

SUBMISSION OF WRITTEN WORK

Class code:

BSGRPRO1KU

Name of course:

Grundlæggende Programmering

Course manager:

Claus Brabrand

Course e-portfolio:

Thesis or project title:

ClickTicket: Et biograf reserveringssystem

Supervisor:

Signe Kyster

Full Name:

Ingrid Maria Christensen

Birthdate (dd/mm/yyyy):

26/01-1997

E-mail:

inch @itu.dk

1. Melissa Høegh Marcher

27/10-1996

mhom @itu.dk

2. Emil Joakim Jensen Bartholdy

27/01-1993

emba @itu.dk

3.

@itu.dk

4.

@itu.dk

5.

@itu.dk

6.

@itu.dk

7.

@itu.dk

ClickTicket: Et biograf reservationssystem

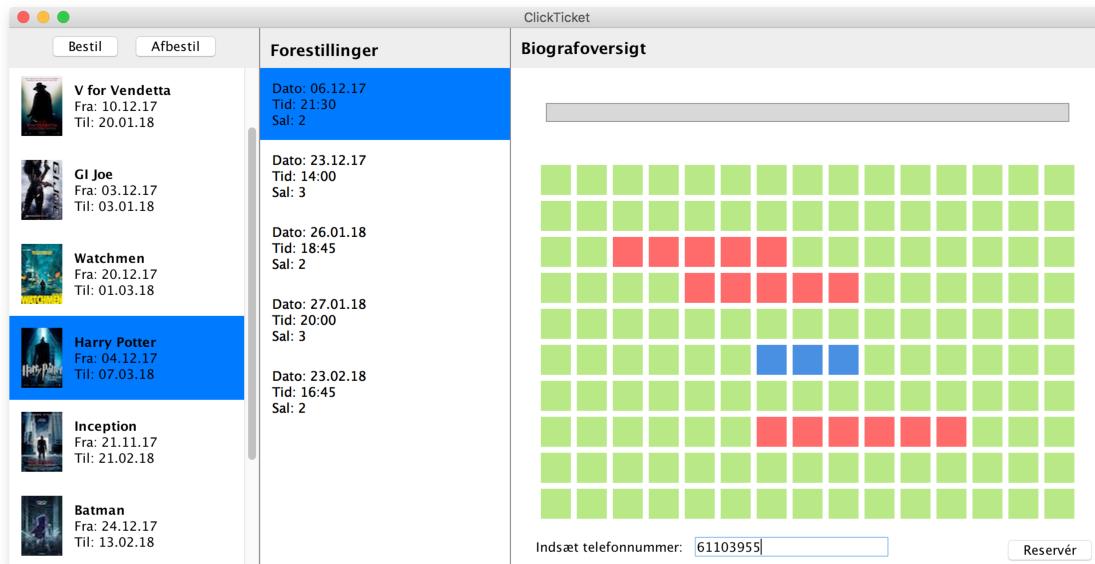
Gruppe Z:

Emil Bartholdy (emba@itu.dk)

Ingrid Maria Christensen (inch@itu.dk)

Melissa Høegh Marcher (mhom@itu.dk)

Afleveringsdato: 15. december 2017



Antal tegn (med mellemrum): 34800 (14,5 normalsider)

Indhold

1 Indledning	3
2 Baggrund og Problemstilling	3
2.1 Problemområde	3
2.2 Brugsmæssige krav til programmet	3
2.3 Systemmæssige krav	4
3 Problemanalyse	5
3.1 Designbeslutninger vedrørende programmets interne struktur	5
3.1.1 Designbeslutninger vedrørende forbindelsen mellem database og program .	5
3.2 Designbeslutninger vedrørende databasens struktur	6
3.3 Designbeslutninger vedrørende brugergrænsefladen	6
4 Brugervejledning	7
4.1 Fejlmeddelelser i programmet	10
5 Teknisk beskrivelse af programmet	10
5.1 Databasedesign	10
5.2 Programmets interne struktur	11
5.3 Brugergrænsefladens struktur	14
5.4 Dokumentation af program	16
6 Afprøvning	16
7 Refleksion over arbejdsprocessen	17
8 Konklusion	18
9 Litteratur	19
10 Bilag	20
10.1 Substantiv-verbum metoden	20
10.2 Papirprototype af ClickTicket	21
10.2.1 Reservering af biografbillet	21
10.2.2 Afbestilling af biografbillet	21
10.3 CRC-kort	22
10.4 Reservationstabell fra databasen	22
10.5 MySQL Commands	22
10.6 Brugertest 2	25
10.7 Testcases - unit test	25
10.7.1 <i>CinemaController</i> unit tests	25
10.7.2 <i>DBAccess</i> unit test	27
10.7.3 <i>Serializer</i> unit test	28
10.7.4 <i>View</i> unit test	31
10.8 Java Doc	31

1 Indledning

Denne rapport er udarbejdet i forbindelse med et 3-ugers projekt i kurset ‘Grundlæggende Programmering’ på 1. semester softwareudvikling (bachelor). Vejleder i projektet har været Signe Kyster (signek@itu.dk). I projektet har vi udviklet et program kaldet ‘ClickTicket’, der skal fungere som et reserveringssystem i en biograf. Tak til biografmedarbejder Rikke fordi hun ville medvirke til interview og brugertests i dette projekt.

2 Baggrund og Problemstilling

2.1 Problemområde

Den sammenhæng, som programmet skal fungere i, er en mindre biograf. Programmet skal bruges af den ekspedient, som håndterer kundehenvendelser i forbindelse med biografbesøg. Kundehenvendelser sker enten i biografens ene billetluge eller over telefonen. Ved en kundehenvendelse giver kunden sit telefonnummer som identifikation. Selve biografen viser flere forskellige film på forskellige tidspunkter og har flere sale. Salene kan have et forskelligt antal stolerækker og antal stole per række.

2.2 Brugsmæssige krav til programmet

De brugsmæssige krav til programmet er hovedsageligt formuleret som arbejdsopgaver. Disse arbejdsopgaver til programmet er fundet gennem tre processer: gennemlæsning af opgaveformuleringen for projektet, interview med en ekspedient fra Nordisk Film Biografer, samt brugertest af en tidlig papirprototype af programmet på samme ekspedient. Kravene til programmet er listet i tabel 1.

ID	Arbejdsopgaver
1	En kunde ringer ind for at reservere tre sidestillede sæder til en film i aften.
2	En kunde ønsker at vide om en film kan ses i biografen i 3D.
3	En kunde ønsker at tilføje og/eller fjerne en eller flere sæder i kundens reservation.
4	En kunde ønsker at kende til tiden på en bestemt forestilling.
5	En kunde ønsker at flytte hendes reservation til et senere tidspunkt på samme dag.
6	En kunde ønsker at reservere en billet til den næste film, der vises i biografen.
7	En kunde ønsker at annullere hendes reservation.
8	En kunde ringer ind for at bestille 17 billeter til samme forestilling.
9	En kunde foretager sig to reservationer i samme telefonopkald.
10	En kunde ønsker at sidde helt forrest eller bagerst i biografen.
11	En kunde ønsker at reservere en billet til en forestilling, der vises om et år.
12	En kunde ønsker at ændre siddepladser i en reservation.
13	En kunde ønsker at opgradere sin billet fra en 2D til en 3D film.
14	En kunde ringer ind og ønsker at bestille sæder, der ikke er sidestillede i biografen.
15	Ekspedienten ønsker at vide hvor mange sæder, hun allerede har markeret som reserverede under oprettelse af en ny reservation.

Tabel 1: Brugsmæssige krav til programmet

Gennem arbejdsopgaverne ses to kernearbejdsopgaver: ekspedienten skal kunne bestille en reservation og slette en reservation til en given forestilling. Ved at opfylde disse to krav, vil man kunne dække mange af arbejdsopgaverne beskrevet i tabel 1. Ved interview af biografekspedienten fik vi også fundet krav, der ikke kunne formuleres som arbejdsopgaver, men nærmere generelle retningslinjer for programmet. I denne forbindelse fortalte biografekspedienten, hvordan det program hun benytter på arbejdet er forvirrende og uoverskueligt med mange knapper og funktioner, hun ikke forstår. Dette fortæller om vigtigheden af brugervenligheden i programmet. Et af vores krav til programmet er derfor, at det skal være nemt, simpelt og overskueligt at benytte.

2.3 Systemmæssige krav

De generelle systemmæssige krav til programmet er listet i tabel 2. Overordnet set skal programmet kun køres på en enkelt computer. Programmet skal kunne indlæse alle nødvendige data fra en database, hvorefter databehandlingen sker i selve programmet (et klienttungt system).

ID	Systemmæssige krav
1	Programmeres i Java.
2	Skal kun køres på én computer.
3	Benytter MySQL til at gemme data i en relationel database.
4	Benytter Java <i>Swing</i> til opbygning af brugergrænsefladen.
5	Er modulært i sit design med lav kobling (coupling) og høj sammenhæng (cohesion).
6	Skal kunne skrive nye reservationer til MySQL databasen, så data gemmes til næste programstart.
7	Vigtige komponenter testes med unit tests.

Tabel 2: Systemmæssige krav til programmet

De data, som programmet skal kunne håndtere er beskrevet og listet i tabel 3. Disse data er nødvendige for at kunne udføre de beskrevne arbejdsopgaver i afsnit 2.2.

Datum	Beskrivelse
Film	De film, som går i biografen.
Sal	Det rum hvori film vises.
Forestilling	Er sammensætningen af sal, film og tidspunkt. Altså hvor og hvornår en given film vises.
Kunde	For hver kunde gemmes et ID i form af et telefonnummer
Sæde	Et sæde i en given sal. Skal gemmes med koordinater for hvor sædet er i en biograf.
Reservation	Er sammensætningen af kunde, forestilling og sæder.

