



ATHENA



UDARBEJDET AF:

ELIAS, KAMILA, ALEXANDER, MADS

AFLEVERET DEN:

12/06/2024

ANSALG:

XXXX



Elias Balle Therkildsen

Computer science
EASV, DMU 23
Denmark, Sønderborg
elias3004@outook.dk / elithe01@easv365.dk
<https://github.com/eliastherkildsen>

Mads Raun Knudsen

Computer science
EASV, DMU 23
Denmark, Sønderborg
mads@rknudsen.dk / madknu01@easv365.dk
<https://github.com/Bolt404>

Kamilla Norborg Appel

Computer science
EASV, DMU 23
Denmark, Sønderborg
kamilla.n.madsen@gmail.com / kamapp01@easv365.dk
<https://github.com/kamapp01>

Alexander Jacob Gerckens

Computer science
EASV, DMU 23
Denmark, Sønderborg
alexander.gerckens@gmail.com / aleger01@easv365.dk
<https://github.com/Gercken>

Indholdsfortegnelse

Problemformulering	4
Afgrænsning.....	4
Konklusion	5
Reflektion.....	6
Metode	8
Inception fasen.....	9
Elaboration fasen.....	9
Construction fasen.....	10
Transition fasen.....	10
Inception fasen	11
Vision	11
Data dictionary	11
Overordnede use cases	12
Domain model	13
Feasibility study.....	14
IT og organisation	17
Analyse af systemets stakeholders	17
Projekt constraints i relation til projektledelse	18
Organisatoriske aspekter i relation til IT-projekter.....	19
Strategi for implementering af bookingsystemet.....	24
IT og organisation delkonklusion	25
Elaboration- og Construction fasen	26
1. iteration:	26
Database struktur.....	31
Kernearkitektur	33
Design	37
Prototype – Usability Test.....	37
Public subscriber.....	54
2. iteration:	59
Transition	67
Installations guide	67
Brugervejledning.....	68
Test.....	69

Bilag	75
Bilag 1	75
Bilag 2	75
Bilag 3	76
Bilag 4	80
Bilag 5	83
Bilag 6	84
Bilag 7	85
Bilag 8	88

Problemformulering

Formålet med dette projekt er at udvikle et informations- og bookingsystem til en uddannelsesinstitution. Behovet opstår på baggrund af mangel på gæsteinformationssystemer, hvorigennem besøgende, ansatte og studerende kan få information om møder, tilhørende lokaler og tidspunkter.

Der ønskes yderligere at systemet kan håndtere at studerende kan booke lokaler ad-hoc.

Projektet lægger op til at være et flere-brugersystem. Dette vurderes ud fra ønsket om en infoskærm til overblik og ad-hoc bookinger samt en administrativ computer, hvorpå de ansatte kan foretage bookinger af lokaler og lignende.

Systemet har to primære brugergrupper: studerende og administrativt personale (alle ansatte). Alle ansatte har adgang til admin-computeren og har den samme admin-rolle med et admin login. Dette rejser spørgsmål om, hvordan man sikrer en passende kontrolmekanisme for administrative handlinger, såsom ændringer af inventar, oprettelse/sletning af lokaler mm., idet alle ansatte har adgang til den samme admin-konto.

Projektet vil fokusere på at opfylde krav om bruger booking, både ad-hoc og igennem admin kontrolpanelet. Derudover vil en central del af systemet bestå i at kunne danne statistik over et givent lokale i en specifiseret tidsperiode.

Der vil ikke blive taget højde for it-sikkerhed i form af kryptering af kodeord, databaselogs m.m. da det ligger uden for pensum.

Afgrænsning

I dette afsnit vil vi sætte rammerne for vores projekt. Vi vil tage stilling til projektets tilvalg og fravalg ud fra de fastlagte kravspecifikationer, givet i casen, samt det møde, vi havde med kunden inden projektets start.

Ved at afgrænse vores projekt sikrer vi, at vi arbejder effektivt inden for vores ressourcer og tidslinjer, og at vi opnår vores mål på en struktureret og målrettet måde.

Afgrænsninger:

- **Funktionalitet:** Selv om idéen med et bookingsystem ligger op til at det skal udvikles som et flerbrugersystem, har vi valgt at lave det som et enkeltbruger system. Dette er valgt, da flerbruger system-udviklingen ligger uden for pensum. Dernæst har vi også valgt at simulere, at der er blevet integreret et ERP-system, ved at lave en tabel i databasen med nogle eksempel e-mails som har samme form som en EASV-e-mail. Dette er besluttet på baggrund at de tidsrestriktioner vi er underlagt.

- **Teknologi:** Databasen vil blive udviklet ved hjælp af Microsoft SQL (MSSQL) og det vil køre som en lokal database. Derudover bliver systemet udviklet i programmeringssproget Java.
- **User interface:** User interfacet vil blive inddelt i to, en til infoskærmen og en til admin siden. Ved udviklingen af infoskærm interfacet vil der blive lagt vægt på en god user experience (UX). Her vil der blive taget højde for den feedback, der bliver givet fra prototypetestene. Modsat på admin interfacet vil der blive lagt vægt på funktionaliteten over UX. Dog vil det ikke blive glemt.
- **Tidsramme:** Projektet skal færdiggøres inden den 12. juni, hvilket sætter begrænsninger for hvor mange ekstra funktionaliteter, der kan tilføjes.

Konklusion

I dette afsnit vil vi overordnet samle op på rapportens delkonklusioner, og holde det færdige produkt op imod problemformuleringen og de tilhørende kravspecifikationer.

På baggrund af feasibility studiet konkluderes det, at markedet er eksisterende men ikke mættet, hvilket understøtter systemets eksistensberettigelse.

Den økonomiske analyse indikerer potentiale for omkostningsbesparelser ved bedre udnyttelse af ressourcer, mens den teknologiske analyse bekræfter systemets muligheder ved at undgå komplekse teknologier. Dog fremhæves en etisk bekymring omkring brugen af personsporbare oplysninger, hvilket kræver omhyggelig håndtering og overholdelse af GDPR. Dette løser vi ved at informere brugeren om at dataene vil blive brugt til analyser. Hernæst kan brugen kun bevæge sig videre i systemet ved at acceptere TOS.

Det konkluderes at Systemet er udviklet i overensstemmelse med de kravspecifikationer som er stillet i problemformuleringen. Med mulighed for at administrere systemet igennem en admin computer. Ydre mere funktionalitet for bruger at booke lokaler ad-hoc igennem en informations skærm.

Reflektion

Det står os klart efter processen med at udvikle ATHENA-systemet, at afgrænsningen har været essentiel for at kunne opnå de mål, vi har sat os. Vi ville fra starten meget gerne udvikle ADHOC-delen af vores produkt, som virkede gennemført og teknisk korrekt. Dette føler vi, at vi har udrettet.

Admin-delen var nedprioriteret, både fordi vores stakeholders er IT-kompetente, men også fordi vi var nødt til at afgrænse os. I reflektion kan vi dog sige, at vi har afleveret noget, som var over forventning. Afgrænsningen af denne del af programmet var dog en stor hjælp til at få ADHOC-delen fuldført.

Use Cases

Vi har været gode til at få nogle brugbare use cases, som vi har kunne arbejde ud fra. I tidlige projekter har vi nogle gange følt, at vores use cases ikke helt har kunnet tilpasses det arbejde, der skulle udføres. Dette har ikke været tilfældet denne gang, hvor 40 use cases virkede som et realistisk antal til et projekt af denne størrelse. Selvom der er forekommet arbejde, der ikke har passet til en use case, men har været nødvendigt, ofte boilerplate, så er langt størstedelen af vores arbejde bundet til en specifik use case.

Prototype

Vores tidlige prototype var et godt værktøj til at læne sig op ad, både med det visuelle design men også med funktionaliteten. Uden denne ville vi have brugt langt mere tid individuelt på at forstå og bearbejde enkelte use cases. Det var en investering af ressourcer, som gav os godt igen på tidsfronten.

DAO og Stored Procedures

Brugen af DAO gav nogle udfordringer. Vi opdagede hurtigt at arbejde ud fra et CRUD-perspektiv var ineffektivt og ikke levede op til vores krav.

Desuden fandt vi det problematisk og uoverskueligt at skulle bruge mange DAOs for at opnå vores mål med databasen. Derfor skiftede vi til stored procedures for de fleste af vores databaseintegrationer, hvilket også afspejles i vores kode.

At kunne sortere data og implementere logik direkte i SQL giver en stor fordel med hensyn til validering og effektivisering på program-siden.

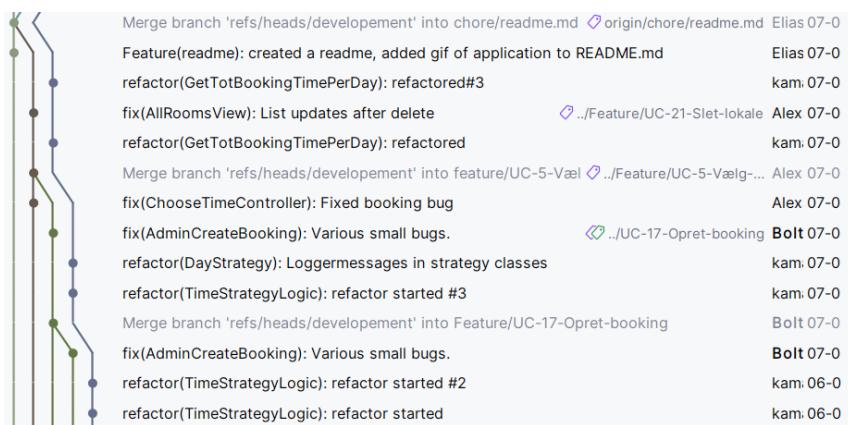
Det ville både være mindre effektivt og tidskrævende at implementere alle vores DB-integrationer med DAOs/CRUD.

PUBSUB Princippet

Vi er stolte over vores brug af PUBSUB-princippet. Det læner sig godt op ad ADHOC-delen og har givet en god indsigt i, hvordan man med builder pattern kan lave et objekt i sekvenser.

GitHub

Implementation af vores GitHub branching strategy var en fantastik tilførelse til vores projekt. At kunne binde feature branches direkte til use cases gjorde en kæmpe forskel i forhold til at finde rundt i vores GitHub repository, og gjorde at samarbejde mellem projekt medlemmer var nemmere. Noget som også kan ses på vores commit history.



billede 1 - GitHub Commit eksempel

Metode

I dette projekt bruger vi den agile metode Unified Process (UP), som et værktøj til projektledelse og systemudvikling. Metoden fungerer ved hjælp af iterationer, hvilket tillader løbende evaluering og forbedring igennem projektets forløb. Vi planlægger i alt to iterationer, hvoraf hver iteration består af både elaboration- og construction-faserne. Efter endt inception fase følges disse to iterationer af transition fasen.

Planlægningen af projektets faser ses illustreret nedenfor:

Fase	Farvekode
Inception	Blå
Elaboration	Orange
Construction	Grå
Transition	Grøn

Tabel 1: iterations farvekoder

Maj 2024						
Mandag	Tirsdag	Onsdag	Torsdag	Fredag	Lørdag	Søndag
	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

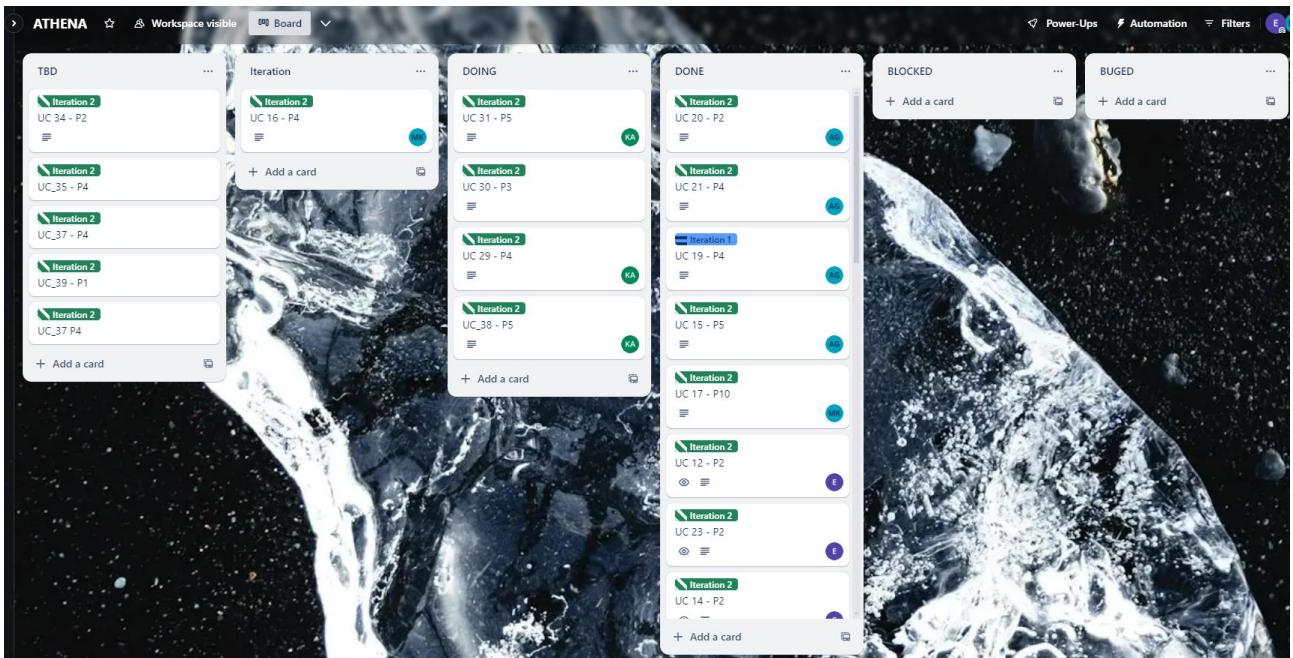
Tabel 2: Kalender med sammenhængende iteration maj 14 - 31

Juni 2024						
Mandag	Tirsdag	Onsdag	Torsdag	Fredag	Lørdag	Søndag
					1	2
3	4	5	6	7	8	9
10	11	12				

Tabel 3 kalender med sammenhængende iterationer juni 1 - 9

For at optimere vores arbejdsproces har vi valgt at adoptere "daily meetings" fra den agile metode Scrum og at anvende et Kanban Board¹. Vi har fra tidligere projekter erfaret, at disse daglige møder samt brugen af Kanban board fungerer rigtig godt for vores gruppe.

¹ <https://www.leankursus.dk/videns-side/hvad-er-kanban-board/>



Billede 2: Billede af kanban board efter 2. iteration.

De daglige møder med inddragelse af vores kanban board skaber overblik og giver indsigt i, om vi er med eller bagud i forhold til tidsplanen. Derudover hjælper det os med tidligt at identificere og håndtere potentielle problemer i processen. Møderne sikrer ligeledes, at alle teammedlemmer er opdaterede om projektets fremgang og mål.

Ved at kombinere dette Scrum element og Kanban boardet med UP's strukturerede tilgang, opnår vi en mere effektiv projektstyring.

Inception fasen

I denne fase vil vi identificere projektets omfang og fastlægge grundlæggende krav. Dette gøres ved at skabe forståelse for projektet og dets domæne gennem generelle kravspecifikationer og en domænemodel. Derudover vil vi gøre det klart, hvad stakeholder ønsker med projektets vision.

Dette vil ske med et indledende kundemøde med key stakeholder og ved hjælp af den udleverede case. Vi vil vurdere, hvorvidt det er muligt at gennemføre projektet gennem en feasibility study. For bedre forståelse af projektet og systemet startes udarbejdelsen af en data dictionary.

Elaboration fasen

I denne fase vil vi fokusere på analysen og designet af systemets arkitektur. Vi vil specificere de generelle use cases fra inception fasen og tidsestimere dem. De mere komplekse use cases udvælges, og uddybes til enten

casual- eller fully dressed use cases. De fully dressed use cases vil yderligere uddybes med tilhørende System Sequence Diagrams (SSD).

Til at holde styr på projekts use cases og fremgang, vil vi anvende et Kanban Board. Her vil vores use cases blive delt ud i iterationer, og i construction fasen dagligt blive fulgt i deres implementering af systemet.

Vi vil anvende Entity-Relationsmodellen (E-R modellen) til at udarbejde databasestrukturen samt tilhørende SQL-script. Ydermere vil vi udvikle, teste og analysere en prototype før construction fasen for at opnå det bedst mulige UI- og UX-design. Vi vil tage beslutninger om systemets kernearkitektur, og besvare spørgsmålene om ”IT og organisation” i relation til projektet og business understanding.

Construction fasen

I denne fase vil vi udvikle systemet. Vi vil implementere funktionaliteterne fra kravsspecifikationerne med tanke på resultatet af prototypeanalysen og de beslutninger, der blev taget om systemets kernearkitektur i elaboration fasen. Vi vil gøre vores bedste for, at alle use cases bliver fuldt ud implementeret, og at systemet opfylder det, ønskede UI- og UX-design. Dog vil der blive lagt fokus på at implementerer de use cases med højst prioritet først.

Transition fasen

Denne fase vil omfatte test af systemet og udarbejdelsen af en brugervejledning. Vi vil færdiggøre den skriftlige opgave, heriblandt lave systemets UML-diagram og SD-diagrammer for dele af systemet, og dokumentere, hvad konklusionen af projektet måtte være.

Inception fasen

I dette afsnit foretages alt forudgående analyse af domænet. Vi vil oparbejde en forståelse af projektets omfang, dets kravspecifikationer og domænemodel, samt komme ind på projektets vision og business perspektivet i forhold til IT og organisation.

Vision

Formålet med systemet er at udvikle et informations- og bookingsystem. Systemet skal gøre det muligt for ansatte og studerende at få et hurtigt overblik over institutionens lokaler og deres tilgængelighed. Dette skal foregå på institutionens infoskærm eller på admin-computeren. Det skal være muligt for de ansatte at booke lokaler til undervisningen, møder mm. og bestille forplejning hvis nødvendigt. Som studerende skal det være muligt at booke ledige lokaler ad-hoc. Ydermere skal systemet kunne håndtere dataudtræk og vise booking statistikker. (version 2.0)²

Data dictionary

NR	ORD	BETYDNING
1	Ad-hoc booking / Ad-hoc	En booking der sker på dagen, kun med tidsrum for denne dag.
2	User	Generel bruger af systemet.
3	Studerende / Stud	Studerende på skolen, anses som user i systemet.
4	Admin	Administration på skolen, ansatte og folk med udvidet adgang.
5	Lokale / Mødelokale / Værksted	Et rum med antal pladser, inventar og navn.
6	DD	Dags Dato.
7	Booking	Reservation af et eller flere tidsintervaller for et lokale.
8	Admin Booking	En nedarvning af booking med en udvidelse af booking valgmuligheder
9	Fejlmelding	Information om fejl / Problemer i lokalet, plus en e-mail på fejlmelder.
10	Arkiveret Fejlmelding	En fejlmelding der er blevet handlet på.
11	Gæsteregistrering	Registrering af forventede antal personer tilknyttet booking.

² Tidligere version (version 1.0) findes i bilag 1

12	Hybrid lokale	Et lokale som egner sig til online- og fysisk undervisning.
13	Stakeholder	Ejere og/eller personer med økonomisk- og arbejdsmæssig interesse i produktet.
14	Sporbar personinformation	Informationer der kan identificere en person.
15	Møde Information	Hvilken overordnet møde type og gæsteregistrering
16	TOS	Terms of Service - Vilkår og betingelser
17	Bloatware	Software som indeholder mange og ofte unødvendige funktioner.

Tabel 4: Data dictionary

Overordnede use cases

Her vises de overordnede use cases fundet i forbindelse med inception-fasen. Use casen tidsestimeres og prioriteres ud fra en lineær skala (1 - 10) hvor 10 er den højest prioriterede.

Prioriteringen sker ud fra et stakeholderperspektiv. I og med at opgaven er en eksamensopgave, og relationen til stakeholderes dermed er anderledes end i virksomhedsøjemed, påtager gruppen selv stakeholderrollen ved prioriteringen af følgende use cases.

Actor	Navn	Prioritet	Estimat i timer
System	Vis bookinger	10	8 timer
Stud	Ad-hoc booking	10	37 timer
System	Vis ledige lokaler	8	37 timer
Admin	Admin booking	10	10 timer
Admin	Rediger booking	4	7 timer
Stud	Slet booking	2	4 timer
Admin	Slet booking	6	7 timer
Admin	Opret lokale	6	6 timer
Admin	Rediger lokale	5	4 timer
Admin	Opret fejlmelding	5	6 timer
Admin	Arkiver fejlmelding	3	2 timer

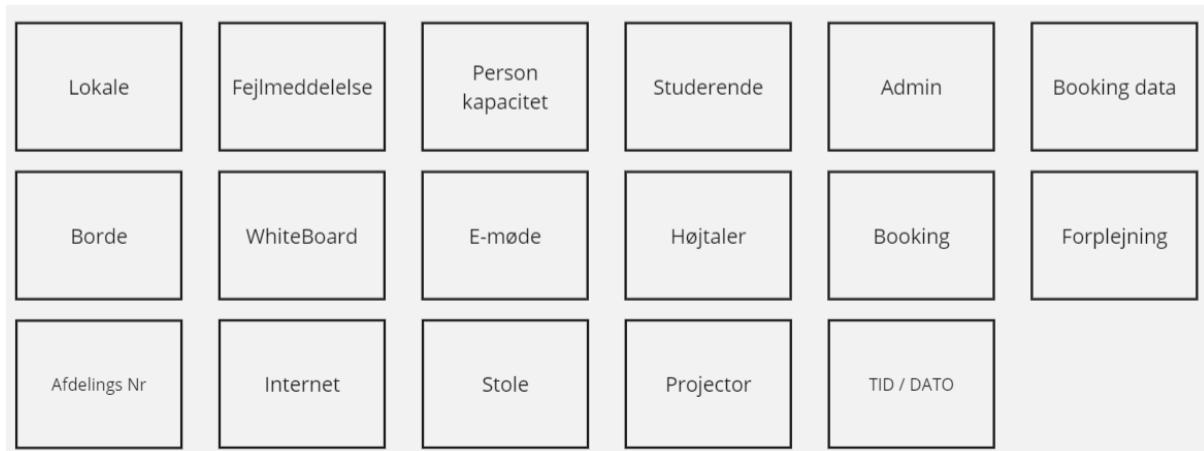
System	Vis booking statistik	4	40 timer
System	Generere CSV-fil	7	4 timer
Version 1.0			

Tabel 5: overordnede use cases

Domain model

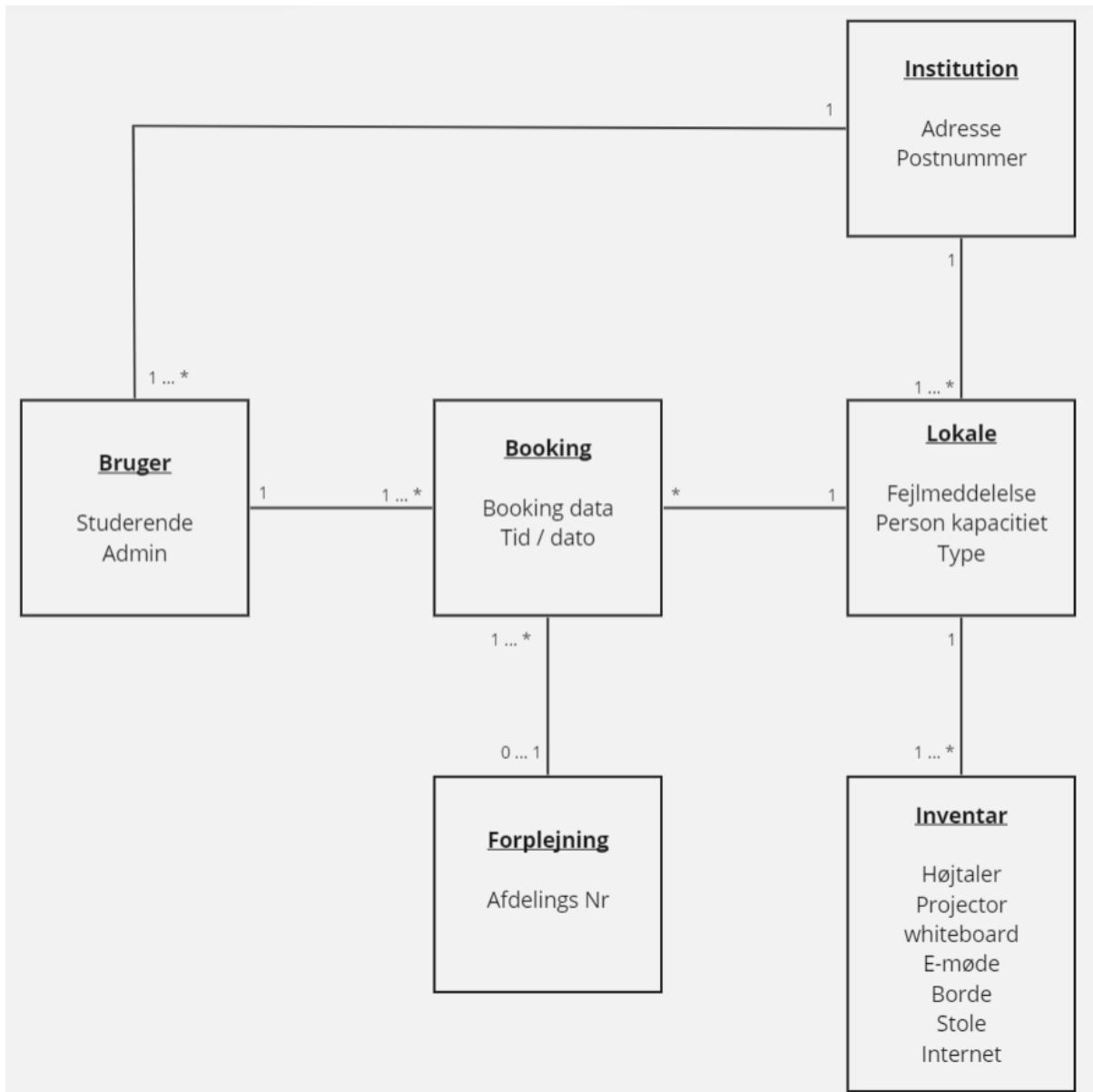
Det generelle domæne i dette projekt er en institution med lokaler, der kan indeholde forskellige typer af udstyr. Dette kan være en projektor, højtaler m.m. Derudover kan der, i tilfælde af at brugerne er administrator, også bestilles forplejning.

