



SUBMISSION OF WRITTEN WORK

Class code: GRPRO

Name of course: GRUNDLÆGGENDE PROGRAMMERING

Course manager: DAN HANSEN

Course e-portfolio: GRPRO-AUTUMN 2014

Thesis or project title: KAFFEKLUBBEN

Supervisor: JESPER MADSEN

Full Name:

Birthdate (dd/mm-yyyy): E-mail:

1. MARK ROSTGAARD MORTENSEN 09/04-1993 MROM@itu.dk

2. AMANDA ENGEL THILO 21/03-1993 AENT@itu.dk

3. FREDERIK HOVMAND JØRGENSEN 20/01-1993 FHOV@itu.dk

4. _____ @itu.dk

5. _____ @itu.dk

6. _____ @itu.dk

7. _____ @itu.dk



EXAM PROJECT
Biobooking

KAFFEKLUBBEN

Amanda Engel Thilo

Frederik Hovmand Jørgensen

Mark Rostgaard Mortensen

DECEMBER 17, 2014

Denne rapport er udarbejdet i December 2014 i et fire ugers projekt på IT-Universitetet i København under vejledning af Jesper Madsen. I projektet udviklede vi et softwaresystem til booking i biografer skrevet i Java. Programmet har til opgave at skulle assisterer en ekspedient i personens arbejdsopgaver der har med kundehenværdelser af gøre, f.eks: Book nye pladser eller slette en tidlige bestilling. Programmet skal ikke kunne håndtere salg.

Indhold

1 Indledning	5
1.1 Baggrund	5
1.2 Problemstilling	5
1.2.1 Krav til programmet	5
1.2.2 Brugerscenario	6
1.2.3 Systemdesign	6
2 Problemanalyse	7
2.1 Vores løsning	7
2.1.1 Build Reservation	7
2.1.2 Alternative løsninger	8
2.2 Databasedesign	8
3 Brugervejledning	9
3.1 Programmet	9
4 Teknisk Analyse	12
4.1 Model View Controller	12
4.1.1 Moduler	12
4.1.2 Algoritmer	12
4.2 Design af Brugergrænseflade	14
4.2.1 Begrænsninger	15
4.2.2 Fejlmeddelelser	15
5 Afprøvning	16
5.1 Brugerafprøvning	16
5.2 Unit test	16
5.3 Resultat	17
6 Arbejdsprocessen	18
6.1 Projektet	18
6.2 Gruppen	18
7 Konklusion	19

Figurer	21
A Tests	22
B Brugergrænseflade	26
C Mock-up	27
D Databasetabeller	28
E Strukturen	30
F Forestillinger	31

Kapitel 1

Indledning

1.1 Baggrund

Det problemområde vi har arbejdet med i forbindelse med vores projekt, er udviklingen af et software-system til brug for en ekspedient der betjener en billetluge i en biograf. Det område der skulle dækkes var reservation både når kunden stod foran billetlugen, samt når kunden ringede ind på telefon. Vores løsning skulle håndtere ekspedientens arbejdsopgaver i sådanne situationer. Løsningen skal derfor fokusere udelukkende på reservationer, og ikke salg.

1.2 Problemstilling

Den overordnede problemstilling som vores program besvarer, er hvordan man udvikler et simpelt, brugervenligt og databasebaseret softwaresystem til brug for en ekspedient i en mindre biografs billetluge.

1.2.1 Krav til programmet

Af krav til programmet [Madsen 2014] kan nævnes at der i forbindelse med reservationerne skal oplyses navn eller anden identitet på kunden. Vi har i vores program valgt at den centrale information omkring brugeren er dennes telefonnummer. Det har vi gjort af den grund af flere kunder godt kan have samme navn, men at ens telefonnummer er unikt.

Derudover skal biografen indeholde flere sale, og sende flere forskellige forestillinger i løbet af dagen. Salenes størrelse, antal pladser mm. skal fremgå af databasen frem for at være indkodet i selve programteksten. Derfor har vi i stedet for at skrive data ind i programteksten, valgt at lave en speciel klasse til at hente data fra databasen. Det samme er gældende for de enkelte forestillinger, der derfor også bliver gemt i databasen.

Kravene er opplistet nedenfor

- Reserverings process
 - vælge forestillinger, herunder dato, tid, film og sal
 - vælge pladser

- vise optaget og ikke optaget pladser
 - bestille pladser
 - afbestille pladser
 - vis ”Dine sæder”
- Kigge process
 - se film der går
 - se tider

1.2.2 Brugerscenario

Signe er ekspedient i en mindre biograf og bruger *biobooking* til at reserverer biletter til kunder. Hun tænder systemet når hun møder på arbejde. Dagens første kunde optræder lidt over kl. 10. Kunden ønsker at booke tre billetter til Fury kl 17.30 den 19. Januar 2015. Signe klikker på den filmknap hvorpå navnet *Fury* optræder. Herefter klikker hun på knappen med tiden *17.30* som står under datoens *19/1*. Signe spørger nu kunden hvor i salen kunden ønsker sine pladser. Kunden siger *ca. midt i salen*. Signe vælger sæde *9, 10 og 11* på række *F*, og viser det på skærmen til kunden. Kunden er tilfreds, så Signe spørger kunde om hvilket navn og telefonnummer reservationen skal laves i. Efter at have indtastet kundens informationer klikker Signe på *Fuldfør reservation*. Kunden tager hjem og opdager, at der er kommet en tilmelding mere til biografturen hun tidligere bookede billetter til. Derfor ringer vedkommende ind til biografen og får fat på ekspedienten Signe. Signe klikker på *Ret reservation* og indtaster kundens telefonnummer. Signe får nu vist en liste med alle reservationer for det pågældende nummer til forestillinger som endnu ikke har været vist. Derefter vælger Signe den pågældende reservation som skal redigeres. Kunden fortæller Signe, at der kommer en ekstra deltager til arrangementet, så de skal have et sæde mere tilføjet til reservationen - de bliver altså fire personer i stedet for tre. Signe kan se de allerede valgte sæder markeret med blå og kan desuden se, at sædet til højre for de reserverede sæder er reserveret til en anden kunde. Heldigvis er der ledigt til venstre, så der klikker Signe og tilføjer derved et ekstra sæde til reservationen. Nu er der fire blå sæder og Signe afslutter redigeringen ved at klikke på *Rediger reservation*. Så er reservationen rettet og kunden er tilfreds med den service vedkommende har fået.

1.2.3 Systemdesign

Vores system er overordnet baseret på data fra databasen. Systemet genererer layout ud fra forskellige data i databasen. Dette indebærer bl.a. hvilke film der er i film-tabellen, hvilke forestillinger der er tilknyttet til en given film, hvilke reservationer der er tilknyttet en given forestilling, hvilke reservationer der er tilknyttet et givent telefonnummer mv. Programmet er delt op, så mest mulig databehandling foretages hos klienten/brugeren. Dette indebærer objekter med informationer om de forskellige forestillinger, arrays med valgte sæder mv. Databaseforespørgsler er lavet så omfattende som muligt, så der udføres så få forespørgsler som muligt og data gemmes i objekter og arrays eller bliver benyttet med det samme i den grafiske fremstilling af programmet som eksempelvis ved generering af film-knapper.

