Nordic Motorhome Project

2. Semester eksamensprojekt

Mads Kjærgaard Christensen (10-03-1996)

Michael Sungsoo Fuglø (24-07-1996)

DAT19B
KEA – Københavns Erhvervsakademi
Afleveret 04-06-2020

GitHub Repository
https://github.com/StortM/2-semester-eksamensprojekt

Indholdsfortegnelse

Introduktion (MF)

I denne rapport vil vi gennemgå vores projektgruppes udviklingsproces fra analyse til implementering af et system. Systemet er lavet til Nordic Motorhome Rental, der er et autocamperudlejningsfirma, som har til ønske at centralisere og digitalisere deres udlejning proces. Projektforløbet er blevet foretaget med løbende kontakt med David Ema, der har ageret kunde under dette projekt.

I rapporten vil vi gennemgå, hvordan vi har valgt at tackle denne problemstilling, hvilke værktøjer vi har valgt at anvende og hvorfor, samt gennemgå vores tankegange under udviklingsprocessen.

Samarbejdskontrakt kan findes i appendix 2.

Værktøjer (MC)

Når det kommer til værktøjer inden for softwareudvikling, er der et bredt udvalg at vælge imellem. Vi har i vores arbejdsprocess gjort brug af værktøjer, som vi er blevet introduceret for i forbindelse med vores undervisning, samt værktøjer vi mente kunne hjælpe os bedst muligt til at komme bedst igennem projektet.

Github

Github er en platform som gør det muligt for udviklere at hoste deres projekt, samt arbejde versioneret på det. Dette opnås via Version Control i form af Git.[1]

En af de smarte features ved Github er, at det er muligt at arbejde på en fælles kodebase, men uafhængigt af andre udviklere.
Det gør man ved at lave en lokal branch af et projekt, hvorefter man kan arbejde på de features man ønsker at implementere og derefter merge den lokale branch med master branchen.

Det er desuden muligt at tilføje kommentarer når man "committer" til projektets master branch, så man kan holde styr på hvilke ændringer der er lavet under hvert commit.
Derudover er en central del af Github som tidligere nævnt Version Control. Det betyder at eventuelle dårlige ændringer ville kunne rulles tilbage.

På den måde har Github givet os en stor fleksibilitet og sikkerhed, hvilket har været grunden for at vi valgte denne platform til at håndtere vores kodebase.

Discord

Discord er en gratis VOIP-applikation, som vi begge kender fra vores hverdag.
På Discord er det muligt at oprette grupperum, hvor vi har haft fri mulighed for at kommunikere via tekst og tale, dele filer og tanker og generelt bare bruge programmet som en standard kommunikationsplatform.

I og med at vi begge kendte programmet, synes det er nemt at bruge og havde brug for en måde at kommunikere under projektforløbet, fandt vi valget af Discord meget naturligt og ligetil.

Google Drive

Google Drive er en fil lagrings- og synkroniseringstjeneste udviklet af Google, som vi begge har brugt før, i forbindelse med andre projekter.

[1] https://kinsta.com/knowledgebase/what-is-github/

Tjenesten gjorde det muligt for os at dele dokumenter, diagrammer og filer på en mere overskuelig og permanent basis end Discord, samt gav os et fælles sted at opbevare alt materiale relateret til projektet. Google Drive har desuden den mulighed at flere personer kan redigere et dokument samtidig, hvilket har været en utroligt nyttig feature for os, blandt andet i forbindelse med udarbejdning af use cases samt denne rapport.

Trello

Trello er et online organiseringsværktøj hvor man har mulighed for at oprette et "board" og i det board danne lister og tilføje "kort" til disse lister. På den måde får man et organiseret overblik over eksempelvis opgaver der skal laves, opgaver der er i gang og opgaver som er løst. Til disse opgaver er det muligt at tilføje medlemmer, og på den måde kan man nemt holde styr på, hvor langt en opgave er nået samt hvem, der har ansvar for de forskellige opgaver.[2]

I projektforløbet benyttede vi 4 lister som var henholdsvis "To Do", "In Progress", "Revisit" og "Done".

Vi forsøgte så vidt muligt at holde boardet opdateret hver dag for på den måde altid at være på samme side omkring hvor langt vi var med en given opgave og hvem der havde ansvaret for den.

IntelliJ IDEA

IntelliJ IDEA er en IDE (Integrated Development Environment), eller integreret udviklingsmiljø, som er udviklet af JetBrains og bruges til udvikling af computersoftware.

Vores primære motivation for at vælge IntellIJ var, at vi i forvejen kendte programmet og synes det er godt at arbejde med. IntelliJ har dog en masse gode features som hjalp os i projektforløbet. F.eks. integreringen med Github, som gjorde det muligt at oprette commits og push/pull direkte fra vores IDE, eller integrationen med Spring og Spring Boot som gjorde, at vi nemt kunne oprette et nyt Spring Boot projekt og hoste projektet via en Tomcat server, uden at skulle bruge for meget tid på konfiguration.

MySQL Workbench

MySQL Workbench er udviklet af Oracle og er et visuelt databasedesignværktøj, der integrerer SQL-udvikling, administration, databasedesign, oprettelse og vedligeholdelse i et enkelt integreret udviklingsmiljø til MySQL-databasesystemet.

Vi benyttede Workbench til at designe vores schema i form af et ER-diagram, lave et SQL script til at opsætte det designede schema og et SQL script til at indsætte testdata som vi

[2] https://trello.com/tour

kunne bruge i forbindelse med udviklingen af systemet. Derudover var programmet særligt brugbart når vi skulle kode vores Repository lag for at teste SQL queries inden vi brugte dem i vores kode.

Visual Paradigm

Visual Paradigm er et industri anerkendt UML-værktøj, som gør det muligt at designe og modellerer diagrammer via godkendt UML-notation.

Programmet er blevet introduceret til os i forbindelse med vores undervisning, og vi mente derfor det var et oplagt valg da vi begge har erfaringer med det.

I projektforløbet brugte vi programmet til at designe og modellere alle diagrammer, som krævede UML-notation såsom vores domæne model, klassediagram og sekvensdiagrammer.

Diagrams.net

Diagrams.net (tidligere draw.io) er et online værktøj til design og modellering af eksempelvis flowcharts og diagrammer.

Vi fandt værktøjet i projektforløbet, da vi under konstruktionen af vores ene controller følte det var svært at holde overblik over de mange metoder og selve flowet i controlleren. Vi havde brug for et letvægts-værktøj, som hurtigt og nemt at kunne lave et diagram, der kunne visualisere flowet, og valgte derfor ikke at bruge Visual Paradigm, men søgte andre veje. Værktøjet gav en god oplevelse af ease-of-use, og gjorde det nemt hurtigt at sætte et flowchart sammen, som kunne give os det manglende overblik, og hjalp os meget til at udvikle vores største controller klasse.

Balsamiq Wireframes

Balsamiq Wireframes er et værktøj til hurtig oprettelse af et mockup af graphical user interfaces til softwareapplikationer.
Vi blev introduceret til dette værktøj i undervisningen til hurtig oprettelse af digitale paper prototypes, og vi så det derfor som oplagt for os at bruge under projektforløbet.

Programmet hjalp os især under vores indledende krav og udarbejdning af use cases, da det hjalp os med at visualisere det “flow” som f.eks. oprettelsen af en ny booking skulle følge.

Frameworks og teknologier (MC)

Spring

"Hovedkomponenten" i vores applikation er Spring frameworket, Spring Boot og de dependencies vi har valgt såsom Spring Security og Spring Web.

Spring kan beskrives som både et udbredt framework til Java og en inversion of control container. Modsat et library, som i bund og grund er en samling af funktioner som en klient kan kalde, hvoraf man kan sige at librariets funktioner indgår som en del af klient koden, hvor klient koden har kontrollen, er det i inversion of control lige omvendt. Du skriver som udvikler en stump kode, som kan indgå i og blive kaldt af frameworket - når det skal bruges. [3]

Det betyder selvfølgelig at man skal lære frameworket at kende for at kunne arbejde med det, men det giver også mange fordele. En af fordelene er, at man kan opnå en meget lav kobling da måden hvorpå Springs inversion of control fungerer er via dependency injection.

En dependency er f.eks. at objekt A indeholder en instans af objekt B og derfor har objekt A, objekt B som en dependency. I helt almindelig java kode ville man instantiere objekt B i A klassen, men via dependency injection er det ikke nødvendigt. Her kan man blot angive variablen af type B og lade være med at instantiere den. Det håndterer, i dette tilfælde, Spring frameworket. På den måde er der altså skabt en lavere kobling end uden dependency injection, da det nu er muligt nemt at fortælle frameworket hvilken implementation af typen B som der ønskes at bruge som objekt.

I vores tilfælde er den største motivation for at bruge Spring dog ikke dens IoC implementering, men nærmere de dependencies der kan tilføjes til et Spring projekt. Heraf Spring Web som har gjort det muligt for os nemt at kunne opsætte en webapplikation som følger MVC pattern.

Spring Boot

Spring Boot er en del af Spring Frameworket, og er en teknologi vi har valgt at bruge, da det er hvad vi har arbejdet med i løbet af semesteret, og da det gør det at udvikle i Spring frameworket meget nemmere.

Spring Boot agerer som en form for mellemmand imellem selve frameworket og udvikleren. Det skal forstås på den måde, at Spring Boot håndterer den konfiguration og opsætning af Spring, som man som udvikler ellers selv ville skulle foretage. Det er selvfølgelig stadig

[3] https://www.youtube.com/watch?v=hQn9Z6bVggk (Spring Framework Guru - Inversion of Control and Spring)

muligt at konfiguere Spring via application.properties filen, eller via annotationer, men via Spring Boot får man et out-of-the-box konfigureret Spring miljø som er klar til brug.[4][5]

Spring Security

Spring Security blev tilføjet som en dependency tidligt i projektet. Det gjorde vi, da vi så en klar fordel i at designe vores system ud fra muligheden for at kunne logge ind. Dette skyldtes både virksomhedens struktur, hvor forskellige ansatte ville have forskellige funktioner i systemet, men også at den udleverede projektbeskrivelse nævnte at autocamper lejeren (kunden), i en senere iteration, selv skulle kunne booke en autocamper via systemet. Vi anså det derfor som en central del af systemet og inkluderede det så tidligt som muligt.

Spring Security indeholder meget forskellig funktionalitet, men i vores system bruges det primært som en form for filter. Filteret fungerer på den måde, at vi konfigurerer Spring Security med en række parametre, som definerer hvilke mappings der kan tilgås af hvilke roller. En rolle (eller role) kan beskrives som en brugertype, eller et brugerniveau. En bruger i systemet kan få tildelt en rolle, som eksempelvis kunne være “SALG”. Ud fra denne rolle, kan vi sørge for at det er muligt for denne bruger at tilgå de sider, som er nødvendige for at kunne varetage en salgsassistents funktioner, men også begrænse adgangen til sider med funktionalitet, som brugeren ikke skal have adgang til. Eksempelvis administrative funktioner som at oprette nye brugere i systemet, eller ændre en eksisterende brugers rolle.

Spring Web

Spring Web var en nødvendig dependency, da det var et krav at vores applikation skulle følge en logisk tre lags arkitektur, som kunne køre på en Tomcat server. Via Spring Web var det muligt for os at opfylde disse krav samt benytte MVC-pattern (Model-View-Controller), som er Springs tilgang til webapplikationer. Via Spring Web og MVC-pattern har det været muligt for os at skabe en klar adskillelse mellem web indhold (i form af views), dirigering af trafik (via controller klasser og metoder heri) og integrering med back-end service og data, som kunne tilknyttes til viewet (i form af model objektet).[6]

Nærmere forklaring og eksempler på MVC i vores applikation vil kunne findes i et senere afsnit.

Thymeleaf

Thymeleaf er en java template engine, som gør det muligt at koble back-end data sammen med en template (i vores tilfælde HTML filer), og via template engine producere et view (en html fil), som kan vises til brugeren.

---

[4] https://dzone.com/articles/understanding-the-basics-of-spring-vs-spring-boot (læst 02-06-2020)
[5] https://www.youtube.com/watch?v=Ch163VfHtvA (set 01-06-2020)
[6] https://spring.io/projects/spring-ws#overview (læst 02-06-2020)

Thymeleaf er et kernekomponent i vores applikation, da vi uden det ikke ville være i stand til at hente data fra vores database og bruge det i vores views. Det bliver blandt andet brugt til at vise eksisterende kunder, bookinger og autocampere i systemet, samt til at sende værdier fra viewet til controlleren.[7]

Bootstrap 4

Da det var et krav til vores applikation, at der skulle bruges CSS til at style vores websider overvejede vi forskellige muligheder. Enten at skrive alt CSS selv, eller benytte et CSS framework, som kunne hjælpe os. Vi var mest interesserede i tanken om at bruge et framework, da vi mente det kunne give os et godt fundament at bygge på. En anden klar fordel var, at det kunne sørge for at vores applikation var kompatibel på tværs af web browsere og forskellige enheder med forskellige viewports.

Grunden til at vi ønskede at kunne skabe kompatibilitet på tværs af enheder var, at det blev nævnt i projektbeskrivelsen, at firmaets kunder i en senere iteration skulle kunne tilgå hjemmesiden og selv booke autocamperer. Dette fremtidige krav mente vi derfor var nemmere at kunne efterkomme, hvis vi så tidligt som muligt begyndte at tænke det ind i applikationen. Alternativt ville man kunne udvikle en mere kundeorienteret side i senere iterationer med fokus på brugervenlighed og multi-device kompatibilitet og beholde den mindre responsive desktop-designede applikation til internt brug for medarbejdere i firmaet.

Vi overvejede forskellige frameworks såsom Foundation (som bl.a. bruges af Facebook) og Bulma. Vi endte dog med at vælge Bootstrap, da det virkede til at være mest udbredt og efter sigende skulle have omfattende dokumentation tilgængelig. Især det at kunne hente hjælp og have et væld af dokumentation tilgængeligt virkede som et af de vigtigste parametre, da CSS er relativt nyt for os begge. Deraf faldt valget på Bootstrap.

Bootstrap er kort fortalt et CSS framework, som begår sig på at være responsivt, samt har en mobile-first tilgang. Mobile-first vil sige, at man starter med at designe sin webside efter en mindre viewport (som en mobil), og gradvist udvider til større viewports. Dette skyldes at der oftest er flere restriktioner på de mindre viewports, og at ikke alle funktioner fungerer her. På den måde sikrer man sig, at ens design virker efter hensigten, og at mere avancerede browserfunktioner anvendes på de større viewports (som en PC), hvor de vil virke efter hensigten. Det skaber kompatibilitet, og gør at ens webside fungerer efter hensigten uanset enheden, der tilgår den.[8]

Trods at frameworket er designet med denne filosofi i mente, valgte vi udelukkende at designe vores side efter en almindelig desktop viewport. Selvom det er modstridende med vores krav om at applikationen skulle fungere optimalt på tværs af enheder, mente vi at det at skulle sørge for at applikationen var optimeret til alle enheder, var en for stor opgave. Især med tanke på, at vi begge er nye indenfor CSS. Dog skal det nævnes, at selvom vi ikke har designet vores websider specifikt efter mobile-first princippet, så opnår vi stadig en vis grad

[7] https://www.thymeleaf.org/faq.html#is-web-framework (læst 02-06-2020)
[8] https://medium.com/@Vincentxia77/what-is-mobile-first-design-why-its-important-how-to-make-it-7d3cf2e29d00 (læst 02-06-2020)

af multi-device kompatibilitet, da frameworket i sig selv er lavet på en måde, så elementer reagerer responsivt på viewportens størrelse.

Maven

Maven er et build automation tool, som primært bruges til Java projekter. Et build tool automatiserer dannelsen af eksekverbare applikationer ud fra source code og dets opgaver involverer blandt andet at hente dependencies, compile source code, pakke buildet og gøre det klar til at kunne køres.[9][10][11]

Når et Spring Boot projekt dannes, kan man både vælge at det skal dannes som et Maven eller Gradle projekt. Og da vi kun har kendskab til Maven gennem vores undervisning, var dette et oplagt valg.

En stor fordel ved at bruge Maven i et projekt som vores er, at det kan holde styr på alle dependencies. Det hjælper utroligt meget, da det gør, at vi på meget nem vis kan tilføje nye dependencies, holde styr på eksisterende og eventuelt fjerne eksisterende. Nye dependencies bliver automatisk downloadet, og ændringer i eksisterende dependencies, som at de slettes, håndteres også automatisk af Maven.

Maven projekter konfigureres via en pom.xml fil (Project Object Management). I denne POM fil kan der bl.a. stå informationer om projektet, såsom navn og version, hvilken java version som source code skal compiles med samt hvilke dependencies der er tilknyttet projektet. Det er altså POM filen, som bruges til at kommunikere med Maven hvorefter Maven håndterer resten.

JDBC

JDBC (Java Database Connectivity) er et Java API, som gør det muligt at forbinde til og eksekvere queries og updates mod en database. Dette sker via en JDBC driver, som i vores tilfælde er MySQL Connector/J driveren som virker mellem en MySQL database og Java sproget. Det er altså via JDBC at vi henter, gemmer, og ændre eksisterende data i vores MySQL database.[12]

[9] https://stackoverflow.com/questions/7249871/what-is-a-build-tool (læst 02-06-2020)
[10] https://maven.apache.org/what-is-maven.html (læst 02-06-2020)
[11] https://en.wikipedia.org/wiki/Build_automation (læst 02-06-2020)
[12] https://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html (læst 02-06-2020)

Problemstilling (MF)

Nordic Motorhome Project har endnu ikke centraliseret og digitaliseret deres ansattes opgaver og selve udlejningsprocessen af autocampere. Dette består bl.a. af at oversigter og dokumenter er en sammenblanding af forskellige medier, herunder Microsoft Excel, Microsoft Word og papir. De ønsker derfor at have et centraliseret system.