Tabel 3: Data i programmet

3 Problemanalyse

3.1 Designbeslutninger vedrørende programmets interne struktur

I vores overvejelser om programmets interne struktur har vi valgt at benytte os af Model-View-Controller (*MVC*) strukturmønstret, for at adskille datamodellen (*Model*) fra brugergrænsefladen (*View*) og programlogikken (*Controller*). Således afkobles store dele af programmet fra hinanden, og gør det nemt for de enkelte programmører at udvikle på programmet parallelt. I *MVC* repræsenterer *Model* det data, som programmet skal holde, *View* repræsenterer visualiseringen af disse data og *Controller* håndterer både *Model* og *View* ved at kontrollere dataflow ind i *Model* og opdaterer *View* når data ændres (Brabrand, 2017, s. 21).

I analysefasen af et projekt er målet at forstå programmets domæne ved at identificere fænomener (konkrete ting fra virkeligheden), som man kan abstrahere til koncepter (generaliseret idé fra en kollektion af fænomener med fælles karakteristika) (Brabrand, 2017, s. 3). Resultatet af dette bruges til at vejlede designet af *Model*-delen af programmet. For at identificere disse koncepter skrev vi en problembeskrivelse af domænet med input fra opgaveformuleringen for projektet og vores interview med en biografekspedient. Herefter benyttede vi substantiv/verbum-metoden (Barnes Költing, 2016, s. 530) til at identificere mulige klasser (se bilag 10.1). For at finde ud af interaktionerne mellem disse klasser lavede vi Class-Responsibilities-Collaborator (CRC) kort (Barnes Költing, 2016, s. 532) (se bilag 10.3). I denne forbindelse besluttede vi, at klasserne *Customer* og *Employee* ikke ville blive inkluderet i implementeringen af programmet. Selvom det godt kunne tænkes at fx en klasse *Employee* skulle holde på loginoplysninger, og at *Customer* skulle holde på oplysninger om kunden (fx kundebonuser), vurderede vi, at dette ville være uden for programmets omfang. Ift. kundrepræsentationen i programmet valgte vi derfor, at en kunde i stedet skulle repræsenteres som et telenummer knyttet til klassen *Reservation*. Dette ville forsimple databasen og lette implementeringen ift. den begrænsede mængde tid til projektet.

Den type data kollektion vi benytter oftest og har valgt til at holde data i programmet er *ArrayLists*. Fx valgte vi at benytte en *ArrayList* til at holde reservationer, da antallet af reservationer ændrer sig som nye reservationer oprettes. I forhold til listerne for *theaters*, *films* og *showings*, kunne vi have valgt at lave disse til *arrays*, da alle tre holder på et konstant antal objekter, som ikke vil ændre sig i løbet af programmets køringstid. Dog valgte vi det andet, så man i principippet har mulighed for udbygge biografen med flere *theaters*, *showings* og *films* uden at omkode det.

3.1.1 Designbeslutninger vedrørende forbindelsen mellem database og program

Vi har valgt, at programmets forbindelse til databasen skal være minimal. Her har vi besluttet, at programmet kun skal gemme nye reservationer ved nedlukning. Dermed benyttes databasen ikke under programkørsel. For det første fører dette til en højere programhastighed ved at programmet ikke hyppigt skal forbinde, læse og skrive til databasen. For det andet simplificerer det implementeringen af forbindelsen mellem program og database. Der er dog også ulemper ved dette design. Fx vil nye reservationer gå tabt, hvis der ikke er internetforbindelse ved programlukning, eller hvis computeren går ud mens programmet kører. Med mere tid til projektet, kunne man have undgået dette problem, ved at oprettet en lokal database, som løbende gemmer nye reservationer,

og som synkroniserer med hoveddatabasen ved programlukning.

En anden beslutning vedrørende forbindelse mellem database og program er, at hele tabellen slettes for reservationer før programmet indsætter de gamle og nyoprettede reservationer ind i tabellen igen. Denne løsning ville ikke fungere i en stor biograf med mange tusinde reservationer af hastighedsmæssige årsager. Vi valgte dog alligevel denne implementering af tidsmæssige årsager, samt at der i opgaveteksten står, at vi skal udvikle bookingsystemet til en mindre biograf. Med mere tid, kunne man i stedet have valgt at tilføje en metode, der gennemløber eksisterende reservationer i databasen og kun tilføjer nyoprettede og sletter afbestilte reservationer, som programmet kører.

3.2 Designbeslutninger vedrørende databasens struktur

Til at designe databasen benyttede vi et *entity-relationship-diagram (ERD)* (se evt. Moore, 2000). Her opstilles tabellerne med dertilhørende felter og relationer, således at databasen fremstår visuelt (se figur 5). I denne forbindelse besluttede vi, at databasens struktur skulle reflektere programnets interne datastruktur mest muligt. Således er det nemmere at oversætte databasen til objekter i programmet og omvendt. Derudover blev det tilstræbt, at databasen er så normaliseret som muligt, bl.a. ved ikke at gemme al data i en enkelt tabel og ved at gemme atomare (ikke opdelelige) værdier i tabellernes felter. Derved forhindres redundans. I vores database gemmes bestilte sæder dog ikke atomart. Hvis en reservation tilknyttes flere sæder, vil disse blive gemt som koordinater i det samme felt, hvilket ikke anses som værende godt databasedesign (se bilag 10.4 for reservationstabell med eksempladata). Et bedre design ville have været at gemme sædernes koordinater, samt deres status (reserveret vs. ikke reserveret) i en separat tabel i separate felter. Dette ville til gengæld også indebære en væsentligt mere kompliceret implementering i programmet og databasen. Kombineret med den begrænsede mængde tid til projektet, valgte vi at gå med den mere simple løsning på at lagre sæder i databasen.

I vores database kunne det have været hensigtsmæssigt at benytte *foreign keys* til at definere relationerne mellem tabellerne. Det var dog ikke muligt at oprette *foreign keys* i databasen. Vi har heller ikke benyttet *join*-funktionen i databasen. Man kunne fx have brugt denne funktion til at finde reservationer baseret på telefonnumre, men det ville kræve, at vi også havde en *customer*-tabel. Trods disse mangler har det stadigvæk været muligt indlæse data til objekter og definere objekternes relation til hinanden ved brug af de enkeltes rækkers ID (se afsnit 5.2).

For at gøre brugergrænsefladen mere visuel, har vi også inkluderet billeder af filmenes plakater. Trods disse billeder ikke fylder meget (ca. 6-7 KB pr. stk) har vi besluttet ikke at lagre billederne i vores database, men i stedet ligge dem i en folder lokalt i programmet. Dette gør at programmet kan starte hurtigere op, fordi programmet ikke skal downloade så meget data ved programmets start.

3.3 Designbeslutninger vedrørende brugergrænsefladen

Tidligt i projektforløbet lavede vi en papirprototype af programmet, som skulle tjene to formål. 1) Papirprototypen skulle hjælpe os med at udrede, hvordan programmet skulle benyttes af en tænkt bruger. 2) Derudover skulle den hjælpe med at validere brugergrænsefladedesignet, før vi begyndte at bruge tid på at kode den. Papirprototypen blev herefter testet ved brugertest med

en biografekspedient. På baggrund af den feedback, som biografekspedienten har givet i både interview og brugertest, har det vejledt vores designbeslutninger vedrørende brugergrænsefladen. Et billede af papirprototypen ses i bilag 10.2 og den tidlige brugertest er beskrevet i afsnit 6.

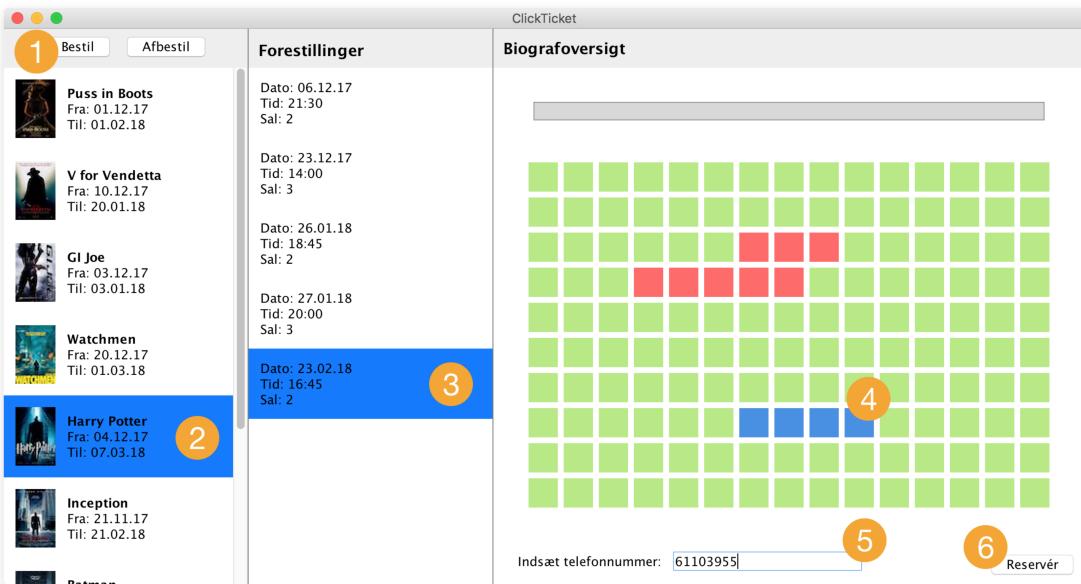
En af de designbeslutninger vi har taget ift. brugergrænsefladen er, at alle arbejdsopgaver skal kunne gøres i et enkelt vindue, fremfor at delopgaver bliver præsenteret i hvert sit vindue (fx valg af film, biografsæder osv.). Dette vindue er delt op i tre hovedkomponenter, med to sidebarer og et hovedpanel. Disse komponenter skifter indhold, afhængig af hvad brugeren vælger. På denne måde mindske forvirringen hos brugeren. For at holde brugergrænsefladen nem og overskuelig, har vi også besluttet, at minimere antallet af inputelementer i brugergrænsefladen, ved at fokusere på de to kernearbejdsopgaver for en biografekspedient (bestilling og afbestilling af forestilling). På denne måde opfyldes behovet om brugervenlighed. Dette reflekteres også i den måde biografekspedienten vælger sæder på. I stedet for at have en menu, hvor man kan vælge hvor mange sæder, en reservation skal bestå af, og så rykke rundt på dette antal sæder på én gang, klikker man blot direkte på sæderne, man ønsker at inkludere i reservationen. Denne mere simple løsning har dog den ulempe, at det fx ville kræve 26 klik at rykke 13 sæder en række tilbage, hvis en kunde ønsker dette.