Kapitel 2

Problemanalyse

2.1 Vores løsning

Den overordnede problemstilling vi har valgt at fokusere på, er hvordan man udvikler et simpelt og implicit program der både og brugervenligt og hvor databasen opbevarer det meste data.

Vores program løser problemstilligen ved at have alt information omkring forestillinger, sale, film og reservationer gemt i en database. Dette gør selve koden og databasen nem at vedligeholde, og vi har på den måde undgået unødvendig kodeduplikering [Brabrand 2014]. For at opfylde ønsket om brugervenlighed har vi valgt at vise programmet i ét vindue, hvor brugeren nemt kan navigere, ved hjælp af tabs, imellem de forskellige funktionaliteter programmet har.

2.1.1 Build Reservation

En af de større udfordringer, som man også har kunne tackle på en masse forskellige måder, var når biografsalen skulle bygges op rent grafisk. Den primære metode vi brugte her var *buildReservationScene(int showID)* i vores *controller.java* class. Alt informationen omkring biografsalen får vi fra vores database, dette bliver behandlet i metoden, her er det bl.a. de forskellige labels der bliver sat (cinemaNameLabel, movienameLabel etc.) Med informationen fra databasen kører vi et for-loop der kører igennem rækkerne, og inden i for loopet kører der endnu et for-loop der kører sæderne igennem for hver kolonne. Det er rimelig lige til og i højgrad den bedste måde vi kunne komme på at gøre det. Koden nedenfor viser hvordan det er lavet.

```
for(number of rows){  
    for(number of collums){  
        if(seats are reserved){  
            make seats red  
        } else {  
            make seats green  
        }  
    }  
}
```

Sæderne bliver grafisk vist som firkanter, her kunne man godt have lavet det som knapper. Vi valgte at bruge rectangle objekter fordi det var nemt at manipulere det udseende vi gerne ville give

dem uden at vi behøvede at gå på kompromis med funktionaliteten.

Til sidst er der måden hvorved du reserverer flere pladser. En oplagt mulighed ville være at have en dropdown menu i bunden af vinduet, der, når den klikkes, viser en liste fra med tal fra 1 til antallet af frie sæder (hvis der er nogle frie sæder tilbage). Denne mulighed er ganske fin, vi besluttede os dog for at gå med noget mere interaktivt hvor ekspedienten blot skal trække musen henover de sæder der skal vælges for at vælge dem. Dette er en feature der gør det nemmere for ekspedienten at vælge en masse sæder på engang og samtidig super fleksibel, hvis en reservation eksempelvis skal deles op på flere rækker.

2.1.2 Alternative løsninger

En alternativ måde at løse problemstillingen på ville være at fokusere på andre måder at booke billetter på. Vi har valgt at man først skal vælge film, derefter tid og tilsidst sæder. Man kunne have valgt at gøre det muligt for brugeren at vælge dato eller tid før valg af film. Vi har dog valgt den løsning vi syntes var mest brugervenlig. En anden mulighed er at man kunne have lavet en liste der viste *alle* forestillinger sorteret efter spilletid - uden at tage hensyn til sortering i forhold til film. Denne mulighed syntes vi designmæssigt ville fungere meget rodet og ikke intuitiv på nogen måde. Derudover kunne vi have valgt at gemme vores reservationer i programkoden frem for i databasen. Vi diskuterede dette i begyndelsen af projektet, men fandt det for besværligt at arbejde med, da det ville kræve større mængder kode, fyldte mere plads samt resultere i at vores program formentlig ville bruge længere tid på at compile og køre. Derudover er det langt mere hensigtsmæssigt at vedligeholde og rette data sådom reservationer i en database fremfor i en evt. tekstfil. Desuden ville en tekstfil med reservationer eller anden programdata kræve forskellige rettigheder affængig af operativsystemmet. Dette problem er slet ikke aktuelt i vores løsning, da alt sammen ligger i databasen.

2.2 Databasedesign

En af de to overordnede tabeller i vores database er *cinemas* som indeholder information omkring vores biografsale. Dette omfatter salens navn, id, antal rækker og antal sæder i hver række. Vores sæder er ikke repræsenteret som en tabel i vores database, men derimod gemt som et 2 dimentioneelt array.

Den anden vigtige tabel er *movies* som indeholder information omkring de film der går i biografen. Det eneste inforamtion denne tabel holder er navn og id. I begge tabeller er det, det pågældende id som fungerer som nøgle. Altså sal_id og movie_id.

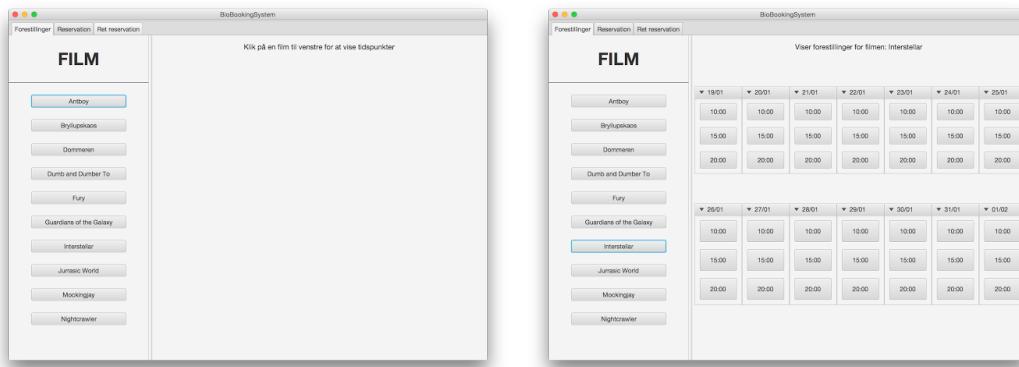
De to tabeller *movie* og *cinema* bliver koblet sammen i tabellen *show*. Her bruges deres nøgler til at sammensætte de foreskellige forestillinger som vores biograf kan vise. I *shows* tilføjes så et nyt ide til de enkelte forestillinger samt et timestamp - som er det tidspunkt forestillingen skal begynde.

Kapitel 3

Brugervejledning

3.1 Programmet

Programmet er designet med 3 tabs i toppen. *Forestillinger*, *Reservation* og *Ret reservation*. Når programmet åbnes starter brugeren i *Forestillinger-vinduet* og der vises en liste i venstre side med de film som kører i biografen. Brugeren klikker på den film, som kunden ønsker at reservere billetter til. Når en film vælges bliver tiderne for filmforestillinger vist for den pågældende film.

