MOTORHOME NORDIC

Dat 19C: 2. Semester: Eksamensprojekt

DAT19C Ali Raza Akhtar
01/02/1998
DAT19C Edmond Bidjeck
29/07/1988
DAT19C Omar Atik
31/08/1998
DAT19C Umit Arabaci
10/10/1994

4. JUNI 2020

HSM

Indhold

2 Problemformulering

Motorhome Nordic har på det seneste fået en øget bestilling af udlejninger af autocampere og har derfor brug for at udvikle en hjemmeside til håndteringer af bestillinger. Hvordan kan et system bedst muligt udvikles, så det tilpasser sig Motorhome Nordic behov og gør deres arbejde mere simplificeret for virksomhedens medarbejdere?

Hvordan udvikler man en hjemmeside på baggrund af et design som er udarbejdet ud fra kundens behov?

- Hvilken udfordringer kan man støde ind på under udviklingen?
- Hvordan beskytter vi persondata bedst muligt?

3 Virksomhedsanalyse

3.1 Om Motorhome Nordic

Alle mennesker elsker at komme lidt ud, især når der er tale om ferie. I Danmark er vi glade for ferie, og oplever i den grad også flere turister, der også vil have lidt af Danmark.

Generelt når der bliver snakket om ferie, så er det første man tænker på en charterferie i det sydlige Spanien eller en flyrejse af en eller anden art, men sådan tænker vi ikke hos Nordic Motorhome.

Nordic Motorhome tilbyder en anden form for ferie, hvor hele din rejse igennem landet kan klassificeres som en ferie, vi tilbyder nemlig et mobilt sommerhus, et sommerhus på fire hjul.

Nordic Motorhome tilbyder forbrugeren en autocamper, som du kan rejse landet rundt i, hvilket ikke gør dig nødsaget til at blive på hotellet, men derimod slå lejr i nærheden af dine favoritplaceringer.

Nordic Motorhome tilbyder forskellige varianter af autocampere der varieres fra 2 til 6 senge. Det vil sige at man både kan komme afsted med dem man lyster.

Nedenfor er en specificeret liste med hvad Nordic Motorhome ellers tilbyder

- 32 Autocampere, fordelt i 8 forskellige typer
- 400 km om dagen
- Forsikring inkluderet i prisen
- Udvendig rengøring inkluderet i prisen
- Aflevering og afhentning af autocamperen
- Ekstra tilbehør såsom cykelholdere, sengelagen, børnesæde etc.

Motorhome Nordic giver dig derfor en mulighed som forbruger at få den perfekte kickstart til din ferie.

3.2 Feasibility study

Et feasibility study har til formål at bevise eller klargør sandsynligheden for, hvor stor succes der er i den pågældende projekt. Derudover klarlægger man hvilken muligheder der findes, så man kan udarbejde den mest profitable løsning. Med feasibility study går man helt i dybden, hvor man finder projektets svagheder og styrker. Det er netop her gruppemedlemmerne tager beslutning om at fortsætte eller stoppe projektet.

3.2.1 Områder feasibility study indeholder

- Tekniske forhold
- Økonomiske og finansielle forhold
- Organisatoriske og operationelle forhold
- Retslige og juridiske forhold
- Planmæssige forhold

3.2.2 Tekniske forhold

De hovedteknologier og værktøjer der bliver brugt i systemet, er følgende:

- Visual paradigm
- IntelliJ
- Java
- PowerPoint
- Word
- MySQL
- Workbench

Hver af teknologier og værktøjer er til rådighed 24 timer i døgnet, og kræver ingen internetforbindelse, hvilket gør det endnu mere produktivt.
Hvis projektet får brug for yderligere ressourcer, så er virksomheden klar til invester i projektet.

3.2.3 Økonomiske og finansielle forhold

- Programmet kommer til at køre på en gratis web server som bliver hostet hos Heroku og en gratis database som bliver hostet hos Gearhost
- Alt værktøj som bliver brugt til at udvikle systemet, er til rådighed, da dette projekt bliver støttet af KEA.
- Vi har programmører/designere til at udvikle systemet gratis, uden nogen omkostninger.
- Vi forventer at hjemmesiden kommer til at udleje en del biler, da der ikke er så mange konkurrenter. Derfor kan vi konkludere at hjemmesiden bliver brugt hyppigt og vil lave en god indtjening for virksomheden.
- Virksomheden kan også gøre brug af google annoncer, som vil være med til at flere person surfer på deres hjemmeside, hvilket vil fører til en bedre indtjening
- Ud fra de ovenstående punkter kan vi se at, projektet er velegnet til at blive gennemført ift. Økonomiske og finansielle rammer.
- System kan give en overskud til virksomheden, hvis de er gode til at præsenterer den på de sociale medier og bruge system hyppigt.

3.2.4 Organisatorisk og operationelle forhold

- Projektet vil påvirke organisationen positivt, hvis projektet bliver en succes, dette vil medføre til at flere kunder henvender sig organisationen.
- Det nye system vil være et godt redskab for ledelsen og medarbejderne i virksomheden, da det vil simplificere deres arbejde i en høj grad dvs. tidsbesparende.
- Brugerne er klar til at give feedback til forbedringer.
- Brugerne vil få lettere adgang til de diverse informationer.

3.2.5 Retslige og juridiske forhold

- Systemet bliver udviklet med stærke sikkerhedsovervejelser.
- Dermed sørger vi også for, at vi overholder persondataloven.
- Alle teknologier og værktøjer er lovligt at anvende.
- Medarbejderne har gode kontraktforhold og overenskomster.
- Ledelsen støtter projektet fuldt ud og ser frem til at have en hjemmeside klar.

3.2.6 Planmæssige forhold

- Det er ekstremt vigtigt at tidsplanen og tidsfristerne bliver overholdt for at projektet kan gennemføres og leveres til kunden.
- Vi forsøger at følge de iterationer som er udarbejdet i faseplanen.
- Derudover har vi en begrænset tid, hvilket betyder at hjemmesiden ikke kan udarbejdes helt færdig, men dog kun indeholde de primær use cases som er grundlaget for virksomheden.

3.2.7 Feasibility study konklusion

Vi kan ud fra denne feasibility study vurdere de forskellige faktorer som er med til at skabe dette projekt, og dermed konkludere at projektet er velegnet til at blive gennemført. Derudover har gruppen klargjort forskellige løsningsforslag til hvordan hjemmesiden bedst muligt kan udvikles og bruges. De tekniske forhold er godt dækket til og er klar til brug. Derudover er økonomien på plads og vi venter ikke at økonomien vil være et problem for projektets udførelse. Gruppens medlemmer vil forsøge at overholde den tidsmæssige planlægning som bliver udarbejde af gruppen, som tager udgangspunkt i de 4 faser som er i Unified Process. Gruppens medlemmer er opmærksomme på under hele forløbet at overholde de retslige og juridiske forhold.

3.3 Risikoanalyse

En risikoanalyse er et redskab der kan defineres som en omhyggelig og detaljeret vurdering af risici, der kan opstå under virksomhedernes funktioner, og kan anvendes til at træffe foranstaltninger for at minimere eller fuldstændig eliminere risici. De forholdsregler, der skal træffes mod potentielle trusler og de analyser der skal udføres, for at fortsætte arbejdet uden problemer, har en betydelig rolle effektive strategier kan tilbydes med omfattende undersøgelser mod de risici, der vil opstå for virksomheder.

En af de effektive strategier kaldes ”Proaktiv risikostrategi” som anvendes til at identificere alle de mulige risikomomenter der kan opstå. Dernæst skal der foretages en risikoanalyse, hvori man undersøger sandsynligheden for risikomomenternes optræden, samt en undersøgelse af hvilke bivirkninger der er forbundet hertil og en afvejning af hvad der vil ske hvis de potentielle risikomomenter forekommer. Der skal derfor laves en aktionsplan for hvorledes risikoen undgås, samt en aktionsplan for hvad der kan gøres hvis et uforudset risikomoment dog opstår.

De risici, der kan opstå i erhvervslivet, er f.eks. undersøgelser der foregår i virksomheder, gennemførte operationer, anvendte materialer, maskiner og udstyr i arbejdsområdet, miljøeffekter og medarbejdere. At sikre medarbejdernes sundhed og sikkerhed ud over arbejdsmiljøet, især på arbejdspladsen. Arbejdsgiveren er ansvarlig for identificering af risici med hensyn til arbejdsmiljø og sikkerhed.

3.3.1 Forklaring af de forskellige parametre

Sandsynligheden S = hvor sandsynligt det er, at risikomomentet indtræffer på en skala fra 1 til 5:

[1. Meget lavt], [2. Lavt] [3. Moderat] [4. Høj] [5. Meget Høj]

Konsekvens K = Hvor alvorlige konsekvenser det har for projektet, hvis risikomomentet indtræffer på en skala fra 1 til 10:

[1.Ubetydelige], [3.Tålelig], [7.Alvorlig], [10.Katastrofal]

Produkt/Risikotal =

[R = K * S]

3.3.1 Risikoanalyse tabel

Risikoanalyse tabel			
Risikomoment	**Sandsynlighed**	**Konsekvens**	**Produkt**
Sygdom	5	10	50
Stress	4	3	12
Internet nedbrud	2	7	14
Opgaver bliver ikke fuldført	1	7	7
Misforståelser af krav	1	10	10

3.3.2 Udvidet risikoanalyse tabel

Udvidet risiko tabel							
Risikomoment	**Sandsynlighed**	**Konsekvens**	**Produkt**	**Præventive tiltag**	**Ansvarlig**	**Løsningsforslag**	**Ansvarlig**
Sygdom	5	10	50	Sikre godt helbred - hold afstand, hygiejne, sund mad, søvn og hygge	Gruppen	Der vil være mulighed for at indgå særaftaler i forhold til sygdom	Gruppen
Stress	4	3	12	Pauser, minimer negative nyheder, hygge	Gruppen	Mere pause og en rolig tilgang til tingene	Gruppen
Internet nedbrud	2	7	14	Mere end en internetforbindelse, bruge tlf som hotspot (midlertidig løsning)	Gruppen	En tekniker bliver tilkaldt eller der bestilles en ny internetforbindelse. Evt. lån af vens forbindelse	Tekniker
Opgaver bliver ikke fuldført	1	7	7	Opfølgning på opgaver dagligt	Gruppen	Gruppen overvåger fuldførelse af opgaver og hjælper til hvis det skønnes nødvendigt	Gruppen

3.4 SWOT-Analyse

SWOT står for Strengths, Weaknesses, Opportunities og Threats. En SWOT-analyse definerer derfor de styrker og svagheder et firma har, både internt og eksternt. Når man har analyseret og gransket disse styrker og svagheder, så kan man derfra komme videre og udarbejde en arbejdsplan, der vil få firmaet til at vokse med de strategier der er blevet udarbejdet. I sammenhæng med dette projekt, kigger vi på de styrker og svagheder der ligger i autocamperudlejningsbranchen, men kan anvendes til alle former for produkter, projekter og firmaer.

INTERNE FAKTORER	
STYRKER (+)	**SVAGHEDER (-)**
• Tilgængelighed - Firmaet er tilgængeligt alle tider på året. • Kundevenligt - Firmaet tilbyder en masse ekstra services, såsom levering af autocamperen, der gør det nemmest for kunden. • Hurtigt voksende - Firmaet startede op i 2019, og er allerede i gang med at ansætte flere. • Placering - Firmaet er placeret lige uden for hovedstaden. • IT-udvikling - Firmaet ønsker sig, at de også skal have webbaserede muligheder til ansatte.	• Sæsonpræget - Klarer sig bedst bestemte tider på året • For få muligheder mht. kilomesterdistance. • Ny virksomhed - Kun eksisteret i 1 år • Ingen webbaserede muligheder for kunderne • Varesortiment - pt. har firmaet kun 32 autocampere til rådighed fordelt på 8 typer der varierer fra 2-6 personers. • Ingen webbaserede muligheder for ansatte

EKSTERNE FAKTORER	
MULIGHEDER (+)	**TRUSLER (-)**
• Faldende benzinpriser vil øge trafikken, og vil resultere i at flere vil bruge bilen som et transportmiddel til sin ferie.	• Stigende benzinpriser • Konkurrence på feriemarkedet - Andre alternativer på rejsemarkedet. • Flere stærke konkurrenter på autocampermarkedet.

3.4.1 Konklusion på SWOT

Som man kan se, er der muligheder for Motorhome Nordic. De kan blandt andet genere en hjemmeside for blandt andet ansatte men derudover også for kunder. Når kunder kan besøge butikken gennem internettet, så vil det også sige, at de kan ramme en større målgruppe, i stedet for at de kun kan ramme den målgruppe der besøger dem fysisk.

3.5 Stakeholder analyse

Stakeholders analyse er en analytisk teknik, som identificerer og analyserer hvor meget interesse og hvor meget indflydelse stakeholders har i et projekt. Stakeholders kan være det enkelte individ, en gruppe af individer og organisationer. De kan have interne eller eksterne relationer med virksomheden.

Miljøstyrelsen er eksempelvis en ekstern stakeholder med lav interesse, men den kan have en høj indflydelse over for virksomheden ved at regulere brug af campingvogne i miljøet. De kan for eksempel tilfredsstilles med virksomhedens brochure til kunderne om lovoverholdelse i miljøet.

Stakeholders	Interesse	Interaktion	Hvad er vigtigt for stakeholder?	Indflydelse
Ejeren	Høj	Skal engageres i projektet så meget muligt og skal tilfredsstilles	At Systemet er bygget til perfektion	Ejeren kan stoppe eller gå videre med projektet som han vil
Miljøstyrelsen	Lav	Skal tilfredsstilles	De har ingen interesse i virksomhed og dens System.	kan vedtage love som kan ramme virksomheden økonomisk for brug af campingvogn i miljøet
Ansatte	Høj	skal informeres om brug af systemet	De vil vide hvordan den nye System fungere	Ingen indflydelse
Kunder	Høj	Skal informeres om brugervenligheden af systemet og service	Brugervenlighed af og service.	Ingen indflydelse

4 Analyse- og designmodel

4.1 Introduktion til UP:

Vi vil i dette projekt gør brug af Unified Software Development Process for at gør vores arbejde mere effektivt, da det er en metode som er objektorienteret og iterativt.
Vores hovedemner er opdelt i 4 forskellige faser.

- Inception
- Elaboration
- Construction
- Transition

Disse 4 punkter deler man yderligere op i flere iterationer helt afhængigt af projektet.
Iterationerne i de forskellige faser skal ikke tage alt for lang tid, man skal helst lave en plan med en tidsbegrænsning for hver iteration. Dette kan ses i vores faseplan som er udarbejdet og vil blive lagt i rapporten. Gruppen har således valgt at dele rapporten op i disse 4 punkter, så hver af de opgaver som bliver løst, bliver kategoriseret inden for den fase den tilhører.

Iterativ er at man step by step udvikler en hver fase. Systemet vokser trinvis(inkrementelt) over tid, iteration for iteration. Resultatet af en iteration er ikke en eksperimentel prototype men derimod en delmængde af det endelige system. Hver iteration håndterer nye krav og udvider systemet.

4.2 Faseplan

Opgaver	Ansvar	Start	slut	Dage	Status
14-04-2020	Alle	12-05-2020	12-05-2020	1	Complete
Første Møde	Alle	5-12	5-12	0	**Complete**
Defineret kravene	Alle	12-maj	14-maj	2	**Complete**
Inception					
Iteration 1					
Risikoanalyse og plan	Umit	13-maj	14-maj	1	**Complete**
Feasibility study	Ali Raza	13-maj	14-maj	1	**Complete**
SWOT analyse	Omar	13-maj	14-maj	1	**Complete**
Stakeholders	Edmond	13-maj	14-maj	1	**Complete**
Phaseplan for Inception	Umit	13-maj	14-maj	1	**Complete**
Elaboration					
Iteration 1					
Supplementary specification	Ali & Edmond	13-maj	14-maj	1	**Complete**
Use case Breif	Omar	13-maj	14-maj	1	**Complete**
Use case casual	Omar	13-maj	14-maj	1	**Complete**
Use Case Fully Dressed	Omar	13-maj	14-maj	1	**Complete**
Use Cases	Omar	14-maj	16-maj	2	**Complete**
Iteration 2					
Domain Model	Ali & Umit	15-maj	16-maj	1	**Complete**
SSD 1 & SSD 2	Ali & Umit	15-maj	16-maj	1	**Complete**
SD	Ali & Umit	15-maj	16-maj	1	**Complete**
Conceptuel model	Ali & Umit	15-maj	19-maj	4	**Complete**
Grasp	Edmond	15-maj	16-maj	1	**Complete**
Class Diagram based on Conceptuel	Ali & Umit	15-maj	16-maj	1	**Complete**
ER-Diagram	Ali & Umit	17-maj	18-maj	1	**Complete**
Iteration 3					
Normalisering	Ali & Umit	18-maj	22-maj	4	**Complete**
Phaseplan for Elaboration	Umit	22-maj	23-maj	1	**Complete**

Planning for next phase	Alle	22-maj	22-maj	0	Complete
Construction					
Iteration 1					
GearHost & Database ready	Umit & Omar	20-maj	24-maj	-4	Complete
MySQL Script & All tables	Umit & Omar	22-maj	24-maj	2	Complete
InteliJ & Spring & Maven ON	Umit & Omar	23-maj	25-maj	2	Complete
UML description & diagrams and updates	Umit & Ali	24-maj	26-maj	2	Complete
Iteration 2					
Architecture Layers	Alle	27-maj	01-jun	5	Complete
HTML & CSS	Alle	27-maj	31-maj	4	Complete
Web Design	Alle	27-maj	01-jun	5	Complete
Collect the code for a finished program	Alle	01-jun	02-jun	1	Complete
Phaseplan for Construction	Umit	02-jun	02-jun	1	Complete
Planning for next phase	Alle	02-jun	02-jun	0	Complete
Transition					
Class Diagram based on Final Code	Umit	02-jun	02-jun	0	Complete
Ready-Made Report	Omar	02-jun	02-jun	0	Complete
Ready-Made phaseplan	Umit	02-jun	04-jun	2	Complete
jUnit Test	Ali	01-jun	04-jun	3	Complete
Comment on special conditions	Alle	01-jun	04-jun	3	Complete
Solutions that require special arguments	Ali	01-jun	04-jun	3	Complete
Finished Project	Alle	01-jun	04-jun	3	Complete

4.3 FURPS+

4.3.1 Funktionelle krav

• Create Booking: UC1
• End Booking: UC2
• Edit booking information: UC3
• Add accessories: UC4
• Cancel booking: UC5
• Create new customer: UC6
• See all customers: UC7
• Edit customer information: UC8
• Delete customer: UC9
• See list of all available cars: UC10
• See list of all cars rented: UC11
• See car details: UC12
• Create bill: UC13
• Edit car details: UC14
• Register new car: UC15

4.3.2 Non-funktionelle krav/ Supplementary specification:

Disse krav er implicit forventninger fra produktet.
Siden disse er forventede funktioner og ikke specifikt dokumenterede krave, så er de også kendt som Quality Attributes(kvalitet egenskaber).

