholdt i beholderen, stadigvæk.

Subclassing and aggregation kan benyttes hvis man ikke har kildekoden til sine frameworks. Her laver man underklasser af hver klasse repræsenteret af et framework og aggregerer disse underklasser. Dette svarer lidt til at lave et mellemled imellem aggregate klasser og frameworks.

Framework Evolution and Maintenance fasen
Denne fase vil have de samme problemer som i ovenstående fase, idet senere udvidelser vil have det samme grundlag at arbejde på. Den store forskel er her, at man skal være opmærksom på, at ændringer i et framework vil have vidtrækkende konsekvenser, hvis de programmer, der benytter det, har ændret i deres framework, eller hvis benyttede klasser ændres radikalt, eksempelvis hvis kald ændres, flyttes eller fjernes. Et eksempel på et område med disse problemer er domæne specifikke frameworks [Michael Mattson, 1998].

8.2.3 Domæne specifikke frameworks

Domæne specifikke frameworks er frameworks, der er udviklet til et specifikt domæne, for eksempel det framework der er udviklet til planlægning af transmissioner for en TV station, som er beskrevet i [Wim Codenie and Vercammen, 1997]. Når der skal arbejdes med domæne specifikke frameworks, er der en række nye problemer, der er vigtige at håndtere. Dette er især tilfældet, hvis et framework udviklet til et specifikt domæne forsøges overført til et andet domæne.

Problemerne her har udspring i, at uden tilstrækkelig dokumentation kan det være meget besværligt at gennemskue, hvilke konsekvenser ændringer i klasse biblioteket til et framework kan have.

Dette problem bliver yderligere kompliceret af den naturlige udvikling i et frameworks levetid. Når et system bliver udviklet efter et framework, vil der med

tiden ske ændringer i dette system, i takt med at brugerne af systemet får brug for nye funktioner. Problemet dukker op, når et framework skal overføres til et nye domæne, eller blot bruges til at udvikle et nyt system for en anden kunde men indenfor samme domæne. Dette beskrives i [Wim Codenie and Vercammen, 1997] igennem en gruppe udvikleres erfaringer. Her blev udviklet et system til planlægning af transmissioner for en TV-station. I dette systems levetid blev der lavet forskellige ændringer, i takt med at kundens behov ændrede sig. Herefter startede forfatterne af artiklen på at udvikle et system til en anden TV-station. Her viste det sig, at denne nye kundes behov krævede, at løsningen i høj grad skulle tilpasses, for at kunne opfylde kundens behov.

Dette betød, at de to systemers arkitekturer bevægede sig i forskellige retninger, i takt med de skiftende forhold og nødvendige ændringer i funktionaliteten. Dette skabte alvorlige vedligeholdelses problemer for det lille udviklingshold. For at kunne genbruge tidligere udviklingsarbejde i højere grad, var der behov for en mere systematisk tilgang, og udviklingsgruppen valgte derfor at begynde at arbejde med framework teknologi som en mulig løsning.

Her stødte udviklingsholdet imidlertid på problemer. En traditionel måde at definere frameworks på er, at arbejdet med frameworks primært består i at udfylde såkaldte "hot-spots"i et framework, for på den måde at tilpasse et framework til et givent system. Dette mener artiklens forfattere imidlertid er en illusion. Ifølge deres erfaringer er det kun et begrænset antal frameworks, der kan tilpasses blot ved at udfylde disse "hot-spots"for praktiske applikationer. I andre områder er arbejdet med at tilpasse et framework til et nyt område ganske enkelt for kompliceret, og i visse tilfælde er det endda nødvendigt at ændre fundamentale dele af frameworkets arkitektur. Dette skyldes, at i takt med at markedet for et framework ændrer sig, er det nødvendigt at ændre tilsvarende i et framework for at kunne følge med udviklingen inden for et givet marked. Fordi det ikke er realistisk at udvikle et framework, der kan tage højde for alle mulige ændringer, er et framework aldrig færdigt. I artiklen gives der tre situationer, hvor der er behov for videreudvikling af et framework:

- Når der er opnået ny indsigt i domænet, og domæne-specifikke koncepter skal integreres i frameworket.
- Når klasser er blevet så komplekse, at der er behov for at overveje et nyt design af framework for at afhjælpe dette. Dette kan dække over introduktion af nye abstrakte klasser eller design mønstre.
- Når der er opnået ny indsigt i designet, og dette har ledt til afdækning af problemer i frameworket

Problemerne kan især opstå, hvis der ikke er tilstrækkelig dokumentation for afhængigheder imellem implementeringer af forskellige metoder. Når et system er tilstrækkelig simpelt, kan det muligvis lade sig gøre at finde disse afhængigheder ved hjælp af kode gennemlæsning. Men i næsten alle praktiske eksempler ville kode gennemlæsning simpelthen ikke være en praktisk løsning, da det både tager for lang tid, og har for store muligheder for fejl.

En mulig løsning ville derfor være bedre dokumentation. Artiklen [Wim Codenie and Vercammen, 1997] foreslår "Reuse Contracts"som en mulig løsning. Disse kan være enten "Multiclass Reuse Contracts", eller "Single-Class Reuse Contracts". Artiklens fokus er lagt på "Single-Class" udgaven, der kan beskrives

som følger: For hver klasse giver en "Single-Class Reuse Contract" en liste over metode signaturerne for de metoder, der er relevante for design af frameworket, de associerede specialiserings klausuler, samt en kommentar for abstrakte metoder. Specialiseringsklausulerne anfører navnene på metoder, der kan kaldes fra klassen. Disse "Reuse Contracts" vil ifølge artiklens forfattere hjælpe på problemet, og skabe bedre muligheder for at undgå fejl og problemer, når domæne specifikke frameworks skal overføres til andre områder.

8.2.4 Framework-udvikling i projektet

I dette projekt bliver der ikke benyttet aktiviteter fra Framework Development eller Maintenance-fasen, idet der benyttes et allerede udviklet generelt framework. Der bliver til gengæld benyttet aktiviter fra Instantiation-fasen, hvor der bliver brugt *redesign og extend*, idet der bliver extendet på Controleum, så det passer til dette domæne, og dermed løser Framework gap problemer.

Legacy Component problemer bliver undgået i projektet, idet der ikke bliver brugt nogen legacy komponenter i dette proof of concept system. I et fuldt implementeret system ville der højest sandsynlig forekomme legacy komponenter. Composition overlap problemerne undgås da de to frameworks der bliver brugt i dette proof of concept ikke overlapper nogen steder.

I dette projekt bliver der benyttet et generelt framework, som bliver specialiseret som et domæne specifikt framework. Grundet dette kan man forudse nogle af de problemer som kunne ske for domæne specifikke frameworks. Det gør det relevant at overveje disse *single class reuse contracts*, og der vil derfor blive givet et eksempel herpå i design-afsnittet.

8.2.5 Pakkestruktur

Valg af pakkestruktur er meget vigtigt for systemet, da dette valg får indflydelse på især skalérbarhed og udvidelsesevne. Der er en række forskellige typer af frameworks, og et som er meget udbredt, er det såkaldte MVC eller Model-View-Control, som især egner sig til objektorienteret softwareudvikling [Maciaszek, 2004]. En videreudvikling på dette er PCMEF, som ud over at have samme styrker som MVC, også har en række regler og principper, som skal hjælpe til med at mindske afhængigheder og gøre det nemmere at vedligeholde.

PCMEF er en forkortelse for de forskellige pakker i frameworket, nemlig Presentation, Control, Mediator, Entity og Foundation [Maciaszek, 2004]. Dette er en lagdelt arkitektur, hvor der øverst er Presentation laget, under det ligger Control laget, under det ligger Domæne-laget som indeholder Mediator og Entity pakkerne, og nederst ligger Foundation laget.

Presentation håndterer al interaktion med brugeren eller brugerne af systemet. *Control* er mellemled mellem Presentation og Domæne laget, så det håndterer forespørgsler fra Presentation. Da dette lag er ansvarlig for brugernes interaktioner med det underliggende system, da har man ofte en klasse her for hvert brugsmønster.

I Domæne laget ligger *Entity* og *Mediator*. *Entity* håndterer forespørgsler fra Control-laget og indeholder klasser som repræsenterer domænespecifikke objekter. *Mediator* sørger for kommunikation mellem Entity og Foundation, således at al kontakt med det understående lag foregår igennem denne pakke.

Foundation er ansvarlig for al kommunikation med databasen.

Ud over disse pakker er der også nogle principper som hjælper til med at øge systemets skalerbarhed og udvidelsesemne [Maciaszek, 2004]:

- **Downward Dependency Principle** sørger for, at øvre lag kun må afhænge af nedre lag, på denne måde kan et øvre lag skiftes eller ændres, uden at det har indflydelse på de nedre lag.
- **Upward Notification Principle** sørger for en lav kobling fra nedre til øvre lag, nedre lag må kun sende notifikationer til øvre lag.
- **Neighbor Communication Principle** sørger for, at et lag kun kan kommunikere med et nabolag lige under eller over det selv, hvilket mindsker koblinger. Eksempelvis betyder det, at Presentation laget aldrig må kommunikere direkte med Domæne laget.
- **Cycle Elimination Principle** eliminerer cykliske afhængigheder ved at man i stedet kunne indsætte en ekstra pakke og et interface.
- **Class Naming Principle** giver mulighed for altid at vide hvor en given klasse er fra, dette gøres ved at tilføje P, C, M, E eller F som præfix, alt efter hvilken en af frameworkets pakker den er indeholdt i.
- **Acquaintance Package Principle** er fra PCMEF+ , hvor + er for Acquaintance pakken, der bliver tilføjet. Dette er en pakke, som er adskilt fra de andre og kun indeholder interfaces. Denne pakke understøtter objektkommunikation mellem ikke-nabo-lag, og bruges til at opretholde en lav kobling under kommunikation mellem "bekendte" (acquaintances).

8.3 Genetiske Algoritmer

8.3.1 Definition

En genetisk algoritme er en algoritme, der bygger på biologiske principper om arvelighed, diversitet, populationer og fitness. Fitness i denne sammenhæng angiver, hvor god en specifik løsning er i forhold til en række parametre. En genetisk algoritme tager udgangspunkt i en tilfældigt skabt population, hvor den måler hvert individs fitness til bestemmelse for forældre til den næste generation. Genetiske algoritmer bruges ofte til parallelle vurderinger af forskellige teser eller ideer, og bruges også til adaptiv programmering [Mitchell, 1996].

8.3.2 Historie

Evolutionære systemer blev først undersøgt af flere uafhængige hold forskere i 1950'erne og 1960'erne [Mitchell, 1996]. I 60'erne introducerede Rechenberg "Evolutions Strategier", der oprindeligt blev brugt til at optimere vingespænd. Bl.a. Fogel, Owens og Walsh (1966) videreudviklede denne strategi i "Evolutionær programmering" med finite-state maskiner, der blev genereret udfra deres state-transition diagrammer. Andre hold, blandt andet biologer, brugte dem til at simulere forskellige kontrollerede populationer. Genetiske algorimer blev opfundet af Holland i 1960'erne og udviklet af hans team og studerende op igennem

1970'erne. Oprindeligt var det for at studere evolution fremfor at løse et specifikt problem. Hans bog "Adaptation in Natural and Artificial Systems" giver fingerpeg til, hvordan man kan behandle data som kromosomer og overføre fra generation til generation, se på "survival of the fittest" samt skabe permutationer og mutationer. Mens disse studier har lagt basis for udviklingen af genetiske algoritmer, så er konceptet sidenhen blevet udvandet og diversificeret til en sådan grad, at når der i dag tales om genetiske algoritmer, er det ikke altid lige nemt at se sammenhængen med den oprindelige research.

8.3.3 Udførsel

Som udgangspunkt skabes der en population af tilfældigt genererede løsningsmuligheder. De ubrugelige individer frasorteres, og resten bruges til næste generation. Der bruges forskellige teknikker kendt fra biologiens verden:

- Selection
- Crossover
- Mutation
- Gemte individer
- Nye individer

Individer med en fitness på nul fjernes eller ses bort fra i en generation. Der skabes nogle gange en mindre mængde nye individer til næste generation for at sikre en divers gen pøl. Yderligere gemmes en mindre del af den originale population for at sikre, at potentielle løsninger ikke glemmes. Her gemmes kun de bedste individer. Nogle algoritmer videresender ikke disse individer, men har derimod et lager, hvori de bedste individer lagres [Coello, 2006]. Herefter skal der skabes en ny generation udfra den nuværende population. Denne skabes ved selection og efterfulgt af crossover og mutationer. Selection er udvælgelse af individer, som udføres på basis af deres fitness funktioner. Jo større fitness, des større sandsynlighed for, at et individs kromosomer indgår i skabelsen af et nyt individ til næste generation. På basis af disse udtages der to individer af gangen, og crossover påbegyndes til skabelse af næste generation. Crossover er, hvor hele alleler byttes imellem mulige løsninger hvorved en krydsning skabes. Et eksempel kunne være strengene **ABCD** og **WXYZ** som forældre, hvor crossover skaber børnene **ABYZ** og **WXCD**. Multi-point crossover algoritmer kan emulere mere komplekse biologiske DNA processer, hvor crossoveren enten kan ske for hver individuel allele, eller hvor der blot kan krydses flere gange. Eksempelvis med strengene **ABCDEFG** og **TUVWXYZ** som forældre, hvor crossover skaber børnene **ABVWEFZ** og **TUCDXYG**. Crossover sker tilfældigt, og nogle gange kan de nye individer ende med at være kloner af deres forældre. Mutationer opstår ved at flippe tilfældige bits i løsningers alleler, dette svarer til dele af et genom i biologiens verden, ofte individuelle tal i en genetisk algoritme. Disse har en meget lav sandsynlighed for at opstå. Afhængig af design sker disse enten før, under eller efter crossover. Efter skabelse af en ny generation vurderes de resulterende individer på ny, og processen gentages i et fastsat antal gange eller indtil de bedste løsninger ikke længere forbedres i et antal generationer.

8.3.4 Fitness

En genetisk algoritme vurderer permutationer af sin population baseret på deres fitness værdi. Oftest bruges en funktion, hvis resultat er en værdi i intervallet fra 0 til 1, hvor 0 er ubrugelig og 1 er optimal. I et simpelt optimeringsproblem som eksempelvis temperatur for dyrkning af solsikker vil der gå en kurve hvor 0 grader er sub-optimal og langsomt stiger fitnessværdien indtil den når et optimum og falder så igen til 0 ved en så høj temperatur, at planten vil dø. Dette er dog langtfra et godt billede af genetiske algoritmer, idet de oftest bruges til løsning af avancerede problemer, hvori en større mængde faktorer spiller ind. Derfor beregnes fitness værdien her ud fra et, ofte vægtet, gennemsnit ud fra samtlige faktorer, hvor der også gives veto mulighed til visse faktorer. I solsikke eksemplet ville en temperatur på under 0 grader, eller temperatur over et dødeligt niveau, have veto. Derfor ville en fitnessfunktion ende med at se ud som [Mitchell, 1996]:

$$\frac{veto_1 \cdot ... \cdot veto_y \cdot (fitness_1 + ... + fitness_X)}{X}$$

8.3.5 Genetiske algoritmer i projektet

I dette projekt bliver der benyttet Controleum, som gør brug af genetiske algoritmer som beskrevet ovenfor. Den genetiske algoritme i frameworket bruger metoderne *selection*, *crossover* og *mutation*. Selection bruges til at udvælge de bedste individer fra hver generation, crossover og mutation bliver så brugt til at generere en ny generation ud fra disse. Der er dog en væsentlig forskel fra genetiske algoritmer som beskrevet ovenfor og det brugte framework, idet fitness værdier i det brugte framework betragter fitness værdier på 0 som optimale, og fitness værdier på 1 som ubrugelige. Controleum uddybes senere i rapporten.

8.4 Multi-Objektiv Optimering

8.4.1 Definition

Et system der har mere end et objektiv at tage højde for, kan optimeres med multi-objektiv optimering. Dette er relevant, da de fleste praktiske eksempler på systemer er for komplicerede til at kunne formulere blot et enkelt objektiv for et sådant system. Problemet består i at opstille et system, der kan optimere på alle objektiver, et problem der bliver yderligere kompliceret af, at mange objektiver ofte er i direkte konflikt med hinanden. Et eksempel på et multi-objektiv optimerings system kunne være et system, der skal kontrollere et spildevands system, som beskrevet i [M. Schütze, 2002]. Ifølge denne artikel kunne det primære objektiv være at skaffe sig af med spildevand på en hygiejnisk måde, og yderligere objektiver kunne være at undgå oversvømmelser, minimere skadelige effekter på miljøet, samt at minimere økonomiske udgifter.

8.4.2 Fremgangsmåde

Artiklen [M. Schütze, 2002] beskriver følgende fremgangsmåde: Hvis det antages at kontrol algoritmen for et sådant system kan beskrives fuldstændigt af en række if-then regler og/eller af en række controllers, som kendt fra "Control Theory", kan kontrol algoritmen beskrives af en række endelige numeriske

parametre. Derfor består arbejdet i to dele: For det første skal der opsættes et framework for de brugte controllers, og der skal bruges en række kontrol regler. Der skal for eksempel specificeres hvilke sensor-informationer, de forskellige kontrol input skal styres afhængigt af. Dernæst skal der sættes numeriske værdier på parametrene til disse controllers.

Når der er sat numeriske parametre på disse controllers, fører dette videre til en matematisk måde at se på et multiobjektiv beslutnings problem. Det følgende er baseret på en artikel, der fokuserer på multi-objektiv beslutnings problemer, der skal bruges indenfor styring af robotter, men principperne kan overføres til andre områder [Pirjanian, 2000]. Ifølge denne artikel kan et sådant multiobjektiv beslutnings problem repræsenteres af følgende ligning:

$$\hat{x} = \arg\max_{\mathbf{x}} [o_1(\mathbf{x}), o_2(\mathbf{x}), ..., o_n(\mathbf{x})] \tag{1}$$

Her er $\mathbf{x} \in X$, hvor $\mathbf{x} = (x_1, ..., x_N) \in \mathbb{R}^N$ er en N-dimensional beslutnings variabel vektor eller et alternativ. Systemets objektiver er så repræsenteret af n objektiv funktioner, $o_1, ..., o_n$. Hvor godt et givet alternativ $\mathbf{x}$ er opfyldt, i forhold til det k'ende objektiv er givet af $o_k(\mathbf{x})$.

I litteraturen er dette problem ofte kendt som "the Vector Optimization Problem"(VOP). Da de forskellige objektiver, som tidligere nævnt, ofte er i konflikt med hinanden, er det muligt at optimale løsninger ikke findes, og der kan derfor kun håbes på at finde løsninger, der tilfredsstiller hver af objektiverne i en tilstrækkelig grad. Denne idé leder til følgende definition fra [Pirjanian, 2000]:

> *Pareto-optimal løsning*
> $\mathbf{x}^*$ er en Pareto-optimal løsning til et VOP hviss der ikke eksisterer et $\mathbf{x} \in X$ sådan at $o_i(\mathbf{x}^*) \leq o_i(\mathbf{x})$ for alle i og $o_j(\mathbf{x}^*) < o_j(\mathbf{x})$ for mindst en j.

En måde at beskrive denne definition i ord er, at en løsning er Pareto-optimal hvis og kun hvis ingen anden løsning kan forbedre en objektiv funktion uden samtidig at forringe mindst en af de andre objektiv funktioner.

En måde at finde en løsning på, er ved at generere et sæt pareto-optimale løsninger, og herefter vægte disse baseret på hvad der er vigtigt i konteksten af optimeringsproblemet. Baseret på disse vægte udvælges den bedste løsning.

8.4.3 Multi-objektiv optimering i projektet

I dette projekt bruges et framework, der udnytter multi-objektiv optimering, med udgangspunkt i det ovenfor beskrevne. Matematikken bag er vigtig at kende til, da det giver en god forståelse for hvordan optimeringen foregår i frameworket. De controllers der omtales i afsnittet kan oversættes til optimeringsframeworkets concerns, som vil blive forklaret i et senere afsnit.

Den store forskel på det benyttede framework og den ovenfor beskrevne metode ligger i måden hvor på de pareto-optimale løsninger findes. Frameworket benytter genetiske algoritmer til skabelsen af en divers løsningsmængde, til modsætning af det ovenstående, hvor der benyttes en mere ligefrem matematisk tilgang, som har nogle svagheder.

8.5 Multiagent Systemer

8.5.1 Introduktion

I løbet af de sidste årtier har der været en enorm forøgelse af, hvad en computer kan præstere i form af hastighed og kompleksitet. Dette hænger sammen med, at forventningerne til hvad en computer skal kunne, samtidigt er forøget. I starten forventedes det blot, at en computer skulle kommunikere direkte med en person ud fra simple input givet af brugeren. Nu forventes der, at et system skal kunne kontrollere eller samarbejde med andre systemer, eller træffe beslutninger på brugerens vegne. Denne ændring i forventninger giver anledning til at give systemer evnen til at kunne arbejde uafhængigt af menneskelig indblanding. Desuden forventes det, at den skal kunne handle på en sådan måde, at den kan arbejde hen imod brugerens bedste ønske, når den samarbejder med andre mennesker eller systemer.

Tilsammen giver alle disse forventninger anledning til at introducere "multi-agent systemer". Idéen bag et multiagent system er simpel, en agent er et system som kan handle uafhængigt af dens bruger eller ejer. Et "multiagent system" er da et system indeholdende flere af sådanne agenter, hvor de interagerer med hinanden i form af forhandlinger, samarbejde eller koordination [Wooldridge, 2009].

8.5.2 Intelligente agenter

For at kunne definere en agent som værende intelligent, kræves der nogle retningslinjer for dette. I [Wooldridge, 2009] foreslås tre punkter: reaktionsevne, proaktivitet og social færdighed. Reaktionsevne er, når intelligente agenter kan forstå deres miljø og reagere i tide i forbindelse med ændringer der sker, for at agenten kan opnå deres design mål. Med proaktivitet hentydes til evnen til at udvise en målrettet opførsel ved at tage initiativet til at tilfredsstille deres design mål. Social færdighed er evnen til at interagere med andre agenter for at opnå deres mål.

Der bør ikke antages et statisk miljø, en agent skal kunne forstå ændringer, der vedrører dens målsætninger, som sker i miljøet, og handle ud fra dette. Det er også vigtigt, at en intelligent agent ikke kun er et reaktivt system, det ønskes at agenten hele tiden skal forsøge at bevæge sig hen i mod dets ønskede mål. Det, der ønskes, er en balance imellem reaktionsevne og målrettethed, således at agenten systematisk forsøger at opnå dens mål, men ikke blindt udfører procedurer for at opnå dem, selv om det er klart, at procedurerene ikke vil virke. Derudover ønskes der også en social adfærd som tillader at forhandlinger omkring ændringer, der skal ske i systemet, kan foregå. Hermed menes det, at der kan opstå situationer, hvor den kollektivt bedste løsning er et kompromis mellem flere agenter. For at opnå den bedste løsning skal agenterne kunne samarbejde og forhandle for at komme frem til dette kompromis.

8.5.3 Agent tilfredshed

For at finde ud af, hvor tilfreds en agent er med en given løsning, er der brug for en måde at beregne dette på. Dette gøres ved hjælp af en såkaldt "utility function", som måler lige præcis dette. En utility function er unik for hver

agent, da metoden skal give en beskrivelse af, hvor tæt på agentens mål dette i virkeligheden er. Idéen er her at give mulighed for, en agent kan være tilfreds nok til, at en løsning er acceptabel for den i denne omgang. Det er vigtigt at pointere, at denne metode ikke kun skal være for et øjebliksbillede, men også gerne tage højde for tidligere løsninger, så hvis den kun har været lidt tilfreds igennem noget tid, og dets målbeskrivelse fortæller at den skal opnå et vist niveau over tid, så vil den arbejde hen imod dette mål ved at mindske sin tilfredshed med den nuværende situation.