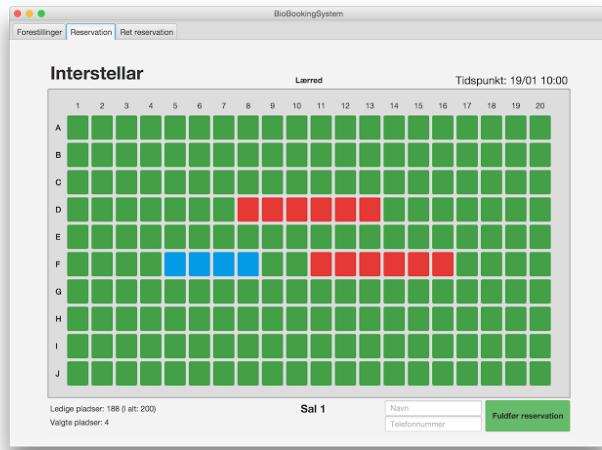


Figur 3.1: Forestillinger-vinduet før og efter en film er valgt

Klikker ekspedienten derimod ikke på noget, vil der stå følgene tekst *Klik på en film til venstre for at vise tidspunkter*, og har man ikke først valgt film og tidspunkt vil *Reservation* være ubrugelige.

Efter at have valgt film, vælger brugeren tidspunktet, som kunden ønsker at reservere til. Når en forestilling vælges bliver brugeren videreført til næste tab - altså reservation. Her vises et vindue med firkanter som illustrerer sæderne i biografen. Ledige sæder er illustreret som en grøn firkant og optagede sæder er illustreret som røde sæder. Brugeren klikker på de sæder som kunden ønsker at reservere - brugeren kan desuden holde musen nede og hive musen over de sæder som skal vælges. De valgte sæder skifter farve til blå og brugeren kan se hvor mange sæder vedkommende har markeret nederst i venstre hjørne.

Når sæderne er valgt skrives kundens navn og telefonnummer ind i tekstmærkerne nederst i

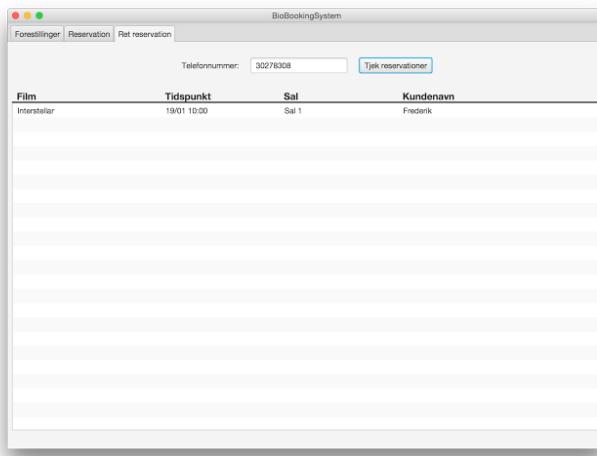


Figur 3.2: Biograf sal med reserverede og valgte sæder

højre hjørne. Afslutningsvist trykkes der på knappen *Fuldfør reservation*.

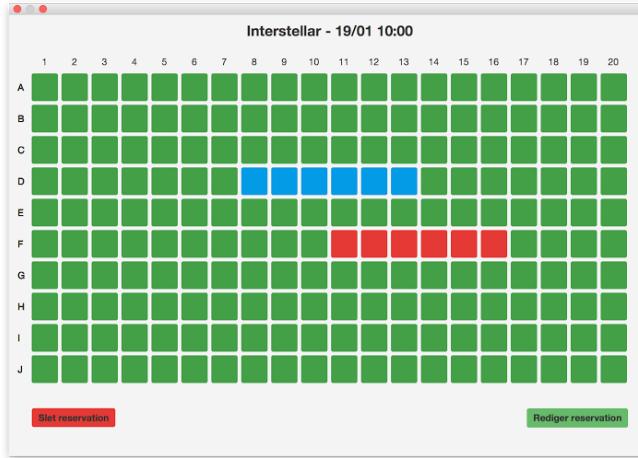
Hvis kunden fortryder sin bestilling eller har ændringer til en eksisterende bestilling, så klikker brugeren på tabben *Ret reservation* i toppen af programmet. Brugeren bliver taget til et vindue hvor kundens telefonnummer skal indtastes. Når telefonnummeret er indtastet og der er trykket på Enter-tasten eller på *Tjek reservationer*-knappen, så vises en liste nedenunder med de reservationer som kunden har tilknyttet til sit telefonnummer. Der vises kun reservationer for forestillinger som ikke er blevet vist - gamle reservationer vises altså ikke. Brugeren dobbeltklikker på den reservation der skal rettes og et nyt vindue åbner som ligner reservationsvinduet. De optagede sæder er røde, de ledige er grønne og de sæder som er tilknyttet til den pågældende reservation vises som blå. Brugeren kan tilføje eller slette sæder. Et valgt sæde er blåt og markeringen fjernes ved at klikke på sædet igen. Når ændringen til reservationen er valgt, så klikker brugeren på *Rediger reservation*. Skal reservationen derimod slettes helt, så klikker brugeren på *Slet reservation*.

Hvis kunden fortryder sin bestilling eller har ændringer til en eksisterende bestilling, så klikker brugeren på tabben *Ret reservation* i toppen af programmet. Brugeren bliver taget til et vindue hvor kundens telefonnummer skal indtastes. Når telefonnummeret er indtastet og der er trykket på Enter-tasten eller på *Tjek reservationer*-knappen, så vises en liste nedenunder med de reservationer som kunden har tilknyttet til sit telefonnummer.



Figur 3.3: Forestillinger-vinduet før og efter en film er valgt

Der vises kun reservationer for forestillinger som ikke er blevet vist - gamle reservationer vises altså ikke. Brugeren dobbeltklikker på den reservation der skal rettes og et nyt vindue åbner som ligner reservationsvinduet.



Figur 3.4: Vinduet hvor brugeren retter sin reservation

De optagede sæder er røde, de ledige er grønne og de sæder som er tilknyttet til den pågældende reservation vises som blå. Brugeren kan tilføje eller slette sæder. Et valgt er blåt og markeringen fjernes ved at klikke på sædet igen. Når ændringen til reservationen er valgt, så klikker brugeren på *Rediger reservation*. Skal reservationen derimod slettes helt, så klikker brugeren på *Slet reservation*.

Kapitel 4

Teknisk Analyse

4.1 Model View Controller

Vi har brugt Model-View-Controller (fork. MVC) til at adskille vores database og buildholder (model), vores controller og vores JavaFx brugergrænseflade (view) fra hinanden. Dette gør vores programkode mere overskuelig og nemmere at vedligeholde.

4.1.1 Moduler

Programmet har 2 klasser: DBConnect og buildHolder. DBConnect foretager alt der har med databasen at gøre. Dette er eksempelvis databaseforespørgsler hvor alle film returneres i et LinkedHashMap, hvor data såsom reservationer, salstørrelse mv. for en given forestilling hentes, hvor data såsom reservationer indsættes eller opdateres i databasen. Den væsentligste del af DBConnect-klassen må være funktionen getCon, der sørger for at der er en gyldig forbindelse til databasen.

```
public Connection getCon() throws SQLException {
    if(!con.isValid(30)) { // isValid takes seconds to wait for timeout as parameter
        con = DriverManager.getConnection(host, username, password);
    }

    return con;
}
```

En anden klasse er *buildHolder*. Et *buildHolder*-objekt lagrer data for en given forestilling. Der bruges getters og setters, så de data der lagres også kan hentes efterfølgende. *buildHolder* objektet bruges efterfølgende til at bygge reservationssalen med data som filmnavn, tidspunkt for forestillingen, størrelsen på salen (rows/columns), reserverede sæder mv.

