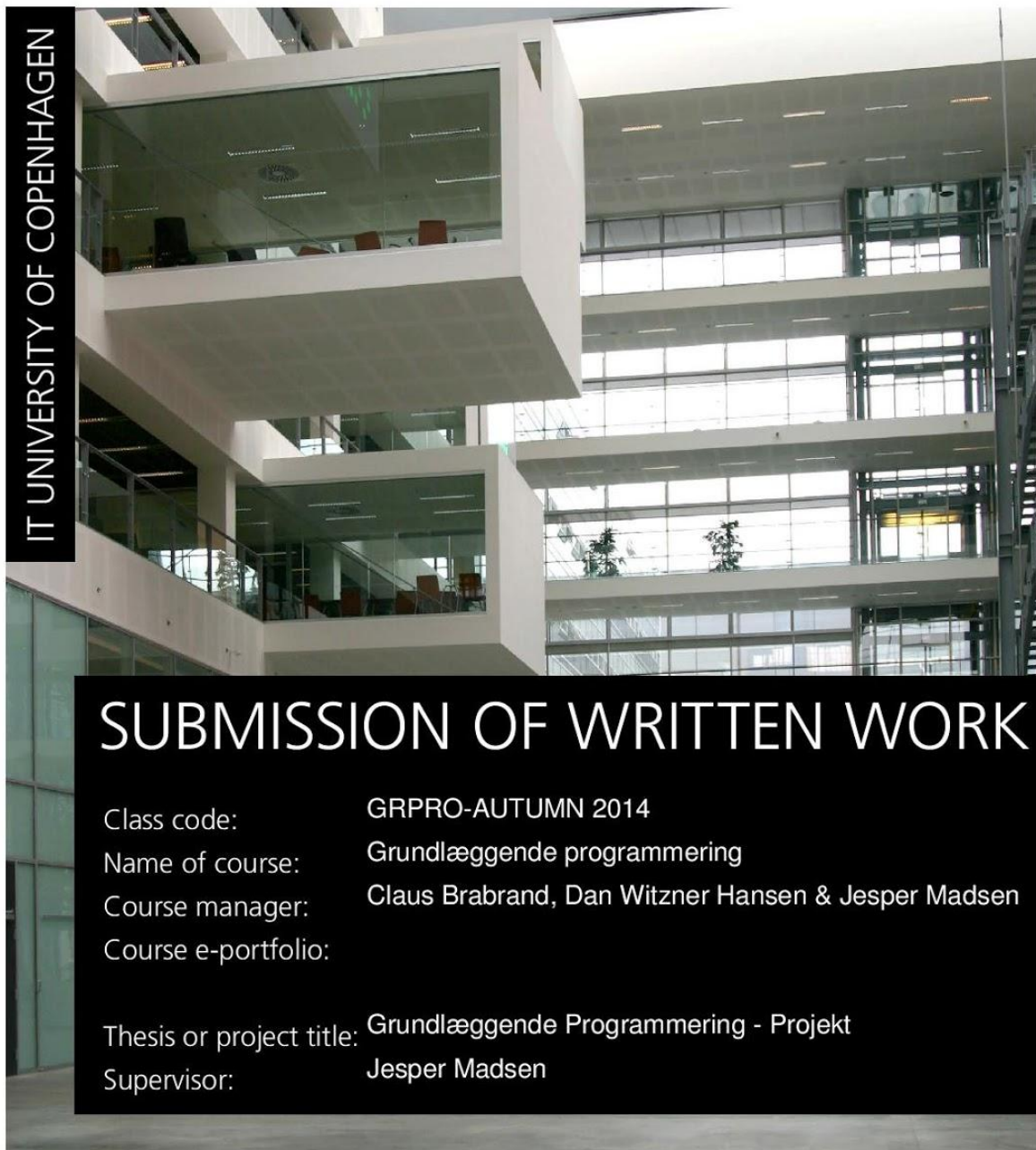


Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand



IT UNIVERSITY OF COPENHAGEN

SUBMISSION OF WRITTEN WORK

Class code: GRPRO-AUTUMN 2014
Name of course: Grundlæggende programmering
Course manager: Claus Brabrand, Dan Witzner Hansen & Jesper Madsen
Course e-portfolio:

Thesis or project title: Grundlæggende Programmering - Projekt
Supervisor: Jesper Madsen

Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1. Dennis Thinh Tan Nguyen	01/04-1993	dttn@itu.dk
2. Thor Valentin Aakjær Nielsen Olesen	14/02-1995	tvao@itu.dk
3. William Diedrichsen Marstrand	25/07-1993	wmar@itu.dk
4. _____	_____	_____@itu.dk
5. _____	_____	_____@itu.dk
6. _____	_____	_____@itu.dk
7. _____	_____	_____@itu.dk

Indholdsfortegnelse

Indholdsfortegnelse	2
Forord	4
Indledning	5
Baggrund	5
Problemanalyse	7
Programmets data	7
Formålet med MVC	8
Substantiv-/verbum metoden	8
Ansvars-drevet design	9
Sammenligning med andre design patterns	9
Iterativ softwareudvikling vs vandfaldsmodellen	9
Flux vs MVC	10
Databasesdesign	11
Fra lokaldatabase til ingen lokaldatabase	12
Fra mange SQL-forespørgsler til få SQL-forespørgsler	12
Prepare statements	13
Databasens indvirkning på klassestrukturen	13
Brugervejledning	14
Teknisk programbeskrivelse	16
Repræsentation og behandling af data	16
Model klasser	16
Reservation klasser	17
Show klasser	17
Customer klasse	17
Theater klasse	18
View, fxml og stylesheet	19
Controller	20
Brugergænsefladen (JavaFX)	21
JavaFX og Scene Builder VS SWING	21
Layout og Komponenter	22
Event Handlers	24
Test af programmet	27
JUnit testing	27
Databaseforbindelse tests	28
Reservationstests	29
Kundehåndteringstests	29
Forestillingstests	30
Testanalyse og exception	31

Dennis Thinh Tan Nguyen	
Thor Valentin Aakjær Nielsen Olesen	
William Diedrichsen Marstrand	
Databaseafhængighed og mock-ups	32
Konklusion	33
Refleksion over arbejdsproces	34
Litteratur	36
E-Litteratur	36
Undervisningsmateriale	37
Forelæsninger	37
Bilag.....	38
Mockups.....	38
Mockup 1	38
Mockup 2.....	39
Mockup 3.....	40
Mockup 4.....	41
Mockup 5.....	42
Mockup 6.....	43
Flowchart	44
Flowchart af reservationssystem	44
Flowchart ved benyttelse af lokal database	45
Flowchart af første data struktur.....	46
Dataflow og relationer mellem tables i databasen.....	47
Gantt Diagram.....	48
Noun/Verb Oversigt	49
Andre programmeringsprincipper	50
Indkapsling (eng.: encapsulation).....	50
Abstraktion (eng.: abstraction)	50
Uddelegering (eng: delegation).....	50
Dokumentation, kommentarer, navnekonventioner og exceptions m.m.	51
Kodeændringer (fjernet)	52

Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

Forord

Da vi startede på grundlæggende programmering i starten af første semester, havde ingen af os regnet med den udvikling, vi ville gennemgå. Vi gik fra ikke, at have en forståelse for hvad et simpelt felt var til at kunne programmere et simpelt reservationssystem med databaseintegration.

Vi vil gerne takke Claus Brabrand og Dan Witzner Hansen for deres lærerige forelæsninger, Jesper Wendel Madsen for hans gode råd til projektet og alle TA's for deres vejledning.

Bemærk at der kan herske inkonsistens mellem brug af termer på engelsk og termer oversat på dansk, da det danske ordforråd på sin vis er begrænset indenfor netop dette felt, men vi har efterstræbt af bruge danske termer, hvor det umiddelbart er muligt, idet rapporten er på dansk.

Selve systemet, dets dokumentation og kommentarer er på engelsk, da det meste materiale/indhold i softwarebranchen er skrevet på engelsk. Det kan derfor med fordel indlæres fra start i målet om at blive dygtigere softwareudviklere.

Indledning

I forbindelse med projektet i grundlæggende programmering har vi fået stillet til opgave at analysere et problem i form af et reservationssystem til en mindre biograf og derefter programmere et softwaresystem, som kan løse dette. Denne rapport er derfor skrevet med henblik på at præsentere systemets formål, opbygning og virkemåde samt testning af programmet og refleksion over disse test og den samlede arbejdsproces.

Baggrund

Projektet omhandler udfærdigelsen af et billetreservationssystem, som kan bruges af en ekspedient. Ekspedienten betjener en billetluge og telefon i en biograf af mindre størrelse. Kunden bestiller via ekspedienten enten pr. telefon eller foran billetlugen. Det er derfor værd at bemærke, at bestillinger fra kunden ikke kan tilgås via nettet, så samtidige opdateringer til databasen er unødvendigt.

Krav til biografen:

- 1) Biografen har kun én billetluge og skal derfor ikke tage hensyn til flere ekspedienter, der laver samtidige ændringer, som risikerer at overskrive hinanden medmindre, man opdaterer databasen.
- 2) Biografen består af flere sale og viser forskellige film til forskellige tidspunkter
- 3) En forestilling består af en film, sal tidspunkt og dag.
- 4) Salene kan have et forskelligt antal rækker med et forskelligt antal stole pr række.
- 5) Antallet af sale og deres struktur skal fremgå af databasen og ikke være indkodet i programmet
- 6) Forestillinger skal fremgå af databasen og må ikke være indkodet i programmet
- 7) Eksempler på sale og forestillinger må godt opdigtes i databasen

Tilgang til løsning af use cases

Vi er i gruppen blevet enige om, at eftersom ekspedienten ikke kan forventes at have nogen særlige IT-kundskaber, og arbejdet skal ske hurtigt og effektivt, da bør softwaresystemet være simpelt og skabe et godt overblik. Vi har derfor valgt at fokusere primært på løsningen af ekspedientens umiddelbare kerneopgaver:

1. Oprette en reservation til en eller flere personer.

Dette indebærer:

- Sædelokation
 - Forestilling
 - Tidspunkt
2. Afbestilling/annulering af en reservation
 3. Finde forestillinger, der vises i biografen
 4. Finde eksisterende reservationer ud fra kundeid

Med dette udgangspunkt in mente har vi valgt, at alle arbejdsopgaver løses manuelt af ekspedienten. Dette forklarer hvorfor, der ikke er implementeret funktioner, som gør det muligt at foretage “samtidige arbejdsopgaver”, eksempelvis at oprette en reservation til to forskellige film samtidigt (nice to have¹).

¹ En funktion der er “smart” men ikke nødvendig i løsningen af problemet og kan udelades til senere implementering uden direkte konsekvenser, i kontrast til “must have features”.

Problemanalyse

Programmets data

Bookingsystemet er bygget op efter designmønstret, MVC(Model View Controller) Pattern, som anvendes i komplekse systemer, hvor der kan være behov for at separere data (Model), præsentationen heraf (View) og håndteringen af brugerinput, der har en indflydelse på viewets og modellens dataændringer (Controller). Forklaringen på dette valg bunder primært i ønsket om at adskille backend og værdier/data fra frontend, der repræsenterer programmet.

Princippet omtales også “Separation of Concerns” i den forstand, at designvalget er med til at adskille problemer og tilhørende ansvarsområder imellem forskellige aktører, der sammen løser den overordnede problemstilling. I vores tilfælde et reservationssystem styret fra en ekspedient, der skal kunne registrere reservationer over telefon og ved fremmøde i systemet til forestillinger i en mindre biograf. Der foretages en klar opdeling mellem domæne objekter og præsentationen af disse objekter.

Domæne objekterne repræsenterer den virkelige verden gennem en samling objekter, der spiller arbejder sammen om at få udført en række arbejdsopgaver og har forskellige ansvarsområder i it-systemet. I vores tilfælde er domænet en biograf, hvori der er en lang række objekter såsom sæder, sale, film, tidspunkter m.m., der både er fysiske og abstrakte.

Præsentationen af disse objekter overlades til den grafiske brugergrænseflade (GUI), her programmet som ekspedienten interagerer med for at foretage reservationer til forestillinger. Der bør ikke indtræde nogen direkte kobling mellem domænets objekter og GUI. Objekterne skal derfor kunne arbejde selvstændigt uden henvisning til præsentationen, så de kan bruges i forskellige frameworks.

Formålet med MVC

I dette projekt har der været behov for objekter, der etablerer en kommunikation mellem ekspedienten og domænemodellen. Samtidig er der behov for at gemme ændringer foretaget på disse objekter og de felter/attributter de besidder, hvilket gøres i en mysql database via ITU. Derigennem opnås en klarere og renere kildetekst, der sammen med tredelingen gør det nemt at genbruge kode og udvide systemet løbende, fordi systemet har en lav kobling. Dette kombineret med en høj sammenhæng (cohesion) gør systemets kode læseligt, koden genbrugelig og designet stabilt, så man ikke risikerer at skulle ændre en masse kode undervejs. Til slut er brugen af moduler til at separere brugerflade, programlogik og data med til at muliggøre tests, så man udefra kan få et indtryk af, om programmet rent faktisk opfylder de krav, der er tilknyttet og håndterer ændringer i data ordentligt.

Substantiv-/verbum metoden

For at kunne klarlægge systemets objekter, klasser og sammenhængen derimellem, har vi benyttet os af verbum/substantiv metoden. Her identificeres klasser, objekter, deres associationer og indbyrdes samspil ved at bruge substantiver til at beskrive “ting” (fx. mennesker, sæder, film m.m.) og verber til at beskrive “handlinger” såsom bestille, forbinde, hente osv. De forenklede sprogstrukturer tilnærmer henholdsvis klasser/objekter og handlinger udført på disse objekter i it-systemet og findes nemmest ved at tage udgangspunkt i problembeskrivelsen samt use cases, der oplyser, hvad programmet bør kunne og hvordan det bør reagere situationelt (Bilag 10.4). Samtidig kan man ud fra oversigten i bilag x observere, hvor objekter gentager sig såsom “Film”, der skal samarbejde med flere objekter på en gang, fx. reservation, sal og forestilling.