8.5.4 Hvornår bør man bruge en agent-baseret løsning

Der er nogle relevante faktorer, som gør sig gældende, når man vælger et agent-baseret system. Den første af disse er, når agenterne er en naturlig metafor [Wooldridge, 2009]. Med dette menes der, at der i mange miljøer naturligt kan modelleres agenter, som repræsenterer et samfund, hvor de enten kan arbejde sammen eller imod hinanden, for at opnå deres mål. Den anden er, når der skal være en distribution af data, kontrol eller ekspertise [Wooldridge, 2009]. Med dette menes der, at en centraliseret løsning er meget svær at få optimal. I sådanne tilfælde er det oftest lettest også her at lade agenter repræsentere naturlige elementer, som har nogle mål de skal opnå, og så samarbejde eller forhandle med hinanden, for at opnå det bedst mulige resultat.

8.5.5 Agenter i projektet

Optimerings frameworket er et multi-agent framework, dog overholder dets agenter ikke definitionen på intelligente agenter. De er ikke reaktive eller proaktive, fordi deres mulighed for at agere er fjernet fra den enkelte agent, og flyttet over i den centrale Negotiator, som styrer processen. Tilsvarende har de ikke sociale færdigheder, fordi de ikke selv kan kommunikere med andre agenter, men må lade Negotiator stå for denne kommunikation.

Agenterne i projektet beskæftiger sig med hver sin del af dambruget, hvis optimering de er ansvarlige for. Dette skal forstås således, at hver agents *utility function* er præcis det samme som *fitness function* fra de genetiske algoritmer. Den præcise virkemåde for optimeringsframeworket beskrives i et senere afsnit.

9 Analyse og Design

9.1 Framework for Multioptimering

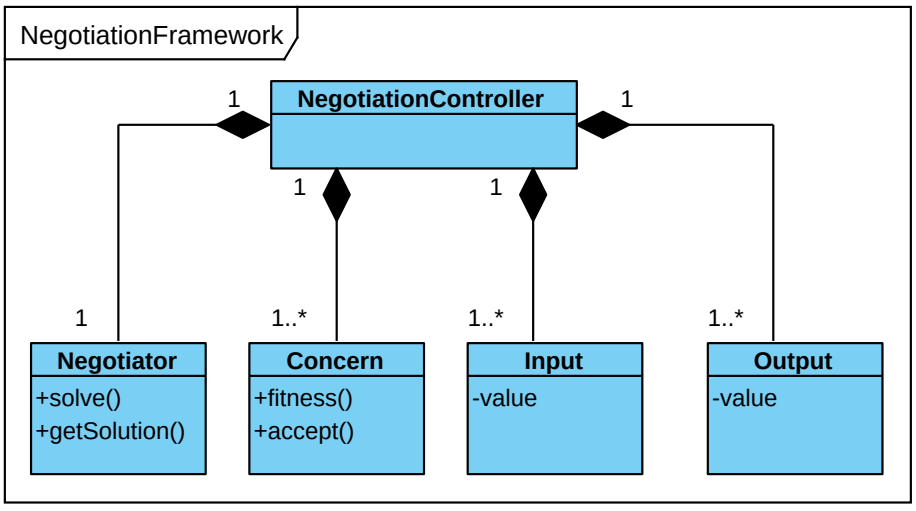
Controleum-frameworket vil her blive forklaret med udgangspunkt i de foregående afsnit omkring software agenter, genetiske algoritmer og multiobjektiv optimering. Frameworket består af 5 større komponenter: Inputs, Outputs, Concerns, Negotiatior og en klasse som samler alle delene. Den sidstnævnte klasse skal derudover indeholde et objekt, som bliver en holder for alle de andre objekter. Dette bliver gjort ved hjælp af dynamisk linking af objekter, se [Martin Rytter, 2010].

- Inputs, objekter der extender typen *Input*, repræsenterer alt det, som ikke er åbent for forhandling (negotiation), dette kunne for eksempel være målinger af temperatur, pH-værdi eller lignende.

- Outputs, objekter der extender typen *Output*, repræsenterer alle de ting, som godt kan forhandles, for eksempel hvor meget kraft en pumpe skal køre med, hvor meget der skal justeres på pH-værdi, eller om fiskene skal fodres på et givent tidspunkt.
- Concerns, objekter der extender typen *Concern*, repræsenterer agenter, der skal forhandle på vegne af systemet. Et mål for et sådan concern kunne være sikring af lav elektricitetspris, at pumperne kører inden for deres begrænsninger, eller at der er den rigtige pH-værdi i vandet. Der bliver skelnet mellem soft concern og hard concern; et soft concern er et concern, som kun har en fitness funktion, et hard concern, er et concern, som kun har en accept funktion. Fitness er et udtryk fra genetiske algoritmer, som beskriver hvor tilfreds en agent er for et givent resultat. I Controleum giver fitness funktioner en værdi fra 0 til 1, hvor 0 er optimalt, og 1 er det dårligst mulige. Acccept er mere hårdt end fitness, og fortæller om en løsning kan bruges eller ej.
- Det er opgaven for en *Negotiator* at finde frem til den bedste løsning for alle agenter. Dette gøres ved, at negotiatoren genererer værdier for output. Her udnyttes teknikkerne fra multiobjektiv optimering, som bruger genetiske algoritmer til at danne løsningerne. Efter hver generation er dannet bliver den dårligste halvdel frasorteret, og der dannes nye generationer ud fra de overlevende. Dette gøres så op til 1.000 gange, eller indtil der ikke længere bliver forbedret på fitness eller accept for de indblandede agenter (concerns). Når et tilfredsstillende resultat er blevet fundet, kan en løsning af typen *Solution* hentes fra den.

Dette leder frem til en domæne model for Negotiator frameworket i dambrug.

Model over frameworket



Figur 2: Model over multioptimeringsframeworket, stor udgave kan ses på Figur 21, s. 66.

9.2 Analyse og Design arbejde i Unified Process

I dette afsnit vil der blive beskrevet, hvordan der er blevet arbejdet med redskaberne indenfor Unified Process [Jim Arlow, 2008] til at udarbejde aktørbeskrivelser, brugsmønster diagrammer, detaljerede brugsmønsterbeskrivelser, navne- og udsagnsordsanalyse, sekvensdiagrammer, analyseklassediagrammer og designklassediagram. Resultaterne af dette arbejde med Unified Process, kan bruges til at basere implementeringen af Proof-of-Concept systemet på.

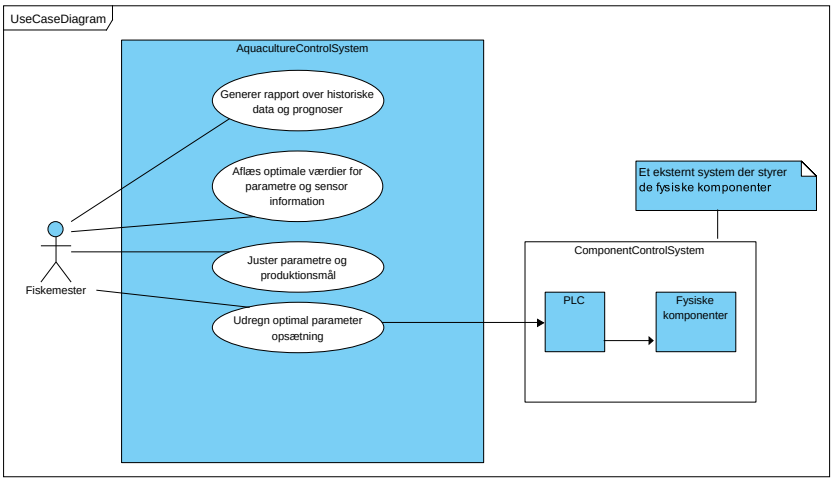
9.2.1 Aktørbeskrivelser

I dette system er der en aktør som bruger systemet; Fiskemesteren. Herunder er en forklaring på rollen i systemet:

Fiskemester	Personen der styrer dambruget, og holder øje med systemet og begynder optimeringsprocessen.

9.2.2 Brugsmønstre

Her vises brugsmønsterdiagrammet, og herefter vil de forskellige brugsmønstre blive forklaret ud fra hver aktør:



Figur 3: Brugsmønsterdiagram, stor udgave kan ses på Figur 22, s. 67.

Fiskemester

- *Få rapport over historiske data og prognoser*
  Der skal gives mulighed for at aflæse historiske data som en rapport med grafer, samt prognoser for fremtiden. Optimeringsprogrammet drager sine konklusioner på baggrund af en række informationer. Disse præsenteres her i rapport-form, sammen med prognoser for fremtidig drift. Disse prognoser kan også bruges til at forudse problemer så som sygdomme eller iltsvind. Prognoser for fremtiden indebærer også et estimat om, hvornår fiskene vil være færdige.

- *Juster parametre og produktionsmål*
  Brugeren skal kunne ændre i produktionsmålene, såsom billigst produktion, deadline for færdig produktion eller lavere udledning af specifikke stoffer. Der foruden skal der være mulighed for at ændre i fodertyper, ønskede pH-værdier, ønsket iltningsprocent og foder tider. Ændringer i produktionsmålene og fastlåsning af parametre forbliver kun et brugsmønster i dette projekt, men ville i et fuldt system kunne modelleres som ændringer i optimeringsframeworkets indstillinger.

- *Aflæs optimale værdier for parametre og sensor information*
  Systemet skal hente informationer fra sensorerne for eksempelvis pH-værdi, iltningsprocent, temperatur eller CO2. Systemet skal samtidigt kunne indhente de sidst udregnede optimale værdier for parametre og vise det hele for fiskemesteren.

- *Udregn den optimale parameter opsætning*
  Systemet skal udregne, på baggrund af input fra sensorer og indstillinger fra fiskemester, den optimale opsætning af parametre. Denne udregning skal ske kontinuerligt, efter fiskemesteren har startet processen en gang. Dette gøres ved hjælp af Negotiator-delen af frameworket. Delen med at inddrage indstillinger fra fiskemester vil ikke blive forfulgt længere end til brugsmønstret, i et fuldt system kunne det modelleres som nævnt under "Juster parametre og produktionsmål".

9.2.3 Detaljeret brugsmønsterbeskrivelse

Der er nu blevet afdækket alle brugsmønstre laves der detaljerede brugsmønsterbeskrivelser for hver af disse. I brugsmønsteret for at udregne den optimale parameter opsætning, er et af punkterne i hovedforløbet sat i parentes. Dette skyldes, at der her er lavet en afgrænsning, og at denne del ikke vil blive implementeret i Proof-of-Concept systemet.

Brugsmønster	Udregn den optimale parameter opsætning (CalculateOptimization)
ID	1
Beskrivelse Systemet skal kontinuerligt, med fastsatte mellemrum, udregne de optimale værdier for fysiske komponenter. Disse værdier udregnes på baggrund af input fra sensorer og indstillinger fra fiskemester, og gøres ved hjælp af Negotiator-delen af frameworket.	
Primære aktør	Fiskemester
Sekundær aktør	Ingen
Forudsætninger (Preconditions)	Der er gemt mindst et sæt sensor information.
Hovedforløb 1. Processen kører i et loop indtil det stoppes: (a) Systemet henter nyeste data fra databasen om sensorer og inputs. (b) Systemet genererer mulige løsningsforslag indtil en acceptabel løsning findes. (c) Denne løsning gemmes i databasen. (d) (Løsningen bliver skubbet ud til de fysiske komponenter.)	
Resultater (Postconditions)	En ny løsning af optimeringsproblemet er fundet.
Alternative forløb	Ingen

Brugsmønster	Få rapport over historiske data og prognoser (GenerateReport)
ID	2
Beskrivelse Der skal opbygges en rapport på baggrund af historiske data, i form af sensor information, optimerede værdier for fysiske komponenter, samt prognoser for vækst, sygdomme mm. Prognoser vil kun forblive et brugsmønster og ikke implementeret.	
Primære aktør	Fiskemester
Sekundær aktør	Ingen
Forudsætninger (Preconditions)	Der er gemt mindst et sæt sensor information, samt mindst en optimering.
Hovedforløb 1. Brugeren vælger et start- og sluttidspunkt for rapportdata. 2. Systemet henter sensor informationer, optimerede værdier fra databasen, og andre input. 3. Der opbygges en rapport på baggrund af de hentede data. 4. Rapporten bliver returneret til brugeren.	
Resultater (Postconditions)	Rapport med behandlede data, samt grafer. Dette vil kun blive implementeret til den grad hvor dataene hentes, ubehandlede, men vil være bygget til at kunne udvides enkelt med en sådan funktionalitet.
Alternative forløb	Ingen

Brugsmønster	Aflæs optimale værdier for parametre og sensor information (GetLatestStatus)
ID	3
Beskrivelse Systemet skal hente informationer fra sensorerne for pH-værdi, iltningsprocent, temperatur, CO2 osv. Systemet skal samtidigt kunne hente de sidst udregnede optimale værdier for parametre og vise det hele for fiskemesteren.	
Primære aktør	Fiskemester
Sekundær aktør	Ingen
Forudsætninger (Preconditions)	Der er gemt mindst et sæt sensor information.
Hovedforløb 1. Systemet henter sensor informationer. 2. Systemet henter optimale værdier for parametre. 3. Informatiorne bliver vist til fiskemesteren.	
Resultater (Postconditions)	Informationerne er blevet hentet til fiskemesteren i et samlet objekt.
Alternative forløb	Ingen

9.2.4 Navne- og udsagnsordsanalyse

For at identificere forskellige mulige analyseklasser er der lavet en navne- og udsagnsordsanalyse på afsnittet om Controleum og de detaljerede brugsmønsterbeskrivelser. Her fremkom følgende resultater:

Fra Negotiator frameworket

- *Navneord*
  Input, Output, Concern (agenter), Negotiator, NegotiationController, Solution
- *Udsagnsord*
  Solve, hent løsning

Fra brugsmønster 1, CalculateOptimization

- *Navneord*
  Systemet, input, sensorer, indstillinger, optimale værdier for fysiske komponenter, Negotiator, tidsinterval, optimeringsprocessen, databasen, sensorer, inputs, løsningsforslag, acceptabel løsning, fysiske komponenter
- *Udsagnsord*
  udregne, henter nyeste data, genererer, løsning findes, gemmes i databasen, skubbet ud

Fra brugsmønster 2, GenerateReport

- *Navneord*
  Rapport, historiske data, sensor information, optimerede værdier for fysiske komponenter, prognoser for vækst og sygdomme, start- og sluttidspunkt, rapportdata, databasen, input, grafer
- *Udsagnsord*
  Opbygges, vælger, henter, returneret

Fra brugsmønster 3, GetLatestStatus

- *Navneord*
  Optimale værdier for parametre, sensor information, informationer, sensorer (for pH-værdi, iltningsprocent, temperatur, CO2 osv.), optimale værdier for parametre
- *Udsagnsord*
  hente

Ud fra disse værdier kan der opbygges et simpelt analyse klasse diagram ved at indentificere hvilke af navneordene, der kan blive til klasser eller attributter, imens udsagnsord kan blive til operationer. Ordene vil nu blive gennemgået og sammenhængene forklaret:

Klasser

Systemet kaldes *Aquaculture Control System* og vil indeholde operationer for hvert brugsmønster. Alle navneordene fra Negotiator frameworket bliver lavet direkte om til klasser i dette system. Dette kan ikke undgås, da et framework benyttes til at finde optimeringerne. Derudover svarer alle *input*, som er nævnt, til input fra en sensor, hvilket også svarer til et input fra Negotiator frameworket.

Sensor informationer vil i dette system blive hentet fra databasen. *Sensor information* bliver lavet om til en klasse indeholdene værdier for en måling sammen med et tidsstempel for målingen, med et tidsstempel per sæt målinger. Denne klasse skal kunne hente den sidst gemte sensor information, eller et sæt af sensor informationer ud fra et givent tidsinterval.

Optimerede værdier for fysiske komponenter, acceptabel løsning, løsningsforslag, optimale værdier for parametre, disse bliver alle lavet om til en enkelt klasse kaldet *Solution* da de dækker over det samme: nye værdier til de fysiske komponenter som er acceptable løsningsforslag. Solution klassen skal kunne hente den sidst gemte løsning, og hente et sæt løsninger som ligger inden for et givent tidsinterval.

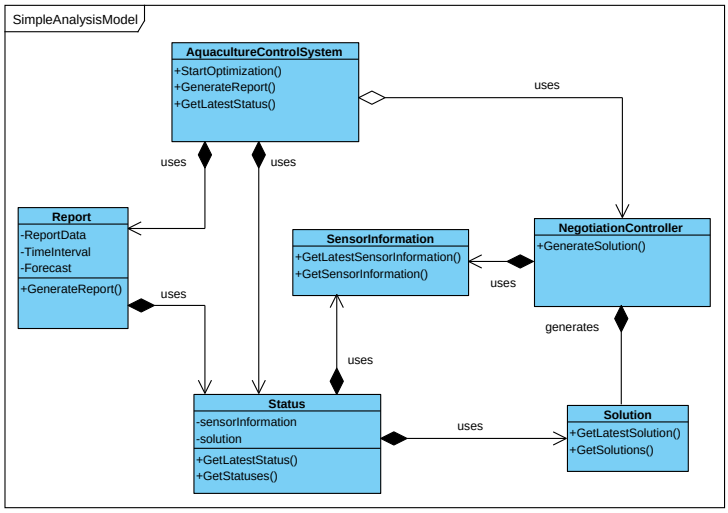
Rapport laves også til en klasse, denne klasse skal kunne hente historiske data, hvilket i dette tilfælde vil sige sæt af *Solution* og *Sensor Information* ud fra et givent tidsinterval. Denne samling af objekter skulle også kunne returneres til fiskemesteren, og derfor vil det samlede objekt blive kaldt *Status*, som så kan holde en liste med *Solution* og en liste med *Sensor Information*. Dette gøres for at kunne overholde både brugsmønster 2 og 3. I brugsmønster 2 vil *Status*-objektet indeholde lister med de førnævnte objekter inden for tidsintervallet, og i brugsmønster 3 vil objektet kun indeholde én af hver, da det er det nyeste sæt der ønskes.

Operationer

Solution får to operationer, så den kan hente den sidste nye løsning, og hente et sæt løsninger ud fra et tidsinterval. Tilsvarende gøres for *Sensor Information*. *Status* vil få to operationer, som vil kalde de tilsvarende operationer på *Solution* og *Sensor Information*. *Rapport* vil da have en operation, som genererer en rapport ved at hente information gennem *Status*. *NegotiationController* vil have en operation, som genererer en *Solution* ved at hente nyeste *Sensor Information*.

9.2.5 Analysemodel

Ud fra navne- og udsagnsordsanalysen kan der laves en overordnet analysemodel for systemets udformning. Der er valgt at oversætte udtrykkene til engelsk, da det letter oversættelsen fra model til kode. Som det kan ses ud fra modellen nedenfor, er alle forbindelser nævnt i analysen blevet beskrevet.

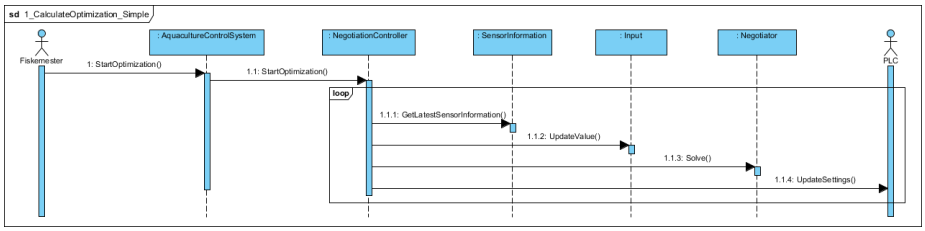


Figur 4: Analysemodel, stor udgave kan ses på Figur 23, s. 68.

9.2.6 Simple Sekvensdiagrammer

For at visualisere bevægelsen af data og operationer igennem programmet, er der lavet en række sekvensdiagrammer. Efter udviklingen af den simple analysemodel, skabes det første sæt af sekvensdiagrammer, der viser hver operations bevægelser igennem systemet. Disse simple sekvensdiagrammer skal give et overblik over hver operation afgivet af brugeren, og vise, hvordan operationer bevæger sig igennem den simple modellering af systemet. I et simpelt sekvensdiagram beskæftiger man sig ikke med persistens, så dette bliver ikke vist. Herunder ses de simple sekvensdiagrammer for metoderne, brugeren kan kalde i den simple analysemodel.

Brugsmønster 1: CalculateOptimization



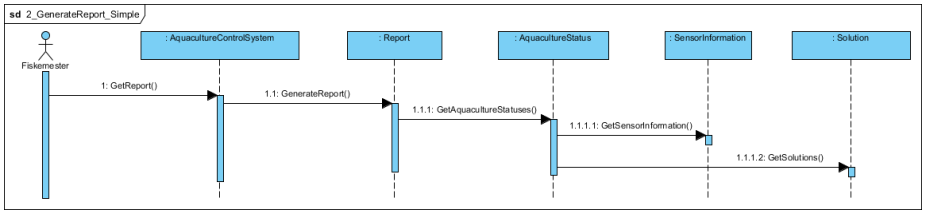
Figur 5: Simpelt sekvensdiagram for brugsmønster 1, stor udgave kan ses på Figur 24, s. 69.

Dette sekvensdiagram viser hvordan brugsmønstret CalculateOptimization vil blive opfyldt. Fiskemesteren starter denne sekvens ved at kalde *StartOptimization*() i systemet. Systemet kalder en metode af samme navn i en NegotiationController

klasse. Optimerings-delen kører i et loop hvor der først bliver hentet de nyeste sensor data fra Sensor Information klassen. Disse informationer gemmes i et objekt, som kædes sammen med Negotiator-delen, således at Negotiator-delen kan hente disse værdier og indsætte værdierne i de korrekte Input klasser. I modellen for Negotiator frameworket svarer dette til ContextClass klassen, som er et objekt der indeholder værdierne. På Negotiator objektet kaldes *Solve*() som løser optimeringen og genererer en løsning.

I et fuldt system ville denne løsning blive sendt ud til en PLC som så kunne reagere på den.

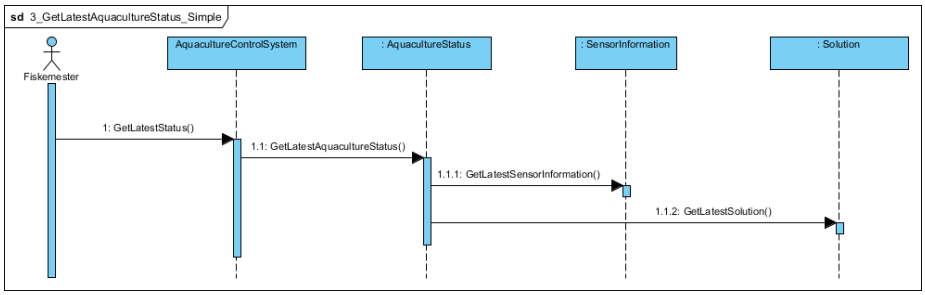
Brugsmønster 2: GenerateReport



Figur 6: Simpelt sekvensdiagram for brugsmønster 2, stor udgave kan ses på Figur 25, s. 70.

Fiskemesteren starter denne sekvens ved at kalde *GetReport*() i systemet. Systemet kalder klassen Report for at generere en rapport. Report kalder herefter AquacultureStatus for at samle de ønskede data. AquacultureStatus kalder endelig Sensor Information og Solution klasserne for at få de korrekte data ud fra et tidsinterval, som bliver givet med.