Overordnet set skal systemet kunne håndtere fem brugertyper, hvoraf en af disse vil være en autocamper lejer, men dette skal systemet først kunne understøtte i en senere udgave. De resterende fire vil være:

- Administrator - håndterer administrativt arbejde, herunder rettelse af priser, sæsoner, oprettelse af ansatte osv.
- Salgsassistent - tager sig af udlejninger af autocampere
- Bogholder - tager sig af budgettering og betalinger
- Servicepersonale, herunder rengøringspersonale og automekanikere. Disse skal være i stand til at kunne notere afleverede autocampere som serviceret.

Systemet har til formål at gøre disse ansattes hverdag nemmere ved at strømline og centralisere alle deres opgaver i et system. Endeligt skal systemet være i stand til at køre i en web browser.

Projektafgrænsning (MF)

Grundet projektets store karakter, har vi valgt at afgrænse projektets første udgave til udelukkende at omhandle salgsassistentens opgaver, altså selve udlejningen af autocampere. Denne del af systemet sås som en af de mest centrale og vigtige dele af systemet, og samtidig også en af de opgaver, som kunne effektiviseres mest muligt ved det nye system, da denne kræver information fra mange dele af domænet.

Denne afgrænsning viste sig dog at være for bred, da vi ikke nåede at implementere dele af bookingprocessen i det endelige system. Dette var noget, som vi desværre indså for sent i udviklingsprocessen, hvilket afspejles i vores design diagrammer og databaseopbygning, da de altså dækker over flere funktioner end vores system indeholder. Vi valgte dog at beholde designelementer, der ikke er implementeret, da de indeholder mange overvejelser og tanker, om hvordan de ikke-implementerede funktioner skal opbygges i senere iterationer. Hvilke dele af designet, der ikke er implementeret, vil blive understreget i den givne sektion.

Unified Process (UP) (MF)

Unified Process, herefter omtalt UP, er en agil og objektorienteret softwareudviklingsprocess, som har til formål at skærpe kundens ofte udviklende krav under udviklingsprocessen. Dette gør den ved at være iterativ og inkrementel. Software bliver altså inddelt i timeboxede iterationer, hvor det er muligt at opbygge et stykke software trinvist. Herved er det altså muligt at mindske risikoen ved processen, ved at være så proaktiv som

muligt. Herudover er UP use case drevet, hvilket betyder, at use cases er kernen i udviklingen, som derfor bruges som fundament for analyse, design og programmering.

I UP opdeler man udviklingsprocessen i fire faser:

- **Inception:** Denne fase er den mindste i projektet, og har til formål at danne overblik over projektets omfang og identificere nøglefunktioner til systemet. Heri analyseres selve organisationen også. Herved bliver det muligt at finde ud af, om projektet overhovedet kan lade sig gøre.

- **Elaboration:** I denne fase analyseres domænet, kravene indskærpes og en grundlæggende arkitektur for programmet fastsættes, for at danne et overblik over systemet. Kravene vurderes og adresseres herfra og bliver prioriteret ud fra deres risikoniveau.

- **Construction:** Denne fase er præget af et markant større fokus på selve skrivningen af systemet. I denne iteration bliver design klassediagram og sekvensdiagrammer også færdigudviklet i takt med systemet, så det er muligt for projektgruppen at holde overblik over systemet og som et visuelt hjælpemiddel.

- **Transition:** I denne fase bliver systemet implementeret hos kunden, hvilket ikke har været en faktor under dette projekt. Derfor udelukkes denne i faseplanen.

Under dette projekt har vi anvendt UP og dets principper til analyse, design og selve kodningen af systemet.

Indledende Krav (MF & MC)

FURPS+

Ud fra den opgivne kravspecifikation samt samtale med kontaktpersonen, har vi fundet frem til følgende kravspecifikation. Disse krav dækker ikke det endelige program, men kun salgsassistentens funktioner jvf. projektafgrænsningen.

Functionality

- Systemet skal have mulighed for login, som udover at være en initiel sikkerhedsbarriere, også sikrer at den ansatte kun har adgang til sit eget ansvarsfelt.
- Det skal være muligt at oprette, redigere, tilgå og slette bookinger.
- Det skal være muligt at oprette, redigere, tilgå og slette kunder, da disse skal tilknyttes en booking.
- Søgefunktion til autocampere for at gøre det mere overskueligt at finde den bedst passende autocamper til kundes behov.

Usability

- Da salgsassistenten/ejeren ikke nødvendigvis er it-kyndige, så skal systemet have en brugervenlig og intuitiv GUI.
- Systemet skal være kompatibelt på tværs af enheder og med de mest populære browsere på markedet da det er påtænkt at kunder senere skal kunne booke autocampere på egen hånd.

Reliability

- Information skal gemmes i en database, for at det er persistent.
- Error handling, i forbindelse med brugerinput, skal sørge for at korrekt type data bliver gemt i databasen.

Performance

- Systemet skal have plads til minimum 32 autocampers, med mulighed for at udvide indenfor hvad databaseteknologien kan understøtte. Da MySQL med InnoDB engine som maksimum understøtter tables på 256TB vil der være rig mulighed for at udvide mængden af autocampers i flåden.[13]
- Søgning i data skal foregå hurtigst muligt. Resultater skal kunne vises indenfor 10 sekunder, 99% af tiden. Derfor skal databasen normaliseres til minimum 3. normalform.
- Størstedelen af sider skal indlæses indenfor 1 sekund. En vis grad af fleksibilitet er tilladt, f.eks. for sider, der udfører store queries.

Supportability

- GRASP principper som loose kobling og polymorfi anvendes, samt seperation of concerns, så der er mulighed for at bygge videre på systemet i senere iterationer.
- Grænsefladen skal være dansk, eftersom firmaet er dansk og andet ønske ikke er specificeret fra firmaets side.

Prototype (MF)

Efter udarbejdelse af kravspecifikationen, lavede vi en paper prototype vha. softwareløsningen Balsamiq Wireframes, der gør en i stand til hurtigt at lave en skabelon af software. Den hovedsagelige motivation for dette var, at vi kunne undersøge om kunden, og vi i projektgruppen, var på samme side, når det kom til vores vision for, hvordan projektet skulle udforme sig.

Dette blev især understreget under udviklingen af prototypen, da den lagde op til en række gode diskussioner om, hvordan vi i gruppen skulle lave det mest brugervenlige og logiske

[13] https://dev.mysql.com/doc/refman/8.0/en/innodb-limits.html

design. Dette førte til at en række store beslutninger skulle tages for workflowet af en bookingoprettelse, som afspejler sig i det endelige system, da det i store træk følger prototypens flow.

Prototypen var til stor hjælp, da vi nu kunne visualisere workflowet for en oprettelse af en booking, samt håndtering af kunder og bookinger. Det agerede også som et referencepunkt til yderligere design og udvikling af diagrammer.

Prototypen kan tilgås via følgende link:
https://drive.google.com/file/d/1B9tPRD-RVQ1J3s9FzOcdI3GNBr8elhy6/view?usp=sharing

Use Cases (MC & MF)

Ud fra de funktionelle krav blev følgende use cases identificeret.

- Søg autocamper
- Håndter autocamper booking
- Håndter kunde

Disse use cases er ikke alle en del af workflowet for oprettelse af en booking, men indebærer stadigvæk salgsassistentens opgaver, f.eks. håndtering af kunder. Af disse use cases valgte vi at beskrive “søg autocamper”, “opret autocamper booking” og “rediger autocamper booking” som fully dressed use cases, da vi vurderede disse som de mest komplekse og udfordrende at implementere. Af disse, har vi valgt at “søg autocamper” og “opret autocamper booking” skal danne fundament for vores design diagrammer. De resterende use cases blev beskrevet i brief format. Disse kan findes i appendix 1.

Fully dressed use case UC1: Opret Autocamper Booking

Use case navn	Opret autocamper booking
Scope	Nordic Motorhome Rental System
Level	User-goal
Primær Aktør	Salgsassistent
Stakeholders og interessenter:	**Salgsassistenten** Ønsker at bookingprocessen skal være nem, intuitiv og så smertefri som mulig. Workflowet skal give mening og være til at lære. Det skal være nemt at oprette nye kunder eller forbinde eksisterende kunder med bookingen. Alle relevante informationer skal blive vist på de

	rigtige tidspunkter så der er et godt overblik, og service kan blive bedst mulig. **Kunden** Ønsker at kunne booke en autocamper nemt og hurtigt. Ønsker kompetent og effektiv service. Ønsker at krav og ønsker såsom bookingperiode, antal senge, tilbehør m.m. kan indfries uden besvær. **Ejer** Ønsker at det er smertefrit at oprette en booking og at fejl håndteres effektivt af systemet og salgsassistenten. **Bogholder** Ønsker at systemet har så lidt prisfejl som muligt.
Preconditions:	En salgsassistent er logget ind i systemet og taler med en kunde som ønsker at booke en autocamper.
Success Guarantee:	Salgsassistenten har oprettet en ny, eller tilknyttet en eksisterende kunde til bookingen. Bookingen er blevet oprettet og kundens ønsker er indfriet bedst muligt. Bookingen er gemt og kan hentes frem senere.
Main Success Scenario:	1. Salgsassistent udfører Søg Autocamper for at finde en autocamper ud fra kundens krav og ønsker. 2. Kunden fortæller hvilken autocamper denne ønsker. 3. Salgsassistenten vælger autocamperen i systemet. 4. Systemet vælger autocamperen og præsenterer salgsassistenten med mulighed for at indtaste kundeinformationer. 5. Salgsassistenten beder kunden om personlig information som fornavn, efternavn, mail og telefonnummer, og indtaster dette samt opretter kunden. 6. Salgsassistenten går videre i bookingprocessen og systemet præsenterer en side med mulighed for valg af tilbehør, pick-up og drop-off point, prisoversigt, kunde- og autocamper information. 7. Salgsassistenten spørger kunden om pick-up og drop-off point samt om ønsker til tilbehør som antal barnesæder, ekstra borde og stole, ekstra sengetøj, cykelstativ. 8. Kunden oplyser de efterspurgte informationer og salgsassistenten indtaster dem. 9. Systemet opdaterer prisoversigten med pris for autocamper pr. dag samt for hele perioden, priser for tilbehør, gebyrer for pick-up og drop-

	off og den totale pris. 10. Kunden accepterer og salgsassistenten fuldfører bookingen. 11. Systemet viser at bookingen er udført og gemmer bookingen.
Extensions:	*a. Hvis et felt på et hvilket som helst tidspunkt udfyldes med ugyldig information: 1. Systemet præsenterer en fejlbesked. 2. Salgsassistenten retter informationen til korrekt format. 6a. Kunden spørger nærmere ind til autocamperen og finder ud af et krav ikke kan opfyldes af den valgte autocamper. 1. Salgsassistenten går tilbage til autocamperlisten og finder en ny autocamper som opfylder kundens behov. 9a. Kunden findes allerede i systemet. 1. Salgsassistenten vælger at finde kunden i systemet. 2. Systemet præsenterer en liste over eksisterende kunder. 3. Salgsassistenten finder kunden på listen ud fra oplysninger som fornavn, efternavn, mail eller telefonnummer. 4. Salgsassistenten tilføjer kunden til ordren. 5. Systemet tilføjer kunden til ordren og går videre i bookingprocessen til valg af pick-up point, drop-off point, tilbehør.

Fully dressed use Case UC2: Rediger Autocamper Booking

Use case name	Rediger autocamper booking
Scope	Nordic Motorhome Rental System
Level	User-goal
Primær Aktør	Salgsassistent
Stakeholders og interessenter:	**Salgsassistent** - Ønsker at opdatering af bookingoplysninger er simpelt og intuitivt, og at informationerne vedr. bookingen bliver vist på en overskuelig måde **Ejer** - Ønsker at opdatering af booking er pålideligt og minimerer fejl i betalingsregistreringer **Kunde** - At booking samt at opdateringen forløber med så få fejl som muligt **Bogholder** - Ønsker at ændriner bliver registreret korrekt og evt. fejl i priser bliver minimeret
Preconditions	En salgsassistent er logget ind i systemet. Booking er oprettet og fundet i systemet. Salgsassistenten snakker med kunden vedr. opdatering af booking
Success guarantee	Booking er blevet opdateret af salgsassistenten, evt. prisændring er udregnet og opdateringen er bekræftet og gemt i systemet.
Main success scenario	1. Salgsassistenten beder systemet om at vise flere detaljer på den pågældende booking 2. Systemet præsenterer en oversigt over booking-, bil- og kundeinformation og tilbehør samt en prisoversigt, der inkluderer total pris, tilbehørsoversigt og -pris, autocamper pris pr- dag og evt. pickup- og drop-off gebyrer. 3. Salgsassistenten retter antallet af barnesæder fra 2 til 1. 4. Prisoversigten opdateres 5. Salgsassistenten bekræfter opdateringen 6. Salgsassistenten får bekræftelse om at den pågældende booking er blevet opdateret succesfuldt
Extensions	*a Hvis et indtastet felt er udfyldt med ugylidge informationer: 1. Systemet prompter salgsassistenten om at ugyldig information er indtastet 2. Salgsassistenten retter informationerne til et korrekt format 3a. Salgsassistent ønsker at rette bookinginfo;

	bookingperiode og pick- og drop-off point 1. Salgsassistenten retter pick-up point. 2. Salgsassistenten retter drop-off point. 3. Salgsassistenten retter fra- og til datoen 3a. Den nye bookingperiode er ugyldig 1. Systemet prompter salgsassistenten om den ugyldige periode. 3b: Salgsassistent ønsker at rette kundeinformation; fornavn, efternavn, email, telefonnummer og adresse 1. Salgsassistenten retter fornavn 2. Salgsassistenten retter efternavn 3. Salgsassistenten retter telefonnummer 4. Salgsassistenten retter adresse 3c: Salgsassistent ønsker at ændre booket autocamper 1. Salgsassistenten vælger at ændre autocamper 2. Salgsassistenten får vist en liste af tilgængelige autocampere i den bookede periode. 3. Salgsassistenten vælger den nye autocamper, der skal bookes 4. Systemet viser nu den opdaterede information i form af: model, navn, antal sæder, senge og prisoversigten opdateres
Special requirements	-
Technology and data variations list	-
Frequency of occurrence	Kan gøres kontinuerligt.
Miscellaneous	-

Fully dressed use Case UC3: Søg Autocamper

Use case name	Søg autocamper
Scope	Nordic Motorhome Rental System
Level	User-goal
Primær Aktør	Salgsassistent
Stakeholders og interessenter	Salgsassistent - Ønsker at søgning kan foregå nemt og intuitivt med mange funktioner og parametre til søgning, således, at de kan finde den ønskede autocamper hurtigst muligt Kunde - Ønsker at det er muligt for salgsassistenten at finde den autocamper, der passer bedst til dens behov Ejer - Ønsker at salgsassistenten har let ved at anvende systemet, så service og salg foregår så let som muligt
Preconditions	En salgsassistent er logget ind i systemet og taler med en kunde som ønsker at booke en autocamper.
Success guarantee	Salgsassistenten søger efter en autocamper ud fra et sæt kriterier. Systemet filtrerer listen af autocamperer. Salgsassistenten finder den ønskede autocamper.
Main success scenario	1. Kunden fortæller salgsassistenten hvilken periode autocamperen skal bookes i. Salgassistenten indtaster de givne fra og til datoer. 2. Systemet præsenterer en filtreret liste over ledige autocampere for den givne periode.
Extensions	2a. Systemet finder ingen autocampere, da ingen er tilgængelige for den indtastede periode. 1. Salgsassistenten oplyser kunden om dette. 2. Kunden genovervejer den ønskede bookingperiode. 2a. Kunden er ikke tilfreds og ønsker at afbryde salget. 1. Salget afbrydes. 2b. Kunden oplyser nye fra og til datoer. 1. Salgsassistenten indtaster disse og præsenteres for en ny liste af ledige autocampere.

Use Case Diagram (MC & MF)

Det er nu muligt for os at opstille et use case diagram for at få mere indblik i, hvordan vores aktører interagerer med systemet. Vi har medtaget autocamper lejer med som supporting actor, da det antages at salgsassistenten taler med denne under de givne use cases.