Af tidsmæssige årsager har vi ikke kunne implementere en mulighed for at ændre en eksisterende reservation i brugergrænsefladen. Hvis kunden derfor ønsker at ændre sin reservation, er biografekspedienten nødsaget til at slette kundens reservation og lave en helt ny en, hvilket igen kan kræve mange klik.

En anden beslutning vi har taget ift. den interne struktur af hvordan brugergrænsefladen stilles op er, at *ActionListeners* inkluderes i *View*-klassen. Her kan der argumenteres for, at *ActionListeners* tilhører *Controller*-delen af *MVC*. Derfor burde disse implementeres i en anden klasse, der ikke har til formål at opstille selve brugergrænsefladen, men blot at lytte og videresende data fra modellen til *View*. Dette er vi første blevet klar over sent i udviklingsforløbet, hvorfor vi ikke har haft tid til at implementere denne struktur i programmet. Hvis dette skulle implementeres, kunne man adskille klassen *View* i to klasser: *View* og *ViewController*. *View* ville kun bestå af metoderne til at opstille de enkelte brugergrænsefladeelementer og *ViewController* implementere lytttere og styre flow af data ind i *View*. *View* kunne implementere *getter*-metoder til de relevante brugergrænsefladeelementer, således at *ViewController* kunne tildele lytttere til disse. På denne måde ville *View* og *Controller* blive bedre adskilt end hvordan det på nuværende tidspunkt er implementeret i vores system.

4 Brugervejledning

Programmet har to brugsflows: et til reservation af sæder til en forestilling og et til afbestilling af en reservation. Det første brugsflow vises i figur 1:



Figur 1: Vindue for bestilling af reservation til en forestilling.

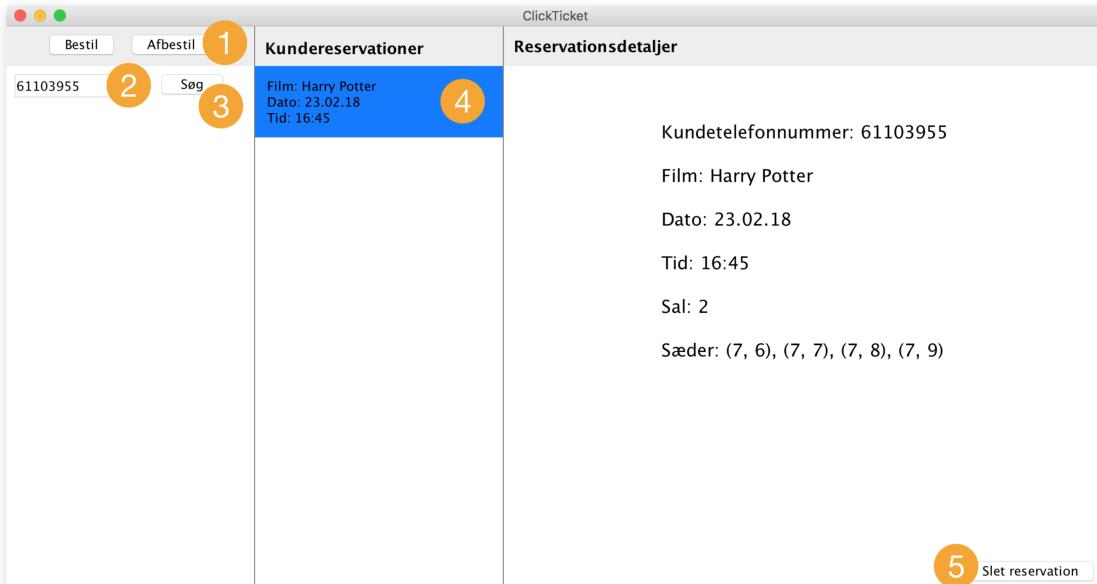
For at bestille en ny forestilling følges disse steps:

1. Ved programmets opstart vil en liste af film vises i det ydre sidepanel og man kan derfor umiddelbart springe til step 2. Hvis ikke klikkes på knappen *Bestil* og en liste af film vil vises i det ydre sidepanel.
2. Klik på den ønskede film. En liste af forestillinger til denne film vil nu vises i det indre sidepanel under *Forestillinger*.
3. Klik på den ønskede forestilling. En biografoversigt til den givne bestilling vil nu vises i vinduets hovedpanel.
4. Vælg position og antal sæder for bestillingen. Dette gøres ved at klikke på ledige sæder (markeret med grøn) i biografen. Sæder, der allerede er reserverede, er markeret med rød. Valgte sæder vil blive markeret med blå.
5. Indtast telefonnummer til registrering af reservation.
6. Klik *Reservér*. Herefter vil følgende besked vises med en oversigt af reservationens detaljer.



Figur 2: Bekræftelse på at reservationsbestilling er gennemført.

Det andet hovedbrugsflow er afbestilling. Dette brugsflow er illustreret i billede 3.



Figur 3: Vindue for afbestilling af en reservation.

For at afbestille en reservation følges disse steps:

1. Først klikkes på knappen *Afbestil*. Herefter vil der vises et inputfelt og knappen *Søg* i det ydre sidepanel.
2. Indtast det telefonnummer i inputfeltet, der blev registreret med den givne reservation, som skal afbestilles.
3. Klik *Søg*. Herefter vil en liste af reservationer tilknyttet det indtastede telefonnummer vises i indre sidepanel under *Kundereservationer*.
4. Klik på den reservation, der skal slettes. Herefter vil reservationsdetaljerne vises i vinduets hovedpanel.
5. Klik på *Slet reservation* for at slette reservationen. Herefter vises følgende besked, som bekräfter at reservationen er blevet slettet.



Figur 4: Bekræftigelse på at den valgte reservation er slettet.

4.1 Fejlmeddelelser i programmet

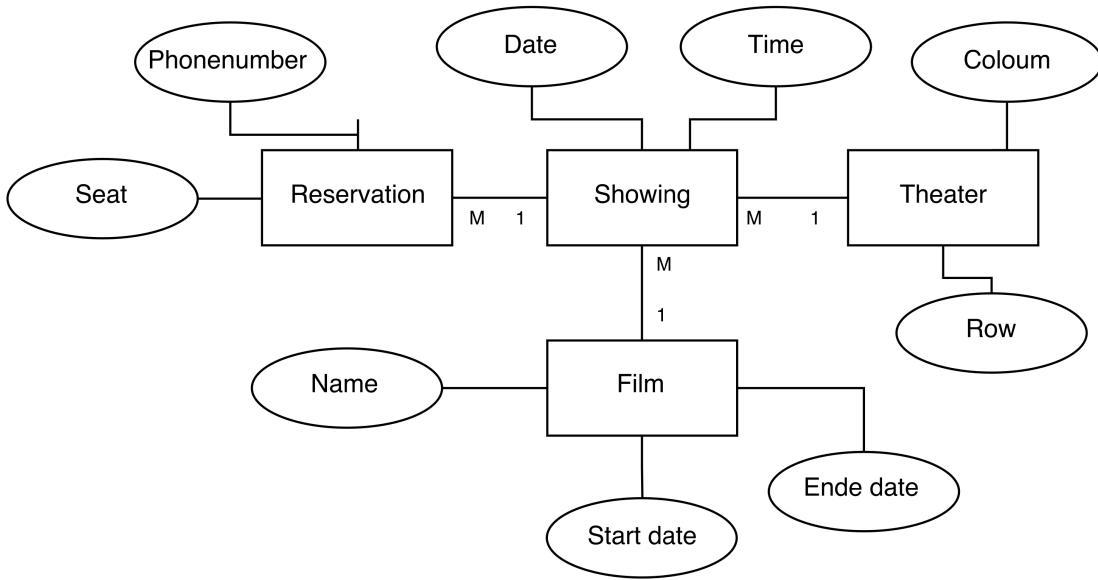
Programmet inkluderer en række fejlmeddelelser, som advarer brugeren hvis der sker fejl:

- *Ingen valgte sæder*: Hvis brugeren klikker på knappen *Reservér* uden at have valgt sæder i reservationen.
- *Ugyldigt telefonnummer*: Hvis brugeren indtaster et ugyldigt telefonnummer som fx “a1276537” eller “9893736233”. Benyttes to steder i brugergrænsefladen: ved bestilling af reservation og søgning efter telefonnummerrelaterede reservationer.
- *Ingen forbindelse*: Hvis programmet ikke kan få forbindelse til databasen og dermed ikke kan hente data såsom film og eksisterende reservationer.
- *Databasefejl*: Hvis der sker en fejl ved at indlæse og skrive data til databasen.
- *Programfejl*: En generisk fejlmeddelse for alle andre typer fejl i systemet.

5 Teknisk beskrivelse af programmet

5.1 Databasedesign

Al data i ClickTicket bliver gemt i en database. Databasen indeholder fire tabeller - *films* (film), *theaters* (sale), *showings* (forestillinger) og *reservations* (reservationer). Tabellerne og deres relationer ses i figur 5.



Figur 5: ER-diagram for databasen.