4.3.2.1 Usability

- Der bliver lagt små spørgsmålstegn forskellige steder på hjemmesiden, som forklarer hvad de forskellige ting betyder f.eks. hvis der et tekstfelt, så vil der være et spørgsmålstegn til rådighed, som vil fortælle mere specifikt hvad feltet skal indeholde.
- For at gør det mere simpelt for kunden, så vil gruppen oprette en brugervejledning og eventuel en lydfil, hvor kunden vil få forklaret hvordan man bruger hjemmesiden korrekt og sikkert.
- Hjemmesiden skal være brugervenlig for kunderne.
- Systemet skal være nemt og brugervenligt for ansatte at arbejde med.
- Derudover vil vi forsøge at gør hjemmesiden brugervenligt til især de nye medarbejder som virksomheden vil ansætte i den nærmeste fremtid.

4.3.2.2 Reliability

- Systemet skal kunne være i stand til at beskytte følsomme data, herunder virksomhedens og kundernes data.
- De fleste steder i kodet vil der være en Try & catch funktion som vil gør system mere robust ved eventuel fejltastninger.

I løbet af hver time vil der automatisk blive lavet en sikkerhedskopi af databasen i tilfælde af at server skulle gå ned og ikke virke igen.

4.3.2.3 Availability

- Hjemmesiden skal være tilgængeligt 24 timer i døgnet.
- Kunderne skal kunne lave en reservation når som helst på hjemmesiden.
- Virksomhedens ansatte skal kunne arbejde på hjemmesiden når som helst.

4.3.2.4 Performance

- Hjemmesiden skal kunne uploades hurtigt og vær klar til brug uden unødig ventetid.
- Hjemmesiden kan håndtere flere brugere på én gang f.eks. hvis der er 2 medarbejdere i gang med at booke en bil til en kunde, så skal der opstå nogen former for problemer.
- Hvis serveren muligvis skulle gå ned, så skal en ny server op og køre indtil den første server er oppe og køre igen.

4.3.2.5 Supportability

- Test af programmet en gang om dagen for at sikre kvaliteten.
- Opdatere programmet løbende for at sikre en bedre arbejdsgang for medarbejderne.
- Hjemmesiden skal være egnet til alle type browsere der er til rådighed.

4.3.2.6 Sub-factors

- Strenge regler for at overholde persondataloven for at sikre en sikkert arbejdsplatform.

4.3.2.7 Kørselsvejledning

For at kunne udføre applikationen, skal der bruges en række software værktøj.

- **Java**

  Java downloades:

  På Javas hjemmeside (www.Java.com), skal der downloades Java ved at vælge **Free Java**, derefter klikkes **Kør** på meddelelseslinjen, og så vælges **Installer**. Java installeres færdigt. Klik på **finish** efter installation. Så er den installeret.
- **MySQL Database:**

  Ved at gå ind på (www.MySQL.com) og der vælges **MySQL Community Edition**, derefter skal der vælges den seneste udgave af programmet. Når man har downloadet og kørt MySQL, skal man vælge **Workbench**.
- **Intellij IDEA:**

  Inde på (www.JetBrains.com) vælges **Download** med Windows eller Mac og Linux. Efter den har downloadet, køres programmet og vælger de agreement som bliver krævet. Derefter installeres Intellij IDEA færdigt, og programmet kan derefter anvendes.

4.4 Use Case

I vores opgave har vi anvendt forskellige former for ' Use Case '. Vi har blandt andet valgt at tilføje et *Use Case Diagram , Brief, Casual og Fully-dressed.* I vores opgave vil vi derfor beskrive de enkelte use cases og forklare hvad de bliver anvendt til.

Til at starte med har vi et Use Case Diagram (**Se Use Case diagram)** I venstre siden af vores diagram kan man se vores aktører, vores aktører er dem der gør brug af programmet, og kan tildeles forskellige scenarios(scenarier) også kaldt use cases som er inde i den blå boks. Der kan være en eller flere aktører der kan gøre brug af en eller flere scenarier, som i vores opgave der både har en *Sales Assistant/Salgsassisten*t samt en *Owner/Ejer*, der har fået tildelt en eller flere scenarier. Disse scenarier kan man kalde en prototype til det program man senere skal programmere, derfor er det vigtigt at få tildelt disse use cases, så man har en forestilling om hvilke aktører der gør hvad i programmet, og hvad programmet til sidst skal ende med at indeholde.

Senere i vores opgave gør vi også brug af en Brief description / kort beskrivelse af vores use cases. Denne Brief description kan kort og kontant beskrives som en kort infotekst der beskriver den enkelte use case, normalt bruges Brief description til at beskrive en main succes scenarie (det bedst mulige udfald i programmet).

Derefter har vi taget brug af en Casual description, som er en lidt længere tekst end brief, der beskriver flere scenarier, eksempelvis som i vores program så kan man bruge en casual description til at vurdere værst tænkelige udfald i programmet.

Dette leder os så til sidst til vores Fully dressed. Alle trin er beskrevet detaljeret ind, og der vil være understøttende sektioner, såsom forudsætninger(preconditions) og succes garantier(succes guarantees).[1]

[1] Craig Larman, Applying UML and patterns, p. 124

4.4.1 Afgrænsning:

Gruppen har med dette projekt opnået virksomheden behov, nemlig at de kan oprette bookings, kunder og aflyse bookings. Gruppen har derudover lavet sider med oplysninger om de forskellige autocampere som virksomheden tilbyder, så medarbejderne kan se de specifikationer køretøjet har. Vi har i det følgende delt vores use cases op ift. hvilken der er blevet implementeret og ikke er blevet implementeret i systemet. Grund til at de manglende use cases ikke er blevet implementeret er pga. manglende tid.

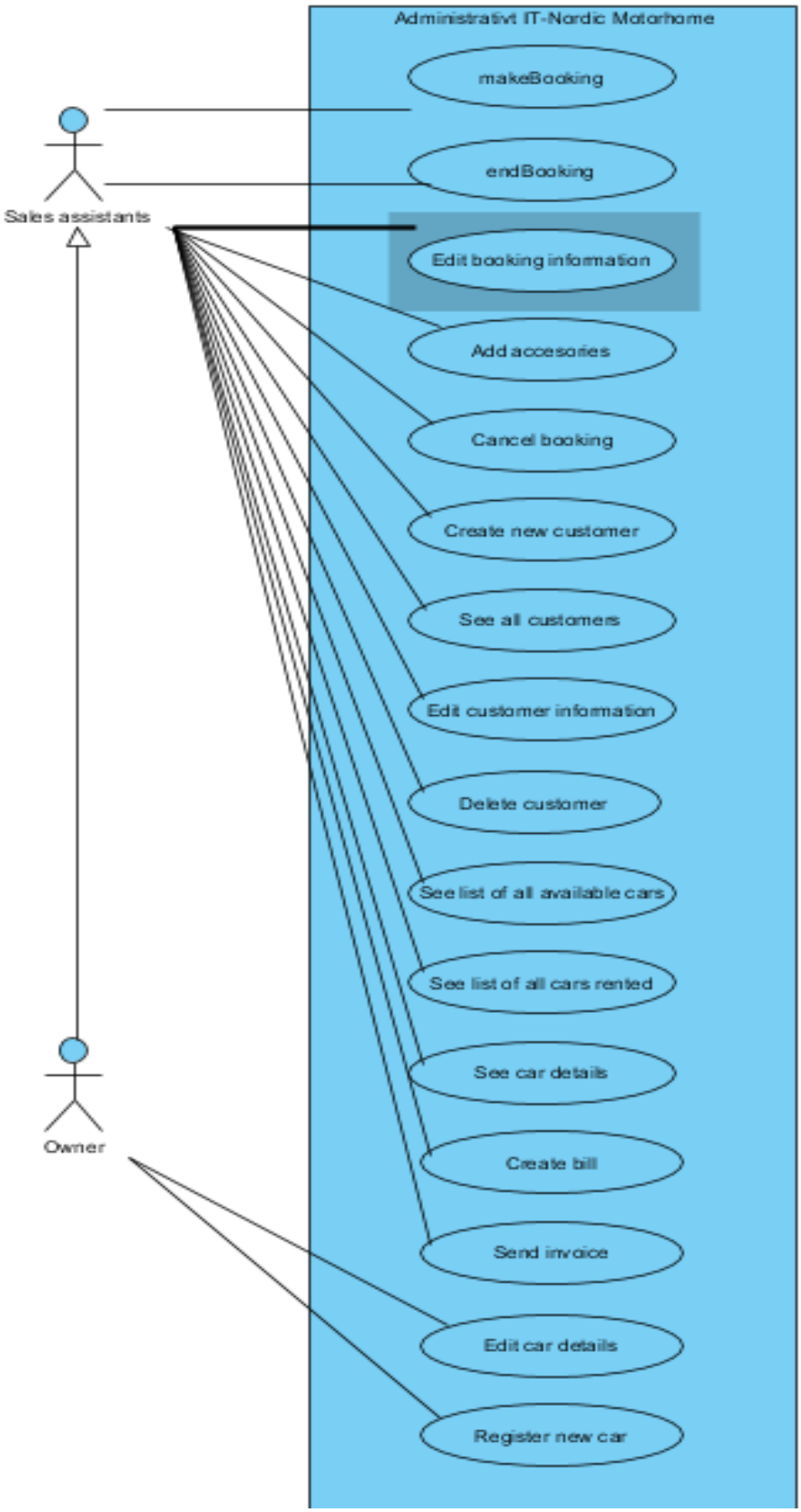
Disse use cases har gruppen implementeret i systemet:

- Create Booking: UC1
- End Booking: UC2
- Edit booking information: UC3
- Cancel booking: UC5
- Create new customer: UC6
- See all customers: UC7
- Edit customer information: UC8
- Delete customer: UC9
- See motorhome details: UC12

Disse use cases har gruppen IKKE implementeret i systemet:

- Add accessories: UC4
- See list of all available motorhomes: UC10
- See list of all motorhome rented: UC11
- Create bill: UC13
- Edit motorhome details: UC14
- Register new motorhome: UC15

4.4.2 Use Case diagram



4.4.3 Use Case brief

- ***Make Booking:***

  Når aktør gør brug af denne funktion, så laver aktøren en ny booking i systemet ved at indtaste diverse informationer han har fået fra kunden.

- ***End Booking:***

  Når aktør gør brug af denne funktion, så afslutter han bookingforløbet hos kunden.

- ***Edit booking information:***

  Når aktør gør brug af denne funktion, så redigerer han i bookingoplysningerne.

- ***Add accessories:***

  Når aktør gør brug af denne funktion, så tilføjer han ekstra tilbehør med til bookingen af autocamperen

- ***Cancel booking:***

  Når aktør gør brug af denne funktion, så annullerer han den booking der er blevet foretaget af kunden

- ***Create new customer:***

  Når aktør gør brug af denne funktion, så tilføjer han en ny kunde inde i systemet.

- ***See all customers:***

  Når aktør gør brug af denne funktion, så kan han se alle kunder der er registreret i systemet.

- ***Edit customer information:***

  Når aktør gør brug af denne funktion, så ændrer han i kundens oplysninger.

- ***Delete customer:***

  Når aktør gør brug af denne funktion, så sletter han en kunde i systemet.

- ***See list of all available autocampers:***

  Når aktør gør brug af denne funktion, så kan aktøren se alle autocampere der står til rådighed.

- ***See list of all autocampers rented:***

  Når aktør gør brug af denne funktion, så kan han se alle autocampere der er lejet ud i øjeblikket.

- ***See autocamper details:***

  Når aktør gør brug af denne funktion, så kan han se autocamperens detaljer.

- ***Create bill:***

  Når aktør gør brug af denne funktion, så laver han en regning til kunden.

- ***Edit car details:***

  Når aktør gør brug af denne funktion, så redigere han i bilens detaljer.

- ***Register new autocamper:***

  Når aktør gør brug af denne funktion, så registrerer han en ny autocamper i systemet

4.4.4 Use case casual

Main success Scenario

Brugeren logger ind i systemet, hvor aktøren kan vælge at oprette en ny kunde, oprette nye bookinger, registrere nye oplysninger til bilerne, ændre kundeoplysninger, ændre bookingoplysninger, slette kunder og annullere bookinger. Aktøren kan også få et overblik over alle disse oplysninger som f.eks. liste af kunder og bookinger.

Alternate Scenarios

Hvis kundens oplysninger bliver skrevet forkert ind eller at de bliver opdateret, så ved hjælp af systemet har brugeren mulighed for at redigere i diverse oplysninger, såsom kundens oplysninger eller bookinger.

4.4.5 Use case fully dressed

4.5.1 makeBooking

Use Case Section	Comment
Use Case Name	Make Booking
Scope	Nordic Motorhome
Level	User goal
Primary Actor	Salgsassistent
Stakeholders and Interests	- Salgsassistenten: Vil gerne registrere/oprette nye bookinger.
Preconditions	Salgsassistenten er identificeret og godkendt
Success Guarantee	Bookingen er fuldført i systemet.
Main Success Scenario	1. Salgsassistenten logger på Systemet. 2. Salgsassistenten registrerer en ny booking med de oplysninger han har fået. 3. Systemet opretter bookingen, så kunden kan få en bekræftelse. 4. Bookingen er oprettet i systemet, og kunden får tilsendt en bookingbekræftelse.
Extensions	**1a. Salgsassistenten kan ikke logge på Systemet.** 1. Salgsassistenten genstarter Systemet. 1a. Salgsassistenten logger ind. 1b. Salgsassistenten kan stadig ikke logge på Systemet og kontakter supporten. **2a. Systemet kan ikke oprettet bookingen** 1. Salgsassistenten prøver at oprette en ny booking i systemet. 1a. Bookingen bliver oprettet i systemet og leverer en bookingbekræftelse. 2a. Systemet kan stadig ikke oprette en ny booking. Salgsassistenten prøver andre metoder. **2. Salgsassistenten genstarter Systemet.** 1a. Salgsassistenten logger på Systemet. 1. Salgsassistenten opretter en ny booking. 1a. Systemet opretter bookingen og sender kunden en bekræftelsesmail. 2a. Systemet kan stadig ikke oprette en booking. Salgsassistenten prøver andre metoder.
Special Requirements	Usability: Brugervejledning, brugervenligt Reliability: Back-up servere Performance: 20 brugere, kort opstartstid Supportability: Test en gang om dagen, opdater løbende, Installation til Mac og Windows.
Technology and Data Variations List	1a. Kontooplysninger bliver indtastet via tastatur. 2a. Kundens oplysninger bliver indtastet via tastatur.
Frequency of Occurrence	Løbende
Miscellaneous	

4.5.2 registerCostumer

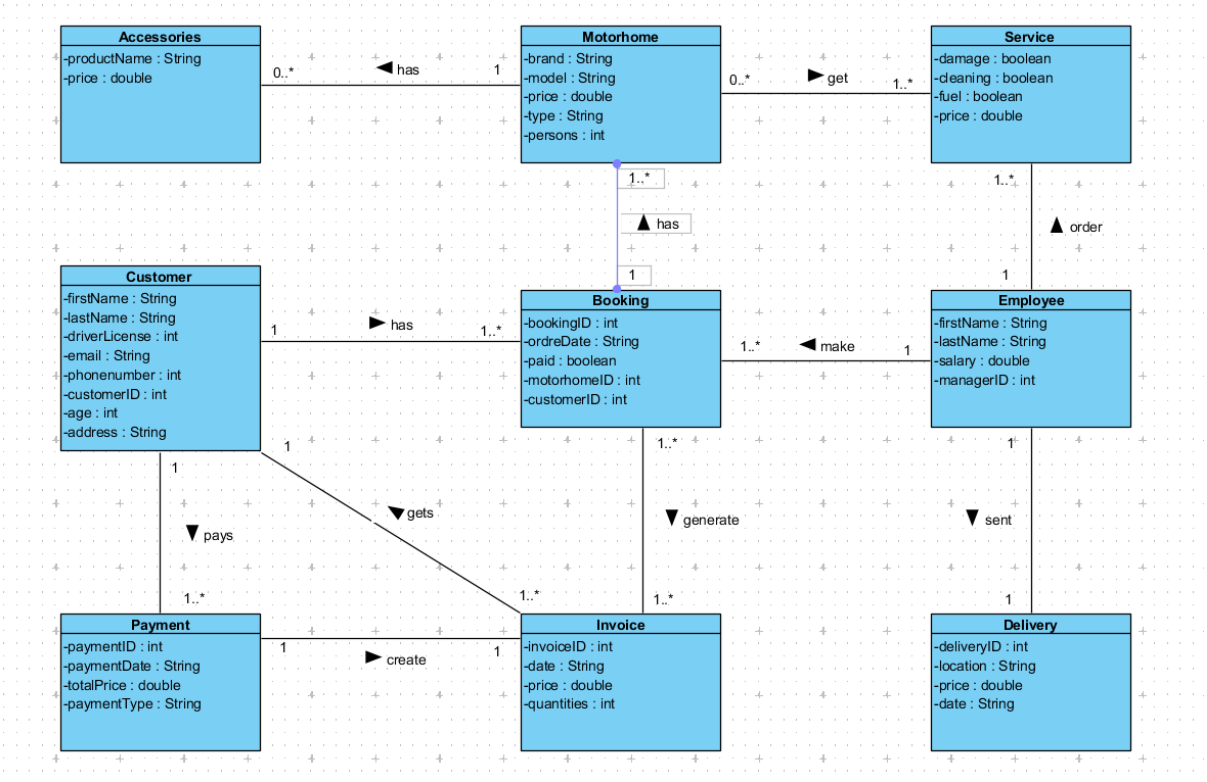
Use Case Section	Comment
Use Case Name	Register costumer
Scope	Nordic Motorhome
Level	User goal
Primary Actor	Salgsassistent
Stakeholders and Interests	- Salgsassistenten: Vil gerne registrere/oprette nye kunder. Ingen kunder = ingen salg - Ejeren: Ønsker at oprette nye biler til forretningen. Jo bredere varesortiment de har desto flere valg har kunderne også.
Preconditions	Salgsassistenten er identificeret og godkendt
Success Guarantee	Kunden er registreret i Systemet
Main Success Scenario	1. Salgsassistenten logger på Systemet. 2. Salgsassistenten registrerer en ny kunde med de oplysninger han har fået. 3. Systemet opretter kunden, så kunden kan foretage sig en booking. 4. Kunden er oprettet i systemet, og kunden får tilsendt en velkomstmail. 5. Kunden kan nu foretage sig en ny booking.
Extensions	**1a. Salgsassistenten kan ikke logge på Systemet.** 1. Salgsassistenten genstarter Systemet. 1a. Salgsassistenten logger ind. 1b. Salgsassistenten kan stadig ikke logge på Systemet og kontakter supporten. **2a. Systemet kan ikke oprettet en ny kunde** 1. Salgsassistenten prøver at oprette en ny kunde i systemet. 1a. Kunden bliver oprettet i systemet og leverer en velkomstmail 2a. Systemet kan stadig ikke oprette en ny kunde. Salgsassistenten prøver andre metoder. **2. Salgsassistenten genstarter Systemet.** 1a. Salgsassistenten logger på Systemet. 1. Salgsassistenten registrerer en ny kunde. 1a. Systemet opretter kunden og leverer en velkomstmail. 2a. Systemet kan stadig ikke oprette en ny kunde. Salgsassistenten prøver andre metoder. **4a. Salgsassistenten kan ikke sende velkomstmail grundet forkert e-mailadresse.** 1. Salgsassistenten overskriver info med opdateret e-mailadresse og sender velkomstmailen. 1a. Salgsassistenten kan stadig ikke sende velkomstmailen. Salgsassistenten prøver andre metoder.
Special Requirements	Usability: Brugervejledning, brugervenligt Reliability: Back-up servere Performance: 20 brugere, kort opstartstid Supportability: Test en gang om dagen, opdater løbende, Installation til Mac og Windows.
Technology and Data Variations List	1a. Kontooplysninger bliver indtastet via tastatur. 2a. Kundens oplysninger bliver indtastet via tastatur.
Frequency of Occurrence	Løbende
Miscellaneous	