Brugsmønster 3: GetLatestStatus

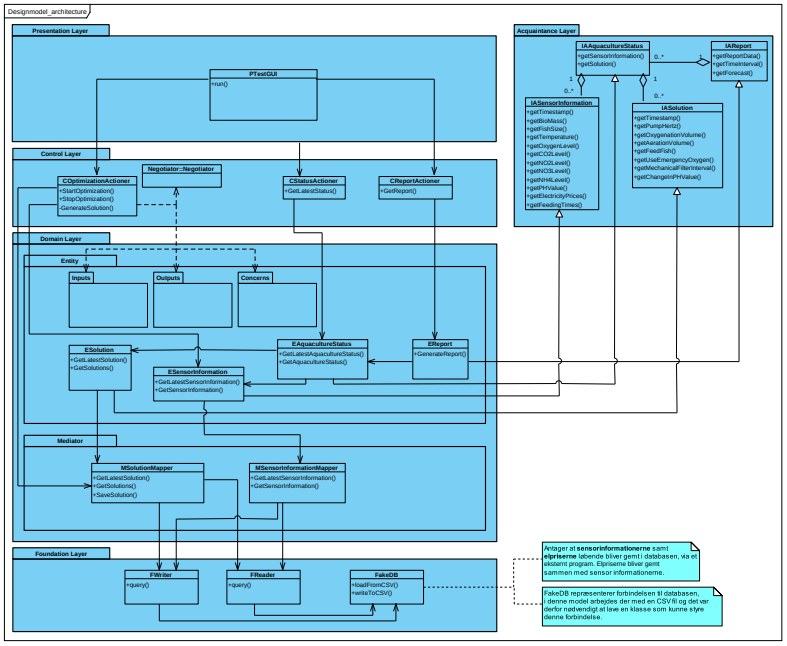


Figur 7: Simpelt sekvensdiagram for brugsmønster 3, stor udgave kan ses på Figur 26, s. 71.

Fiskemesteren starter denne sekvens ved at kalde *GetLatestStatus*() i systemet. Systemet kalder AquacultureStatus klassen for at samle de ønskede data. Denne klasse kalder herefter Sensor Information, og Solution på samme måde.

9.2.7 PCMEF+ og designklassemodel

Ud fra de foregående afsnit angående brugsmønstre, analyser og modeller, blev det fundet bedst passende at benytte PCMEF+. Der blev valgt denne struktur da klare afgrænsninger mellem programmets forskellige dele, gør det overskueligt at arbejde med, mindsker afhængigheder og gør det nemmere at udvide og vedligeholde. Dette er især vigtigt, da der bruges et udefrakommende framework under udvikling, der er designet til nemt, at kunne udvides med yderligere concerns. På denne måde bibeholdes en skalerbar og udvidelsesvenlig arkitektur. Følgende er designklassediagrammet for arkitekturen, som er blevet tilpasset PCMEF+ i forhold til analyseklassediagrammet. Dette diagram er valgt med den viste detaljegrad for at give et overblik over arkitekturen. Havde der været valgt en større detaljegrad på dette diagram, ville det have trukket opmærksomheden fra arkitekturen, hvilket er fokusområdet for dette diagram. Et mere detaljeret uddrag af diagrammet kan findes efter forklaringen af PCMEF+ designet.



Figur 8: Design diagram for arkitektur, stor udgave kan ses på Figur 27, s. 72.

Presentation-laget

- *PTestGUI* er den grafiske brugerflade, som benyttes til at illustrere, hvordan brugen af de forskellige dele af Control-laget ser ud, samt give en overskuelig måde at se hvordan systemet virker.

Control-laget

- *COptimizationActioner* har til opgave at løse brugsmønster 1: Udregn den optimale parameter opsætning. Dette gøres ved hjælp af Controleum, og

har derfor kun tilknytning til objekter i Entity-laget som har relevans for dette, sammen med Negotiator-klassen som bliver brugt direkte fra frameworket. Derudover har den også forbindelse til MSolutionMapper fra Mediator, som kan gemme fundne løsninger.

- *CReportActioner* har til opgave at løse brugsmønster 2: Få rapport over historiske data og prognoser. Disse informationer hentes igennem Entity-lagets EReport, som så benytter EAquacultureStatus, som så henter de relevante data fra ESolution og ESensorInformation.
- *CStatusActioner* har til opgave at løse brugsmønster 3: Aflæs optimale værdier for parametre og sensor information. Dette gøres på samme måde som CReportActioner for at få resultaterne samlet i et objekt.

Domæne-laget, Entity pakken

- *ESolution* repræsenterer en løsning fundet i Martin Rytters framework, og indeholder også operationerne til at hente løsningerne. Dette sker igennem MSolutionBroker.
- *ESensorInformation* repræsenterer et sæt af sensor informationer, også denne klasse indeholder operationer for at hente tidligere sæt, eller det seneste.
- *EAquacultureStatus* er et objekt, som samler sæt af løsninger og sensor informationer, for at have et samlet objekt der kan returneres til EReport og CStatusActioner, der begge skal bruge sæt af begge to.
- *EReport* repræsenterer et objekt indeholder informationer vedrørende tidligere løsninger og sensor informationer.
- *Inputs-, Outputs- og Concerns-pakke* repræsenterer de klasser der nedarver fra multioptimerings frameworket, disse pakker bliver udspecificeret senere.

Domæne-laget, Mediator pakken

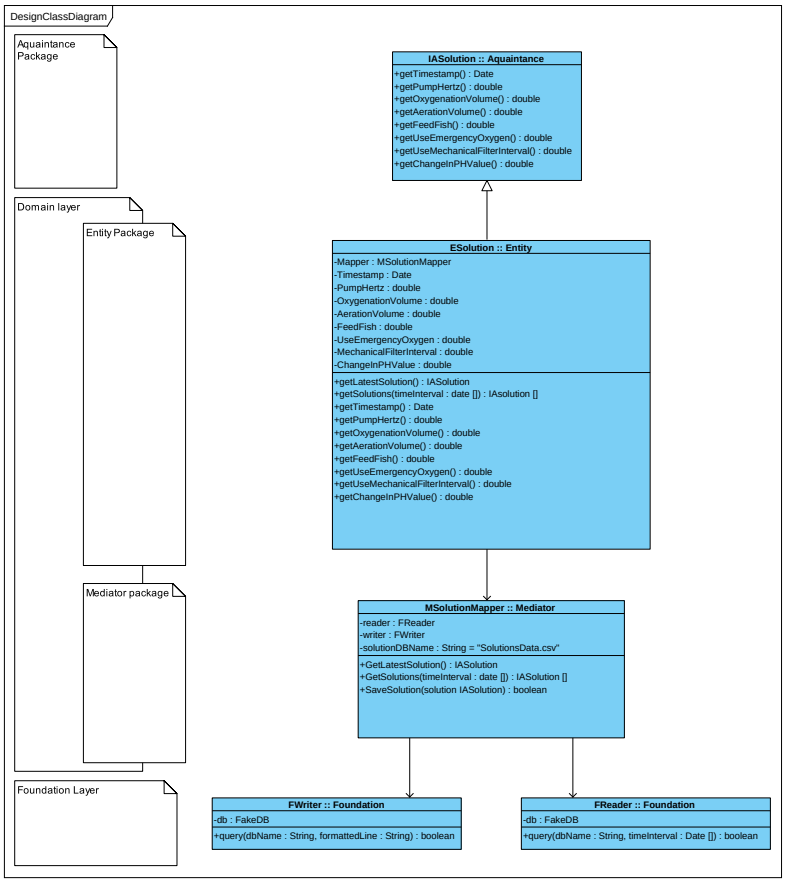
- *MSolutionMapper* håndterer forbindelsen mellem øvre lag og Foundation laget, når det er løsninger (Solutions), der skal gemmes eller hentes.
- *MSensorInformationMapper* håndterer på samme måde forbindelsen, når det er sensor informationer, der skal gemmes eller hentes.

Foundation-laget

- *FReader* sørger for forbindelse til database som vedrører læsning. I dette proof of concept system består det af at læse fra en CSV-fil, som indeholder linjer med tidligere løsninger eller sensor informationer. I dette proof of concept bliver elpriserne betragtet som en del af dem, så de er gemt sammen med sensor informationerne.
- *FWriter* sørger også for forbindelse til databasen, i dette tilfælde når der skal skrives til den førnævnte CSV-fil med tidligere løsninger.

Detaljeret udsnit af design model

Der er valgt fokus på solutions i systemet. Diagrammet viser en detaljeret fremstilling af ESolutions og udvalgte dele af systemet, som er relevante for ESolutions. Tilsvarende diagram kunne laves for andre dele af systemer, for eksempel kunne der laves et diagram der fulgte ESensorInformation igennem systemet, men forskellene er så små, at det ikke blev fundet relevant at lave.



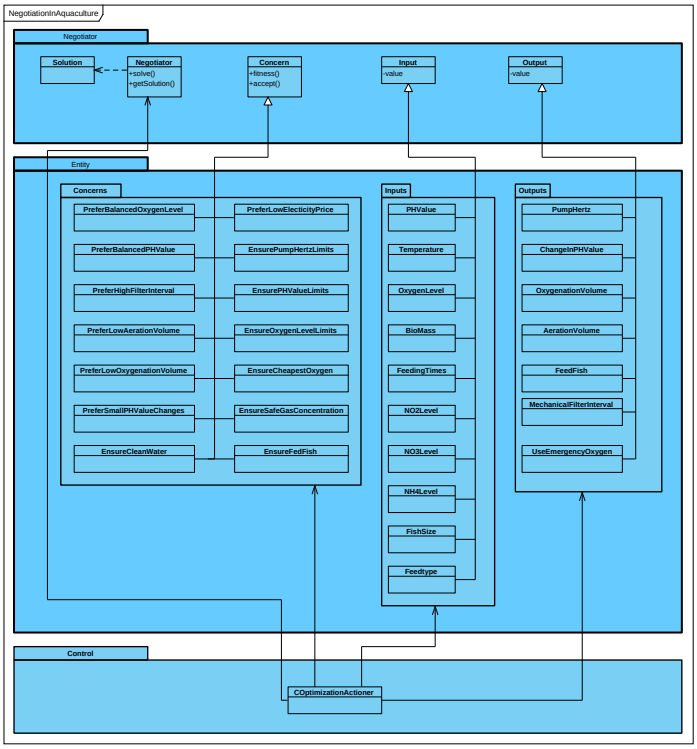
Figur 9: Detaljeret designklassediagram , stor udgave kan ses på Figur 28, s. 73.

Da der nu er udarbejdet en grund designklassemodel, kan der dannes avancerede sekvensdiagrammer for at komme et skridt nærmere implementering.

9.2.8 Multioptimeringsframework i dambrug

Der er gjort brug af frameworket til at lave nye klasser af Input, Output og Concerns, som arver fra disse. På følgende diagram kan man se sammenhængen, og herefter følger en beskrivelse af, hvordan de forskellige dele hænger sammen, forklaret ud fra de udarbejdede Concerns.

Som det fremgår af diagrammet, ligger klasser, der nedarver fra Input, Output, og Concerns i en under-pakke til Entity, dette gøres for at overholde PCMEF+ strukturen. Det er COptimizationActioner der benytter sig af disse klasser, og når forbindelsen går fra denne klasse og til hele pakken, er det fordi



Figur 10: Multioptimerng i dambrug, stor udgave kan ses på Figur 29, s. 74.

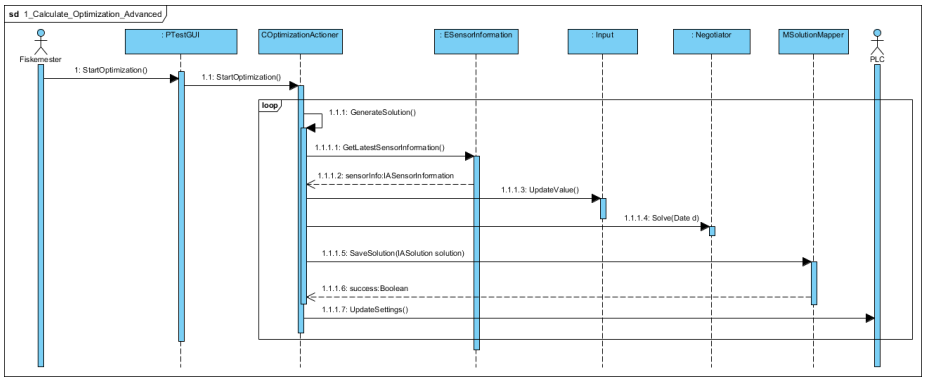
den bruger alle klasser heri. Derudover bruger den også Negotiator-klassen som den er, da denne klasse kun indeholder logikken for optimeringen og skal derfor ikke specialiseres til dambruget. De forskellige Input-, Output- og Concern-klasser vil blive forklaret i implementeringsafsnittet.

9.2.9 Avancerede Sekvensdiagrammer

På basis af den simple model og de simple sekvensdiagrammer, udarbejdedes der en designmodel, der går i dybden med objekter, klasser og undermetoder. På basis af dette udarbejdes der så avancerede sekvensdiagrammer. For de allerede modellerede metoder er det en uddybning af de simple sekvensdiagrammer, mens det for undermetoder inden i systemet består af helt nye diagrammer. Herunder ses først de udvidede sekvensdiagrammer for brugsmønstrene, herefter sekvensdiagrammer for metoder, der er specificeret i designmodellen, navnlig hvordan der hentes og gemmes data om Sensor Information og Solution objekter.

Brugsmønster 1: CalculateOptimization

Fiskemesteren benytter sig af en brugerflade, som ligger i Presentation-laget for PCMEF+ frameworket, kaldet PTestGUI, så det overholder navngivningsprincippet. Ud fra dette trykkes der på en knap, som initierer denne sekvens.



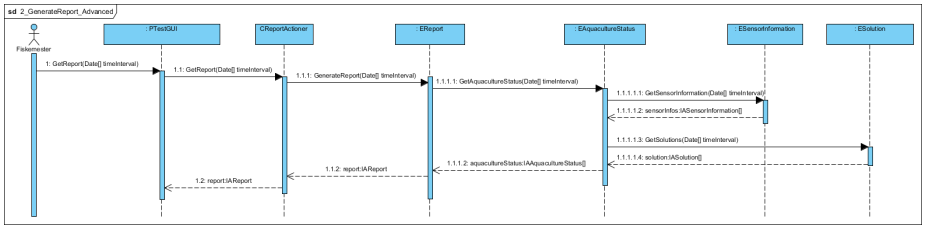
Figur 11: Avanceret sekvensdiagram for brugsmønster 1, stor udgave kan ses på Figur 30, s. 75.

PTestGUI kalder da metoden *StartOptimization*() på COptimizationActioner klassen, som ligger i Control-laget. Denne metode opretter en tråd hvorpå der, med et interval på 2 sekunders pause imellem, kaldes *GenerateSolution*(), som så laver optimeringen. Da dette kører på en anden tråd, forbliver brugerfladen responsiv for brugeren. I metoden *GenerateSolution*() kaldes klassen ESensorInformation, beliggende i Domain-lagets Entity-pakke, for at indhente de sidste nye sensor informationer. Informationerne kommer på form af et interface objekt, som ESensorInformation nedarver, på denne måde overholdes Acquaintance delen af PCMEF+.

Når Negotiator delen har fundet en løsning, gemmes den fundne løsning ved at danne et nyt Solution-objekt som kan blive gemt ved hjælp af klassen MSolutionMapper, beliggende i Domain-lagets Mediator-pakke, der står for persistens og hentning af Solution-objekter fra Foundation-laget. Hvordan denne persistens og hentning foregår vil blive forklaret i et senere sekvensdiagram.

Også i dette diagram bliver der vist hvordan man kunne notificere en PLC, dette bliver ikke forfulgt i implementeringen, og bliver kun vist for at vise hvordan det kunne være gjort.

Brugsmønster 2: GenerateReport

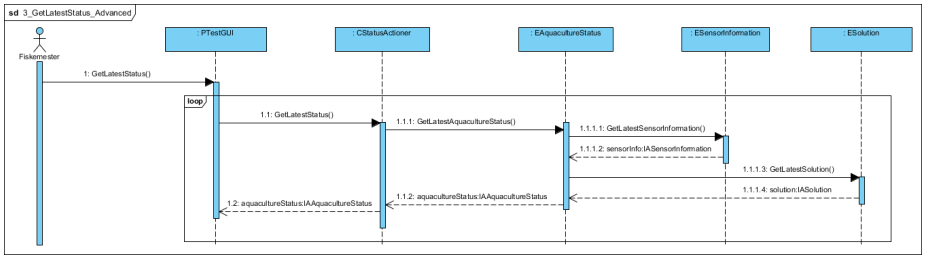


Figur 12: Avanceret sekvensdiagram for brugsmønster 2, stor udgave kan ses på Figur 31, s. 76.

På brugerfladen PTestGUI angiver fiskemesteren et tidsinterval som der ønskes data indhentet fra. Da bliver metoden *GetReport*() kaldet på CReportActioner

klassen, som igen kalder EReport klassens metode *GenerateReport*(). I denne metode bliver EAquacultureStatus klassen kaldt for at indhente de ønskede data inden for tidsintervallet. EAquacultureStatus kalder så de passende metoder på ESensorInformation og ESolution. Fra EAquacultureStatus returneres et interface objekt, kaldet IAAquacultureStatus, til EReport. EReport gemmer dataen og returnerer et interface objekt kaldet IAReport som så returneres af CReportActioner. Det er så brugerfladens opgave at vise disse data på en overskuelig måde. I et fuldt system kunne dette være på graf form, men her bliver det bare vist som lister.

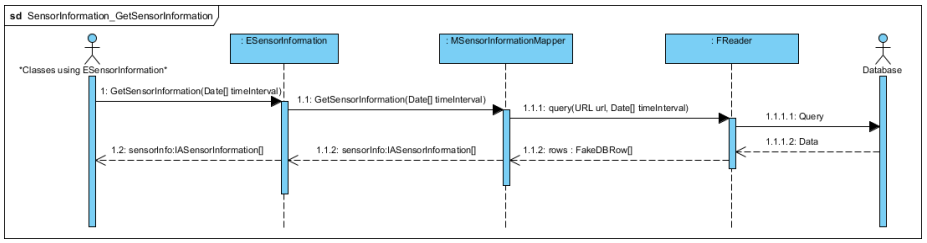
Brugsmønster 3: GetLatestStatus



Figur 13: Avanceret sekvensdiagram for brugsmønster 3, stor udgave kan ses på Figur 32, s. 77.

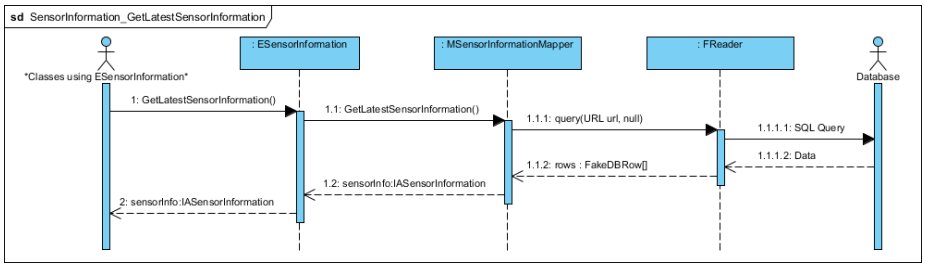
Det ønskes løbende fra fiskemesterens side at se de nyeste værdier for sensor informationer og optimeringsløsninger. Denne sekvens startes af fiskemesteren. Når først denne sekvens er påbegyndt, bliver der med et par sekunders mellemrum indhentet de nyeste data i et loop på en ny tråd. PTestGUI kalder *GetLatestStatus*() CStatusActioner, som benytter metoden *GetLatestAquacultureStatus()* på EAquacultureStatus til at indhente informationerne fra ESensorInformation og ESolution. Fra EAquacultureStatus returneres interface objektet indeholdende de ønskede data. Dette resultat returneres til PTestGUI som så viser de fundne data på en overskuelig måde.

Hentning af data fra Foundation-laget



Figur 14: Sekvensdiagram for GetSensorInformation, stor udgave kan ses på Figur 33, s. 78.

De viste diagrammer er sekvensdiagrammer af, hvordan sensor informationer indhentes i Domain-laget. Dette sker igennem en MSensorInformationMapper

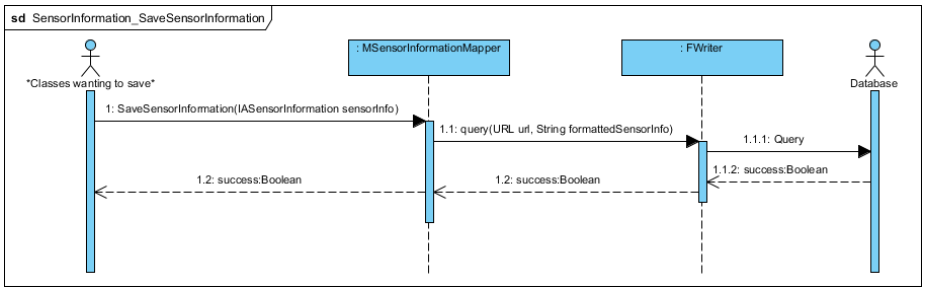


Figur 15: Sekvensdiagram for GetLatestSensorInformation, stor udgave kan ses på Figur 34, s. 79.

som i princippet skulle stå for at lave SQL og håndtere records fundet herfra, men da der ikke arbejdes med en egentlig database, består dette i at der angives en url på den database, der ønskes brugt, i dette tilfælde den for Sensor Informationer, og så et tidsinterval som afgrænser søgningen. For at hente de nyeste data skal tidsintervallet angives som null. MSensorInformationMapper kalder da metoden *query*() på FReader som så skulle formidle forbindelsen til databasen. Det, der returneres fra FReader, er et array med objekter af typen FakeDBRow. Dette gøres for at have et objekt som kan repræsentere en record. Dette arraylaves om til et sæt af Sensor Informationer somn returneres som interface objektet for typen. Dette returneres igen af ESensorInformation til klassen, som har kaldt metoden. På næsten samme måde indhentes de nyeste data gjort.

På tilsvarende måde hentes Solution objekter, her bare ved hjælp af en MSolutionMapper. Ud over dette, og at den returnerede værdi er af typen IASolution, er metoderne for hentning af Solution-data ens med SensorInformation, og vil derfor ikke blive vist.

Persistens af data i Foundation-laget



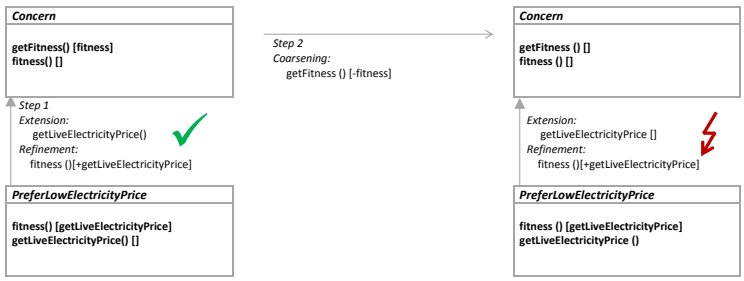
Figur 16: Sekvensdiagram for SaveSensorInformation, stor udgave kan ses på Figur 35, s. 80.

På ovenstående sekvensdiagram kan det ses, hvordan ny sensor data gemmes i databasen. Dette sker ved at en klasse, der ønsker at gemme, kalder MSensorInformations metode $SaveSensorInformation()$ med et interface objekt for Sensor Information. I $SaveSensorInformation()$ benyttes $toString()$-metoden

på interface objektet, som det andet argument i kaldet til FWriter. Grunden til at det kan gøres på denne måde, er fordi ESensorInformation og ESolution overskriver *toString*() med en metode, som returnerer en String som er formateret til at blive skrevet som en ny linje i databasen. Selve metoden til at skrive til, og hente fra CSV-filen vil blive forklaret i afsnittet om implementering. Ligesom ved hentning af data foregår dette tilsvarende for MSolutionMapper når der skal gemmes nye optimeringsløsninger.

9.2.10 Eksempel på Single-class reuse contract

Som tidligere nævnt hjælper en single-class reuse contract med at få overblik over ændringer, der kunne føre til konflikter i systemet. Da sådanne konflikter skal undgås til alle tider, er denne form for dokumentation et godt værktøj til at afhjælpe problemet. I dette afsnit vil der blive givet et eksempel på, hvorledes en kontrakt kunne se ud, sammen med ændringer og de konflikter der opstår eller ændringer der går godt. En single-class reuse contract skal forstås som en dokumentation for en nedarvning og extension af en class der er blevet lavet. Nedenstående figur er et eksempel på en kontrakt, hvor der bliver håndteret forskellige eksempler.



Figur 17: Single-class reuse contract, stor udgave kan ses på Figur 36, s. 81.