I figuren ses det fx, at en *showing* kun kan indeholde en enkelt *film*, men at en *film* kan tilhøre mange *showings*. På samme måde kan en *reservation* kun indeholde én *showing*, mens en *showing* kan tilhøre mange *reservations*. Hver af disse tabeller svarer til klasser i programmet. *Films*, *theaters* og *showings* er konstante og kan ikke opdateres gennem programmet, mens *reservations* er dynamisk og bliver opdateret hver gang programmet lukkes. På figur 6 ses et eksempel på en tabel i databasen.

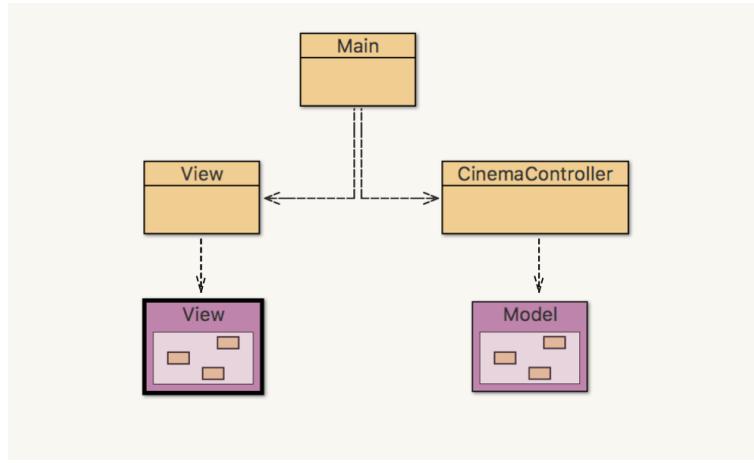
mysql> SELECT * FROM showings;				
showing_id	film_id_FK	theater_id_FK	date	time
1	1	2	01.12.17	12:00
2	1	4	05.12.17	18:00
3	1	2	21.12.17	12:00
4	1	4	04.01.18	18:00
5	1	2	24.01.18	12:00
6	2	1	10.12.17	17:30

Figur 6: Del af tabellen *showings*.

Tabellen *showings* indeholder fire attributter: *showing_id*, *film_id_FK*, *theater_id_FK*, *date* og *time*. Hver forestilling henviser til en bestemt film (*film_id_FK*) og sal (*theater_id_FK*), som den bliver vist i. Den indeholder også dato og tidspunkt. MySQL-koden, der er blevet brugt til at oprette databasen med eksempeldata findes i bilag 10.5.

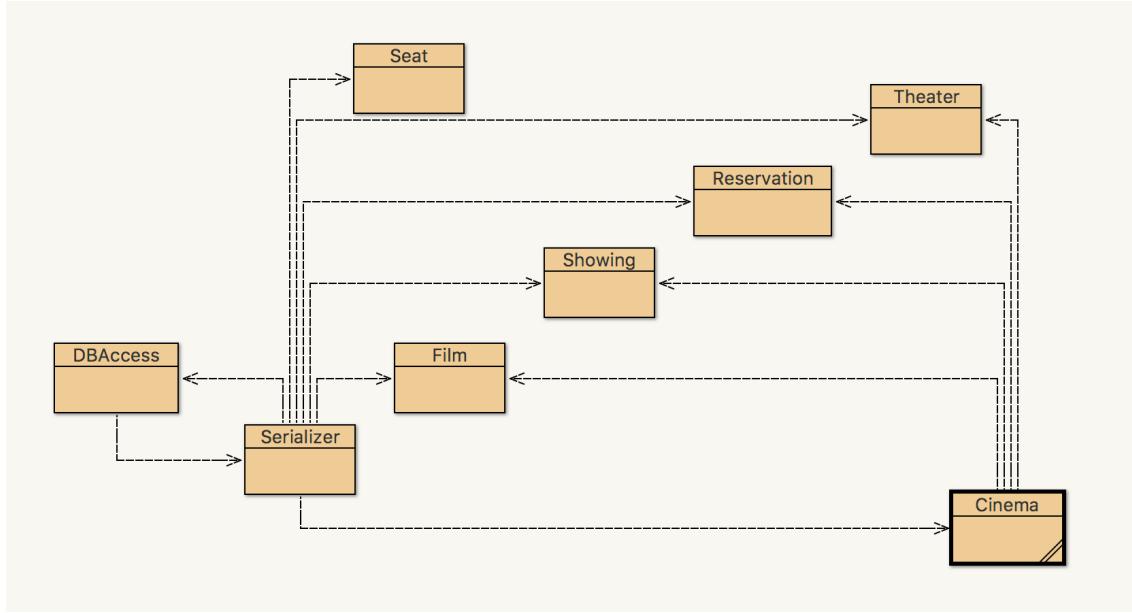
5.2 Programmets interne struktur

Nedenstående viser klassediagrammet for *Controller*-delen af programmets interne struktur:



Figur 7: Klassediagram for *Controller*-delen af programmet.

Klassen *Main* starter programmet op ved at oprette en *CinemaController* og et *View*. Overordnet set står klassen *CinemaController* for at kontrollere flowet af data fra modellen til *View*. Dette gøres ved at *CinemaController* indlæser det data, som bl.a. klassen *Cinema* holder på (*Model*), filtrerer det og sender det videre til *View*-klassen (*View*).



Figur 8: Klassediagram for *Model*-delen af programmet.

Cinema er ansvarlig for at holde på alt data fra databasen, oprettet som objekter. *DBAccess* står for at udtrække data fra databasen og gemme det som strenge. *Serializer* opretter dem derefter som objekter og sender dem videre til *Cinema*. De øvrige klasser bruges alene til at oprette objekter modelleret efter virkeligheden.

DBAccess indeholder fire *extract*-metoder, der bruges til at hente data fra databasen og om-danne hver række i en tabel til en streng, som sammensættes til en *ArrayList* af strenge. Der er tilhørende *getter*-metoder til hver *ArrayList*, som kaldes af *Serializer*. Herudover er der en

updateReservations-metode, der kaldes som det sidste inden programmet lukker. Den sletter alle rækker i tabellen *reservations* og opretter derefter alle eksisterende reservationsobjekter som rækker i samme tabel.

Serializer har fire *setUp*-metoder, der omdanner *ArrayLists* af strenge fra *DBAccess* til objekter. Derudover har den en *convertReservationsToString*-metode, der omdanner alle *Cinema*'s *reservations* til SQL-kommando-strenge, som kan kaldes af *DBAccess*' *updateReservations*. Metoden *convertReservationsToString* fungerer på den måde, at den tager en *ArrayList* af *Cinema*'s reservationer og løber igennem hver af dem i en *foreach*-løkke. Herefter løber den igennem alle sæder i hver reservation i en anden *foreach*-løkke. For hvert sæde laver den en streng af det nævnte sædes række- og sædenummer adskilt af et kolon. Hvis der er flere sæder i reservationen, tilføjer den yderligere en bindestreg for at adskille sæderne i reservationen (fx "3:4-3:5-3:6", se evt. bilag 10.4). Dette tjekker den i et *if-statement*. Herefter hopper den ud af det inderste *foreach*-løkke og opretter en streng med SQL-kommandoen, hvor den tilføjer de relevante data til en *reservation* i strengformat.

Cinema initialiseres af *CinemaController*. Dens konstruktør modtager alle nyoprettede objekter fra *Serializer* og opbevarer disse data i sine felter af *ArrayLists*. Derudover har *Cinema* en metode til at tilføje en ny *reservation* til sin *ArrayList* af *reservations*, samt en metode til at slette en *reservation* fra samme liste.

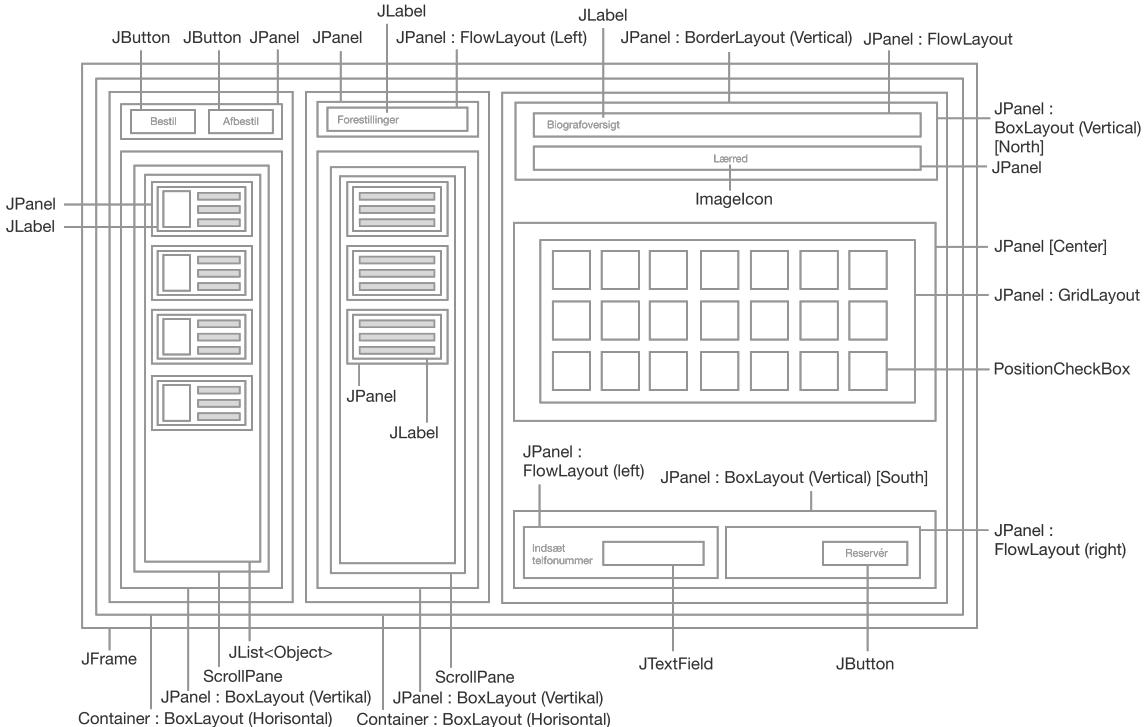
CinemaController er ansvarlig for al kontakt mellem *View* og *Model*. Den indeholder derfor metoder, der kan videregive informationer fra databasen og modellen, som *View* skal bruge. Heriblandt metoderne *listAllFilms* og *listAllPhoneNumbers*. Herudover har den tre metoder, der filterer i *Cinema*'s lister for et eller flere objekter, der har et bestemt objekt som felt. Fx *showingsRelatedToFilm*-metoden, der tager en film som parameter og derefter løber igennem alle *showings* og returnerer de *showings*, der viser den valgte film. På samme måde løber de to andre metoder igennem hhv. alle *reservations*, der er til en bestemt *showing* og alle *reservations*, der er lavet i et bestemt *phoneNumber*. Derudover har *CinemaController* følgende metoder *createReservation*, *deleteReservation* og *updateReservations*, der har ansvar for at tilføje og fjerne *reservations* fra sin egen kollektion af *reservation* objekter.

