

# Zealand - Sjælland

## Erhvervsakademi

Datamatiker 1. Års Projekt 2020



**Case:** Frederiksberg Sportsklinik  
**Vejledere:** András Pedersen, Anders Petersen, Karsten Vandrup  
**Forfattere:** Kelvin Aggerholm, Benjamin Kyhn, Tommy Hansen  
**Github**  
**Repository:** <https://github.com/BenjaminKyhn/Foersteaarsprojekt/>  
**Official** <https://github.com/BenjaminKyhn/Foersteaarsprojekt/>  
**Release:** [releases/tag/v1.0](https://github.com/BenjaminKyhn/Foersteaarsprojekt/releases/tag/v1.0)  
**JavaDoc:** <https://benjaminkyhn.github.io/Foersteaarsprojekt/>  
**Antal Tegn:** 92689

# Indholdsfortegnelse

Indledning (Kelvin, Tommy og Benjamin)	4
Produktbeskrivelse	4
Virksomhed	5
Mission (Tommy)	5
Vision (Tommy)	6
Forretningsmodel (Benjamin)	6
Customer segment	7
Value proposition	8
Revenue streams	8
Channels	8
Customer relationship	8
Key activities	8
Key resources og key partners	9
Cost structure	9
Opstart og finansiering	9
Strategi for skalering af forretningen (Kelvin)	10
Systemudvikling	10
Projektstyring	10
Metodevalg	10
Javadokumentation (Benjamin)	10
Brugertest (Kelvin)	10
Metrikker	11
Eksperimentering (Tommy)	11
Proof of concept	11
Mockups (Kelvin)	15
Unified Process (Tommy)	17
Faseplaner	19
Refleksioner til Unified Process	21
UML (Kelvin)	21
Sporbarhed (Tommy)	21

Use Case Diagrams (Kelvin)	27
Use Cases	28
Domænemodeller	30
Systemsekevensdiagrammer	31
Operationskontrakter (Benjamin)	32
Sekvensdiagrammer	34
Klassediagrammer (Kelvin)	38
Test (Benjamin)	39
Code coverage	39
Unittest	40
Systemtest	43
Analyse af Brugertest (Kelvin)	46
Programmering	49
Designmønstre (Tommy)	49
Clean architecture med observer pattern	49
Tanker til Clean Architecture:	51
Singleton pattern	51
Facade pattern	53
Persistens (Benjamin)	53
MySQL vs. SQL	53
Firebase	54
Udviklingsmiljøer (Kelvin)	55
Github (Kelvin)	57
Gradle (Kelvin)	59
Androidudvikling (Tommy)	61
Trådprogrammering (Tommy)	65
Polymorfi (Benjamin)	66
Datastrukturer (Tommy)	68
Sikkerhed (Benjamin)	69
Encapsulation	69
Hashing	71
Fejlhåndtering(exceptions) (Kelvin)	73
Konklusion (Kelvin, Tommy, Benjamin)	75

## Indledning (Kelvin, Tommy og Benjamin)

Denne rapport omhandler udviklingen af et eksamensprojekt for første år af datamatikerstudiet. Projektet tager udgangspunkt i en problemstilling fra erhvervslivet, som er stillet af vores kunde fra Frederiksberg Sportsklinik. Kunden ønskede at få udviklet et IT-system til kommunikation med sine patienter. I rapporten vil vi præsentere en løsning på problemstillingen, samt redegøre for vores proces i udviklingen af denne. Løsningen er baseret på kravindsamling fra kunden, som vi har fremskaffet gennem kundemøder. Vi vil i rapporten også inddrage vores fag Programmering, Systemudvikling og Virksomhed fra datamatikerstudiet og forklare, hvordan vi opfylder fagenes læringsmål. Rapporten er opdelt i tre dele efter vores fag og vi vil i hver af de tre afsnit begrunde de valg, vi har truffet undervejs i vores udviklingsproces. I virksomhedsafsnittet fremlægger vi først en målrettet forretningsmodel, og forklarer ud fra den, hvordan virksomhedens vision og mission kan realiseres både i opstartsfasen, men også med henblik på senere skalering. I systemudviklingsafsnittet vil vi beskrive vores arbejdsproces samt redegøre for hvilke værktøjer, vi har benyttet i modellering af systemet. Vi vil se på udvalgte artefakter, som vi har udarbejdet til at afbilde systemets funktionelle krav, med henblik på at identificere sporbarheden i disse. I programmeringsafsnittet vil vi begrunde vores valg af softwarearkitektur og datastrukturer samt diskutere de udfordringer, vi har haft med at implementere dem i programmerne. Vi vil også diskutere de udfordringer, der er, når man udvikler software i en arbejdsgruppe og til sidst tager vi de kritiske briller på, når vi gennemgår sikkerheden af vores program og kommer med forslag til, hvordan den kan forbedres.

## Produktbeskrivelse

Vores kunde, Frederiksberg Sportsklinik, ønskede et IT-system, som han kan bruge til den daglige interaktion med sine patienter. Kunden havde følgende funktionelle krav:

- Løsningen skal bestå af en androidapp og en desktopapp
- Patient og behandler skal kunne sende beskeder til hinanden med de to apps
- Patienten skal kunne booke en tid hos behandleren på app'en
- Behandleren skal kunne tildele patienten træningsprogrammer på app'en
- Patienten skal kunne se træningsprogrammet på app'en som video

De helt specifikke funktionelle krav samt ikke-funktionelle krav, kan ses i bilag 7.

Vi har udviklet en løsning, som lever op til de to første funktionelle krav, med henblik på senere implementering af de resterende krav. Vi har udarbejdet to videoer til

demonstration af løsningen; en for androidapp'en og en for desktopapp'en.

Videoerne findes på følgende links:

- Præsentation af desktopapp: <https://www.youtube.com/watch?v=6ooYH2aHg2M>
- Præsentation af androidapp: <https://www.youtube.com/watch?v=QyYi-xggBXA>

Vi håber, at læserne af denne rapport vil se videoerne inden læsningen, da det giver en meget bedre forståelse for vores løsning.

## Virksomhed

### Mission (Tommy)

At have en mission er vigtigt for ethvert firma, da det giver firmaets ansatte en klar reference til hvad de skal gøre i nuet. Derfor skal vores softwarefirma ligeledes have en mission også selvom dette er et eksamensprojekt og ikke en reel forretning. Fordi selvom vi ikke har et reelt firma kan tankerne bag processen være lærerig og er en del af kravene til eksamen.

Vores tanker bag missionen ligger i at vi er fire studenter med gennemsnitlig studenterindkomst og derfor skal vores mission til det firma vi 'skaber' være passende til sådan et scenarie. Vores nuværende fokus er på danske virksomheder da det ville være mest realistisk at begynde med som dansk startup. Et andet fokus er, da vi i eksamenssammenhængen arbejder med et firma i sundhedssektoren, så indskrænker vi også vores fokus til at vores kunder er dem i dette erhverv, af den årsag at her har vi nu mest erfaring. Med disse tanker kom vi frem til denne mission for vores firma:

- At skabe personlige og erhvervspraktiske software til vores kunder i sundhedssektoren
- At engagere og give support både under udviklingen af softwaren samt efter, hvori møderne er fastlagte af begge parter på aftalte tidspunkter
- Såfremt det er passende, at være åbne for kundeinput og aktivt udforme softwaren så kunden får den bedste tilfredsstillelse med deres brug af denne

Fremadrettet ville denne mission kun bruges til dette eksamensprojekt og hvis vi oprigtigt lavet et firma sammen ville vi kun tage inspiration af denne.

## Vision (Tommy)

Udover mission som fokuserer på nuet, så har firmaer også en vision der ser fremad. Vores vision, ligesom missionen, vil være baseret på det gældende scenarie hvor vi er fire studerende med gennemsnitlig studenterindkomst der arbejder sammen med en kunde fra sundhedssektoren. Til dette scenario vurderer vi at det er passende at udvide vores horisont over tid. At starte i Danmark men blive internationale når en god mulighed opstår at udvide fra kun at servicere erhvervspraktikanter i sundhedssektoren til at lave software til alle erhvervsbranche der kunne have gavn af vores software. Fremadrettet vil vi også arbejde på at lave skabe vores egen software der ikke er udarbejdet med en kunde, og denne kunne være ved siden af vores nuværende indtægtskilde. Ved at softwaren ikke er bundet til en specifik kunde vil dette produkt være tilgængeligt til flere og det vil være mere fleksibelt i hvordan det kan profiteres fra, og ud fra dette kan man nemmere bruge en skalerbar model og dermed i helhed også gøre firmaet skalerbart.

Med alle disse overvejelser vil vores vision for firmaets fremtid være:

- Lokaliser og gør brug af kanaler der gør det muligt at finde kunder uden for Danmark
- Udvide vores service til at have en bredere målgruppe men stadig inden for erhverv
- Skabe et produkt der kan udgives gennem egne kanaler eller en forretningspartner samt at dette produkt udvikles uafhængigt af andre således at en skalerbar model kan indføres

Med disse overvejelser om firmaets vision har vi i eksamensprojektet gjort tanker om hvordan et reelt firma kunne udformes. Om det bliver det eller ej er ikke vigtigt men bare ved at overveje har vi i processen udvidet vores viden om de scenarier der er gældende når man opretter sit eget firma.

## Forretningsmodel (Benjamin)

I dette afsnit vil jeg vise, hvordan vi opfylder læringsmålene:

- Den studerende kan vurdere praksisnære forretningsprocesser baseret på centrale analysemetoder
- Den studerende kan håndtere sammenhængen mellem design af forretningsprocesser og design af IT-systemer

Vi har lavet et Business Model Canvas (BMC) til at beskrive vores forretningsmodel, som jeg vil referere til løbende i dette afsnit. Modellen kan ses nedenfor og i bilag 6:

KEY PARTNERS	KEY ACTIVITIES	VALUE PROPOSITIONS	CUSTOMER RELATIONSHIPS	CUSTOMER SEGMENTS
Google (Android)	Udvikle Desktop & Androidapplikationer.	Terapeuter / personale underlagt sundhedssektoren får bedre mulighed for at have en god kontakt/relation til deres klienter.	Abonnement – aftale om vedligeholdelse af systemet.	Værdi for private eller offentlige sundhedsydere, som ønsker online patientkontakt.
Jetbeans(IntelliJ)	Kundemøder	Fysioterapeuten vil kunne have kontakt til klienterne, uden for klinikken.	Anbefalinger gennem diverse reviews som fx trustpilot.	Virksomheder, som praktiserer inden for behandling og genoptræning.
Google(Firebase)	Reklame (Telefon/Fysisk fremmøde)	Applikationen giver fysioterapeuten mere tid og større berøringsflade.		Virksomheder, som fører en klinik og har kundekontakt.
Adobe(Illustrator, XD)				
Revisorer				
Investorer(Seed funding)				
	<b>KEY RESOURCES</b>		<b>CHANNELS</b>	
	Sundhedspersonale / Behandlingsydere	Terapeutens patienter kan nemmere få kontakt (feedback), gennem hele deres forløb.	Google Play Store	
	IDE platform (Android Studio / IntelliJ)	Minimumsydelse – Kontakt mellem serviceyder og kunde.	Virksomhedens hjemmeside	
			Sociale Medier	
<b>COST STRUCTURE</b>		<b>REVENUE STREAMS</b>		
Lancering af app Google Play Store 172,75kr.		Salg af nye apps		
Reklame		Abonnementer		
Softwarelicenser		Vedligeholdelse af apps		

## Customer segment

Kundesegmentet er først og fremmest være virksomheder, der tilbyder en sundhedsydelse af en eller anden art, men da der findes andre former for klinikker med lignende kundekontakt, så forventer vi at kunne udbrede kundesegmenter til at omfatte en hvilken som helst form for klinik med kundekontakt.

## Value proposition

Vi leverer en multifacetteret løsning, som kan løse mange af klinikkernes hverdagsproblemer. Den primære værdi for kunden vil være øget patientkontakt, men vores løsning vil også give klinikkerne mulighed for at kunne indsamle feedback på deres ydelser og udvide deres målgruppe.

## Revenue streams

Vi er en virksomhed, der udvikler software, så den primære indkomstkilde vil selvfølgelig være salg af nye softwareløsninger, men i softwareverdenen er der også konstant behov for opdateringer. En app kan fx ikke få lov til at ligge på Google Play Store i længere tid end 1 år, uden at modtage opdateringer. Derfor vil vores sekundære indtægtskilde være vedligeholdelse af de softwareløsninger, som vi allerede har solgt, da vi ikke forventer, at kunden selv vil være hverken i stand til eller motiveret for selv at vedligeholde dem.

## Channels

Da vi er en nystartet virksomhed vil vores primære salgskanaler i først omgang være sociale medier, Google Play Store, virksomhedens egen hjemmeside og diverse review sites som Trustpilot. Sociale medier som fx Facebook er effektive måder at få udbredt kendskabet til vores virksomhed både gennem word of mouth og betalte reklamer. Derudover vil vi selvfølgelig have vores egen hjemmeside, hvor det vil være muligt at kontakte os, og så vil vores app's være tilgængelige på Google Play Store, så potentielle kunder altid kan se vores tidligere løsninger og selv vurdere, om en lignende løsning vil passe til dem. Trustpilot er en meget velkendt og udbredt forbrugerhjemmeside, så vi vil selvfølgelig sørge for at opretholde vores profil på deres hjemmeside og håndtere anbefalinger og kritik professionelt. En positiv score på Trustpilot er vigtigt for en IT-virksomhed med virke i Danmark.

## Customer relationship

Vi vil få et tæt forhold til vores kunder, da hele vores arbejdsgang involverer at inddrage dem i udviklingen af en meget specifik løsning, der passer til netop den kunde. Vi forventer derfor at kunne opretholde forholdet til kunden også efter salget, da der fortsat vil være et behov fra deres side om at vedligeholde app'en.

## Key activities

Virksomhedens vigtigste aktiviteter vil være dem, der medvirker til at skabe salg. Derfor vil vi fokusere vores arbejde omkring programmering, kundemøder og reklamering for vores ydelser.



## Key resources og key partners

Vi er en IT-virksomhed, så vi er afhængige af mange andre IT-virksomheder, der leverer den software, som vi gør brug af i vores arbejde. Der er mange på markedet, så det er altid muligt at skifte partnere, men vi har nævnt nogle eksempler i modellen. Derudover er kunden i dette tilfælde også en vigtig ressource, fordi vi ikke kan udvikle løsninger, uden inddragelse af kunden, og vi vil være i stand til at skabe større salg, hvis vi kan skræddersyge løsningen til netop det, kunden ønsker. Finansiering er også yderst vigtig, da det er grundlaget for at vores virksomhed kan fungere - i hvert fald indtil vi får skabt et stort nok salg til at kunne klare os selv.

## Cost structure

Der er heldigvis ikke så mange udgifter forbundet med udvikling af software, så vi forventer at kunne holde omkostningerne forholdsvis lave. De eneste umiddelbare omkostninger ved udvikling af en enkelt app vil være softwarelicenser og udgivelse af app'en på Google Play Store. Der vil komme flere udgifter i takt med at virksomheden opskaleres, så derfor vil vi berøre finansiering i næste afsnit.

## Opstart og finansiering

Vi forventer at kunne sælge systemet til kunden for 10.000 kr. Vi har vurderet, at da produktet er et eksamensprojekt og da kunden er en forholdsvis lille spiller på markedet, som tilmed får mange tilbud på lignende systemer fra de andre eksamensgrupper, så kan vi nok ikke tillade os at tage mere for produktet. Det betyder, at da vi kun tjener 10.000 kr. på systemet, så vælger vi at starte firmaet uden et registreret CVR-nummer, da det i Danmark er lovligt at have en momspligtig omsætning på under 50.000 kr. Det betyder selvfølgelig også, at vi skal betale skat af beløbet, men da der kun er tale om en indtægt 10.000 kr., der skal deles mellem 4 personer, så vurderer vi, at det er den bedste løsning.

Det fremgår af investeringsbudgettet, at udgifterne er forholdsvis lave. Den eneste umiddelbare udgift er at lægge app'en ud på Google Play Store. Firebase er i udviklingsfasen gratis, men har nogle begrænsninger som fx maks. 1 GB lagringsplads og maks. 50.000 læsninger fra databasen om dagen. Når app'en bliver taget i brug, vil denne udgift påfalde kunden, og er derfor ikke en udgift, som vi skal tage højde for i vores budget. Vi vil derfor fordele de resterende 9.827,25 kr. til løn for vores arbejde. Vi har valgt ikke at blande personlige udgifter og indtægter ind i virksomhedens budget.

## Strategi for skalering af forretningen (Kelvin)

Efter første succesfulde projekt for Frederiksberg Sportsklinik forventer vi at kunne udvikle og sælge lignende løsninger til andre klinikker for 25000 kr, da vi vurderer efter et succesfuldt salg har vi mere legitimitet på markedet og derfor vil vi stå i en stærkere forhandlingsposition. På dette tidspunkt vil vi stifte et Anpartsselskab (ApS), og derfor kræver det 50.000 kr i opstart. Derudover vil vi anskaffe os nogen arbejdscomputere, hæve sænkeborde, en arbejdsbil, IntelliJ Ultimate brugerlicenser og kontorstole til 319654,40kr (bilag 2). Vi har brug for disse ting til en officiel arbejdsplads. Bilen bruger vi eksklusivt til kundemøder, da det skaber mere tillid og interesse for kunderne. Vi vil finansiere dette via Seed Funding/Venture Capital, da vi er en startende virksomhed med gode vækstmuligheder og da vi har et produkt som de ville være interesseret i at finansiere. Da vores budget ligger på 319654,40.kr ville vi søge seed fund kapital hos Northzone eller Northcap, som begge er Venture Capital/Seed funding virksomheder der specialiserer sig IT virksomhed startup. En Venture Capital vi endte med at fravælge var Sunstone Life Science Ventures, da de fokuserer på IT virksomheder der udvikler lægeudstyr.

## Systemudvikling

### Projektstyring

#### Metodevalg

##### Javadokumentation (Benjamin)

Vi bruger Javadokumentation til at kommentere vores kode. Javadokumentation indledes med syntaksen `/**` og afsluttes med `*/`. Man kan se, hvem der har skrevet koden, fordi den er annoteret med `@author`. Alle klasser har en `@author`, som betyder, at denne person er forfatter til klassen. Andre bidrag til koden i en klasse, vil være annoteret med `@author` umiddelbart før den del, personen har skrevet. Alle kodelinjer, der kommer efter en `@author` annotation skal fortolkes, som om personen har skrevet dem. Der er indsat kommentarer for klasser og metoder i logikken, og der er udarbejdet et Javadoc for koden, som findes i vores repository for projektet. Linket kan findes på forsiden af denne rapport.

##### Brugertest (Kelvin)

Vi bruger Brugertest til at specificere kravene til vores program. Vores brugertest gør det muligt for os at få både Kundens og brugeres input til at designe programmet. Brugertesten er sat op med generelle spørgsmål som giver brugeren et mål som de

skal opnå i Android app'en eller desktop app'en for at løse opgaven. Brugertesten skal have så lidt til ingen input fra testeren, for at give bedre resultater af de testede.

### Metrikker

Vi har brugt metrikken fuldendte og ikke fuldendte opgaver til at evaluere processen undervejs i vores program. Vi har metrikken fuldendte opgaver og ikke fuldendte opgaver i vores dagsorden. Denne metrik brugte vi dag til dag ved at gennemse hvor mange af opgaverne vi lavede dagligt og om vi opnåede den arbejds mængde vi havde sat os for at lave den dag. Det holdte vi så styr på ved at sætte hak i de opgaver vi havde lavet og kryds i dem vi ikke nået at lave den dag, som vi så skulle tage med til næste dag. Når vi udgiver vores program vil vi i fremtiden bruge code coverage metrikken til at kvalitetssikre vores program.

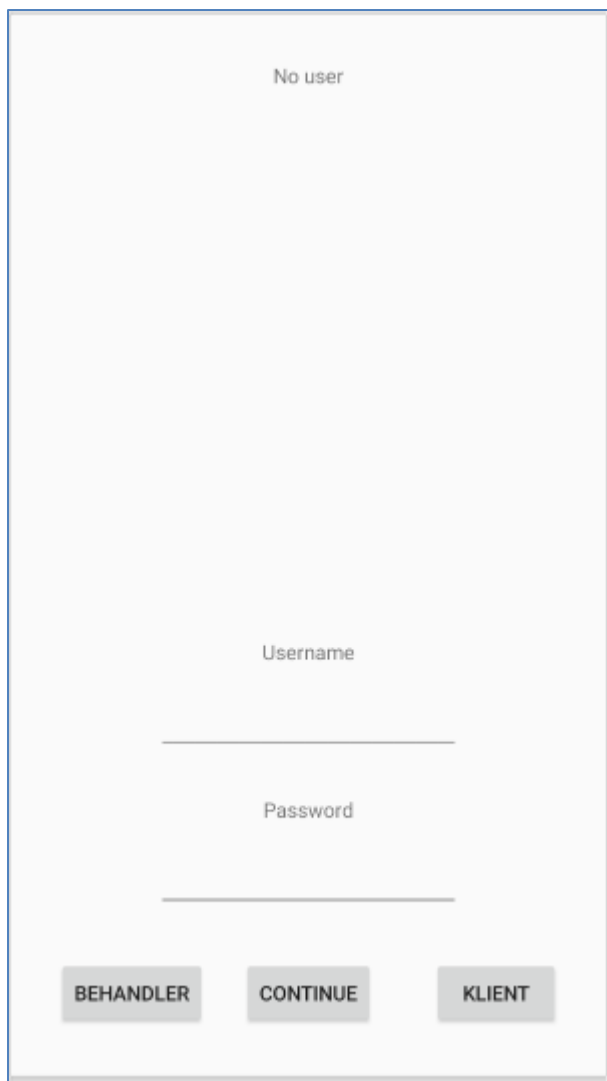
### Eksperimentering (Tommy)

I projektet fandt vi det nødvendigt at eksperimentere inden vi reelt gik i gang med at designe programmet. Vi ville bl.a. inddrage et ikke-afprøvet bibliotek samt lave mockups så vi kunne få feedback til UI'et inden vi fortsatte med at arbejde med det og i denne proces har vi opfyldt dette læringsmål:

- forståelse for eksperimenters betydning som del af eller supplement til systemudviklingsmetoden

### Proof of concept

Til at starte med så var en af problemstillinger at vi ikke vidste om Googles firebase API var et passende fit til vores program. Vi skulle bruge en database der både kunne arbejde med en mobile app samt en desktop app og samtidig skulle vi lære at bruge Cloud Firestore, Firebases mere moderne database bibliotek, da vi kun havde kendskab til det tidligere bibliotek, Realtime Database. Vi skulle også lære samt afprøve Firebase Authentication for at se om dette kunne fungere som vores brugersystem. I denne proces lavede vi en prototype app, eller proof of concept, til mobile hvor fokus var at man skulle kunne logge ind og sende beskeder imellem to mobiler i realtime.



A mobile app login screen prototype. At the top, it says "No user". Below that, there are two input fields: "Username" and "Password". At the bottom, there are three buttons: "BEHANDLER", "CONTINUE", and "KLIENT".