Ansvars-drevet design

Indenfor programmering og god kodestil har vi lært, at en lav kobling og få afhængigheder er ønskværdigt. Vi har derfor i projektet efterstræbt, at hver klasse har deres eget ansvar, der handler om at vide ting og gøre ting. Med andre ord, når en klasse ejer nogle felter, så har den ansvar for beregninger/ændringer på disse. Eksempelvis har vi en `SeatModel`-klasse, der udelukkende håndterer sædets tilgængelighed ved at tilgå databasen. Dette er alt sammen med henblik på en øget læselighed, kode-genbrug og design-stabilitet, så vedligeholdelse og fremtidigt arbejde på projektet udefra er muligt for andre programmører, eksaminatorer og ikke mindst os selv.

Sammenligning med andre design patterns

Som tidligere nævnt har vi genbrugt MVC design pattern. Dette er blevet gjort med henblik på at organisere projektets struktur og kode, og samtidig er det en god løsning, der er blevet brugt før i mange andre sammenhænge, som har vist gode resultater (fx. til spil). Da designmønstret henvender sig til et typisk problem og beskriver en generel løsning dertil kan MVC bruges i mange forskellige sammenhænge - her i opbygningen af et reservationssystem til en biograf, hvor ekspedienten skal have adgang til en oversigt af forestillinger og reservationer (View), som skal gemmes i en database, hvor ændringer via brugerinput (Controller) udføres på objekter (Model).

Iterativ softwareudvikling vs vandfaldsmodellen

Vi har under hele projektforsløbet arbejdet iterativt på systemets grundform baseret på MVC og den indledende klassestruktur (Bilag 10.2.3). Systemet er blevet forbedret gradvist på baggrund af de udfordringer og den feedback, som vi har mødt. Dette står i kontrast til fx. “vandfaldsmodellen”, hvor man kigger på problemstillingen (analyse), klassestrukturen og metoder (design), laver koden (implementation), tester enkelte dele (unit testing) og til sidst afleverer noget (integration testing).

Ved at benytte sig af denne fremgangsmåde, risikerer man at ende med et system, der ikke opfylder det egentlige behov, fordi man ikke imødekommer

samtidige ændringer, der tilgodeser brugerens behov. Dette problem minimeres ved at arbejde iterativt, fordi man gradvist “skyder” sig tættere på målet ved at løse de vigtigste opgaver først og så måske lave en knapt så perfekt løsning.

Herudover er andre teknikker og arbejdsværktøjer også blevet brugt, hvilket kan ses i bilag.²

Flux vs MVC

Da vi valgte et strukturelt designmønster til systemet var det velvidende, om at der findes utallige andre designmønstre, men eftersom vi gennem kurset blev introduceret til MVC og fik det anbefalet af vores lektor til projektet holdt vi fast i designet. Til gengæld kan man diskutere, om hvorvidt det er hensigtsmæssigt i alle situationer. En mere tidssvarende og nærliggende kandidat til MVC er Flux, som Facebook bruger. Løsningen stræber at skabe en et ensidigt dataflow i et system i modsætning til MVC, hvor dataflowet kan gå begge veje, fra model til view og vice versa. MVC er et nærliggende designvalg, hvis der er tale om et mindre system ligesom vores, men så snart systemet bliver større, vil kompleksiteten øges dramatisk gennem modeller og deres korresponderende views. Herved ender man med et system, der er svært at forstå og debugge, da dataændringer går begge veje. I Flux erstattes controlleren med en “Dispatcher”, der styrer hvordan data bliver opdateret, når en handling aktiveres. Opdateringen registreres i View før andre handlinger kan aktiveres og “dispatcheren” kan ligefrem nægte metodekald før en forhenværende metode er kørt færdig. Dette er med til at mindske “tosidige” dataændringer i en klassestruktur, hvilket kan være en fordel i større komplekse systemer, hvor der kan optræde flere afhængigheder end normalt. Som nævnt før er MVC dog stadigvæk en udmærket tilgang til mindre systemer, hvor størrelsen ikke er en bærende faktor endnu sammenlignet med Facebook, der er en kæmpe organization med en stor kodedatabase.³

² Bilag 10.5

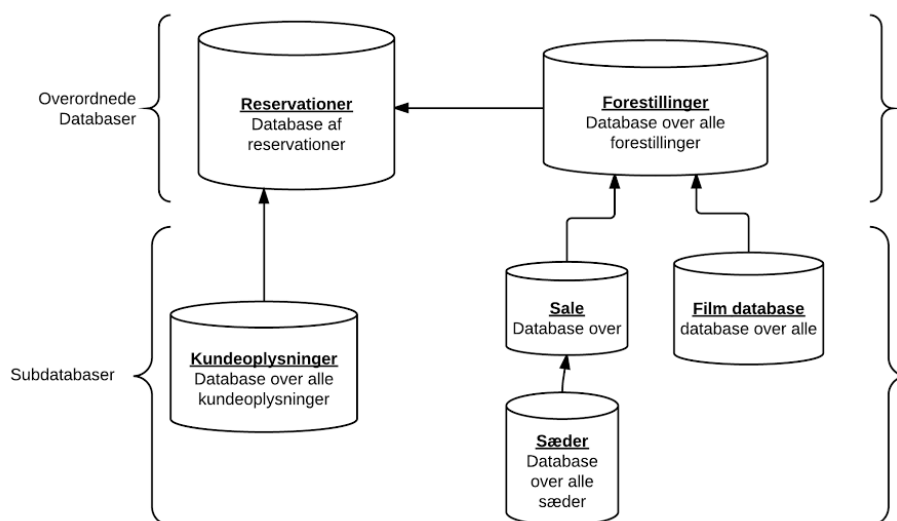
³ <http://www.infoq.com/news/2014/05/facebook-mvc-flux>

Databasedesign

Et billetreservationssystem arbejder med store mængder af data, som man skal kunne tilgå effektivt. Her kan man med fordel benytte en database, der er opdelt efter relevante tabeller, da man med en database effektivt kan oprette, hente og skabe relationer mellem store mængder af data.

Databasen som vores reservationssystem skal tilgå bør struktureres i to dele. Den ene del repræsenterer overordnede elementer af en biograf, heraf:

- Reservationer fra kunder
- Forestillinger med forskellige film på forskellige tidspunkter



Figur 4.1 – Relationerne i databasen

Den anden halvdel repræsenterer “subelementer” af de overordnede elementer, heraf:

- Kundeoplysninger
- Sædeoplysninger
- Filmoplysninger
- Sale i en biograf

Ved at opdele dataen, giver det os mulighed for at specificere de overordnede elementer og samtidig have mulighed for at ændre hvert enkelte subelement uafhængig af hinanden.

Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

Dette er afgørende i tilfælde af at biografen får tilføjet eller fjernet sale, sæder, film m.m., hvorved man hurtig vil kunne ændre dette med minimal risiko for redundans.

Denne relation mellem overordnede elementer og subelementer er illustreret på (figur 4.1. & bilag 10.2.4) Databasestrukturen tager udgangspunkt i en reservation, der er tilknyttet en forestilling og nogle kundeoplysninger. Alle disse bestanddele har hver deres tabel, som de overordnede tabeller kan joine gennem deres primære nøgler.

Hvert element tildeles en unik nøgle for at kunne skelne i data i databasen og undgå redundant data. For eksempel bliver hver kunde tildelt en unik nøgle i tilfælde af at to kunder har det samme navn. Disse unikke nøgler gør det mulig for systemet at skelne mellem de to kunder, og derved oprette passende reservationer ved brug af nøglerne som identifikation.

Fra lokaldatabase til ingen lokaldatabase.

Oprindeligt startede vi ud med at ville benytte os af en lokal database, der delvis lagrede data hentet ned fra en globale database (Se bilag 10.22 for flowchart). Herfra skulle man kunne oprette reservationer vha. data fra den lokale database. Et praktisk problem, der kan opstå i dette tilfælde er dog strømsvigt eller overskrivninger af lokal forankret data ved samtidig brug af flere systemer med egne lokaldatabaser. I så fald kan man risikere, at ens data bliver overskrevet, redundant eller i værste tilfælde mistets. Derfor vil vi gerne ud fra et pragmatisk standpunkt have adskilt data fra reservationssystemet, således at man kun kan hente eller oprette data fra et sted.

Fra mange SQL-forespørgsler til få SQL-forespørgsler

Under udviklingen af reservationssystemet og dets måde at tilgå databasen, stødte vi på et problem, som vi ikke havde forudset. Paradoksalt har vi separeret alle metoder, hvilket også indebærer metoder der omhandler databasekald, i så høj grad, at vi er endt med en masse metoder, der kan få udføre få, specifikke queries. Som udgangspunkt er dette godt, men problemet er at de i sidste ende oprettes alle sammen og skiftevis skaber en forbindelse til databasen, som kan risikere at blive overbelastet eller køre langsomt.

Løsningen har derfor været at samle mange af disse “delmetoder” eller SQL-forespørgsler i et samlet kald, således at man kan hente, ændre eller indsætte større mængder af data på en gang. Dermed skal forbindelsen til databasen kun oprettes få gange, hvilket minimerer kørselstiden markant.

Prepare statements

Når man arbejder med databaser, opstår der en risiko for, at ondsindede requests igennem navne og telefonnumre til databasen benyttes. For eksempel vil en bruger kunne indsætte SQL-forespørgsler, som eksempelvis kunne slette hele reservationstabellen. Vi er dog kun blevet introduceret til det kort og har derfor fravalgt det i dette projekt, hvor vi stedet har taget højde for andre krav ved fremtidig udvikling.

Databasens indvirkning på klassestrukturen

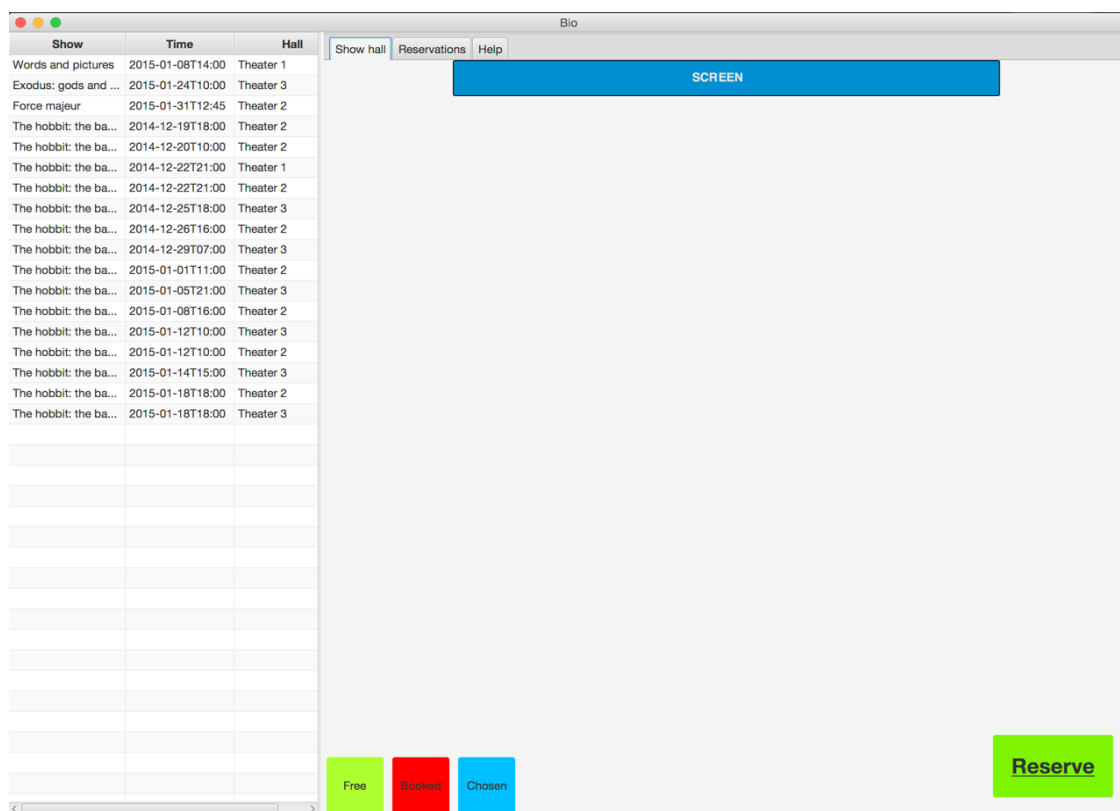
Det er værd at bemærke, at klassestrukturen har ændret sig markant i det endelige system, hvor vi har fået et større kendskab til databaser og SQL. Eksempelvis har vi lagret oplysninger på sæder, sale, forestillinger og kunder i en server database frem for at gøre det lokalt med henblik på at sikre, at det kun er vores database, der gemmer data. Selvom opgavebeskrivelsen oplyser, at systemet kun skal benyttes af én ekspedient, kunne man forestille sig, at ændringer i reservationer i værste tilfælde ville blive overskrevet, hvis flere ekspedienter benytter sig af systemet samtidig. Dette er gældende i de fleste biografer i dag og er derfor værd at medtage, hvis systemet skal fremtidssikres til større biografer. Til slut er det vigtigt at adskille lagringen og håndteringen af data i tilfælde af strømsvigt, hvorved lokale ændringer risikerer at gå tabt. For at gøre dette muligt har det været nødvendigt at oprette nye kolonner i databasen med informationer tilknyttet kunde, forestilling, reservation, sal og sæde, som vores database tildeler et unikt ID.

Brugervejledning

A: Ekspedienten præsenteres først for forsiden herunder. Herfra kan ekspedienten få et overblik over de forestillinger, der går i biografen ved at kigge i venstre tabel, som viser filmtitlen (Show), tidspunktet hvorpå filmen starter (Time) og hvilken sal den spiller i (Hall).

Ekspedienten har følgende muligheder vha. museklik:

- B1) Tryk på “Reserve”, hvor et popup vindue fremkommer men ingen reservation er mulig
- B2) Tryk på fanen “Reservations” for oversigt over eksisterende reservationer
- B3) Tryk på “Help” for informationer til hjælp omkring hvordan programmet benyttes.
- B4) Tryk på en forestilling i tabellen, hvorved en ny oversigt over sædepladser for salen vises i “Show hall” fanen (valgt ved start af programmet).