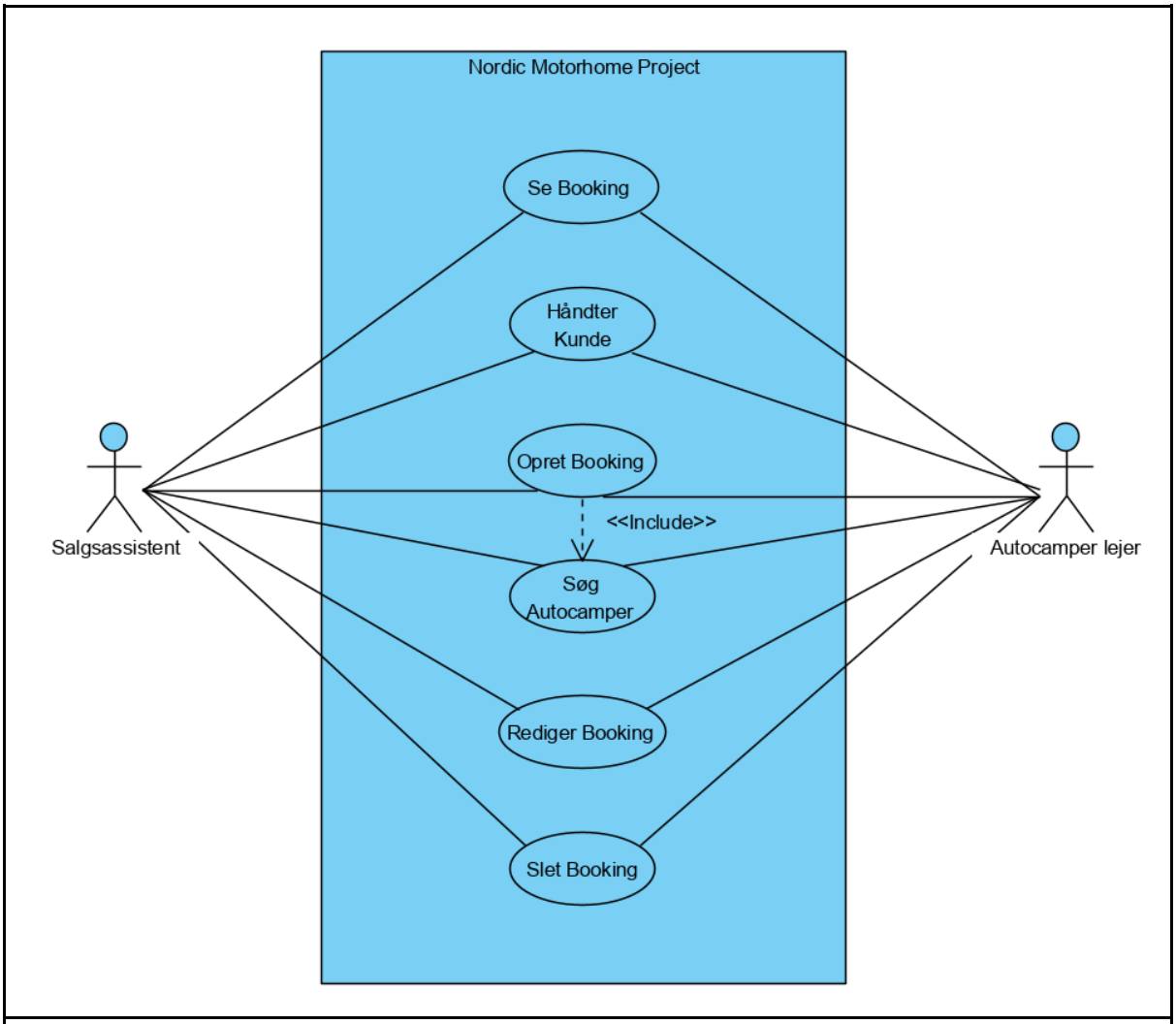


Diagram 1: Use case diagram for Nordic Motorhome Project

Her ses det at vores use case “søg autocamper” er inkluderet i “opret booking” use casen, da denne funktion er en del af workflowet for en booking oprettelse. Dog er det også muligt blot at tilgå “søg autocamper”, hvis aktør ønsker at finde en specifik autocamper uafhængigt af en booking oprettelse. Derfor er der også en direkte association mellem salgsassistenten og denne use case.

Da vi har afgrænset projektet til kun at medtage salgsassistentens ansvarsområde, har vi undladt at tage andre oplagte primære aktører med såsom ejeren og bogholderen, selvom det ville være naturligt, at de også vil kunne tilgå nogle af disse use cases i en senere iteration.

Andre supporting actors som f.eks. “NETS” ville også være oplagte at medtage i en senere iteration, da vores system skal kunne håndtere betalinger, men da betalinger ikke er en del af vores scope endnu, er disse udeladt.

Projekt- og organisationsanalyse

SWOT-analyse (MF)

Vi har udarbejdet en SWOT-analyse efter samtale med kunden, for at præcisere virksomhedens styrker og svagheder, både internt og eksternt. Det giver os et overblik over virksomhedens situation og over hvilke områder virksomheden klarer sig godt eller mindre godt.

Et af de helt centrale punkter i denne SWOT-analyse er coronakrisen, der har ramt virksomheden hårdt. Deres lille størrelse samt et svækket marked, da 50% af deres kunder er turister, stiller dem i en meget udfordrende situation. Dette var også noget, som kunden var bevidst om, hvilket han gav udtryk for ved blot at håbe på, at virksomheden overlevede 2020.

	Positive	**Negative**
Interne forhold	**Styrker** - Stærk og kompetent ledelse - Ledelsen har en stor viden indenfor kerneområdet - God virksomhedskultur - Har vækstet det seneste halvandet år, hvilket har givet økonomisk overskud samt øget virksomhedsmoralen	**Svagheder** - Kunder har ikke mulighed for at booke autocampere igennem et website, men er nødsaget til at skrive/ringe ind til firmaet. Mangel på dette kan i kundens øjne virke uprofessionelt og 'gammeldags' samt afskrække kunder fra at foretage en booking, da det er nemmere ved et online system. - Personale har svært ved at overskue udlejninger samt inventar grundet et decentraliseret IT-system. - Relativt lille mængde autocampere sammenlignet med konkurrenter (85 stk. www.out2camp.dk) - Grundet firmaets lille størrelse, er det mindre robust overfor finansielle kriser såsom coronakrisen
Eksterne forhold	**Muligheder** - Da danskere ikke længere kan rejse udenlands, er der opstået en unik mulighed for at markedsføre overfor det danske marked yderligere, da det antages at danskerne vil lede efter nye feriemuligheder indenfor landets grænser. - Hvis firmaet er i stand til at	**Trusler** - Grundet coronakrisen har virksomheden mistet 50% af dets kunder, da turister ikke længere kan komme ind i landet. - Kundernes købekraft er svækket, da en spekulation om en eventuel finanskrise er opstået.

	overleve coronakrisen, vil konkurrencen formodentlig være formindsket eller svækket - Kompetent udviklingshold, der kan anvende ny teknologi til udvikling af hjemmeside.	- Da grænserne er lukkede, er der muligvis et svækket behov for at leje autocampere, da man kunne forestille sig at lejere ofte tager på længere ture igennem Europa - Stærke konkurrenter med mange års erfaring der allerede har etableret et ry, sammenlignet med firmaets 1,5 års erfaring.

Risikoanalyse (MF)

Risikoanalysen er et godt redskab til at vurdere, hvilke risici der findes for et projekt eller en virksomhed. Herfra er det muligt at opstille risici i en tabel, hvorefter de vurderes ud fra produktet af sandsynligheden for at risikomomentet træder til og konsekvensen af dette. Det er hermed muligt at finde ud af hvilke risici, der kunne true en virksomhed eller et projekts gennemførsel.

I vores tilfælde har vi lavet en risikoanalyse for projektets gennemførsel. Alt i alt blev projektet vurderet til at have en lav risiko for at fejle. Vi tog derfor risikomomenter ved cutoff 6 med i den udvidede risikotabel. Det blev vurderet at det mest truende risikomoment for projektet var medtagelse af nye teknologier, herunder især Spring Framework, da dette ikke var noget, som vi havde arbejdet med i den her skala tidligere.

Udvidet risikotabel							
Risikomoment	**Sandsynlighed**	**Konsekvens**	**Produkt**	**Præventive tiltag**	**Ansvarlig**	**Løsningsforslag**	**Ansvarlig**
Kollektiv sygdom	2	3	6	Sikre at man ved sygdom på forhånd aftaler om et eller flere syge medlemmer mødes med resten af gruppen. Mulighed for at arbejde hjemmefra ved sygdom. Undgå for stor eksponering til andre mennesker for at undgå Corona-smitte.	Projektgruppen, det enkelte medlem	Benytte Discord, Trello til kommunikation og koordinering af hjemmearbejde. Sørge for at alt projektmateriale ligger på Google Drive samt GitHub.	Projektgruppen
Gruppemedlem forlader gruppen	1	10	10	Tag fat og løse konflikter når de opstår. Skabe et forum med mulighed for åben kommunikation.	Projektgruppen	Tage kontakt til vejleder. Sørge for at revurdere størrelsen af projektet samt fordele den nye arbejdsbyrde ligeligt.	Projektgruppen
Gruppekonflikt der skaber splid	1	10	10	Tag fat og løse konflikter når de opstår.	Projektgruppen	Tage fat i de involverede i en given konflikt og samlet i gruppen	Projektgruppen

				Skabe et forum med mulighed for åben kommunikation.		få talt problemerne ud og derved få konflikten løst.	
Mangelfuld afgrænsning af projekt	2	3	6	Kontinuerlig kontakt med vejleder. Løbende i projektets forløb sammenligne opgaven og afgrænsningsbeskrivelsen med projektbeskrivelsen.	Projektgruppen	Vurdere omfanget af eventuelt ekstraarbejde og lave det, eller afgrænse projektet yderligere og skære ned på omfanget af det endelige resultat.	Projektgruppen
Tab af essentielle projekt filer	1	10	10	Vi gemmer løbende filer på Google Drive. Vi anvender versionskontrol i form af GitHub, for at kunne dokumentere og se udvikling samt som eventuel backup.	Projektgruppen	Tjek hvorvidt der ligger en tidligere version og byg videre på den. Er der ingen tidligere version må den ansvarlige for tabet, eller personen med mest indsigt i området, lave filen igen.	Projektgruppen
Fejlfortolkning af projektkrav	1	7	7	Hvis et medlem i gruppen er i tvivl om hvordan en opgave skal udføres, vil	Projektgruppen	Klargøre præcis hvad der er blevet	Projektgruppen

				det blive taget op i gruppen. Hvis gruppen er i tvivl, vil vi kontakte en vejleder.		fejlfortolket. Herefter må hvert gruppemedlem rette i de filer som vedkommende er ansvarlig for.	
Manglende evne til at anvende Spring eller andre nye teknologier til at implementere de opstillede krav for projektet	2	7	14	Vi sætter ekstra tid af projektet til at udvide forståelse og kompetencer indenfor Spring frameworket og andre nye teknologier.	Projektgruppen	Yderligere research. Konsultation med fagkyndige (Nicklas). Alternative måder at løse en given opgave.	Projektgruppen

Interessentanalyse (MC)

En interessentanalyse omhandler interessenter, der af Lars Zwisler defineres som:

"Personer eller persongrupper, som har en interesse i dit projekt, bliver påvirket af det eller som kan have indflydelse på dets resultat".

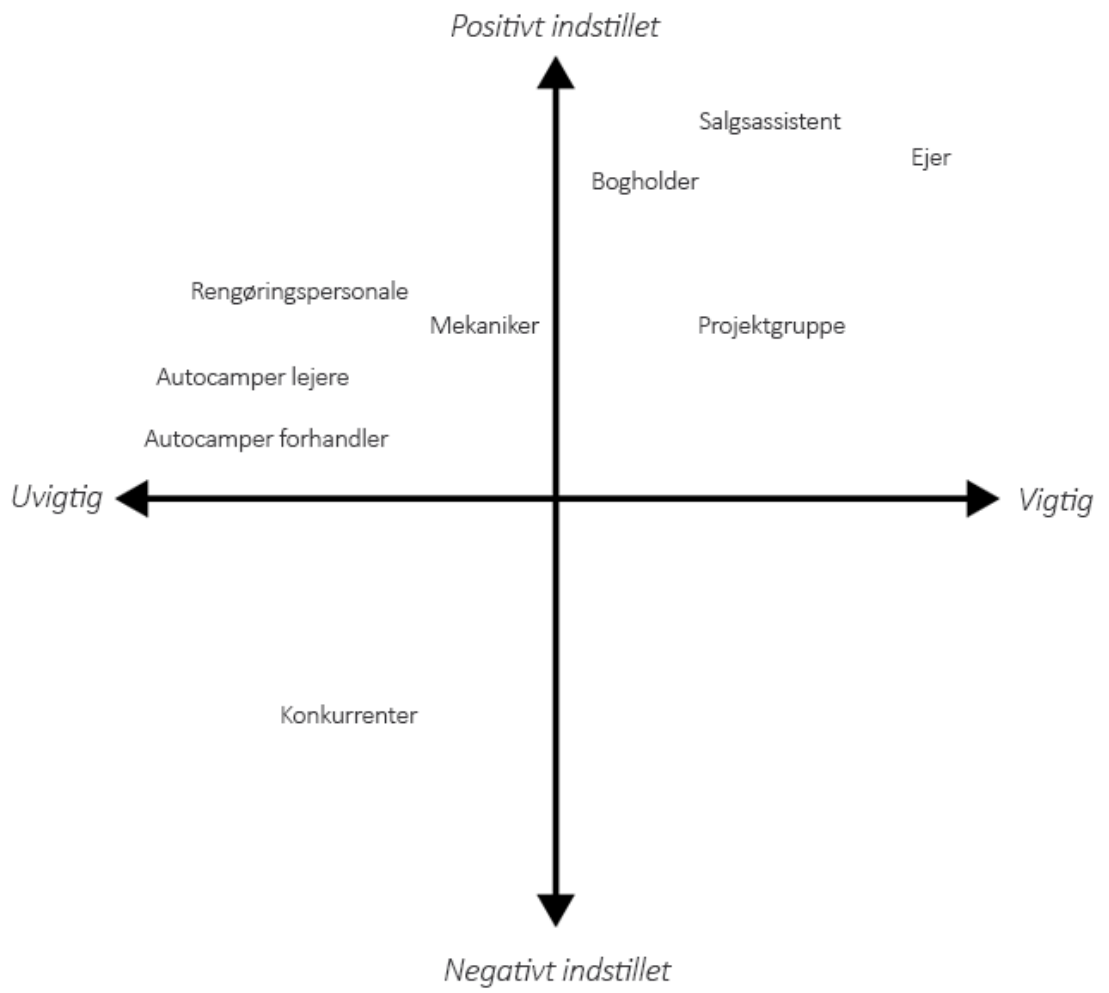
Denne type af analyse, der er god til at give udviklingsholdet en fornemmelse for, hvilke personer eller grupper, der skal holdes særligt øje med, og hvem der ikke behøver at holdes orienteret om projektudviklingen.

Vi startede med at identificere alle interessenter, ved at stille os selv en række spørgsmål som f.eks. "Hvem får gavn og udbytte af projektet?" og "Hvem berøres, generes eller lider afsavn af projektet". Ud fra dette fandt vi følgende liste af interessenter:

Interessenter:

- Ejer
- Salgsassistent
- Rengøringspersonale
- Mekaniker
- Bogholder
- Autocamper forhandler
- Autocamper lejere (firmaets kunder)
- Konkurrenter
- Projektgruppen

For at klarlægge interessenternes indflydelse med henblik på at identificere nøgleinteressenter, lavede vi nu en kategorisering af interessenter:



Billede 1: Kategorisering af interessenter

Ud fra dette var det nu muligt for os at identificere følgende nøgleinteressenter og deres succeskriterier:

Ejer

- Ansatte er tilfredse med systemet
- Systemet effektiviserer arbejdsprocesser
- Systemet minimerer antallet af fejl fra ansatte
- Systemet er brugervenligt og nemt at lære
- Systemet fungerer uden eller med acceptabel mængde fejl
- Systemet opfylder de opstillede krav
- Virksomheden har mulighed for at vokse ind i systemet, f.eks. i form af flere ansatte som bruger det i fremtiden.
- Projektet overholder tidsplanen

Salgsassistent

- Systemet er brugervenligt, intuitivt og nemt at lære
- Systemet gør det nemmere at arbejde
- Systemet er overskueligt og giver et godt overblik over udlejninger, ledige autocampere m.m.
- Systemet er hurtigt og fungerer med minimalt antal fejl

Projektgruppen

- Projektarbejdet forløber godt
- Projektgruppen fungerer godt
- Krav til systemet er godt beskrevet
- Krav ændrer sig mindst muligt og nye features eller ønsker til systemet bliver kommunikeret klart og tydeligt og gerne så tidligt i projektets forløb som muligt.
- Kunden er tilfreds med systemet
- Kunden er god til at komme med feedback.

Vi kunne nu opstille vores nøgleinteressenter i en tabel, så vi kunne få et overblik over dem og deres indvirken på projektet. Vi kunne bruge denne tabel som et referencepunkt for, hvornår nøgleinteressenter skulle informeres om ændringer, fremskridt eller hvis større beslutninger skulle tages:

Interessentanalyse						
Interessenter	**Deres mål**	**Tidligere reaktion**	**Hvad der kan forventes**	**Indvirkning +/-**	**Mulig fremtidig reaktion**	**Ideer**
Ejer	Systemet lykkedes og virker efter hensigten.	Spændt over at få et IT system op at køre Håber på at systemet er skalerbart og kan udvides i fremtiden	Bliver utilfreds hvis projektet bliver forsinket	Vil være mere tilstrækkelig til god feedback såfremt han er positivt stemt Vil føle sig presset og muligvis være mindre samarbejdsvillig såfremt tingene ikke går godt	Kan være tilbøjelig til ikke at bruge systemet såfremt det ikke lever op til forventninger.	Involvere ham i testfaser samt prototyper Ajourføre ham med projektet undervejs i projektforløbet Vi vil være åbne overfor feedback og idéer fra ham
Salgsassistent	Systemet hjælper til at gøre arbejdet nemmere	Træt af at arbejde med et decentraliseret system og papir	Bliver frustreret og demotiveret hvis systemet ikke er brugervenligt og derfor en tilbøjelighed til ikke at acceptere det nye system	Kan komme med idéer og indsigt hvis de er positivt stemt over for den retning som det nye system udfolder sig i Dårlige reaktioner eller stemning kan smitte af på ejeren, hvilket kan påvirke kvaliteten/mængden af feedback og samarbejdsvillighed fra hans side	Kan have et ønske om at gå tilbage til det gamle system, såfremt det nye ikke er brugervenligt, gør hverdagen nemmere o.l.	Involvere en eller to salgsassistenter i tests (gerne black-box) og prototyper Få indsigt i deres arbejdsflow, f.eks. via interviews med henblik på at gøre systemet effektivt

Projektgruppe	Systemet virker og ejeren er tilfreds	Nysgerrighed omkring hvilke teknologier projektet kan implementeres via. God fornemmelse for at projektet kan færdiggøres inden for tidsplanen.	Analytisk tilgang til projektet. Opsøger feedback fra ejer. Er omstillingsparate.	Går tingene godt kan systemet opnå en højere grad af kvalitet og muligvis flere features end forventet Går tingene dårligt vil kvaliteten sandsynligvis forringes og flere ønsker til systemet vil blive indfriet. I værste tilfælde vil systemet blive forsinket.	Kunne nægte fremtidigt arbejde med ejer såfremt han viser sig som en udfordrende samarbejdspartner.	Forsøge at skabe en god dialog med ejeren og involvere ham i udviklingen Sørge for at vidensdele omkring nye teknologier og løsninger

Feasibility study (MC & MF)

Et feasibility study er et godt redskab til at mindske tab, hvis nu et projekt skulle fejle, da dette er relativt prævalent for softwareprojekter. Ved at lave et feasibility study 10-20% inde i projektet, kan man konkludere om projektet burde fortsætte eller aflyses. Man forsøger at bevise sandsynligheden for at et projekt vil have succes, ved at finde styrker og svagheder. Disse opdeles i en række kategorier, hvoraf vi ikke har medtaget retlige og juridiske forhold og politiske og planmæssige forhold, da vi ikke har nok viden indenfor disse felter til at diskutere dette. Vi har derfor valgt at kigge markedet, den økonomiske gennemførsel samt de tekniske forhold for projektet.

