

Nordic Motorhome Project

2. semester Interdisciplinary Exam Project

KEA Computer Science

Uddannelsessted: Københavns Erhvervsakademi – Datamatiker

Dato: 4 juni 2020

Studerende: Jonathan Kasper Clement Iversen

Underskrift: _____

Studerende: Jonas Mørkhøj Hansen

Underskrift: _____

Studerende: Jacob Ravn

Underskrift: _____

Studerende: Tobias Salling Jensen

Underskrift: _____

Indhold

Github:	5
Problemformulering (Alle)	5
Samarbejdsaftale (Alle)	6
Kørselsvejledning (Jonas):	7
ITO	7
Feasibility Study (Alle)	7
Risk Analysis (Alle)	9
SWOT Analysis for Nordic Motorhome (Alle)	9
Stakeholder Analysis (Alle)	9
Software design	11
Phase Plan (Alle)	11
Unified process (Alle)	11
Supplementary Specification (Non-functional requirements) (Alle)	12
Use Case Diagram (Alle)	12
Use Cases	13
Brief (Use Case: Different Drop-off) (Alle)	13
Brief (Use Case: View Details) (Alle)	13
Casual (Use Case: Register Pick-up) (Alle)	14
Fully Dressed (Use Case: Rent Motorhome) (Alle)	14
Fully Dressed (Use Case: Cancel Motorhome) (Alle)	15
Fully Dressed (Use Case: Return Motorhome) (Alle)	17
Conceptual Model (Alle)	18
System Sequence Diagram (Use Case: Rent Motorhome) (Alle)	19
Sequence Diagram (Use Case: Rent Motorhome) (Alle)	20
System Sequence Diagram (Use Case: Return Motorhome) (Alle)	21
Sequence Diagram (Use Case: Return Motorhome) (Alle)	22
Design Class Diagram (før vi begyndte at kode) (Alle)	23
Design Class Diagram (efter vi var færdige med at kode) (Jonas og Jonathan)	24
GRASP Responsibilities	25
Polymorphism (Jonathan)	25
Controller (Jacob)	25
Creator (Jacob)	26

Software Construction	28
HashMap (Jonathan)	28
Deprecated getMonth() (Jonas)	28
HTTP error handling (Jonathan).....	29
Filter ift. Forskellige parametre (Jonathan)	30
Valide Data Input (Jonathan)	31
Fragments (Jacob).....	32
Dropdown menu (Jacob)	32
HTML, CSS.....	34
Input felter (HTML), CSS connection med fontawesome (Tobias)	34
Test (Jacob).....	34
CustomerRepository test:.....	35
Test af motorhome filter:	35
ActiveMotorhomeRepository test:	36
DatabaseConnectionManager test:	36
Konklusion af test.....	36
MySQL:	37
Beskrivelse af ER diagram og tabeller:	37
Motorhomes (Jacob):	37
Custusemotor (Jacob):	37
Customers (Jacob):.....	37
Motorhometype (Jacob):.....	38
Damages (Jacob):	38
Season (Jacob):	38
ER diagram (Jacob og Jonathan):	38
Normalformer (Alle):.....	39
Normalform 1:	39
Normalform 2:	40
Normalform 3:	40
SQL dokumentation (Alle):	40
Prepared statement:	40
Create og drop (DDL):.....	41
Databasens integritet:	42

On delete/update no action:	43
DML:	43
Insert:	43
Update:.....	44
Delete:	44
DQL:.....	44
Konklusion (Alle):.....	45
Kildeliste:.....	46
Bilag	47
Design Class Diagram (bilag 1).....	47

Github:

<https://github.com/Nicklas-homies/Nordic-Motorhome-project>

Problemformulering (Alle)

Nordic Motorhome er en succesfuld virksomhed med stigende vækst, som er stiftet i 2019. Nordic Motorhome ønsker derfor en webapplikation der kan holde styr på deres information omkring deres motorhomes, se udlejninger og se deres nuværende kunder i en samlet database, med tilhørende interface som køres lokalt via en browser. Nordic Motorhome vil gerne følge med tiden og have lavet en denne PaaS webapplikation, applikationen skal kunne køres lokalt på en computer, som de ansatte hos Nordic Motorhome skal have adgang til. Det er denne applikation som den følgende rapport med dokumentation vil omhandle. Projektet vil foregå som et udviklingsprojekt for softwareudviklere.

Samarbejdsaftale (Alle)

I forbindelse med vores første års eksamensprojekt vedrørende case om Nordic Motorhome Project, har vi udarbejdet en samarbejdsaftale for bedre at kunne administrere vores gruppearbejde.

Reglerne er som følgende:

§ 1 Jonas fungerer som projektorganisator, alt beslutningsmæssigt fastlås af projektorganisatoren.

§ 2 Hvis der skulle være uenigheder, henvises der til § 1.

§ 3 Der bliver taget udgangspunkt i faseplanen, i løbet af projekt arbejdet.

Stk. 1. Hvis en milepæl ikke bliver nået, vil dette skulle indhentes i næste fase.

§ 4 Der vil være minimum 2 obligatoriske samlinger hver uge, hvor der bliver arbejdet sammen.

Stk. 1. Disse samlinger kan foregå hjemmefra, ved brug af discord. Grundet nuværende Covid-19 situation.

Jacob Ravn

Jonathan Iversen

Jonas Hansen

Tobias Salling

Jacob Ravn

Jonathan Iversen

Jonas Hansen

Tobias Salling

Kørselsvejledning (Jonas):

Softwaremæssige forudsætninger for at kunne køre applikationen:

- Have MySQL Workbench installeret
- IntelliJ installeret.

Brugeren starter med at åbne både IntelliJ og MySQL workbench på deres computer. Først skal source koden åbnes i IntelliJ, det kan ske enten ved at clone det via vores github link¹ eller downloades som ZIP fil.

Når koden er åbnet er det vigtigt at tjekke "application.properties" som findes under resources i projekt overview.

```
db.url=jdbc:mysql://localhost:3306/NMH_company  
db.user=rootTest  
db.password=root
```

Disse oplysninger skal stemme overens med brugerens Database oplysninger, som kan findes i SQL Workbench. Hvis disse oplysninger ikke står korrekt, vil det ikke være muligt at forbinde til databasen. Bemærk dette er kun nødvendigt, hvis brugeren kører databasen lokalt.

Det er muligt at forbinde databasen til en online server således, at man ikke behøver at åbne SQL workbench, når man skal benytte databasen. Dette kan gøres ved at forbinde sin database til f.eks. GearHost², så databasen ligger på en server online.

Vores database kører lokalt, så når brugeren har ændret at application.properties stemmer overens med sine SQL oplysninger, så skal brugeren åbne vores SQL dokument "Create DB and Tables" og execute det (knappen der ligner et lyn). Brugeren skal nu køre programmet i IntelliJ.

Dette gøres ved at finde "NMHprojectApplication" i IntelliJ source kode src-main-java-com.nmh.project, hvor man derefter kører applikation. En succesfuld kørsel af applikationen vil kunne ses ved følgende udskrift kommer i run: *Tomcat started on port(s): 8080 (http)* (det er muligt at ændre til hvilken port man vil køre på).

Åben en browser og indtast "localhost:8080" som URL. Dette vil føre brugeren til startside af vores applikation.

ITO

Feasibility Study (Alle)

Hvis man kigger på om vores projekt er teknisk feasible, så kan vi hurtigt konkludere at det ikke er et særligt stort produkt, dog kan visse dele være en smule komplicerede. Dette betyder at vi

¹ <https://github.com/Nicklas-homies/Nordic-Motorhome-project>

² <https://www.gearhost.com/>

teknisk set ikke burde støde ind i nogle problemer. Da vi allerede har arbejdet med java og Spring før og lavet applikationer der minder meget om, er det heller ikke et problem at producere produkter. Produktet kommer også til at være meget sikkert da den opgave vi har fået, ikke kræver at der er noget online, oveni dette arbejder vi også med SQL-databaser hvilket er en stabil måde at opbevare sin data uden at frygte tab.

Kigger man i stedet på om det er økonomisk feasible, så kan man igen hurtigt se at produktet er meget simpelt og det hurtigt kan produceres. I forhold til cost/benefit er produktet billigt at producere, billigt at vedligeholde og billigt at opretholde, dog skal man også tage højde for hvis applikationen skal rulles ud og ligge på en online server. Hvis man ønsker dette, er der andre økonomiske trin på vejen, dog ikke noget der springer banken. Derimod er det ikke en særligt kompleks applikation og hvis man beslutter sig for at man gerne vil have nye funktioner kan dette nemt implementeres.

I forhold til de operationelle krav der er til produktet, er de som sagt ikke særligt komplicerede og derfor burde der ikke være nogle problemer. Ledelsen er fuldt ombord på produktet og vi kan nemt gå i dialog angående mulige problemstilling der kunne opstå. Dette gælder også efter produktet originalt er blevet deployed hvis der skulle findes fejl. Dog forventer vi ikke at der kommer til at være problemer efter "launch" og vi forventer at det kommer til at løse det forretningsmæssige behov.

Den eneste åbne problemstilling for produktet er persondataloven, dog burde dette ikke være et problem da det hele foregår offline, vi skal dog også tage højde for hvis applikationen skal rulles ud til en online funktion. Vi har valgt at tage højde for dette ved at sikre vores database fra SQL injections og lignende. Dog skal man også tage forbehold til licenser og lignende, vi sælger webapplikationen og licensen til Nordic Motorhome og de får derefter ansvaret for vedligeholdelse og andre udgifter der kunne fremkomme. Dog fikser vi eventuelle bugs eller fejl der er hvis det er nødvendigt.

Vi foreslår som sædvanlig at checke om der allerede eksisterer applikationer der kunne løse de administrative problemer kunden har da det er et almindeligt problem, der bare skal specialiseres. Fordelene ved at de får lavet deres egen applikation er at det kommer til at være mere simpelt og derved mere brugervenligt.

For at konkludere til sidst om det er et go eller no go til applikationen vil vi nok foreslå kunden at undersøge om der ikke allerede eksisterer en applikation der lever op til deres behov. Det kommer til at være nemmere og hurtigere for dem at finde en applikation og nemmere på længere sigt i forhold til vedligeholdelse. Hvis de beslutter sig for at få lavet applikationen hos os vil det blive mere simpelt og brugervenligt i forhold til deres behov. Dog er produktet ellers muligt at lave hvis de ikke finder en gyldig applikation.

Risk Analysis (Alle)

Vi har som udviklingerne af vores applikation til Nordic Motorhome valgt at lave en risikoanalyse, det har vi gjort for at være så godt forberedte som muligt til at gribe vores opgave an og for at danne os en oversigt over de mulige risici der kunne være forbundet med projektet. Vi har sammen brainstormet og fundet frem til hvilke risikoer der kunne være forbundet med dette projekt, dette har vi gjort under kolonnen "*Risiko beskrivelse*", derefter har vi givet den givne risiko en sandsynlighed fra værdien 1 – 5 og en konsekvens i værdierne 1, 3, 7, 10 (1 ubetydelig – 10 katastrofal). Sandsynligheden ganger vi med vores konsekvens og får derved et produkt. Herefter beskriver vi vores mulige præventive løsninger til risikoen, hvem der er ansvarlig, løsningen og de ansvarlige for risikoen.

Risiko beskrivelse	Sandsynlighed	Konsekvenser	Produkt	Mulige præventive løsninger	Ansvarlig	Løsning	Ansvarlig
Applikationen crasher	1	3	3	flere afprøvninger af programmet	projekt deltagere	genstart serveren hvor applikationen kører	bruger
Fil tab ved at der er noget der går i stykker	1	7	7	back-up af databasen til en sekundær database	projekt deltagere	hent data fra back-up	projekt deltagere
Mangelfulde afprøvninger	2	3	6	planlæg flere diverse afprøvninger af programmet	projekt deltagere	gør mere brug af unit tests	projekt deltagere
Ændring i kravene under udvikling	1	7	7	gennemgå designdelen af projektet grundigt	projekt deltagere	ændre projektet	projekt deltagere
Hvis man ikke opnår de milepæle der er sat	2	3	6	holde sig til faseplanen	projekt deltagere	mere arbejde den følgende uge	projekt deltagere

SWOT Analysis for Nordic Motorhome (Alle)

I vores gruppe har vi valgt at lave en SWOT-analyse for virksomheden Nordic Motorhome. Dette har vi gjort for at få skabt et bedre overblik over deres virksomhed, samt en bedre forståelse over deres hvilke forskellige forretningsområder og stillinger de har i virksomheden. En SWOT-analyse hjælper også virksomheden med at vurdere et nyt produkt og i dette tilfælde har vi vurderet at Nordic Motorhomes svage side er tidsplanlægning, fordi de har meget papirarbejde og manuelt arbejde i at indtaste bookinger, kunder og registrer motorhomes. Derfor ville det give god mening at lave et program der kan sørge for dette. Under Nordic Motorhomes eksterne forhold har vi valgt at sige at de har mulighed for ekspansion, til at starte med i de skandinaviske lande og deraf vil vores program også være relativt let at opdatere på sproget og udenlandske valutaer.

Interne forhold	
Strengths	Weakness
Stort sortiment/udvalg af Motorhomes	Tidsplanlægning
Flere pick-up points	Risiko for centrale medarbejdere stopper
Bred målgruppe	
Eksterne forhold	
Opportunity	Threats
Blande flere produkter sammen	Konkurrenter
Mulighed for ekspansion	Stigende modstand på dieslbiler

Stakeholder Analysis (Alle)

En interessentanalyse er et godt værktøj til at få et overblik og beskrive de forskellige interessenter der kunne have en interesse i det projekt vi bygger til deres organisation. Derfor har vi valgt at lave en interessentanalyse over Nordic Motorhomes ledere, medarbejdere og deres kunder. Nordic Motorhomes ledere er en vigtig interessent i vores projekt da de netop har taget fat i os for at optimere deres virksomheds daglige arbejdsgang, mht. at gøre arbejdet mere

struktureret og overskueligt for deres medarbejdere. Nordic Motorhomes medarbejdere har vi også valgt at lave en interessentanalyse over, da det er dem som kommer til at arbejde mest med vores færdige produkt, her vil det primært være salgssassisterne og bookkeeper som kommer til at bruge vores produkt. Til sidst har vi valgt at lave en interessentanalyse over Nordic Motorhomes kunder, da de også vil være interesseret i en nem og praktisk fremgangsmåde hvor alle tingene forløber lige til ved udlejning af et motorhome.

