

FØRSTEÅRSPRØVE

IT-System til Scooterland

UCL Erhvervsakademi og Professionshøjskole

Datamatiker, 2. semester – DMVF241 – Vejle



Studerende: Amjad Nizar Renno, Karina Kristensen, Mathias Dalby Sternberg, René Rønning Hertz og Thomas Rud Nielsen

Undervisere: Henrik Steen Krogh, Martin Knudsen og Per Larsen

13 – 12 – 24

| | |
|---|----|
| Indledning - Karina | 4 |
| Problemstilling - Karina | 4 |
| Problemformulering - Karina | 4 |
| Tilgangsvinkler - Karina | 5 |
| Opgaveløsning | 5 |
| Agilt arbejde og SCRUM - Karina | 5 |
| Product Backlog - Karina | 6 |
| Analyse | 6 |
| Risikoanalyse - Thomas | 6 |
| Funktionalitet - Karina | 8 |
| User Stories | 9 |
| Use Cases | 10 |
| Design | 14 |
| Sekvensdiagram (SD) - Amjad | 14 |
| Systemsekvensdiagram (SSD) - Amjad | 16 |
| GoF-Mønstre - Karina | 18 |
| GRASP-Mønstre - Karina | 18 |
| Arkitektur - Karina | 19 |
| Hosting - Karina | 19 |
| Brugergrænsefladedesign - Karina | 20 |
| Processen | 20 |
| Sprint 0 | 20 |
| Use Case Diagrammer, SSD- og SD-diagram (Amjad Renno) | 21 |
| Risikoanalyse og rapportopsætning (Thomas) | 21 |
| Domænemodel (Karina) | 21 |
| Første iteration af Klassediagrammet (Rene) | 22 |
| E/R Diagram (Rene) | 23 |
| Sprint 1 | 24 |
| Database og Entity Framework (Rene) | 24 |
| Ordre side for mekaniker (Thomas) | 25 |
| Faktura Page (Amjad) | 25 |
| Opret kunde (Mathias) | 25 |
| Opret ordre - Første iteration (Karina) | 27 |
| Sprint 2 | 31 |
| Ordre Oversigt (Rene) | 31 |
| Forbindelser mellem mekaniker og mærke (Thomas) | 32 |
| Ydelse Oversigt (Rene) | 32 |
| Faktura formatering (Amjad Renno) | 33 |
| Sprint 3 | 34 |
| Opret Ordre - Anden iteration (Rene) | 34 |
| Lejning af Scooter (Rene) | 37 |
| BackupSystem og Login (Thomas) | 38 |

| | |
|--|----|
| Forbedring af brugeroplevelsen (Amjad Renno) | 38 |
| Sprint 4 | 39 |
| Kunde Oversigt (Rene) | 39 |
| Login og CSS + HTML (Thomas) | 40 |
| Slette faktura og tilføje en søgning (Amjad Renno) | 41 |
| GRASP, Overskrift, UNIT-tests, ForbindMærkeTilMekaniker (Karina) | 42 |
| Sprint 5 | 45 |
| Validation (Rene) | 46 |
| Rapportskrivning og final touch-ups (Karina) | 47 |
| Rense koder (Amjad) | 47 |
| Visuelle justeringer og fejlfinding (Thomas) | 48 |
| Perspektivering | 48 |
| Konklusion | 49 |
| Litteraturliste | 49 |
| Systemsekvensdiagram (SSD): https://www.edrawsoft.com/article/uml-system-sequence-diagram.html | 49 |
| Bilag: | 49 |

Indledning - Karina

Scooterland er en virksomhed, hvor vi specialiserer os i at tilbyde ydelser og produkter til kunderne, som har en interesse for scootere. Vi vil gerne skabe en enkel og effektiv platform, der møder behovene hos både kunderne og vores medarbejdere. Vi har derfor udviklet et system, der inkluderer roller som kontordamer, værkførere og mekanikere. Vi har kombineret en desktop-applikation til intern brug og en Blazor-baseret hjemmeside for kunder, som gør det brugervenligt for kunder at oprette og administrere ordrer.

Problemstilling - Karina

Som gruppe ønsker vi at lave en Scooterland hjemmeside, hvis formål er at yde ydelser til kunderne, altså udlejning af scootere samt reparation og service, hvorved vi også ønsker at have et system oppe at køre til administration af ordre, kunder, ydelser, mekanikere mm. Og et login system til vores mekanikere, værkfører og kontordamerne.

Hvordan kan vi sikre, at kunder nemt kan oprette ordrer på vores hjemmeside og få adgang til relevante ydelser som reparationer, service og udlejning af scootere?

Hvordan kan vi skabe en effektiv integration mellem vores WinForms-baserede system og den Blazor-baserede hjemmeside?

Hvordan kan vi forbedre kommunikationen og arbejdsprocesserne mellem kontordamerne, værkførerne og mekanikerne, hvor vi samtidigt behandler kundedata sikkert og effektivt?

Hvordan sikrer vi en brugeroplevelse, der er instinktiv for kunderne, og samtidig et internt system, der understøtter virksomhedens arbejdsprocesser?

Problemformulering - Karina

Vi vil gerne løse de problematikker der kommer, ved at lave en hjemmeside, som er brugervenlig, både overfor kunderne såmænd også medarbejderne. Hvordan kan vores Scooterland projekt udvikle en brugervenlig og effektiv hjemmeside, som gør det muligt at kunder kan oprette ordrer på vores ydelser, såsom service, reparation og udlejning af scootere, hvor vi så på samme tid kan sikre en god integration med vores interne systemer, og også forbedre kommunikationen mellem vores medarbejdere og kunder?

Hvilke funktioner skal hjemmesiden have for at opfylde kundernes behov?

Hvordan kan vi lave en løsning, der hjælper forskellige brugerroller, som kontordamer, værkfører og mekanikere?

Hvordan kan vi sikre datasikkerhed og en instinktiv brugeroplevelse?

Tilgangsvinkler - Karina

I Scooterland-projektet har vi arbejdet på at udvikle et system der gør hverdagen nemmere for både ansatte og kunder. For at skabe struktur i vores arbejde bruger vi SCRUM, hvor vi planlægger og udfører opgaver gennem sprints. I sprint 0 fokuserede vi på at lave ER-diagrammer, domænemodeller og skitser til hjemmesiden.

Vi bruger Trello som værktøj til at holde styr på opgaverne. Her inddelte vi opgaverne i tre kategorier: "To do", "Doing", og "Done".

Vores tilgang er brugercentreret. Vi udvikler et system, der kombinerer en WinForms-app til ansatte og en Blazor-hjemmeside til kunder. Brugercentreret handler det om at tilfredsstille mekanikernes, værkførernes og andre ansattes behov, ved at sikre at de kan udføre deres arbejdsopgaver nemt og effektivt. Samtidig sørger vi for, at kunderne får en god oplevelse med systemet, f.eks. ved at gemme deres ordrehistorik og give dem oversigt over deres regninger.

Målet er at skabe et sammenhængende system, hvor ansatte kan registrere ordrer, planlægge opgaver, og opkræve betalinger, mens kunder kan føle sig trygge ved, at deres data er gemt sikkert. Vi prioriterer også meget at gøre systemet fleksibelt, f.eks. ved at ansatte kan ændre priser eller planlægge opgaver baseret på mekanikernes specialer indenfor mærker.

Denne tilgang sikrer, at vi arbejder målrettet og opfylder både virksomhedens og kundernes behov i udviklingen af Scooterland.

Opgaveløsning

Agilt arbejde og SCRUM - Karina

Iterative sprints: Vores projekt er delt op i mindre dele, som har gjort at vi kunne fokusere på specifikke funktioner.

Trello: Vi har brugt Trello for at holde styr på vores opgaver, opdelt i To do, Doing og Done, hvilket har givet os overblik.

Daily stand ups: Vi har haft daglige møder på Discord for at dele status, planlægge næste skridt og fortælle hvilke udfordringer vi har stødt på.

Kundeorienteret udvikling: Vores user stories har fokus på hvad de forskellige brugere (ansatte, mekanikere, værkfører) ønsker og har behov for, hvilket sikrer at systemet er hvad brugerne ønsker.

Product owner - Henrik

- Prioriterer opgaverne i Trello (vores product backlog) og sikrer at funktionaliteten afspejler kundens behov.
- Afklarer hvad han ønsker.
- Giver opgaven og siger hvad han vil have. Kommer med forslag til hvad han gerne vil have i næste sprint.

SCRUM master - Rene

- Sørger for at SCRUM-processen følges, og at teamet kan arbejde uden forhindringer.
- Facilitator for daily standups og hjælper med problemløsning.
- Har ansvaret for at fortælle hvilke funktioner der er vigtigst.

Udviklingsteam - Amjad, Mathias, Thomas, Karina

- Består af udviklere, der implementerer funktioner og løser de tekniske opgaver.

Product Backlog - Karina

Dette er vores overblik over de funktioner der skal udvikles til Scooterland. De er opdelt i to do, doing og done. Vi har brugt Trello til at holde styr på tingene.

Analyse

Risikoanalyse - Thomas

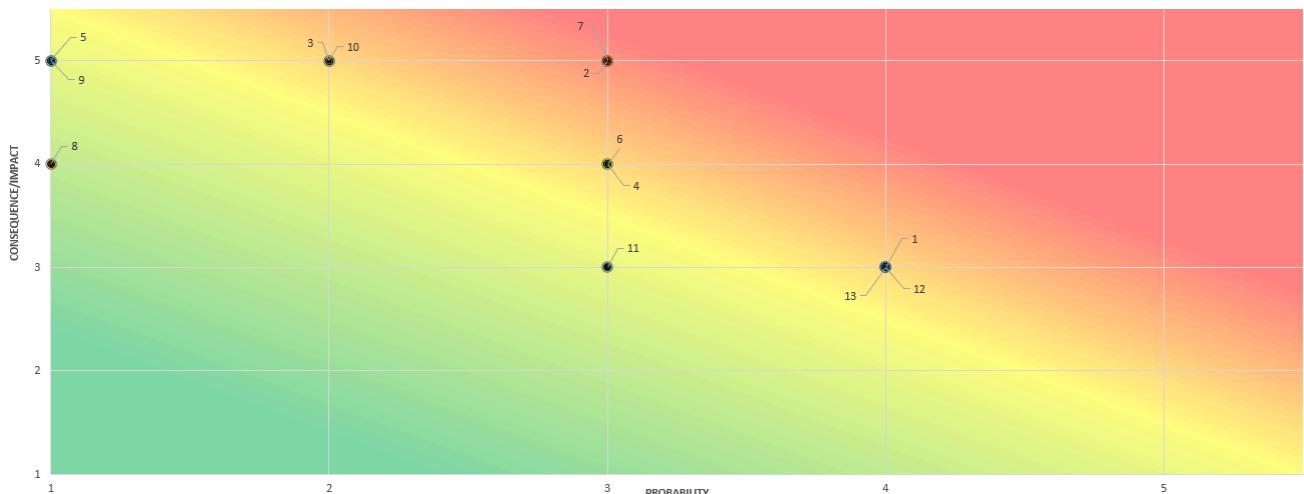
Som team blev vi hurtigt enige om vigtigheden i at identificere samtlige risici under udvikling af projektet. Da dette ville give os et bedre indblik i hvad der kan gå galt, samt give os værktøjerne til, at forhindre fejl i at opstå, samt at have en plan klar, hvis uheldet er ude. Dette blev fokus for i sprint 0, med udgangspunkt i Henrik Steen Kroghs skabelon for risikoanalyser, udformede vi en analyse med 14 identificerede risici.