Dennis Thinh Tan Nguyen

Thor Valentin Aakjær Nielsen Olesen

William Diedrichsen Marstrand

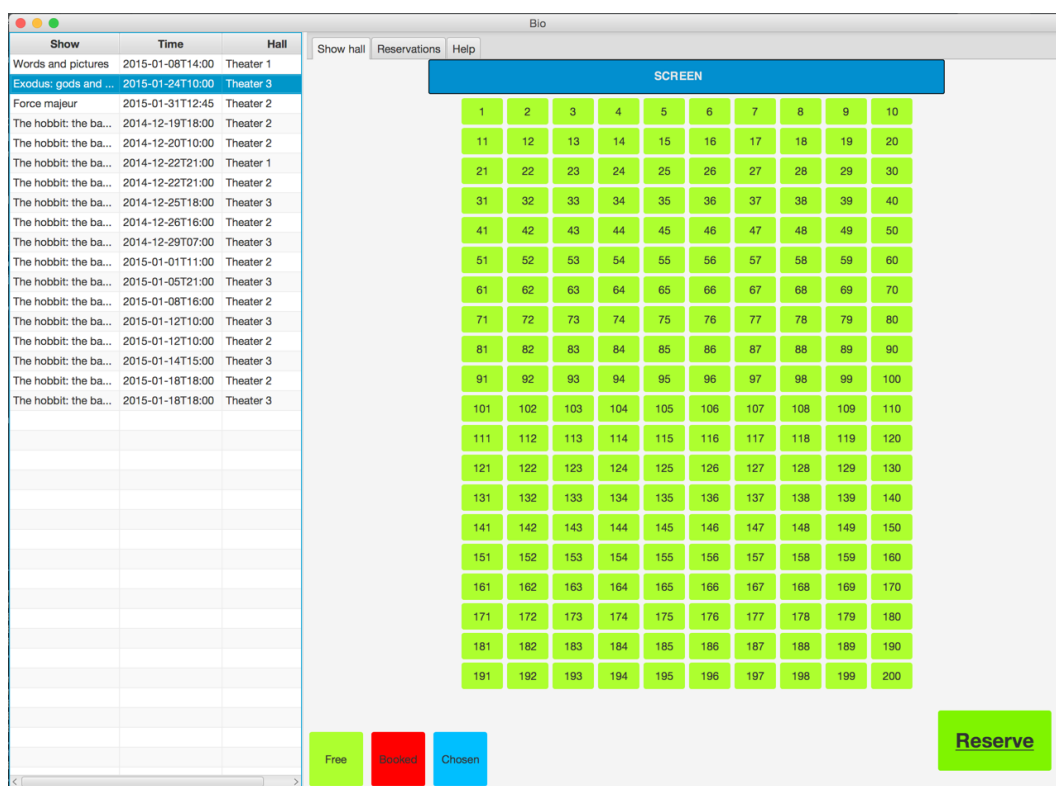
Følgende scenarier kan forekomme ved interaktion med sæderne i ønsket om en reservation:

C1) Ekspedienten trykker på et eller flere grønne sæder, der skifter farve til blå for at indikere, at de er klar til at indgå i en ny reservation.

D) Ekspedienten trykker “Reserve” i nederste højre hjørne. Herfra bliver ekspedienten bedt om at oplyse kundens navn og telefonnummer i et nyt popup vindue tilknyttet reservationen. Dette gemmes i databasen og kan nu tilgås i fanen “Reservations”, som henter reservationsoplysninger fra databasen.

C2) Ekspedienten trykker på et eller flere røde sæder, hvorved intet sker, da disse sæder i forvejen er reserveret til en anden kunde.

C3) Såfremt en kunde ønsker at ændre en eksisterende reservation, skal den oprindelige reservation først slettes manuelt af ekspedienten, hvorefter en ny reservation oprettes (i stedet for at overskrive den oprindelige).



Teknisk programbeskrivelse

Repræsentation og behandling af data

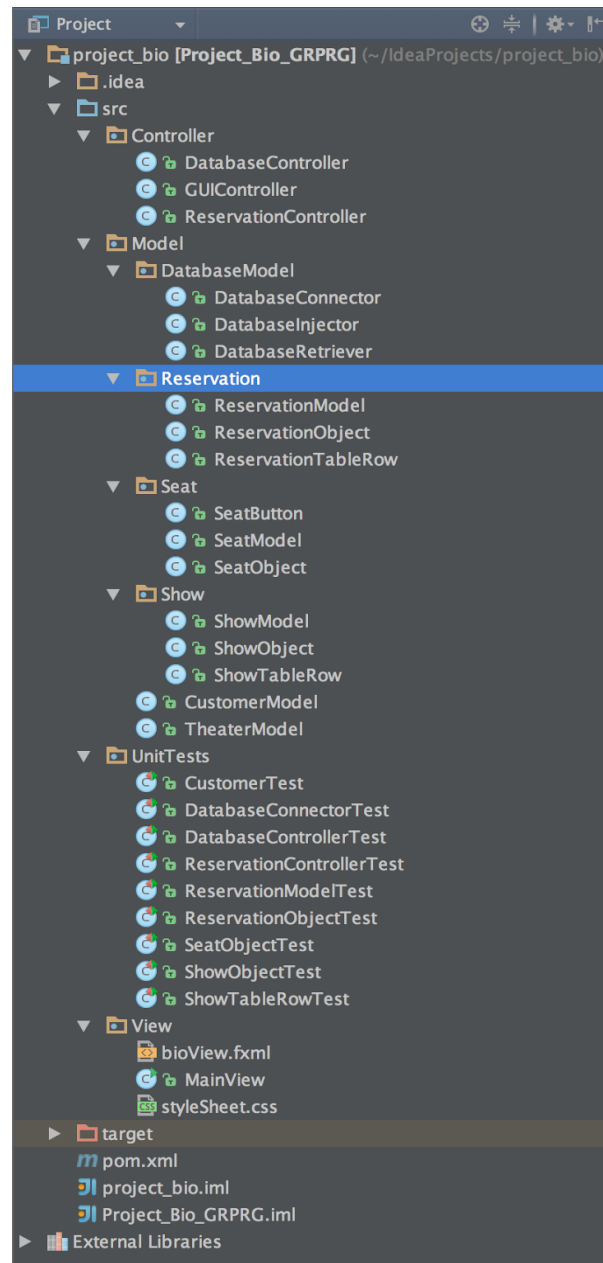
Programmet er opbygget efter et MVC Pattern og ud fra et **ansvars-drevet design**. For at skabe et overblik blev de forskellige klasser derfor inddelt i pakker afhængig af ansvarsområde.

(se billede til højre).

Model klasser

Klasserne i pakken DatabaseModel står for forbindelser til databasen, hvor **DatabaseConnector** opretter og lukker forbindelsen til databasen. Dette sker ved hjælp af to metoder, hvor den ene er “*initConnection()*” som opretter forbindelse til database og den anden “*closeConnection()*” som sørger for at forbindelsen termineres.

DatabaseInjector indeholder metoder, der håndterer forespørgelse om at indsætte data i databasen og extender **DatabaseConnector** på samme måde som **DatabaseRetriever**, der håndterer forespørgelser om at hente data.



Reservation klasser

Modellerne i pakken **Reservation** håndterer alle reservationer i databasen. **ReservationModel** står for at hente, oprette og slette reservationer i databasen. Når information hentes ned fra databasen lagres det i instanser af **ReservationObject**, som gemmes i en ArrayList, der via **ReservationTableRow** wrappes med **SimpleProperty**. Denne håndtering af reservationer er uhensigtsmæssig, da brugen af **ReservationObject** er overflødig i og med, at man ville kunne have overført dataene direkte til **ReservationTableRow**. Grunden til at dette ikke blev gjort skyldes, at **ReservationObject** var implementeret i den første udgave af designet, som blev ændret inde i forløbet. Grundet manglende tid måtte klassen derfor i stedet indbygges i det nye design. Pakken **Seat** indeholder klasser med ansvar for dataene på sæderne, som hentes fra databasen. Den data, som **Seat** objekterne indeholder bruges til at skabe det grafiske view for sæderne i GUI'en, og er forbundet til objekter fra klassen **GUIController**, som så kan informere **View** om ændringer for sæderne.

Show klasser

ShowModel håndterer data om forestillingerne, og den kan hente ID på alle forestillinger med ledige sæder. Derudover kan den indsætte nye shows ind i databasen, hvilket dog ikke nødvendigvis er afgørende for ekspedienten. Det er nemlig ikke en funktion, som er tiltænkt vedkommende, men derimod blot en nem måde, hvorpå nye forestillinger i biografen oprettes.

ShowObject bruges til at lave objekter ud fra show data fra databasen, og var på samme måde som **ReservationObject** indtænkt i et design, der måtte ændres med kort tid tilbage, og blev derfor også vedligeholdt. **ShowTableRow** er klassen til at wrappe vores show data med **SimpleProperty**, der kan lyttes på i en ObservableList, så dataene i GUI'ens show tabel holdes opdateret.

Customer klasse

CustomerModel håndtere alle kunderelaterede opgaver og instanser af klassen kan oprette og slette nye kunder samt hente information om eksisterende ud fra parametre, som bruges i queries til databasen.

Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

Theater klasse

TheaterModel indeholder en væsentlig metode **setSeats()**, der returnerer et **GridPane** med **ToggleButton**s, som repræsenterer hvert sæde til en specifik forestilling i den valgte biografsal. **ToggleButton**s får et sæde ID ud fra deres placeringen, en event handler, der håndterer klik i GUI'en samt en farve og dertil en status ud fra oplysninger i databasen, så de enten er ledige eller reserverede. Denne metode sikrer en dynamisk visning af biografسالene, når sæder skal vælges.

```
public GridPane setSeats(ArrayList<SeatObject> list)
{

    int rows = list.get(0).getRowsInTheater();

    int seats = list.get(0).getSeatsInEachRow();

    final GridPane root = new GridPane();

    root.setId("TheaterSeats");
    root.setPadding(new Insets(5));
    root.setHgap(5);
    root.setVgap(5);

    root.setAlignment(Pos.CENTER);

    final ToggleButton[][] buttons = new ToggleButton[rows][seats];
```

(screenshot fortsat på næste side)

Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

```
int counter = 0;
for (int y = 0; y<rows; y++)
    for(int x = 0; x<seats; x++)
    {
        String styleClass;
        if(list.get(counter).getIsAvailable())
        {
            styleClass = "free";
        }
        else
        {
            styleClass = "reserved";
        }
        //Add seat_id as property to seatButton
        final SeatButton button = new SeatButton(list.get(counter).getSeat_id());
        button.setText(""+(counter+1));
        buttons[y][x] = button;
        buttons[y][x].getStyleClass().add(styleClass);
        buttons[y][x].setPrefSize(50,50);

        root.add(buttons[y][x], x, y);

        counter++;

        buttons[y][x].setOnAction(event -> {
            GUIController guiController = new GUIController();
            guiController.onSeatPressEvent(button);
        });
    }
return root;
}
```

View, fxml og stylesheet

Pakken **View** indeholder klassen **MainView**, hvorfra programmet loades ved at sætte stage og scene for JavaFX samt loade fxml-filen og sætte data via **GUIController** klassen. Fxml-filen kan redigeres via det visuelle UI værktøj JavaFX Scene Builder, og er knyttet til vores **GUIController**, som er sat til at være controllerklasse for fxml-filen, der indeholder beskrivelsen til viewet for vores javaFX applikation og loades ind i MainView.

Controller

Controller klasserne håndterer forbindelsen mellem Model og View og består af tre klasser **DatabaseController**: Står for kommunikation med **DatabaseRetriever** og **DatabaseInjector** ved at sende queries til disse, når databasen skal tilgås. Den indeholder alle de nødvendige metoder til at sende forespørgsler(queries) til databasen, som enten vil ændre databasens indhold eller hente indhold fra databasen til klienten. Klassen udfører dog ikke kodelinjerne selv, men kalder de passede metoder fra Databaseklasserne nævnt ovenfor. **GUIController** kommunikerer mellem **View** og **Model** klasser, så GUI'en hele tiden kan opdateres, hvilket gøres ved FXML tags, der gør metoder og felter tilgængelige for fxml-filen. Brugen af properties gør det hele tiden muligt at lytte efter ændringer. **ReservationController** sender besked til reservation model klasserne, når information skal ændres eller hentes og bruges i **GUIController** klassen til oprettelse og sletning af reservation, så database og GUI begge opdateres i forhold til hinanden.

Grundet et manglende kendskab til JavaFX startede vi med et udgangspunkt, som resulterede i en uheldig kodelinje. Dette måtte vi rette op på senere hen i forløbet, hvor vi var nødsagede til at lave omfattende ændringer i source coden. Som konsekvens gik det ud over tidsaspektet og den mængde tid, som vi havde troede, det ville tage at færdiggøre projektet. Dette endelige system indeholder derfor også enkelte svært gennemskuelige metoder, som i stedet er forsøgt forklaret med frekvent kommentering, for at øge gennemskueligheden på bedste vis. Dette bryder dog med det vores ønske om et **ansvars-drevet design** fra MVC Pattern, som vi ellers har forsøgt at arbejde ud fra, hvor klasser så vidt muligt håndterer hver deres ansvarsområde og metoderne lige så opdeles i mindre metoder, hvilket er gjort i et forsøg på en mere organiseret og derved læsbar kode med en lav kobling.

Brugergrænsefladen (JavaFX)

JavaFX og Scene Builder VS SWING

Til udfærdigelsen af systemets brugergrænseflade blev det visuelle layout tool javaFX Scene Builder benyttet, da dette blev anbefalet af vores supervisor. Her blev videomateriale med en guide stillet til rådighed, så vi hurtigt kunne opbygge en viden om de grundlæggende funktioner, som vi derefter byggede videre på i vores arbejde.