For at få en bedre forståelse for domænet er det relevant at lave en domainmodel. Se **Fejl! Henvisningskilde ikke fundet.**



Billede 3: Domain model for en institution med lokaler der kan udlejes.

Brugen af en domain model sikrer, at udviklingsgruppen danner et overordnet billede over fx en institution, hvilket understøtter de efterfølgende faser i Unified Process. Objekterne i domænet har som oftest en relation til hinanden. For at styrke forståelsen af disse relationer dannes kardinaliteter imellem selv samme. Se Billede 4: Domæne model og de tilhørende kardinaliteter.



Billede 4: Domæne model og de tilhørende kardinaliteter.

Feasibility study

I dette afsnit undersøges, hvorvidt systemet kan udvikles. Dette sker ud fra følgende perspektiver:

1. Konkurrenter
2. Økonomi
3. Teknologi
4. Etik
5. Lovgivning

Konkurrenter

Under feasibility studier er det vigtigt at danne sig et overblik over eventuelle konkurrenter, og hvorledes disse differentierer sig fra markedet.

Under denne fase er følgende interessante konkurrenter til vores system fundet:

1. Gecko³
2. Outlook Calendar⁴
3. Google Calendar⁵

Hvis man kigger nærmere på Gecko, som en mulig løsning, så opdager man, at deres bookingmodul giver en masse andre muligheder, end bare at kunne booke lokaler. Det kan give mange funktioner, som ikke er nødvendige, og bare bliver betragtet som bloatware.

Er det økonomisk muligt?

Det er centralt at have det økonomiske perspektiv i mente under konceptudviklingen af systemet. Det følgende afsnit vil derfor indeholde relevante perspektiver inden for økonomi.

Systemets formål er at udnytte de faste udgifter bedst muligt på en given institution ved at højne lokalernes belægningsprocent. Heraf findes et af de økonomiske incitamenter fra stakeholderne til udvikling af programmet.

For at sikre at systemet er økonomisk rentabelt, er det vigtigt at undersøge, hvorvidt der er et marked for systemet, og hvorvidt lignede programmer allerede findes.

Programmet er tiltænkt små og mellemstore virksomheder (SMV) med lokalekapacitet, som ikke bliver udnyttet. Grunden til at vi har valgt at afgrænse programmet til kun at inkludere SMV er, at større virksomheder oftest benytter programmer inkluderet i deres ERP-system.

Er det teknologisk muligt?

På baggrund af det allerede eksisterende marked for lignende systemer står det klart, at der ikke er nogen teknologiske begrænsninger for udviklingen af dette system.

Da systemet er en del af et 2. semesters eksamsprojekt, er der klart definerede teknologier, som underviserne ønsker brugt. Disse er som følgende:

1. MS-SQL
2. Java
3. Java-FX

³ <https://www.geckobooking.dk/>

⁴ <https://support.microsoft.com/en-us/office/welcome-to-your-outlook-calendar-6fb9225d-9f9d-456d-8c81-8437bfcd3ebf>

⁵ <https://calendar.google.com/>

Ovenstående er alle modne teknologier med bred anerkendelse blandt udviklere. Det skønnes derfor, at der ikke vil opstå teknologiske risici på baggrund af overståendes brug af systemet.

Er det etisk muligt?

Ud fra den givne problemformulering og det efterfølgende kundemøde antager vi, at brugernes informationer blandt andet skal bruges til statistiske formål. Vi har derfor valgt at gøre brug af en e-mail, som brugeren skal indtaste ved booking af et lokale. Dette gør dog, at der skal tages hensyn til GDPR, da en e-mail er person sporbare information.

Er det lovgivningsmæssigt muligt?

Da systemet primært er beregnet til intern brug og ikke indebærer nogen form for brugerbetaling, er der kun minimalt abonnement på persondata ved booking af lokaler. Brugen af e-mail og navn til statistiske formål er underlagt GDPR, Artikel 6, stk. F⁶, der tillader behandling af data til videnskabelige formål. Dette inkluderer indsamling af statistik.

Feasibility study konklusion

På baggrund af feasibility studiet konkluderes følgende:

Det konkluderes, at markedet allerede er eksisterende, men ikke mættet. Vi finder derfor, at systemet er eksistensberettiget.

Ud fra den økonomiske analyse fremgår det, at de faste omkostninger hos institutionen kan nedbringes ved bedre udnyttelse af lokalerne.

Den teknologiske analyse konkluderer, at systemet ikke gør brug af nye eller eksotiske teknologier. Dette sikrer, at systemet er teknologisk muligt.

Under den etiske analyse af systemet tydeliggøres et problem, hvilket er, at der skal bruges person sporbare oplysninger, som skal tages højde for i udviklingsprocessen.

Systemet berører GDPR lovgivning, da der indtastes, opbevares og bruges person sporbare informationer til legitime interesser. Derfor skal brugeren samtykke TOS for at kunne fortage en booking.

⁶ <https://www2.phabsalon.dk/studienet/studiehjaelp/informationssikkerhed/politikker-og-vejledninger/lovlige-behandling-artikel-6/>

IT og organisation

I dette afsnit foretages en business relateret analyse af systemet i forhold til IT og organisation. Vi vil udarbejde en stakeholders analyse, analysere systemets projekt constraints, se på organisatoriske aspekter i relation til IT-projekter og give vores bud på en strategi for implementering af bookingsystemet

Analyse af systemets stakeholders

Følgende stakeholders er identificeret:

- Lære
- Studerende
- Pedel
- Ledelse

Disse stakeholders klassificeres ud fra en interesse/influence matrix. Her bliver de tildelt en værdi i forhold til, hvor indflydelsesrige de er - denne er enten lav eller høj. Derudover tildeles de også en værdi i forhold til, hvor interesseret de er i systemet - denne er også inddelt i enten lav eller høj. Her er det værd at nævne, at der med interesse i systemet skal forstås, hvor villige de er til at bruge systemet.



Billede 5: Matrix over interessenter.

Ovenstående matrix viser stakeholders med hhv. høj indflydelse og høj interesse for systemet. Disse er dernæst blevet involveret tæt i systemudvikling, og bliver kaldt key stakeholders. Dette er blandt andet sket ved at benytte disse som testpersoner under test af prototypen.

Derudover er der også nogle stakeholders, som er placeret i kvadranten med høj interesse og lav indflydelse (Elever samt pedel). Det er vigtigt at holde disse stakeholders informeret, men der er ingen behov for at involvere dem, når der skal træffes beslutninger om systemets udvikling.

I projektets opstartsfase blev der foretaget et interview af key stakeholder Jesper Mølgaard Axelsen, som også er projektejer. Her er systemkravene blevet specificeret, og yderligere spørgsmål er blevet besvaret. Disse findes i bilag 2, men her er et udpluk af interessante spørgsmål samt svar fra interviewet:

1. **Spørgsmål:** Hvilken grad af forplejning ønskes der at systemet tilbyder?

Svar: Forplejning skal kun være tilladt for personale, og er en menu, som ikke kan ændres. Den består altså altid af det samme.

2. **Spørgsmål:** Forventes det at en lærer kan slette elevers booking?

Svar: Ja, en lærer skal kunne slette elevers booking, og reservere lokalet selv.

3. **Spørgsmål:** Hvilken form for statistik forventes der at kunne udledes fra systemet?

Svar: Information om et givent lokales antal bookinger.

I og med at key stakeholders er vigtige i alle projektfaserne, har vi derfor lavet en plan for, hvordan de skal inkorporeres i de forskellige faser:

- Idea phase: Dele idéer og tanker samt have samtaler om udviklingen af systemet.
- Preparation phase: Forventningsafstemme samt videregive information.
- Realization / implementation phase: Sørge for at opretholde kontakt
- Evaluation phase: Afslutte kontakten

Dette er gjort for at sikre opretholdelsen af kontakt og samtaler mellem projektteam og key stakeholders.

Projekt constraints i relation til projektledelse

I relation til projektledelse er det vigtigt at indtænke de fire forskellige projekt constraints:

- Kvalitet (Quality)
- Tid (Time)
- Økonomi (Economy)
- Mål (Goals)

idet alle fire aspekter influerer hinanden. Derfor bør der være nogle overvejelser og forventningsafstemninger med key stakeholder forud for projektets start – hvad vægtes højest? Hvad er fastlåst? Og hvad er målet?

I vores projekt er det tidsmæssige aspekt fastlåst, idet projektperioden ikke er til forhandling. Det er noget, der kan komme til at have indflydelse på kvaliteten af det færdige system. Hvis ikke alle kravsspecifikationer er opfyldt til projektets deadline, kan det være nødvendigt at slække på kvaliteten og hermed ikke komme helt i mål med den ønskede funktionalitet.

Hvad angår økonomistyringen, er det ikke noget, vi som projektgruppe har nogen indflydelse på, og vi tænker derfor ikke særlig meget over det. Key stakeholder har stillet lokaler og personale til rådighed gennem hele processen, og kravsspecifikationerne er låst (kravsspecifikationerne i casen skal være opfyldt). Vores egne timer skal ikke finansieres, og tæller derfor ikke som en del af det økonomiske regnskab.

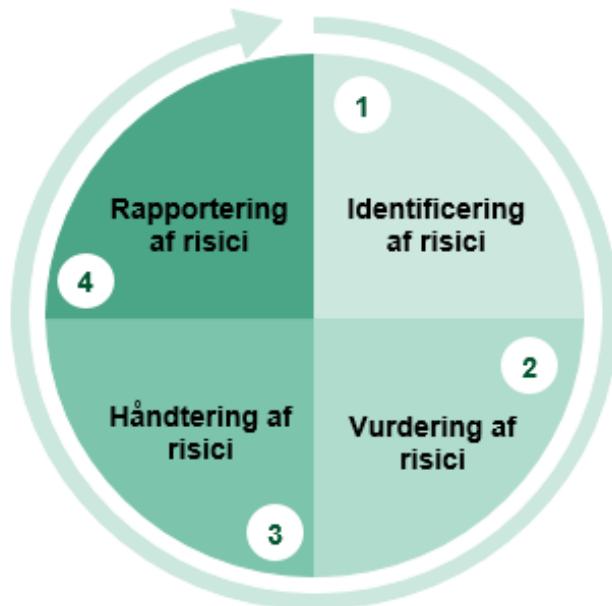
Projektgruppens ambitioner for det færdige produkt er at levere et system af høj kvalitet. Vi er opmærksomme på, at tiden kan volde os problemer, og tænker derfor meget i genbrug af kode, komponenter og gode design patterns. Afgrænsninger ift. projektet er også indtænkt som en faktor, der kan give mere tid til vores tilvalg, og derved være med til at højne kvaliteten af det færdige produkt.

Organisatoriske aspekter i relation til IT-projekter.

I relation til IT-projekter i en organisation er der flere centrale punkter at tage højde for. I det følgende har vi valgt at tage udgangspunkt i risikostyring og forandringsledelse.

- **Risikostyring**

Riskostyring i en virksomhed består af 4 søjler:



Billede 6: Risikostyrings 4 søjler⁷.

Først identificeres eventuelle risici. Dernæst vurderes de fundne risici ud fra sandsynlighed og konsekvens. Så håndteres de fundne risici ved at finde de bedst mulige strategier til effektivt at håndtere disse risici og reducere risikoen for, at de sker. Herfra undersøges der løbende, om de fundne risicis konsekvens eller sandsynlighed ændrer sig – hvis dette er tilfældet, revurderes risiciens og dens håndtering.

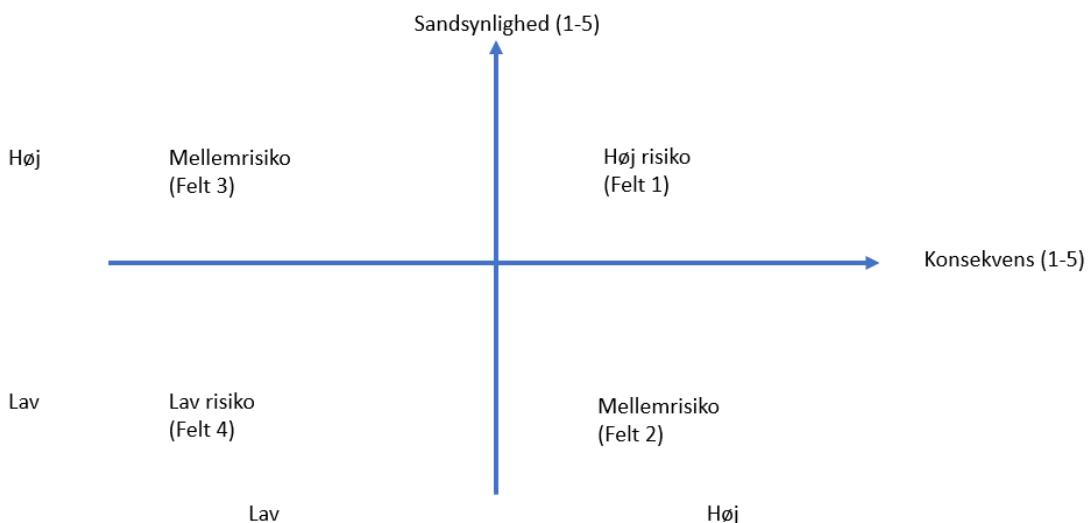
At identificere risici og rangere disse efter deres konsekvens og sandsynlighed, kan gøres som matematisk formel:

$$Risiko_a = Konsekvens * sandsynlighed$$

$$Risiko_b = konsekvens pr tid * tidsenhed * sandsynlighed$$

Dette visualiseres ofte i en matrix, baseret på udregningen af formel ”Risiko_a”:

⁷ Kilde: <https://oes.dk/it-og-oekonomistyring/risikostyring/>



Billede 7: Risiko analyse matrix⁸ der inddeler fundne risici i fire risikogrupper ud fra deres konsekvens og sandsynlighed

I relation til vores system er der flere forskellige risici, som skal tages i betragtning. Følgende er fundet:

Risiko	Sandsynlighed (1 – 10)	Konsekvens (1 – 10)	Risiko(a)
Brand	2	8	$2 \cdot 8 = 16$
Tyveri	3	8	$3 \cdot 8 = 24$
Brugerfejl	5	3	$5 \cdot 3 = 15$
Uautoriseret adgang	4	5	$4 \cdot 5 = 20$
Vandskade	1	8	$1 \cdot 8 = 8$

Tabel 6: Her opgøres sandsynlighed fra 1 til 10, hvor 10 er det mest sandsynlige. Ligeledes inddeltes konsekvens fra 1 til 10, hvor 10 har størst konsekvens.

Ud fra overstående identificeres to risici med en høj risikoværdi. Det er hhv. Tyveri (risikoværdi på 24) og Uautoriseret adgang (risikoværdi på 20). For at imødegå disse kan flere værktøjer tages i brug.

I forhold til tyveri kan systemet designes til at sænke konsekvensen ved ofte at lave backups, som gemmes på en anden lokation.

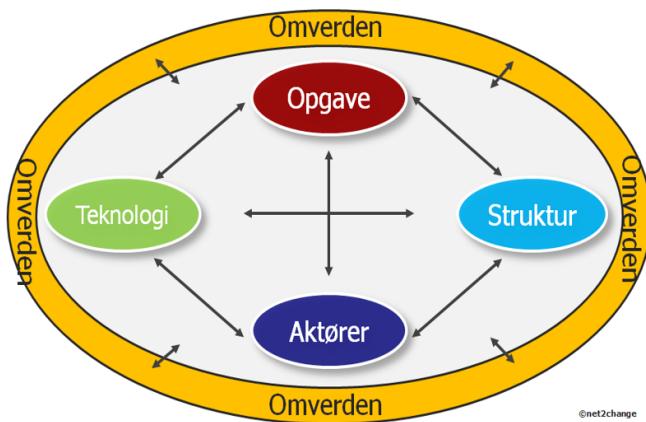
I forhold til uautoriseret adgang til systemet menes der, at en person, der ikke er tilknyttet skolen, kan få adgang til systemet og booking af lokaler. Dette kan imødegås på flere måder. Vi har blandt andet designet systemet til, at man skal have en aktiv e-mail tilhørende institutionen for at kunne booke et lokale. Derudover kunne man også indtænke, at der bliver sendt en bekræftelsesmail til mailadressen, der anvendes til bookingen. Herved tages der højde for, at en brugers e-mail kan misbruges af en anden

⁸ Business Process Optimisation

person. Det udelukker ikke sandsynligheden for at det sker, men ejeren af den misbruges e-mail bliver gjort opmærksom på det, og kan handle på det.

- **Forandringsledelse**

Når forandringer sker, er det vigtigt at have lagt en plan på forhånd, for hvordan den skal håndteres / inkorporeres. Det skyldes, at alle dele af en virksomhed hænger sammen på en eller anden måde. Det ses også i understående model (Leawitt's diamond):



Billede 8: Leawitt's diamond⁹

Leawitt modellen er her inddraget for at vise, at en organisation opfører sig som en organisme. Laver man ændringer et sted, påvirker man et andet. Hvis der tages udgangspunkt i den udvidede version af modellen, kan man se, at omverdenen også spiller en rolle i forhold til påvirkning af de forskellige dele af organisationen.

John P. Kotter og Todd Jick har udviklet teorier om, hvordan man effektivt kan lede og implementere forandringer i organisationer. De benytter forskellige tilgange og metoder til at opnå succesfulde forandringsprocesser. Nedenfor ses en tabel, der viser deres forskellige tilgange og metoder til at opnå succesfulde forandringsprocesser:

⁹ PowerPoint "Chapter 11 and 12 - project management and change management" fra business understanding undervisningen

	Jick (1991)	Kotter (1995)
1	Analyse the organisation and its need for change	Establish a sence of urgency
2	Create a shared vision and common direction	Form a powerful guiding coalition
3	Separate from the past	Create a vision
4	Create a sence of urgency	Communicate the vision
5	Support a strong leader role	Empower others to act on vision
6	Line up political sponsorship	Plan and create short-term wins
7	Craft an implementation plan	Consolidate improvements – produce more change
8	Develop enabling structures	Institutionalise new approaches
9	Communicate, involve people and be honest	
10	Reinforce and institutionalise the change	

Billede 9: materiale taget fra business understanding undervisning, PowerPoint "Chaptr 11 and 12"

Her er nogle af de centrale forskelle til, hvordan man kan implementere IT-systemer i organisationer med udgangspunkt i ovenstående tilgange:

Jick og Kotter har ultimativt samme målsætning, men hver deres måde at opnå resultatet på. Jick fokuserer på at få hele organisationen involveret fra starten gennem analyse og behovsdækning, mens Kotter arbejder ud fra et ledelsesperspektiv, hvor organisationens behov kommer i første række. Dette har også den ulempe, at motivationen skal komme fra ledelsen.

Med Jicks tilgang vil motivationen ofte komme fra projektet selv.

For at præcisere, går Kotters teori om at skabe en brændende platform ikke ud på at skabe panik hos medarbejderne. Teorien går derimod ud på at signalere, at der ude i horisonten er et bedre sted for virksomheden at eksistere. Dette skal være drivende for at en virksomhed udvikler sig. Den brændende platform betyder altså, at en virksomhed stagnerer, og ikke følger med udviklingen.

Udvikling sker ikke ad sig selv, men bygger på virksomhedens kultur. Derfor er det fra et ledelsesperspektiv nærlæggende at understøtte og arbejde bevidst med kulturen, for at sikre, at forandringen i organisationen kan ske.

Når man snakker forandringsledelse i relation til organisationer, er det ekstremt centralt at kunne orientere sig, og måle på hvordan en forandring i virksomheden klarer sig. Dette findes der flere værktøjer til - SMART mål er et af disse.

SMART

Specifikt	Målbart	Attraktivt	Realistisk	Tidsbestemt
Hvad er det helt konkrete mål, I ønsker at opnå?	Kan I måle, hvornår I har opnået målet?	Hvorfor er det vigtigt for jer, at nå det mål?	Er det realistisk, at I kan opnå målet?	Hvornår ønsker I, at målet skal være opnået?

Billede 10: SMART mål, organisatorisk værktøj for at måle virksomhedsudvikling.

Når en organisation står over for en forandring, hvilket den som regel gør i en moderne verden, kan man gøre det på flere måder.

Kotter nævner 8 acceleratorer i forbindelse med forandringsledelse. Det er nogle trin, som skal sætte skub i tingene og skabe flow. Her nævner han for eksempel kortsigtede gevinster - altså at aktøren skal kunne se en gevinst her og nu. Kotter taler også om en hær af frivillige. Med dette tænker han på, at forandringsprocessen ikke kun bæres frem af ledelsen men også af almindelige ildsjæle i organisationen. En sidste vigtig pointe der nævnes, er at opretholde accelerationen - flammen må ikke brande ud.

Her gælder det om at få tydeliggjort de gode historier, og løst eventuelle problemstillinger løbende.

Strategi for implementering af bookingsystemet

Under implementeringen af systemet kan der opstå flere forskellige udfordringer, som er relevante at indtænke. Der kan opstå fejl i systemet, som ikke er fundet i udvikling- og test fasen. Disse kan, i værste fald, føre til, at systemet i en periode ikke kan bruges, indtil fejlen er løst.

Flere forskellige strategier kan benyttes til at udrulle systemet:

- Pilot project
- Big bang
- Phase implementation
- Concurrent implementation

På baggrund af ovenstående, og for at sikre at systemet er moden, når det erstatter det allerede eksisterende system, Office Outlook-kalenderen, falder Concurrent implementation af projektet naturligt. Dette vil give

læreren mulighed for at have et system at falde tilbage på i tilfælde af nedbrud. Dette vil foregå over en fastsat tidsperiode, hvorefter systemet vil overtage helt, og det gamle system dermed udfases.