Overordnet, satte vi vores risici ind i forskellige overkategorier; Tekniske Risici, Kvalitetsrisici, Tidsmæssige Risici, Ressourcebegrænsninger, Kommunikationsrisici og Erfarings Risici.

| Risiko Analyse - Matrix | | | | | | | |
|-------------------------|---|-------------------|----------------------------|------------|--------|--|--|
| ID | Risk: What can go wrong? | Probability (1-5) | Consequence / Impact (1-5) | Risk score | Resp. | Preventive action | Mitigation action |
| 1 | Tekniske Risici : Merge Konflikter ved brug af Git | 4 | 3 | 12 | (Alle) | Implementér klare branch- og merge-strategier; sørг for regelmæssige code reviews. | Løs konflikter under scrum møder. |
| 2 | Tekniske Risici : Fejl i Database opsætning. | 3 | 5 | 15 | (Alle) | Ved opstart af projektet, lav et gennemført E/R Diagram med rigtige relationer, for at undgå problemer med databasen gennem projekts forløb. | Opset en backup-database eller caching-mekanisme for at minimere påvirkning under nedbrud. |
| 3 | Kvalitetsrisici: Manglende funktionalitet. | 2 | 5 | 10 | (Alle) | Afhold møder med Vejledere for at sikre tydelige funktionskrav. | Brug feedback fra vejleder til at identificere manglende funktioner og prioriter dem i fremtidige sprints. |
| 4 | Kvalitetsrisici: Teknologisk kompleksitet i UI og applikationen. | 3 | 4 | 12 | (Alle) | Forenkle UI-design; vælg velunderstøttede og kompatible løsninger, samt designfilosofier. | Sæg hjælp hos vejleder eller brug gestaltlovene som reference. |
| 5 | Kvalitetsrisici: Fejlforklaring af opgaven, casen misforståes. | 1 | 5 | 5 | (Alle) | Afhold indledende møder for at klarlægge projektets omfang og krav med hele teamet. | Gennemgå regelmæssigt kravene med teamet for at sikre sammenhæng med projektmålene. |
| 6 | Tidsmæssige Risici : Sprint-overbelastning - Fejlestimeringer ift. Hvad der kan nås i sprint. | 3 | 4 | 12 | (Alle) | Brug fornuftig sprint planlægning, som læner sig op af productbackloggen. | Juster sprint-mål under sprinten, hvis nødvendigt, for at reducere overbelastning. |
| 7 | Tidsmæssige Risici: Sprint-underbelastning - For lidt arbejde bliver lavet i sprint, fører til overbelastning i næste sprint. | 3 | 5 | 15 | (Alle) | Analysér arbejdsbelastningen under sprint-planlægningen og tildele ekstra opgaver om nødvendigt. | Omfordel arbejdshylden i kommende sprints baseret på teams aktuelle tempo og kapacitet. |
| 8 | Tidsmæssige Risici: Planlægning henger ikke sammen med aktuelt produkt. | 1 | 4 | 4 | (Alle) | Gennemgå og opdater produkt-backlog regelmæssigt for at sikre sammenhæng med projektmålene. | Juster planlægningen gennem sprint reviews for at matche produkts faktiske status. |
| 9 | Ressourcebegrænsninger: Gruppemedlemmer mister interesse. | 1 | 5 | 5 | (Alle) | Involver teamet i beslutningstagning for at øge engagementet. Sørg for at teamet er motiveret. | Teamleder tager ansvar for at motiverve teamet, brug Pink's teori eller Herzberger. |
| 10 | Ressourcebegrænsninger: Gruppemedlen(mer) bliver fraværende, f.eks. Grundet sygdom. | 2 | 5 | 10 | (Alle) | Förbered teammedlemmer, så de kan dække hinandens roller ved behov. | Omfordel opgaver eller tilkald vejleder for kompensation. |
| 11 | Kommunikationsrisici: Manglende kommunikation i gruppen. | 3 | 3 | 9 | (Alle) | Opsæt daglige stand-ups og check-ins for at sikre løbende opdateringer. | Brug samarbejdsværktøjer som Discord og spor opgaver i et fælles projektsystringsværktøj som Trello. |

De tekniske risici, vurderede vi til at være de farlige, overordnet er det også den kategori, som scorer højest på risiko grafen. Dette skyldes, at de tekniske risici, er risici som har en stor sandsynlighed og en ret høj konsekvens. I vores tilfælde er det merge konflikter i Git, samt fejl i database opsætning eller migration. Hvilket er grunden til, at tekniske risici forud for projektet er vægtet som nogle af de farligste.

Ved kvalitets risici, har vi risici som er ret usandsynlige, men har en rigtig voldsom konsekvens for projektets helhed. Det er risici som manglende funktionalitet og teknisk kompleksitet ift. både UI og Funktionalitet. Hvis vi som gruppe mangler funktionalitet pga. misforståelser eller fejlfortolkning af opgave casen, har det store konsekvenser. Det samme kan siges hvis udviklerne finder funktionalitet al for teknisk komplekst.



Vi lavede tidsmæssige risici, da vi så sandsynligheden for dårlig sprint planlægning. Dette dækker over alt fra, under- og overbelastning i sprints, til at planlægningen ikke hænger sammen med produkt-backloggen. Dette er væsentlige risici at undgå, da planlægningen af sprints styrer hele workflowet for teamet. Det er dog også første gang at dette team arbejder sammen, så indbyrdes kender vi ikke hinandens arbejdstempo mm. Dette gør det sværere at planlægge sprints til perfektion.

Ressourcer begrænsninger dækker over de ressourcer vi har som team, altså vores udviklerteam. Det vil få katastrofale konsekvenser for udviklingsprocessen, hvis en udvikler bliver ramt af alvorlig sygdom eller mister motivation. Det er op til teamet indbyrdes at motivere hinanden. Ved at identificere disse risici, kan vi forberede os på ressource begrænsninger og dække efter hinanden ved behov.

Kommunikation er essentielt for projektets fremgang. Ved at give opmærksom på dette fra start af, kan teamet eliminere diverse konsekvenser. Det kan være en udvikler oplever problemer med udviklingen, hvis dette ikke bliver kommunikeret, kan resten af teamet ikke hjælpe. Derfor er kommunikation i høj fokus.

Sidst kiggede vi på risici ved manglende erfaring. Denne kategori er også vigtig, da alle teammedlemmer er uerfarne i versionsstyring og SCRUM i teams. Det er den manglende erfaring, som kan skabe nogle af de fornævnte risici. Derfor er det vigtigt, at teamet læner sig op ad SCRUM-mytenologien, for at vænne sig til den arbejdsgang. Teamet skal sætte klare retningslinjer for hvordan man arbejder agilt, og tænke dette ind i versionstyring.

Funktionalitet - Karina

Scooterland-projektet er udviklet som en moderne og brugervenlig platform. Dens funktioner er administration af scooterer, kunder, mekanikere og deres relaterede oplysninger. Programmet er designet til at forenkle arbejdsprocesser og skabe overblik over virksomhedens data gennem en instinktiv brugerflade og god backend.

Kernefunktionalitet:

- Kunde- og mekanikeradministration. Programmet gør det muligt at create, read, update og delete data for både kunder og mekanikere. Dette inkluderer funktioner som
 - Tilknytte en fast mekaniker til kunde. Sikre at kundens oplysninger er opdaterede og let tilgængelige.
- Mærkeforbindelse til mekanikere og kunder. Mekanikere kan knyttes til scooter-mærker, så arbejdsopgaver kan fordeles til mekanikernes ekspertise. Kunder kan tilføje mærker, f.eks. Honda eller Yamaha, hvilket gør det nemmere at matche dem med den rette mekaniker.
- Integration mellem kunder, scooter og mekanikere. Systemet understøtter en direkte forbindelse mellem kunder, deres scooter og den mekaniker, der servicerer dem. Dette skaber en problemfri proces for både planlægning og udførelse af opgaver.
- API-integration og datalagring. Backend-programmet benytter sig af API'er til at hente og opdatere data dynamisk. Det betyder at ændringer straks afspejles i systemet uden behov for genindlæsning af siden. F.eks: Hentning af mekanikerlist og mærker via Rest API'er. Opdatering af kundeoplysninger med tilknytning til mekanikere og scooter-mærker.
- Brugergrænseflade i Blazor. Vi har brugt Blazor for en interaktiv frontend. Funktionaliteten inkluderer: Dropdown menuer til valg af mekanikere og mærker. Formularer til at opdatere kundedata og forbinde dem til specifikke mekanikere. Dynamisk opdatering af data med feedback til brugeren, såsom fejlmeddelelser og success meddelelser.

Anvending af programmet kan se sådan her ud:

En kunde kan oprettes med deres scooter oplysninger, og en passende mekaniker kan tilknyttes. Administratoren kan hurtigt finde og opdatere kundedata for at holde systemet aktuelt. Arbejdsfordelingen optimeres med at kunne knytte mekanikere til mærker, så kunder får den bedst mulige service.

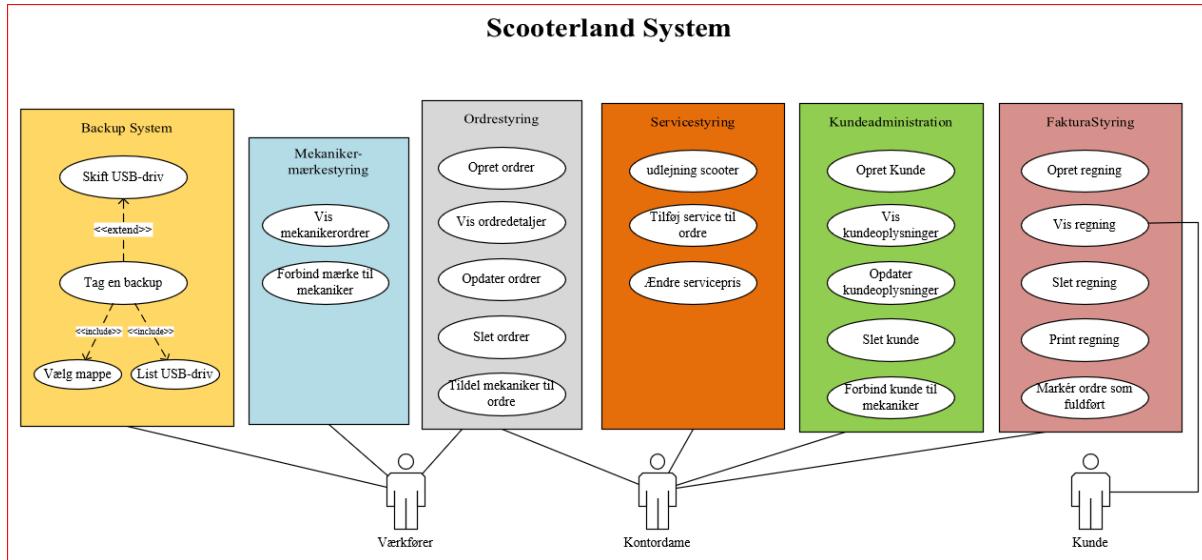
User Stories

As a _ I want to _ so that _

Som _ vil jeg _ så _

- Som ansat vil jeg kunne logge ind så programmet kan huske mig.
- Som mekaniker vil jeg have at programmet regner ud hvor meget jeg skal opkræve af en kunde for en lejet scooter så jeg ikke behøver selv at regne det ud.
- Som mekaniker vil jeg at alle vores ydelser og varer er oprettet i systemet, så jeg kan tilgå disse ved ordreoprettelse.
- Som værkfører vil jeg kunne oprette eller slette ydelser og varer i systemet, så jeg kan opdatere vores virksomheds udbud.
- Som værkfører vil jeg gerne kunne forbinde en kunde med en mekaniker, så de kan blive betjent af en mekaniker de kender og er tryg ved.
- Som værkfører vil jeg kunne markere en mekaniker med et speciale inde for et mærke, så jeg kan give dem arbejde, de specialiserer sig for.
- Som mekaniker vil jeg kunne ændre prisen på en ydelse for en kunde, hvis de har forhandlet det til en anden pris, så jeg kan holde vores løfte.
- Som værkfører vil jeg have at systemet gemmer data om en kundes ordre så jeg kan have oversigt over deres ordre historik.
- Som værkfører vil jeg have at systemet registrerer en kunde i systemet, hvis de ikke allerede er der, så jeg kan få gemt nye kunder i systemet.
- Som mekaniker vil jeg kunne lave en regning for alle ydelserne en kunde har købt så jeg kan nemt opkræve penge for dem.
- Som værkfører vil jeg planlægge reparationsopgaver og deler dem til de relevante mekanikere, så kan jeg organisere arbejdet processen.
- Som mekaniker vil jeg have at systemet opretter en ordre der er forbundet til en kunde, så jeg ved hvad der skal laves og til hvem.
- Som ansat skal jeg kunne se fakturaen i programmet, så jeg ved hvor meget der skal opkræves af kunden og hvad ordren består af.
- Som bruger skal jeg kunne se en oversigt for en kunde for hvor registrerede timer og forbrug er på en given ordre, så jeg har data over enkelte kunder.
- Som ansat vil jeg registrere kontante betalinger for køb som olie og dæk, så jeg spore det daglige salg.
- Som ansat skal jeg have en winforms app der kan kopiere den valgte folder til en usb stick, så jeg kan lave backups på virksomhedens data.

