Software Engineering
Via University College
SoftwareEngineering
Banegårdsgade 4
8700 Horsens

Titel:

2. Semesterprojekt - Holdtræningssystem

ECTS:

10 ECTS

Semester:

2. Semester

Projektperiode:

4/2/2026 - 27/5/2026

Projektgruppe:

SEP2 Gruppe 2

Studienummer	Navn	Studieretning
362108	Devin Paul Jaworek	SW
361830	Victor Høggaard	SW
362962	Kasra Hojjatifar	SW
362181	Mussa Mohammad	SW
362947	Hussein Abed	SW

Vejledere:

Mona Wendel Andersen, Steffen Vissing Andersen

Sidetal: 62 Antal bilag: 14

1. Resume

Dette projekt omhandler udviklingen af et fitnesscenter-system til administration af holdtræning i moderne fitnesscentre. Formålet med projektet var at undersøge de organisatoriske og teknologiske udfordringer, der opstår ved administration af holdtræning, samt at udvikle et system, der kan understøtte medlemmer, instruktører og ejere i håndteringen af aktiviteter og ressourcer.

Projektet tager udgangspunkt i de udfordringer, der kan opstå i forbindelse med planlægning, koordinering og kommunikation mellem de forskellige aktører i et fitnesscenter. Derudover undersøges behovet for fleksible systemer, der kan håndtere ændringer i aktiviteter, medlemsstruktur, lokaler og udstyr.

På baggrund af analysen blev der udviklet et klient-server system med JavaFX som brugergrænseflade og en Postgres database. Systemet giver medlemmer mulighed for at se aktiviteter, tilmelde og afmelde sig hold samt blive tilføjet til ventelister. Instruktører kan oprette, redigere og aflyse aktiviteter, mens ejeren kan administrere instruktører, lokaler og udstyr.

I systemets design blev der anvendt MVVM-arkitektur samt design patterns som Observer, Proxy, Adapter, State og DAO. Derudover blev systemet udviklet med fokus på principper fra SOLID og GRASP for at skabe en mere struktureret og vedligeholdelsesvenlig løsning.

Projektet blev udviklet ved brug af Scrum i kombination med Unified Process, hvor analyse, design, implementering og test blev videreudviklet gennem flere iterationer og sprints. Under projektforløbet blev der løbende arbejdet med use-cases, domænemodel, systemsekvensdiagrammer, aktivitetsdiagrammer og test af systemets funktionalitet.

Projektet resulterede i et fungerende fitnesscenter-system, der understøtter administration af holdtræning samt de centrale behov hos medlemmer, instruktører og ejere.

Indhold

2. Indledning

2.1. Problemfelt

Fitness er i bred forstand sundhed. Her er det specifikt træningsformer, der er målrettet sundhedsgavn. Disse træningsformer indebærer alt fra 70'ernes bodybuilding til 2010'ernes crossfit. Derfor skal der være et sted at dyrke disse træningsformer på – og fitnesscentre er det sted, hvor det foregår (Salminen, 2025a).

Over tid er de gamle motionscentre blevet til de moderne fitnesscentre, hvor det kommercielle og teknologiske fylder mere end nogensinde. Dog omhandler de centrale værdier blandt fitnesscentre stadig sundhed og sociale aspekter (Salminen, 2025b).

Det er generelt videnskabeligt anerkendt, at træning og daglig motion giver fordele til sundheden både mentalt og fysisk. Regulær træning kan have massive fordele for mental og social sundhed. Regulær træning i fitnesscentre giver også mulighed, for at socialisere med andre folk. Dette sker specielt på træningshold (Santini, 2023).

Holdtræning kan give en følelse af fællesskab, socialstøtte og selvtillid. Når en person deltager i et fitnesshold, bliver de en del af et fællesskab, der deler de samme mål og udfordringer. Et fællesskab som dette kan reducere ens følelse af isolation og ensomhed, samtidigt med at det styrker ens motivation til at møde op til træningen regelmæssigt. På baggrund af dette er holdaktiviteter blevet en central del af fitnesscentre. Ved kombination af styrketræning og socialt fællesskab bliver holdtræning til et sted, hvor fysisk og mentalt trivsel bliver understøttet (MOMENTUM OP, 2025).

Træningscentre er blevet mere moderne igennem tiden, hvor der har været teknologiske fremskridt bl.a. i form af komplicerede medlemssystemer, mangelfulde bookingsystemer og systemintegration i maskiner. Disse fremskridt fortsætter og fitnesscentrene er derfor meget dynamiske. Det kan være alt fra nyt udstyr til nye medlemmer. Af den grund kan det være en udfordring for de nuværende systemer til fitnesscentre at holde sig adaptive og afspejle virkeligheden. Hertil kan det være et problem for medlemmer at holde styr på mulige træningshold, hvilke lokaler det er i, hvor mange der er tilmeldt og mere til. Alt i alt er der problemer i organiseringen af holdtræninger i fitnesscentre. Dette inkluderer alle parter; fitnesscenter-ejere, instruktører og medlemmer. Desuden forventes det i moderne samfund, at registrering af konti og betaling for services sker automatisk (BOX12, 2025; Johansen, 2025).

2.2. Projektets formål

Formålet med projektet er at analysere de organisatoriske og teknologiske udfordringer, der opstår ved administration af holdtræning i moderne fitnesscentre, samt at udvikle et system, der kan understøtte medlemmer, instruktører og ejere i håndtering af holdaktiviteter.

Systemet skal skabe bedre overblik over aktiviteter, tilmeldinger, lokaler og udstyr samt forbedre kommunikationen mellem de involverede aktører. Derudover skal systemet understøtte administration af holdtræning gennem funktioner som booking, oprettelse af aktiviteter og håndtering af ressourcer.

2.3. Problemformulering

Projektet undersøger følgende problemformulering:

Hvordan kan de organisatoriske og teknologiske udfordringer ved administration af holdtræning i moderne fitnesscentre forstås og analyseres i relation til de involverede aktørers behov? For at besvare problemformuleringen arbejdes der desuden med følgende underspørgsmål:

- Hvilke organisatoriske udfordringer opstår der, i forbindelse med planlægning og koordinering af holdtræning?
- Hvordan påvirker eksisterende medlemssystemer og bookingsystemer kommunikationen mellem medlemmer, instruktører og ejere?
- Hvilke behov og forventninger har de forskellige aktører til administration af holdtræning?
- Hvordan påvirker fitnesscentres dynamiske udvikling kravene til administrative systemer?
- Hvordan kan systemet understøtte fleksibilitet, overblik og effektiv organisering?

3. Analyse

3.1. Introduktion

Der er lavet en analyse baseret på problemfeltet og problemformuleringen fra afsnit 2. Her er der udarbejdet funktionelle og ikke-funktionelle krav med udgangspunkt i systemets overordnede mål og de udfordringer, der blev identificeret i forbindelse med administration af holdtræning i fitnesscentre.

På baggrund af kravene er der udarbejdet use-cases og et use-case diagram, som beskriver systemets, funktionalitet og interaktionen mellem systemets aktører. Derudover er der udarbejdet en domænemodel, som viser systemets vigtigste objekter og relationer, der er også lavet en systemsekvensdiagram.

Analysen danner grundlag for systemets videre design og implementering.

3.2. Krav

Som beskrevet i analyseintroduktionen i afsnit 3.1 er der udarbejdet krav på baggrund af problemfeltet og problemformuleringen fra afsnit 1. Kravene er opdelt i funktionelle og ikke-funktionelle krav.

3.2.1. Funktionelle krav

De funktionelle krav beskriver de funktioner, systemet skal understøtte for medlemmer, instruktører og ejere. Der er særligt fokus på administration af holdtræning, herunder tilmelding til aktiviteter, administration af hold, håndtering af lokaler og administration af udstyr. De funktionelle krav er desuden udarbejdet med udgangspunkt i SMART- og INVEST-kriterierne for at sikre, at kravene er tydelige, realistiske og mulige at implementere og teste gennem projektforløbet.

De funktionelle krav er opstillet som User Stories ud fra formatet: "Som [rolle] ønsker jeg [funktion], for at [formål]". Dette gør det tydeligt, hvilken aktør kravet omhandler, hvilken funktion systemet skal understøtte, samt hvorfor funktionen er relevant.

SMART kan blandt andet ses i krav [1]: "Som medlem ønsker jeg at kunne tilmelde mig et hold, for at kunne deltage på et hold". Kravet er specifikt, da det præcist beskriver, hvem der udfører handlingen, og hvad handlingen består af. Samtidig er kravet målbart og testbart, da det er tydeligt, hvornår funktionaliteten virker korrekt , nemlig når et medlem kan tilmelde sig et hold gennem systemet. De øvrige krav er formuleret på samme måde med konkrete handlinger og tydelige aktører.

INVEST kan ses ved, at kravene er opdelt i mindre og selvstændige funktioner. Eksempelvis er login [8], registrering [9], holdtilmelding [1] og afmelding [10] opdelt i separate krav frem for at være samlet i en stor funktion. Dette gør kravene lettere at implementere og teste individuelt. Samtidig giver hvert krav værdi for en bestemt aktør i systemet, da funktionerne direkte understøtter behov hos enten medlemmer, instruktører eller ejere.

Funktionelle krav:

[1] Som medlem ønsker jeg at kunne tilmelde mig et hold, for at kunne deltage på et hold.

[2] Som medlem ønsker jeg at kunne se hold ved (titel, tidspunkt, tid, instruktør og status) som jeg kan melde mig til, for at have overblik over hvilke mulige hold der er.

[3] Som medlem ønsker jeg at kunne se detaljer om et hold (titel, beskrivelse, tid, tidspunkt, lokale og instruktør), for at kunne give mig et detaljeret indblik i holdet.

[7] Som medlem ønsker jeg at kunne se medlemmer ved (fornavn og efternavn) som deltager på hold, for at holde overblik over hvem og hvor mange der kommer.

[4] Som instruktør ønsker jeg at kunne oprette et hold ved (titel, beskrivelse, tid, tidspunkt, lokale, udstyr og antal pladser), for at gøre holdet tilgængeligt.

[5] Som instruktør ønsker jeg at kunne redigere et hold ved (titel, beskrivelse, tid, tidspunkt, lokale, antal pladser og udstyr) med lokale og udstyr, for at kunne tilpasse ændringer til holdet.

[6] Som instruktør ønsker jeg at kunne aflyse et hold, for at jeg har mulighed for at annullere, hvis noget forhindrer mig i at afholde aktiviteten.

[8] Som medlem ønsker jeg at kunne tilgå hold via brugernavn og adgangskode for at se de hold som jeg har tilmeldt mig.

[9] Som medlem ønsker jeg at kunne registrere mig selv ved (brugernavn, fornavn, efternavn og adgangskode), for at kunne anvende aktiviteter i fitnesscenteret.

[10] Som medlem ønsker jeg at kunne afmelde mig fra et hold, for at give plads til andre medlemmer, hvis jeg ikke længere kan.

[11] Som ejer ønsker jeg at kunne se hvilke træningsrum mit fitnesscenter har ved lokale-id, for at danne mig et overblik over træningsrum.

[12] Som ejer ønsker jeg at kunne se hvilket udstyr mit fitnesscenter har ved (navn og antal), for at danne mig et overblik over udstyr.

[13] Som ejer ønsker jeg at kunne se hvilke instruktører mit fitnesscenter har ved (fornavn og efternavn), for at danne mig et overblik over instruktører.

[14] Som ejer ønsker jeg at kunne oprette instruktører ved hjælp af en liste over alle medlemmer, for at kunne tilbyde aktiviteter.

[15] Som ejer ønsker jeg at kunne oprette og redigere træningsrum ved lokale-id, for at lokaler kan anvendes til hold.

[16] Som ejer ønsker jeg at kunne oprette og redigere udstyr ved (navn og antal), for at udstyr kan anvendes til hold.

[17] Som ejer ønsker jeg at kunne slette træningsrum, for at fjerne rum, der ikke kan anvendes længere.

[18] Som ejer ønsker jeg at kunne slette udstyr, for at fjerne udstyr, der ikke længere er i centeret.

[19] Som ejer ønsker jeg at kunne slette instkruktører, for at fjerne instruktører, der ikke længere arbejder i centeret.

[20] Som instruktør ønsker jeg ikke at kunne oprette et hold med et lokale og tidspunkt, der allerede er oprettet, for at undgå dobbeltbookinger.

[21] Som medlem ønsker jeg at kunne melde mig på ventelisten ved på et fuldt booket hold, for at kunne være med, hvis nogen melder fra.

3.2.2. Ikke Funktionelle krav

De ikke-funktionelle krav beskriver systemets tekniske struktur og kvalitetskrav. Kravene omhandler blandt andet systemets arkitektur, brugergrænseflade, datahåndtering og vedligeholdelse.

Kravene er samtidig vurderet ud fra FURPS+-modellen for at sikre, at systemet ikke kun understøtter funktionalitet, men også brugervenlighed, pålidelighed, ydelse og vedligeholdelse.

[22] Der skal anvendes en database for at hylde styr på medlemmer, hold, instruktører, udstyr, ejere og lokaler.

[23] Der skal anvendes JavaFX til en flere-vinduet GUI.

[24] Systemet skal være et multitrådet klient-server system.

[25] Der skal anvendes følgende designmønstre: MVVM, observer, singleton / multiton, state.

Krav [22] om anvendelse af en database understøtter reliability, da data om medlemmer, aktiviteter og instruktører skal lagres stabilt og kunne genbruges mellem sessioner uden tab af information. Krav [23] om anvendelse af JavaFX understøtter usability, da systemet skal have en overskuelig og brugervenlig brugergrænseflade med flere vinduer til de forskellige brugerroller.

Krav [24] om et multitrådet klient-server system understøtter performance, da flere klienter skal kunne anvende systemet samtidigt uden at blokere hinanden. Dette er særligt relevant i et fitnesssystem, hvor medlemmer, instruktører og ejere kan være aktive i systemet på samme tid.

Krav [25] om anvendelse af designmønstre som MVVM, Observer, Singleton og State understøtter supportability, da systemet bliver mere struktureret og lettere at vedligeholde og videreudvikle. Designmønstrene bidrager samtidig til en mere fleksibel arkitektur, hvor ændringer i systemet lettere kan implementeres.

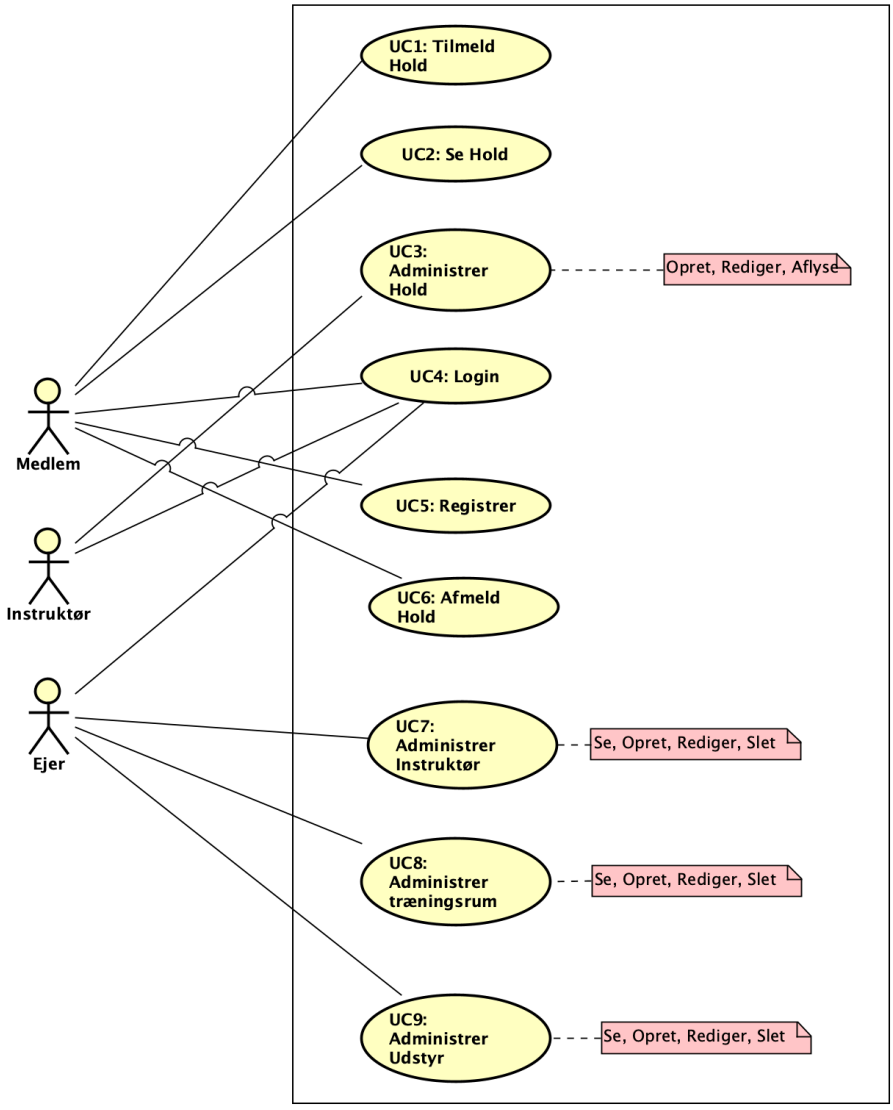
Flere af de ikke-funktionelle krav fungerer desuden som konkrete design- og implementeringskrav i FURPS+, da teknologier som JavaFX, database og klient-server arkitektur er centrale valg i systemets implementering

3.3. Use-cases

Der er lavet use-cases for holdtræningssystemet. Disse use-cases er baseret på de funktionelle krav beskrevet i afsnit 3.2. Use-cases beskriver systemets funktionalitet set fra en aktørs perspektiv og viser samspillet mellem aktører og system.

Systemet indeholder tre primære aktører: Medlem, Instruktør og Ejer.

Hver aktør har forskellige rettigheder og funktioner i systemet. Medlemmer fokuserer primært på tilmelding til aktiviteter, instruktører administrerer hold, og ejeren administrerer systemets ressourcer.



Figur 1: Use-case diagram

Figur 1 viser use-case diagrammet for holdtræningssystemet. Diagrammet viser systemets centrale use-cases samt relationerne mellem systemets aktører.

Medlemmet har adgang til følgende use-cases: Registrer, login, se hold, tilmeld hold og afmeld hold. Instruktøren har adgang til administration af aktiviteter, mens ejeren har adgang til administration af instruktører, lokaler og udstyr.

Use-case diagrammet blev anvendt til at skabe overblik over systemets funktionalitet samt sammenhængen mellem aktører og use-cases.

Use-case	UC1: Tilmeld Hold
Resume	Et medlem skal kunne tilmelde sig et hold. Et medlem skal også kunne tilmelde sig ventelisten, hvis et hold er fuldt.
Aktører	Medlem.
Forudsætninger	Medlem skal være logget ind.
Post-betingelser	Medlem er tilmeldt holdet eller dets venteliste.
Basis-sekvens	1. Medlem vælger et hold. 2. Medlem vælger “detaljer”. 3. Systemet viser mulighed, for at tilmelde sig på holdet. 4. Medlem vælger “tilmeld hold”. 5. Systemet validerer plads på holdet. 6. Systemet spørger om bekræftelse på tilmeldingen (bekræft/annuller).ALT[2] 7. Medlem vælger bekræft ALT[1]. 8. Holdet opdateres med tilføjelse af medlem på medlemsliste.
Alternative sekvenser	ALT[*] Processen kan anulleres på alle trin ALT[1] Medlem vælger “annuller” og systemet annullerer tilmeldingen. ALT[2] 1. Hvis holdet er fyldt bliver medlem spurgt om de vil tilføjes til ventelisten. 2. Medlem vælger bekræft. 3. Holdet opdateres med tilføjelse af medlem på venteliste.
Bemærkninger	Dækker krav: [1], [21].

Tabel 1: Use-case beskrivelse for UC1: Tilmeld Hold

Tabel 1 beskriver processen, hvor et medlem tilmelder sig et hold i systemet. Medlemmet vælger først et hold fra aktivitetslisten og kan derefter se detaljer om aktiviteten, herunder tidspunkt, instruktør og ledige pladser. Herefter kan medlemmet vælge at tilmelde sig holdet. Når medlemmet vælger “tilmeld hold”, kontrollerer systemet, om der er ledige pladser på aktiviteten. Hvis der er plads, tilføjes medlemmet til holdets deltagerliste, og hvis holdet er fyldt, får medlemmet mulighed for at blive tilføjet til en venteliste.

I projektet blev use-cases først anvendt i en kort form for at skabe et overordnet overblik over systemets funktionalitet og relationen mellem aktører og use-cases gennem use-

case diagrammet. Herefter blev udvalgte use-cases videreudviklet til et detaljeret format med basis- og alternative sekvenser. Dette blev gjort for at beskrive systemets adfærd mere præcist samt identificere forskellige scenarier og undtagelser i funktionaliteten.

Den detaljerede use-case for UC1: "Tilmeld Hold" blev efterfølgende anvendt som grundlag for aktivitetsdiagrammer, systemsekvensdiagrammer og implementeringen af funktionaliteten. Basis- og alternative sekvenser blev samtidig brugt i testfasen til udarbejdelse af systemets testcases. Use-casen er central for systemet, da tilmelding til aktiviteter er den vigtigste funktion i holdtræningssystemet.

Use-casen er central for systemet, da tilmelding til aktiviteter er den vigtigste funktion i holdtræningssystemet.

Ud over den centrale use-case blev der også udarbejdet use-cases for:

Use-case	**Dækkede krav**
UC1: Tilmeld Hold	Denne use-case dækker kravene 1, 21.
UC2: Se Hold	Denne use-case dækker kravene 2, 3.
UC3: Administrer Hold	Denne use-case dækker kravene 4, 5, 6, 20.
UC4: Login	Denne use-case dækker krav 8.
UC5: Registrer	Denne use-case dækker krav 9.
UC6: Afmeld Hold	Denne use-case dækker krav 10.
UC7: Administrer Instruktør	Denne use-case dækker kravene 13, 14, 19.
UC8: Administrer træningsrum	Denne use-case dækker kravene 11, 15, 17.
UC9: Administrer Udstyr	Denne use-case dækker kravene 12, 16, 18.

Tabel 2: Oversigt over dækkede use-cases og krav

Tabel 2 viser både de andre use-cases samt hvilke krav de hver især dækker.

UC2: Se Hold: Et medlem skal kunne se en liste over alle træningshold – både fulde og ledige hold .

Ud over den fuldt detaljerede use-case beskrivelse, er systemets use-cases også udarbejdet i det såkaldte "kort format" . Formålet med det korte format er at give læseren et hurtigt og præcist overblik over aktørens overordnede mål med den pågældende use-case, uden at gå i dybden med de specifikke system-interaktioner og trin.

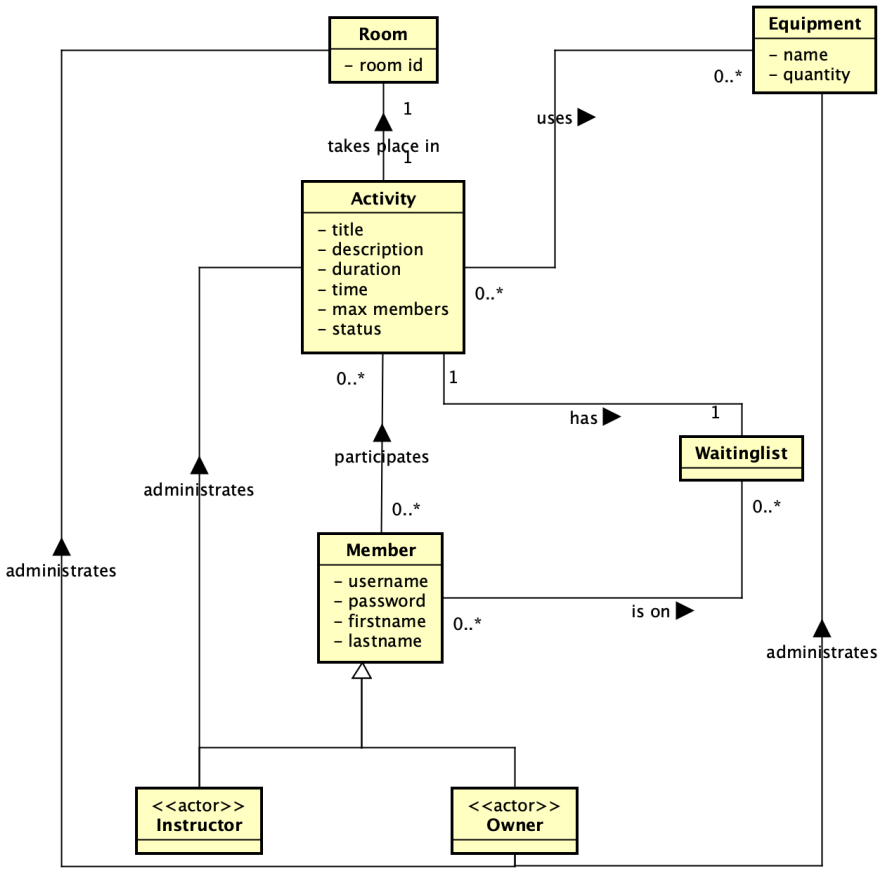
Alle use-case beskrivelser findes i Bilag[A].