Indledende analyse

Systemet udarbejdes til Nordic Motorhome Rental, der er et autocamperudlejningsfirma, som blev startet allerede i 2019. Trods den unge alder, har firmaet opnået stor succes, men deres ekspansion hindres af et decentraliseret administrationssystem, der som konsekvens mindsker effektiviteten af de ansatte, øger chancen for fejl og som et resultat af dette forringer kundeoplevelsen. Systemet har til formål at centralisere og strømline arbejdsprocessen, så de ansatte får en mere overskuelig hverdag og profit optimeres.

Firmaet står i en bemærkelsesværdig situation, da deres overgang til et nyt system forekommer under corona-krisen, hvilket vil påvirke 2020's overskud markant. Som et resultat af denne krise, ville udviklingsprocessen af et nyt it-system, under normale forhold, hindres markant, men da systemet udvikles gratis som et eksamensprojekt af studerende på Københavns Erhvervsakademi (KEA), vil dette ikke være en faktor.

Markedsundersøgelse

Digitalisering samt centralisering af data har mange fordele, som Nordic Motorhome Rental vil kunne drage fordele af. Dette er også et medie, som konkurrenter allerede gør sig brug af, da det er muligt at kunne booke autocampere via en hjemmeside hos bl.a. www.out2camp.dk og www.camperworld.dk. Overgangen til det nye system vil bl.a. frigive timer for de ansatte, der ellers ville have gået til administrativt arbejde. Herudover vil de ansatte også kunne give kunden en mere effektiv og hurtig service, da de nu anvender et centraliseret system, hvoraf det formodes at informationer er hurtigere og lettere at tilgå. Dette vil bidrage til en reduktion af fejl, da færre opgaver vil være foretaget af mennesker. Ejeren vil også kunne få et hurtigere overblik over sin organisation, da bookinger, ansatte, kunder osv. alle er samlet i et system.

Alt i alt ses systemet som en meget positiv indflydelse på virksomheden, der vil kunne bidrage til deres vækst.

Økonomisk gennemførsel

Som nævnt, udvikles systemet gratis af studerende på KEA, hvilket betyder at de økonomiske omkostninger for selve udviklingen af systemet vil være nært nul. Det vurderes derfor, at den økonomiske gennemførsel af projektet, vil være meget let givet.

Det skal dog nævnes, at systemet kan medføre en række mindre vedligeholdelsesmæssige samt initiale omkostninger. Hvis firmaet ikke allerede ejer et domæne, skal dette købes og en server opstilles, enten cloud-baseret eller on-premise.

Igennem Amazon Web Services (AWS), kan en server indeholdende Tomcat, Java og MySQL fås til $0.50[14] i timen, hvilket på en måned med 30 dage og en nuværende dollarkurs på 6,72 svarer til lige godt 2.400, - DKK om måneden.[15] Uden nærmere indblik i virksomhedens økonomi vurderes dette til at være en overskuelig omkostning. Det skal dog også nævne at en Cloud-baseret service på sigt nok vil være en dyrere løsning, og at det er uvist om en autocamper virksomhed har brug for at kunne skalere som jo er en af fordelene ved at vælge en Cloud løsning. Derfor kunne virksomheden antageligt drage fordel af en on-premise løsning for at spare penge og eventuelt ansætte en konsulent til at varetage IT support og vedligeholdelse.

Det forventes ikke, at de ansattes computere eller anden hardware skal udskiftes ved håndtering af dette system, da det ikke forventes at behovet for klient computerkraft vil stige signifikant. Til slut skal systemet muligvis vedligeholdes, hvilket selvfølgelig også er en omkostning.

Det forventes dog, at den økonomiske besparelse ved en effektivisering af de ansatte samt bedre service overfor potentielle kunder vil opveje disse relativt små økonomiske omkostninger for udvikling samt vedligeholdelse af systemet.

Tekniske forhold

Som nævnt tidligere, vil hardware formodentlig ikke blive et problem, derudover anvendes der velkendte og konventionelle softwareløsninger, der heller ikke forventes at give nogle problemer.

Første iteration af systemet udvikles på 3 uger af 2 studerende, hvori der skal implementeres to use cases. Kravene er præcise og mulige, hvilket hjælper denne proces. Herudover er kunden let at arbejde med, og viser høj viden indenfor domænet og systemudvikling. Projektet vurderes derfor til at have en *høj struktur*.

Systemet vil blive udviklet ved hjælp af diverse Frameworks, herunder Spring, Spring Web, Spring Security, Bootstrap og Thymeleaf som template engine. Vi som projektdeltagere har mindre erfaring med Spring, Spring Web og Thymeleaf, men Spring Security og Bootstrap er nye, hvilket er en risikofaktor. Herudover har vi ikke arbejdet med nogle af disse Frameworks

[14] https://aws.amazon.com/marketplace/pp/Dongguan-SainStore-e-Commerce-Co-Ltd-Tomcat-Java-M/B00W2LG0RW
[15] https://www.valutakurser.dk/ (læst 02-06-2020)

og teknologier på en større skala. Dette påvirker risikoniveauet for dette projekt markant, og vurderes som dette projektets største risikofaktor. Risikoniveauet vil derfor bemærkes som betydeligt.

Der vurderes dog, at projektets høje struktur samt iterative tilgang til udviklingsprocessen, vil kunne udveje risikoniveauet, og hermed vurdere projektet som teknisk muligt.

Go/No go beslutning

Med udgangspunkt i dette feasibility study for Nordic Motorhome Rental vurderes følgende:

Der findes ingen forlæg for at projektet skal stoppes efter en indledende analyse, og de økonomiske omkostninger anses som overkommelige, i forhold til den økonomiske gennemførsel at systemet. Markedsundersøgelsen viser, at der næsten kun er fordele ved at skifte til et program der vil digitalisere og centralisere virksomhedens opgaver.

Den tekniske risiko er vurderet til at være projektets mest betydelige risikofaktor. Denne vurderes dog til at blive opvejet af projektets høje struktur samt generelt lave risiko.
Med alt dette taget i betragtning vurderes det at projektet er egnet til at blive fuldført.

Endelig beslutning ***Go***

Design

Faseplan (MC & MF)

Inception:

Iteration 1: Start 11/05 - slut 12/05

- Analyse og udarbejdelse af risikoanalyse, SWOT-analyse og interessentanalyse.
- Første udkast af funktionelle og ikke-funktionelle krav til programmet (FURPS+)
    - Supplementary Specification
- Use cases identificeres
- Indledende research af frameworks (Spring Boot, Bootstrap) og dependencies, der skal ligge til bund for programmet.

Elaboration

Iteration 1: Start 13/05 - slut 19/05

- Første udkast af use cases samt diagram
- SSD til fully dressed use cases laves
- Første udkast til domænemodel laves
- Videre research og begyndende implementering af frameworks: Spring og Bootstrap
    - Research af Spring Security dependency i forhold til login funktionalitet
- Identifikation af high-risks samt implementering af disse og core arkitektur
- ER-diagram designes og laves
- Feasibility Study

Construction

Iteration 1: Start 20/05 - slut 23/05

- Videre arbejde med core arkitektur. Gerne muligt at få opbygget et mock-up/skelet af designet (via bootstrap) i denne fase.
- Low-risk segmenter påbegyndes
- Raffinering og justering af uses cases og SSD'ere + SD i kombination med implementeringen af disse i programmet, så implementering matcher beskrivelse og diagram.
- Dannelse af SQL DDL-script samt SQL DML-script til at indsætte "test data".
- Klassediagram laves løbende i takt med kodning af programmet.

Iteration 2: Start 24/05 - slut 30/05

- Implementering af high-risk segmenter af programmet færdiggøres
- Low-risk segmenter færdiggøres
- Styling færdiggøres
- Generel struktur i programmet gennemgås, for at forsikre konsistens
- GET/POST/Redirect-pattern gøres konsistent igennem programmet

Domænemodel (MC & MF)

Vi har ud fra vores use cases og domæne opstillet en domænemodel, der beskriver relationen mellem vores konceptuelle klasser.

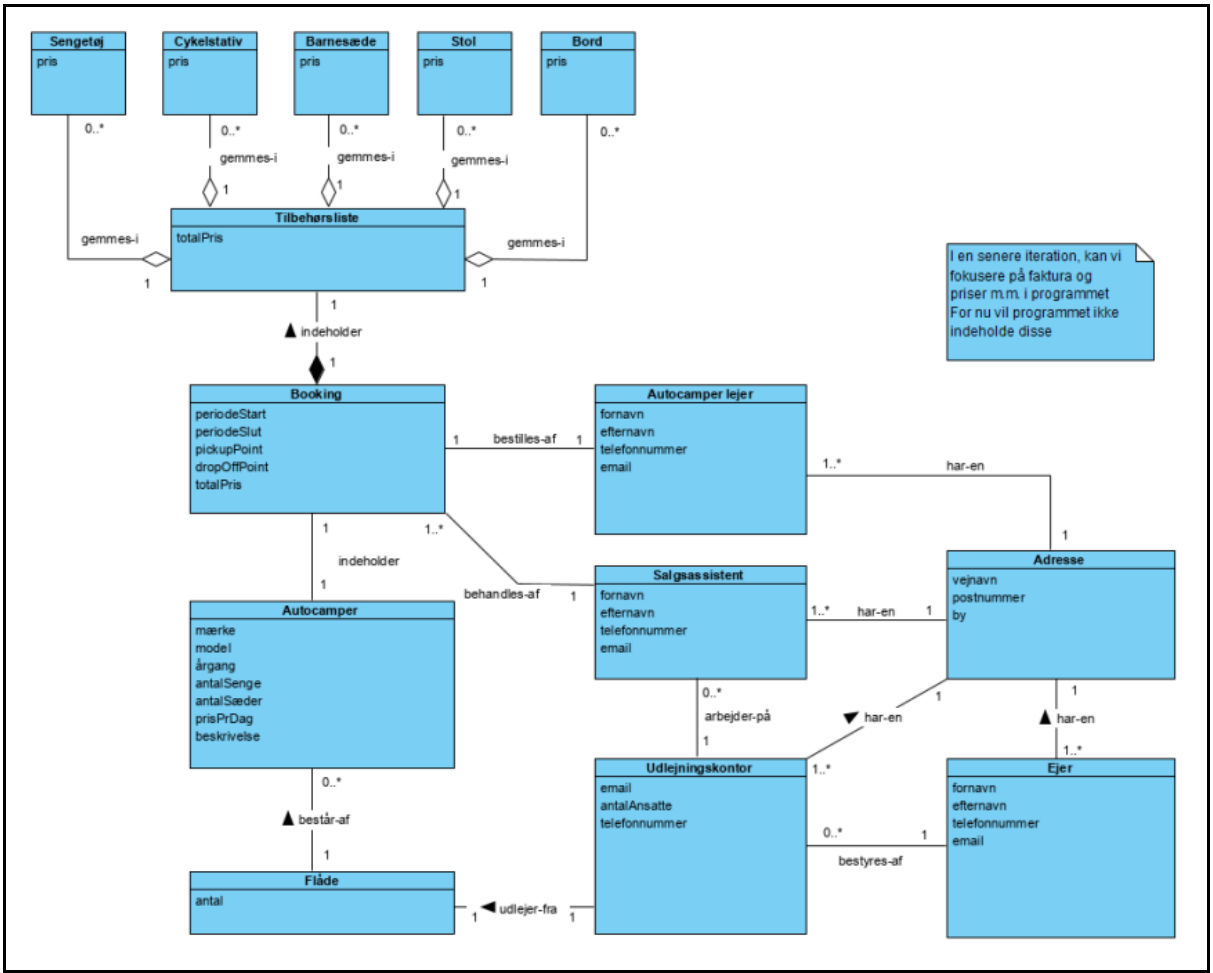


Diagram 2: Domænemodel for Nordic Motorhome Project

Det ses at en booking er tilknyttet en lejer, en autocamper og en tilbehørsliste der er opbygget af hver sin tilbehørsklasse. Bookingen behandles af en salgsassistent.

Derudover indeholder en booking sine egne attributter, der udelukkende omhandler bookingen selv såsom “periodStart” og “pickupPoint”, der henholdsvis beskriver startdatoen for en booking og hvor autocamperen ønskes at kunne hentes.

Det er måske blevet bemærket at klassediagrammet ikke indeholder en faktura klassen, som ellers kunne være oplagt. Vi har valgt ikke at at medtage en faktura klasse grundet vores afgrænsning, der ikke omhandler betalinger. For nu er det eneste der repræsenterer en betaling attributten “totalPris” i Booking, der blot er med for forståelsens skyld.

Vi har valgt at have en tilbehørsliste aggregeret af en masse tilbehørs klasser. Vi kan hermed have en samlet liste i en booking, så det er let at overskue hvilket tilbehør og hvor meget af hvert der er tilknyttet en booking.

Da mange klasser tilgår en adresse, har vi opdelt det således at adresser har sin egen klasse. Dette giver god mening, da vi nu har denne delte information centraliseret et sted.

Det ses også, at hvert udlejningskontor på dette tidspunkt har sin egen flåde, da deres multiplicity er 1-1. Her kunne man have valgt at have en centraliseret flåde, som alle de forskellige udlejningskontorer udlejer fra, men for at holde systemet så simpelt som muligt, og da ejeren ikke har nævnt andet, har vi valgt ikke at gøre dette. I stedet arbejder vi med en enkelt flåde pr. udlejningskontor, som dette udlejer fra.

Database (MC)

Kort efter domænemodellen var udarbejdet, gik vi i gang med at designe vores database. Vi mente det var essentielt at have fastlagt domænet, og derfor ikke designe databasen for tidligt, men også at det ikke skulle være for sent. I databasen ligger mange af vores tanker omkring hvordan forskellige dele af domænet interagerer i en mere kodenær sammenhæng, og derfor vil vi gerne knytte et par tanker til, hvorfor den er som den er.

ER-Diagram

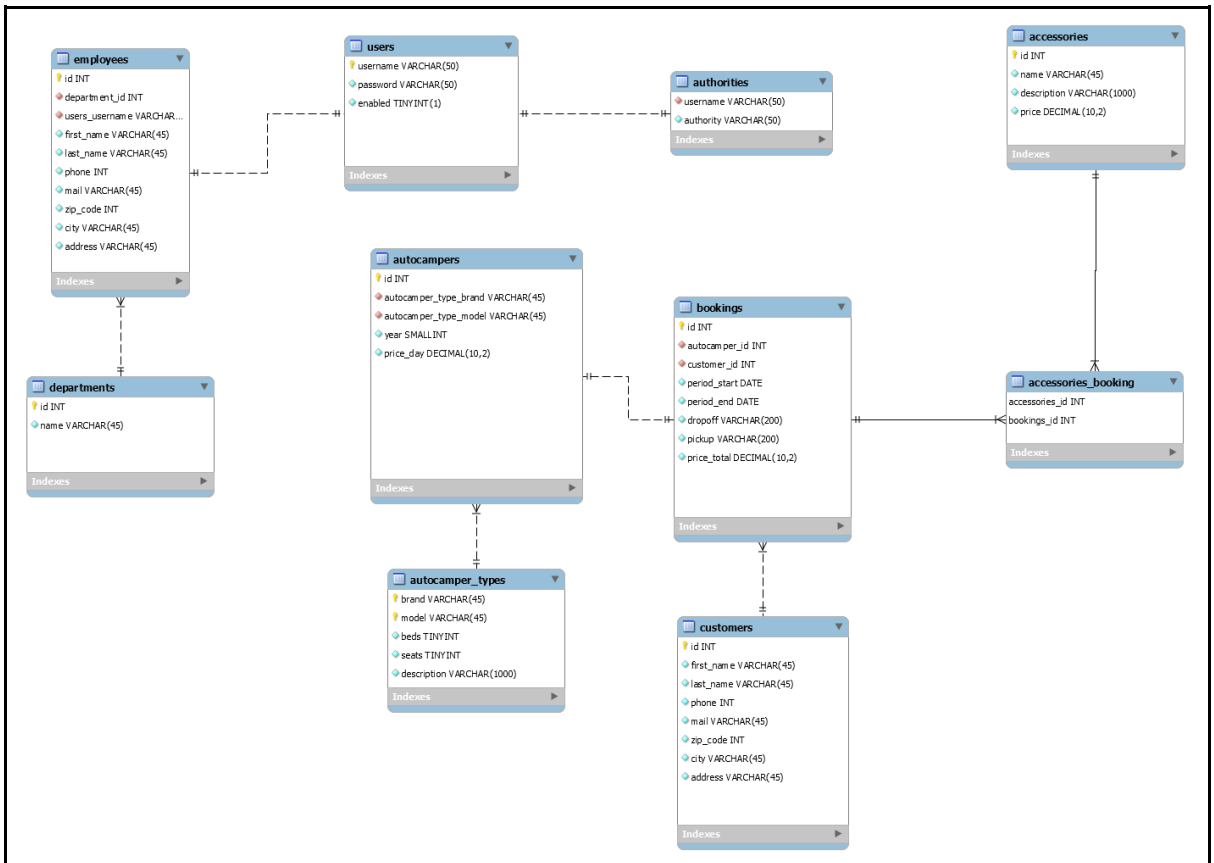


Diagram 3: ER-diagram overdatabase

Da dette diagram kan være vanskeligt at læse, henvises der til den vedhæftede fil “ER-diagram” for bedre læsning.