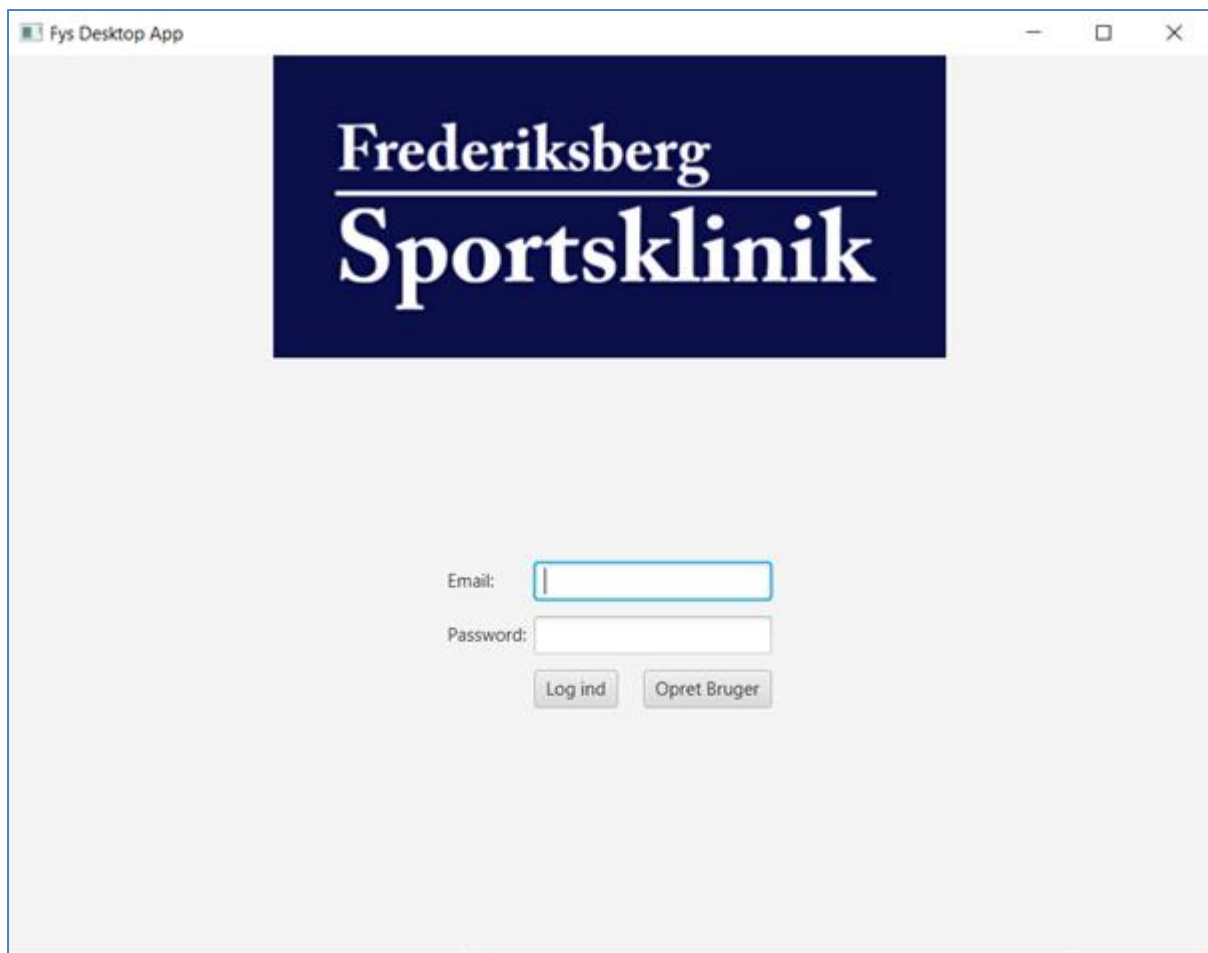
Startsiden til prototypeappen på mobilen.

I en prototype af denne slags er UI design ikke vigtigt og UI'et har derfor kun det essentielle. Der blive desuden også brugt hjælpe knapper til hurtigt at facilitere login. Efter man er loggede ind så skiftes til beskedbrugerfladen.



Dette er et billede af UI'en som den er vist i Android Studio. I den rigtige har beskeder et reelt layout og ikke item 0, item 1, osv.

Med prototypen formåede vi at sende beskeder i realtime og dermed blev argumentet for at bruge Firebase stærkere. Alligevel var der stadig spørgsmål der skulle afklares. Vi vidste stadig ikke om Firebase kunne bruges til en desktop app samt om at brugen af Firebase i denne ville facilitere kommunikation mellem en desktop app og en mobile app. Mere eksperimentering var derfor nødvendigt inden vi kunne fastsætte os på Firebase. Vi måtte derfor lave en lignende prototype for desktopprogrammet, hvor vi her også anvender Firebase biblioteket.



Startside for desktopprototypen. Det skal nævnes at denne prototype blev viderearbejdet til det endelige program og derfor eksisterer desktopprototypen ikke på samme måde som Androidprototypen.

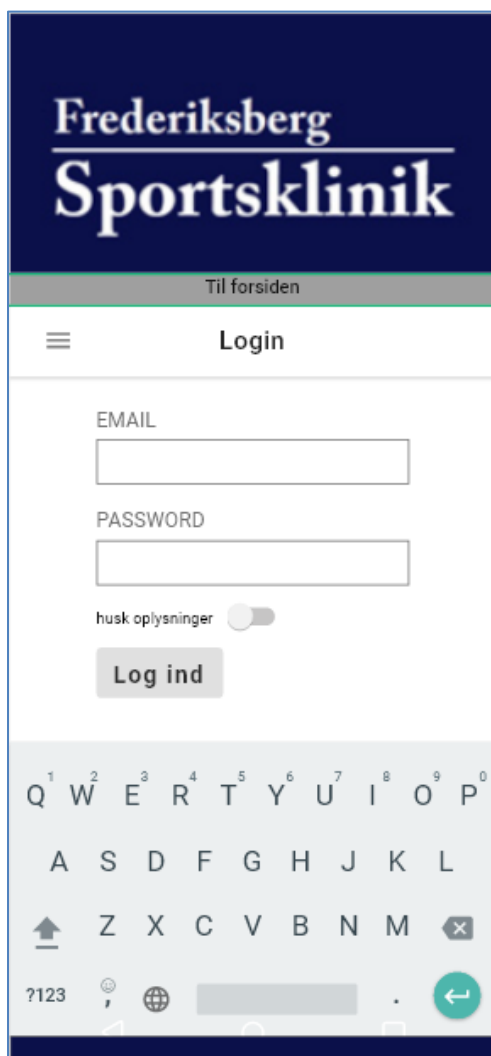
Allerede fra start i vores afprøvning af Firebase sammen med et desktopprogram fandt vi ud af at samme API ikke kunne anvendes. Vi opdagede at Firebase primært er målrettet mod Iphone, Android og web, hvori denne sammenhæng at desktop ikke indgår. Vi kom uden om problemet ved at bruge Firebase Admin SDK som strengt taget ikke er beregnet til dette men alligevel kunne bruges til vores formål. Brugen af denne Admin SDK bragte dog stadig problemer med sig. Firebase Authentication, som var biblioteket vi brugte for at håndtere et brugersystem, er ikke tilgængelig med Admin SDK og dermed kunne vi ikke bruge det hvis vi både skulle have en mobile app og en desktop app. Vi valgte derfor at lave vores eget login system, da det stadig var muligt for begge programmer at bruge Cloud Firestore hvilket var det vigtigste. Dermed endte vi til sidst med to prototyper på to forskellige platforme der begge kunne logge ind og sende beskeder, samt fik vi bekræftet at Firebase kunne bruges for at afvikle kommunikation mellem disse to. Så med disse eksperimenter udfald konkluderede vi at Firebase var passende at inddrage i projektet.

Som kan ses her, så har eksperimenter været en vigtig del af processen og uden disse ville vi have stødt ind i problemerne der opstod længere inde i processen,

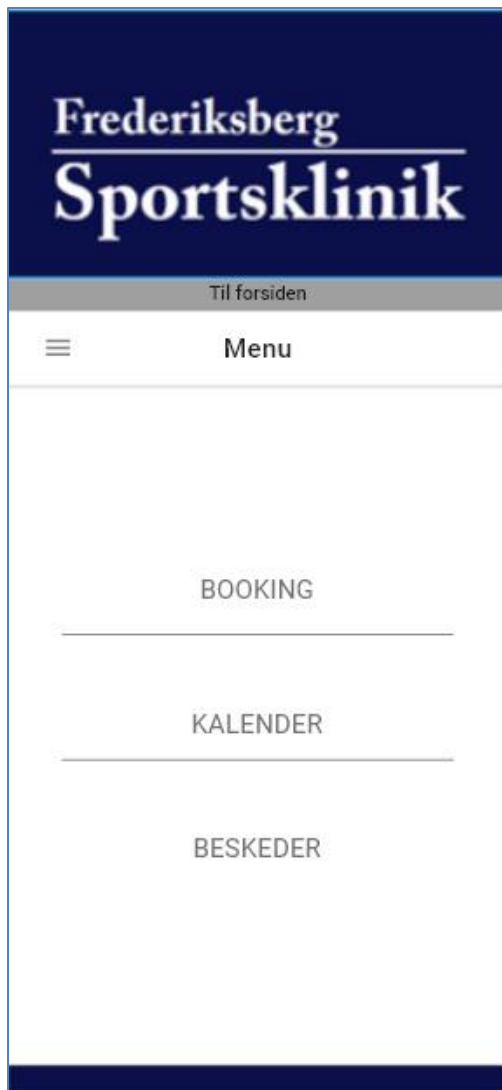
hvilket kunne have haft store konsekvenser for den gældende plan. Vi kunne i værste fald have været nødt til at bremse al produktion for først at afklare problemerne og dette kunne have opstået på et kritisk tidspunkt. Vi kan derpå se at eksperimenter kan mindske risici og give overblik og at i det hele taget har været gavnende for projektet at gøre brug af. Også i senere projekter vil det sandsynligvis være fyldestgørende at eksperimentere inden man fastsætter sig på f.eks. et bibliotek eller en arkitektur da disse projekter også kunne have gavn af eksperimenternes belønning vedrørende mindskningen af risici samt skabe et overblik over hvad er muligt.

### Mockups (Kelvin)

Vi startede med at lave en mockup til programmet som vi i gang med at designe, til dette formål valgte vi at bruge AdobeXD. Det valgte vi at gøre da det gjorde det muligt for at visualisere på en mere detaljeret måde hvordan vores app skulle se ud end, vi tog også dette valg fordi manuel tegninger kræver mere tidsinvestering og har større chance for feature creep i designet af programmet.

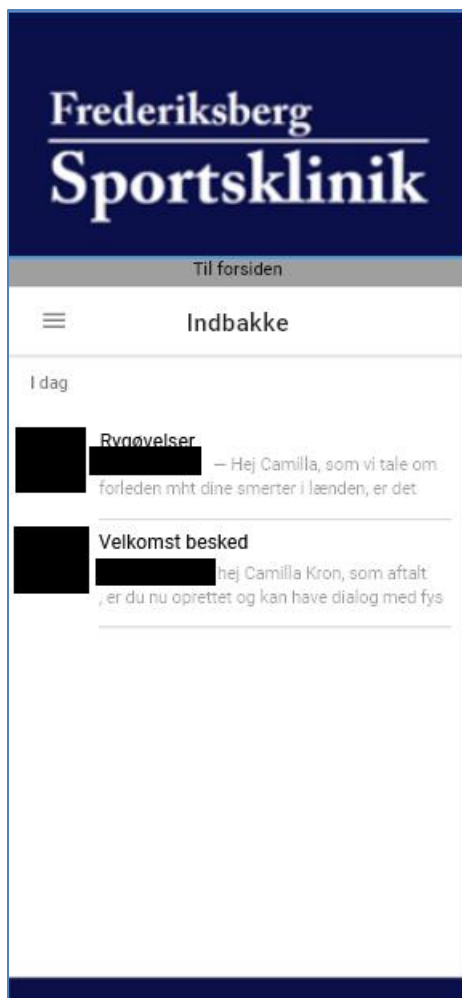


Dette er login siden for AdobeXD prototypen til Android app'en. I en Mockup er der en generel ide for hvordan at den endelige model skal se ud selvom designet ikke er helt færdigt på det tidspunkt. I dette tilfælde er det den færdiggjort version af et mockup for login siden, som også ligner designet som vi bruger i android appen.



Hovedmenuen for AdobeXD mockup af Android app. Designet til mockup har forblevet det samme igennem det største delen af projektet, hvilket er begrundet i brugertestende og kundemøderne. Baseret på den feedback vi fik om hvor intuitivt programmet var programmeret, samt hvor godt appen så ud. Det gjorde det nemmere at retfærdiggøre vores beslutning om at bruge AdobeXD.





Indbakke i AdobeXD prototypen, indbakken blev senere hen omdøbt til beskeder. Vores valg af AdobeXD var et eksperiment i at få specificeret kundens krav, da kunden selv ikke vidste hvad for et design til programmet som de ville have. Det gjorde at vi med brugertestene kunne eksperimentere for finde de elementer af prototypen som kunden kunne bedst lide og ændre eller fjerne dem som de havde problemer med. Udfra dette kan vi konkludere at AdobeXD har gjort det muligt at eksperimentere med UI til vores Android-app, med en tilfreds kunde som endemål. Eksperimentering i UI kommer stadig til at være en vigtig del når vi fortsætter med at udvikle Android appen.

## Unified Process (Tommy)

I dette projekt blev der brugt Unified Process da det er en måde at strukturere et softwareprojekts forløb således at der er større chance for at projektet bliver en succes. Ved at anvende Unified Process viste vi også at vi kunne løse et problem på systematisk vis og at vi kunne koordinere som et hold fra start til slut og sammen udvikle et endeligt softwareprodukt hvilket fuldendte dette læringsmål:

- varetage udviklingsorienterede situationer under anvendelse af systemudviklingsmetoder og tilhørende teknikker

Til at starte med lavede vi en risikoanalyse hvor vi analyserede de risici der kunne opstå i projektet. Vores hensigt med denne risikoanalyse var at have et dokument vi kunne referere til og have i baghovedet gennem projektets forløb. Her er et udsnit delt i to. (Bilag 68)

Del 1:

Risiko	Sandsynlighed i %	Konsekvensskala
En eller flere artefakter lever ikke op til kravene og skal laves om	0,90	3
En person er fraværende en dag til to dage	0,90	3
Mere end to personer er fraværende en til to dage	0,70	3
Et af artefakterne lever ikke op til kravene og skal laves om	1,00	2
Rapport bliver ikke skrevet en eller flere gange	0,50	4
Prototypen kan ikke vises	0,40	4
Løsningen lever ikke op til de funktionelle krav	0,50	3
En person er fraværende mere end to dage	0,40	3

Del 2:

Konsekvens	Prioritet	Revideret sandsynlighed	Revideret ko	Revideret po
Tid spildes på revidering	2,70	0,70	3	2,10
Artefakter/kode bliver ikke lavet	2,70	0,90	2	1,80
Artefakter/kode bliver ikke lavet	2,10	0,70	2	1,40
Tid spildes på revidering	2,00	0,90	2	1,80
Mere tid bliver brugt på at skrive de manglende sider	2,00	0,40	3	1,20
Kunden bliver utilfreds og vi får ikke feedback	1,60	0,15	4	0,60
Kunden bliver utilfreds	1,50	0,35	3	1,05
Artefakter/kode bliver ikke lavet	1,20	0,40	3	1,20

Samt konsekvensskalaen:

Grad	Definition
5	Projektet kan ikke færdiggøres til levering.
4	Projektet forsinkes så meget, at 3 eller flere use cases ikke bliver færdiggjorte.
3	Projektet forsinkes så meget, at 1-2 use cases ikke bliver færdiggjorte.
2	Projektet forsinkes med en halv dag.
1	Mindre frustrationer og forsinkelser på få timer.
0	Ingen konsekvens

I dette projekt fokuserede vi mest på de risici der medførte tidsspilde da det var de konsekvenser vi fandt mest væsentlige eftersom dette kunne være en stærk indikator på en lavere karakter da vi så ville ende med ikke at bruge vores tid fornuftigt. I en anden sammenhæng kunne vi have fokuseret på konsekvenser der kunne have påvirket profitten, men dette var ikke gældende da projektet er en eksamensopgave og ikke en rigtig business case.

Vores brug af risikoanalysen medførte et overblik over de scenarier der kunne opstå i projektet men alligevel blev risikoanalysen ikke særlig anvendt i projektet. Desuden fandt vi ud af at vi undervurderede hvor meget tid der skulle bruges på at implementere forskellige UI-elementer eller på at implementere Firebase databasen

og fremadrettet kunne vi have manglende erfaring som en risiko der kan medføre væsentligt tidsspilde.

## Faseplaner

Med henblik på UPs opdeling i faser ville vi lave en faseplan der tilnærmelsesvis kunne svare til dem der kræves af UP. Vi siger tilnærmelsesvis da vi i eksamensprojektet ikke kunne ændre vores bemandingsprofil ligesom det ville foregå i et normalt UP projekt, og derfor ville det ikke være praktisk at følge processen helt stringent. Alligevel vurderede vi at Unified Process' metoder kunne hjælpe med at strukturere projektet også selvom vi løst implementerede dem.

Id	Opgavetil	Opgavenavn	Varighed	Startdato	Slutdato	Foregående opgaver
1	✓	<b>Inception</b>	4 dage	ti 12-05-20	fr 15-05-20	
2	✓	<b>iteration #1</b>	4 dage	ti 12-05-20	fr 15-05-20	
3	✓	Oprette et repo	1 dag	ti 12-05-20	ti 12-05-20	
4	✓	Oprette et google doc til opgave	1 dag	ti 12-05-20	ti 12-05-20	
5	✓	Indholdsfortegnelse	1 dag	ti 12-05-20	ti 12-05-20	
6	✓	Indledning	1 dag	ti 12-05-20	ti 12-05-20	
7	✓	Problemformulering	1 dag	ti 12-05-20	ti 12-05-20	
8	✓	Use case diagram	1 dag	ti 12-05-20	ti 12-05-20	
9	✓	Færdiggøre PoC for Cloud Firestore	1 dag	ti 12-05-20	ti 12-05-20	
10	✓	Mockup af mobilapp	1 dag	ti 12-05-20	ti 12-05-20	
11	✓	Fully dressed use case	1 dag	on 13-05-20	on 13-05-20	
12	✓	Domænemodel	2 dage	on 13-05-20	to 14-05-20	
13	✓	PoC for firestore i en desktop app	1 dag	on 13-05-20	on 13-05-20	
14	✓	Skrive et afsnit om unified process	1 dag	on 13-05-20	on 13-05-20	
15	✓	Project structure i implementationmappen	1 dag	to 14-05-20	to 14-05-20	
16	✓	Mockup af desktop app i JavaFX	2 dage	to 14-05-20	fr 15-05-20	
17	✓	Risikoanalyse	1 dag	to 14-05-20	to 14-05-20	
18	✓	Få afklaring fra Anders på Complimenta og Exorgo Live integration	1 dag	fr 15-05-20	fr 15-05-20	
19	✓	Review risikoanalyse	1 dag	fr 15-05-20	fr 15-05-20	
20	✓	Skrive et afsnit om UML	1 dag	fr 15-05-20	fr 15-05-20	

Et snit af inceptionfasen lavet i Microsoft Project. (Bilag 69)

I vores første fase fokuserede vi på at lave proof of concepts for at forstå de muligheder vi havde med de libraries vi valgte at inkorporere i projektet. Der var især mange uhensigtsmæssigheder angående brugen af Googles Firebase database i både en desktop app og en mobile app da denne for det meste var egnet til mobile og web og ikke desktop. Dette medførte at vi brugte mere tid på inception fasen end der var hensigten selvom vi nåede vores milepæle med fungerende prototyper af alle use cases involverende en samtale mellem to brugere i en app.

24	✓	✚	<b>Elaboration</b>	<b>9 dage</b>	<b>ma 18-05-20</b>	<b>to 28-05-20</b>	
25	✓	✚	<b>iteration #2</b>	<b>5 dage</b>	<b>ma 18-05-20</b>	<b>fr 22-05-20</b>	
26	✓	✚	Review SSD03 og SSD04	1 dag	ma 18-05-20	ma 18-05-20	
27	✓	✚	Lav SSD01 og SSD02	1 dag	ma 18-05-20	ma 18-05-20	
28	✓	✚	Operationskontrakter	1 dag	ma 18-05-20	ma 18-05-20	
29	✓	✚	Business Model Canvas	1 dag	ma 18-05-20	ma 18-05-20	
30	✓	✚	Arbejde med feedback fra Chris	1 dag	ma 18-05-20	ma 18-05-20	
31	✓	✚	Lav SD til UC01 og UC02	1 dag	ti 19-05-20	ti 19-05-20	
32	✓	✚	Lav DCD til UC01 og UC02	2 dage	ti 19-05-20	on 20-05-20	
33	✓	✚	Opstart af androidprojekt (UI kode)	1 dag	ti 19-05-20	ti 19-05-20	
34	✓	✚	Lav SD til ref frames	1 dag	on 20-05-20	on 20-05-20	
35	✓	✚	Lave UML til UC03 og UC04	1 dag	to 21-05-20	to 21-05-20	
36	✓	✚	Lav android UI færdig	3 dage	on 20-05-20	fr 22-05-20	33
37	✓	✚	<b>iteration #3</b>	<b>5 dage</b>	<b>fr 22-05-20</b>	<b>to 28-05-20</b>	
38	✓	✚	Overgå til Clean Architecture	5 dage	fr 22-05-20	to 28-05-20	
39	✓	✚	Koble logik og UI sammen i desktop app	1 dag	fr 22-05-20	fr 22-05-20	
40	✓	✚	Lav UI til sletBruger i desktop app	1 dag	ma 25-05-20	ma 25-05-20	

Et snit af elaboration fasen lavet i Microsoft Project. (Bilag 69)

Da vi nåede elaboration fasen og, måske uhensigtsmæssigt, allerede havde et program der kunne vises til kunden, fokuserede vi fortsat på at implementere use cases involverende chat. I denne proces kom vi frem til at Clean Architecture ville være den bedste fit for vores program og derefter havde vi en passende arkitektur vi kunne arbejde videre med fremadrettet.

49	✓	✚	<b>Construction</b>	<b>10 dage</b>	<b>fr 29-05-20</b>	<b>to 11-06-20</b>	
50	✓	✚	<b>iteration #4</b>	<b>10 dage</b>	<b>fr 29-05-20</b>	<b>to 11-06-20</b>	
51	✓	✚	UI for vis beskeder opdateres live i desktop app'en	1 dag	fr 29-05-20	fr 29-05-20	
52	✓	✚	Skriver afsnit om finansiering færdig	1 dag	fr 29-05-20	fr 29-05-20	
53	✓	✚	Lav UC05 Tilknyt Patient	1 dag	fr 29-05-20	fr 29-05-20	
54	✓	✚	Ret diagrammer for UC01-04	1 dag	fr 29-05-20	fr 29-05-20	
55	✓	✚	Udvikle UI til kalender android	1 dag	fr 29-05-20	fr 29-05-20	
56	✓	✚	Udvikle UI til booking android	1 dag	fr 29-05-20	fr 29-05-20	
57	✓	✚	Få styr på brugertesten	1 dag	ma 01-06-20	ma 01-06-20	
58	✓	✚	Lave unit tests	1 dag	ma 01-06-20	ma 01-06-20	

Et snit af Construction fasen lavet i Microsoft Project. (Bilag 69)

Med milepælen til elaboration fasen nået, fortsatte vi med construction fasen. Vi tænkte at vi ville implementere så mange use cases som vi kunne nå men at projektet havde forløbet sig således at vi ikke kunne nå alt og måske aldrig blev klar til en acceptabel release vores kunde. Vores prioritet var at en use case skulle implementeres godt og ikke hurtigt så vi kunne nå flere da dette virkede mest praktisk i et eksamenssammenhæng også selvom det gik ud over vores kunde.

## Refleksioner til Unified Process

Brugen af Unified Process i vores projekt har været givende på den måde at vi har kunnet opnå en systematisk fremgang for vores proces, men alligevel kan implementeringen af systemudviklingsframework'et stadig forbedres. Vores faser og iterationer gav os et klart mål vi skulle nå men et specifikt dokument man kunne referere til for at se dette mål blev sjældent fremstillet, da vi i stedet prioriterede andre punkter såsom programmering og rapportskrivning, og derfor manglede der en klar organisering af vores brug af Unified Process. Dog som førnævnt kan vi ikke ændre vores bemandingsprofil i projekter såsom disse og derfor kan man også tale om at Unified Process slet ikke er passende i et eksamensprojektsammenhæng såsom denne og den ide kan vi tage med os fremadrettet.

## UML (Kelvin)

Vores løsning består af 2 programmer, som grundlæggende er meget ens. Der er dog nogle få mindre forskelle. Fx kan man som behandler oprette brugere både til patienter og andre behandlere, men det kan man ikke som patient. Vi har valgt at udforme alt UML fra patientens synspunkt, men i implementeringen har vi selvfølgelig taget højde for de små forskelle, der er i behandlerudgaven af programmet.

I dette afsnit vil vi forklare, hvordan vi lever op til studieordningens færdighedsmål for systemudvikling:

- Den studerende kan anvende fagområdets centrale teknikker og værktøjer til modellering af IT-systemer på analyse- og designniveau

Det vil vi gøre ved at vise, hvilke UML-artefakter, vi har udarbejdet, samt forklare, på hvilken baggrund, vi har udviklet dem.