Med javaFX Scene Builder kan en brugergrænseflad opbygges ud fra et drag and drop-princip, hvor de forskellige komponenter kan trækkes til et arbejdsområde, hvorfra brugergrænsefladen opbygges visuelt. Denne arbejdsform er specielt effektiv, hvis man ikke har store erfaringer med kodning af GUI. JavaFX tilskynder nemlig en til at prøve sig frem rent visuelt og giver øjeblikkelig respons på ens ændringer sammen med et indtryk af sammenhængene imellem komponenterne. Komponenternes properties kan desuden let modificeres, CSS stylesheets kan tilføjes og JavaFX Scene Builder genererer automatisk den tilhørende FXML-kode, som så kan kombineres med det ønskede Java projekt ved at koble UI'en til applikationen.

Visuelle layout tools tilbydes også til SWING som bl.a. Netbeans Swing GUI Builder, men JavaFX har dog nogle interessante egenskaber i forhold til, hvad der grafisk er muligt. Da SWING er bundet til CPU'en, som ikke er specialiseret til kørsel af grafik, skalerer SWING ikke i samme omfang som JavaFX, der benytter grafikortet og derfor kan køre grafiske programmer hurtigere. Da vores program ikke er særlig grafisk tungt, ville SWING stadig være en mulighed i dette tilfælde. Eftersom JavaFX umiddelbart er fremtidssikret til at arbejde med større grafiske programmer, og desuden er blevet udmeldt som den nye standard, vil det nok alligevel være en stor fordel for os at opbygge et godt kendskab hertil. Af disse grunde har vi valgt JavaFX til at skabe brugerfladen.

Layout og Komponenter

Vores brugerflade design bygger på en idé om overskuelighed og effektivitet, så ekspedienten let og hurtigt kan betjene så mange kunder som muligt. Vi startede med forskellige mockups, og arbejdede os hen imod et endeligt resultat, der implementerede enkelte dele fra de forskellige mockups.⁴

For at opbygge et overskueligt inddelt layout brugte vi et **BorderPane Layout**, hvori vi nastede flere forskellige containers, så relateret data blev grupperet. Da det valgte contentpane er et **BorderPane**, gav det god mulighed for opdelingen af et Table til shows og et Tab layout indeholdende tre faner, hvor ekspedienten kan se salen med reserverbare sæder, reservationerne for det valgte show og finde hjælp.

Herudover blev en **MenuBar** også overvejet og forsøgt implementeret, men efter grundig overvejelse blev den fjernet bevidst for at undgå unødige distraktioner og i stedet fokusere på hovedopgaven hos ekspedienten, som er at foretage en reservation for en kunde til en given forestilling i en mindre biograf med forskellige sale af forskellig størrelse. Vi udeladte desuden at fokusere på et visuelt flot design og gik i stedet efter et simpelt design, som blot skulle give mulighed for at løse arbejdsopgaverne og ikke nødvendigvis virke tiltalende for ekspedienten. Et mindre problem i forhold til repræsentation af filmtitler i venstre **Table**, hvor Show navne længere end 20 karakterer ikke kan ses fuldstændigt.

Show	Time	Hall
Words and pictures	2015-01-08T14:00	Theater 1
Exodus: gods and ...	2015-01-24T10:00	Theater 3

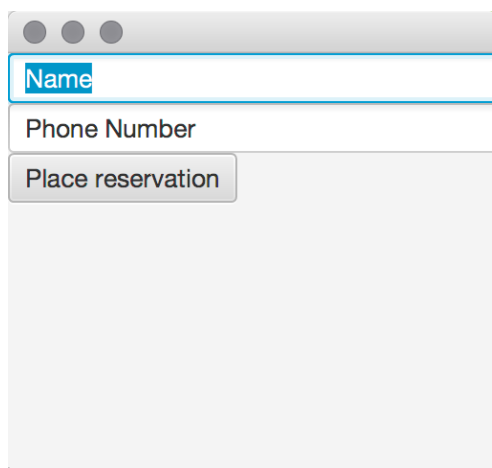
Nederst i venstre hjørne er der indsat tre farvede kvadrater med en tekst, der indikerer, hvilken status et givent sæde har afhængig af dets tilgængelighed i en reservation. Sædet er tilgængeligt hvis det er grønt, reserveret i forvejen hvis det er rødt eller markeret til at indgå i en ny reservation, hvis det er blå. Sæderne er desuden placeret foran et blå rektangel, der forestiller skærmen i biografen, så det bliver let at kende rækkernes placering.

⁴ Bilag 10.1

Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

Dette anså vi dog ikke som et væsentligt problem, i forhold til vores primære fokus på løsning af arbejdsopgaverne, da vi antog, at man i så tilfælde stadig kan resonere sig frem til showet.

De væsentligste benyttede komponenter er **Tables** til visning af shows og reservationer, da det er en effektiv måde til at skabe overblik, som ikke kræver meget plads i GUI'en. **Buttons** blev brugt til at foretage reservationer, hvor et specialiseret reservations-pop-up--vindue kom frem, hvilket blev gjort, fordi muligheden for reservation kun var nødvendig når pladserne var valgt, og derfor kun skulle være tilgængelig på dette tidspunkt.



ToggleButtons anvendtes til repræsentation af biografsæder, da disse knapper kan ændre tilstand afhængig af, om de er valgte eller ej, hvilket netop passer til en reservation. **TextArea** blev brugt til indskrivning af hjælp til ekspedienten, **Text** og **Rectangle** til visning af, hvor lærredet befinder sig i salen. For **Tables**, **Buttons** og **ToggleButton**s tilføjedes et fxid i Scene Builder, som blev brugt som reference i den egentlige source code.

Event Handlers

I JavaFX kan events af forskellig art let knyttes til en valgt komponent. Dette sker blot ved at skrive navnet på den metode, som skal kaldes når eventet forekommer i en kolonne for det ønskede event, der findes i højre side under drop down menuen **Code** og tilføje et fxid til metoden i source coden. På denne måde tilføjedes en række eventhandlers:

Vores venstre TableView blev tildelt **setShowBorderPane()**, som event handler **on mouse clicked** for hver række med et show i tabellen. Den kalder to metoder

setShowHall() og **setReservationTable()**, der gør brug af den førnævnte metode **setSeats()**:

```
private void setShowHall()
{
    ShowHallModel showHallModel = new ShowHallModel();
    ShowModel showModel = new ShowModel();
    int show_id;
    int theater_id;

    int index = bioTable.getSelectionModel().getSelectedIndex();
    //storing information from tableRow
    String hallColumnValue = hallColumn.getCellData(index);
    String showColumnValue = showColumn.getCellData(index);
    LocalDateTime timeColumnValue = timeColumn.getCellData(index);

    theater_id = showModel.getTheaterID(hallColumnValue);
    Timestamp ts = java.sql.Timestamp.valueOf(timeColumnValue); // Parses the value of timeColumnValue to Timestamp

    //Timestamp ts = new Timestamp();
    show_id = showModel.getShowID(ts, theater_id);
    //GridPane which store all seatButton in GUI
    showHallGrid = showHallModel.setSeats(hallColumnValue, show_id);
    //Adds the seat to the GUI pane ShowHall
    showBorderPane.setCenter(showHallGrid);

    currentShowID = show_id;
}
```


setReservationTable():

```
private void setReservationTable()
{
    ObservableList<ReservationTableModel> reservations = FXCollections.observableArrayList();
    ShowModel showModel = new ShowModel();
    ReservationModel reservationModel = new ReservationModel();
    int show_id;
    int theater_id;

    int index = bioTable.getSelectionModel().getSelectedIndex();
    //storing information from tableRow
    String hallColumnValue = hallColumn.getCellData(index);
    String showColumnValue = showColumn.getCellData(index);
    LocalDateTime timeColumnValue = timeColumn.getCellData(index);

    theater_id = showModel.getTheaterID(hallColumnValue);
    Timestamp ts = java.sql.Timestamp.valueOf(timeColumnValue); // Parses the value of timeColumnValue to Timestamp

    //Timestamp ts = new Timestamp();
    show_id = showModel.getShowID(ts, theater_id);

    reservationModel.getReservationData(reservations, show_id);

    reservationTableView.setItems(reservations);
}
```

Disse to metoder sørger for, at informationen i TabPanet **Show hall** og **Reservations** bliver opdateret, så antallet af sæder til den pågældende forestilling og deres status laves i **Show hall** samtidig med, at reservationerne for den pågældende forestilling også loades i **Reservations**.

Ekspedienten kan tilgå sæderne, som oprettes vha. metoden: **setSeats(String theater_name, int show_id)**.

Denne metode er skrevet for at lave en dynamisk visning af sæderne i de forskellige sale, som forestillingerne er knyttet til. For at undgå mange tabs/vinduer med hver sal og dennes sæder, blev metoden derfor skrevet, så et antal af **ToggleButtons** svarende til antallet af rækker i salen og sæderne heri blev oprettet. Derudover bliver hver **ToggleButton** tildelt en farve ud fra deres status i databasen (reserveret/ ikke reserveret).

TableView for shows har fået en event handler, som gør at alt inklusive fanerne for oven: show, time og hall reagerer med et database kald, når de klikkes. Dette blev dog løst i det andet table for reservations, hvor det kun var den enkelte column cell, som reagerede på eventet. Vi fik ikke implementeret funktionen i TableViewet for shows pga. tidsaspektet. Den samme nedprioritering gjorde sig gældende for reserv-knappen, som stadig delvist kan benyttes, selvom der ingen sal eller sæde er valgt. Vi valgte i stedet at fokusere på, at knappen skulle kunne udføre en reservation, såfremt ekspedienten fulgte anvisningerne i brugsvejledningen.

Test af programmet

JUnit testing

For at sikre at reservationssystemet virker som det skal, har vi skrevet nogle tests, som efterprøver de fundamentale funktioner af et reservationssystem. Vi har taget udgangspunkt i unit-testing, hvor vi opdeler programmet i mindre moduler, som vi kan afprøve selvstændigt. Hvert modul isoleres, hvorefter der opstilles et scenarie som er passende i forhold modulets funktionalitet, og hvad der forventes som resultat. Da systemet er opbygget ud fra et MVC(Model-View-Controller) designmønster, har det været oplagt at skrive test for model klasserne, fordi de har ansvaret for håndteringen og repræsentationen af dataen samt controller klasserne, fordi de repræsenterer bindeleddet mellem modelklasserne og view.

Taget dette i betragtning, så har vi opbygget vores tests med hensyn til de primære use cases for et billetreservationssystem.

Derfor er det oplagt at udføre tests med henblik på, at disse funktioner virker som de skal.

Testene der er blevet oprettet er opdelt i fire kategorier med følgende klasser:

- **Database forbindelsetests**
 - DatabaseControllerTest
 - DatabaseConnectorTest

- **Reservationstests**
 - ReservationControllerTest
 - ReservationModelTest
 - ReservationsObjectTest

- **Kundehåndteringstests**
 - CustomerModelTest

- **Forestillingstests**

- ShowObjectTest
- ShowTableTest
- SeatObjectTest

Databaseforbindelse tests

Disse testklasser afprøver forbindelsen mellem reservationssystemet og serveren, hvor databasen er lokaliseret. Dette er essentielt netop, fordi databaseforbindelsen er bindeleddet mellem reservationssystemet og alt den data, som systemet skal bruge for at fungere korrekt. Derfor skal man sikre sig, at forbindelsen hele tiden kan oprettes, hvilket kan tjekkes gennem tests.

Herunder har vi oprettet to testklasser, hvor den ene er DatabaseConnectorTest, der undersøger om vi kan oprette forbindelse og indsende og hente data fra databasen. DatabaseConnectorTest kalder et sæt af metoder med foruddefinerede SQL-forespørgsler (queries), der ligger i DatabaseInjector og DatabaseRetriever. Førstnævnte har til formål at modificere databasen, imens sidstnævnte blot henter data fra databasen.

Den anden klasse er DatabaseControllerTest, der undersøger om databasecontrolleren fejlfrit kan videresende foruddefinerede SQL-forespørgsler til de passende databasemodeller.

Begge testklasser returnerer nogle værdier, som sammenlignes med de foruddefinerede queries, der blev brugt. Hvis testene blev udført fejlfrit og det sammenlignede resultat er ens, kan det være med til at bekræfte, at testene var vellykkede.

Reservationstests

Udover databaseforbindelsen er det vigtigt at teste, om systemet kan håndtere reservationer korrekt og fejlfrit af den simple grund af at systemet er et reservationssystem.

Reservationstests består af tre tests klasser. De vigtigste testklasser er ReservationModelTest, som har ansvaret for at teste oprettelsen af nye reservationer og sletning af eksisterende reservationer. Testklassen bruger foruddefinerede reservationsoplysninger til at oprette testreservationer i database. Herefter vil den undersøge, om reservationen eksisterer i databasen og afslutningsvis prøve at slette testreservationen.

For at opnå en vellykket test vil der blive lavet en sammenligning af de foruddefinerede reservationsoplysninger og de returnerede oplysninger. Ved oprettelse og undersøgelse om testreservationens eksistens, skal de returnerede oplysninger og de foruddefinerede oplysninger være ens. Ved sletning af testreservationen skal de returnerede oplysninger være lig null, fordi dette er en indikation på, at testreservationen ikke længere eksisterer i databasen - og den kan derfor antages slettet.

ReservationsControllerTests, udfører lignende tests, men den undersøger samtidig om bindeledet mellem reservationscontrolleren og de øvrige reservationsklasser er stabile. Det vil sige, at man kan kalde de forskellige metoder som de øvrige reservationsklasser har, gennem controlleren.

Kundehåndteringstests

CustomerModelTest under kundehåndteringstests har ansvaret for at teste oprettelsen af nye kunder, samt det at trække oplysninger om eksisterende kunder ud fra databasen. Ligesom de forrige tests, benytter vi foruddefinerede oplysninger, som vi bruger til at oprette testkunder og sammenligne returnerede oplysninger med.

Forestillingstests

Testklasserne til forestillingsklasserne har ansvaret for at afprøve, om der bliver returneret en liste med forestillingsobjekter, der indeholder oplysninger om hver forestilling i databasen. Formålet med testen er, at sikre at disse forestillingsobjekter bliver returneret, således at GUI'en kan vise dem.