ER-diagrammet blev ligesom det meste af vores design lavet med en for stor afgrænsning. Derfor vil ting som tilbehør, ansatte i virksomheden og deres tilhørende afdeling være at finde, selvom det på nuværende tidspunkt ikke er implementeret i programmet. Det mener vi dog ikke gør det store, da det at tage fat i en stor del af domænet gjorde, at vi havde flere overvejelser med i designet af databasen.

Vi har så vidt muligt forsøgt at identificere primary keys som ikke nødvendigvis er et id, men nærmere en unik identifier, som kunne være at finde i hver tabel. Dette kan eksempelvis ses i autocamper_types, hvor kombinationen af brand og model skaber en composite primary key, eller i users hvor username også er unikt, da en bruger i systemet må have et unikt brugernavn.

Som det kan ses, er den centrale del helt klart bookings tabellen, som i bund og grund samler alle foreign keys fra omkringliggende tabeller, og på den måde har en reference til entries i disse. Det skyldes selvfølgelig, at en booking består af informationer fra mange af disse tabeller, som skal kunne findes igen.

Derudover har vi forsøgt at normalisere databasen til 3. normalform for at undgå data redundans og gøre tabellerne så nemme at arbejde med som muligt.

Det burde kunne ses at databasen er normaliseret til 1. normalform ved, at hver værdi i et givent row er skalar, og at ingen kolonner gentager sig selv.

Det burde kunne ses at databasen er normaliseret til 2. normalform ved, at hver kolonne som ikke indeholder en primary key, afhænger af hele tabellens primary key.

Til sidst burde det kunne ses at databasen er normaliseret til 3. normalform ved, at hver kolonne som ikke indeholder en primary key kun er afhængig af tabellens primary key.
Her kan vi dog knytte en lille kommentar, da det ikke er tilfældet for f.eks. kolonner i customers og employees tabellen, som har at gøre med en adresse. Nærmere kolonnerne “zip_code”, “city” og “address”. Dette er et bevidst valg.

Ved at skulle normalisere dette, ville det kræve en ny tabel med navnet addresses. Her ville vi skulle gemme adresse informationer for både kunder og ansatte. Det ville selvfølgelig være bedre normaliseret til 3. normalform, men på den anden side mente vi, at det ikke nødvendigvis ville have en positiv påvirkning. Eksempelvis vil der være markant forskel på mængden af data i customers og employees. Der vil rent naturligt ende med at være flere kunder end ansatte i systemet. På sigt ville dette kunne påvirke hastigheden for, hvor hurtigt en adresse til en ansat ville kunne hentes, da det nu ville være nødvendigt at kigge igennem en stor mængde adresser tilhørende kunder.

Det ville helt klart give mening i et større system, hvor man eventuelt indlæste alle adresser i Danmark til sin database, men i vores “lille” booking system, ville det ikke nødvendigvis have den ønskede gavnlige effekt og derfor har vi valgt at bibeholde adresse-relaterede kolonner i begge tabeller.

I forhold til datatyper, har vi forsøgt at tilpasse dem bedst muligt til de værdier de skal indeholde:

- Beskrivelser (descriptions) i accessories og autocamper_types tabellerne er af typen VARCHAR(1000), da vi ville være sikre på der var plads til at opbevare en længere beskrivelse uden problemer.
- Alle steder hvor datoer skal gemmes har vi brugt typen DATE
- Priser har datatypen DECIMAL(10,2), som beskriver at det skal være et kommatal med maks 2 decimaler og en længde på maks 10.
- year i autocampers tabellen og beds og seats i autocamper_types tabellen, har datatypen TINYINT der maks kan indeholde værdien 65535 kontra INT som maks kan indeholde værdien 4294967295. Dette er et bevidst valg, da vi ikke ser nogen situation, hvor så store værdier ville være nødvendige for disse kolonner.[16]
- enabled i users tabellen har datatypen TINYINT(1), som vi umiddelbart kunne finde frem til, var måden at beskrive en boolean værdi i MySQL. Default værdien for denne er 1 for at sørge for at en nyoprettet bruger vil være aktiv.

System sekvens diagrammer (MF)

For at få et overblik over, hvordan vores aktører skal tilgå systemet i en given use case, har vi lavet system sekvens diagrammer. Disse diagrammer behandler systemet som en “black box”, hvor fokus ligger på interaktionen mellem bruger og system.

[16] https://tableplus.com/blog/2018/08/mysql-the-difference-between-int-bigint-mediumint-smallint-tinyint.html

System sekvens diagram 1 - Opret autocamper booking

Se Visual Paradigm projekt: Sequence Diagram => SSD: Opret booking

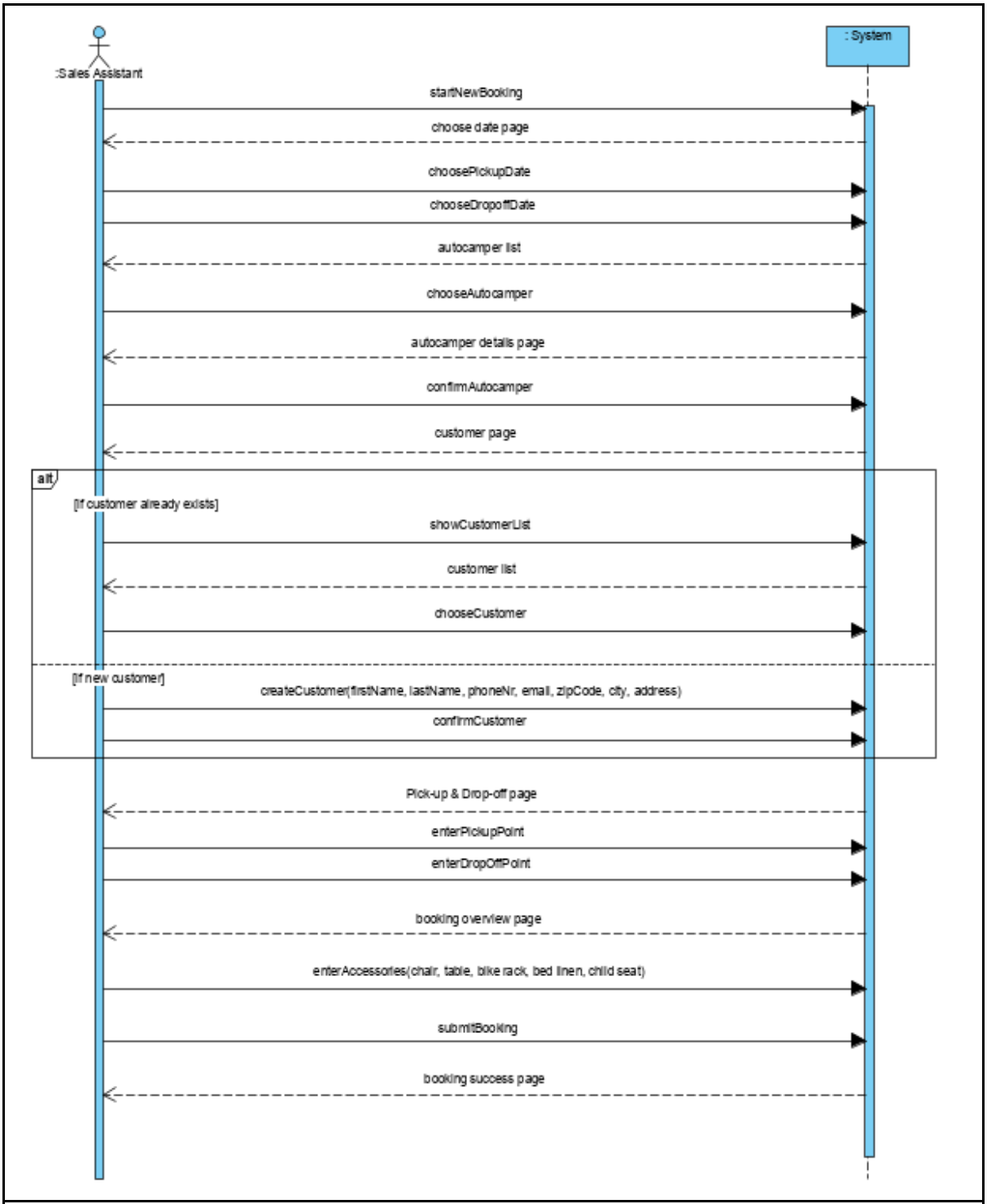


Diagram 4: System sekvens diagram - Opret booking

Dette diagram er lavet over vores use case “Opret autocamper booking”. Diagrammet starter med at salgsassistenten initialiserer bookingstart ved “createBooking”. Herefter præsenterer

systemet en side, hvor bookingdatoer skal indtastes. Datoer bekræftes af salgsassistenten, og en en liste af ledige autocampere vises. Nu kan salgsassistenten vælge en autocamper, hvorefter systemet præsenterer en side med yderligere information af autocamperen.

Dette mønster “indtast information ->“bekræft information -> ny side” går igennem hele flowet for opret autocamper booking. Det eneste sted dette mønster brydes er ved tilhæftning af en kunde til bookingen, hvor vi har lavet et alternative flow. Det har vi gjort, da det er muligt enten at oprette en ny kunde eller vedhæfte en allerede eksisterende kunde til en booking.

Hvis salgsassistenten vil oprette en ny kunde, skal de blot udfylde en form og bekræfte, men hvis de vil tilføje en allerede eksisterende kunde, bliver salgsassistenten vist en liste af kunder, der kan tilføjes til bookingen. Salgsassistenten kan herefter vælge den korrekte kunde, der skal tilføjes til bookingen. Begge flows ender med at salgsassistenten bliver præsenteret en pick-up og drop-off side.

Det bemærkes her, at det er muligt at tilføje tilbehør til bookingen. Dette er ikke muligt i vores system, da vi ikke havde tid til at implementere denne funktion. Vi havde dog forestillet os, at efter man havde valgt de mest essentielle dele af bookingen, såsom autocamper og bookingperiode, havde mulighed for at kunne tilføje ekstra tilbehør til ens booking.

System sekvens diagram 2 - Søg autocamper

Se Visual Paradigm projekt: Sequence Diagram => SSD: Søg autocamper

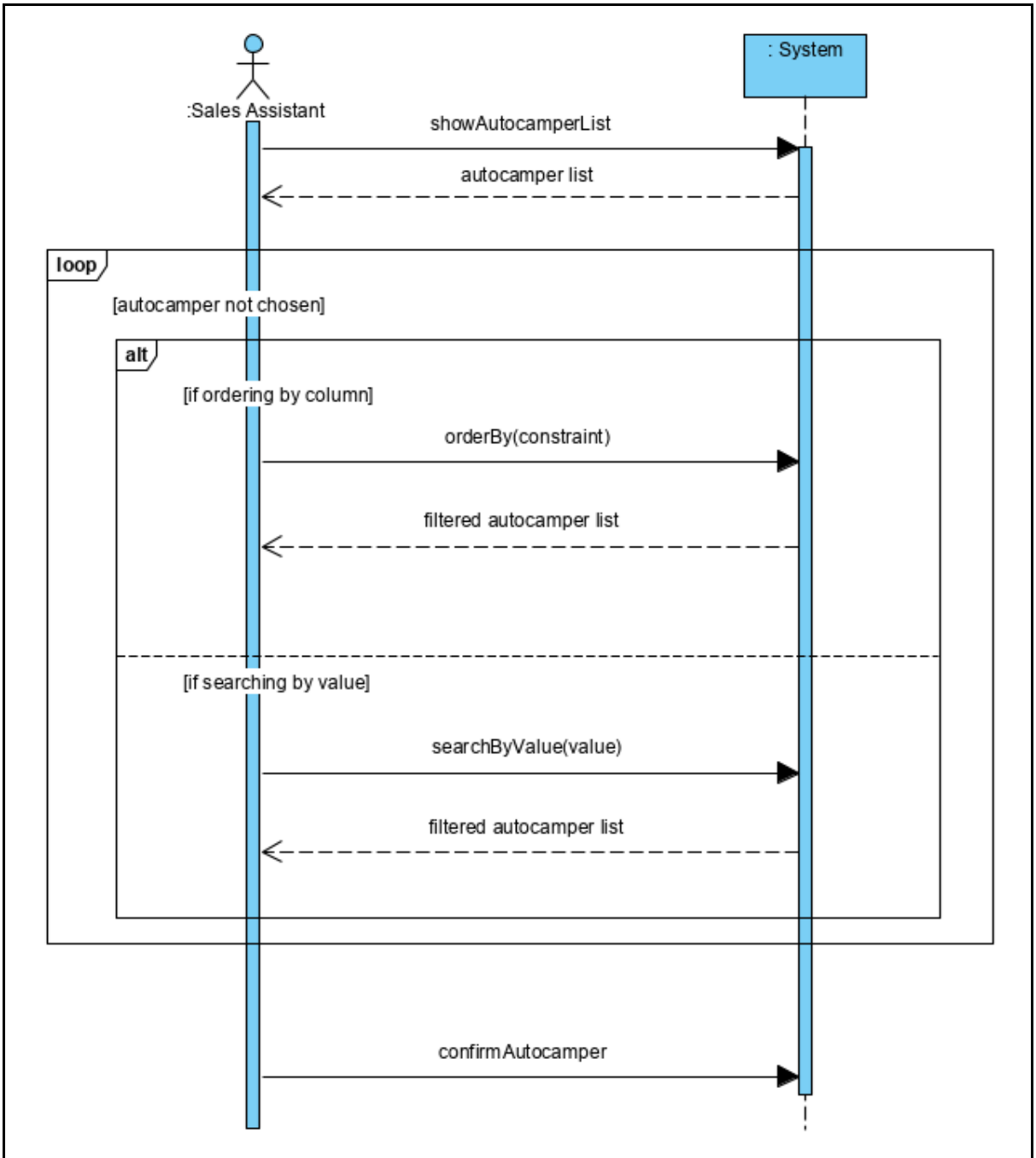


Diagram 5: System sekvens diagram - Søg autocamper

I denne use case får salgsassistenten vist en liste af autocampere, hvorefter salgsassistenten træder ind i et loop med den condition, at en autocamper ikke er valgt. I loopet har vi et alternative flow, da listen kan filtreres på to måder. Enten kan listen sorteres ud fra en kolonneværdi (såsom pris) eller også kan der søges efter en specifik værdi (såsom Fiat).

Da dette alternative flow er inde i et loop, kan der filtreres og sorteres så mange gange, som salgsassistenten finder passende inden den korrekte autocamper er valgt. Herefter træder salgsassistenten ud af loopet, hvorefter de bekræfter valget af autocamper.

Vi valgte at have disse to filtreringsmetoder, da vi mener at begge disse filtreringsmetoder kan være passende til hver deres situation - nemlig specifikke og uspecifikke forespørgsler. Her ville en specifik forespørgsel f.eks. være, at autocamperen skal have fem senge, og en uspecifik forespørgsel kunne være at finde den billigste autocamper. Vi mener dog, at en kombination af de to filtreringsmetoder ofte vil være salgsassistentens bedste bud. For at eksemplificere dette, kan vi bruge vores eksempel fra før, hvor kunden skulle have en autocamper med fem senge. På dette allerede aktive filter kan salgsassistenten f.eks. nu tilføje et uspecifikt kolonnefilter og finde den billigste autocamper med fem senge. Herfra mener vi, at listen formodentlig vil være filtreret tilstrækkeligt, men vi kan ikke udelukke, at yderligere filtre kunne være brugbare. Dette kunne være interessant at undersøge i en senere iteration.

Igen bemærkes det dog, at disse filtreringsmetoder ikke er implementeret i systemet endnu grundet tidsmangel. For at følge det aktuelle system sekvens diagram, skal man blot undgå at træde ind i loopet.