4.5.3 cancelBooking

Use Case Section	Comment
Use Case Name	Cancel Booking
Scope	Nordic Motorhome
Level	User goal
Primary Actor	Salgsassistent
Stakeholders and Interests	- Salgsassistenten: Vil gerne annullere bookinger.
Preconditions	Salgsassistenten er identificeret og godkendt
Success Guarantee	Annulleringen af bookingen er fuldført.
Main Success Scenario	1. Salgsassistenten logger på Systemet. 2. Salgsassistenten annullerer en booking med de oplysninger han har fået. 3. Systemet annullerer bookingen, så kunden kan få en bekræftelse på annulleringen. 4. Bookingen er annulleret i systemet, og kunden får tilsendt en bekræftelse på annulleringen.
Extensions	**1a. Salgsassistenten kan ikke logge på Systemet.** 1. Salgsassistenten genstarter Systemet. 1a. Salgsassistenten logger ind. 1b. Salgsassistenten kan stadig ikke logge på Systemet og kontakter supporten. **2a. Systemet kan ikke annullere bookingen** 1. Salgsassistenten prøver at annullere bookingen i systemet. 1a. Bookingen bliver annulleret i systemet og leverer en annulleringsbekræftelse. 2a. Systemet kan stadig ikke annullere en booking. Salgsassistenten prøver andre metoder. **2. Salgsassistenten genstarter Systemet.** 1a. Salgsassistenten logger på Systemet. 1. Salgsassistenten annullerer en bookingen. 1a. Systemet annullerer bookingen og sender kunden en bekræftelsesmail. 2a. Systemet kan stadig ikke annullere en booking. Salgsassistenten prøver andre metoder.
Special Requirements	Usability: Brugervejledning, brugervenligt Reliability: Back-up servere Performance: 20 brugere, kort opstartstid Supportability: Test en gang om dagen, opdater løbende, Installation til Mac og Windows.
Technology and Data Variations List	1a. Kontooplysninger bliver indtastet via tastatur. 2a. Kundens oplysninger bliver indtastet via tastatur.
Frequency of Occurrence	Løbende
Miscellaneous	

4.6 Konceptuel model

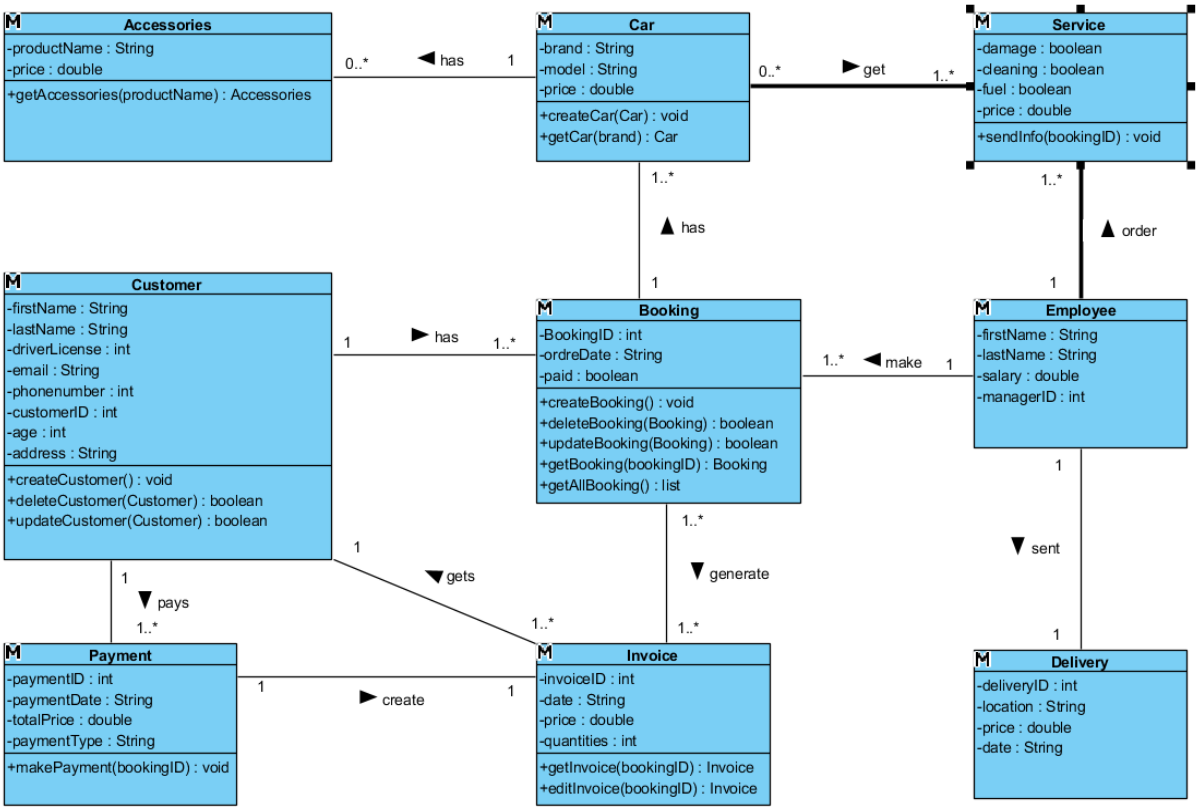
En domænemodel er en konceptuel (idemæssig) model af et system som beskriver de forskellige enheder som er i konceptet og beskriver deres relation til hinanden. Konceptuelle klasser (som ikke indeholder kode eller andre step), som indeholder attributter (data i klasserne) og associeringer (relation og sammenhæng). Den store forskel mellem domæne og klassediagrammet er at domæne ikke indeholder metoder i sin skabelon hvorimod klassediagram indeholder det.



4.7 Klassediagram

Klassediagrammer er designet baseret på OOP (Object Oriented Programming). Målet er at definere klasserne i vores software og deres forhold til disse. Vi vil gerne give et eksempel fra vores kode **(Se nedenstående kode 4.5.1.1)** og sammenligne dem med klassediagrammet.

4.7.1 Første version af klassediagrammet

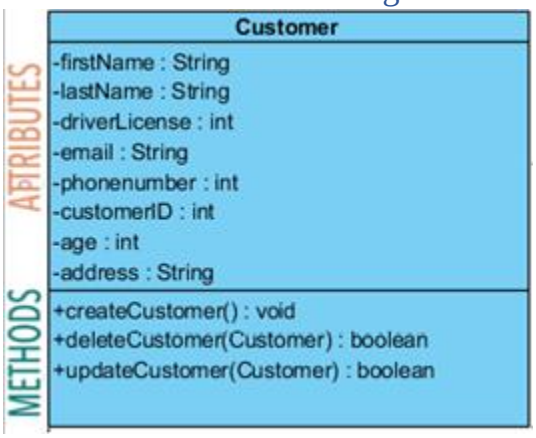


4.7.1.1 Eksempel af klasse I kode

```
public class Customer {
        private String firstName;
        private String lastName;
        private String email;
        private int phonenumber;
        private int customer_id;
        private int age;
        private String address_id;
```

Bemærk: Ikke alle getter-setter-metoder er skrevet for at undgå yderligere udvidelse af teksten. Sådan skrev vi ”Customer” klassen i Java. Men hvad nu hvis vi ville udtrykke denne klasse, så den kunne kodes på alle andre objektbaserede sprog? Så hvis vi skrev på et sprog, som andre softwareudviklere ville forstå. Så måtte vi udtrykke det i klasseskemaet.

4.7.1.2 Udsnit fra klassediagram



Ovenfor ser vi repræsentation af ”Customer” i klasseskemaet. Klasser udtrykkes i klassediagrammer på denne måde. Der er "Attributter" øverst, kaldt som klasseattributter og i dette diagramudsnit har vi (f.eks. Navn, efternavn, alder osv.) og for neden har vi metoderne/funktioner/operationer. Tegnet ”+” foran metoderne er tegnet på adgangsmodifikation (access modifier). Den viser, at attributten enten er public, package/default, privat eller protected. I dette tilfælde er vores operationer public. Hvis klassen var abstrakt, kunne vi udtrykke den ved at skrive kursiv i form af ”Customer” i stedet for ”Customer”.

4.7.2 Access-modifikationer:

- private
+ public
# protected
~ package/default

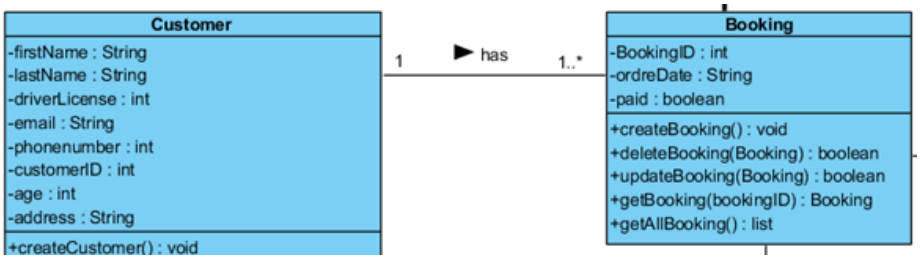
4.7.3 Generalisering / Arv

Generalisering er et UML-element som bruges til at vise relationen for en underordnet (subclass) til en overordnet (superclass). En sub class som er en underklasse kan arve fra en superklasse. Det kan for eksempel være en interfaceklasse man arver fra. Disse typer arverelationer kaldes ”IS-A” relation.

4.7.4 Association

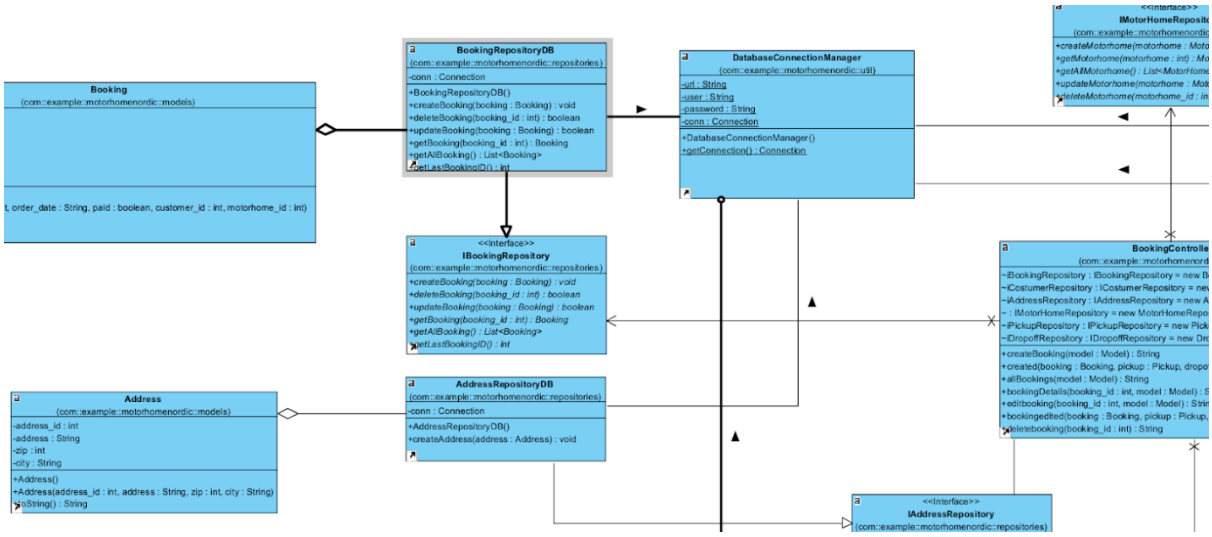
Associering er en af de forholdstyper i klassediagrammet, der fortæller os om sammenhængen mellem to klasser. Associations er delt i flere typer af forhold, men vi vil specifikt kun fokusere på de forhold vi har udarbejdet. F.eks. kan associationer fortælle os om forholdet mellem to klasser som kan befinde sig i en bi-directional (ensrettet forhold),
eller uni-directional (dobbeltretning forhold) eller det vi også kalder en aggregering. Den nedenstående tabel fortæller os om Multiplicity (mangfoldigheder).

0	No instances (rare)
0..1	No instances, or one instance
1	Exactly one instance
1..1	Exactly one instance
0..*	Zero or more instances
*	Zero or more instances
1..*	One or more instances



Multiplicity kan forståes, som en association mellem klasser, men at en bestemt klasse, kan have en eller flere af den klasse, klassen er associeret med. Som eksemplet viser ovenfor, så kan en Customer have flere Booking.

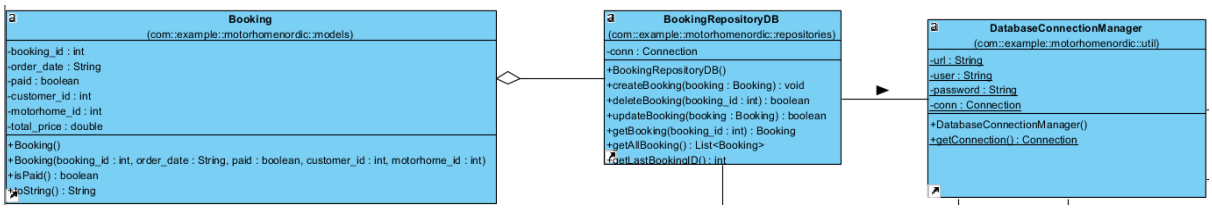
4.7.5 Klassediagram baseret på slutkode



Dette bilag er en del af vores klassediagram som er baseret på den endelig kode. Vi har valgt at bruge Visual Paradigm til at generere et klassediagram. Forskellen mellem det første og det sidste klassediagram er enorm. Vi har tilføjet en del klasser og opdateret vores associationer. Der er blevet tilføjet flere repositories-klasser og controller-klasser plus database-connection klassen. Vi har valgt at bruge "navigability" i nogle af vores associationer.

Dette kryds som er markeret fortæller os noget om navigability. BookingController kender til interfacen men interfacen kender ikke til BookingController.

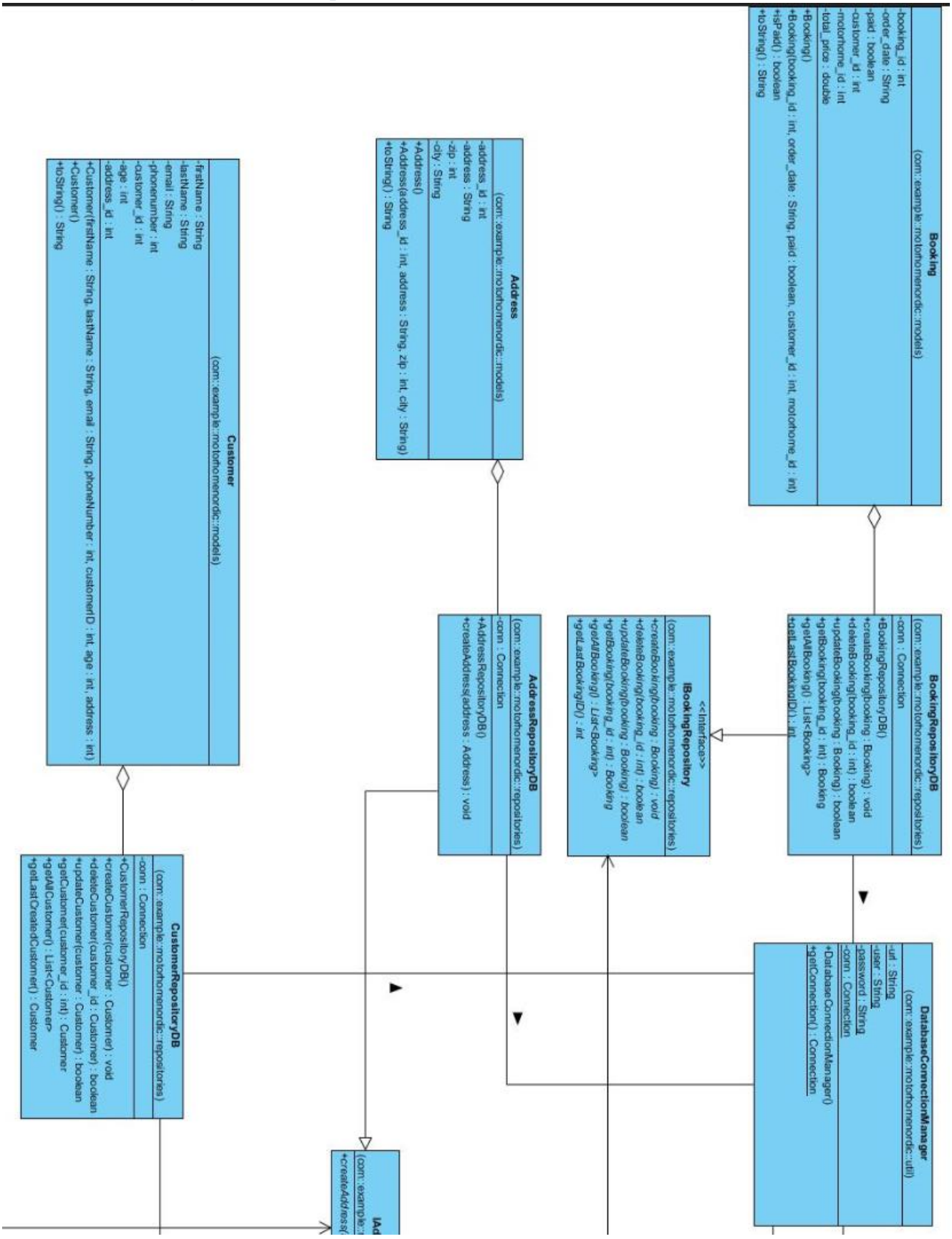
Vi har så også brugt en andet relations som kaldes Aggregation.