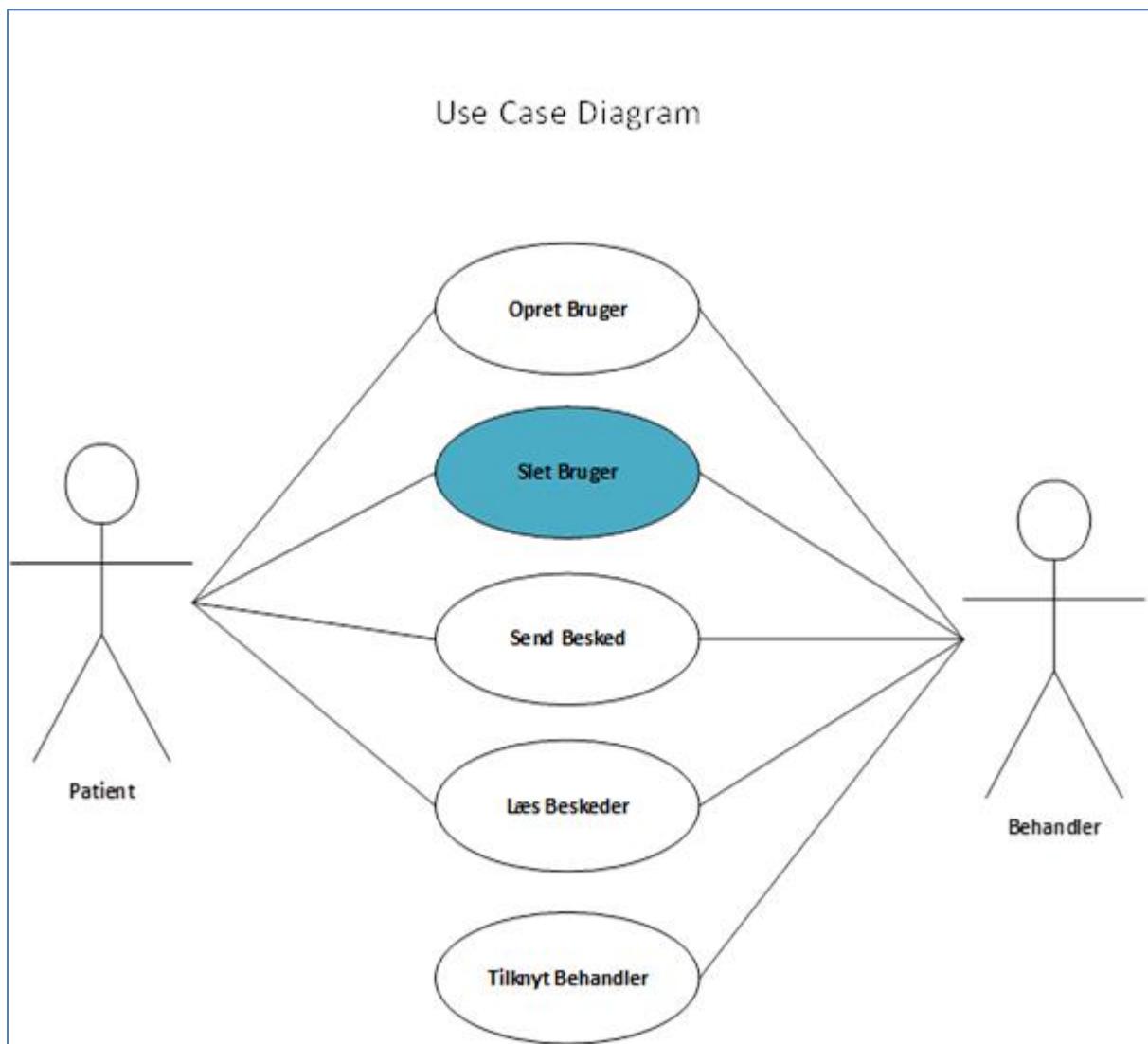
## Sporbarhed (Tommy)

Før vi starter UML afsnittene så har vi et afsnit til sporbarhed således at de dybdegående afsnit til specifikke artefakter ikke behøver at tage udgangspunkt i samme use case.

Et vigtigt element i brugen af UML er at det giver sporbarhed samt et referencedokument til alle parter der kunne have gavn af det, såsom eksterne programmører der kunne deltage i projektet samt kunden selv der kunne have behov for dem i et juridisk sammenhæng. Ved at bruge UML og vise sporbarhed så opfylder vi dette læringsmål:

- formidle systemudviklingens proces og produkt til relevante interessenter – herunder sikre sporbarhed

For at vise sporbarhed vil vi her bruge Slet Bruger use casen som eksempel hvor vi starter med at se den som en del af use case diagrammet, her oplyst i blå.

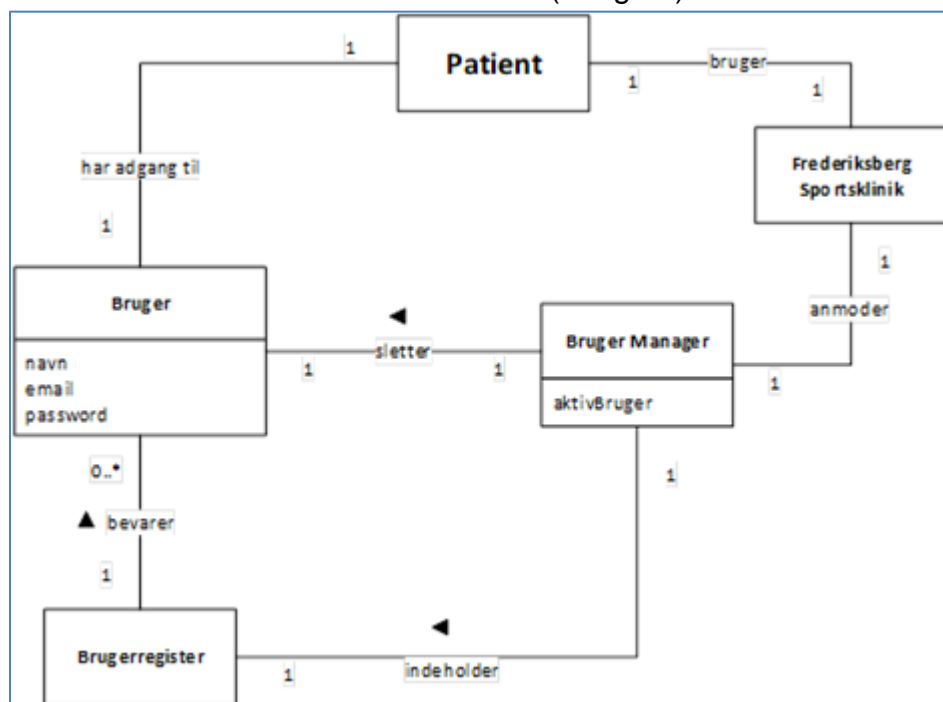


(Bilag 8) Vi kan se at det er en patient eller en behandler der interagerer med use casen som en del af systemet men de nærmere detaljer fås fra den fully dressed use case, her vist i snit.

<b>Stakeholders and Interest</b>	Patient - de vil gerne have en brugerkonto slettet da de ikke længere har brug for at have en.
<b>Preconditions</b>	N/A
<b>Success Guarantee</b>	Patientens brugerkonto bliver slettet fra registeret.
<b>Main Success Scenario</b>	<ol style="list-style-type: none"> <li>1. Patienten vil gerne have en bruger slettet.</li> <li>2. Patienten angiver password.</li> <li>3. Frederiksberg Sportsklinik tjekker, om passwordet er korrekt.</li> <li>4. Frederiksberg Sportsklinik accepterer passwordet.</li> <li>5. Frederiksberg Sportsklinik sletter brugeren.</li> </ol>

(Bilag 10) I Denne er det værd at nævne at den er udarbejdet sammen med kunden og derfor bruger den generelt sprog der ikke refererer til noget programmering. Dette åbner også op for at modellering af designet ikke er fastlåst til en bestemt implementering og derpå kan give vi os selv større rammer til at udtænke en løsning. Ellers er det væsentlige at patienten kan spores både fra use case diagrammet samt den fully dressed use case. Behandleren er her ikke vist og heller ikke efterfølgende da vi nu i eksemplerne kun vil fokusere på patienten som aktør, eftersom Slet Bruger use casen udføres på samme måde for begge.

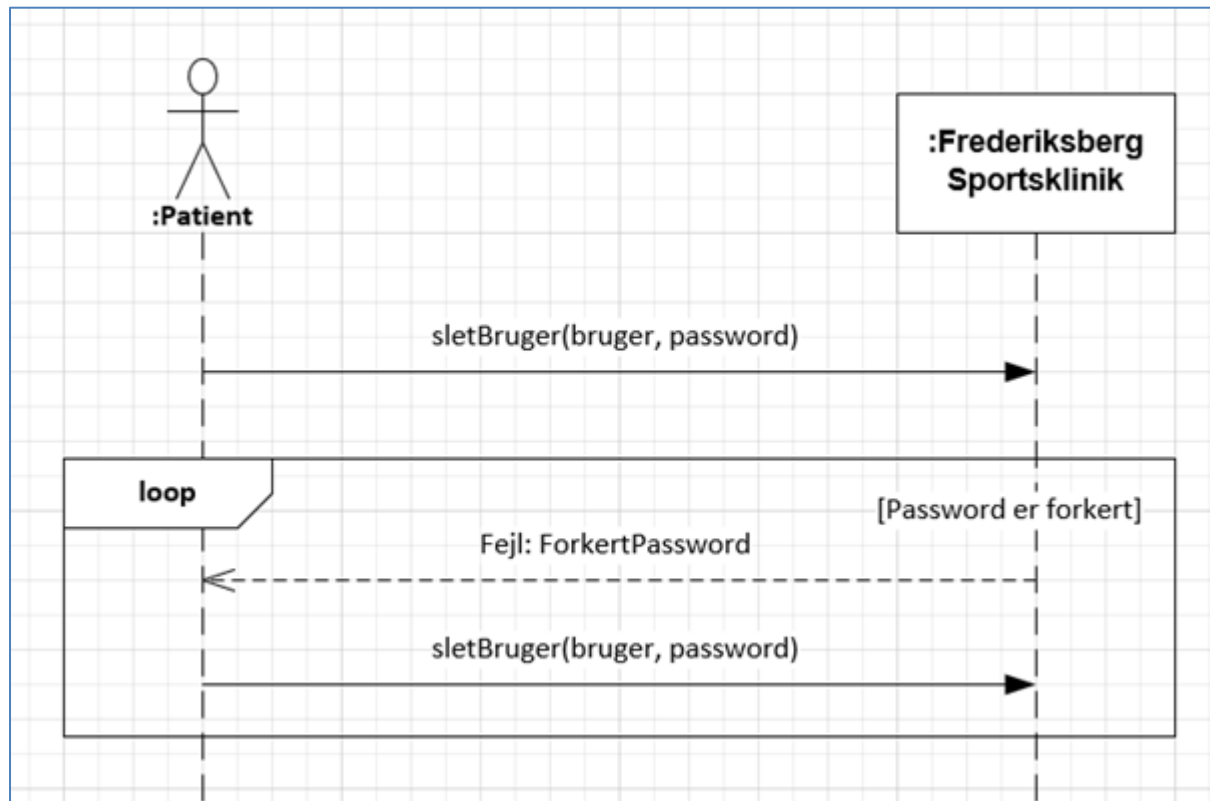
Efter use casen er domænemodellen. (Bilag 15)



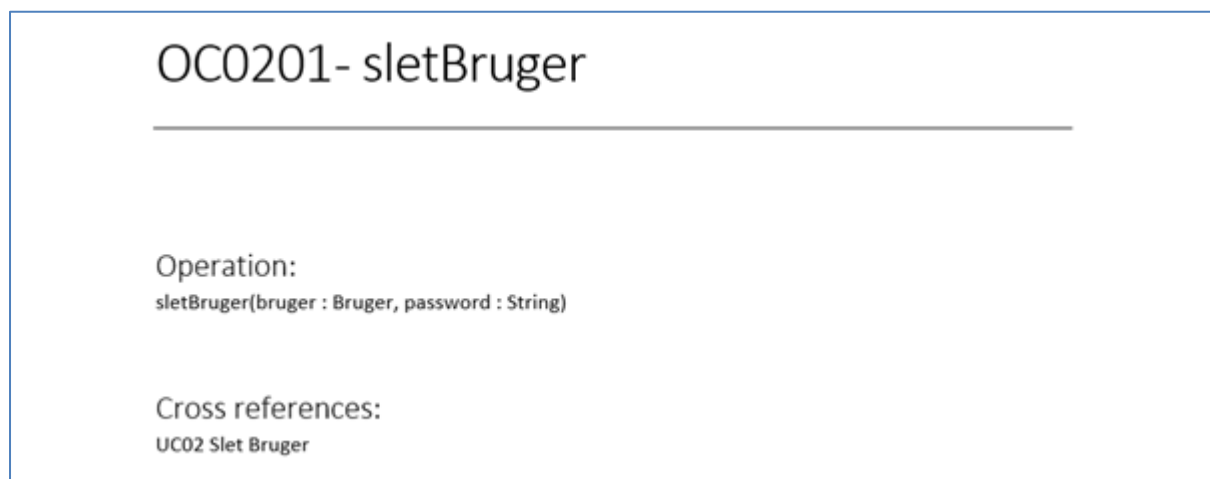
Vi bruger de samme udtryk som f.eks. sletter, hvilket også var i den fully dressed use case og de koncepter der vises her er overført fra samme. Vi udvider dog med



koncepter der er nødt til at eksistere i domænet før en løsning kan udtænkes. Fra det statiske fortsætter vi med det ikke-statiske i systemsekvensdiagrammet.



(Bilag 20) Igen er de samme udtryk gennemgående i artefakterne og det kan ses at patienten kun behøver at interagere med systemet en gang hvis han vil slette sin bruger, medmindre at personen indtaster et forkert password. Af denne ene interaktion med systemet udspringer en operationskontrakt.



Samt dens preconditions og postconditions.



#### Preconditions:

En Bruger bruger eksisterer

En liste af brugere brugerliste eksisterer

brugerliste indeholder bruger

Aktiv bruger eksisterer

Et password pw er angivet

pw er hashed til hashed password hpw

hpw stemmer overens med bruger.password

#### Postconditions:

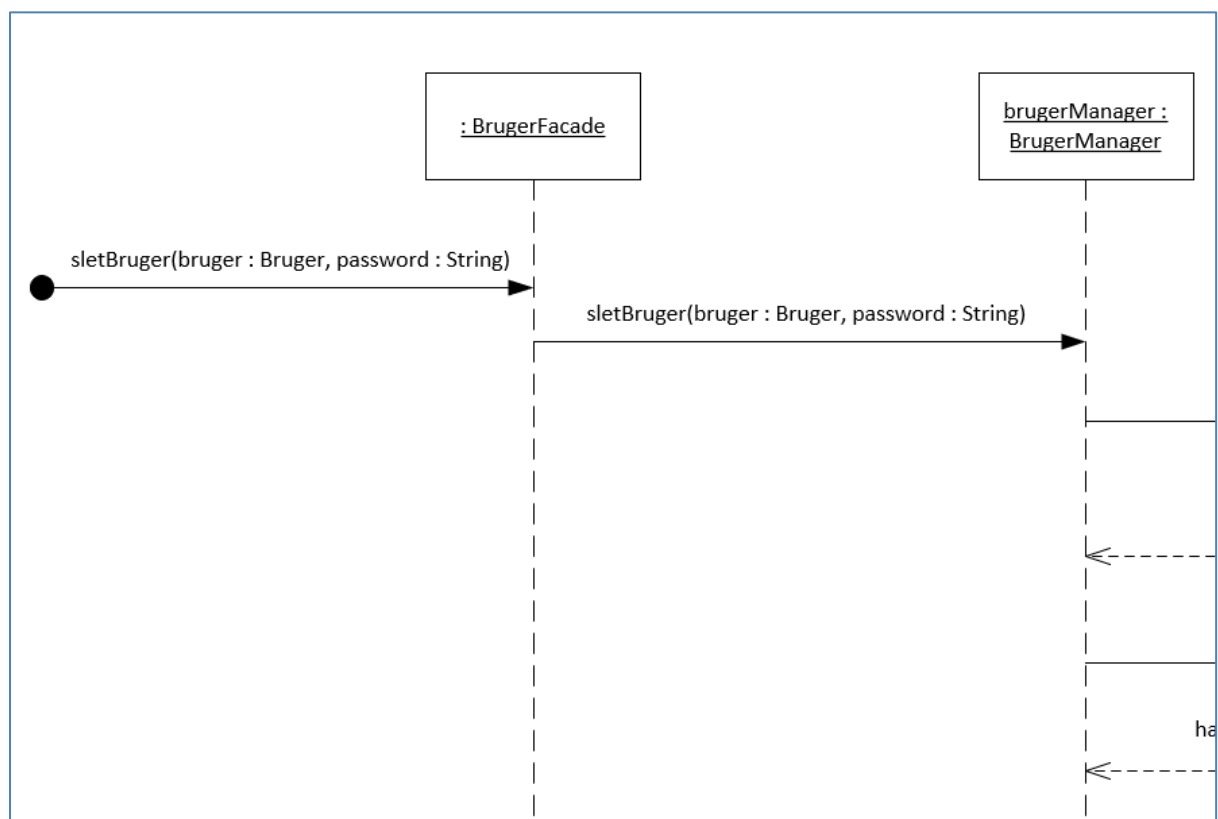
Brugerliste blev sat til ikke at indeholde bruger

Aktiv bruger blev sat til null

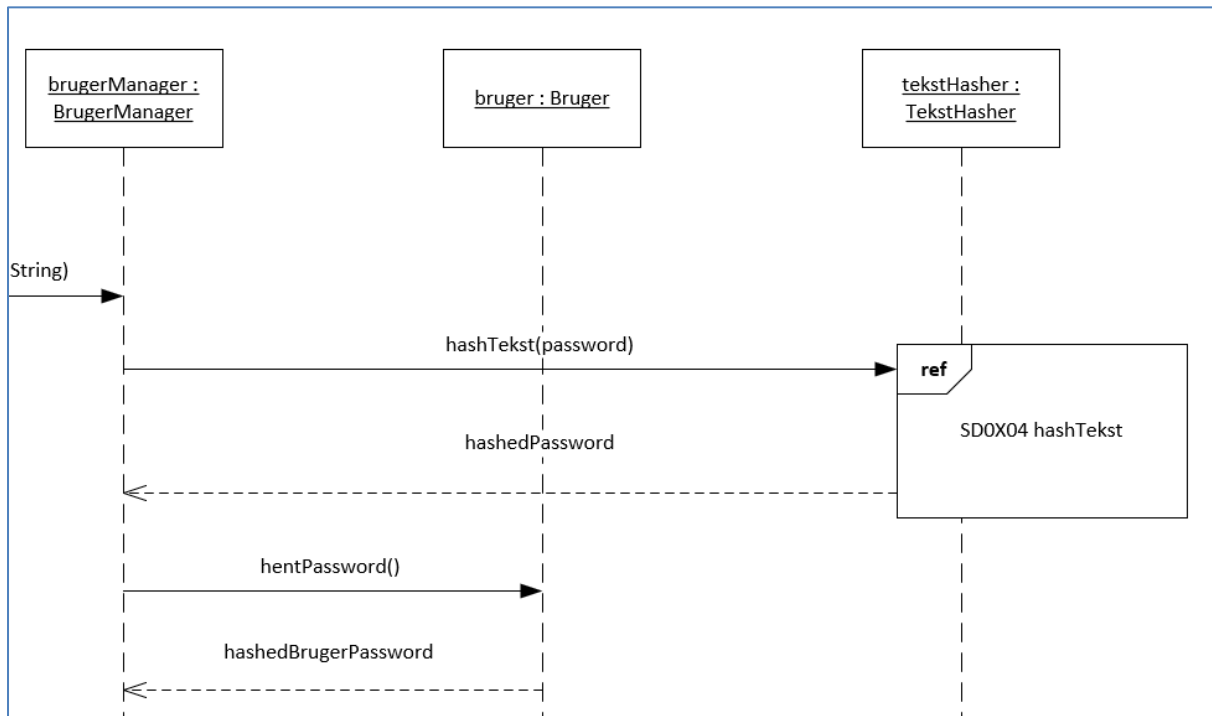
(Bilag 28) Her vises helt specifik de tilstande der må være gældende før operationen kan udføres samt det der er gældende i domænet efterfølgende. På grundlaget af denne kan vi designe systemet.

Sekvensdiagram i to dele: (Bilag 41)

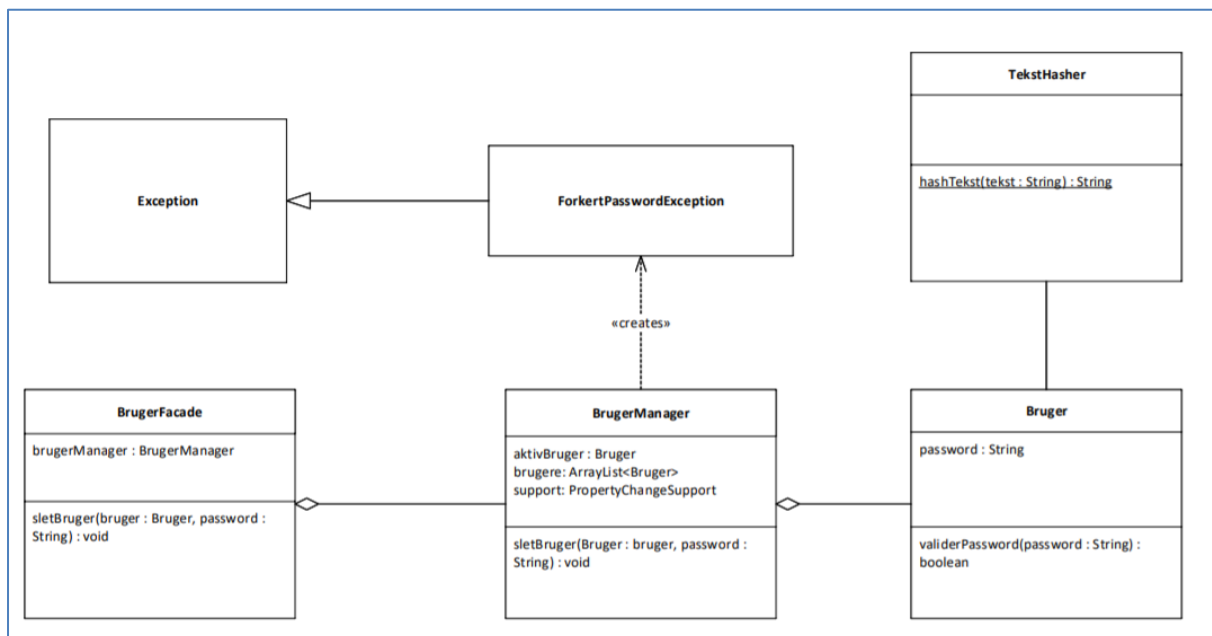
Del 1:



## Del 2:



## Klassediagram: (Bilag 60)



Baseret på operationskontrakten kan vi repræsentere den hele sekvens når patienten anmoder om at få slettet sin bruger, samt vise klassediagrammet, og dermed har vi fuldt overblik over alle elementer vedrørende use casen Slet Bruger, i hvert fald når en patient er aktøren.

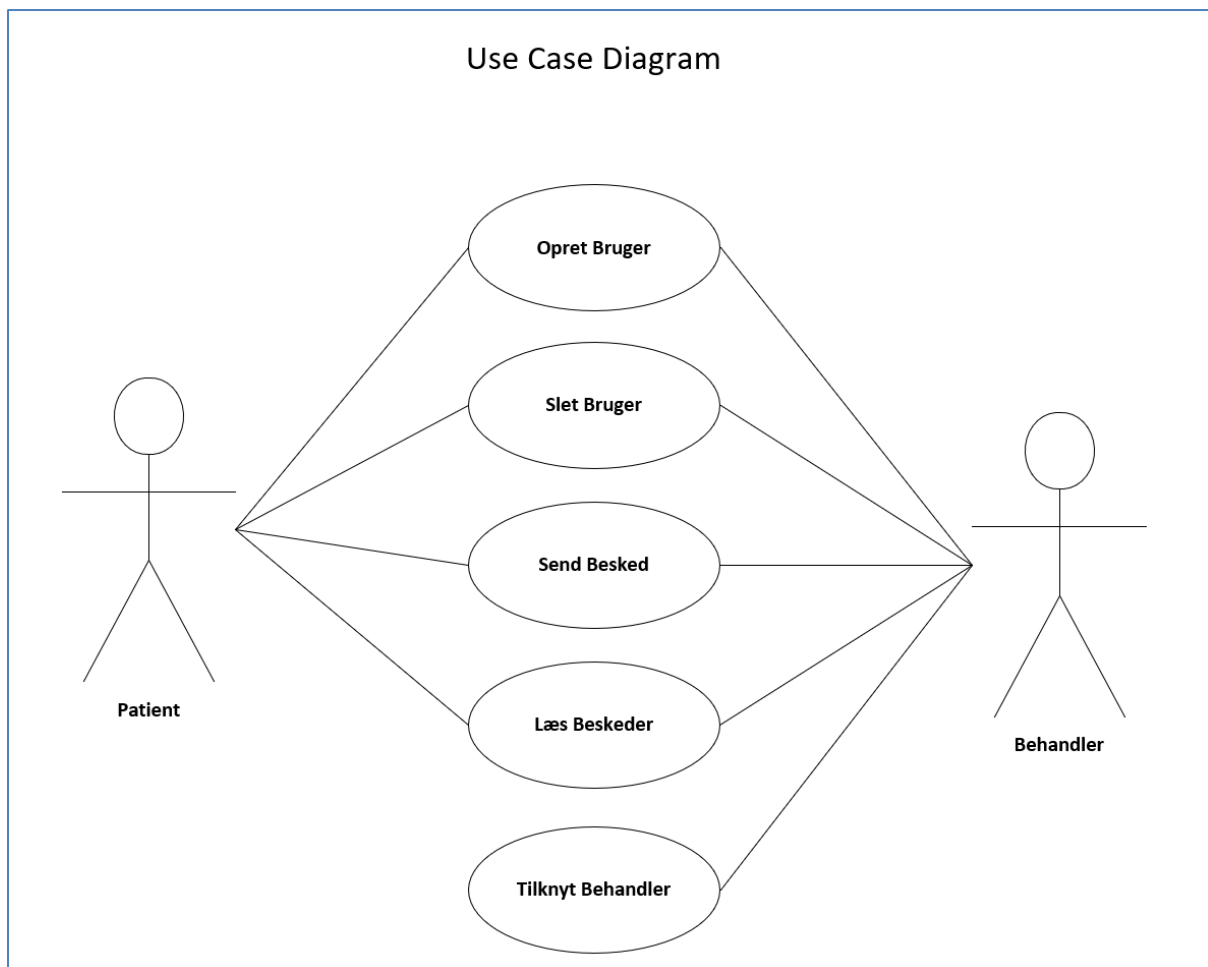
Med dette kan sporbarhed ses i vores projekt, her i en specifik case således at de næste afsnit ikke nødvendigvis behøver at bruge samme use case, og der er givet indblik i hvordan vi har indarbejdet sporbarhed i projektet. Samlet for alt UML er at vi med den har skabt faglig dokumentation for den proces vi har gennemgået under modellering af systemet, dog ikke alle steder. Af tidsmæssige årsager og da der kun er marginale forskelle i de use cases hvor patient og behandler overlapper, så har vi valgt i disse tilfælde kun at have UML-artefakter tilpassede til når aktøren er patient. Dette gør naturligvis at der er små ukendtheder vi ikke kan dokumentere for i vores program, men overordnet mener vi ikke at dette gør stor skade på det overblik som vi har fået med de resterende UML-artefakter.