For at sikre kvaliteten og relevansen af systemets use-cases har vi evalueret dem ud fra Boss-testen, Elementary Business Process-testen og Størrelses-testen. Tager vi udgangspunkt i vores primære use-case, UC1: Tilmeld Hold, består de n Boss-testen, da tilmelding af medlemmer skaber direkte forretningsværdi for fitnesscenteret og er en meningsfuld opgave for brugeren. Den består EBP-testen, da handlingen udføres af en person på ét tidspunkt og efterlader systemet i en stabil tilstand (medlemmet er enten tilmeldt eller på venteliste). Endelig består den Størrelses-testen, da use-casen

indeholder et passende antal trin (8 trin i basis-sekvensen) og derved hverken er for banal (kun et enkelt knaptryk) eller for kompleks til at kunne overskues i én proces.

3.4. Domænemodel og Ordliste

Der er lavet en domænemodel for holdtræningssystemet. Domænemodellen er baseret på kravene fra afsnit 2.2 samt use-cases fra afsnit 2.3.



Figur 2: Domænemodel

Figur 2 viser systemets centrale objekter og relationerne mellem dem. Modellen viser blandt andet sammenhængen mellem medlemmer, aktiviteter, instruktører, lokaler og udstyr.

Et medlem kan være tilmeldt flere aktiviteter, mens en aktivitet er tilknyttet både et lokale og en instruktør. Derudover kan aktiviteter anvende forskelligt udstyr, og systemet understøtter ventelister ved fyldte hold.

Domænemodellen blev anvendt som grundlag for det videre design af systemet, da modellen gav et samlet overblik over systemets struktur og centrale relationer.

Domænemodellen er vedlagt i Bilag[B].

3.4.1. Oversættelsestabel

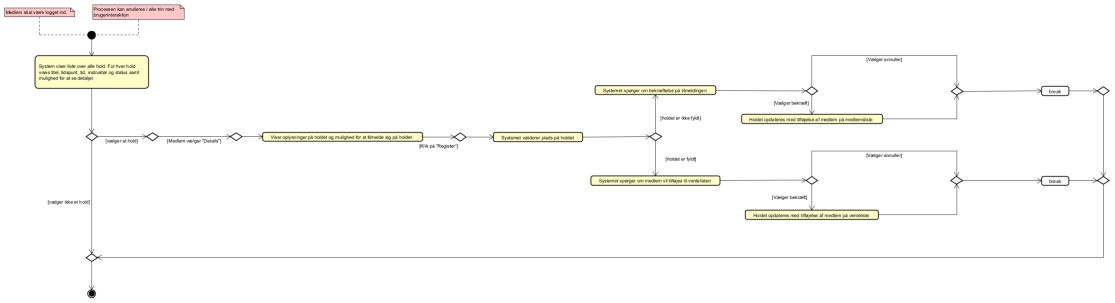
Analysetermer (Dansk)	Designtermer (Engelsk)	Beskrivelser
Lokale	Room	Et rum, hvor holdtræning forgår.
Aktivitet	Activity	En planlagt klasse, som medlemmer kan deltage.
Udstyr	Equipment	Redskaber, der bruges til holdtræning.
Venteliste	Waitinglist	En liste over medlemmer, der venter på ledige pladser.
Medlemsliste	Memberlist	En liste over alle registrerede medlemmer.
Medlem	Member	En person som er tilknyttet systemet og kan deltage i holdtræning
Instruktør	Instructor	En person som underviser holdtræning
Ejer	Owner	En person med overordnet rettigheder i systemet

Tabel 3: Oversættelsestabel

Tabel 3 viser oversættelsestabellen mellem analysebegreberne og de designtermer, der anvendes i systemets implementering. Tabellen blev udarbejdet for at skabe en tydelig sammenhæng mellem analyse, design og kode, da flere af klasserne og systemets struktur er navngivet på engelsk i implementeringen. Tabellen beskriver samtidig de vigtigste objekter i systemet og deres funktion.

3.5. Aktivitetsdiagram

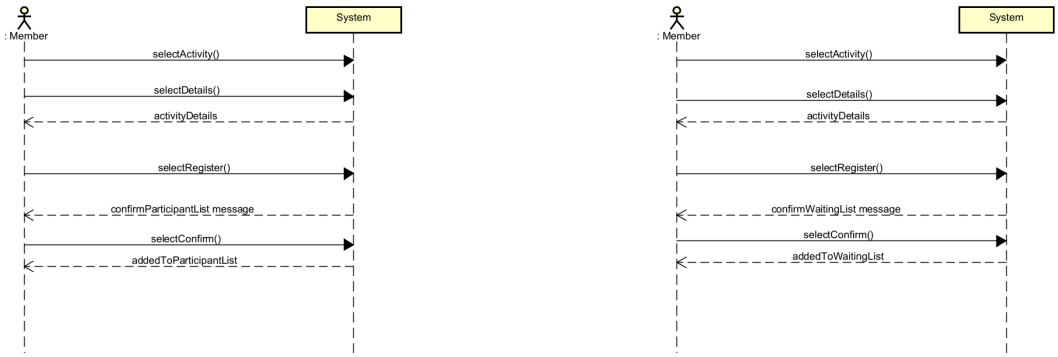
Der er lavet aktivitetsdiagram for use-casen tilmeld hold. Den vises herunder. Diagrammet er lavet med udgangspunkt i kravene, som er beskrevet i afsnit 2.2. Her har aktivitetsdiagrammer lignende funktion som use-case beskrivelserne og beskriver, hvordan programmet skal opføre sig overfor brugeren.



Figur 3: Aktivitetsdiagram for tilmeld hold

Figur 3 viser aktivitetsdiagrammet for "Tilmeld Hold" use-casen. Diagrammet illustrerer flowet for, hvordan et medlem interagerer med systemet for at tilmelde sig en aktivitet. Her ses de to primære scenarier baseret på systemets validering: enten tilføjes medlemmet direkte til holdets medlemsliste, hvis der er ledige pladser, eller også tilbydes medlemmet en plads på ventelisten, hvis holdet allerede er fuldt. Derudover ses det, at processen til enhver tid kan annulleres af brugeren. Dette er blevet brugt til implementation af denne funktionalitet. Dette aktivitetsdiagram kan findes i Bilag[C].

3.6. Systemsekvensdiagram



Figur 4: Systemsekvens diagram

Figur 4 viser systemsekvensdiagrammet for use-casen "Tilmeld Hold". Diagrammet beskriver interaktionen mellem medlemmet og systemet ved tilmelding til en aktivitet.

På det første diagram vælger medlemmet først en aktivitet og ser detaljer om holdet. Herefter vælger medlemmet at tilmelde sig aktiviteten, hvorefter systemet bekræfter tilmeldingen og tilføjer medlemmet til deltagerlisten.

Det andet diagram viser scenariet, hvor holdet er fyldt. Medlemmet vælger igen en aktivitet og ser detaljer om holdet, men i stedet for en normal tilmelding får medlemmet mulighed for at blive tilføjet til ventelisten. Systemet bekræfter herefter, at medlemmet er blevet tilføjet til ventelisten.

Systemsekvensdiagrammerne blev udarbejdet for at vise den overordnede kommunikation mellem bruger og system samt de vigtigste handlinger i forbindelse med tilmelding til aktiviteter, hvilket er brugbart når det skal implementeres.

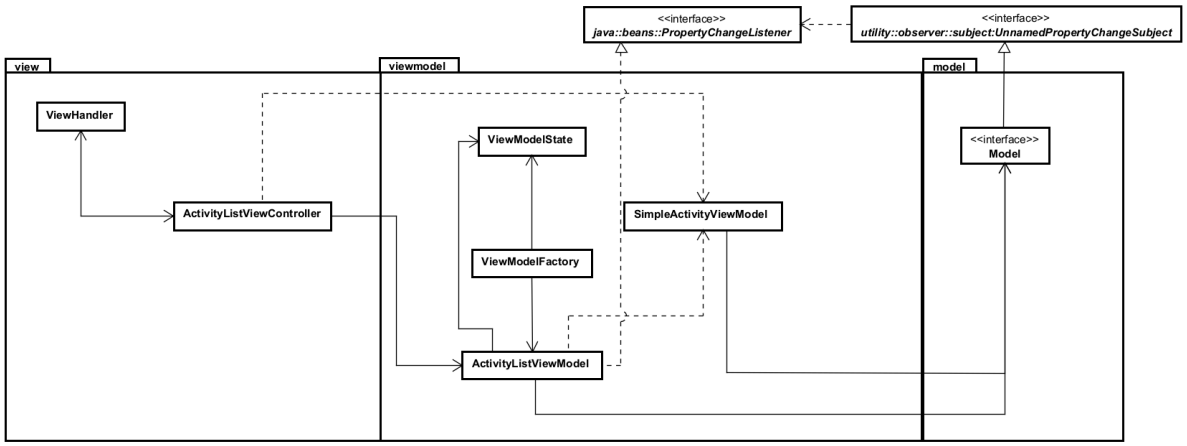
Alle systemssekvensdiagrammer kan findes i Bilag[D].

4. Design

4.1. Introduktion

På baggrund af analysen udarbejdes design. Specielt er systemsekvensdiagrammet, domænemodel og use-case beskrivelser brugt til at designe. Design er hovedsageligt udført med stort fokus på SOLID og GRASP. Afsnittet er udgjort af udvalgte designelementer som MVVM, Observer, Proxy og Adapter. Det overordnede klassediagram kan findes i Bilag[E].

4.2. MVVM



Figur 5: MVVM-Struktur i systemet (kun ActivityList)

Systemet er udviklet med MVVM-arkitekturen (Model-View-ViewModel) for at skabe en tydelig opdeling mellem brugergrænseflade, logik og datahåndtering. Formålet med anvendelsen af arkitekturen er at reducere koblingen mellem systemets lag og gøre løsningen mere overskuelig og vedligeholdelsesvenlig. Dette sørger for lav kobling og høj sammenhængskraft (GRASP) samt følger Open/Closed Principle fra SOLID-principperne. Dermed bliver systemet nemmere at videreudvikle og vedligeholde.

Som det kan ses på Figur 5, består MVVM-arkitekturen af tre lag:

- Model, som håndterer data og systemlogik
- View, som håndterer brugergrænsefladen
- ViewModel, som fungerer som mellemled mellem View og Model

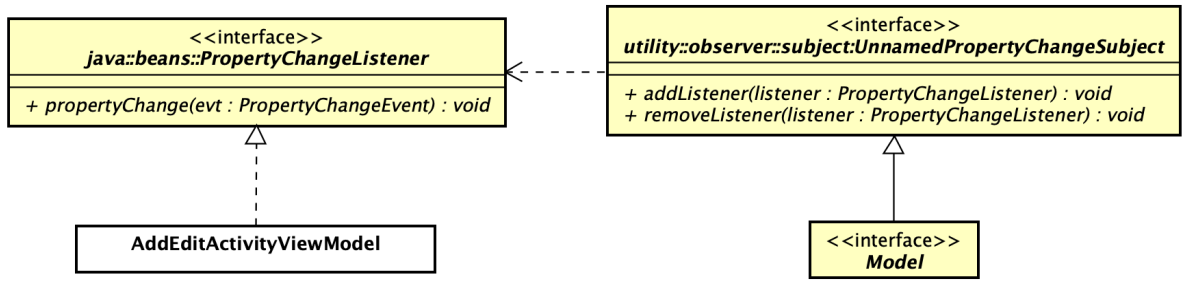
View-laget består blandt andet af klasser som ViewHandler og ActivityListViewController. Disse klasser håndterer navigation mellem vinduer samt brugerens interaktion med systemet og viser data i brugergrænsefladen.

ViewModel-laget indeholder blandt andet ActivityListViewModel og SimpleActivityViewModel. Disse klasser håndterer kommunikationen mellem View og Model samt databinding gennem JavaFX properties og ObservableLists. ViewModelFactory bruges til at oprette og dele ViewModels, mens ViewModelState anvendes til at gemme data mellem forskellige views.

Model-laget består af interfacet Model samt implementeringen Gym og domæneklasser som Activity, Member og Equipment. Dette lag håndterer systemets data og forretningslogik.

Som vist på Figur 5 kommunikerer View-laget ikke direkte med Model-laget, men går gennem ViewModel-laget. Dette sørger for, at kommunikationen som udgangspunkt kun sker en vej i systemet fra View til ViewModel til Model. Ved ændringer i modellen kan der dog sendes opdateringer tilbage gennem observer-patterns og JavaFX properties, så brugergrænsefladen automatisk opdateres med de nyeste data.

4.3. Observer



Figur 6: Eksempel på observermønster i systemet

Til kommunikationen fra Model til ViewModel anvendes observermønstret. Som vist på Figur 6 extender Model interfacet UnnamedPropertyChangeSubject, hvilket gør det muligt at tilføje og fjerne lyttere gennem addListener() og removeListener().

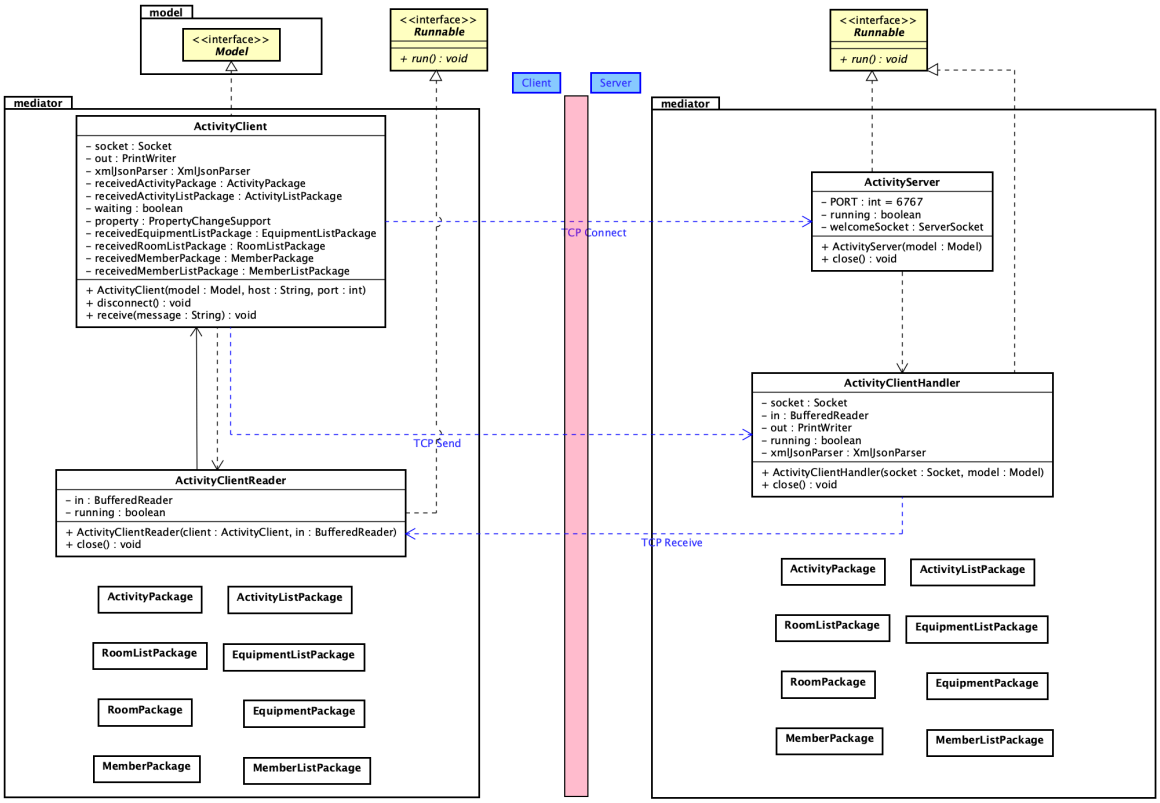
ViewModel-klasserne, f.eks. AddEditActivityViewModel, implementerer PropertyChangeListener. Det betyder, at de kan lytte efter ændringer i modellen og reagere, når der bliver sendt en PropertyChangeEvent.

Når der sker en ændring i Model-laget, kan modellen give besked videre til de tilknyttede ViewModels. ViewModel kan derefter opdatere sine properties, som View-laget er bundet til. På den måde bliver ændringer i modellen vist i brugergrænsefladen uden at View har direkte adgang til Model.

Figuren viser kun et eksempel på anvendelsen af observermønstret. Mønstret bruges flere steder i systemet, hvor ændringer i modellen skal sendes videre til andre dele af programmet. Dette understøtter MVVM-strukturen og hjælper med at holde koblingen mellem lagene lav

4.4. Client-Server

Systemet er designet som et client-server system, hvor klienterne håndterer brugerinteraktion, mens serveren håndterer logik, data og kommunikation med databasen.



Figur 7: Klient-server arkitektur

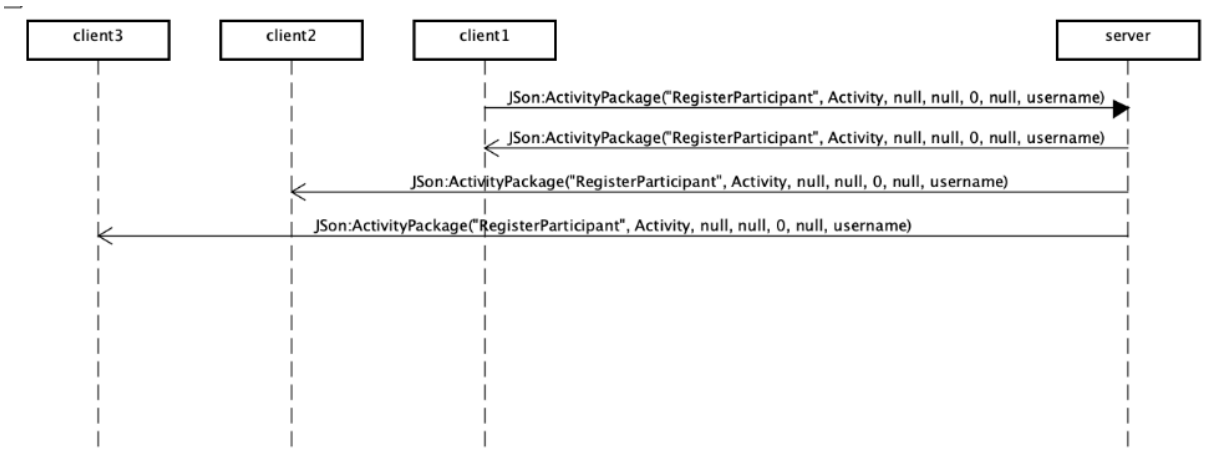
Som vist på Figur 7 lytter ActivityServer efter forbindelser fra klienter gennem en serversocket. Når en klient forbinder til serveren, oprettes en ActivityClientHandler, som håndterer kommunikationen med den enkelte klient i en separat tråd. Dette gør det muligt for flere klienter at være forbundet til systemet samtidig.

Klienterne kommunikerer med serveren ved at sende forespørgsler som JSON-beskeder gennem ActivityClient. Serveren modtager beskederne, behandler dem og sender derefter et svar tilbage til klienten. På klientsiden anvendes ActivityClientReader til at modtage opdateringer og svar fra serveren.

Kommunikationen mellem klient og server sker gennem forskellige packages, herunder ActivityPackage, MemberPackage, RoomPackage og EquipmentPackage, som anvendes til at overføre data mellem systemets dele.

Ved ændringer i systemet broadcastes opdateringer til de øvrige klienter, så alle klienter holdes synkroniseret. Arkitekturen gør det muligt for flere klienter at arbejde med samme server samtidig og sikrer, at alle klienter modtager de nyeste opdateringer fra serveren.

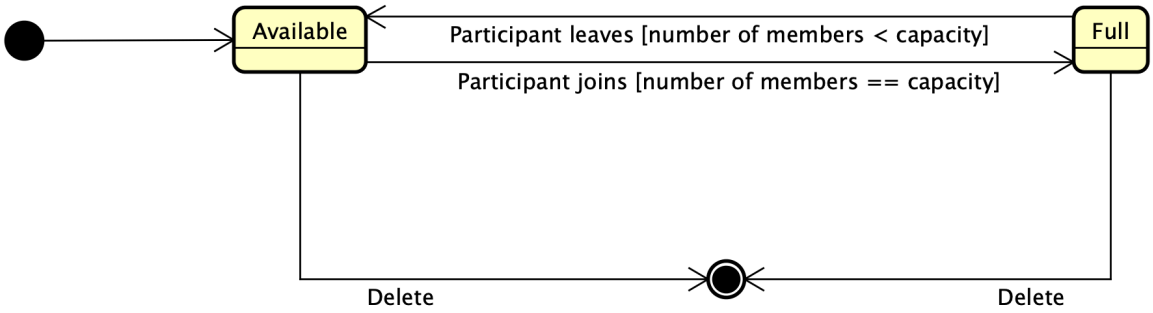
Systemet anvender JSON-baserede protokoller til kommunikationen mellem klient og server.



Figur 8: Protokol for RegisterParticipant

Den vigtigste protokol i systemet er tilmelding til hold (RegisterParticipant som vist på Figur 8). Når et medlem tilmelder sig en aktivitet, sender klienten en forespørgsel til serveren i form af et ActivityPackage, hvor der er angivet hvilken aktivitet og hvem der vil tilmelde sig. Serveren behandler herefter forespørgslen og opdaterer systemets data. Efter behandlingen sendes opdateringen videre til de øvrige klienter gennem broadcast, så alle klienter får opdateret deres visning af aktiviteten. Protokollen viser dermed, hvordan systemet håndterer kommunikation mellem flere klienter og samtidig holder data synkroniseret i hele systemet. Systemet indeholder flere lignende protokoller, som er vedlagt Bilag[F].

4.5. State

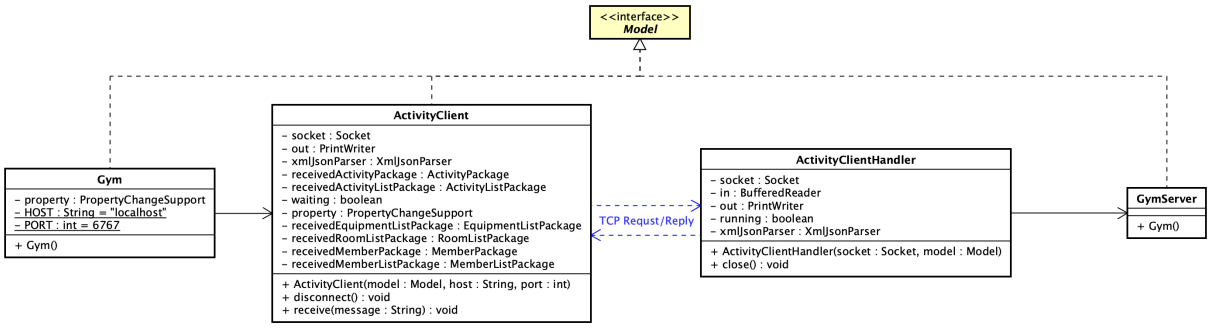


Figur 9: Protokol for RegisterParticipant

State-diagrammet (Figur 9) viser, hvordan en aktivitet skifter mellem tilstandene Available og Full. En aktivitet starter i tilstanden Available, hvor der er ledige pladser. Når en deltager tilmelder sig, og antallet af deltagere når kapaciteten, skifter aktiviteten til Full. Hvis en deltager forlader holdet, og der igen er ledige pladser, skifter tilstanden tilbage til Available. Derudover kan aktiviteten slettes fra begge tilstande gennem handlingen

Delete, hvilket afslutter aktivitetens livscyklus. Der er gjort brug af et statemønster, for at holde systemet Open/Closed (SOLID), hvis der i fremtiden bliver brug for en ny status på en aktivitet som "midlertidigt lukket", eller lignende. Det bliver derfor nemt at tilføje, uden at ændre i det fortidige.