Det er *controlleren* der gör brug af disse objekter, når data skal visualiseres og altså fremvises for brugeren.

4.1.2 Algoritmer

Når et DBConnect objekt initialiseres bliver der oprettet og gemt en connection til databasen i et felt. Funktionen *getCon* tester om denne forbindelse stadig er gyldig. Hvis den ikke er gyldig, så oprettes

der en ny gyldig forbindelse og denne returneres. Derved kan forbindelsen eksempelvis kaldes som `getCon().createStatement()`; og altid bruge en gyldig forbindelse. `getMovies` tager ingen parametre og returnerer et LinkedHashMap med filmens id som key og filmens navn som value. Sorteret alfabetisk efter filmnavn (A-Z).

`getMovieSchedule` tager film id som parameter og returnerer et LinkedHashMap med show id som *key* og tidspunktet for forestillingen (timestamp) som *value*. Data er hentet fra databasen med en sql-query hvor film id'et er det ene *where*-parametre og tiden for forestillingen er større/senere end det nuværende tidspunkt. Hele forespørgslen er sorteret med den nærmeste forestilling først - altså *time ASC*.

`getBuildSceneInfo` tager et forestillings id som parameter og laver et *buildHolder* objekt med information om forestillingen. Først hentes generel information såsom filmnavn, tidspunkt for forestillingen, størrelsen på salen (højde og bredde) m.fl. Efterfølgende hentes alle de reserverede sæder til den pågældende forestilling. Disse reservationer gemmes i et multidimensional array `[x][y]` af typen *Boolean*, hvor værdien i det ydre array er x-koordinaten til sædet og værdien i det indre array er y-koordinaten. Et reserveret sæde sættes til at være *true* på sædets plads - eksempelvis vil sæde 5:7 være `[5][7] = true`, hvis det er reserveret.

`insertReservation` er den metode, som sørger for at de valgte sæder og tilknyttede kundeinformationer indsættes i databasen. Der tages fire parametre: en *ArrayList* af typen string med sæderne, en *Integer* med forestillingens id, en *String* med kundens navn og ligeledes en med kundens telefonnummer. Sæderne gemmes som string med et kolon som splitter mellem sædets *x*- og *y-værdi* fx 5:8.

Først indsættes kundens navn, telefonnummer og forestillingens id i tabellen *reservations*. Derefter køres funktionen `getLastReservationId`, som returnerer senest indsatte reservations id. Dette id skal nemlig bruges til at linke de valgte sæder med reservationen. Dernæst løbes vores *ArrayList* igennem for at indsætte hvert sæde i reservation ind i vores database.

```
for(String seat : seats) {
    query = "INSERT INTO reservationlines (reservation_id, seat_x, seat_y) VALUES ('"
        + lastid + "', '" + seatInfo[0] + "', '" + seatInfo[1] + "');"
}
```

Koden ovenfor illustrerer hvordan vi indsætter seat i databasen. Vores sæde-string skal splittes op, så vi kan indsætte en x-værdi og en y-værdi i databasen. Man splitter vores string op i et *string-array* hvor index 0 er alt tekst før første kolon. Index 1 er alt tekst før andet kolon osv.

```
String[] seatInfo = seat.split(":");
```

Derved kan vi nu tilgå vores x-værdi som `seatInfo[0]` og vores y-værdi som `seatInfo[1]`. Hvis hele metoden forløber fejlfrit, så returneres der boolean true. Hvis der forekommer en *exception* så returneres false.

`getReservations` henter alle reservationer tilhørende et telefonnummer. Parametret er altså et telefonnummer og metoden returnerer et LinkedHashMap med reservations id som *key* og en *HBox* med

informationer om forestillingen som *value*. Databaseforespørgslen henter kun fremtidige reservationer - altså reservationer til forestillinger som endnu ikke er vist. Denne metode bliver brugt til at liste alle fremtidige reservationer op for en kunde, så de enten kan rettes eller slettes.

getResSeat bruges til at bygge den sal, hvor en reservation kan rettes eller slettes. Den tager to parametre. Det ene er reservations id'et og det andet er et buildHolder objekt. Reservations id'et er essentielt, da de reserverede sæder er bundet op på dette. BuildHolder objektet bruges til at bygge det multidimensionelle array som returneres, da vi skal bruge salens bredde og højde (columns/rows).

```
Boolean[][] resSeat = new Boolean[bh.getColumns() + 1][bh.getRows() + 1];
```

Vi ligger 1 til, da sæderne starter med 1 og et array starter med index 0. På den måde undgår vi en *NullPointerException*, da salen ellers vil prøve et tilgå et felt uden for vores array - også kaldet *ArrayIndexOutOfBoundsException*. Et optaget sæde vil returnere værdien true og et ledigt sæde false. Dette tjekker vi når vi looper igennem salens højde og bredde, som bliver gjort i vores controller, der også bygger den grafiske sal som er det vi ser.

getShowIdFromResId er en forholdsvis simpel metode. Den tager et reservations id som parameter og laver ud fra dette en database forespørgsel og finder derved ud af hvilken forestilling det pågældende reservations id tilhører. Dette forestillings id returneres som en int. På den måde kan vi sætte et buildHolder objekt op med det rette informationer, når en forestilling skal redigeres eller slettes. Metoden returnerer 0 hvis der ikke er fundet nogen forestilling til det pågældende reservations id.

updateReservation opdaterer en reservation. Der tages tre parametre: to *ArrayLists* af typen *String* og et reservations id af typen *int*. Den første arrayliste er de gamle reserverede sæder og den anden arrayliste er de nye reserverede sæder. Først slettes de gamle sæder og derefter indsættes sæderne fra den redigerede og dermed nye reservation. Hvis opdateringen af reservationen forløber uden fejl, så returnerer metoden *true* ellers returnerer metoden *false*.

deleteReservation sletter en reservation og dertilhørende sæder fra databasen. Metoden har et parameter som er reservations id af typen *int*. Da sæderne står er linket til dette reservations id, kan vi bare fjerne alle rækkerne i tabellen med reserverede sæder ud fra et givent reservations id. Først fjernes reserverede sæder og afslutningsvist fjernes selve orden med kundenavn, telefonnummer og forestillings id.