Der er et argument for at køre systemet som et pilotprojekt for isoleret at teste det. Dog vil dette være meget mere kompliceret, da det gamle system stadig vil blive brugt, hvilket betyder, at de to systemer vil være ude af sync.

IT og organisation delkonklusion

Analysen af IT og organisation har fremhævet væsentlige aspekter samt potentielle løsninger på de identificerede udfordringer.

Vi identificerede fire forskellige stakeholders: lærere, studerende, pedeller og ledelse, hvoraf lærere og ledelse blev klassificeret som key stakeholders. En plan blev udarbejdet for, hvordan disse key stakeholders skal involveres i systemets forskellige udviklingsfaser.

Vi identificerede projektbegrensninger, hvor tidsbegrænsningen blev vurderet som den vigtigste, da den er uforhandlingsbar. Dette påvirker mål- og kvalitetsbegrænsningerne, da den kan begrænse, hvad der kan opnås inden for den givne tid.

En risikoanalyse blev udført, hvor sandsynlighed og konsekvens af risici blev vurderet. Strategier blev udviklet for at håndtere risici som tyveri (backups gemt på en anden lokation) og uautoriseret adgang (systemtjek for at afvise uautoriserede brugere).

For at implementere ATHENA systemet i en institution, anbefales den concurrent metode, hvor det nye system kører samtidig med det gamle. Dette muliggør sammenligning af resultater fra begge systemer. Fremgangsmåden skal analyseres for hver enkelt institution.

Medarbejderinvolvering er vigtig ved implementeringen af ATHENA. John Kotter beskriver, at forandringsledelse kan ske ved at skabe en følelse af presserende nødvendighed (sense of urgency). Dette kan være relevant, afhængigt af institutionen. Vi anbefaler at følge Kotters teori og inddrage key stakeholder Jesper Mølgård Axelsen som motivationsfaktor.

Elaboration- og Construction fasen

I dette afsnit vil vi fokusere på kernearkitekturen, design og udviklingen af systemet. Vi vil udvikle arkitekturen, foretage designvalg og påbegynde implementeringen af systemets kernefunktioner. Yderligere vil afsnittet opdeles blive opdelt i 1. og 2. iteration.

1. iteration:

I denne iteration lægges der vægt på udviklingen af systemets ad-hoc del, herunder infoskærmen.

Elaboration fasen

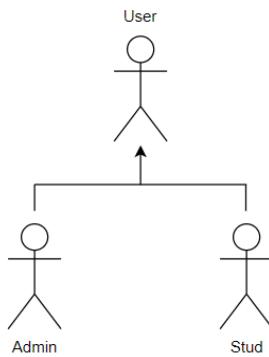
Brief use cases

I følgende afsnit omskrives de overordnede use cases til mere specifikke brief use cases. Disse vurderes ud fra følgende faktorer: Estimat i timer og prioritet. Estimat i timer dækker over den tid, som udviklergruppen, mener opgaven vil tage at udføre. Prioriteten angiver, hvor vigtig udviklergruppen finder casen. I en virkelig kontekst, f.eks. ved et softwarehus, vil man her have stakeholder ind over. Dette er dog ikke muligt i vores situation. Derfor agerer gruppen også stakeholder under prioriteringen af disse.

ID	Actor	Navn	Prioritet (1-10)	Estimat (timer)
UC_0	User	Vælg system	10	1 Time
UC_1	System, User	Vis reserverede lokaler	10	5 Timer
UC_2	System	Sorter reserverede lokaler (FD) lokale/tid	4	4 Timer
UC_3	System	Vis aktuel tid	1	1 Time
UC_4	System	Vis aktuel dato	5	1 Time
UC_5	User	Vælg starttidspunkt / sluttidspunkt	10	7 Timer
UC_6	System	Vis ledige lokaler	8	5 Timer
UC_7	System, User	Indtast Booking Oplysninger	7	4 Timer
UC_8	System	Vis booking bekræftelse	3	4 Timer
UC_9	System	Vis samtykkeerklæring	7	1 Time
UC_10	User	Svar på samtykkeerklæring	7	1 Time
UC_11	User	Slet booking	2	3 Timer

Billede 11: viser de udvalgte use cases til 1. iteration. De resterende use cases fundet i denne iteration findes i bilag X

Som det fremgår af ovenstående use cases, anvendes der en "User" som actor. Denne actor er et resultat af UP's generalisering. I vores tilfælde deler de studerende og de ansatte den samme rolle i ad-hoc delen af systemet. Vi har derfor generaliseret disse actors under "User", som det ses nedenfor:



Billede 12: Abstract user i ad-hoc delen af systemet.
De studerende og ansatte deler samme rolle

Casual use cases

I følgende afsnit bliver en af de ovennævnte brief use cases uddybet som casual use cases. Dette sker for at præcisere, hvad den indebærer.

UC_11	Slet Booking
Actor	System, User
1. User vælger Slet Booking 2. System prompter Email 3. System sends user til UC_1.	
Version: 1.0	Forfatter: Alexander, Kamilla, Elias, Mads

Tabel 7: Casual use case af UC_11

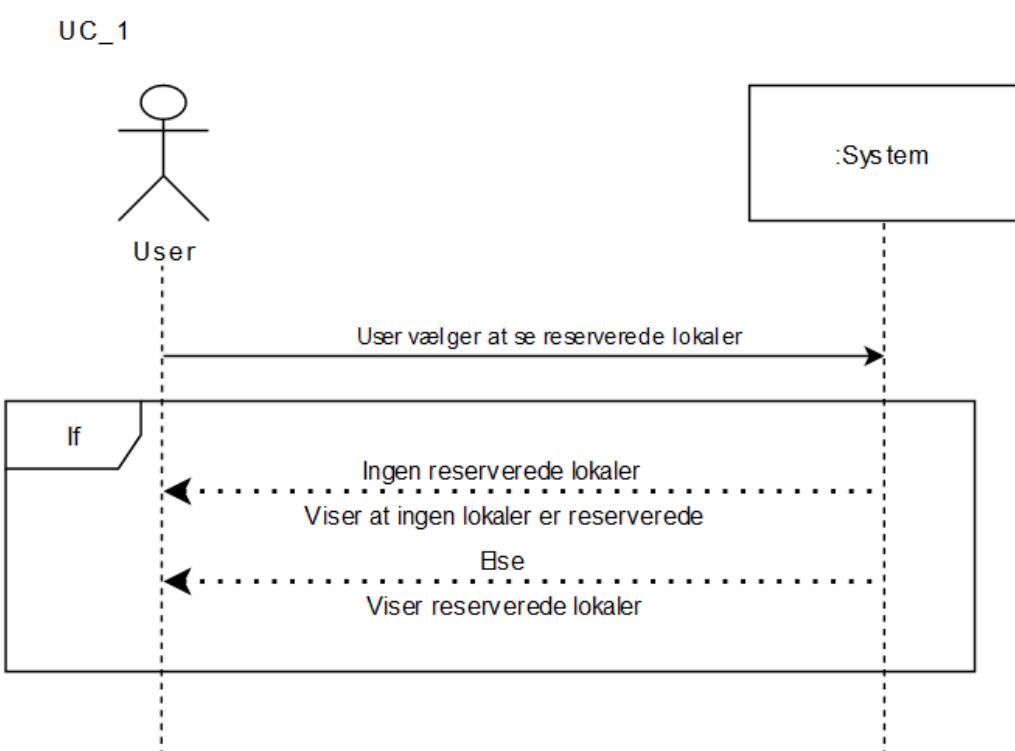
Fully Dressed UC og tilhørende SSD

I følgende afsnit bliver tre af de ovennævnte brief use cases uddybet som fully dressed use cases. Dette sker for at præcisere, hvad de indebærer og for at mindske kompleksiteten.

UC_1	Vis reserverede lokaler
Actor	System, User

Trigger	User vil se reserverede lokaler
Precondition	Der er oprettet lokaler i systemet.
	<ol style="list-style-type: none"> 1. User vælger at se reserverede lokaler. 2. Sorter reserverede lokaler/tid (UC2) 3. Systemet henter reserverede lokaler fra databasen. <ol style="list-style-type: none"> 3.1 IF (ingen lokaler er reserverede) <ol style="list-style-type: none"> 3.1.1 Systemet præsenterer på brugerfladen at ingen lokaler er reserverede. 3.2 Else <ol style="list-style-type: none"> 3.2.1 Systemet viser reserverede lokaler på brugerfladen.
Version: 1.0	Forfatter: Alexander, Kamilla, Elias, Mads

Tabel 8: Fullydresed uc_1



Figur 1: viser SSD over UC_1 – vis reserverede lokaler

UC_5	Vælg starttidspunkt / sluttidspunkt
Actor	System, User
Trigger	User booker tider til rum.
Precondition	Rum har ledige tider.
Postcondition	Nye booket tider er gemt for rum.

1. System præsenterer User for tider for gældende rum.
2. System tillader valg af ledige tider.
3. User vælger starttidspunkt ud fra block af (X) minutter.
4. User vælger sluttidspunkt ud fra block af (X) minutter.
5. User bekræfter valg.
6. System sender videre til indtast booking oplysninger (UC-7)

4.a User vælger sluttidspunkt der er før starttidspunkt.

1. System viser besked omkring invalidt valg.

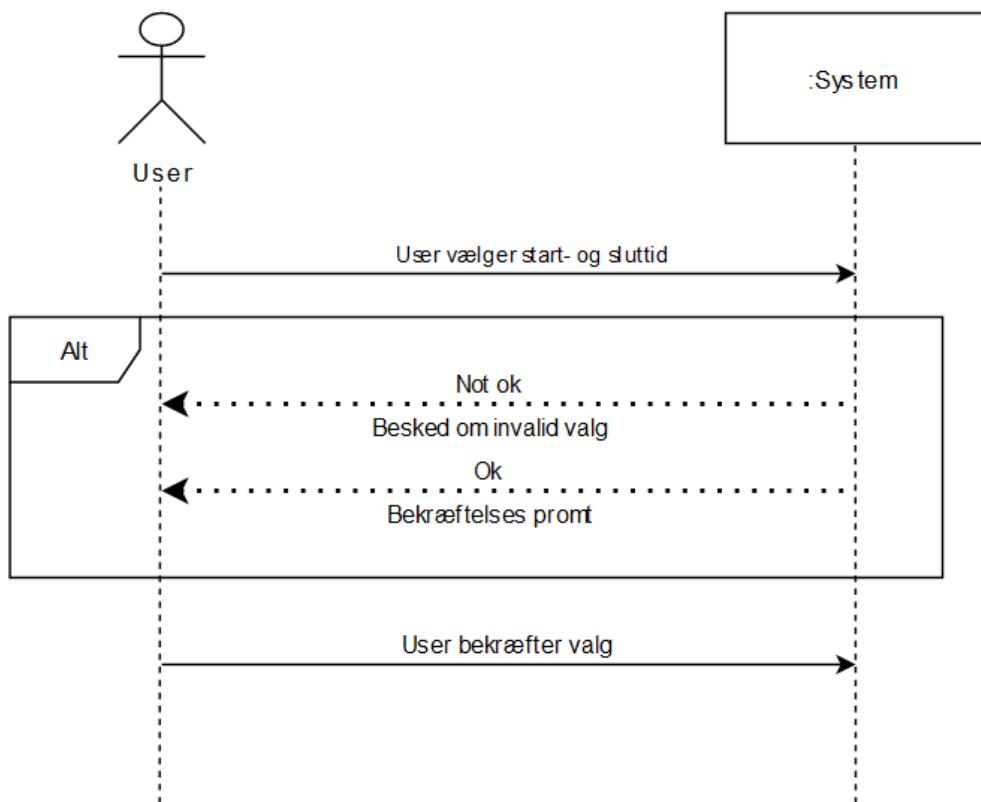
Sender user tilbage til (Step 3)

Version: 1.0

Forfatter: Alexander, Kamilla, Elias, Mads

Tabel 9: Fully dressed use case uc_5

UC_5

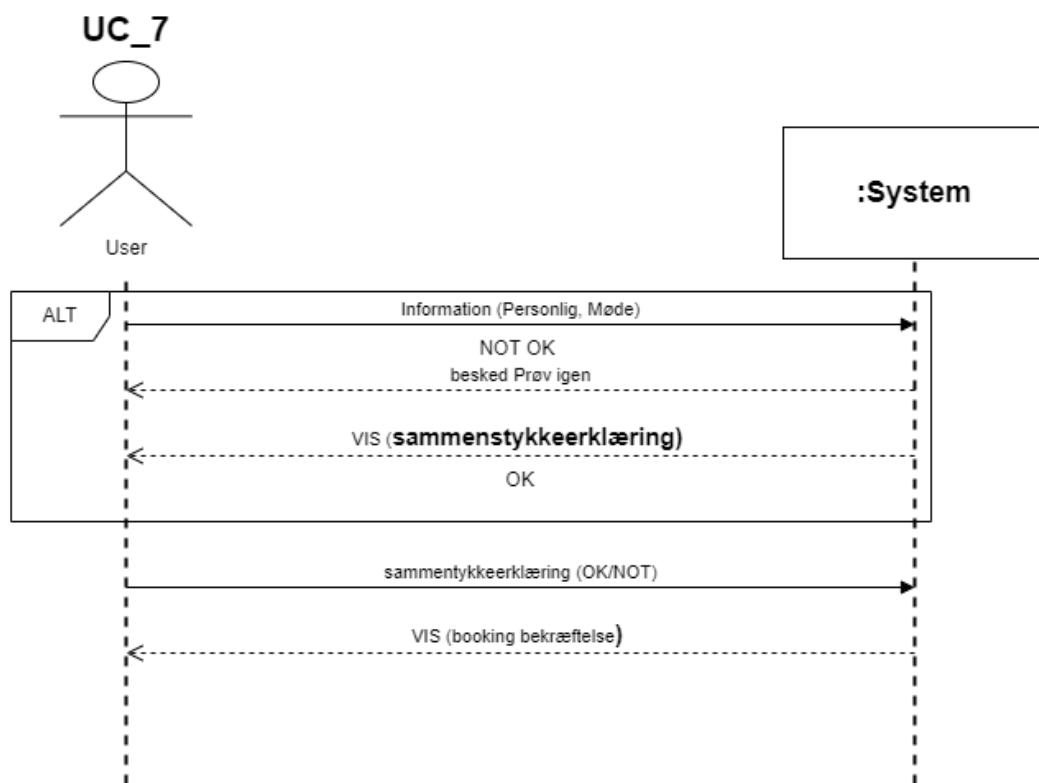


Figur 2: viser SSD over UC_5 – vælg starttidspunkt / sluttidspunkt

UC_7	Indtast booking oplysninger
Actor	System, User
Trigger	Useren har valgt rum og tidsperiode.

Precondition	Rummet er ledigt i den givne tidsperiode.
Postcondition	Rummet er booket i den givne tidsperiode.
<ol style="list-style-type: none"> 1. System promoterer brugeren til at indtaste møde informationer. 2. System promoterer brugeren til at indtaste personinformationer. 3. Brugeren indtaster informationer. 4. Systemet validerer informationer. 5. IF NOT (Vis sammenstykkeerklæring (UC_9)) <ol style="list-style-type: none"> 4.1 Systemet sætter personfølsom information til anonym. 6. Systemet sender brugeren til booking bekræftelse (UC_8) 	
<ol style="list-style-type: none"> 3a. Hvis informationerne ikke er valide <ol style="list-style-type: none"> 1. Systemet informerer hvilken information der ikke er valid. <i>Systemet sender useren til (1).</i> 	
Version: 1.0	Forfatter: Alexander, Kamilla, Elias, Mads

Tabel 10: fully dressed use case uc_7



Figur 3: viser SSD over UC_7 – Indtast booking oplysninger

Database struktur

E-R diagram

I det følgende afsnit udarbejdes databasestrukturen ved at anvende E-R modellen og normaliseringsreglerne.

E-R modellen bidrager med et overblik over de forskellige entities og deres tilsvarende relationer og kardinaliteter. Til at organisere og strukturere tabellerne korrekt har vi anvendt de tre normaliseringsregler og look-up tables¹⁰. Resultatet af dette er som følgende:

Som det fremgår af E-R diagrammet på næste side, havde vi en mange – mange relation mellem `tbl_inventory` og `tbl_room`. Dette ses ved den implementerede mellemtabel `tbl_roomInventory`. Med mellemtabellen håndterer vi, at mange lokaler kan have meget inventar, og at meget inventar kan findes i mange lokaler.

I forhold til relationen mellem `tbl_userEmail` og `tbl_adminLogin` ses en kardinalitet 1:1. Kardinaliteten ses meget sjældent, og medfører, at de to entities har den samme primary key. Vi kunne godt have valgt at sammensætte de to tabeller i `tbl_userEmail`. Grundlaget for at lade vær var, at en user i vores system altid har en email, men ikke nødvendigvis et login – kun admins har et login. Herved er hvert admin login (één login pr. admin) koblet op på tilhørende admin-email.

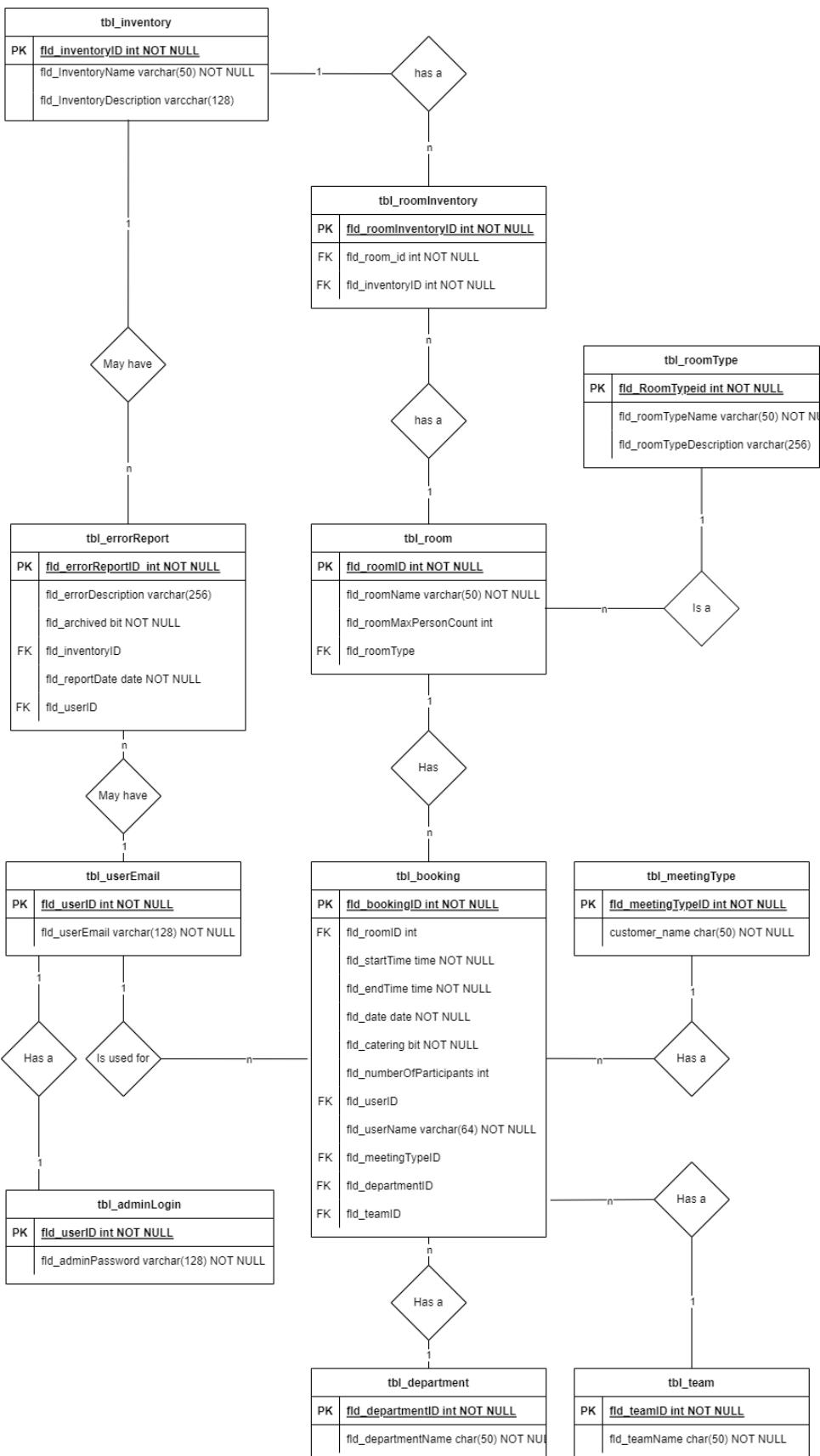
Vores `tbl_inventory` er resultatet af en normalisering efter 1. normalform. Ved at lægge inventardata i en anden tabel end `tbl_room` undgår vi repeating groups, og derved at inventar kan optræde et variabelt antal gange alt efter lokale.

For at overholde 2. normalform har vi sørget for at oprette de entities, der var nødvendige, for at alle ikke-nøgle-attributter i en entity er afhængige af samme primary key.

Til at forenkle dataindtastning og minimere fejl, har vi valgt at anvende look-up tables. Herved forud defineres fx mødetype i `tbl_meetingType` og holdnavn i `tbl_team`, så der tages højde for blandt andet stavfejl og forskellige måder at skrive det samme på.

Slutteligt er det værd at nævne, at vi i tabellerne `tbl_errorReport` og `tbl_booking` har attributter med datatypen ”bit”. Dette er tiltænkt håndtering af booleans fra det færdige system hvor 1 = true og 0 = false.

¹⁰ <https://kyligence.io/plp/understanding-basics-lookup-table-database-management/>



Billede 13: Endelige E-R diagram (version 1.0) med alle entities og deres tilhørende attributter repræsenteret

Kernearkitektur

Vores kernearkitektur er en afgørende del af vores proces, hvor vi fastlægger vores grundlæggende principper. Det er vigtigt at etablere denne arkitektur, før vi går videre til Construction fasen.

Nogle af disse beslutninger træffes på baggrund af vores use cases, mens andre baseres på erfaringer, der er opnået gennem studiet, såsom vores Branch Strategy for GitHub.

Software Arkitektur:

Systemet tiltækkes at være opbygget omkring en 6 lags arkitektur, med følgende lag:

1. View
2. Controller
3. Model
4. Persistence
5. Service
6. Database

Vi bestræber os på at overholde denne arkitektur så præcist som muligt. Konceptet er enkelt: Jo længere ned i systemet vi kommer, desto mere fokus lægger vi på genbrug af kode. Dette er et mål, som vi bestræber os efter at opnå.

Afhængigt af processen og behovene i construction fasen, kan det være nødvendigt at tilføre yderligere lag. I så fald vil disse lag tilføres for at bevare overblikket over vores system, og derved have nemmere ved at gruppere relevante klasser.

Et eksempel kunne være vores databaselag, som i så fald kommer til at indeholde vores JDBC.

JDBC vil i dette tilfælde stadig være genbrugelig og som udgangspunkt ikke have en effekt på genbrugelighed, kun læsbarhed.

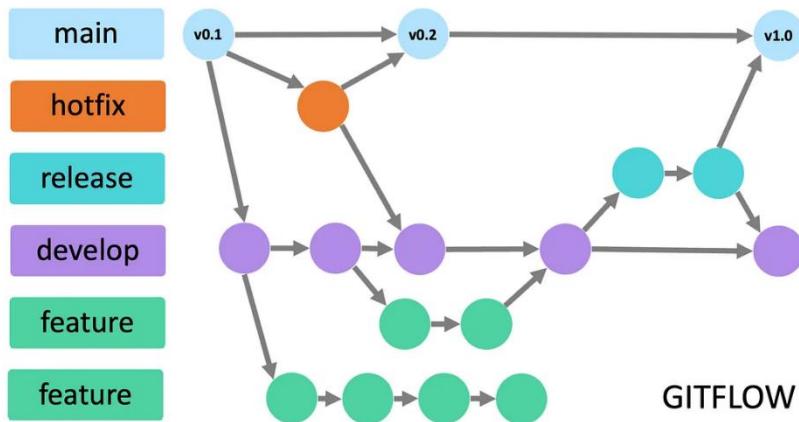
JDBC

For at håndtere kommunikationen imellem ATHENA systemet og databasen, har vi valgt at gøre brug af JDBC api'en¹¹ fra Oracle., som en del af kernearkitekturen. For at sikre, at udviklingsmiljøet er korrekt opsat, er dette sket forud for den egentlige Construction fase.