4.6. Proxy



Figur 10: Eksempel på proxy

Ovenstående Figur 10 viser anvendelsen af proxy-mønstret i kommunikationen mellem klient og server. Både Gym, ActivityClient og GymServer implementerer interfacet Model (Subject). Hertil håndterer ActivityClientHandler alle metoder fra Model-interface, hvilket gør at kommunikation sker gennem samme interface uden at kende den konkrete implementering.

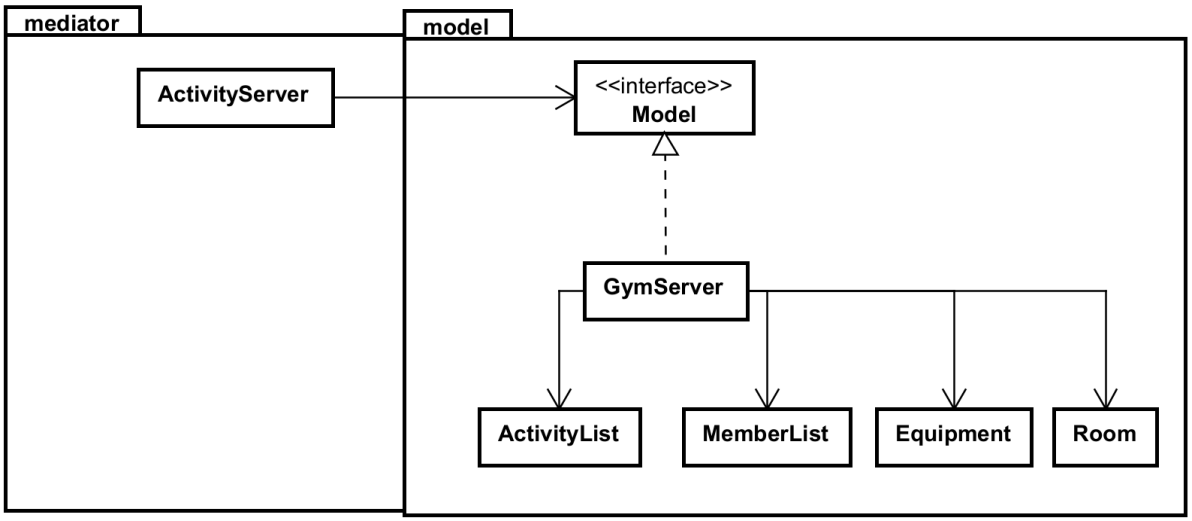
Her fungerer Gym og ActivityClient som proxy, fordi de videredelagerer ansvaret til det rigtige subjekt (Real Subject) som er ActivityClientHandler og GymServer. Når en forespørgsel sendes til modellen, modtages den først af Gym, som derefter videresender opgaven til ActivityClient, der håndterer kommunikationen med serveren.

På den måde skjules netværkskommunikationen for andre lag, samtidig med at koblingen mellem lagene holdes lav. Proxy-patternet gør det derfor muligt at adskille klientkommunikation fra resten af systemets logik.

Som tidligere nævnt implementerer ActivityClientHandler ikke Model-interfacet, men håndterer request af alle metoder derfra, derfor konkluderes at dette proxy-mønster designes som et Remote Proxy-lignende.

State-diagrammet er vedlagt i Bilag[G].

4.7. Adapter



Figur 11: Eksempel på adapter

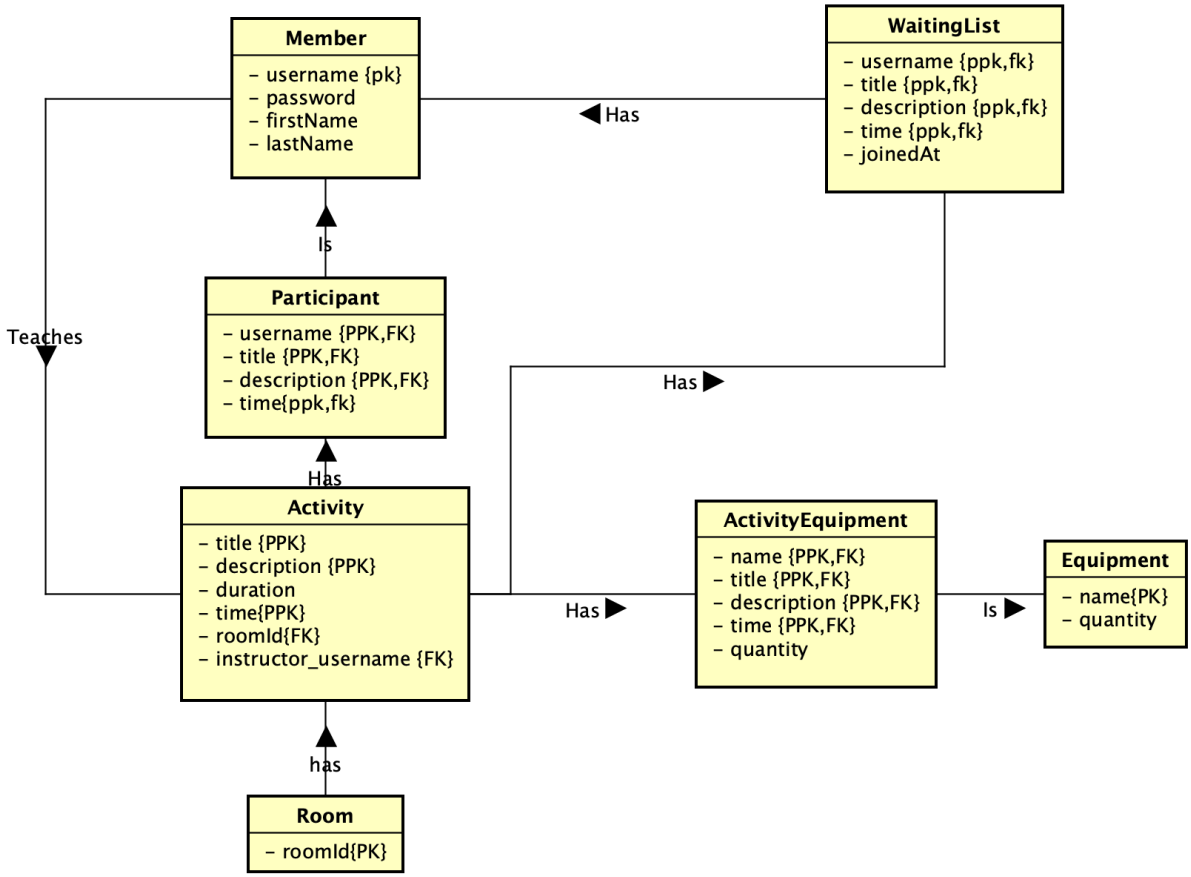
Ovenstående Figur Figur 11 viser anvendelsen af adapter-patternet i systemet. GymServer fungerer som adapter ved at implementere interfacet Model, hvilket gør det muligt for ActivityServer at kommunikere med systemet gennem et fælles interface.

GymServer forbinder serverlaget med systemets underliggende objekter og lister, herunder ActivityList, MemberList, Equipment og Room. På den måde skjules den interne struktur for ActivityServer, som kun behøver at kommunikere gennem Model-interfacet.

Adapter-patternet gør det dermed muligt at samle adgangen til systemets data i en klasse og reducerer samtidig koblingen mellem serverlaget og model-laget

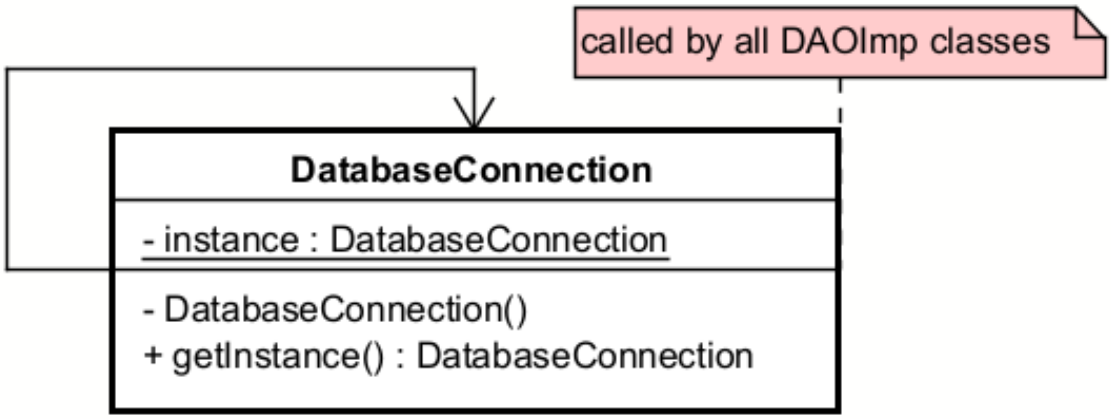
4.8. Database

Systemet har brug for at lagre informationer over aktiviteter, medlemmer, udstyr og rum. Det er dog essentielt at systemet også gemmer på, hvilke medlemmer der er tilmeldt på hold og hvilke medlemmer der er tilmeldt på ventelisten for at kunne komme på et hold så snart der kommer en ledig plads.



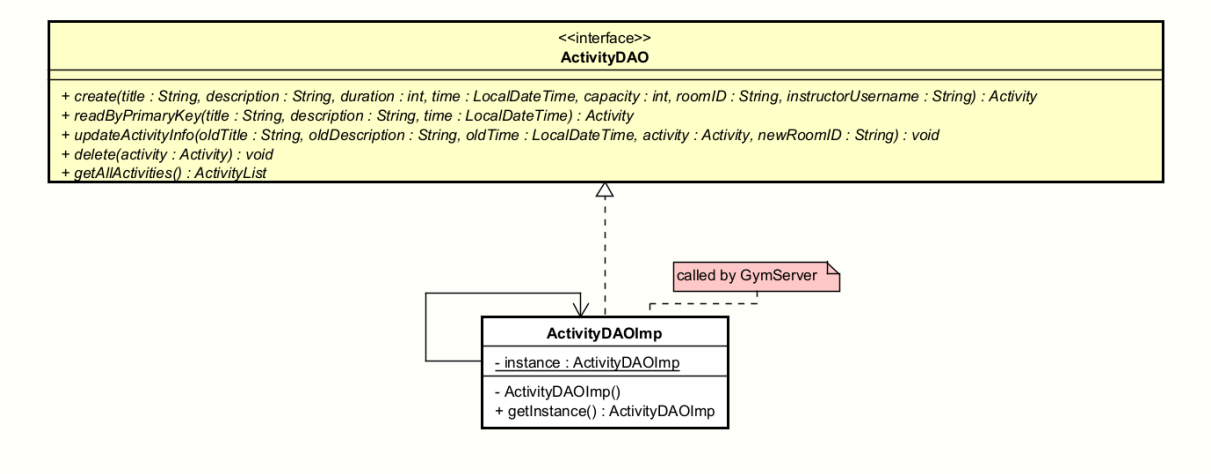
Figur 12: EER diagram for database

På ovenstående Figur 12 ses EER diagrammet for databasen her ses at den i tredje normalform og er blevet udarbejdet over flere sprints. Her beskrives alle relationer som er på databasen. Her ses at en aktivitet har flere relations med member og equipment som skaber en mellemrelation med Participant og ActivityEquipment siden at hver aktivitet har deres egen liste med medlemmer og udstyr. Der kan ses at Member kan være en del af en aktivitet hvorpå de "Teacher" det vil sige det er relationen for at have en instruktør på en aktivitet. Hver aktivitet har en venteliste med medlemmer på ventelisten. Baseret på EER diagram på Figur 12 kan udvikles en database. Her er det dog nødvendigt at kunne tilgå databasen og skabe en forbindelse til databasen. Astah filen til Figur 12 kan findes under Bilag[H].



Figur 13: Forbindelse oprettes til database

Ovenstående Figur 13 viser klassediagrammet for at oprette en forbindelse med databasen her er det designet til at være en singleton sådan at der kun kan eksistere en databaseforbindelse som kan kaldes på.



Figur 14: DAO til aktivitet

For at adskille databasekode fra resten af systemets logik anvendes DAO-mønstret (Data Access Object). DAO-laget fungerer som et mellemled mellem systemets model og databasen. Som ses på ovenstående Figur 14 beskriver DAO-interfacet de tilgængelige operationer, mens DAO-implementeringerne indeholder SQL-forespørgsler og forbindelsen til databasen. Dette gør systemet mere struktureret og reducerer koblingen mellem model-laget og databasen. Når serveren starter, hentes data fra databasen og indlæses i systemets model. Herefter arbejder serveren videre med den opdaterede model, mens databasen opdateres ved ændringer som oprettelse, redigering eller sletning af aktiviteter og medlemmer. Anvendelsen af DAO-mønstret gjorde det muligt at holde databasehåndtering adskilt fra systemets øvrige logik og gjorde løsningen lettere at vedligeholde.

5. Implementation

5.1. Introduktion

På baggrund af de udvalgte designelementer fra designafsnittet udarbejdes dette implementations afsnit. Derfor tages der udgangspunkt i de mønstre beskrevet i design-afsnittet, hvor der videreforklares deres brug i projektet med fokus på, hvordan det er blevet implementeret. Kildekoden er vedlagt i Bilag[I].

5.2. MVVM & Observer

I design afsnittet blev der forklaret, hvordan MVVM-mønstret er blevet brugt til at designe systemet. Hvordan det er blevet implementeret, gennemgås i dette delafsnit. Det er valgt at vise DetailsActivityViewModel, hvilket er siden der vises til medlemmer hvis der trykkes på detaljer over en aktivitet. Der vælges ikke at vise hele klasser men kun de vigtigste elementer siden at hele klassen kan findes i Bilag[I].

```
titleField.textProperty().bindBidirectional(detailsActivityViewModel.getTitleFieldProperty());

stateLabel.textProperty().bind(detailsActivityViewModel.getStateProperty());

durationField.textProperty().bindBidirectional(
    detailsActivityViewModel.getDurationFieldProperty(),
    new IntStringConverter()
);
```

Kodestykke 1: DetailsActivityViewController Init()

Ovenstående Kodestykke 1 viser de vigtigste linjer fra Init metoden i DetailsActivityViewControlleren. Her kan ses bindingerne i init metoden der er både anvendt bindBidirectional() og bind(). Bindbidirectional giver os mulighed for at at den kan opdateres fra ViewController og ViewModel. Hvor .bind() kun kan opdateres fra ViewModel.Her kan også ses at der er anvendt en .IntStringConverter() dette er gjort for at håndtere Int stil String konvertering på View siden.

```
usernameColumn.setCellValueFactory(cellData ->
cellData.getValue().getUsernameProperty());
memberList.setItems(detailsActivityViewModel.getAll());
);
```

Kodestykke 2: DetailsActivityViewController data binding til tabel

Ovenstående Kodestykke 2 viser hvordan UI-tabellen med medlemmer bliver kobblet på vha. viewmodellen.

```
@FXML private void backButtonPressed()
{
  switch(detailsActivityViewModel.getRole())
  {
    case "Owner":
      viewHandler.openView("owner-activity-list");
      break;
    case "Member", "Instructor":
      viewHandler.openView("list");
      break;
  }
}
```

Kodestykke 3: Brug af ViewHandler i ViewController

Ovenstående Kodestykke 3 viser hvordan viewControlleren ikke selv styrer logikken men anvender viewmodel til at finde ud af hvilken fxml-view instruktøren, medlemmet eller ejeren skal smides tilbage på.

```
@Override
public void propertyChange(PropertyChangeEvent evt)
{
  switch (evt.getPropertyName())
  {
    case "RegisterParticipant", "RemoveMemberFromActivity":
      reset();
      break;
  }
}
```

Kodestykke 4: Viewhandler for DetailsActivityView

Nu kigges på viewmodel deleen af MVVM mønstret.Som ses på ovenstående Kodestykke 4 er det her at observer mønsteret bliver implementeret ved at der lyttes på en propertychange gøres at når der sker en ændring tvinges clientens program til at reset hvilket resulterer i at informationerne på klienten opdateres.

```
public void loadFromModel()
{
  Platform.runLater(() -> {
    LocalDateTime exactTime = viewModelState.getTime();
    ArrayList<Member> modelList = model.getActivity(viewModelState.getTitle(),
        viewModelState.getDescription(), exactTime).getParticipantList();
    list.clear();
    if (modelList != null) {
      for (int i = 0; i < modelList.size(); i++) {
        list.add(new SimpleMemberViewModel(modelList.get(i)));
      }
    }
  });
}
```

Kodestykke 5: Viewhandler for DetailsActivityView

Ovenstående Kodestykke 5 viser hvordan at data bliver hentet fra model asynkront her kan ses at der anvendes Platform.runLater()→, dette gøres sådan at når deltagerlisten bliver opdateret vil ændringen blive lagt i trådkøen og ikke crashe programmet.

```
if(status.equals("Full"))
{
  alert.setHeaderText("Do you wish to join the waiting list?");
} else {
  alert.setHeaderText("Do you wish to join the member list?");
}
```

Kodestykke 6: Viewhandler for DetailsActivityView

Ovenstående Kodestykke 6 viser anvendelsen af en dialog-box for medlemmer som beskrevet i kravene i afsnit 1. Sættes et krav for at der skal være en almindelig medlemsliste og en venteliste her er der lavet logik baseret på anvendelse af hvilken status aktiviteten har. Hvis holdets status er "Full" tilmeldes et medlem til ventelisten og hvis det er alt andet ergo "Available" tilmeldes medlemmet aktiviteten.

```
public void registerButtonPressed()
  {
    try
    {
      Activity activity = model.getActivity(titleFieldProperty.get(),
          descriptionFieldProperty.get(), viewModelState.getTime());
      Member member = model.getMember(viewModelState.getUsername());
      if(activity.isParticipantOnList(member))
      {
        model.removeFromParticipantList(member, activity);
      }
      else
      {
        if(registerConfirmation(activity.getStatus()))
        {
          model.addToParticipantList(member, activity);
        }
      }
    }
    catch(Exception e)
    {
      errorProperty.set(e.getMessage());
    }
  }
```

Kodestykke 7: DeatilsActivityViewModel Register Button Pressed

Ovenstående Kodestykke 7 håndterer hvad der sker hvis at der bliver trykket Register på client siden. Her vil den bruge model til at fjerne fra medlemslister og tilføje på medlemslister. Baseret på om medlemmet allerede er tilmeldt aktiviteten eller ej. Der vil gåes mere i detalje på hvordan data gemmes i database afsnittet under implementation.

```
Activity activity = model.getActivity(
    viewModelState.getTitle(),
    viewModelState.getDescription(),
    viewModelState.getTime()
);
```

Kodestykke 8: Viewhandler for DetailsActivityView

Ovenstående Kodestykke 8 her vises hvordan viewmodelstate anvendes til at kunne gemme på informationer mellem forskellige views. Når et medlem har valgt en aktivitet som de gerne vil se aktiviter på er man nødt til at gemme informationer på aktiviteten et sted. Det er viewmodelstate det gøres i her gemmes informationerne på en valgt aktivitet.

5.3. Proxy

I afsnit 3.6 blev proxy-mønstret og dens indflydelse på designet udfoldet. Der tages udgangspunkt i det og forklares, hvordan det er implementeret.

```
public interface Model extends UnnamedPropertyChangeSubject
{
   // UnnamedPropertyChangeSubject metoder er udeladt

    ArrayList<Activity> getAllActivities();

    // Flere getters og setters er udeladt
}
```

Kodestykke 9: Model Interface

På Kodestykke 9 vises et kodestykke fra Model interfacet, hvor et eksempel på en getter "getAllActivities" er vist. Model interfacet fungerer som subjektet i Proxy-mønstret, fordi den er en erstatning for det rigtige subjekt, som er ActivityClient. Dette bliver implementeret, ved at de andre to klasser implementerer Model.

```
public class Gym implements Model, PropertyChangeListener
{
    //Udeladt instanser og konstruktør

    @Override public ArrayList<Activity> getAllActivities()
    {
        return activityClient.getAllActivities();
    }

    // Udeladt flere lignende metoder, implementeret fra Model Interface
}
```

Kodestykke 10: Gym getAllActivities()

På Kodestykke 10 vises et kodestykke fra klientsidens Gym-klasse. Det vises klart og tydeligt, at den implementerer Model. Ved alle de implementerede metoder fra Model-interfacet videredelegeres opgaven videre til ActivityClient. Derfor er Gym Proxy-klassen i Proxy-mønstret.

```
public class ActivityClient implements Model
{
  // Udeladt instansvariabler, konstruktør og receive metode.
    @Override public synchronized ArrayList<Activity> getAllActivities()
    {
        try
        {
            out.println(xmlJsonParser.toJson(
                    new ActivityListPackage("GetAllActivities", null), false));
            while (receivedActivityListPackage == null)
            {
                waiting = true;
                wait();
            }
            ArrayList<Activity> activityArrayList =
receivedActivityListPackage.getAllActivities();
            waiting = false;
            receivedActivityListPackage = null;
            return activityArrayList;
        }
        catch (ParserException | InterruptedException e)
        {
            throw new RuntimeException(e);
        }
    }
}
```

Kodestykke 11: ActivityClient getAllActivities()

ActivityClient som er vist på Kodestykke 11, håndterer den faktiske handling. Det er den, der kontakter serveren og spørger om alle aktiviteter, når det efterspørges i klienten. Her er der taget et udsnit af metoden getAllActivities(), for at vise, hvordan den kontakter serveren (ved out.println). Det bliver uddybet i afsnit 4.5.

```
public class ActivityClientHandler implements PropertyChangeListener, Runnable
{
  // Udeladt instansvariabler, konstruktør, close og propertyChange metode.
    @Override public void run()
    {
      // try-catch udeladt
      while (running)
      {
          String messageIn = in.readLine();
          if (messageIn == null) break;

          Map<String, Object> jsonMap = xmlJsonParser.fromJson(messageIn, Map.class);
          String type = (String) jsonMap.get("type");

          switch (type)
          {
            case "GetAllActivities":
                out.println(xmlJsonParser.toJson(new
ActivityListPackage("GetAllActivities", model.getAllActivities()),
ActivityListPackage.class, false));
            break;
          }
      }

    }
}
```

Kodestykke 12: ActivityClientHandler

På Kodestykke 12 viser håndteringen af modtagelsen af en GetAllActivities-request. Da ActivityClientHandler ikke implementerer Model-interfacet, men stadig håndterer alle de samme metoder, kaldes det et Remote proxy-lignende proxy-mønster. Den kalder model.getAllActivities(), som spørger serverens model om alle aktiviteter og returnerer det. Dermed bliver det sendt tilbage til ActivityClient, der venter på det.