Test og design er tæt forbundet, hvilket kommer til udtryk gennem opdelingen af tests i kategorier, der afspejler klassestrukturen. Dette er med til at sikre, at de fundamentale funktioner altid vil kunne testes ved ændringer eller nye implementationer. Dette gjorde det lettere at fejlfinde og debugge programmet i tilfældet af, at en exception opstår.

For det andet blev testene udviklet, så snart grundstrukturen af vores klasser blev klargjort. Som resultat kunne vi teste klasserne isoleret set og således reducere koblingen i reservationssystemet. Princippet benævnes også dependency injection⁵ hvor man lader afhængigheden koble sig til modelklasserne og ikke omvendt. I dette tilfælde kunne vi for eksempel udskifte controller klasserne ud med test klasserne, når vi ville udføre tests på model klasserne. Samlet set blev klasserne mere modulære, hvilket betød, at vi lettere kunne skifte input komponenter til klasserne ud med andre og derved opnå et mindre koblet system.

⁵ Introduktion til Unit Testing - S3. L.26 - "Ekstra undervisningsmateriale"

Testanalyse og exception

Testene tjekker generelt om de forskellige klasser kan tilgå databasen afhængig af formålet. De benytter alle foruddefinerede oplysninger som ønskes oprettet, for så at hente disse oplysninger ned igen til sammenligning og sidst slette dem fra databasen.

På den måde får vi bekræftet, om der etableres en forbindelse til databasen, samt at vi kan oprette hente og slette data . Hvis der opstår exceptions undervejs, kan man hurtigt og effektivt finde frem til fejlen, og i hvilken del af processen den foreligger i, hvorfra man kan debugge isoleret. Derved undgår man at skulle søge hele systemet igennem for fejl.

Vi fandt blandt andet en fejl, hvor forbindelsen til databasen forblev åben, men vi har måtte nedprioritere dette i forhold til andre arbejdsopgaver, da det først er blevet medtaget på et sent tidspunkt i forløbet. Den åbne forbindelse kan være problematisk, hvis flere skal have adgang til databasen samtidig, men eftersom systemet er målrettet en mindre biograf, hvor kun en enkelt ekspedient skal have forbindelse, har vi nedprioriteret løsningen herom.

Udover dette har vi også så vidt muligt prøvet, at få håndteret exceptions via passende metoder, som kan kaste dem. Blandt andet har vi oprettet try/catch statements i alle vores metoder i DatabaseControlleren, fordi SQL-forespørgsler går igennem controllerklassen. Hvis der endelig opstår en exception vedrørende SQL-forespørgelser, så kan vi også hurtig lokalisere fejlen, og dermed få et indblik i, hvad systemet prøver at udføre, når fejlen opstår.

I starten af projektet oprettede vi egne exceptions, der blev kastet ved sletning af en kunde eller reservation, som ikke eksisterede i databasen. Dette fjernede vi dog igen, da vi senere i forløbet, fandt frem til, at man kan udvide SQL-forespørgslerne, så der kun skal oprettes en kunde, hvis det vedkommende ikke eksisterer i databasen (...*WHERE NOT EXISTS*...).

Et andet exception som vi har fjernet skulle kastes i tilfælde af, at vi ville slette en reservation, som ikke eksisterede i databasen. I dette tilfælde vil reservationssystemet dog ikke termineres uventet, når vi fravalgte at implementere exceptionen.

Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

Dette skyldes højst sandsynligt blot, at vi sender en SQL query til databasen. Umiddelbart vil der ikke være nogen logiske grunde for at systemet ville termineres. Tværtimod er formålet med denne type exceptions blot at informere systemet eller brugeren om, at sletningen fejlede, fordi det ønskede element ikke eksisterer.

Denne exception er blevet fjernet efter implementationen af en slettefunktion, hvor ekspedienten markerer en reservation under reservationsfanen og derefter trykker på slet knappen. Taget dette i betragtning, så kan brugeren ikke give noget forkert input, såsom tastefejl da alle reservationer allerede er listet og tilgængelig visuelt. I sidste ende brugte vi ikke disse exceptions, da vi endte med at refaktorere en større del af koden til systemet.

Databaseafhængighed og mock-ups

Det skal siges at vores tests stadigvæk er afhængig af eksterne faktorer, heraf den reelle database forbindelse. Dette betyder dog at vi er bundet til at databasen virker og at vi har en internetforbindelse. Disse faktorer er forholdsvis utilregnelige og er derfor ikke helt optimale end hvis vi havde benyttet et simpelt og forudsigeligt mock-up. Man kunne have lavet disse mock-ups i Mockito, hvilket havde givet et indtryk af “dataflowet” inden forbindelsen til databasen var oprettet og vi kunne tilgå data derfra.

Dette fravalgte vi dog udelukkende, fordi vi i forvejen skulle arbejde med mange andre nye værktøjer i projektet, som vi ikke havde prøvet før, herunder versionsstyring i Git, IDE’et IntelliJ og JavaFX til den grafiske brugergrænseflade.

Konklusion

Vi har gennem dette indledende softwareprojekt erfaret, hvor vigtigt det er at reflektere over ens design og arkitektur i et system, før man påbegynder kodningen. Vi har under hele forløbet forsøgt at gøre os nogle tanker omkring, hvad problemet er, og hvordan vi vil løse det. I gruppen har vi arbejdet iterativt på de problemstillinger, der afstedkom i udviklingen af et biograf reservationssystem.

Selvom vi ikke har løst alle problemer og dertilhørende scenarier mest hensigtsmæssigt, har vi gennem en mindre perfekt løsning skudt os gradvist tættere på målet ved at løse de vigtigste opgaver først. For at kunne gøre dette har vi brudt det overordnede problem op i simple delproblemer, der samtidig har givet os indsigt i, hvordan det komplekse problem løses. Dette kommer til udtryk i vores klassestruktur, der er skabt med henblik på at opdele ansvarsområderne mellem flere aktører, som løser det overordnede problem i samspil. For at kunne gøre dette har vi dog måtte nedprioritere andre aspekter såsom vores GUI design, der kun er optimeret til de hyppigste arbejdsopgaver hos ekspedienten.

I forlængelse af dette har det været nødvendigt med en større omstrukturering på trods af vores MVC-struktur, men dette skyldes primært et manglende kendskab til databaser og JavaFX. På trods af dette har vi stadig fået et større kendskab til grundprincipperne i objektorienteret programmering end før kurset samt brugen af professionelle udviklingsværktøjer fx IDE'er som IntelliJ og Visuelle UI Værktøjer som JavaFX Scene Builder. Ved at applicere disse principper på et ikke-trivielt program har vi således opnået en større viden omkring håndteringen af softwareprojekter, og samtidig er vi alt i alt blevet dygtigere programmører.

Refleksion over arbejdsproces

Det har for alle tre gruppemedlemmer i projektet været en kæmpe omvæltning at gå fra at have et “perifert” kendskab til en masse grundprincipper i Java kodning til at skulle applicere disse på et af virkelighedens problemer. Vi har tilegnet os ny viden på flere fronter herunder, hvordan man strukturerer et softwareprojekt i praksis med udgangspunkt i objektorienteret programmering, der kan bruges til at opdele koden i klasser, som håndterer hver sit ansvarsområde. Denne tankegang fik vi introduceret gennem kurset i grundlæggende programmering, som har skærpet vores forståelse for dens virkning i praksis.

Samtidig har vi nu, med egne øjne erfaret, hvilke udfordringer der rent faktisk eksisterer i et softwaremiljø, hvor man ofte skal administrere et projekt mellem flere mennesker. Vi fik bekræftet, hvor effektiv versionsstyring er til at administrere et softwareprojekt mellem flere programmører, der skal kunne se hinandens ændringer og arbejde samtidig. Dette er alfa omega, da man ofte arbejder uafhængigt af tid og sted indenfor softwarebranchen. Ved at bruge versionsstyring har vi kunne genoptage arbejdet uden videre og undgå unødvendige distraktioner, der kan risikere at afbryde arbejdet undervejs.

Derudover har det været en fordel, at migrere over til et nyt IDE (IntelliJ) i stedet for BlueJ, selvom BlueJ retmæssigt har tjent sit formål under kurset. Forskellen ligger i behovet. BlueJ imødekommer behov hos nye studerende som os selv, der aldrig har programmet før kurset, fordi man nemt kan forholde sig til brugergrænsefladen, der er simpel, interaktiv og visuel. Derved kan man fokusere på at lære koncepterne i programmering frem for at lære at benytte sig af et andet IDE som kan blive en distraktion fra hovedformålet - nemlig det at kunne skrive og forstå god kode. I IntelliJ og flere andre IDE’er bruger man tid på at kigge på mange linjers kode, der kan distrahere en fra dette læringsmål. Vi valgte dog alligevel at tage springet, da man professionelt bruger lignende IDE’er, og kendskabet til disse værktøjer er en fordel senere hen, når man skal ud på arbejdsmarkedet og arbejde på større projekter.

Arbejdsopgaver er blevet uddelt via applikationen Wunderlist, som er et redskab til projektstyring og uddeling af opgaver, mens deadlines er blevet indsat i et fælles Gantt diagram⁶, der giver en oversigt over hvilke aktiviteter, der skal nås hvornår.

Disse redskaber har i høj grad bidraget til at strukturere og prioritere vores arbejde, hvilket vi har fundet nødvendigt, da man i forvejen kan have svært ved at give slip på et stykke software i et større projekt, der umiddelbart altid kan forbedres. Med dette bør man blot medtage, at det er afgørende at kunne give slip på et stykke kode, når det bliver for komplekst eller måske endda irrelevant, hvilket vi har måtte opleve flere gange i processen (jf. Bilag 10.6). Undervejs er vi desuden blevet introduceret til mange nyere og bedre måder at implementere dele af vores løsninger på, men samtidig har vi været bundet op på en deadline. I så fald har vi været nødsaget til at aflevere et produkt, der ikke efterstiller alle krav, men forsøger at gøre det muligt for ekspedienten at løse de mest gængse arbejdsopgaver manuelt. Dette søger vi at afspejle i vores strukturelle designmønster, der er skabt med henblik på at dele problemet op i mindre bestanddele. De simple problemer kan samtidig give indsigt i løsningen af det overordnede problem og gør det nemmere for os at arbejde videre på projektet efterfølgende.

Selvom det har været svært for os at give slip på nogle ting, har vi accepteret omstændighederne og erkendt, at et program aldrig er fejlfrit og uforanderligt. Dette er baseret på præmissen om at ens prioriteter ændrer sig løbende, samtidig med at man bliver udsat for mange nye måder at se tingene på. Man bør derfor være åben overfor de udfordringer, der foreligger både i ens kode, men også i valget mellem det mangfoldige udbud af teknologier, der er målrettet samme problem. I stedet for at frygte fejl, bugs og exceptions i vores program har vi brugt disse som drivkraft til løsningen i, hvordan vi fikser programmet eller optimerer koden. Dette bringer os til den sidste erfaring i projektet; programmering ændrer måden hvorpå man ser på livet og har en indvirkning på ens personlige kompetencer. Vi har i kurset og dette projekt fået en fornemmelse for, hvad der karakteriserer en softwareudviklers “mindset”, hvilket til dels kommer til udtryk gennem lysten til at “afprøve noget nyt”, selvom det ikke nødvendigvis er forsøgt tidligere.

I udviklingen af software har vi altså lært at omfavne forandring sammen med den hurtige tilpasning til nye teknologier, der skaber incitament til at prøve nye ting.

⁶ Se bilag 10.3

Litteratur

E-Litteratur

- Stackoverflow: “Javascript what’s the difference between event handlers listeners
<http://stackoverflow.com/questions/6929528/javascript-whats-the-difference-between-event-handlers-listener>(senest aflsæt 17/12 2014) event handler forklaring
- Sencha: “What’s the difference between handlers and listeners
<http://www.sencha.com/forum/showthread.php?86997-What-s-the-difference-between-handler-and-listeners> (senest aflsæt 17/12 2014) event handler vs listener
- Javabog:
<http://javabog.dk/> (senest aflsæt 17/12 2014) danske java termer
- Timothyasiimwe: Reflections of a programmer.
<http://timothyasiimwe.me/blog/2014/08/02/reflections-of-a-programmer/> (senest aflsæt 17/12 2014)
- Microsoft: Model-View-Controller
<http://msdn.microsoft.com/en-us/library/ff649643.aspx> (senest aflsæt 17/12 2014)
- Java API Oracle Documentation
<http://docs.oracle.com/javase/7/docs/api/> (senest aflsæt 17/12 2014)
- JavaFX Oracle Documentation
<http://docs.oracle.com/javase/8/javase-clienttechnologies.htm> (senest aflsæt 17/12 2014)
- code.makery
<http://code.makery.ch/> (senest aflsæt 17/12 2014)

Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

Undervisningsmateriale

- David Barnes & Michael Kölling , 5. Edition: “Objects First with Java - A practical Introduction Using BlueJ”. - Person Education
- Sestoft, Peter (2000): ”*Udformning af rapporter*”. IT Universitetet i København (Upubliceret undervisningsmateriale).
- Mappe “Ekstramateriale” fra Jesper
 - Introduktion til Unit Testing - “Ekstra undervisningsmateriale” :unittesting.pdf
 - Video: JavaFX-intro.mp4 - Jesper Madsen
 - Video: Git_Intro.mp4 - Jesper Madsen
 - Video: Introduktion_til_IntelliJ.mp4 - Jesper Madsen
 - Video: testing.mp4

Forelæsninger

- Brand, C. (2014): Lektion 14: .SQL II: Joins, Keys, Group-By, & Java+SQL ! GRPRO AUTUMN 2014, Grundlæggende Programmering. IT Universitetet, København, 28. september 2014, upubliceret.
- Brand, C. (2014): Lektion 15: .SQL II: Joins, Keys, Group-By, & Java+SQL ! GRPRO AUTUMN 2014, Grundlæggende Programmering. IT Universitetet, København, 30. september 2014, upubliceret.