Interessentanalyse for Nordic Motorhome lederne/ejerne:

Interessentanalyse NMH-Leder						
Interessent	Deres mål	Tidligere reaktion	Hvad der kan forventes	Indvirkning Pos/neg	Mulige fremtidig reaktion	ideer
NMH-leder	At lette papirarbejde for sine medarbejdere så de kan bruge deres tid mere effektivt	Implementerings problemer da medarbejdere er blevet vant til papirarbejde	Bliver meget negativ hvis produktet ikke følger sin produktionsplan	positiv kan være det er svære at dobbelt booke motorhomes eller lign. problemer. Negative kan ske hvis applikationen fungerer langsomt og ikke opfylder hans krav	positivt indstillet ift. lettere arbejdsryk	holde lederen up to date med udviklingen af applikationen, og løbende diskutere produktkrav

Interessentanalyse for Nordic Motorhome medarbejderne:

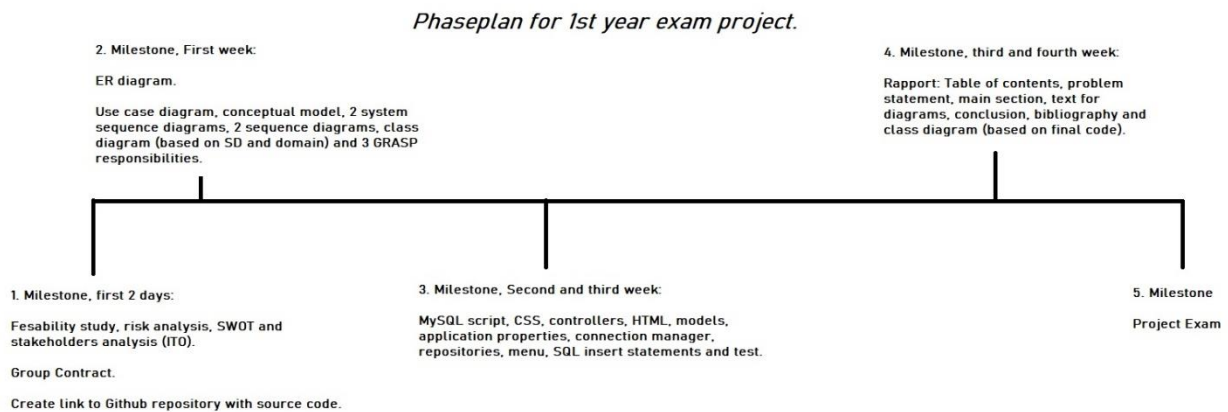
Interessentanalyse NMH-medarbejder						
Interessent	Deres mål	Tidligere reaktion	Hvad der kan forventes	Indvirkning Pos/neg	Mulige fremtidig reaktion	ideer
NMH-medarbejder	at få en nemmere hverdag der er mindre stresset og mere struktureret arbejdsform	skeptisk over for det nye system da de er vant til deres gamle metoder	Bliver meget negativ hvis produktet ikke er funktionelt eller langsomt	positiv kan være det er sværere for medarbejderne at lave fejl. Negative kan ske hvis applikationen fungerer langsomt og ikke opfylder deres krav	stadig skeptisk ift. til det nye system og problemer med om det er lige så nemt som papir	diskuter med medarbejderne hvordan de gør det nu og se om vi kan implementere nogle af de nuværende metoder

Interessentanalyse for Nordic Motorhomes kunder:

Interessentanalyse NMH-kunde						
Interessent	Deres mål	Tidligere reaktion	Hvad der kan forventes	Indvirkning Pos/neg	Mulige fremtidig reaktion	ideer
NMH-kunde	det skal være en let process at bestille et motorhome	papirsystem som blev brugt tidligere var rodet og langsomt	at applikationen virker	positiv kan være det er sværere for medarbejderne at lave fejl og derfor bliver det en bedre process for kunden. Negative kan ske hvis applikationen ikke fungerer	tilfredse med processen der er ved at leje et motorhome	lav en rundspørge blandt tidligere kunder og hør hvad deres forventninger er ift. teknologisk hjælp ved udlejning

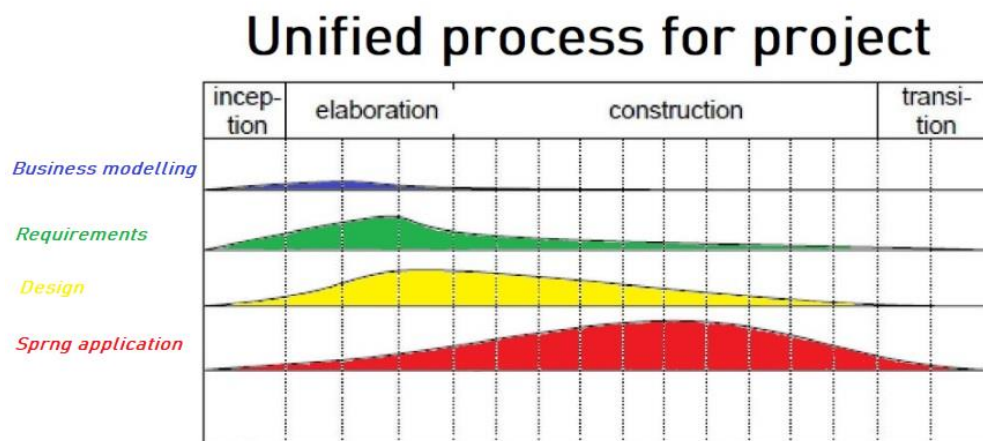
Software design

Phase Plan (Alle)



Unified process (Alle)

Vi har lavet vores Unified Process i starten af vores projekt, lige efter vores faseplan. Unified Process er en model som giver et godt overblik over hvordan vi kommer til at gå igennem de fire faser, Inception, Elaboration, Construction og Transition. Forberedelsen (Inception) er den fase der giver os det fulde overblik over hvilke krav vores system skal have implementeret. Etableringsfasen (Elaboration) er der hvor vi har fokus på at udvikle de centrale dele af systemet, vi ser herunder en stor del på vores requirements til Nordic Motorhomes krav til vores projekt. Under Elaboration kigger vi også på design, hvor vi udvikler Use Cases og diagrammer indenfor SWD. I konstruktionsfasen (Construction) udvikler vi på vores kode i Java, MySQL, HTML og CSS. I vores overdragelse af programmet (Transition) bliver de sidste små rettelser i koden lavet og programmet overdrages til Nordic Motorhome, i form af projektfremlæggelse og rapport.



Supplementary Specification (Non-functional requirements) (Alle)

Usability:

Vores program skal være let at bruge, man skal ikke igennem en lang proces for at udføre trivielle funktioner, som at oprette en lejeaftale. Det skal derfor også være nemt at tilgå redigering af aktive motorhomes, tilføjelser af nye motorhomes når firmaet udvider eller lignende. Det er en del af projekt at det er skalerbart og derfor skal det også være nemt at udvide det. Her kunne man tænke at Nordic Motorhome kunne overveje at udvide til andre skandinaviske lande og dertil vil det være nemt at implementere programmet så det er tilpasset andre skandinaviske lande. Nordic Motorhome kunne også overveje at lave en portal til deres kunder, så de selv ville kunne leje et motorhome. Dette vil være en større udviklingsfase, men bestemt muligt da mange af funktionerne og metoderne vil blive brugt igen.

Reliability:

Applikationen kommer til at læne sig op ad en SQL server ift. at gemme sine motorhome og kunde data, i sammenhæng med SQL sikrer vi os også mod SQL injections og andre problemer SQL kan medføre.

Performance:

Applikationen skal være skalerbar og skal derfor være brugbar på et større plan så vel som et småt, der er derfor taget højde for memory management.

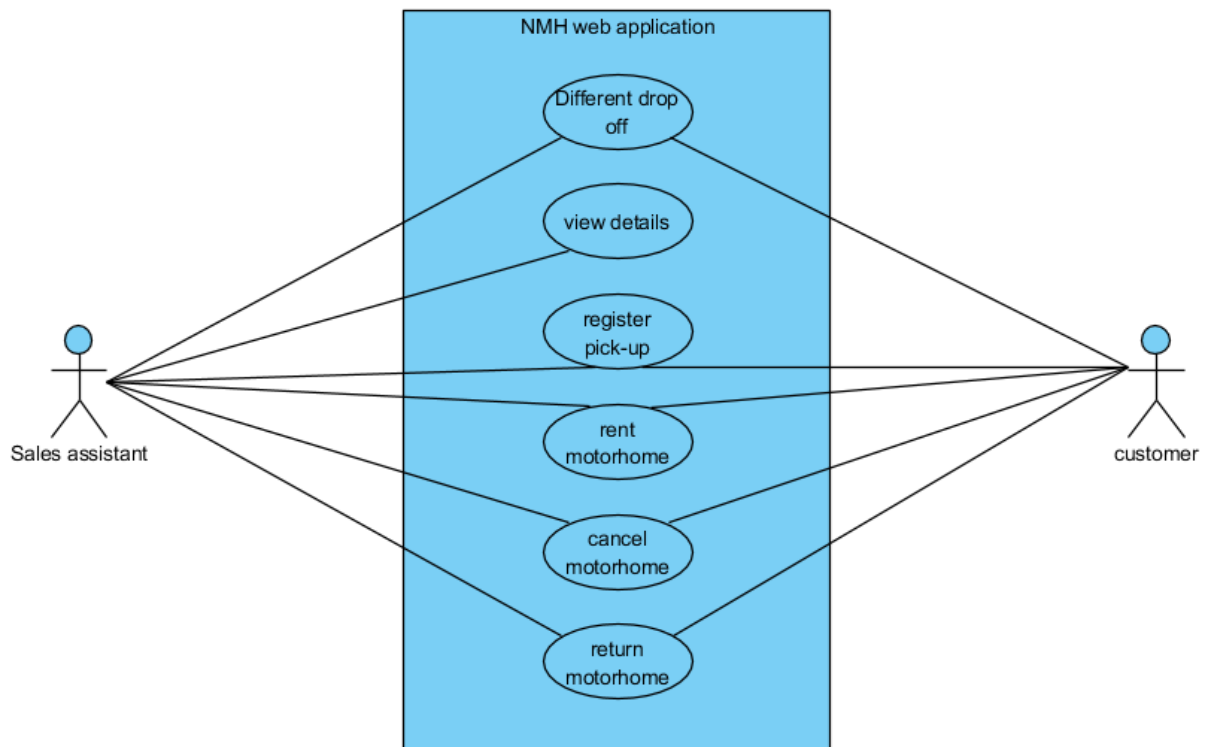
Supportability:

Applikationen skal være nem at ændre uden der skal redigeres i mange forskellige klasser. Oven i dette skal det også være nemt at tilføje funktioner, så det tager vi også højde for.

Use Case Diagram (Alle)

(billede på næste side)

Vores Use Case diagram er et simpelt diagram som viser brugernes interaktion med vores system. I Use Case diagrammet har vi derfor vores "Actor" til venstre for diagrammet, som i dette tilfælde er Nordic Motorhomes Salgs assistent. Vores salgsassistent har associationer til alle vores Use Cases, da de kommer til at have en indflydelse igennem hele vores program. På højre side af vores diagram er Nordic Motorhomes kunder (customer) som vi har valgt at associationer til nogle af vores Use Cases, da de har indflydelse på at den specifikke Use Case kan fungere.



Use Cases

Brief (Use Case: Different Drop-off) (Alle)

Salgs Assistenten åbner programmet og vælger fra Homepage "*Hand In Motorhome*". Derefter vælger medarbejderen "*Edit details about drop-off*" for det gældende Motorhome. Sales Assistant ændrer "*Drop-off location*" til den nye lokation og trykker derefter "*Submit*"

Brief (Use Case: View Details) (Alle)

Salgs Assistenten åbner programmet og vælger fra Homepage "*Rent Motorhome*" og trykker "*See Details*" på det specifikke Motorhome. Salgs Assistenten tjekker information på Motorhomet og lukker derefter programmet.

Casual (Use Case: Register Pick-up) (Alle)

Main success scenario: Salgs Assistenten åbner programmet og vælger fra Homepage "*Register Pick-up*" vælger det Motorhome som kunden ønsker. Salgs Assistenten indtaster derefter nødvendige data på kunden og bliver derefter promptet med "*Correct Motorhome*" og "*Correct Data*". Hvis data er korrekt oprettes en pick-up til kunden.

Alternative scenarios:

- a) Hvis den indtastede data er forkert.
 - a.1) Salgs Assistenten redigerer dataene fra "*waiting for Pick-up*" listen.
 - a.2) Akcepterer data er korrekt og gemmer.
- b) Hvis der efter Pick-up, bliver fundet fejl i indtastningen af Pick-up.
 - b.1) Salgs Assistenten vælger "*See active Motorhomes*".
 - b.2) Redigerer data for gældende motorhome og gemmer.

Fully Dressed (Use Case: Rent Motorhome) (Alle)

Use case: Rent Motorhome

Scope: Nordic Motorhome Webapplikation.

Level: User goal.

Primary actor: Salgsassistenten.

Stakeholders and interest:

- Salgsassistent: Ønsker at programmet er let anvendeligt, til brug ved udlejning, afslut leje og indtastning af data.
- Ejer: Ønsker at programmet virker nemt og ligetil for sine ansatte, som kommer til at bruge programmet. Ønsker også at der ikke er fejl der leder til problemer ved Motorhomes når de skal afleveres eller data skal ændres.
- Kunder: Ønsker nem proces ved registrering af deres lejeperiode.

Preconditions: Salgsassistenten og kunde har aftalt at der skal lejes et eller flere motorhomes.

Success guarantee (postcondition): Registreringen af leje af motorhome er oprettet og kunden har fået prisen.

Main success scenario (basic flow):

- 1) Ansat modtager lejeansøgning og skal oprette ny lejeperiode for motorhome.

- 2) Vælger fra homepage "*Rent Motorhome*".
- 3) Ansæt indtaster gældende dato for leje.
- 4) Vælger et af de tilgængelige motorhomes, efter kundens behov.
- 5) Indtaster kundens data og andre kontraktmæssige forhold.
- 6) Godkender indtastet oplysninger.
- 7) Den ansatte får prisen og videregiver den til kunden.

Extension (alternative flow):

6.a) Skriver forkert information.

a.1) Vælger "*See active motorhomes*".

a.2) Vælger motorhome der skal redigeres.

a.3) Retter sin fejl under valgte motorhome.

Special requirements:

- Salgsassistenten skal bruge en pc.
- En database over motorhomes, kan købes som server hos Nordic Motorhome eller hostes i skyen.

Frequency of occurrence:

- Hver gang en ny anmodning om udlejning af motorhome finder sted.

Open issues:

- GDPR.
- Der kan kun vælges motorhome ud fra dato, og ikke ud fra motorhome type.

Fully Dressed (Use Case: Cancel Motorhome) (Alle)

Use case: Cancel Motorhome.

Scope: Nordic Motorhome Webapplikation.

Level: User goal.

Primary actor: Salgsassistent.

Stakeholders and interest:

- Salgsassistent: Ønsker at programmet er let anvendeligt, til brug ved udlejning, afslut leje og indtastning af data.
- Ejer: Ønsker at programmet virker nemt og ligetil for sine ansatte, som kommer til at bruge programmet. Ønsker også at der ikke er fejl der leder til problemer ved Motorhomes når de skal afleveres eller data skal ændres.

- Kunder: Ønsker nem proces ved aflysning af deres lejeperiode.