5.4. Adapter

I afsnit 3.7 blev Adapter og dens indflydelse på designet udfoldet. Der tages udgangspunkt i det og forklares, hvordan det er implementeret for at sikre, at ActivityServer kan kommunikere med systemets domænelister uden at være bundet til deres specifikke implementeringer.

```
public interface Model extends UnnamedPropertyChangeSubject
{
    // ... Observer metoder udeladt
    ArrayList<Activity> getAllActivities();
    Activity getActivity(String title, String description, LocalDateTime time);
    void addToParticipantList(Member member, Activity activity);
    // ... Flere metoder for Members, Rooms og Equipment udeladt
}
```

Kodestykke 13: Model Interface

På Kodestykke 13 ses Model-interfacet, som agerer Target i Adapter-mønstret. Dette interface definerer de operationer, som klienten (ActivityServer) forventer at kunne kalde for at få adgang til systemets data. Ved at afhængigheder peger mod dette interface i stedet for konkrete klasser, bevares lav kobling.

```
public class Gym implements Model
{
    private PropertyChangeSupport property;
    private MemberList memberList;
    private ActivityList activityList;
    private ArrayList<Room> rooms;
    private ArrayList<Equipment> equipment;

    // ... Konstruktør der initialiserer listerne via DAO'er udeladt

    @Override public ArrayList<Activity> getAllActivities() {
        return activityList.getActivities();
    }

    @Override public Activity getActivity(String title, String description,
LocalDateTime time) {
        if (title == null || description == null || time == null) {
            throw new IllegalArgumentException("Search parameters cannot be null");
        }
        return activityList.getActivity(title, description, time);
    }

    @Override public Member getMember(String username) {
        if (username == null || username.isBlank()) {
            throw new IllegalArgumentException("Username cannot be null or empty");
        }
        return memberList.getMember(username);
    }
    // ... Flere delegerende metoder udeladt
}
```

Kodestykke 14: Gym server

Gym-klassen (som på serversiden ofte refereres til som GymServer) udgør selve Adapteren. Som det ses på Kodestykke 14, implementerer Model-interfacet. Denne klasse indkapsler de underliggende Adaptees, som i dette system er domænelisterne (ActivityList, MemberList samt lister for Room og Equipment).

Når Gym modtager et kald gennem Model-interfacet, eksempelvis getActivity, tilpasses og videresendes kaldet til den specifikke ActivityList-instans. Adapteren fungerer dermed som et samlende lag, der konverterer de standardiserede anmodninger fra serveren til operationer på de individuelle dataobjekter.

```
public class ActivityServer implements Runnable
{
    private Model model;
    private static final int PORT = 6767;
    private ServerSocket welcomeSocket;

    public ActivityServer(Model model) throws IOException
    {
        this.model = model;
        this.welcomeSocket = new ServerSocket(PORT);
        // ... udeladt logik
    }

    @Override public void run()
    {
        try
        {
            while (!welcomeSocket.isClosed())
            {
                Socket socket = welcomeSocket.accept();
                ActivityClientHandler handler = new ActivityClientHandler(socket,
model);
                // ... trådhåndtering udeladt
            }
        }
        // ... catch blok udeladt
    }
}
```

Kodestykke 15: ActivityServer

På Kodestykke 15 vises ActivityServer. I konteksten af Adapter-mønstret er dette den klasse, der gør brug af det fælles Model-interface (Target) for at tilgå systemets data.

Dette ses ved, at ActivityServer udelukkende kender til Model. I konstruktøren modtager et objekt af typen Model (som i praksis er Gym-adapteren). ActivityServer har dermed intet kendskab til, hvordan domænelisterne, som f.eks. ActivityList eller MemberList, fungerer internt eller er struktureret. Referencen til modellen videregives derefter til ActivityClientHandler, som behandler de faktiske forespørgsler fra systemets brugere. Denne struktur sikrer, at serverens netværks- og trådhåndteringslogik er fuldstændig afkoblet fra den underliggende datahåndtering.

5.5. Client-Server

I afsnit 3.6 blev implementeringen af proxy-mønstret dokumenteret. Eksemplet på proxy-implementationen var til kommunikation mellem klient og server. Derfor vil der blive henvist til det afsnit og taget udgangspunkt i nogle af dets kodestykker. Ved relevans tilføjes nye kodestykker, der muligvis er fra de samme klasser som vist i kodestykkerne fra afsnit 3.6.

```
public class ActivityClient implements Model
{
    // Instansvariabler er udeladt

    public ActivityClient(Model model, String host, int port) throws Exception
    {
        this.model = model;

        xmlJsonParser = new XmlJsonParser();
        // xmlJsonParser.registerPolymorphicAdapter er udeladt
        socket = new Socket(host, port);

        in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        out = new PrintWriter(socket.getOutputStream(), true);

        // Initialisering af packages er udeladt

        waiting = false;

        this.property = new PropertyChangeSupport(this);

        ActivityClientReader activityClientReader = new ActivityClientReader(this,
                in);
        Thread thread = new Thread(activityClientReader);
        thread.setDaemon(true);
        thread.start();
    }
}
```

Kodestykke 16: ActivityClient Konstruktør

I Kodestykke 16 vises ActivityClients konstruktør. Dette indebærer, hvordan klienten opretter forbindelse til serveren. Det bliver gjort ved at oprette en socket. Dertil oprettes en måde at modtage information og give information (request / reply). Hvilket henholdsvis bliver til in (BufferedReader) og out (PrintWriter).

```
public class ActivityClient implements Model
{
    // Instansvariabler og alle andre metoder er udeladt
    @Override public synchronized void addActivity(Activity activity) {
        try
        {
            receivedActivityPackages.clear();
            out.println(xmlJsonParser.toJson(
                    new ActivityPackage("CreateActivity", activity, null, null, 0,
                            null, null), false));
            while(receivedActivityPackages.isEmpty()){
                waiting = true;
                wait();
            }
            waiting = false;
            for (int i = 0; i < receivedActivityPackages.size(); i++) {
                if (receivedActivityPackages.get(i).getErrorMessage() == null) {
property.firePropertyChange(receivedActivityPackages.get(i).getType(),
                            receivedActivityPackages.get(i).getUsername(),
receivedActivityPackages.get(i).getActivity());
                }
                else
                {
                    throw new
IllegalArgumentException(receivedActivityPackages.get(i).getErrorMessage());
                }
                receivedActivityPackages.clear();
            }
        }
        catch (ParserException | InterruptedException e)
        {
            throw new RuntimeException(e);
        }
    }
}
```

Kodestykke 17: ActivityClient addActivity()

I Kodestykke 17 vises, hvordan klienten kontakter serveren ved en request. Det er med udgangspunkt i metoden addActivity(). Den sender en request via en ActivityPackage, der følger protokollen for "CreateActivity". Den sendes til serveren ved at konvertere den til en JSon-streng med XmlJsonParser. Dernæst sendes JSon-strengen til serveren via out.println.

```
public class ActivityServer implements Runnable
{
    // Instansvariabler, prints og close() er udeladt
    public ActivityServer(Model model) throws IOException
    {
        this.model = model;
        this.welcomeSocket = new ServerSocket(PORT);
    }

    @Override public void run()
    {
      // try catch udeladt
      while (!welcomeSocket.isClosed())
      {
          Socket socket = welcomeSocket.accept();

          ActivityClientHandler handler = new ActivityClientHandler(socket, model);
          Thread t = new Thread(handler);
          t.setDaemon(true);
          t.start();
      }
    }
}
```

Kodestykke 18: ActivityServer Klient-server

I Kodestykke 18 vises, hvordan serveren initialiseres. Ved at initialisere server-socket, vente på at modtage en ny klient-forbindelse og delagerer en ny tråd til den klient, der opretter forbindelse til serveren via ActivityClientHandler. På den måde håndteres flere klient-forbindelser på en gang af serveren.

```
public class ActivityClientHandler implements PropertyChangeListener, Runnable
{
  // Udeladt instansvariabler, close og propertyChange metode.

    public ActivityClientHandler(Socket socket, Model model) throws IOException
    {
        this.socket = socket;
        this.model = model;
        this.in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        this.out = new PrintWriter(socket.getOutputStream(), true);
        this.running = true;
        this.xmlJsonParser = new XmlJsonParser();
        // xmlJsonParser.registerPolymorphicAdapter udeladt

        model.addListener(this);
        System.out.println("Connection established with " +
socket.getInetAddress().getHostAddress());
    }

    @Override public void run()
    {
      // try-catch udeladt
      while (running)
      {
          String messageIn = in.readLine();
          if (messageIn == null) break;

          Map<String, Object> jsonMap = xmlJsonParser.fromJson(messageIn, Map.class);
          String type = (String) jsonMap.get("type");

          switch (type)
          {
            case "CreateActivity":
              try {
                  ActivityPackage pkg = xmlJsonParser.fromJson(messageIn,
ActivityPackage.class);
                  model.addActivity(pkg.getActivity());
              } catch (Exception e) {
                  out.println(xmlJsonParser.toJson(new ActivityPackage("Error",
e.getMessage()), false));
              }
            break;
          }
      }
    }
}
```

Kodestykke 19: ActivityClientHandler CreateActivity

I Kodestykke 19 ses et udsnit fra ActivityClientHandler-klassen, hvor der kun er vist håndtering af CreateActivity og konstruktøren.

Konstruktøren viser, hvordan den opretter forbindelse til klienten på baggrund af ActivityServer, som giver client-socket ind i ActivityClientHandlerens konstruktør ved oprettelse.

Metoden run() søger efter en ny modtaget request, når den har fået en request, tjekker den efter typen på requesten, som blev angivet "CreateActivity" på klientens forespørgsel. Derfor bliver den håndteret i switch casens "CreateActivity" case.

Derefter oprettes en ny pakke ift. den type request der er modtaget. I dette tilfælde er det en ActivityPackage. På den måde kan der nu hentes det Activity-objekt, der blev sendt med klientens request. Den bliver nu oprettet i serverens model, ved model.addActivity(). Hvis der opstår en fejl i oprettelsen af aktivitets-objektet, samler en try-catch blok det op og sender en reply med "Error"-typen og exception-beskeden.

```
        @Override public void addActivity(Activity activity) {
               // Udeladt database håndtering
               activityList.addActivity(activity);
               property.firePropertyChange("CreateActivity", null, activity);
         }
```

Kodestykke 20: GymServer addActivity()

I Kodestykke 20 vises modellens metode addActivity. Det ses at den fyrer et nyt event via firePropertyChange() på det PorpertyChangeSupport-objekt, der intialiseres i konstruktøren (ikke vist). Den sender derudover de informationer som den bruger til at tilføje et nyt aktivitets-objekt.

```
  @Override public void propertyChange(PropertyChangeEvent evt)
  {
    //try-catch blok udeladt
    String eventName = evt.getPropertyName();
    switch (eventName) {
        case "CreateActivity":
          Activity act = (Activity) evt.getNewValue();
          out.println(xmlJsonParser.toJson(new ActivityPackage("CreateActivity", act,
null, null, 0, null, null), ActivityPackage.class, false));
        break;
    }
  }
```

Kodestykke 21: ActivityClientHandler CreateActivity

Dette bliver samlet op i ActivityClientHandler, fordi den implementerer PropertyChangeListener og tilføjer sig selv som lytter på subjektet (modellen). Det vil sige at alle ActivityClientHandler modtager denne ændring.

I Kodestykke 21 ses, hvordan det håndteres i propertyChange, når en aktivitet tilføjes i server-modellen og eventet bliver fyret. Den sender en reply til klienten med typen "CreateActivity" og aktivitets-objektet. Det modtager alle klienter nu, og det håndteres derfor på alle klienter, så alle klienter altid er opdateret til samme stadie. Dette kaldes også broadcast.

5.6. State

I afsnit 3.5 blev State-mønstret og dens indflydelse på designet udfoldet.hvor det blev beskrevet, hvordan en aktivitet (Activity) skifter mellem tilstandene Available og Full. Der tages udgangspunkt i det og forklares, hvordan det er implementeret.

```
public abstract class ActivityState
{
  // equals metode udeladt

  public String status()
  {
      return getClass().getSimpleName();
  }

  public abstract void setAvailable(Activity activity);

  public abstract void setFull(Activity activity);
}
```

Kodestykke 22: ActivityState

På Kodestykke 22 vises et uddrag af ActivityState klassen, som fungerer som den abstrakte tilstandsklasse (State) i mønstret. Den definerer de abstrakte metoder setAvailable og setFull, som de konkrete tilstande skal implementere. Metoden status() returnerer navnet på den specifikke underklasse som en streng (f.eks. “Available” eller “Full”), hvilket gør det nemt for systemet at kontrollere den nuværende tilstand.

```
public class Available extends ActivityState
{
    @Override public void setAvailable(Activity activity)
    {
        // Activity is already available; no action needed.
    }

    @Override public void setFull(Activity activity)
    {
        activity.setState(new Full());
    }
}
```

Kodestykke 23: Availabe

På Kodestykke 23 vises den konkrete tilstand Available, der arver fra ActivityState. Som det ses i setFull-metoden, videredelegeres ansvaret for at skifte tilstand til denne klasse. Kaldes der setFull() på en aktivitet, der er i Available-tilstand, ændres aktivitetens tilstand til et nyt Full-objekt. Kaldes derimod setAvailable(), sker der intet, da aktiviteten allerede er ledig. Full-klassen er implementeret på præcis samme måde, blot med omvendt logik, hvor setAvailable() skifter tilstanden tilbage til Available.

```
public class Full extends ActivityState
{
    @Override public void setAvailable(Activity activity)
    {
        activity.setState(new Available());
    }

    @Override public void setFull(Activity activity)
    {
        // Activity is already full; no action needed.
    }
}
```

Kodestykke 24: Full

På samme måde som den foregående klasse, viser Kodestykke 24 implementeringen af Full-tilstanden. Her er logikken håndteret omvendt. Kaldes setAvailable() på en aktivitet, fordi en deltager for eksempel har afmeldt sig, ændres aktivitetens tilstand til et nyt Available-objekt. Forsøges derimod et kald til setFull(), sker der ingen ændring, da aktiviteten allerede er fyldt op.

```
public class Activity
{
    private ActivityState activityState;

    // Udeladt instansvariabler og konstruktør

    public String getStatus() { return activityState.status(); }

    public void setAvailable() { activityState.setAvailable(this); }

    public void setFull() { activityState.setFull(this); }

    public boolean addParticipant(Member member)
    {
        // ... validering udeladt
        if (getStatus().equals("Full"))
        {
            waitingList.addMember(member);
            return false;
        }
        else
        {
            participantList.addMember(member);
            if (participantList.isFull()) setFull();
            return true;
        }
    }
}
```

Kodestykke 25: Activity

Activity-klassen, som ses på Kodestykke 25, agerer Context i state-mønstret. Den indeholder en reference til det nuværende ActivityState-objekt. I konstruktøren (ikke vist) initialiseres activityState som standard til new Available().

Når et medlem tilføjes via addParticipant(), kontrolleres tilstanden først via getStatus(). Hvis der er plads, tilføjes medlemmet til participantList. Herefter tjekker systemet, om listen nu er fuld (participantList.isFull()). Er dette tilfældet, kaldes setFull(), som delegerer kaldet videre til activityState.setFull(this), hvorefter tilstanden skifter til Full.

Ved at anvende denne struktur holdes Activity-klassen ren for avanceret tilstandslogik. Aktiviteten reagerer blot på ændringer i deltagerantallet, mens selve logikken for at oprette og skifte mellem tilstandsobjekter er indkapslet korrekt i klasserne for State-mønstret.

5.7. Database

For at kunne holde på data i systemet uden at Serveren skal køre 24/7 er der lavet en database. Som beskrevet i tidligere Design afsnit anvendes der DAO filer for at holde styr på hvordan dataen gemmes. Her er der lavet DAO filer for hver element på Figur 12 dette kan ses i Design afsnitet. Det vil sige der er lavet.

- ActivityDAO
- ActivityEquipmentDAO
- EquipmentDAO
- MemberDAO
- ParticipantDAO
- RoomDAO
- WaitListDAO

Her er der lavet et interface og en klasse som implementerer interfacet disse kan findes i Bilag[I] under repository packen. Dog for at kunne gemme data på en database skal der oprettes en forbindelse her er der valgt at bruge NeonDB for at holde på data. For at forbinde til neonDB er der lavet en Singleton connection klasse som alle dao implementations filer denne kan også ses i design afsnitet på Figur 13.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class DatabaseConnection
{
  private static DatabaseConnection instance;
  private Connection connection;

  private DatabaseConnection() throws SQLException
  {
    DriverManager.registerDriver(new org.postgresql.Driver());
    this.connection = DriverManager.getConnection(
        "jdbc:postgresql://ep-polished-bonus-al3ktuj0-pooler.c-3.eu-central-1.aws.
neon.tech/neondb",
        "neondb_owner", "npg_NW5bHvMsV3GI");
  }
  public static synchronized DatabaseConnection getInstance()
      throws SQLException
  {
    if (instance == null)
    {
      instance = new DatabaseConnection();
    }
    return instance;
  }
  public Connection getConnection()
  {
    return connection;
  }
}
```

Kodestykke 26: Database connection kode

Som ses på Kodestykke 26 er der gjort brug af singleton her er der en private static instance, en private konstruktør og en public static synchronised metode som enten skaber instance eller returnerner den eksisterende instance. Denne kode er skrevet for at der ikke skal skrives den samme kode følgende DRY. Her er det vigtigt at den er synchronized sådan at der kun oprettes forbindelse trådsikkert en gang.

```
public interface ActivityDAO
{
  Activity create(String title, String description, int duration,
      LocalDateTime time, int capacity, String roomId, ArrayList<Equipment>
equipment, Instructor instructor) throws SQLException;
  Activity readByPrimaryKey(String title, String description,
      LocalDateTime time) throws SQLException;
  void updateActivityInfo(String oldTitle, String oldDescription,
      LocalDateTime oldTime, Activity activity) throws SQLException;
  void delete(Activity activity) throws SQLException;
  ActivityList getAllActivities(ArrayList<Room> rooms) throws SQLException;
}
```

Kodestykke 27: Activity DAO Interface

Det ovenstående Kodestykke 27 viser interfacet for activityDAO her er der metoder for at kunne oprette, slette, hente ogdatere elementer i databasen. Dette gøres via anvendelsen af primary keys, newValues og oldValues.

```
 @Override
  public Activity create(String title, String description, int duration,
LocalDateTime time, int capacity, String roomId, ArrayList<Equipment> equipment,
Instructor instructor) throws SQLException {
    Connection connection = DatabaseConnection.getInstance().getConnection();
    try (PreparedStatement statement = connection.prepareStatement(
            "INSERT INTO Activity(title, description, duration, time, capacity,
roomid, instructor_username) VALUES (?, ?, ?, ?, ?, ?, ?)")) {
      statement.setString(1, title);
      statement.setString(2, description);
      statement.setInt(3, duration);
      statement.setObject(4, time);
      statement.setInt(5, capacity);
      statement.setString(6, roomId);
      statement.setString(7, instructor.getUsername());
      statement.executeUpdate();
      // tilføjelse af equipment på activity er undladt i dette eksempel
      }
      return new Activity(title, description, duration, time, equipment, new
Room(roomId), capacity, instructor);    }
  }
```

Kodestykke 28: Activity DAO Create

Ovenstående Kodestykke 28 viser hvordan at ActivityDAOimp filens create metode ser ud her kan ses at det første som gøres er at oprette forbindelse til databasen ved at bruge DatabaseConnection().getInstance().getConnection(), herefter vil den lave et PreparedStateent hvor at SQL statement bliver oprettet med de nødvendige variabler. Her oprettes en aktivitet ved at give en titel, beskrivelse, varighed, tid, kapacitet, rum og instruktør. Disse variabler kan sendes over databasen vha. statement.executeUpdate().

```
  @Override public Activity readByPrimaryKey(String title, String description,
      LocalDateTime time) throws SQLException
  {
    Connection connection = DatabaseConnection.getInstance().getConnection();
    try (PreparedStatement statement = connection.prepareStatement(
        "SELECT * FROM Activity WHERE title = ? AND description = ? AND time = ?");)
    {
      statement.setString(1, title);
      statement.setString(2, description);
      statement.setObject(3, time);

      try (ResultSet resultSet = statement.executeQuery())
      {
        if (resultSet.next())
        {
          String instructorUsername = resultSet.getString("instructor_username");
          Instructor instructor = null;
          if (instructorUsername != null) {
            instructor =
MemberDAOImp.getInstance().getInstructor(instructorUsername);
          }
          return new Activity(resultSet.getString("title"),
              resultSet.getString("description"), resultSet.getInt("duration"),
              resultSet.getObject("time", LocalDateTime.class), null, null,
              resultSet.getInt("capacity"),instructor);
        }
      }
    }
    return null;
  }
```

Kodestykke 29: Activity DAO Reading

Ovenstående Kodestykke 29 viser hvordan at man kan hente information fra databasen her bruges primary key på en aktivitet for at få et resultset tilbage fra databasen, her kan man prøver at skabe en ny aktivitet for at returnere denne her har man dog et tilfælde af at en dao fil anvender en anden DAOfil for at kunne hente en instruktør siden at man ikke kan have en aktivitet uden en instruktør. På Kodestykke 27 kan ses at der er en metode getAllActivities() som returnerer alle aktiviteter. Dette anvendes på model siden for at hente alle aktiviteter ved opstart af serveren.

```
private PropertyChangeSupport property;
    private MemberList memberList;
    private ActivityList activityList;
    private ArrayList<Room> rooms;
    private ArrayList<Equipment> equipment;

    public Gym() throws SQLException
    {
      this.property = new PropertyChangeSupport(this);
      this.memberList = MemberDAOImp.getInstance().getAllMembers();
      this.rooms = RoomDAOImp.getInstance().getAllRooms();
      this.activityList = ActivityDAOImp.getInstance().getAllActivities(rooms);
      this.equipment = EquipmentDAOImp.getInstance().getAllEquipments();
      loadActivityDetails();
    }
    private void loadActivityDetails()throws SQLException{
      for (int i = 0; i < activityList.getActivities().size(); i++) {
        Activity activity = activityList.getActivities().get(i);
        loadParticipants(activity);
      }
    }
    private void loadParticipants(Activity activity)throws SQLException{
      List<Member> members =
ParticipantDAOImp.getInstance().getMembersForActivity(activity);
      List<Member> waitingMembers =
WaitListDAOImp.getInstance().getWaitListForActivity(activity);
      for (int i = 0; i < members.size(); i++) {
        activity.addParticipant(members.get(i));
      }
      for (int i = 0; i < waitingMembers.size(); i++) {
        activity.addParticipant(waitingMembers.get(i));
      }
    }
```

Kodestykke 30: Datainitialisering på serverside modelpackage

Ovenstående Kodestykke 30 viser hvordan at dataen bliver indlæst på server siden. Her kaldes på DAO implementations klasserne som alle sammen er singleton for at hente data ved opstart. Det vil sige at når serveren starter henter den alt data på engang for at man undgår at skulle kontakte databasen unødvendigt i runtime. Der er undtagelser til dette og databasen opdateres når at der tilføjes eller fjernes medlemmer fra databasen.