Denne figur skal ses som to forskellige tidspunkter i udviklingsarbejdet. Den venstre "søjle" er en single-class reuse contract. Den øverste kasse repræsenterer en abstrakte klasse, som den eksisterer i frameworket der arbejdes med, før der er blevet nedarvet og/eller udført en extension på den. Under dette ses en nedarvning fra den abstrakte klasse, med et par notater der skal hjælpe med at forstå nedarvningen og hvilke konsekvenser den har for underklassen.

I eksemplet ses den abstrakte klasse Concern, hvor der er udvalgt to bekrivende metoder for at illustrere brugen af single-class reuse contract. Da der ikke har været adgang til kildekoden, er det antaget, at metoden *getFitness*() kalder metoden *fitness*(). Denne figur viser hvordan der gerne vil tilføjes et metodekald hos *fitness*(), og hvordan det kan gå galt, hvis der bliver lavet en ændring i den abstrakte klasse.

- *Step 1*
  Det første skridt er en nedarvning fra den abstrakte klasse, dette kaldes også at extende en klasse. Der er her ændringer fra den abstrakte klasse, hvor de metoder, der overskrives, beskrives. Ved "Step 1"er beskrevet den ændring der skal ske i systemet. I dette tilfælde er der to ændringer. Den

første er en "Extension" til klassen i form af $getLiveElectricityPrice()$-operationen. Den anden ændring er en "Refinement", hvor der bliver lavet et kald til operationen $fitness()$. Argumentationen for at lægge operationen her er, at der skal være mulighed for indfangning af nyeste elektricitetspriser hver gang der skal udregnes et concern. Dette er ikke en optimal måde at indhente oplysningerne da metoden bliver kaldt mange hundrede gange i sekunder, men er gjort for at skabe et overskueligt eksempel. Her er der ingen konflikt, så denne ændring kan godt implementeres, hvilket er indikeret med et grønt rettemærke.

- *Step 2*
  I det andet skridt, sker der en "Coarsening", hvor der bliver lavet en ændring i den abstrakte klasse. Her er det kaldet til $fitness()$ der bliver forsøgt fjernet fra $getFitness()$. Da denne ændring ikke indeholder en måde at indhente de nyeste elektricitetspriser via $getLiveElectricityPrice()$, så er der en konflikt med "Step 1"som skal håndteres, dette er vist ved hjælp af den røde konflikt-pil.

Som det fremgår af eksemplet, vil sådanne kontrakter give en god mulighed for at se, hvordan ændringer ville påvirke resten af systemet, da man holder øje med evolutionen for hver klasse. Dette eksempel bliver givet som et forslag til en måde at undgå udvidelsesproblemer, der kunne forekomme ved udbygning til et fuldt system, og burde laves for alle relevante klasser.

10 Implementering

10.1 Introduktion

Da der, som følge af Unified Process aktiviteterne fra analyse og design, er en detaljeret beskrivelse af, hvordan systemet skal implementeres, vil der i dette afsnit være fokus på integrering af Multi-optimerings frameworket og de dele af systemet, som ikke er blevet beskrevet i foregående afsnit. Disse dele er den grafiske brugerflade, forbindelse til database, samt implementering af mulighed for test i dette proof of concept system.

Et Proof of Concept program, også kaldet et POC, er et program med begrænset omfang, udviklet for at vise en idé eller koncept. På basis af et POC er det meningen, at man kan se, om et fuldbyrdet program er muligt at udvikle og drage andre konklusioner. Aquaculture POC programmet er udviklet med basis i Controleum til løsning af multi-objective problemer, hvor POC programmet skal vise, at dette også kan benyttes indenfor dambrugs domænet. POC programmet blev udviklet med UP metoden, hvor de første skridt tog udgangspunkt i et fuldbyrdet program. Ud fra resultaterne af UP aktiviteterne blev der herefter truffet beslutninger om afgrænsninger i forhold til et fuldt program. Følgende er en liste over dele, som henholdsvis skulle og ikke skulle implementeres.

Skulle implementeres

- *PCMEF+ framework*
  Efter arbejdet med UP aktiviteterne er der fremkommet en model for

systemet, der skal implementeres for at vise det også fungerer i praksis.

- *Multi-optimerings framework der virker med snapshot billeder*
  For at kunne vis,e at multi-optimerings frameworket også virker i dambrugs domæne,t skal det kunne påvises, at det faktisk er tilfældet. Den løbende optimering skal også kunne slås til og fra, så hver optimering kan aflæses af brugeren. Snapshot billeder betyder her, et sæt sensor informationer eller sensor data, som repræsenterer dambrugets nuværende status. Forskellen fra et fuldt system ville være at i det fulde system kunne der også tages højde for historiske data.

- *Brugerflade til fremvisning af dambrugets status*
  For at kunne vise brugeren resultaterne af optimeringen, er det nødvendigt med en brugerflade. Med dambrugets status menes der den sidst gemte løsning på optimering, samt det sæt sensor informationer, som optimeringerne er blevet beregnet ud fra.

- *Brugerflade til fremvisning af rapporter*
  Det var også ønsket, at der kunne fremvises rapporter over dambrugets status og ønskede ændringer. I det fulde system kunne dette blive vist som grafer eller udprint, men i denne implementering vil der kun kunne fremvises lister med sensor informationer og optimeringsløsninger, som ligger inden for nogle dato og tidsparametre defineret af brugeren.

- *Persistens for optimeringsløsninger og sensor informationer*
  For at kunne give et godt billede af et fuldt systems udseende, er det nødvendigt med en form for persistens. I dette projekt bliver det i form af CSV filer.

Skulle ikke implementeres

Følgende punkter er dele af systemet, som ikke vil blive implementeret i projektet, men er ønsket til et fuldt system. Punkterne bliver ikke implementeret, da de ikke ligger inden for dette projekts mål, som er at vise at der kan optimeres på et dambrug ved at udnytte multi-optimerings frameworket.

- Brugerflade til fremvisning af prognoser (sygdomme, produktets udvikling)
- Kommunikations hardware til systemets komponenter (PLC)
- Avanceret fremvisning af sensor data mm. i form af grafer
- Mulighed for ændringer eller fastsættelse i indstillinger af negotiator eller parametre hertil
- Understøttelse for flere forskellige typer af komponenter (forskellige iltningsmetoder, filtermetoder, mm.)

10.2 Implementering af PCMEF+

Ud fra design klasse diagrammet ses implementeringens udformning. Diagrammet er præcist i forhold til det overordnede design, og sekvensdiagrammerne beskriver også præcist de vigtigste forløb med parameter- og resultat-typer.

PCMEF+-lagdelingen er overholdt, så der er alle 4 lag og Acquaintance: Presentation, Control, Domain med Mediator og Entity, Foundation. Følgende er en beskrivelse af hvordan hvert princip er blevet overholdt fra PCMEF+:

- *Downward Dependency Principle*
  Ud fra design klasse diagrammet fremgår det tydeligt at øvre lag kun har afhængigheder til de nedre lag, og ikke omvendt.

- *Upward notification Principle*
  Der er ikke nogen notifikationer i dette proof of concept system, men i et fuldt system ville der være noget lignende med hensyn til forbindelsen til en PLC.

- *Neighbor Communication Principle*
  Lag må kun kommunikere med tilstødende lag, det vil sige lag, som er lige under eller over, og dette er blevet overholdt, for eksempel ved COptimizationActioner, beliggende i Control-laget, denne klasse kommunikerer kun med laget lige under, Domæne-laget, hvor den har forbindelse til ESensorInformation og MSolutionMapper.

- *Cycle Elimination Principle*
  Ud fra design klasse diagrammet fremgår det, at der ikke er nogen cykliske afhængigheder, så dette princip er også overholdt.

- *Class Naming Principle*
  Hver klasse, med undtagelse af dem, som nedarver fra optimeringsframeworket, overholder dette princip. Eksempelvis PTestGUI som ligger i Presentation, eller FWriter, som ligger i Foundation. Grunden til dette princip ikke er overholdt for optimeringsframeworket er, at det gør det nemmere at arbejde med, så Inputs, Outputs og Concerns har fået deres egen pakke i implementeringen så der er en tydelig opdeling, derudover bliver de kun brugt i COptimizationActioner.

- *Acquaintance Package Principle*
  Design klasse diagrammet viser, hvordan objekterne i Entity-pakken nedarver fra lignende interface objekter fra Acquaintance-pakken. Disse interface objekter er så det eneste, som bliver transporteret på tværs af lagene, som det også fremgår af design klasse diagrammet.

10.3 Implementering af optimeringsframework

Implementeringen af Controleum består af Inputs, Outputs, Concerns og en Negotiator, der alle bliver oprettet og håndteret af Control-lagets COptimizationActioner-klasse. Følgende er en overordnet beskrivelse af de forskellige dele, samt en forklaring af fremgangsmåden for generering af en ny løsning.

Inputs
Disse repræsenterer sensorer i dambruget, for eksempel en sensor for ilt-indhold i vandet. Disse inputs henter de nyeste værdier fra en database. I denne implementering er databasen blot en tekst-fil, hvori nogle værdier er gemt til demonstrering af systemet. I et færdigt system ville der her skulle laves et system, der aflæser sensorerne og gemmer de aflæste værdier i databasen.

Outputs
Outputs repræsenterer forskellige handlinger og processer, der kan skrues op og ned for. Et eksempel her kunne være ilt-tilførsel til vandet, hvor det tilsvarende output er implementeret som en værdi for hvor mange liter ilt der tilføjes i sekundet. Et output er implementeret til at have to funktioner, en funktion der genererer en tilfældig værdi, indenfor et passende interval. I eksempel med ilt-tilførsel ville dette interval være de mulige værdier for hvor mange liter ilt der kan tilføjes i sekundet. Den anden funktion et output har, er en funktion, der muterer den genererede værdi. Dette vil sige, at i stedet for at generere en helt ny tilfældig værdi, returneres en værdi der er enten lidt højere eller lidt lavere end den tidligere værdi.

Concerns
Tilsammen giver disse to grupper de indstillinger og værdier som Concerns arbejder ud fra. Et concern modellerer et aspekt af det fulde system, som man ønsker minimeret eller maksimeret, imens det skal holde sig inden for visse grænser, ofte opstillet af inputs afgrænsninger eller andre concerns. Hvert concern har enten en boolean accept metode eller en fitness metode. Hvis et concern har en accept metode returnerer denne enten true eller false, alt efter om inputs og outputs overholder de grænse-værdier, som concernet er programmeret til at varetage. Hvis et concern derimod har en fitness metode, returnerer det en tal værdi fra 0 til 1, alt efter hvor tæt på en ønsket løsning input og output er. Des tættere fitness er på 0, des mere tilfreds er det givne concern.

10.3.1 Fremgangsmåde for optimering

De førnævnte objekter og en instans af Negotiator-klassen, også fra optimeringsframeworket, oprettes i COptimizationActioner klassens private metode kaldet *GenerateSolution*(). Denne metode henter så de seneste sensor informationer igennem ESensorinformation. Herefter tilføjes alle inputs, outputs og concerns, som skal bruges til optimeringen. Disse tilføjes ved hjælp af dynamisk linking af objekter ved hjælp af decouplink [Martin Rytter, 2010], således at et tomt objekt bliver fyldt med dynamiske links til de førnævnte inputs, outputs og concerns. I praksis betyder dette, at man får et stort objekt, der kan bruges til at tilgå de enkelte inputs, outputs og concerns. Herefter bruges Negotiator-objektet til at finde en løsning på optimeringen. Negotiator-klassen er brugt som en "black box", forstået på den måde, at det er en eksisterende funktion fra Controleum, der bruges som den er, uden at have ændret i klassen. Forklaringen på hvordan denne Negotiator-klasse virker, kan læses i afsnittene som omhandler multi-objektiv optimering, genetiske algoritmer og multi-agent systemer.

Når Negotiator objektet har fundet løsningen, gemmes denne i en ny solution, som så gemmes i databasen ved hjælp af MSolutionMapper, så det følger PCMEF+ strukturen. Da optimeringen skal ske løbende, kaldes metoden *GenerateSolution*() altid fra en anden tråd end den, som styrer systemet, da dette giver en mere responsiv brugerflade. Denne tråd, som kalder *GenerateSolution*(), oprettes i metoden *StartOptimization*(), men der sørges for, der altid kun er 1 tråd, som løser optimeringen af gangen.

Der er også implementeret en anden metode kaldet *StopOptimization*(), som stopper den tråd der er i gang med at løse optimeringsproblemet. Dette

håndteres på en pæn måde, så optimeringen kan sættes i gang igen.

10.4 Inputs og Outputs

Aqualculture Control System benytter sig af negotiatoren fra Controleum, der via genetiske algoritmer skaber en mængde permutationer af output og, via concerns, rangérer disse i forhold til inputs. I den forbindelse skal systemet modellere in- og outputs.

10.4.1 Inputs

Inputs gemmes i programmets database af de enkelte sensorer. Metoden er ikke modelleret, men det er oplagt at antage at hver elektonisk sensor har sit eget interface eller medfølgende program, der kan køres på en server eller computer, hvorefter dataene gemmes. Programmet har modelleret en del sensorer, men listen er hverken udtømmende, eller garanteret korrekt. Afhængig af hvor avanceret det endelige program skal være, kan mængden og typerne af sensorer justeres til at passe til de implementerede concerns.

Inputs

- CO_2 - mængden af kuldioxid opløst i vandet i mg/L
- NH_4 - mængden af ammoniak opløst i vandet i mg/L
- NO_2 - mængden af nitrit opløst i vandet i mg/L
- NO_3 - mængden af nitrat opløst i vandet i mg/L
- O_2 - mængden af ilt opløst i vandet i mg/L
- Biomass - den samlede biomasse i en dam
- pH - pH værdien i vandet
- Temperatur - Temperaturen i vandet
- Fodertype - Hvilken type foder, der benyttes
- Fodertider - et skema over fodertider
- El priser - et skema over priserne på strøm de næste 24 timer

Nogle inputs, såsom foderskema, kan være inputs, der kun benyttes, hvis brugerne vælger at fastlåse et concern. Alternativt kan optimeringsalgoritmen selv sørge for fodring, når dette er mest gunstigt. Fælles for inputs er, at de benyttes af optimeringsalgoritmen når denne kalder sine concerns. Her benyttes disse inputs til at bestemme fitness og accept funktioner baseret på algoritmer i concerns. Eksempelvis benytter algoritmen for sikring af iltniveau sensorer for ikke blot ilt, men også pH og temperatur til at finde iltens opløsningsværdier, samt biomassen til at beslutte, hvor meget ilt der skal til for at modvirke fiskenes optag. Endelig kigges der på et fodringsskema for at sikre, at der tages højde for fiskenes øgede iltindtag under fodring.

10.4.2 Outputs

Foreslåede outputs permuteres af optimeringsalgoritmen, og de mest optimale lagres i programmets database. Disse gemmes som tal i form af eksempelvis liter per sekund eller ændring af pH værdi. Disse tal-værdier er valgt ud fra præmisset, at de skal være nemt læselige og passe med optimerings programmets interface. For at føre optimeringerne ud i livet er systemet designet med en PLC. En Programmable Logic Controller er valgt, idet denne er nemmere at justere end i Aquaculture Control System. Dette betyder, at der ikke skal ske store ændringer i programmet ved anskaffelse af ny hardware, men at PLC'en blot skal omprogrammeres til at oversætte data i databasen til et forståeligt sprog for de nye komponenter. Denne metode betyder ikke blot mere overskuelig kode, men gør også systemet langt mere fleksibelt når man ser på udvidelse til brug i flere forskellige dambrug.

Outputs

- Afgasning - angiver mængden af atmosfærisk luft, der pumpes igennem vandet i L/s
- pH ændring - angiver ændring af pH værdien
- Foder fisk - angiver, om der skal tændes for fodringssystemet eller ej
- Mekanisk filter interval - angiver, hvor ofte det mekaniske filter skal rense sig selv
- Ilt volumen - angiver mængden af ilt, der skal tilsættes, i L/s
- Pumpehertz - angiver hvor mange hertz, pumperne skal køre med
- Benyt nød-ilt - angiver, om der skal tændes for nødilten

10.5 Implementering af Concerns

I proof of concept systemet er der implementeret de vigtigste concerns. Til et fuldt system skal der ikke kun skrives flere concerns til de mere detaljerede dele af et funktionelt dambrug, men disse, og de implementerede concerns, skal også kunne tilgå historiske data samt fremtidige data, som for eksempel vejrudsigter eller foderskemaer. De implementerede concerns er basale og bruger hovedsageligt afrundede værdier og simplificerede beregninger, hvor et fuldt implementeret system ville skulle benytte mere avancerede beregninger, der er nøje afstemt med det enkle dambrug og dets komponenter. Hard concerns, EnsureXXX i programmet, kan returnere TRUE eller FALSE. Disse sikrer grænseværdier osv. Soft concerns, PreferXXX i programmet, returnerer en fitness værdi imellem 0 og 1. Her er det vigtigt at bemærke, at en fitnessværdi på 0 er det optimale, og en fitnessværdi på 1 er den dårligste løsning.

10.5.1 EnsureCheapestOxygen

Dette concern skal sikre, at der ikke benyttes unødvendigt dyre ilt-tilførselsprocesser, og at der ikke bruges unødig energi på tilførsel af for megen ilt. Concernet har en accept-funktion, der sikrer, at der kun bruges nød-ilt ved en

opløsning af ilt i vandet på under 5 mg/L efter den almindelige ilt tilførsel, hvilket er minimummet for regnbueørreders trivsel, hvor 3 er dødeligt [Raleigh, 1984]. I et fuldendt system ville dette concern samtidig kun tillade brug af nødilt, hvis der samtidigt var skruet helt op for den almindelige ilttilførsel samt sikre, at der ikke bliver spildt nødilt, hvis der er planer om pumpning af mere ilt der vil betyde et kortvarigt dyp til ikke mindre end 3 mg/L.

10.5.2 EnsureCleanWater

Dette concern skal sikre en god vandkvalitet for fiskene. En god kvalitet er afhængig af stoffer i vandet i form af kemiske forbindelser. Yderligere skal der ses på partikulater i vandet i form af foder rester, fækalier og andet, der falder i vandet. Den første del af concernet sikrer, at det mekaniske filter arbejder hårdere under fodring end dets minimums-indstillinger. Her kommer der flere partikulater i vandet som følge af små foderstykker, der ikke bliver spist af fiskene. I et fuldbyrdet system skal dette concern også sikre pumpning i et nøje udregnet tidsinterval efter endt fodring samt et senere interval, hvor beregninger kan vise, at fiskene her udskiller fækalier som resultat af fodringen. Yderligere kan der evt. eksistere sensorer, der kan kvantificere mængden af partikulater fjernet af filteret eller i vandet, eventuelt beregnet af gennemlysning, der kan sikre sig et mere snævert system for det mekaniske filter. Kemiske forbindelser i vandet er resultater af opløste fækalier, mad og bakterie udskilninger. Nogle af de vigtigste ting at se på her er ammonium, som er ammoniak opløst i vand, og nitrit. For disse eksisterer grænseværdier, der nødvendigvis skal overholdes. NH_4 (ammonium) skal ligge på højest 0.03 mg/L. I et fuldt system, hvor man kan se på systemets historik, kan denne grænse løftes til 0.05 mg/L i kortere perioder. NO_2 (nitrit) skal ligge på højest 0.55 mg/L. Begge stoffer, og mange andre, der ikke er modeleret i POC programmet, fjernes via det biologiske filter. Dette filter styres af pumperne, der pumper vand ind og sikrer bevægelse i bassinet med biokuglerne. Af denne grund beregner concernet en acceptfunktion, hvis der pumpes nok til at fjerne stofferne fra vandet. I et fuldt implementeret system ville dette concern skulle supplementeres af et concern med en fitness funktion, der kunne benytte formler afhængige af temperatur, pH, lys etc. for at se, hvor stor en mængde af disse stoffer et givent biologisk filter ville kunne fjerne. Herefter kunne disse udregninger kombineres med pumpe Hz og via accept funktionen sikre acceptable niveauer, imens fitness funktionen ville foretrække et løsningsforslag, der fjerner store mængder af stoffer. Dette ville så blive modarbejdet af bl.a. PreferLowElectricityPrice concernet.

10.5.3 EnsureFedFish

Fisk skal have mad for at vokse sig store og sunde. Dette concern skal sikre sig at fiskene modtager nok mad. I concernet arbejdes der med et fastlagt foderskema. Dette illustrerer, hvordan et concern kan fastsættes i et fuldt implementeret system. I stedet for at beregne fitness for en permutation, ændres concernet til et hard concern, hvor løsninger kun accepteres, hvis de passer overens med det fastlagte data. I dette tilfælde er det fastlagte data et array med fodringstider, hvor concernet tvinger programmet til at finde løsninger, der fodrer fra de fastlagte tidspunkter og et kvarter frem i tiden. I et fuldt implementeret program kan man stadig bruge faste foderprogrammer. Alternativt vil man kunne de-

signe concerns, der samtidig beregner gunstige fodringstidspunkter mht. lys og herved beregner fitness baseret på disse. Samtidig vil der være en simpel accept funktion, der kun accepterer fodringsskemaer, der sikrer fiskene nok foder uden at overfodre.

10.5.4 EnsureOxygenLevelLimits

Dette concern bekymrer sig om niveauet af ilt i vandet. Det har en accept funktion, der sikrer, at ilten i vandet ligger imellem 5 mg/L og 13 mg/L. Regnbueørreder kan tåle kortere perioder med ned til 3 mg/L, men de vil her begynde at gå i panik og stresse, hvilket medfører yderligere aktivitet og iltsvind som følge [Raleigh, 1984]. Ydermere er 13 mg/L sat som et loft.

Der er et maksimum iltindhold for vand, over hvilket fiskenes gæller ikke kan udskille CO_2. Da regnbueørreder ikke er fanget i vand med højere koncentrationer end 13 mg/L, er dette sat som øvre grænse [Molony, 200]. I et fuldt system ville dette concern skulle kigge på historikken og sikre sig, at dette niveau kun eksisterer i meget korte perioder. Som følge vil en acceptfunktion skulle hæve minimums-niveauet efter et så lavt niveau, for at sikre en høj nok ilttilførsel til at negere fiskenes øgede forbrug som følge af det kritisk lave niveau.

10.5.5 EnsurePHValueLimits

Fisk trives bedst ved pH-værdier indenfor et bestemt interval. Et dambrug benytter vand, der sendes ind i systemet, samt recirkuleret vand. Tilføjet vand, hvad enten det er regnvand, kildevand etc. kan være enten mere eller mindre surt i forhold til fiskenes optimale pH-værdi. pH-værdien kan ændres via et system der tilføjer enten saltsyre eller kalk.

Dette concern sikrer sig derfor at pH-værdien for vandet holdes indenfor et acceptabelt niveau, der for regnbueørreder bør ligge imellem en pH-værdi på 6,7 og 8,5 [Raleigh, 1984]. Derfor har dette concern en accept-funktion, der kun accepterer løsningsforslag, hvor pH-værdien efter den foreslåede ændring i pH-værdi ligger inden for dette område. Implementeringen er en simplificeret løsning, der blot får foreslået en ændring i pH-værdi. I et mere avanceret system, ville det blandt andet være nødvendigt at regne ud, hvor meget pH-værdien ændres, alt efter hvor meget saltsyre eller kalk der bliver tilsat.

10.5.6 EnsurePumpHertzLimits

Dette er et sikkerhedsconcern. De modellerede pumper er lavet af Lykkegaard A/S og har en sikkerhedsmargen, som de kan arbejde indenfor. Det optimale hertz-niveau er ikke det eneste, de kan køre med. Man kan også øge eller sænke hertzene for at ændre på vandflowet. Imidlertid kan for store ændringer resultere i rystelser i pumperne, der kan skade dem, eller få pumperne til at udvikle mere varme, end deres kølesystem kan fjerne. Samtidig har den nedre grænse også betydning for pumpens effektivitet. Ved et vist niveau går al energien til at køre pumpen og intet til at flytte vandet, hvormed pumpen kun bibeholder løftehøjden uden at flytte vandet. Ifølge interviewet med Lykkegaard var begrænsningerne for deres pumper til et setup som det i Lerkenfeld i intervallet 48-52Hz [Svend Christensen, 2012].