Use Cases



Der er ingen tvivl om, at use cases er en vigtig del af systemdesign, da dette diagram giver et overblik over de roller og funktioner, der er knyttet til hver enkelt bruger. I ScooterLand-systemet er use cases designet til specifikt at definere hver enkelt medarbejders opgaver såsom værkføreren og kontordamen, hvor opgaver er fordelt mellem medarbejderne såsom oprettelse af ordrer, fakturering og sikkerhedskopiering af virksomhedsdata.

For at få et overblik over, hvilke funktioner hver ansat rolle har i ScooterLand-systemet, er lavet use cases i et **brief format**. Disse use cases er baseret på funktionerne beskrevet under systemets design og udvikling. Her kan ses de vigtigste roller og deres funktioner.

UC1. Værkører

- 1.1. Opretter ordre.
- 1.2. Ser detaljer om ordrer.
- 1.3. Opdater ordre.
- 1.4. Sletter ordre.
- 1.5. Tildeler mekanikere til ordrer.

- 1.6. Viser en oversigt over mekaniker ordrer.

1.7. Forbinder scootermærker til mekanikere.

1.8. Tager backup af systemdata.

1.9. Skift mellem USB-drev under backup.

1.10. Vælger en mappe.

1.11. Vis Liste Driv.

UC2. Kontordame

2.1. Opretter ordre.

2.2. Ser detaljer om ordrer.

2.3. Opdater ordre.

2.4. Sletter ordre.

2.5. Tildeler mekanikere til ordrer.

2.6. Håndterer scooterudlejning.

2.7. Tilføjer ekstra services til ordrer.

2.8. Kan ændre servicepris

2.9. Opretter kunder inde i systemet.

2.10. Vis kundeoplysninger

2.11. Opdater kundeoplysninger.

2.12. Sletter kunder fra systemet.

2.13. Forbinder kunder til mekanikere.

2.14. Genererer fakturaer for afsluttede ordrer.

2.15. Printer fakturaer til kunder.

2.16. Markerer ordrer som afsluttet.

UC3. Kunde

3.1. Ser detaljerte fakturaer.

3.2. få en kopi af regningen.

For at få en dybdegående detaljeret, hvordan specifikke funktioner i systemet fungerer, er der lavet **fully dressed use cases** for FakturaStyring. Disse beskriver dybdeinteraktion mellem roller og systemet i nøglefunktioner.

Use Case Name: Opret Faktura

Scope: FakturaStyring i ScooterLand System

Primary Actor: Kontordame

Stakeholders and Interests:

- Kontordame: Hun vil oprette en korrekt faktura baseret på ordrenummeret.
- Kunde: forventer en nøjagtig faktura, der inkluderer alle relevante servicedetaljer og priser.

Garanti for succes:

- Fakturaen genereres og gemmes korrekt i systemet og er tilgængelig for både kontordame og klient.

Main Flow:

1 Kontordamen logger ind i ScooterLand-systemet.

2. Kontordamen indtaster ansøgningsnummeret.

3. Kontordamen klikker på "Opret faktura".

4. Systemet viser en meddelelse om, at fakturaen blev fundet

5. Systemet viser ordrelisten med dens detaljer, herunder reservedele, serviceydelser og status.
6. Kontordamen vurderer, at anmodningen er fuldstændig
7. Systemet viser en meddeelse om, at anmodningen blev gennemført
8. Kontordamen bestemmer metoden for at sende fakturaen til klienten.
9. Systemet sender en besked om, at handlingen er gennemført
10. Kunden modtager fakturaen

Alternativt flow:

- Faktura ikke fundet:
 - Systemet viser en fejlmeddeelse og beder kontordamen om at indtaste det korrekte ordrenummer for at oprette fakturaen.

Særlige krav:

- Fakturaen skal indeholde:
 - Kundens navn og oplysninger.
 - Ansøgningens status er komplet eller ej.
- Navnet på mekanikeren, hvis det anmodes om, afhænger af tilstedeværelsen af en mekaniker
- Ansøgningsnummer og dato.
- Detaljerede priser for hver service eller reservedel. - ScooterLand navn, adresse og kontaktoplysninger.

Konklusion

Use cases i ScooterLand-systemet illustrerer rollerne mellem medarbejdere og deres respektive roller. Såsom oprettelse af fakturaen af kontordamen for at give et tip til udviklerne om at implementere systemet fungerer korrekt.

Ved at fordele opgaver mellem medarbejderne og give hver af dem en rolle, letter det en smidig implementering af systemfunktioner, og gør ScooterLand-systemet til et brugervenligt site, der imødekommer virksomhedens behov for håndtering af ordrer, fakturaer og backups. Det er også en vigtig ressource til at vedligeholde systemet eller udvide det i fremtiden.

Design

GoF-Mønstre - Karina

GoF er inddelt i 3 kategorier. Creational patterns, structural patterns og behavioral patterns. GoF står for gang of four og er altså de 4 forfattere der har skrevet bogen Design patterns: Elements of reusable object-oriented software.

Nogle af de eksempler på GoF-mønstre vi bruger er:

Factory method: Mønstret bruges når vi vil oprette objekter som f.eks. kunder, mekanikere eller mærker. Det sikrer, at vi altid skaber objekter på en ensartet måde uden at koden ved præcist, hvordan de bliver oprettet. Vi kan f.eks. bruge Factory method til at oprette kunder baseret på specifikke krav, som f.eks. deres type scooter eller ønsket mekaniker.

Singleton: Sikrer at der kun er én instans af en bestemt klasse i hele programmet. Vi bruger singleton til at sikre, at der kun er én global liste over mærker, som vores mekanikere kan arbejde med.

Decorator: Med decorator tilføjer vi dynamisk funktioner til et objekt uden at ændre dets oprindelige kode. Når en kunde tilføjer ekstra ydelser til sin reparation, som f.eks. en særlig vask eller tuning, bruger vi decorator til at opbygge ydelserne uden at lave helt ny kode for hver kombination af service.

Observer: Mønstret giver os mulighed for at oprette et system, hvor objekter kan ”lytte” til ændringer i et andet objekt. Vi bruger Observer, hvis vi vil give kunder besked når der sker ændringer, f.eks. hvis deres mekaniker ikke længere kan servicere deres mærke, eller hvis en ny rabat er tilgængelig.

Strategy: Mønstret giver os mulighed for at definere forskellige algoritmer og bruge dem udskifteligt. Vi bruger strategy til at håndtere forskellige prisberegninger, f.eks. baseret på scootermærke, mekanikere eller rabatter.

Jeg har brugt ChatGPT for at søge efter kilder der handler om GoF, hvor jeg så fandt bogen nævnt foroven og skrevet ind i vores litteraturliste. Jeg har brugt bogen og dens 3 kapitler om GoF, for at få information om GoF.

GRASP-Mønstre - Karina

GRASP står for General Responsibility Assignment Software Patterns, som er et sæt principper der hjælper med at organisere og fordele ansvar i vores software.

Creator: En klasse skal oprette en ny instans af en anden klasse, hvis den har brug for det. Vi bruger mønstret til at lade en mekaniker oprette nye scootermærker, fordi mekanikeren allerede arbejder med mærker og har ansvaret for dem.

Low coupling: Gør klasser uafhængige af hinanden. Vi har f.eks. sørget for i Scooterland at KundeService og MekanikerService kan fungere hver for sig, selvom de samarbejder. Hvis vi ændrer noget i MekanikerService, påvirker det ikke direkte KundeService.

Controller: Hjælper os med at bestemme, hvilken klasse der skal håndtere brugerens input. I vores projekt bruger vi controllers til at håndtere, når brugeren vil forbinde en kunde til en mekaniker eller tilføje et mærke til en mekaniker. F.eks. har vi en KundeController der tager sig af alle de operationer, der handler om kunder.

High cohesion: Fokuserer på at holde klasserne fokuserede på deres egne opgaver. Vi sørger f.eks. for, at vores MekanikerService kun arbejder med funktioner relateret til mekanikere og ikke håndterer noget med kunder eller mærker.

Jeg har brugt ChatGPT til at finde kilder om GRASP, hvor den så nævnte den bog vi allerede har: Applying UML and patterns, skrevet af Craig Larman, hvor jeg så sprugte ChatGPT på hvilke kapitler jeg kunne finde noget om GRASP, hvor jeg så selv fandt bogen frem og læste de kapitler (kapitel 17 og 18). Kilden er skrevet ind i vores litteraturliste.

Arkitektur - Karina

Frontend: Det er vores brugergrænseflade og er udviklet i Blazor. Brugerne kan interagere med systemet gennem dropdowns og formularer. Vi har f.eks. en side hvor man kan forbinde kunder til mekanikere eller mærker til mekanikere. Vi bruger komponentbaseret design, som gør at vi kan genbruge koden.

Backend: Vi implementerer forretningslogikken og al kommunikation med databasen i vores backend. Vi bruger forskellige services som f.eks. KundeService, til at håndtere alt logikken relateret til kunder, som f.eks. CRUD fra databasen. MekanikerService, til alt mekaniker-relateret logik. Såsom at forbinde mærker og mekanikere. MærkeService, til at håndtere scootermærker og deres relationer til mekanikere.

Database: Vi bruger MSSQL som vores database, hvor alle data om kunder, mærker og mekanikere gemmes. Vi bruger en web API til datatilgang, som er ligesom en bindeled mellem vores services og databasen. API'et sørger for at levere data i JSON-format til Blazor så data kan vises eller opdateres dynamisk. F.eks. når vi henter mærker til dropdown'en, sker det gennem et API kald til api/maerkeapi.

Hosting - Karina

Vi har valgt at lade være med at hoste vores Blazor hjemmeside, API'er og database til det offentlige internet ved brug af Microsoft Azure, men i stedet for valgt bare at aflevere vores API'er, database og Scooterland projekt som zip-filer og .bacpac filer.

Det er dog muligt at hoste vores Blazor, API'er og database ved brug af Microsoft Azure cloud hosting løsning, som er en nem og let tilgængelig måde at få sine projekter ud i virkeligheden.

Brugergrænsefladedesign - Karina

Vi har valgt et simpelt layout, så vores brugere ikke bliver overvældet. Der er kun de nødvendige knapper og felter, så brugeren kan fokusere på opgaven uden at blive overvældet.

Vi har dropdown menuer, som gør at brugeren hurtigt kan vælge mærker, mekanikere og ydelser fra listen. Vi har også knapper med klare handlinger, som f.eks. bekræft booking, forbind, gem og annullér.

På vores forbind mærke til mekaniker side, kan en administrator tildele mærker til mekanikere, og på vores forbind kunde til mekaniker side, får kunder tildelt en fast mekaniker, så de altid ved hvem der reparerer deres scooter.

Vi har brugt farver og beskeder til at give feedback til brugerne, såsom rød ved "fejl" og grøn ved "success".

Systemet er designet til at fungere på computere, og vi har testet grænsefladen på computere, for at sikre, at det fungerer problemfrit.

Hovedmenuen gør det hurtigt at skifte mellem sider som mærker, kunder og mekanikere.

Vi har sikret at alle sider i systemet har samme opbygning og designprincipper. Det gør det nemmere for brugeren at lære systemet at kende.

Processen

Sprint 0

Selvom det ikke er typisk for Scrum at have en Sprint 0, følte vi det vigtigt at have en for at kompilere alt af arbejdet der skal foregå før vi begynder på programmet selv. Derfor lavede vi de diagrammer, som problemstillingen kræver.