¹¹ <https://docs.oracle.com/javase/tutorial/jdbc/overview/>

Git & GitHub

Branch Strategy - GITFLOW



Billede 14: diagram over Git flow strategi.

Vi har brugt GITFLOW som vores branching strategi i forhold til Git & Github.

Her lægges der vægt på at vi har følgende branches;

- **Main:** Som altid er en virkende version af vores software (Product Ready).
- **Development:** En forholdsvis virkende version, dog med til tider mangler for ‘fuld funktionalitet’ og et sted hvor vi implementerer vores features.
- **Feature:** En specifik features som er under udvikling.
- **Release:** Forberedende arbejde til at merge ind i main.
- **Hotfix:** Små fixes der er nødvendige for at main og Dev respektivt virker som forventet.

I vores tilfælde vil vores feature branches være bundet op til Use Cases. Intentionen er, at vi vil have en lettere måde at bidrage til hinandens arbejde på, mens vi indskrænker vores branches til kun, at fokusere på mindre områder.

Yderligere, da projektets tidshorisont er fastlåst til et par uger, vil vores Main være noget vi uploader til på en ugentlig basis. Ideen vil stadig være at have noget der virker - vi har dog ikke i sinde, at Main skal stå som et selvstændigt produkt ud over ved aflevering.

Release vil være en “fredags-branch”, hvor fokus vil være på at lave mindre fixes for at få ugens arbejde helt på plads.

Hotfix, vil blive brugt som beskrevet af GITFLOW, og er derfor ikke relevant for vores eksamsprojekt. Vi vælger dog at bibeholde den for fleksibilitetens skyld.

Navngivningskonventioner

Branches

- **Main :** ‘main’
- **Development :** ‘development’
- **Feature :** ‘feature/{usecase-Id}-{kort beskrivelse}’
Eksempel: feature/UC-1-Vis-Reserved-Lokaler
- **Release :** ‘release/{version-nummer}’
Eksempel : release/1.0.0
- **HotFix :** ‘hotfix/{ kort beskrivelse }’
Eksempel : hotfix/vælg-tid-bug

Branch navne skal være korte, men beskrivende nok til at formidle deres formål. Derfor har vi valgt at bruge vores use cases. Ikke nok med at det giver os en rød tråd i forhold til vores design dokument, så formåede det at holde vores scope i check for den respektive branch. Man burde ikke arbejde ud over sin use-case i sin branch.

Commits - Conventional Commits¹²

En fuld implementering af Conventional Commits er lidt uden for scoop for os lige nu.

Men vi har valgt at gå med en ‘lite’ version, der følger den uden skærpet krav i forhold til formatering i commits:

1. **Type:** Angiver formålet med commit.
2. **Scope:** Angiver den del af kodebasen, der er påvirket, vi har i sinde at bruge Class navne hvor muligt.
3. **Subject:** En kort beskrivelse af ændringen.
4. **Body:** Valgfri, detaljeret forklaring af ændringen.

Format:

Formatet for et commit er som følgende:

```
type(scope): subject  
Body
```

¹² SOURCE: <https://www.conventionalcommits.org/en/v1.0.0/>

Typer:

- **feat:** En ny funktion
Eksempel: *feat(timeSelector) : Added ability to select time.*
- **fix:** En fejlrettelse
Eksempel: *fix(timeSelector) : Changed class variable time from double to int.*
- **docs:** Dokumentation Ændringer
Eksempel: *docs(readme) : Updated readme to include new dependencies.*
- **style:** Kodestil ændringer (formatering, manglende semikolon osv.)
Eksempel: *style(timeSelector) : semikolon added after method loop.*
- **refactor:** Kodeændringer, der hverken retter en fejl eller tilføjer en funktion
Eksempel: *refactor(timeSelector) : rewritten method for datetime.*
- **perf:** Ydelsesforbedringer
Eksempel: *perf(timeSelector) : removed unnecessary loop.*
- **test:** Tilføjelse eller opdatering af tests
Eksempel: *test(JDBC) : Added test for JDBC connection.*
- **chore:** Ændringer i build-processen eller hjælpeværktøjer og biblioteker
Eksempel: *chore(dependency) : updated JavaFX from 18 to 21.*

Fuldt eksempel på et commit:

feat(database): improve JDBC connection handling

- Implemented connection pooling to enhance performance and resource management.
- Replaced direct DriverManager calls with a DataSource.
- Updated database configuration in application.properties.
- Added logging for connection lifecycle events.
- Refactored existing database access code to use the new connection handling approach.

Design

Til udarbejdelsen af systemet tiltænker vi at anvende SOLID- og Grasph principperne. Ved at følge disse principper sikrer vi fleksibel- og skalerbar arkitektur, der fremmer maintainability og muliggør udvidelse af systemet.

Ydermere tiltænker vi at anvende custom components til at opnå det tænkte visuelle design og kunne genbruge dette.

Vi vil implementere Public Subscribers til kommunikation mellem vores controllere og bruge builder patterns i relation til dette.

Vi vil anvende DAO pattern, som muliggør CRUD-operationer til databasen ved hjælp af JDBC, og Stored procedure til SQL logik.

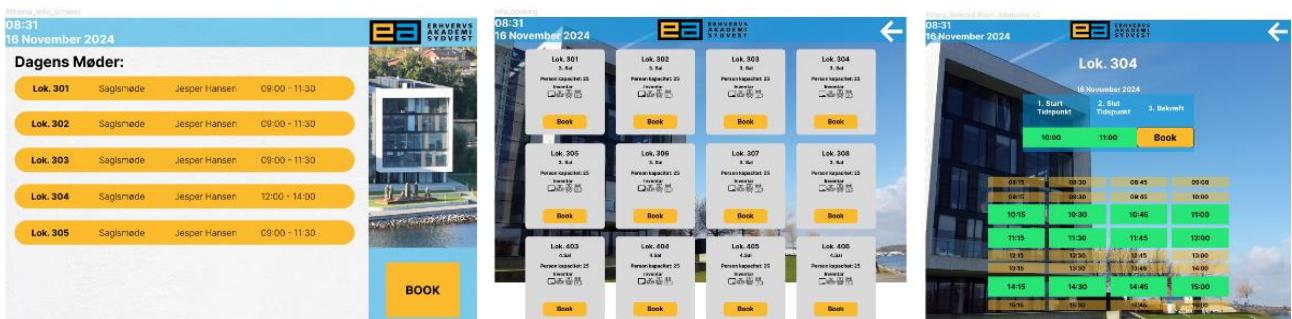
Disse designvalg vil alle blive nærmere beskrevet i construction fasen, når de er blevet implementeret i vores kode.

Prototype – Usability Test

Under udviklingen af systemet har vi udviklet en prototype af systemets infoskærm (Se bilag X). Dette har vi valgt for at identificere og rette eventuelle fejl og misforståelser inden for user experience (UX) og user interface (UI) tidligt i processen. Desuden er prototypen blevet brugt til at teste et diskuteret emne i gruppen – om der skal være et billede af skolen som baggrund på vores bookingsider.

Vi har valgt at teste vores prototype i kundens miljø og med personer, som vi forventer vil blive brugere af systemet. Vi anvendte "Think aloud"-metoden, hvor testpersonerne blev bedt om at tænke højt under hele processen. Dette gav os direkte indsigt i nogle af vores stakeholders' tanker og holdninger samt deres interaktion med prototypen.

Vi fandt tre testpersoner, som skulle teste vores prototype. De blev stillet opgaven: "Book lokale 304 i tidsrummet 10:00 – 11:00." Sluttligt blev de præsenteret for to udgaver af den samme bookingside – én med skolen som baggrund og en med hvid neutral baggrund. Her blev de bedt om at udvælge den version, de bedst kunne lide, og efterfølgende beskrive, hvorfor valget faldt på netop den.



Billede 15: viser uddrag af vores prototype
(se bilag 3 for den fulde prototype og større visning).

Analyse af Usability Testen

For at forstå resultaterne af vores usability test, gennemgik vi feedbacken fra testpersonerne og analyserede deres oplevelser og kommentarer.

Testpersoner:

Multimediedesigner (4. semester)

Anders (lærer)

Jacob (datamatiker, 4. semester)

Positivliste:

- Prototypen er intuitiv
- Testpersonerne havde nemt ved at navigere
- Testpersonerne gik ”de rigtige veje”
- Korrekt valg af tider

- Kasseopbygning af programmet fungerer godt for indtastning af information
- Baggrund med billede fungerer godt, hvis kasser er mindre gennemsigtige
- Advarsel ved fejl er en god funktion

Negativliste:

- Symboler/ikoner for inventar er ikke tydelige – kunne ikke afkodes
- Manglende overblik over bookede lokaler og inventar under booking
- Baggrundsbilledet på prototypen fjerner fokus fra datafelter
- Forvirring over ADHOC og begrænset mulighed for at booke flere dage
- Forvirrende lokaloversigt
- Usikkerhed omkring PIN-kode funktionalitet – er det til døren? Hvor bruger man den?
- Inkonsistent farvebrug på knapper

Overraskelser:

- PIN-kode er overflødig uden digital kvittering (e-mail)
- "Indtast bookinginformationer" mangler opsummering af lokalet, inventar og pladser
- "Dagens Møder" er forvirrende, da brugerne forventer at kunne klikke på et møde for at booke det pågældende lokale
- Foretrækker listevisning over "flot" design i lokaloversigten

Analyse af fund

Generelt viste testene, at prototypen fungerer som forventet og at den er intuitiv, men at der er mangler i forhold til UX-design og brugervenlighed. Vores fund indikerer, at "info-view" og "lokalebooking-view" er opsat på en forvirrende måde og mangler nødvendige informationer. Der var en tendens til, at brugerne forsøgte at bruge "info-view" til at booke lokaler, hvilket ikke var hensigten.

Til trods for enkelte problemer, var der positiv feedback omkring det overordnede UX-design. Dette er noget, vi vil fastholde og videreudvikle i det endelige produkt. Med undtagelse af pin-koden, som blev kritiseret som

overflødig, fandt vi, at den negative og overraskende feedback primært var lokalt isoleret til Ad-Hoc-funktionerne og derfor ikke vil have den store påvirkning på resten af programmet. Vores PIN-kode-funktionalitet bør dog genovervejes.

Feedbacken om brugen af billeder som baggrund viste sig også at være værdifuld. Vi valgte at beholde billedet af skolen, da det harmonerer godt med brugen af de firkantede "informationsbokse".

Anbefalinger

På baggrund af testresultaterne kan vi nu forbedre designfasen af vores system:

1. Behold det firkantede design.
2. Sørg for, at information og ikoner hænger sammen, da ikoner alene ikke altid er forståelige.
3. Det trinvise design til AD-HOC booking virker godt og bør fastholdes.
4. Sikr konsistens i farver og knapper for at guide brugeren bedre gennem processen.
5. Programmet bør afspejle skolens farvevalg for en bedre brugeroplevelse.

Disse indsigtter vil hjælpe os med at raffinere vores prototype og sikre et mere brugervenligt og effektivt system.

Construction fasen

I det følgende vil de enkelte designvalg blive beskrevet med eksempler fra vores system.

Custom Component

Vi har gjort brug af en del custom components i projektet

Et eksempel er vores BookingComp:

```

public class BookingComp extends HBox { 10 usages ± Bolt404 +1
    □ 5 ✓ 3 ▲ ▼
    private void generateComp (String roomName, String userName, String meetingType, String startAndEndTime){ 2 usage
        Label labelBookingRoom = buildLabel(roomName,   fontSizE: 18, FontWeight.BOLD);
        labelBookingRoom.setPrefWidth(100);

        Label labelBookerName = buildLabel(userName,   fontSizE: 18, FontWeight.NORMAL);
        labelBookerName.setPrefWidth(200);

        Label labelBookingName = buildLabel(meetingType,   fontSizE: 18, FontWeight.NORMAL);
        labelBookingName.setPrefWidth(150);

        Label labelBookingTime = buildLabel(startAndEndTime,   fontSizE: 18, FontWeight.NORMAL);

        // setting booking comp color.
        setBookingCompColor(BookingCompColors.NORMAL);

        // Setting this in regard to how we want it to look.
        this.setMinHeight(35);
        this.setAlignment(Pos.CENTER_LEFT);
        this.setStyle("-fx-background-color: "+ this.bookingCompColor.getColor() + "; -fx-background-radius: 40");
        this.getChildren().addAll(labelBookingRoom, labelBookingName, addPane(), labelBookingTime);
    }
}

```

Billede 16: BookingComp.java - src/main/java/org/apollo/template/View/UI/BookingComp.java

som skaber en HBox, der allerede er fyldt med vores information og styling. Dette muliggør, at vi kan skabe en UI til vores users med de relevante informationer.

Nedenfor ses et eksempel på implementeringen af BookingComp:

```

for (Booking i : bookingList) {
    //Getting our Variables sorted.
    String time = getStringTimeFormated(i);
    String roomName = i.getRoom().getRoomName();
    String userName = i.getUsername();
    String meetingType = i.getMeetingType().getMeetingTypeName();
    int bookingId = i.getBookingID();

    BookingComp bookingComponent;
    LoggerMessage.trace( instance: this,   message: roomName + " | " + userName + " | Adding to vbox");
    |
    if (bookingId == -1){
        LoggerMessage.debug( instance: this, message: "Making BookingComp without ID");
        bookingComponent = new BookingComp(roomName,userName,meetingType,time);
    } else {
        LoggerMessage.debug( instance: this, message: "Making BookingComp with ID: " + bookingId);
        bookingComponent = new BookingComp(roomName,userName,meetingType,time,bookingId);
    }
}

```

Billede 17: ReservedRoomsVBox.java - src/main/java/org/apollo/template/View/UI/ReservedRoomsVBox.java

Som det ses på eksemplet ovenfor, er det muligt at kalde og genere mange components ud fra en Liste. Dette gør vi brug af en del gange i systemet.

Vi fandt hurtigt ud af, at brugen af components har stort potentiale i et projekt som dette, men vi kan også se, at vi har en del kode, der kunne genbruges. Dette er højest sandsynlig på grund af manglende erfaring eller fordi vi ikke gik nok i dybden under designfasen.

Vi fandt en elegant løsning på dette, men manglede tid til at implementere denne fuldt ud. Den bygger på en default Component, som man bygger videre på ved at extende fra den.

Konceptet er, at vi i dette tilfælde har en HBox, hvor vi styler den, for herefter at bygge videre på den.

Se billede på næste side:

```
1 package org.apollo.template.View.UI;
2
3 > import ...
11
12 @ public abstract class DefualtComponent extends HBox { 11 usages 4 inheritors
13
14     private CompColors compColors; 2 usages
15     private final int FONT_SIZE = 18; 1 usage
16     private CompColors color = CompColors.NORMAL; 1 usage
17
18     >     public DefualtComponent() { styleHbox(this); }
19
22     /**
23      * Method for styling a label
24      * @param label Label
25      */
26 @ >     public void styleLabel(Label label){...}
27
28     /**
29      * Method for styling a Hbox
30      * @param hbox Hbox
31      */
32
33 @ >     public void styleHbox(HBox hbox){...}
34
35     >     public CompColors getCompColor() { return compColors; }
36
37
38 @ >     public void setCompColor(CompColors compColors) {...}
39
40
41
42
43
44
45     >     public CompColors getCompColor() { return compColors; }
46
47
48
49 @ >     public void setCompColor(CompColors compColors) {...}
50
51
52
53 // endregion
54
55 }
```

Billede 18: DefualtComponent.java, her ses vores default component som nedarver fra HBox

```

1  package org.apollo.template.View.UI;
2
3 > import ...
5
6
7 public class RoomComp extends DefaultComponent{ 5 usages
8
9     private Room room; 4 usages
10
11 >     public RoomComp(Room room) {...}
15
16 >     private void loadRoom(){...}
18
29 <     // region getter & setter
30 >     public Room getRoom() { return room; }
33
34     // endregion
35 }
36

```

billedet 19 - RoomComp.java

Her ses nedarvning fra DefaultComponent. Componenten er lavet specifik til at visualisere et Room Object. Kort sagt foderer vi den med et data holdende Object, som hedder Room. Ud fra dette laver vi vores component. Her bruger vi style fra Default Componenten og lægger den nødvendige logik og funktionalitet ind.

På den måde vil vi kunne genbruge koden i stor stil. Havde vi implementeret denne måde, at lave vores Componentens på, så ville vi også kunne have sparet en del tid.

Builder Pattern

Vi har især gjort brug af Builder Pattern sammen med public subscriber pattern i ad-hoc delen af vores program. Dette var meget aktuelt, da vi ønskede at opbygge et dataobjekt over tid, for derefter at kunne sende dette objekts data til databasen. Eksempler:

```

public class AvailableRoomsController implements Initializable {

    @FXML
    private VBox vbox_Listview;

    private Booking booking = new Booking(); 4 usages

```

Billedet 20: AvailableRoomsController.java, her skaber vi vores objekt "Booking".

```

public class AvailableRoomsController implements Initializable {
    private void button_bookOnAction(Button button_book, Room availableRoom) { 1 usage
        button_book.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent actionEvent) {

                // create booking information object
                booking.setRoom(availableRoom);
                booking.setAdHoc(true);

```

Billede 21: AvailableRoomsController.java, her gøres brug af builder pattern.

```

public class ChooseTimeController implements Initializable, Subscriber {
    protected void onButton_book(){
        // Sets the start- and endtime of the booking object
        booking.setStartTime(Time.valueOf(startTime));
        booking.setEndTime(Time.valueOf(endTime));
        // Sets the date of the booking object
        LocalDate currentDate = LocalDate.now();
        booking.setDate(Date.valueOf(currentDate).toLocalDate());

```

Billede 22: ChooseTimeController.java, her bygges videre på booking objectet.

Som det ses, har vi mulighed for at bygge objektet op over tid, i stedet for at skulle bygge hele objektet på én gang i en constructor. Dette er ekstremt brugbart i et program som vores, hvor brugere løbende får mange valg.

```

public class Booking { 1 inheritor

    // this is an empty constructor, due to this class being a builder pattern.
    public Booking() {}

    public Booking setBookingID(int bookingID) { 5 usages
        this.bookingID = bookingID;
        return this;
    }

    public Booking setEmail(Email email) {
        this.email = email;
        return this;
    }

    public Booking setNumberOfParticipants(int NumberOfParticipants) { 5 usages
        this.NumberOfParticipants = NumberOfParticipants;
        return this;
    }

```

Billede 23: Booking.java, her ses vores Booking Object, hvor vores set-metoder returnerer 'this', altså 'Booking'.

For at kunne bygge vores objekt med setters, er man nødt til at returnere objektet. Dette muliggør også, at man kan bygge det i én sekvens med .setData. For læsbarhedens skyld har vi dog undgået dette i vores ad-hoc del af programmet. Hvis vi gjorde det, kunne det se sådan ud:

```
booking.setStartTime(Time.valueOf(startTime)).setEndTime(Time.valueOf(endTime)). setDate(Date.valueOf(currentDate).toLocalDate());
```

Billede 24: Eksempel på BuilderPattern. Her vises, hvordan Builder Pattern kan bruges i praksis.

SOLID

Hvis vi tager DAO som et eksempel, gør den brug af alle dele af SOLID-principperne:

- **Single Responsibility Principle (SRP):** DAO har kun ansvar for at håndtere databaseoperationer for en bestemt tabel i databasen, hvilket betyder, at den har én specifik opgave og gør dette godt.
- **Open/Closed Principle (OCP) og Liskov Substitution Principle (LSP):** Ved at benytte interfaces til vores DAO kan vi nemt udvide funktionaliteten uden at ændre den eksisterende kode. Samtidig sikrer vi, at alle implementeringer af interfacet opfylder de forventninger, vi har til vores DAO.
- **Interface Segregation Principle (ISP):** Vores interface er begrænset til CRUD-operationer. Hvis der er behov for yderligere funktionalitet, kan vi oprette nye interfaces, der inkluderer disse nye områder, hvilket betyder, at klienter kun behøver at kende til de interfaces, der er relevante for dem.
- **Dependency Inversion Principle (DIP):** DAO'en afhænger af abstraktioner (interfaces) snarere end konkrete implementeringer. Dette gør systemet mere fleksibelt og testbart, da vi kan udskifte implementeringer uden at påvirke de afhængige klienter.

Med disse principper implementeret, kan vores DAO-arkitektur nemt udvides og vedligeholdes, samtidig med at den forbliver robust og pålidelig.

DATABASE

Database scripts

For at kunne konstruere databasen, udarbejdede vi et database creation script baseret på det ER-diagram, der blev udarbejdet i den tidlige elaboration fase. Ved at anvende et sådant script sikrer vi, at alle gruppemedlemmer har præcis den samme databasestruktur.

Dette muliggør også nemme tilpasninger, hvis der af en eller anden grund opstår fejl eller der kræves ændringer i databasen under udviklingen. Se hele database creation scriptet i bilaget (Bilag 6).

Derudover har vi anvendt et insert script til at tilføje testdata, som systemet skal bruge for at kunne køre. Ved at skabe dette som et script sikrer vi endnu engang, at alle gruppemedlemmer har den samme datasæt at teste på. Samtidig er der udviklet et script til at tilføje alle stored procedurer, så alle har de nødvendige stored procedurer til rådighed. Disse to scripts kan ses i deres helhed i bilagene (Bilag 7 og Bilag 8).

DAO

Vi har delvist brugt Data Access Objects (DAO) i vores system, hvor det har givet mening. Når der har været behov for større logik i forhold til dataudtræk, har vi brugt stored procedures. Da DAO'er primært håndterer CRUD-operationer, har det ofte været problematisk at finde en god måde at implementere dem på, især når vi har haft behov for at sortere data. Dette er langt nemmere at gøre gennem stored procedures i stedet for en kombineret DAO- og Java-logikløsning.

En af fordelene ved DAO, er at man derved kan hæve abstraktionsniveauet. Derved følger vi Liskov substitution princip (LSP). Dette giver et bedre overblik, og en mere genbrugelig kode base. Dette gør også systemet mere skalerbart.

Efter vores implementering af DAO'er har vi dog indset, at vi kunne have brugt INNER JOIN for at skabe mere komplette dataobjekter. Dette har vi ikke gjort, fordi vi havde en forståelse for at DAO kun bestod af tabel til model relationer. Vi kan dog konkludere, at dette kunne gøres uden at forvirre eller på anden måde bryde DAO-pattern. Faktisk ville det gøre vores kode mere læsbar og nemmere at vedligeholde. Denne misforståelse har resulteret i mindre brug af DAO'er.

Et eksempel på en af vores implementeringer:

```

package org.apollo.template.persistence.JDBC.DAO;

import java.util.List;
public interface DAO<T>{ 7 implementations

    void add(T t); 7 implementations

    void delete(T t); 7 implementations

    void update(T t); 7 implementations

    T read(int id); 2 usages 7 implementations
    List<T> readAll(); 7 implementations

}

```

Billede 25: DAO.java, DAO Interface

DAO implementeres som et interface for at holde en lav kobling mellem klassens DAO og objektet, der ønsker at bruge den. Dette er også et godt eksempel på GRASP-princippet: Low Coupling.

```

public abstract class DAOAbstract<T> implements DAO<T>{ 7 usages 7 inheritors
    // reference to connection obj
    private final Connection CONN = JDBC.get().getConnection(); 1 usage

    /**
     * Method for fetching the connection obj
     * @return Connection
     */
    public Connection getCONN() { return CONN; }

    /** Method for closing a PreparedStatement, Checks for obj null ...*/
    public void closePreparedStatement(PreparedStatement ps) {...}

    /** Method for closing a ResultSet, Checks for obj null ...*/
    public void closeResultSet(ResultSet rs) {...}

}

```

billede 26 - DAOAbstract.java

Mellem vores Interface og DAOs har vi så valgt at have en abstract klasse, som holder på funktionalitet i forhold til vores connection. Dette er for at undgå gentagende kode samt for at give øget funktionalitet i forhold til connectionen.