10.5.7 EnsureSafeGasConcentration

Dette concern skal sikre at værdierne for forskellige gasser opløst i vandet forbliver indenfor sikre grænser. I POC programmet kigges der kun på CO_2-koncentrationen, hvor et fuldbyrdet program ville have flere gasser at skulle se på. Ydermere bruges der en lineær funktion, der antager fuldendt diffusion, hvor dette ikke er realistisk i den virkelige verden. Der ønskes et sikkert CO_2-niveau, hvilket er maksimum 2 mg CO_2/L [Raleigh, 1984]. Til POC funktionen modelleres der, at der kan pumpes imellem 0 og 200 liter atmosfærisk luft ind. Atmosfærisk luft vejer 1,2 gram per liter og indeholder 392 ppm CO_2, hvilket er 0,4704 mg [Wikipedia, 2012b]. Concernet antager, at der udlignes CO_2 imellem 1 liter vand og 1 liter luft og udregner herefter den nye koncentration i vandet. I virkeligheden sker der her diffusion imellem luft og væske, hvilket hverken er lineært eller simpelt. Samtidig er der en vekselvirkning pga. vands opførsel og bevægelse samt manglende kontaktflade imellem alt luften og vandet. Afhængig af disse faktorer, pH, temperatur og andre kemiske formler vil dette concern, fuldt modelleret, være langt mere avanceret.

10.5.8 PreferBalancedOxygenLevel

Dette concern har en fitness funktion der bedømmer hvor tæt på de optimale værdier et løsningsforslag er. Denne er afhængig af temperaturer, idet iltniveauet for regnbueørreder har forskellige optimale værdier, jævnfør concernet PreferCheapestOxygen. På basis af temperaturen beregnes en fitnessværdi for løsningen, hvor fitnessen er højest ved de optimale niveauer for den givne temperatur, 7 mg/L for temperaturer 15 grader eller derunder, 9 mg/L for temperaturer højere end det [Raleigh, 1984].

Disse tal er dog subjektive, idet et fuldt program ville skulle bruge en graf der beregner det bedste ilt-niveau løbende for den pågældende temperatur, hvilket ville kunne visualiseres som en S-kurve.

Desuden bør en færdig løsning forsøge at undgå for store udsving i iltindholdet i vandet, da dette ellers kan stresse fiskene.

10.5.9 PreferBalancedPHValue

Concernet beregner en fitnessværdi baseret på, hvor langt den foreslåede ændring er fra den optimale pH-værdi for et dambrug. I implementeringen er der valgt en arbitrær værdi på 7,5 som den optimale værdi, og dette concern returnerer så en fitness værdi imellem 0 og 1, alt efter hvor langt fra denne værdi den foreslåede pH-værdi er.

10.5.10 PreferHighFilterInterval

Det mekaniske filter bruger en stor mængde strøm på jetdyserne, der bruger vandtryk til at rense filteret for slam og dreje tromlen rundt. Derfor foretrækkes et højere interval imellem rensningerne. Det antages, at filteret mindst skal køre en gang hvert femte sekund, hvilket er et kvalificeret gæt baseret på Lerkenfeld, hvor intervallet var sat til syv sekunder. Det maksimale interval er sat arbitrært til femten sekunder. Dette concern konflikter med EnsureCleanWater og et uimplementret concern, der kan have foretrukne værdier for forskellige stoffer

i vandet. På denne måde sikres rent vand uden at spilde energi på unødvendig rensning.

10.5.11 PreferLowAerationVolume

Dette concern sikrer, at der kun bruges den nødvendige mængde afgasning ved at have en bedre fitnessværdi for lavere mængder afgasning. Afgasning bruger blæsere, der er meget dyre i drift, og derfor arbejder dette concern i spænd med PreferLowElectricityPrice og modvirker de concerns, der vil sikre et højt iltniveau eller sikre specifikke foretrukne grænser for gasser i vandet.

10.5.12 PreferLowElectricityPrice

Dette concern er en constraint for en stor del af de andre concerns. Det har kun en fitness-funktion, der ser på prisen for strøm på det pågældende tidspunkt og strømforbruget i systemet. Et fuldt udviklet system ville her skulle have et noget mere indviklet concern. Der ville skulle ses på historik og på et komplet sæt af dagspriser. Herudfra ville man kunne nedjustere fitnessen for strømspild i tidsintervaller med meget høje priser og øge fitnessværdien i intervaller med lavere priser i forhold til dagsprisernes gennemsnitspriser. Yderligere kunne det overvejes at skabe løsningsforslag med indstillinger for en periode på et døgn af gangen. Ud fra dette sæt ville man bedre kunne sikre at forbruget af strøm var mere hensigtsmæssig i forhold til priserne, samtidig med at ting som fodring ville kunne koordineres i samarbejde med døgnets lyse timer, vejret og elpriser.

10.5.13 PreferLowOxygenationVolume

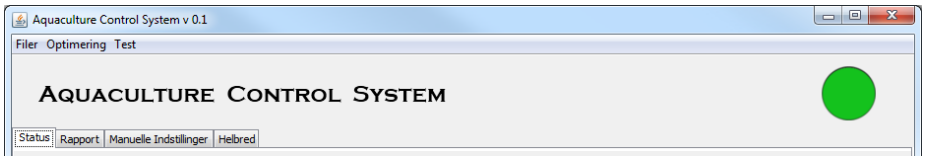
Dette concern sikrer via fitnessfunktionen, at der foretrækkes et lavt forbrug af iltning. Hvad enten der benyttes jetdyser eller keramiske sten til iltning, koster det penge at pumpe ilten ind i vandet og ikke al ilten opløses heri. Derfor bruges dette concern til at modvirke EnsureOxygenLevelLimits for at sikre, at der kun pumpes det nødvendige ilt ind. I et fuldt implementeret system ville dette concern også tage højde for planlagte fodringer til at ændre fitnessværdien på bestemte tidspunkter samt sikre, at der ikke er store fluktueringer i iltniveauerne. Alternativt kunne fluktueringerne rykkes ud i et separat concern på samme måde som PH fluktueringerne.

10.5.14 PreferSmallPHValueChanges

Concernet benytter sig af en eksponentiel funktion til at udregne en fitnessværdi baseret på ændringerne i pH-værdi. Idet en stor ændring ikke er godt for fiskene, vil denne sikre, at ændringer i pH-værdi holdes så små som muligt, dog uden at forhindre optimeringsprogrammet i at lave store ændringer for at sikre habitable PH værdier.

10.6 Grafisk brugerflade

Selve dambrugs POC programmet er designet med tabs til de individuelle dele, både dem, der er implementeret, men også tabs til at vise, hvad et fuldt program skal have. Følgende er først et billede af den grafiske brugerflade, sammen med en beskrivelse af de forskellige dele af brugerfladen.



- *"Filer"*-menuen
  I Filer-menuen findes kun en knap kaldet "Afslut", der lukker programmet. Programmet kan også lukkes ved at trykke på krydset i højre hjørne.

- *"Optimering"*-menuen
  Denne menu indeholder muligheden for enten at starte eller stoppe optimeringstråden, ved hjælp af to knapper kaldet "Start" og "Stop". Som udgangspunkt er det kun Start-knappen, der kan trykkes på, hvilket skyldes, at det er brugeren som skal sætte gang i brugsmønstret. Den grønne cirkel i højre side indikerer, at optimeringen allerede er igang. Hvis der trykkes på Start-knappen ændres den røde cirkel til en grøn. Dernæst bliver der forsøgt at kalde metoden *StartOptimization*() på en instantiering af COptimizationActioner-klassen. Når der trykkes på Stop-knappen, ændres den grønne cirkel til en rød, og COptimizationActioner kaldes med *StopOptimization*().

- *"Test"*-menuen
  I denne menu findes muligheden for at gemme sæt af sensor informationer, der hver repræsenterer en given situation i dambruget. Dette er eksempelvis "Ændringer i pH-værdi", hvor det ønskes at påvise, at systemet kan finde en ændring i pH-værdien, som bringer den tættere på det optimale.

- *"Status"*-tab
  Denne tab viser sidste sæt sensor informationer samt indstillinger for systemets komponenter, som udregnet af negotiatoren på basis af sensor informationenerne. I et fuldt implementeret system skulle felterne for hver sensor og komponent udvides med en graf, der kan hjælpe brugeren med at visualisere systemets gang. Endelig har systemet en start knap samt et felt, der hjælper til med at visualisere systemets status ved hjælp af tekst og farveskift.

- *"Rapport"*-tab
  Rapport skal, i et fuldt implementeret program, vise grafisk historik med mulighed for at se på individuelle tidsperioder, samt vise prognoser for fiskebestandens vækst. I denne implementering er der kun tale om en historik på tabel-form, derudover er prognoser ikke implementeret.

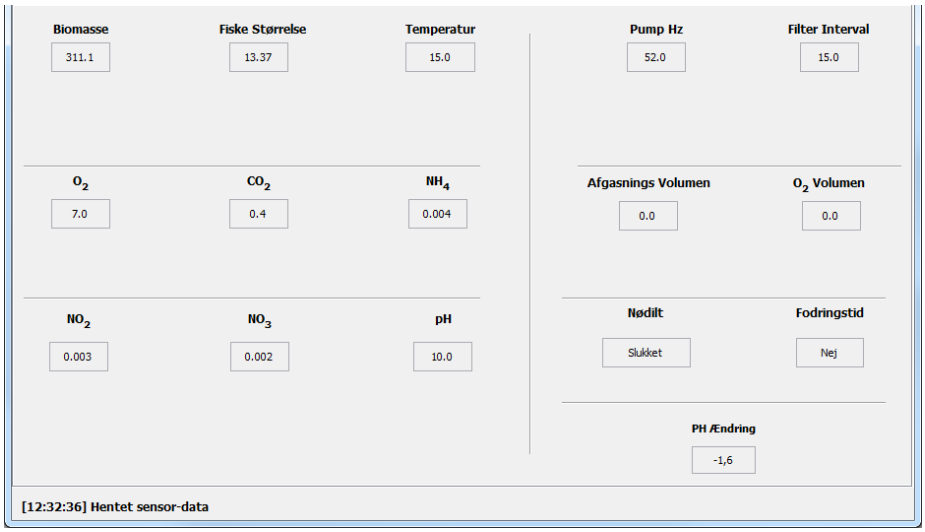
- *"Manuelle indstillinger"*-tab
  Manuelle indstillinger er, hvor man i et fuldt implementeret system eksempelvis ville kunne ændre på fodertyper eller fastlåse concerns til et bestemt interval. Denne funktionalitet er ikke implementeret.

- *"Helbred"*-tab
  Helbred skal vise prognoser for sundhed, evt. baseret på et intelligent system, der lærer af tidligere episoder. Denne funktion udnytter under historisk data til at lave disse prognoser. Denne funktionalitet er ikke implementeret.

Der vil nu blive gennemgået de to tabs som er implementeret.

Status-tab

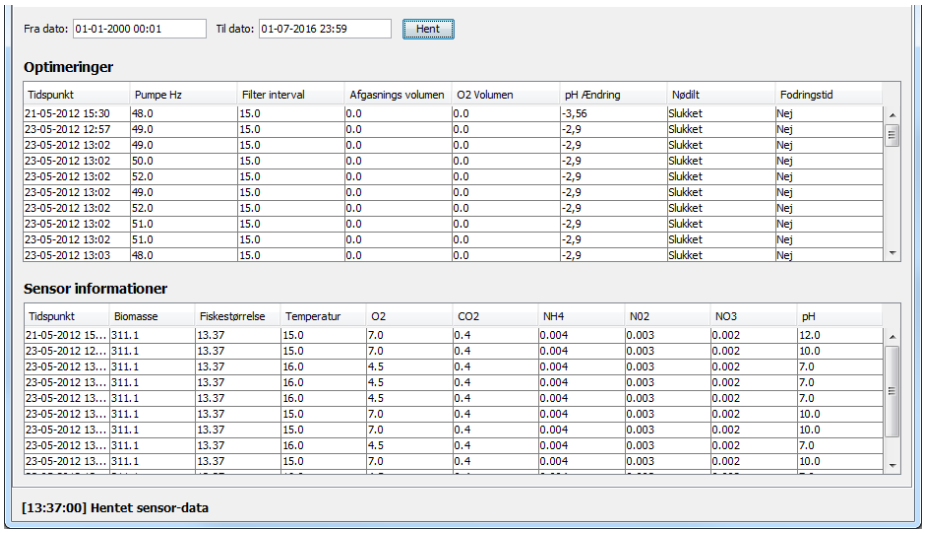
Denne del af brugerfladen viser status på dambruget i snapshot øjeblikket. I venstre side ses de overvågede sensor informationer, og i højre side de optimeringer som skal udføres. I eksemplet på billedet er pH-værdien den eneste værdi, der skal ændres så den kommer under 8,5. I bunden ses en status linje som opdateres hver gang, der hentes et sæt sensor informationer, også kaldet sensor data, eller hvis optimeringen bliver sat igang.



Når systemet starter, bliver der oprettet en ny tråd, som med korte mellemrum indhenter den nyeste status. Dette bliver gjort ved et kald igennem CStatusActioner klassens metode *GetLatestStatus*(), og på den måde overholdes PCMEF+ frameworket. Der returneres et interface objekt af typen IAAquacultureStatus. Ud fra dette objekt kan interface objekterne for dets IAStatus og IASensorInformation hentes. Fra disse objekter bliver alle de relevante værdier hentet og vist pænt på brugerfladen. Til sidst bliver status-linjen opdateret med et klokkeslæt og definition af statusen.

Rapport-tab

Denne del står for at vise historiske data. I det fulde system burde der være en form for eksport metode. I denne tab ses først to input felter: et med en fra dato, og et med en til dato. Når disse felter er udfyldt korrekt, trykkes på Hent-knappen, som så henter data der ligger inden for det angivede interval.



Når der trykkes på Hent-knappen, kaldes *getReportButtonActionPerformed*() i PTestGUI, som først bruger regulære udtryk for at tjekke, at interval værdierne er gyldige. Hvis dette er tilfældet, konverteres disse værdier til Date-typer og gemmes i et array, som gemmer på tidsintervallet. Dette tidsinterval bliver så givet med som parameter til et kald til CReportActioners metode *GetReport*(). *GetReport*() returnerer et objekt af typen IAReport, som indeholder alle data, der er behov for. Dernæst bliver de to tabeller tømt, og de nye værdier sat ind. Hvis der sker en fejl, opdateres linjen.

10.7 Persistens af data

10.7.1 CSV som database

I dette projekt bliver der brugt CSV-filer som database. Der arbejdes med to CSV-filer: "SensorInformationData.csv"og "SolutionsData.csv". Disse to CSV-filer bliver nu vist nedenfor i nævnte rækkefølge:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	InputID	Timestamp	PHValue	Temperature	OxygenLevel	BioMass	NO3Level	NO2Level	FishSize	CO2Level	NH4Level	ElectricityPrice	FeedingTimes	
2	2	23-05-2012 13:51	10.0	15.0	7.0	311.1	0.002	0.003	13.37	0.4	0.004	[1.0,3.0,1.0,2.0,4:	[6:00,9:00,12:00,15:00,18:00]	
3														

Figur 18: SensorInformationData.csv, stor udgave kan ses på Figur 37, s. 82.

	A	B	C	D	E	F	G	H	I
1	InputId	Timestamp	PumpHertz	OxygenationVolume	AerationVolume	FeedFish	UseEmergencyOxygen	MechanicalFilterInterval	ChangeInPHValue
2	1	21-05-2012 15:30	48.0	0.0	0.0	0.0	0.0	15.0	-3.56
3									

Figur 19: SolutionsData.csv, stor udgave kan ses på Figur 38, s. 83.

- *SensorInformation.csv*
  Denne fil repræsenterer linjer, som kunne hentes af en database vedrørende

sensor informationer, og første linje er kolonne-navne. InputId, PHValue, Temperature, OxygenLevel, NO2Level, NO3Level, FishSize, CO2Level og NH4Level er tal-værdier enten heltal eller komma-tal. Timestamp er en dato på formen "dd-mm-åååå HH:mm"for at have det præcist ned til et minuts usikkerhed. I denne implementering er FeedingTimes bare en tabel, som viser de forskellige tider, hvor der skal fodres. ElectricityPrice er en liste med 24 værdier, som alle repræsenterer prisen på el for den pågældende time i døgnet. Værdierne for ElectricityPrice er ikke præcise og bruges kun til at vise, at man kan inddrage elektricitetsprisen i udregningerne. I et fuldt system skal der indhentes de faktiske værdier fra udbyderne.

- *SolutionsData.csv*
  Denne fil repræsenterer det samme som SensorInformation.csv, bare for optimeringsløsninger. Her er InputId, PumpHertz, OxygenationVolume, FeedFish, UseEmergencyOxygen, MechanicalFilterInterval og ChangeInPHValue allesammen tal-værdier, hvor Timestamp er en dato på samme måde som Timestamp i SensorInformation.csv.

10.7.2 FakeDB-klassen

Som vist på design modellen, foregår al persistens af data via en CSV-fil, der tilgås ved hjælp af en FakeDB klasse, beliggende i Foundation-laget. Denne klasse indeholder tre metoder: $writeToCSV()$, $loadFromCSV()$ og $GetNextRowIdFromCSV()$. Følgende er en beskrivelse af disse metoder og hvordan de bruger CSV-filen.

- *GetNextRowIdFromCSV(URL url)*
  Denne metode læser linjerne i den CSV-fil, hvis url bliver givet med som parameter, og returnerer ID'et for hvad, en indsættelse af en ny linje skulle være.

- *loadFromCSV(URL url, Date[] interval)*
  Som parameter tager denne metode en url på CSV-filen, der skal hentes fra, og et tidsinterval i form af et Date-array. Hvis objektet for tidsintervallet er null, så søges der efter det nyeste, eller senest gemte, linje i filen. Filen indlæses så ved hjælp af en BufferedReader, som så læser den første linje og gemmer kolonne-navnene herfra. Dernæst gennemgås de følgende linjer og lægges over i et FakeDBRow-objekt, som kan indeholde disse elementer. Der undersøges herefter, om der er tale om den sidste linje i filen, og om det er den der ledes efter. I dette tilfælde bliver kun denne række gemt i et array som skal returneres. Hvis der ikke ledes efter den nyeste linje, og tidsstemplet for den givne række ligger inden for det tidsinterval, der angives i inputtet, tilføjes rækkerne. Til slut bliver de fundne rækker, hvis nogen, lavet om til den korrekte form og returneret.

- *writeToCSV(URL url, String formattedLine)*
  Som input tager denne metode en url på CSV-filen på samme måde som $loadFromCSV()$, men det andet input er en streng, som repræsenterer den linje der skal tilføjes til CSV-filen. Metoden laver et FileWriter-objekt, som har filen, der svarer til url'en fra inputtet. Dernæst hentes det næste ID fra $GetNextRowIdFromCSV()$ og FileWriter-objektet skriver linjen,

gemmer ændringen og lukker sig. Dernæst bliver der returneret om det var en succes.

11 Test og Resultater

11.1 Test og resultater

Med et færdigt proof of concept program er den sidste fase at undersøge, hvorvidt programmet fungerer og fremskaffer de forventede resultater. For at gøre dette, er det nødvendigt med en serie af test-cases, der undersøger forskellige funktioner af programmet og behandler resultaterne af disse. Idet programmet ikke har lige så snævre margener for de forskellige concerns og ej heller har evnerne til at se på historik, vil resultaterne variere mere end i et færdigt program. Visse concerns indsnævrer ændringer, men indsnævringerne er langtfra så stramme og udbredte som de ville skulle være i et fuldt program. Af samme grund vil nogle værdier fluktuere en del under tests, idet de blot skal ligge inden for relativt brede grænser. Herunder er beskrevet forskellige tests af det færdige program, hvor de fleste indbefatter adskillige concerns i en enkelt test. Undervejs i skabelsen af programmet er der foretaget mere snævre tests, der har sikret, at hvert enkelt concern virker korrekt. Derfor er følgende tests beregnet på at sikre en overdnet korrekthed af programmet, samt at de forskellige concerns virker i fællesskab, når de benyttes af negotiatoren. Af denne grund vil hver test fokusere på et enkelt element. Samtlige andre elementer vil være sat til at ligge indenfor acceptable grænser, hvorfor de største ændringer i systemets løsningsforslag vil være i relation til det enkelte input, der ændres i. Hermed vil man også kunne se, hvordan ikke relaterede outputs ændres indenfor acceptable grænser, idet disse stadig udregnes sideløbende med outputs relaterende til det aktulle input.

Acceptable parametre

- Biomasse: 311,1
- Fiske Størrelse: 13,37
- Temperatur: 15,0
- O_2 level: 9,0
- CO_2 level: 0,4
- NH_4 level: 0,004
- NO_2 level: 0,003
- NO_3 level: 0,002
- pH: 7,0

Output ved acceptable parametre

- Pump Hz: 48-52
- Filter Interval: 15
- Afgasnings Volumen: 0
- O_2 volumen: tæt på 0
- Nødilt: Slukket
- Fodringstid: Nej
- pH Ændring: 0,0-0,5

11.1.1 Test - pH værdi

Med alle værdier indenfor acceptable parametre ændres pH værdien til 10. Idet denne skal ligge imellem 6,7 og 8,5, er dette uacceptabelt. pH-værdien kan maksimalt ændre sig 2,9, men den har et ekstra stringent concern, der foretrækker små ændringer. Dette concern vil derfor fortrække, at ændringerne sænker pH-værdien langsomt mod dens mål på 7,5, som et tredje concern sikrer, er den optimale værdi. Når programmet køres på testen, ses det på figur 39, s. 84, at pH-værdien sænkes med 1,6-1,7, der er de to laveste værdier mulige, for at få pH-værdierne skubbet ned til et acceptabelt niveau. Herforuden er der små ændringer i O_2 volumen samt Pump-Hz, der, som forventet, fluktuerer inden for deres accept-begrænsede concerns.

11.1.2 Test - Aktivering af nød-ilt

Her sættes alle parametre til acceptable niveauer. Herefter sættes iltniveauet til værende 4,5 mg/L, hvilket er under den minimale grænse på 5 mg/L. Idet fuld åbning af almindelig iltning af vandet kun kan tilsætte 0,5 mg/L, tvinges nødilten til også at åbnes. Nødilten er en tænd/sluk funktion, der tilføjer 0,25 mg/L O_2. Grundet dette ses det på figur 40, s. 85, at den almindelige ilt tilsætning begrænses af PreferLowOxygenationVolume til kun lige at presse iltniveauet op over 5.0 mg/L. Igen fluktuerer Pump Hz grundet det non-stringente concern, mens pH værdien forsøges øget imellem 0,3 og 0,5, idet den ligger på 7 og det optimale er 7,5.

11.1.3 Test - Tid til fodring

Med de samme acceptable parametre ændres foderskemaet, således der skal fodres på det givne tidspunkt. Nu tændes der for fodermekanismen, imens pH-værdien igen fluktuerer imellem 0,3 og 0,5 samt mindre fluktueringer i iltningsvolumen og Pump-Hz. Denne test afprøver også intervallet hos det mekaniske filter, hvor dette normalt reguleres af PreferHighFilterInterval. Under fodring trumfer EnsureCleanWater imidlertid, hvor dette sikrer et ineterval på maksimum 8 sekunder. Se Figur 41, s. 86.

11.1.4 Test - Højt CO_2-niveau

Igen sættes samtlige parametre til de samme acceptable indstillinger, hvorefter CO_2-niveauet hæves til 2,5. Idet dette er et alt for højt niveau, vil der skulle afgasses. Ved kørsel kan det ses, at afgasningen kører omkring maksimum belastning, hvilket er krævet for at sikre et acceptabelt CO_2-niveau. Igen ses også de forventede fluktueringer omkring Pump-Hz og pH-værdien. Se Figur 42, s. 87.