Klassediagram (MF)

Se Visual Paradigm projekt: Class Diagram => Klassediagram

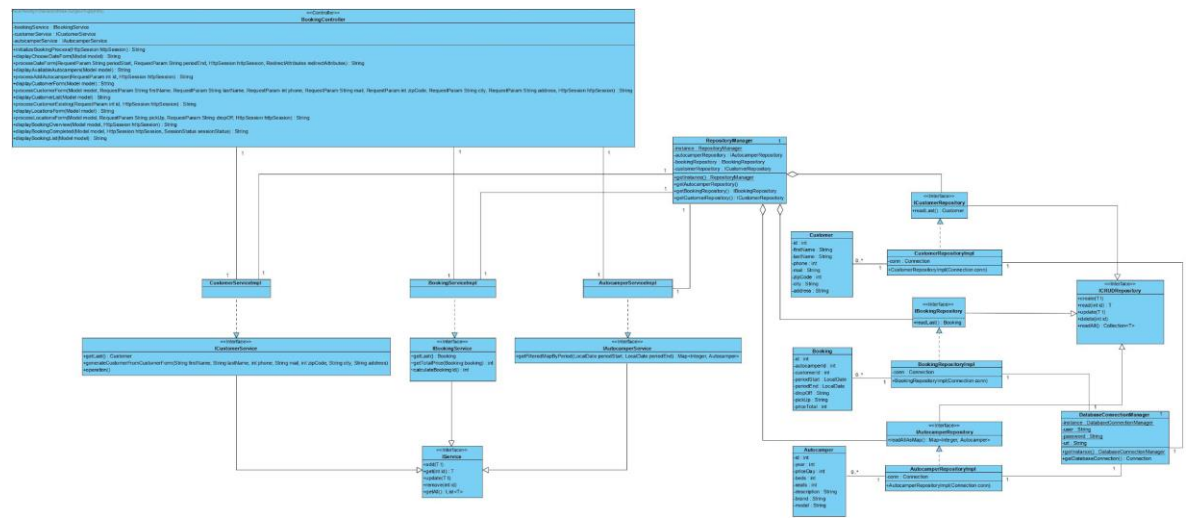


Diagram 6: Klassediagram

Da dette diagram er meget stort, anbefales det at kigge i selve Visual Paradigm projektet under læsningen af diagrammet.

Vi valgte at lave et klassediagram, både fordi det var et krav, men også for at danne et overblik over systemets klasser og udtænke de forskellige attributter og metoder som de indeholder. Klassediagrammet var også med til at hjælpe os med at udtænke den

sammenhæng som klasserne interagerer i og diagrammet var på den måde et værktøj som hjalp os til bedre forståelse af vores kode.

Da klassediagrammet indeholder mange patterns samt koncepter såsom Singletons, har vi valgt for overskuelighedens skyld at uddybe disse senere i rapporten og bare præsentere et overblik her og nu.

Vi har valgt at medtage en enkelt controller klasse, “BookingController", for at eksemplificere, hvordan vi har lavet vores controller klasser. Denne controller vil altså repræsentere alle controllers i programmet.

Klassen kan dog være en anelse uoverskuelig hvilket skyldes dens mange metoder og at det ikke er muligt at beskrive både flow og hvordan metoder i klassen kalder hinanden. Af denne grund valgte vi under udviklingen af booking controlleren at lave et flow-chart som kunne visualisere klassen og dermed give et overblik.

BookingController flow-chart (MC)

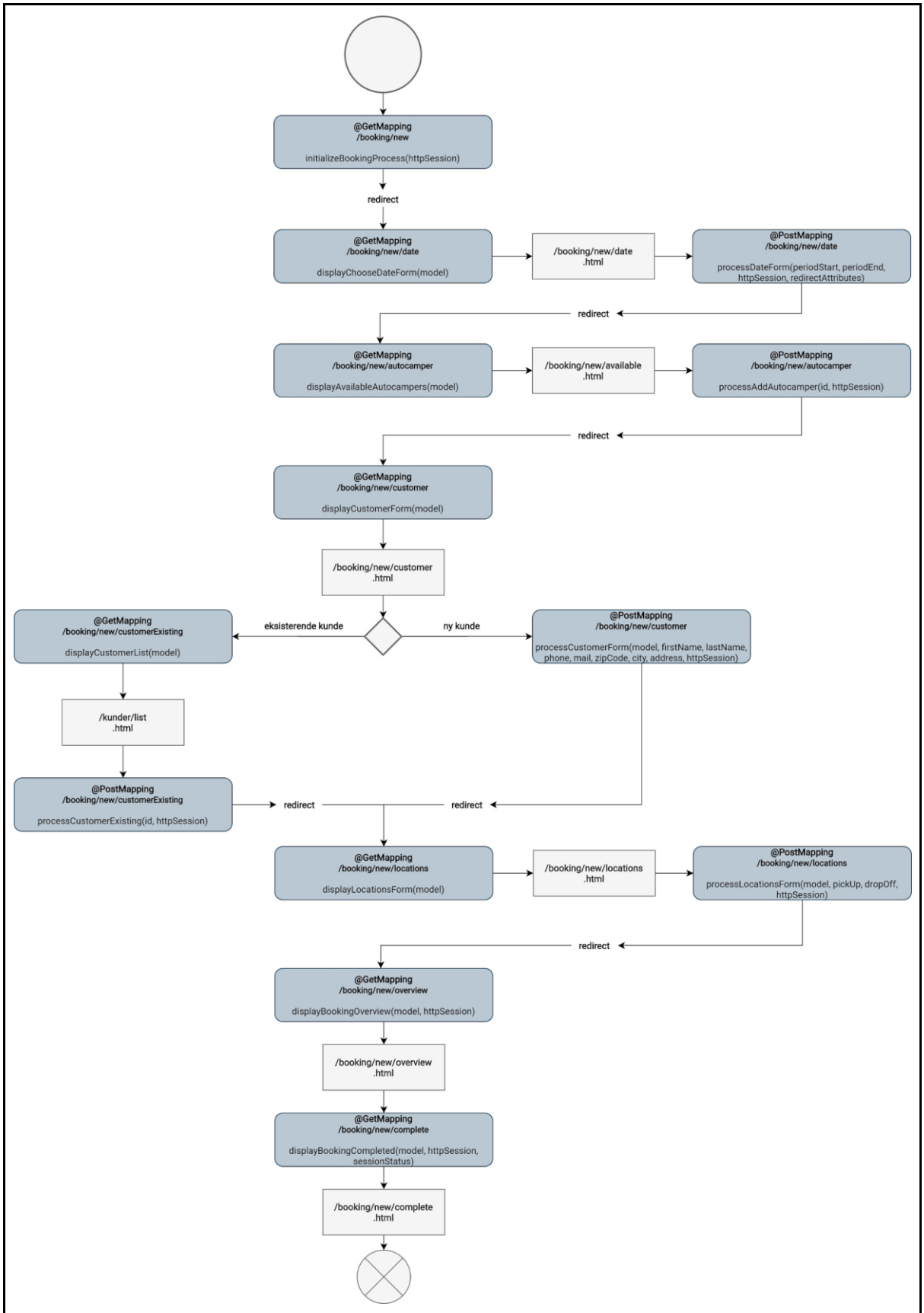


Diagram 7: Flow-chart for BookingController klassen

Da dette diagram kan være vanskeligt at læse i fuld størrelse i rapporten, henvises der til det vedhæftede flow-chart for bedre læsning.

Vi er opmærksomme på, at notationen ikke nødvendigvis er korrekt, men da diagrammet gav os et rigtig godt overblik og blev brugt meget under udvikling af klassen, har vi valgt at medtage det for at vise vores arbejdsprocess.

Diagrammet viser flowet for BookingController klassen tydeligt, blandt andet rækkefølgen de forskellige metoder kaldes i, hvilke views de returnerer o.l. Bemærk også at det viser vores forsøg på et Post/Redirect/Get pattern, som vil blive beskrevet mere udførligt senere i rapporten.

Jvf. vores system sekvens diagrammer ses det igen, hvordan bookingprocessen er relativt repetitiv, hvor forms bliver submitted, herefter behandles submitted data ved en POST mappet metode, og til slut bliver brugeren redirected til en ny GET mappet side. Igen ses det også, hvordan det er muligt at vælge enten at oprette en ny bruger eller tilføje en allerede eksisterende bruger til bookingen i den decision node der forekommer i diagrammet.

Sekvensdiagrammer

For at danne overblik og visualisere interaktionen mellem klasser i et program, kan det være brugbart at lave sekvensdiagrammer. I vores tilfælde har vi valgt at lave et sekvensdiagram for henholdsvis “opret kunde” use casen og “søg autocamper”.

Dette valgte vi at gøre, da vores use case “opret autocamper booking” i store træk gør det samme igen og igen, og at et sekvensdiagram over hele processen derfor vil gå hen og få en redundant karakter. Vi mener derfor at disse to sekvensdiagrammer vil give et fint overblik over, hvordan hele “opret autocamper booking” use case vil skulle implementeres. Vi ser disse to dele af “opret autocamper booking” use case som de mest interessante at undersøge ved et sekvensdiagram.

Som en lille ekstra kommentar vil man når man udarbejder sekvensdiagrammer gøre det med henblik på at skabe en bedre forståelse af et specifikt kodesegment. Vi havde dog lidt problemer med dette. Sekvensdiagrammer i tidligere projekter har kunne afbilde kode som 100% er vores egen, men nu hvor der bliver blandet frameworks ind i det (Spring) har et stort lag af funktionalitet været abstraheret væk fra os. Vi har derfor forsøgt på bedste vis at forklare ”vores” del af processen for et givent kodesegment og forsøgt at ”gemme Spring væk”.

Sekvensdiagram - Opret kunde (MC & MF)

Se Visual Paradigm projekt: Sequence Diagram => SD: Opret kunde

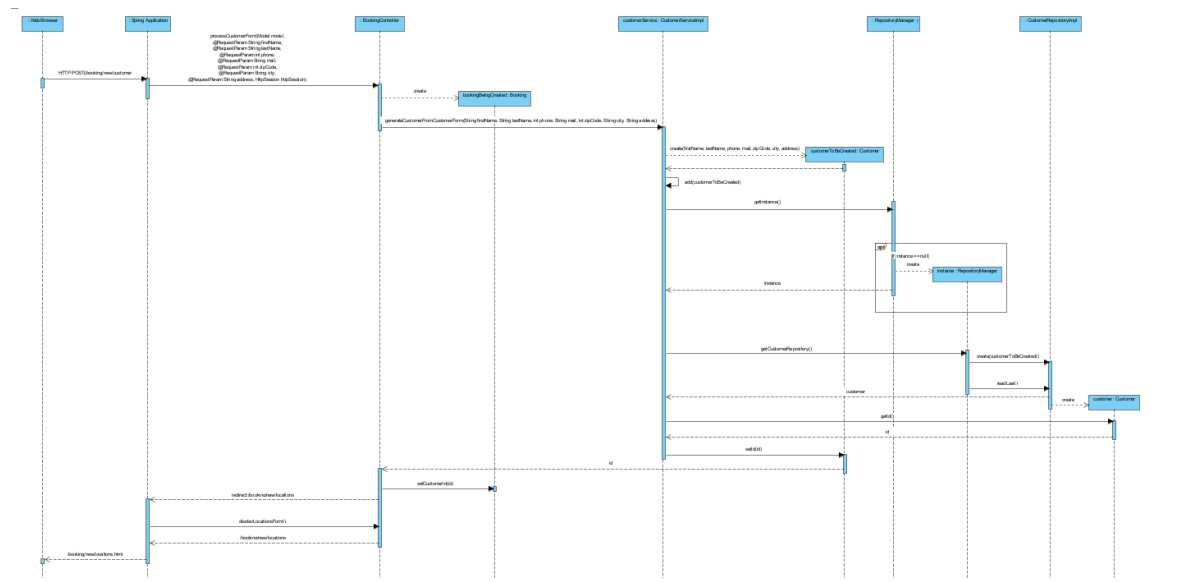


Diagram 8: Sekvensdiagram over “opret kunde” use case

Da dette diagram er vanskeligt at læse i rapporten, henvises der til det vedhæftede Visual Paradigm projekt for lettere læsning.

Sekvensdiagrammet er en kodenær visualisering af vores ”Opret Kunde” use case og indeholder en klasse afbildet af typen “Spring Application”. Da vi ikke har stor erfaring med Spring og ikke kender alle interne gear og hjul blev vi enige om at en abstraktion af disse ukendte mekanismer ville være en god idé. På den måde har vi kunne fokusere på at beskrive de dele af koden som vi selv har skruet sammen.

En anden “hjemmelavet” klasse er typen “Web Browser” som repræsenterer en web browser. Den er udelukkende inkluderet for at skabe en form for forståelse for hvad der starter den beskrevne funktionalitet. Den er med til at vise hvilken type af request (post/get) som Spring håndterer samt hvilket view der til sidst returneres til browseren.

Sekvensdiagram - Søg autocamper (MC & MF)

Se Visual Paradigm projekt: Sequence Diagram => SD: Søg autocamper

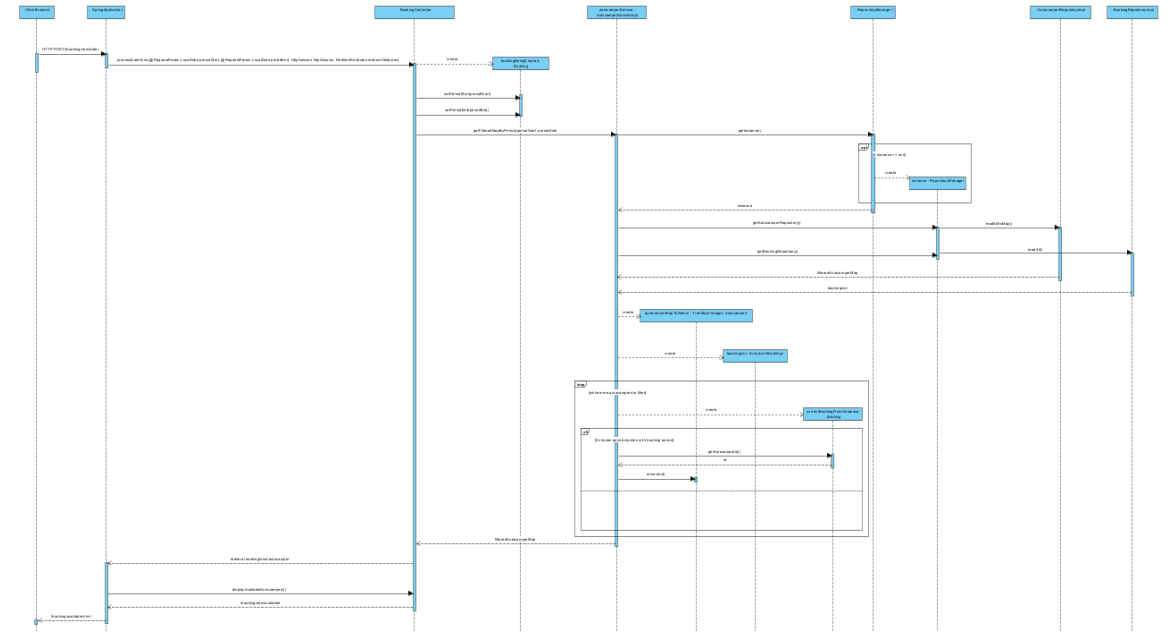


Diagram 9: Sekvensdiagram over “Søg autocamper” use case

Da dette diagram er vanskeligt at læse i rapporten, henvises der til det vedhæftede Visual Paradigm projekt for lettere læsning.

Sekvensdiagrammet er en visuel og mere kodenær repræsentation af vores ”Søg Autocamper” use case.

I dette diagram anvender vi igen de hjemmelavede klasser “Spring Application” og “Web Browser” til at holde fokus på vores egen kode.

Implementering

I dette afsnit vil vi komme ind på selve implementeringen af designet samt udvalgte dele af vores kode og de patterns vi har gjort brug af.

Programstruktur (MF)

Overordnet set har vores program et hierarki, der ser således ud:

Controller ⇔ servicelag ⇔ managerklasse ⇔ repositorylag ⇔ database & model

Hvoraf "⇔ " markerer kommunikation.

Controller

Dette lag tager sig af af alle requests fra viewet, og navigerer dem herefter videre til servicelaget. Med andre ord kan de ses som en form for "trafiklys".

Service

Det er heri at alle businessrelaterede operationer foregår, som f.eks. beregninger og listeoperationer. Dette lag bekymrer sig ikke om, hvor objekter kommer fra, eller hvordan de bliver opbevaret, men arbejder kun med objekterne selv.

RepositoryManager

Servicelag og repositorylag er bundet sammen med en RepositoryManager klasse, der indeholder instanser af repository klasserne. Tanken bag denne manager klasse er, at den agerer som en "bro" mellem servicelaget og de specifikke repository implementationer.

Repository

Dette lag tager sig udelukkende af databaserelaterede opgaver. Altså hvordan objekter bliver hentet, indsat eller på anden måde interagerer med databasen. Disse klasser anvender derfor også vores DatabaseConnectionManager til kommunikation med databasen. Alle repositories skal bruge en database forbindelse til at instantieres og dette sker i RepositoryManager klassen, hvor vi instantiere vores repositories med hjælp fra DatabaseConnectionManager klassen, som returnere en forbindelse.

Model

Det er her, hvor vores POJOs af bookinger, autocampere og kunder ligger. Disse klasser bliver primært behandlet af repository laget, der udfører databaseoperationer på disse.

Patterns

Vi vil nu gennemgå en række af de patterns, som vi har brugt under dette projekt og dele af vores systemstruktur. Vi vil forsøge give jer et indblik i, hvorfor vi har valgt at anvende netop disse patterns og principper.

GRASP

GRASP er et akronym, der står for **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**attern. Idéen med GRASP er, at man opsætter en række principper eller guidelines for, hvordan ansvar til sammenspillende softwareklasser skal fordeles. Princippet er, at hvis man følger disse designprincipper, så får man god og forståelig kode.

Da GRASP er opbygget af 9 principper, hvoraf vi vil gå i dybden med 4 af disse.

Controller (MF)

En controllers funktion er at delegere opgaver imellem brugerinterface og selve applikationen. Controlleren har altså ansvaret for at videresende forespørgsler fra UI-laget til den korrekte klasse i applikationslaget, hvor forespørgslen vil blive behandlet. Herfra kan en klasse i applikationslaget videresende en forespørgsel til controlleren om at vise det korrekte view. Controlleren koordinerer altså kommunikationen imellem views og applikationslaget.