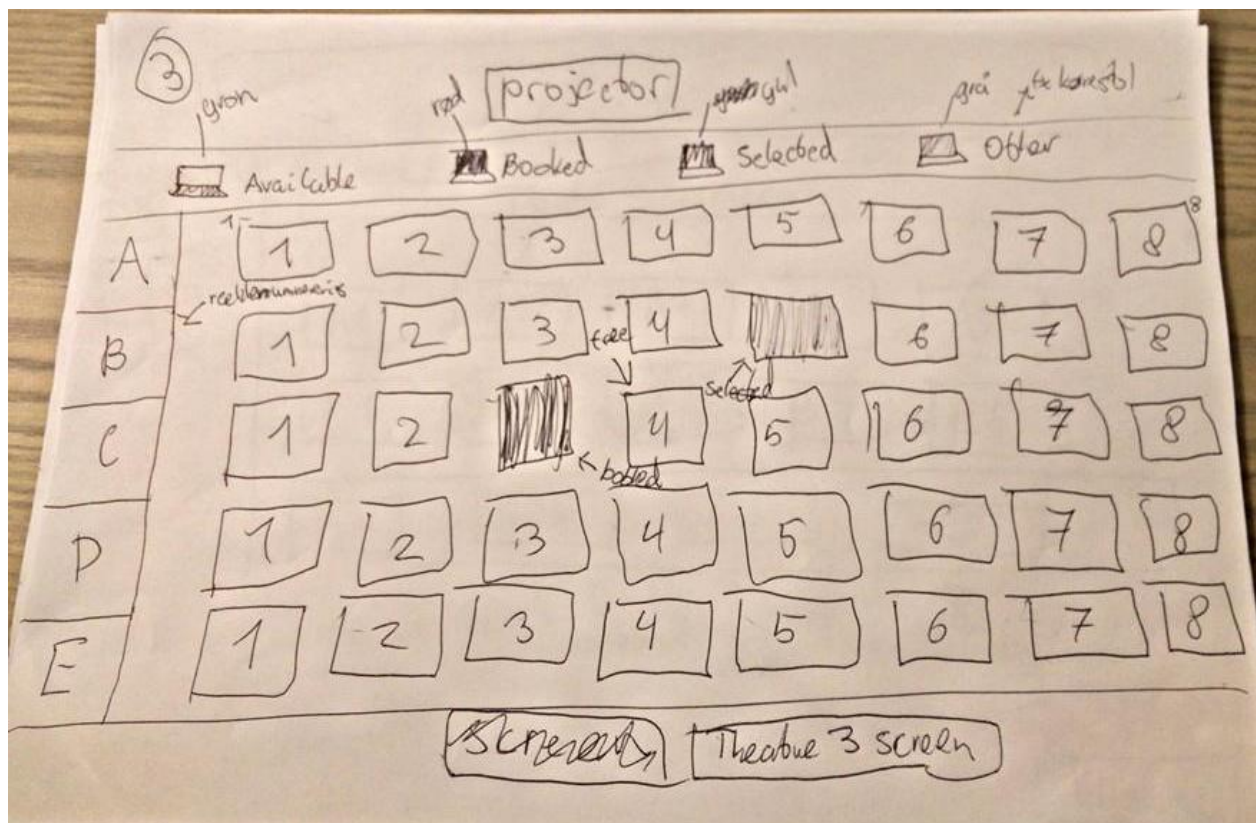
Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

Bilag

Mockups

I mockup afsnittet har vi alle skitser af hvordan vores gui kunne se ud.

Mockup 1



Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

Mockup 2

③

Movie Title
Cinema, Date, Time

① Choose tickets
Tickets
0
price
xxx DKK

② Choose seats
Theatre 3 - screen

A1	A2	A3	A4	A5
B1	B2	B3	B4	B5

grøn	Rød	Blå	Gul
------	----------------	-----	-----

↑ Available / free ↓ Booked / reserved ↑ other ↑ Selected / marked

Back Reserve

Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

Mockup 3

②

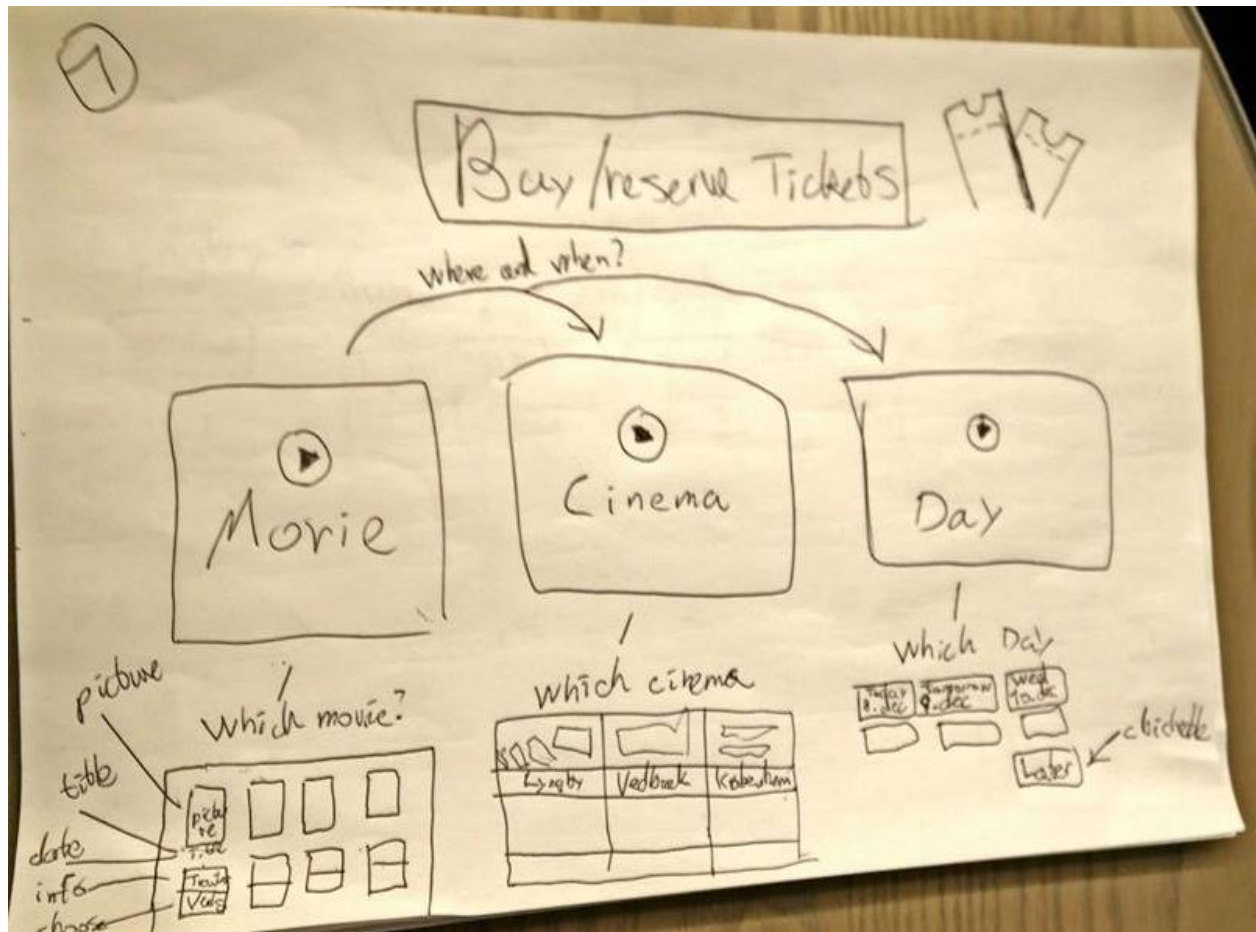
Where and when?

	Sun 7/12	Mon 8/12	Tue 9/12	Wed 10/12	Thu 11/12	Fri 12/12	Sat 13/12	Sun 14/12
Lyngby								
prcbue	16:00	13:30			12:00		18:00	18:00
København	20:00	18:30						
Århus								

Next Week

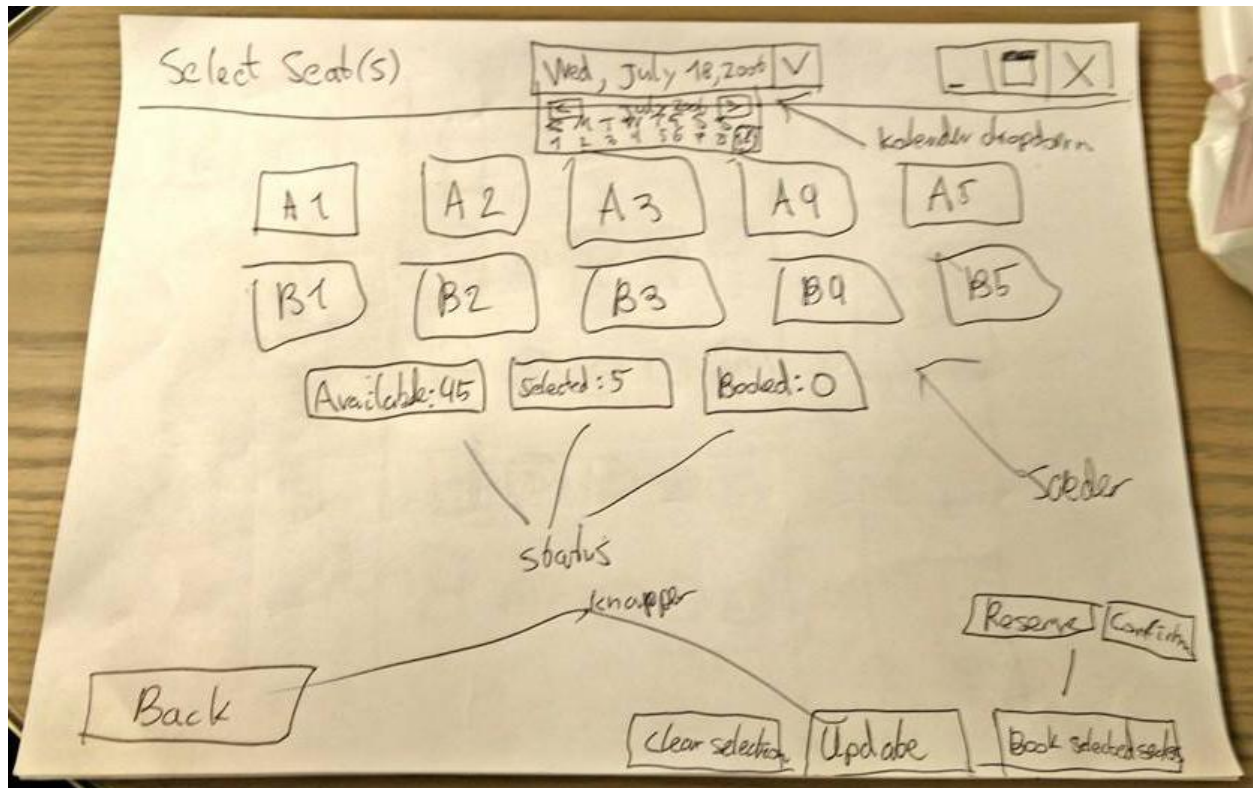
Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

Mockup 4



Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

Mockup 5







☐ Seat Selection proceed

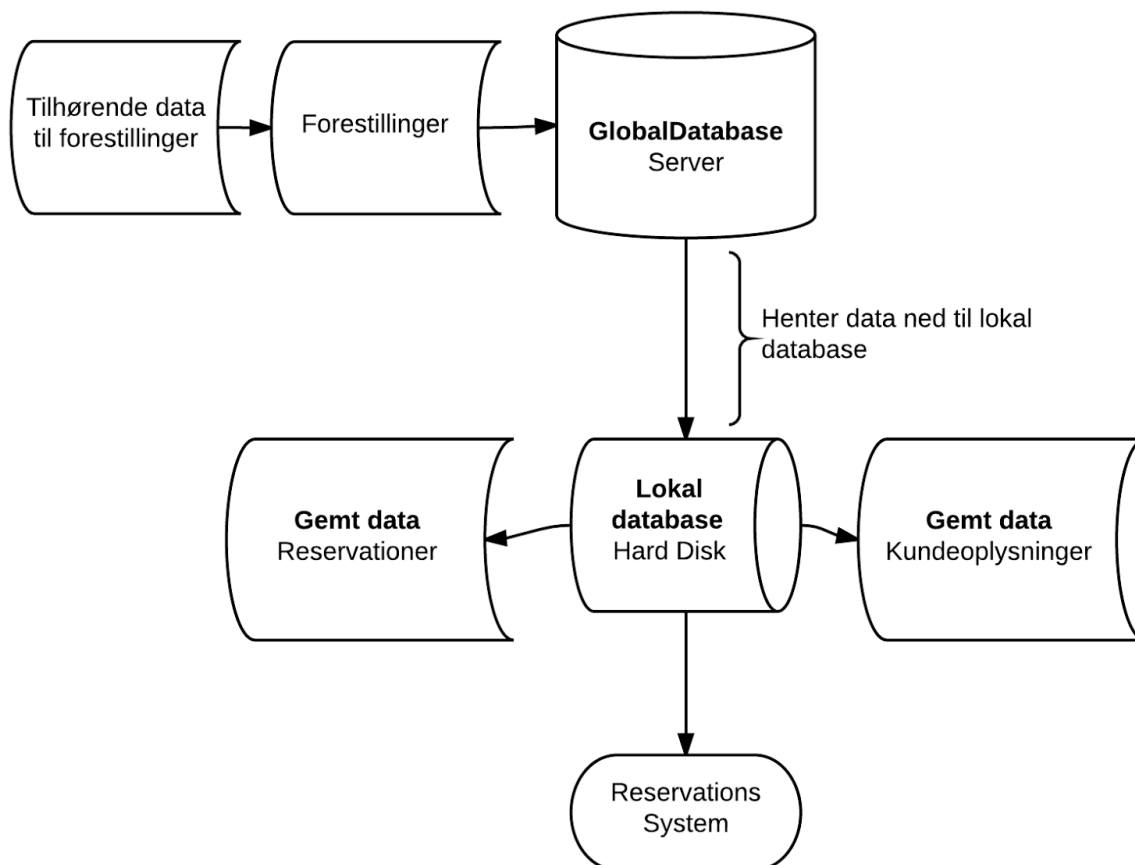
Movie Title
Show date: Mon, Dec 9, 2014 (10:00 pm)
Screen: Theatre 3