11.1.5 Test - Iltning

Her sættes parametrene til at ligge indenfor de acceptable niveauer, hvorefter iltniveauet sættes til 5,0 Dette betyder, at nødilten ikke må aktiveres, men idet temperaturen ligger på 15 grader, vil PreferBalancedOxygen gerne have, at der opnås det optimale ilt niveau på 7 mg/L. Derfor skulle der gerne pumpes ilt ind i systemet. Ved test ses det, at der ikke tilføjes ilt. Ved forsøg på ændringer af O_2-niveauet indenfor de acceptable grænser, hvor der burde skulle tilføjes O_2,

sker der ikke noget, dette gælder både ved 15 og 16 grader. Ved 16 grader er det optimale niveau 9 mg/L, hvilket burde få concernet til at veje tungere ved kun 5 mg/L. Et kvalificeret gæt her er, at concernet PreferLowOxygenationVolume trumfer PreferBalancedOxygenLevel. I et fuldt udviklet system ville der derfor skulle sættes højere prioritet på PreferBalancedOxygenLevel og sikres, at dette concern får lov at spille ind. Alternativt kan concernet ændres fra at være linært til et eksponentielt concern, på samme måde som PreferSmallPHValueChanges. Se Figur 43, s. 88.

11.1.6 Test - Pump-Hz

Med optimale parametre ændres niveauerne af NO_2 og NH_4 til farlige niveauer. Dette betyder, at der skal renses mere vand via det biologiske filter, hvorfor pump-Hz vil stige. Normalt holdes dette parameter nede af PreferLowElectricityPrice, men EnsureCleanWater vil, ved disse niveauer, tvinge Pump-Hz til at ligge på mindst 50 Hz. Testen resulterer i, se figur 44, s. 89, at Pump-Hz ligger imellem 50 og 52, som afgrænset af de forskellige hard concerns her. Under tests, hvor der kunne bruges PreferLowElectricityPrice til at minimere Hz kommer den også ofte op over 50 Hz. Dette viser, at dette concern, i et færdigt system, skal enten have et højere prioritet eller laves om til en eksponentiel fitness funktion, der er bedre til at tvinge de bedste solutions hen imod en billig løsning rent elektricitets-mæssigt.

11.1.7 Resultat

De fleste tests viser, at systemet fungerer som designet. I tidligere iterationer blev der fundet fejl i hard concerns, hvor visse opsætninger skabte umulige situationer. De resulterende optimeringer var milevidt fra en reel løsning, hvilket gjorde det nemt at opdage disse fejl. Den eneste test der her har opført sig uforudset, var aktivering af iltning. Denne test viser behovet for enten mere stejle concerns eller udnyttelse af prioritering af concerns, en funktionalitet der er tilføjet i de nyeste versioner af frameworket end det brugte.

12 Diskussion

Et af projektets mål har været at vise, om optimeringsteknikker kunne benyttes til dambrugsdomænet. Udgangspunktet for projektet var et case study af de stigende energipriser og den fluktuerende prismodel i det kommende SmartGrid. Hypotesen for dette case study går på, at der også i dambrugsdomænet kan optimeres, således at energiforbruget minimeres.

Den oprindelige research fokuserede på dambruget Lerkenfeld i Himmerland, et moderne dambrug med et lukket kredsløb. Efter endt research af dambruget, samt et møde med repræsentanter fra Lykkegaard A/S, blev det indset, at en del af de oprindelige tanker omkring optimering af dambrug ikke kunne bruges i et anlæg så simpelt som Lerkenfeld. Anlægget havde ikke den ønskede mængde sensorer, ej heller mulighed for justering af de fleste komponenter. Grundet dette blev programmets udgangspunkt ændret til et fiktivt anlæg, med løst udgangspunkt i Lerkenfeld. Dette anlæg blev modelleret med samtlige komponenter som værende justerbare samt en mængde sensorer der kunne måles fra. En del af designet af dette blev lavet på basis af mødet med Lykkegaard A/S,

hvor det også blev klargjort, at en optimering i stor grad ikke kun skulle kigge på energipriser, men også foder samt tidsrum for produktets færdiggørelse. Resultatet af dette betyder et mere avanceret optimeringssystem, som ikke kun optimerer på minimering af energiforbrug, men samtidig også optimerer på foderforbrug og dermed samlet profit.

Til projektet blev der udviklet et proof of concept program, hvor valget af fokusområde faldt på den oprindelige plan, navnlig energiforbrug. Derfor blev der fokuseret på de forskellige energi-intensive komponenter, og sensorer hvis input påvirker bruget af disse. Imens projektet generelt har vist at optimering kan have en betydelig positiv indflydelse på dambrugets drift, kræver det et mere avanceret biologisk og teknisk grundlag at lave et program, som kan vise eventuelle reelle besparelser i form af energiudgifter. Samtidig er det blevet tydeligt, at et fuldt implementeret Aquaculture Control System til dambrug vil kunne give en fiskemester flere muligheder. Disse inkluderer besparelser i foder, energi, ilt, udledninger samt større indflydelse på produktets færdiggørelse. Dette behandles yderligere i perspektivering.

I projektet blev der valgt at benytte PCMEF+ for at vise, hvorledes et system kunne udvikles, så det kan styre et dambrug og er nemt at udvide. Grundet PCMEF+ opbygning, hvis fulgt korrekt, er der en del sikkerheder vedrørende arkitektur og udvidelse. Model-View-Control blev overvejet, da den indeholder mange af de samme kvaliteter som PCMEF+, men da PCMEF+ derudover også indeholder en række regler og principper til forbedring af systemudvikling, blev PCMEF+ valgt som den bedre løsning af disse to. I projektet blev der benyttet CSV-filer som database, men i et fuldt implementeret system burde der benyttes en relationel database da dette simplificerer arbejdet med generering af rapport-data. Derudover blev der ikke implementeret forbindelse til PLC til styring af fysiske komponenter. Dette blev ikke fundet nødvendigt for at opnå målet med proof of concept programmet, hvilket var implementering af optimering i dambrug.

Til implementeringen af optimeringsdelen blev der benyttet Controleum, udviklet under ledelse af Bo Nørregaard Jørgensen, der bruger genetiske algoritmer sammen med multi-objektive optimeringsteknikker. Frameworket er blevet udviklet med den tanke, at det skulle være så generisk som muligt for derved at gøre det nemt udvideligt. Dette har resulteret i et framework, hvor optimeringen styres af en central Negotiator, imens logikken for domænet er placeret i Concerns. Frameworket viste sig at være smertefrit at specialisere til dambrugsdomænet, hvilket også er frameworkets største kvalitet.

Alternativet til brug af genetiske algoritmer til multiobjektiv optimeringsproblemer kunne være metoder som for eksempel det beskrevne i afsnit 8.4 om multiobjektiv optimering, der forsøger at vandre langs pareto fronten for at finde en optimal løsning. Imidlertid har disse metoder en svaghed i forhold til genetiske algoritmer, idet disse metoder kan risikere at sidde fast i et lokalt optimum i stedet for et globalt. Brugen af genetiske algoritmer minimerer risikoen for at overse disse globale optima, der til tider kan være overordentlig svære at finde. Da genetiske algoritmer beror på tilfældigt genererede generationer, er der stadig en risiko, for at den optimale løsning overses.

13 Perspektivering

På basis af erfaringerne fra dette projekt, er der grundlag for videreudvikling af det implementerede projekt til et fuldt system. Dette underbygges i konklusionen. Ved videreudvikling er der nogle punkter som bør behandles eller overvejes.

For det første bør der være mere hensigtsmæssig og uddybende rapport funktionalitet. Denne skal blandt andet indeholde muligheden for visning af grafer til skabelse af bedre overblik samt prognoser for produktets færdiggørelse, sygdomme og lignende. Disse prognoser kan baseres på en intelligent logik som kigger på historiske data og lærer ud fra disse.

Ydermere bør der i et fuldt system være mulighed for at specificere hvilken type dambrug der skal styres. Dette inkluderer mulighed for indstilling af eksempelvis typer af komponenter, fiskearter og basin-udformning. Dette kunne gøres ved at implementere et bibliotek af indstillinger for de førnævnte typer.

Til udvikling af Concerns, anbefales det at der inddrages ekspertviden fra biologi og kemi, idet projektets research har vist at der ikke er tilstrækkelig med tilgængelig viden på disse områder inden for dambrug. Biologisk set mangler blandt andet information om optimale grænseværdier frem for acceptable intervaller. Desuden mangler der research i hvordan mange af de modellerede parametre har indflydelse på fiskevækst og trivsel. Angående kemi skal der foretages mere præcise beregninger af effekten af iltning, afgasning og det biologiske filter.

14 Konklusion

Imod forventning har projektet ikke utvetydigt vist, at der vil kunne opnås besparelser i elpriser via optimering i et dambrug. Til gengæld har projektet givet anledning til en række andre konklusioner.

For det første er det blevet påvist, at det er muligt at benytte optimeringsframeworket inden for dambrugsdomænet. Det er forfatternes klare overbevisning, at et fuldt udviklet system med tilstrækkeligt præcise Concerns vil kunne skabe store besparelser for dambruget samt give fiskemesteren bedre mulighed for at tilpasse sig det nuværende marked. Dette indebærer ikke kun forbrug af el og ilt, men også et bedre overblik, og dermed styring, af udledning fra dambrug samt dato for det færdige produkt. Ved at kunne minimere udledninger fra dambruget kan den totale biomasse i produktionen øges, og styring af dato for det færdige produkt tillader fiskemesteren at kunne sælge sit produkt på et tidspunkt, hvor markedspriserne er gunstige.

Det brugte optimeringsframework har vist sig at være effektivt til formålet. Grundet de mange varierende optimeringsmål er det forfatternes holdning, at genetiske algoritmer er det rigtige valg, idet disse minimerer risikoen for at overse globale optima.

Det implementerede projekt var ikke tilstrækkeligt avanceret til at kunne vise en kvantificerbar besparelse, men viste et tydeligt potentiale for at kunne gøre dette ved tilstrækkelig udbyggelse, hvilket blev underbygget i de tests der blev udført. Disse tests viser, at optimeringsframeworket laver de ønskede optimeringer med kun få fejl, som kunne skyldes en implementeringsfejl fra forfatternes side. Potentialet for projektet ligger i at udvikle mere avancerede Con-

cerns såsom *PreferLowElectricityPrice*, som opfylder ønsker om minimering af energiforbrug, der giver muligheden for at vægte løsningsforslag mere præcist.

Yderligere er der observeret potentiale til besparelser på andre hovedområder af dambrugets driftsomkostninger. Her hentydes blandt andet til foder, som har vist sig at stå for en stor del af budgettet. Med en intelligent styring af dambrug kan spild af foder minimeres, og dermed kan der også skæres ned på restprodukter.

Projektet har vist, at det er en klar mulighed at benytte optimeringsteknikker til at hjælpe med driften af dambruget ved at optimere den daglige drift. Det er forfatternes overbevisning, at der er potentiale for ikke blot optimering af energiforbrug, men også en generel optimering og effektivisering af moderne dambrug generelt. Især optimering af energiforbrug i forhold til variable prislister, som følge af implementeringen af SmartGrid har potentiale for at skabe besparelser. Her kan optimeringssystemet sørge for, at energitunge processer hovedsageligt køres i perioder med lave energipriser.

Litteratur

Carlos A. Coello Coello. Evolutionary multiobjective optimization a historical view of the field, 2006.

Christian W. Dawson. Projects in computing and information systems, a student's guide, second edition, 2009.

Rajiv Sabherwal Irma Beceraa-Fernandez, Avelino Gonzalez. Knowledge management: Challenges, solutions and technologies, 2010.

Ila Neustadt Jim Arlow. Uml 2 and the unified process, 2008.

Odd-Ivar Lekang. Aquaculture engineering, 2007.

U. Jumar D. Butler M. Schütze, T. B. To. Multi-objective control of urban wastewater systems, 2002.

Bills Maciaszek, Liong. Practical software engineering, chapter 5, 2004.

Bo Nørregaard Martin Rytter. Decouplink: Dynamic links for java, 2010. URL `http://http://decouplink.com/overview`.

Jan Bosch Michael Mattson. Framework composition: Problems, causes and solutions, 1998.

Melanie Mitchell. An introduction to genetic algorithms, 1996.

Brett Molony. Environmental requirements and tolerances of rainbow trout (oncorhynchus mykiss) and brown trout (salmo trutta) with special reference to western australia:a review, 200. URL `http://dialogue.dpi.wa.gov.au/docs/frr/frr130/frr130.pdf`.

COO for AquaPri Morten Priess. Interview med aquapri om dambrug, 2012.

EnergiNet og Dansk Energi. Smart grid i danmark, 2010.

P. Pirjanian. Multiple objective behavior-based control, 2000.

Robert F. Raleigh. Habitat suitability information: Rainbow trout, 1984. URL `http://www.nwrc.usgs.gov/wdb/pub/hsi/hsi-060.pdf`.

Regeringen. Danmark 2020, http://www.stm.dk/publikationer/arbprog_10/danmark%202020_viden_vaeks
2010.

Salgs Ingeniør for Lykkegaard AS Svend Christensen, Eksport Manager og Lars Bjerregaard. Interview med lykkegaard as om dambrug og pumper, 2012.

Wikipedia. Software framework, 2012a. URL `http://en.wikipedia.org/wiki/Software_framework`.

Wikipedia. Carbon dioxide, 2012b. URL `http://en.wikipedia.org/wiki/Carbon_dioxide`.

Wikipedia. Unified process, 2012c. URL `http://en.wikipedia.org/wiki/Unified_Process`.

Patrick Steyaert Wim Codenie, Koen De Hondt and Arlette Vercammen. From custom applications to domain-specific frameworks, 1997.

Michael Wooldridge. An introduction to multiagent systems, 2009.

A Rapport redaktør

Følgende er en liste over de forskellige afsnit i rapporten, med en angivelse af hvem der har skrevet hvilke afsnit. Arbejdet med projektet har foregået i samarbejde, med løbende feedback til afsnit, og især indenfor Unified Process øvelserne er der blevet arbejdet i fællesskab. Nedenstående liste er en liste over hvem der har skrevet de enkelte afsnit ind i rapporten.

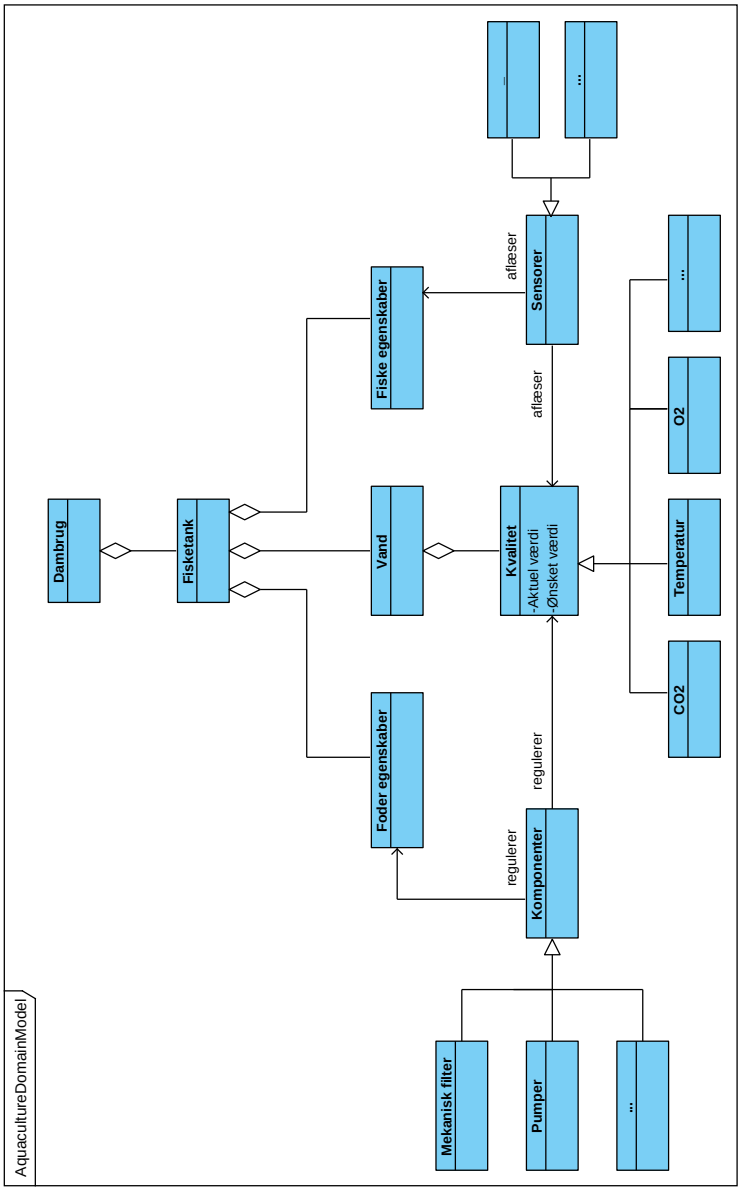
Afsnit:	**Skrevet af:**
Afsnit 1-5:	Martin Dissing-Hansen, Morten Olsen og Peter Nellemann
Afsnit 6.1	Morten Olsen
Afsnit 6.2	Peter Nellemann
Afsnit 6.3	Morten Olsen
Afsnit 7.1:	Peter Nellemann
Afsnit 7.2.1-10:	Peter Nellemann og Morten Olsen
Afsnit 7.2.11:	Peter Nellemann
Afsnit 7.2.12:	Martin Dissing-Hansen
Afsnit 7.3:	Martin Dissing-Hansen
Afsnit 8.1:	Morten Olsen
Afsnit 8.2.1-2:	Peter Nellemann
Afsnit 8.2.3:	Morten Olsen
Afsnit 8.2.4:	Martin Dissing-Hansen, Morten Olsen og Peter Nellemann
Afsnit 8.2.5:	Martin Dissing-Hansen
Afsnit 8.3.1-8.3.4:	Peter Nellemann
Afsnit 8.3.5:	Martin Dissing-Hansen, Morten Olsen og Peter Nellemann
Afsnit 8.4.1-8.4.2:	Morten Olsen
Afsnit 8.4.3:	Martin Dissing-Hansen, Morten Olsen og Peter Nellemann
Afsnit 8.5.1-4	Martin Dissing-Hansen
Afsnit 8.5.5:	Martin Dissing-Hansen, Morten Olsen og Peter Nellemann
Afsnit 9.1-9.2.9	Martin Dissing-Hansen
Afsnit 9.2.10:	Martin Dissing-Hansen og Morten Olsen
Afsnit 10.1-2:	Martin Dissing-Hansen
Afsnit 10.3-5:	Peter Nellemann
Afsnit 10.6-7:	Martin Dissing-Hansen
Afsnit 11:	Peter Nellemann
Afsnit 12-14:	Martin Dissing-Hansen, Morten Olsen og Peter Nellemann

B Program ansvarlig

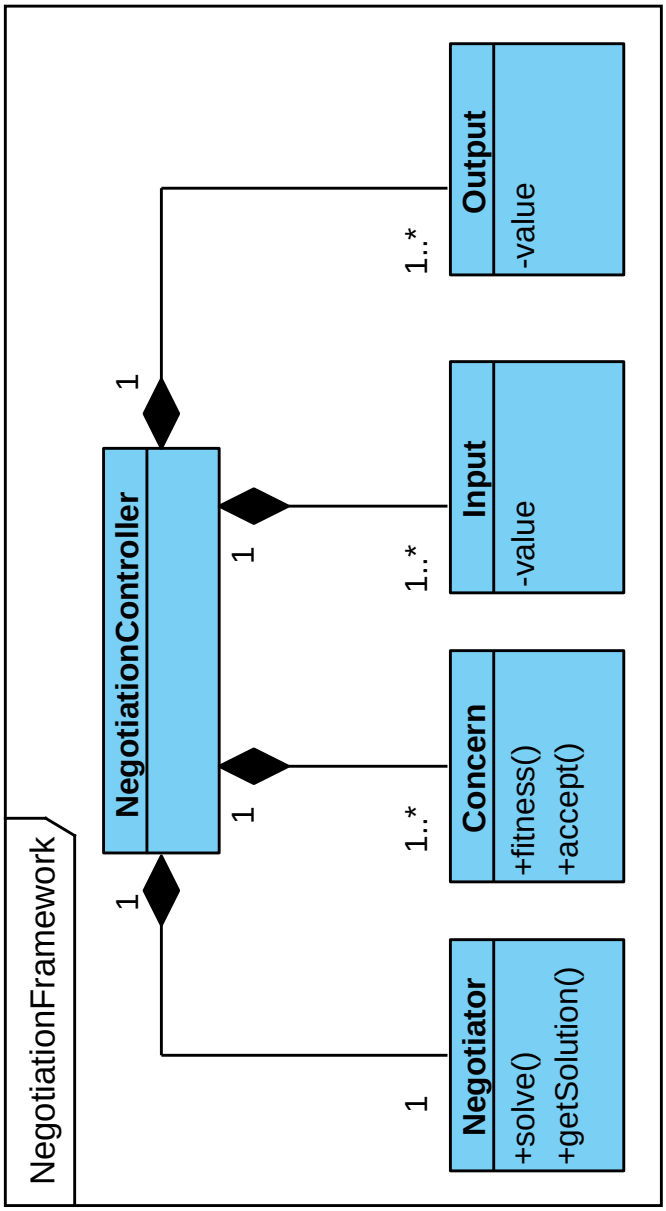
Følgende er en liste over hvem der har skrevet hvad i det udviklede program, generaliseret til hver pakke for at få et hurtigere overblik over fordelingen.