Enkelte steder, hvor det har givet mening, er der implementeret *exceptions* i programmet. Vi har udelukkende benyttet os af *checked exceptions* og bruger dem til at informere brugeren om, at der er sket en fejl via brugergrænsefladen. Dette kunne fx være, når computeren ikke har net og derfor ikke kan forbinde til databasen eller når brugeren ikke har valgt sæder.

I forhold til håndtering af databaseforbindelse benyttes også *SQLException*. Denne *exception* har vi valgt at kaste videre op i systemet, indtil *CinemaController*. Dette har vi gjort, da det er *CinemaController*, der har forbindelse til *View* og dermed kan sende fejlbeskeden til brugergrænsefalden, som kan underrette brugeren. Vi mente derfor, at det gav bedst mening, at det var denne klasse, der fangede denne *exception*.

5.3 Brugergrænsefladens struktur

Brugergrænsefladen er bygget med Javas *Swing* og *AWT* biblioteker. Brugergrænsefladen opnår sit udseende ved at indsætte flere såkaldte *JPanels* ind i hinanden. En oversigt af hvordan brugergrænsefladen er opbygget af *JPanels* med dertilhørende *layout managers* (fx *FlowLayout* og *BorderLayout*) og andre brugerfladekomponenter (fx *JList*, *JButton*) ses på figur 9.



Figur 9: En oversigt af de forskellige brugergrænsefladeelementer i ClickTicket.

Overordnet ses at vinduets *Container* har et horisontalt *BorderLayout*, så de tre hovedpaneler sættes ved siden af hinanden. Herefter er de to sidepaneler bygget af *JPanels* sat med et vertikalt *BorderLayout*. Listerne i brugergrænsefladen er bygget ved, at en *JList* indsættes som komponent i et *JScrollPane*. Her kommer en listes data fra de objekter, som listen skal vise til brugeren, fx er filmlisten bygget ved hjælp af et *array* af *film*-objekter. Når en listes (*JList*) data kommer fra et *array* af objekter, nødvendiggør det, at man definerer hvordan den enkelte celles udseende skal se ud med en såkaldt *ListCellRenderer*. Her laves en klasse som implementerer *ListCellRenderer* interfacet. I denne klasse implementeres en metode, der så fortæller listen hvordan objekterne skal repræsenteres visuelt. I forbindelse med bl.a. listen af film, blev der også benyttet *html*-kode til at formattere teksten i de enkelte celler til at lave linjeskift.

Det mest komplicerede komponent i brugergrænsefladen er det gitter, som repræsenterer sæderne i en given sal. Dette komponent består af et *JPanel* med *GridLayout* bestående af et antal *PositionCheckboxes*, der svarer til det antal sæder, der findes i salen. En *PositionCheckBox* nedarver fra en *JCheckBox* med den forskel at en *PositionCheckBox* indeholder felter, således at dens position i det givne *GridLayout* kan bestemmes. Således kan koordinaterne for de valgte sæder til en given reservation indsamles når brugeren klikker på *Reservér* knappen. Dette gøres ved brug af en *for*-løkke som gennemløber alle afkrydsede *PositionCheckboxes* gemt i en *ArrayList*.

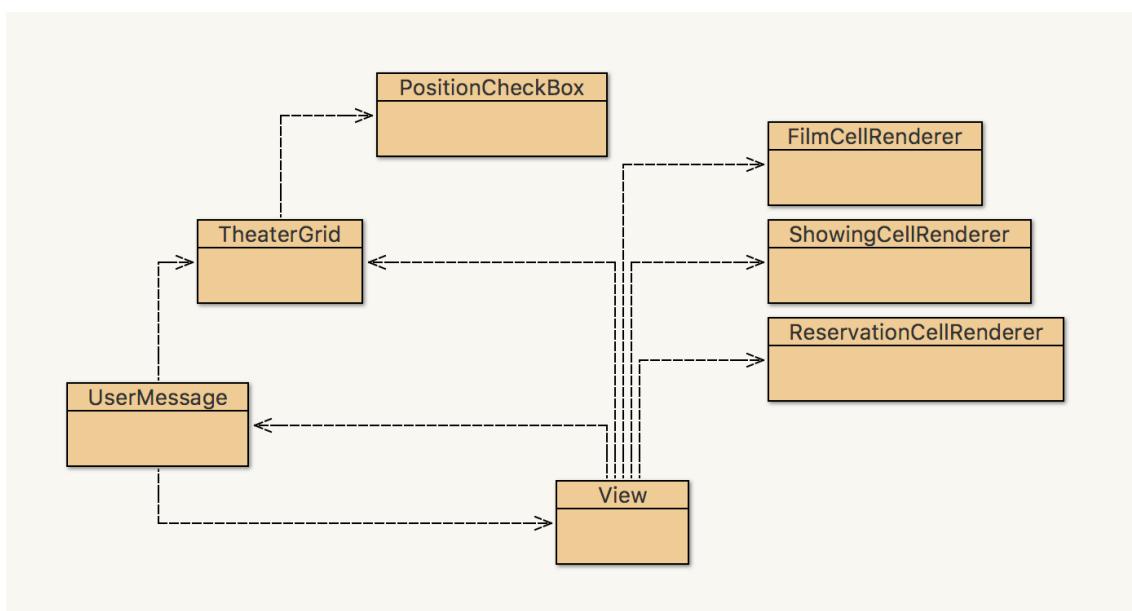
En af de store udfordringer i dette projekt har været at oversætte allerede bookede sæder fra en *ArrayList* af reservationer til deaktiverede *PositionCheckboxes* i *GridLayoutet*, således at brugeren ikke kan vælge et allerede booket sæde. Både *Seat* og *PositionCheckBox* objekter holder på et *r*-koordinat (*row*) og et *c*-koordinat (*column*). Derudover holdes *PositionCheckboxes* i en *ArrayList*, hvor det er et index, der identificerer den enkelte *PositionCheckBox*. Hvordan går man fra et sæt af 2D-koordinater til et endimensionelt index? Svaret var en simpel ligning, der udregner et index *k* ud fra et sædes *r*-koordinat, et *c*-koordinat og salens bredde *C*.

$$k = r \cdot C + c$$

Så hvis fx et sæde har koordinatet (2, 5) i en biografsal med *C* = 4, vil dens tilsvarende index *k* i *ArrayListen* af *PositionCheckboxes* være:

$$k = 13 = 2 \cdot 4 + 5$$

Hver *PositionCheckBox* kan dermed sammenlignes med hvert sæde (*Seat* objekt) og når der findes et match deaktiveres den givne *PositionCheckBox*. Metoden findes i klassen *TheaterGrid* og dens navn er *disableReservedSeatPositionCheckboxes*.



Figur 10: Klassediagram for brugergrænsefladedelen af programmet.

I figur 10 vises klassediagrammet for brugergrænseflade-delen (*View*) af programmet. Her ses det bl.a., at det er klassen *View*, som har en reference til *TheaterGrid*, hvilket er fordi hovedformålet med *View* er at udskifte brugerfladekomponenter og opdatere disse brugerfladekomponenter med data fra modellen, som brugeren klikker gennem brugergrænsefladen. For at kunne skifte indholdet ind og ud af brugergrænsefladen benytter *View* to slags *Listeners*: *ListSelectionListeners*, der lytter efter om brugeren vælger en ny celle i en af listerne og *ActionListener*, som lytter efter om brugeren klikker på en knap. *ActionListeners* er skrevet med den simple syntaksform for *lambda* udtryk (Barnes Költing, 2016, s. 445). De er også benyttet, når brugeren klikker på knapperne

Reservér og *Slet* reservation.

View-klassen har også det ansvar at validere brugerens input til brugergrænsefladen. Her sikrer *View*, at brugeren ikke glemmer at vælge sæder til en reservation før reservationen registreres, og at brugerens indtastede telefonnummer kun kan være et 8-cifret tal. Hvis brugeren ikke giver et korrekt input, gives der besked herom ved brug af en *JOptionPane*. Både fejlbeskeder og successbeskeder findes i klassen *UserMessage*.

5.4 Dokumentation af program

For at fremtidssikre programmet har vi skrevet Java Doc kommentarer til vigtige metoder og klasser. De forklarer hvad den enkeltes klasses overordnede ansvar og funktion er, og hvad metoderne gør. Se evt. bilag 10.8.

6 Afprøvning

For at sikre, at programmet virkede som det skulle, har vi gennem implementeringsprocessen udført en række tests.

I den tidlige brugertest med biografekspedienten, havde vi fokus på at høre om hendes typiske arbejdsopgaver. Vi havde tegnet vinduerne i brugergrænsefladen på forskellige stykker papir, som vi havde klippet ud, for på den måde at kunne vise det dynamiske flow i programmet. Hun fortalte, at det vigtigste er, at kunne bestille og afbestille billetter, fordi en billet-aændring kan ske ved først at afbestille og derefter bestille nye billetter. Hun kunne godt lide idéen om at klikke sæder fra og til, men synes, at det kunne være brugbart med en tæller i siden, som viser hvor mange sæder man har klikket på. Overordnet var hun glad for det meget simple og overskuelige design. Efter at havde snakket med hende, lavede vi små-justeringer på papirprototypen, for så vidt muligt at opfylde hendes ønsker. På denne måde blev vores design valideret, og dermed har vi kunne tage valg om afgrænsninger af systemet ift. hvad hun oplyste om sine typiske arbejdsopgaver.