4.2 Design af Brugergrænseflade

Vi har valgt at opdele vores brugergrænseflade i tre dele: *Forestillinger*, *Reservation* og *Ret Reservation*. Vi diskuterede om det var mest brugervenligt at lave et vindue som skiftede mellem forskellige sider når man klikkede *næste side*, men blev enig om at det var mere simpelt og brugervenligt at lave et vindue med mulighed for maulet at skifte mellem siderne via en *tap-menu*. Siden *Forestillinger* er desuden også delt op i to dele - en del der oplister de forskellige film, og en der oplister spilletiderne for den valgte film. Vi har valgt det på den måde, fordi vi syntes det var designmæssigt bedre at udfylde hele vinduet ved at opdele det i de to dele, frem for at have en stor næsten tom side. En anden beslutning vi diskuterede omkring vores design, var hvorvidt man skulle kunne klikke på *Reservation* uden først at have valgt en forestilling. Vi havde overvejet at have en default biografsal til hvis man

ikke havde valgt forestilling, men besluttede at den løsning vi har nu, hvor der blot står *Du har ikke valgt nogen forestilling! Klik "Forestillinger" og vælg en film*, var mere brugervenlig. Under *Reservation* diskuterede vi hvorvidt rækker og sæder skulle have navne, og valgte at kalde rækkerne ved bogstaver og sæderne ved tal. At man både kan klikke på de enkelte sæder *og* trække hen over dem for at vælge sæder, er et resultat af at vi både ønskede at man skulle kunne vælge mange sæder på én gang, men også at man skulle kunne vælge sæderne til eller fra enkeltvis. Vi snakkede om at det skulle laves således at brugeren indtastede det ønskede antal sæder, og når brugeren så klikkede på ét sæde, ville det resterende antal sæder brugeren ønskede, automatisk blive valgt i form af de efterfølgende sæder. Vi valgte dog denne løsning fra, da vi syntes vores løsning var mere flexibel. Vores *View* er kun én klasse, nemlig *Controller*. Det er et bevidst valg vi har taget, og diskuteret. Vi besluttede os for at holde os til en klasse for enkelthedens skyld. Normalt ville man inddele *View* i flere klasser med hver deres funktion. Fx. kunne vi have tre *View*-klasser, en for *Forestillinger*, en for *Resrvation* og en for *Ret reservation*. Vi syntes dog ikke det var et problem at samle dem i én klasse pga programmets størrelse. Vi har således, for vores egen forståelses skyld, valgt at gøre det i én klasse.

4.2.1 Begrænsninger

Systemet er ikke begrænset til små biografer og kan derfor også bruges af større biografer med store sale. Systemet har ikke en administration til forestillinger og betalling, men det kan implementeres rimeligt simpelt i den nuværende kode, da systemet er lavet så dynamisk som muligt. Systemet baserer nemlig alt indhold i programmet fra databasen, pånær de statiske elementer som textfields og almindelige buttons. Man kan forestille sig, at systemet ville være svært at benytte for en biograf med over 3000 sæder, da sæderne i salene vil blive utroligt små og svære at markere. Som biografen er opbygget nu kan salene ikke være større end 25 rækker, medmindre man navngiver dem efter noget andet end alfabetet. Vores system er udviklet til mindre biografer og håndterer fint sale på op til 1000 sæder - dog skal man være god med museføringen, når vi har med så mange sæder at gøre, da størrelsen på sæderne skalerer til en statistisk scene.

4.2.2 Fejlmeddelelser

Programmet viser exceptions ved hjælp af metoden *catchPopUp*, som gör brug af vores dialog boks *newPopUp* der tager en string som skal vises i dialogvinduet. Metoden til at vise fejlmeddelelser tager en exception som parameter og denne exception fremvises i dialogen. Systemets *catchPopUp* vises altid teksten "Der er opstået en fejl" efterfulgt af den pågældende exception. *newPopUp*-metoden derimod bruges direkte, når beskeder skal printes til brugeren. Dette indebærer beskeder som "Bestllingen er gennemført", "Reservationen kan ikke udføres. Felterne navn og telefonnummer skal være udfyldt. Desuden skal du vælge sæder", "Reservationen er rettet!" m.fl. Når en reservation opdateres eller indsættes returnerer metoden til dette en Boolean. Hvis den returnerer false bliver en dialog vist med enten "Der er sket en fejl. Prøv igen.", "Der er sket en fejl! Reservationen kunne ikke rettes. Luk vinduet og prøv igen.".

Kapitel 5

Afprøvning

5.1 Brugerafprøvning

Vi har valgt at lave vores testing af programmets JavaFx-baserede brugergrænseflade som manuel brugerafprøvning. Den beskrevne fremgangsmåde i brugervejledningen er den tilsigtede måde brugeren skal forstå og bruge programmet. Brugeren havde ingen problemer med at reservere en forestilling, og det tog heller lang tid før brugeren havde fundet ud af at rette eller slette i reservationen. Vores bruger prøvede at indtaste bogstaver istedet for sit telefonnummer, men vores system reagerede som forventet på fejlen og et popup-vindue med en fejlmeddeelse dukkede frem. Den eneste uoverenstemmelse med hvad vi havde forventet, var at brugeren ikke opdagede at man kunne trække musen hen over sæderne for at markere flere sæder.

5.2 Unit test

Vores klasse *DBConnectTest* er en testklasse og indeholdende derfor udelukkende tests. Den primære funktion for for disse test er at teste om vores programtekst snakker rigtigt sammen med databasen. Vi har oprettet en testdatabase, som indeholder de samme tabeller som vores oprindelige database. Testdatabasens tabeller er dog tomme, og bruges kun til test. Fordi vi primært tester koden i relation til databasen er det ikke rene unit test vi udfører. Faktisk er det det man kalder integration tests. Integration test er tests der bruger en database, et netværk eller et andet eksternt system som fx en mailserver. Typisk for en integration test er også at den udfører I/O. I vores tilfælde tager vores tests input fra en database og udskriver et grafisk output til skærmen.

Testen *testGetMovies()* tester om film og ide passer sammen. Dette gøres ved at vi giver testen et forventet output, som er bestemte id's tilknyttet bestemte film. Når testen kører, testes derfor om det forventede output passer overens med det faktiske uddata.

testTimeStamp() tjekker hvorvidt en given films forestillinger bliver sorteret korrekt efter tid. Det gøres ved at lave et while-loop som kører igennem alle tidspunkterne og tjekker om det nuværende timestamp er lavere end det foregående.

For at tjekke om et reserveret sæde vise med rød farve, og dermed ikke har nogle funktioner. Sædernes funktioner er nemlig tilknyttet deres farve, således at de røde sæder ikke er mulige at klikke på. *testReservedSetColor()* henter et reserveret sæde og derefter sætter dens farve lig med den forventede

røde farve.

testInsertReservation() tester om metoden *InsertReservation()* rent faktisk opretter en reservation.

```
@Test  
public void testInsertReservation(){  
    ArrayList<String> seats = new ArrayList<String>();  
    seats.add("1:2");  
    dbConnect.insertReservation(seats, 120, "Amanda", "26802103");  
    dbConnect.getLastReservationId();
```

Ovenfor ses en del af testmetoden *testInsertReservation*. I metoden begynder vi med at oprette en *ArrayList* til sæder, og herefter indsætter vi et sædet i Array'et som har plads [1:2]. Tredje linje kalder *insertReservation* metoden fra *DBConnect* klasse, og giver metoden de værdier den skal bruge for at oprette en reservation. Den sidste linje i *testInsertReservation* bruger metoden *getLastReservation* metoden, igen fra *dbConnect*, til at hente det sidste oprette reservations id - som er den reservation vi lige har oprettet i linjen ovenover.