```
@Override public void removeFromParticipantList(Member member,
    Activity activity)
{
  activity = getActivity(activity.getTitle(), activity.getDescription(),
      activity.getTime());
  if (member == null || activity == null)
  {
    throw new IllegalArgumentException("Member and activity cannot be null");
  }
  try
  {
    if(activity.removeParticipant(member))
    {
      ParticipantDAOImp.getInstance().removeParticipant(member, activity);
    }
    else
    {
      WaitListDAOImp.getInstance().removeFromWaitList(member, activity);
    }
    property.firePropertyChange("RemoveMemberFromActivity", member, activity);
  }
  catch (SQLException e)
  {
    throw new RuntimeException(e);
  }
}
```

Kodestykke 31: Databasehåndtering ved fjernelse af medlemmer

Ovenstående Kodestykke 31 viser hvad der sker hvis man vælger at fjerne sin registrering på en aktivitet, angående på om man er på medlemslisten eller ventelisten bliver man fjernet ved at kalde på DAO implementation filen og fjernet fra databasen. Dette gøres med det samme for at dataen på model og database altid er den samme og ikke kommer ud af sync. Alt oprettelse og fjernelse af aktiviteter, rum, instruktører og redskaber fungerer på samme måde og anvender getInstance() kald for at tilgå databasens funktionaliteter.

6. Test

6.1. Introduktion

I dette kapitel redegøres der for de test, der er udført på holdtræningssystemet. Formålet med testfasen er at sikre systemets kvalitet, stabilitet og at det opfylder de opstillede krav. Der tages udgangspunkt i V-modellen, og der er valgt en teststrategi, der dækker systemet på både et detaljeret kodeniveau og et overordnet brugerniveau. Kapitlet er bygget op omkring Test via usecases (Systemtest), Unit-test og Integrationstest.

6.2. Test-cases

I dette afsnit testes systemet ud fra use-cases for at sikre, at alle stier i systemet fungerer som forventet. Der tages udgangspunkt i UC1: Tilmeld Hold. De resterende test-cases kan findes i Bilag[J]. Formålet med at anvende testscenarier er systematisk at verificere, at systemets funktionalitet i praksis opfylder de definerede krav.

Nedenstående bliver der vist tre testscenarier, som bruges til at verificere, at alle trin i processen udføres korrekt. De tre scenarier er udvalgt, fordi de tilsammen dækker både hovedflowet og de mest centrale alternative forløb i UC1: Tilmeld Hold. TC1 tester det normale succesfulde forløb, hvor et medlem tilmelder sig et hold med ledige pladser. TC2 dækker det hyppigste alternative flow, hvor brugeren aktivt vælger at annullere tilmeldingen i bekræftelsesdialogen. TC6 er valgt for at teste systemets håndtering af et fuldt hold, hvor medlemmet i stedet skal tilbydes en plads på ventelisten. Disse tre scenarier repræsenterer dermed både det forventede brugerflow og de vigtigste afvigelser, som systemet skal kunne håndtere stabilt. Hvis alle tre scenarier gennemføres som forventet, kan implementationen betragtes succesfuld.

Signaturforklaring:

- **R** = Regular (Normalt flow)
- **T** = Tom (Ingen handling/fejl)
- **ALT** = Alternativ sekvens
- **-** = Ikke relevant for dette scenarie

Testscenarier for UC1: Tilmeld Hold

Trin	TC1	TC2	TC3	TC4	TC5	TC6
1. Medlem vælger et hold	R	R	R	R	T	R
2. Vælger 'detaljer'	R	R	R	R	-	R
3. System viser tilmeldingsmulighed	Vist	Vist	Vist	Vist	-	Vist
4. Vælger 'tilmeld hold'	R	R	R	T	-	R
5. System validerer plads	OK	OK	OK	-	-	Fuldt
6. Bekræftelsesdialog vises	Vist	Vist	Vist	-	-	Venteliste
7. Valg af Bekræft/annuller	Bekræft	Annuller	T	-	-	Bekræft
8. Hold opdateres	Tilmeldt	Annulleret	-	-	-	Venteliste

Tabel 4: Testscenarie

Tabel 4 illustrerer seks specifikke testscenarier (TC1-TC6) for use-casen "Tilmeld Hold", som trin-for-trin tester systemets reaktion på brugerens handlinger fra start til slut. Det antages, at processen stopper, hvis et trin ikke gennemføres. TC6 dækker den alternative sekvens, hvor holdet er fuldt, og medlemmet tilmeldes ventelisten i stedet.

Oversigt over testscenarierne:

- **TC1:** R, R, R, R (Fuldt gennemløb - tilmelding lykkedes)
- **TC2:** R, R, R, Annuller (Annuller ved bekræftelsesdialog)
- **TC3:** R, R, T (Annuller ved bekræft-knappen)
- **TC4:** R, T (Ingen tilmeldingshandling valgt)
- **TC5:** T (Intet hold valgt)
- **TC6:** R, R, R, ALT[2] (Fuldt hold → tilmeld venteliste)

ALT[*] gælder for alle trin - brugeren kan altid annullere. Dette er dækket af TC2, TC3 og TC4. Forudsætningen for UC1 er, at medlemmet er logget ind (UC4 skal være implementeret først).

Nedenfor gennemføres de primære testcases (TC1, TC2 og TC6) for at verificere systemets funktionalitet i praksis.

TESTCASE 1 (Hovedscenarie - Succesfuld tilmelding)

Trin	Inputfelt	Værdi	Forventet resultat	Faktisk resultat
1	Hold-liste	Box Hold	Hold vises på listen	Hold vises på listen
2	Detalje-knap	Klik 'detaljer'	Detaljeside vises	Detaljeside vises
3	(System)	-	Tilmeld-knap vises	Tilmeld-knap vises
4	Tilmeld-knap	Klik 'tilmeld hold'	Systemet påbegynder tilmelding	Systemet påbegynder tilmelding
5	(System)	-	Plads valideres - OK	Plads valideres - OK
6	(System)	-	Bekræftelsesdialog vises	Bekræftelsesdialog vises
7	Bekræft-knap	Klik Bekræft	Tilmelding gennemføres	Tilmelding gennemføres
8	(System)	-	Medlem tilføjet på holdets medlemsliste	Medlem tilføjet på holdets medlemsliste

Tabel 5: TESTCASE 1

Tabel 5 verificerer hovedscenariet, hvor et medlem tilmelder sig et hold med ledige pladser.

Den bekræfter, at systemet korrekt validerer pladsen, viser bekræftelsesdialogen og gennemfører handlingen ved brugerens accept.

Til sidst sikres det, at systemet opdaterer dataen og tilføjer medlemmet korrekt til holdets medlemsliste.

TESTCASE 2 (Alternativ sekvens - Annullering)

Trin	Inputfelt	Værdi	Forventet resultat	Faktisk resultat
1	Hold-liste	Box Hold	Hold vises på listen	Hold vises på listen
2	Detalje-knap	Klik 'detaljer'	Detaljeside vises	Detaljeside vises
3	(System)	-	Tilmeld-knap vises	Tilmeld-knap vises
4	Tilmeld-knap	Klik 'tilmeld hold'	Systemet påbegynder tilmelding	Systemet påbegynder tilmelding
5	(System)	-	Plads valideres - OK	Plads valideres - OK
6	(System)	-	Bekræftelsesdialog vises	Bekræftelsesdialog vises
7	Annuller-knap	Klik 'annuller'	Tilmelding annulleres, ingen ændring	Tilmelding annulleres, ingen ændring
8	(System)	-	Medlem tilføjes ikke på holdets medlemsliste	Medlem tilføjes ikke på holdets medlemsliste

Tabel 6: TESTCASE 2

Tabel 6 undersøger det alternative flow, hvor et medlem starter tilmeldingsprocessen, men vælger at afbryde den. Den tester, at systemet udløser bekræftelsesdialogen, men respekterer brugerens valg, når der trykkes "annuller". Testen sikrer dermed, at tilmeldingen afbrydes fuldstændigt, og at medlemmet ikke tilføjes på holdets liste.

TESTCASE 6 (Alternativ sekvens - Venteliste)

Trin	Inputfelt	Værdi	Forventet resultat	Faktisk resultat
1	Hold-liste	Fuldt booket hold	Hold vises som 'fuldt'	Hold vises som 'fuldt'
2	Detalje-knap	Klik 'detaljer'	Detaljeside vises	Detaljeside vises
3	(System)	-	Venteliste-mulighed vises	Venteliste-mulighed vises
4	Venteliste-knap	Klik 'tilmeld venteliste'	Bekræftelsesdialog vises	Bekræftelsesdialog vises
5	Bekræft-knap	Klik Bekræft	Medlem tilføjes på ventelisten	Medlem tilføjes på ventelisten
6	(System)	-	Venteliste opdateres på server og DB	Venteliste opdateres på server og DB

Tabel 7: TESTCASE 6

Tabel 7 evaluerer systemets adfærd, når et medlem forsøger at tilmelde sig et hold, der er fuldt booket. Den bekræfter, at systemet korrekt identificerer holdet som fuldt og i stedet tilbyder en plads på ventelisten frem for en normal tilmelding. Afslutningsvis sikres det, at medlemmet ved bekræftelse tilføjes ventelisten, og at denne handling opdateres korrekt på både server og i databasen.

6.3. Unit-test

Der anvendes Black-box tests for at sikre, at systemet opfylder de specificerede krav. Testene er designet ud fra forventet adfærd (input og output) frem for intern implementering, og formålet er at verificere, at systemet håndterer gyldige og ugyldige inputs korrekt.

Derudover anvendes White-box testing for at sikre, at kodelogikken fungerer korrekt. Formålet er at opnå dækning af logikken i systemet frem for udelukkende at verificere input/output. Testene er derfor designet ud fra en forståelse af implementeringen

6.3.1. Unit-Testcase

Testcases for metoden addToParticipantList er designet ud fra ZOMBIE-principperne for at sikre dækning af både normale og fejlscenarier.

Alle Unit-Testcases findes i Bilag[K]

AddparticipantList

- Z:

null member → IllegalArgumentException null activity → IllegalArgumentException null member + nullactivity → IllegalArgumentException

- O:

1 gyldigt member + 1 aktivitet → medlem tilføjes korrekt

- M:

samme member tilmeldes flere gange → IllegalStateException flere members til samme aktvitet → alle tilføjes korrekt samme member til flere aktiviteter → tilføjes korrekt

- B:

aktivitet med kapacitet = 1 → første member accepteres aktivitet med kapacitet = 1 → første member afvises

- E:

duplicate registration → IllegalStateException null input → IllegalArgumentException

6.3.2. Blackbox

Det nedenstående Kodestykke 32 verificerer, at systemet ikke tillader, at den samme bruger bliver tilmeldt den samme aktivitet flere gange.

```
 @Test void exception_alreadyRegistered_throwsIllegalStateException()
    throws Exception
{
  Member member = createValidMember("testReg8");
  gym.createMember(member);
  Activity activity = createValidActivity("Yoga", FIXED_TIME);
  gym.addActivity(activity);

  gym.addToParticipantList(member, activity);

  assertThrows(IllegalStateException.class,
      () -> gym.addToParticipantList(member, activity));
}
```

Kodestykke 32: Blackbox test

Som det kan ses, oprettes der først en gyldig bruger testReg8 og en aktivitet Yoga, hvorefter brugeren tilmeldes aktiviteten en gang via gym.addToParticipantList(). Derefter forsøges det at tilmelde den samme bruger til den aktivitet igen. Ifølge systemets kravspecifikation skal dette afvises, og systemet skal kaste en IllegalStateException.

Valget af IllegalStateException frem for IllegalArgumentException er bevidst, inputtet i sig selv er gyldigt, da både bruger og aktivitet eksisterer i systemet. Det er derimod systemets tilstand der er ugyldig, da bruger allerede er registreret. Det vil sige at IllegalStateException er det korrekte valg.

Testen verificerer derudover at databasen forbliver konsistent efter en exception, at der stadig kun eksisterer en registrering i Participant tabellen. Dette er vigtigt fordi en fejl her kunne betyde at en bruger optager flere pladser og dermed forhindrer andre i at tilmelde sig.

I en ZOMBIE-sammenhæng placerer denne test sig under E (Exception), da den tester et fejlscenarie der opstår ved ugyldig systemtilstand. Den dækker dog også M (Many), da den tester, hvad der sker, når samme specifikke handling bliver gentaget.

De resterende Black-box test findes i Bilag[L] og Bilag[M].

6.3.3. Whitebox

Denne test, som er blevet vist på Kodestykke 34 tager udgangspunkt i en direkte analyse af implementationen af addToParticipantList. Metoden indeholder fire kontrolflow-branches som testen er blevet designet ud fra

```
@Override public void addToParticipantList(Member member, Activity activity)
 {
   if (member == null || activity == null)
   {
     throw new IllegalArgumentException("Member and activity cannot be null");
   }
   if(activity.getParticipantList().contains(member))
   {
     throw new IllegalStateException("Cannot register. You're already on the
activity");
   }
   try
   {
     if(activityList.getActivity(activity.getTitle(), activity.getDescription(),
             activity.getTime()).addParticipant(member))
     {
       ParticipantDAOImp.getInstance().addParticipant(member, activity);
     }
     else
     {
       WaitListDAOImp.getInstance().addToWaitList(member, activity);
     }
     property.firePropertyChange("ParticipantAdded", member, activity);
   }
   catch (SQLException e)
   {
     throw new IllegalArgumentException(e);
   }
 }
```

Kodestykke 33: addToParticipantList metode på serversiden

Linje 3-6 udgør Branch 1, en null-guard der kaster IllegalArgumentException hvis enten member eller activity er null. Dette dækkes af testen branch1a_nullMember og branch1b_nullActivity.

Linje 7-10 udgør Branch 2, en duplikat-tjek der kalder activity.getParticipantList().contains(member). Hvis betingelsen er sand, kastes en IllegalStateException. Det er denne branch som nedestående test dækker. Testen tilmelder først member en gang, så contains() returnerer false og member tilføjes. Derefter forsøges tilmelding igen, hvor contains() nu returnerer true og exception kastes.

Linje 13-18 udgør Branch 3 og 4, hvis aktiviteten ikke er fuld, persisteres member i Participant tabellen via ParticipantDAOimp. Hvis aktiviteten er fuld,

placeres member i ventelisten via WaitListDAOImp. Disse branches dækkes af branch3_validMemberAndActivity_memberIsAdded.

Ved at tage udgangspunkt i implementations if statments sikres det at alle grene af logikken testes og ikke kun det forventeded adfærd. Whitebox testen er skrevet fordi vi kan se at contains() på linje 7 eksisterer i koden, og vi ved præcis hvad der skal til for at aktivere den branch

De nævnte branches 1a, 1b og 3 kan findes i Bilag[N]

```
@Test void branch2_memberAlreadyRegistered_throwsIllegalStateException()
  {
    Member member = createValidMember("john1");
    Activity activity = createValidActivity();
    gym.addActivity(activity);

    gym.addToParticipantList(member, activity);

    assertThrows(IllegalStateException.class,
        () -> gym.addToParticipantList(member, activity));
  }
```

Kodestykke 34: Whitebox test

Testen udføres ved først at tilmelde et medlem til en aktivitet korrekt og derefter forsøges at tilmelde det samme medlem til den samme aktivitet. Dette udløser, Ifølge implementering en IllegalStateException fordi systemt forhindrer dubletter.

White-box testen er baseret på en analyse af implementeringen og har til formål at dække forskellige kontrolflows i koden, herunder betingelser i if-statements, for at sikre at alle grene af logikken tests.

Testen er navngivet branch2_memberLreadyRegistered_throwsIllegalStateException for eksplicit at signalere hvilken gren af koden der testes, i modsætning til black-box tests der navngives efter ZOMBIE-kategorier.

Forskellen på denne white-box test og den tilsvarende black-box test er hensigten bag testen. Black-box testen tester ud fra systemets specifikation, at duplilate registreringer skal afvises. White-box testn tester ud fra koden, at den konkrete if-betingelse med contains() tjekket aktiveres og kaster den forventede exceotion. Begge tests bekræfter den samme adfærd, men af forskellige årsager.

Whitebox testene er skrevet efter en gennemgang af implementationens kontrolflow på serversiden Gym. for hver metode med if statements er branches identificeret og dækket med mindst en test per gren. DummyGym blev udviklet specifikt til dette formål, da den gengiver præcis samme logik som den rigtige Gym, men uden databasekald, hvilket gør testene hurtige og isolerede. Dette sikrer at en fejlende whitebox test altid peger på en logikfejl i selve metoden og ikke på netværk eller databasetilstand.

Der findes flere White-box tests i Bilag[N].

6.4. Integrationstest

```
@Test void createActivity_isPersistedInDB() throws Exception
  {
    Activity activity = createValidActivity("Yoga", FIXED_TIME);

    clientGym.addActivity(activity);