Use Case Diagrammer, SSD- og SD-diagram (Amjad Renno)

I Sprint 0 handler det om at skabe et godt fundament for projektet igennem at læse, planlægge og forstå CASE-beskrivelser til ScooterLand opgave .

Det var vigtigt for hjemmesiden at kunne håndtere basale opgaver som processen med at oprette en kunde, bestille en ordre, koble kunden til en mekaniker og til sidst oprette en faktura.

Vi blev enige om at skrive User Stories for at repræsentere de påkrævede funktioner ved at oprette et Googledocs-dokument.

Jeg oprettede en branch på GitHub for at sikre, at jeg arbejder på den, så den ikke påvirker resten af teamet.

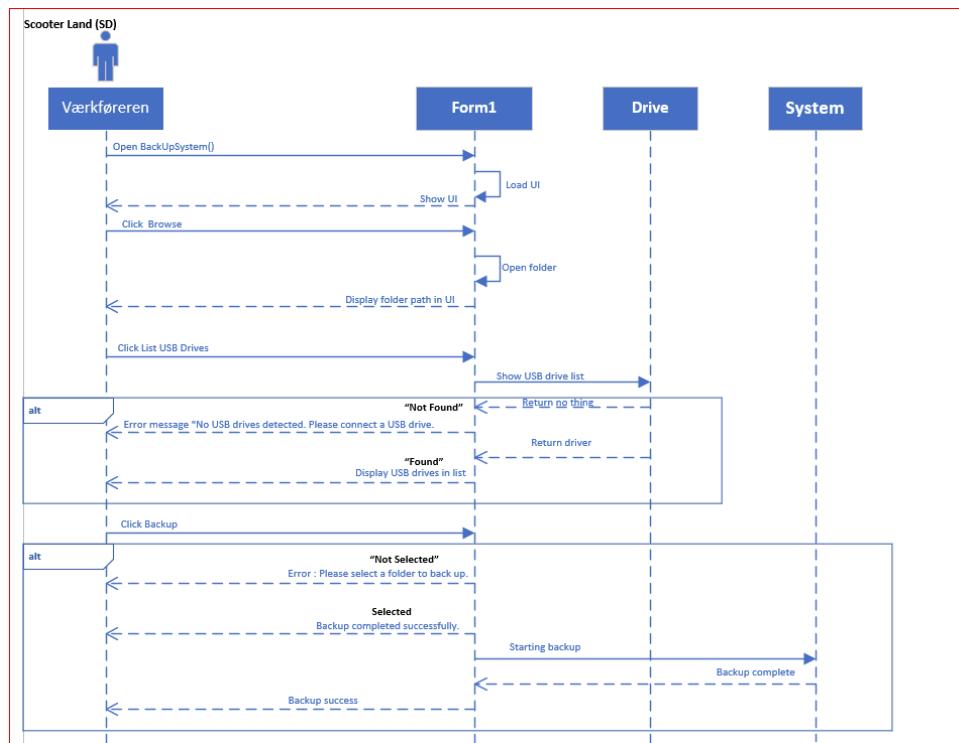
Jeg begyndte at lave Use Case Diagrammer baseret på User Stories.

Og fokus på kontordame og værkfører opgaver.

Jeg lavede et SSD- og SD-diagram: for at illustrere rækkefølgen af grundlæggende operationer og hvordan brugeren vil interagere med systemet.

Disse grafer bidrager til at skabe en stærk struktur for hjemmesiden og hjælper i processen med opdatering og udvikling.

Sekvensdiagram (SD) - Amjad



Beskrivelse:

Et sekvensdiagram er bruges til at visualisere detaljeret, og hvordan komponenter i et system interagerer med hinanden.

I dette diagram, lavede jeg, beskriver det processen funktionalitet ved Backup System til Scooterlandsoplysninger ved at bryge Backup Utility App via usb-drev.

Diagrammet fokuser på hvordan de interne systemprocesser interagerer sammen for at fuldføre backup proces.

1. Hvordan processen fungerer:

- Værkfører klikker på knappen Gennemse for at vælge en mappe, som der skal laves en sikkerhedskopi af.
- Brugeren klikker på "Liste over USB-drev" for at få vist listen over USB-drevet, hvor data gemmes.
- Brugeren vælger drevet og klikker på "Backup".

2. Systemprocedurer:

- Systemet kontrollerer valget af mappe og USB-drev.
- Hvis kontrollerne er opfyldt, starter systemet backup-processen:
- Kopier valgte filer og mapper fra den valgte mappe til et USB-drev.

3. Resultat:

- Systemet fuldfører backup-processen og viser en succesmeddelelse.
- Viser en fejlmeddelelse som f.eks. ikke at vælge en korrekt path.

Analyse og diskussion

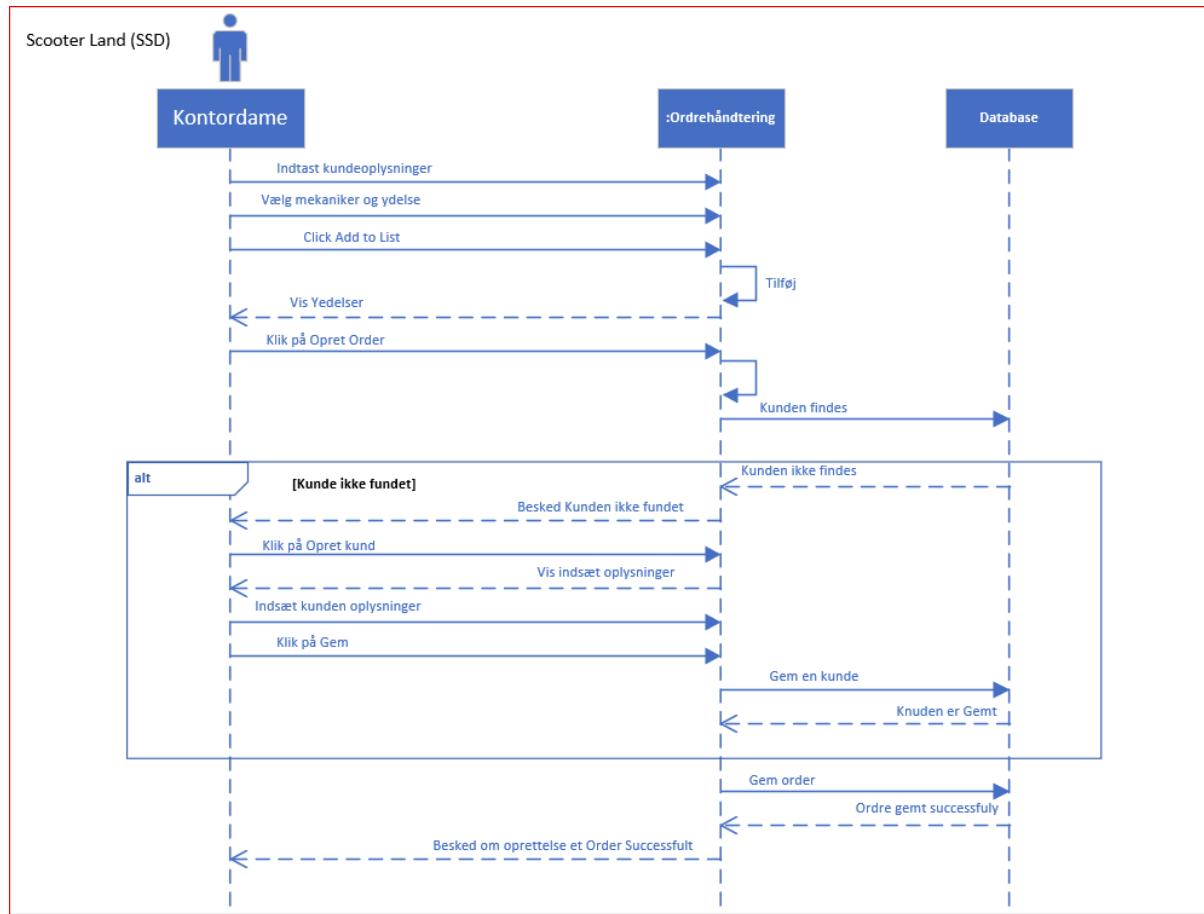
Sekvensdiagrammet giver et detaljeret overblik over processen bag backupfunktionen. Det inkluderer valideringstrin og interaktion mellem komponenter. Diagrammet understøtter en forståelse af, hvordan man laver en sikkerhedskopi problemfrit, samtidig med at man undgår fejl og tydeligt viser den potentielle fejlmeddelelse

konklusion

Endelig, ved at tegne ScooterLand data backup processen ved hjælp af et Sequence Diagram (SD), vises det, at det er en vigtig del af udviklingsprocessen ved at visualisere processen bag backup funktionen.

Fordi det giver en detaljeret forståelse af systemets funktioner indbyrdes. Det hjælper udviklere med at implementere funktioner på en overskuelig og korrekt måde, og ordningen tillader også at tilføje ændringer senere med lethed, håndtere fejl og sikre problemfri udvikling og vedligeholdelse af backupsystemet.

Systemsekvensdiagram (SSD) - Amjad



Introduktion til SSD

System Sequence Diagram (SSD) er en væsentlig del af ScooterLands systemudviklingsproces, da det giver et overblik over, hvordan brugeren vil interagere med systemet.

I dette eksempel har jeg lavet det viser, hvordan kontordamen reagerer på at håndtere ordrer Helt fra tilføjelse af en anmodning til tilføjelse af en service og oprettelse af en kunde, hvis den ikke eksisterer

Derfor er det vigtigt at bruge et systemsekvensdiagram til at visualisere brugsscenariet mellem brugeren og systemet, identificere vigtige funktioner og visualisere forventede fejl under processen med at oprette en anmodning.

Opret ordrer:

Diagrammet viser processen for, hvordan en bruger interagerer med systemet for at oprette en ordre. Dette omfatter validering af kundeoplysninger, valg af tjenester og oprettelse af en ordre.

1. Procedure:

- Brugeren åbner siden "Opret ordrer" via hjemmesiden.
- Han indtaster oplysninger som kundens navn, telefonnummer, mekanikerens navn og den nødvendige mængde og tilføjer dem til listen, så han kan tilføje yderligere ordrer.

2. Systemprocedurer:

- Bekræft, at kunden faktisk findes i databasen:
- Hvis kunden ikke er registreret i databasen, vises brugeren i grænsefladen med mulighed for at oprette en ny kunde og registrere kundens oplysninger.
- Efter at have registreret kunden eller sikret sig, at han eksisterer, vælger brugeren tjenesten og tilføjer den.
- Systemet gemmer anmodningen og giver bekræftelse.

3. Resultat:

Enten oprettes kunden og ordren, eller også får brugeren besked om eventuelle fejl såsom en fejl ved oprettelse af en kunde.