Under implementeringsfasen har vi gjort brug af *System.out.print*-metoden og lavet unit test, for at teste de enkelte metoder i programmet (se bilag 10.7 med unit tests). Ved at bruge unit tests kunne vi sikre os, at hvert enkel del af programmet virker, inden man sætter det sammen med andre dele. Vi har også gjort brug af IntelliJ's Debugger, når testene ikke er gået som forventet, så vi har kunne finde roden til problemet og rette fejlene. Vi har benyttet unit tests i metoder, der henter data fra databasen og opretter data som objekter i programmet. I *View*-delen af programmet har vi dog kun lavet en unit test for at teste *validatePhoneNumber*-metoden. Resten af brugergrænsefladen har vi testet vha. Debuggeren og ved at køre programmet.

Til sidst har vi udført endnu en brugertest med samme biografmedarbejder, som vi foretog den indledende brugertest med. Til denne udarbejdede vi en vejledning (bilag 10.6), hvor vi opsatte arbejdsscenerier for brugeren. Her bad vi hende om at italesætte de ting, hun gjorde og tænkte, mens hun brugte programmet. Fordi vi udførte anden brugertest sent i projektet, var vi klar over, at vi ikke ville have tid til at ændre i programmet efter at have snakket med hende. Det var dog aligevel vigtigt at få testet brugertilfredsheden for det endelige program. Til denne brugertest læste

vi scenarierne op et af gangen, mens hun løste den tilsvarende opgave beskrevet i scenariet. Hun gennemførte alle scenarier uden problemer og havde ikke de store forslag til ændringer. Generelt syntes hun, at programmet var “*super fedt, nemt at finde ud af og smart*”. Dog syntes hun, det var svært at finde ud af hvilke film, der gik på en specifik dag.

I starten af projektet blev der stillet en række brugsmæssige og systemmæssige krav til programmet. Ift. de brugsmæssige krav, har vi opfyldt langt de fleste. Manglerne i programmet i forhold til arbejdsopgaverne er:

- At biografen ikke viser 3D film (krav nr. 2 og 13). Dog har dette ingen indflydelse på, hvordan programmet skulle programmeres.
- At man ikke kan bestille en billet til om et år (krav nr. 11). Vores film i databasen bliver kun vist t.o.m. marts 2018.
- At ekspedienten ikke har en tæller, så han/hun kan se, hvor mange billetter, der er blevet valgt (krav nr. 15).

Se tabel 1 for arbejdsopgaverne. Brugervenligheden har vi lagt stor vægt på, da det også var en af de ting, vi konkluderede var vigtigt ud fra den indelede brugertest. Ud fra vores sidste vellykkede brugertest, vurderer vi, at vi opnået det ønskede niveau af brugervenlighed i programmet.

I forhold til de systemmæssige krav, så lever programmet op til næsten dem alle. Som diskuteret under afsnit 3.3, kunne vi bedre adskille *View* og *Controller* fra hinanden ud fra den eksisterende *View*-klasse og derved gøre programmet mere modulært (se krav 4 i tabel 2) og opnå lavere kobling. Derudover har vi udført flere tests både på delkomponenter af programmet, og på programnets flow, hvorfor vi vurderer, at programmet højst sandsynligt fungerer som forventet. Der findes dog unit tests som mangler at blive implementeret, fx i forbindelse med om *TheaterGrid* altid returnerer korrekt 2D-array af sæder. Derudover har vi mest været fokuseret på positive unit test og vi kunne med fordel have implementeret flere negative unit tests som ved test af *validatePhoneNumber*-metoden (se test i bilag 10.7.4).

7 Refleksion over arbejdsprocessen

Det første vi gjorde som gruppe, da projektperioden begyndte, var at mødes for at forventningsafstemme. På dette møde udarbejdede vi en gruppekontrakt, hvor vi blev enige om retningslinjer for gruppearbejdet. Derudover lavede vi en fælles online kalender, hvor vi med det samme tilføjede alle mødetidspunkter for de næste tre uger. Ved at have både en tidsmæssig og samarbejdsmæssig ramme for projektarbejde har vi lettere kunne forhindre gruppekonflikter og samtidig vide, at vi gik mod et fælles mål.

Inden projektperioden begyndte, blev vi introduceret til Git. Vi har benyttet os af Git gennem hele projektforløbet, da det gjorde det muligt at dele kode indbyrdes uden mange filoverførsler og sammenkoblingsproblemer. Dette til trods for at det tog tid fra at arbejde på projektet. I sidste ende har Git dog muliggjort, at vi kunne bruge mere tid på den faktiske kodning af programmet

fremfor at sammensætte koden fra de enkelte gruppemedlemmer.

Vi har arbejdet efter den såkaldte vandfaldsmodel, hvor vi har taget faserne (analyse, design og implementering) i rækkefølge og ikke vendt tilbage til tidligere faser, efter vi var gået videre til næste. Da vi ikke har haft nok tid, har vi ikke kunne have en iterativ tilgang til projektet. En cyklus varer nemlig oftest 3-uger afhængig af systemudviklingsmetoden (se fx Scrum-metoden). Til gengæld brugte vi en del tid på både vores analyse- og designfase. Vi fik snakket mange elementer af det samlede program igennem inden, vi begyndte at kode, så vi havde en fælles forståelse for, hvordan programmet skulle implementeres. Dette har gjort, at implementeringsprocessen gik mere glat, da vi på forhånd var enige om mange beslutninger.

I implementeringsfasen brugte vi *interfaces* som et værktøj til at samarbejde om kode. Vi oprettede to *interfaces*; et mellem *Model* og *Controller*, og et andet mellem *View* og *Controller*. På den måde kunne vi bruge metoder fra hinandens dele, uden at de nødvendigvis var færdigimplementeret. En forbedring ift. samarbejdet i dette projekt kunne have været at benytte *interfaces* til flere klasser (fx alle klasserne i *Model*).

I løbet af projektarbejdet har vi arbejdet meget opdelt, hvor hver enkelt person har arbejdet på enten *Model*, *View* eller *Controller*. Dette har gjort, at vi alle har kunne fordybe os meget i vores enkelte arbejdsopgaver. Til gengæld har det også gjort, at alle ikke har lige dyb indsigt i samtlige dele af programmet. Vi kobledes først alle dele sammen i den sidste uge af projektet som et “Big Bang”. Dette kunne man have valgt at gøre løbende, så man ikke risikerede, at intet virkede ved sammenkoblingen til sidst. Det lykkedes dog i dette projekt, uden de store problemer. I et fremtidigt projekt vil vi nok gøre det anderledes og koble færdiggjorte dele af programmet sammen undervejs.

Generelt set synes vi alle tre, at vi har fået enormt meget ud af dette projekt og har lært en helt masse nyt samtidig med, at vi er blevet mere sikre i det, vi har lært til undervisningen. Vi vurderer alle, at vi i høj grad lever op til læringsmålene. Vi kunne dog have været bedre til at teste programmet under implementeringsfasen af projektet.

8 Konklusion

I dette projekt har vi gennemført en analysefase af et biografdomæne, hvor vi fandt en række krav til det reserveringssystem vi skulle bygge. Resultaterne fra denne domæneanalyse blev derudover brugt til at designe den objektorienterede måde at lagre data på i programmet (*Model*). Herefter blev programmet, brugergrænsefladen og den dertilhørende relationelle database implementeret i Java og MySQL. Vi har forsøgt løbende at teste systemet ved at udføre flere unit test på vigtige dele af programmet samt brugertest, som viser at programmet fungerer hensigtsmæssigt. Alt i alt har projektsammenarbejdet i gruppen ført til et velimplementeret biograf reserveringssystem kaldet *ClickTicket*.

9 Litteratur

Brabrand, C. (2017). *Grundlæggende Programmering: Design Patterns* [Slides fra forelæsning].

Tilgået d. 11. december 2017 på

<https://learnit.itu.dk/mod/resource/view.php?id=78374>.

Barnes, D. J. Kölking, M. (2016). *Objects First with Java: A Practical Introduction using BlueJ*. 6. udgave, Pearson.

Moore, Scott A. 2000. *Entity-Relationship Modelling*.

Fundet d. 3.12.17 på <http://www.inf.unibz.it/franconi/teaching/2000/ct481/er-modelling/Appendix>

10 Bilag

10.1 Substantiv-verbmetoden

The cinema booking system should help a cinema employee create and cancel seat-bookings for showings. A showing is a combination of movie, date, time and theater. A movie is shown for a limited period of time. A theater is a hall with rows of seats and a screen. Different theaters have different numbers of seats. A seat is identified by a row and a number in that row. A customer is a person, who makes a reservation at the cinema or via telephone. A phone number is used to identify customers.