Preconditions: Salgs Assistent og kunde har aftalt at der skal aflyses et eller flere motorhomes og der er oprettet en lejekontrakt.

Success guarantee (postcondition): registreringen af aflysning af motorhome er gennemført og kunden har fået en pris.

Main success scenario (basic flow):

- 1) Ansæt modtager aflysnings anmodning og skal aflyse lejeperiode for motorhome.
- 2) Vælger fra homepage "*See active Motorhome*".
- 3) Vælger det gældende motorhome.
- 4) Trykker "*Cancel Motorhome*".
- 5) Bliver promptet og accepterer.
- 6) Den ansatte får pris for aflysning og videregiver information.

Extension (alternative flow):

3.a) Hvis Salgs Assistenten vælger et forkert Motorhome.

- a.1) Annullerer og går tilbage.
- a.2) Vælger det korrekte motorhome og aflyser.

5.b) Aflyser forkert motorhome.

- b.1) Vælger "*Rent Motorhome*" fra homepage.
- b.2) Vælger "*Create from archive*".
- b.3) Vælger et gældende motorhome fra liste og opretter igen.
- b.4) Går til "*Cancel Motorhome*" siden og aflyser det rigtige motorhome.?????

Special requirements:

- Salgs Assistenten skal bruge en pc.
- En database over motorhomes.

Frequency of occurrence:

- Hver gang en ny anmodning kommer ind.

Open issues:

- GDPR.

Fully Dressed (Use Case: Return Motorhome) (Alle)

Use case: Return Motorhome.

Scope: Nordic Motorhome Webapplikation.

Level: User goal.

Primary actor: Salgs Assistenten.

Stakeholders and interest:

- Salgs Assistent: Ønsker let brug af program, ved indtastning af data.
- Ejer: Ønsker at webapplikationen virker nemt og let. Ønsker også at der ikke er fejl der leder til problemer ved Motorhomes når de skal afleveres eller data skal ændres.
- Kunder: Ønsker nem proces ved aflysning af lejeperiode.

Preconditions: Lejekontrakt er udløbet og kunden har afleveret motorhomet

Success guarantee (postcondition): motorhome er afleveret og kunden har fået prisen

Main success scenario (basic flow):

- 1) Vælger fra homepage "*See active Motorhome*".
- 2) Vælger det gældende motorhome.
- 3) Trykker "*Return Motorhome*".
- 4) Indtaster kørte kilometer, tank status og om der er nogle skader.
- 5) Bliver promptet og accepterer.
- 6) Den ansatte får pris for lejekontrakten og står for indkrævning.

Extension (alternative flow):

2.a) Vælger forkert motorhome.

a.1) Annullerer og går tilbage.

a.2) Vælger det korrekte motorhome og returnerer.

5.b) Indtastet data forkert.

b.1) trykker nej ved prompt og indtaster korrekt data.

5.c) Returnerer forkert motorhome.

c.1) Vælger "*Rent Motorhome*" fra homepage.

c.2) Vælger "*Create from archive*".

c.3) Vælger et gældende motorhome fra liste og opretter igen.

c.4) Går til page til return siden og returnerer det rigtige motorhome.

Special requirements:

- Salgs Assistenten skal bruge en pc.
- En database over motorhomes.

Frequency of occurrence:

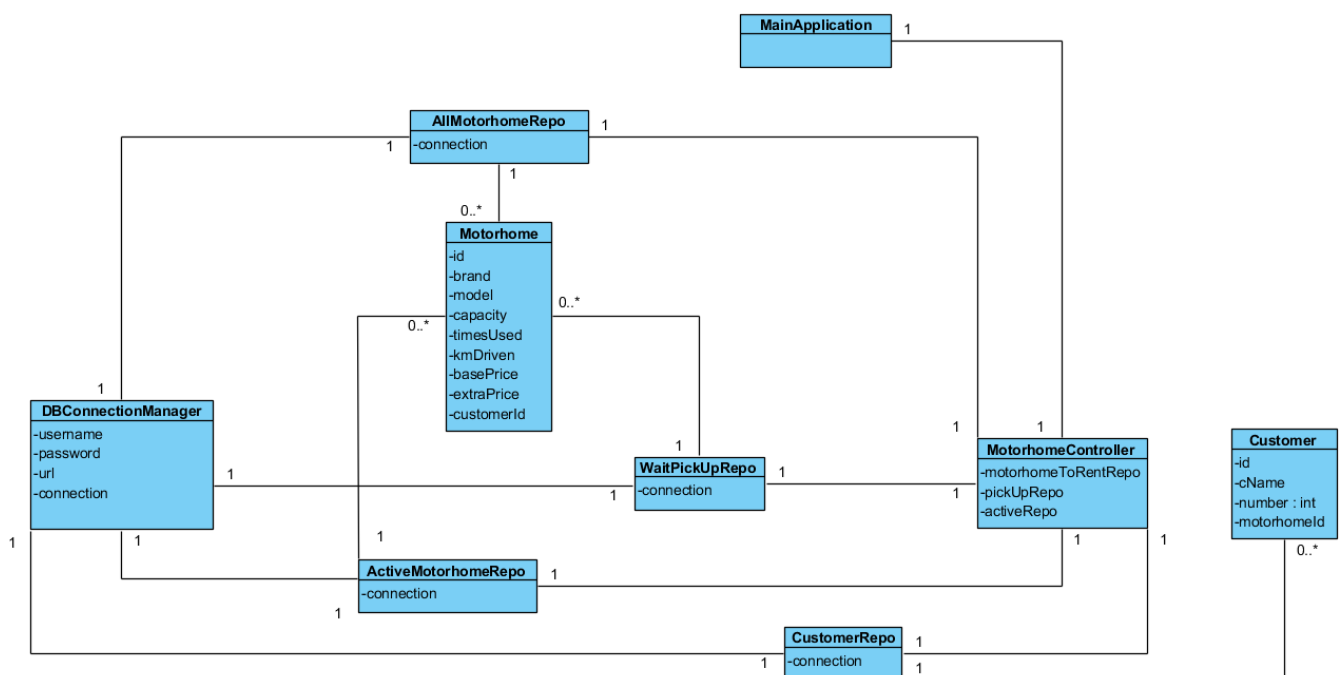
- Hver gang et nyt motorhome skal returneres.

Open issues:

- GDPR.

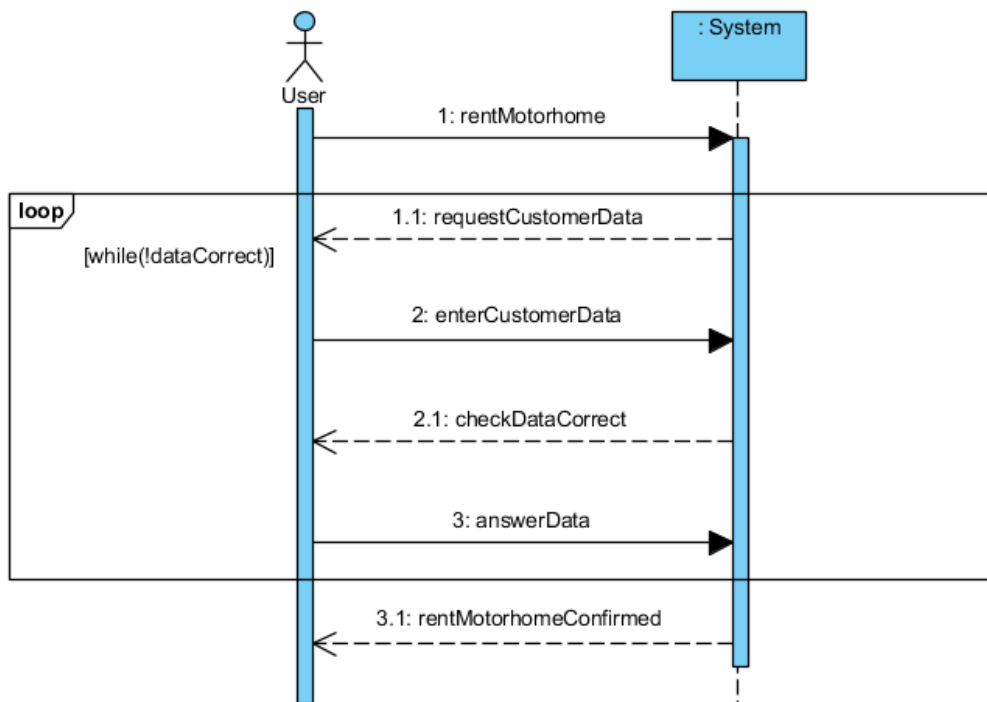
Conceptual Model (Alle)

Den konceptuel model repræsenterer vores system, som vi har tænkt os at lave til Nordic Motorhome. Den konceptuelle model giver andre personer en god viden og forståelse for hvordan vi vil opbygge vores program. Modellen indeholder vores controllere, models og repositories, som vi har tænkt os at udvikle i Java. Hver klasse indeholder både attributter og associationer til andre klasser. Denne model er lavet inden vi begyndte at skrive selve koden og vi har efterfølgende i vores kode tilføjet og ændret flere controllere, models og repositories, herunder bl.a. en CustomerController, SeasonRepository, Season class osv. Disse ville kunne ses i vores [Design Class Diagram](#).



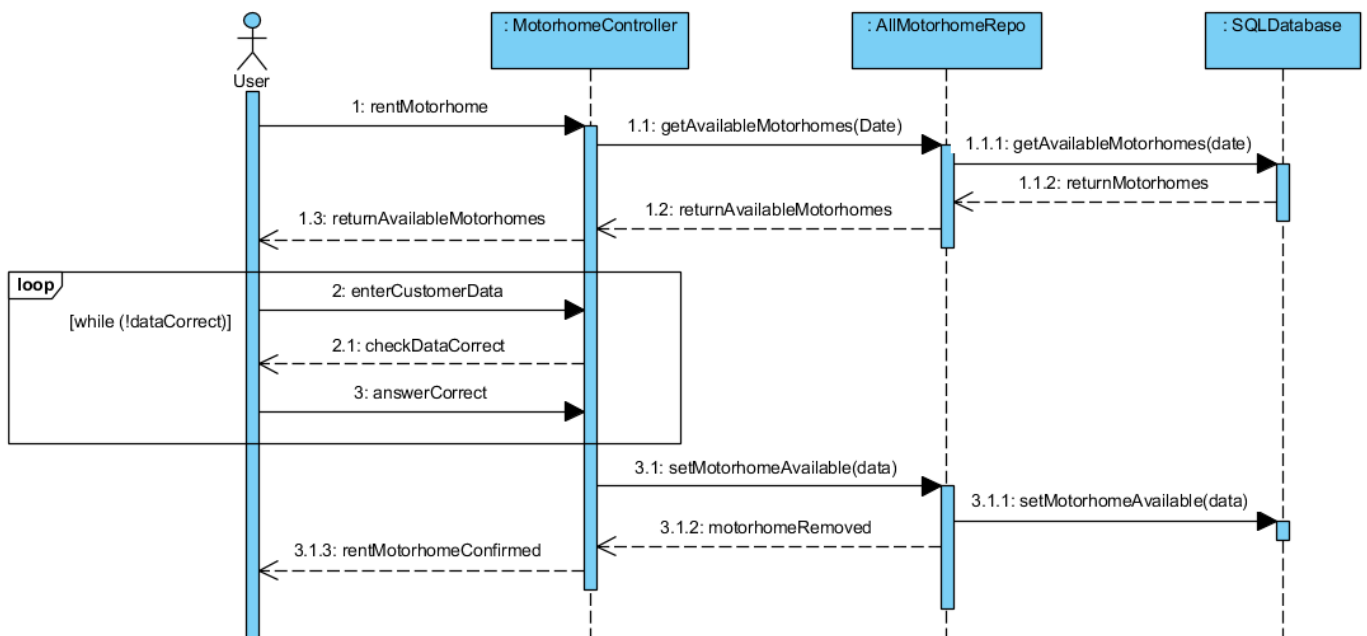
System Sequence Diagram (Use Case: Rent Motorhome) (Alle)

Vores System Sekvens Diagram er lavet over vores Use Case "*Rent Motorhome*". System Sekvens diagrammet viser en sekvens af et input og output mellem vores Actor og Systemet. Vores Actor starter med at klikke på en knap så vores *rentMotorhome* metode aktiveres. Derefter kører et loop indtil alt data er udfyldt korrekt. Når alt data er korrekt, så godkender vores Actor og systemet sender en besked med at udlejningen er foretaget.



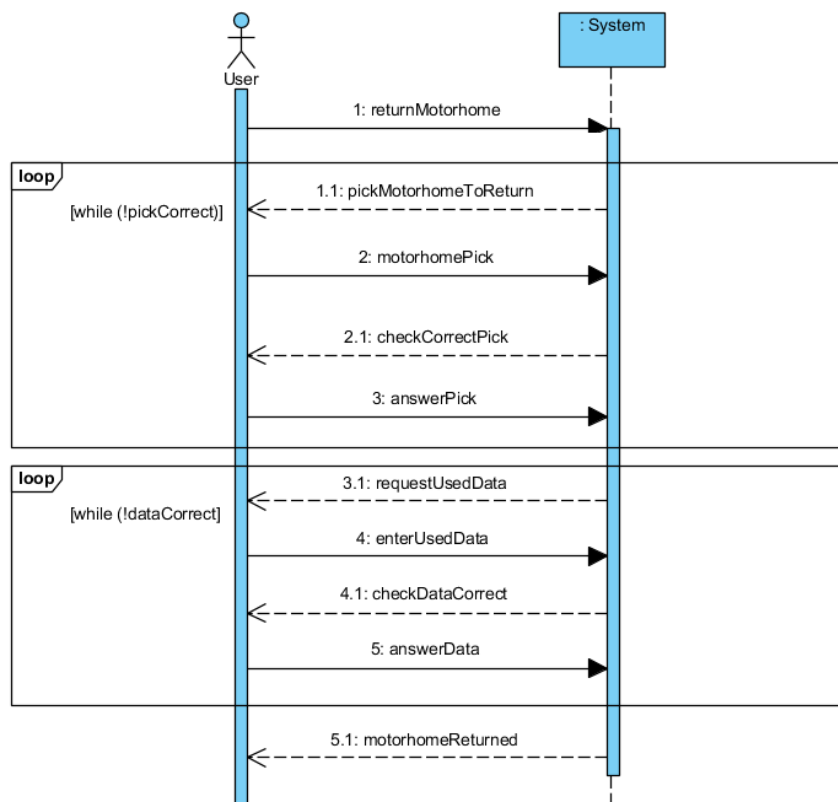
Sequence Diagram (Use Case: Rent Motorhome) (Alle)

Vores sekvens diagram har vi valgt at lave over Use Casen "Rent Motorhome", sekvensdiagrammet viser os interaktionerne mellem vores Actor og de tre klasser / objekter. I vores sekvens diagram har vi en *MotorhomeController*, *AllMotorhomeRepo* og en *SQLDatabase*. Vores Actor (User: Salgsassistent) ville i vores system vælge at leje et Motorhome for en kunde. *MotorhomeController* integrerer sammen med *AllMotorhomeRepo* for at få en liste over tilgængelige Motorhomes. *AllMotorhomeRepo* integrerer så sammen med vores *SQLDatabase*, som returner tilgængelige motorhomes. Salgsassistenten bliver herefter promptet for at indtaste lejerens personlige data korrekt, denne interaktion sker kun sammen med *MotorhomeController* klassen og interfacet. Hvis dataene er korrekte, så godkender salgsassistenten at ville leje det valgte Motorhome, og motorhomet bliver herefter fjernet fra listen over tilgængelige Motorhomes i *SQLDatabase*.