A	1	2	3	4	
B	(1)	(2)	(3)	(4)	(5)
C	(1)	(2)	(3)	(4)	(5)
D	(1)	(2)	(3)	(4)	(5)
E	(1)	(2)	(3)	(4)	(5)

Screen

			
---	---	--	---

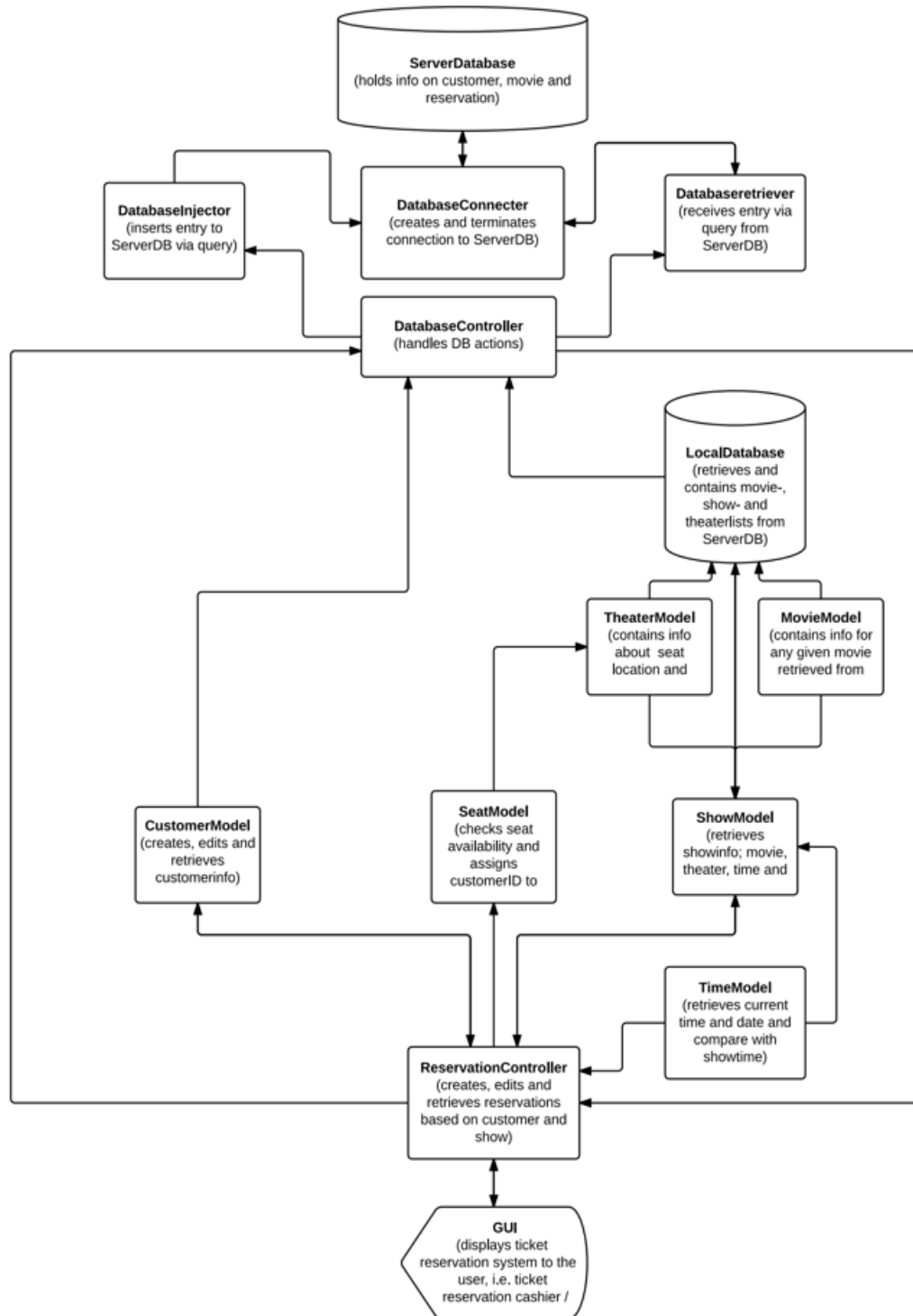
Flowchart ved benyttelse af lokal database



Flowcharten viser hvordan dataen bliver hentet og lagret mellem lokal databasen og global databasen

Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

Flowchart af første data struktur

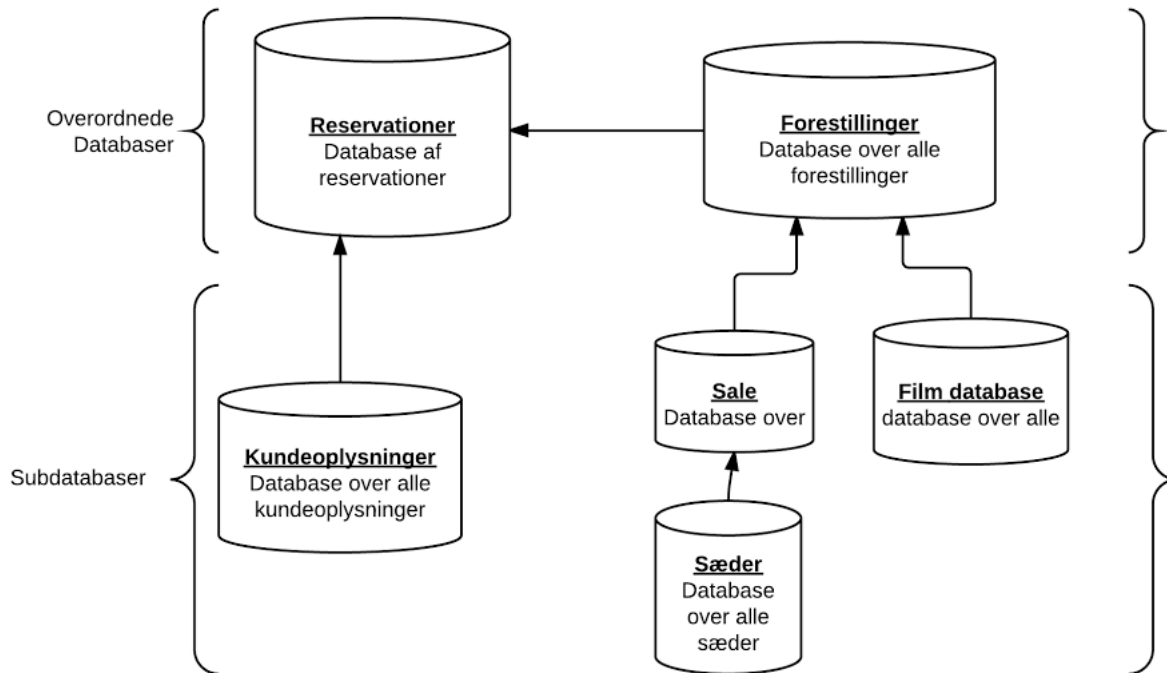


Dennis Thinh Tan Nguyen

Thor Valentin Aakjær Nielsen Olesen

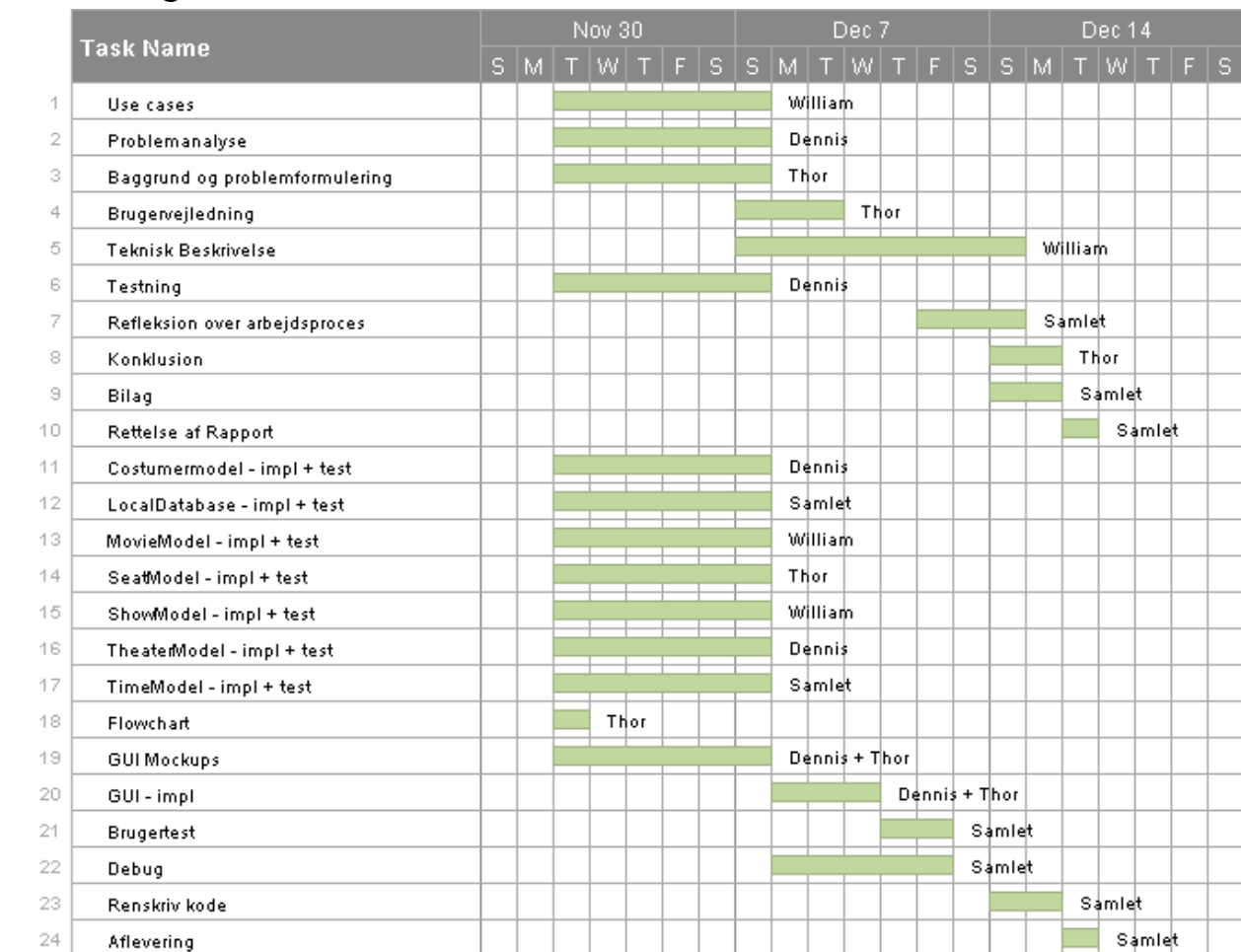
William Diedrichsen Marstrand

Dataflow og relationer mellem tables i databasen.



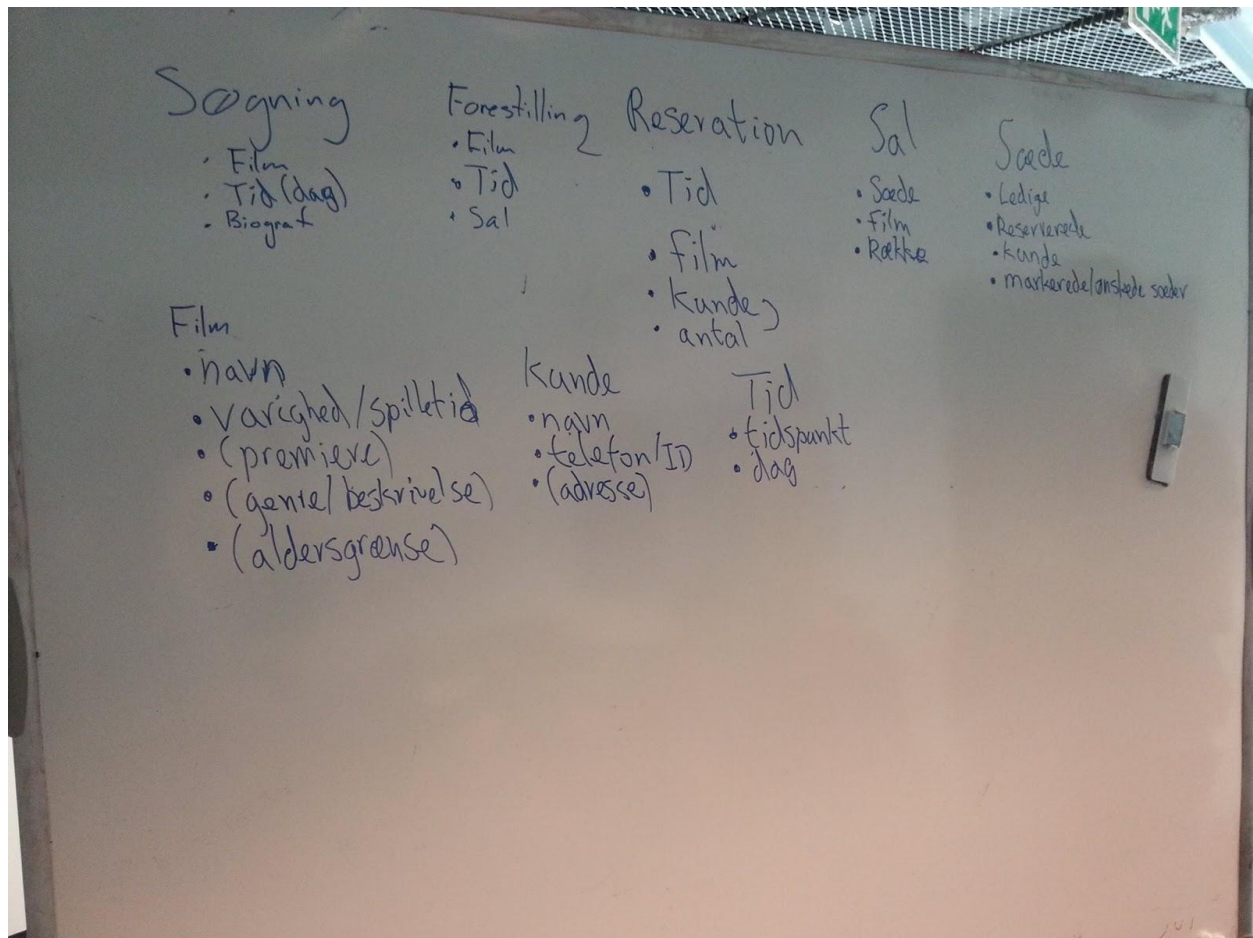
Flowchartet repræsenterer relationerne mellem de forskellige tabeller i databasen. Hvert element præsenterer de forskellige tabeller, mens pilene viser hvordan de er relateret i forhold til hinanden. De er forbundet med hinanden ved hjælp af tabellernes unikke nøgle.