Den sidste testmetode *testGetReservation* tester metoden *getReservation*. Dette gøres ved at testen først tjekker om en reservation med telefonnummeret 11223344 eksisterer, og returner *true* hvis reservationen *ikke* eksisterer. Herefter opretter testen selv en reservation under det samme telefonnummer, og til sidst tjekker testen igen om der eksisterer en reservation med det givne telefonnummer. Denne gang skal testen returnere *true* hvis der *eksisterer* en reservation.

Både *testInsertReservation* og *testGetReservation* har en *@After* test, som sletter de reservationer der bliver lavet i de to tests. Dette gøres ved at *@After* kalder *deleteReservation* fra *dbConnect*.

5.3 Resultat

Ud fra brugerafprøvningen kom vi frem til den konklusion at vores brugergrænseflade fungerer som forventet. Vores indtryk fra brugeren er at programmets design er rimelig nemt at overskue, og simpelt at bruge. Gennem vores unit test har vi tjekket at databasen fungerer optimalt og at reservationer bliver lavet og gemt rigtigt. Konklusionen ud fra vores forskellige tests er derfor at vores program fungerer som vi har tænkt det.

Kapitel 6

Arbejdsprocessen

6.1 Projektet

Vi har i løbet af udarbejdelsen af vores løsning lært at konstruere et bookingsystem der skal betjenes af en ekspedient i en mindre biograf. Vi har analyseret den givne problemstilling og kommet frem til et optimalt programdesign.

Systemets formål, opbygning og virkemåde står klart præsenteret både for brugere af systemet og eventuelle udviklere der skal videreudvikle systemet.

De forskellige metoder i systemet er blevet afprøvet i muligt omfang og metoderne fungerer efter hensigten. Det har været svært at teste brugergrænsefladen, men de brugere, der har testet den, mener at den lever op til formålet, hvilket har været hensigten, da vi udviklede den.

Vores system gør brug af en- og to-dimensionelle arrays, når vores sæder skal reserveres og rettes. Systemet gør brug af flere klasser fra Javas klassebibliotek heriblandt collections. Vi gør brug af klassehierarkier i vores test af systemet, og vi har organiseret klasserne så hensigtsmæssigt og effektivt som muligt. Systemet gør brug af programløkker, men sortering sker i databaseforespørgslerne. Databasen er struktureret med gode relationer, som skaber en god rød tråd mellem tabellerne.

6.2 Gruppen

Arbejdet med projektet er gået meget godt. Vi startede med at diskutere hvordan programmet skulle opbygges rent grafisk og hvordan det skulle repræsenteres i tabeller. Da vi fandt enighed om at fornuftigt design gik vi i gang med at udvikle det. Vi delte projektet op, således to personer stod for kodning af selve systemet og en person stod for at teste systemet og dets metoder. Vi har løbende set tilbage på arbejdsprocessen og vi har optimeret den oprindelige idé, hvis vi er kommet på mere effektive metoder eller designmæssige forbedringer.

Kapitel 7

Konklusion

Gennem Java har vi opbygget et simpelt program der bygger på *Model View Controller*. Vores *Model*, som er vores brugergrænseflade, er skabt primært i JavaFx, og der har designmæssigt være fokus på brugervenlighed. Systemet lever op til de stillede krav og kan nemt og effektivt udføre alle arbejdsopgaverne. Systemet er brugervenligt og designet på en rimelig intuitiv måde. Brugeren kan se en liste over de film der går i biografen og ud fra disse vælge en forestilling til en specifik film på et specifikt tidspunkt. Der er mulighed for at reservere, redigere og slette reservationer i systemet. Alt vores data er opbevaret i vores database, og styres således fra *DBConnect* klassen. Vi har testet vores program og nået det resultat at programmet fungerer efter hensigten. Vores program klarer de respektive fejlmeddelelser med forskellige pop-up vinduer der fortæller brugeren hvilken fejl der er opstået. Vi har valgt at lave biografsalene relativ små, fordi det var hvad der stod i opgavebeskrivelsen, men vores program er lavet således at der ikke er begrænsning på salendes størrelse, selvfølgelig indenfor rimelighedens grænse. Skulle vi arbejde videre med programmet har vi nogle ændringer vi gerne ville lave:

- Mulighed for køb/salg af billetter
- Mulighed for køb/reservering over nettet
- Flere billetluger
- Trailer for hver film
- Mulighed for at oprette en bruger

Vi finder vores løsning tilfredstillende og kan ikke, udover overnævnte ændringer, tænke på implementationer eller bedre alternativer.

Bibliografi

- Brabrand, Claus (2014). *Klassedesign*. URL: https://learnit.itu.dk/pluginfile.php/113876/mod_resource/content/0/GRP0-08.pdf (sidst set 23.09.2014).
- Madsen, Jesper (2014). *Projekt problemformulering*. URL: https://learnit.itu.dk/pluginfile.php/116426/mod_resource/content/1/Projekt%20Problemformulering.pdf (sidst set 03.12.2014).

Figurer

3.1	Forestillinger-vinduet før og efter en film er valgt	9
3.2	Biograf sal med reserverede og valgte sæder	10
3.3	Forestillinger-vinduet før og efter en film er valgt	11
3.4	Vinduet hvor brugeren retter sin reservation	11
B.1	Skitse over brugergrænseflade	26
C.1	Mock-up af biografsalen	27
D.1	Overvejelse omkring tabellen Shows	28
D.2	Overvejelse omkring tabellen Reservationer	28
D.3	Overvejelse omkring tabellen Cinemas	29
D.4	Overvejelse omkring tabellen Reservationlines	29
E.1	Overvejelse omkring programstrukturen strukturen	30
F.1	Udkast til forestillingsoversigt	31