Her ses et godt eksempel på brug af SRP, hvor klassen har ansvar for database connection.

```
    } finally {
        closeResultSet(rs);
        closePreparedStatement(ps);
    }
```

Her ses de i brug, hvor vi lukker vores preparepet statement og resultset efter en try-catch.

```
public class InventoryItemDAO extends DAOAbstract<InventoryItems> { 10 usages
    @Override
    public List<InventoryItems> readAll() {
        PreparedStatement ps = null;
        ResultSet rs = null;
        List<InventoryItems> inventoryItemsList = new ArrayList<>();
        try {
            ps = getCONN().prepareStatement( sql: "SELECT * FROM tbl_inventory");
            rs = ps.executeQuery();

            while (rs.next()) {
                int inventoryID = rs.getInt( columnLabel: "fld_inventoryID");
                String inventoryName = rs.getString( columnLabel: "fld_inventoryName");
                String inventoryDescription = rs.getString( columnLabel: "fld_inventoryDescription");

                inventoryItemsList.add(new InventoryItems(inventoryID, inventoryName, inventoryDescription));
            }
        } catch (SQLException e) {
            LoggerMessage.error( instance: this, message: "IN READALL; an error occurred: " + e.getMessage());
        } finally {
            closeResultSet(rs);
            closePreparedStatement(ps);
        }
        return inventoryItemsList;
    }
}
```

Billede 27: InventoryItemDAO.java, readAdd method

Overstående metode viser, at der extendes vores DAOAbstract klasse, metoden gør brug af en prepareStatement som holder vores SQL, og returer et resultset som en list.

Dette er et godt eksempel på Open / closed princippet, fordi metoder fra den abstrakte klasse kan overrides og tilpasses til brugsmønsteret.

Yderligere giver det os mulighed for at følge DRY princippet (Don't repeat yourself).

Plus vi har den yderligere funktionalitet med den abstrakt klasse, som gør at vi ikke skal genskrive den samme kode igen og igen.

```
public class InventoryItems {  
  
    private int id; 5 usages  
    private String name; 7 usages  
    private String description; 4 usages
```

Billede 28: InventoryItems.java, Vores data object

```
// creates inventory item obj, and stores it in the database.  
InventoryItems = new InventoryItems(textField_itemName.getText(), textArea_itemDescription.getText());  
DAO<InventoryItems> dao = new InventoryItemDAO();  
dao.add(inventoryItems);
```

Billede 29: CreateInventoryItemController.java

Ovenstående viser implementeringen i Java Logik.

Store procedures

I vores applikation anvender vi flere Stored Procedures (sProcedures). En sProcedure er en navngivet samling af SQL-udtryk, som kan eksekveres sammen. De bruges ofte til at forbedre sikkerhed og ydeevne.¹³

Fordelen ved sProcedures er, at de tilbyder fleksibilitet og modellering i forhold til DAO, der primært omhandler CRUD-operationer. Med sProcedures kan vi nemt nesse flere forskellige databaseoperationer i én søgning og endda udføre beregninger. Derudover udføres en sProcedure direkte i databasen, hvilket resulterer i hurtigere eksekveringstider. En anden fordel ved sProcedures er sikkerhedsaspektet. Ved at gemme SQL-koden bag en databaseforbindelse forhindrer vi, at koden eksponeres i applikationen eller bliver tilgængelig for uautoriseret adgang.

Stored procedure – eksempel 1:

En af de sProcedures vi benytter i vores applikation er getAvailableRooms. Denne sProcedure tager en booking dato som parameter og returnerer et resultatsæt bestående af alle ledige rum (ikke fuldt bookede) på den givne dato.

¹³ <https://hightouch.com/sql-dictionary/sql-stored-procedures>

SProceduren starter med at beregne den samlede bookingtid pr. rum. Dette gøres ved at udregne antal minutter mellem fld_startTime og fld_endTime for hver booking på den specificerede dato. Resultatet gemmes som et midlertidigt resultatsæt (BookedTime), som indeholder felterne fld_roomID og total_booked_time_minutes.

Efter denne operation listes de felter, vi ønsker at vælge, og fra hvilke tabeller. For at få informationer om rumtypen (roomType), laver vi en INNER JOIN mellem tabellerne tbl_room og tbl_roomType. Herved kombinerer vi de to tabeller baseret på fld_roomTypeID. Vores midlertidige resultatsæt BookedTime kombinerer vi med tbl_room ved hjælp af en LEFT JOIN baseret på fld_roomID. Dette sikrer, at alle rum medtages, også selvom de ikke har nogen bookings på den specificerede dato.

Resultaterne af vores selection bliver herefter filtreret på baggrund af hvert rums total_booked_time_minutes. Hvis et rum har en total bookingtid på mindre end 480 minutter (8 timer) eller den er NULL (dvs. ingen booking), anses rummet som værende ledigt. Herved bliver rummet en del af det endelige resultatsæt, som slutteligt sorteres i stigende rækkefølge efter rumnavn.

```
-- Stored procedure to find available rooms on a given date
CREATE PROCEDURE getAvailableRooms (@BookingDate DATE)
AS
BEGIN

    -- Calculates the total booking time for each room on the specified date ('NULL' if no booking yet)
    WITH BookedTime AS (
        SELECT
            fld_roomID,
            SUM(DATEDIFF(MINUTE, fld_startTime, fld_endTime)) AS total_booked_time_minutes
        FROM
            tbl_booking
        WHERE
            fld_date = @BookingDate
        GROUP BY
            fld_roomID
    )

    -- Selects selected fields from 3 tables related to each other using INNER JOIN and LEFT JOIN - these are to be presented when displaying available rooms
    SELECT
        tbl_room.fld_roomID,
        tbl_room.fld_roomName,
        tbl_room.fld_floor,
        tbl_room.fld_roomMaxPersonCount,
        tbl_room.fld_roomTypeID,
        tbl_roomType.fld_roomTypeName,
        tbl_roomType.fld_roomTypeDescription
    FROM
        tbl_room
        LEFT JOIN BookedTime ON tbl_room.fld_roomID = BookedTime.fld_roomID
        INNER JOIN tbl_roomType ON tbl_room.fld_roomTypeID = tbl_roomType.fld_roomTypeID

```

Billede 30: første del af Stored procedure "getAvailableRoom" – sidste del findes på næste side

```

-- Filters the results to include only rooms that are either not booked on the specified date or have a total booked time less than 8 hours (480 minutes)
WHERE
    BookedTime.total_booked_time_minutes < 480 OR BookedTime.total_booked_time_minutes IS NULL

-- Groups the results by five columns to ensure that we only see each room once in the list, even if there are multiple bookings on the selected day.
GROUP BY
    tbl_room.fld_roomID,
    tbl_room.fld_roomName,
    tbl_room.fld_floor,
    tbl_room.fld_roomTypeID,
    tbl_roomType.fld_roomTypeName,
    tbl_roomType.fld_roomTypeDescription,
    tbl_room.fld_roomMaxPersonCount

-- The available rooms are sorted by room name (ascending) and start time (ascending)
ORDER BY
    tbl_room.fld_roomName ASC;

END;

```

Billede 31: Stored procedure "getAvailableRoom", der returnerer et resultatsæt med alle ledige lokaler på den give dato.
Taget fra [src/main/java/org/apollo/template/JDBC/databasseInsert.sql](#)

Denne sProcedure kaldes i JAVA-metoden getAvailableRooms i GetAvailableRooms ved hjælp af en PreparedStatement. Den sætter bookingdatoen som parameter og udfører forespørgslen, hvilket returnerer et ResultSet (som længere nede i metoden bliver behandlet).

```

public static List<Room> getAvailableRooms(Date dateToday){

    List<Room> availableRooms = new ArrayList<>();

    try {
        PreparedStatement ps = JDBC.get().getConnection().prepareStatement(sql: "EXECUTE getAvailableRooms @BookingDate = ?");
        ps.setDate(parameterIndex: 1, dateToday);
        ResultSet rs = ps.executeQuery();
    }
}

```

Billede 32: Kald af Stored Procedure getAvailableRooms i JAVA-metoden getAvailableRooms.
Taget fra [src/main/java/org/apollo/template/persistence/JDBC/StoredProcedure/GetAvailableRooms.java](#)

Stored procedure – eksempel 2:

Formålet med denne sProcedures, AddEmailIfNotExists, er at checke om en e-mail eksisterer i tbl_userEmail. Hvis den ikke eksisterer, oprettes den. Fordi dette sker som en sProcedure, slipper vi for at spørge databasen, modtage et svar og derfra handle som det vil være tilfældet hvis dette forgik igennem en DAO.

```

CREATE PROCEDURE AddEmailIfNotExists
@EmailAddress NVARCHAR(255)
AS
BEGIN
    -- Check if the email already exists
    IF NOT EXISTS (SELECT 1 FROM tbl_userEmail WHERE fld_userEmail = @EmailAddress)
        BEGIN
            -- Insert the email into the table
            INSERT INTO tbl_userEmail(fld_userEmail)
            VALUES (@EmailAddress);

            -- Indicate success
            SELECT 'Email added successfully' AS Result;
        END
    ELSE
        BEGIN
            -- Indicate that the email already exists
            SELECT 'Email already exists' AS Result;
        END
END;

```

Billede 33: Her ses en store procedure, der indsætter en email i tabel tbl_userEmail, hvis den ikke allerede eksistere. Taget fra `src/main/java/org/apollo/template/Database/databasseInsert.sql`

Denne sProcedure kaldes i JAVA-metoden `onButton_book` i `BookingInformationController` ved hjælp af en `PreparedStatement`. Den sætter emailen som argument og udfører forespørgslen.

```

Method for adding a email to the database, checks if it already exists before adding it.

Params: email – Email

public static void addEmailIfNotExists(Email email) { 1 usage  ✎ Elias Therkildsen
try {
    PreparedStatement ps = JDBC.get().getConnection().prepareStatement( sql: "EXEC AddEmailIfNotExists @EmailAddress = ?");
    ps.setString( parameterIndex: 1, email.getEmail());
    ResultSet rs = ps.executeQuery();
    LoggerMessage.debug( name: "AddEmailIfNotExists", message: "JDBC: " + rs.next());

} catch (SQLException e) {
    throw new RuntimeException(e);
}
}

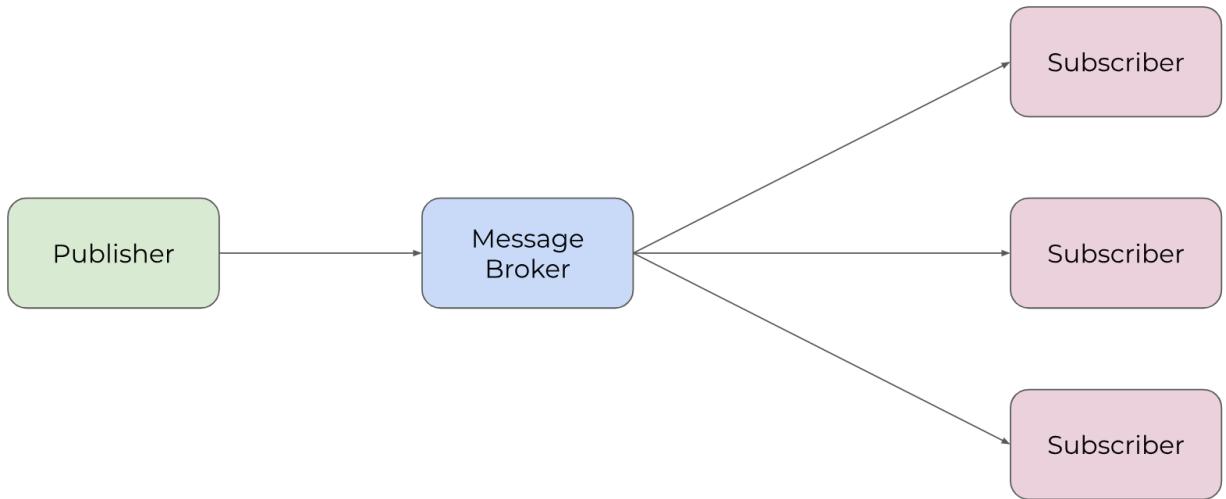
```

Billede 34: Her ses implementeringen af overstående stored procedure. taget fra `src/main/java/org/apollo/template/Database/databasseInsert.sql`

Public subscriber

Vores lokalebooking system gør brug af public subscriber pattern til at flytte objekter imellem controllerne. Dette sker f.eks. med objektet *Booking.java* (src/main/java/org/apollo/template/Model/Booking.java).

En public subscriber består af flere lag som det ses på billedet nedenfor (**Fejl! Henvisningskilde ikke fundet.**). Lagene er som følgende; Publisher, Messages Broker og subscriber.



Billede 35: billedet viser de forskellige komponenter i public subscriber pattern

Formålet med Publisher er at sende værdier eller beskeder til en eller flere subscribers. I vores tilfælde sender vi et objekt til en eller flere subscribers for at dele information mellem forskellige controllerne. Denne overførsel af information sker gennem en Message Broker. Det er denne klasse, der sørger for, at videregive det publicerede til de rette subscribers. Messages Brokerens ansvar er at holde styr på, hvad disse subscribers er interesseret i at vide. For at sikre en løs kobling mellem klasserne implementerer subscribers ofte et interface, der definerer, hvordan de modtager og håndterer meddelelser fra publishers.

I vores system muliggør denne implementering af subscriber-interface, at controllers kan kommunikere uden at bryde med vores 6-layer architecture, hvor vi ønsker nedadgående dependencies.

I vores tilfælde implementerer de interface:

```

public interface Subscriber { 14 usages 3 implementations

    void update(Object o); 3 implementations

}

```

Billede 36: viser et billede af det interface, som vores subscribers implementerer

Som det ses nedenfor, implementerer BookingCompleteController'en interfacet, og ønsker at subscribe på information om booking information (enum: BOOKING_INFORMATION).

```

// subscribing to messages broker for topic BOOKING_INFORMATION
MessagesBroker.getInstance().subscribe( subscriber: this, MessagesBrokerTopic.BOOKING_INFORMATION);

```

Billede 37: viser hvordan BookingCompliteController subscriber til information om BOOKING_INFORMATION-

Herved modtager klassen BookingCompleteController et nyt objekt fra Message Brokeren, hver gang publisher sender ny update om BOOKING_INFORMATION.

Dette sker via metoden ”publish”:

```

Method for publishing an object.

Params: topic - MessagesBrokerTopic
        messages - String

public synchronized void publish(MessagesBrokerTopic topic, Object messages) throws Exception { 11 usages

    // validating that the messages object is not null
    if (messages == null){
        throw new Exception();
    }

    // looping through all pairs in subscriberList
    for (Pair<MessagesBrokerTopic, Subscriber> pair : subscriberList){
        LoggerMessage.debug( instance: this, message: pair.getKey() + " - " + pair.getValue());
        if (topic.equals(pair.getKey())){
            LoggerMessage.trace( instance: this, message: "Object : " + messages.toString());
            pair.getValue().update(messages);
        }
    }
}

```

Billede 38 39: viser publish metoden i vores MessageBroker klasse

hvor Message Brokeren gennemgår en Pair/value liste ud fra det medsendte topic. De som er subscriptet til dette topic, modtager det nye objekt.

Fordi Messages Brokeren bruger GoF-patternet singleton, kan referencen til objektet nås i hele applikationen. Logik for at abonnere og fjerne abonnement på et emne er Messages Brokerens ansvar. Derudover er det, som tidligere nævnt, også dens ansvar at publicere ændringer. Alt dette kan ses på nedenstående billede:

```
public class MessagesBroker {

    private static MessagesBroker instance;  3 usages
    // list of subs and their topic
    private List<Pair<MessagesBrokerTopic, Subscriber>> subscriberList = new ArrayList<>();  5 usages

    // making messagesBroker private, to follow singleton principal.
    private MessagesBroker() {}  1 usage

    /** Method for subscribing to a topic. ...*/
    public synchronized void subscribe(Subscriber subscriber, MessagesBrokerTopic topic){...}
    /** Method for unsubscribing from a topic. ...*/
    public synchronized void unSubscribe(Subscriber subscriber, MessagesBrokerTopic topic){...}
    /** Method for publishing an object. ...*/
    public synchronized void publish(MessagesBrokerTopic topic, Object messages){...}

    public synchronized static MessagesBroker getInstance(){
        if (instance == null){
            instance = new MessagesBroker();
        }

        return instance;
    }
}
```

Billede 40: viser Messages Broker-klassen i vores applikation. Klassen fungerer som en singleton og er ansvarlig for at håndtere abonnementer, offentliggørelse af beskeder og logikken for abonnement og fjernelse af abonnement på emner.

Derudover er keywordet synchronized brugt i metodesignaturen, i Message Borkerens metoder, for sikre at kun en tråd kan få adgang til en metode ad gangen. Dette er Javas svar på en semaphor - den sikrer at der ikke sker nogle datakonflikter og race conditions.

Der er flere fordele ved at benytte sig af Publisher-Subscriber pattern. En af fordelene er skalerbarhed. Dette tillader, at systemet kan vokse uden at miste ydeevne, da publishers og subscribers er uafhængige af hinanden. Det betyder, at hvis vores system skal udvides, kan nye publishers og subscribers tilføjes uden problemer.

Afkobling er en anden fordel, idet beskeder kan sendes og modtages asynkront, uden forsinkelser. Dette gør det muligt for vores controllere at kommunikere effektivt uden ventetid.

Reaktionsevnen er endnu en fordel, da denne arkitektur muliggør realtidsreaktioner, hvor subscribers straks modtager beskeder. Dette sikrer, at opdateringer om foreløbige bookinginformationer hurtigt bliver delt mellem forskellige controllere i systemet.

Endelig er vedligeholdelsesvenlighed også en stor fordel, da ændringer i én komponent ikke påvirker andre. Dette gør det lettere at opdatere og vedligeholde vores system.

Delkonklusion 1. iteration

Under første iteration har fokus været på at udvikle use cases relateret til infoskærmen og ad-hoc bookingdelen af systemet. Denne opdeling har været relevant, da flere use cases i admin bookingdelen indeholder udvidet funktionalitet, som kan nedarves fra infoskærm/ad-hoc.

UI/UX og implementering

Med hensyn til vores UI/UX, har vi fulgt analysen af vores prototype og implementeret de anbefalinger, vi fandt, så grundigt som muligt. Resultatet af ad-hoc bookingdelen følger prototypen og leverer et tilfredsstilende resultat. Vi har lagt særlig vægt på at sikre et sammenhængende flow, der guider brugeren fra start til slut.

Tidsestimering og læringspunkter

Generelt har vores tidsestimering for use cases været præcis, selvom der har været afvigelser. Optimistiske tidsestimeringer har ført til nogle tidsoverskridelser, mens vi i andre tilfælde har overestimeret tidsforbruget. På trods af disse udfordringer har vi overholdt tidsplanen. Erfaringerne fra denne iteration vil vi tage med videre, og vi vil justere tidsestimatorne for den kommende iteration. Derudover vil vi inddrage nye use cases, som er afledt af den første iteration.

Gruppearbejde og samarbejdsstrategier

Gruppearbejdet har fungeret godt, og den demokratiske tilgang har gjort det nemt at håndtere uenigheder og forskellige meninger. Vi supplerer hinanden godt. Vores branching strategi og naming convention for commits har fungeret effektivt, hvilket har givet os et bedre overblik og et mere struktureret arbejdsflow. Det har været muligt at samarbejde omkring branches, da det har været nemt at få og bevare overblikket over, hvilket arbejde der bliver udført og hvornår. Navngivning af branches i relation til use cases har været særligt nyttigt, da det har skabt en klar sammenhæng mellem Unified Process og udført arbejde. En relativt fleksibel developer branch har været et fremragende værktøj til at implementere og rette mindre fejl i forbindelse med merges af feature branches.

Ændringer og justeringer

Vi har fulgt SSD'er, dog med ændringer i forhold til UC7. Dette skyldes nye og bedre tilvalg i forhold til implementering af samtykkeerklæring. Derfor har vi besluttet, at det ikke skal være muligt at oprette en booking, hvis brugeren ikke accepterer TOS.

Construction-fasen

Vi har opnået de mål, vi satte i elaboration-fasen, hvilket understreger, at vores tilgang i construction-fasen har været effektiv.

2. iteration:

I denne iteration lægges der vægt på udviklingen af systemets admin del.

Elaboration fasen

Nedenfor ses vores brief use cases – nogle er reviderede og andre er nytilkomne. De som står med kursiv, blev implementeret i 1. iteration.