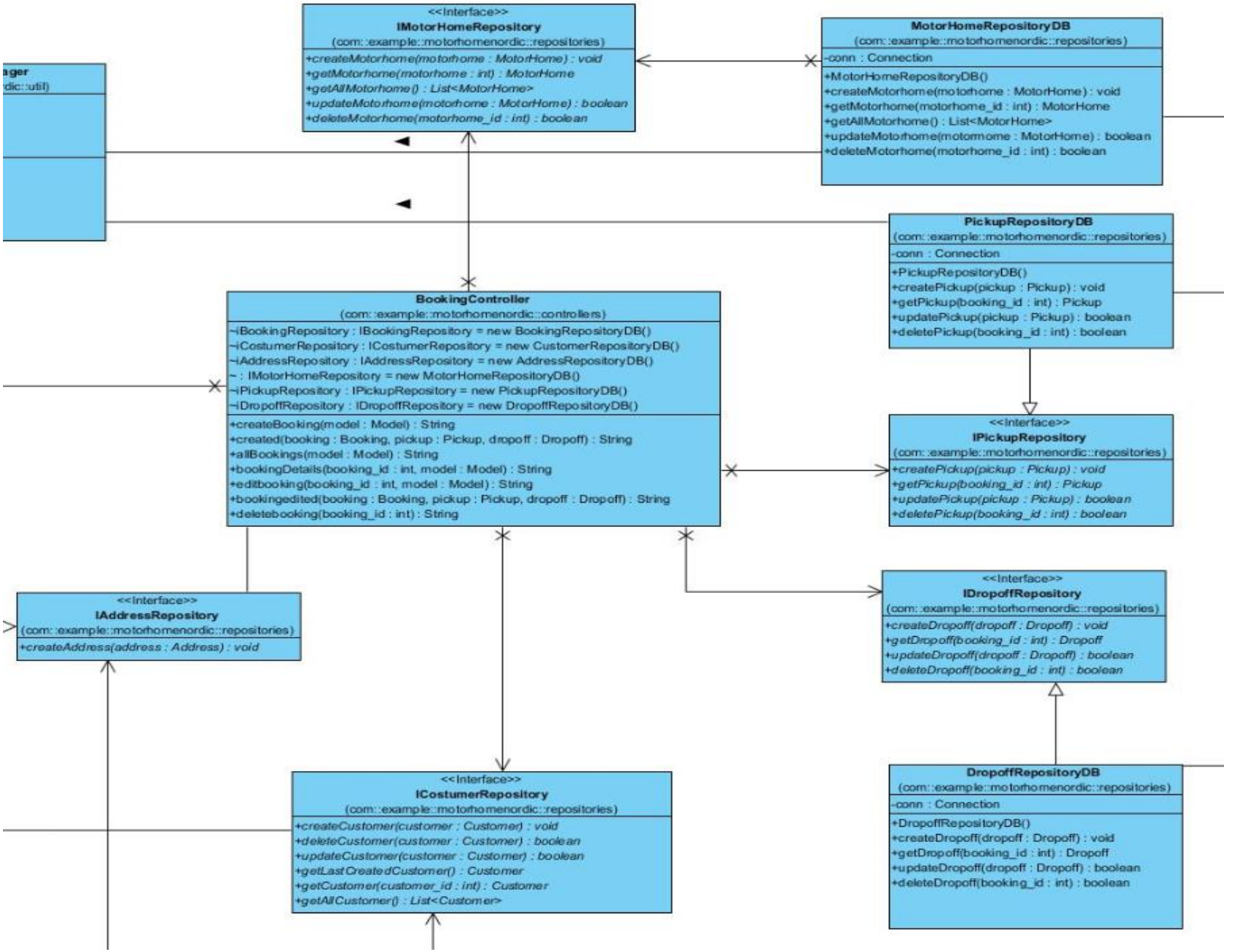


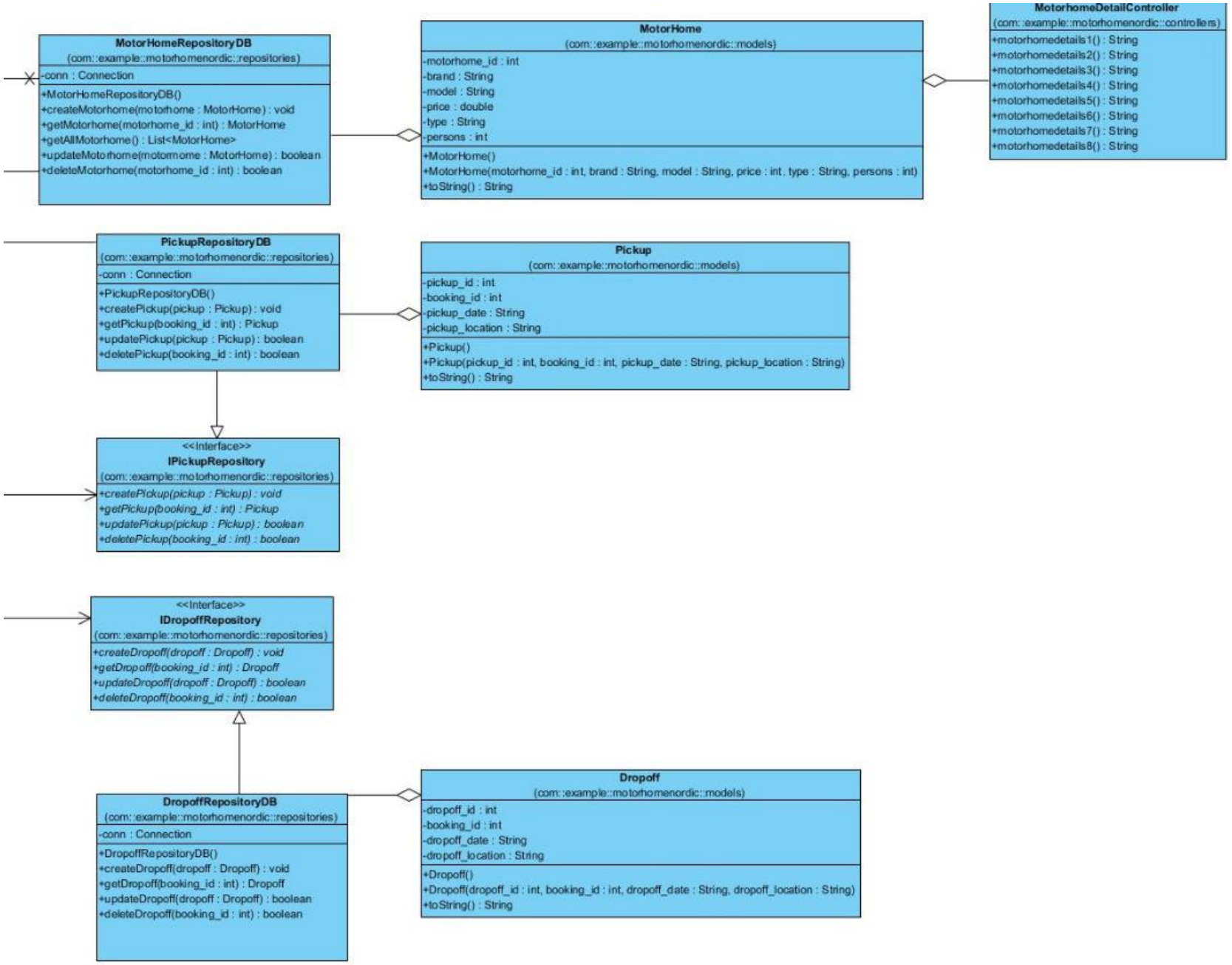
Aggregationen forholdet kan ses mellem BookingRepository og Booking klassen hvor BookingRepository er en del af Booking klassen. Når Booking klassen forsvinder, behøver BookingRepository imidlertid ikke at blive ødelagt der er et svagt forhold imellem dem. Dette forhold havde været modsat hvis der var tale om composition.

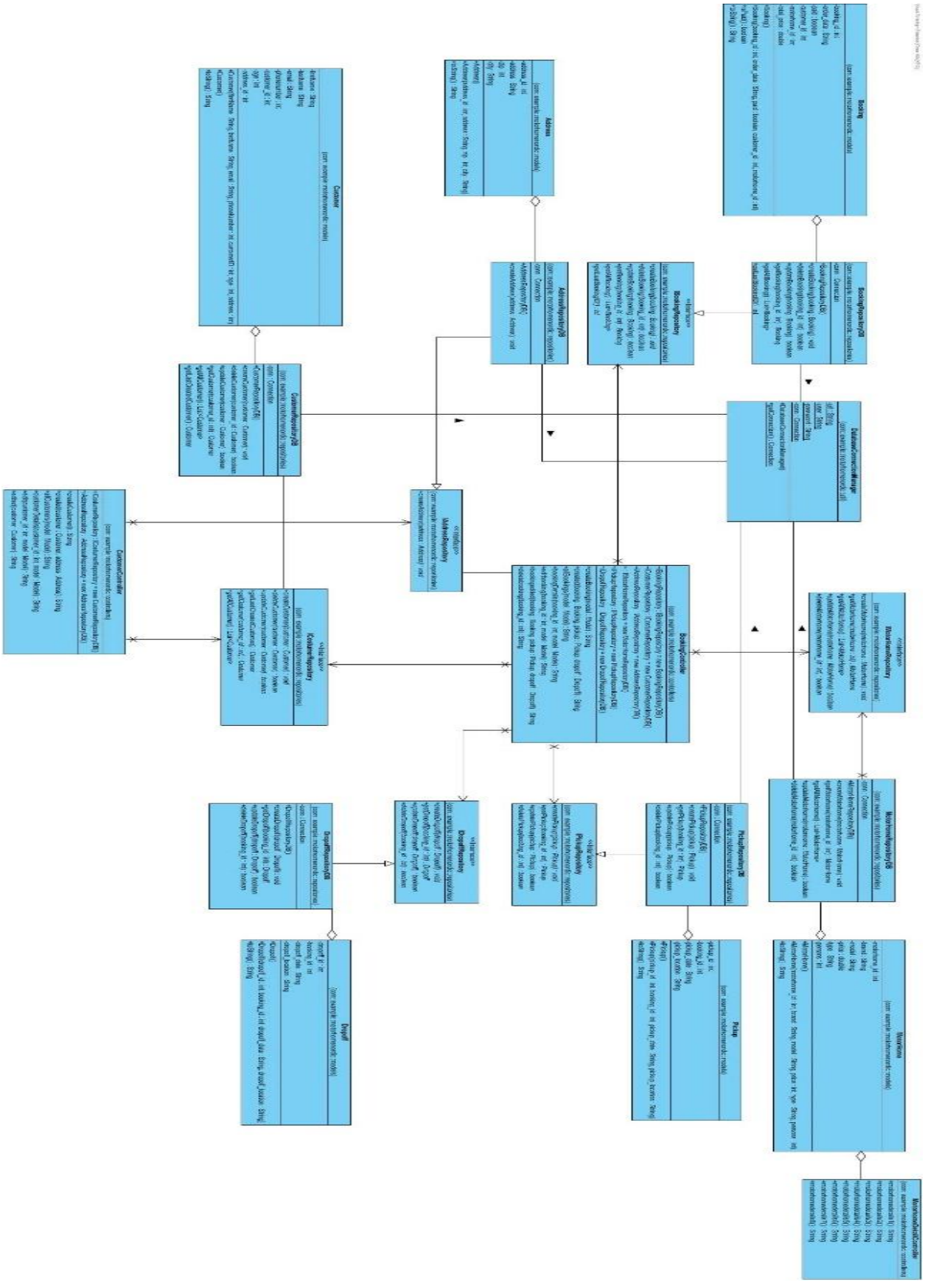
Det sidste vi har tilføjet, er at alle vores Repositories har en HAS relation til vores databaseConnection.

4.7.5.1 Klassediagram baseret på slutkode (Hel)




ager
dic::util)
<<Interface>>
IMotorHomeRepository
(com::example::motorhomenordic::repositories)
+createMotorhome(motorhome : MotorHome) : void
+getMotorhome(motorhome : int) : MotorHome
+getAllMotorhome() : List<MotorHome>
+updateMotorhome(motorhome : MotorHome) : boolean
+deleteMotorhome(motorhome_id : int) : boolean
MotorHomeRepositoryDB
(com::example::motorhomenordic::repositories)
-conn : Connection
+MotorHomeRepositoryDB()
+createMotorhome(motorhome : MotorHome) : void
+getMotorhome(motorhome_id : int) : MotorHome
+getAllMotorhome() : List<MotorHome>
+updateMotorhome(motorhome : MotorHome) : boolean
+deleteMotorhome(motorhome_id : int) : boolean
PickupRepositoryDB
(com::example::motorhomenordic::repositories)
-conn : Connection
+PickupRepositoryDB()
+createPickup(pickup : Pickup) : void
+getPickup(booking_id : int) : Pickup
+updatePickup(pickup : Pickup) : boolean
+deletePickup(booking_id : int) : boolean
BookingController
(com::example::motorhomenordic::controllers)
~iBookingRepository : IBookingRepository = new BookingRepositoryDB()
~iCostumerRepository : ICostumerRepository = new CustomerRepositoryDB()
~iAddressRepository : IAddressRepository = new AddressRepositoryDB()
~ : IMotorHomeRepository = new MotorHomeRepositoryDB()
~iPickupRepository : IPickupRepository = new PickupRepositoryDB()
~iDropoffRepository : IDropoffRepository = new DropoffRepositoryDB()
+createBooking(model : Model) : String
+created(booking : Booking, pickup : Pickup, dropoff : Dropoff) : String
+allBookings(model : Model) : String
+bookingDetails(booking_id : int, model : Model) : String
+editbooking(booking_id : int, model : Model) : String
+bookingedited(booking : Booking, pickup : Pickup, dropoff : Dropoff) : String
+deletebooking(booking_id : int) : String
<<Interface>>
IPickupRepository
(com::example::motorhomenordic::repositories)
+createPickup(pickup : Pickup) : void
+getPickup(booking_id : int) : Pickup
+updatePickup(pickup : Pickup) : boolean
+deletePickup(booking_id : int) : boolean
<<Interface>>
IAddressRepository
(com::example::motorhomenordic::repositories)
+createAddress(address : Address) : void
<<Interface>>
IDropoffRepository
(com::example::motorhomenordic::repositories)
+createDropoff(dropoff : Dropoff) : void
+getDropoff(booking_id : int) : Dropoff
+updateDropoff(dropoff : Dropoff) : boolean
+deleteDropoff(booking_id : int) : boolean
<<Interface>>
ICostumerRepository
(com::example::motorhomenordic::repositories)
+createCustomer(customer : Customer) : void
+deleteCustomer(customer : Customer) : boolean
+updateCustomer(customer : Customer) : boolean
+getLastCreatedCustomer() : Customer
+getCustomer(customer_id : int) : Customer
+getAllCustomer() : List<Customer>
DropoffRepositoryDB
(com::example::motorhomenordic::repositories)
-conn : Connection
+DropoffRepositoryDB()
+createDropoff(dropoff : Dropoff) : void
+getDropoff(booking_id : int) : Dropoff
+updateDropoff(dropoff : Dropoff) : boolean
+deleteDropoff(booking_id : int) : boolean