    assertTrue(activityExistsInDB("Yoga", "Fitness class", FIXED_TIME));
  }
```

Kodestykke 35: Integrationstest

Som vist på Kodestykke 35 verificerer denne integrationstest, at en aktivitet korrekt bliver oprettet gennem klient og persisteret i databasen via serveren. Testen dækker hele flowet fra client til server til database og sikrer, at integrationen mellem systemets lag fungerer korrekt.

Testen opretter først en aktivitet, Yoga, med et fast timestamp FIXED:TIME via createValidActivity. Derefter kaldes clientGym.addActivity(activity), som sender en request gennem socket-forbindelsen til ActivityClientHandler på serveren, der videredelegerer til server-sidens Gym, som persisterer aktiviteten i databasen via activityDAOImp. Til sidst verificeres det direkte i databasen via en SQL-query i hjælpmetoden.

Den direkte DB-assertion sikrer at alle lag i systemet fungere korrekt, klient, sokcet, handler, model og DAO

Transaktionisolation sikres via connection.setAutoCommit(false) i `@BeforeEach` og connection.rollback() i `@AfterEach`, så alle ændringer rulles tilbage efter hver test. Dette betyder at testen ikke efterlader data i databasen og kan køres gentagne gange uden at påvirke systemets tilstand

Denne test klassificeres som en systemintegrationstest frem for en ren unit test, da den tester samspillet mellem flere lag, klient. netværk server og databse, i stedet fot at isolere en enkelt komponent.

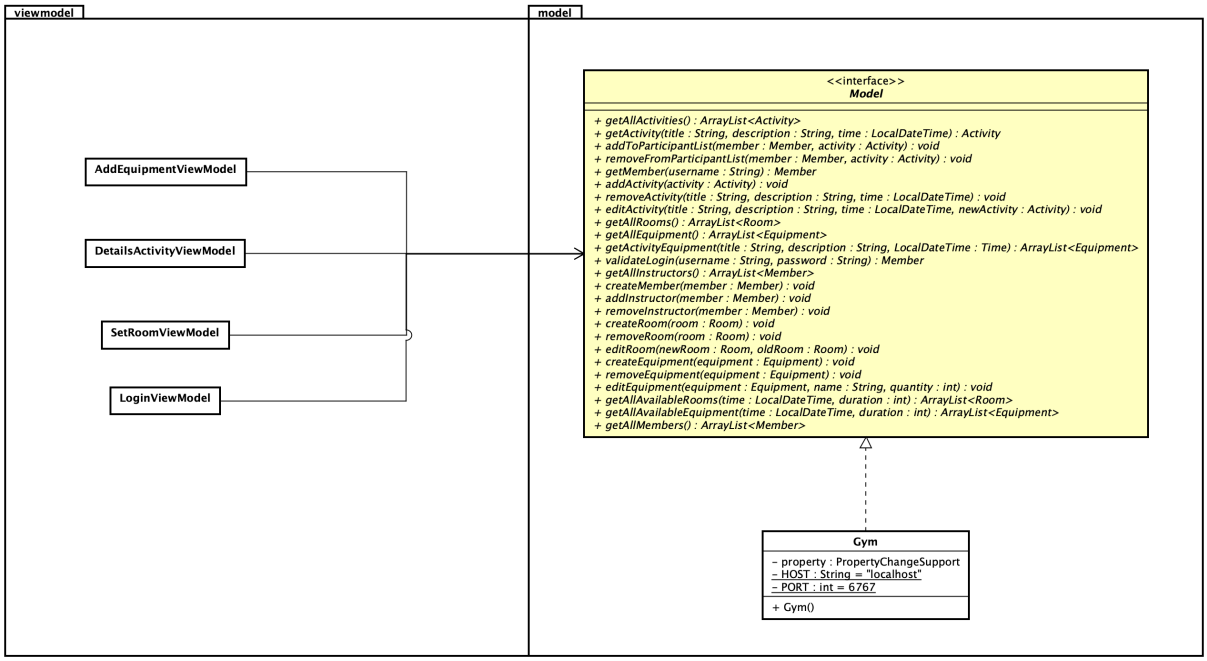
De resterende integrationstest findes i Bilag[M].

7. Diskussion

7.1. Introduktion

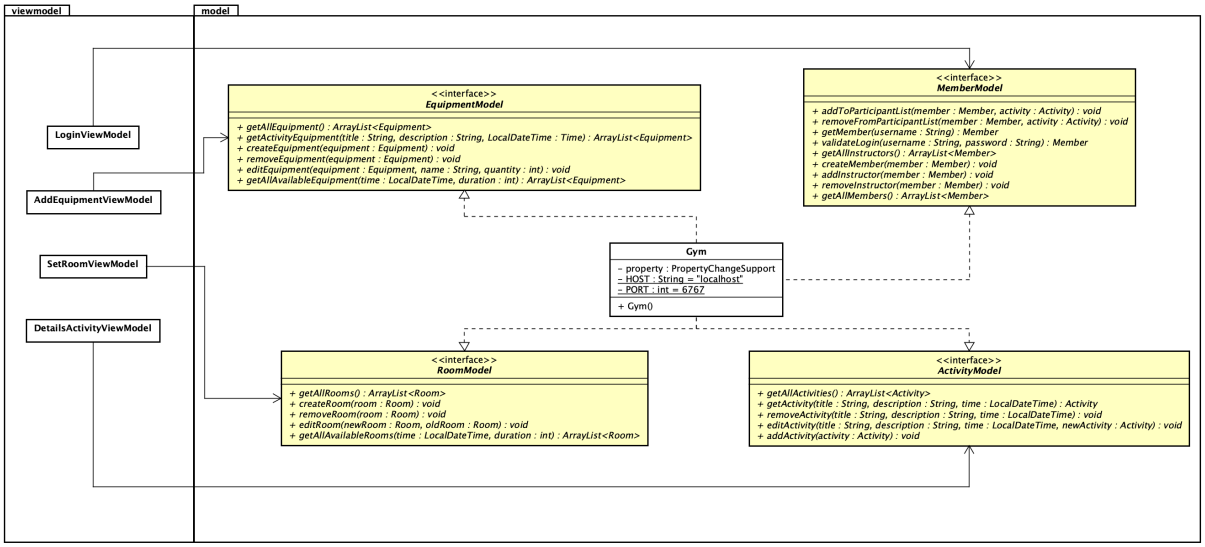
På baggrund af alle de forrige afsnit i rapporten, skrives et diskussions-afsnit, hvor der drøftes valg taget i de forskellige discipliner, der indgår i projektet. Der vurderes om hvorvidt andre valg ville gavne projektet, hvis der skulle videreudvikles, eller laves et lignende igen.

7.2. Interface Segregation



Figur 15: Model-interface brug i systemet (udeladt andre viewmodels)

Som det fremgår af Figur 15, tog det oprindelige designudkast af model-laget udgangspunkt i ét samlet Model-interface, som klassen Gym implementerede. Dette interface indeholder samtlige af systemets forretningsmetoder, lige fra håndtering af udstyr og lokaler til aktiviteter og medlemslogin. Selvom denne tilgang indledningsvist kan virke simpel, afslører en nærmere analyse, at designet bryder med flere centrale objektorienterede designprincipper fra både SOLID og GRASP.



Figur 16: Eksempel på hvordan interface segregation kunne forbedre design

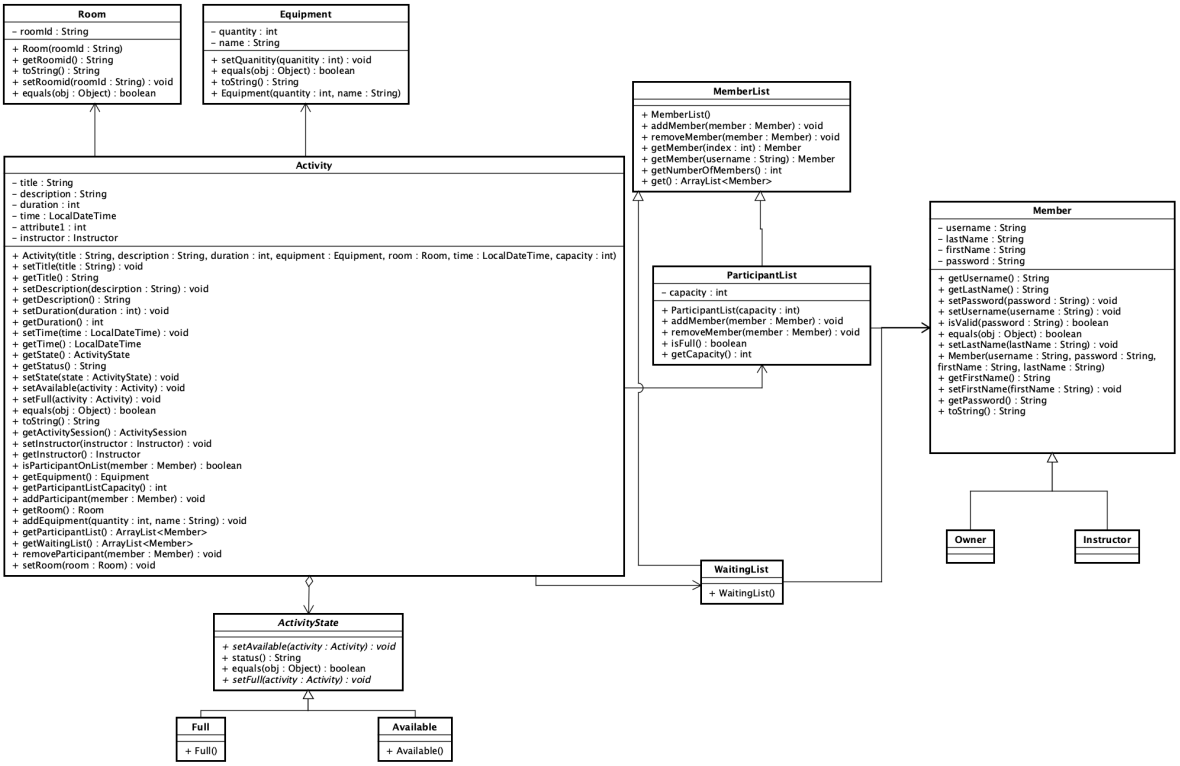
Det primære problem ved designet i Figur 15 er overtrædelsen af Interface Segregation Principle (ISP) fra SOLID. ISP dikterer, at en klient (i dette tilfælde systemets ViewModels) aldrig bør tvinges til at afhænge af metoder, den ikke bruger. I det oprindelige design afhænger eksempelvis LoginViewModel af det fulde Model-interface, hvilket betyder, at den rent strukturelt har kendskab til irrelevante metoder som createRoom() og addEquipment().

Set gennem GRASP-principperne fungerer det store Model-interface nærmest som en "god-klasse", der kender til alt i systemet. Dette medfører en uhensigtsmæssig lav kohæsion, da interfacet mangler ét veldefineret ansvarsområde, samt en problematisk høj kobling, da alle ViewModels er tæt bundet til denne ene, centraliserede kontrakt.

For at imødekomme disse strukturelle svagheder præsenterer Figur 16 et revideret design, hvor ISP er anvendt aktivt til at bryde interfacet op i fire domænespecifikke interfaces: EquipmentModel, MemberModel, RoomModel og ActivityModel. Gym-klassen fungerer fortsat som den underliggende facade, men klienterne eksponeres nu kun for den del af systemet, de reelt har brug for.

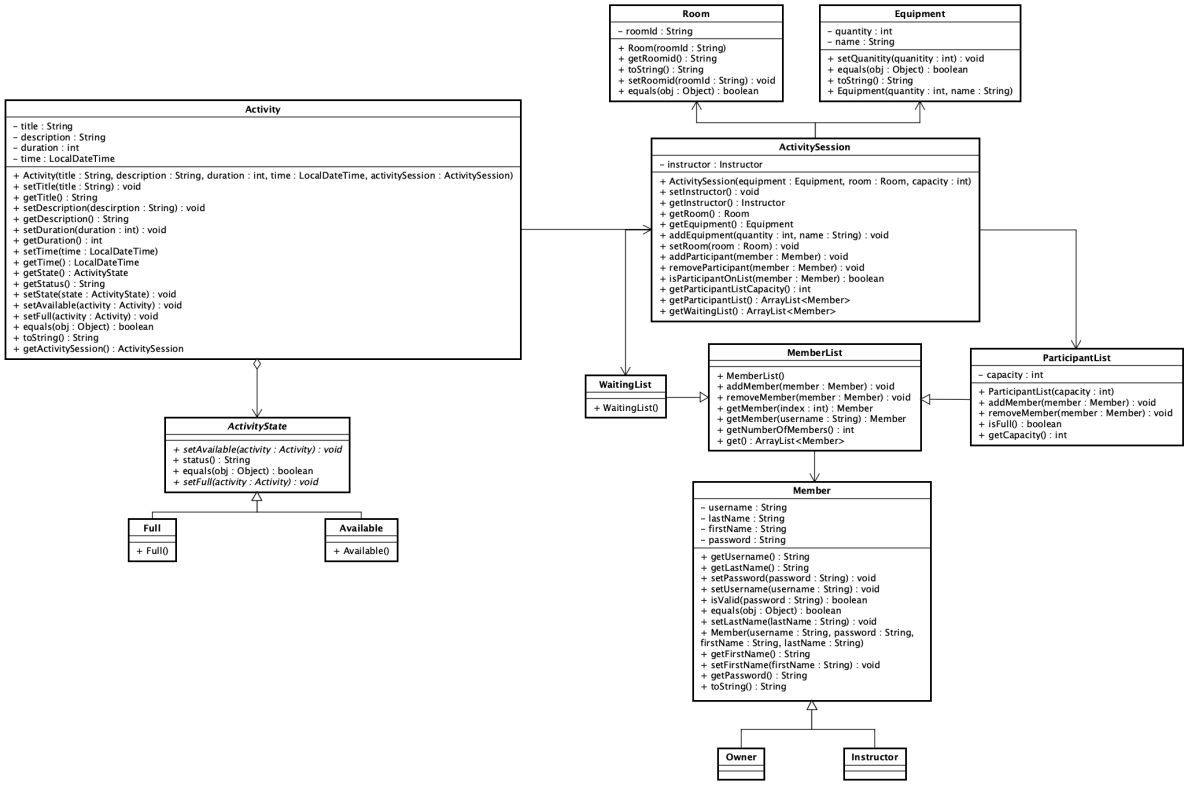
Gennem denne opdeling opnås der en langt lavere kobling, da afhængighederne minimeres eksempelvis kender AddEquipmentViewModel nu udelukkende til EquipmentModel og påvirkes ikke af ændringer i medlemshåndteringen. Samtidig sikres en højere kohæsion, idet hvert interface nu har et logisk og skarpt afgrænset ansvarsområde. Ved at anvende Interface Segregation-princippet opnås derved et mere robust, testbart og vedligeholdelsesvenligt system, der i langt højere grad efterlever standarderne for god softwarearkitektur.

7.3. Single Responsibility Principle & Law of Demeter



Figur 17: Activity fra det nuværende design

Et centralt aspekt af systemets arkitektoniske udvikling er refaktoreringen af den oprindelige, overbelastede Activity-klasse vist i Figur 17. Her agerede klassen som en stor “god-klasse” med ansvar for alt fra stamdata til ressource-håndtering og medlemsregistrering.



Figur 18: Activity opdelt i Activity og ActivitySession

Ved at udskille håndteringen af ressourcer, såsom lokaler, udstyr og instruktører, samt deltager- og ventelister i en dedikeret ActivitySession-klasse, som illustreret i Figur 18, opnås en markant forbedring i forhold til Single Responsibility Principle (SRP). I dette nye design er ansvarsområderne mere fokuserede, idet Activity nu primært indkapsler aktivitetens definition og overordnede tilstand, mens ActivitySession håndterer den operationelle afvikling. Selvom denne opdeling øger kohæsionen i de enkelte klasser, introducerer den dog et nyt, velkendt objektorienteret kompromis: brud på Law of Demeter (LoD) og en deraf følgende høj kobling. For at få adgang til dybereliggende data tvinges systemets klienter nu til at navigere gennem lange kæder af objektreferencer. Et illustrativt eksempel på dette ud fra strukturen i Figur 18 er et potentielt metodekald som activity.getActivitySession().getParticipantList().getCapacity(). Denne type kædede opslag medfører en uhensigtsmæssig høj strukturel kobling, fordi den kaldende klasse ikke længere kun afhænger af Activity, men også påtvinges et indgående kendskab til den interne opbygning og relationerne mellem ActivitySession og WaitingList. Hvis implementeringen af sessions- eller listestrukturen ændres i fremtiden, vil det potentielt kræve rettelser i fjerne dele af kildekoden, der oprindeligt blot skulle aflæse en simpel kapacitet. Sammenligningen mellem Figur 17 og Figur 18 afspejler dermed et klassisk arkitektonisk trade-off: Valget står mellem at overholde SRP for at undgå uoverskuelige klasser og at overholde LoD for at beskytte systemet mod skrøbelige dot-dot notation. For at profitere fuldt ud af den forbedrede ansvarsfordeling uden at lide under den

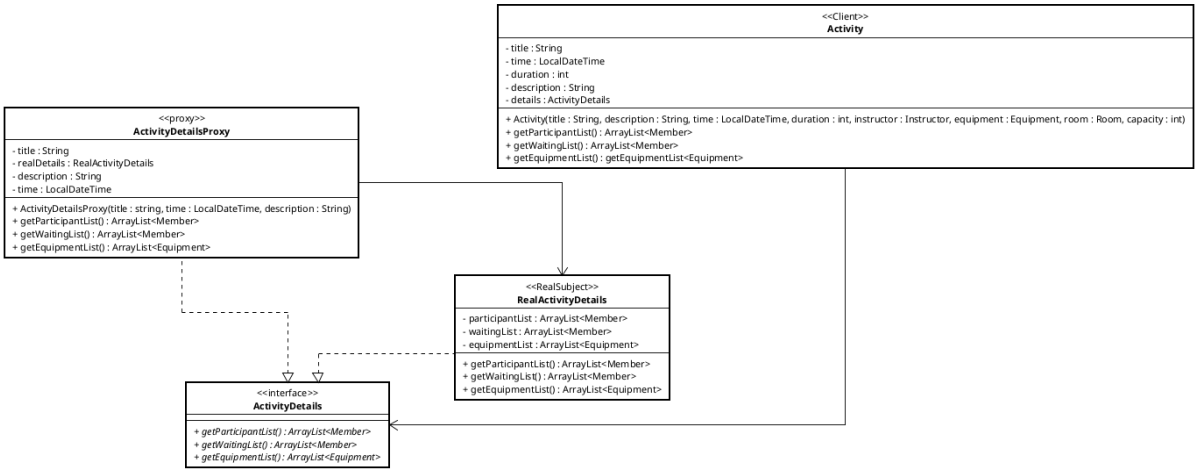
stramme, tværgående kobling, kan det overvejes at introducere strukturer, der indkapsler disse dataudtræk, så klienterne skærmes mod domænemodellens interne kompleksitet.

7.4. Datahåndtering

Som beskrevet i afsnit 4 Implementation er der lavet datahåndtering vha. en database som lagrer informationer så snart et medlem tilmelder sig, et rum bliver tilføjet eller en medarbejder opretter en aktivitet og mm. Der er dog nogle problemer med designet hvis at det skal skaleres op til flere tusind brugere, i den nuværende implementation læses alt applikationsdata ind i serverens hukommelse ved opstart via loadActivityDetails(). Her er fordelen ved den her tilgang at man kan minimere kommunikation med databasen fremadrettet kan minskes ved at man så kun kontakter databasen ved opdatering, tilføjelse, hentning eller fjernelse af et tupple.

Set ud fra et skalerbarheds- og performancemæssigt perspektiv præsenterer denne løsning dog markante begrænsninger i et virkeligt produktionsmiljø. Hvis databasen over tid vokser til at indeholde tusindvis af registrerede medlemmer samt historiske aktivitetsdata, vil serverens initiale opstartstid øges eksponentielt, og RAM-forbruget vil stige drastisk, hvilket truer systemets stabilitet.

Det er muligt at immødekomme Open/Closed principle ved en mere hensigtsmæssig ressourceallokering kan man anvende en virtuel proxy. I stedet for at lade serveren præ-load alle deltagere, medlemmer og rum vil den virtuelle proxy fungere som en pladsholder. Når en klient anmoder om et samlet oversigt hentes i føste omgang kunde basale metadata som titel, tidspunkt og varighed. Det er så først når et medlem vælger et hold for at se uddybende informationer eller for at timlelde sig at den virtuelle proxy vil gribe ind. Her vil proxyen så kunne kalde på WaitListDAO hvilket vil reducere serverens behov for at holde på alt data i RAM.



Figur 19: Virtuel proxy for bedre datahåndtering

Ovenstående figur Figur 19 illustrerer denne alternative løsning og det forbedrede systemflow for datahentning. Arkitekturen er bygget op omkring adskillelsen af letvægts- og tungvægtsdata for at sikre lav kobling og høj performance. Når systemet

initialiseres, oprettes Activity-objektet (Klienten) udelukkende med letvægtsdata. For at håndtere de tunge lister instansieres en ActivityDetailsProxy. Proxyen gemmer kun de attributter, der udgør aktivitetens Composite Primary Key i databasen (titel, tidspunkt og beskrivelse). Som det fremgår af modellen, afhænger Activity-klassen udelukkende af interfacet ActivityDetails. Klassen har intet kendskab til den underliggende proxy eller selve databaselogikken. Dette overholder Dependency Inversion-princippet fra SOLID, da klienten afhænger af en abstraktion frem for en konkret implementering. Når en bruger interagerer med systemet for at se detaljer for et hold, kalder Activity metoden getParticipantList() via interfacet. Dette kald opfanges af ActivityDetailsProxy. Proxyen tjekker først, om det tunge dataobjekt (RealActivityDetails) allerede eksisterer i hukommelsen. Er referencen null, benytter proxyen sin gemte Primary Key til at foretage et databasekald via DAO-laget. Efter databasekaldet instansierer proxyen RealActivityDetails-objektet med de fulde lister og returnerer dataen. Ved efterfølgende anmodninger på samme aktivitet vil proxyen opfange, at objektet allerede er indlæst, og returnere listen øjeblikkeligt uden yderligere netværksbelastning.

Yderligere en af problematikerne kommer fra at det er et multrådet client server system. Hver client får tildelt sin egen tråd via ActivityClientHandler. Det muliggør at flere brugere kan interagere med systemet samtidigt. Hvis man dog kigger på DatabaseConnection så er den implementeret som en Singleton som opretter en enkelt java.sql.Connection, hvilket alle tråde deles om. Her er en standard JDBC-forbindelse ikke designet til at håndtere mange samtidige forespørgsler. En mulig løsning havde været at tilføje connection pooling for at give adgang til en dynamisk håndtering af JDBC connections. Der er dog ikke yderligt blevet tænkt over hvordan dette problem kunne løses.

8. Konklusion

Projektet har haft fokus på de organisatoriske og teknologiske udfordringer, der opstår ved administration af holdtræning i moderne fitnesscentre. Gennem analysearbejdet blev det tydeligt, at koordinering af aktiviteter, medlemmer, instruktører, lokaler og udstyr skaber et behov for et system, der kan samle administrationen og skabe bedre overblik for de involverede aktører.

Undersøgelsen viste, at medlemmer, instruktører og ejere har forskellige behov til et administrativt system. Medlemmer har behov for nem adgang til information om aktiviteter samt mulighed for tilmelding og afmelding af hold. Instruktører har behov for at kunne administrere aktiviteter, mens ejeren har behov for et samlet overblik over instruktører, lokaler og udstyr i fitnesscenteret og mulighed for administration deraf.

På baggrund af analysen blev der udviklet et holdtræningssystem som et klient-server system med JavaFX som brugergrænseflade og en database til opbevaring af data. Systemet understøtter blandt andet login og registrering, oprettelse og administration af aktiviteter, tilmelding og afmelding af hold, ventelister samt administration af lokaler, instruktører og udstyr.

I designet af systemet blev der anvendt MVVM-arkitektur samt designmønstre som Observer, Proxy, Adapter og State. Disse blev anvendt for at skabe en mere struktureret og vedligeholdelsesvenlig løsning med tydelig opdeling mellem systemets lag og ansvarsområder. Hertil blev der lagt stor vægt i at følge SOLID og GRASP.

Projektet viste samtidig, at der opstår flere arkitekturmæssige udfordringer ved udvikling af større systemer. I diskussionsafsnittet blev der blandt andet arbejdet med problemstillinger omkring Interface Segregation Principle, Single Responsibility Principle, Law of Demeter samt håndtering af data og skalering af systemet. Dette viste, at der ofte må foretages kompromiser mellem lav kobling, høj kohæsion og systemets kompleksitet.

Gennem systemtest, unit-test og integrationstest blev det verificeret, at systemets centrale funktioner fungerede korrekt. Testene viste blandt andet, at systemet håndterer synkronisering mellem klienter og server samt korrekt opdatering af database og brugergrænseflade ved ændringer i systemet.

Projektet konkluderer dermed, at et centralt administrationssystem kan forbedre overblik, koordinering og organisering af holdtræning i fitnesscentre.

7. Referencer

BOX12. (2025,). *The Common Fitness Programming Challenges And How To Overcome Them*. BOX12. https://box12fitness.com/the-common-fitness-programming-challenges-and-how-to-overcome-them/

Johansen, M. (2025,). *Automatiseret betalingsystemet*. OneMoneyWay. https://onemoneyway.com/da/blog/automatiseret-betalingssystem/

MOMENTUM OP. (2025,). *How Group Fitness Boosts Mental Health*. MOMENTUM OP. https://momentumop.fitness/articles/how-group-fitness-boosts-mental-health-3/

Salminen, M. A. (2025a,). *Fitness*. Danmarks Nationalleksikon. https://lex.dk/fitness

Salminen, M. A. (2025b,). *Fitnesscenter*. Danmarks Nationalleksikon. https://lex.dk/fitnesscenter

Santini, Z. (2023,). *Mentalt overskud kædes sammen med mere fysisk aktivitet*. SDU. https://www.sdu.dk/da/sif/ugens_tal/ut_8_mental_sundhed_og_fysisk_aktivitet_haenger_sammen

8. Bilag

Bilag A – Use Cases

Bilag B - Domænemodellen

Bilag C – Aktivitetsdiagram

Bilag D – Systemsekvensdiagrammer

Bilag E – Designklassediagram

Bilag F – Protokoller

Bilag G – Statediagram

Bilag H – eerdiagram

Bilag I – Kildekode

Bilag J – Testcases UC2–UC9

Bilag K – Testcases (Oversigt)

Bilag L – Klient-tests

Bilag M – Server-tests

Bilag N - Whitebox-tests

Software Engineering
Via University College
SoftwareEngineering
Banegårdsgade 4
8700 Horsens

Titel:

Procesrapport - Holdtræningssystem

ECTS:

10 ECTS

Semester:

2. Semester

Projektperiode:

4/2/2026 - 27/5/2026

Projektgruppe:

SEP2 Gruppe 2

Studienummer	Navn	Studieretning
362108	Devin Paul Jaworek	SW
361830	Victor Høggaard	SW
362962	Kasra Hojjatifar	SW
362181	Mussa Mohammad	SW
362947	Hussain Abed	SW

Vejledere:

Mona Wendel Andersen, Steffen Vissing Andersen

Sidetal: 20 Antal bilag: 6

Indhold

1. Indledning

Denne procesrapport beskriver arbejdsprocessen bag udviklingen af holdtræningssystemet, som blev udviklet gennem semesterprojektet. Rapporten har fokus på, hvordan vi har arbejdet gennem projektforløbet, hvilke arbejdsmetoder vi anvendte, samt hvilke erfaringer og udfordringer vi oplevede undervejs.

Vi arbejdede gennem hele projektet med Scrum som projektstyringsværktøj, hvor arbejdet blev opdelt i sprints med løbende planlægning, daily scrum møder, sprint reviews og retrospectives. Samtidig anvendte vi Unified Process (UP) som overordnet udviklingsmodel, hvilket betød, at analyse, design, implementering og test blev udviklet iterativt gennem hele projektforløbet.

Gruppen bestod af medlemmer med forskellige kompetencer, arbejdsformer og personprofiler. Dette havde betydning for samarbejdet, kommunikationen og måden opgaver blev fordelt på. Under projektet oplevede vi blandt andet udfordringer med koordinering, planlægning og forskellige arbejdsmetoder, men samtidig bidrog de forskellige profiler også med forskellige styrker til projektet. Derudover arbejdede vi med kulturforståelse gennem projektforløbet, hvor forskelle i blandt andet kommunikation og opfattelse af tid havde betydning for samarbejdet i gruppen.

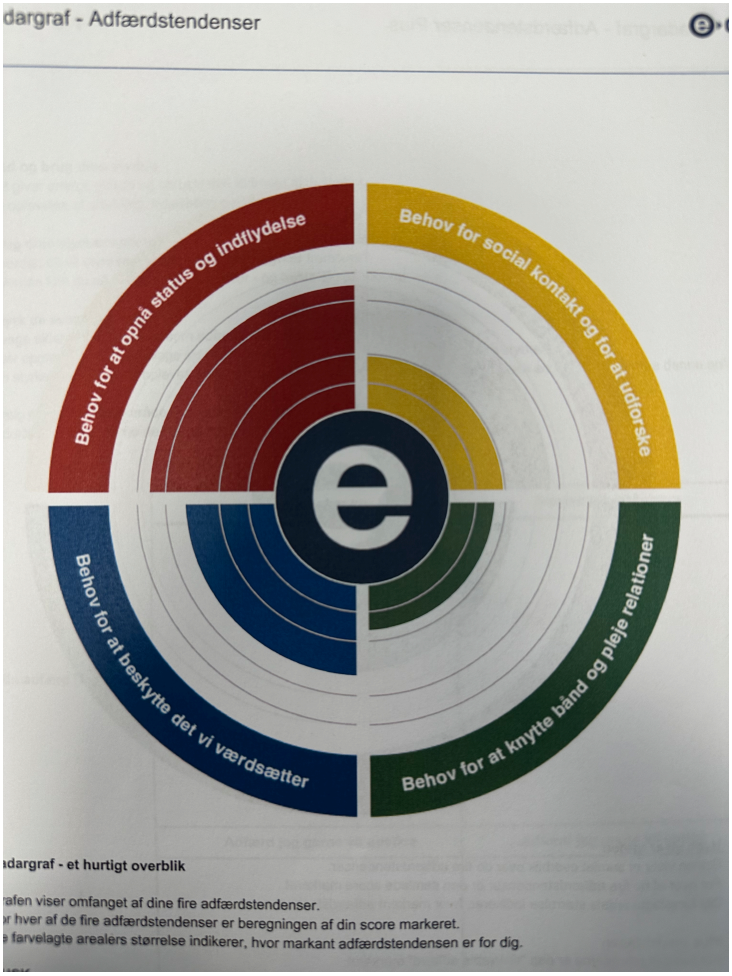
Rapporten indeholder derfor refleksioner over gruppesamarbejdet, anvendelsen af Scrum og UP, samarbejdet med vejleder samt erfaringer med planlægning, kommunikation og arbejdsfordeling gennem projektet.