Analyse og diskussion:

SSD'en demonstrerer, hvordan brugeren og systemet kommunikerer i forbindelse med brugssagen "Opret ordrer". Det fokuserer på de grundlæggende funktioner i systemet, såsom at verificere kundedata for at være korrekte og håndtere ordrer.

konklusion:

Der er ingen tvivl om, at System Sequence Diagram (SSD) er en vigtig del af forståelsen af, hvordan brugeren interagerer med systemet ved at tydeliggøre de grundlæggende funktioner i processen med at oprette en ordre, tilføje en service og oprette en ny kunde, og forskellige

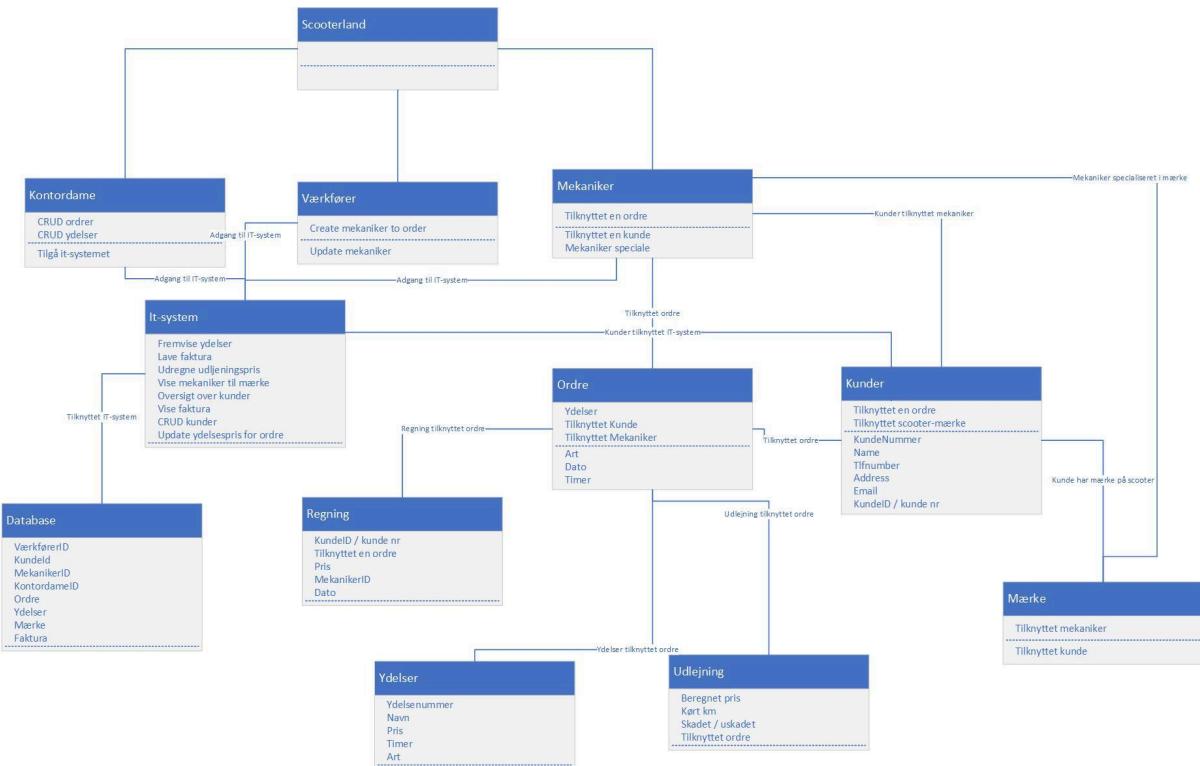
funktioner, som kontordamen interagerer med, foreslås også mulige fejl, såsom fravær af en kunde i databasen.

Risikoanalyse og rapportopsætning (Thomas)

Vi vurderede, at identifikation af diverse risici skulle gøres hurtigst muligt. Derfor lavede jeg en risikoanalyse for at give os et overblik over konsekvenserne ved projektet, samt hvordan vi kan forhindre dem. Jeg gjorde brug af Henrik Steen Krogh's skabelon til risikoanalyser. Jeg lavede opsætningen af vores indholdsfortegnelse i Sprint 0, da dette ville give os en ide om omfanget af rapporten, samt dens struktur.

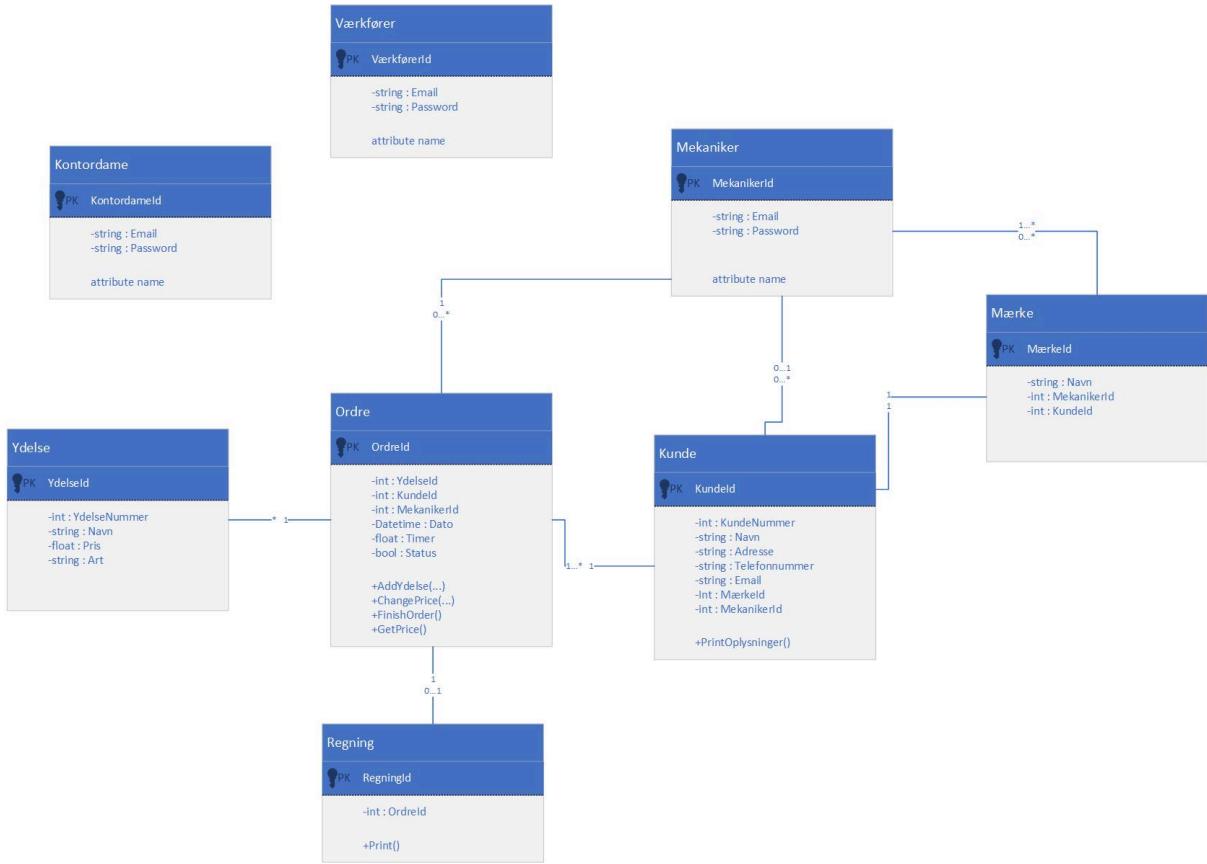
Domænemodel (Karina)

It-systemet viser ydelser, laver fakturaer, beregner priser, viser oversigter over kunder og mekanikere og håndterer data som kunder og ordrer. Kunder til ordrer relationen viser at kunder kan bestille ordrer med flere ydelser. Mekaniker til ordrer relationen viser at hver ordre er en mekaniker tilknyttet. Mærker til mekaniker og kunder relationerne viser at mekanikerne er specialiseret i et mærke, og kunderne kan have scooterer med et bestemt mærke. Udlejning er knyttet til ordre og ydelser, da man kan tilføje ydelsen til at leje en scooter til sin ordre. Regning (faktura) er knyttet til ordre og indsamler kundernes data og ordrer. Kontordamen står for at oprette og opdatere ordrer og ydelser. Værkføreren står for at tildele mekanikere til ordrer og opdatere deres oplysninger.



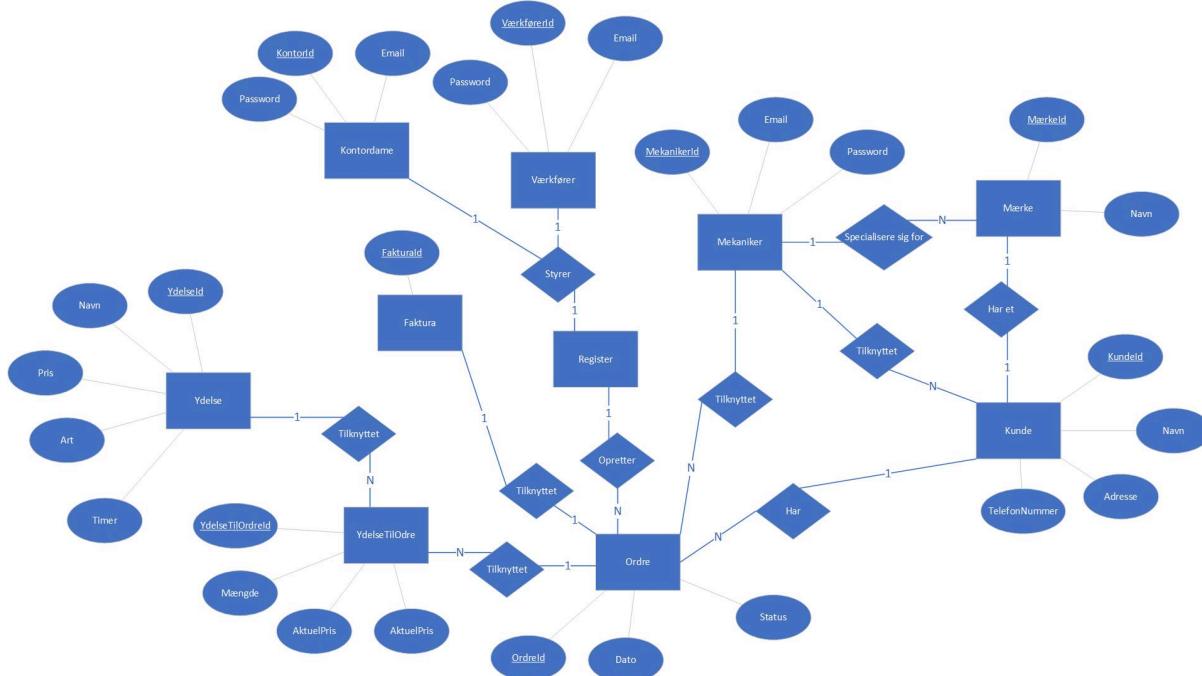
Første iteration af Klassediagrammet (Rene)

Før vi begyndte vores sprint, lavede jeg den første iteration af vores klassediagram. Den vil give os et overblik over de basisklasser, som vi skal oprette i vores program, og de metoder der er indkapslet i dem. Dette er kun det første pass for diagrammet, og jeg vil derfor iterere på den, efter vi ved præcist hvilke klasser der er i vores program.



E/R Diagram (Rene)

For at modellere forskellige entiteter og deres relationer, samt hvilken attributter de består konkret af, opretter jeg et E/R diagram. E/R diagrammer vil også hjælpe mig når jeg skal opsætte databasen, ved at konkret visualisere klasserne og deres attributter som databasen vil bestå af.



Sprint 1

I første sprint fokuserede vi på bygningen af den første iteration af vores program. Med det fokuserede vi på at skabe et minimum levedygtigt produkt. Sprint 1 fokuserede så på de kerne metoder som resten af programmet derefter bygges op på.

Database og Entity Framework (Rene)

For at gøre det muligt at udføre CRUD operationer gennem vores hjemmeside, har vi gjort brug af Entity Framework. Entity framework skaber nemlig en sammenhæng mellem programmet og en database. Og gennem en controller, kan en bruger få adgang til databasen. Vi brugte også repositories til at indkapsle vores metoder, samt services for at få adgang til de metoder.

Ud fra ER diagrammet opretter jeg alle de klasser i en model mappe på shared, da både serveren og klienten skal kunne få adgang til dem.

Ved brug af DbSet, kan jeg få bestemt hvilke af de klasser jeg vil have oprettet i databasen som tabeller. Efter en brug af add-migration og update-database, vil min ScooterlandDB nu have de tabeller

Nu hvor jeg har oprettet databasen, vil jeg oprette rammerne for at få adgang til den. Vi har derfor brug for at oprette controllers, services og repositories, for klasserne: "Ordre",

“Ydelse”, “Mærke”, “Mekaniker” og “Kunde”, for det er klasserne brugeren skal kunne udøve Crud operationer på.

Ordre side for mekaniker (Thomas)

For at muliggøre funktionalitet for mekanikerne, fokuserede jeg på at lave en side, som viser ordrer tilknyttet til en bestemt mekaniker. I starten af sprinten, oplevede vi problemer med implementeringen, da der var en fejl i opsætningen af vores services, controllers og repositories. Vi lavede fejlsøgning som team, og optimerede opsætningen, hvilket gjorde implementeringen af siden "MekanikerOrdre" succesfuld. Dette skabte essentiel funktionalitet for mekanikerne, så de kunne se hvilke ordrer de er tilknyttet.