## Use Case Diagrams (Kelvin)

I dette afsnit viser jeg hvordan vi anvender use case diagrammer til at identificere use cases, definere problemstillingen og derved demonstrere hvordan at vi lever op til studieordnings mål:

- udviklingsbaseret viden om kvalitetskriteriers betydning for systemudviklingsprocessen og systemets endelige udformning

Use case diagrammet et af de første led i kravindsamling, som vi arbejder med. Use case diagrammet bruges til at vise actoren og use casenes relationer på et simpelt niveau. Vi har simplificeret vores workflow, da de to programmer vi har udviklet er meget ens, et eksempel kunne være UC04 Læs Beskeder:



I læs beskeder for man vist teksten i de beskeder man har modtaget. Det er præcis samme use case om man er klient eller behandler og der også præcis samme workflow i systemet. Derfor giver det mening at have denne use case præsenteret i et enkelt diagram. Man kan argumentere for at man skulle have lavet diagrammet som to forskellige diagrammer siden use case opret bruger har i nogle tilfælde forskellige workflows. Der ville også være manglende overblik hvis vi havde flere use cases en nødvendigt. Vores supporting actors i dette tilfælde ville være exorlive og complimenta, siden vores program skal kommunikere med de to apps. Man kan forestille sig en situation som behandler hvor man skal oprette en klient. Men i de fleste situationer er det det samme og derfor giver det mere overblik at sætte dem sammen. Det viser at vi har tænkt over at de endelige systemer kommer til at fungere overvejende ens.

## Use Cases

I dette afsnit viser jeg hvordan vi anvender use cases til at analysere systemet samt forklare, hvordan at vi lever op til første del af studieordningsmål:

- **vurdere praksisnære problemstillinger under inddragelse af brugere** og anvende hensigtsmæssige mønstre i modelleringen

Vi arbejder videre på analyseprocessen som er beskrevet i use case diagrammet. Et eksempel på dette kunne være UC01 Opret Bruger:

### Fully-dressed Use Case

Use Case Section	Comment
Use Case Name	UC01 Opret Bruger
Scope	Frederiksberg Sportsklinik
Level	Brugermål
Primary Actor	Patient, Behandler

I UC01 Opret Bruger (bilag 9), ser vi hvordan en bruger bliver opret og hvilket trin der skal gennemgås for at brugeren kan oprettes. Trinene til at opfylde Opret Bruger bliver lavet af brugeren selv, derfor er det et Brugermål. I main success scenario for oprettelse af en bruger ser vi hvad en bruger skal gøre for at oprette en bruger:

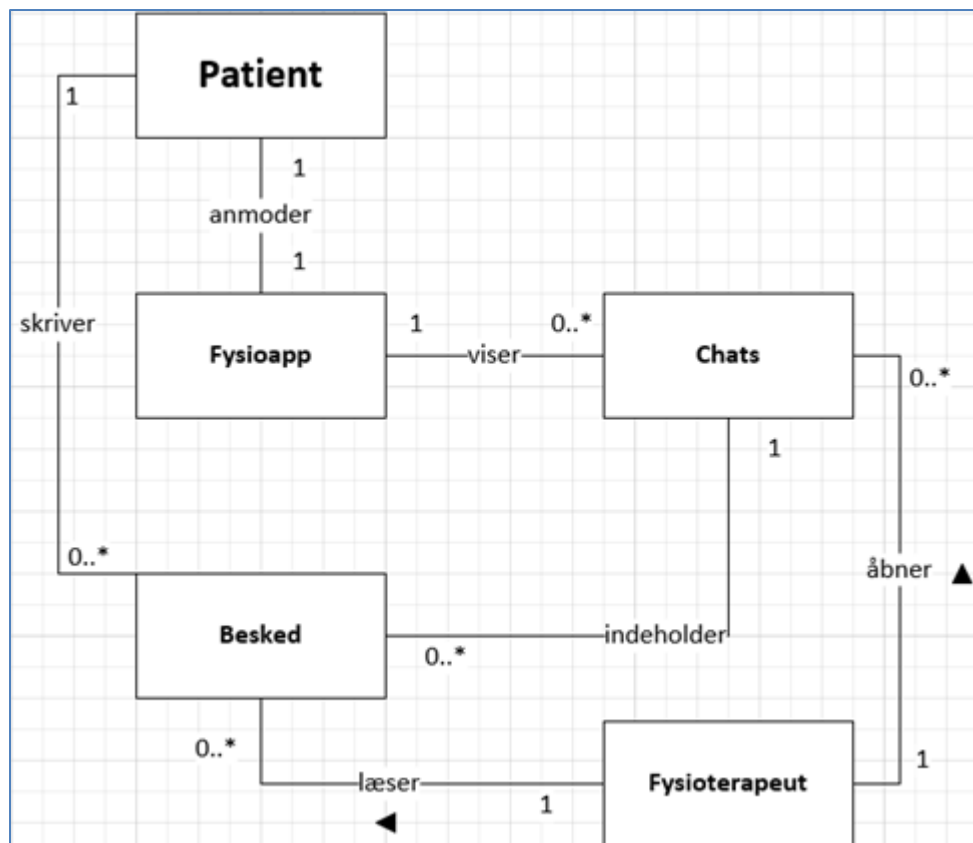
Main Success Scenario	<ol style="list-style-type: none"> <li>1. Patient vil gerne skabe en bruger.</li> <li>2. Frederiksberg Sportsklinik accepterer anmodningen.</li> <li>3. Patient angiver navn, password og email.</li> <li>4. Frederiksberg Sportsklinik tjekker patientregisteret, om der allerede er en bruger med denne email.</li> <li>5. Frederiksberg Sportsklinik accepterer e-mailen.</li> <li>6. Frederiksberg Sportsklinik tjekker, om navnet er gyldigt.</li> <li>7. Frederiksberg Sportsklinik accepterer navnet.</li> <li>8. Frederiksberg Sportsklinik tjekker, om passwordet er gyldigt.</li> <li>9. Frederiksberg Sportsklinik accepterer passwordet.</li> <li>10. Frederiksberg Sportsklinik skaber brugeren.</li> <li>11. Patienten tager sin nye bruger i brug.</li> </ol>
-----------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

De funktionelle krav kan man se i main success scenario hvor processen for brugeroprettelse vises med trin, som at brugeren skal angive et navn, password og e-mail som så tjekkes og godtages af systemet. I Extensions ser vi så de alternative

scenarier programmet går igennem før den acceptere og fortsætter i main success scenario. De adfærdsmæssige krav bliver vist i stakeholders samt level. Man kan argumentere for at opret bruger kan erstattes med en online bruger som for eksempel en google konto som kan linkes til Frederiksberg Sportsklinik i stedet for vores eget brugersystem. En situation hvor brugeren allerede har en google konto eller lignende kan forestilles. En lokal bruger vil gøre det mere sikkert for klienten når der ikke er ekstra data om brugeren som bliver delt.

## Domænemodeller

I dette afsnit viser jeg hvordan vi anvender Domænemodeller til at analysere systemet. Vi arbejder videre på den analyseproces som er beskrevet i use casene. Det eksempel som vi bruger er DOM 03 Send Besked:



I DOM03 send besked (bilag 16) ser man relationerne mellem de forskellige klasser, som viser de bemærkelsesværdige koncepter og/eller elementer af Send Besked funktionen. Vi har tilføjet domænerne ude fra vores kundes krav. Kunden ville have at patienterne kan få vist beskeder fra fysioterapeuten og at begge parter kan skrive

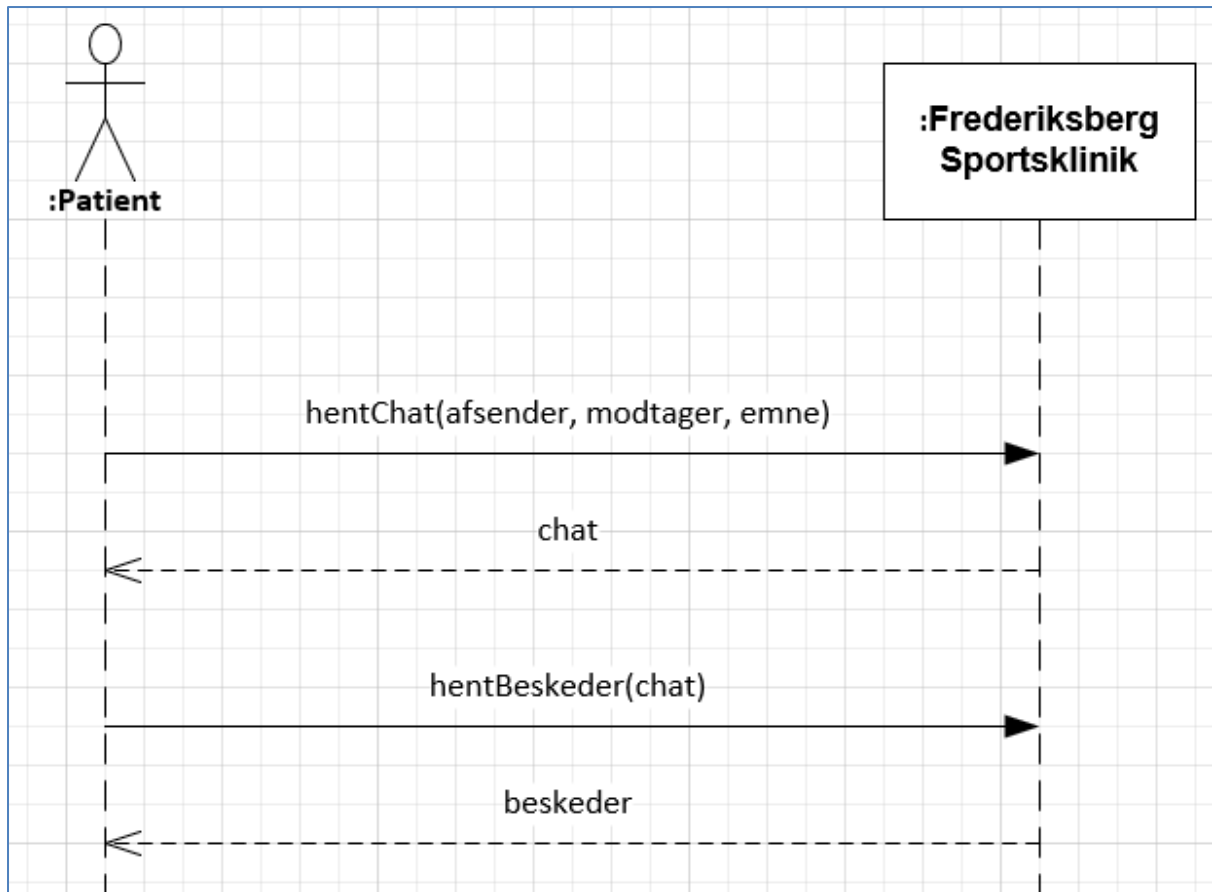
til hinanden. Man kan argumentere om chat funktionen er nødvendig når fysioterapeuten bare læser beskederne fra patienten, siden det skaber flere trin som fysioterapeuten skal igennem. Vi vælger alligevel at bruge chat funktionen da det skaber overblik og gør det nemmere at organisere beskeder for fysioterapeuterne og patienterne.

## Systemsekevensdiagrammer

Systemsekvensdiagram bruger vi til at vise hvordan at programmerne fungerer ved at vise hvordan at koden ville eksekveres. Systemsekvensdiagrammet er relateret til use case 03 som den er baseret på og Domænemodellen som også trækker fra use casen. Her er et billede af UC04 - Læs Beskeder:

<b>Success Guarantee</b>	Patient og behandler kan læse beskeder fra hinanden.
<b>Main Success Scenario</b>	<ol style="list-style-type: none"><li>1. Patient vil gerne læse beskeder fra sin behandler.</li><li>2. Patient vælger samtale.</li><li>3. Frederiksberg Sportsklinik viser beskedhistorik for samtalen.</li><li>4. Patient læser beskeden.</li></ol>

I UC04 har vi et Main Success Scenario som beskriver trin for trin hvad der skal ske for at vi opnår succes. Disse trin skal vi omskrive, så det passer med SSD syntax. Det kan vi så se i SSD04 Læs Beskeder:

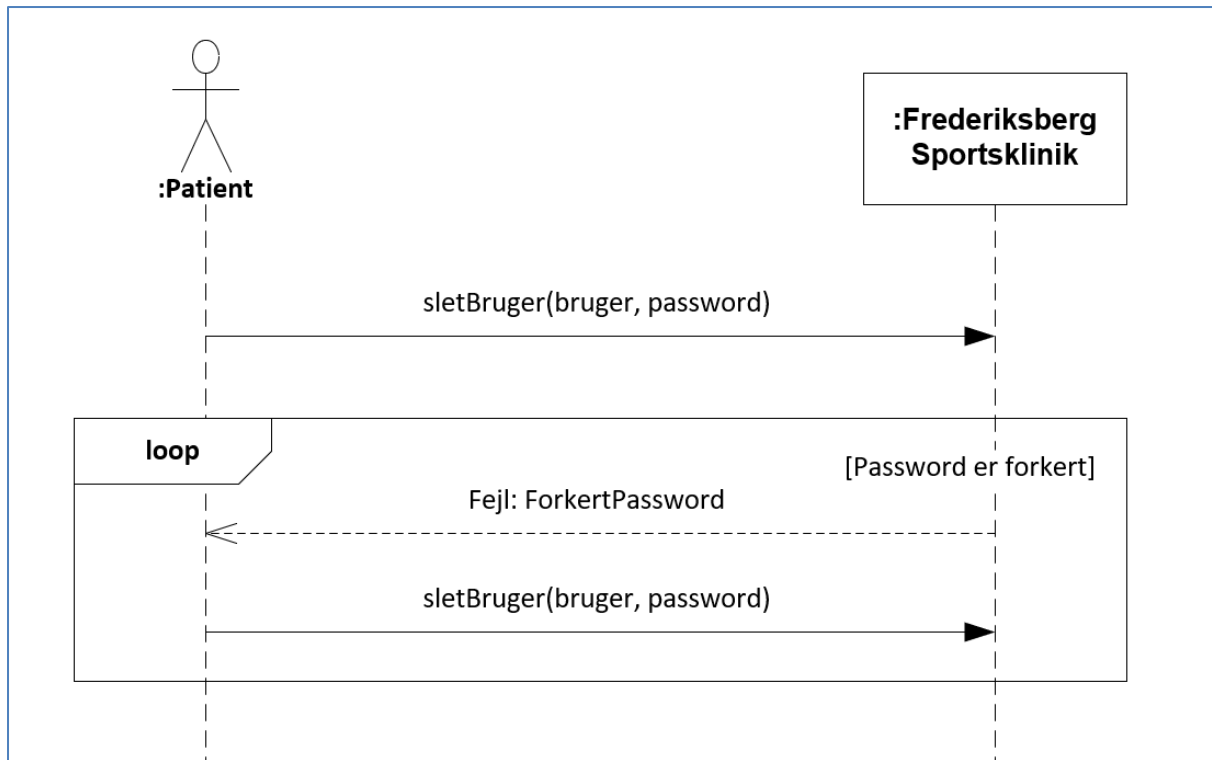


I SSD04 - læs besked viser vi hvordan vi omskriver main success scenario, samt viser diagrammets funktion. SSD04 er en simpel process som kunden går igennem for at se sine beskeder. Først beder kunden Frederiksberg Sportsklinik om at han kan se sine tidligere chats via `hentChat(afsender, modtager, emne)`. Frederiksbergsportsklinik viser så patienten's tidligere chats i return statementen `chat`. Bagefter spørger kunden om de kan se deres tidligere beskeder inde i `hentChat`, hvilket de gør med `hentBeskeder(chat)`. Frederiksberg Sportsklinik returnere så patientens beskeder. Det kan argumenteres at det ikke er nødvendigt at brugeren skal først ind i sine chat historie før at de kan se deres beskeder. Grunden til at vi har gjort det alligevel er fordi det gøre det mere muligt for programmet at organisere de forskellige chats til den individuelle patient i stedet for en hel masse uorganiseret beskeder.

### Operationskontrakter (Benjamin)

Operationskrakterne arbejder videre med operationerne fra SSD'et og beskriver dem i detaljer. Her bliver operationerne uddybet med preconditions og postconditions, hvor postconditions er de vigtigste (Larman 2004). Lad os se på SSD'et for Slet Bruger:





I dette SSD er der altså kun en enkelt operation, sletBruger. Før sletBruger kan eksekveres, er der nogle preconditions, som skal være mødt. Vi ved fra domænemodellen, at brugeren af programmet skal være logged ind på en Bruger og at denne bruger opbevares i et brugerregister. Vi kan se på SSD'et, at der skal angives et password, som skal matche med brugerens password. Derudover mangler vi at tage højde for, at password skal opbevares i en database, og derfor skal hashes for at overholde GDPR-lovgivningen. Nu har vi nok information til at kunne formulere vores preconditions:

- En Bruger bruger eksisterer
- En liste af brugere brugerliste eksisterer
- brugerliste indeholder bruger
- Aktiv bruger eksisterer
- Et password pw er angivet
- pw er hashed til hashed password hpw
- hpw stemmer overens med bruger.password

Selvom der kun er tale om en enkelt operation, så har den forholdsvis mange preconditions, men det giver også god mening, da sletning af brugere fra systemet kan have alvorlige konsekvenser for både brugen af systemet, men kan også forårsage runtime errors, hvis der ikke er taget højde for det. Derfor skal vi være helt sikre på, at alle de gældende preconditions er overholdt, inden en bruger slettes fra systemet. Når en bruger er slettet, har vi formuleret følgende postconditions:

- Brugerliste blev sat til ikke at indeholde bruger

- Aktiv bruger blev sat til null

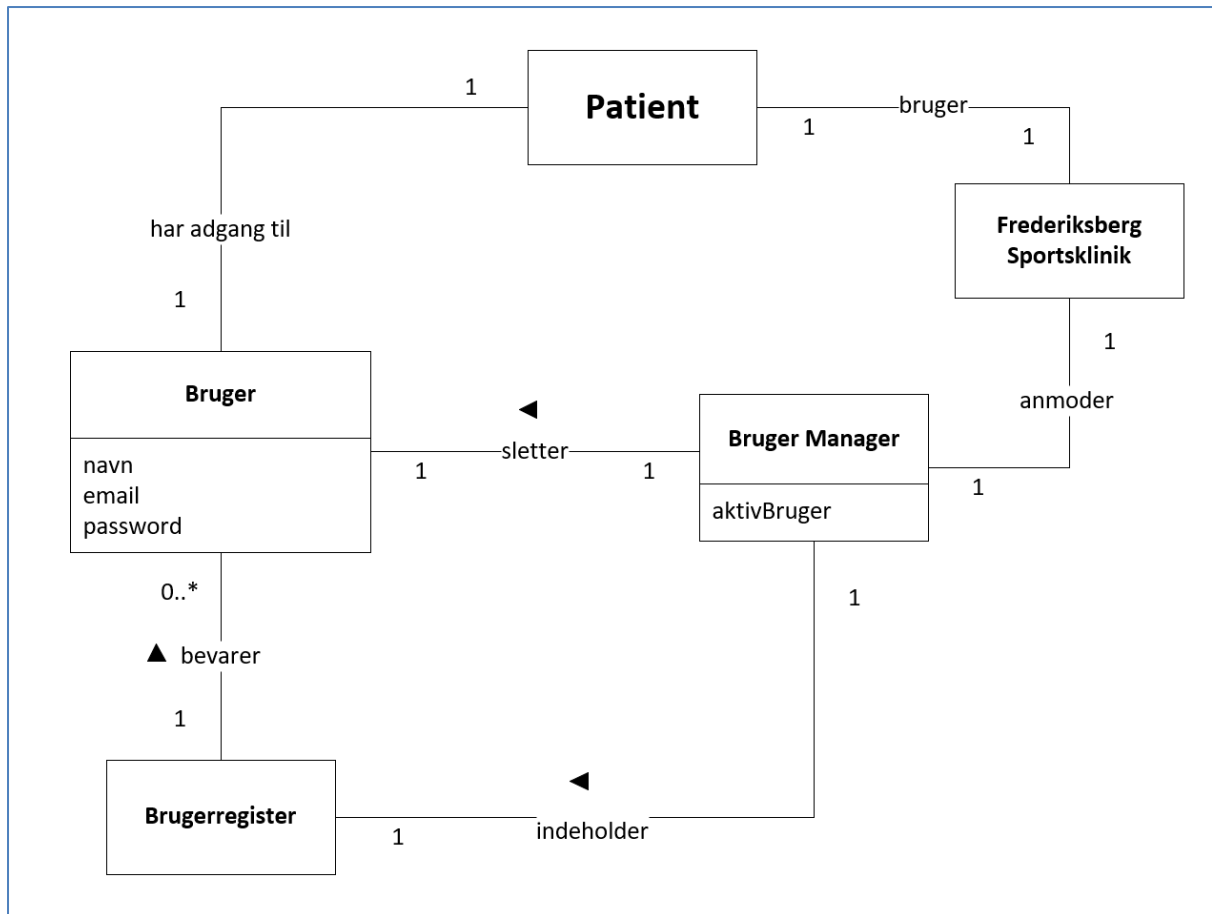
Når en bruger slettes fra systemet, så opbevares den ikke længere i brugerlisten, og da brugeren ikke længere eksisterer, kan den heller ikke være sat til at være den aktive bruger, så derfor sættes denne til null. Det skal nævnes, at `sletBruger` kan have lidt forskellige work flows i de to programmer, fordi der er forskellige situationer, hvor en bruger skal slettes:

- 1) Brugeren er patient og kan kun slette sin egen bruger
- 2) Brugeren er behandler og ønsker at slette sin egen bruger
- 3) Brugeren er behandler og ønsker at slette en patients eller en anden behandlers bruger

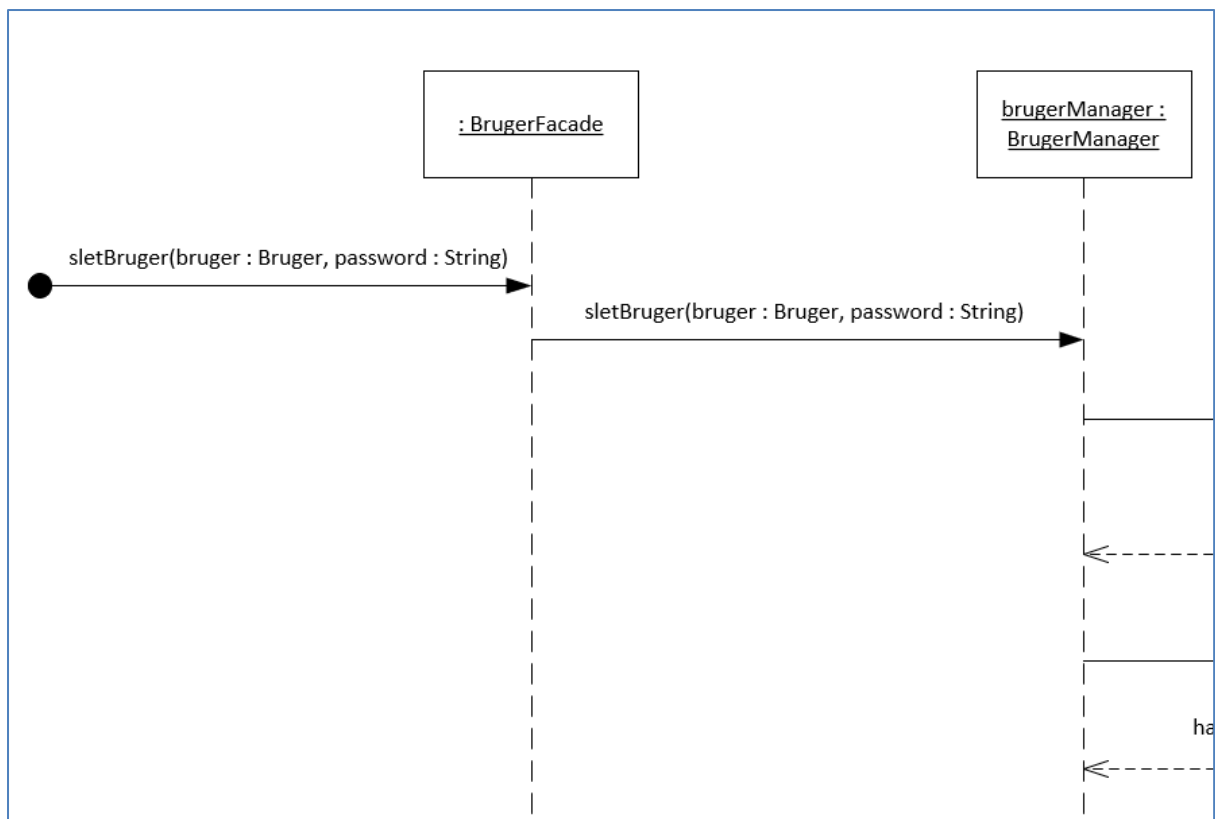
Situation 1 og 2 passer på det workflow, der allerede er beskrevet, men situation 3 har andre preconditions og postconditions. I dette tilfælde kender behandleren ikke brugerens password, så operationen laver derfor et emailtjek i stedet. Behandleren vil også stadig være den aktive bruger af systemet efter operationen er fuldført, så derfor skal den aktive bruger ikke ændres. Vi har dog valgt at beskrive disse situationer som samme operation i vores UML, da de grundlæggende er meget ens, men har selvfølgelig taget højde for forskellen i implementationen. Den fulde operationskontrakt for `sletBruger` kan ses i bilag 28.