Figur 11: Substantiver er grønne. Verber er blå

Substantiver	Verber
Cinema Booking System	Help
Cinema Employee	Create (reservation)
Seat Booking	— Make
— Reservation	Cancel Reservation
Showing	— Is
Movie	Show (showing)
Date	Have (row, seat)
Time	Identify (customer, phone number)
— Period of time	— Use
Theater	
— Hall	
— Screen	
Row	
Seat	
Number	
Customer	
— Person	
Phone Number	
Cinema	
Theater	
Telephone	
Reservation	

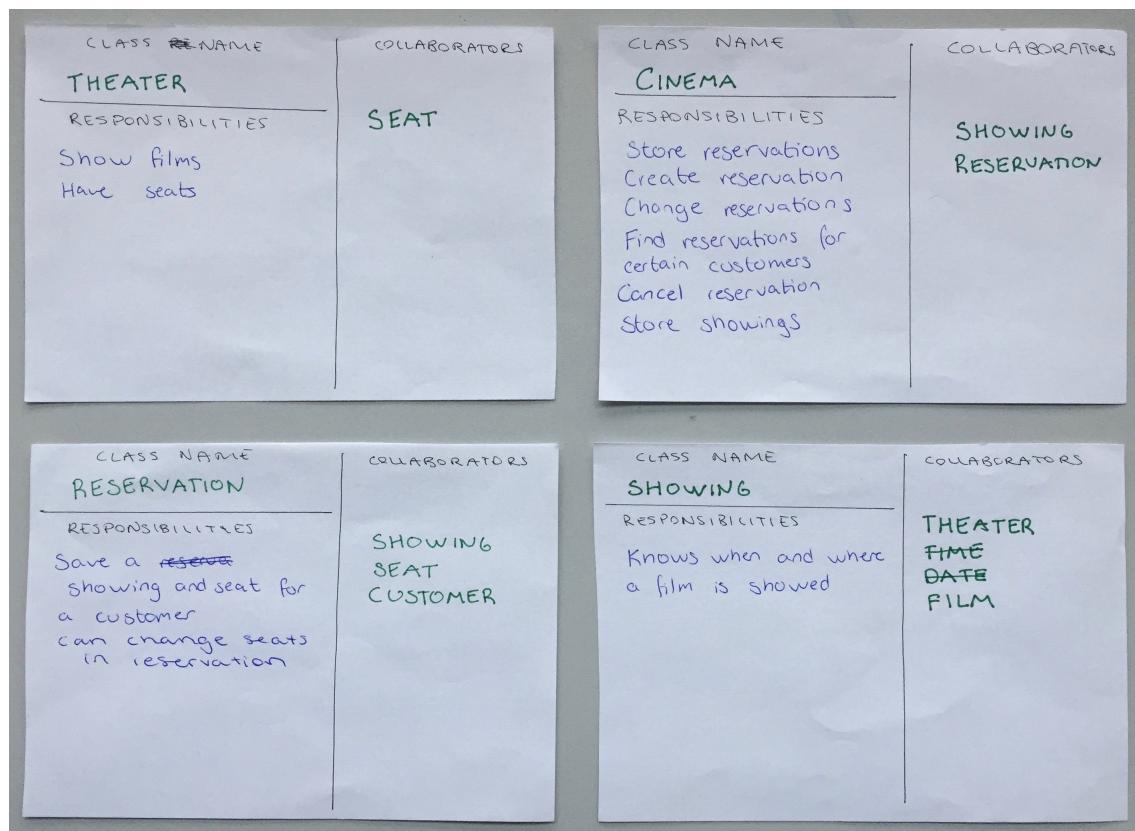
Tabel 4: Systemmæssige krav til programmet

10.2 Papirprototype af ClickTicket

10.2.1 Reservering af biografbillet

10.2.2 Afbestilling af biografbillet

10.3 CRC-kort



10.4 Reservationstabel fra databasen

```
[mysql> SELECT * FROM reservations;
+-----+-----+-----+
| reservation_id | phone_number_FK | seat |
+-----+-----+-----+
| 191 | 09876543 | 0:1-0:2 |
| 192 | 87654321 | 5:1-5:2-5:3 |
| 193 | 12345678 | 0:2-0:3 |
| 194 | 34567890 | 4:5-4:6 |
| 195 | 30324561 | 4:4-4:5-4:6-4:7-4:8 |
| 196 | 64789288 | 0:5-0:6-0:7 |
| 197 | 60220146 | 5:2-5:3-5:4 |
| 198 | 64789288 | 0:8 |
| 199 | 60621330 | 6:5-6:6-6:7 |
| 200 | 61103955 | 0:6-0:7-0:8 |
| 201 | 65834523 | 10:2-10:3-10:4-10:5 |
+-----+-----+-----+
11 rows in set (0.01 sec)
```

10.5 MySQL Commands

```
--films
CREATE TABLE films
(film_id INT PRIMARY KEY AUTO_INCREMENT,
```

```

name VARCHAR(50) NOT NULL,
start_date VARCHAR(20) NOT NULL,
end_date VARCHAR(20) NOT NULL);

INSERT INTO films (name, start_date, end_date)
VALUES ('Puss in Boots', '01.12.17', '01.02.18'),
('V for Vendetta', '10.12.17', '20.01.18'),
('GI Joe', '3.12.17', '03.01.18'),
('Watchmen', '20.12.17', '01.03.18'),
('Harry Potter', '04.12.17', '07.03.18'),
('Inception', '21.11.17', '21.02.18'),
('Batman', '24.12.17', '13.02.18'),
('Unforgiven', '01.01.18', '01.03.18'),
('Kick Ass', '25.12.17', '22.02.18');

--theaters
CREATE TABLE theaters
(theater_id PRIMARY KEY AUTO_INCREMENT,
row INT NOT NULL,
seat INT NOT NULL);

INSERT INTO theaters (row, seat)
VALUES (20, 8),
(15, 10),
(15, 15),
(8, 7);

--showings
CREATE TABLE showings
(showing_id INT PRIMARY KEY AUTO_INCREMENT,
film_id_FK INT NOT NULL,
theater_id_FK INT NOT NULL,
date VARCHAR(20) NOT NULL,
time VARCHAR(20) NOT NULL);

INSERT INTO showings (film_id, theater_id, date, time)
VALUES
-- Puss in Boots
(1, 2, '01.12.17', '12:00'),
(1, 4, '05.12.17', '18:00'),
(1, 2, '21.12.17', '12:00'),
(1, 4, '04.01.18', '18:00'),
(1, 2, '24.01.18', '12:00'),
-- V for Vendetta
(2, 1, '10.12.17', '17:30'),

```

```

(2, 3, '20.12.17', '21:30'),
(2, 1, '26.12.17', '17:30'),
(2, 3, '10.01.18', '21:00'),
(2, 3, '13.01.18', '19:30'),
-- GI Joe
(3, 2, '04.12.17', '20:15'),
(3, 2, '10.12.17', '17:30'),
(3, 2, '11.12.17', '13:30'),
(3, 2, '25.12.17', '19:45'),
(3, 2, '05.01.18', '20:00'),
-- Watchmen
(4, 3, '24.12.17', '21:30'),
(4, 1, '05.01.18', '19:15'),
(4, 3, '08.01.18', '14:30'),
(4, 1, '22.02.18', '20:15'),
(4, 3, '01.03.18', '17:45'),
-- Harry Potter
(5, 2, '06.12.17', '21:30'),
(5, 3, '23.12.17', '14:00'),
(5, 2, '26.01.18', '18:45'),
(5, 3, '27.01.18', '20:00'),
(5, 2, '23.02.18', '16:45'),
-- Inception
(6, 3, '01.12.17', '19:00'),
(6, 3, '14.12.17', '12:00'),
(6, 3, '20.01.18', '20:15'),
(6, 3, '02.02.18', '17:30'),
(6, 3, '20.02.18', '21:15'),
-- Batman
(7, 1, '27.12.17', '20:30'),
(7, 2, '03.01.18', '22:00'),
(7, 2, '11.01.18', '19:45'),
(7, 2, '18.01.18', '18:30'),
(7, 1, '19.01.18', '20:30'),
-- Unforgiven
(8, 4, '06.01.18', '14:00'),
(8, 4, '17.01.18', '21:15'),
(8, 4, '19.01.18', '20:30'),
(8, 1, '14.02.18', '19:45'),
(8, 1, '30.02.18', '18:00'),
-- Kick Ass
(9, 4, '26.12.17', '20:00'),
(9, 1, '30.12.17', '17:45'),
(9, 1, '16.01.18', '21:15'),
(9, 4, '05.02.18', '19:30'),

```

```

(9, 1, '17.02.18', '20:00'),

--reservation
CREATE TABLE reservations
(reservation_id INT PRIMARY KEY AUTO_INCREMENT,
phone_number_FK VARCHAR(20) NOT NULL,
seat VARCHAR(150) NOT NULL);

```

10.6 Brugertest 2

Brugertest

Når jeg viser dig programmet og du udfører de scenarier som jeg læser op, må du meget gerne italesætte hvad du gør, hvad du synes om det og hvad du leder efter eller savner.

Præmissen: Du arbejder i en lille biograf med kun en billetluge. Du er på arbejde og sidder alene i billetlugen.

Scenarie 1: Et par vil gerne Batman d. 11. januar, kl. 19.45. Telefonnummer: 54637899.

Scenarie 2: En familie på fem vil gerne se Harry Potter kl. 18.45 en dag. Telefonnummer: 30324561.

Scenarie 3: Tre venner vil gerne se en film d. 13. januar. Telefonnummer: 64789288.

Scenarie 4: En mor med sine to børn vil gerne se Puss in Boots. De er ligeglade med hvornår, bare det er en lille biograf. De vil gerne sidde ca. midt i. Telefonnummer: 60220146.

Scenarie 5: Parret, der skal se Batman, vil gerne afbestille deres billetter. Telefonnummer: 54637899.

Scenarie 6: De tre venner har fået en ny ven, der skal med i biffen d. 13. januar. Telefonnummer: 64789288.

Scenarie 7: En person spørger hvilke forestillinger, der går i marts.