Faktura Page (Amjad)

Jeg begyndte at arbejde på at introducere de grundlæggende funktioner på fakturasiden, såsom oprettelse, sletning og søgning efter en faktura.

Opret sider

IFakturaService.cs, FakturaService.cs, FakturaController.cs, IFakturaRepository, IFakturaRepository.cs

Jeg testede forbindelsen ved hjælp af Postman for at sikre, at grundlæggende funktioner fungerer og gemmes i databasen.

Postman gør det nemt at sende http-kommandoer til webstedets API, såsom GET, POST, PUT, DELETE-kommandoer. Det gør det nemt at oprette CRUD-operationer.

```

1 {
2   "ordredate": "2024-11-22T00:00:00",
3   "status": false,
4   "ydelsesListId": 1,
5   "kundeId": 1,
6   "mekanikerId": 1
7 }
  
```

Body Cookies Headers Test Results
200 OK - 156 ms - 92 B | Save Response

Opret kunde (Mathias)

Der skal laves en side hvor man kan oprette en kunde med navn, telefonnummer og adresse. En kunde skal også have et kunde-ID.

```

<EditForm Model="newKunde" OnValidSubmit="HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />
    <div>
        <label for="KundeName">Navn:</label>
        <InputText id="KundeName" @bind-Value="newKunde.Navn" class="form-control" />
    </div>
    <div>
        <label for="KundeAddress">Adresse:</label>
        <InputText id="KundeAddress" @bind-Value="newKunde.Adresse" class="form-control" />
    </div>
    <div>
        <label for="KundePhone">Telefonnummer:</label>
        <InputNumber id="KundePhone" @bind-Value="newKunde.TelefonNummer" class="form-control" />
    </div>
    <button type="submit" class="btn btn-primary mt-2">Gem Kunde</button>
</EditForm>

```

Først bliver der lavet selv hvad man ser, siden hvor man kan indtaste navn, adresse og telefonnummer med en “gem” knap. Det bliver gjort via en EditForm.

```

@if (kunder.Any())
{
    <h4>Gemte Kunder:</h4>
    <ul>
        @foreach (var kunde in kunder)
        {
            <li>@kunde.KundeId - @kunde.Navn - @kunde.Adresse - @kunde.TelefonNummer</li>
        }
    </ul>
}

```

hver kunde bliver gemt med deres Id, Navn, Adresse, TelefonNummer

```

@code {
    private Kunde newKunde = new Kunde();
    private List<Kunde> kunder = new List<Kunde>();

    [Inject]
    private IKundeService KundeService { get; set; }

    int ErrorCode { get; set; } = 0;

    protected override async Task OnInitializedAsync()
    {
        kunder = (await KundeService.GetAllKunder()).ToList();
    }
}

```

```

private async void HandleValidSubmit()
{
    ErrorCode = await KundeService.AddKunde(newKunde);
    if (ErrorCode == (int) HttpStatusCode.OK)
    {
        kunder = await KundeService.GetAllKunder().ToListAsync();
    }
    newKunde = new Kunde();
}
}

```

kunder bliver tilføjet til en liste. servicen IKundeService bliver brugt så der kan bliver tilføjet og få information om kunden lettere.



Opret ordre - Første iteration (Karina)

Jeg har lavet en Ydelser side på Blazor med dropdown-menu til de ydelser man kan vælge mellem og formular til at indtaste navn,

```

@if (ydelseList.Count == 0)
{
    <p><em>Loading...</em></p>
    // Viser en loading skærm indtil dataen er blevet hentet
}
else
{
    <div class="form-group">
        <div class="list-group-item d-flex flex-column">
            <label class="form-label">Kunde Navn:</label>
            <input type="text" class="form-control" @bind="selectedKundeNavn" placeholder="Indtast navn på kunden" required />
            <label class="form-label">Telefon Nummer:</label>
            <input type="text" class="form-control" @bind-value="selectedKundeTelefon" placeholder="Indtast telefonnr på kunden"
                @bind-value:event="onchange" @bind-value:after="FindAssignedMekaniker" required />
        </div>
    </div>

    <div class="list-group-item d-flex flex-column">
        <label class="form-label">Vælg en Ydelse:</label>
        <select class="form-control" name="ydelse" id="ydelse" @bind-value="selectedYdelseId" @bind-value:event="onchange"
            @bind-value:after="() => ShowPopup(false)" placeholder="Vælg en Ydelse" required>
            @foreach (var ydelse in ydelseList)
            {
                <option value="@ydelse.YdelseId">
                    @ydelse.Navn (
                    @if (aktueltPris == 0)
                    {
                        @ydelse.Pris
                    }
                    else
                    {
                        @aktueltPris
                    } dkk
                </option>
            }
        </select>
        <div>
            <button class="btn btn-primary btn-sm" @onclick="() => ShowPopup(true)">ændre pris</button>
            <button class="btn btn-primary btn-sm" @onclick="ResetAktuelPris">Nulstil pris</button>
        </div>
    </div>
}

```

samt en prisberegner til at beregne total-prisen på den ydelse man har valgt,

```

        <div class="align-content-lg-end">
            <label class="form-label">Mængde:</label>
            <input type="number" class="form-control" @bind="selectedMængde" min="1" />
        </div>
        <button class="btn btn-primary" @onclick="AddYdelse">Add to List</button>
    </div>

    <h4>Ydelse i Ordrer:</h4>
    @if (ordrerYdelser.Any()) // Viser ydelserne i ordrerne hvis der er nogen
    {
        <ul class="list-group">
            @foreach (var ydelse in ordrerYdelser)
            {
                <li class="list-group-item d-flex justify-content-between align-items-center">
                    @ydelse.YdelseNavn - Quantity: @ydelse.YdelseMængde - Pris: @ydelse.YdelsePris
                    <button class="btn btn-sm btn-danger mt-2" @onclick="() => RemoveYdelse(ydelse)">Fjern</button>
                </li>
            }
        </ul>
        <h4>
            Pris i alt = @totalPrice dkk.
        </h4>
    }

```

og lavet en DateTime.now for at vælge en dato for den ydelse man gerne vil booke, ud fra dags dato.

og så tilføjet en “Book” knap-funktion som gør at ordenen tilføjes faktura, som så blev ændret til “Opret Ordre”, og man kan se ordenen inde på Ordre siden.

```

try
{
    var addedOrdre = new Ordre
    {
        KundeId = valgteKundeId,
        MekanikerId = mekanikerList.Single(m => m.MekanikerId == selectedMekaniker).MekanikerId,
        OrdreDate = DateTime.Now,
        YdelseTilOrdre = ordrerYdelser.Select(p => new YdelseTilOrdre
        {
            YdelseId = p.YdelseId,
            Mængde = p.YdelseMængde,
            AktuelPris = p.YdelsePris
        }).ToList()
    };
    var response = await OrdreService.AddOrdre(addedOrdre);

    if (response == (int) HttpStatusCode.OK)
    {
        successMessage = "Ordren blev oprettet!";
        ordrerYdelser.Clear();
        selectedKundeNavn = "";
        selectedKundeTelefon = "";
        totalPrice = 0; // Resetter variablerne
    }
}
else
{
    errorMessage = "Ordren oprettelsen mislykkedes.";
}
}
catch (Exception ex)
{
    errorMessage = "Ordren indsendelsen mislykkedes: " + ex.Message;
}
}

```

Dataene ville så blive gemt i databasen, altså kunderne og ordren.

Jeg startede med at benytte mig af Radzen, som er ligesom Bootstrap. Det lykkedes til at starte med, indtil jeg så valgte at undgå at bruge Radzen alligevel da de andre ikke kunne finde ud af det da vi kodede videre til sprint 2, 3... osv.

Efter et par dage inde i sprint 1 valgte jeg så også at gøre så man kan vælge en mekaniker til ordren. Det gjorde jeg også ved hjælp af en dropdown menu.

```

<div class="list-group-item d-flex flex-column">
    <label class="form-label">Mekaniker:</label>
    <select class="form-control" @bind="selectedMekaniker">
        @foreach (var mekaniker in mekanikerList)
        {
            <option value="@mekaniker.MekanikerId"> @mekaniker.Navn </option>
        }
    </select>
</div>

```

For at det hele ville virke, injectede jeg IOrdreService, IYdelseService, IMekanikerService og IKundeService.

```

@code {
    [Inject]
    private IOrdreService OrdreService { get; set; }
    [Inject]
    private IYdelseService YdelseService { get; set; }
    [Inject]
    private IKundeService KundeService { get; set; }
    [Inject]
    private IMekanikerService MekanikerService { get; set; }

    private string selectedKundeNavn = "";
    private string selectedKundeTelefon = "";
    private int selectedYdelseId = 1;
    private int selectedMængde = 1;
    private int selectedMekaniker = 1;

    private List<Ydelse> ydelseList = new();
    private List<Kunde> kundeList = new();
    private List<Mekaniker> mekanikerList = new();
    private List<OrdrerYdelse> ordrerYdelser = new();

    private string successMessage;
    private string errorMessage;

    private double aktuelPris = 0;
    private double totalPrice = 0;

    private bool isKundePopupVisible, isOpretKundeVisible = false;
    private Kunde newKunde = new Kunde();
    private DateTime selectedDate = DateTime.Now;
}

```

Så hentede vi alle data fra databasen ind i vores program, og hvis det ikke var muligt så ville fejlmeldelsen ”kunne ikke loade fra databasen” blive læst.

```

protected override async Task OnInitializedAsync()
{
    try
    {
        ydelseList = (await YdelseService.GetAllYdelser()).ToList();
        kundeList = (await KundeService.GetAllKunder()).ToList();
        mekanikerList = (await MekanikerService.GetAllMekaniker()).ToList();
    }
    catch (Exception ex)
    {
        errorMessage = "Kunne ikke loade fra databasen: " + ex.Message;
    }
}

```

Sprint 2

I Sprint 2 fortsatte vi med implementeringen af kernefunktionerne.

Ordre Oversigt (Rene)

For at værkføreren kan have kontrol og overblik over alle de ordrer virksomheden har oprettet, har de brug for en "Ordre Oversigt" page. Siden skal derfor visse alle ordrer i databasen, så jeg opretter en liste af ordrer og ved initialiseringen, bruger OrdreServicen til at sætte listen lig med alle ordrerne i databasen. Med en table og en foreach loop, kan jeg nemt skabe et overblik over alle ordrer i databasen, med hvilken kunde, mekaniker og hvilke ydelser der er tilknyttet hvert ordre. For at vise mekanikeren og kunderne, henter jeg også dem alle sammen ned til en liste fra databasen. Derfra bruger jeg et lambda udtryk til at vise navnet på mekanikeren/kunde som har den matchende id med orden.

```
<td>
    @if (@ordre.KundeId != null) //Viser kun kunden hvis der er en kunde tilknyttet
    {
        @kundeList.Single(k => k.KundeId == ordre.KundeId).Navn
    }
</td>
<td>
    @if (@ordre.MekanikerId != null) //Viser kun mekanikernavn hvis der er en mekaniker tilknyttet
    {
        @mekanikerList.Single(x => x.MekanikerId == ordre.MekanikerId).Navn
    }
</td>
```

Da jeg også vil vise alle ydelser som ordren består af, har jeg brug for at modificere vores OrdreRepository, så når jeg henter data'en for ordrerne, henter jeg også information om de ydelser der er tilknyttet. Jeg inkluderer YdelseTilOrdre, da den er forbindelsen mellem ordre og ydelser, og Ydelser.

```
2 references
public List<Ordre> GetAllOrdre()
{
    return db.Ordrer.Include(o => o.YdelseTilOrdre).ThenInclude(y => y.Ydelse).ToList();
}
```

Men hvis jeg bare gør det, løber jeg ind i et problem. Siden at Ydelse klassen også vil have en reference til Ordre klassen, som dermed har en reference til Ydelse klassen, vil det skabe en evig cyklus af data hentning. Derfor laver jeg en reference handler i program.cs der sørger for at cyklus som denne vil blive ignoreret.

```
builder.Services.AddControllers().AddJsonOptions(x =>
    x.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles);
```

Hvis det er nødvendigt, skal mekanikeren for en ordre også kunne ændres, så hvis f.eks, hvis de er blevet syge og en anden mekaniker bliver nødt til at erstatte dem. Det gøres simpelt med den samme Mekaniker-Service jeg brugte før. Jeg opretter en metode der tager imod to variabler, ordren der skal ændres og det mekaniker id som den skal ændres til. Derfra sætter jeg ordrens mekaniker id og bruger update metoden fra min service til at ændre databasen.