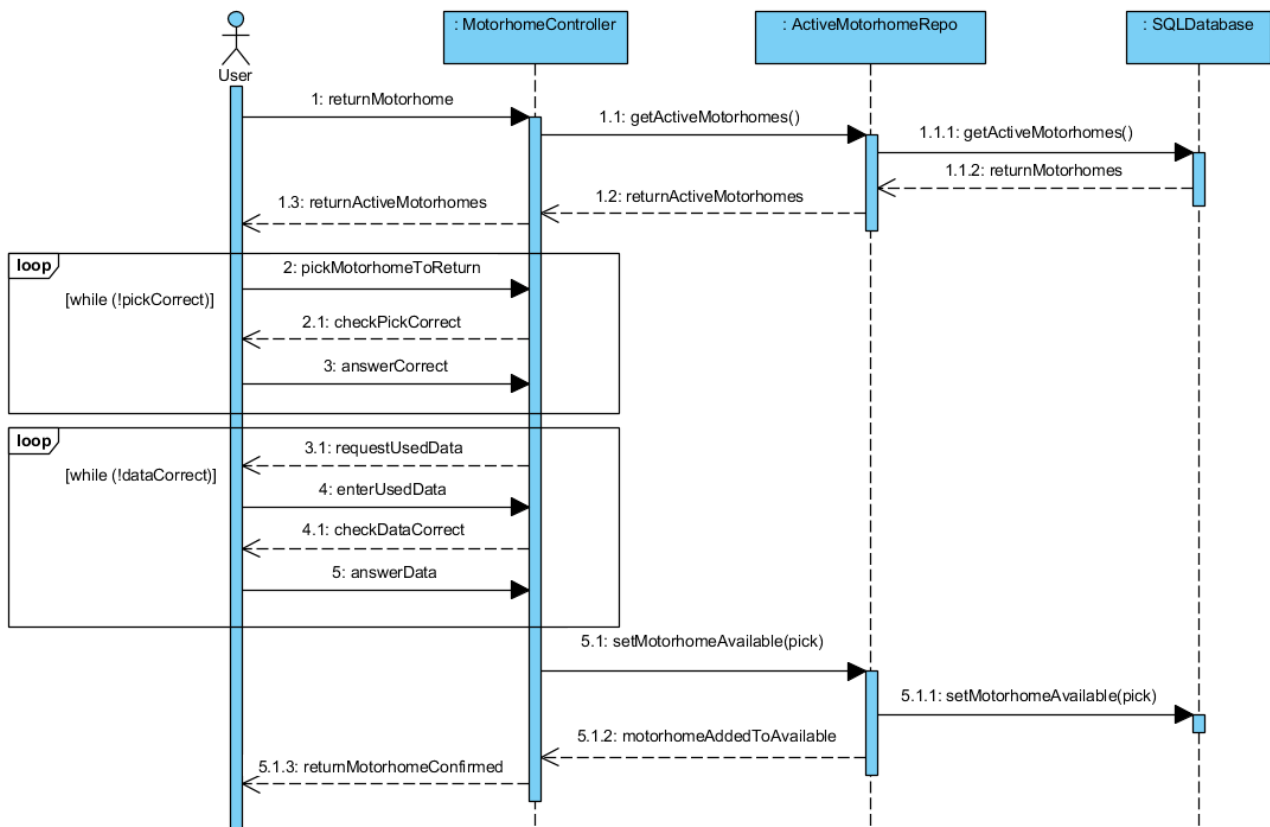
System Sequence Diagram (Use Case: Return Motorhome) (Alle)

Vores andet system sekvens diagram har vi valgt at lave over use casen *"Return Motorhome"*. Hvor vi har vores actor på venstre side og vores System på højre side. Vores actor (User: SalgsAssistent) salgsassistenten klikker i interfacet på knappen *"Return Motorhome"* så interaktionen med vores system begynder, vores system sender en besked tilbage til vores user omkring hvilket Motorhome som skal returneres. Sådan kører interaktionerne mellem vores user og system hele vejen igennem. Vi har også gjort brug af et loop, de steder hvor vi mener at dette er nødvendigt. Loopet kører indtil vores Actor har udfyldt alle nødvendige felter/data korrekt og derefter bliver det valgte motorhome returneret.



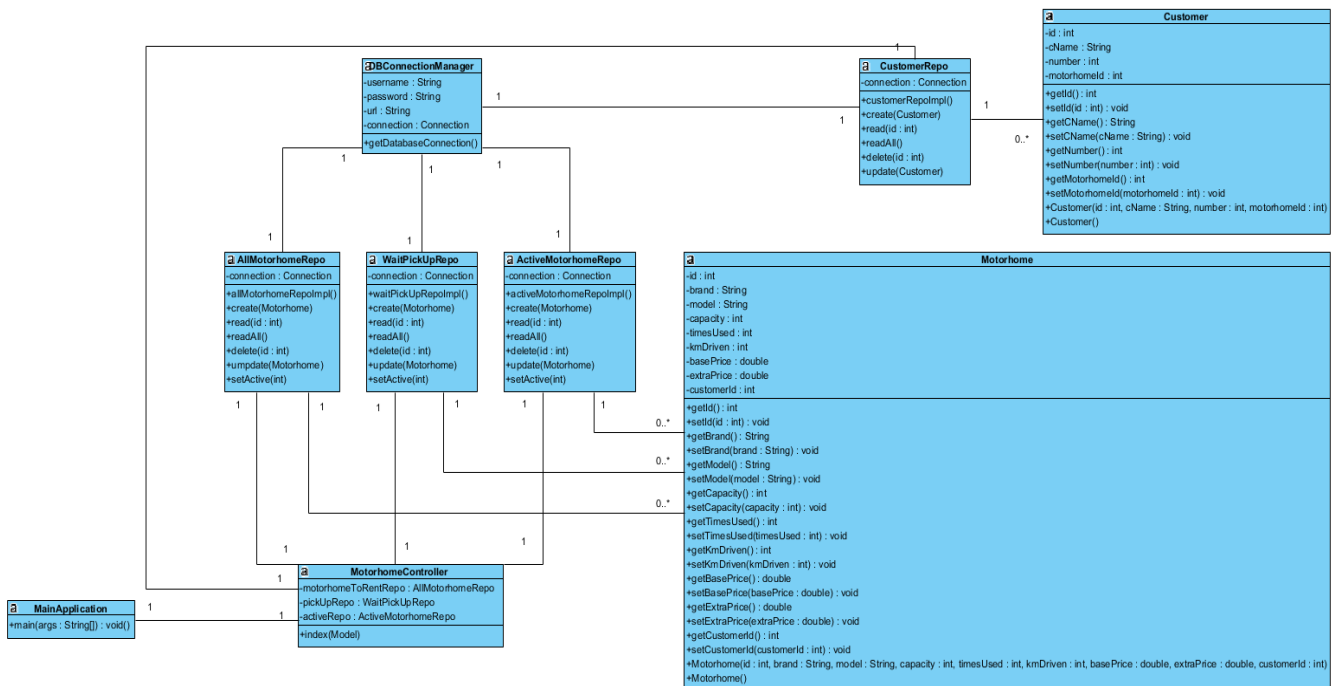
Sequence Diagram (Use Case: Return Motorhome) (Alle)

Vores andet sekvensdiagram har vi valgt at lave over "Return Motorhome". Vi har vores Actor ude til venstre i diagrammet, og klasserne *MotorhomeController*, *ActiveMotorhomeRepo* og databasen *SQL-database*. I første sekvens af diagrammet vælger vores Actor at trykke Return Motorhome i vores interface, som integrerer med vores *MotorhomeController* sender så metoden *getActiveMotorhomes()* til klassen *ActiveMotorhomeRepo*, som har en metode der integrere med vores *SQL-database*. Herefter får vores Actor en liste over aktive udlejet motorhomes. Så kører vi et loop indtil vores Actor har valgt det korrekte motorhome som skal returnere, et andet loop begynder herefter indtil vores Actor har udfyldt data på det returneret motorhome. Så bliver det valgte motorhome sat til at være ledigt, hvor der foregår en sekvens fra *MotorhomeController* til *ActiveMotorhomeRepo*, som så sender en metode til *SQL-databasen* og det valgte motorhome bliver sat som ledigt.



Design Class Diagram (før vi begyndte at kode) (Alle)

Nedenstående design klasse diagram repræsenterer vores system, som vi har tænkt os at lave til Nordic Motorhome. Diagrammet bruges til at give os et godt overblik og forståelse for hvordan vi vil opbygge vores program. Diagrammet indeholder vores controller og de basismodeller vi har tænkt os at lave i selve vores applikation. Man kan også se vores repositories som er en af de vigtigste dele af vores program. Hver klasse indeholder både attributter og metoder med hvad for nogle attributter der skal bruges. Udover dette er der også associationer til andre klasser. Mange af disse associationer er 1 til 1 hvilket indikerer at der kun skal eksistere en instans af det objekt for den gældende association. Dog kan man også se at vores to modeller Motorhome og Customer har 1 til 0..*, 0..* betyder at der kan eksistere alt fra 0 instanser af det objekt til mange, dette bruger vi da de repositories de er forbundet til har både kan slet og opret metoder. Dette diagram er lavet inden vi begyndte at skrive selve koden og vi har efterfølgende i vores kode ændret i stort set alle klasserne, her i blandt controllere, models og repositories. Vi er også blevet nødt til at tilføje mange nye klasser herunder bl.a. en CustomerController, SeasonRepository, Season class osv. Mange af disse ændringer sker som følge af GRASP responsibilities da der på daværende tidspunkt var et problem med bl.a. high coupling, vi kommer til at skrive mere om dette under vores GRASP-diagrammer.



Design Class Diagram (efter vi var færdige med at kode) (Jonas og Jonathan)

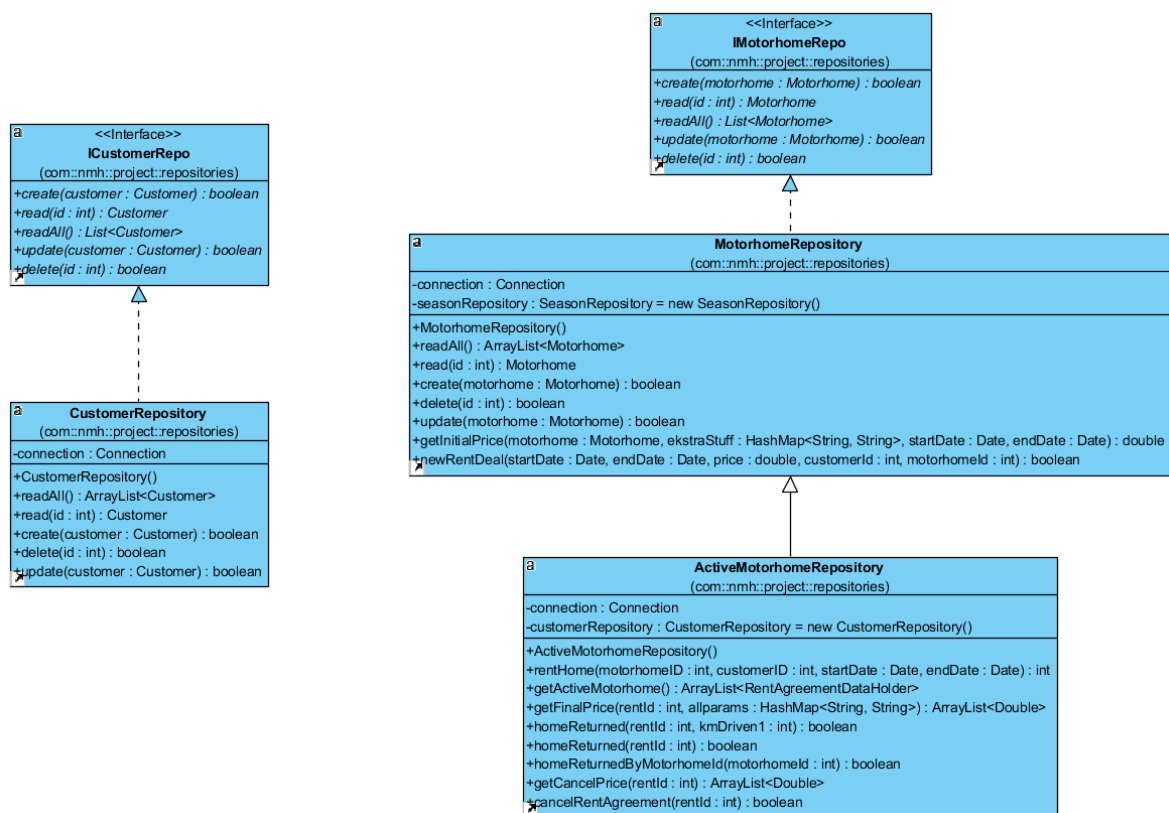
Grundet størrelsen på diagrammet er det vedlagt som bilag, under [Design Class Diagram \(bilag 1\)](#), det kommer også til at følge med som billede med afleveringen da det kan være svært at se i Word.

Diagrammet kan bruges til at give os og andre der skal se kode-delen af vores et godt overblik og forståelse for hvordan programmet er opbygget. Diagrammet indeholder vores fem controllere, der i det originale design klasse diagram kun blev vist som et. Det viser også vores to nye modeller Season og RentAgreementDataHolder, disse bruges henholdsvis til at fastlægge en pris ud fra sæson periode og til at opbevare midlertidige data for en lejeperiode. Med de nye modeller er vi også blevet nødt til at ændre og lave flere repositories, dette gjordes også for at holde fokus på GRASP. De nye repositories er DamageRepo, FilterRepo og SeasonRepo som bruges flere steder igennem vores pprogram. Sammenligner man igen de to klassediagrammer kan man også se at ActiveMotorhomeRepo har fået en række nye metoder, dette kommer af at vi ikke har arbejdet så meget med SQL databaser før og derfor har set anderledes på hvordan vi ville kreere objekter i lister og lignende. Man kan også se at ActiveMotorhomeRepo inheriter fra vores MotorhomeRepo, dette er illustreret med den fuldt optrukkede linje og en hvid pil, dette gør vi da den bl.a. bruger sin super classes metode. Et andet punkt hvor vi også bruger polymorphismen er ved vores interfaces. Både MotorhomeRepo og CustomerRepo har et interface og dette er indikeret med en stiplet linje og en blå pil. Vi bruger interfaces for at sige hvad de forskellige klasser skal kunne. Udover dette har vi også tilføjet vores DatabaseConnectionManager som er den klasse der står for at forbinde til vores SQL-database, kigger man på diagrammet kan man se at alle dens attributter og metoder er understregede, dette indikerer at de er statiske. Vi bruger statiske metoder og attributter for at vi ikke behøver at oprette en instans af klassen for at bruge den, og for at sørge for at attributterne kun kan ændres ved brug af den gældende metode.