ID	Actor	Navn	Prioritet (1-10)	Estimat (timer)
UC_0	User	Vælg system	10	1 Time
UC_1	System, User	Vis reserverede lokaler	10	5 Timer
UC_2	System	Sorter reserverede lokaler (FD) lokale/tid	4	4 Timer
UC_3	System	Vis aktuel tid	1	1 Time
UC_4	System	Vis aktuel dato	5	1 Time
UC_5	User	Vælg starttidspunkt / sluttidspunkt	10	7 Timer
UC_6	System	Vis ledige lokaler	8	5 Timer
UC_7	System, User	Indtast Booking Oplysninger	7	4 Timer
UC_8	System	Vis booking bekræftelse	3	4 Timer
UC_9	System	Vis samtykkeerklæring	7	1 Time
UC_10	User	Svar på samtykkeerklæring	7	1 Time
UC_11	User	Slet booking	2	3 Timer
UC_12	Admin	Opret hold	2	3 Timer
UC_13	Admin	Rediger hold	1	2 Timer
UC_14	Admin	Slet hold	2	1 Time
UC_15	Admin	Slet booking	5	3 timer
UC_16	Admin	Rediger booking	4	2 Timer
UC_17	Admin	Opret booking	10	3 Timer
UC_18	Admin	Se bookinger	6	3 Timer

UC_19	Admin	Opret lokale	4	3 Timer
UC_20	Admin	Rediger lokale	2	2 Timer
UC_21	Admin	Slet lokale	4	1 Time
UC_22	Admin	Opret inventar	3	3 Timer
UC_23	Admin	Rediger inventar	2	2 Timer
UC_24	Admin	Slet inventar	3	1 Time
UC_25	Admin	Opret fejlmelding	7	4 Timer
UC_26	Admin	Rediger fejlmelding	4	2 Timer
UC_27	Admin	Slet fejlmelding	7	1 Time
UC_28	Admin	Arkiver fejlmelding	2	2 Timer
UC_29	Admin	Vælg statistik periode	4	1 Time
UC_30	Admin	Sammenlign booking statistik	3	2 Timer
UC_31	System	Vis grafisk booking statistik	5	2 Timer
UC_32	Admin	Vis booking statistik grafisk	4	8 Timer
UC_33	System	Generer CSV-file	9	5 Timer
UC_34	Admin, System	Vis dashboard	2	4 Timer
UC_35	Admin	Ændre kodeord	4	2 Timer
UC_36	User	Gå til info view	6	2 Timer
UC_37	System	Gå til info view efter X tid	4	3 Timer
UC_38	Admin, Database	Anvend Strategi pattern til statistik	5	16 Timer
UC_39	Admin, System	Udvælg data til CSV-File	1	7 Timer
UC_40	Admin	Login med kode	10	2 Timer
TOTAL				137 Timer
				Version 2.0

Version 1.0 findes i bilag XXX

Casual use cases:

UC_38	Anvend Strategi pattern til statistik
Actor	System, Database
1. System genererer ud fra valgt strategi 2. System snakker med Database 3. Systemet viser resultat	
Version: 1.0	Forfatter: Alexander, Kamilla, Elias, Mads

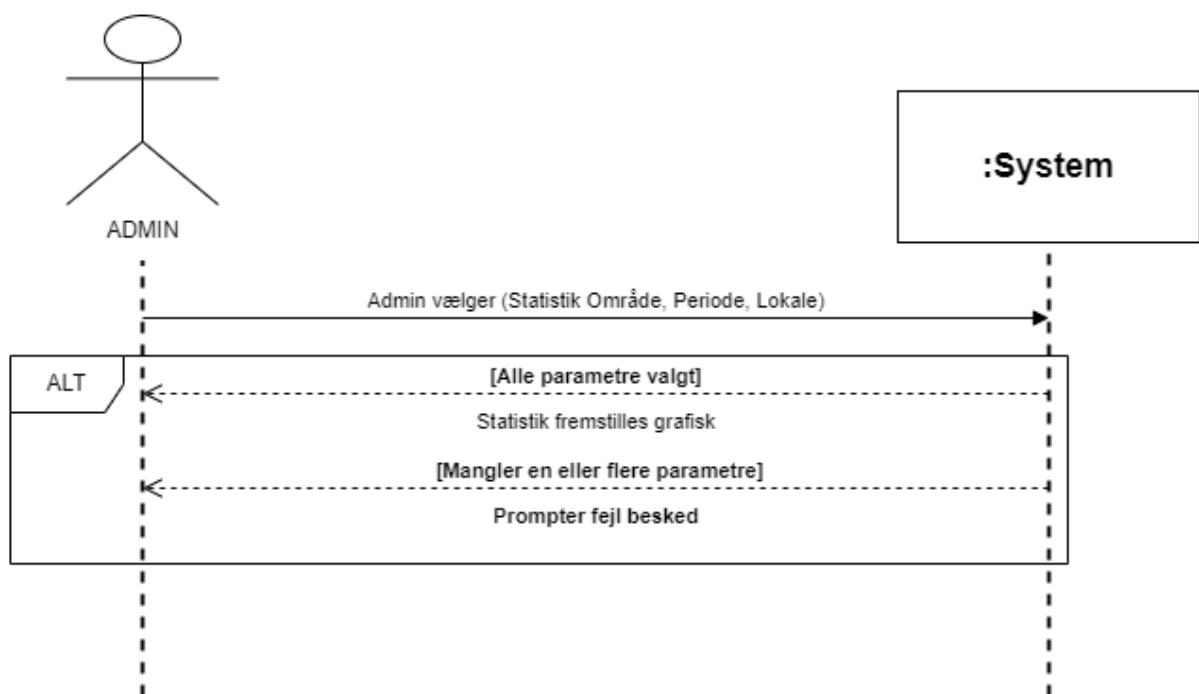
Tabel 11: casual use case af uc_38

Fully dressed use cases:

UC_32	Se booking statistik grafisk
Actor	Admin
Trigger	Admin vælger lokale
Precondition	Admin er logget ind
Postcondition	System viser Statistik
1. Admin vælger Statistik i sidemenu 2. System viser område med muligheder for statistik 3. Admin vælger Statistik område 4. Admin vælger Periode 5. Admin vælger lokale 6. System (Vis grafisk booking statistik - UC31)	
5a. Admin har ikke valgt Periode og/Eller Statistik område. 1. System prompter Admin om at vælge <i>Gå til Step 3</i>	
Version: 1.0	Forfatter: Alexander, Kamilla, Elias, Mads

Tabel 12: Fully dressed use case af uc_32

UC_32



Billede 41: SSD af uc_32

Kernearkitektur:

Under construction fasen i 1. iteration, blev vi klar over, at der manglede et field i tabellen tbl_room. Dette førte til opdatering af tbl_room, hvor fld_floor blev tilføjet (Se bilag 5).

Design

I admin delen har vi valgt at anvende et Strategy Pattern til at håndtere forskellige tidsperioder til den visuelle fremstilling af bookingstatistikken. Brugerne får mulighed for at vælge mellem tre forskellige tidsperioder: én dag, én uge eller én måned. Hver af disse perioder vil repræsentere en unik strategi (algoritme) i vores system.

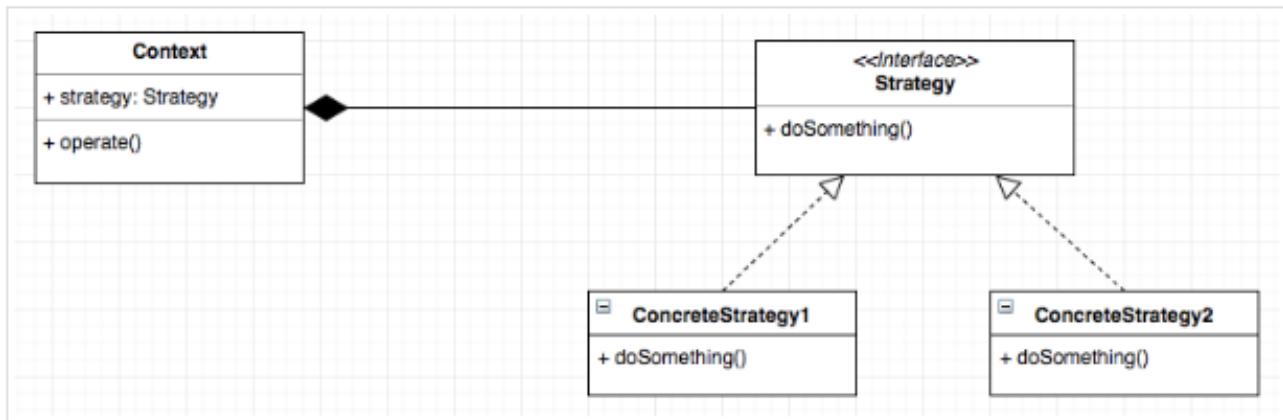
Dette designvalg vil alle blive nærmere beskrevet i construction fasen, når det er blevet implementeret i vores kode.

Construction fase

I det følgende vil Strategy Pattern blive beskrevet med eksempler fra vores system.

Strategy Pattern

Strategy Pattern er et designmønster, der bruges til at definere en familie af algoritmer inden for samme kontekst.¹⁴ Dette mønster muliggør, at hver algoritme (strategi) kan indkapsles og gøres udskiftelig. Dette opnås ved at have et fælles interface for alle de familiære strategier og en context-klasse, der bruger en reference til dette interface til at udføre den ønskede strategi, som det ses på billedet nedenfor.



Billede 42: viser et simpelt UML-diagram over designmønsteret "Strategy Pattern".¹⁵

Formålet med interfacet er at definere en fælles kontrakt, som alle strategier i familien skal implementere. Herved fastlægger interfacet de fælles krav, som alle strategierne skal opfylde. I vores tilfælde drejer det sig om én metode til at generere et StatObj (Statistik objekt):

```
public interface TimeStrategy { 5 usages 3 implementations

    /**
     * This method generates a StatObj based on a specific strategy and a roomID
     * @param statisticsArea the area of statistics to be generated
     * @param roomID the roomID for a selected room
     * @return a StatObj that contains the variables needed to create a barChart.
     */
    StatObj generateObj(StatisticsArea statisticsArea, int roomID); 1 usage 3 implementations

}
```

Billede 43: viser det interface, som vores strategier implementerer.
Som det fremgår af billedet, er fælleskravet én metode, som genererer et StatObj.

¹⁴ <https://refactoring.guru/design-patterns/strategy>

¹⁵ <https://heidyhe.github.io/strategy-pattern/>

Til at anvende de forskellige strategier, har vi en context-klasse. Denne klasse fungerer som en klient, der sørger for at håndtere den valgte strategi og udføre den nødvendige opgave. Context-klassen indeholder en reference til det fælles interface for strategierne og bruger denne reference til at kalde den relevante metode på den aktuelle strategi.

I vores tilfælde drejer det sig om TimeContext-klassen:

```
public class TimeContext { 2 usages

    private TimeStrategy strategy; 3 usages

    public void setStrategy(TimeStrategy strategy) { 3 usages

        this.strategy = strategy;
        LoggerMessage.info(instance: this, message: "Strategy set to " + strategy);
    }

    /**
     * This method generates a StatObj based on the specified strategy and roomID.
     * @param statisticsArea the area of statistics to be generated
     * @return a StatObj that contains the variables needed to create a barChart
     */
    public StatObj generateObj(StatisticsArea statisticsArea, int roomID){ 1 usage
        if (strategy == null){
            throw new IllegalStateException("Strategy is not set");
        }
        return strategy.generateObj(statisticsArea, roomID);
    }
}
```

Billede 44: viser TimeContext-klassen i vores applikation

hvor det ses, at den valgte strategi bliver sat ved brug af setStrategy-metoden, og metoden generateObj bliver kaldt ud fra den valgte strategi. Denne metode genererer et StatObj baseret på den specificerede strategi og et roomID.

I vores system er konteksten "tid", hvorved TimeContext-klassen altså styrer valget og anvendelsen af forskellige tidsstrategier.

Som tidligere nævnt, har brugerne af vores system, mulighed for at vælge imellem 3 forskellige tidsperioder. Disse tre perioder danner vores nuværende strategier:

```

public class DayStrategy implements TimeStrategy{ 1 usage

    // number of days used in the strategy by default
    private int FINAL_DAY = 1; 1 usage

    /**
     * This method generates a StatObj based on a specific strategy and its corresponding number of days and a roomID
     * @param statisticsArea the area of statistics to be generated
     * @param roomID the roomID for a selected room
     * @return a StatObj that contains the variables needed to create a barChart
     */
    @Override 1 usage
    public StatObj generateObj(StatisticsArea statisticsArea, int roomID) {

        // gets today's date
        LocalDate currentDate = TimeStrategyLogic.getInstance().currentDate();
        LoggerMessage.debug( instance: this, message: "Current Date: " + LocalDate.now());

        // generates the StatObj using the TimeStrategyLogic
        // NOTE: start- and end date is the same - that's why 2 x currentDate
        return TimeStrategyLogic.getInstance().generateStatObj(FINAL_DAY, Date.valueOf(currentDate), Date.valueOf(currentDate), statisticsArea, roomID);
    }

}

```

Billede 45: viser vores "DayStrategy"-klasse med en final variabel, der definerer antallet af dage i den specifikke strategi.

```

public class WeekStrategy implements TimeStrategy{ 1 usage

    // number of days used in the strategy by default (7 = days in a week)
    private int FINAL_DAYS_WEEK = 7; 2 usages

    /**
     * This method generates a StatObj based on a specific strategy and its corresponding number of days and a roomID
     * @param statisticsArea the area of statistics to be generated
     * @param roomID the roomID for a selected room
     * @return a StatObj that contains the variables needed to create a barChart
     */
    @Override 1 usage
    public StatObj generateObj(StatisticsArea statisticsArea, int roomID) {

        // gets today's date
        LocalDate currentDate = TimeStrategyLogic.getInstance().currentDate();
        LoggerMessage.debug( instance: this, message: "Current Date: " + LocalDate.now());

        // calculates the start date based on today's date given the number of days used in this strategy
        Date startDate = Date.valueOf(TimeStrategyLogic.getInstance().startDate(currentDate, FINAL_DAYS_WEEK));
        LoggerMessage.debug( instance: this, message: "Start Date: " + startDate);

        // generates the StatObj using the TimeStrategyLogic
        return TimeStrategyLogic.getInstance().generateStatObj(FINAL_DAYS_WEEK, startDate, Date.valueOf(currentDate), statisticsArea, roomID);
    }
}

```

Billede 46: viser vores "WeekStrategy"-klasse med en final variabel, der definerer antallet af dage i den specifikke strategi.

```

public class MonthStrategy implements TimeStrategy{ 1 usage

    // number of days used in the strategy by default (31 = days in a month)
    private int FINAL_DAYS_MONTH = 31; 2 usages

    /**
     * This method generates a StatObj based on a specific strategy and its corresponding number of days and a roomID
     * @param statisticsArea the area of statistics to be generated
     * @param roomID the roomID for a selected room
     * @return a StatObj that contains the variables needed to create a barChart
     */
    @Override 1 usage
    public StatObj generateObj(StatisticsArea statisticsArea, int roomID) {

        // gets today's date
        LocalDate currentDate = TimeStrategyLogic.getInstance().currentDate();
        LoggerMessage.debug( instance: this, message: "Current Date: " + LocalDate.now());

        // calculates the start date based on today's date given the number of days used in this strategy
        Date startDate = Date.valueOf(TimeStrategyLogic.getInstance().startDate(currentDate, FINAL_DAYS_MONTH));
        LoggerMessage.debug( instance: this, message: "Start Date: " + startDate);

        // generates the StatObj using the TimeStrategyLogic
        return TimeStrategyLogic.getInstance().generateStatObj(FINAL_DAYS_MONTH, startDate, Date.valueOf(currentDate), statisticsArea, roomID);
    }
}

```

Billede 47: viser vores "MonthStrategy"-klasse med en final variabel, der definerer antallet af dage i den specifikke strategi.

Som det ses, implementerer hver af disse strategi-klasser interfacet TimeStrategy. Ydermere ses der, at strategi-klasserne har deres egen unikke implementering af metoden generateObj, der genererer et StatObj baseret på den specificerede tidsperiode (strategiens final variabel) og roomID.

Til at håndtere alt logikken i forbindelse med genereringen af et StatObj har vi valgt at anvende en TimeStrategyLogic klasse, som hver strategi samarbejder med for at oprette det ønskede objekt. Denne klasse indeholder metoder, der udfører beregninger, databasekald og datahåndtering – alt det nødvendige for at generere det korrekte StatObj. Hver strategi sørger for at sende de relevante argumenter til TimeStrategyLogic-klassen, hvorved logikken til generering af StatObj'et adskilles fra de enkelte strategier. Dette muliggør den føromtalte indkapsling og muligheden for at udvide applikationen med flere strategier uden at skulle ændre noget i den eksisterende kode – altså følger vi open / closed principippet.

Delkonklusion 2. iteration

Under anden iteration har fokus været på at udvikle use cases relateret til administrator delen af systemet. Derudover har vi på baggrund af den tidligere iteration fundet nye use cases. Disse vil også blive udviklet i 2. iteration.

Tidsestimering

Vores tidsestimering har i denne iteration været mere præcis end tidligere. Dette skyldes til dels at vi generelt er blevet bedre til at estimere. Men også at vi har haft et skærpet fokus på at arbejde målrettet imod ikke at gå udover vores use case.

Gruppearbejde

Gruppearbejdet har fortsat fungeret god. Vi blev enige om at vores GitHub strategi fortsat var den rigtige vej fremad.

Ændringer og justeringer

Det stod os i miden af 2. iteration klart at vi kunne have udnyttet abstrakte klasser bedre f.eks. når vi oprettet nye komponenter. Dog grundet tidspresset valgte vi i fællesskab at dette fremad rettet skulle ændres. Men ikke at refraktor tidligere skrevet kode. Overordnet har vi ikke lavet store ændringer.

Construction-fasen

Vi har ikke opnået alle de mål, vi satte i elaboration-fasen, hvilket skyldes tidspres. Dog har vi grundet valget om at programmere use case med høj prioritet først. Sikret mest mulige funktionalitet. Programmet fungerer, og opfylder alle krav.

Transition

Transitionsfasen er den sidste og afsluttende fase i unified process. Her gennemtestet programmet, og bruger vejledning samt installations guide udvikles. Ydermere udvikles der et UML-diagram over systemets klasser.

Installations guide

For at installere Athena systemet skal følgende steps gennemføres. Da systemet er et enkelt, bruger system sker alt dette på samme PC.

Database

Installer MSSQL-database. Opret en bruger i MSSQL-databasen, og erstat brugernavn, adgangskode, port og IP-adresse i (src/main/java/org/apollo/template/Database/db.properties).

Kør testen (src/test/java/org/apollo/template/Database/JDBCTest.java) for at sikre, at en forbindelse er oprettet.

Kør databaseCreateScript fra (src/main/java/org/apollo/template/Database/databaseCreateScript.sql).

Kør databaseInsertScript fra (src/main/java/org/apollo/template/Database/databaseInsertScript.sql)

Kør databaseStoredProcedureScript fra (src/main/java/org/apollo/template/Database/databaseStoredProcedure.sql)

For at sikre, at systemet er installeret korrekt, skal du køre alle tests fra (src/test/java/org/apollo/template).

System

System hentes direkte ned fra GitHub, branchen (Main), dernæst startes programmet.

Brugervejledning

Athena Systemet består af to dele: ad-hoc booking og admin panel. Admin panel sidens formål er at skabe en brugerflade til at administrere bookinger, og lokaler med mere. Ad-hoc-delen af systemet består i en infoskærm hvor dagens møder er præsenteret og møder kan bookes på selve samme dag.

Ad-hoc.

For at oprette et møde vælges knappen “BOOK” på infoskærmen, hvorefter et lokale og en tidshorisont vælges. Dernæst bedes brugeren om nogle informationer som er relevante for systemet at vide. Efter dette kan bookingen bekræftes.

Ønsker man at slette sin booking sker dette ved at vælge knappen “SLET BOOKING”. Dernæst skal brugeren indtaste den email som bookingen er oprettet under, og vælge mødet der skal slettes.

Admin

Admin-panelet har meget forskellig funktionalitet. Dette kan opdeles i to sektioner.

System administration:

System administrationsdelen går ud på at sikre at relevante hold, lokaler og information om ovenstående altid passer. Derfor er der i systemet flere forskellige faner, som en admin kan navigere til for at sikre dette.

Admin booking:

Admin-booking er en udvidelse af ad-hoc booking, som giver mulighed for at booke et lokale flere dage ad gangen, booke lokaler i fremtiden og slette bookinger.

Test

Introduktion

Unit-testing er et essentielt værktøj til at finde runtime errors, teste funktioner og metoder.

Test Public subscriber - Message Broker.

Det har været relevant at udvikle test af Public subscriber – Message Broker grundet kompleksiteten og vigtigheden. Under test af Messages brokeren se: (src/test/java/org/apollo/template/persistence/PubSub/MessagesBrokerTest.java) er flere problematikker opstået.

En af fejlene var at vi ikke tog højde for hvis et null objekt blev sendt, en anden var vi i unsubscribe metoden havde byttet rundt på topic og subscriber. Fordi disse fejl opstod under runtime og ikke compiletime, var det kun muligt at finde ved hjælp af Unit testene.

Fordi testen omhandler nogle komplekse funktioner, har vi gjort brug af JUnit tests BeforeEach og TearDown annotations, som sikrer at disse metoder kaldes hhv. før og efter testen. Se (**Fejl! Henvisningskilde ikke fundet.**)

```

// Setting up an environment for the tests.
@BeforeEach
void setUp() {

    // setting up debugger environment.
    ConfigLoader.get();

    // creating a mock object to run the test on.
    booking = new Booking();
    room = new Room();
    meetingType = new MeetingType();
    email = new Email( emailID: 16,   email: "testEmail@easv365.dk");

    // constructing meetingType
    meetingType.setMeetingTypeID(2);
    meetingType.setMeetingTypeName("Møde");

    // constructing room
    room.setRoomID(10);
    room.setRoomName("301");
    room.setFloor(2);
    room.setRoomMaxPersonCount(35);

    // constructing the booking.
    booking.setMeetingType(meetingType);
    booking.setUsername("bob");
    booking.setStartTime(Time.valueOf( s: "08:00:00"));
    booking.setEndTime(Time.valueOf( s: "10:00:00"));
    booking.setEmail(email);
    booking.setBookingID(12);
    booking.setNumberOfParticipants(16);
    booking.setAdHoc(true);
    booking.setUsername("peter");

    pair = new Pair<>(MessagesBrokerTopic.BOOKING_INFORMATION, v: this);
}

}

```

Billede 48: `beforeEach`: kaldes før alle tests.

Ligeledes ses her tearDown.

```
@AfterEach  
void tearDown() {  
    // setting all obj to null  
    booking = null;  
    email = null;  
    room = null;  
    meetingType = null;  
}
```

Billede 49: teardown metode, bruges i messagesbroker tests src/test/java/org/apollo/template/persistence/PubSub/MessagesBrokerTest.java

Messages broker klassen består af flere forskellige metoder, dog er nogle mere relevante at teste end andre. De relevante metoder er: subscribe, unSubscribe og publish.

Structure

- ▼ **MessagesBroker**
 - (m) **MessagesBroker()**
 - (m) **subscribe(Subscriber, MessagesBrokerTopic): void**
 - (m) **unSubscribe(Subscriber, MessagesBrokerTopic): void**
 - (m) **publish(MessagesBrokerTopic, Object): void**
 - (m) **getInstance(): MessagesBroker**
 - (m) **getSubscriberList(): List<Pair<MessagesBrokerTopic, Subscriber>>**
 - (f) **instance: MessagesBroker**
 - (f) **subscriberList: List<Pair<MessagesBrokerTopic, Subscriber>> = new ArrayList<>()**

Billede 50: messageBroker class structure.

Fordi messagesbrokeren ikke håndtere tal eller andre grænser har vi ikke fundet det relevant at bruge boundary value analysis (BVA). Vi har dog fundet det meget interessant at bruge equivalence partitioning (EQ).

```

/*
This test aims to ensure that its possible to publish an object, and that the right object is afterwards received.
*/
@Test
void publishObjectValid() {

    // subscribing to the messages broker
    MessagesBroker.getInstance().subscribe( subscriber, this, MessagesBrokerTopic.BOOKING_INFORMATION);

    // publishing object.
    try {
        MessagesBroker.getInstance().publish(MessagesBrokerTopic.BOOKING_INFORMATION, booking);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }

    Object o = receivedObject;
    // casting object to booking.
    Booking actualBooking = (Booking) o;

    assertEquals(actualBooking, booking);
}

```

Billede 51: Publish metode test.

Overstående metode tester hvorvidt der opstår sideeffekts når et objekt publiceres, dette sker ved at sammenligne det med et objekt som er identisk. Derved kan vi se om det publiceret objekt er ens. Underforstået ligger der også en test hvorvidt et objekt reelt kan publiceres.

```

/*
This test aims to ensure that it is not possible to publish an object which is null
*/
@Test
void publishObjectInvalid() {

    boolean exspected = false;
    boolean actual;

    // subscribing to the messages broker
    MessagesBroker.getInstance().subscribe( subscriber, this, MessagesBrokerTopic.BOOKING_INFORMATION);

    // publishing object.
    try {
        MessagesBroker.getInstance().publish(MessagesBrokerTopic.BOOKING_INFORMATION, messages: null);
        actual = true;
    } catch (Exception e) {
        actual = false;
    }

    assertEquals(exspected, actual);
}

```

Billede 52: Messages broker test, publishers null object

Denne test resulteret I at en mangel blev fundet, nemlig at messages broker tidligere kunne publicere null objekter. Der er udviklet flere test, men disse har været de mest relevante at beskrive.

Test JDBC

Da vi bruger JDBC til vores database, er det nødvendigt at være sikker på at vores JDBC fungerer som planlagt.

Derfor har vi en relative simple test som først får/skaber vores JDBC object (SingleTon).

Derfor sætter vi en connection op, og sætter vores testkriterier op, så prøver vi med en try-catch.

Og til sidst sørger vi for at vores connection til databasen er closed igen da vi er færdig med testen.