Dennis Thinh Tan Nguyen
 Thor Valentin Aakjær Nielsen Olesen
 William Diedrichsen Marstrand
Gantt Diagram



Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

Noun/Verb Oversigt



Andre programmeringsprincipper

Indkapsling (eng.: encapsulation)

En udfordring under udviklingen af et system, der skal benyttes af andre mennesker er risikoen ved, at der pilles ved data og metoder i objekter, som påvirker systemets adfærd. For at undgå dette indkapsler man data og metoder, hvilket betyder at man ikke lader andre udefra bruge objekter efter eget ønske fuldstændigt. Ved at angive metoder og data med nøgleord (public, private, protected) kan man styre adgangen til et objekts data og metoder og derved sikre sig mod uønskede dataændringer. Eksempelvis bør metoder, der opretter og lukker forbindelsen til databasen være private eller protected, så man undgår inkonsistens i forbindelsen, hvilket er afgørende i dette projekt, da alle informationer vedrørende reservationer lagres og tilgås i databasen.

Abstraktion (eng.: abstraction)

For at få et hurtigt indtryk af systemet, dets opbygning og virkemåde har vi gjort brug af abstraktion af data. Ved at tage udgangspunkt i klassestrukturen illustreret i vores **Flowchart** har vi først defineret klasserne i java sammen med de variabler og den adfærd (metoder) de bør have. Metoderne har vi tildelt en passende signatur ift. deres funktion. Metodekroppen har vi blot sat til at returnere en endelig værdi eller den paramter, som vi tror skal benyttes i metoden. Først derefter har vi skabt de rigtige objekter, der bør handle som angivet i klassen og hentet data gemt i databasen dertil.

Uddelegering (eng: delegation)

Som ordet antyder er der tale om, at man afleverer ansvaret for en bestemt opgave til en anden klasse eller metode. Hvis man skal bruge en funktionalitet i en anden klasse, men ikke ønsker at ændre funktionaliteten ydeligere bruges delegation frem for nedarvning. Eksempelvis skal systemets modelklasser ofte oprette en forbindelse til databasen, hvilket gøres gennem databasecontrollerens metoder, der dog ikke skal “ændres” undervejs processen.⁷

⁷ <http://www.vogella.com/tutorials/DesignPatterns/article.html#chapter20s1s2a>

Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

Dokumentation, kommentarer, navnekonventioner og exceptions m.m.

I kildeteksten til systemet bør man stræbe efter en kort, overskuelig og præcis dokumentation, der sammen med kommentarer af klasser og metoder forklarer klassernes ansvarsområde, metodernes formål og måske endda rationalet bag disse valg, hvis det kræver ydeligere specificering. Herudover bør man benytte sig af navnekonventioner i metodesignaturen, der afspejler metodens indhold og formål, så andre der læser koden hurtigt kan få et indblik i systemets virkemåde eller arbejde videre på et eksisterende system. Dokumentationen, kommentarerne og navnekonventioner bør tilsammen give vedkommende en idé om, hvad der sker og hvorfor.

Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

Kodeændringer (fjernet)

Removed MovieModel due to lack of search function

```
1 package Model;
2
3
4 +import ...
5
6
7
8
9 +/**...*/
14 public class MovieModel {
15
16     //Fields
17
18     private final DatabaseController mDBController;
19
20
21     //Constructor
22
23 +/**...*/
26 +public MovieModel() { mDBController = new DatabaseController(); }
29
30     //Methods
31
32     public String getMovieGenre(String movieTitle)
33 +{...}
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48     public String getAgeLimit(String movieTitle)
49 +{...}
50
51
52
53
54
55
56
57
58
59
60     public String getMovieLength(String movieTitle)
62 +{...}
63
64
65
66
67
68
69
70
71
72
73
74
75     public ArrayList<String> getMoviesByGenre(String movieGenre)
76 +{...}
77
78
79
80
81
82
83
84
85
86
87
88 +public ArrayList<String> getAllMovieTitles(){...}
89
90
91
92
93
94
95
96
97
98
99
100
101     public ArrayList<String> getAllFromMovieList()
102 +{...}
103
104
105
106
107
108
109
110
111
112
113
114 +/**...*/
119 public int getMovieIDFromTitle(String title)
120 +{...}
121
122
123
124
125
126
127
128 +/**...*/
133 public String getMovieTitleFromID(int movieID)
134 +{...}
135
136
137
138
139
140
141
142 }
```

Dennis Thinh Tan Nguyen

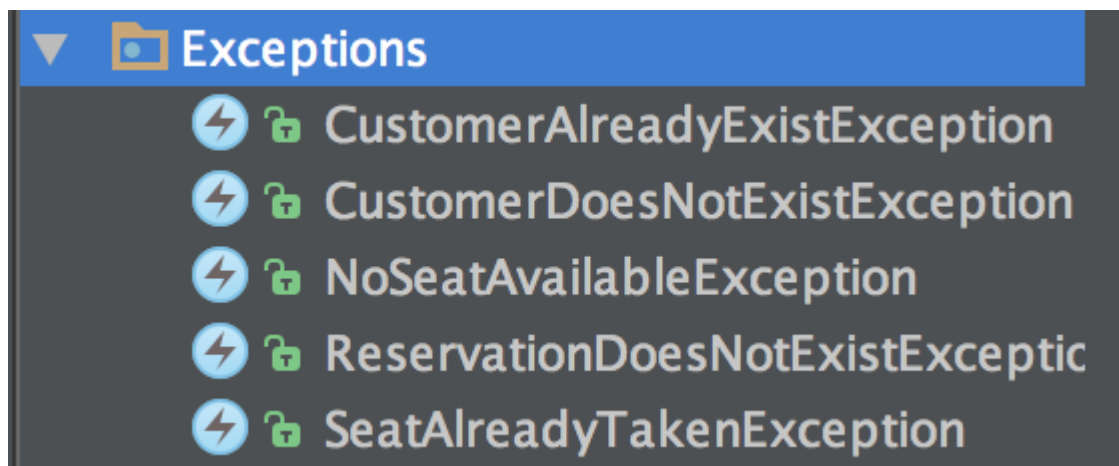
Thor Valentin Aakjær Nielsen Olesen

William Diedrichsen Marstrand

Removed ShowModel Test due to GUI involvement & lack of skills

```
1 package UnitTests;
2
3 import ...
4
5
6
7 public class ShowModelTest extends TestCase {
8
9     private ShowModel show;
10
11     @Before // Initialize objects related to Show Model before each test
12     public void setUp() { show = new ShowModel(); }
13
14
15
16
17     public void testGetShowData() throws Exception {...}
18
19
20
21     public void testGetTheaterID() throws Exception {...}
22
23
24
25     public void testGetShowID() throws Exception {...}
26
27
28
29     public void testCreateShowSeats() throws Exception {...}
30
31
32 }
```

Removed Exceptions previously used to prompt the user



Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

```
1 package Model.Exceptions;
2
3 /**
4  * Exception used if a customer already exist in the database
5  * Created by Dennis Thinh Tan Nguyen - William Diedrichsen Marstrand - Thor Valentin Olesen
6  */
7 public class CustomerAlreadyExistException extends IllegalAccessException{
8
9     public CustomerAlreadyExistException()
10    {
11        super("The customer is already registered");
12    }
13 }
```

```
1 package Model.Exceptions;
2
3 /**
4  * Exception if a customer does not exist in the database.
5  * eg. thrown if you try to delete a non-existing customer
6  * Created by Dennis Thinh Tan Nguyen - William Diedrichsen Marstrand - Thor Valentin Olesen
7  */
8 public class CustomerDoesNotExistException extends IllegalAccessException{
9
10    public CustomerDoesNotExistException(String errorMessage)
11    {
12        super(errorMessage);
13    }
14 }
```

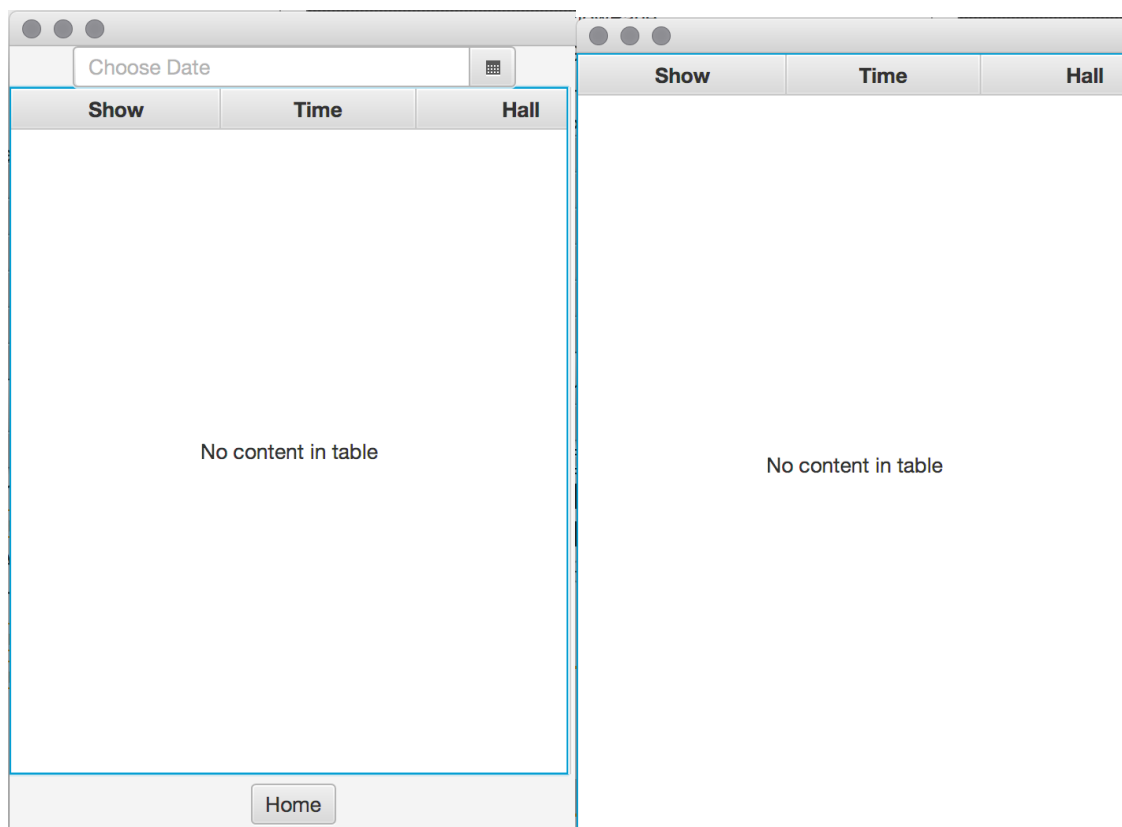
```
1 package Model.Exceptions;
2
3 /**
4  * Indicates that there are no seats available.
5  * Created by Dennis Thinh Tan Nguyen - William Diedrichsen Marstrand - Thor Valentin Olesen
6  */
7 public class NoSeatAvailableException extends IllegalAccessException {
8     public NoSeatAvailableException(String errorMessage)
9     {
10        super(errorMessage);
11    }
12 }
13
14
```

```
1 package Model.Exceptions;
2
3 /**
4  * Exception that indicates that a reservation doesn't exist.
5  * eg. thrown if you try to delete a non existing reservation
6  * Created by Dennis Thinh Tan Nguyen - William Diedrichsen Marstrand - Thor Valentin Olesen
7  */
8 public class ReservationDoesNotExistException extends IllegalAccessException {
9     public ReservationDoesNotExistException(String s)
10    {
11        super("There is no existing reservation with the given info");
12    }
13 }
```

Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

```
1 package Model.Exceptions;
2
3 /**
4  * Exception that indicates a seat is already reserved.
5  * eg. thrown if you try to reserve on a reserved seat.
6  * Created by Dennis Thinh Tan Nguyen - William Diedrichsen Marstrand - Thor Valentin Olesen
7  */
8 public class SeatAlreadyTakenException extends IllegalArgumentException{
9
10     public SeatAlreadyTakenException(String errorMessage)
11     {
12         super(errorMessage);
13     }
14 }
```

Removed DatePicker and Home button from GUI (didn't work)



Removed method refreshShow()

Dennis Thinh Tan Nguyen
Thor Valentin Aakjær Nielsen Olesen
William Diedrichsen Marstrand

```
/**
 * Event handler for GUI DatePicker component, which refreshes bioTable data.
 */
@FXML
private void refreshShow()
{
    LocalDate localDate = showDatePicker.getValue();//Get timestamp from chosen day from calender

    LocalDateTime localDateTime= localDate.atStartOfDay();//Convert Date to right class
    Timestamp timestamp = java.sql.Timestamp.valueOf(localDateTime);

    ShowModel showModel= new ShowModel();

    showGuiData.clear();//Clear the previous list

    showModel.getShowData(showGuiData,timestamp); // Add the correct shows to the time table

    setShowData(showGuiData);//Update display
}
```

Removed method showReset()

```
/**
 * Event handler for GUI Home Button component On Mouse Clicked, which resets the bioTable data to the current date.
 */
@FXML
private void showReset()
{
    ShowModel showModel= new ShowModel();

    showGuiData.clear(); //Clear the previous list

    showModel.getShowData(showGuiData);//Add the correct shows to the time table

    setShowData(showGuiData);//Update display
}
```