MotorHomeRepositoryDB
(com::example::motorhomenordic::repositories)
-conn : Connection
+MotorHomeRepositoryDB()
+createMotorhome(motorhome : MotorHome) : void
+getMotorhome(motorhome_id : int) : MotorHome
+getAllMotorhome() : List<MotorHome>
+updateMotorhome(motorhome : MotorHome) : boolean
+deleteMotorhome(motorhome_id : int) : boolean
MotorHome
(com::example::motorhomenordic::models)
-motorhome_id : int
-brand : String
-model : String
-price : double
-type : String
-persons : int
+MotorHome()
+MotorHome(motorhome_id : int, brand : String, model : String, price : int, type : String, persons : int)
+toString() : String
MotorhomeDetailController
(com::example::motorhomenordic::controllers)
+motorhomedetails1() : String
+motorhomedetails2() : String
+motorhomedetails3() : String
+motorhomedetails4() : String
+motorhomedetails5() : String
+motorhomedetails6() : String
+motorhomedetails7() : String
+motorhomedetails8() : String
PickupRepositoryDB
(com::example::motorhomenordic::repositories)
-conn : Connection
+PickupRepositoryDB()
+createPickup(pickup : Pickup) : void
+getPickup(booking_id : int) : Pickup
+updatePickup(pickup : Pickup) : boolean
+deletePickup(booking_id : int) : boolean
Pickup
(com::example::motorhomenordic::models)
-pickup_id : int
-booking_id : int
-pickup_date : String
-pickup_location : String
+Pickup()
+Pickup(pickup_id : int, booking_id : int, pickup_date : String, pickup_location : String)
+toString() : String
<<Interface>>
IPickupRepository
(com::example::motorhomenordic::repositories)
+createPickup(pickup : Pickup) : void
+getPickup(booking_id : int) : Pickup
+updatePickup(pickup : Pickup) : boolean
+deletePickup(booking_id : int) : boolean
<<Interface>>
IDropoffRepository
(com::example::motorhomenordic::repositories)
+createDropoff(dropoff : Dropoff) : void
+getDropoff(booking_id : int) : Dropoff
+updateDropoff(dropoff : Dropoff) : boolean
+deleteDropoff(booking_id : int) : boolean
DropoffRepositoryDB
(com::example::motorhomenordic::repositories)
-conn : Connection
+DropoffRepositoryDB()
+createDropoff(dropoff : Dropoff) : void
+getDropoff(booking_id : int) : Dropoff
+updateDropoff(dropoff : Dropoff) : boolean
+deleteDropoff(booking_id : int) : boolean
Dropoff
(com::example::motorhomenordic::models)
-dropoff_id : int
-booking_id : int
-dropoff_date : String
-dropoff_location : String
+Dropoff()
+Dropoff(dropoff_id : int, booking_id : int, dropoff_date : String, dropoff_location : String)
+toString() : String

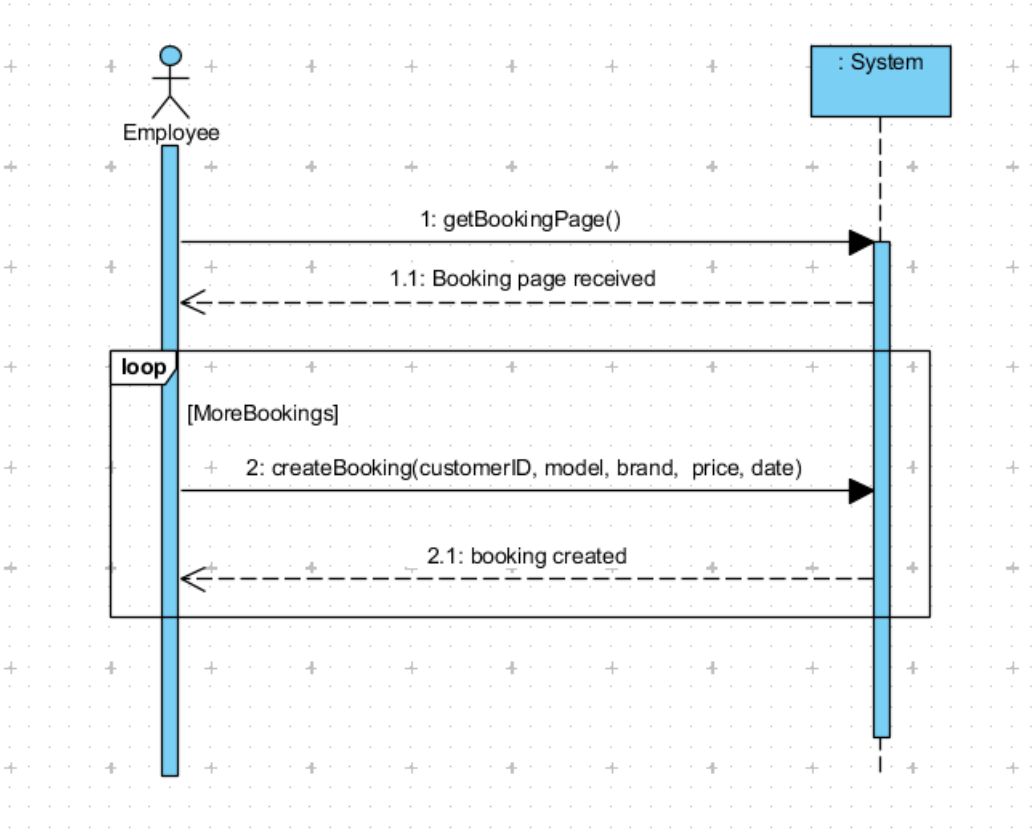


4.8 System sequence diagram

Gruppen har i forbindelse med opgaven valgt at udarbejde et System sequence diagram, som er et kort diagram fra UML som viser hvordan brugeren interagerer med system. På figur 1 har gruppen valgt at bruge en af de primære funktioner som er *Use case: 1 Make booking*.

Vi kan ud fra *Figur 4.8.1* se at, medarbejderen sender en anmodning til systemet på punkt 1, hvor anmodningen bliver besvaret ved at systemet svarer med en side, hvor bookingen bliver oprettet.

4.8.1 Figur 1: System sequence diagram: UC1- Make Booking

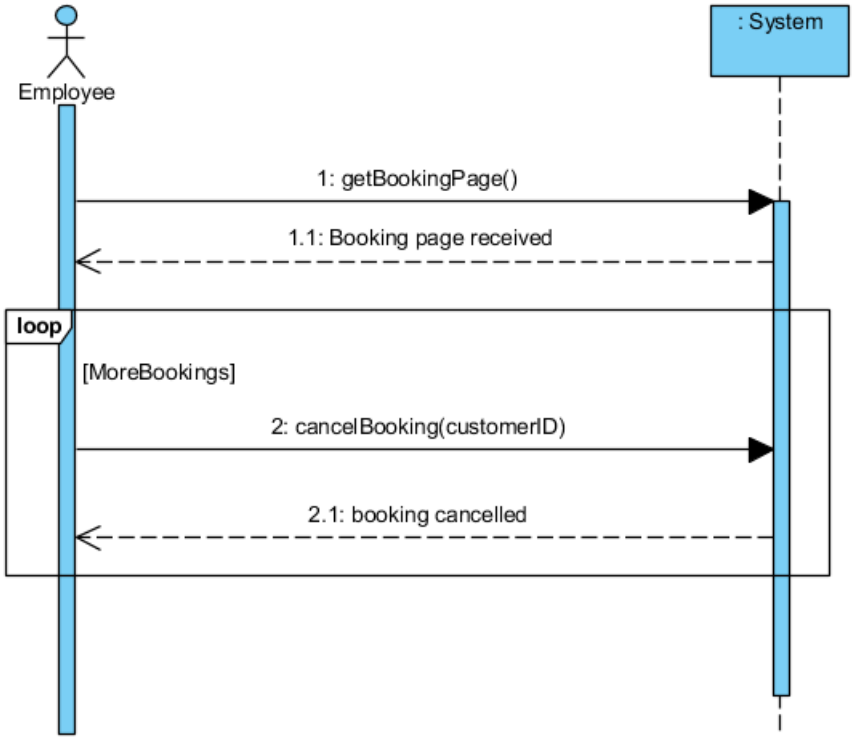


Derefter går medarbejderen i gang med at indtaste de nødvendige oplysninger inde på punkt 2, hvor han efterfølgende trykker på create booking-knappen, som sender en metode afsted i systemet og gemmer bookingen.

Vores System sequence diagram giver gruppen en god forståelse for hvordan systemet kommunikerer med brugeren og hvilke metoder der bliver brugt for dette. Det giver gruppen et godt overblik over hvilken kode gruppen skal tage brug af, når systemet bliver udviklet.

I vores andet System sequence diagram som kan ses på figur 4.8.2, har gruppen valgt at tage udgangspunkt i UC5 som er en funktion som sletter booking, hvis den bliver aflyst. Denne use case er også helt klart relevant ift. til systemet. Vi kan ud fra figur 4.8.2 se hvordan medarbejderen sender en request til systemet (punkt 1), hvor han efterfølgende får sendt en side tilbage. Brugeren vælger en booking som han gerne vil slette og trykker på slet knappen (punkt 2), hvor der så bliver sendt en bekræftelse tilbage til medarbejderen.

4.8.2 Figur 2: System sequence diagram: UC5 - Cancel booking



Dette System sequence diagram giver ligesom det sidste diagram en god forståelse for hvordan kommunikationen foregår mellem brugeren og system, hvilket igen givet os et overblik over hvordan koden skal sætte i værk.

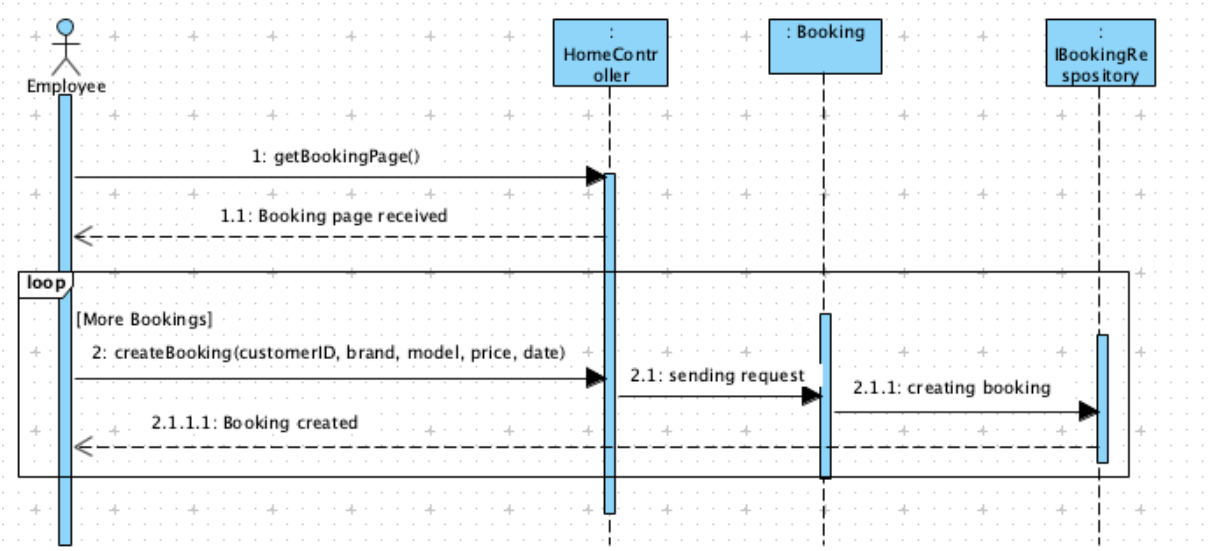
4.9 Sequence diagram

Gruppen har valgt at lave et Sequence diagram, som er en udvidet udgave af System sequence diagrammet. Forskellen her er, at vi kan se hvilke objekter brugeren interagerer med, hvilket gør at gruppen kan dykke yderligere ned i de enkelte use cases. På vi *Figur 4.9.1* har vi udarbejdet et Sequence diagram som tager udgangspunkt i *UC1: Make booking*, som vi også har udarbejdet et System sequence diagram af. Det har vi gjort, så vi kan forstå og se kodestrukturen af den pågældende use case, så kan man derefter udarbejde klasse objekter til systemet.

Figur 4.9.1 viser hvordan en medarbejder opretter en booking ved hjælp af systemet. Vi kan se at medarbejderen afsender en request til HomeController(punkt 1), som sender en bookingside tilbage.

Herefter ser vi at medarbejderen udfylder bookinginformationerne og trykker opret, som sender informationerne videre til *HomeControlleren* (Punkt 2), som derefter sender data videre til *Bookingklassen* (Punkt 2.1), hvor metoden befinder sig. Herfra bliver koden i metoden eksekveret som gemmer data inde på databasen (Punkt 2.1.1).

4.9.1 Figur 3: Sequence Diagram: UC1 - Make booking



4.10 GRASP

General responsibility assignment software patterns eller Grasp er regler for hvordan man tildeler ansvar til klasser og objekter i objektorienteret design. Den sikrer et godt design af et system.

4.10.1 Expert

Hvilke ansvarsområder skal de enkelte klasser have?

Her tildeles den klasse, der har den nødvendige information til at udføre handlingen, ansvar for handlingen.

f.eks: Objekt A skal have alt information om objekt B for at initialiserer den.

4.10.2 Creator

Hvem er ansvarlig for at oprette nye objekter (instanser) af en given klasse?

En klasse A skal være ansvarlig for at oprette nye objekter af en anden klasse B, hvis en eller flere af de følgende ting gælder:

- A indeholder objekter af typen B
- A består af objekter af typen B
- A kender de data, som B skal initialiseres med
- A bruger objekter af typen B meget

4.10.3 Low coupling

Lav kobling omhandler, at klasserne i programmet er lette at udskifte og eventuelt genbruges.

Programmer med høj kobling (forbindingen mellem klasser), kan genkendes ved at ændringer i en klasse har større indflydelse på de andre dele af programmet, og at koden i den enkelte klasse kan være svær at forstå.

Man skal derfor sørge for klasserne har lav kobling, så klasserne ikke er særlige afhængig af hinanden, således at programmet bliver mere overskueligt og at det bliver lettere at udskifte en eller flere dele af programmet.

Fordele ved lav kobling:

- Genanvendelighed - programmet kan anvendes til en anden opgave.
- Fleksibilitet - de anvendte dele af programmet som ikke ønskes, kan lettere udskiftes med noget andet.
- Overskuelige dele af programmet kan forstås uafhængigt af hinanden.

Hvordan opnås lav kobling?

- Man sørger for, at klasserne har referencer til så få andre klasser som muligt.
- Man nævner specifikke egenskaber i interfaces, og lader variable og parametre være af interfacets type, så man undgår at være ud på en bestemt implementering.
- Man skal give den enkelte klasse et ansvarsområde, som er ensartet og let at forstå.

4.10.4 High cohesion

High cohesion (Høj kohæsion) er en måling for hvor stærk relationer data/metoder har til hinanden inde i et objekt/klasse.

Klasserne med lav cohesion har data som ikke relatere til hinanden. Et eksempel kan være at man kun tildeler kundeklassen opgaven til at registrere ordrerne. Her vil kundeklassen have en høj kohæsion, fordi den kun er tildelt et ansvarsområde. Hvis man også tildeler kunde klassen opgaven til at tage sig af produktpris, så vil kohæsion af kunde klassen falde drastisk, fordi prisklassen ikke er direkte relateret med kunde klassen.

Målet er at dele klasserne i ansvarsområder der er overskuelige og lette at forstå. Men jo flere ansvarsområder et objekt eller en klasse har, jo lavere bliver kohæsionen.

4.10.5 Pure fabrication

Eksemplet her kunne være at der noget som skal være med i en applikation som ikke kan ses som en klasse eller et objekt i virkelighedens verden. Eller at vi har opførelse (metode) som ikke tilpasses med vores klasser. I stedet for at presse det i vores eksisterende klasser, kan vi opfinde en ny klasse selvom den har ikke eksisteret i vores konceptuel model. Fordelen vil være at vi undgår lav kohæsion. Husk at vi vil have lav kobling og høj kohæsion.

4.10.6 Indirection
Formålet med indirection er at reducere coupling(kobling) ved at introducere et/en mellemobjekt/mellemklasse. Det er en ide, hvor vi kan reducere kobling med objekterne. Hvis vi f.eks. har for mange objekter som skal kommunikere med hinanden, vil det være uundgåeligt at have høj kobling mellem dem, samt mange afhængigheder(dependencies). Det man kan gøre i stedet, er at reducere direkte forbindelser, ved tilføje et indirekte objekt imellem dem. På den måde forkortes forbindelser til enhver objekt.

4.10.7 Controller
Controller er ligesom en mellemmand mellem vores business klasse og brugerinterface. Man vil helst undgå at have høj kobling mellem dem, ved at binde dem direkte sammen, hvor business klassen skal vide noget om brugerinterface, og brugerinterface skal vide noget om businessklassen. Man kan derfor oprette en kontroller klasse udelukkende for at være intermediærer mellem de to klasser. Et godt eksempel er designmønster Model- view-controller (MVC) som bruges til programmer med brugergrænseoverflade.

Vores primære controller i vores program som står for at opretholde alle objekter og køre hele programmet er BookingControlleren. Controlleren henter data fra databasen til Viewet og gemmer data fra Viewet til databasen uden at påvirke systemet.

4.10.8 Polymorphism
Polymorfisme er et objekts evne til at antage mange former. F.eks. at et Animal klasse nedarver sin attributter og metoder til en Kat klasse. Det betyder at når vi opretter et objekt af klassen Kat, så kan vi sætte datatypen som Animal, da det er en del af Animal klasse.

4.10.9 Protected variations
Beskytte systemet fra ændringer og udvekslinger.

Hvordan man kan designe et system, så ændringer og udvekslinger har en minimal indflydelse på det der allerede eksisterer. Vi skal derfor identificere ramme (framework, abstraction) omkring de meste sårbare data. Her taler vi om indkapsling af data, gøre vores data privat, og brug af interface.

4.10.10 Grasp anvendt i koden

Kode 1: Expert

Gruppen har under udvikling af systemet anvendt de forskellige grasp punkter for at give et godt design og fleksibilitet i koden.

Et af de punkter vi har brugt i vores kode, er Expert som vi hyppigt har gjort brug af i vores controller klasser, hvor vi har oprettet flere repository-objekter. Dette kan ses i dette eksempel:

```
@Controller
public class BookingController {
  IBookingRepository iBookingRepository;
  ICostumerRepository iCostumerRepository;
  IAddressRepository iAddressRepository;
  IMotorHomeRepository iMotorHomeRepository;
  IPickupRepository iPickupRepository;
  IDropoffRepository iDropoffRepository;
  public BookingController() {
    iBookingRepository = new BookingRepositoryDB();
    iCostumerRepository = new CustomerRepositoryDB();
    iAddressRepository = new AddressRepositoryDB();
    iMotorHomeRepository = new MotorHomeRepositoryDB();
    iPickupRepository = new PickupRepositoryDB();
    iDropoffRepository = new DropoffRepositoryDB();
  }
```

Ud fra dette afsnit fra vores bookingController klasse kan vi se hvordan vi initialiser vores repository klasser i controller klassen. Controller klassen har alt den information for at kunne oprette disse objekter, med andre ord så er controller klassen ekspert til hvordan repository klassen bliver oprettet.