2. Gruppearbejde og Kulturforståelse

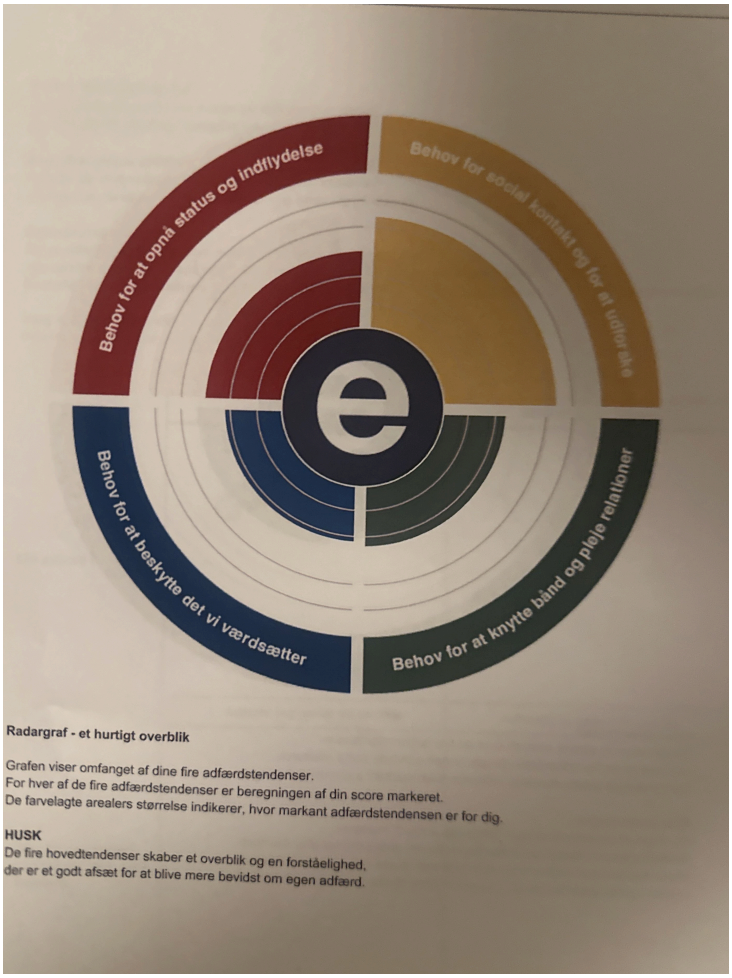
Det her semester har vi i gruppearbejdet haft stor fokus på anvendelsen af Scrum. Her har vi anvendt personprofiler og kulturforståelse, til at fremme forståelse for personlige styrker og svagheder både i arbejdsproces og kommunikation. Dette blev blandt andet brugt til udvælgelse af Scrum-roller.

Vi har fået udleveret personprofiler i starten af semesteret, ved opstart af gruppearbejdet og initiering af projektarbejdet. Her har vi samtidigt fået undervisning i betydningen af personprofiler både styrker og mulige svagheder, samt samarbejde på tværs af forskellige personprofiler, og hvordan der tages højde for det. Ved opstart af projektet er noget af det første vi har lavet, at sammle op på hvilket personprofiler vi har repræsenteret i gruppen.

Gruppens personlighedsprofiler består af:

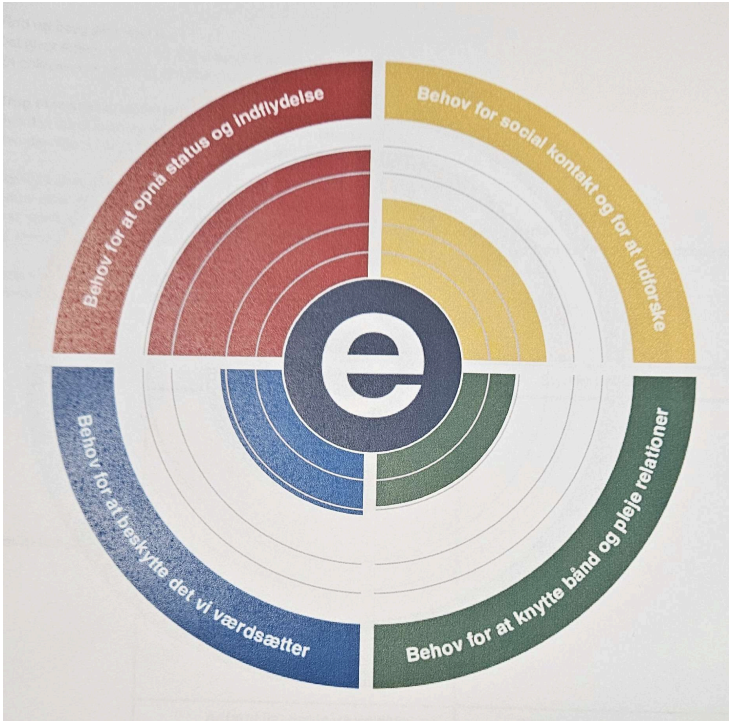


Figur 1: Personprofil tilhørende Victor



Figur 2: Personprofil tilhørende Kasra

Figur 3: Personprofil tilhørende Devin



Figur 4: Personprofil tilhørende Hussein

Som vores personprofiler viser, er vi i gruppen meget rød-domineret.

En personprofil domineret af den røde farve har normalt tendenser til, at være handlekraftige og målrettede. Dette oplevede vi have en stor effekt i starten af projektet, da det førte til at projektinitieringen skete hurtigt og uden tøven. Gruppe-discorden, Youtrack, Github og sharepoint har været sat op lige siden første lektion med grupperne.

Der er blevet valgt at Victor Høggaard skal fungere som Scrum-master og Devin Jaworek som Product Owner. Denne beslutning er blevet taget på baggrund af personprofilernes styrker.

Blandt andet skal Scrum-master stå for de Scrum-cermonierne, dvs. daily-scrum, review og retrospective. Af den årsag kan en rød personprofil være fordelagtig, da den er kendetegnet ved en udadvendt, initiativrig og beslutsom person. Derfor passer den røde profil godt til at stå for møderne, og holde styr på projektets retning, samt meddele gruppemedlemmerne ved dårlig kurs eller lignende.

Product-owner har til opgave, at fordele prioriteringen af backlog-elementer (opgaver), fremme kundens og godkendelse af user-stories ved slutning af et sprint. Her er den blå-personprofil fordelagtig til opgaven, da den er detaljeorienteret, fokuserer på fakta og kvalitetssikring. Dette sikrer, at vores user-stories er veldefinerede, og at acceptkriterierne er tydelige, før vi påbegynder selve udviklingen.

Et konkret eksempel på forskellig adfærd i gruppen kom til udtryk under vores idéudvikling. Her oplevede vi flere gange, at de udadvendte og idérige profiler (Hussein og Kasra) hurtigt kom med en masse idéer. I disse situationer insisterede Devin (Blå) og Mussa (Grøn) på, at vi først skulle have styr på hvad der var muligt, samt af det

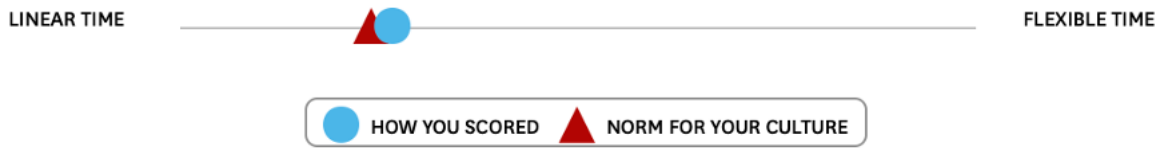
rette ambitionsniveau. Ved at være bevidste om vores profiler kunne vi bruge dette konstruktivt frem for at lade det skabe gnidninger.

Vores individuelle styrker og svagheder har også været tydelige i vores faste Scrum-roller. Som Scrum Master har Victor anvendt sin rød/blå profil til at fastholde strukturen i Scrum-ceremonierne og overholde tidsplanen. En svaghed ved den stærkt røde profil kan dog være mangel på tålmodighed, og det har derfor krævet en bevidst indsats fra Victors side at træde et skridt tilbage under vores retrospectives, så de mere reflekterende profiler også fik taletid. Devin har som Product Owner brugt sin primære blå profil til at udarbejde en stærk og detaljeret product backlog, mens hans sekundære røde farve har hjulpet ham med at skære igennem og træffe endelige beslutninger, når teamet har været i tvivl om prioriteringen af opgaver.

Det er dog ikke kun farven i personprofilerne, der har betydning for gruppearbejdet, men også kulturbaggrund på gruppemedlemmerne.

I gruppen har vi meget forskellige kulturelle baggrunde blandt dette Danmark, Tyskland, Vietnam, Iran, Syrien og Irak. Vi fandt ud, hvordan vores kulturforståelse påvirkede gruppearbejde ved at tage en kutlurprofil-test.

Blandt andet har vi en meget forskellig holdning til nogle af de 8 kulturelle dimensioner, specielt tid.



Figur 5: Kutlurprofil Victor (Tid)

På Figur 5 ses Victors opfattelse af den kulturelle dimension "Tid". Lineær tid er en forståelse af, at tiden og aftaler er meget punktlige. Det vil sige at Victor forventer at man kommer før, eller senest til tiden.

Figur 6: Kutlurprofil Devin (Tid)

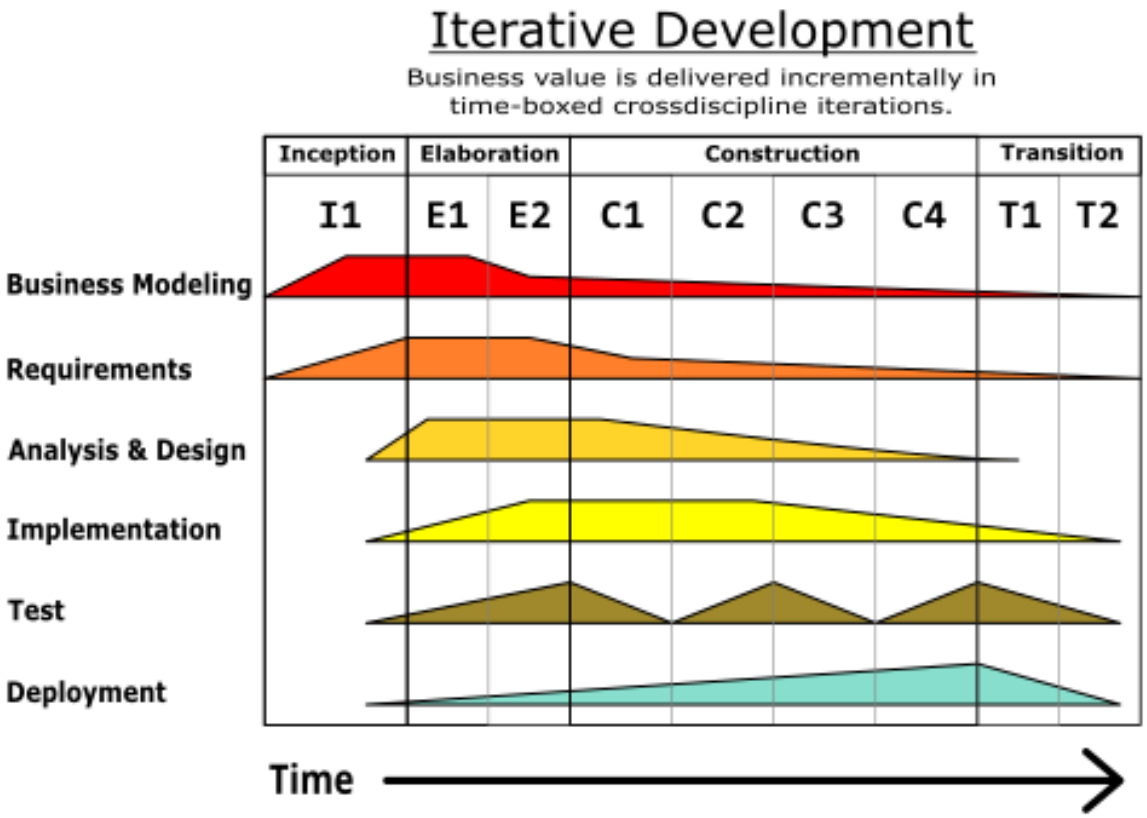
På Figur 6 ses Devins opfattelse af tid. Han har en mere fleksibel tilgang til tid.

Denne difference på opfattelse af tid, har ført til misforståelse i forhold til mødetidspunktet, da der står i vores gruppekontrakt Bilag[A], at vi skal møde kl 9:00 med 10 akademiske minutter, hvilket Victor opfatter som for sent efter kl 9, hvorimod Devin opfatter det som for sent efter 9:10.

Resten af vores kulturprofiler kan findes i Bilag[B]. I vores kulturprofiler har vi alle valgt Danmark, så vi kan tydeligt se differencen i vores forståelse af de 8 kulturelle dimensioner.

3. Unified Process & Scrum

Unified process (UP) er den overordnede ramme for projektet, som er blevet fulgt ved at bruge Scrum som projektstyringsværktøj. Vi har fulgt UP's 4 faser og vores arbejde i de enkelte faser vil i dette afsnit blive uddybet.

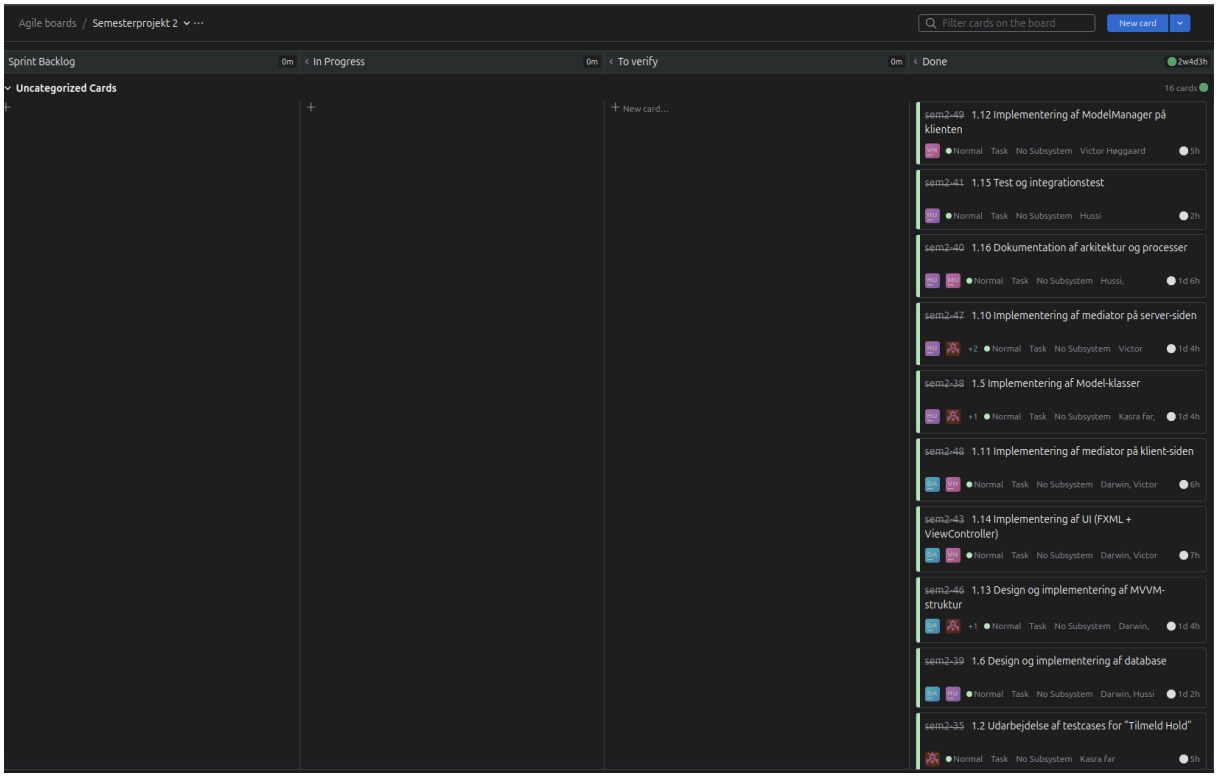


Figur 7: Unified Process

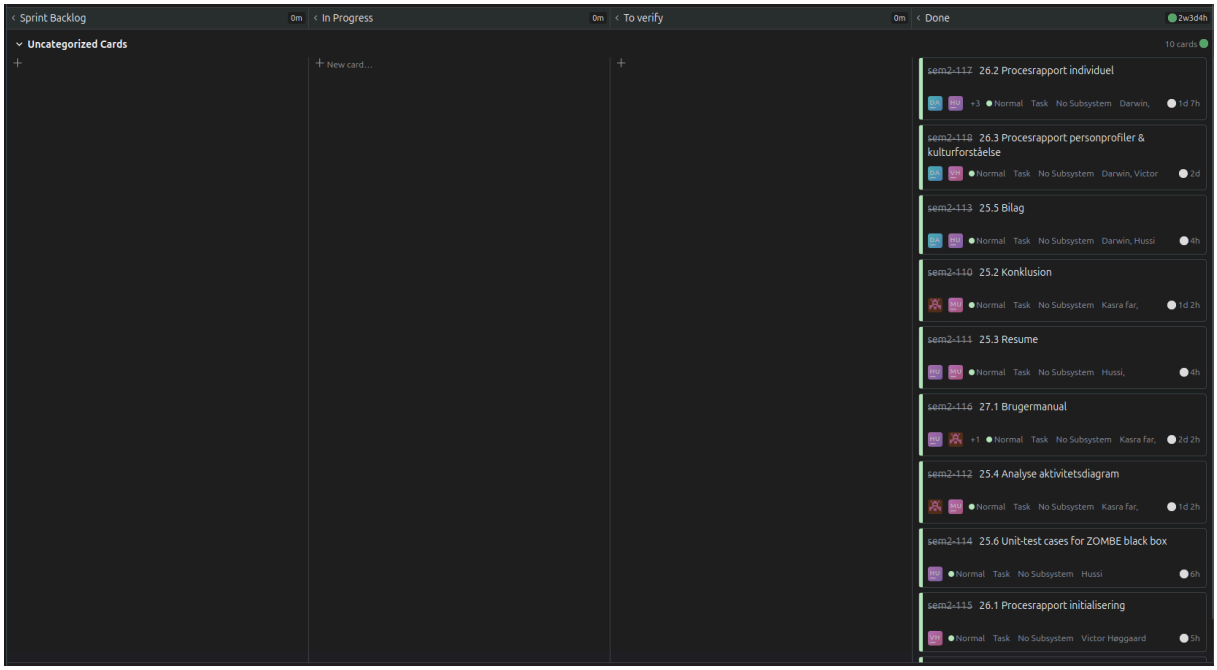
Ovenstående Figur 7 viser den fulgte UP-model. UP har fungeret som en iterativ proces skabelon, der er fulgt i takt med at vi kom videre i projektet.

UP blev anvendt som en guide for, hvor meget de forskellige discipliner fyldte i løbet af vores projekt. I takt med at det gav mening at skifte fokus på andre discipliner anså vi det som, at vi skiftede UP-fase. Det er den måde vi har anvendt UP på.

I inception-fasen har der været fokus på et kort og udetaljeret første udkast, for at få et overblik. Her lavede vi for-analyse (problemfelt, afgrænsning og problemformulering), domænemodel, user-stories, ikke-funktionelle krav, use-case diagram, kort format af use case beskrivelser, klassediagram, system-sekvensdiagram, tilstandsdiagram, kode-eksperimenter i form af database-integrations test og test deraf. Dette blev brugt til at danne et overblik over, hvad projektet dækkede over, og det meste af det blev kasseret til næste sprint, hvor vi måtte starte forfra på blandt andet domænemodellen.



Figur 8: Youtrack Backlog Sprint 1



Figur 9: Unified Process

På Figur 8 kan man se, hvordan analyse, design, business modeling og implementation har fyldt mest. Dette skyldes at det er et elaboration-sprint, som er kendetegnet for at have fokus på disse discipliner. Dette er sprint 1, for inception-fasen foregik uden backlog. I de første to sprints, lignede backloggen meget hinanden, som på Figur 8.

Hvis man sammenligner det med Figur 9 er fokus fuldstændig ændret, og ingen af opgaverne er nu i hverken implementation eller design, nu er fokus skiftet til leverancer, fordi det er et transition-sprint. Dette viser den store kontrast i projektet, ved forskellige faser.

Vi har i alt brugt 8 sprints, hvor sprint 1-2 har været elaboration, 3-5 har været construction og 6-8 har været transition. Dette er blevet vurderet på baggrund af vores disciplin-fordeling.

Vi har igennem projektet siden start af inception-fasen anvendt Scrum. Vi havde tidligere i projektet oprettet vores user-stories og kunne derfra skabe en product backlog som prioriteres og deles om til opgaver som skal fuldføres. Under Sprint-planning møderne er der blevet dannet product-backlog, hvor der er blevet valgt, hvor mange user-stories, der skulle laves og hvilke acceptkriterier, der skal til, for at udføre user-storien, for at opgaven kunne godkendes til, at være verified og kunne vises til Product Owner, ved review-møde.

1	7	Design af protokol for tilmelding af hold 6 timer Acceptkriterier - Der er lavet kommunikationsprotokol som dækker klient-server kommunikationen. - Protokollen dokumenterer både succesfulde og exception scenarier. - Protokoldokumentation via diagram. - Kommunikation mellem klient og server er via JSon. - Gennemgået og godkendt af teamet.	Devin, Victor	Ikke startet

Figur 10: Sprint backlog med acceptkriterier

viser et sprint-backlog element for userstory 1, hvor der er sat acceptkriterier, timeantal, ansvarlige og status. Den resterende product backlog kan findes i Bilag[C]. Vi har anvendt sprint-backlog elementerne ved at tilføje dem til vores YouTrack, som har gjort det muligt for at holde styr på deres status, og hvem der er ansvarlige for opgaven. Når en opgave har været bedømt færdig af ansvarlige er den blevet trukket ind i to-verify, hvorefter opgaven er blevet vurderet af gruppen, efter næste daily standup, hvis den blev godkendt, blev den flyttet til Done.

I forbindelse med anvendelsen af Scrum har der været en række scrum ceremonier som er blevet overholdt, som beskrevet tidligere er der lavet sprint planning møder efter afslutningen af et sprint. Der er dog også lavet Sprint Review møder og Sprint retrospektiv møder. Sprint review mødet har inkluderet Product owner, hvor product owner har skullet accepterer færdiggørelsen af user-stories. De første 2 Sprints har dog været udfordret af at istedet for at Product Owner har godkendt user-storien har product owneren godkendt på accept-kriterier udformet af user-storien. Det vil sige at hvis et accept-kriterie har manglet noget eller har været uspecifikt har godkendelsen af accept-kriteriet ikke været det samme som godkendelsen af en user storie. Alle Scrum ceremonier kan findes i Bilag[D].

30-04-2026

Person	Hvad lavede du sidst?	Status	Problemer
Devin	4.2, 2.8, 3.2, 2.7	Færdig	Energi, træner ikke ben
Hussein	2.10, 3.3, 4.3, 4.5	Igangværende, Færdig med 4.5	Problemer med test model-klassen fejler, fordi exceptions ikke er lavet.
Kasra	2.10, 3.3, 4.3, 4.5	Igangværende, Færdig med 4.5	
Mussa	4.5, 2.10, 3.3, 4.3	Færdig	
Victor	4.2, 2.8, 3.2, 2.7	Færdig	Observer på memberList under details på et hold Modellering af DBS (klassediagram)

Figur 11: Daily-scrum møde ceremoni

Figur 11 viser et notat fra et tilfældig udvalgt daily-scrum møde. Disse ceremonier blev udført hver dag, som det første. Hver gang er der blevet anvendt en tabel, der viser en rækkefølge på gruppens team-members, dvs. at vi kommer til at høre fra alle personer. Et daily-scrum møde bestod af at Scrum-master skrev en besked på discord med et mødelokale, 20-30 minutter før mødetid. Mødet startede først, når alle gruppemedlemmer var ankommet. Her blev der sagt godmorgen og Scrum-master valgte, hvem der først skulle fortælle om, hvad de havde lavet, deres status og eventuelle problemer. Alle i gruppen skulle igennem disse tre punkter, før mødet var slut. Disse møder resulterede i, at vi altid vidste hvad hinanden lavede og om de havde brug for hjælp.

Sprint 2 (30-04-2026)

Person	Start	Stop	Fortsæt
Devin	Mere selvstændig og sørge for at sine opgaver er ordentligt lavet, før sidste dagen. Spørg ved tvivl.	Bruge git forkert. Fetch før pull.	Med at arbejde 2 og 2
Hussein	Jeg skal være bedre til at tjekke discord for git-opdateringer.	Lade som om vi ved hvad Kasra siger	Gode daily scrum møder
Kasra	Bedre grupperelation og dynamik ved fællesarrangementer som vinsmagning, eller fredagsbar.	Med merge konflikter	Kommunikation med respekt.
Mussa	Snak mere om opgaverne. Mere kommunikation i gruppen.	At lave noget, uden at fortælle det til de andre.	Fortsæt med gode review og retrospective møder.
Victor	Fælles spisepauser, hvor vi ser Olsen Banden på storskærm.	Kalde røde tests færdige. Så mange kort på YouTrack, der gør de samme ting.	At gennemføre sprints.

Figur 12: Retrospektiv møde ceremoni

Figur 12 viser, hvordan vi har håndterete vores retrospektive møder efter hvert sprint. Scrum-master har spurgt hver person, hvad de ønskede at henholdsvis starte, stoppe og fortsætte med ift. projektets arbejde. Her har vi igen sørget for, at alle medlemmer bidrager til samtalen, ved at notere i en tabel, der kræver at alle har et bud på start, stop og fortsæt. Dette brugte vi til, at videreudvikle på vores samarbejde. Det blev brugt som en reference til, hvordan vi kunne blive bedre i gruppen til at arbejde sammen med hinanden, og hvad vi individuelt tænkte om gruppearbejdet. Som det kan ses i Scrum ceremonierne Bilag[D], går mange ting igen, hvilket er et tegn på, at gruppen ikke har fået det ud af disse retrospektive møder, som vi potentielt kunne.

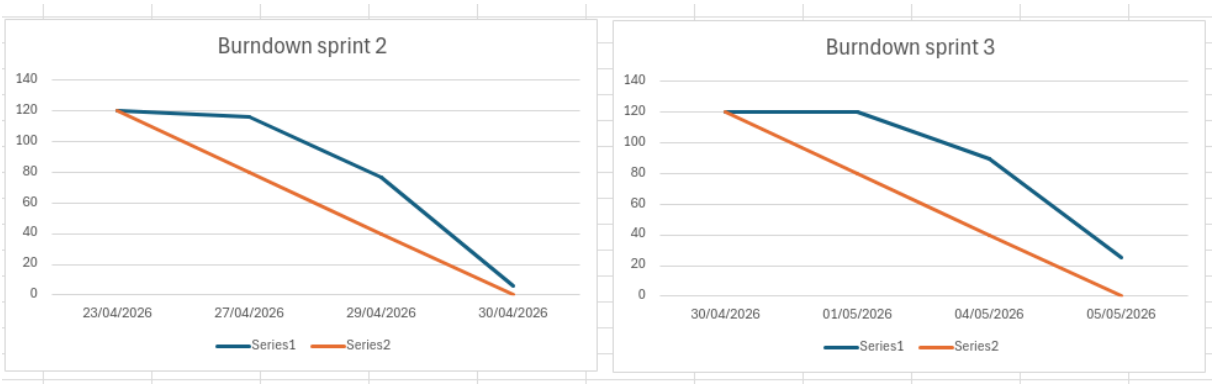
Sprint 1 (23-04-2026)

Alle acceptkriterier fra product backlog er godkendt af product owner:

- Medlem kan se oplysninger på hold
- Medlem kan trykke på tilmeld hold
- Medlem bliver sat på medlemslisten på holdet
- Når medlemmet bliver sat på medlemslisten skal det opdateres både på serveren og i databasen.
- Når det opdateres i serveren skal alle klienter opdateres med de nye informationer.

Figur 13: Sprint review møde med acceptkriterier istedet for user storie

Ovenstående Figur 13 viser at product owner er gået igennem acceptkriterier istedet for, det er gjort både i det første og andet sprint. Som sådan ville det have været en mulighed at gørre dette korrekt ved anvendelsen af acceptkriterier, hvis acceptkriterierne var perfekte og dækkede hele user-storien. Det viste sig dog også under sprint-retrospektive møder at det ikke er nemt at lave perfekte acceptkriterier og man nemt kan komme til at undlade noget vigtigt. Det vil sige at det er mere ligetil at godkende user-stories da man undgår fejl ved et ekstra abstraktions lag. Det har dog gennerelt fungeret godt at anvende product owner til godkendelsen af user-stories og størstedelen af user-stories har været godkendt af product-owner. Der har dog været omprioriteringer af ting som skulle implementeres, da product owner lagde mærke til at der manglede noget som skulle have været implementeret tidligere. User-stories har dog ikke ændret sig i løbet af projektet og har ikke krævet en omprioritering/ændring af product-owner løbende. Selvom størstedelen af user-stories har været godkendt har der været enkelte Sprints som har fejlet dette skyldes opgaver som ikke er blevet godkendt af teamet ved udgangen af et sprint.



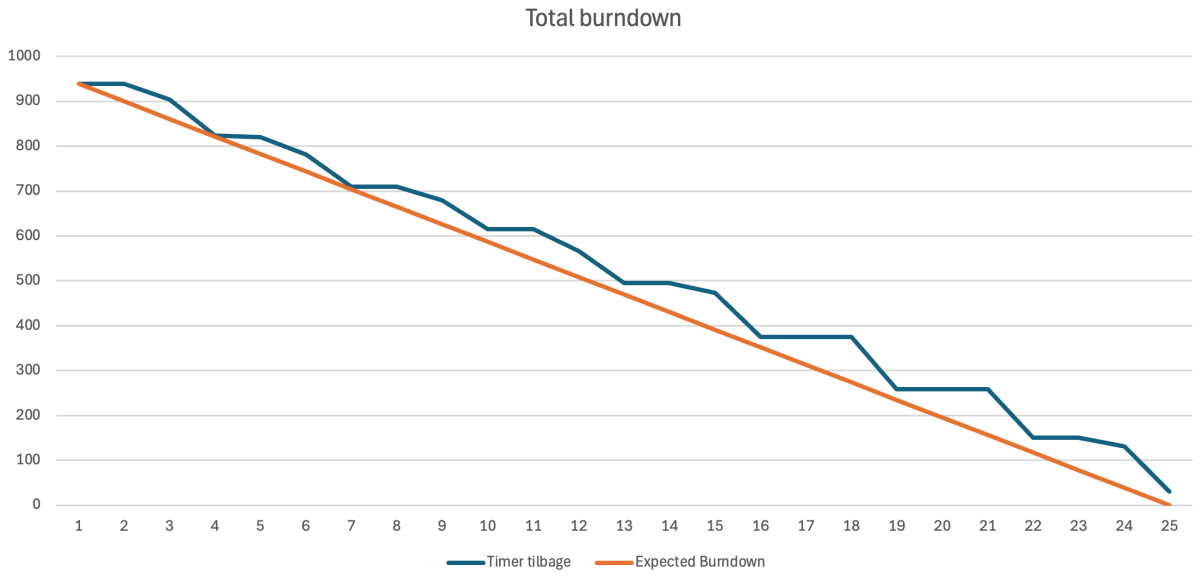
Figur 14: Burndown for fejlede sprints

Figur 14 viser burndown for begge de fejlede sprints. Vi brugte burndown til at få et overblik over, om vi var foran eller bagud på de enkelte sprints. Resten af vores burndown charts, det totale burndown chart og vores sprint fordeling er vedlagt i Bilag[E]. Det ses

på Youtrack på Bilag[F] at test har fejlet begge sprint og javadoc har fejlet på det ene. Her har testen i første omgang fejlet, da den viste fejl i kode-implementationen som ikke blev håndteret og i anden omgang fejlede testen ,da den ikke fulgte ZOMBE standarden. Javadocen fejlede, da oprettelsen af Javadoc havde ændret på kode og der skulle laves et rollback til en tidligere version, før javadocens oprettelse. Da testen ikke var blevet fuldført i Sprint 2 og 3, er der blevet tilføjet en ny opgave i sprint-backlog, sprint 4 inkluderer oprettelsen af nye teste efter forventet ZOMBE standard, som også er blevet godkendt.

Gennerelt har review møderne været et godt værktøj til, at se hvorvidt vi overholder de krav som vi havde sat tidligere i projektet, og har fungeret som en god sikkerhedsfunktion der sørger for at vi ikke glemmer vigtige features, som product-owner ønsker sig. Dog kan manglen af tilføjelsen af user-stories løbende i projektet og manglen af omprioritering være et tegn på at de første stories har været virkelig gode, eller at product-owner manglede den nødvendige seperation fra sin anden rolle som team-member, for at finde disse mangler.

Det kan dog konstateres at projektet er nået i mål, fordi alle user-stories har været opfyldte. På nedenstående figur kan man se, hvordan projektarbejdet er gået ved anvendelsen af et burndown chart, som viser hele projektarbejdet fra sprint 1 til og med sprint 8.



Figur 15: Total Burndown