Det er her vigtigt, at en controller ikke udfører arbejde selv, men derimod delegerer opgaver videre til andre klasser.

I vores program anvendes controllere til netop dette formål. Af dette er fremsprunget fire controllere, der hver har ansvar for hver deres område; (AutocamperController, BookingController, CustomerController og LoginController).

For at give et eksempel på, hvordan en af vores controllere virker, har vi her en simpel controller metode i BookingController, der returnerer en liste af bookinger til brugeren.

```
/*
Returnerer en liste over bookinger i systemet. Ikke en del af ny booking flowet.
 */
@GetMapping("/booking/list")
public String displayBookingList(Model model) {
    model.addAttribute( s: "title",  o: "Bookinger");
    model.addAttribute( s: "bookingList", bookingService.getAll());

    return "/booking/list";
}
```

Det ses her, hvordan bookingController modtager et mapping fra viewet, hvorefter controller metoden kalder en serviceklasse, der nu kan hente alle bookinger som en liste. Dette tilføjes herefter til modellen, som sendes videre til et view.

High Cohesion (MC)

High Cohesion handler i bund og grund om at skabe forståelse. Måden man indenfor objekt-orienteret programmering opnår en højere forståelse, sker ved at designe klasser med et så koncentreret formål, eller ansvar, som muligt. Man kan med andre ord sige, at jo mere fokuseret en klasse er, jo mere forståelse (eller cohesion) opnår den.

Der er flere fordele ved at skabe high cohesion. Primært opnår man klasser, som er nemmere at vedligeholde og rette, og som ikke lige så tit har brug for at blive rettet. Derudover vil klasser med high cohesion også typisk være nemmere at genbruge, da deres fokuspunkt og ansvar er klart defineret, og på den måde nemmere kan bruges sammen med andre klasser. Det gør altså at klassen kan bruges bedre på tværs af klasser.

Vi har i vores applikation forsøgt at skabe high cohesion ved at prøve at definere en klasses ansvarsområde så tydeligt som muligt. Dette kan også ses igennem vores program kommentarer, som henvender sig overordnet til vores klasser, som i eksemplet nedenfor:

```
/*
    Service klasse med ansvar for at udfører operationer relevante for en autocamper.
    Dette gøres via klassens tilsvarende repository som står for kontakt med databasen.
 */
public class AutocamperServiceImpl implements IAutocamperService {
```

Her er der altså både en AutocamperService klasse, som står for operationer med relevans for autocampere. F.eks. det at lave en liste af alle autocampere fra systemet via getAll() metoden. Går man videre ned i programmets lag sker dette ved hjælp af en tilsvarende AutocamperRepository klasse, som har til ansvar at udføre den givne operation imod vores database.

På den måde er der et klart hierarki og en klar fordeling af ansvar for hver klasse, hvilket skaber en høj forståelse, da det er tydeligt hvad en klasse skal kunne. Dette gør det nemmere at genbruge klassen og dens metoder, frem for at have alt funktionalitet samlet i nogle få klasser.

Low Coupling (MC)

Low coupling læner sig op ad high cohesion i den forstand at low coupling som regel også vil skabe high cohesion. Der er dog stadig forskel.

Kerneprincippet i low coupling handler om at klasser, eller moduler, skal være så uafhængige af hinanden som muligt. Det skal gerne være sådan at når man foretager en ændring i en klasse, så skal det helst ikke kræve for mange ændringer i andre klasser. Det gør det nemmere at foretage ændringer, vedligeholde kode og tilføje ny funktionalitet og let kunne koble det sammen med eksisterende klasser og metoder.

Der er flere måder man kan forsøge at skabe low coupling, men det er også vigtigt at pointere at for lav kobling heller ikke er at foretrække. For høj kobling vil betyde, at det bliver svært at foretage ændringer, og vil oftest medføre low cohesion, men det samme vil ske ved for lav kobling. Ved for lav kobling vil klasser, ansvar og funktionalitet være spredt så meget ud og så opdelt, at det kan være svært at arbejde med. Derfor er det vigtigt at pointere at det handler om at finde en mellemvej mellem for høj og for lav kobling.

Vi har forsøgt at skabe low coupling på flere måder. Både via brugen af interfaces, opdelingen af vores klasser og måden hvorpå vores metoder virker.

Et eksempel på en metode i vores BookingServiceImpl klasse som demonstrerer vores filosofi kan ses nedenfor:

```
@Override
public int getTotalPrice(Booking booking) {
    int priceDay = 0;
    int numberOfDays = (int) ChronoUnit.DAYS.between(booking.getPeriodStart(), booking.getPeriodEnd());

    try {
        priceDay = RepositoryManager.getInstance().getAutocamperRepository().read(booking.getAutocamperId()).getPriceDay();
    } catch(SQLException e) {
        e.printStackTrace();
    }

    return priceDay * numberOfDays;
}
```

Her tager getTotalPrice() metoden imod et booking objekt for at kunne beregne den totale pris for et booking objekt. Metoden er i bund og grund ligeglad med implementationen af booking klassen, så længe at booking objektet har en periode og en pris pr. dag, som kan hentes kan metoden udregne en totalpris og returnere denne.

Man kan i metoden argumentere for, at der er høj kobling mellem den og repository laget da måden pris pr. dag hentes er på følgende måde:

1. RepositoryManager klassens getInstance() metode kaldes,
2. på instansen kaldes getAutocamperRepository, som returnerer en instans af dette repository.
3. read() metoden kaldes på autocamperRepository instansen og getAutocamperId() kaldes på det booking objekt som getTotalPrice modtager. Altså finder AutocamperRepository den autocamper, i databasen, som tilhører den givne booking.
4. Til sidst kaldes metoden getPriceDay() på booking objektets tilhørende autocamper.

På den anden side er prisen nødt til at komme et sted fra. Alternativt kunne man gemme en instans af autocamperen i booking objektet for på den måde at hente pris pr. dag. Det vil dog også skabe en form for høj kobling, da booking objektet nu er afhængigt af autocamper objektet. Deraf vores tidligere argument for, at der altid vil være en form for kobling mellem klasser og metoder.

Vi mener umiddelbart at have fundet en løsning, som virker og skaber low coupling til et godt punkt. RepositoryManager håndterer instanser af de forskellige repositories, og derfor er der ikke en direkte dependency til et repository i vores metode. Det er også muligt at ændre implementationen af read() metoden i AutocamperRepository klassen så længe den returnerer et autocamper objekt ud fra et givent id. På den måde er der et sæt "rammer", eller ting som en metode skal kunne tage imod og evt. returnere, som skal være opfyldt, men hvordan disse "rammer" opfyldes kan ændres uden problemer.

For at illustrere et eksempel på low coupling har vi her et eksempel på en af vores model klasser; Customer.

```
public class Customer {
    private int id;
    private String firstName;
    private String lastName;
    private int phone;
    private String mail;
    private int zipCode;
    private String city;
    private String address;

    // Default constructor så vi kan oprette objekter til nye kunder
    public Customer() {

    }

    public Customer(int id, String firstName, String lastName, int phone, String mail, int zipCode, String city, String address) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.phone = phone;
        this.mail = mail;
        this.zipCode = zipCode;
        this.city = city;
        this.address = address;
    }

    // Constructor uden id da vi lader DB håndterer tildeling af id til hvert entry
    public Customer(String firstName, String lastName, int phone, String mail, int zipCode, String city, String address) {
        this.firstName = firstName;
        this.lastName = lastName;
```

Customer klassen indeholder udelukkende fields af simple typer, og har ingen dependencies til andre klasser. På den måde har den altså low coupling, da ændringer andre steder i programmet ikke vil have en påvirkning på denne klasse. Men som det også er pointeret før, så er dette ikke et realistisk niveau af kobling for de fleste klasser. Hele idéen med objekt-orienteret programmering er, at objekter interagerer med hinanden, og det i sig selv skaber en form for kobling imellem dem.

Polymorphism (MF)

Polymorfi betyder "mange former", og er konceptet omkring, hvordan en java klasse kan tage forskellige udformninger. Det er hermed muligt at anvende den samme klasses metoder med forskellige udformninger, hvilket øger genbrugeligheden af kode samt reducerer mængden af redundant kode i et program. Det har også fordelen at ens kode bliver lettere overskuelig, da man kan opsætte et tydeligt hierarki imellem sine klasser.

Vi har anvendt polymorfi i vores projekt ved at anvende en række interfaces. Vi har f.eks. et generisk interface kaldet ICRUDRepository, der indeholder en række metoder med

generiske typer. Vi så dette som en god mulighed for at anvende polymorfi, da bookinger, autocampere og kunder alle skal indeholde disse metoder.

```
public interface ICRUDRepository<T> {
      void create(T t);
      T read(int id);
      void update(T t);
      void delete(int id);
      List<T> readAll();
}
```

Disse metoder nedarves herefter til nogle repositories, der giver hver metode en type. Vi har i disse repositories også nogle typespecifikke metoder, hvoraf vi stadigvæk forsikrer os, at vi stadigvæk har samme basale funktioner. Vi kan hermed anvende de samme metoder på objekter af forskellige typer, hvor de vil have specialiseret behaviour. Disse interfaces nedarves her til javaklasser af den specifikke type, hvor de bliver implementeret.

Vi så ikke så mange andre muligheder for at anvende polymorfi, da vores domæne er relativt opdelt, og derfor ikke lægger så meget op til delt funktionalitet. I en senere iteration kunne det være interessant at undersøge, hvordan polymorfi kunne anvendes yderligere. Dette kan f.eks. blive oplagt, hvis man valgte at implementere de resterende ansatte i programmet (bogholder, rengøringsassistent, mekaniker). Disse kunne alle være en del af en gruppe “employees”, hvoraf de kunne dele en række metoder, såsom getSalaray() eller getId(). Dette ville formentlig kræve en ændring af databasen, men det kunne helt klart være værd at undersøge inden videre implementation.

Model-View-Controller (MC)

Model-View-Controller, eller MVC, pattern er et design pattern som især bruges indenfor webapplikationer. Formålet er at opnå seperation of concerns.

Vi har brugt Spring Web til at opnå dette pattern, da denne dependency fra Springs side er lavet med MVC pattern i mente. Vi vil forsøge at forklare dette pattern med udgangspunkt i en metode fra vores BookingController, som kan ses efter forklaringen af hvert element i MVC.

Model

Modellen er i Spring sammenhæng et objekt, som kan indeholde attributter. Disse attributter er i vores sammenhæng objekter, som vi ønsker at vores view kan bruge til f.eks. at vise de fields som en booking indeholder til vores bruger, der tilgår den specifikke mapping. Modellen passeres altså til viewet efter at man via controlleren har tilføjet de ønskede objekter.

Controller

Det centrale element i MVC pattern er controlleren, som ligesom navnet angiver kontrollerer de to andre elementer. Det er controlleren som bestemmer hvad der skal gemmes i modellen, og hvilket view som skal vises til brugeren og i hvilken sammenhæng. Det er med andre ord en form for trafiklys.

Det vil være sådan, at controlleren modtager en request, sørger for at klargøre modellen med de ønskede attributter/objekter, og herefter sørger for at det ønskede view vises til brugeren.

View

Et view er i vores sammenhæng en fil med filtypen html, som viser hvad vi ønsker overfor brugeren. Et view er altså en specifik side i vores webapplikation, som brugeren kan se og interagere med. I vores applikation benytter vi os af Thymeleaf, som tidligere beskrevet, og det er via Thymeleaf, at vi har mulighed for at kommunikere med modellen igennem vores view.

Dette kan ses i nedenstående uddrag fra /booking/new/overview.html filen, hvor bookingens informationer indlæses i en liste via en Thymeleaf variable expression *(læs: ${booking.autocamperId})*

```
<li class="">
    <p class="font-weight-bold">Autocamper ID: <span class="font-weight-normal" th:text="${booking.autocamperId}"></span></p>
</li>
<li class="">
    <p class="font-weight-bold">Kunde ID: <span class="font-weight-normal" th:text="${booking.customerId}"></span> </p>
```

For at give nærmere forklaring og konkretisere vores brug af MVC pattern, vil vi tage udgangspunkt i nedenstående metode fra vores BookingController klasse.

```
@GetMapping("/booking/new/overview")
public String displayBookingOverview(Model model, HttpSession httpSession) {
    model.addAttribute( s: "title",  o: "Booking Overblik");

    Booking bookingBeingCreated = (Booking) httpSession.getAttribute( s: "bookingBeingCreated");
    bookingBeingCreated.setPriceTotal(bookingService.getTotalPrice(bookingBeingCreated));
    model.addAttribute(bookingBeingCreated);

    return "/booking/new/overview";
}
```

I metoden er der tilføjet en GetMapping annotation, som peger på adressen "/booking/new/overview", som betyder at denne metode skal håndtere den GET request, der sendes til denne adresse.

Derudover er der tilføjet så lidt "reel kode" som muligt i controlleren, da den kun skal dirigere trafik. Den kode vi har tilføjet, har udelukkende til ansvar at udfylde modellen med de attributter, vi ønsker at bruge i vores view.

1. Der passeres en titel til viewet som i vores html fil bruges i <title> tagget.
2. Vi henter det booking objekt, som er gemt som en SessionAttribute og som indeholder alle informationer som salgsassistenten indtil videre har udfyldt omkring bookingen. Dette objekt gemmer vi i en variabel af typen Booking, for at kunne tilføje yderligere information til den.

3. setPriceTotal() metoden kaldes på vores booking objekt, og via servicelaget, finder vi den totale pris for bookingen, som udregnes ud fra perioden og autocamperens pris pr. dag.
4. Det nyopdaterede booking objekt tilføjes til modellen, hvorefter vi i vores view præsentere bookingens attributter overfor salgsassistenten, så han kan tjekke at alle informationer er korrekte.

Post/Redirect/Get (MF)

Post/Redirect/Get (PRG) er et design pattern, der anvendes til at forsikre at form submissions ellller andre POST-requests kun forekommer en gang. Dette er et klassisk problem, hvis en side sat sådan op, så hvis siden efter et POST-request bliver refreshed, så vil ens POST-request blive resubmitted, hvilket resulterer i at endnu et POST-request vil blive sendt afsted. Dette kan have slemme konsekvenser, da et køb f.eks. vil kunne blive gentaget flere gange.

Dette kommer man udenom ved brug af PRG-pattern, da man i sin POST-request returnerer et redirect i stedet for en ny side. Dette redirect bliver herefter samlet op af et GET-request til den nye URL, som kan returnere den korrekte side. Hvis kunden nu refresher siden, så vil de blot blive returneret den nuværende side, og ingen skade er sket.

Dette pattern har vi implementeret i vores program, hvilket kan ses i POST metoderne i BookingController klassen.

```
@PostMapping("/booking/new/autocamper")
public String processAddAutocamper(@RequestParam int id, HttpSession httpSession) {
    Booking bookingBeingCreated = (Booking) httpSession.getAttribute( s: "bookingBeingCreated");

    bookingBeingCreated.setAutocamperId(id);

    return "redirect:/booking/new/customer";
}
```

Her er eksempel på dette. Efter bruger har valgt en autocamper, bliver de redirected til kundesiden ved et GET request på, /booking/new/customer. Implementeringen af dette pattern vises også godt i vores flow-chart over BookingController.

Repository Pattern (MF)

Hovedidéen med et repository pattern er, at man opdeler sin kode i et servicelag og et repository lag. Her vil servicelaget tage sig af alle business mæssige operationer, men aldrig arbejde med databasen selv. Alle databaseoperationer bliver nemlig varetaget af ens repositories. Målet er at man vil opnå en mere klar fordeling af ansvar indenfor programmet, hvilket vil øge anvendeligheden og overskueligheden af programmet, da man f.eks. ikke behøver at tænke på databaseimplementation, når man arbejder med servicelogik og omvendt.

Disse har vi som nævnt tidligere implementeret vha. interfaces, hvor vi øverst i hierarkiet har et CRUD-repository med en generisk type. Dette interface extendes af et mere specifikt interface for hver service. Det giver mulighed for mere specialiserede metoder for hver service, og sørger for at de stadig har de samme basale funktioner. Dette giver os også mulighed for nemmere at udskifte implementation, hvis det skulle være nødvendigt. Disse interfaces bliver herefter implementeret i repository implementationsklasser.

Samme mønster bliver anvendt til implementeringen af servicelaget, dog adskiller service metodenavnene sig fra navnene i ICrudRepository, selvom add i servicelaget kalder create i repository laget osv. Det skyldes at det virkede mere brugervenligt på denne måde, da det efterligner metodenavne fra f.eks. List interfacet.

For at forbinde vores repository lag med servicelaget, har anvendt en manager klasse, som vi har kaldt RepositoryManager. Denne klasse indeholder instanser af vores respository implementationer, hvilket gør at servicelaget kun behøver et objekt af denne manager for at kalde databaseoperationer. Vi mener at denne implementation øger forståeligheden og anvendeligheden af programmet, da hver serviceklasse ikke behøver at instantiere en række repositoryklasser. Som det ser ud lige nu, er servicelagets metoder relativt få, men formodningen er, at vi senere vil få markant flere metoder, hvorfor dette paterns brugbarhed vil blive understreget yderligere.

For at forbinde disse repositoryklasser til databasen har vi anvendt JDBC, der er implementeret vha. en DatabaseConnectionManager. Denne manager indeholder en singleton instans af et Connection objekt, så vi er sikre på kun at arbejde med denne ene instans.

Singleton (MC)

Vi har i vores program valgt at implementere Singleton pattern i 2 forskellige klasser; DatabaseConnectionManager og RepositoryManager.