Kode 2: Creator

Vi har også anvendt Creator i vores kode både i vores BookingController og CustomerController. Det kan vi se her på dette eksempel fra vores CustomerController klasse:

```
@Controller
public class CustomerController {
  ICostumerRepository iCostumerRepository;
  IAddressRepository iAddressRepository;
  public CustomerController() {
    iCostumerRepository = new CustomerRepositoryDB();
    iAddressRepository = new AddressRepositoryDB();
  }
```

Vi kan her se hvordan CustomerController initialiser andre objekter i konstruktøren, når klassen selv bliver initialiseret.

Kode 3: High cohesion

Gruppen har også anvendt high cohesion som betyder at de enkelte klasser kun har de ansvarsområder som de er ansvarlig over. Med andre ord betyder det klassen kun skal lave det den er skabt til. Vi vil nu vise eksempel fra vores kode, hvor vi anvender dette:

```
public interface IBookingRepository {
  public void createBooking(Booking booking);
  public boolean deleteBooking(int booking_id);
  public boolean updateBooking(Booking booking);
  public Booking getBooking(int booking_id);
  public List<Booking> getAllBooking();
  public int getLastBookingID();
}
```

Vi har valgt at tage udgangspunkt i Interface-klassen: IBookingRepository for at vise et kort eksempel. Her ser vi at den klasse som kommer til, implementerer fra IBookingRepository kommer kun til at indeholde disse 6 metoder som kun er relateret til Booking. Derfor har vores klasse en high cohesion, da den kun er ansvarlig for det den er skabt til.

Kode 4: Controller

Vores primære controller i vores program som står for at opretholde alle objekter og køre hele programmet er BookingControlleren. Controlleren henter data fra databasen til Viewet og gemmer data fra Viewet til databasen uden at påvirke systemet.

5 Database

5.1 Normalisering af database

Normalisering har til opgave at sikre rettelser i databasen på en mere sikker og simpel måde uden at have en stor indflydelse på anden data. Dette gør den ved at formindske redundant data, som er det data som bliver gentaget flere steder. Derfor er normalisering oplagt at anvende i databasen, da det gør det markant lettere for systemet at foretage rettelser i databasen.

En af de primære problemer med redundans data er at systemet kan blive langsommere, da det redundante data kan være skyld i at der bliver brugt unødvendige ressourcer som hukommelse, netværk og cpu.[2]

En anden ulempe kan være, at det vil være besværligt at ændre data i databasen flere steder end hvad man behøver. Derfor er normalisering et stort plus, da man kun behøver at ændre data et sted i databasen. Normalisering består af flere normalformer, men vi vil i denne opgave gør brug af 1, 2 og 3.normalform.

Gruppen har udarbejdet nogle tabeller som skal være en del af databasen og har valgt at lave normalisering på alle tabellerne, så vi kan fjerne alt unødvendigt data der gentages, og derved øge brugervenligheden af databasen. Gruppen har valgt at vise normaliseringsprocessen for den primære booking tabel, som kan ses på *Figur 1*.

[2] https://balslev.io/programmering/database/normalisering-af-databaser/

5.1.1 1. Normalform

Det første step i normaliseringsprocessen er 1. Normalform som tager udgangspunkt i følgende fire punkter som betegnes for at være de basiske regler for at opnå en 1. Normalform:

1. Hver kolonne skal indeholde en enkel værdi
2. En kolonne skal kun indeholde den samme datatype som er angivet
3. Hver kolonne skal have et unikt navn
4. Rækkefølgen af hvordan værdierne gemmes har ingen betydning f.eks. det behøver ikke at være højst til lav

Vi kan ud fra disse 4 punkter se hvor vigtig det er at skabe tabeller med 1.normalform, uden disse punkter vil tabellen blive betegnet som en dårlig database design. Disse punkter er med til at strukturerer database designet på en måde så der ikke opstår forvirringer og problemer. Derfor er oplagt at have en database design som er funktionelt og godt designet, da det vil have en stor indflydelse på det pågældende system som skal bruge databasen. Det er anbefalet at man som minimum bruger 1.normalform for sit databasedesign.[3]

Gruppen vil nu se om vores tabel følger disse 4 punkter på *Figur 4*.

5.1.2 Figur 4: 1. Normalform

Booking_id	Customer_id	Orderdate	Invoice_total	Invoice_date	Payment_total	Payment_date	Paid
1012	1	10.06.20	4500	10.06.20	4500	12.06.20	1
1020	2	21.07.20	3200	21.07.20	3200	23.07.20	1
1023	3	01.07.20	3000	01.07.20	3000	03.07.20	0
1015	4	01.07.20	2000	01.07.20	2000	03.07.20	1

Figur 4:

En databasetabel der indeholder bookinginformationer om udlejning (OBS: i kolonnen ”paid” bruger vi en ’ boolean ’ for at sikre os at kunden har betalt.)

[3] https://www.studytonight.com/dbms/first-normal-form.php

Vi kan ud fra *Figur 4*, se at vores tabel allerede opfylder det første punkt ved at have en enkel værdi i kolonner. Derudover opfylder tabellen også punkt Nr. 2, Nr. 3 og Nr. 4, da der kun bliver angivet samme datatype i alle kolonner og at hver kolonne har et unikt navn. Derfor kan vi konkludere at tabellen allerede på forhånd er sat i 1. Normalform. Det vil sige at vi kan begynde at udarbejde 2. Normalform.

5.1.3 2. Normalform

Det andet step i normaliseringsprocessen er 2. Normalform som tager udgangspunkt på i følgende tre punkter:

1. Det er et krav at tabellen er i 1.normalform
2. Vigtigt at have viden om domænet
3. Ingen partiel funktionel afhængighed

Det vigtigste element i 2. Normalform er partiel funktionel afhængighed og det eksisterer når primærnøglen i tabellen har nogle kolonner som enten er delvis eller slet ikke afhængig af primærnøglen. Derfor vil vi nu separere de kolonner som er delvist eller ikke afhængig af primærnøglen som i vores tilfælde er Booking_id og foreign key customer_id som tilsammen bliver en sammensat nøgle.[4]

Vi kan ud fra *Figur 4*, se at alle kolonner er afhængig af både booking_id og customer_id, hvilket betyder at vores tabel allerede er i 2. Normalform, da de andre kolonner ikke er afhængige af andre kolonner udover den sammensatte nøgle og derfor er der ikke noget partiel funktionel afhængighed tabellen.

[4] https://www.studytonight.com/dbms/second-normal-form.php

5.1.4 Figur 5: 2. Normalform

Invoice_id	Invoice_total	Invoice_date
10012	4500	10.06.20
12034	3200	21.07.20
12054	3000	01.07.20
12009	2000	01.07.20

Figur 5: Invoice

Gruppen har også valgt at separere alt der er relateret til Invoice i *Figur 4*, da de indbyrdes er afhængig af hinanden og derfor udgør en transitiv funktionel afhængighed på *Figur 4*. Vi kan på *Figur 5*, se hvordan invoice_total og invoice_date har fået deres egen primærnøgle, hvilket er med til at gør kolonner kun afhængige af primærnøglen som er Invoice_id.

5.1.5 3. Normalform

Det tredje step i normaliseringsprocessen er 3. Normalform som tager udgangspunkt i følgende tre punkter:

1. Det er et krav at tabellen er i 2. Normalform.
2. Kolonner må ikke eksistere uden for primærnøglen, der er afhængig af hinanden[5].
3. Ingen transitive funktionelle afhængigheder.

Det vigtigste element i 3. Normalform er transitive funktionelle afhængigheder, som betyder at en kolonne er afhængig af en anden kolonne udover primærnøglen. Derfor skal tabellen deles op i flere små bidder, så alle kolonner som er afhængige af hinanden, får deres egen tabel med deres egen primærnøgle. Gruppens tabel er nu klar til 3. Normalform, da tabellen er godkendt for 2. Normalform.

Gruppen har nu delt tabellen i *Figur 4*, op i tre figurer med hver deres tabel, hver figur indeholder en tabel med en primærnøgle, hvor alle kolonner er afhængige af primærnøglen.

[5] https://balslev.io/programmering/database/normalisering-af-databaser/

5.1.6 Figur 6: 3. Normalform

Payment_id	Payment_total	Payment_date
20012	4500	12.06.20
22034	3200	23.07.20
22054	3000	03.07.20
22009	2000	03.07.20

Figur 6: Payment

På *Figur* 3, kan man se hvordan vi har separeret payment fra *Figur 4*, hvilket sætter begge tabeller i 3. Normalform. Payment_total og Payment_date er nu begge afhængige af den nye primærnøgle payment_id.

5.1.7 Figur 7: Overblik

Booking_id	Customer_id	Invoice_id	Payment_id	Orderdate	Paid
1012	1	10012	20012	10.06.20	0
1020	2	12034	22034	21.07.20	1
1023	3	12054	22054	01.07.20	1
1015	4	12009	22009	01.07.20	1

Figur 7: *Booking*

Vi kan ud fra *Figur 7*, se at customer_id, ordredate og paid kun er afhængig af primærnøglen, hvilket gør at vores tabel kommer i 3. Normalform. Derudover har vi tilføjet invoice_id og payment_id som foreign keys, da de refererer til de andre tabeller.

5.1.8 Konklusion på normalisering

Vores booking tabel har nu været igennem normaliseringsprocessen og er derfor velegnet til at blive anvendt i vores system, som et godt databasedesign. Vi forventer at normalisering har gjort vores arbejde en del nemmere og vil give systemet en øget effektivitet.

Vi kan ud fra de opdelte tabeller se, at der er en betydningsfuld understøttet integritet mellem tabellerne, da Parent-tabellen, som er booking nu har to Child-tabeller som har en direkte forbindelse til Parent-tabellen. Child-tabellerne er nu afhængig af Parent-tabellen. Hvis der muligvis skulle komme en ændring i Parent-tabellen, så vil det også have en indflydelse på Child-tabellerne. En af de primære årsager til at vi har valgt at lave det på denne måde og følge normaliseringsprocessen er, at vi ikke behøver at ændre data i tabellerne flere steder end nødvendigt og derved gør databasen mere kompleks.

5.2 ER-Diagram

Entity Relationship Diagram viser forholdet mellem enheder som er gemt i databasen. Diagrammet hjælper med at forstå den logiske struktur i databasen. Derudover giver det mulighed for at designe og visualisere databasen, der giver et bedre indblik i de forskellige enheder, da man kan identificere de enheder der eksisterer i systemet og deres relationer mellem de forskellige enheder[6]. Gruppen kan ved hjælp af ER-diagrammet finde frem til hvilke enheder der har en forbindelse til hinanden ved hjælp af noget der hedder, ER-cardinality som kan ses på *Figur 8*. Med disse cardinalities er det muligt at fortælle ud fra en numerisk kontekst hvad minimum og maksimum relationen er mellem enhederne. ER-diagrammet hjælper med at definerer enhederne, attributterne og relationerne der til sammen danner et grundlag for et blueprint til databasen.

Hvad er en enhed?

- En enhed er således et objekt som kan være levende som et menneske, eller noget ikke levende som en bil. Enheden består primært af en attribut, som er primærnøglen som de andre attributter er afhængige af.

Hvad er en attribut?

- Attributter er det der definerer det pågældende objekt som f.eks. at bilen har 4 hjul osv.

Hvad er en relation?

- En relation er blot en forbindelse mellem to enheder

Databasen som gruppen har udarbejdet, indeholder i alt 11 tabeller som kommer til at gemme vigtige informationer om virksomheden, deres kunder og autocampere. For at visualisere alle disse tabeller har gruppen valgt at lave et ER-diagram som vil vise alle tabellerne, attributterne og deres relationer til hinanden. Det vil også tydeliggøre integriteten af databasen. Til at oprette vores ER-diagram har gruppen anvendt programmet Lucidchart som er en webapplikation til visualisering af diverse diagrammer.

[6] https://www.guru99.com/er-diagram-tutorial-dbms.html

I vores ER diagram som kan ses på *Figur 9,* har alle vores tabeller en primærnøgle som afsluttes med ordet ”id” som i databasen bliver sat til at være auto_increment, hvilket betyder at hver gang der bliver oprettet en række i tabellerne, så stiger primærnøglen automatisk, uden at vi selv skal indtaste et id-nummer hver gang.

Alle de enheder som har en relation til hinanden, har vi mærket med en ER-cardinality som viser forholdet med en numerisk pil. Derudover har nogle af vores tabeller en foreign key som tydeliggøre integriteten mellem de forskellige tabeller, da den viser at de har en forbindelse til hinanden. Vi har også valgt at lave vores tabeller med to SQL-koder ”ON UPDATE CASCADE” og ”ON DELETE CASCADE” der har en vigtig betydning for databasen.

Det betyder at hvis en række som er blevet slettet i en Parent-tabel med en primærnøgle som har foreign key til andre Child-tabeller, så bliver data også slettet i Child-tabellen. Det samme gælder for update i Parent-tabeller dvs. at hvis det bliver ændret i parent tabel, så sker ændringen også child tabellen.

Vores primære tabeller i databasen er Booking, Motorhome og Customer. Booking-tabellen indeholder de vigtige elementer, der gør det muligt at beskrive en bookingordre som har 2 foreign keys fra customer og motorhome. Customer-tabellen indeholder alle kundeoplysninger. Motorhome-tabellen indeholder alle oplysninger vedrørende køretøjer, virksomheden har sat til udlejning. Tabellen beskriver blandt andet hvor mange personer køretøjet er til, og hvad er prisen er for det enkelte køretøj.

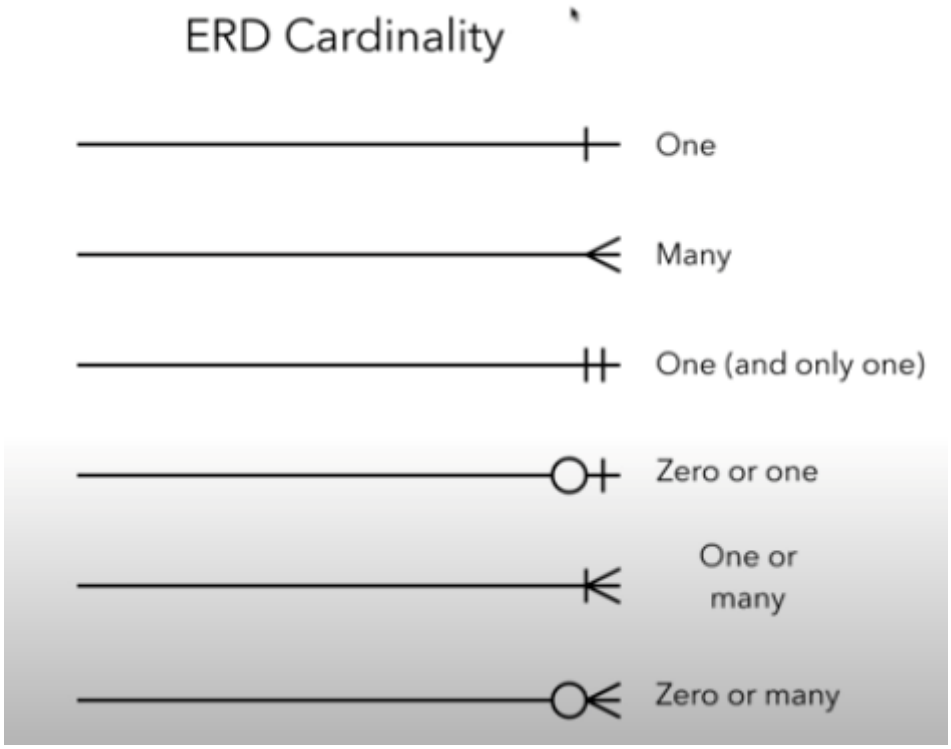
Tabellerne invoice og payment har hver deres tabel, for at virksomheden får et bedre overblik over regninger som er sendes til kunden, og hvorvidt de har betalt eller ej.
Begge tabeller har deres egen primærnøgle. Invioce-tabellen har en foreign key fra Booking-tabellen og Payment-tabellen har en foregn key fra Invoice-tabellen.

Tabellerne Pickup og Dropoff er oprettet, for at holde styr på hvor og hvornår kunden skal aflevere den udlejede campingvogn tilbage på den angivet adresse som kunden ønsker. Begge tabeller har hver deres primærnøgle som refererer til Booking-tabellen.

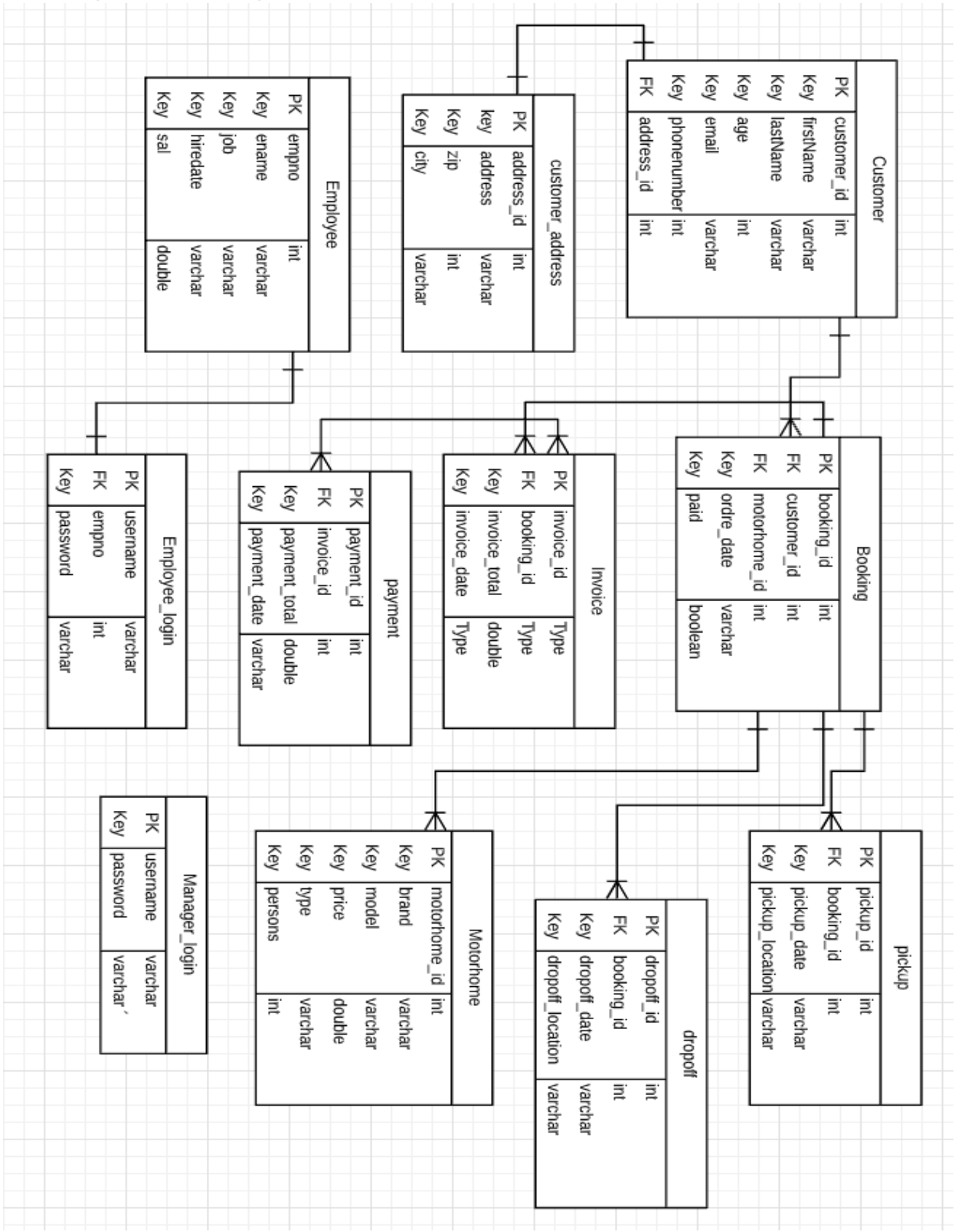
Derudover har vi også valgt at inddrage en tabel til medarbejdere, da virksomheden har 8 ansatte. Virksomhed kan ved hjælp af tabellen kan holde styr på den enkelte medarbejders løn osv. Gruppen forventer også at lave en loginside til de medarbejder der står for salg, derfor har gruppen valgt at lave 2 tabeller i databasen, som kommer til at indeholde login oplysninger som ”username” og ”password”. Dette har gruppen valgt for at gøre hjemmesiden mere sikker for virksomheden.