Klasser:	**Kodet af:**
Foundation-lagets klasser:	Martin Dissing-Hansen
Domain.Mediator pakkens klasser:	Martin Dissing-Hansen
Domain.Entity pakkens klasser:	Martin Dissing-Hansen
Control-lagets klasser:	Martin Dissing-Hansen
Concerns, Inputs og Outputs:	Peter Nellemann, Morten Olsen og Martin Dissing-Hansen
Presentation-laget (GUI)	Peter Nellemann og Martin Dissing-Hansen
Acquaintance-lagets klasser:	Martin Dissing-Hansen
Test pakkens klasser:	Martin Dissing-Hansen

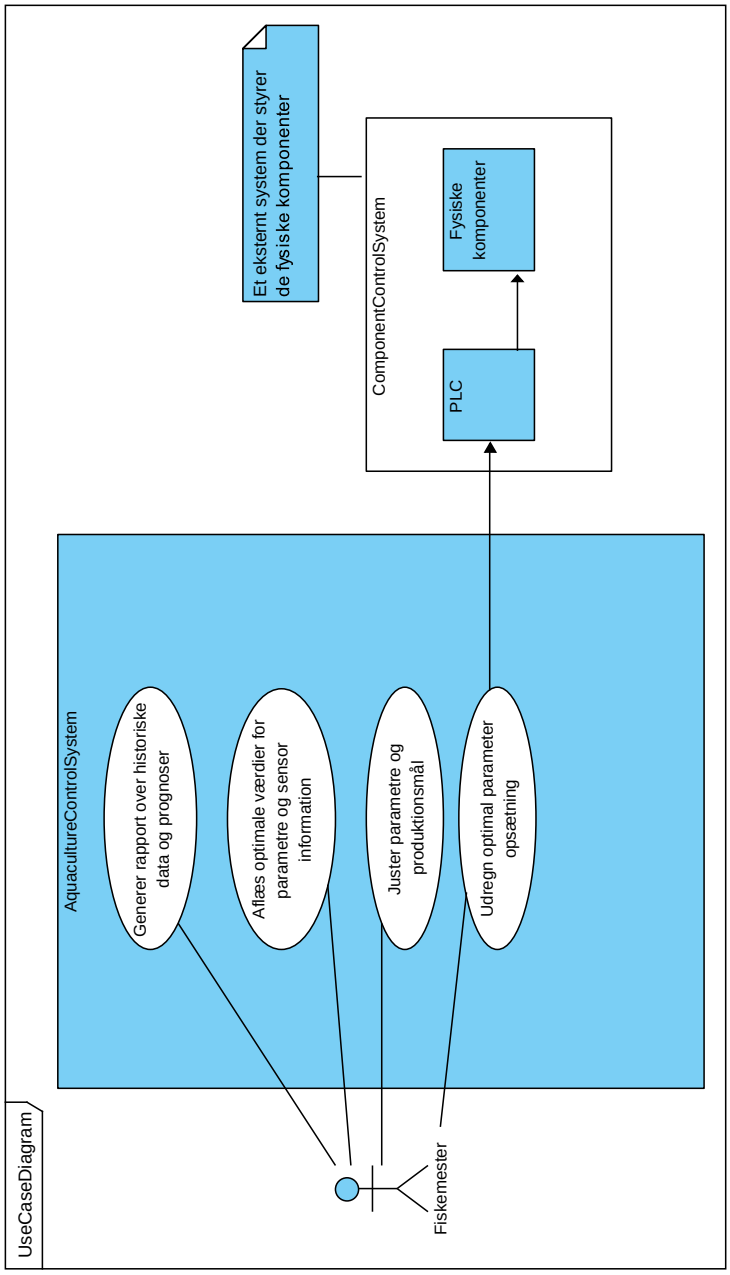
C Diagrammer



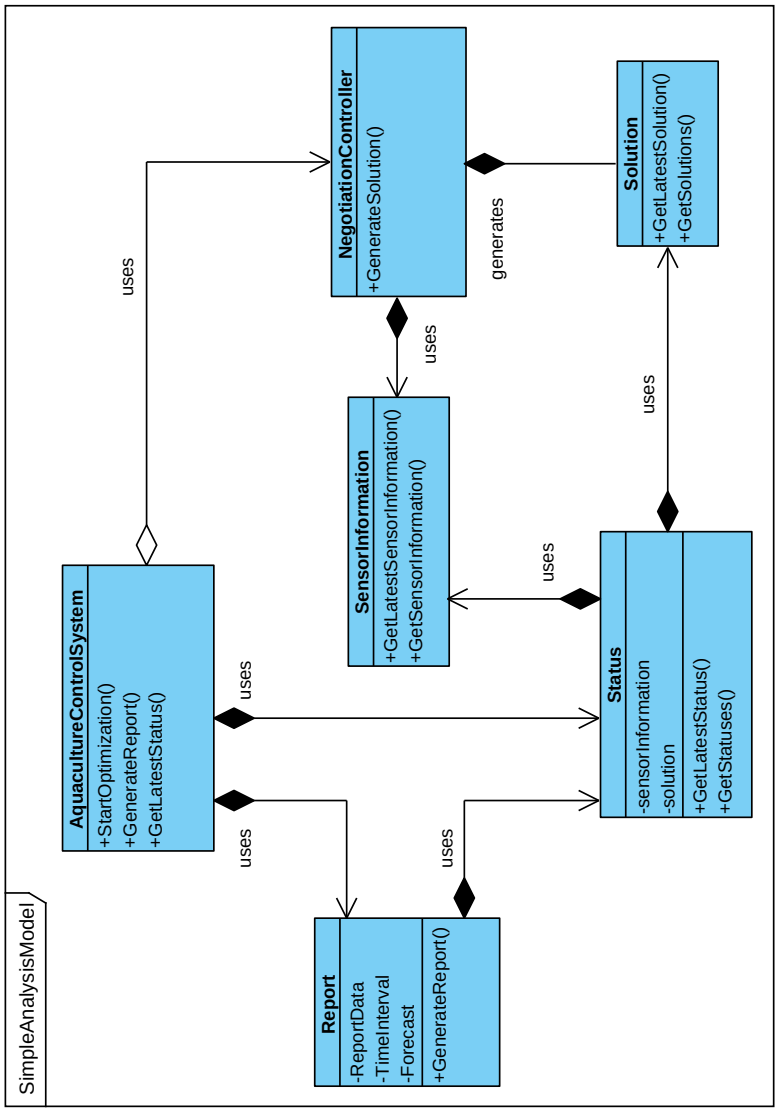
Figur 20: Domæne model



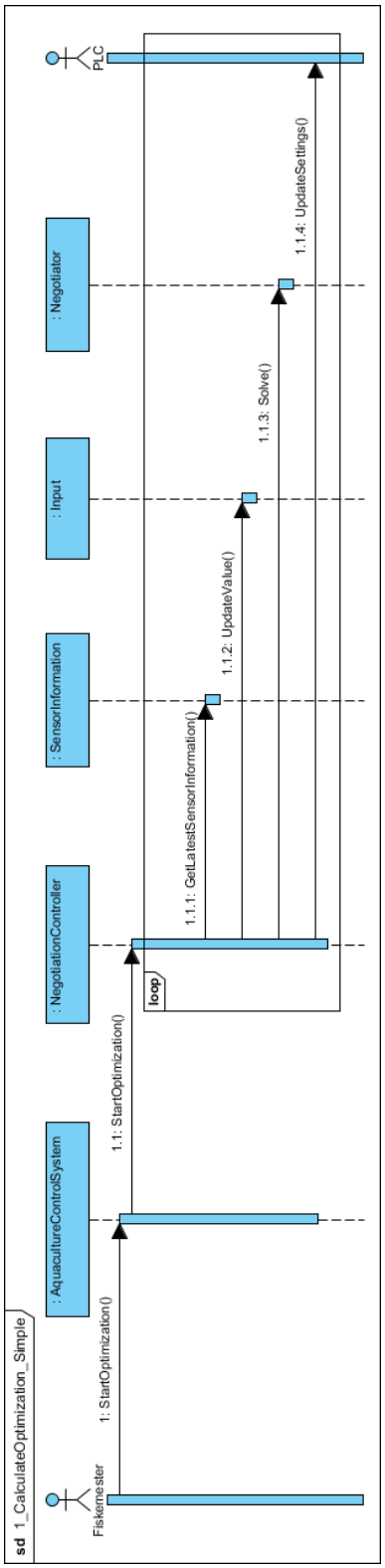
Figur 21: Model over framework



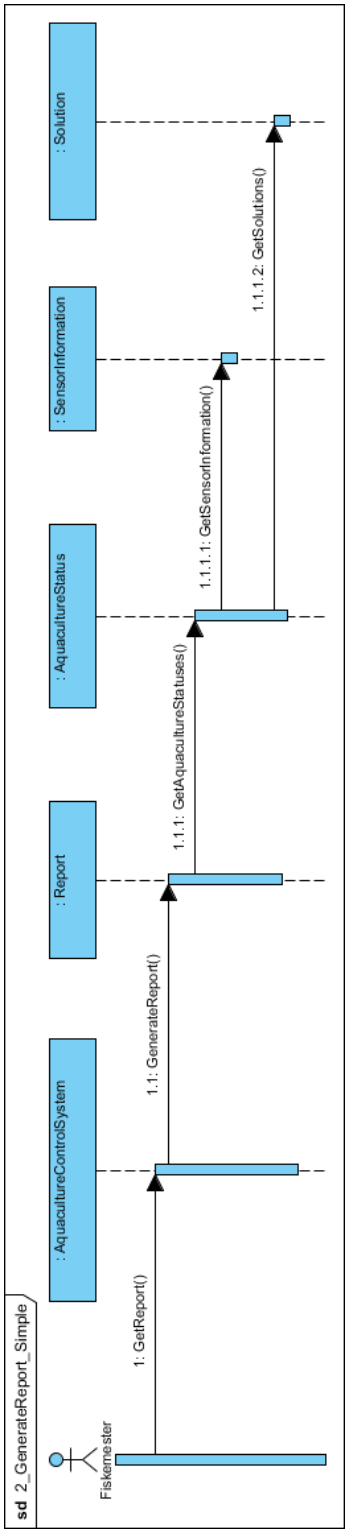
Figur 22: Brugsmønsterdiagram



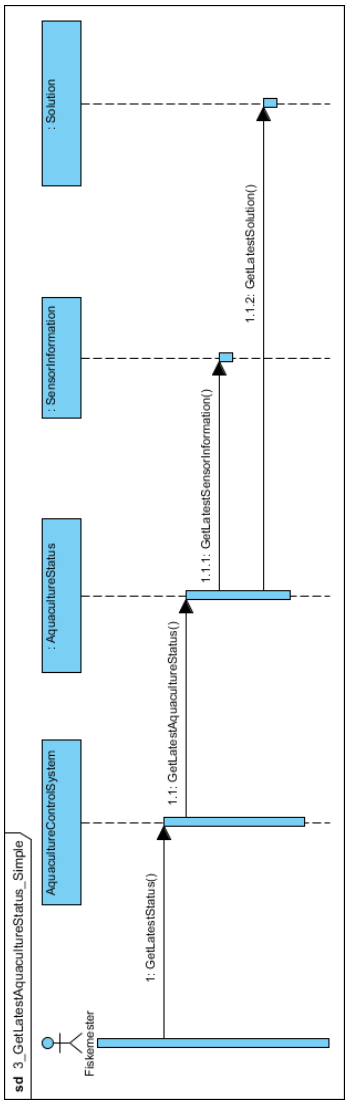
Figur 23: Analysemodel



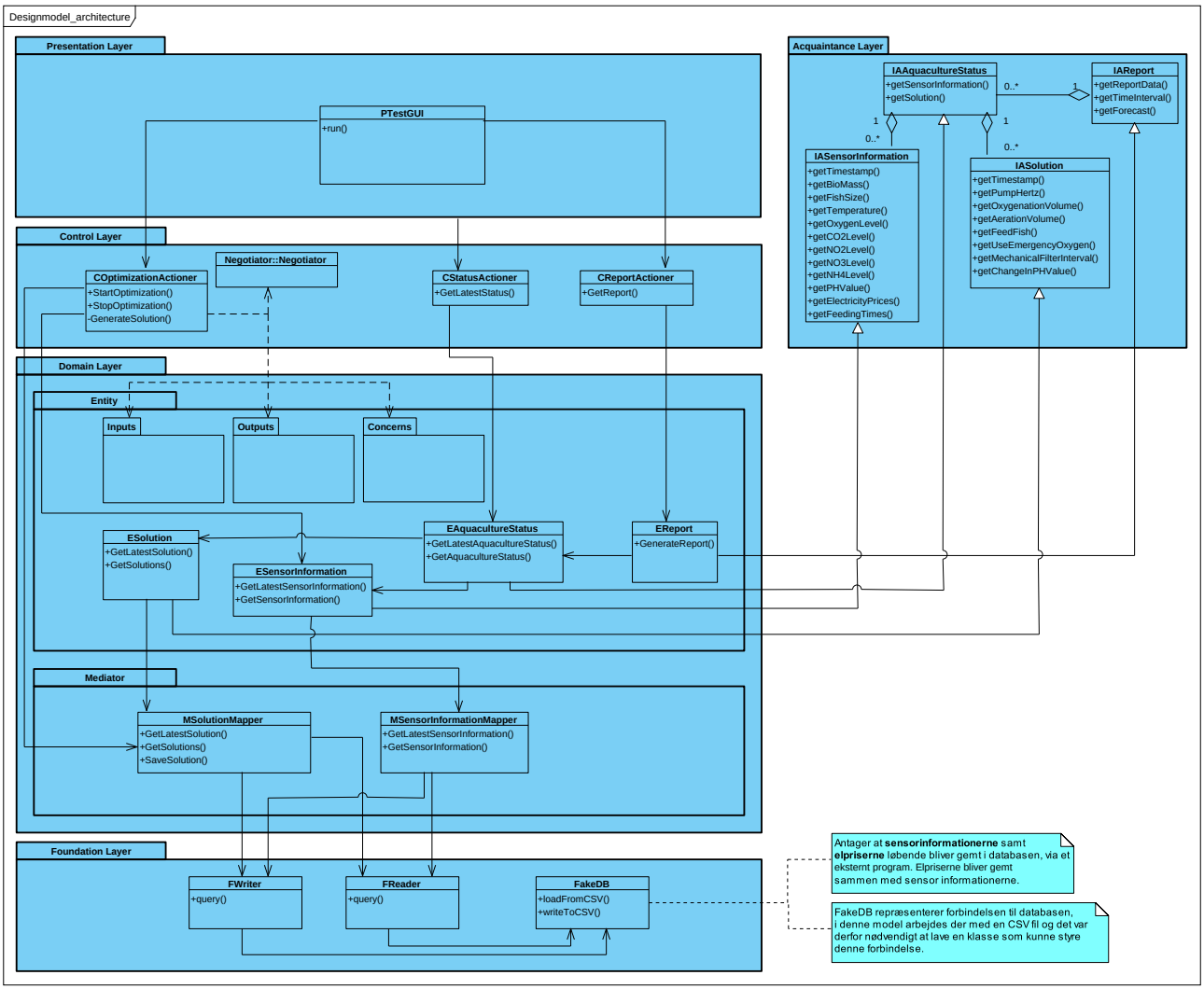
Figur 24: Simpelt sekvensdiagram for brugsmønster 1



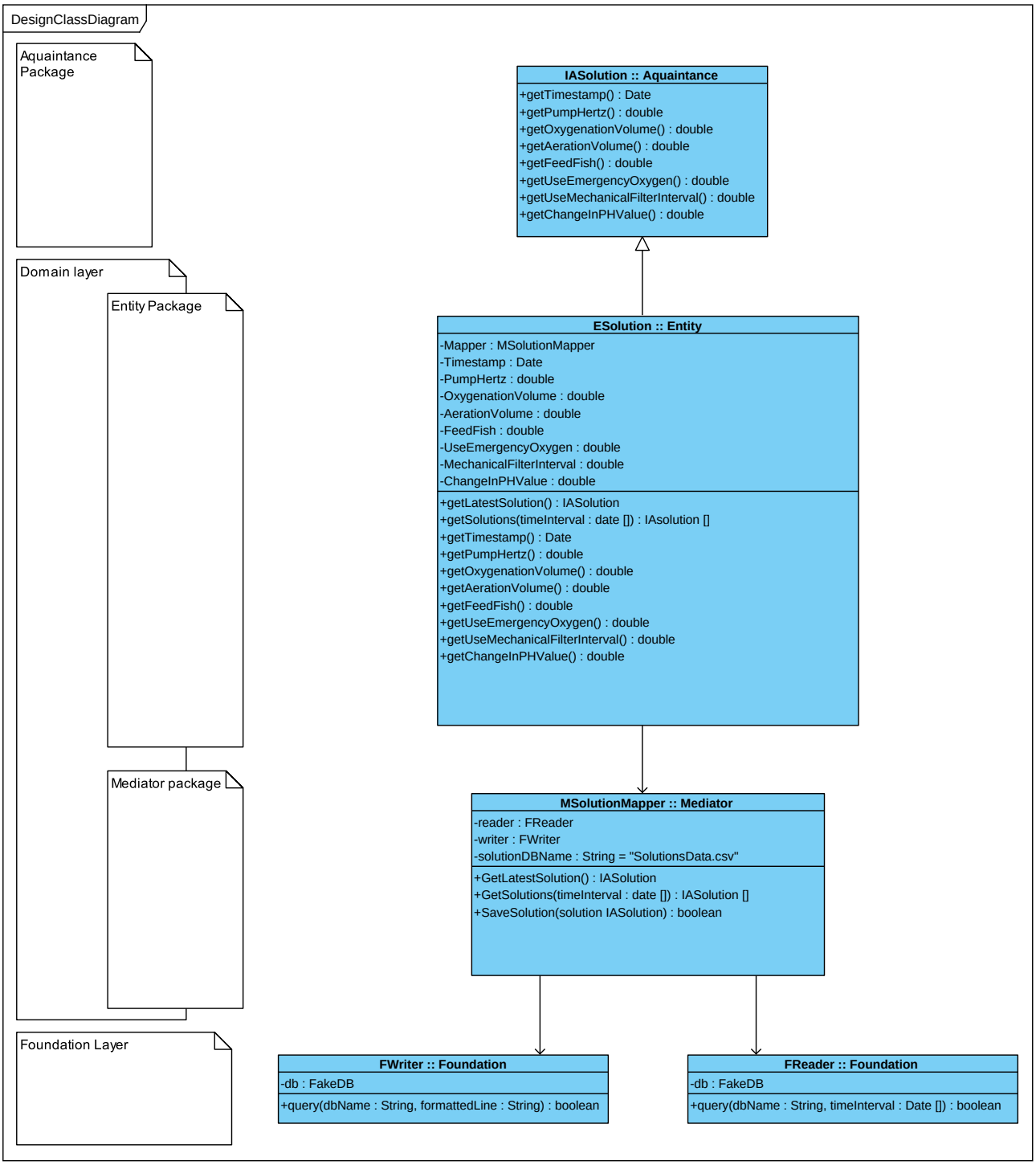
Figur 25: Simpelt sekvensdiagram for brugsmønster 2



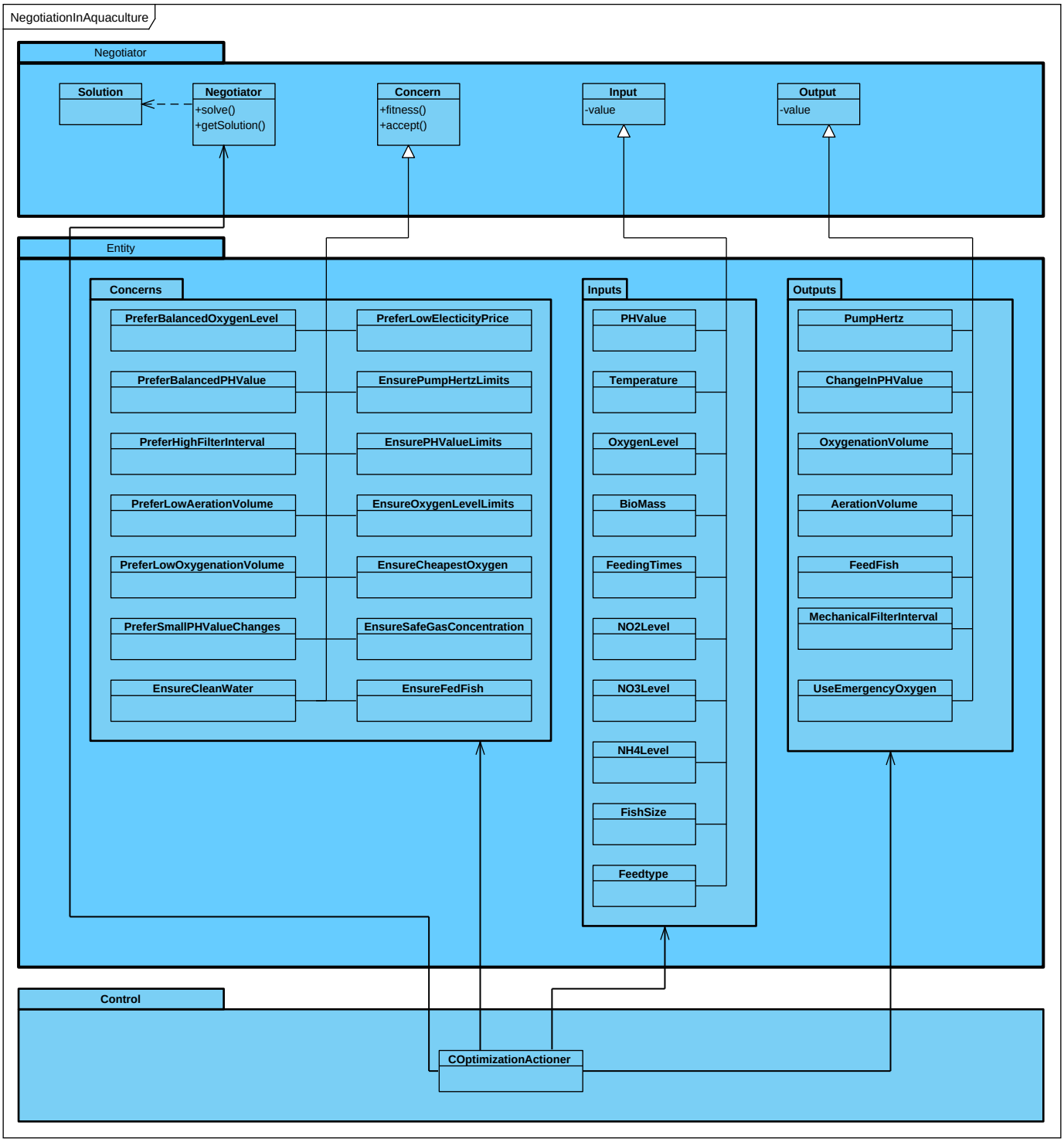
Figur 26: Simpelt sekvensdiagram for brugsmønster 3