## Sekvensdiagrammer

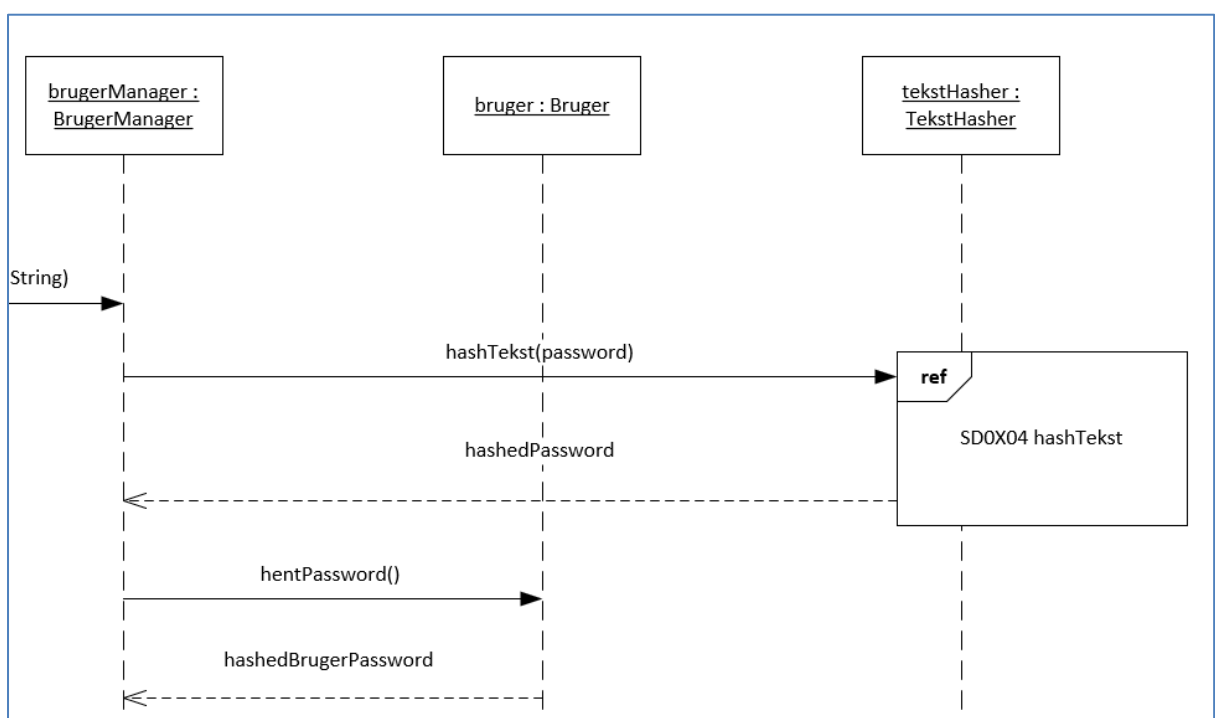
I dette afsnit vil jeg vise, hvordan vi anvender sekvensdiagrammer til at designe systemet samt forklare. I designdelen af Unified Process arbejder vi videre på metoderne, som blev beskrevet med operationskontrakter. Sekvensdiagrammet bruges til at vise sekvensen af metodekald i systemet. Vi er nu forpligtet til at repræsentere objekter, metodenavne, variable og parametre mm. ved deres egentlige navne og typer, som de kommer til at fremstå i koden (Larman 2004). Vi har brugt sekvensdiagrammer til at designe koden til vores system, så lad os se på et eksempel fra UC02 Slet Bruger. Vi starter med at se på domænemodellen, for at forstå, hvilke koncepter, klasserne stammer fra:



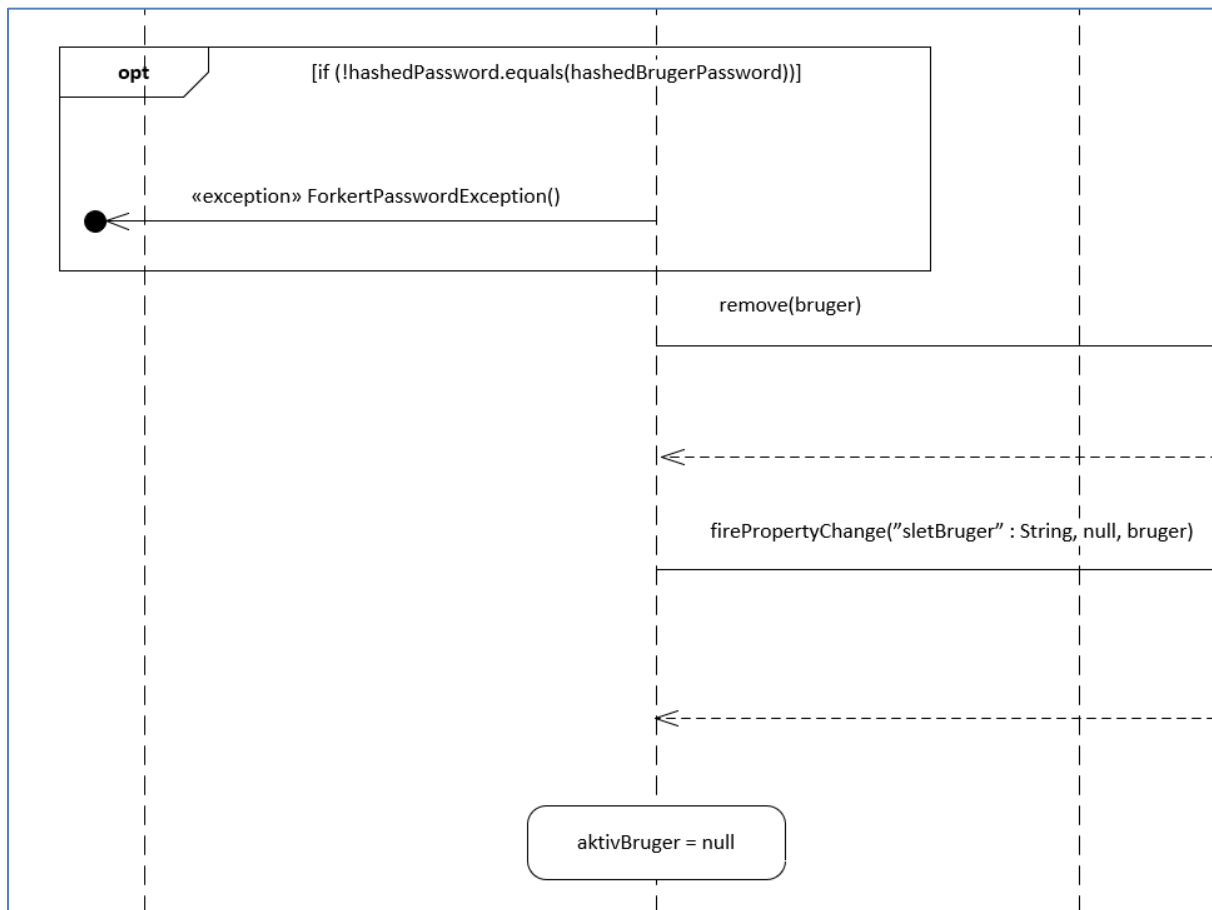
Patienten er brugeren af programmet, hvilket vil sige at det er et eksternt koncept og derfor ikke noget, der skal være en del af programmet, og Frederiksberg Sportsklinik er selve programmet. Disse to koncepter er ikke interessante i designfasen, fordi de ikke kommer til at være en del af implementeringen fremadrettet. Vi kan derfor fokusere på koncepterne Bruger, Brugerregister og Bruger Manager. Som vi kan se, så er Bruger Manager et koncept, som styrer Brugerregisteret ved at slette brugere (og i UC01 oprette brugere). Med den styrende rolle, som Bruger Manager nu engang har, så vil det være naturligt at gøre Bruger Manager til en klasse, som tager imod de metodekald, der foretager ændringer i brugerregisteret. Den første metode i UC02 beskrives i OC0201 som `sletBruger(bruger : Bruger, password : String)` (se bilag 28), og vi har lige set på dens preconditions. Alt denne information skal så anvendes i sekvensdiagrammet. Lad os se på første del af SD0201 (SD'et kan ses i sin helhed i bilag 41):



Her ser vi først et objekt af klassen BrugerFacade, som tager imod metodekaldet sletBruger. Vi anvender et facademønster, fordi vi har valgt at bruge en clean arkitektur, og da er det praktisk med et enkelt koblingspunkt for UI-elementer, men det kan du læse mere om i afsnittet om clean architecture. Derfra sendes metodekaldet direkte videre til klassen BrugerManager, hvor det interessante sker:



En precondition var, at det angivne password skal matche med det password, som brugeren indeholder, men da vi gerne vil sørge for en passende sikkerhed i varetægt af brugerens informationer, skal passwordet først hashes, inden det sammenlignes. Derfor går der et metodekald hashTekst til et objekt af TekstHasher-klassen. Jeg vil ikke gå i dybden med, hvordan hashing foregår, da det kan ses i bilag 52 SD0X04 hashTekst. Efter det er blevet hashed henter vi passwordet fra brugeren, og så er vi klar til at sammenligne de to password:



Hvis de to password ikke matcher, fanger vi en exception ForkertPasswordException, som bliver lost til det punkt, metodekaldet oprindede fra. Hvis passwordtjekket ikke returnerer en exception, bliver der sendt et remove-kald til brugerlisten brugere. Derefter sker der et observerkald, firePropertyChange, så observerne kan se, at der er sket en ændring, også sættes aktivBruger til null. Til sidst sendes kontrollen tilbage som lost til det punkt, metodekaldet oprindede fra. Grundet skemaets størrelse er det svært at vise alle objekterne her, så jeg vil igen henvise til det fulde SD i bilag 41.

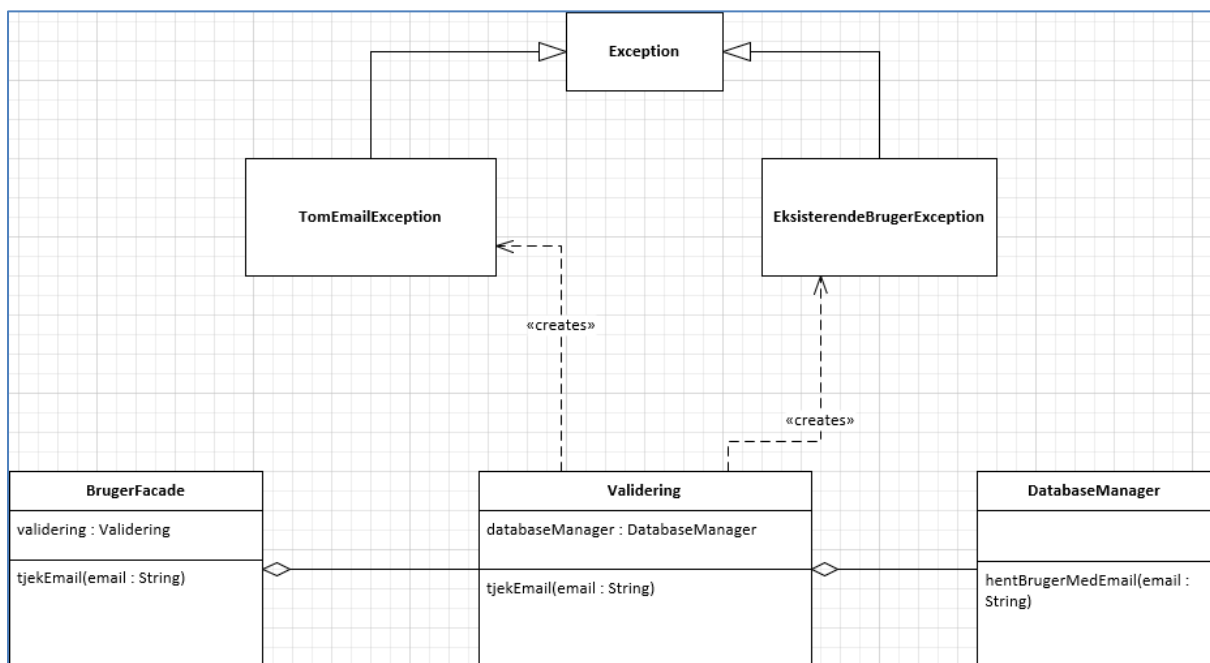
I operationskontrakten for sletBruger havde vi følgende postconditions:

- Brugerliste blev sat til ikke at indeholde bruger
- Aktiv bruger blev sat til null

Og vi kan se, at sekvensdiagrammet opfylder disse.

## Klassediagrammer (Kelvin)

I dette afsnit vil jeg vise hvordan at vi designer systemet via Klassediagrammer samt forklare. Klassediagrammer bliver lavet samtidig med sekvensdiagrammer. Klassediagrammer bruges til at illustrere klasser, interfaces, association. De bruges også til at visualisere domæne modeller (Iarman 2004). Et eksempel kunne være DCD0101 tjekEmail:



I DCD0101 tjekEmail ser vi processen for hvordan at programmet tjekker om der eksistere en email i systemet. Det starter den med at gøre i klassen **BrugerFacade**, som har attributten `validering : Validering` og metoden `tjekEmail(email : String)`. **BrugerFacade** indeholder klassen **Validering** som bliver vist via en aggregation line. **BrugerFacade** sender et kald til Klassen **Validering** som skaber to exceptions: **TomEmailException** og **EksisterendeBrugerException** som bliver instantiseret hvis **Validering** får en tom email eller en eksisterende bruger tilbage fra **DatabaseManager**. **Validering** sender kaldet videre til **DatabaseManager**, som så sender `hentBrugerMedEmail(email : String)` tilbage igennem **Validering** hvor den tjekkes for **Exception** og derefter sendes videre til **BrugerFacade** og vises til brugeren.

## Test (Benjamin)

I dette afsnit vil vi vise, hvordan vi lever op til læringsmålet fra **systemudvikling**:

- Den studerende kan anvende erhvervets teknikker og redskaber til planlægning og afvikling af test og kvalitetssikring

og læringsmålet fra **programmering**:

- Den studerende kan anvende moderne teknikker og værktøjer til afvikling af test og kvalitetssikring samt udfærdige dokumentation i forhold til gældende standarder i professionen

### Code coverage

Vi valgte at bruge JUnit5 til test, fordi det er den nyeste version af JUnit. Vores mål med test var at teste alle use cases. Det giver ikke 100% code coverage, men vi følte, at et bedre mål var at leve op til kundekravene i så høj grad som muligt, så derfor valgte vi at prioritere tiden med fokus på dette.

Element	Class, %	Method, %	Line, %
exceptions	92% (13/14)	100% (0/0)	100% (13/13)
Besked	100% (1/1)	50% (5/10)	55% (11/20)
BeskedFacade	100% (1/1)	90% (9/10)	91% (21/23)
BeskedManager	100% (1/1)	88% (8/9)	90% (29/32)
Bruger	100% (1/1)	57% (8/14)	59% (16/27)
BrugerFacade	100% (1/1)	55% (10/18)	60% (20/33)
BrugerManager	100% (1/1)	73% (14/19)	69% (60/86)
Chat	100% (2/2)	58% (10/17)	61% (22/36)
TekstHasher	100% (1/1)	100% (1/1)	80% (8/10)
Validering	100% (1/1)	100% (8/8)	100% (31/31)

Her ses det samlede resultat af vores unit- og systemtest, når vi kører dem med code coverage. Der er nogle ting at bide mærke i, inden vi går i detaljer med testene. Klasserne Besked, Chat og Bruger er entities og har derfor en masse setter/getter metoder, som ikke testes. Det giver lavere code coverage end de andre klasser. BrugerFacade har forholdsvis lav code coverage, fordi den er en controller, som ikke rigtig gør andet end at sende metodekald videre til BrugerManager og Validering. Den indeholder bl.a. en masse tjek-metoder, som sendes videre til Validering, og disse testes ikke i systemtest, fordi de er meget simple og ikke bruger andre klasser end BrugerFacade og Validering. BeskedFacade har modsat meget høj code coverage, fordi alle metoderne testes i systemtest. Det gode ved vores testresultater er, at vi i det mindste tester alle klasser i vores program.

## Unittest

Vi bruger unittest til at teste de enkelte metoder i programmet. Vi laver dem ved at mocke de klasser, som metoden, der bliver testet, arbejder sammen med. På den måde isolerer vi den enkelte metode, så vi er sikre på, at vi kun tester den.

Da vi planlagde unittest, startede vi i mange tilfælde med at lave ækvivalensklasser for at sikre os, at vi testede alle vigtige edge cases. Følgende er et eksempel på ækvivalensklasser for test af `tjekPassword(String password)`-metoden i `Validering`-klassen:

Ækvivalensklasser for <code>tjekPassword(String password)</code>				
Class	Dimension	Definition	Validity	Boundary Values
EC01	password er null	<code>password = null</code>	invalid	<code>password: {null}</code>
EC02	password er tom	<code>password.length = 0</code>	invalid	<code>password.length: {0}</code>
EC03	password er for kort	<code>password.length &lt; 6</code>	invalid	<code>password.length: {5}</code>
EC04	password er for langt	<code>5 &lt; password.length &gt; 20</code>	invalid	<code>password.length: {21}</code>
EC05	password er gyldigt	<code>5 &lt; password.length &lt; 20</code>	valid	<code>password.length: {7}</code>

Som vi kan se, er det eneste gyldige password en String med en længde på 6-20 tegn. Derudover bliver grænseværdierne 5 og 21 testet, og for god ordens skyld er der også test til situationer, hvor passwordet er null eller tomt. Ækvivalensklasser giver ikke altid mening, da nogle test kun har to udfald (fx null og not-null) og derfor ingen grænseværdier. Efter ækvivalensklasser lavede vi en beskrivelse af testene:

Unittest for <code>tjekPassword(String password)</code>					
Id	ECs		Input		Forventet Output
UT010301	[EC01]		password	null	<code>NullPointerException</code>
UT010302	[EC02]		password	""	<code>TomPasswordException</code>
UT010303	[EC03]		password	12345	<code>PasswordLaengdeException</code>
UT010304	[EC04]		password	123456789123456789123	<code>PasswordLaengdeException</code>
UT010305	[EC05]		password	1234567	Ingen exception

Testene beskrives ud fra ækvivalensklasserne med input og forventet output. Fx forventer vi en `PasswordLaengdeException`, hvis man forsøger at inputte et password med en længde, der ikke er 6-20 tegn. Her ses UT010303 implementeret:

```
76      @Test
77      ▶ public void tjekPasswordUT010303() {
78          Validering validering = new Validering();
79          String password = "12345";
80          assertThrows>PasswordLaengdeException.class, () -> validering.tjekPassword(password));
81      }
```

Selve testen foretages i linje 80, hvor `assertThrows`-metoden kaldes. Vi forventer en `PasswordLaengdeException` og det faktiske output er `validering.tjekPassword(password)`. Vi kan se at testen består, når vi kører vores testsuite:



✓ UTD01 (model)	51 ms
✓ tjekPasswordUT010301	13 ms
✓ tjekPasswordUT010302	0 ms
✓ tjekPasswordUT010303	1 ms
✓ tjekPasswordUT010304	1 ms
✓ tjekPasswordUT010305	1 ms
✓ tjekEmailUT010101	2 ms
✓ tjekEmailUT010102	1 ms
✓ tjekEmailUT010103	2 ms
✓ tjekEmailUT010104	0 ms
✓ opretBrugerUT010401	3 ms
✓ opretBrugerUT010402	25 ms
✓ tjekNavnUT010201	1 ms
✓ tjekNavnUT010202	1 ms
✓ tjekNavnUT010203	0 ms

Der er ingen mockobjekter i unittesten af tjekPassword, fordi metoden kun foregår inden i Validering-klassen. Jeg har valgt dette eksempel, fordi det har gode ækvivalensklasser. Lad os derfor se på et andet eksempel, hvor vi bruger mockobjekter:

```

30 public void opretChat(String navn, String emne) throws BrugerFindesIkkeException {
31     Bruger afsender = brugerManager.getAktivBruger();
32     Bruger modtager = brugerManager.hentBrugerMedNavn(navn);
33     long sidstAktiv = System.currentTimeMillis();
34     if (modtager == null)
35         throw new BrugerFindesIkkeException();
36     Chat nyChat = new Chat(afsender.getNavn(), modtager.getNavn(), emne, sidstAktiv);
37     chats.add(nyChat);
38 }

```

Her ses metoden opretChat(String navn, String emne) i BeskedManager-klassen. opretChat er afhængig af en anden klasse, BrugerManager, for at kunne bestemme afsender (linje 31) og modtager (linje 32). Da vi skal isolere opretChat i unittesten, er det altså nødvendigt at mocke BrugerManager-klassen. Vi løser problemet ved at lave en TestbarBeskedManager:

```

139     private class TestbarBeskedManager extends BeskedManager{
140
141         public TestbarBeskedManager(){
142             setChats(new ArrayList<>());
143         }
144
145         @Override
146         protected BrugerManager newBrugerManager() {
147             return new MockBrugerManager();
148         }
149     }

```

TestbarBeskedManager extender BeskedManager (linje 139). BeskedManager har metoden newBrugerManager(), som vi overrider til at returnere en MockBrugerManager i stedet (linje 145-148). Lad os se på MockBrugerManager:

```

69     private class MockBrugerManager extends BrugerManager {
70         @Override
71         public Bruger getAktivBruger(){
72             return new MockBruger( navn: "Hans");
73         }
74
75         @Override
76         public Bruger hentBrugerMedNavn(String navn) {
77             if (navn == null) {
78                 return null;
79             }
80
81             if (!navn.equals("Boris")) {
82                 return null;
83             }
84
85             return new MockBruger(navn);
86         }
87     }

```

I mock-klassen overrider vi de metoder, som opretChat kalder, så vi får et output, der kan bruges til test. Vi kan se, at getAktivBruger() altid returnerer "Hans" (line 70-73) og hentBrugerMedNavn kun kan returnere null eller "Boris" (linje 75-83). Nu kan testen af opretChat laves:

```

25      @Test
26      public void opretChatUT030301() throws BrugerFindesIkkeException {
27          BeskedManager beskedManager = new TestbarBeskedManager();
28          beskedManager.opretChat( navn: "Boris", emne: "Skulderskade");
29          String output = beskedManager.hentChats().get(0).getModtager();
30          assertEquals( expected: "Boris", output);
31      }

```

TestbarBeskedManager sørger som sagt for et output, som vi kan forudse. Dvs. at det eneste vi nu tester er, om en Chat bliver oprettet. Vi forventer, at den Chat har modtageren "Boris", og vores faktiske output er beskedManager.hentChats().get(0).getModtager() (linje 38). Testen består, så vi ved, at forventet output og faktisk output er det samme:

▼ ✓ BeskedManagerTest (unittests.usecases)	11 ms
✓ opretChatUT030301	11 ms

## Systemtest

Vi bruger systemtest til at teste, hvordan klasserne i programmet arbejder sammen, dvs. i disse test er det den reelle kode, der bliver testet, og der bruges ingen mocks. Dog er der en enkelt undtagelse til denne regel, og det er DatabaseManager.java. Der er to problemer ift. at teste med DatabaseManager. For det første, så anvender vi et singleton pattern, fordi Firebase ikke tillader, at metoden initializeDB kaldes mere end en gang; der må altså kun være én forbindelse til Firebase. For det andet, så udgør det et væsentligt problem at teste metoder som fx sletBruger, hvis det medvirker, at testen rent faktisk sletter en bruger fra vores database. Derfor har vi valgt at mocke DatabaseManager i systemtest, så vi kan sørge for, at test ikke påvirker driften af programmet.

```

110 private class MockDatabaseManager {
111     public ObserverbarListe<Chat> hentChats() {
112         ObserverbarListe<Chat> chats = new ObserverbarListe<>();
113         Chat chat = new Chat( afsender: "Karsten Wiren", modtager: "Christian Iuul",
114                               emne: "Hold i nakken", System.currentTimeMillis());
115         chats.add(chat);
116         return chats;
117     }
118
119     public ObserverbarListe<Bruger> hentBrugere() {
120
121         TekstHasher tekstHasher = new TekstHasher();
122         String password = tekstHasher.hashTekst("testpw");
123
124         ObserverbarListe<Bruger> brugere = new ObserverbarListe<>();
125         Bruger behandler1 = new Bruger( navn: "Christian Iuul", email: "fys@frbsport.dk",
126                                         password, erBehandler: true);
127         Bruger patient1 = new Bruger( navn: "Camilla Kron", email: "camillak@gmail.com",
128                                       password, erBehandler: false);
129         Bruger patient2 = new Bruger( navn: "Karsten Wiren", email: "karstenw@gmail.com",
130                                       password, erBehandler: false);
131         brugere.add(behandler1);
132         brugere.add(patient1);
133         brugere.add(patient2);
134         return brugere;
135     }
136 }

```

Her ses MockDatabaseManager, som er oprettet som en private class i STD0301.java. I linje 111 laves en metode, hentChats, som skal efterligne metoden hentChatsMedNavn fra DatabaseManager, og i linje 119 laves en metode, hentBrugere, som skal efterligne metoden hentBrugere fra DatabaseManager. Den væsentligste forskel er, at hvor den rigtige DatabaseManager-klasse skal skabe forbindelse til Firebase, for at fylde listerne, så fylder MockDatabaseManager sine lister med data, som den selv indeholder. Nu har vi altså en fungerende MockDatabaseManager, som fylder vores lister med data, uden at forstyrre driften af programmet, og vi kan bruge disse til at udføre vores systemtest. Lad os se på et eksempel:

Systemtest til opretChat(String navn, String emne)			
id	Input		Forventet output
opretChatST030301	navn	Karsten Wiren	En chat er oprettet
	emne	Ondt i ryggen	
opretChatST030302	navn	Ejnar Gunnarsen	BrugerFindesIkkeException
	emne	Dårligt knæ	
opretChatST030303	navn	Karsten Wiren	TomEmneException
	emne	""	
opretChatST030304	navn	Karsten Wiren	ForMangeTegnException
	emne	testtesttesttesttesttesttesttesttesttesttesttesttesttesttest	

Her er systemtest til opretChat beskrevet med input og forventet output. Jeg vil gå i dybden med opretChatST030301, men lad os først se på metoden, som den ser ud i BeskedManager.java:

```
30 public void opretChat(String navn, String emne) throws BrugerFindesIkkeException {
31     Bruger afsender = brugerManager.getAktivBruger();
32     Bruger modtager = brugerManager.hentBrugerMedNavn(navn);
33     long sidstAktiv = System.currentTimeMillis();
34     if (modtager == null)
35         throw new BrugerFindesIkkeException();
36     Chat nyChat = new Chat(afsender.getNavn(), modtager.getNavn(), emne, sidstAktiv);
37     chats.add(nyChat);
38 }
```

Vi kan se, at opretChat tager imod to parametre, navn og emne (linje 30), som var beskrevet i testen. Den ender med at lave et nyt Chat-objekt, nyChat (linje 36), og tilføjer det til en liste af Chat-objekter, chats (linje 37), som BeskedManager indeholder. Chatten tager også nogle parametre, som først skal defineres. afsender og modtager defineres vha. metoderne getAktivBruger (linje 31) og hentBrugerMedNavn (linje 32) fra BrugerManager.java. Derudover skal vi bruge deres navne som String-variable, så vi bruger Bruger-objektets getter-metoder til at få disse i constructoren til nyChat (linje 36). sidstAktiv defineres vha. klassen System (linje 33) og til sidst genbruges det parameter, som opretChat fik med, da den blev kaldt, som emne i constructoren til nyChat (linje 36). Vi teste opretChat således:

```
14 @Test
15 public void opretChatST030301() throws BrugerFindesIkkeException, ForkertPasswordException,
16     TomEmneException, ForMangeTegnException {
17     MockDatabaseManager mockDatabaseManager = new MockDatabaseManager();
18     BeskedFacade beskedFacade = BeskedFacade.getInstance();
19     beskedFacade.setChats(mockDatabaseManager.hentChats());
20
21     BrugerFacade brugerFacade = BrugerFacade.getInstance();
22     brugerFacade.setBrugere(mockDatabaseManager.hentBrugere());
23     brugerFacade.logInd( email: "fys@frbsport.dk", password: "testpw");
24
25     beskedFacade.opretChat( navn: "Karsten Wiren", emne: "Ondt i ryggen");
26     String output = beskedFacade.hentChats().get(beskedFacade.hentChats().size() - 1).getEmne();
27     assertEquals( expected: "Ondt i ryggen", output);
28 }
```

Her tester vi oprettelse af en Chat med afsender "Christian luul", modtager "Karsten Wiren" og emne "Ondt i ryggen". Det starter med at der instantieres en ny MockDatabaseManager (linje 17) og en BeskedFacade ved at kalde BeskedFacade.getInstance() (linje 18). Derefter sættes listen af chats i BeskedFacade til den, vi får fra MockDatabaseManager (linje 19). Vi skal også bruge BrugerManagers liste af brugere, så derfor instantierer vi BrugerFacade som

BrugerFacade.getInstance (linje 21) og sætter derefter listen til den, vi fik fra MockDatabaseManager (linje 22). Derefter kalder vi logInd-metoden (linje 23), og den vil vi se nærmere på om lidt. Vi er nu klar til at kalde opretChat på BeskedFacade med parametrene modtager = "Karsten Wiren" og emne = "Ondt i ryggen", som de var beskrevet i testen (linje 25). Pointen med testen er at teste, om chatten rent faktisk blev oprettet, så derfor kalder vi assertEquals på den (linje 27). Vi forventer, at emnet, den er oprettet med, er "Ondt i ryggen", og det faktiske emne findes i linje 26 ved først at hente listen med Chats i BeskedManager og derefter hente den første Chat i listen, da vi ved, at listen var tom, da vi fik den fra MockDatabaseManager.

▼ ✓ STD0301 (model)	21 ms
✓ opretChatSTD0301	21 ms

Testen består, så det forventede output var altså lig med det faktisk output. Lad os lige vende tilbage til logInd-metoden, for at se, hvorfor vi kaldte den i testen:

```

88     public boolean logInd(String email, String password) throws ForkertPasswordException {
89         for (int i = 0; i < brugere.size(); i++) {
90             String hashedPassword = tekstHasher.hashTekst(password);
91             if (brugere.get(i).getEmail().equals(email)) {
92                 if (brugere.get(i).getPassword().equals(hashedPassword)){
93                     aktivBruger = brugere.get(i);
94                     return true;
95                 }
96                 else
97                     throw new ForkertPasswordException();
98             }
99         }
100         return false;
101     }

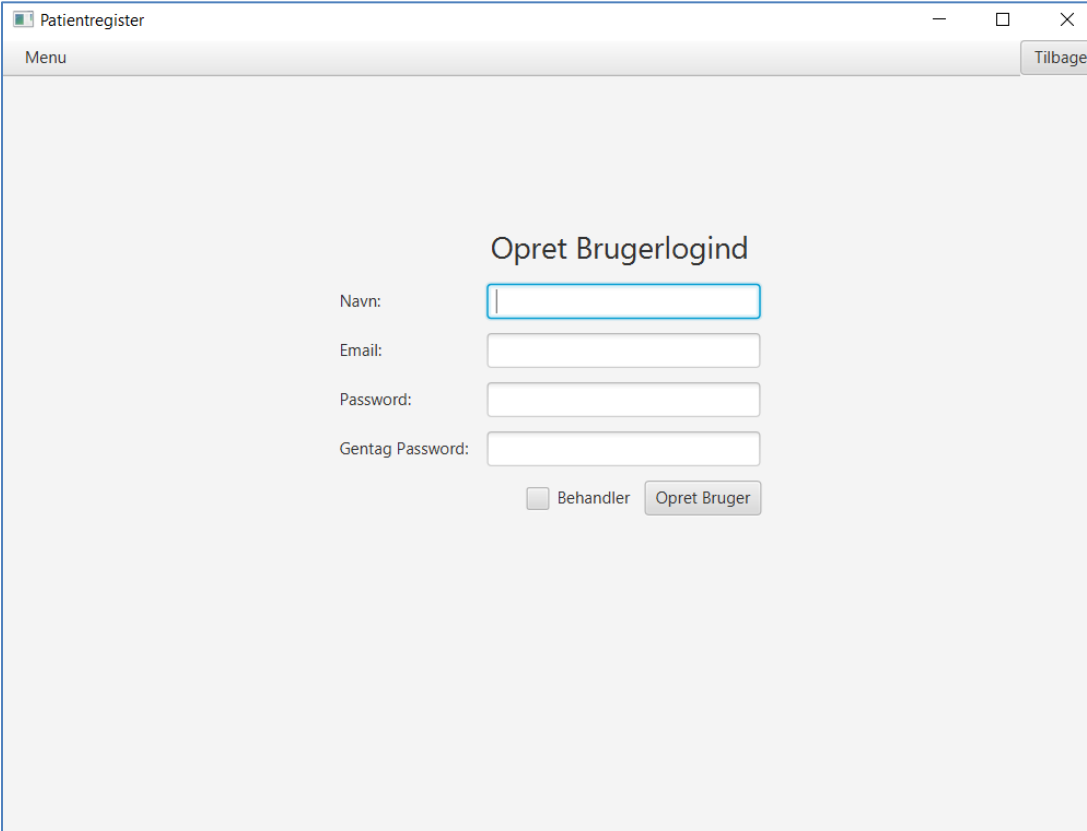
```

afsender til Chat-objektet så vi tidligere, er bestemt ved at hente aktivBruger fra BrugerManager. Derfor var vi i testen nødt til at kalde logInd-metoden. Det interessante her er, at instansvariabelt aktivBruger i BrugerManager bliver sat den bruger, som passer til de parametre, vi giver metoden (linje 93).

## Analyse af Brugertest (Kelvin)

Brugertesten har til formål at få feedback fra kunden, som vi skal bruge til at specificere kundekrav og udvikle en brugervenlig brugerflade. Testene udformer sig i at lade kunden manøvrere rundt i programmet, hvor han ud fra et fiktivt scenarie, skal løse en række opgaver, med mindst mulig vejledning.

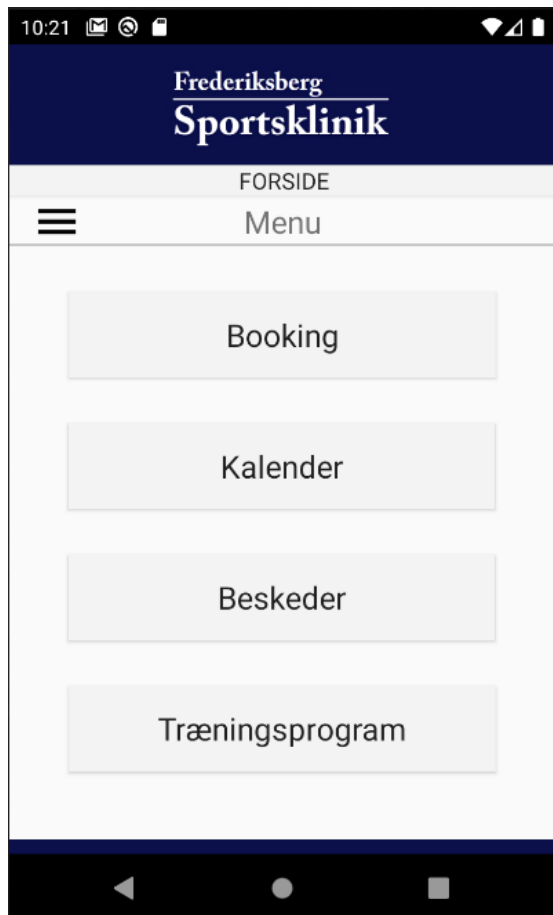
Det feedback som vi fik fra vores kundens brugertest er brugbart for hvordan vores program kommer til at udvikle sig designmæssigt. Design er et vigtigt komponent for at guide folk rundt i appen, fordi hvis en app er designet dårligt, så er det lige meget hvor god appen er når den får et dårligt ry på grund af dets design. Det er et vigtigt punkt i gestalt psykologi at man forstår vigtigheden af de æstetisk kvaliteter i forhold til personers opfattelse og tankegang. Det kan ses i Gestaltlovene, som er de love der forklare de psykologiske aspekter af visuel design. Et eksempel er Law of Symmetry, som siger at objekter skal være balanceret eller symmetriske for at blive set som et helt program. Lovene hjælper os med at designe vores program baseret på vores brugertests, da de fortæller os hvad der giver mening i vores program, et eksempel kunne være desktop-apps testen for vores kunde:



The screenshot shows a web browser window with the title 'Patientregister'. The browser's address bar shows 'Menu' and a 'Tilbage' button. The main content area has a light gray background. In the center, there is a form titled 'Opret Brugerlogind'. The form contains four input fields: 'Navn:', 'Email:', 'Password:', and 'Gentag Password:'. Below the 'Gentag Password:' field, there is a checkbox labeled 'Behandler' and a button labeled 'Opret Bruger'.

I eksemplet kan man se opret bruger. I vores første test skulle Kunden finde ud hvordan han oprettede en bruger til patient Anders. Kunden havde ingen problemer med at finde rundt i app'en og synes det var intuitivt indtil han kom til opret bruger, da han skulle lave en bruger til patienten så klikkede han på behandler boxen og oprettede brugeren. Behandler boxen markere den nye bruger som en behandler i stedet for patient. Brugertesten viste os at kunden ikke havde den samme forstand

på teknik som vi har. Det kan være på grund af Gestaltloven Law of Focal Point, som siger at en bruger vil fokusere på det der står ud, at Christian hakkede behandler boxen. Vi valgte ikke at fjerne knappen fordi at programmet er afhængig af at man kan lave brugere til både patienter og behandlere. Et andet eksempel kunne være bruger testen for android-appen på en midaldrende dame:



Det andet eksempel som bliver vist er en brugertest til android app'en. testpersonen er en midaldrende dame som ikke er vant til teknologi. Personen fik at vide at de ville gerne i kontakt med behandleren. Testperson havde problemer igennem hele processen, det var mest tydeligt da personen først gik ind på kalenderen for at finde en tid, så ind på booking for at booke en tid og til sidst ind på beskeder og fandt beskeden de skulle skrive tilbage på. Fejlen i denne brugertest var spørgsmålet selv, da det var alt for generelt (testscenarie 3, opgave 2). I dette eksempel kan vi se at brugerens mentale modeller er formet af deres manglende erfaring med teknologi og hvordan at der er en fejl i Domain level i de mentale modeller, som er opgave og data niveauet i et applications domæne, specifikt Task Description. Opgavebeskrivelsen var, som nævnt tidligere, for generelt og derfor skabte det



forvirrelse hos testeren, men selvom testen fejlede var der brugbar resultater som hjælp med kvalitetssikring for vores projekt.

## Programmering

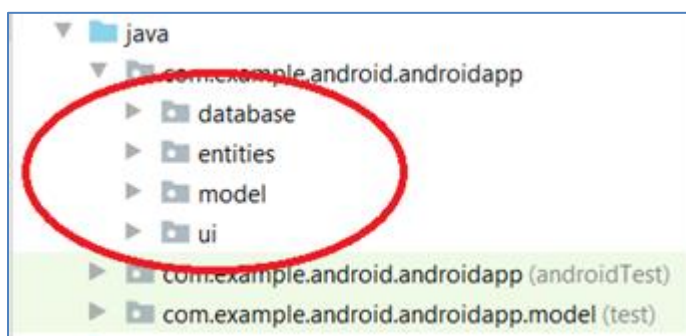
### Designmønstre (Tommy)

#### Clean architecture med observer pattern

I vores projekt bruger vi Clean Architecture såfremt det er muligt. Til at starte med var model og database koblet sammen på den måde at databasen kun kom i brug når der skete noget i modelklassen men denne kobling bragte problemer i forhold til test, samt at modelklassernes opbygning blev drejet til at passe databasen hvilket gjorde programmet mindre fleksibelt. Løsningen på dette var som sagt at bruge clean architecture og med brugen af denne opfyldes til dels dette læringsmål:

- anvende centrale faciliteter i programmeringssproget til realisering af algoritmer, designmønstre, abstrakte datatyper, datastrukturer, designmodeller og brugergrænseflader

Til at starte med ser vores pakke opdeling således ud:



En ting der er værd at nævne allerede nu er at vi ikke har presenters til UI og Use cases. Dette er et bevidst valg da vores brug af Firebase ikke faciliterede en nem brug af disse. I stedet må UI klasserne kalde på use cases selv og så formatere om end det er nødvendigt, hvilket ikke er hensigtsmæssigt eftersom UI'et får for mange opgaver. Alligevel blev dette den mest praktiske løsning med hensyn til tid brugt og arbejde krævet.

Det der faciliterer at vi kan afkoble database og model er observer pattern også kendt som listener. I vores programmer bruger vi PropertyChangeSupport sammen

med `PropertyChangeListener` hvilket er klasser fra Java biblioteket der var beregnet til at erstatte de deprecated `Observer` og `Observable` klasser og interfaces. Vi kunne have gjort det samme med de deprecated klasser men mente at det var bedre at sætte sig ind i den nye måde at bruge observer pattern som Java har implementeret.

```
13 class BrugerManager {
14     // aktiveBruger er det bruger objekt som er loggede ind.
15     private Bruger aktivBruger;
16     private ArrayList<Bruger> brugere;
17     private PropertyChangeSupport support;
18
19     BrugerManager() {
20         support = new PropertyChangeSupport( sourceBean: this);
21         brugere = new ArrayList<>();
22     }
```

I stedet for at extend `Observer` i den klasse der skal observeres så skal den observeret klasse instantiere en `PropertyChangeSupport` med sig selv som parameter. Med denne får man metoder til at tilføje og fjerne observers/listeners. Man sender observerkald med `firePropertyChange`:

```
53 // Observerkald hvor det nye bruger objekt passerer som argument.
54 support.firePropertyChange( propertyName: "opretBruger", oldValue: null, bruger);
```

Her har vi oprettet et nyt bruger objekt og derefter kalder vi `firePropertyChange`, hvor vi navngiver ændringen og sender det nye bruger objekt tilføjet til listen med som det objekt observerne kan referere til.

```
44 brugerFacade.tilfoejListener(new PropertyChangeListener() {
45     @Override
46     public void propertyChange(PropertyChangeEvent evt) {
47         if (evt.getPropertyName().equals("opretBruger")) {
48             DatabaseManager databaseManager = new DatabaseManager();
49             Bruger bruger = (Bruger) evt.getNewValue();
50             databaseManager.gemBruger(bruger);
51             progressDialog.dismiss();
52             finish();
53             startActivity(new Intent(getApplicationContext(), MenuActivity.class));
54         }
55     }
56 });
```

Som kan ses her så kan databasen observere på brugerFacaden (Faktisk brugerManager men via brugerFacaden) og derpå vide når den skal opdater. Med dette er modelklasserne og databasen afkoblet og dette tillader os at f.eks. ændre databasen uden at behøve at ændre på koden i modelklasserne.

Tanker til Clean Architecture:

Vores brug af Clean Architecture har været gavnligt for projektet på den måde at programmet bliver mere fleksibelt og afkoblet som er gode ikke-funktionelle krav at varetage, men det kræver til gengæld lidt mere ekspertise at implementere. Det er derfor godt at vi formået at inddrage det i projektet da vi nu har et godt designmønstre at vise frem, men alligevel, med hensyn på arbejdet krævet, så kan vi også konkluderer at Clean Architecture ikke er det rette fit i alle programmer og fremadrettet i andre projekter vil vi altid vurdere om det er passende at bruge dette designmønster fremfor et der kunne være simplere at implementere såsom GRASPs opdeling med model, view, persistence og domain.

## Singleton pattern

I programmets udformning har vi vurderet det passende at bruge singleton pattern. Selvom det ikke er pensum, så er det alligevel værd at nævne det eftersom det er et designmønster og at det er et der brugt i vores program. Til at starte med, så er vi klar over den kritik der eksisterer med singleton hvor nogen referer det som værende et anti-pattern ( <https://www.michaelsafyan.com/tech/design/patterns/singleton> ). Stadig har vi fundet det passende da det giver os en nem måde at vedvare data når vi f.eks. skifter activity i Android eller skifter scene i JavaFX.

Et eksempel vises her hvor vi gør brug af det i BrugerFacade klassen.

```

17 public class BrugerFacade {
18     private BrugerManager brugerManager;
19     private Validering validering;
20     private static BrugerFacade brugerFacade;
21
22     private BrugerFacade() {
23         brugerManager = new BrugerManager();
24         validering = new Validering(brugerManager);
25     }
26
27     public static synchronized BrugerFacade hentInstans() {
28         if (brugerFacade == null) {
29             brugerFacade = new BrugerFacade();
30         }
31         return brugerFacade;
32     }

```

Med et facade designmønstre kan vi nøjes med lave én klasse til singleton, i stedet for f.eks. at de viste klasse BrugerManager og Validering på henholdsvis linje 18 og 19 også behøver at være det, og derpå kan vi formindske klasser som er singleton. Dette er hensigtsmæssigt da et af problemerne med singleton pattern er at det er svært at teste. Grunden til at BrugerFacade, som er den udvendige interface UI'et skal interagere med om alt omhandlende brugersystemet i vores program, er singleton er netop fordi denne data om brugere som den passerer omkring sig skal være persisterende i programmet selvom man har skiftede scene i UI'et. I dette format, når den ene instans af BrugerFacade er instantieret og dennes instans af BrugerManager har fået fyldt sine liste af brugere, så vil, hver gang UI'et tilgår BrugerFacade, altid få samme liste af brugere, og derpå slipper vi f.eks. for at hente listen af brugere ned fra databasen hver gang vi skifter scene i UI'et. Selvfølgelig er det værd at nævne at singleton pattern ikke er den eneste løsning på denne problemstilling men kun er i brug fordi de bivirkninger den fører med sig vurderer vi til at være acceptable i modsætning til de praktiske egenskaber som den giver os. Alligevel, fremadrettet, så kunne vi i stedet for singleton pattern kun lave én instans uden at det er singleton pattern og så passere referencen til denne instans med når vi navigerer rundt i vores program. Dette vil give mere arbejde i at programmere et scenskift men til gengæld slipper man for alle de problemer som singleton fører med sig som f.eks. problemer med unit tests og at den låser programmets design fast i at værende singleton selvom programmet i fremtiden ikke behøver at være det.

## Facade pattern

Som vist i det forrige afsnit så bruger vi Facade designmønster og dette er også noget der er værd at nævne. Vi har i vores program tre klasser der fungerer som facade: BrugerFacade, BeskedFacade, og TraeningsprogramFacade. Som navnene hentyder til arbejder de henholdsvis med brugersystemet, chatsystemet og visning af træningsvideoer fra ExorGO-live. Grunden til vi bruger facade er at det hjælper til at indkapsle programmet og øge maintainability, en af de ikke-funktionelle krav som indgår i FURPS+. Ved at der kun er et access point til funktioner så isolerer vi de steder hvor forskellige systemer må koble sig til hinanden. Med dette kan vi nemt f.eks. skifte UI'et ud med et andet, da der kun er en klasse som UI'et må koble sig sammen med og alt det der sker bagved den behøver den slet ikke have kendskab til. Vi får samtidig en klar oversigt over de funktioner der skal stilles til rådighed frem for dem der bare kan ske bag ved facaden og dette er en anden ting der giver koden mere maintainability eftersom det bliver mere overskueligt for en selv eller andre om hvad kode de skal arbejde med hvis de vil koble ting sammen.