ER-diagrammet ses på *Figur 9*, diagrammet er klar til at blive anvendt som en rigtige database. Alle tabeller der havde brug for at blive normaliseret er blevet gjort igennem normaliseringsprocessen ved hjælp af de 3 normalformer.

5.2.1 Figur 8: ERD Cardinality



5.2.2 Figur 9: ER-Diagram



5.3 SQL SCRIPT

5.3.1 Oprettelse af database

```
CREATE TABLE customer_adress(
address_id int(6) PRIMARY KEY,
address varchar(40) NOT NULL,
zip int(8) NOT NULL,
city varchar(40) NOT NULL,
PRIMARY KEY(address_id)
);
```

```
CREATE TABLE Employee(
empno int(6) AUTO_INCREMENT,
ename varchar(40) NOT NULL,
job varchar(40) NOT NULL,
hiredate varchar(40) NOT NULL,
sal double NOT NULL,
PRIMARY KEY(empno)
);
```

```
CREATE TABLE Employee_login(
username varchar(40) NOT NULL,
empno int(6) AUTO_INCREMENT,
password varchar(40) NOT NULL,
PRIMARY KEY(username),
FOREIGN KEY(empno)
REFERENCES Employee(empno)
);
```

```
CREATE TABLE Manager_login(
username varchar(40) NOT NULL,
password varchar(40) NOT NULL,
PRIMARY KEY(username)
);
```

```
CREATE TABLE Customer(
customer_id int(6) AUTO_INCREMENT,
firstName varchar(40) NOT NULL,
lastName varchar(40) NOT NULL,
age int(2),
email varchar(40) NOT NULL,
phonenumber int(8) NOT NULL,
address_id int(6),
PRIMARY KEY (customer_id)
FOREIGN KEY (address_id)
REFERENCES customer_address(address_id)
);
```

```
CREATE TABLE motorhome(
motorhome_id int not null auto_increment PRIMARY
KEY,
brand varchar (30),
model varchar (30),
price double,
type varchar (30),
persons int
);
```

```
CREATE TABLE booking (
booking_id INT NOT NULL AUTO_INCREMENT,
customer_id INT NOT NULL ,
motorhome_id INT NOT NULL ,
order_date varchar (30) NOT NULL,
total_price double NOT NULL,
paid Boolean NOT NULL,
 PRIMARY KEY(booking_id),
FOREIGN KEY(customer_id) REFERENCES
customer (customer_id),
FOREIGN KEY (motorhome_id) REFERENCES
motorhome (motorhome_id)
);
```

```
CREATE TABLE pickup (
pickup_id int not null auto_increment,
booking_id int not null,
pickup_date varchar (30),
pickup_location varchar (30),
Primary key(pickup_id),
foreign key (booking_id) references
booking(booking_id)
);
```

```
CREATE TABLE dropoff(
dropoff_id int not null auto_increment PRIMARY
KEY,
booking_id int not null,
foreign key (booking_id), references booking
(booking_id)
dropoff_date varchar (30),
dropoff_location (30)
);
```

```
Create table  invoice(
invoice_id int not null auto_increment PRIMARY
KEY,
booking_id int not null,
invoice_total double,
invoice_date int,
 foreign key (booking_id) references
booking(booking_id));
```

```
Create table  payment(
Payment_id int not null auto_increment PRIMARY
KEY,
Invoice_id int not null,
invoice_total double,
invoice_date int,
 foreign key (invoice_id) references
invoice(invoice_id)
);
```

5.3.2 Kode indsat i database

```
INSERT INTO motorhome (motorhome_id,brand,model,price,type,persons) VALUES (1," Coachmen RV
","2013","5528","F",5),(2,"Coachmen RV","2013","5670","D",3),(3,"Forest River
RV","2018","6561","C",2),(4,"Entegra Coach ","2014","4304","C",2),(5,"Holiday
Rambler","2010","7022","K",3),(6,"Holiday Rambler","2014","7312","G",6),(7,"American
Coach","2012","7527","F",3),(8," Coachmen RV ","2012","6897","A",6),(9,"American
Coach","2018","5626","K",6),(10,"American Coach","2013","4557","B",2);
```

```
INSERT INTO motorhome (motorhome_id,brand,model,price,type,persons) VALUES  (11,"Tiffin
Motorhomes","2009","7481","F",4),(12,"Forest River RV","2014","6067","E",3),(13,"Coachmen
RV","2009","3434","E",6),(14,"Forest River RV","2011","4178","G",5),(15,"Coachmen
RV","2009","4188","F",6),(16,"Entegra Coach ","2009","6063","B",4),(17,"Forest River
RV","2010","6969","B",6),(18,"Entegra Coach ","2018","6829","C",4),(19,"Entegra Coach
","2011","5910","F",3),(20,"Forest River RV","2011","6325","D",4);
```

```
INSERT INTO motorhome (motorhome_id,brand,model,price,type,persons) VALUES (21,"Holiday
Rambler","2014","5341","D",2),(22,"Forest River RV","2014","3395","A",6),(23,"Tiffin
Motorhomes","2010","4403","D",6),(24,"Forest River RV","2009","5563","B",6),(25,"American
Coach","2009","3501","F",5),(26,"Tiffin Motorhomes","2018","5355","D",3),(27,"Holiday
Rambler","2018","7837","A",6),(28,"Entegra Coach ","2018","7798","D",2),(29,"American
Coach","2009","3960","E",2),(30," Coachmen RV ","2012","7369","B",2);
```

```
INSERT INTO motorhome (motorhome_id,brand,model,price,type,persons) VALUES (31,"Tiffin
Motorhomes","2011","5889","A",3),(32,"Tiffin Motorhomes","2012","6451","C",5);
```

5.3.3 Test data

Gruppen har valgt at lave enkelte test for at sikre at vores system virker optimalt, dette har vi gjort ved hjælp af Junit. Vi har kun valgt at teste database-klasserne som står for at hente og gemme data, for at sikre om metoderne virker. Grundet mangel på tid har vi ikke valgt lave test på alle klasser og metoder.

Gruppen har udarbejdet en tabel for de metoder som er blevet testet, dette kan ses på *Figur* 10:

5.3.3.1 Figur 10: Test tabel

Klasse	Metoder	Testet
BookingRepository	getLastBooking()	JA
BookingRepository	getBooking(id)	JA
CustomerRepository	getCustomer(id)	JA

5.3.3.2 Testeksempel kode

```
@Test
public void getBookingTest(){
  //Arrange
  bookingRepositoryDB = new BookingRepositoryDB();
  Booking booking = new Booking(10, "2020-05-28", true, 16, 1);
  //ACT
  int expected = booking.getBooking_id();
  int actuel = bookingRepositoryDB.getBooking(10).getBooking_id();
  //Assert
  assertEquals(expected, actuel);
}
```

Vi kan i dette eksempel se at vi tester et objekt som vi har oprettet, og sammenligner den med det objekt som kommer fra databasen.

6 Programdokumentation

6.1 Særlig komplekse programdele

I vores program har vi programdele, der har en særlig kompleks karakter. De programdele vi synes, der i projektet har været en kompleks opgave at arbejde med, er blandt andet *@Pathvariable*, *@ModelAttribute* og Databasen.
Vi har heldigvis ikke haft de store problemer med de forskellige metoder og programdele, men dog har det været en udfordring for os, fordi vi nyligt er blevet introduceret til disse nye metoder. Pathvariable og ModelAttribute, er noget vi for nyligt er blevet introduceret til i Spring, men database har vi tidligere arbejdet med på samme måde.

6.1.1 @PathVariable

Til at starte med har vi @Pathvariable, som vi har et udklip af ***(Se koden nedenfor).*** Vi plejer normalt at anvende metoden @RequestParam som henter data fra samme @GetMapping den er kodet i. Men sådan er det ikke med @Pathvariable, da den har til opgave at hente data fra andre sider end den @GetMapping den er kodet i. Dette er en fordel, da man kan hente data som id fra andre sider og indsætte det i en @GetMapping, hvor man kan bruge id'et til at hente flere oplysninger som nu bliver vist på en side for sig selv.

I vores tilfælde anvender vi @Pathvariable i BookingController klassen under @GetMapping"/bookingdetails{id}", hvor vi i parameter indsætter vores @Pathvariable som kommer fra {id}. Måden id'et bliver hentet på kan ses i vores HTML kode (se koden nedenfor), hvor man anvender thymeleaf med href og henviser til den getmapping id'et skal transporteres til, ved at anvende symbolet "+" mellem getMappingen og id'et. Det koden så går ind og gør er, at den udskriver en bookingDetails side, som er forbundet med den booking_id. Det vil sige, at hvis man f.eks. har en kunde der hedder ' Bo ', og han har booking_id'et 8, så vil siden vise bookingdetails/8, og vise alle de detaljer der er forbundet ' Bo'.

6.1.1.1 @PathVariable kode

```
@GetMapping ("/bookingdetails{id}")
   public String bookingDetails(@PathVariable("id") int booking_id, Model model){
      Booking booking = iBookingRepository.getBooking(booking_id);
      Pickup pickup = iPickupRepository.getPickup(booking_id);
      Dropoff dropoff = iDropoffRepository.getDropoff(booking_id);

      Customer customer = iCostumerRepository.getCustomer(booking.getCustomer_id());

      model.addAttribute("booking_id", booking.getBooking_id());
      model.addAttribute("order_date", booking.getOrder_date());
      model.addAttribute("total_price", booking.getTotal_price());
      model.addAttribute("paid", booking.isPaid());
      model.addAttribute("motorhome_id", booking.getMotorhome_id());
      model.addAttribute("customer_id", booking.getCustomer_id());



</tr>
<tr th:each="booking:${list}">

   <td th:text="${booking.booking_id}"></td>
   <td th:text="${booking.order_date}"></td>
   <td th:text="${booking.paid}"></td>
   <td th:text="${booking.customer_id}"></td>
   <!--<td th:text="${booking.motorhome_id}"></td>-->
   <td th:each="motor : ${motorlist}" th:if="${booking.motorhome_id == motor.motorhome_id}"
th:text="${motor.brand} + ' ' + ${motor.type}"></td>

   <td><a th:href="@{'/bookingdetails' + ${booking.booking_id}}" class="btn btn-success">Details</a></td>
   <td><a th:href="@{'/editbooking' + ${booking.booking_id}}" class="btn btn-info">Edit</a></td>
   <td><a href="${pageContext.request.contextPath }" th:href=" @{'/deleteBookings' +
${booking.booking_id}}" class="btn btn-danger" onclick="return confirm('Are you sure you want to delete
this booking?')">Delete</a></td>


</tr>
```

6.1.2 @ModelAttribute

@ModelAttribute har den egenskab at den henter alle de attributter fra en side, som matcher klassens attributter. Efterfølende opretter den et objekt med disse attributter, i dette tilfælde (**Se koden nedenfor**) har vi f.eks. Booking klassen som får sin data fra siden ved hjælp af @ModelAttribute. For at siden kan overføre disse data til getMappingen, så skal vi anvende HTML koden <form> og henvise til en getMapping, hvor vi efterfølgende taster vores data ind i <input> og trykker på submit knappen (dette kan ses nedenfor). Efter dette er gjort, så kan vi anvende Booking objeketet til at oprette bookingdata i databasen.

6.1.2.1 @ModelAttribute kode

```
@PostMapping("/created")
   public String created(@ModelAttribute Booking booking, @ModelAttribute Pickup pickup, @ModelAttribute
Dropoff dropoff){
      iBookingRepository.createBooking(booking);


      pickup.setBooking_id(iBookingRepository.getLastBookingID());
      iPickupRepository.createPickup(pickup);
      dropoff.setBooking_id(iBookingRepository.getLastBookingID());
      iDropoffRepository.createDropoff(dropoff);
```

```
<form action="/created" method="post">


   <br/>

   <div class="from-group">
      <label for="customer">Customer_id</label>
      <input type="number" name="customer_id" id="customer" th:value="${customer_id}" class="form-control"
required>
   </div>


  <div>
      <button type="submit" name="submit" class="btn btn-primary">Create</button>
   </div>


</form>
```

6.1.3 Database

Til at slutte af med har vi databasen. Vi har anvendt en database i MySQL, hvor vi skulle tilslutte en forbindelse mellem vores program og databasen. Dette har vi gjort således (**Se kode 6.1.3.1**). Til at starte med har vi vores adgang til databasen:

6.1.3.1 Database kode

```
private static String url;

private static String user;

private static String password;

private static Connection conn;
```

Disse data har vi gemt i vores application.properties, så den er beskyttet fra omverden, og man kun kan se koden igennem programmet. For at programmet skulle kunne registrere disse data som vi har sat ind i vores application.properties, så skal vi oprette en metode der læser vores fil, som er application properties, som vi har gjort nedenfor(**Se kode 6.1.3.2)**.

6.1.3.2 Database kode

```
Properties prop = new Properties();

    try{

        FileInputStream propFile = new FileInputStream("src/main/resources/application.properties");

        prop.load(propFile);

        url = prop.getProperty("db.url");

        user = prop.getProperty("db.user");

        password = prop.getProperty("db.password");

    }
```

Da vi som sagt havde brugt database i stedet for tekstfiler, så har vi også oprettet en online server, så alt data er tilgængelig for alle brugere. Dette gjorde vi igennem GearHost, der tilbyder fri online servere. Serveren er som sagt brugt til at få alle brugere med til at arbejde under et specifik database, i stedet for flere forskellige database, som også vil være grund til at der vil være forskellige kode i de individuelle steder, hvor gruppemedlemmerne har oprettet deres kode som eksempelvis *url, username og password.* MySQL er også nemmere at tage hånd om, og danner et meget større overblik end hvad filer kan.

6.2 Beskrivelse og dokumentation af løsninger

6.2.1 Construction

Construction er den fase hvor der bliver der bliver samlet koder til et færdigt program. Transition er så den sidste fase hvor man så afprøver programmet, laver et opdateret program, prøver og finde frem til mulig fejl bliver fundet.

6.2.2 Patterns

Gruppen har ved hjælp af Java udarbejdet forskellige løsninger til virksomheden. Java har i de seneste årtier været et meget populær programmeringssprog, som har udviklet kæmpe systemer som vi kan se i dag. Derfor er gruppen meget tilfreds ved at anvende dette sprog til at udvikle det nye system for virksomheden. For at kunne udvikle systemet i Java har vi også gjort brug af Frameworket Spring som er lavet specielt til Java for at gør det lettere for udviklerne. Spring er et kæmpe framework som indeholder virkelig mange funktioner, og derfor vil vi ikke anvende alt hvad Spring indeholder, men kun små dele af det. Vi kommer primær kun til at anvende Spring Boot med et design pattern som hedder MVC (Model View Controller).

6.2.3 MVC (Model View Controller)

MVC er et design pattern arkitektur som gør koden mere fleksibelt og holdbart samt reducerer kompleksiteten af designet. MVC består primært af 3 punkter:

Model

- Det er de forskellige objekter som kodet indeholder og bliver brugt som en skabelon til at transporterer data frem og tilbage fra f.eks. databasen og ud til viewet.