10.7 Testcases - unit test

10.7.1 *CinemaController* unit tests

```

package Controller.ControllerTest;

import Controller.CinemaController;
import Model.*;
import org.junit.Test;
import java.util.ArrayList;
import static org.junit.Assert.*;

```

```

public class CinemaControllerTest {
    private Cinema cinema;

    @Test
    public void listAllFilms() throws Exception {
        CinemaController cinemaController = new CinemaController();
        Film f = new Film(2, "V for Vendetta", "10.12.17", "20.01.18");
        ArrayList<Film> a = cinemaController.listAllFilms();
        assertEquals(f.getId(), a.get(1).getId());
    }

    @Test
    public void showingsRelatedToFilm() throws Exception {
        DBAccess dbAccess = new DBAccess();
        Serializer serializer = new Serializer(dbAccess, cinema);

        ArrayList<Film> films = serializer.setUpFilms(dbAccess.getFilmStrings());
        ArrayList<Theater> theaters = serializer.setUpTheaters(dbAccess.getTheaterStrings());
        ArrayList<Showing> showings = serializer.setUpShowings(dbAccess.getShowingStrings());
        ArrayList<Reservation> reservations =
            serializer.setUpReservations(dbAccess.getReservationStrings());
        cinema = new Cinema(films, theaters, showings, reservations);

        Film f = new Film(2, "V for Vendetta", "10.12.17", "20.01.18");
        Theater t = new Theater(3, 15, 15);
        Showing s = new Showing(7, f, t, "20.12.17", "21:30");

        assertEquals(s.getId(), cinema.getShowings().get(6).getId());
    }

    @Test
    public void reservationsRelatedToShowing() throws Exception {
        DBAccess dbAccess = new DBAccess();
        Serializer serializer = new Serializer(dbAccess, cinema);
        CinemaController cinemaController = new CinemaController();

        ArrayList<Film> films = serializer.setUpFilms(dbAccess.getFilmStrings());
        ArrayList<Theater> theaters = serializer.setUpTheaters(dbAccess.getTheaterStrings());
        ArrayList<Showing> showings = serializer.setUpShowings(dbAccess.getShowingStrings());
        ArrayList<Reservation> reservations =
            serializer.setUpReservations(dbAccess.getReservationStrings());
        cinema = new Cinema(films, theaters, showings, reservations);
    }
}

```

```

        ArrayList<Seat> as = new ArrayList<>();
        as.add(new Seat(0, 2));
        as.add(new Seat(0,3));
        Film f = new Film(4, "Watchmen", "20.12.17","01.03.18");
        Theater t = new Theater(1, 8, 20);
        Showing s = new Showing(19, f, t, "22.02.18", "20:15");
        Reservation r = new Reservation("12345678", as, s);
        ArrayList<Reservation> ar = new ArrayList<>();
        ar.add(r);

        assertEquals(ar.size(), cinemaController.reservationsRelatedToShowing(s).size());
    }
}

```

10.7.2 DBAccess unit test

```

package Model.ModelTest;

import Model.DBAccess;
import org.junit.Test;
import java.sql.SQLException;
import java.util.ArrayList;

import static org.junit.Assert.*;

public class DBAccessTest {

    @Test
    public void getFilmStrings() throws SQLException {
        DBAccess DBAccess = new DBAccess() ;
        String s = "2,V for Vendetta,10.12.17,20.01.18";
        ArrayList<String> a = DBAccess.getFilmStrings();
        assertEquals(s, a.get(1));
    }

    @Test
    public void getTheaterStrings() throws Exception {
        DBAccess DBAccess = new DBAccess() ;
        String s = "3,15,15";
        ArrayList<String> a = DBAccess.getTheaterStrings();
        assertEquals(s, a.get(2));
    }

    @Test
    public void getShowingStrings() throws Exception {

```

```

        DBAccess DBAccess = new DBAccess() ;
        String s = "33,7,2,11.01.18,19:45";
        ArrayList<String> a = DBAccess.getShowingStrings();
        assertEquals(s, a.get(32));
    }

    /**
     * Reservation ID number changes each time the program is opened
     * which is why ID is not included in this test.
     * @throws Exception
     */
    @Test
    public void getReservationStrings() throws Exception {
        DBAccess DBAccess = new DBAccess() ;
        String s = "12345678,0:2-0:3,19";
        ArrayList<String> a = DBAccess.getReservationStrings();
        assertEquals(s, a.get(2).substring(4));
    }
}

```

10.7.3 Serializer unit test

```

package Model.ModelTest;

import Model.*;
import org.junit.Test;
import java.sql.SQLException;
import java.util.ArrayList;

import static org.junit.Assert.*;

public class SerializerTest {

    private Cinema cinema;

    @Test
    public void setUpFilms() throws SQLException {
        DBAccess dbAccess = new DBAccess();
        Serializer serializer = new Serializer(dbAccess, cinema);

        ArrayList<Film> films = serializer.setUpFilms(dbAccess.getFilmStrings());
        ArrayList<Theater> theaters = serializer.setUpTheaters(dbAccess.getTheaterStrings());
        ArrayList<Showing> showings = serializer.setUpShowings(dbAccess.getShowingStrings());
        ArrayList<Reservation> reservations =
            serializer.setUpReservations(dbAccess.getReservationStrings());
    }
}

```

```

cinema = new Cinema(films, theaters, showings, reservations);

Film f1 = new Film(2, "V for Vendetta", "10.12.17", "20.01.18");
assertEquals(f1.getId(), films.get(1).getId());

Film f2 = new Film(9, "Kick Ass", "25.12.17", "22.02.18");
assertEquals(f2.getStartDate(), films.get(8).getStartDate());

Film f3 = new Film(5, "Harry Potter", "04.12.17", "07.03.18");
assertEquals(f3.getName(), films.get(4).getName());
}

@Test
public void setUpTheaters() throws SQLException {
    DBAccess dbAccess = new DBAccess();
    Serializer serializer = new Serializer(dbAccess, cinema);

    ArrayList<Film> films = serializer.setUpFilms(dbAccess.getFilmStrings());
    ArrayList<Theater> theaters = serializer.setUpTheaters(dbAccess.getTheaterStrings());
    ArrayList<Showing> showings = serializer.setUpShowings(dbAccess.getShowingStrings());
    ArrayList<Reservation> reservations =
        serializer.setUpReservations(dbAccess.getReservationStrings());
    cinema = new Cinema(films, theaters, showings, reservations);

    Theater t = new Theater(3, 15, 15);
    assertEquals(t.getId(), theaters.get(2).getId());
}

@Test
public void setUpShowings() throws SQLException {
    DBAccess dbAccess = new DBAccess();
    Serializer serializer = new Serializer(dbAccess, cinema);

    ArrayList<Film> films = serializer.setUpFilms(dbAccess.getFilmStrings());
    ArrayList<Theater> theaters = serializer.setUpTheaters(dbAccess.getTheaterStrings());
    ArrayList<Showing> showings = serializer.setUpShowings(dbAccess.getShowingStrings());
    ArrayList<Reservation> reservations =
        serializer.setUpReservations(dbAccess.getReservationStrings());
    cinema = new Cinema(films, theaters, showings, reservations);

    Film f = new Film(2, "V for Vendetta", "10.12.17", "20.01.18");
    Theater t = new Theater(3, 15, 15);
    Showing s = new Showing(9, f, t, "10.01.18", "21:00");
    assertEquals(s.getId(), showings.get(8).getId());
}

```

```

@Test
public void setUpReservations() throws Exception {
    DBAccess dbAccess = new DBAccess();
    Serializer serializer = new Serializer(dbAccess, cinema);

    ArrayList<Film> films = serializer.setUpFilms(dbAccess.getFilmStrings());
    ArrayList<Theater> theaters = serializer.setUpTheaters(dbAccess.getTheaterStrings());
    ArrayList<Showing> showings = serializer.setUpShowings(dbAccess.getShowingStrings());
    ArrayList<Reservation> reservations =
        serializer.setUpReservations(dbAccess.getReservationStrings());
    cinema = new Cinema(films, theaters, showings, reservations);

    ArrayList<Seat> as = new ArrayList<>();
    as.add(new Seat(0, 2));
    as.add(new Seat(0,3));
    Film f = new Film(4, "Watchmen", "20.12.17", "01.03.18");
    Theater t = new Theater(1, 8, 20);
    Showing s = new Showing(19, f, t, "22.02.18", "20:15");
    Reservation r = new Reservation("12345678", as, s);

    assertEquals(r.getPhoneNumber(), reservations.get(2).getPhoneNumber());
}

@Test
public void convertReservationsToStrings() throws Exception {
    DBAccess dbAccess = new DBAccess();
    Serializer serializer = new Serializer(dbAccess, cinema);

    ArrayList<Film> films = serializer.setUpFilms(dbAccess.getFilmStrings());
    ArrayList<Theater> theaters = serializer.setUpTheaters(dbAccess.getTheaterStrings());
    ArrayList<Showing> showings = serializer.setUpShowings(dbAccess.getShowingStrings());
    ArrayList<Reservation> reservations =
        serializer.setUpReservations(dbAccess.getReservationStrings());
    cinema = new Cinema(films, theaters, showings, reservations);

    assertEquals("INSERT INTO reservations (phone_number_FK, seat, showing_id_FK) VALUES
                  ('12345678', '0:2-0:3', 19);",
                  serializer.convertReservationsToStrings(reservations).get(2));
}
}

```

10.7.4 View unit test

```
package View.ViewTest;

import Controller.CinemaController;
import Exceptions.InvalidPhoneNumberException;
import View.View;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

public class ViewTest {
    @Rule
    public final ExpectedException exception = ExpectedException.none();

    @Test
    public void negativeValidatesPhoneNumber() throws Exceptions.InvalidPhoneNumberException {
        CinemaController cinemaController = new CinemaController();
        View view = new View(cinemaController);

        exception.expect(InvalidPhoneNumberException.class);
        view.validatePhoneNumber("a1234567");
    }

    @Test
    public void positiveValidatesPhoneNumber() throws Exceptions.InvalidPhoneNumberException {
        CinemaController cinemaController = new CinemaController();
        View view = new View(cinemaController);

        view.validatePhoneNumber("12345678");
    }
}
```

10.8 Java Doc

For at se den generede JavaDoc fra dokumentationen af programmet, åben den medfølgte zip-fil og åben filen *index.html*. Herefter vil et browservindue med JavaDoc af programmet åbnes.