Det en simple men brugbar test i forhold til om en udviklers miljø er korrekt sat op.3

```
class JDBCCTest {
    private static JDBC jdbc;  3 usages

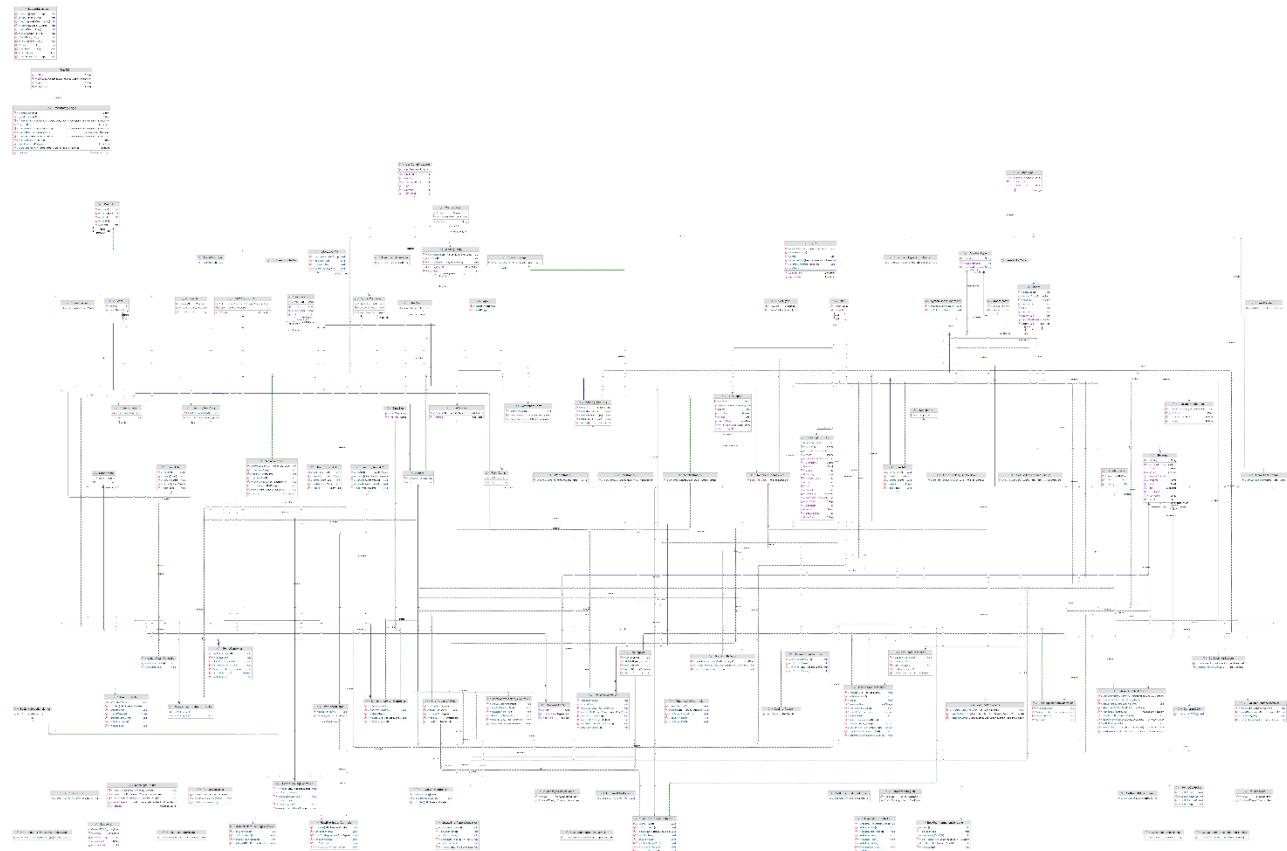
    @BeforeAll
    static void setUp() {
        // Create an instance of JDBC before running the tests
        jdbc = JDBC.get();
    }

    @Test
    void getConnection() {
        // Check if the getConnection method returns a non-null connection obj
        Connection connection = jdbc.getConnection();
        assertNotNull(connection);
        try {
            // Check if the connection is valid by calling isValid method
            assertTrue(connection.isValid(timeout: 5));
        } catch (SQLException e) {
            fail("SQLException occurred while checking connection validity: " +
                e.getMessage());
        }
    }

    @AfterAll
    static void tearDown() {
        // Close the database connection after running the tests
        jdbc.databaseClose();
    }
}
```

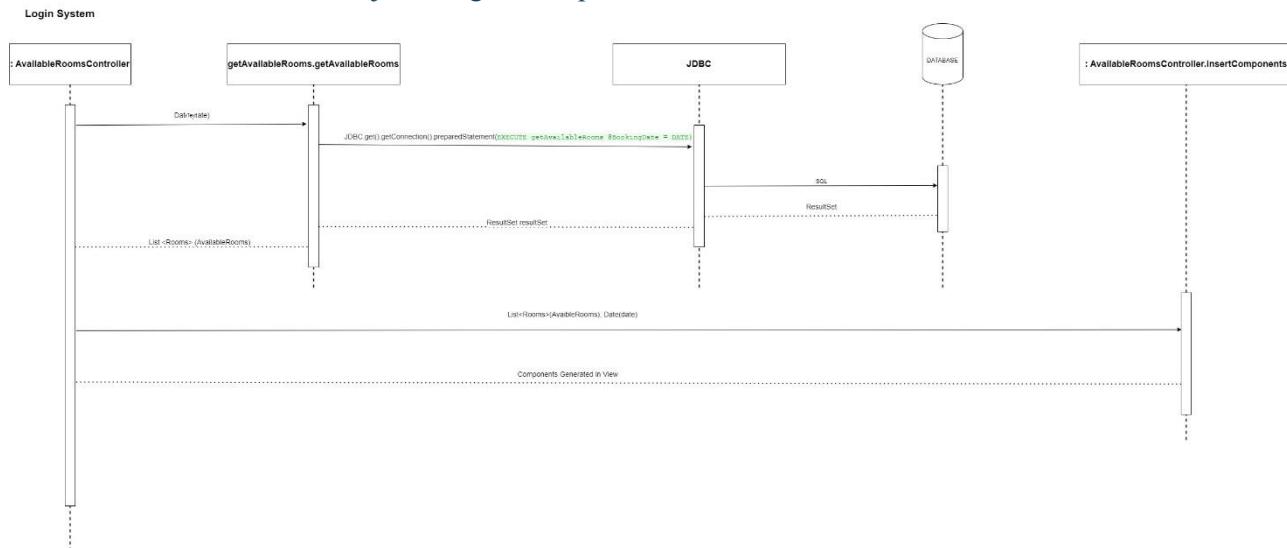
billede 53 - JDBC.java, vores JDBC test.

UML ATHENA



billede 54 - UML Athena - <https://github.com/eliastherkildsen/ATHENA/blob/developement/src/main/resources/org/apollo/template/images/mainUMLAthena.png>

SD AvailableRoomsController.java brug af Component of database



billede 55 - SD over Generation af Component for AvailableRoomsController

En SD, der viser vejen fra Controller til Database tilbage til Controller, hvor efter der kan laves components med information fra databasen, intentionen var at vise hvordan klasser snakker sammen for ultimative at kunne vise den relevante information.

Bilag

Bilag 1

Vision: 1.0

Formålet med systemet er at udvikle et informations- og bookingsystem. Systemet skal gøre det muligt for ansatte og studerende at få et hurtigt overblik over institutionens lokaler og deres tilgængelighed. Dette skal foregå på institutionens infoskærm eller på admin-computeren. Det skal være muligt for de ansatte at booke lokaler til undervisningen, møder mm. og bestille forplejning hvis nødvendigt. Som studerende skal det være muligt at booke ledige lokaler ad-hoc. Ydermere skal systemet kunne vise booking statistikker. (version 1.0)

Bilag 2

Spørgsmål til stakeholder

1. Ønskes der stram booking? Eller afsat tid mellem hver booking?
Stram booking

2. Er stakeholder interesseret i antal af personer til mødet, eller at "Poul-Erik" har været med til mødet? (Er det til fremmøde statistik?)
Antal – skrives ved bookingen

3. Er infoskærmen tiltænkt touchskærm?
Computer

4. Kan studerende booke lokale?
Ja – ad hoc

5. Er det et system med fri adgang? Eller er det et lukket system? (Er login nødvendigt?)
Fri adgang – unik adgang?

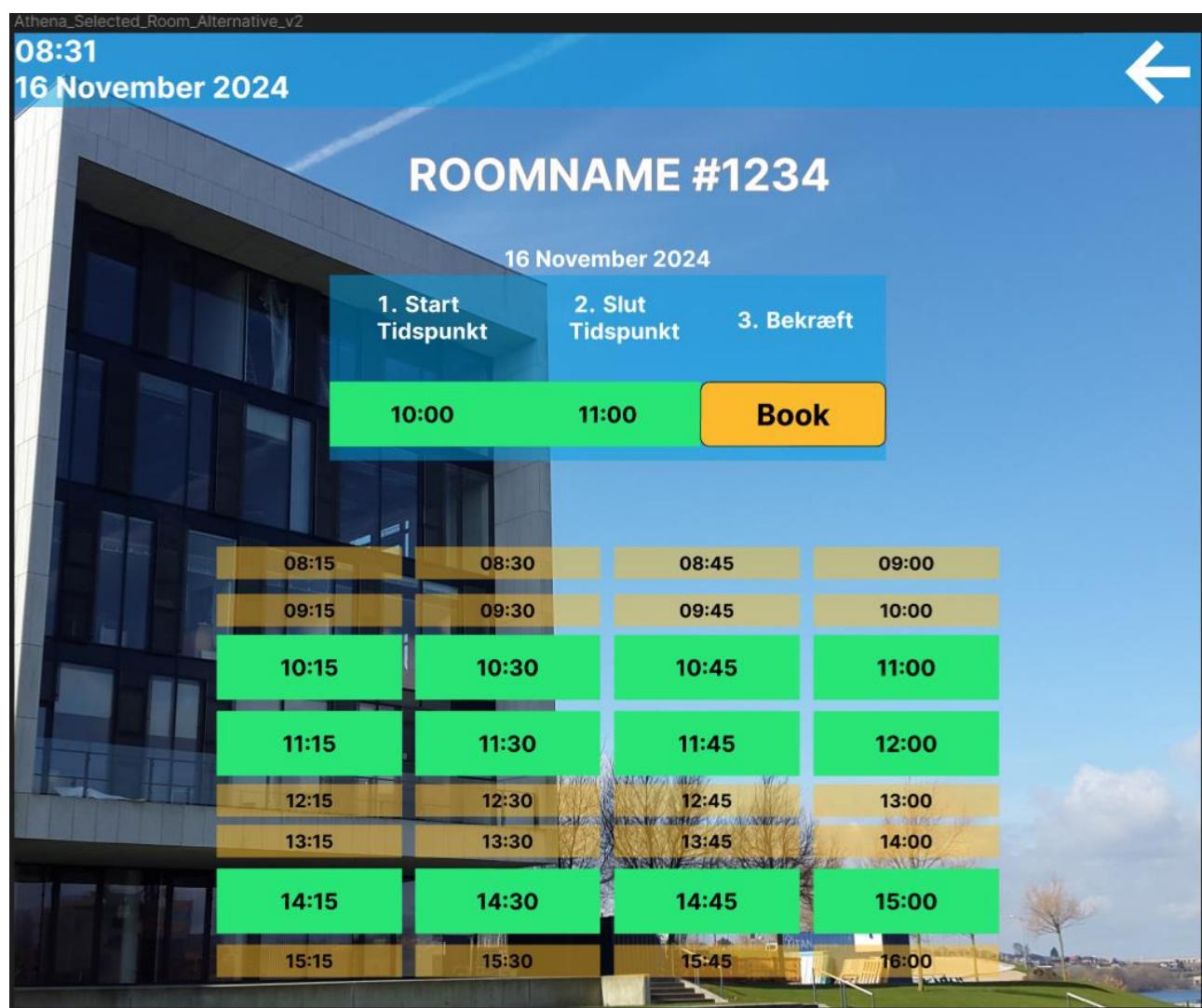
6. I forhold til forplejning: hvilken grad?
(boolean eller lister? Skal der være en oversigt over lokaler med forplejning?)

7. I forhold til fejlmelding: hvad skal der ske med fejlmeddelelsen?
Bokse der kan krydses af + note
Projekter

Højtaler
Inventar
Boks der hedder andet

8. Hvad er faciliteter?
9. Hvilken statistik får stakeholder noget ud af? Tal/diagrammer/mix? Hvis diagram – hvilket/hvilke?
Tal, dashboard nice to have

Bilag 3



08:31

16 November 2024

**Dagens Møder:**

Lok. 301 Saglsmøde Jesper Hansen 09:00 - 11:30

Lok. 302 Saglsmøde Jesper Hansen 09:00 - 11:30

Lok. 303 Saglsmøde Jesper Hansen 09:00 - 11:30

Lok. 304 Saglsmøde Jesper Hansen 12:00 - 14:00

Lok. 305 Saglsmøde Jesper Hansen 09:00 - 11:30

**BOOK**

08:31

16 November 2024



Lok. 301 3. Sal Person kapacitet: 25 Inventar 	Lok. 302 3. Sal Person kapacitet: 25 Inventar 	Lok. 303 3. Sal Person kapacitet: 25 Inventar 	Lok. 304 3. Sal Person kapacitet: 25 Inventar
Lok. 305 3. Sal Person kapacitet: 25 Inventar 	Lok. 306 3. Sal Person kapacitet: 25 Inventar 	Lok. 307 3. Sal Person kapacitet: 25 Inventar 	Lok. 308 3. Sal Person kapacitet: 25 Inventar
Lok. 403 4.Sal Person kapacitet: 25 Inventar 	Lok. 404 4.Sal Person kapacitet: 25 Inventar 	Lok. 405 4.Sal Person kapacitet: 25 Inventar 	Lok. 406 4.Sal Person kapacitet: 25 Inventar

08:31
16 November 2024



Lok. 304

16 November 2024

1. Start Tidspunkt 2. Slut Tidspunkt 3. Bekræft

Book

08:00	08:15	08:30	08:45
09:00	09:15	09:30	09:45
10:00	10:15	10:30	10:45
11:00	11:15	11:30	11:45
12:00	12:15	12:30	12:45
13:00	13:15	13:30	13:45
14:00	14:15	14:30	14:45
15:00	15:15	15:30	15:45

08:31
16 November 2024



Lok. 304

16 November 2024

1. Start Tidspunkt 2. Slut Tidspunkt 3. Bekræft

Book

08:15	08:30	08:45	09:00
09:15	09:30	09:45	10:00
10:15	10:30	10:45	11:00
11:15	11:30	11:45	12:00
12:15	12:30	12:45	13:00
13:15	13:30	13:45	14:00
14:15	14:30	14:45	15:00
15:15	15:30	15:45	16:00

08:31
16 November 2024



Lok. 304

Møde navn:

Salgsmøde

Dit navn:

Alexander

Email:

alexander@gmail.com

Antal forventede deltager:

20

Book

08:31
16 November 2024



Lok. 304

Succes!

Rummet er nu booket for dit møde:

Salgsmøde

10:00 - 11:00

16 November 2024

PIN

9

8

7

6

Brug PIN for at slette booking

08:31

16 November 2024



Lok. 304

16 November 2024

1. Start Tidspunkt	2. Slut Tidspunkt	3. Bekræft
10:00	11:00	Book

08:15	08:30	08:45	09:00
09:15	09:30	09:45	10:00
10:15	10:30	10:45	11:00
11:15	11:30	11:45	12:00
12:15	12:30	12:45	13:00
13:15	13:30	13:45	14:00
14:15	14:30	14:45	15:00
15:15	15:30	15:45	16:00

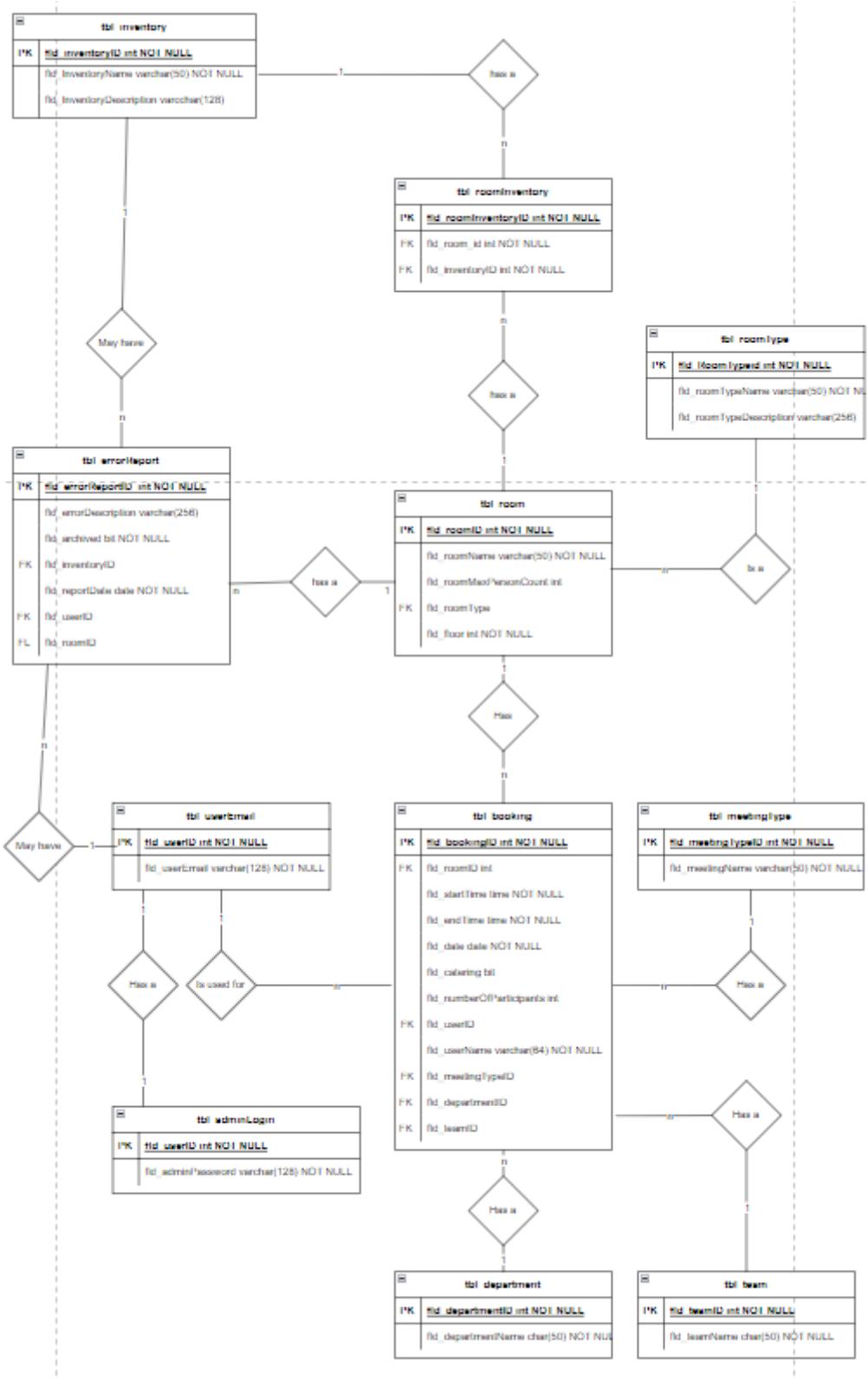
Bilag 4

ID	Actor	Navn	Prioritet (1-10)	Estimat (timer)
UC_0	User	Vælg system	10	1 Time
UC_1	System, User	Vis reserverede lokaler	10	5 Timer
UC_2	System	Sorter reserverede lokaler (FD) lokale/tid	4	4 Timer
UC_3	System	Vis aktuel tid	1	1 Time
UC_4	System	Vis aktuel dato	5	1 Time
UC_5	User	Vælg starttidspunkt / sluttidspunkt	10	7 Timer

UC_6	System	Vis ledige lokaler	8	5 Timer
UC_7	System, User	Indtast Booking Oplysninger	7	4 Timer
UC_8	System	Vis booking bekræftelse	3	4 Timer
UC_9	System	Vis samtykkeerklæring	7	1 Time
UC_10	User	Svar på samtykkeerklæring	7	1 Time
UC_11	User	Slet booking	2	3 Timer
UC_12	Admin	Opret hold	2	3 Timer
UC_13	Admin	Rediger hold	1	2 Timer
UC_14	Admin	Slet hold	2	1 Time
UC_15	Admin	Slet booking	5	3 timer
UC_16	Admin	Rediger booking	4	2 Timer
UC_17	Admin	Opret booking	10	3 Timer
UC_18	Admin	Vis bookinger	6	3 Timer
UC_19	Admin	Opret lokale	4	3 Timer
UC_20	Admin	Rediger lokale	2	2 Timer
UC_21	Admin	Slet lokale	4	1 Time
UC_22	Admin	Opret inventar	3	3 Timer
UC_23	Admin	Rediger inventar	2	2 Timer
UC_24	Admin	Slet inventar	3	1 Time
UC_25	Admin	Opret fejlmelding	7	4 Timer
UC_26	Admin	Rediger fejlmelding	4	2 Timer
UC_27	Admin	Slet fejlmelding	7	1 Time
UC_28	Admin	Arkiver fejlmelding	2	2 Timer
UC_29	Admin	Vælg statistik periode	4	1 Time
UC_30	Admin	Sammenlign booking statistik	3	2 Timer

UC_31	System	Vis booking statistik tal	5	2 Timer
UC_32	System	Vis booking statistik grafisk	4	8 Timer
UC_33	System	Generer CSV-file	9	5 Timer
TOTAL				92 Timer
	Version 1.0			

Bilag 5



Bilag 6

Database creation script:

```
-- Creating the database
CREATE DATABASE db_Athena;
GO

-- Selecting the database
USE db_Athena;

-- Creating the tables in the database.
CREATE TABLE tbl_inventory(
    fld_inventoryID INTEGER IDENTITY (1,1) PRIMARY KEY,
    fld_inventoryName VARCHAR(50) NOT NULL,
    fld_inventoryDescription VARCHAR(128)
);

CREATE TABLE tbl_roomType(
    fld_roomTypeID INTEGER IDENTITY (1,1) PRIMARY KEY,
    fld_roomTypeName VARCHAR(50) NOT NULL,
    fld_roomTypeDescription VARCHAR(512)
);

CREATE TABLE tbl_room(
    fld_roomID INTEGER IDENTITY (1,1) PRIMARY KEY,
    fld_roomName VARCHAR(50) NOT NULL,
    fld_floor int,
    fld_roomMaxPersonCount INTEGER,
    fld_roomTypeID INTEGER,
    FOREIGN KEY (fld_roomTypeID) REFERENCES tbl_roomType(fld_roomTypeID)
);

CREATE TABLE tbl_roomInventory(
    fld_roomInventoryID INTEGER IDENTITY (1,1) PRIMARY KEY,
    fld_roomID INTEGER,
    fld_inventoryID INTEGER,
    FOREIGN KEY (fld_roomID) REFERENCES tbl_room(fld_roomID),
    FOREIGN KEY (fld_inventoryID) REFERENCES tbl_inventory(fld_inventoryID)
);

CREATE TABLE tbl_userEmail(
    fld(userID INTEGER IDENTITY(1,1) PRIMARY KEY,
    fld_userEmail VARCHAR(128) NOT NULL
);

CREATE TABLE tbl_errorReport(
    fld_errorReportID INTEGER IDENTITY (1,1) PRIMARY KEY,
    fld_archived BIT NOT NULL,
    fld_reportDate DATE NOT NULL,
    fld_inventoryID INTEGER,
    fld(userID INTEGER,
    fld_reportDescription VARCHAR(256),
    fld_roomID INTEGER
    FOREIGN KEY (fld_inventoryID) REFERENCES tbl_inventory (fld_inventoryID),
    FOREIGN KEY (fld(userID) REFERENCES tbl_userEmail(fld(userID),

```

```

        FOREIGN KEY (fld_roomID) REFERENCES tbl_room(fld_roomID)
);

CREATE TABLE tbl_adminLogin(
    fld(userID INTEGER IDENTITY(1,1) PRIMARY KEY,
    fld_adminPassword varchar(128) NOT NULL
);

CREATE TABLE tbl_meetingType(
    fld_meetingTypeID INTEGER IDENTITY(1,1) PRIMARY KEY,
    fld_meetingType VARCHAR(50) NOT NULL
);

CREATE TABLE tbl_team(
    fld_teamID INTEGER IDENTITY(1,1) PRIMARY KEY,
    fld_teamName VARCHAR(50) NOT NULL
);

CREATE TABLE tbl_department(
    fld_departmentID INTEGER IDENTITY(1,1) PRIMARY KEY,
    fld_departmentName VARCHAR(50) NOT NULL
);

CREATE TABLE tbl_booking(
    fld_bookingID INTEGER IDENTITY(1,1) PRIMARY KEY,
    fld_startTime TIME(0) NOT NULL,
    fld_endTime TIME(0) NOT NULL,
    fld_date DATE NOT NULL,
    fld_catering BIT NOT NULL,
    fld_numberOfParticipants INTEGER NOT NULL,
    fld_userName VARCHAR(50) NOT NULL,
    fld(userID INTEGER,
    fld_roomID INTEGER,
    fld_meetingTypeID INTEGER,
    fld_departmentID INTEGER,
    fld_teamID INTEGER,
    FOREIGN KEY (fld(userID) REFERENCES tbl_userEmail(fld(userID)),
    FOREIGN KEY (fld_roomID) REFERENCES tbl_room(fld_roomid),
    FOREIGN KEY (fld_meetingTypeID) REFERENCES
tbl_meetingType(fld_meetingTypeID),
    FOREIGN KEY (fld_departmentID) REFERENCES tbl_department(fld_departmentID),
    FOREIGN KEY (fld_teamID) REFERENCES tbl_team(fld_teamID),
)