Forbindelser mellem mekaniker og mærke (Thomas)

Jeg lavede mere funktionalitet for mekanikerne, ved at muliggøre forbindelser mellem mekanikere og mærker. Denne forbindelse er vigtig for programmet, da den gør, at det viser hvilke mærker som mekanikerne specialiserer sig inden for. Mekanikere kan derfor blive tildelt de rette ordrer efter deres kunnen.

Fokus blev derefter rapporten, skrive om vores indledning, samt at en analyse af vores risikomatrix, for at bedre forstå hvordan vi kan forhindre risici.

Ydelse Oversigt (Rene)

I samme stil som Ordre oversigten, skabte jeg også en “Ydelse Oversig” page.

Ligesom med ordrerne, skaber både overblik over hvilke ydelser virksomheden, men den lader også værkføreren ændre på deres ydelseskatalog. Enten ved at opdatere en eksisterende ydelse, eller ved at oprette en ny ydelse.

Jeg tilføjede derfor en opret ydelse knappe, plus en redigere og slet knap for hver ydelse.

For oprettelse og redigering gør jeg brug af den samme forms, hvor en bool værdi “isEditMode”, holder styr på hvilken metode der skal bruges. Hvis opret knappen trykkes sættes den til false og hvis redigere knappen trykkes sættes den til true.

```
private void ShowPopup(Ydelse ydelse, bool isEdit)
{
    selectedYdelse = ydelse;
    isEditMode = isEdit;
    modalTitle = isEdit ? "Redigere Ydelse" : "Tilføj Ydelse";
    modalButtonText = isEdit ? "Gem Ändringer" : "Gem";
    isPopupVisible = true;
}
```

Derfra vises der et pop up, hvis titel og tekst afhænger også af isEditMode variablen. Efter brugeren enten indtaster eller redigerer information for ydelsen og trykker submit, vil variablen bestemme om add eller update metoden fra Ydelse repositorien bruges.

```

private async Task HandleFormSubmit()
{
    if (isEditMode)
    {
        // Update ydelsen hvis isEditMode er true
        ErrorCode = await Service.UpdateYdelse(selectedYdelse);
        if (ErrorCode == (int) HttpStatusCode.OK)
        {
            ydelseList = (await Service.GetAllYdelser()).ToList();
        }
    }
    else
    {
        // Create ydelsen hvis isEditMode er falsk
        ErrorCode = await Service.AddYdelse(selectedYdelse);
        if (ErrorCode == (int) HttpStatusCode.OK)
        {
            ydelseList = (await Service.GetAllYdelser()).ToList();
        }
    }
    ClosePopup();
}

```

Faktura formatering (Amjad Renno)

Flere detaljer ind.

Jeg koblede fakturaer til ordrer, så der automatisk oprettes en faktura, når ordrenummeret indtastes.

Jeg tror, at denne metode er nemmere for kontordamen at oprette en faktura automatisk i stedet for at skrive den manuelt, så når kontordamen opretter en ordre, vises alle detaljer ud fra kundens ønske, hvilket sparer tid for kontordamen. .

Jeg tilføjede funktionen til at vise ordre detaljer såsom kundens navn, ordrenummer, mekanikerens navn og dato og Faktura Status.

Tilføjelse af en funktion for at se status for ordren på fakturaen, så brugeren kan skelne ordren som komplet eller ufuldstændig via Tilføj farver Faktura Status:

Grøn for "fuldført" status.

| Faktura Detaljer | |
|------------------|-----------------|
| Faktura ID: | 12 |
| Ordre ID: | 8 |
| Kunde Navn: | Jack Kunde |
| Mekaniker: | Jane Smith |
| Tilføjelsesdato: | 01/01/0001 |
| Status: | Fuldført |

| Ydeler i Ordren | | | |
|-----------------|------------|--------|------------------|
| Navn | Pris (dkk) | Mængde | Total Pris (dkk) |
| Lej Scooter | 500 | 2 | 1000 |

Luk Detaljer

Rød for "Ikke fuldført"-status.

| Faktura Detaljer | |
|------------------|----------------------|
| Faktura ID: | 14 |
| Ordre ID: | 10 |
| Kunde Navn: | Jack Kunde |
| Mekaniker: | Jane Smith |
| Tilføjelsesdato: | 01/01/0001 |
| Status: | Ikke fuldført |

| Ydeler i Ordren | | | |
|-----------------|------------|--------|------------------|
| Navn | Pris (dkk) | Mængde | Total Pris (dkk) |
| Lej Scooter | 500 | 1 | 500 |

Sprint 3

I Sprint 3 afsluttede vi med implementeringen af kernefunktioner.

Opret Ordre - Anden iteration (Rene)

Ligesom med ydelserne, skal programmet kunne oprette en ordre. Men i modsætning til oprettelsen af en ydelse, bliver ordre oprettelsen kompliceret af at det skal være muligt at ændre prisen på en ordre og have en ordre bestå af flere af den samme ydelse.

Det er her hvor jeg vil bruge to klasser, YdelseTilOrdre og OrdrerYdelse. YdelseTilOrdre vil indeholde alt den information om en ydelse der vil være specifikt for orden. Altså mængden af ydelsen der er i orden og dens aktuelle pris, hvis prisen skal være anderledes end ydelsens standard. OrdrerYdelse klassen vil holde information om både ydelsen selv og de ændringer som brugeren har foretaget på den. Den gør jeg brug af for at gøre indsættelsen af orden i databasen nemmere.

Først indtaster brugeren navnet og telefonnummeret på kunden. Så vælger de mekanikeren, hvis kunden har en tilknyttet mekaniker, vil ordrens mekaniker automatisk sættes til kundens mekaniker. På den måde kan kunden få deres foretrækkelige mekaniker. Til dette mål bruger jeg en bindvalue:event, der aktiveres når telefonnummer feltet ændres.

```
@bind-value="selectedKundeTelefon" placeholder="Indtast telefonnr på kunden" @bind-value:event="onchange" @bind-value:after="FindAssignedMekaniker" required />
```

Den først søger om der er en kunde i listen af kunder i databasen med et matchende telefonnummer, og hvis der er en, sætter den valgte mekaniker til kundens tilknyttede mekaniker.

```
private void FindAssignedMekaniker() // Sætter mekanikeren til at være den mekaniker der er tilknyttet kunden, hvis der er en mekaniker tilknyttet dem.
{
    if (kundeList.Any(k => k.TelponNummer == Int32.Parse(selectedKundeTelefon)))
    {
        Kunde tempKunde = kundeList.Single(k => k.TelponNummer == Int32.Parse(selectedKundeTelefon));
        selectedMekaniker = mekanikerList.Single(m => m.MekanikerId == tempKunde.MekanikerId).MekanikerId;
    }
}
```

Når en bruger tilføjer en ydelse til ordren, vil den valgte ydelser derfor tilføjes til en liste af ordrerYdelser.

```
var ydelse = ydelseList.FirstOrDefault(p => p.YdelseId == selectedYdelseId);
if (ydelse != null)
{
    if (aktuelpriis == 0)
    {
        ordrerYdelser.Add(new OrdrerYdelse
        {
            YdelseId = ydelse.YdelseId,
            YdelseNavn = ydelse.Navn,
            YdelsePris = ydelse.Pris,
            YdelseArt = ydelse.Art,
            YdelseTimer = ydelse.Timer,
            YdelseMængde = selectedMængde,
        });
        totalPrice += ydelse.Pris * selectedMængde;
    }
}
```

For at gøre det muligt at ændre prisen opretter jeg en double variable, “aktuelpriis”, og sætter den til at være lig med 0. Hvis brugeren trykker på knappen “Ny Pris”, vil der vises et pop up, hvor den ønskede pris kan indtastes, som aktuel pris variablen bliver sat til at være lig med.

Når en kunde så vælger at tilføje en ydelse, tjekker programmet om aktuel pris er lig med 0. Hvis den er lig med 0, vil den oprettede OrdrerYdelses ydelsePris, bare være sat til at være ydelsens standardværdi. Derimod, hvis variablen er noget andet, sættes OrdrerYdelsens ydelsePris til at være variablens værdi.

```

if (aktuelpriis == 0)
{
    ordrerYdelser.Add(new OrdrerYdelse
    {
        YdelseId = ydelse.YdelseId,
        YdelseNavn = ydelse.Navn,
        YdelsePris = ydelse.Pris,
        YdelseArt = ydelse.Art,
        YdelseTimer = ydelse.Timer,
        YdelseMængde = selectedMængde,
    });
    totalPrice += ydelse.Pris * selectedMængde;
}
else // Kører hvis prisen på ydelsen skal være noget andet end standardprisen
{
    ordrerYdelser.Add(new OrdrerYdelse
    {
        YdelseId = ydelse.YdelseId,
        YdelseNavn = ydelse.Navn,
        YdelsePris = aktuelPris,
        YdelseArt = ydelse.Art,
        YdelseTimer = ydelse.Timer,
        YdelseMængde = selectedMængde,
    });
}

```

Når brugeren trykker på “Opret ordre” knappen, tjekker den først om de indtastede kundeinformation matcher en kunde i databasen. Fordi hvis der ikke er en matchende kunde i systemet, skal brugeren nemlig have muligheden for at nu oprette dem i databasen.

```

private async Task SubmitOrder() // Tilføjer ordrer til databasen
{
    int valgteKundeId = 0;
    try // Tjekker om den indtastede kunde er i systemet.
    {
        valgteKundeId = kundeList.Single(k => k.Navn == selectedKundeNavn && k.TlfNummer == int.Parse(selectedKundeTlf));
    }
    catch
    {
        ShowPopupKundeNotFound(); // Viser et popup hvis kunden ikke kan findes
        return;
    }
}

```

I pop up'en kan de så enten indtaste den manglende information for kunden, eller genindtastes deres information, hvis de nu har lavet en stavfejl.

Efter at programmet enten finder en matchende kunde, eller opretter den nye kunde, samler programmet liste af ordrerYdelse med den valgte kunde og mekaniker.

```

var addedOrdre = new Ordre
{
    KundeId = valgteKundeId,
    MekanikerId = mekanikerList.Single(m => m.MekanikerId == selectedMekaniker).MekanikerId,
    OrdreDate = DateTime.Now,
    YdelseTilOrdre = ordrerYdelser.Select(p => new YdelseTilOrdre
    {
        YdelseId = p.YdelseId,
        Mængde = p.YdelseMængde,
        AktuelPris = p.YdelsePris
    }).ToList()
};
var response = await OrdreService.AddOrdre(addedOrdre);

```

Ved brug af en Select() metode, omdannes ordrerYdelserne listen til en YdelseTilOrdre liste som så tilknyttes ordren. Derefter kan ordren nu gemmes i databasen.

Lejning af Scooter (Rene)

Udlejning af scooter er et specielt tilfælde når det kommer til ydelser. Fordi i modsætning til de andre ydelser, vil den opkrævede pris være afhængig af flere variabler. Hermed prisen per dag, dage lejet, kilometer kørt og om der foregået en skade på scooteren. For at håndtere dette opretter jeg en klasse for lejet scooter som holder en reference til den tilsvarende ydelse. I dette tilfælde har jeg sat ydelse id'en for lejet scooter til 13.