## Persistens (Benjamin)

I dette afsnit vil jeg vise, hvordan vi lever op til fagets læringsmål:

- anvende centrale metoder og teknikker til at realisere modeller i et databasesystem og konstruere programmer, der benytter en databasegrænseflade

Persistens er det, der gemmer data i programmet, så det kan blive hentet frem på et andet tidspunkt, eller af et andet program, og blive brugt igen. et program uden persistens har meget begrænset anvendelse, da den data, der bliver brugt i et sådant program, forsvinder igen umiddelbart efter afslutning af en session; altså når programmet lukkes. Persistens er en nødvendighed i dette projekt, da et af kundekravene er, at der skal sendes en besked fra et program til et andet. Beskeden er i denne situation data, som skal kunne gemmes og tages frem igen. Her bruges en database, som tilgås gennem et database management system (DBMS)(Liang 2019).

## MySQL vs. SQL

Data kan persisteres på forskellig vis, men er typisk standard datatyper som String, integer og double. Der findes grundlæggende to typer databaser; relationelle- og ikke-relationelle databaser. I en relationel database, er dataene koblet opstillet i tabeller, som har kolonner og rækker, og er derudover koblet sammen med en nøgle, der afgør, hvordan de er relateret til hinanden. Ikke-relationelle databaser derimod, gør ikke brug af denne struktur. Strukturen af disse kan variere, men er ofte

optimeret til den type data, de skal indeholde. MySQL er det mest udbredte relationel database management system (RDBMS), som bruger Structured Query Language (SQL). SQL er et sprog, som muliggør querying i en relationel database, hvilket betyder at man kan tilgå og sortere i flere tabeller med en enkelt kommando. Eksempler på SQL queries:

- `SELECT * FROM table` (vælger alt fra nuværende tabel)
- `SELECT ... FROM ... WHERE condition` = vælger et specifikt felt fra nuværende tabel, som opfylder en condition
- `DELETE FROM table1 WHERE condition` = sletter et specifikt felt fra nuværende tabel, som opfylder en condition
- `CREATE TABLE table (field1 type1, field2 type2, ...)` = laver en ny tabel med de specificerede felter (<https://devhints.io/mysql>)

Når en database ikke bruger SQL, siger man, at det er en NoSQL database. NoSQL databaser fungerer på forskellige måder alt efter, hvilken database, der er tale om. I nogle NoSQL databaser kan man slet ikke bruge queries.

## Firestore

Vi valgte i vores løsning at bruge Firebase Firestore, som er en document database, som lagrer data som JSON-objekter (Javascript Object Notation). Det er dermed en ikke-relationel NoSQL database. Beslutningen faldt på Firebase af forskellige årsager, bl.a. fordi vi får et færdigt DBMS fuldstændt med database, server og hosting. Det gør det enkelt at implementere. Et af kundekravene var et chatsystem, hvilket betyder at vi med Firebase ikke skulle tænke på hosting, og det gjorde denne løsning meget attraktiv. Hvis vi havde valgt at bruge MySQL, så skulle vi først hoste databasen på en server, før den kunne tilgås af flere apps på samme tid. En database kan også være lokal på den maskine, som kører applikationen, men det var selvfølgelig ikke tilstrækkeligt til denne løsning. Firebase er optimeret til implementering i androidapplikationer, hvilket gav anledning til en forholdsvis problemfri integration i vores system. Der findes masser af dokumentation og det supporter både udvikling i android- og rene javamiljøer. Firestore er NoSQL, men det har sit eget query-system, som ikke kræver, at dataen er opsat i tabeller:

```
89 public Bruger hentBrugerMedNavn(String navn) {  
90     Bruger bruger = null;  
91     Query query = firestore.collection("brugere").whereEqualTo("navn", navn);
```

Her ser vi et eksempel på en query i Firestore (linje 91). Firestore er inddelt i collections, som indeholder documents, men documents kan også indeholde deres egne collections, som igen kan indeholde documents osv. Det er altså muligt at skabe meget lange grene af documents, som indeholder data, og er organiseret i

collections. I linje 91 i eksemplet skrives en query, som skal kigge i den collection, der hedder "brugere", og kun skal hente de documents, der har et field, navn, med det data, som svarer til det navn, metoden får med som parameter. Allerede her har vi altså lavet en ret stor sortering i det data, som ligger gemt i databasen. Andre eksempler på query-kommandoer i Firestore er whereIn og whereGreaterThan, men dem havde vi ikke behov for at bruge i vores program. Firestore kommer dog også med nogle problemer, hvoraf det største, vi har oplevet, er at Google kræver betaling pr. read og write i databasen. Det observer pattern, vi anvender, resulterer desværre i et forholdsvis stort antal reads, så derfor ville vi på sigt skulle finde en måde at optimere dette på, så det ikke bliver alt for dyrt for kunden. Løsningen kunne sagtens ligge i et overgå til et andet DBMS.

## Udviklingsmiljøer (Kelvin)

Vi har gjort brug af forskellige udviklingsmiljøer i forløbet af vores projekt til udvikling og kommunikation. Dette afsnit vil vi forklare hvorfor vi har brugt de udviklingsmiljøer og hvordan vi har brugt dem.

IDE (Integrated Development Environment) er software applikationer som for eksempel IntelliJ, som har en source code editor, build tools og en debugger, hvilket kendetegner en IDE. Det er baseret på disse grunde at vi bruger AndroidStudio og IntelliJ.

Vi har brugt Discord som kommunikationsplatform og ugelige fysiske møder til at facilitere vores udviklings process. I de dage har vi haft dagsplaner til at strukturere vores opgavefordeling. Vores forhold har været stabile på grund af den daglige opgavefordeling, så vi altid vidste hvad der prioriteten for iterationen.

Et andet udviklingsmiljø er Android Studio som vi har brugt til at udvikle mobilappen til vores projekt. Android Studio har hjulpet med vores udviklingsprocess med funktioner som auto-genereret activities og code coverage, som gøre det muligt at streamline vores arbejdsprocess samt kvalitetssikre vores produkt. I både IntelliJ og AndroidStudio er code coverage funktionen brugbar til at se hvor meget af koden er testet. Eksempelvis er der code coverage for UC03:

Element	Class, %	Method, %	Line, %
exceptions	28% (4/14)	100% (0/0)	100% (4/4)
Besked	100% (1/1)	40% (4/10)	45% (9/20)
BeskedFacade	100% (1/1)	90% (9/10)	91% (21/23)
BeskedManager	100% (1/1)	88% (8/9)	90% (29/32)
Bruger	100% (1/1)	35% (5/14)	44% (12/27)
BrugerFacade	100% (1/1)	22% (4/18)	30% (10/33)
BrugerManager	100% (1/1)	36% (7/19)	25% (22/86)
Chat	100% (2/2)	58% (10/17)	61% (22/36)
TekstHasher	100% (1/1)	100% (1/1)	80% (8/10)
Validering	100% (1/1)	62% (5/8)	54% (17/31)

I UC03 ser vi i hvilken grad vores klasser, metoder og kodelinjer der er dækket af code coverage. Grunden til vores code coverage er lavt er fordi vi kun tager udgangspunkt UC03, hvilket har effekt på hvor meget der bliver vist. Code coverage for de forskellige elementer er vigtig i forhold til hvad man vil teste. I eksemplet vil man kun se at alle Classes er dækket, men det svinger i Line coverage med linje dækning.

Vi lavede unittest og systemtest ved at oprette en test mappe hvor vi så lavede testklasserne. Formålet med vores tests er vi viser vores program virker, og testene gør det muligt for os at fange fejl som vi ville ellers misse. I det eksempel som vi viser lavede vi tests til tilknytBehandlerUT050103:

```

77  @Test
78  public void tilknytBehandlerUT050103() {
79      String test123 = "";
80      MockBruger patient = new MockBruger( navn: "Warwick Davis", test123, erBehandler: false);
81      MockBruger behandler = new MockBruger( navn: "Charles Manson", test123, erBehandler: true);
82      patient.setBehandlere(null);
83      BrugerManager brugerManager = new TestbarBrugerManager();
84      assertThrows(NullPointerException.class, () -> brugerManager.tilknytBehandler(patient, behandler));
85  }

```

I denne test viser vi hvordan vi fanger en NullPointerException, som kan ses ved linjen patient.setBehandlere(null); og hvor vi så viser hvordan vi fanger den med en assertThrows. Der kan læses mere om unittests og systemtests i kapitel tests

IntelliJ har vi også brugt i vores projekt til udvikling af vores desktop app designet til behandlerne. AndroidStudio og IntelliJ har ens på mange områder og det gør det muligt for dem at snakke sammen, for eksempel er de begge lavet af de samme firma og de er begge IDEer.

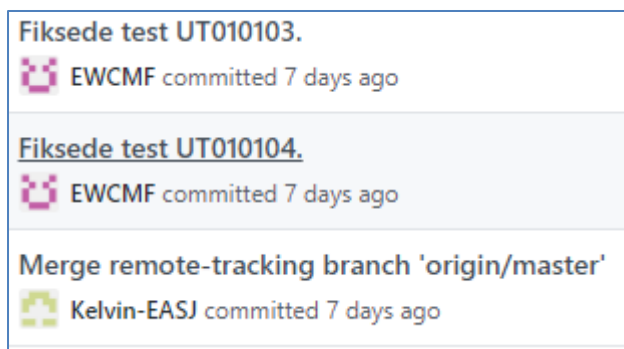


## Github (Kelvin)

Vi bruger Git til at vise projekt opdateringer og skabe struktur i projektet.

Versionsstyring har vi gjort brug af i vores projekt. Grunden til at vi har gjort det er fordi det gør det muligt for hele gruppen at holde deres version af projektet opdateret og sørger for at merge conflicts blive fikset før det kan blive uploadet til master branchen. Det giver os også mulighed for at hoppe frem og tilbage mellem forskellige versioner og giver gruppemedlemmer muligheden for at arbejde på samme tid. Github fungerer også som en backup af vores projekt i tilfældet af en computer uploader et commit og derefter går død. Det gør Github attraktiv for vores gruppe, grundet af dets kvalitetssikring og kontrol.

Committing og pushing er vigtig for vores projekt, da der gør det muligt som tidligere nævnt at kvalitetskontrollere projektet. Når man committer i projektet gør det det også muligt at se projektprocessen og revert hvis der er sket en fejl i opdateringen. Et eksempel på et dårligt commit kunne være:



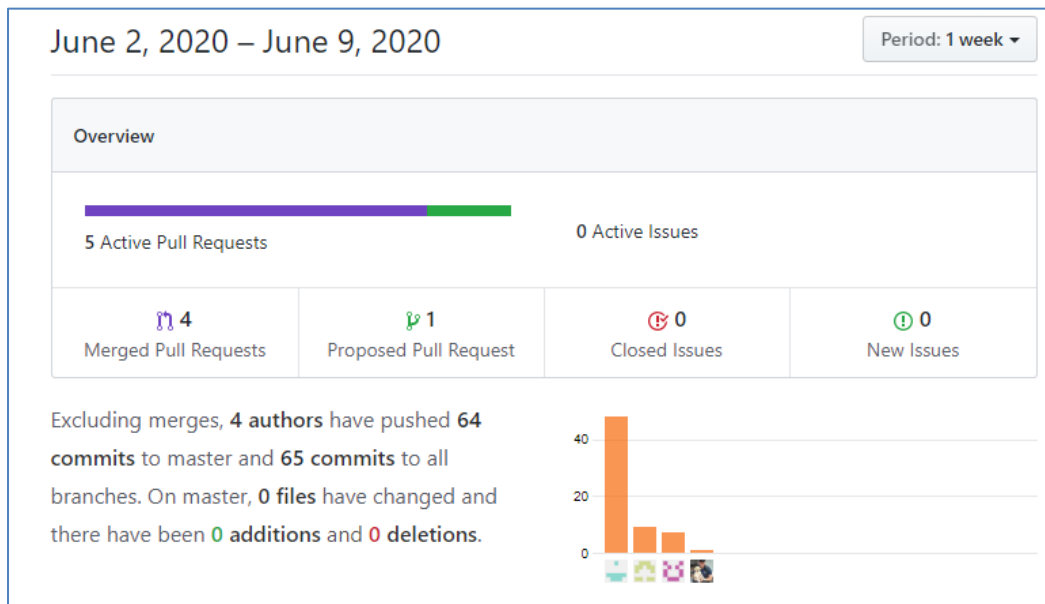
En af vores gruppemedlemmer ødelagde programmet med et commit som han pushede direkte i master branchen, i stedet for at lave en separate branch. På grund af revert funktionen sparede vi den tid vi ville have brugt på at fixe programmet.

Et godt eksempel på en commit ville være et andet gruppemedlems commit om sletPatient:



I billedet ser man hvordan at gruppemedlemmet helt specifikt skriver hvad commitet handler om og hvad den gør. Dette giver med det samme et overblik over hvad der er sket.

Insight viser hvordan at udviklingen af projektet har foregået over dets levetid. Det fortæller også om hvor mange commits og merges alle gruppemedlemmer har og den individuelle persons arbejde:



På dette billede kan man se hvor mange pull requests der er blevet merged, hvor mange issues der er kørende eller er blevet løst og arbejdsindsatsen for gruppen. Der er et problem ved insight i Github og det er at den viser hvor ofte et gruppemedlem hard committed noget, i stedet for hvor faktisk det commit var. Det skaber det forkerte indtryk af at dem som ikke har mange commits ikke laver noget, hvilket kan i nogle tilfælde være forkert hvis en person har eksempelvis lavet et stort commit med mange ændringer, hvor en anden har rettet et par stavefejl over en del commits. Det giver dog stadig et overblik over arbejdsprocessen for gruppen og hvad de har beskæftiget sig med over dagene og ugerne.

Et eksempel på hvad vi har brugt Git kan være vores brug af branches:

kobling-af-ui-til-uc05	Updated 2 hours ago by BenjaminKyhn	3   0	#27	Merged
Implementering-af-systemtests	Updated 2 days ago by BenjaminKyhn	18   0	#25	Merged
Implementering-af-unit-tests	Updated 2 days ago by BenjaminKyhn	79   2	#24	Open
Clean-architecture-i-desktop	Updated 7 days ago by BenjaminKyhn	171   0	#21	Merged

Som det kan ses i billedet, bruger vi branches som implementering-af-systemtest til kvalitetssikring, hvilket vi har gjort ved at sætte den i en separat branch, hvor vi så har implementeret vores systemtest og derefter så har merged det ind i vores projekt efter at vi havde sikret det virkede. Issues er en anden funktion i git som vi kunne have brugt, men valgte ikke at bruge, da vi har både fysiske muligheder for at møde og sociale programmer som discord som erstatter Issues.

I vores gruppe bruger vi både Discord, som er den sociale platform vi har brugt igennem hele projektet til at kommunikere og Github i tandem med hinanden. Vi bruger git til at sende dokumenter og modeller til hinanden så vi kan bruge dem til at strukturere vores kode og demonstrere virksomhedsopstart. Discord gør det også muligt at opdatere hinanden om ændringer eller spørg om reviews til dokumentationen eller kode.

## Gradle (Kelvin)

Gradle er et build tool, som er et program der automatiserer processen for at skabe et executable program. Gradle konfigurerer de dependencies samt den version af SDK vores program kræver for at kan køre. Vi bruger Gradle i både desktop og Android appene og grunden til at vi anvender det er fordi at det er den eneste måde at vi kan bruge Firebase i vores java desktop program og fordi at gradle er default i AndroidStudio, så skal vi bruge Gradle I IntelliJ er der en kommando kaldet Javac vi bruger hvis vi vil bygge vores projekt til en executable når vi har færdiglavet vores program. Vi har også gjort brug af Gradle til at forbedre vores software:

```
16  javafx {
17      version = "13"
18      modules = ['javafx.controls', 'javafx.fxml']
19  }
20
21  dependencies {
22      implementation 'com.google.firebase:firebase-admin:6.12.2'
23      testCompile group: 'junit', name: 'junit', version: '4.13'
24      testImplementation('junit:junit:4.13')
25  }
```

Her i vores build.gradle i desktop appen kan vi se de dependencies som vores program har (junit, firebase), hvilket version af javafx vi kører og de moduler som javafx bruger. I desktopappen gjorde vi brug af Firebase library, da vi bruger det som vores online database som vi trækker og uploader data til, som vi så kan bruge i desktop og Android appen. I Android appen implementere vi en del flere libraries:

34	implementation 'androidx.appcompat:appcompat:1.1.0'
35	implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
36	implementation 'androidx.legacy:legacy-support-v4:1.0.0'
37	implementation 'androidx.lifecycle:lifecycle-extensions:2.2.0'
38	implementation 'com.google.firebase:firebase-firestore:21.4.3'
39	implementation 'androidx.legacy:legacy-support-v4:1.0.0'
40	implementation 'com.google.android.material:material:1.1.0'
41	implementation 'androidx.appcompat:appcompat:1.1.0'
42	implementation 'androidx.navigation:navigation-fragment:2.3.0-rc01'
43	implementation 'androidx.navigation:navigation-ui:2.3.0-rc01'
44	implementation 'androidx.lifecycle:lifecycle-extensions:2.2.0'

I androidappen er der flere implementationer, fordi de kommer indbygget i programmet. Ud af de 11 implementationer er 5 af dem eksterne libraries som vores gruppe har tilføjet. constraintLayout tilføjet vi for at bedre kontrollere hvor visuelle elementer som Buttons og ImageViews ved at låse dem fast på deres plads via constraints. Material brugte vi til at edit visuelle elementer, som for eksempel profilbilleder, så de passede bedre med layout formatet. Navigation brugte vi til at lave en navigation menu som overlayer siderne i Android appen. I build.gradle kan man også se vores applicationId, hvilket er unikt for vores projekt:

```

4  android {
5      compileSdkVersion 29
6      buildToolsVersion "29.0.3"
7
8      defaultConfig {
9          applicationId "com.example.android.androidapp"
10         minSdkVersion 21
11         targetSdkVersion 29
12         versionCode 1
13         versionName "1.0"
14
15         testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
16     }

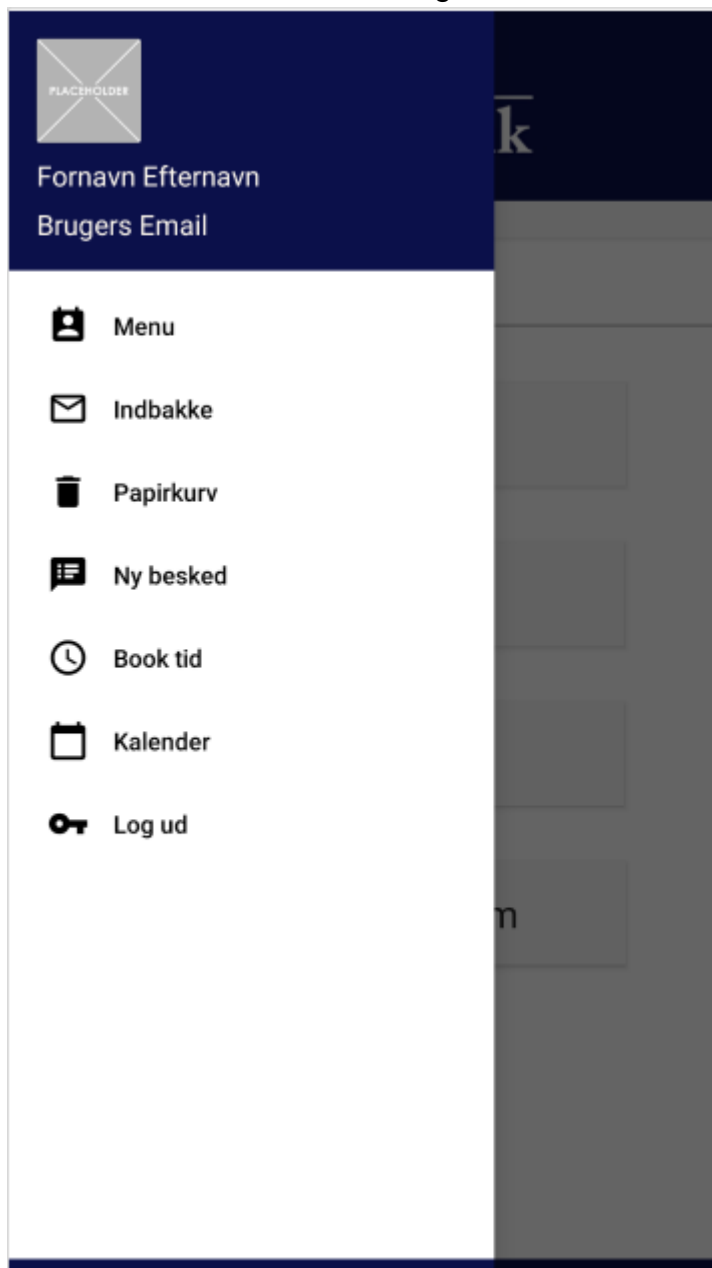
```

I Android appen kan man se SDK versionen som vi bruger til projektet (SDK version 29) og hvad minimums versionen af SDK i vores defaultConfig skal være, hvilket er 21. Vi kan også se hvad versionsnavnet på vores projekt når vi officielt udgiver det. I projektet har vi haft problemer med SDK ikke var den rigtige version for det individuelle gruppemedlem og kunne dermed ikke starte projektet, hvilket gjorde vi mistede arbejdstimer på at fikse det.

## Androidudvikling (Tommy)

Til projektet er der lavet en app til Android. Denne app er beregnet eksklusivt til vores kundes patienter, så derfor indeholder den ikke alle features som kan findes i desktopversionen. Det er features som at oprette et træningsprogram til en patient samt at knytte en behandler til en patient. Vi mente at et let tilgængeligt program på telefonen, og med kun den nødvendige funktionalitet for patienter, ville være mest passende for den demografi som vores kunde behandler. Af den årsag ville vi derfor ikke bruge ekstra resurser på at udvikle en lignende app på Pc'en.

En af måderne vi gjorde vores app let tilgængelig var at gøre den nem at navigere vha. af en menu som kunne tilgås i alle activities efter at man er loggede ind.



Til hver activity er der en header som er en del af UI'et og i denne er der, hvad normalt kaldes, en hamburger knap som man kan trykke på for at åbne menuen. Man kan også trække menuen ind med et swipe fra venstre skærmkant til midten og denne funktionalitet kommer af at menuen er et view i en DrawerLayout container.

I vores brug af denne navigationsmenu er processen for at skabe en activity ændret. Alle activities har to dele til deres XML-layout, den der gør at menuen kan åbnes og det aktuelle indhold som er vist i activity'en. Et eksempel kan ses her:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.drawerlayout.widget.DrawerLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     xmlns:app="http://schemas.android.com/apk/res-auto"
4     xmlns:tools="http://schemas.android.com/tools"
5     android:id="@+id/drawer_layout"
6     android:layout_width="match_parent"
7     android:layout_height="match_parent"
8     tools:context=".ui.MainActivity"
9     tools:openDrawer="start">
10    <!-- @author Patrick + Kelvin -->
11    <include
12        android:layout_width="match_parent"
13        android:layout_height="match_parent"
14        layout="@layout/include_menu">
15
16    </include>
17
18    <com.google.android.material.navigation.NavigationView
19        android:id="@+id/navigation_view"
20        android:layout_width="wrap_content"
```

Dette er activity\_menu.xml og det kan ses at root layoutet kun indeholder to children. Dette er fordi at det indhold der bliver vist på skærm ligger i en anden XML-fil, include\_menu.xml, som kan ses på linje 14.