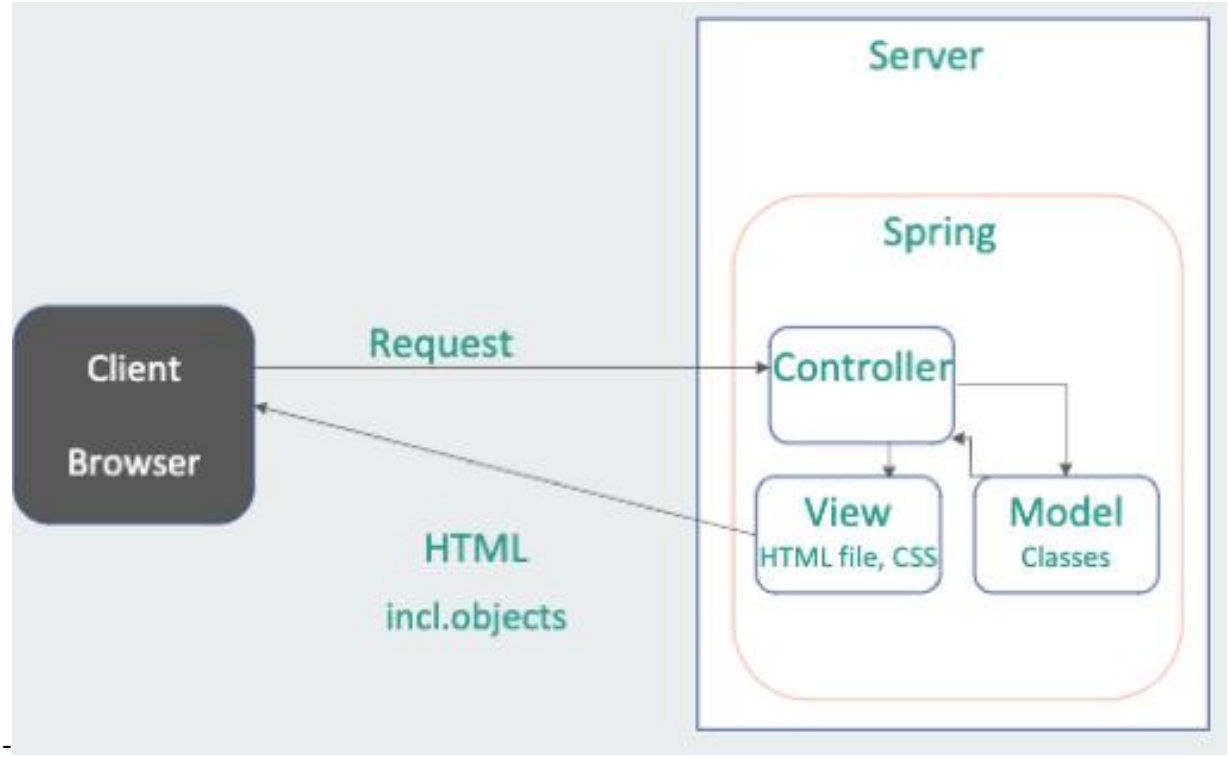
View

- View står primært for at visualiserer det data som modellerne indeholder så brugeren kan se det som User Interface.

Controller

- Controlleren har til opgave at modtage anmodning fra browseren, hvor den efterfølgende sender anmodningen videre til en model for at indsamle eller indsætte data. Her efterfølgende får den en model tilbage med data, hvor Controlleren sender data ud som View i form af HTML og CSS.

6.2.4 Figur 10: MVC



- [7]

For at få en bedre forståelse af MVC kan vi ud fra *Figur 10*, se hvordan en anmodning kommer ind til Controlleren, hvor den efterfølgende sender den videre til model for at hente eller indsætte data. Efter data er blevet hentet eller indsat, så får Controlleren en model tilbage som den så sender videre til View, som bliver vist til brugeren.

6.2.5 Brug af Model i kode

Gruppen vil nu vise hvordan vi har gjort brug af denne pattern. Derfor har vi valgt at tage udgangspunkt i vores "Make booking" use case, hvor vi gør brug af vores Booking klasse. For at kunne hente data til View, så gør vi brug af Thymeleaf som er en template engine som transporterer data fra controlleren til View. Dette vil blive illustreret igennem vores kode ved at vise et udklip **(Se kode 6.2.5.1)**

[7] [1] Understanding Spring (2) – Read-Only-pdf Fronter

6.2.5.1 Model kode

Vi kan i det følgende se modellen for Booking, hvor vi har nogle variabler og getter og setter som vil blive brugt for at kommunikerer med databasen og Thymeleaf.

```
public class Booking {
    private int booking_id;
    private String order_date;
    private boolean paid;
    private int customer_id;
    private int motorhome_id;
    private double total_price;

    public Booking() { }

    public Booking(int booking_id, String order_date, boolean paid, int customer_id, int motorhome_id) {
        this.booking_id = booking_id;
        this.order_date = order_date;
        this.paid = paid;
        this.motorhome_id = motorhome_id;
        this.customer_id = customer_id;
    }
     public int getBooking_id() { return booking_id; }

    public void setBooking_id(int booking_id) { this.booking_id = booking_id; }

    public String getOrder_date() { return order_date; }

    public void setOrder_date(String order_date) { this.order_date =order_date; }

    public boolean isPaid() { return paid; }

    public void setPaid(boolean paid) { this.paid = paid; }

    public int getCustomer_id() { return customer_id; }
```

```
public void setCustomer_id(int customer_id) { this.customer_id = customer_id; }

public int getMotorhome_id() { return motorhome_id; }

public void setMotorhome_id(int motorhome_id) { this.motorhome_id = motorhome_id; }

public double getTotal_price() {
    return total_price;
}

public void setTotal_price(double total_price) {
    this.total_price = total_price;
}
```

6.2.6 Brug af view i kode

Vi kan i den nedenstående kode **(Se kode 6.2.6.1)** se at vi anvender thymeleaf ved brug af ”th”, der indeholder flere forskellige metoder. I dette tilfælde har vi valgt at bruge th:each som er et loop som kører igennem en liste med Booking-objekter. Vi kan inde i th:each se, at det første i listen som kommer fra controlleren via Booking repository bliver sat lig med booking.
Herefter skal de variabler som er i objektet booking udskrives i Viewet. Dette sker igennem thymeleaf igen, hvor vi anvender th:text som sætter værdien fra Booking objektet ind i tabel data (td). Nu bliver det muligt for brugeren at se teksten som kommer igennem thymeleaf.

6.2.6.1 View kode

Vi vil nu vise et lille afsnit af vores kode i View, hvor vi viser hvordan data fra model bliver hentet til controlleren og videre til Viewet.

```
<tr th:each="booking:${list}">

                <td th:text="${booking.booking_id}"></td>

                <td th:text="${booking.order_date}"></td>

                <td th:text="${booking.paid}"></td>

                <td th:text="${booking.customer_id}"></td>

                <td th:each="motor : ${motorlist}" th:if="${booking.motorhome_id == motor.motorhome_id}" th:text="${motor.brand} + ' ' + ${motor.type}"></td>

                <td><a th:href="@{'/bookingdetails' + ${booking.booking_id}}" class="btn btn-success">Details</a></td>

                <td><a th:href="@{'/editbooking' + ${booking.booking_id}}" class="btn btn-info">Edit</a></td>

                <td><a href="${pageContext.request.contextPath }" th:href=" @{'/deleteBookings' + ${booking.booking_id}}" class="btn btn-danger" onclick="return confirm('Are you sure you want to delete this course?')">Delete</a></td>

</tr>
```

6.2.7 Brug af Controller i kode

Vi kan ud fra det nedenstående kodeeksempel (**Se kode 6.2.7.1)** i vores BookingController klasse se hvordan et PostMapping bliver anvendt til at fører modeller med data videre til databasen, ved at anvende det reserverede ord i Spring @ModelAttribute, så indsætter vi alle de data som bliver indtastet i Viewet ind i objektet(modeller) der står foran det reserverede ord. Efterfølgende bliver de objekter(modeller) indsat i en metode fra de forskellige Repository klasser der er, hvor de derefter bliver gemt i databasen. Afslutningsvis returnerer controlleren en side.
Dette eksempel viser hvilken rolle controlleren har, da den står for at hente og gemme data via modeller.

6.2.7.1 Controller kode

Vi vil i dette eksempel vise et udklip af vores Controller som vi har anvendt ift. At oprette en booking. Som nævnt tidligere står controlleren for at modtage en anmodning som sendes videre til modellen. Vi vil i det følgende kode eksempel demonstrere hvordan controlleren står for oprettelsen af bookings.

```
@PostMapping("/created")

public String created(@ModelAttribute Booking booking, @ModelAttribute Pickup pickup,
@ModelAttribute Dropoff dropoff){

		iBookingRepository.createBooking(booking);

		pickup.setBooking_id(iBookingRepository.getLastBookingID());

		iPickupRepository.createPickup(pickup);

		dropoff.setBooking_id(iBookingRepository.getLastBookingID());

		iDropoffRepository.createDropoff(dropoff);

		return "redirect:/";

}
```

6.2.8 Brug af Interface i kode

Vi har i vores kode brugt interface flere gange, for at kunne implementere det i Repository til hver model. Interfacen indeholder i alt 5 metoder som er CRUD (Create, read, update & delete) operations. Disse metoder står for at hente og gemme data fra databasen. I det koden nedenfor(**Se kode 6.2.8.1)** kan vi se et udsnit af vores interface IBookingRepository som indeholder CRUD til vores booking system.

6.2.8.1 Interface kode

```
public interface IBookingRepository {



		public void createBooking(Booking booking);

		public boolean deleteBooking(int booking_id);

		public boolean updateBooking(Booking booking);

		public Booking getBooking(int booking_id);

		public List<Booking> getAllBooking();

		public int getLastBookingID();



}
```

6.2.9 Brug af Exceptions i kode

For at undgå at systemet bryder sammen, så har gruppen flere steder udarbejdet en try & catch, som står for at håndtere exceptions. Dette er primært i vores repositories med SQL-statements, da den står for at hente og gemme fra databasen. Vi oplever tit at der er fejl ved at hente og gemme data, derfor er håndtering af exception meget vigtig, da det gør gruppen opmærksom på hvor fejlen kan være i koden og dermed løse koden hurtigere.

6.2.9.1 Exception kode

Her har vi et eksempel på hvordan vi har håndteret exceptions i vores kode, vi har i vores tilfælde valgt at kaste en SQLException, hvor der bliver printet en besked om den mulig fejl der kan være i koden.

Try and Catch er vigtig at have, da fejlen bliver fanget og dermed ikke stopper hele programmet.

```
@Override
public void createBooking(Booking booking) {


try{
   PreparedStatement prepared = conn.prepareStatement("INSERT INTO booking(customer_id, motorhome_id, order_date, total_price ,paid) VALUES (?,?,?,?,?)");
   prepared.setInt(1, booking.getCustomer_id());
   prepared.setInt(2, booking.getMotorhome_id());
   prepared.setString(3, booking.getOrder_date());
   prepared.setDouble(4, booking.getTotal_price());
   prepared.setBoolean(5, booking.isPaid());


   prepared.executeUpdate();


} catch (SQLException e) {
   e.printStackTrace();
   System.out.println(e.getMessage());
}
}
```

6.2.10 Login sikkerhed i kode

Gruppen har valgt at anvende et loginsystem til hjemmesiden, da hjemmesiden indeholder følsomme oplysninger om deres kunder. For at implementere dette loginsystem har vi anvendt Spring Security som gør arbejdet lettere for os. Spring security låser automatisk alle siderne på hjemmesiden, bortset fra de sider vi selv giver lov til at vise, indtil der ikke bliver indtastet et login. Gruppen har oprettet logintabeller i databasen, som indeholde username og password for at logge ind i hjemmesiden.
Gruppen har ikke valgt at tage alle de klasser der relateret til loginsystemet med i vores diagrammer, da det er noget vi endnu ikke har lært i undervisningen og derfor ikke kan forklare Spring Security i detaljer.

6.3 SQL i løsningen

6.3.1 Hovedfunktioner i MySQL

- Brugervenlighed - Download og brug af softwaren er meget let.
- Høj ydeevne - Giver dig hurtig indlæsning værktøjer med forskellige hukommelse cacher.
- Kompatibilitet - Kompatibel med alle moderne platforme som MySQL, Windows, Linux, Unix, Mac.
- Ydeevne - MySQL giver dig resultater med høj ydeevne uden at miste grundlæggende funktioner.
- Datasikkerhed - Kun autoriserede brugere kan få adgang til databasen. Det giver fuld sikkerhed for dataene.
- Lav pris - gratis at bruge.
- Hukommelses Effektivitet - Hukommelsesforbrug i MySQL er meget lav.

6.3.2 DML

Vi skriver de mest anvendte DML-kommandoer (Data Manipulation Language), som er data oprettelse, redigering og forespørgsel om sprog. Det bruges generelt til DML-data læsning og -behandling. Vi har valgt og definere de DML-kommandoer vi har brugt i vores SQL-kode;

***1-INSERT**-kommandoen er den kommando, vi bruger til at indtaste ny data i tabellen.*

I MYSQL bruger vi **INSERT INTO**-kommandoen til at tilføje data til en tabel. Vi har brugt denne kommando i vores SQL-kode som i kan se på det nedenstående afsnit.

6.3.2.1 Insert-eksempel

Feltnavne i tabellen skrives i rækkefølge efter kommandoen "INSERT **INTO TABLE**" med et kommategn mellem dem. Derefter bruger vi et andet kommando ”VALUES” for at indsætte vores værdier i vores tabel.

INSERT INTO motorhome (motorhome_id,brand,model,price,type,persons) VALUES (1," Coachmen RV ","2013","5528","F",5)

2-SELECT-kommandoen er den kommando, vi bruger til at indlæse det vi indtaster.
Med denne kommando kan vi vælge den tabel, vi vil behandle tabellerne fra vores MySQL-database. Hvis man for eksempel kun ønsker address_id i customer_address tabellen kan man gøre således:

6.3.2.2 Select-eksempel

Eksempel: **SELECT** * **FROM** customer_address **WHERE** address_id

SELECT *FROM viser alle data i tabellen.

3-UPDATE kommandoen bruger vi til at opdatere tabellen.

Denne kommando bruges til at erstatte information, der allerede findes i tabellen.

6.3.2.3 Update-eksempel

Eksempel: **UPDATE** booking **SET** motorhome_id=19 where booking_id=’10’

Hvad der er skrevet med store og fede bogstaver i eksemplet, er vores hovedkommandostruktur. Det anføres, at vi har ændret værdien af det afsnit, der hedder motorhome_id i bookingtabellen til 10.

4-WHERE-kommandoen er en kommando, der tillader, hvor ændringen skal foretages. (**Se også i kode 6.3.4.4).**

5-DELETE: Denne kommando bruges til at slette oplysninger, der allerede findes.

6.3.2.4 Delete- og Where-Eksempel

DELETE FROM customer **WHERE** custumer_id=’11’

Her er **DELETE** vores hovedkommando, **WHERE** er tabeloplysningerne, der skal slettes. Med andre ord fjernes linjen, der hedder customer_id som er 11 i tabellen.7

6.3.3 DDL

DDL (Data Definition Language) bruges til at definere, tilføje, slette tabeller og kolonner i databasen. DDL-kommandoer;

CREATE opretter et objekt i databasen.

DROP sletter objekter fra DROP-databasen.

6.3.5.1 CREATE-eksempel

```
CREATE TABLE motorhome (

motorhome_id int not null auto_increment PRIMARY KEY,

brand varchar (30),

model varchar (30),

price double,

type varchar (30),

persons int

);
```

For at kunne oprette en ny tabel, skal man navngive den. Definerer tabellens kolonner og hver kolonnes datatype. Efter CREATE TABLE-sætningen navngiver vi tabellen. I dette tilfælde har vi navngivet tabellen motorhome. Motorhome_id som en primær nøgle (primary key) og NOT NULL fortæller os at disse felter ikke kan være NULL, når der oprettes data i denne tabel. Årsagen til at vi definerer motorhome_id AUTO_INCREMENT er at den kan bruges til at generere en unik identitet til nye rækker. Så for hver gang man opretter en ny kolonne, vil motorhome_id stige inkrementelt. Derefter brugte INT (heltal) som datatype.

Varchar som er en række af karakter der har en længde på maksimum 255 karakter. Vi har valgt og sætte den til 30 karakter i vores tabel.

Double som kan indeholde decimaltal i sig.

DROP-kommandoen bruges til at slette en tabel eller database. Omhyggelig håndtering kan være påkrævet, når du bruger denne kommando. Ellers kan din database eller tabeller muligvis forsvinde på én gang, og det er muligvis ikke muligt at returnere dem.

7 Konklusion

Ud fra vores arbejde med denne opgave, kan vi konkludere at Motorhome Nordic vil klare sig bedre med en portal for sine medarbejdere, denne portal er blevet udarbejdet igennem forskellige programmer, såsom IntelliJ Idea, MySQL, GearHost og HerokuApp.

Vi har igennem hele projektet anvendt Unified Process modellen for at kunne skabe de bedste rammer for projektet. Vi har i Inception fasen samlede alle de relevante oplysninger og krav, som skulle til for at projektet skal være en succes.
I Elaboration fasen begyndte gruppen at designe systemet ved hjælp af de forskellige UML-diagrammer for at få et bedre overblik over hvordan systemet skal udvikles. Vi har derudover også designet databasen ved hjælp af ER-diagrammet og normaliseringsprocessen.
I vores konstruktion fase var der primær fokus på at udvikle systemet, da designet var klar til at blive udviklet. Her brugte vi værktøjer som Intellij Idea som har til formål at udarbejde Java, Spring, HTML og css kode, samt oprette en forbindelse med vores MySQL database. GearHost har givet os mulighed for at oprette en server som bliver forbundet med vores system, som kommer til at indeholde vores database.
Transition fasen var til at udarbejde en guide til hvordan vi anvender produktet. Her lavede vi også nogle test for at se om systemet virkede optimalt. Vi vil også forsøge at få feedback fra virksomheden, så systemet kan forbedres efter kundens behov.

De udfordringer man har stødt på igennem projektets forløb, er blandt andet at sikkerheden spiller en meget stor faktor i et projekt af dette omfang, så vi har altid prøvet at opgradere sikkerheden, så sikkerheden er i top, og møder kundens behov.

Derfor kan vi konkludere at, ved hjælp af denne portal der er blevet udarbejdet, så vil ansatte i firmaet bestemt se en forskel fra den daglige arbejdsdag med papirarbejde til en portal der understøtter deres behov. Disse behov vil gøre det muligt for den enkelte ansat at få et større overblik over blandt andet bookinger samt kunder.

8 Applikation og kode

Vi er nu færdige med vores projekt, og har udviklet den endelig hjemmeside til kunden. Vi deler hermed vores link til Github og hjemmesiden.

8.1 Hjemmeside:

Link: *http://motorhomenordic.herokuapp.com/*

Brugernavn:

- kea20

Kode:

- kea20

8.2 Github-link

Link: https://github.com/HolySupremeMaster/motorhome

9 Litteraturliste

Bøger

Craig Larman, Applying UML and patterns,

Hjemmesider

https://www.guru99.com/er-diagram-tutorial-dbms.html

https://balslev.io/programmering/database/normalisering-af-databaser/

https://www.studytonight.com/dbms/second-normal-form.php

https://www.studytonight.com/dbms/first-normal-form.php

https://balslev.io/programmering/database/normalisering-af-databaser/

Understanding Spring (2) – Read-Only-pdf Fronter[PDF anvendt fra Fronter - brugerlogin]