GRASP Responsibilities

Polymorphism (Jonathan)

Nedenstående diagram bruges til at vise de steder vi har brugt polymorphism i vores program. Vi bruger i vores kode både interface og inheritance. Vi bruger interfaces for at fastslå et basisgrundlag for hvad vores klasser skal kunne. Og vi bruger inheritance i vores ActiveMotorhomeRepo for at undgå at skulle genskrive kode unødvendigt og derved øge reuseability. Interfacet er vist med en stiplet linje og blå pil og inheritance bliver vist med den fuldt optrukkede linje og hvide pil.



Controller (Jacob)

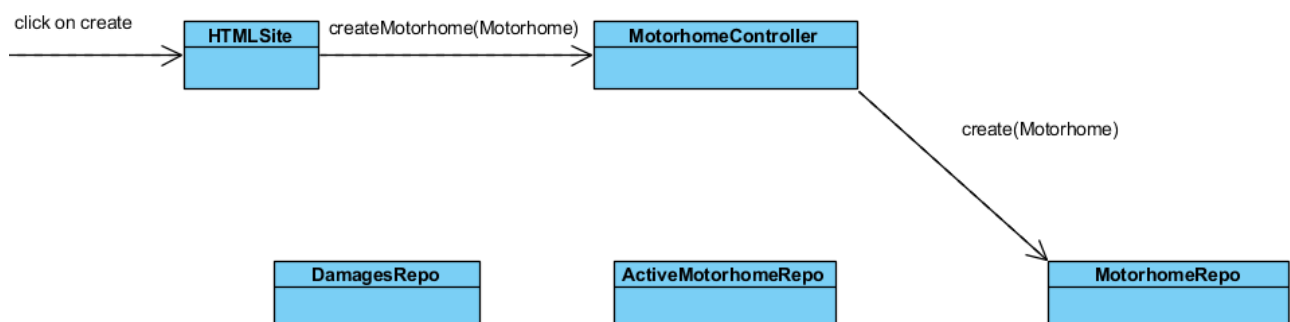
I GRASP-design bruges controller til at håndtere den data der kommer fra UI. Det er den første klasse data møder, som er ansvarlig for at håndtere og videresende information. Et helt konkret eksempel vil være: Hvis man trykker på login, hvor bliver password og login modtaget I en teoretisk applikation?

I en Spring applikation er det rimelig let at identificere hvilke java klasser som er controllere, fordi Spring kræver at bruger en `@Controller` annotation ved Controller klasser. De ligger alle sammen I samme mappe og har formodentlig controller som en del af deres navn. Det gøres så spring ved, at her er en klasse som modtager html requests. Springs UI er naturligvis alle html filerne med

tilhørende css filer, og det er her controllererne får deres information fra. Informationen kan være <form> submit data eller et link til en ny html adresse.

I vores applikation har vi 5 controller klasser, hvor 1 af dem kun håndterer /error, nemlig HomeController. De andre controller klasser er CustomerController, MotorhomeController, RentMotorhomeController og ActiveMotorhomeController. De fleste af controllerklassernes funktion giver sig selv, CustomerController håndterer alle html sider hvor Customer er i fokus, fx at oprette en ny kunde i systemet. MotorhomeController håndterer siderne hvor modellen Motorhome er i fokus, fx at hvis NMH har købt et nyt motorhome. RentMotorHome beskæftiger sig med at oprette nye leje aftaler, og modtager altså information såsom dato fra starten af lejeaftalen. ActiveMotorhomeController er måske den, hvor navnet giver mindst mening, hvis man ikke ved hvad der menes med active. I den her applikation skal active forstås som leje aftaler, som er oprettet og ligger i systemet. Derfor håndterer ActiveMotorhomeController ting som fx at aflyse en allerede eksisterende aftale eller afslutte en aftale når kunden returnerer et motorhome.

De modtager alle sammen information fra html siderne, som de sender videre til de relevante repositories. På nedenstående figur kan man se et diagram over hvordan controller modtager information og sender den videre.



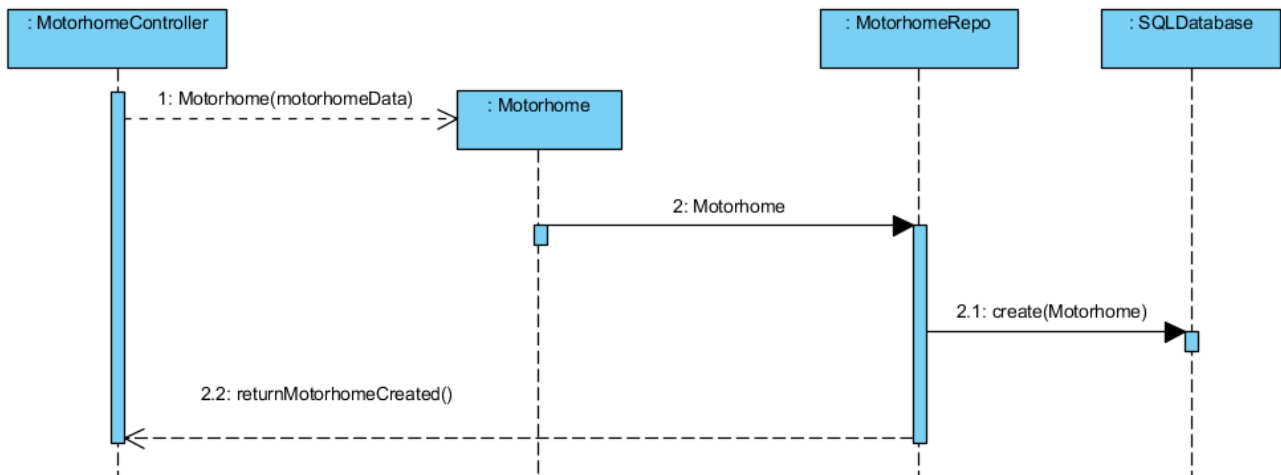
Creator (Jacob)

Creator GRASP-design princippet beskæftiger sig med, hvem opretter hver enkelt klasse. Hvem har ansvaret for at skrive new. Det er naturligvis vigtigt, at det er ikke er tilfældigt hvem der gør det, både for læsbarheden af koden, men også så man har en klasse der er information expert, så den klasse der opretter et objekt også har den information den skal bruge for at oprette klassen.

I vores application kan dette bedst ses ved brug af Controller klasserne, som har ansvaret for at instantiere de respektive repositories. Det giver bedst mening, at netop disse klasser instantiere repositories, fordi det er her klasserne skal bruges.

Det bliver også brugt når de forskellige klasser, både controllerer og repositories, opretter modeller motorhome. De bliver oprettet i klassen selv, da det er her de skal bruges og det er her de har informationen. Det er altså ikke en handler klasse eller lignende, som står for at lave nye Motorhome modeller.

Hvis man skal se på både creator og controller sammen, vil det være controllerklassen som får data, og her bruger den informationen til enten at oprette modeller som den selv bruger, eller sender den relevante information videre til et repository som vil overtage ansvaret.



Software Construction

HashMap (Jonathan)

Ser vi på metoden `getDmgByMotorhomeId()` i `DamageRepository`, ser man vi bruger en ny data struktur til at opbevare vores data, nemlig et `HashMap` som også er retur typen af metoden. Et `HashMap` er et hash table der implementerer `Map`. Det smarte ved et `HashMap` er dets hastighed og at den ikke kan have den samme key to gange. Udover dette er den dog også `unordered`, dog betyder dette ikke noget for os i de tilfælde vi skal bruge den.

Et `HashMap` har det der hedder en key og en value, den kan som sagt ikke have to af den samme key, i vores tilfælde er vores key af typen `Integer` og vores value skal være en `String`. Vi bruger en `PreparedStatement` til at hente data fra vores SQL-database og i linje 91 tager vi så vores data og sætter ind i vores `HashMap`. Vi bruger `getInt()` i put metoden til at sætte key-værdien. Vi har kaldt den værdi den henter den henter fra vores database, `damageld`. Går man ind og kigger i vores SQL-script der opretter databasen, kan man se at denne værdi auto-inkrementere derfor behøves vi ikke at bekymre os om at miste en value da der aldrig vil være to ens keys.

```
83 public HashMap<Integer,String> getDmgByMotorhomeId(int id){
84     HashMap<Integer,String> damages = new HashMap<>();
85     try {
86         String getAllString = "SELECT * FROM damages where motorhomeDmgId = ?";
87         PreparedStatement statement = connection.prepareStatement(getAllString);
88         statement.setInt( parameterIndex: 1, id);
89         ResultSet results = statement.executeQuery();
90         while (results.next()){
91             damages.put(results.getInt( columnIndex: 1), results.getString( columnIndex: 2));
92         }
93     }
94     catch (SQLException e){
95         System.out.println("error in DamageRepository : getDmgByMotorhomeId()");
96         System.out.println(e.getMessage());
97     }
98     return damages;
99 }
```

Deprecated `getMonth()` (Jonas)

Kigger man på nedstående billede, så vil vi gerne rette fokus på kode linje 148 og 149. Her benytter vi nemlig `getMonth` fra `java.util.Date` som man kan se er overstreget. Dette betyder at metoden er "deprecated", som i kort forstand betyder at der ikke er fokus på den metode længere. Den kan ende med at bliver superseded (afløst) af en anden metode og slet ikke eksistere i fremtiden. Vi har dog valgt at fortsætte med den, idet vi har arbejdet med den før. Hvis man skulle forsøge at optimere dette, ville man måske benytte sig af `java.util.Calendar` eller evt. bruge `java.time`, så man benyttede sig af `LocalDate` i stedet. Grunden til dette er at man har valgt at opdatere brugen af date og time.

```

128 @ public double getInitialPrice(Motorhome motorhome, HashMap<String,String> extraStuff, Date startDate, Date endDate){
129     double totalPrice = 0; //price is in whole euro
130     double dayPrice = 0;
131     long rentedDays = ChronoUnit.DAYS.between(startDate.toInstant(),endDate.toInstant());
132
133     try {
134         String getDayPrice = "SELECT price FROM motorhomes INNER JOIN motorhometype ON " +
135             "motorhomes.typeId = motorhometype.typeId WHERE motorhomes.motorhomeId = ?";
136         PreparedStatement statement = connection.prepareStatement(getDayPrice);
137         statement.setInt( parameterIndex: 1,motorhome.getId());
138         ResultSet resultSet = statement.executeQuery();
139         while (resultSet.next()){
140             dayPrice = resultSet.getDouble( columnLabel: "price");
141         }
142     }
143     catch (SQLException e){
144         System.out.println("error in MotorhomeRepository : at getInitialPrice()");
145         System.out.println(e.getMessage());
146     }
147
148     dayPrice *= seasonRepository.seasonPrice(seasonRepository.seasonType(startDate.getMonth() + 1, endDate.getMonth() + 1));
149     totalPrice += (dayPrice * rentedDays);

```

HTTP error handling (Jonathan)

Det er dvs. meget usandsynligt at vores program ikke fejler så derfor bliver vi nødt til at tage højde for de forskellige fejl der kunne komme. Når det kommer til http, kan der opstå mange fejl på forskellige tidspunkter, vi håndterer i vores kode 4 generelle fejl som vi har stødt ind i undervejs i programudviklingen. Udover de fire har vi også lavet en generel fejl håndtering, så hvis den fejl der opstår, ikke er fejl 404, 500, 403 eller 400 så vises der bare en generel besked.

Den klasse vi håndterer fejlene i har vi kaldt HomeController og den implementerer et spring interface der hedder ErrorController. For at vi kan lave en custom error side bliver vi nødt til at returnere den html side vi ønsker at vise når der opstår en fejl. Det er det vi gør fra linje 61 til 64. Vi bruger så RequestMapping og laver en metode der håndterer hvad der sendes når vi får en fejl. På linje 24 ses at vi bruger et af metodens parametre til at finde statuskoden på den fejl vi har modtaget, denne bruges så igen på linje 28 og laves om til en integer så vi kan lave logik på den. Udover at fortælle brugen hvilken fejl kode der er blevet produceret vil vi også fortælle lidt om den, til dette opretter vi først en String på linje 25 som er vores generelle fejlkode, denne ses kun hvis den fejl der opstår, ikke er en af de 4 fejl vi har valgt at håndtere. Er det dog en af de fejl der opstår så laves der en custom besked til den fejl, dette kan man se på linje 33. Så hvis der opstår en fejl aktiveres vores handleError() metode, og så længe status koden ikke er null går programmet videre. Hvis fejl koden f.eks. er 404 er if-statementet på linje 31 sandt og derfor bruger vi vores model til at sende både statuskoden og vores custom besked til error siden.

```

61 @Override
62 public String getErrorPath() {
63     return "/error";
64 }

```

```

22 @RequestMapping("/error")
23 @ public String handleError(HttpServletRequest request, Model model) {
24     Object status = request.getAttribute(RequestDispatcher.ERROR_STATUS_CODE);
25     String strError = "We did not expect that to happen";
26
27     if (status != null) {
28         Integer statusCode = Integer.valueOf(status.toString());
29
30
31         if(statusCode == HttpStatus.NOT_FOUND.value()) {
32             model.addAttribute( s: "message", statusCode);
33             strError = "Sorry, we couldn't find what you were looking for, try using the top menu to navigate.";
34             model.addAttribute( s: "stringError", strError);
35             return "/error";
36         }

```

Filter ift. Forskellige parametre (Jonathan)

Grundlaget for vores produkt gør det næsten umuligt ikke på et eller andet tidspunkt at filtrere ud fra hvad det er kunden eller andre skulle have lyst til at se. På grund af dette har vi selvfølgelig lavet en filter metode. Vores filter metode bruges når der skal udlejes et motorhome og ved hjælp af de parametre man indtaster, sender den så en liste tilbage. På nuværende tidspunkt kalder den bare en masse andre metoder, som modtager en liste og fjerner motorhomes fra den liste ud fra de gældende parametre. Lad os kigge på en af de metoder.

```

35 public ArrayList<Motorhome> filter(int activeState, int typeId, int maxPrice, int minPrice,
36                                     @DateTimeFormat(pattern = "yyyy-MM-dd") Date startDate,
37                                     @DateTimeFormat(pattern = "yyyy-MM-dd") Date endDate){
38     ArrayList<Motorhome> filteredList = returnAvailableMotorhomeByState(activeState);
39     filteredList = filterByTypeId(filteredList, typeId);
40     filteredList = filterByMaxPrice(filteredList, maxPrice);
41     filteredList = filterByMinPrice(filteredList, minPrice);
42     filteredList = filterByTwoDate(filteredList, startDate, endDate);
43     return filteredList;
44 }