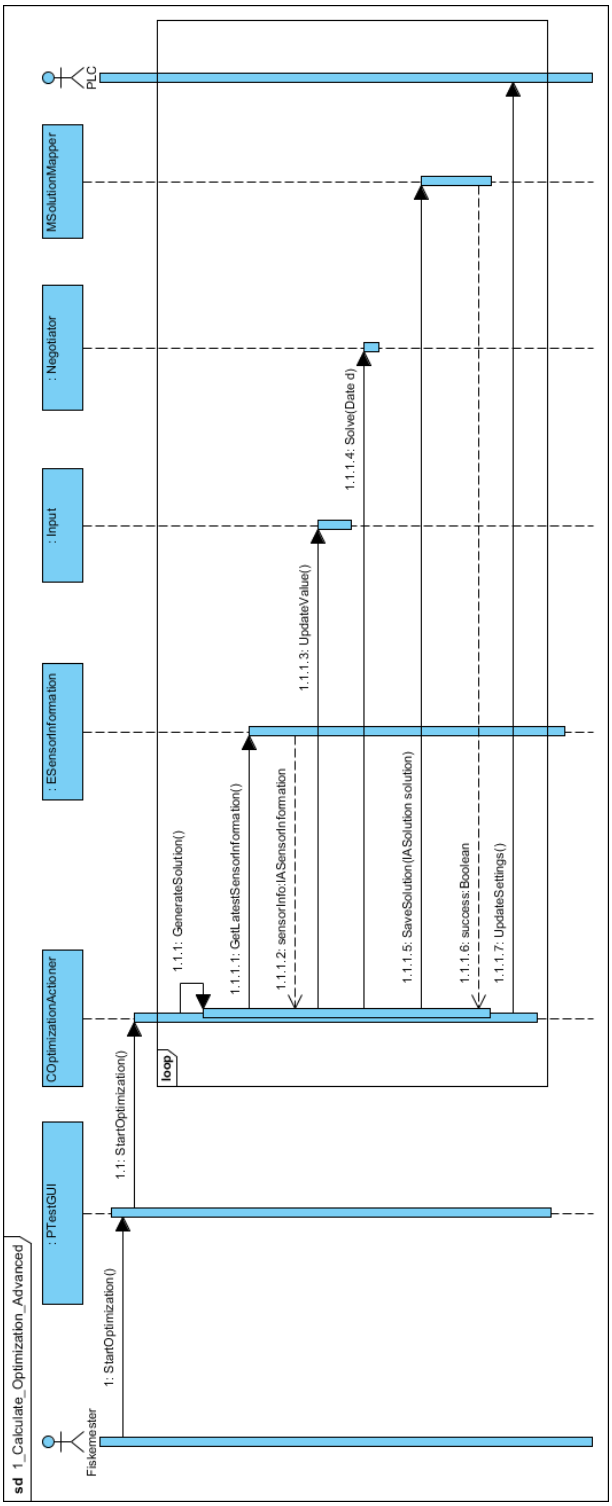
Figur 27: Design model, arkitektur



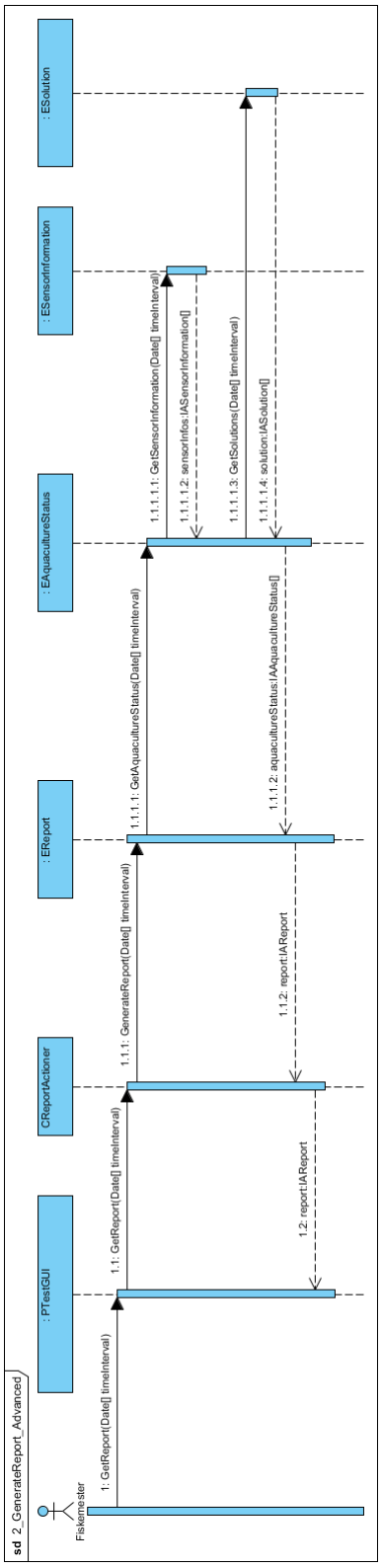
Figur 28: Design diagram, detaljeret



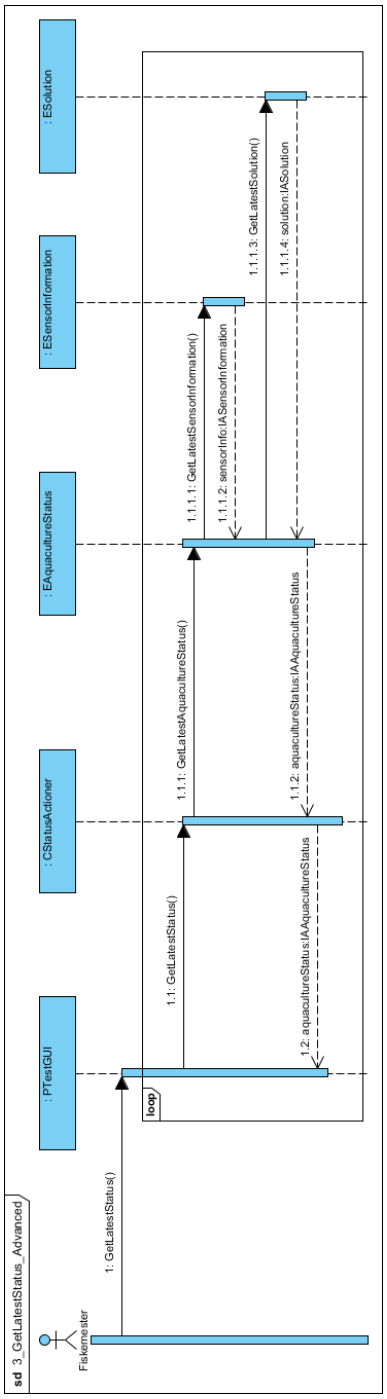
Figur 29: Multioptimering i dambrug



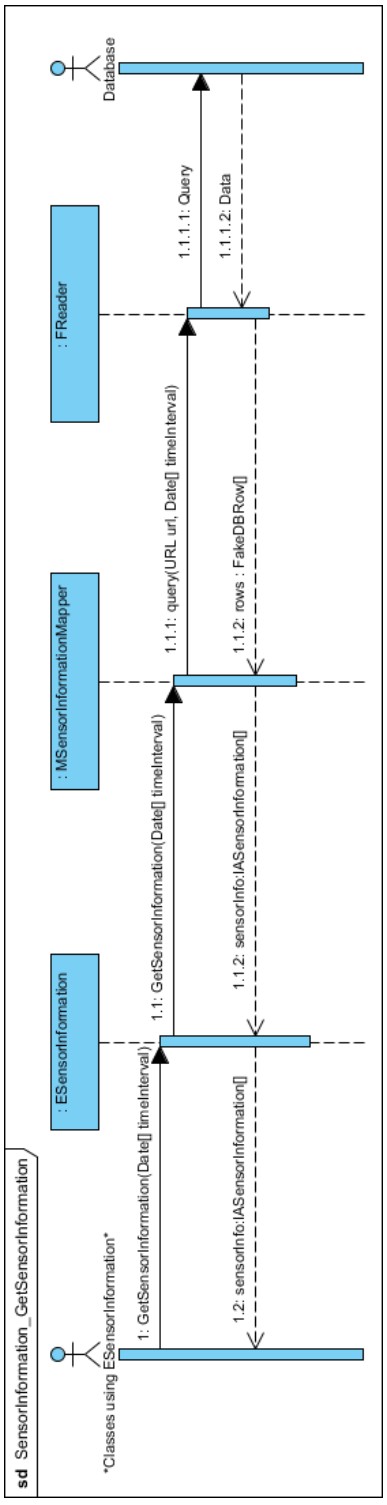
Figur 30: Avanceret sekvensdiagram for brugsmønster 1



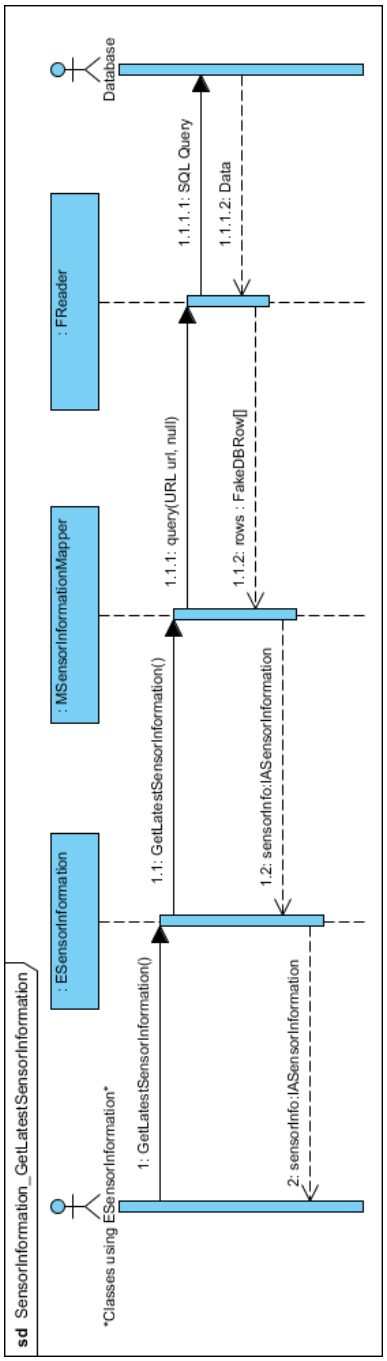
Figur 31: Avanceret sekvensdiagram for brugsmønster 2



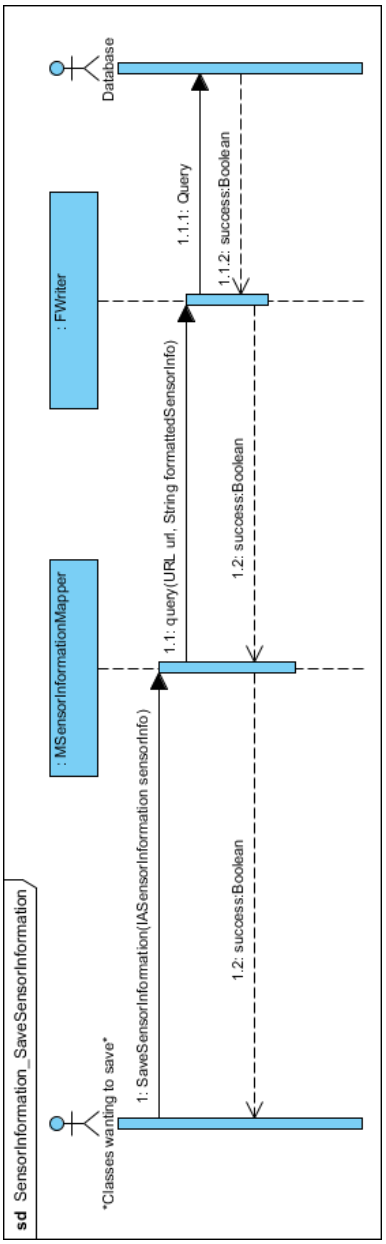
Figur 32: Avanceret sekvensdiagram for brugsmønster 3



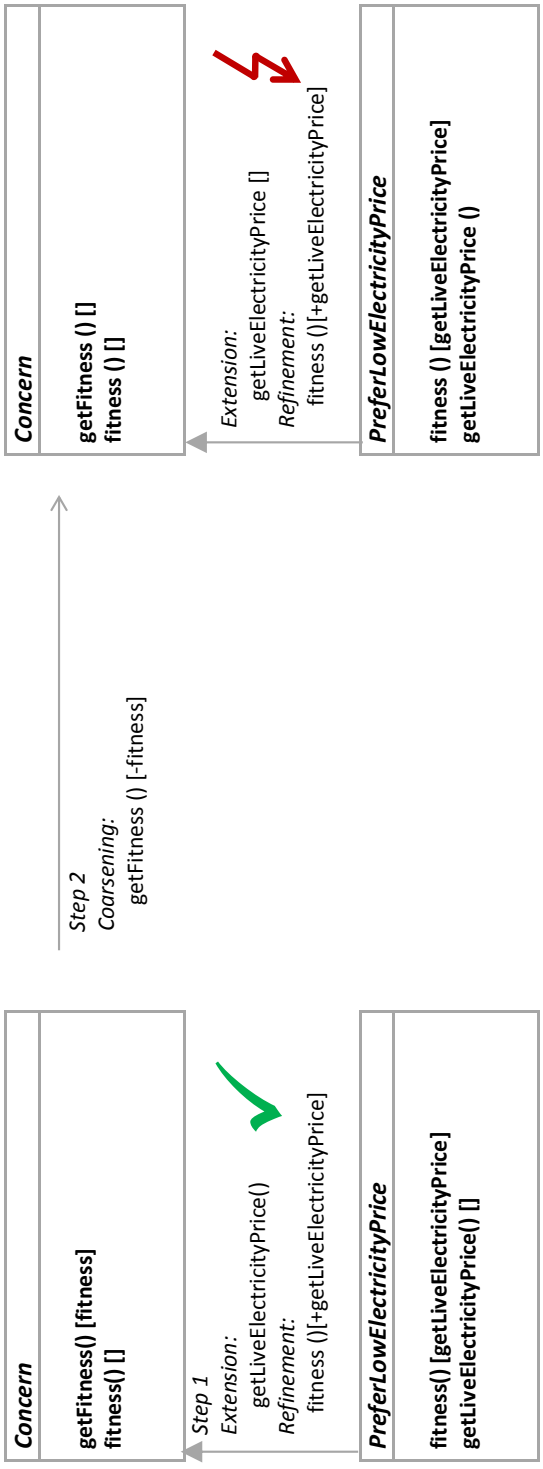
Figur 33: Sekvensdiagram for GetSensorInformation



Figur 34: Sekvensdiagram for GetLatestSensorInformation



Figur 35: Sekvensdiagram for SaveSensorInformation



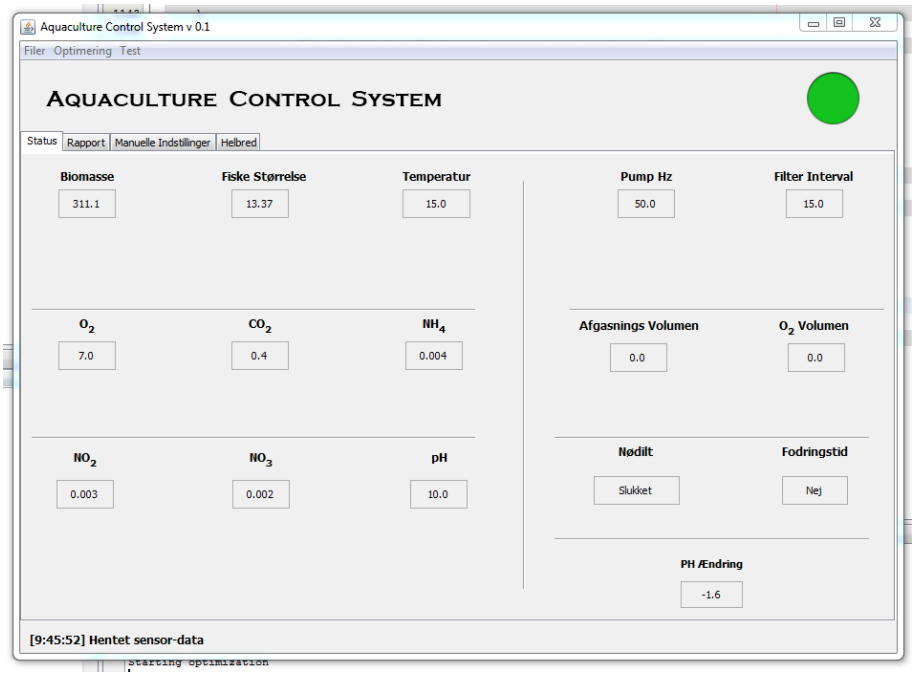
Figur 36: Single-class reuse kontrakt

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	InputID	Timestamp	PHValue	Temperature	OxygenLevel	BioMass	NO3Level	NO2Level	FishSize	CO2Level	NH4Level	ElectricityPrice	FeedingTimes	
2	2	23-05-2012 13:51	10.0	15.0	7.0	311.1	0.002	0.003	13.37	0.4	0.004	[1.0,3.0,1.0,2.0,4.(	[6:00,9:00,12:00,15:00,18:00]	
3														

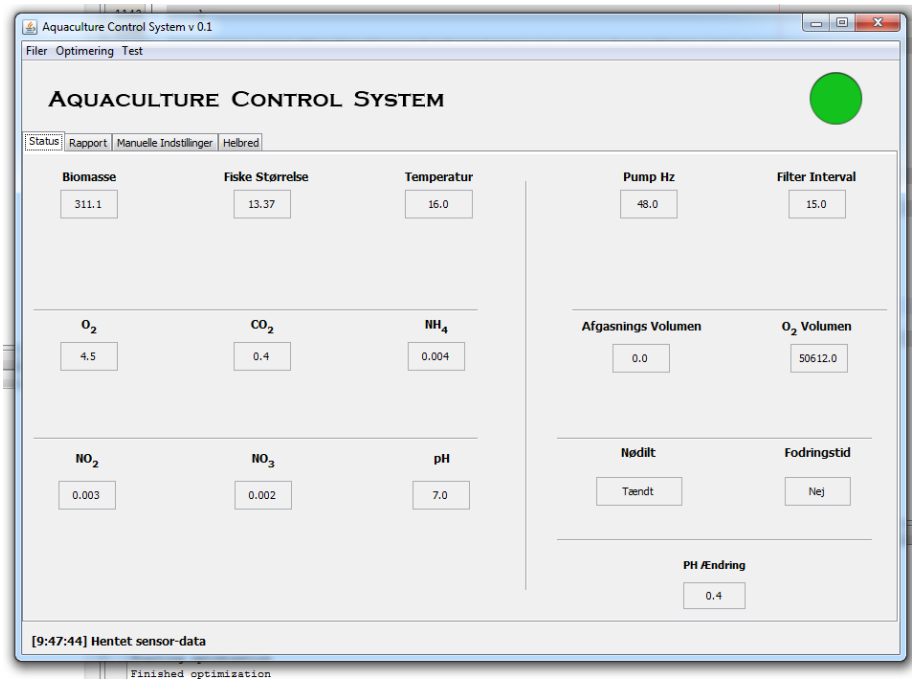
Figur 37: SensorInformationData.csv

	A	B	C	D	E	F	G	H	I
1	InputId	Timestamp	PumpHertz	OxygenationVolume	AerationVolume	FeedFish	UseEmergencyOxygen	MechanicalFilterInterval	ChangeInPHValue
2	1	21-05-2012 15:30	48.0	0.0	0.0	0.0	0.0	15.0	-3.56
3									

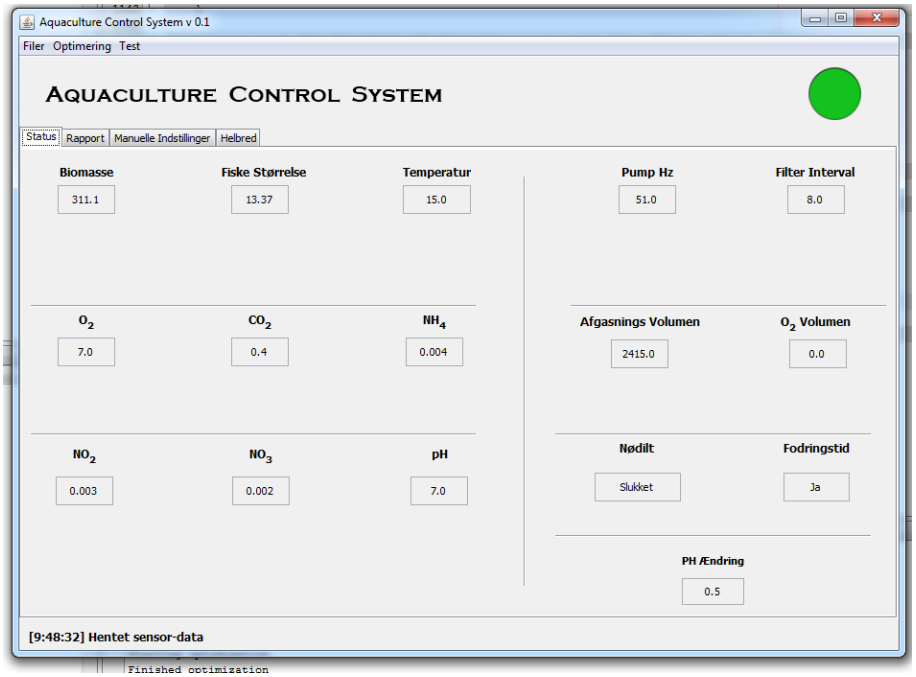
Figur 38: SolutionData.csv



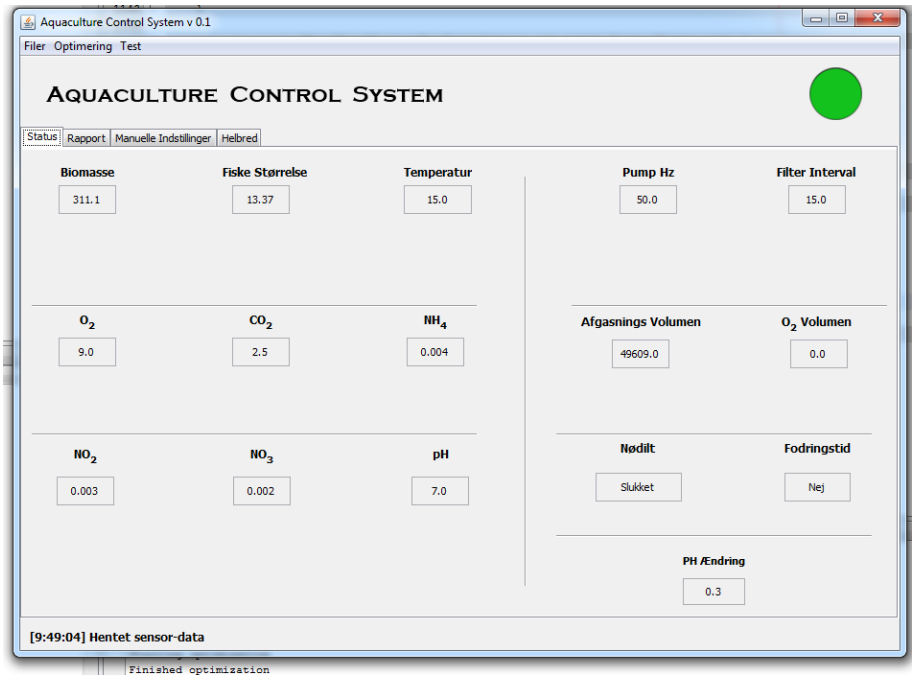
Figur 39: Test - pH værdi



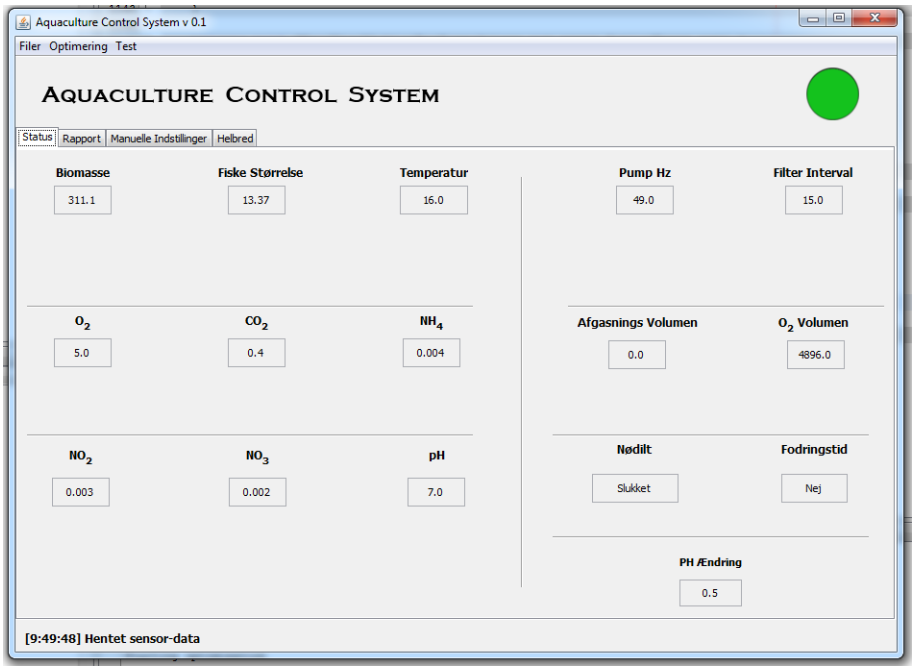
Figur 40: Test - Aktivering af nød-ilt



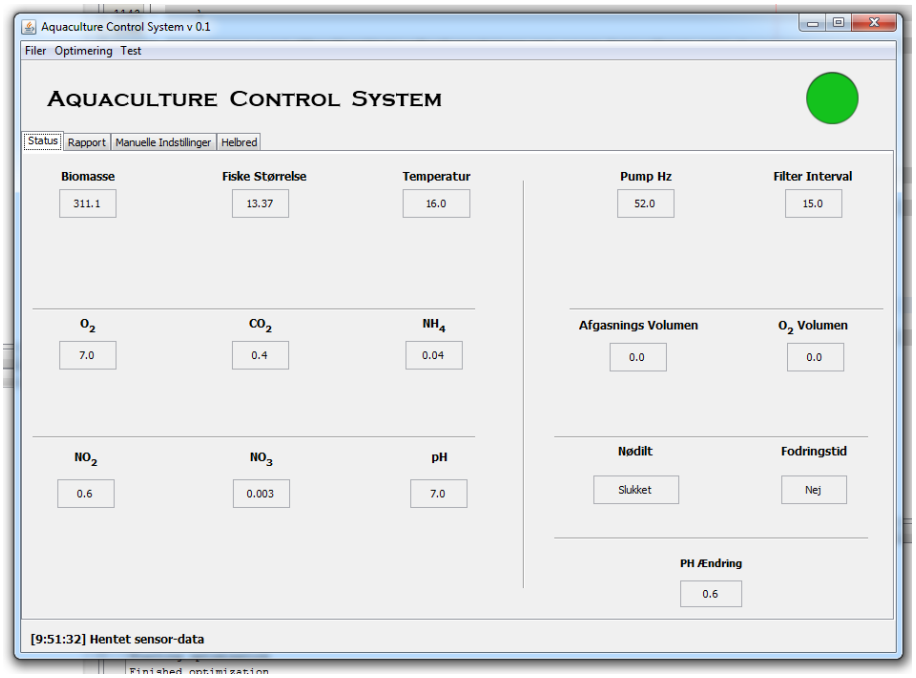
Figur 41: Test - Tid til fodring



Figur 42: Test - Højt CO_2 niveau



Figur 43: Test - Iltning



Figur 44: Test - Pump Hz