Singleton pattern har til formål at sørge for, at kun en enkelt instans af en klasse er tilgængelig, og såfremt andre klasser skal bruge denne instans, kan de få den udleveret. Man opnår dette resultat ved at gøre constructoren for klassen private, sådan at det kun er klassen selv der kan kalde den. Derudover laver man en getInstance() metode som har til ansvar at instantiere klassen i en lokal privat variabel, såfremt det er første gang metoden kaldes, og/eller returnere denne instans. Dette kan ses eksemplificeret i nedenstående billede fra vores DatabaseConnectionManager klasse.

```java
private static DatabaseConnectionManager instance;
private static String user;
private static String password;
private static String url;

private DatabaseConnectionManager() {

}

public static DatabaseConnectionManager getInstance() {
    if (instance == null) {
        instance = new DatabaseConnectionManager();
    }

    return instance;
}
```

I tilfælde af vores DatabaseConnectionManager klasse, har klassen til ansvar at returnere en database forbindelse. Dette gør vi for at lade DriverManager klassen (en del af JDBC), som bruges til at hente en forbindelse, håndtere oprettelsen af nye database forbindelser. Det er altså kun igennem getConnection() metoden, at det er tilladt at hente en database forbindelse. Denne forbindelse bruges når vi i vores RepositoryManager skal instantiere vores repositories i RepositoryManager klassens constructor.

Vores RepositoryManager klasse er også implementeret som en Singleton. Det er præcis samme fremgangsmåde som med DatabaseConnectionManager klassen, forskellen er bare at RepositoryManager indeholder en instans af hver repository implementation, og returnerer en af disse alt afhængigt af hvilken get metode der kaldes i klassen. Tanken bag dette er, at det kun er nødvendigt at have 1 instans af et repository instantieret for at kunne tilgå databasen. Derudover gør det også, at det ikke er nødvendigt at have en instans af vores repositories i vores forskellige service klasser, da de bare kan kalde den statiske getInstance() metode i manager klassen, og derved få fat i det repository og de metoder som der er brug for.

Thymeleaf (MF)

Som nævnt tidligere er Thymeleaf et kernekomponent i vores system, da det tillader os at kunne hente database information til vores templates. Et eksempel på dette er, hvordan vi indlæser vores liste af autocampere:

```
<tr th:each="autocamper : ${filteredAutocamperMap}">
    <td th:text="${autocamper.getValue().getId()}"></td>
    <td th:text="${autocamper.getValue().getBrand()}"></td>
    <td th:text="${autocamper.getValue().getModel()}"></td>
    <td th:text="${autocamper.getValue().getYear()}"></td>
    <td th:text="${autocamper.getValue().getBeds()}"></td>
    <td th:text="${autocamper.getValue().getSeats()}"></td>
    <td th:text="${autocamper.getValue().getPriceDay()}"></td>
    <td>
        <form action="#" th:action="@{/booking/new/autocamper}" method="post">
            <input type="hidden" name="id" id="id" th:value="${autocamper.getValue().getId()}">
            <button class="btn btn-outline-primary" name="submit" type="submit" value="Vælg">Vælg</button>
        </form>
    </td>
</tr>
```

Dette gør vi ved at lade Thymeleaf iterere henover attributten “filteredAutocamperMap”, med Thymeleafs version af et for-each loop. Vi kan hermed hente disse værdier og indsætte dem i en liste, så listen bliver dynamisk oprettet ud fra hvilke autocampere, der er ledige i en given bookingperiode. Det givne id fra den valgte autocamper bliver herefter sendt videre til den korrekte controller metode, hvilket heller ikke ville være muligt uden brug af Thymeleaf.

Et andet eksempel på anvendelse af Thymeleaf er ved brug af fragments, der hjælper os til ikke at skrive redundant html-kode, der skal gå igen på mange sider. Dette ses i vores navigation bar:

```
<nav th:fragment="navbar" class="site-header sticky-top py-1">
    <div class="container d-flex flex-column flex-md-row justify-content-between">
        <span class="font-weight-bold py-2 d-none d-inline-block" th:text="Velkommen " th:sec:authentication="name"> bruger</span>
        <a class="py-2 d-none d-md-inline-block" href="/">Hjem</a>
        <a class="py-2 d-none d-md-inline-block" href="/booking/new">Opret Booking</a>
        <a class="py-2 d-none d-md-inline-block" href="/booking/list">Bookinger</a>
        <a class="py-2 d-none d-md-inline-block" href="/autocampere">Autocampere</a>
        <a class="py-2 d-none d-md-inline-block" href="/kunder">Kunder</a>
    </div>
</nav>
```

Dette gøres ved at annotere et element som et fragment ved “th:fragment”, hvorefter det nu kan tilføjes til html sider. På den måde vil en ændring i dette fragment betyde at navbar elementet på alle html sider som benytter dette fragment blive ændret. Det gør det altså en del nemmere at foretage ændringer, da det gentagende kodestykke er centraliseret et sted.

Service layer (MC)

Vi har også valgt at komme lidt ind på servicelaget, da vi mener det har en essentiel rolle i programmet og giver udtryk for en del af vores overvejelser.
Servicelaget agerer som en form for dør til repository laget, og metoder fra servicelaget bliver kaldt i vores controller klasser.

Vi mente det var nødvendigt at implementere et servicelag, for at sørge for at vores controller klasser udelukkende ville varetage controller specifikke opgaver. F.eks. er det at udregne totalprisen for en booking relevant for controlleren, da denne værdi skal bruges under bookingprocessen, og tilføjes til den model som viewet skal bruge, men vi mente ikke det nødvendigvis var controlleren, som skulle udføre denne handling. Derfor er servicelaget

et lag, hvor det var muligt at bortabstrahere metoder, som i deres natur handlede mere om business logic end egentlig navigering.

For at inddrage et eksempel på en metode fra servicelaget, som også er meget essentiel for at foretage en booking, har vi her en metode som genererer et map af ledige autocampers for en given periode:

```
@Override
public Map<Integer, Autocamper> getFilteredMapByPeriod(LocalDate periodStart, LocalDate periodEnd) {
    Map<Integer, Autocamper> autocamperMapToReturn;
    List<Booking> bookingList;

    try {
        autocamperMapToReturn = RepositoryManager.getInstance().getAutocamperRepository().readAllAsMap();
        bookingList = RepositoryManager.getInstance().getBookingRepository().readAll();

        for(int i = 1; i <= bookingList.size(); i++) {
            Booking currentBookingFromDatabase = bookingList.get(i - 1);

            if((periodStart.isBefore(currentBookingFromDatabase.getPeriodEnd().plusDays(1))
                    && periodEnd.plusDays(1).isAfter(currentBookingFromDatabase.getPeriodStart()))) {

                autocamperMapToReturn.remove(currentBookingFromDatabase.getAutocamperId());
            }
        }

        return autocamperMapToReturn;

    } catch(SQLException e) {
        e.printStackTrace();
    }

    return null;
}
```

Metoden bliver kaldt og brugt i processDateForm() metoden i BookingController klassen.

Metoden har til formål at returnere et map af autocampere, som ikke er booket i en given periode defineret af periodStart og periodEnd variablerne, som metoden får fra de datoer som salgsassistenten indtaster under oprettelsen af en ny booking.
Dette gøres ved at definere et for loop, hvis iterationer er defineret af størrelsen på bookingList, som er en ArrayList af alle bookinger fra databasen.

Hver individuel booking fra listen har en start og slut dato, som sammenlignes med de datoer der passeres ind i metoden. Skulle perioderne være ens eller overlappe, vil den tilknyttede autocamper bliver fjernet fra mappet og derved får man altså et map, som kun indeholder ledige autocampere for perioden. Bemærk at der for begge periodEnd bliver tilføjet 1 dag, som sørger for at bookingen på begge sider forlænges med 1 dag. Tanken bag dette er, at der skal være en “buffer dag”, som kan bruges på at servicere autocamperen, inden den kan bookes igen. Dette er udelukkende et valg fra vores side, og burde blive konsulteret med ejeren af Nordic Motorhome Rental.

Idéen bag at bruge et map som datastruktur og ikke en liste, er der flere grunde til:

1. Med en liste som f.eks. en ArrayList, vil det at fjerne autocampere ændre på autocamperens index, og der vil på den måde være et skel imellem autocamper objektets index i listen, og det id der findes i databasen.
2. En liste starter fra index 0, hvilket heller ikke er repræsentativt for databasen.

Ved at gemme autocamper objekter i et map, undgår vi disse problemer, da en key i mappet, med et givent autocamper id ikke vil ændres, når et Entry fjernes fra mappet. Mappet instansieres som et TreeMap i BookingController. Grunden til at vi har valgt TreeMap skyldes, at Entries i mappet bliver sorteret efter keys, og da Integer klassen har en naturlig orden, gør det at hver entry står i rækkefølge efter key fra lavest til størst.

Konklusion (MC & MF)

Til slut, kan vi konkludere at vi indledende i projektet endte med ikke at afgrænse nok, og derfor endte med en stor mængde design, som ikke endte i det endelige program. Det har dog også betydet, at vi har dannet et solidt fundament hvori, der er lagt mange tanker og som understøtter den eksisterende implementering. Dette betyder, at det i fremtidige iterationer bør være nemmere at videreudvikle på systemet.

Vores valg af frameworks og teknologier viste sig igennem vores risikoanalyse at være den største faktor for at projektet kunne fejle. Vi så det dog som en risiko, der var værd at tage med og lagde af den grund også mange kræfter i løbet af elaboration fasen i at undersøge og forsøge os med implementering af disse frameworks og teknologier. Et udbytte af dette valg ville kunne vise sig i fremtidige iterationer, hvor fundamentet for en god systemstruktur allerede er lagt.

Igennem vores Feasability Study og SWOT-analyse fandt vi frem til at virksomheden ville have stor gavn af et centraliseret IT-system, og at det ville kunne gavne deres vækst og omsætning, samt effektivisere arbejdet i firmaet.

I forhold til vores implementering, har vi ved at bruge design patterns hvor det gav mening forhåbentligt skabt en kodebase, som er letforståelig og via high cohesion og low coupling mulig at kunne redigere, tilføje og på anden vis ændre uden for meget besvær.

Dette er det første projekt af denne skala, hvor vi har benyttet så mange teknologier, frameworks og design patterns, og af den grund har vi også lært meget af projektet. Da vi begge ikke er erfarne programmører, betyder det også at vi har flere ting som kunne være gjort anderledes.

Eksempelvis har BookingController klassen været svær at arbejde med, da den har håndteret et stort workflow, og derfor indeholdte mange metoder. En eventuel ændring til denne kunne være at opdele den i flere under controllere, som hver især ville have ansvar for en specifik del af bookingprocessen.

En anden ting som vi mener burde være en del af systemet (og kunne tilføjes i en senere iteration), er fejlhåndtering i forbindelse med modtagelse af brugerinput på hjemmesiden.

Dette ville være med til at sikre korrekt data i databasen og forbedre brugeroplevelsen markant.

Appendices

Appendix 1 – Brief Use Cases

Søg autocamper

En salgsassistent vælger at få vist en liste for autocampere i systemet. Systemet viser listen. Salgsassistenten leder efter en bestemt autocamper og vælger at bruge søgefunktionen. Salgsassistenten indtaster bilens model i søgefeltet. Systemet opdaterer listen og viser alle biler med modeller matchende det indtastede input.

Opret autocamper booking

En salgsassistent ønsker at booke en autocamper. Salgsassistenten vælger at få vist listen for autocampere. Salgsassistenten vælger den autocamper der skal bookes og får vist detaljer for autocamperen. Salgsassistenten indtaster en fra/til dato og vælger at booke. Salgsassistenten indtaster kundens information. Salgsassistenten vælger pick-up og drop-off point samt tilføjer 1 cykelstativ og 1 barnesæde som tilbehør til bookingen. Systemet fortæller salgsassistenten at bookingen er gennemført.

Se autocamper booking

En salgsassistent ønsker at se en eksisterende autocamper booking. Salgsassistenten vælger at se eksisterende bookinger. På listen over bookinger finder salgsassistenten den ønskede booking og beder systemet vise flere detaljer. Systemet præsenterer en oversigt over bookinginfo, bilinformation, kundeinformation, tilbehør prisinformation.

Rediger autocamper booking

En salgsassistent ønsker at se en eksisterende autocamper booking for at rette hvilket tilbehør der er bestilt. Salgsassistenten vælger at se eksisterende bookinger. På listen over bookinger finder salgsassistenten den ønskede booking og beder systemet vise flere detaljer. Systemet præsenterer en oversigt over bookinginfo, bilinformation, kundeinformation, tilbehør prisinformation. Salgsassistenten retter antallet af barnesæder fra 2 til 1. Systemet opdaterer de præsenterede priser. Salgsassistenten bekræfter ændringen og gemmer.

Slet autocamper booking

En salgsassistent ønsker at slette en autocamper booking. Salgsassistenten vælger at se eksisterende bookinger. På listen over bookinger finder salgsassistenten den ønskede booking og beder systemet slette bookingen.

Opret kunde

En salgsassistent ønsker at oprette en ny kunde, udenom en booking. Salgsassistenten vælger at få vist listen for kunder. Salgsassistenten trykker på opret. Salgsassistenten indtaster kundens information. Salgsassistenten gemmer kunden i systemet.

Se kunde

En salgsassistent ønsker at finde en kunde ud fra telefonnummer i systemet. Salgsassistenten vælger at få vist en liste over kunder. På listen finder salgsassistenten en kunde med et matchende telefonnummer.

Rediger kunde

En salgsassistent ønsker at redigere oplysninger for en eksisterende kunde. Salgsassistenten vælger at få vist en liste over kunder. Salgsassistenten finder den ønskede kunde ud fra navn, efternavn og email. Salgsassistenten vælger kunden og beder systemet vise detaljer. Systemet præsenterer yderligere information om kunden. Salgsassistenten redigere kundens email og gemmer.

Slet kunde

En salgsassistent ønsker at slette en kunde. Salgsassistenten vælger at få vist en liste over kunder. Salgsassistenten finder den ønskede kunde ud fra navn og efternavn. Salgsassistenten vælger kunden og beder systemet slette kunden.

Appendix 2 – Samarbejdskontrakt

Denne samarbejdskontrakt er gældende for projektet ”Datamatikeruddannelsen – Dat19B – 2. semester eksamensprojekt” gældende fra d. 11. maj 2020 – 04. juni 2020. Gruppemedlemmerne er som følger: Mads Kjærgaard Christensen og Michael Fuglø.

§ 1 Alle gruppemedlemmer har en stemme i gruppedemokratiet.

§ 1.1 Ved brud på paragrafsæt kan et gruppemedlem instantiere en afstemning, som kan medføre en advarsel. Ved 3 advarsler vil vejleder kontaktes og en straf vil blive forhandlet.

§ 1.2 Gruppekontrakten er gældende for alle gruppemedlemmer, der har underskrevet kontrakten.

Fagligt relaterede regler:
§ 2 Hvert gruppemedlem skal som minimum bestræbe sig efter at overholde de aftalte opgaveinddelinger.

§ 2.1 Hvis gruppemedlem ikke er i stand til at overholde den givne aftale, skal gruppemedlem henvende sig til gruppen via Discord og begrunde sin dvaskhed. Herefter finder gruppen en passende løsning på opgaveinddelingen, hvis begrundelsen findes tilstrækkelig af et demokratisk flertal i gruppen.

Regler for fravær:
§ 3 Hvert gruppemedlem skal bestræbe sig på at overholde de aftalte mødetider. I tilfælde af forsinkelse skal det kommunikeres med gruppen hurtigst muligt.

§3.1 I tilfælde af sygdom/fravær skal gruppemedlem henvende sig til gruppen via Discord og begrunde sit fravær. Herefter finder gruppen en passende løsning på problemstillingen, hvis begrundelsen findes tilstrækkelig af et demokratisk flertal i gruppen.

Sociale regler:
§ 4 Vi skal lytte og respektere hinanden og hinandens forskelligheder.
§ 4.1 Konflikter bliver løst ved demokratisk flertal og majoriteten vinder.

Mads K. Christensen *Michael Fuglø*

Litteraturliste

1. https://kinsta.com/knowledgebase/what-is-github/ (læst 31-05-2020)
2. https://trello.com/tour (læst 31-05-2020)
3. https://www.youtube.com/watch?v=hQn9Z6bVqgk (Spring Framework Guru - Inversion of Control and Spring) (set 01-06-2020)
4. https://dzone.com/articles/understanding-the-basics-of-spring-vs-spring-boot (læst 02-06-2020)
5. https://www.youtube.com/watch?v=Ch163VfHtvA (Telusko – What is Spring Boot? | Introduction) (set 01-06-2020)
6. https://spring.io/projects/spring-ws#overview (læst 02-06-2020)
7. https://www.thymeleaf.org/faq.html#is-web-framework (læst 02-06-2020)
8. https://medium.com/@Vincentxia77/what-is-mobile-first-design-why-its-important-how-to-make-it-7d3cf2e29d00 (læst 02-06-2020)
9. https://stackoverflow.com/questions/7249871/what-is-a-build-tool (læst 02-06-2020)
10. https://maven.apache.org/what-is-maven.html (læst 02-06-2020)
11. https://en.wikipedia.org/wiki/Build_automation (læst 02-06-2020)
12. https://docs.oracle.com/javase/tutorial/jdbc/basics/gettingstarted.html (læst 02-06-2020)
13. https://dev.mysql.com/doc/refman/8.0/en/innodb-limits.html (læst 12-05-2020)
14. https://aws.amazon.com/marketplace/pp/Dongguan-SainStore-e-Commerce-Co-Ltd-Tomcat-Java-M/B00W2LG0RW (læst 01-06-2020)
15. https://www.valutakurser.dk/ (læst 02-06-2020)
16. https://tableplus.com/blog/2018/08/mysql-the-difference-between-int-bigint-mediumint-smallint-tinyint.html (læst 03-06-2020)