```

Metoden filterByTwoDate() modtager to datoer og en ArrayList. Til at starte med checker den så at det er to gyldige datoer, er det ikke det returnerer den bare listen uden at fjerne nogle motorhomes fra den. Er de derimod gyldige oprettes der en ny tom liste kaldet found, denne bruges til at holde styr på hvilke motorhomes der skal fjernes fra listen senere i metoden. Da vi arbejder med SQL bliver vi også nød til at være forberedte på mulige fejl der kan opstå, dette gør vi med vores try-catch. Vi forbereder så en String vi kan indsætte i vores statement på linje 53, denne String skal bruges til at finde data i vores SQL fra vores custUseMotor table som passer med vores parametre. Vores filterString indsættes i vores PreparedStatement på linje 56 så vi senere kan eksekvere den med vores SQL-database. Fra linje 58 til og med 62 konverterer vi vores datoer til Strings så de kan forstås af SQL, dette kan gøres på andre måder, men vi mente dette var mest readable i tilfældet. Disse Strings indsættes så i vores PreparedStatement og vi eksekverer vores query. Vores data bliver indsat i et ResultSet, vi bruger så vores ResultSet til at gennemgå dataene

og checker om nogle af de motorhomes der er i vores ArrayList også er i ResultSettet. Hvis de findes, tilføjes de til found listen, som så bruges til at fjerne dem fra den originale liste. Når de er fjernet, returneres listen og så er metoden slut.

```

46 public ArrayList<Motorhome> filterByTwoDate(ArrayList<Motorhome> theList, @DateTimeFormat(pattern = "yyyy-MM-dd") Date startDate,
47                                     @DateTimeFormat(pattern = "yyyy-MM-dd") Date endDate){
48     if (endDate == null && startDate == null){
49         return theList;
50     }
51     ArrayList<Motorhome> found = new ArrayList<>();
52     try {
53         String filterString = "SELECT * FROM motorhomes INNER JOIN custusemotor ON motorhomes.motorhomeId = custusemotor.motorhomeId" +
54                               "WHERE (? between startDate and endDate) OR (? between startDate and endDate) OR (startDate between ? and ?)" +
55                               "OR (endDate between ? and ?)";
56         PreparedStatement statement = connection.prepareStatement(filterString);
57
58         SimpleDateFormat startDateFormat = new SimpleDateFormat( pattern: "yyyy-MM-dd HH:mm:ss" );
59         String currentStartDateTime = startDateFormat.format(startDate);
60
61         SimpleDateFormat endDateFormat = new SimpleDateFormat( pattern: "yyyy-MM-dd HH:mm:ss" );
62         String currentEndDateTime = endDateFormat.format(endDate);
63
64         statement.setString( parameterIndex: 1, currentStartDateTime);
65         statement.setString( parameterIndex: 2,currentEndDateTime);
66         statement.setString( parameterIndex: 3,currentStartDateTime);
67         statement.setString( parameterIndex: 4,currentEndDateTime);
68         statement.setString( parameterIndex: 5,currentStartDateTime);
69         statement.setString( parameterIndex: 6,currentEndDateTime);
70         ResultSet resultSet = statement.executeQuery();
71         while (resultSet.next()){
72             for (Motorhome home : theList){
73                 if (home.getId() == resultSet.getInt( columnIndex: 1)){
74                     found.add(home);
75                 }

```

Valide Data Input (Jonathan)

Da vi arbejder med bruger input er det også svært at undgå at der til tider vil være forkerte inputs fra brugeren. Dog kan vi ved hjælp af java annotationer nemt reducere disse fejl. Kigger man på billedet til højre kan man se vi skriver **@NotEmpty**, **@NotNull**, **@Min** og **@Max**, disse annotationer bruges bl.a. i vores Motorhome model til at sige at disse attributter ikke må være tomme eller skal indeholde bestemte værdier. Man skriver annotationen og den attribut den skal være gældende for.

```

11 @NotEmpty
12 private String brand;
13 @NotEmpty
14 private String model;
15 @NotNull
16 private int timesUsed;
17 @NotNull
18 private int kmDriven;
19 private int price;
20 private int activeState;
21 @NotNull
22 @Min(1)
23 @Max(8)
24 private int typeId;

```

Man kan bruge disse krav i en anden annotation, **@Valid**. Kigger man på nedenstående billede kan man se der på linje 29 står **@Valid**, denne annotation bruges til at checke om de attributter det gældende motorhome har, opfylder de krav dets models annotationer har stillet.

```

28 @PostMapping("motorhomes/createMotorhome/add")
29 @Valid
30 public String createMotorhomeAdd(@Valid Motorhome motorhome, BindingResult bindingResult){
31     if (bindingResult.hasErrors()){
32         return "redirect:/motorhomes/createMotorhome";
33     }

```

Udover at vi checker det i vores controller så bruger vi Thymeleaf til at checke om værdierne er korrekte i html delen. I dette tilfælde bruger vi Thymeleafs if funktion til at checke om den opfylder kravene til typeld attributten. Det vil sige hvis den værdi der står i typeld's input felt ikke er minimum 1, maksimum 8 og ikke er "null" så kan man submitte sin form. Det minder lidt om at skrive required, men den checker også om værdien opfylder de givne krav.

```
43 <td><input type="number" th:field="*{typeId}" value="1" min="1" step="1"></td>
44 <td th:if="{#fields.hasErrors('typeId')}" th:errors="*{typeId}">typeId error</td>
```

Fragments (Jacob)

Fragments er en Thymeleaf function, som gør det meget let at indsætte stykker af html kode i en html fil. Som navnet antyder, indsætter den en fragment af noget kode.

Vi har brugt funktionen til at indsætte topnav menuen i alle vores html filer. Det er gjort, så man sikre at alle sider i applikationen indeholder den samme menu, og hvis man ønsker at ændre den, skal det kun ske 1 sted.

Man kalder et fragment ved at indsætte følgende kode i html:

```
<body>

<div th:insert="fragments/menu :: menu"></div>

<div class="container">
```

Hvor fragments er mappen, det første menu er navnet på filen og det sidste er "metoden" som er blevet kaldt menu. Alt css, html kode osv. Bliver derefter hentet fra filen, og skrevet ind her. Det betyder at når en browser åbner siden, ser de ikke thymeleaf, men den kode som thymeleaf har indsat.

Thymeleaf koden der navngiver den kode der skal indsættes ser således ud:

```
<div th:fragment="menu">
```

Derefter indskrives man bare alt den kode man vil, i den tilhørende div.

I vores application har vi brugt det til 2 funktioner, menuen og en footer. Footeren er ikke speciel, det er bare noget tekst der beskriver hvem der har lavet projektet.

I vores menu, har vi lavet 1 ting som er værd at fremhæve:

Dropdown menu (Jacob)

I vores topnav menu er der dropdown menuer:


```

<div class="dropdown a-topNav-border" th:classappend="${#httpServletRequest.getRequestURI().startsWith('/customer') ? 'active':''}">
  <button>Customer ↓</button>
  <div class="dropdown-content" >
    <a class="dropdown-a" href="/customer/create">New customer</a>
    <a class="dropdown-a" href="/customer/list">Customer list</a>
  </div>
</div>

```

Det som får menuen til at være en dropdown, ligger i vores CSS stylesheet. Thymeleaf funktionen classappend tilføjer en klasse. Klassen som vil være tilføjet, er fundet ved en thymeleaf if statement, som enten returnerer active eller "". Active er den css klasse der bruges når man er på siden, i udsnittet hvis man er på en side der ligger under customer. Ellers tilføjer man "", som ikke giver nogen ændring eftersom det ikke betyder noget.

Den magiske kode, som får hele dropdown menuen til at virke er:

```

27 .dropdown-content { /*can't see content*/
28     display: none;
29     position: absolute;
30 }
31 .dropdown:hover .dropdown-content { /*can see content when hover*/
32     display: block;
33 }

```

Det betyder meget simpelt, at den div med class dropdown-content er skjult i form af display: none;

Når man så hoverer, vil den ændre display til at være en block. En anden del af css kode som er vigtig er:

```

.dropdown {
    display: inline-block;
    position: relative; /*makes the dropdown be relative to button*/
}

```

Uden position: relative; vil dropdown menuen dukke op så langt til venstre som resten af css tillader. Det gør det umuligt at vælge et punkt i menuen, derfor er den linje også en vigtig del af dropdown menuen. Resten af css er for at gøre menuen smukt.

HTML, CSS

Input felter (HTML), CSS connection med fontawesome (Tobias)

Vores input felter i HTML er lavet i en div vi har givet en klasse "myTextBox", inde i denne div ligger flere <div></div> hvori der er en label der fungerer som en slags overskrift til det

efterfølgende input felt. Vi har valgt at

lave div inde i div fordi det giver os et

bedre grundlag for at kunne designe

interfacet i css. Vores input felter har

nogle forskellige attributter f.eks.

bruger vi required, som fungerer som

en boolean, hvilket betyder at der skal

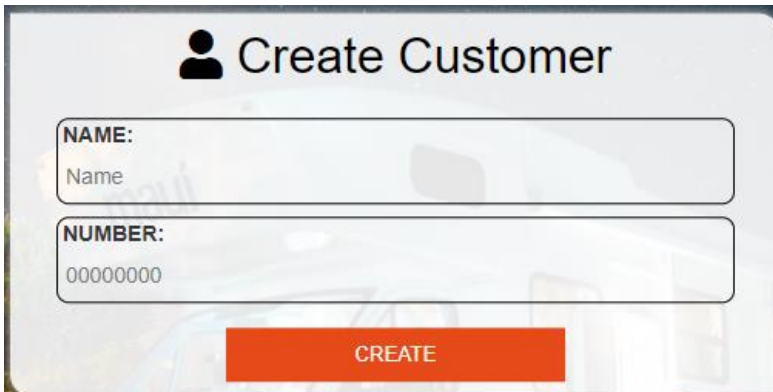
være tekst i inputfeltet før man kan klikke submit. En anden feature vi vil fremhæve, er vores <i>

tag inde i vores <h1> tag. Dette <i> med klassen fas fa-user fungerer kun fordi vi i vores <head> tag har et <script> som refererer til en side som hedder fontawesome.com.:

```
<div class="myTextBox">
  <h1><i class="fas fa-user"></i> Create Customer</h1>
  <div class="txtb">
    <label>Name: </label>
    <input type="text" name="cName" required placeholder="Name">
  </div>
  <div class="txtb">
    <label>Number: </label>
    <input type="number" name="Number" min="0" required placeholder="00000000">
  </div>
  <input class="buttonNew" type="submit" name="submit form via POST" value="Create">
</div>
```

```
<script src="https://kit.fontawesome.com/7898533ecb.js" crossorigin="anonymous"></script>
```

Dette script fra fontawesome.com giver os adgang til at benytte utallige af ikoner fra deres hjemmeside. I dette eksempel har vi brugt et user ikon under vores input sektion til at lave en ny kunde. Ikonet kan man så via klassen fas fa-user, style på i sit css dokument.



Test (Jacob)

Der er udført test i junit på de dele af projektet, der er vurderet til at være væsentlige dele af backend. Tilgangen til at lave testene har været, at de kun tester en enkelt del af projektet, uden at ændre den data der ligger på MySQL serveren. Det er tiltænkt at testene tester dele af projektet, der ligger på et så lavt afhængighedsniveau som muligt, da testene ellers ville undersøge mange metoder på én gang og det ikke er tiltænkt at teste hele strukturen på én gang. Det gør det lettere at finde ud af hvorhenne en eventuel fejl måtte ligge, da der kun ses på små dele af gangen.

I figuren nedenfor ses de test som er udført.

Name of class with the Tests	Name of tested class	Name of test:	Name of tested method
customerTests	customerRepository	testEmptyCustomerCreation	create(Customer)
customerTests	customerRepository	testCustomerCreateAndDelete	create(Customer) and delete(id)
customerTests	customerRepository	testOfReadAll	readAll()
motorhomeFilterTests	activeMotorhomeRepository	testOfTypeFilter	filterByTypeid(ArrayList,filterId)
motorhomeFilterTests	activeMotorhomeRepository	testOfMaxPrice	filterByMaxPrice(ArrayList,maxPrice)
NmHprojectApplicationTests	DatabaseConnectionManager	setup	getDatabaseConnection()
rentAHomeTest	activeMotorhomeRepository	testCancelRentAgreement	cancelRentAgreement(rent id)
rentAHomeTest	ActiveMotorhomeRepository	testOfRentHomeAndReturnHome	rentHome(rent details) and homeReturned(rentId)

CustomerRepository test:

De metoder der bliver testet, er create, delete og readAll. Create metoden bør returnere det nye customer id, som den ny-oprettede customer har fået i MySQL table customers. Create bliver testet 2 gange, en gang hvor man tester om man kan indsætte en "tom" kunde, hvor alle variable er null, i customers MySQL table. Den anden create indsætter en komplet test kunde, som gerne skulle returnere et frisk customerId, som kan bruges til at teste delete metoden.

Der er ikke fundet større udfordringer ved at teste customerRepository, eller Customer class. Det kommer af, at funktioner ikke går længere end CRUD. Det er altså nemt at se, hvorvidt informationen kan udføre meget simple operationer.

Test af motorhome filter:

Der er blevet testet filterByMaxPrice og filterByTypeid. Det er gjort ved at give filterne en komplet liste fra motorhomeRepository.readAll(). Det er derefter forholdsvis simpelt at tjekke om de motorhomes med den givne maxpris og typeid er blevet fjernet fra listen. Grundet til at minprisen ikke bliver testet er, at den er skrevet på næsten samme måde som min. prisen.

Der er ikke blevet undersøgt om filterByStartDate eller endDate virker. Det er ikke gjort, da der ikke ligger nogen metoder som kan undersøge det. Det vil betyde, at vi skulle skrive nye metoder for at teste den del af projektet, eller ændre strukturen på vores modeller, MySQL serveren eller en kombination af de to. Det kommer sig af, at start datoen på en lejeperiode ligger i en tabel der hedder custusemotor, og ikke hos hver enkelt motorhome. Det er naturligvis godt, fordi normalitet reglerne siger man ikke må have en liste liggende i en tabel celle. Det giver dog den udfordring, at intern kan vi ikke undersøge om hver enkelt motorhome er lejet ud i den bestemte periode andre steder end i filtermetoden selv.

En løsning ville være at indsætte en test lejeperiode i MySQL databasen, også kigge så langt ud i fremtiden at der ikke bør være nogle overlap.

ActiveMotorhomeRepository test:

Der er udført 2 test på den her class, selvom der er rig mulighed for at udføre flere. Der er blevet testet 2 af de vigtigste funktioner i programmet, nemlig oprettelse af nye lejeaftaler og metoden der håndterer aflysninger. Hvis 1 af disse metoder ikke fungerer, vil det betyde at hele programmet har kritiske fejl, fordi man mister overblikket over hvor et motorhome befinder sig.