```

Bilag 7

Database Insert script:

```

USE db_Athena;

-- tbl_inventory
INSERT INTO tbl_inventory (fld_inventoryName, fld_inventoryDescription)
VALUES
    ('Andet', 'Et ikke defineret objekt'),
    ('Tavle', 'Stor tavle til at skrive på med kridt eller whiteboard markers'),
    ('Projektor', 'Enhed til at projicere computerens skærm på væggen eller et

```

```

lærred'),
('Stol', 'Standard stol'),
('Bord', 'Standard bord'),
('Mikrofon', 'Mikrofon til brug ved præsentationer eller forelæsninger'),
('Højttaler', 'Højttaler til lydudstyr i ilokalet');

-- tbl_roomType
INSERT INTO tbl_roomType (fld_roomTypeName, fld_roomTypeDescription)
VALUES
    ('Andet', 'Et ikke defineret lokale'),
    ('Møde lokale', 'Et lokale til at holde møder'),
    ('Klasseværelse', 'Et lokale til at undervise studerende'),
    ('Hybrid lokale', 'Et lokale til at undervise studerende, med mulighed for
online undervisning'),
    ('Værksted', 'Et lokale med maskiner til at arbejde med rå materialer');

-- tbl_room
INSERT INTO tbl_room (fld_roomName, fld_roomMaxPersonCount, fld_roomTypeID,
fld_floor)
VALUES

    -- 3 sal på alsion.
    ('305', 30, 3, 3),
    ('306', 18, 2, 3),
    ('307', 18, 2, 3),
    ('315', 20, 2, 3),
    ('316', 35, 2, 3),
    -- 4 sal på alsion
    ('400', 12, 4, 4),
    ('401', 30, 2, 4),
    ('402', 30, 2, 4),
    ('403', 12, 2, 4),
    ('404', 30, 2, 4),
    ('405', 16, 2, 4),
    ('406', 18, 2, 4),
    ('407', 18, 2, 4),
    ('409', 26, 2, 4),
    ('410', 18, 2, 4);

-- tbl_roomInventory
INSERT INTO tbl_roomInventory (fld_roomID, fld_inventoryID)
VALUES
    -- Room 305
    (1, 1),
    (1, 2),
    (1, 3),
    (1, 4),

    -- room 306
    (2, 1),
    (2, 2),
    (2, 3),
    (2, 4),
    (2, 5),

    -- room 307
    (3, 1),
    (3, 2),
    (3, 3),

```

```

(3, 4);

-- tbl_userEmail
INSERT INTO tbl_userEmail (fld_userEmail)
VALUES
    ('peterRasmussen@easv365.dk'),
    ('MadsPetersen@easv365.dk'),
    ('KimSøndergård@easv365.dk'),
    ('LarsHansen@easv365.dk');

-- tbl_errorReport

-- tbl_adminLogin

-- tbl_meetingType
INSERT INTO tbl_meetingType (fld_meetingType)
VALUES
    ('Anonymt'),
    ('Andet'),
    ('Møde'),
    ('Eksamens'),
    ('Undervisning');

-- tbl_team
INSERT INTO tbl_team (fld_teamName)
VALUES
    ('Andet'),
    ('Anonymt'),
    ('d22'),
    ('d23'),
    ('e22'),
    ('e23'),
    ('pt22'),
    ('pt23'),
    ('md22'),
    ('md23');

-- tbl_department
INSERT INTO tbl_department (fld_departmentName)
VALUES
    ('Andet'),
    ('Anonymt'),
    ('Rengøring'),
    ('Ledelse'),
    ('Underviser');

-- tbl_booking
INSERT INTO tbl_booking (fld_startTime, fld_endTime, fld_date, fld_catering,
fld_numberOfParticipants, fld_userName, fld(userID), fld_roomID,
fld_meetingTypeID, fld_departmentID, fld_teamID)
VALUES
    -- Booking for today (current date)
    ('09:00:00', '10:00:00', CAST(GETDATE() AS DATE), 1, 10, 'Mads', 3, 1, 3, 5,
4),
    ('11:30:00', '13:00:00', CAST(GETDATE() AS DATE), 0, 8, 'Lars', 5, 2, 3, 5,
4),
    ('14:00:00', '16:00:00', CAST(GETDATE() AS DATE), 0, 6, 'Peter', 2, 2, 3, 5,
4),

```

```

-- Booking for tomorrow
('08:30:00', '09:45:00', CAST(DATEADD(DAY, 1, GETDATE()) AS DATE), 1, 9,
'Kim', 4, 7, 2, 1, 1),
('10:30:00', '11:45:00', CAST(DATEADD(DAY, 1, GETDATE()) AS DATE), 1, 8,
'Mads', 3, 1, 3, 5, 4),
('13:00:00', '13:45:00', CAST(DATEADD(DAY, 1, GETDATE()) AS DATE), 0, 5,
'Lars', 5, 2, 3, 5, 4),
-- Booking for the day after tomorrow
('09:15:00', '10:15:00', CAST(DATEADD(DAY, 2, GETDATE()) AS DATE), 1, 10,
'Peter', 2, 2, 3, 5, 4),
('11:30:00', '13:00:00', CAST(DATEADD(DAY, 2, GETDATE()) AS DATE), 0, 8,
'Kim', 4, 7, 2, 1, 1),
('14:00:00', '16:00:00', CAST(DATEADD(DAY, 2, GETDATE()) AS DATE), 0, 6,
'Mads', 3, 1, 3, 5, 4),
-- Booking for 3 days from now
('08:45:00', '09:45:00', CAST(DATEADD(DAY, 3, GETDATE()) AS DATE), 1, 9,
'Lars', 5, 2, 3, 5, 4),
('10:30:00', '11:45:00', CAST(DATEADD(DAY, 3, GETDATE()) AS DATE), 1, 8,
'Peter', 2, 2, 3, 5, 4),
('13:00:00', '13:45:00', CAST(DATEADD(DAY, 3, GETDATE()) AS DATE), 0, 5,
'Kim', 4, 7, 2, 1, 1),
-- Booking for 4 days from now
('09:00:00', '10:00:00', CAST(DATEADD(DAY, 4, GETDATE()) AS DATE), 1, 10,
'Mads', 3, 1, 3, 5, 4),
('11:30:00', '13:00:00', CAST(DATEADD(DAY, 4, GETDATE()) AS DATE), 0, 8,
'Lars', 5, 2, 3, 5, 4),
('14:00:00', '16:00:00', CAST(DATEADD(DAY, 4, GETDATE()) AS DATE), 0, 6,
'Peter', 2, 2, 3, 5, 4);

```

Bilag 8

Database Stored Procedures:

```

CREATE PROCEDURE AddEmailIfNotExists
@EmailAddress NVARCHAR(255)
AS
BEGIN
    -- Check if the email already exists
    IF NOT EXISTS (SELECT 1 FROM tbl_userEmail WHERE fld_userEmail =
@EmailAddress)
        BEGIN
            -- Insert the email into the table
            INSERT INTO tbl_userEmail(fld_userEmail)
            VALUES (@EmailAddress);

            -- Indicate success
            SELECT 'Email added successfully' AS Result;
        END
    ELSE
        BEGIN
            -- Indicate that the email already exists
            SELECT 'Email already exists' AS Result;
        END
END;
GO

-- Function for InfoScreen
CREATE PROCEDURE GetBookingsByDate

```

```

@BookingDate DATE -- We are doing this by date.
AS
BEGIN
    -- SELECT the relevant data fields i want it to return.
    SELECT
        tbl_booking.fld_startTime,
        tbl_booking.fld_endTime,
        tbl_booking.fld_userName,
        tbl_room.fld_roomName,
        tbl_meetingType.fld_meetingType
    FROM
        tbl_booking
            INNER JOIN tbl_room ON tbl_booking.fld_roomID = tbl_room.fld_roomID
            INNER JOIN tbl_meetingType ON tbl_booking.fld_meetingTypeID =
tbl_meetingType.fld_meetingTypeID
    WHERE
        tbl_booking.fld_date = @BookingDate
    ORDER BY
        tbl_booking.fld_startTime ASC,
        tbl_booking.fld_endTime ASC;
END;
GO

-- Stored procedure to find available rooms on a given date
CREATE PROCEDURE getAvailableRooms (@BookingDate DATE)
AS
BEGIN

    -- Calculates the total booking time for each room on the specified date
    ('NULL' if no booking yet)
    WITH BookedTime AS (
        SELECT
            fld_roomID,
            SUM(DATEDIFF(MINUTE, fld_startTime, fld_endTime)) AS
total_booked_time_minutes
        FROM
            tbl_booking
        WHERE
            fld_date = @BookingDate
        GROUP BY
            fld_roomID
    )

    -- Selects selected fields from 3 tables related to each other using INNER
    JOIN and LEFT JOIN - these are to be presented when displaying available rooms
    SELECT
        tbl_room.fld_roomID,
        tbl_room.fld_roomName,
        tbl_room.fld_floor,
        tbl_room.fld_roomMaxPersonCount,
        tbl_room.fld_roomTypeID,
        tbl_roomType.fld_roomTypeName,
        tbl_roomType.fld_roomTypeDescription

    FROM
        tbl_room
            LEFT JOIN BookedTime ON tbl_room.fld_roomID = BookedTime.fld_roomID
            INNER JOIN tbl_roomType ON tbl_room.fld_roomTypeID =
tbl_roomType.fld_roomTypeID

```

```

-- Filters the results to include only rooms that are either not booked on
the specified date or have a total booked time less than 8 hours (480 minutes)
WHERE
    BookedTime.total_booked_time_minutes < 480 OR
BookedTime.total_booked_time_minutes IS NULL

-- Groups the results by five columns to ensure that we only see each room
once in the list, even if there are multiple bookings on the selected day.
GROUP BY
    tbl_room.fld_roomID,
    tbl_room.fld_roomName,
    tbl_room.fld_floor,
    tbl_room.fld_roomTypeID,
    tbl_roomType.fld_roomTypeName,
    tbl_roomType.fld_roomTypeDescription,
    tbl_room.fld_roomMaxPersonCount

-- The available rooms are sorted by room name (ascending) and start time
(ascending)
ORDER BY
    tbl_room.fld_roomName ASC;

END;
GO

-- Gets Room Name from a given Room ID
CREATE PROCEDURE getRoomNameFromID (@RoomID int)
AS
BEGIN
    SELECT
        tbl_room.fld_roomName
    FROM
        tbl_room
    WHERE
        fld_roomID = @RoomID
END
GO

-- Stored procedure to find all bookings on a specific and in a specific room
CREATE PROCEDURE getBookingsFromDate (@BookingDate Date, @RoomID INTEGER)
AS
BEGIN

    SELECT
        tbl_booking.fld_startTime,
        tbl_booking.fld_endTime
    FROM
        tbl_booking
    WHERE
        fld_date = @BookingDate AND fld_roomID = @RoomID

END;
GO

-- Stored procedure to finde email id by email adress

CREATE PROCEDURE getEmailIDByEmailAdress (@emailAdress varchar(256))
AS

```

```

BEGIN
    SELECT
        fld(userID
    FROM
        tbl_userEmail
    WHERE
        fld(userID) = @emailAddress

END;
GO

-- Stored procedure for getting the meetingTypeID by passing the meetingType name
CREATE PROCEDURE getMeetingTypeIDByMeetingType(@meetingType varchar(50))
AS
BEGIN
    SELECT
        fld(meetingTypeID
    FROM
        tbl_meetingType
    WHERE
        fld(meetingType) = @meetingType
END;
GO

CREATE PROCEDURE FindbookingByEmail (@EmailAddress NVARCHAR(255))
AS
BEGIN
    SELECT tbl_booking.fld_bookingID, tbl_userEmail.fld(userID),
tbl_room.fld_roomName, fld(startTime, fld(endTime, fld(userID,
tbl_meetingType.fld(meetingType

    FROM tbl_booking
        INNER JOIN tbl_userEmail ON tbl_booking.fld(userID =
tbl_userEmail.fld(userID
            INNER JOIN tbl_room ON tbl_booking.fld(roomID = tbl_room.fld(roomID
                INNER JOIN tbl_meetingType ON tbl_booking.fld(meetingTypeID =
tbl_meetingType.fld(meetingTypeID

    WHERE
        tbl_userEmail.fld(userID) = @EmailAddress;
END;
GO

CREATE PROCEDURE getAllRoomTypeNames
AS
BEGIN
    SELECT
        fld(roomTypeName
    FROM
        tbl_roomType
END;
GO

CREATE PROCEDURE getErrorReports
AS
BEGIN
    SELECT

```

```

-- Selecting data from error report entity
tbl_errorReport.fld_archived, tbl_errorReport.fld_errorReportID,
tbl_errorReport.fld_reportDate, tbl_errorReport.fld_reportDescription,

-- selecting data from email entity
tbl_userEmail.fld_userEmail, tbl_userEmail.fld(userID),
tbl_room.fld_roomName, tbl_room.fld_roomID,
tbl_inventory.fld_inventoryName, tbl_inventory.fld_inventoryID

FROM tbl_errorReport
    INNER JOIN tbl_userEmail ON tbl_errorReport.fld(userID) =
tbl_userEmail.fld(userID)
        INNER JOIN tbl_room ON tbl_errorReport.fld_roomID =
tbl_room.fld_roomID
            INNER JOIN tbl_inventory ON tbl_errorReport.fld_inventoryID =
tbl_inventory.fld_inventoryID
END;
GO

CREATE PROCEDURE GetAllBookingsFromTodayAndOnwards (@date Date)
AS
BEGIN
    SELECT tbl_booking.fld_bookingID, tbl_userEmail.fld_userEmail,
tbl_room.fld_roomName, fld_startTime, fld_endTime, fld_userName,
tbl_meetingType.fld_meetingType

    FROM tbl_booking
        INNER JOIN tbl_userEmail ON tbl_booking.fld(userID) =
tbl_userEmail.fld(userID)
            INNER JOIN tbl_room ON tbl_booking.fld_roomID = tbl_room.fld_roomID
                INNER JOIN tbl_meetingType ON tbl_booking.fld_meetingTypeID =
tbl_meetingType.fld_meetingTypeID

    WHERE
        tbl_booking.fld_date >= @date;
END;
GO

CREATE PROCEDURE getNumberOfBookingsFromRoomID (@roomID INTEGER)
AS
BEGIN
    SELECT COUNT(fld_bookingID)
    FROM tbl_booking
    WHERE fld_roomID = @roomID
END;
GO

CREATE PROCEDURE GetRoomIDFromName (@roomName varChar(50))
AS
BEGIN
    SELECT fld_roomID
    FROM tbl_room
    WHERE fld_roomName = @roomName
END;

```

```

GO

CREATE PROCEDURE GetAvailableRoomsForDateTimeRange
    @startDate DATE,
    @startTime TIME,
    @endDate DATE,
    @endTime TIME,
    @maxPersonCount INT
AS
BEGIN
    SELECT
        tbl_room.fld_roomID,
        tbl_room.fld_roomName,
        tbl_room.fld_roomMaxPersonCount,
        tbl_room.fld_roomTypeID,
        tbl_room.fld_floor,
        tbl_roomType.fld_roomTypeID AS roomType_ID,
        tbl_roomType.fld_roomTypeDescription,
        tbl_roomType.fld_roomTypeName
    FROM
        tbl_room
        INNER JOIN
        tbl_roomType ON tbl_room.fld_roomTypeID = tbl_roomType.fld_roomTypeID
    WHERE
        tbl_room.fld_roomMaxPersonCount >= @maxPersonCount AND
        NOT EXISTS (
            SELECT 1
            FROM tbl_booking
            WHERE tbl_room.fld_roomID = tbl_booking.fld_roomID
                AND tbl_booking.fld_date >= @startDate
                AND tbl_booking.fld_date <= @endDate
                AND ((tbl_booking.fldStartTime < @endTime AND
tbl_booking.fldEndTime > @startTime)
                    OR (tbl_booking.fldStartTime < @endTime AND
tbl_booking.fldEndTime > @endTime))
                OR (tbl_booking.fldStartTime < @startTime AND
tbl_booking.fldEndTime > @startTime))
        );
END;
GO

```

```

CREATE PROCEDURE getRoomTypeIDFromName (@RoomTypeName varChar(50))
AS
BEGIN
    SELECT *
    FROM
        tbl_roomType
    WHERE
        fld_roomTypeName = @RoomTypeName
END;
GO

```

```

-- Stored procedure to find total booking time in minutes per booking in a given
room
CREATE PROCEDURE getTotalBookingTimePerBooking(@roomID int, @date Date)
AS
BEGIN

```

```

SELECT
    fld_bookingID,
    -- Calculates total booking time per booking
    SUM(DATEDIFF(MINUTE, fld_startTime, fld_endTime)) AS total_booking_time

FROM
    tbl_booking

WHERE
    fld_roomID = @roomID
    AND fld_date = @date

    -- Groups the results by bookingID so each bookingID only appears once (with
    -- the total booking time)
    GROUP BY
        fld_bookingID
    -- Orders the results by bookingID
    ORDER BY
        fld_bookingID

END;
GO

-- Stored procedure to find total booking time in minutes per day in a given
-- room
CREATE PROCEDURE getTotalBookingTimePerDay(@roomID INT, @startDate DATE,
@endDate DATE)

AS
BEGIN
    -- Generates all dates in the DateRange (from startDate - endDate)
    WITH DateRange AS (
        SELECT @startDate AS fld_date
        -- Combines all SELECT results
        UNION ALL
        -- Adds one day to fld_date
        SELECT DATEADD(DAY, 1, fld_date)
        FROM DateRange
        -- Ensures that we only continue adding dates as long as the new date
        -- (after adding one day) is less than or equal to @endDate
        WHERE DATEADD(DAY, 1, fld_date) <= @endDate
    )
    -- Takes a date, sums the total booking time (minutes) for that day in
    -- the selected room - 0 if no booking
    SELECT DateRange.fld_date,
        ISNULL(SUM(DATEDIFF(MINUTE, tbl_booking.fld_startTime,
        tbl_booking.fld_endTime)), 0) AS total_booking_time

```

```

FROM
    -- Joins DateRange with tbl_booking, filtering by room ID
    DateRange
        LEFT JOIN tbl_booking ON DateRange.fld_date = tbl_booking.fld_date
    AND tbl_booking.fld_roomID = @roomID

    GROUP BY
        DateRange.fld_date
    ORDER BY
        DateRange.fld_date;
END;
GO

```

Figur

billede 1 - GitHub Commit eksempel	7
Billede 2: Billede af kanban board efter 2. iteration	9
Billede 3: Domain model for en institution med lokaler der kan udlejes.	13
Billede 4: Domæne model og de tilhørende kardinaliteter.	14
Billede 5: Matrix over interesserter.	17
Billede 6: Risikostyrings 4 søjler.	20
Billede 7: Risiko analyse matrix der inddeler fundne risici i fire risikogrupper ud fra	21
Billede 8: Leawitt´s diamond	22
Billede 9: materiale taget fra business understanding undervisning, PowerPoint "Chaptor 11 and 12"	23
Billede 10: SMART mål, organisatorisk værktøj for at måle virksomhedsudvikling.	24
Billede 11: viser de udvalgte use cases til 1. iteration. De resterende use cases fundet i denne iteration findes i bilag X.....	26
Billede 12: Abstract user i ad-hoc delen af systemet.	27
Billede 13: Endelige E-R diagram (version 1.0) med alle entities og deres tilhørende attributter repræsenteret	32
Billede 14: diagram over Git flow strategi.....	34
Billede 15: viser uddrag af vores prototype	38
Billede 16: BookingComp.java - src/main/java/org/apollo/template/View/UI/BookingComp.java.....	41
Billede 17: ReservedRoomsVBox.java - src/main/java/org/apollo/template/View/UI/ReservedRoomsVBox.java	41
Billede 18: DefualtComponent.java, her ses vores default component som nedarver fra HBox	43
billede 19 - RoomComp.java	44
Billede 20: AvailableRoomsController.java, her skaber vi vores objekt "Booking".	44
Billede 21: AvailableRoomsController.java, her gøres brug af builder pattern.....	45
Billede 22: ChooseTimeController.java, her bygges videre på booking objectet.	45
Billede 23: Booking.java, her ses vores Booking Object, hvor vores set-metoder returnerer 'this', altså 'Booking'.....	45
Billede 24: Eksempel på BuilderPattern. Her vises, hvordan Builder Pattern kan bruges i praksis.	46
Billede 25: DAO.java, DAO Interface	48
billede 26 - DAOAbstract.java	48
Billede 27: InventoryItemDAO.java, readAdd method.....	49
Billede 28: InventoryItems.java, Vores data object.....	50

Billede 29: CreateInventoryItemController.java	50
Billede 30: første del af Stored procedure "getAvailableRoom" – sidste del findes på næste side.....	51
Billede 31: Stored procedure "getAvailableRoom", der returnerer et resultatsæt med alle ledige lokaler på den given dato. Taget fra src/main/java/org/apollo/template/JDBC/databasselInsert.sql	52
Billede 32: Kald af Stored Procedure getAvailablesRooms i JAVA-metoden getAvailablesRooms.	52
Billede 33: Her ses en store procedure, der indsætter en email i tabel tbl_userEmail, hvis den ikke allerede eksistere. Taget fra src/main/java/org/apollo/template/Database/databasselInsert.sql.....	53
Billede 34: Her ses implementeringen af overstående stored procedure. taget fra src/main/java/org/apollo/template/Database/databasselInsert.sql	53
Billede 35: billedet viser de forskellige komponenter i public subscriber pattern	54
Billede 36: viser et billede af det interface, som vores subscribers implementerer	55
Billede 37: viser hvordan BookingCompliteController subscriber til information om BOOKING_INFORMATION-	55
Billede 38 39: viser publish metoden i vores MessageBroker klasse.....	55
Billede 40: viser Messages Broker-klassen i vores applikation. Klassen fungerer som en singleton og er ansvarlig for at håndtere abonnementer, offentliggørelse af beskeder og logikken for abonnement og fjernelse af abonnement på emner.	56
Billede 41: SSD af uc_32.....	62
Billede 42: viser et simpelt UML-diagram over designmønsteret "Strategy Pattern".	63
Billede 43: viser det interface, som vores strategier implementerer.	63
Billede 44: viser TimeContext-klassen i vores applikation	64
Billede 45: viser vores "DayStrategy"-klasse med en final variabel, der definerer antallet af dage i den specifikke strategi.....	65
Billede 46: viser vores "WeekStrategy"-klasse med en final variabel, der definerer antallet af dage i den specifikke strategi.....	65
Billede 47: viser vores "MonthStrategy"-klasse med en final variabel, der definerer antallet af dage i den specifikke strategi.....	66
Billede 48: beforeEach: kaldes før alle tests.	70
Billede 49: teardown metode, bruges i messagesbroker tests src/test/java/org/apollo/template/persistence/PubSub/MessagesBrokerTest.java	71
Billede 50: messageBroker class structure.	71
Billede 51: Publish metode test.	72
Billede 52: Messages broker test, publishers null object	72
billed 53 - JDBC.java, vores JDBC test.	73
billed 54 - UML Athena - https://github.com/eliastherkildsen/ATHENA/blob/developement/src/main/resources/org/apollo/template/images/mainUMLAthena.png	74
billed 55 - SD over Generation af Component for AvailableRoomsController	74

Fodnoter

<https://www.iamachs.com/p/introduction-to-the-publisher-subscriber-pattern-for-cloud-native-application-development/>