Bilag A

Tests

```
package test;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import model.DBConnect;
import model.buildHolder;
import org.junit.After;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import java.sql.*;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.Map;

public class DBConnectTest {
    private Connection con;
    private Statement st;
    private ResultSet rs;
    private int lastid;
    private int lastid1;
    private DBConnect dbConnect;

    public DBConnectTest() throws SQLException {
        dbConnect = new DBConnect("jdbc:mysql://mysql.itu.dk:3306/KaffeklubbenTest",
                               "Kaffekluben2", "kp8473moxa");

        con = DriverManager.getConnection("jdbc:mysql://mysql.itu.dk:3306/KaffeklubbenTest",
                               "Kaffekluben2", "kp8473moxa");
        st = con.createStatement();
    }
}
```

```

//denne test tjekker om film og ide passer med det forventede - ØG om rækkefølgen er rigtig.
@Test
public void testGetMovies() throws SQLException {
    Map<Integer, String> expected = new LinkedHashMap<>();
    expected.put(3, "Antboy");
    expected.put(7, "Bryllupskaos");
    expected.put(6, "Dommeren");
    expected.put(9, "Dumb and Dumber To");
    expected.put(2, "Fury");
    expected.put(8, "Guardians of the Galaxy");
    expected.put(1, "Interstellar");
    expected.put(11, "Jurassic World");
    expected.put(5, "Mockingjay");
    expected.put(10, "Nightcrawler");

    assertEquals(expected, dbConnect.getMovies());
}

//denne test tjekker tidspunkterne på film nr 1 virker - 1 kan udskiftes med andre id's.
// Virker stadig.
@Test
public void testTimeStamp() throws SQLException {
    st = con.createStatement();
    String query = "SELECT * FROM shows WHERE movie_id=1 ORDER BY time ASC";
    rs = st.executeQuery(query);
    Timestamp tmstmp = new Timestamp(0);
    while (rs.next()) {
        //System.out.println("Id eksisterer"); //id 4 eksisterer ikke
        assertTrue(tmstmp.getTime() < rs.getTimestamp("time").getTime());
        if (rs.getTimestamp("time").getTime() > tmstmp.getTime()) {
            tmstmp = rs.getTimestamp("time");
        }
    }
}

//denne test tjekker om et reserveret sæde for farven rød
@Test
public void testReservedSetColor() throws SQLException {
    buildHolder bh = dbConnect.getBuildSceneInfo(109);
    Boolean[][] resSeat = bh.getResSeat();
    for(int i = 1; i < 15; i++) { //hvorför 15?
        for(int j = 1; j < 11; j++) { //hvorför 11?
            double width = (879-8*bh.getColumns()-8)/bh.getColumns();
}

```

```

        double height = (521-8*bh.getRows()-8)/bh.getRows();
        final Rectangle r = new Rectangle(width,height);
        int x = i;
        int y = j;
        if(resSeat[i][j] != null) {
            if(resSeat[i][j]) {
                r.setFill(Color.web("#E53935"));
            }
        } else {
            r.setFill(Color.web("#43A047"));
        }
        if(i==15 && j==9){
            assertTrue(r.getFill().toString().equals("0xe53935ff"));
        }
    }
}

//tester om reservationer der bliver lavet, gemmes i databasen
@Test
public void testInsertReservation() throws SQLException {
    ArrayList<String> seats = new ArrayList<String>();
    seats.add("1:2");
    dbConnect.insertReservation(seats, 1, "Amanda", "26802103");
    dbConnect.getLastReservationId();

    rs = st.executeQuery("SELECT reservations.id, seat_x, seat_y FROM reservations," +
        " reservationlines WHERE reservationlines.reservation_id" +
        " = reservations.id AND show_id = '1' AND customer_name = 'Amanda'" +
        " AND customer_phone = '26802103'");
    while(rs.next()){
        lastid = rs.getInt(1);
        System.out.print(lastid);
        //assertEquals(rs.getString("id"), (lastid));
        //assertEquals(rs.getInt("seat_x"), 1);
        //assertEquals(rs.getInt("seat_y"), 2);
    }
}
//sletter den reservation vi har lavet ovenfor.
@After
public void deleteReservationtest() throws SQLException {
    dbConnect.deleteReservation(lastid);
}

```

```

//Denne test tester om der kommer en liste ud når man har reserveret for et bestemt nummer.

@Test
public void testGetReservation() throws SQLException {
    Boolean reservations = dbConnect.getBooleanReservations("11223344");
    //    assertTrue(!reservations);
    ArrayList<String> seats = new ArrayList<String>();
    seats.add("2:2");
    String customerName = "Markus";
    String phoneNumber = "11223344";
    int showId = 109;
    dbConnect.insertReservation(seats, showId, customerName, phoneNumber);
    reservations = dbConnect.getBooleanReservations("11223344");
    assertTrue(reservations);

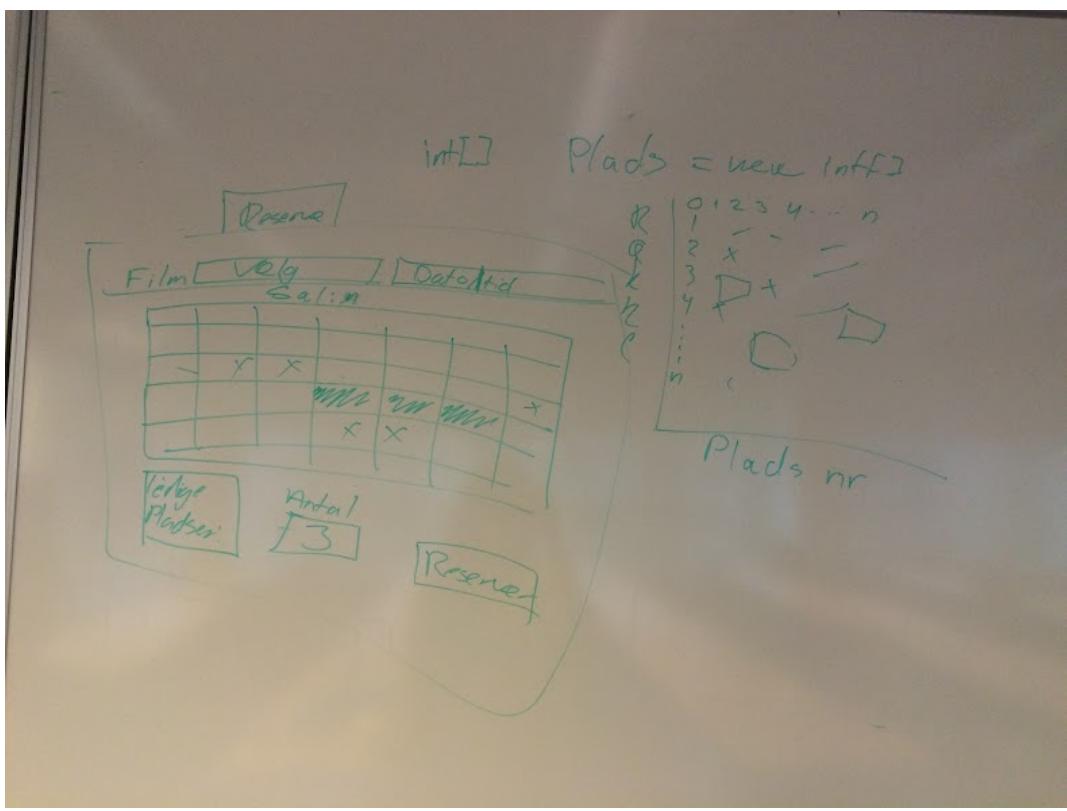
    rs = st.executeQuery("SELECT id FROM reservations ORDER BY id DESC LIMIT 1");
    if (rs.next()) {
        lastid1 = rs.getInt(1);
    }
}

//sletter indput til databasen som blev lavet opover.
@After
public void deleteInputReservation() throws SQLException {
    dbConnect.deleteReservation(lastid1);
}
}