Testen virker ved først at oprette en ny aftale i begge tilfælde, og dernæst henholdsvis at betragte den oprettede testaftale som henholdsvis afsluttet og aflyst.

Det output man får fra metoden er boolean, og vil forhåbentlig give en god indikation om metoden virker. Der er i testen stadig en mulighed for fejl, da metoderne kan have SQL statement fejl der stadig bliver kørt, men ikke udfører den korrekte funktion. Det er derfor ikke en helt fyldestgørende test, men det er ikke muligt at lave den bedre uden at lave større metoder i et nyt repository.

DatabaseConnectionManager test:

Testen er meget simpel, den tester om man får en SQL Connection til sin server, hvis den ikke får et svar og kan validere Connection indenfor 10 sekunder fejler testen. Det må antages at man kan etablere et svar fra en gyldig server indenfor 10 sekunder, især fordi vi på dette eksamensprojekt bruger lokalhost.

Konklusion af test:

Det har i det større billede været forholdsvis svært at lave nogle gode test til det her projekt, det kommer af, at det er nyt at teste på MySQL og derfor har vi ikke haft test i tankerne. Som tidligere nævnt har fx filterByStartDate og filterByEndDate været svære at få testet på en fornuftig måde, uden at være nødsaget til at ændre i selve projektet eller skrive nye metoder. Der er ikke nogen metode, som giver et fornuftigt billede over hvad der er udlejet, så det er svært at teste om fx cancelRentAgreement metoden virker. Den giver selvfølgelig en false/true output, men der er ingen fornuftig måde at verificere at et bestemt rentId er fjernet fra MySQL.

I fremtidige projekter vil det være smart at have test i tankerne, så man kan lave nogle test, der giver et fyldestgørende indblik i programmets fejl. Det som ville have haft den største betydning, ville nok være at have en bedre måde at trække information om leje aftaler ud af programmet. I vores projekt har meget af fokus ligget på, at informationen om motorhomes er vigtig, men det er senere, især ved test, blevet klart at fokus skulle have ligget på lejeaftalerne og den information der ligger i forbindelse med det. Det ville sandsynligvis også have gjort prisudregning lettere i forbindelse med udlejning, aflysninger og tilbageleveringer af motorhomes.

Det er lykkedes med de fleste test, da alle metoder har et output som man kan teste på.

Der er fundet nogle begrænsninger for, hvad der var muligt at teste på en effektiv måde.

MySQL:

Beskrivelse af ER diagram og tabeller:

Motorhomes (Jacob):

Vores tabel "motorhomes", er en "liste" over alle de 32 motorhomes, som vores fiktive firma ejer. Der er mulighed for at oprette flere i vores applikation, i tilfælde af, at flere motorhomes skulle købes i fremtiden. Tabellen består af et motorhomeld, som er den primære key og bliver brugt igennem hele applikationen til at identificere det enkelte motorhome. Brand, model, timesUsed og km Driven beskriver alle egenskaber omkring det enkelte motorkøretøj, navnlig hvilket mærke og model køretøjet er, hvor mange gange det er blevet brugt, og hvor langt det har kørt. ActiveState bliver brugt til at se om det enkelte køretøj er leget ud, eller ej. Typeld viser hvilken type motorhomet er, sengepladser, dagspris, og typeld er en foreign key til "motorhometype" tabellen som bliver beskrevet senere.

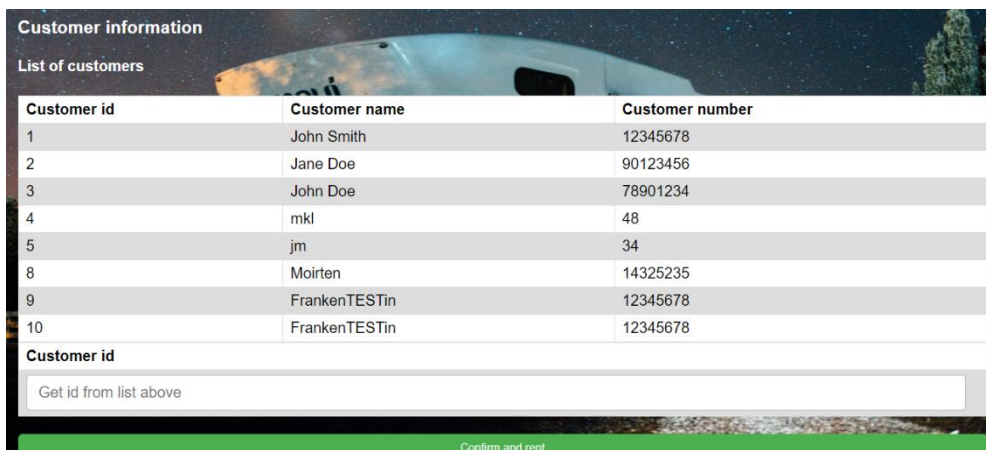
Der findes en model klasse i applikationen som hedder Motorhome, og data fra tabellen kan direkte oversættes til tabellen.

Custusemotor (Jacob):

Custusemotor er vores tabel, som beskriver lejeaftaler, navnet er en sammentrækning af customers use of motorhomes. Den har dens primary key rentId som også bliver brugt af java, til at identificere de forskellige lejeaftaler. Tabellen indeholder start datoen og slut datoen for lejeperioden, startDate og endDate, ekstraPrice som er den pris som kunden vil betale, hvis han ikke får flere gebyrer, fx kørt mere end de tilladte 400 km gennemsnit om dagen. EkstraPrice har allerede fået indregnet en eventuel stigning af pris, hvis kunden lejer i en højsæson. CustomerId og motorhomeld, referer til det motorhome som er blevet udlejet, og den kunde som lejer motorhomet.

Customers (Jacob):

Customers er vores database over kunder, som er registreret i vores system. Man skal være registreret før man får lov at leje, da man vælger fra en liste over allerede registrerede kunder når man lejer, som ses på billedet under:



The screenshot shows a web interface titled "Customer information" with a subtitle "List of customers". It displays a table with three columns: "Customer id", "Customer name", and "Customer number". The table lists 10 customers. Below the table is a form with a label "Customer id" and a text input field containing the placeholder "Get id from list above". At the bottom of the interface is a green button labeled "Confirm and rent".

Customer id	Customer name	Customer number
1	John Smith	12345678
2	Jane Doe	90123456
3	John Doe	78901234
4	mkl	48
5	jrn	34
8	Moirten	14325235
9	FrankenTESTin	12345678
10	FrankenTESTin	12345678

Customer id
Get id from list above

Confirm and rent

Customers primary key er customerId, som også bliver brugt til at identificere kunder i java koden. Hver kunde har meget lidt information stående om dem, nemlig et navn og et nummer. Det er gjort, da det ikke har nogen betydning for de usecases der er udviklet. Det er set som spild af energi at gøre klassen større og inkludere information, som ville være en del af et virkeligt projekt, fx cpr-nummer, e-mail, mm.

Hvis man skulle fuldt udvikle dette projekt, ligger det frit for, at man tilføjer flere søjler i Customer tabellen, og tilføjer felterne i de relevante html filer. Det vil ikke ændre eller forstyrre nogen logik i applikationen (low coupling).

Motorhometype (Jacob):

Motorhometype tabellen beskriver de forskellige typer af motorhomes. Det er en tabel for sig selv, så man kun skal ændre informationen om typen 1 sted. Tabellen beskriver hvad prisen er, for hver enkelt motorhome type, og hvor mange sengepladser der findes. På samme måde som customer tabellen, er type tabellen også ret tom. I den virkelige verden er det meget muligt, at der vil være mere relevant information, fx om der er toilet, køkken, etc. men der er igen ingen grund til at bruge tid på at finde på en masse information.

Damages (Jacob):

Damages tabellen indeholder beskrivelser om hvilke skader der er på et køretøj (damageDesc), hvilket køretøj der har skaden (motorhomeld), og tabellens primary key som bliver brugt til identifikation af hver enkelt skade. Grunden til at informationen ikke ligger i motorhomes tabellen, er at normalform 1 siger: Man ikke må have lister i en database. Hvis man har 2 skader på samme motorhome, vil det være en liste over skader.

Season (Jacob):

Season indeholder information om høj, middel og lav sæson, som har betydning for prisudregningen. Hvert primært id har tilknyttet en startdato, slut dato og hvilken type sæson der snakkes om (høj, middel og normal). Java tjekker ved prisudregning den her tabel, og season tabellen har derfor ingen relation til de andre tabeller i databasen. Det er gjort, da det er java, der står for prisudregningen og ikke MySQL.

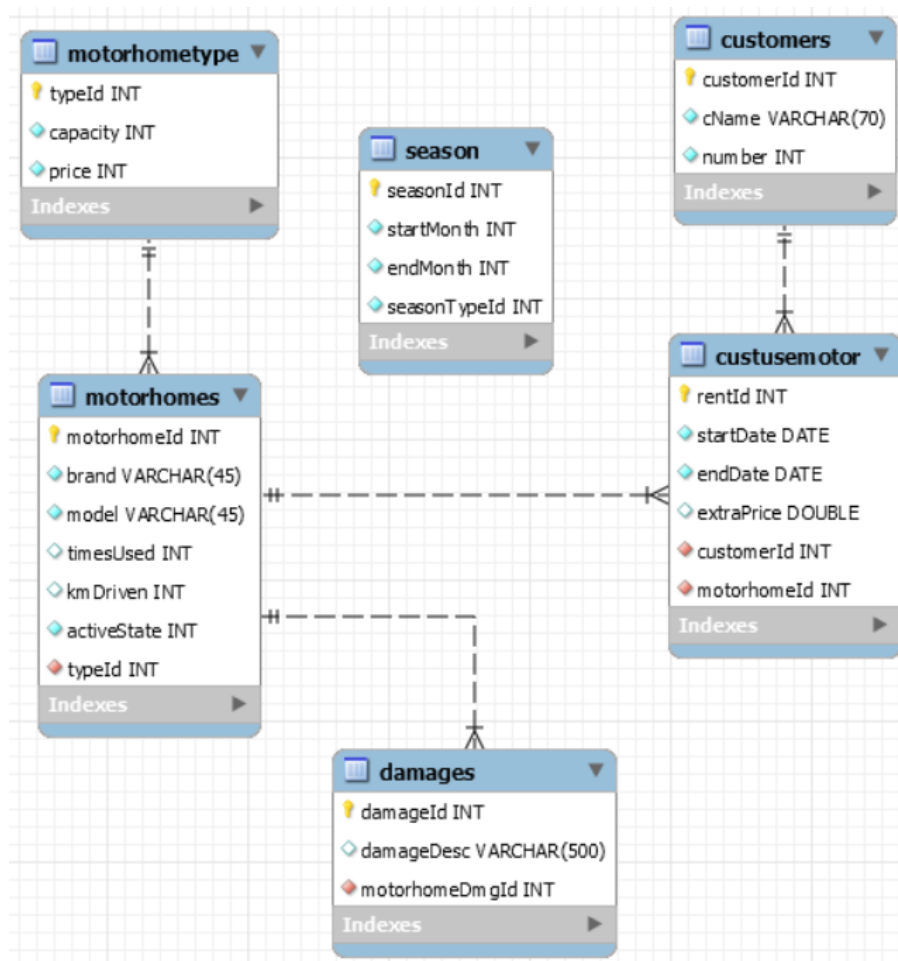
ER diagram (Jacob og Jonathan):

Vi har i vores arbejde valgt at oprette ER diagrammet som noget af det første i projektets forløb. Det er gjort, fordi det giver mulighed for at se, hvordan vores data kommer til at se ud med det samme. Det gør det lettere i resten af udviklingsprocessen, når man kan se, hvordan de forskellige dele af projektet skal udarbejdes, som fx domænemodellen, html sider osv.

Kigger man på diagrammet kan man også se vi har linjer imellem vores tabels, disse viser kardinalitet mellem dem. Kigger man på vores linjer kan man hurtigt se der kun er en slags, dog har den to forskellige måder at vise forekomster af forhold mellem de to tables. Disse vises ved to streger og en streg med en fordeling efter. De to streger betyder at det er en og kun en. Kigger vi på linjen imellem motorhomeType og motorhomes kan vi se at motorhomeType har to streger,

det betyder altså at et motorhome kan have en, og kun en, motorhomeType. Kigger man derimod på motorhomes symbol kan man se strengen der fordeler sig, dette betyder at motorhomeType kan have mange motorhomes. Et eksempel er hvis man har et typeId der hedder 1, mange motorhomes kan være af type 1, dog kan et motorhome kun have en type, det vil sige den kan ikke være type 1 og 2.

Den endelige version af diagrammet kan ses nedenfor. Diagrammet har ændret sig minimalt siden første version, men der har været småting. Blandt andet ændrede vi prisen fra at ligge i motorhomes, til at ligge i motorhomeType. Det betyder at prisen er baseret på typen, og ikke på det enkelte motorhome. Det gør det lettere at ændre prisen for flere hjem, men giver den ulempe at det er svære at differentiere mellem de forskellige motorhomes. Diagrammet er udviklet med de 3 normalformer i tankerne.



Normalformer (Alle):

Normalform 1:

Vores database overholder alle normalformerne. Første normalform siger kort sagt, at ingen kolonner gentager en andens værdier, og at databasen skal være atomisk (når man googler atomisk, kommer der kun fysik frem, hvilket jeg fandt lidt morsomt).

Det kan bedst ses, at vi ikke har nogen gentagne kolonner ved tabellen damages. Damages

indeholder information om motorhomes, men fordi at motorhome kan have flere skader vil det være nødvendigt med flere kolonner der indeholder skader, eller lave en liste i en celle. Begge ting er naturligvis imod normalform 1, og derfor har vi lavet en tabel som indeholder skadebeskrivelsen, og et foreign key som linker til det motorhome der har skaden.

Normalform 2:

Normalform 2 gælder for tabeller med sammensatte primary keys. Alt information i en tabel med sammensatte primary keys skal direkte beskrive og være afhængige af primary key.

Normalform 2 bliver ikke brugt i databasen, ikke fordi den er uvæsentlig, men fordi vi ikke har nogen tabeller med sammensatte primary keys. Det har været meget let at undgå, fordi alle tabellerne har en simpel integer id key. Det eneste sted, hvor det kunne have været relevant er motorhomes. Det kunne have været muligt at lave model og brand til unikke keys, men det vil kun være muligt, hvis man har et af hvert køretøj. Det er meget tænkeligt at et firma køber 2 eller flere af same mærke og model, derfor har motorhomes deres egen individuelle key.

Normalform 3:

Normalform 3 siger, at alle kolonner kan bestemmes ud fra primary key, og at der ikke må være transitive funktionelle afhængigheder.

Det kan bedst ses i vores database ved motorhometype tabellen. Den beskriver forskellen på de forskellige typer motorhomes. Den kunne i teorien ligge under motorhomes tabellen, men det ville ikke overholde 3 normalform, da motorhomes tabellen ville indeholde data der ikke afhænger af motorhomeld, men derimod af typeId. Derfor er den blevet lagt ud i sin egen tabel, og motorhomes tabellen har så fået et foreign key, der referer til type tabellen.