Ovenstående Figur 15 viser det totale burndown chart for projektet. Her vises forventet burndown (timer) og reelle timer tilbage. Der kan dog ses, at selvom alle user-stories har været fuldført er der nogle sprints, der fejlede som resulterede i oprettelsen af nye opgaver i sprint backloggen der skulle dække de ting som manglede. Dette har resulteret i, at de reelle timer ikke rammer expected til sidst på modellen.

4. Personlige reflektioner

4.1. Victor

Jeg har i projektet haft rollen som Scrum-master, hvor jeg blandt andet har nydt at stå for, at styre møderne og informere holdet om, hvorvidt vi var bagud eller foran. Derudover at give ordet til de forskellige Team-Members. Jeg har været klar over den utålmodighed, der måske følger med en rød domineret personprofil, og jeg har prøvet at gøre de andre gruppemedlemmer klar over, at det er en egenskab jeg har.

Jeg har forsøgt at tage højde for, at alle i gruppen fik indflydelse, når der skulle tages beslutninger. Det har jeg blandt andet gjort ved retrospective-møderne, hvor alle fik lov at sige, hvad de tænkte om gruppearbejdet. Dertil skal det pointeres, at gruppearbejdet har væsentligt gavnet af, at alle blev hørt, da indvendinger ofte blev mødt af stor enighed blandt andre gruppemedlemmer, hvilket førte til hurtig ændring af adfærd i gruppen.

I selve projektet har jeg været med på det hele, og hjulpet alle gruppemedlemmer med deres opgaver, hvis de fandt dem besværlige. Hovedsageligt har jeg dog siddet på implementation og design.

Jeg mener dog ikke jeg er blevet mødt af samme ambitionsniveau, som vores forventningsafstemning lød til at love, fra nogle af de andre gruppemedlemmer. Jeg forventede højt humør, god arbejdsmorale og højt ambitionsniveau, hvor alle tog ansvar for de opgaver de blev stillet, samt gik til opgaven nysgerrig – med idéen om at få det bedst mulige resultat på opgaven. Dog mener jeg ikke, at nogle af de andre gruppemedlemmer har stået ordentligt til ansvar for deres opgave, eller har gået til det med ambitionen om at levere det bedst mulige, hvilket jeg ærger mig over.

Jeg sidder også tilbage med en følelse af frustration over, at gruppekontrakten ikke er blevet overholdt. F.eks. er mødetider blevet overtrådt gentagende gange, men der er ikke blevet handlet på det, hvilket jeg tror skyldes at idéen med gruppekontrakten generelt var, at den overhovedet ikke skulle brydes, så der ikke var grund til at handle. Den havde til formål at sætte en ramme for, hvad gruppemedlemmerne i overensstemmelse fandt acceptabelt. Derudover er gruppekontrakten ikke blevet ændret, selvom der ikke er sket handling og den derfor ikke blev overholdt.

Jeg tror dette skyldes en form for konfliktsky i gruppen, trods vores personprofiler og kutlurprofiler siger noget andet. Det virker som om, at der ikke er nogen, der har lyst til at konfrontere hinanden og optrappe konflikten, for muligvis at skabe et bedre samarbejde.

I fremtiden vil jeg være bedre til, at sørge for at gruppekontrakten bliver overholdt og vedligeholdt, så alle gruppemedlemmer trives. Til en anden gang vil jeg også være bedre til at opsøge en mulig optrappende konflikt, så den kan blive løst, hvilket kan gavne gruppearbejdet.

4.2. Devin

I projektets arbejde har jeg primært arbejdet i 2 roller et normalt team-medlem og product owner. Her har jeg sørget for ændringer af prioritering når det har været nødvendigt og opfylt product owner rollen. Det har dog været en udfordring at tage afstand fra at være et normalt medlem og tage product owner rollen på ved sprint review. Den blå personprofil har understøttet mig med min rolle og detaljeroienteret arbejde dog har der været et par krav som løbende har været overset de er dog blevet fikset i et efterfølgende sprint som et nyt backlog element.

Som normalt team-medlem har jeg dog arbejdet mest med design og implementation af projektet. Jeg er en stærk blanding af blå og rød her har jeg taget initiativ implementation og design af databasen. Siden det var noget vi ikke havde arbejdet med før nu og det virkede som en meget interessant opgave at gå igang med.

Generelt i projekt arbejdet har vi haft gode fremskridt og er kommet frem til et produkt som jeg kan være tilfreds med, dog har der været udfordringer i processen. Som nok både stammer fra forskellene i vores personprofiler i grupppen og forskellige forventninger af projektet. I starten af projektet var vi blevet enige om at vi gerne ville op på et højt ambitionsniveau og gerne vill lavet et større projekt hvor alle ville bidrage til det. Det kunne dog mærkes specielt i sprint retrospektiv møderne der ikke var enighed om hvad det ville betyde at tage ansvar for sit eget arbejde. Mange af opgaverne i gruppen har været præget af at det føles som om man skulle tvinge andre gruppemedllemer til at lave opgaverne til både at opfylde hvad der skulle være et mindstekrav for dem men også at komme igang med opgaverne som det ville forventes. Det skyldes nok en manglende kommunikation i gruppen om problemer som dog aldrig kom, da der samtidig med mange opgaver og forventninger også har været en forsvindende motivation til projektet.

Der har været nogle opgaver som ikke er blevet overholdet som det har været tiltænkt som f.eks gruppekontrakten, her havde vi i starten af semesterprojektet masser af energi og holdte styr på det. Dog som der kom mere arbejde og mere tidspress, for at få færdiggjort opgaverne er nogle opgaver røget til side som f.eks at følge op på gruppekontrakten og opdatere den. Flere af disse problemstillinger kunne formentlig have været løst med en mere åben kommunikation. I stedet for at tage de nødvendige konfrontationer valgte jeg dog ofte at undgå dem for at bevare den gode stemme i gruppen og undgå konflikter.Jeg har lært at jeg fremadrettet skal blive bedre til at tage den direkte og konstruktive diskussion med mine gruppemedlemmer, når problemerne opstår, i stedet for at udskyde det eller blot berøre det overfladisk.

4.3. Kasra

I projektets arbejde har jeg fungeret som almindeligt team-medlem, hvor mine arbejdsopgaver har været fordelt mellem kodning og dokumentation. På den tekniske side har jeg haft meget fokus på implementeringen af Model- og Mediator-lagene, og jeg har brugt en del tid på at skrive Javadoc for at sikre en ordentlig læsbarhed i koden. I forhold til projektrapporten har jeg ikke blot bidraget til afsnittene om Analyse,

Implementering, Test og Konklusion samt udarbejdelsen af vores use-case beskrivelser; jeg har også lært utrolig meget fagligt af at arbejde med netop disse områder.

Vi har haft et forløb præget af en del udfordringer i forhold til forventningsafstemning, gruppens dynamik og vores gruppekontrakt. Den absolut største udfordring i projektet var, at vi i opstarten lagde ud med et meget højt ambitionsniveau og faste rammer, hvilket dog viste sig udfordrende for gruppen at opretholde i praksis. Min personprofil hælder mest til det gule, hvilket betyder, at jeg fokuserer meget på nysgerrighed, idérigdom og den gode stemning i gruppen. I dette projekt har jeg dog i høj grad arbejdet med mine egne svagheder og fokuseret på struktur. Jeg er altid mødt op til tiden til vores aftaler, og jeg kan mærke, at jeg er blevet en meget mere punktlig person i forhold til sidste semester. Hvad jeg især har lært om gruppearbejde er, at det ikke er nok blot at udforme en kontrakt i starten; det kræver løbende opfølgning og ærlig kommunikation fra alle. I min næste gruppe vil jeg helt klart ændre min tilgang ved at være endnu mere proaktiv og turde tage de nødvendige snakke, hvis aftalerne ikke bliver overholdt i teamet.

Hele dette forløb har lært mig utrolig meget på flere fronter. Fagligt har det været spændende og udfordrende at lære om designprincipper. Jeg er særligt stolt af at have arbejdet i dybden med SOLID og GRASP under implementeringen, da det i høj grad har rykket min forståelse for, hvordan man strukturerer kode ordentligt. Sammenlignet med første semester kan jeg desuden tydeligt mærke, at jeg er blevet meget bedre til selve projektskrivningen, ligesom mine evner inden for design af diagrammer og forståelse af systemarkitektur har fået et markant løft. Det har alt sammen givet mig et meget bedre grundlag for at kunne evaluere vores endelige produkt i konklusionen. Hvis jeg skal fremhæve de tre vigtigste ting, jeg har lært i dette projekt, så er det for det første at anvende designprincipper og forstå systemarkitektur i praksis. For det andet har jeg lært vigtigheden af læsbarhed og formidling gennem veldokumenterede use-cases, diagrammer og Javadoc. Den tredje vigtige læring har været af personlig karakter – nemlig indsigten i, at en velfungerende gruppe kræver aktiv vedligeholdelse, og at struktur og punktlighed er afgørende for et godt samarbejde.

4.4. Mussa

I projektet arbejdede jeg primært med dokumentation, protokoller og dele af implementeringen. Derudover deltog jeg også i arbejdet med projektrapport og procesrapport. Jeg bidrog blandt andet til beskrivelser af systemets design og kommunikation mellem klient og server.

Gruppen arbejdede generelt godt sammen gennem hele projektet. Opgaverne blev fordelt ud fra gruppemedlemmernes kompetencer og styrker, hvilket gjorde samarbejdet mere effektivt. Derudover arbejdede vi ofte sammen to og to om opgaverne, hvilket gav mulighed for at samarbejde med forskellige gruppemedlemmer og hjælpe hinanden undervejs. Hvis nogen blev færdige med deres egne opgaver, blev der ofte tilbudt hjælp til resten af gruppen.

Vi arbejdede med Scrum og UP gennem projektet, hvor arbejdet blev opdelt i sprints. Dette gav et godt overblik over opgaverne og gjorde det lettere løbende at videreudvikle projektet. Kommunikation i gruppen fungerede godt, og alle havde mulighed for at komme med deres egne meninger og idéer.

I forhold til personprofiler identificerede jeg mig mest med den grønne profil. I starten af projektet var jeg derfor mere tilbageholdende og stille i gruppearbejdet, især når der skulle diskuteres idéer og løsninger. Efterhånden som jeg lærte gruppen bedre at kende, blev jeg mere tryg ved at deltage aktivt i diskussioner og komme med mine egne forslag og meninger.

Gennem projektet har jeg udviklet mig både fagligt og personligt. Jeg er blevet bedre til at arbejde struktureret med dokumentation og design samt til at planlægge og færdiggøre mine opgaver til tiden. Derudover har projektet givet mig mere erfaring med samarbejde, MVVM-arkitektur og udvikling af større systemer i grupper.

4.5. Hussein

I projektet arbejdede jeg primært med protokoller, Junit-tests og dele af implementeringen samt database, og bidrog desuden til projektrapport, hvor jeg beskrive test delen.

I forhold til personprofiler identificerede jeg mest rød og gul. Det betød at jeg fra staren gik engageret og handlingsorienteret til opgaven, jeg tog initiativ til at sætte gang i arbejdet med protokoller og tests. Den gule side kom til udtrk i min entusiasme for at dele viden med gruppen og i min nysgerrighed over for løsninger og tilgange.

Gennem projektet har jeg udvilklet mig både fagligt og personligt. Jeg er blevet bedre til brug mit energi ind i struktureret arbejde og til at planlægge min opgaver. derudover har projktet givet mig mere erfaring med samarbejde i grupper, MVVM-arkitektur og udvikling af større systemer.

Generalt i gruppen har vi haft godt samarbejde igennem forløbet. vi havde fokus på at dele opgaverne ud fra kompetencer og styrker og havde fokus på at opgaverne blev lavet i par af to og to, hvilke gave mulighed for få hjælp og diskuter valg. Var man nu færdig med sine opgaver, var der muligheed for at tilbyde hjælp til andre opgaver, men der var dog stadig udfordringer i gruppen som mulighvis stemmer fra forskellige personprofiler, arbejdsmåder og mangel på kommunikation. Der blev brugt Scrum og UP med sprints, hvilke gav et overblik og gjorde det lettere at prioritere projektet.

5. Reflektioner over vejledning

Gennem projektforløbet har vi haft vejledning med vores vejleder Steffen i forbindelse med udvalgte sprint reviews og gennemgang af projektets fremgang. Vejledningen blev primært brugt til at få feedback på vores løsning, arbejdsproces og dokumentation samt til at diskutere eventuelle problemer og usikkerheder i projektet.

Vores tilgang til vejledning var, at vi forsøgte at møde forberedte op til vejledningsmøderne med konkrete spørgsmål og materiale, som kunne gennemgås. Dette gjorde det muligt at få feedback på både implementering, design og strukturering af rapporten. Derudover blev vejledningen brugt som en form for kvalitetssikring af projektets retning og de valg, vi traf undervejs i udviklingsprocessen.

Feedbacken fra vejledningen var ofte overordnet og fungerede derfor mere som vejledning til områder, vi selv skulle undersøge og arbejde videre med. Dette betød, at gruppen selv skulle reflektere over løsningerne og identificere, hvordan eventuelle problemer kunne forbedres i systemet eller rapporten. På den måde blev vejledningen ikke kun brugt til at få svar, men også til at skabe refleksion over vores egne valg og arbejdsmetoder.

Vejledningen blev desuden brugt til at få et bedre overblik over projektets progression mellem sprints. Her fik vi mulighed for at præsentere de løsninger og funktioner, der var blevet udviklet siden sidste vejledning, samt få vurderet om projektet bevægede sig i den ønskede retning. Dette gav samtidig gruppen mulighed for løbende at justere på både arbejdsprocessen og projektets fokusområder.

Generelt oplevede vi, at vejledningen var med til at give overblik over projektets fremgang og sikre, at vi bevægede os i den rigtige retning gennem projektforløbet. Samtidig gav vejledningsmøderne anledning til refleksion over både vores tekniske valg og måden, vi samarbejdede på i gruppen.

6. Konklusion

Gennem projektforløbet har vi arbejdet med udviklingen af et holdtræningssystem ved anvendelse af Scrum og Unified Process (UP). Arbejdet blev opdelt i flere sprints, hvor analyse, design, implementering og test løbende blev videreudviklet gennem iterative processer. Dette gav os mulighed for at udvikle systemet og tilpasse arbejdsprocessen undervejs under projektforløbet.

Brugen af Scrum hjalp generelt med struktur og overblik i projektet gennem sprint planning, daily scrum, sprint reviews og retrospectives. Særligt brugen af sprint backlogs, acceptkriterier og burndown charts gjorde det muligt for os at følge projektets progression og identificere problemer undervejs. Samtidig viste projektet også, at Scrum kræver tydelig kommunikation og løbende opfølgning for at fungere optimalt i praksis. Dette kom blandt andet til udtryk gennem udfordringer med gruppekontrakten, forskellige forventninger til ambitionsniveau og manglende opfølgning på tilbagevendende problemer fra retrospectives.

Projektforløbet har også lært os, hvordan forskellige personprofiler og kulturforståelser har betydning for samarbejdet i gruppen. Gruppen bestod af medlemmer med forskellige styrker, arbejdsformer og tilgange til kommunikation og planlægning. Dette skabte både udfordringer og fordele i samarbejdet. De forskellige profiler bidrog blandt andet med initiativ, refleksion og idéudvikling, hvilket havde en positiv betydning for projektets udvikling. Samtidig viste projektet, at forskelle i forventninger og opfattelse af blandt andet tid og ansvar kunne skabe misforståelser i gruppearbejdet.

Gennem projektet har vi opnået erfaring med udvikling af et større system i en gruppe samt anvendelse af udviklingsmetoder og designprincipper i praksis. Derudover har projektforløbet bidraget med erfaring inden for samarbejde, kommunikation, planlægning og håndtering af udfordringer i et længerevarende projektarbejde.

Samlet set tænker vi at projektforløbet var meget lærerigt både fagligt og samarbejdsmæssigt. Selvom der opstod udfordringer undervejs, lykkedes det os som en gruppe at færdiggøre projektet og opnå de opstillede mål gennem løbende samarbejde og videreudvikling af både systemet og arbejdsprocessen.

8. Bilag

Bilag A – Gruppekontrakt

Bilag B – Kulturelle profiler

Bilag C – Product backlog

Bilag D – Scrum ceremonier

Bilag E – Sprintskema

Bilag F - Youtrack Sprints