```

Bilag B

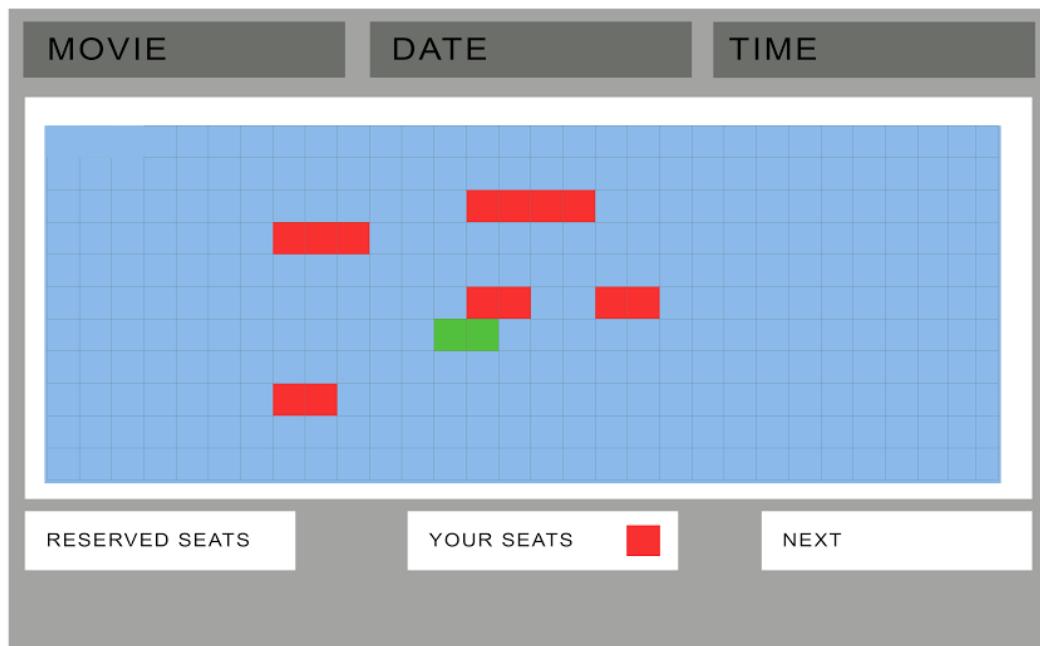
Brugergrænseflade



Figur B.1: Skitse over brugergrænseflade

Bilag C

Mock-up



Figur C.1: Mock-up af biografsalen

Bilag D

Databasetabeller

Shows		
id	int(5)	NOT NULL
cinema_id	int(5)	NOT NULL
movie_id	int(5)	NOT NULL
time	int(12)	NOT NULL

Figur D.1: Overvejelse omkring tabellen Shows

Reservations		
id	int(10)	NOT NULL
show_id	int(5)	NOT NULL
customer_name	varchar(50)	NOT NULL
customer_phone	int(8)	NOT NULL

Figur D.2: Overvejelse omkring tabellen Reservationer

Cinemas					
id	int(5)	NOT NULL	// auto increment	//primary key	
name	varchar(30)	NOT NULL	// name of cinema	//max 30 lenght	//unique key
columns	int(2)	NOT NULL	//width of cinema		
rows	int(2)	NOT NULL	//height of cinema		
			//number of seats = columns*rows		

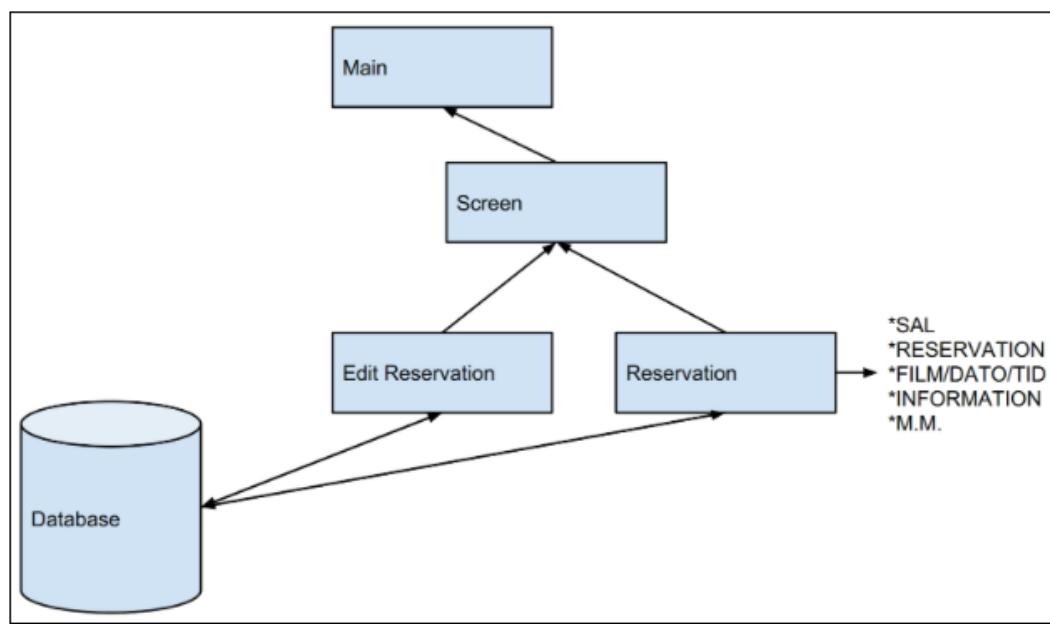
Figur D.3: Overvejelse omkring tabellen Cinemas

Reservationlines		
id	int(10)	NOT NULL
reservation_id	int(10)	NOT NULL
seat_x	int(3)	NOT NULL
seat_y	int(3)	NOT NULL

Figur D.4: Overvejelse omkring tabellen Reservationlines

Bilag E

Strukturen



Figur E.1: Overvejelse omkring programstrukturen strukturen

Bilag F

Forestillinger

Sal 1	Film	MovielD	Sal 2	Film	MovielD	Sal 3	Film	MovielD
10:00	Interstellar	1	10:00	Antboy	3	10:00	Dommeren	6
12:30	Fury	2	12:30	Mockingjay	4	12:30	Bryllupskaos	7
15:00	Interstellar	1	15:00	Antboy	3	15:00	Dommeren	6
17:30	Fury	2	17:30	Mockingjay	4	17:30	Bryllupskaos	7
20:00	Interstellar	1	20:00	Antboy	3	20:00	Dommeren	6
22:30	Fury	2	22:30	Mockingjay	4	22:30	Bryllupskaos	7

Sal 4	Film	MovielD	Sal 5	Film	MovielD
10:00	Guardians of the Galaxy	8	10:00	Nightcrawler	10
12:30	Dumb and Dumber To	9	12:30	Jurrasic World	11
15:00	Guardians of the Galaxy	8	15:00	Nightcrawler	10
17:30	Dumb and Dumber To	9	17:30	Jurrasic World	11
20:00	Guardians of the Galaxy	8	20:00	Nightcrawler	10
22:30	Dumb and Dumber To	9	22:30	Jurrasic World	11

Figur F.1: Udkast til forestillingsoversigt