SQL dokumentation (Alle):

Prepared statement:

Gennem hele vores projekt gør vi brug af PreparedStatement, som er en del af java.sql. Det gøres for at undgå SQL injektion angreb. SQL-injektion fungerer ved, at man som bruger indsætter en SQL-statement, ofte med et ødelæggende bagtanke, i et felt som fx brugernavn, og håber at logikken der ligger inde bagved kører ens statement. Fx kunne en hacker skrive som brugernavn, "SELECT* FROM user WHERE 1=1" i håb om at modtage en liste, der indeholder passwords eller e-mails.

```
172 @ public boolean newRentDeal(Date startDate, Date endDate, double price, int customerId, int motorhomeId){
173     try {
174         String insertRentDeal = "INSERT INTO custusemotor(startDate, endDate, extraPrice, customerId, motorhomeId) VALUES (?, ?, ?, ?, ?)";
175         PreparedStatement statement = connection.prepareStatement(insertRentDeal);
176         statement.setDate( parameterIndex: 1, new java.sql.Date(startDate.getTime()));
177         statement.setDate( parameterIndex: 2, new java.sql.Date(endDate.getTime()));
178         statement.setDouble( parameterIndex: 3, price);
179         statement.setInt( parameterIndex: 4, customerId);
180         statement.setInt( parameterIndex: 5, motorhomeId);
181         statement.executeUpdate();
182         return true;
```



```

172 @ public boolean newRentDeal(Date startDate, Date endDate, double price, int customerId, int motorhomeId){
173     try {
174         String insertRentDeal = "INSERT INTO custusemotor(startDate, endDate, extraPrice, customerId, motorhomeId) VALUES (?, ?, ?, ?, ?)";
175         PreparedStatement statement = connection.prepareStatement(insertRentDeal);
176         statement.setDate( parameterIndex: 1, new java.sql.Date(startDate.getTime()));
177         statement.setDate( parameterIndex: 2, new java.sql.Date(endDate.getTime()));
178         statement.setDouble( parameterIndex: 3, price);
179         statement.setInt( parameterIndex: 4, customerId);
180         statement.setInt( parameterIndex: 5, motorhomeId);
181         statement.executeUpdate();
182         return true;

```

Java koden til at lave en PreparedStatement, ser ud som billedet ovenfor. Man skriver sit SQL-statement, og alt data der skal indsættes i statement, bliver lavet annoteret med "?". Den data man indsætter i sin string, bliver derefter tjekket og ens preparedStatement kører ikke hvis den fx har "select * from user" i sig.

Create og drop (DDL):

I vores projekt har vi gjort brug af DDL og DML. DDL står for Data definition Language, og beskæftiger sig med at oprette databaser og ændre på allerede oprettede databasers struktur. DDL bliver derfor brugt i vores create script, til at oprette selve databasen og de tabeller der skal være i databasen. De DDL-funktioner vi har brugt, er create og drop. Begge funktioner bliver kun brugt i forbindelse med vores create script. Create bliver brugt til at oprette schema (databasen)³ og tabellerne. Eksempelvis ved oprettelse af tabellen motorhometype:

```

21 • DROP TABLE IF EXISTS `NMH_company`.`motorhomeType` ;
22
23 • CREATE TABLE IF NOT EXISTS `NMH_company`.`motorhomeType` (
24     `typeId` INT NOT NULL,
25     `capacity` INT NOT NULL,
26     `price` INT NOT NULL,
27     PRIMARY KEY (`typeId`))
28     ENGINE = InnoDB;

```

Her oprettes en ny tabel i databasen "NMH_company", som er navnet på vores database. Drop table if exists bliver brugt, i tilfælde af, at man har kørt en ældre version af vores script eller scriptet fejler halvvejs igennem og man er nødt til at køre scriptet igen. Det har naturligvis den ulempe, at man risikere at droppe en masse vigtige data.

Motorhometype tabellen er en af de få tabeller, hvor id ikke har brugt AUTO_INCREMENT som en del af oprettelsen af primary id. Det er gjort fordi typeId bliver brugt til at beskrive en bestemt type, og det er ikke ligegyldigt hvad nummer de har. Da det helst skal give nogenlunde mening, og fx ikke springe tal over i tilfældigheder. Det er mest så mennesker lettere får et overskud, det er lettere at forstå type 1,2,3..8 end 1,9,19,2...8. Et eksempel hvor AUTO_INCREMENT bliver brugt:

³ I MySQL 5.0.2 og nyere er database og schema det samme. Det er ikke det samme i andre SQL produkter.

```

59 • CREATE TABLE IF NOT EXISTS `NMH_company`.`customers` (
60     `customerId` INT NOT NULL AUTO_INCREMENT,
61     `cName` VARCHAR(70) NOT NULL,
62     `number` INT NOT NULL,
63     PRIMARY KEY (`customerId`))
64     ENGINE = InnoDB;

```

Her har customerId ikke større betydning, om man har fat i kunde 1 eller 199 har ingen betydning. Det er fuldstændig ligegyldig information, både for mennesker og for maskinen. Forhåbentlig for vores firma bliver customerId på et tidspunkt rigtig højt, og det giver endnu mindre mening med manuel styring af id.

Databasens integritet:

Den engine der bliver brugt er InnoDB, det er standard i MySQL i dag. Vi har i løbet af undervisningen ikke haft noget information omkring MySQL engines. Tidligere var det ikke InnoDB, som blev brugt som standard, men derimod MyISAM. Den gamle engine havde den ulempe at den ikke tjekkede, om det var "valid data" man indsatte i tabellerne. Derfor er det min forståelse, at man kunne indsætte "invalid data", fx en dato som fx 30 februar ville blive accepteret af MyISAM. Vi bruger InnoDB derfor bliver invalid data ikke godtaget i databasen, fx ville d. 30 februar blive afvist og MySQL ville give beskeden Error Code: 1292. InnoDB har også andre fordele, som vi ikke gør brug af. Med InnoDB har man fx mulighed for rollback i tilfælde af fejlagtige opdateringer. Vi gør brug af NOT NULL funktionen når vi opretter vores tabeller, det betyder at vi ikke kan indsætte tom information. Det bliver også testet hvorvidt det er muligt i vores customerTests klasse:

```

23 • CREATE TABLE IF NOT EXISTS `NMH_company`.`motorhomeType` (
24     `typeId` INT NOT NULL,
25     `capacity` INT NOT NULL,
26     `price` INT NOT NULL,
27     PRIMARY KEY (`typeId`))
28     ENGINE = InnoDB;

```

Derudover bruger vi AUTO_INCREMENT ved oprettelse af nye primary keys, hvilket giver sikkerhed for, at vi kun har 1 primary key.

Der er ikke nogen default værdier for nogen kolonner i vores database. Det kommer af, at der som udgangspunkt aldrig bør være tom information uploadet til vores database (bortset fra primary key, som bliver håndteret af AUTO_INCREMENT). Da vi forventer, at data der bliver sendt til vores MySQL database er udfyldt, har vi heller ikke gjort mere for at tjekke hvorvidt det er valide data udover NOT NULL og type (int, varchar, date osv.).

Der vil også være nogle tjeks i selve Java delen af koden. Blandt andet kan man ikke lave et validt motorhome hvis ens typeId ligger over 8, eller under 1, som er henholdsvis max og min type

nummer. Det er opnået ved koden:

```
21 @NotNull
22 @Min(1)
23 @Max(8)
24 private int typeId;
```

I motorhome modellen.

I det store hele skulle der ikke være mulighed for at indtaste data, som ikke giver mening i deres respektive celle.

On delete/update no action:

On delete/update no action betyder, at hvis man sletter noget data, der har en foreign key, kan man påvirke de tabeller den respektive foreign key hænger sammen med. Det er ikke gjort i vores database, da vi ikke forventer at slette noget, der har en påvirkning andre steder, vi ønsker fx ikke at slette vores kundekontakt information, bare fordi de ikke længere lejer et motorhome (i tilfælde af, at man finder skader fx).

Et sted i vores database, hvor det ville være ønskværdigt, ville være ved damages. Det giver ikke stor mening at gemme skadesrapporter om et motorhome, man ikke længere ejer. Det er ikke gjort, da motorhomes tabellen ikke gemmer noget information om damages, og man derfor ikke kan slette data i damages, ved hjælp af "action on delete". Hvis man sletter en skade fra damages, ønsker man ikke at slette ejerskabet om det motorhome man lige har fixet.

DML:

DML står for data manipulation language. Det beskæftiger sig med manipulation af den data der ligger i de forskellige tabeller, de MySQL statements vi bruger i forbindelse med DML er insert, update og Delete.

Insert:

Insert bliver brugt til at indsætte nye rækker i en tabel. Alt efter hvordan ens tabel er blevet oprettet, kan man vælge at udfylde dele af rækken, og dermed udelade felter, eller fylde alle cellerne i rækken ud. De fleste kolonner i vores database er af typen NOT NULL, som betyder at de skal have information før MySQL godtager et statement. Derfor vil man være tvunget til at udfylde alle kolonnens data når bruger insert i vores database, den største undtagelse for dette er de primary key id, som bliver udfyldt automatisk af MySQL. Det sker da de er blevet oprettet med primary keys der har auto_increment, som betyder at MySQL automatisk giver primary key 1 højere end tidligere.

Et eksempel på data der bliver indsat i vores tabel er i metoden newRentDeal som ligger i MotorhomeRepository filen. Her bliver der oprettet en ny lejeaftale (som navnet indikerer), og

derfor skal den indsættes i vores MySQL database. Der bliver brugt følgende insert statement:

```
172 @ public boolean newRentDeal(Date startDate, Date endDate, double price, int customerId, int motorhomeId){
173     try {
174         String insertRentDeal = "INSERT INTO custusemotor(startDate, endDate, extraPrice, customerId, motorhomeId) VALUES (?, ?, ?, ?, ?)";
175         PreparedStatement statement = connection.prepareStatement(insertRentDeal);
```

Man skriver hvilken tabel man indsætter data og herefter values som er den data der skal indsættes.

Update:

Update funktionen bliver brugt til at opdatere en allerede eksisterende række, hvis man fx skal ændre en kundes efternavn fordi de blevet gift. Det gøres meget lig insert statement, med den meget vigtige del er, at man skal huske at bruge WHERE, så man ikke opdaterer alle rækkerne i en tabel, det ses fx i update metoden fra MotorhomeRepository:

```
108 @Override
109 @ public boolean update(Motorhome motorhome) {
110     try{
111         String update = "UPDATE motorhomes SET brand=?,model=?,timesUsed=?,kmDriven=?,activeState=?,typeId=? WHERE motorhomeId=?";
112         PreparedStatement updateStatement = connection.prepareStatement(update);
108 @Override
109 @ public boolean update(Motorhome motorhome) {
110     try{
111         String update = "UPDATE motorhomes SET brand=?,model=?,timesUsed=?,kmDriven=?,activeState=?,typeId=? WHERE motorhomeId=?";
112         PreparedStatement updateStatement = connection.prepareStatement(update);
```

som ses i eksemplet ovenfor, skriver man update, den tabel man vil opdatere, også sætter man de værdier man gerne vil ændre. Her er WHERE motorhomeId=? Den vigtigste del af linjen, fordi uden den ville den her metode opdatere ALLE rækker i hele motorhomes tabellen med de nye værdier. Hvis man kommer til det, er det godt innoDB har mulighed for rollbacks, hvis altså man slår det til... Hvilket vi ikke har gjort...

Delete:

Delete er den SQL-statement der fjerner en række fra en tabel i databasen, det skal helst ikke forveksles med drop statement, da der er væsentlig forskel. Når man bruger en delete statement skal man, på samme måde som i update, være meget opmærksom på, at man kun delete lige præcis den række man gerne vil, og altså ikke alle de andre. Derfor er det også vigtigt at huske den rigtige WHERE-statement. Et eksempel på en delete metode findes ved homeReturnedByMotorhomeId metoden, der findes i klassen ActiveMotorhomeRepository:

```
188 public boolean homeReturnedByMotorhomeId(int motorhomeId){
189     try{
190         String deleteFromCustusemotor = "DELETE FROM custusemotor WHERE motorhomeId=?";
191         PreparedStatement statement = connection.prepareStatement(deleteFromCustusemotor);
192         statement.setInt( parameterIndex: 1,motorhomeId);
193         statement.executeUpdate();
```

DQL:

DQL står for Data Query Language, og er select statementen. Det er brugt til at trække data ud fra tabellerne i databasen. Select bliver brugt meget ofte i vores projekt, også i forbindelse med

funktioner, som giver os behandlet data i stedet for rå data som man ville få med `select * from xx`. Et eksempel hvor vi henter behandlet data, og ikke information direkte fra cellerne vil være metoden `getCancelPrice` i `activeMotorhomeRepository`, hvor vi trækker forskellen på 2 datoer:

```
211 String getCustusemotor = "SELECT extraPrice,DATEDIFF(CURRENT_DATE,startDate) FROM custusemotor WHERE rentId=?";
```

Her får vi prisen, som bliver læst fra cellen, og forskellen mellem start dato på lejeperioden. Det skal bruges til at udregne hvor meget af prisen man skal betale, baseret på hvornår man aflyser lejeperioden.

Konklusion (Alle):

Grundet størrelsen af vores projekt kan det være svært at konkludere hele projektet i en beskrivelse. Dog kan vi konkludere for hver del. Kigger vi på ITO delen først, kan vi konkludere at NMH nemt kan udvide sig og vokse, og med en optimering af den store mængde papirarbejde som de formentligt havde, kan det nu gå endnu hurtigere. Vores applikation er forhåbentlig et skridt i en digitaliserende retning for NMH. Software design delens indflydelse på vores endelige produkt er også unægtelig, vi har udredt forskellige krav inden vi er gået i gang og vi har så ved hjælp af dette opbygget vores program. Selve konstruktionen af applikation er som sagt stærkt påvirket af design delen, men også meget af de krav der er givet af NMH. Vi har efter egen mening opfyldt alle de krav vi har fået stillet og er selv tilfredse med det produkt der er lavet. En overordnet konklusion af projektet er at vi har fået lavet den applikation vi skulle og har med den lavet en lang række dokumentation af hele processen, som forhåbentlig kan give mere indsigt i applikationen og dens fordele og ulemper.

Kildeliste:

Link til vores github: <https://github.com/Nicklas-homies/Nordic-Motorhome-project>

Link til sted som kan hoste SQL online: <https://www.gearhost.com/>

Link til ikoner brugt på webapplikation: <https://fontawesome.com/>

Link til HTML menu: <https://css-tricks.com/solved-with-css-dropdown-menus/>

Link til baggrundsbillede: https://unsplash.com/photos/-Avc2AiE1_Q

Link til table HTML: https://www.w3schools.com/html/html_tables.asp

Link til CSS buttons: https://www.w3schools.com/css/css3_buttons.asp

Design Class Diagram (bilag 1)