```

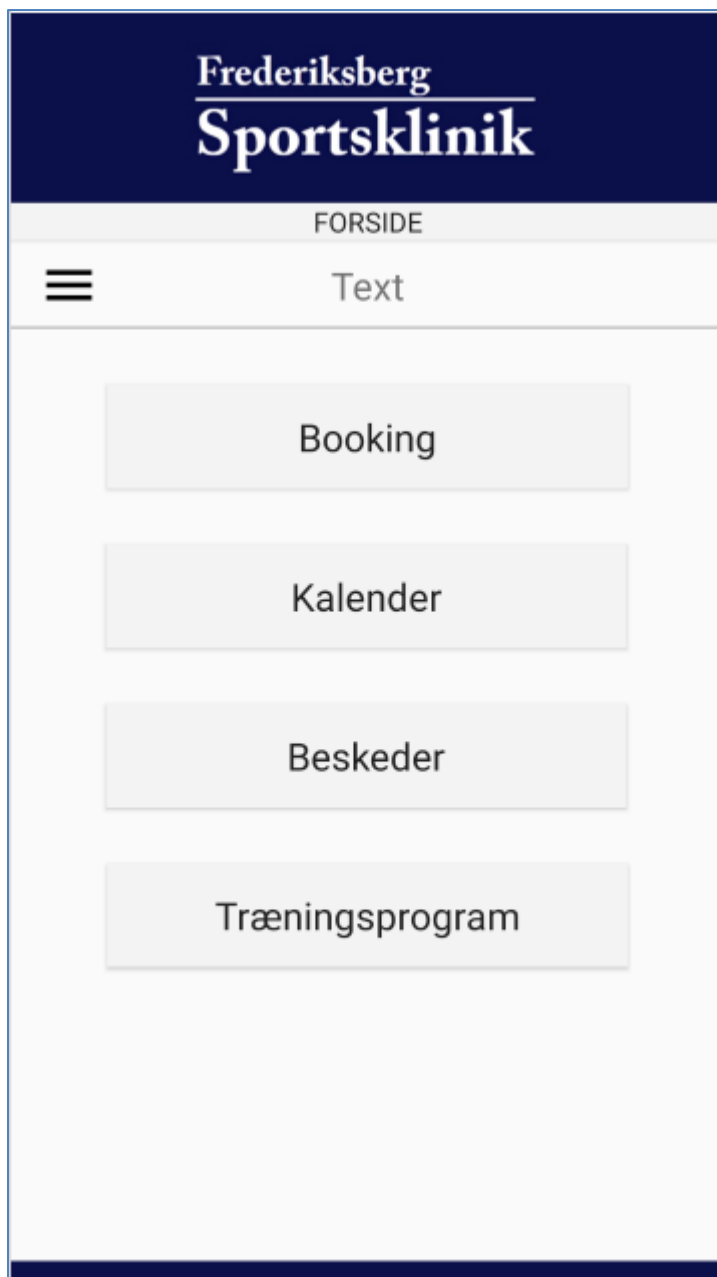
1  <?xml version="1.0" encoding="utf-8"?>
2  <androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:app="http://schemas.android.com/apk/res-auto"
4      xmlns:tools="http://schemas.android.com/tools"
5      android:layout_width="match_parent"
6      android:layout_height="match_parent"
7      tools:context=".ui.MenuActivity">
8      <!-- @@author Patrick + Kelvin -->
9      <include
10         android:id="@+id/header"
11         layout="@layout/header_skabelon"
12         android:layout_width="0dp"
13         android:layout_height="0dp"
14         app:layout_constraintBottom_toTopOf="@+id/guideline"
15         app:layout_constraintEnd_toEndOf="parent"
16         app:layout_constraintHorizontal_bias="1.0"
17         app:layout_constraintStart_toStartOf="parent"
18         app:layout_constraintTop_toTopOf="parent"
19         app:layout_constraintVertical_bias="0.0"/>
20
21     <androidx.constraintlayout.widget.Guideline
22         android:id="@+id/guideline"
23         android:layout_width="wrap_content"
24         android:layout_height="wrap_content"
25         android:orientation="horizontal"
26         app:layout_constraintGuide_percent="0.25" />

```

Denne har så alle de resterende Android views, dog med den undtagelse af de views der indgår i headeren da disse også bliver indlæst med en include som kan ses på linje 9.



Som kan ses, så med includes kan vi genbruge meget af layoutet når det er passende og I det hele set er der tre xml filer brugt for at vise dette:



Det fulde UI for MenuActivity.java hvor navigationsmenuen ikke er vist.

Dette valg af strukturen for vores UI har givet os en template vi kan bruge til alle activities men det er værd at nævne hvordan den stadig kunne gøres anderledes og stadig have samme generelle funktionalitet. Fordi vi stadig har en masse activities, så skal hver activity indeholde sit eget drawerLayout og vi ender med en masse XML-filer som kun har til forskel at deres include er anderledes. I stedet kunne man have én XML-fil til navigationsmenuen og derefter indlæse alt UI i den som fragments. Med denne model vil antallet af gentagende kode i XML-filerne formindskes og det ville være hurtigere at designe. Grunden til at vi ikke brugte denne model er således fordi at tilbageknappen på mobilen ville miste sin funktionalitet til at gå tilbage mellem activities samt at vi ville miste transitionsanimationen ved aktivitetsskiftet. Vi vurderede derfor at vi hellere vil have



en smule gentagende kode så vi kunne beholde disse to ting da brugeroplevelsen ender med at være bedre. I et andet projekt hvor det visuelle var mindre væsentligt ville det være mere praktisk i udviklingsfasen hvis der var blevet brugt fragments i kun et activity.

## Trådprogrammering (Tommy)

I vores program bruges der tråde i både Android versionen og desktopversionen hver gang der er et kald til databasen. At have databasekald i en sideløbende tråd ses som at være god form og derfor valgte vi også at inkludere det i vores projekt. Desuden opfylder det dette læringsmål:

- anvende centrale metoder og teknikker til at designe og konstruere programmer som samarbejdende processer/tråde

I brugen af tråde i projektet har vi to forskellige implementeringer der varierer mellem Android og desktop. Dette udspring af at vores database, Firebase, har forskellige API til mobile og desktop. Deres endelige funktioner er ens men måden det er kodet er anderledes. Et eksempel er metoden hentBrugerMedEmail i klassen DatabaseManager.

I Android:

```
58 public void hentBrugerMedEmail(String email) {
59     firestore.collection( collectionPath: "brugere").document(email).get().addOnSuccessListener(documentSnapshot -> {
60         Bruger bruger = null;
61         if (documentSnapshot.exists()) {
62             bruger = documentSnapshot.toObject(Bruger.class);
63         }
64         support.firePropertyChange( propertyName: "hentBrugerMedEmail", oldValue: null, bruger);
65     });
66 }
```

I desktop:

```
79 public void hentBrugerMedEmail(String email) {
80     ApiFuture<DocumentSnapshot> document = firestore.collection( path: "brugere").document(email).get();
81
82     Thread thread = new Thread(() -> {
83         try {
84             Bruger bruger = null;
85             if (document.get().exists()) {
86                 bruger = document.get().toObject(Bruger.class);
87             }
88             support.firePropertyChange( propertyName: "hentBrugerMedEmail", oldValue: null, bruger);
89         } catch (InterruptedException | ExecutionException e) {
90             e.printStackTrace();
91         }
92     });
93     thread.start();
94 }
95 }
```

Den vigtigste forskel på de to udførelser her er at Android kun kan hente data sammen med listeners, vist med lambdaudtryk på linje 59, og at i desktopprogrammet behøver man kun en try and catch, vist på linje 83 til 92. Når en onSuccessListener instantieres i Android programmet og sættes i arbejde, så vil den automatisk gøre det i en sideløbende tråd som en del af API'et. Dette er i modsætning til desktop som vil gøre det sekventielt medmindre man selv starter en tråd. Vi bruger derfor Thread klassen fra Java library for at gøre så desktop programmet agerer ligesom Android programmet. Måden de begge så kan returner data fra deres tråde er ved at andre klasser observer på DatabaseManager klassen og får den hentede data via et observerkald, ses på linje 64 og 89 i henholdsvis Android og desktop.

```
85 databaseManager.tilfoejListener(evt -> {  
86     if (evt.getPropertyName().equals("hentBrugerMedEmail")) {  
87         progressDialog.dismiss();  
88         Bruger bruger = (Bruger) evt.getNewValue();
```

Androidprogrammets databaseManager får tilføjet en listener som kan modtage kald fra hentBrugerMedEmail metoden.

Med listeners kan callbacks håndteres også selvom de sker i andre tråde samt i andre klasser og i sidste ende har vi en implementering af en database som ikke behøver at fjerne brugerkontrollen hver gang en read laves på denne database, hvilket er giver vores program en bedre brugeroplevelse.

Med dette eksempel kan brugen af tråde ses i vores program. En sidste ting der er værd at nævne er at hvis Firebase ikke startede en tråd automatisk, så havde vi flere muligheder for at gøre det alligevel da i Android så findes der AsyncTask klassen ved siden af Javas Thread klasse. I sådan et tilfælde ville vi stadig vælge Javas klasse da brugen af den anden ville have skabt en større afhængighed af Android library'et, samt at AsyncTask på gældende tidspunkt er deprecated i de nyeste versioner af Android.

## Polymorfi (Benjamin)

Polymorfi er når et objekt kan påtage mere end én form gennem nedarvning fra abstrakte klasser og interfaces (Liang 2019 kapitel 11). Der er ingen eksempler på interfaces eller abstrakte klasser i vores program, da vi ikke har haft brug for dem, men vi bruger nedarvning, når vi laver mock-klasser til test og fejlhåndtering i form af exceptions. Her er et eksempel fra unittest UTD01:

```

160     private class TestbarBrugerManager extends BrugerManager {
161         public TestbarBrugerManager() {
162             setBrugere(new ArrayList<>());
163         }
164
165         @Override
166         protected Validering newValidering() {
167             return new MockValidering();
168         }
169     }

```

Vi kan se, at TestbarBrugerManager-klassen nedarver fra BrugerManager-klassen, fordi der står extends i dens class header (linje 160). Da unittest går ud på at isolere enkelte metoder, er det smart at bruge polymorfi, da vi så kan override metoder og constructors til at gøre præcis, hvad vi vil have skal ske i testen. Da TestbarBrugerManager nu er sin egen klasse, men samtidig en BrugerManager, så kan vi give den sin egen constructor, som i dette tilfælde udelukkende fylder den liste af brugere, som den har arvet fra BrugerManager (linje 161-163). Derudover overrider vi metoden newValidering i BrugerManager (linje 165-168), fordi vi ønsker at bruge en MockValidering-klasse til at udføre test. MockValidering-klassen nedarver så igen fra Validering-klassen:

```

153     private class MockValidering extends Validering{
154
155         @Override
156         public void tjekEmail(String email) {
157     }

```

MockValidering-klassen overrider også en metode, tjekEmail (linje 155-157), som vi i dette tilfælde har brug for ikke skal gøre noget, for at få isoleret vores unittest.

Vi gør også brug af nedarvning til fejlhåndtering, som fx ForkertEmailException.java:

```

1     package entities.exceptions;
2
3     public class ForkertEmailException extends Exception{
4     }

```

- umiddelbart en ret kedelig klasse, da det eneste, den gør, er at nedarve fra Exception-klassen, men det giver os nogle interessante muligheder for fejlhåndtering. Exception nedarver nemlig fra Throwable-klassen:

```

45     public class Exception extends Throwable {

```

Throwable er en indbygget klasse i Java, så altså ikke noget vi kan tage æren for, men vi kan bruge det i vores program, fordi en exception, der er thrown, kan fanges og håndteres i en try-catch, som fx i SletPatientController.java fra ui-pakken:

```
65      try {
66          brugerFacade.sletPatient(patient, email);
67          patient = null;
68      } catch (ForkertEmailException fee) {
69          popupWindow( infoText: "Fejl: Forkert email indtastet");
70      }
```

ForkertEmailException fanges i linje 68 og håndteres i linje 69. I dette tilfælde har vi behov for at kalde en metode popupWindow og give den en String som parameter. Denne String bliver så udskrevet på skærmen i et popupvindue, og brugeren bliver gjort opmærksom på, at den angivne email altså er forkert.

## Datastrukturer (Tommy)

I dette afsnit vil vi nævne de overvejelser der er gældende angående brugen af datastrukturer og dermed vise at vi til dels opfylder dette læringsmål:

- Kan anvende centrale faciliteter i programmeringssproget til realisering af algoritmer, designmønstre, abstrakte datatyper, datastrukturer, designmodeller og brugergrænseflader

Brugen af datastrukturer i projektet har været baseret på hvad vi har haft behov for, og på det grundlag vurderede vi at ArrayList fra Javas Collections framework ville være den mest passende i alle tilfælde. Et eksempel kan ses her:

```
17      class BeskedManager {
18          private ArrayList<Chat> chats;
```

En chat er en samtale mellem to personer, hvor de også i vores program har et fast emne. Dette gør at flere chats kan eksistere iblandt de samme personer og vi kan derpå ikke vide hvor mange af dem der kommer til at være.

Som blandt andet kan ses før, så i programmet bevarer vi lister af ukendte størrelser, bestående af elementer som brugere, beskeder, og chats, hvor vi i alle tilfælde itererer igennem dem når vi skal finde specifikke elementer. For at sammenligne andre datastrukturer der kunne nævnes til sådan en use case, så er der simple arrays og LinkedList, en anden datastruktur der indgår i Collections frameworket og samtidig også bruger List interfacet, men for begge af disse er der

egenskaber der gør at vi ikke vil tage dem i brug. For simple arrays, så kan vi aldrig vide det nøjagtige antal på de objekter vi bruger i programmet og derfor er en dynamisk liste bedre også selvom den kræver mere hukommelse. Desuden, selvom det ville være muligt at arbejde med arrays alligevel hvis man havde en fast makskapacitet, så kan en startkapacitet på ArrayLists også sættes for samme effekt og disse er nemmere at arbejde med da tilføjelse af elementer altid kan gøres med add metoden. Af disse årsager vælger vi ArrayList frem for arrays.

Den anden datastruktur som ArrayList kan sammenlignes med til brug i vores program er LinkedList. Denne kunne faktisk fungere fint i vores program og endda overgå ArrayList i de tilfælde vi behøver at tilføje og hent et noget som ligger først eller sidst i listen. Alligevel så har LinkedList ikke random access og dette kan være nødvendigt fremadrettet da vi f.eks. kunne implementere redigering af specifikke beskeder valgt i UI'et af brugeren.

Med alle disse overvejelser har datastrukturer således indgået i projektet, men brugen af dem har alligevel ikke fyldt så meget da de use cases som vores kunde har krævet ikke har behøvet det. I andre sammenhænge ville vi måske kunne inddrage flere da vi nu har vist at vi kan håndtere at veje dem mod hinanden og vælge den der ender med at passe bedst.

## Sikkerhed (Benjamin)

### Encapsulation

En af måderne, vi opnår sikkerhed i vores program, er gennem encapsulation, som jeg vil demonstrere med et udsnit af Bruger.java:

```

6 public class Bruger {
7     private String navn;
8     private String email;
9     private String password;
10    private String fotoURL;
11    private boolean erBehandler;
12    private ArrayList<String> behandlere;
13
14    public Bruger(){
15    }
16
17    public Bruger(String navn, String email, String password, boolean erBehandler){
18        this.navn = navn;
19        this.email = email;
20        this.password = password;
21        this.erBehandler = erBehandler;
22        behandlere = new ArrayList<>();
23    }
24
25    public ArrayList<String> getBehandlere() {return behandlere;}
26
27    public void setBehandlere(ArrayList<String> behandlere) {this.behandlere = behandlere;}

```

Bruger.java er en klasse i vores entities package. Den bruges til at indeholde oplysninger om alle brugere i vores program. Vi kan se, at den har variable som navn, email og password (linje 7-12). Med nogle få undtagelser, så er vi generelt ikke interesserede i, at disse variable bliver ændret. Derfor bruger vi encapsulation. Det gøres ved at give klassens variable en private access modifier (Liang 2019 afsnit 9.9). Nu kan Bruger-klassens variable ikke tilgås uden for klassen. Den eneste måde, hvorpå værdien af disse variable kan manipuleres, er 1) når en instans af klassen skabes med en af dens to constructors (linje 14 og 17), eller 2) når en af klassens setter-metoder kaldes (linje 27). Constructoren og setter-metoderne har en public access modifier, hvilket betyder, at de kan tilgås uden for klassen. En tommelfingerregel er, at variable skal være private og constructors og metoder skal være public. Vi har dog også undtagelser i vores program:

```

21 public class DatabaseManager {
22     private static DatabaseManager databaseManager;
23     private Firestore firestore;
24     private boolean write;
25
26     private DatabaseManager() {
27         initializeDB();
28         firestore = FirestoreClient.getFirestore();
29     }
30
31     public static synchronized DatabaseManager getInstance() {
32         if (databaseManager == null) {
33             databaseManager = new DatabaseManager();
34         }
35         return databaseManager;
36     }

```

Her ser vi på DatabaseManager.java, som har en private constructor (linje 26). Pga. Firebasetilknytning, må der aldrig være mere end én instans af DatabaseManager-klassen, så derfor er det ikke praktisk med en public constructor i dette tilfælde, da der skabes en ny instans af en klasse, hver gang dens constructor kaldes. Til gengæld indeholder den et static instansvariabel af sig selv, fordi vi anvender singleton pattern. Singleton pattern sikrer sammen med den private constructor, at der ikke kan skabes mere end én DatabaseManager, fordi den eneste måde at instantiere en ny DatabaseManager, er ved at kalde getInstance-metoden (linje 31). En tredje access modifier, som vi bruger til at øge sikkerheden i vores program, er protected. Lad os se på BeskedManager.java:

```

79 protected BrugerManager newBrugerManager() {
80     return BrugerManager.getInstance();
81 }

```

BeskedManager har en metode, newBrugerManager med en protected access modifier (linje 79). Protected sørger for, at det kun er klasser inde i pakken, som kan tilgå metoden. Vores program er inddelt i pakkerne database, entities, model og ui, hvor BeskedManager findes i model-pakken. newBrugerManager er en metode, som kun bruges i test til at mocke BrugerManager. Derfor ville det være meget upraktisk, hvis en af de andre pakker, fx ui, kunne kalde newBrugerManager.

## Hashing

Vores programmer anvender et selvudviklet loginsystem, hvilket vil sige, at vi opbevarer brugernes informationer. Derfor er det nødvendigt at tilpasse systemet, så

GDPR-lovgivningen overholdes (<https://gdpr.dk/>). Et af lovgivningens 7 principper er fortrolighed, hvilket betyder, at uvedkomne som fx hackere, ikke kan få adgang til brugernes informationer. For at sikre vores database mod hacking, er vi derfor nødt til at kryptere eller hashe informationerne. Kryptering og hashing forvrænger informationerne, så de ikke kan læses. Forskellen på kryptering og hashing er, at kryptering foregår begge veje, så hvis man har krypteringsnøglen, kan man dekryptere informationerne, så de bliver læselige igen. Det ønsker vi ikke at gøre med brugernes password, da det heller ikke er hensigtsmæssigt, at systemadministratoren kender brugernes password. Derfor har vi valgt at anvende en hashing algoritme til at forvrænge passwordet. Hashing foregår kun én vej, hvilket vil sige, at når passwordet først er hashed, så kan det ikke gendannes. Resultatet er, at det kun er brugeren, som indtastede passwordet i sin originale form, som kender det. Det betyder også, at brugeren ikke kan få gendannet sit password, hvis han glemmer det. I kommercielle programmer vil man i denne situation i stedet tillade brugeren at oprette et nyt password. Denne funktion er på nuværende tidspunkt ikke implementeret i vores programmer, men det er noget, som vi vil udvikle, inden programmerne er færdige. De resterende informationer om brugeren, som fx navn og email, skal selvfølgelig krypteres. Dette vil vi også implementere på sigt.

Vi har lavet en klasse til hashing af tekst, som vi kalder TekstHasher.java:

```
1 package model;
2
3 import org.apache.commons.codec.binary.Hex;
4
5 import java.nio.charset.StandardCharsets;
6 import java.security.MessageDigest;
7 import java.security.NoSuchAlgorithmException;
8
9 /** @author Benjamin */
10 public class TekstHasher {
11     @ public String hashTekst(String tekst){
12         String sha256hex = null;
13         try {
14             MessageDigest digest = MessageDigest.getInstance("SHA-256");
15             byte[] bytes = tekst.getBytes(StandardCharsets.UTF_8);
16             byte[] hash = digest.digest(bytes);
17             sha256hex = new String(Hex.encodeHex(hash));
18         } catch (NoSuchAlgorithmException e) {
19             e.printStackTrace();
20         }
21         return sha256hex;
22     }
23 }
```



Klassen importerer nogle libraries (linje 3-7), som bruges i metoden hashTekst (linje 11-22). Den String, som bliver givet med som parameter til metoden, skal først omdannes til et array af bytes (linje 15). MessageDigest-biblioteket leverer SHA256-algoritmen til hashing, når digest-metoden kaldes (linje 16). Til sidst bliver det lavet til en String igen vha. encodeHex-metoden (linje 17) fra Hex biblioteket (linje 3), som returneres til det sted, metoden blev kaldt fra.

Denne hashing giver en meget grundlæggende sikkerhed i vores program, da man ikke umiddelbart kan få adgang til brugernes passwords. Vi valgte at bruge SHA-256, fordi det var en nem implementation i Java, som vi kunne bruge under udviklingen af vores programmer, og stadig sikre, at de kan snakke sammen. Efterfølgende er vi dog blevet mere opmærksomme på sikkerhedsrisiciene, som denne algoritme udgør. En hacker vil stadig kunne få adgang til informationerne, hvis han får nok tid. Der findes fx tabeller, kaldet rainbow tables, med hashes, som en hacker kan køre igennem med en computer, og dermed finde det password, som originalt blev indtastet (<https://www.baeldung.com/java-password-hashing>). Derfor er vores programmer i deres nuværende stadie ikke sikkerhedsmæssigt gode nok til at blive udgivet. Det er nødvendigt at tage yderligere skridt, for at forbedre sikkerheden:

- Salting: et hash kan saltes for at tilføje et tilfældigt element, som gør det sværere at knække.
- Konfigurerbar styrke: hashes bliver lettere at knække i takt med at computerne bliver hurtigere. Man bør derfor anvende en algoritme med konfigurerbar styrke, hvilket betyder, at man kan ændre på, hvor længe algoritmen er om at blive udført.

Med disse forbedringer i tankerne, vil vi vælge en bedre hashingalgoritme i den endelige version af Frederiksberg Sportsklub som fx. PBKDF2, BCrypt, SCrypt eller Argon2.

## Fejlhåndtering(exceptions) (Kelvin)

I vores projekt har vi gjort brug af exceptions til at fange fejl i vores programmer, her er et eksempel fra sletBruger-metoden:

```
71 @ public void sletBruger(Bruger bruger, String password) throws ForkertPasswordException {
72     String hashedPassword = tekstHasher.hashTekst(password);
73     String hashedBrugerPassword = bruger.getPassword();
74
75     if (!hashedPassword.equals(hashedBrugerPassword))
76         throw new ForkertPasswordException();
77
78     brugere.remove(bruger);
79
80     aktivBruger = null;
81 }
```

sletBruger viser hvordan at ForkertPasswordException virker, hvilket kan ses på linje 75-76 hvis hashedPassword ikke er lig med hashedBrugerPassword, så skal programmet returnere ForkertPasswordException hvorefter programmet håber ud af metoden. Nu kigger vi på hvordan exceptionen bliver brugt i sletBrugerController:

```
48 public void sletBruger() {
49     String password = tfPassword.getText();
50     String gentagPassword = tfGentagPw.getText();
51     if (!password.equals(gentagPassword))
52         popupWindow( infoText: "Fejl: password matcher ikke");
53
54     try {
55         brugerFacade.sletBruger(aktivBruger, password);
56         skiftTilStartscene();
57     } catch (ForkertPasswordException fpe){
58         popupWindow( infoText: "Fejl: passwordet er forkert");
59     } catch (Exception e){
60         e.printStackTrace();
61     }
62 }
```

Her ser vi hvordan ForkertPasswordException bliver brugt i Slet Bruger. Først skriver man password og så gentage passwordet i linje 49-50. Hvis gentag password er forkert fanger if statementen det og udskriver en fejl på linje 51-52. I try catch metoden prøver behandleren at slette en bruger, hvor hvis passwordet er forkert så kommer et popup window som informere om fejlen som set på linje 54-58.

Relateret til exceptions, kompileringsfejl og runtime fejl handler om de fejl man får på de forskellige niveauer. En kompileringsfejl sker i koden, det kan for eksempel være at man mangler et semikolon når man skriver kode hvilket er en syntax fejl.

Runtime fejl sker når man rent faktisk prøver at køre et program og det enten ikke virker ordentlig eller ikke virker overhovedet. De fejl kalder man normalt for exceptions. Eksempler på runtime fejl kan være logik fejl eller et memory leak.

ForkertPasswordException er en checked exception, det betyder at det er en fejl som bliver fanget på compile niveauet. En unchecked exception ville så være det modsatte, hvor det er en fejl som ikke bliver checked på compile niveauet og som ikke bliver registreret hvis programmøren ikke specificere eller fanger de exceptions.

## Konklusion (Kelvin, Tommy, Benjamin)

Med denne rapport har vi præsenteret vores løsning på problemstillingen, som blev stillet af kunden fra Frederiksberg Sportsklinik, og vi har dokumenteret de mest væsentlige emner om vores proces samt overvejelse under eksamensprojektet. På baggrund af dette har vi opfyldt læringsmålene for Programmering, Systemudvikling og Virksomhed til bedste evne, og vi mener, at vi har et dokument, der bevilger gruppen at gå til eksamen. Yderligere nævnes det igen at programmet vil blive udgivet som det var ved rapportens afslutning, men at der fortsat vil blive arbejdet videre på det indtil eksamen.

# Litteraturliste

Y. Daniel Liang, Introduction to Java Programming and Data Structures, 11th edition, Pearson, Armstrong State University, 2019, ISBN: 1-292-22187-9

Craig Larman, Applying UML and Patterns: An introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd edition, Upper Saddle River, NJ, USA, Prentice Hall PTR, 2004, ISBN: 0-13-148906-2

GDPR Lovgivningen - <https://gdpr.dk/> besøgt 8/6-2020

Hashing a Password in Java - <https://www.baeldung.com/java-password-hashing> besøgt 8/6-2020

MySQL Cheatsheet - <https://devhints.io/mysql> besøgt d. 8/6-2020

Singleton Anti-Pattern - <https://www.michaelsafyan.com/tech/design/patterns/singleton> besøgt d. 9/6-2020