Når brugeren så vælger en ydelse, tjekker jeg om ydelsen id er lig med 13

```

name="ydelse" id="ydelse" @bind-value="selectedYdelseId" @bind-value:event="onchange" @bind-value:after="() => ShowPopup(false)"

private void ShowPopup(bool isChange) // Viser et popup for enten Lejning af scooter eller ændring af pris
{
    modalTitle = isChange ? "Ændre Pris på Ydelse" : "Udlejning af Scooter";
    modalButtonText = isChange ? "Ændre Pris" : "Tilføj Ydelse";
    if (selectedYdelseId == lejetScooter.YdelseId) // Viser pop up for lejning af scooter
    {
        isPopupVisible = true;
    }
}

```

Og hvis den er, viser pop up'en der indeholder det forms for indtastning af relevante informationer. Efter brugeren har indtastede information, bliver den nu oprettet som en ordrerYdelse, ligesom med de andre ydelser. Men i stedet for at bruge en standard eller en indtastet pris, gøres der brug af en metode for at regne prisen.

```

ordrerYdelser.Add(new OrdrerYdelse
{
    YdelseId = ydelse.YdelseId,
    YdelseNavn = ydelse.Navn,
    YdelsePris = addedLejetScooter.GetLejeScooterTotalPris(),
    YdelseArt = ydelse.Art,
    YdelseMængde = 1
});

```

Denne metode samler alle de indtastede informationer om den lejet scooter og returner en total pris:

```
2 references
public double GetLejeScooterTotalPris()
{
    if (Skadet)
    {
        totalPris = (LejePris * DageLejet) + Forsikring + (KortKilometer * kmPris) + selvRisiko;
    }
    else
    {
        totalPris = (LejePris * DageLejet) + Forsikring + (KortKilometer * kmPris);
    }
    return totalPris;
}

double totalPris;
double kmPris = 0.53;
double selvRisiko = 1000;
```

På den måde er det muligt at tilføje en ydelse til ordrer med en kalkuleret pris.

BackupSystem og Login (Thomas)

For at gøre det muligt, at lave backups, som en sikkerhedsforanstaltning. Jeg lavede derfor et backupsystem, separat fra Scooterlands systemet. Jeg udformede det i WinForms, med et simpelt UI i fokus. Funktionaliteten af systemet er også simpel, da man kan trykke på en knap 'List USB Drev' for at udfylde en listbox med alle tilsluttede USB Drev. Efter man har valgt et USB Drev, kan man vælge hvilken mappe man vil lave en backup af. Backuppen bliver fuldført ved et tryk på 'Backup'.

Jeg oplevede store problemer med opsætningen af login systemet under sprint 3. Jeg fik lavet en metode som tjekkede om password og e mail var korrekt, dette var fint, men havde ikke en fuld login funktionalitet.

Forbedring af brugeroplevelsen (Amjad Renno)

For at undgå fejl har jeg lavet flere funktioner ved at tilføje og teste knapper feks.

Tilføj knapper for opdatering og sletning af faktura status:

Jeg tilføjede funktionen til at opdatere fakturaen ved at tilføje knappen "Markér som Fuldført".

For at forbedre brugen og undgå fejl, har jeg implementeret mange funktioner ved at tilføje knapper, som hjælper kontordamen med at håndtere fakturaer mere

Jeg tilføjede opdatering af faktura funktionen ved at tilføje en "Marker som fuldført" knap. Når man trykker på denne knap, indikerer status for ordre, at den er komplet i databasen, hvilket gør det nemmere for kontordamen tydeligt at se anmodningen, hvis den er ufuldstændig i brugergrænsefladen.



Knappen Markér som Fuldført forsvinder, hvis ordrestatus er Fuldført



Jeg tilføjede en knap til at slette fakturaen fra databasen med en bekræftelsesmeddelelse om, at den er blevet slettet.

Sprint 4

I sprint 4 ryddede vi op i programmet og udførte de sider og metoder som vi havde misset i de forrige sprint.

Kunde Oversigt (Rene)

Ligesom med ordene og ydelerne, vil jeg oprette en oversigt over kunderne. Til det, skal siden kunne vise alt information om en kunde, plus en liste over alle de ordrer som er tilknyttet dem. Det skal også være muligt at skifte en kundes tilknyttet mærke eller mekaniker.

For at vise historikken, bruger jeg et simpelt lambda udtryk for at finde alle ordrer med det matchende kundeId.

```
private void ShowOrdreHistorik(int kundeId)
{
    valgtKunde = kundeList.Single(k => k.KundeId == kundeId);
    valgtOrdreList = ordreList.Where(o => o.KundeId == valgtKunde.KundeId).ToList();
    showPopupOrdrer = true;
}
```

Så vises den liste af ordrer i et popup.

```
<div class="modal-body">
    <ul>
        @foreach (var ordre in valgtOrdreList)
        {
            <h4> @ordre.OrdreDateString() </h4>
            <p><b>Ydelses i ordren: </b> </p>
            @foreach (var ydelse in ordre.YdelseTilOrdre)
            {
                <li><b>@ydelse.Ydelse.Navn</b><br/></li>
            }
            <br>
            <p>Total Pris for ordre: <b> @CalculatePrice(ordre)</b> dkk</p>
            <p>Total Timer for ordre: <b> @CalculateHours(ordre)</b> timer</p>
        }
    </ul>
```

For at ændre mekaniker og mærke bruger jeg også et pop up. Den giver brugeren et valg af alle mekaniker/mærker i databasen, så de kan vælge hvilken skal nu tilknyttes kunden.

```
private void ShowMærke(int kundeId)
{
    valgtKunde = kundeList.Single(k => k.KundeId == kundeId);
    showPopupMærke = true;
}
```

Derefter bruges en Service til at update kunden i databasen med den nye information.

```
private async Task ChangeMærke()
{
    valgtKunde.MærkeId = selectedMærke;
    var success = await KundeService.UpdateKunde(valgtKunde);
    CheckSuccess(success);
    ClosePopup();
}
```

Og med er det muligt både at ændre og se informationer om individuelle kunder!

Login og CSS + HTML (Thomas)

Jeg prøvede at muliggøre login funktionalitet i Sprint 4, eftersom jeg var mislykkedes med implementeringen i Sprint 3. Jeg løb dog ind i flere problemer med at gøre det nyttigt, da jeg

ikke kunne forstå hvordan jeg skulle gemme sessions eller gøre brug af cookies. Vi valgte derfor at droppe login funktionalitet grundet dets kompleksitet.

Jeg begyndte derefter på det visuelle af vores Scooterland system. Finde et tema til siden, hvor vi valgte et mørke/lyseblå design tema. Jeg rettede også margins, da det visuelt så klaustrofobisk ud.

Slette faktura og tilføje en søgning (Amjad Renno)

Opdateret funktionen med at slette faktura og tilføje en søgning

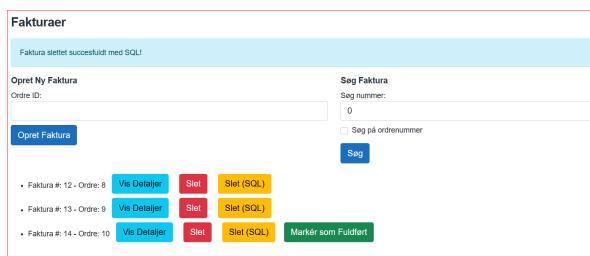
Jeg oprettede en funktion til at slette faktura fra databasen ved hjælp af SQLClient, som fremskynder faktura sletning processen.

```
public bool DeleteFakturaWithSql(int fakturaId)
{
    string connectionString = db.Database.GetDBConnection().ConnectionString;
    bool isDeleted = false;

    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        try
        {
            connection.Open();
            string query = "DELETE FROM Fakturaer WHERE FakturaId = @FakturaId";
            using (SqlCommand command = new SqlCommand(query, connection))
            {
                command.Parameters.AddWithValue("@FakturaId", fakturaId);
                int rowsAffected = command.ExecuteNonQuery();
                isDeleted = rowsAffected > 0;
            }
        }
        catch (SqlException sqlEx)
        {
            Console.WriteLine($"SQL Error: {sqlEx.Message}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }

    return isDeleted;
}
```

Jeg sørgede for at bekræfte sletningsprocessen ved at sende en besked til brugeren med besked om, at fakturaen var slettet



Der er tilføjet en fakturasøgningsfunktion for at gøre det nemmere for brugeren at finde fakturaer baseret på Fakura ID eller Ordre ID.

Ordre ID:

Fakturaer

Fakura fundet

Opret Ny Fakura

Ordre ID:

Søg Fakura

Søg nummer:

Søg på ordrenummer

Søg

- Fakura #: 13 - Ordre: 9 **Vis Detaljer** **Slet** **Slet (SQL)**

Søgefunktionen blev testet ved at sikre, at en besked blev sendt til brugeren i visse tilfælde. Ligesom fakturaen blev fundet.
Eller ingen faktura fundet.

Fakturaer

Der er ingen Order med dette ID. Prøv igen

Opret Ny Fakura

Ordre ID:

Søg Fakura

Søg nummer:

Søg på ordrenummer

Søg

GRASP, Overskrift, UNIT-tests, ForbindMærkeTilMekaniker (Karina)

Først startede jeg med at lave overskriften til vores Blazor hjemmeside, opgaven var at lave en shared header som viser Scooterland navnet, adressen og dags dato, øverst på hver side / tab man går ind på i vores Blazor hjemmeside:

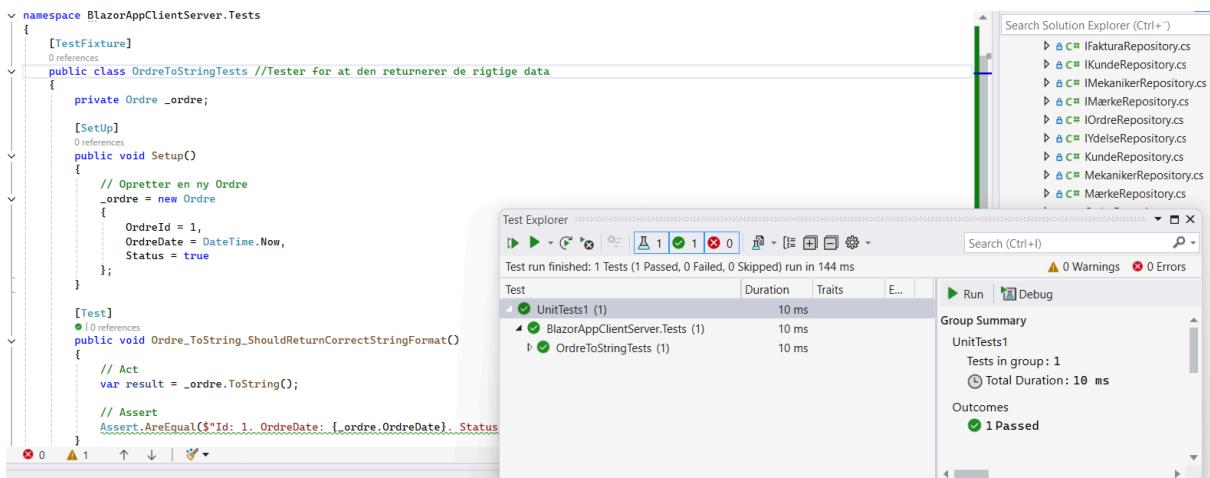
```
<h3>SharedHeader</h3>

@code {
    private string todayDate = DateTime.Now.ToString("dd-MM-yyyy");
}

<div class="text-center mb-4">
    <h2>Scooterland</h2>
    <p>Adresse: Boulevarden 25, 7100 Vejle</p>
    <p>Dags dato: @todayDate</p>
    <hr />
</div>
```

Og så brugte jeg bare <SharedHeader /> øverst på alle razor komponenterne siderne.

Det næste jeg gjorde var så at lave UNIT-tests, jeg valgte at lave dem ud fra vores C# models klasser da vi endnu ikke har lært at bruge mocking til unit tests af HttpClient:



The screenshot shows the Visual Studio interface with the Test Explorer window open. It displays two successful test runs:

- Test run 1:** UnitTests1 (1) - Duration: 10 ms. Contains BlazorAppClientServer.Tests (1) which contains OrdreToStringTests (1). Status: 1 Passed, 0 Failed, 0 Skipped.
- Test run 2:** UnitTests1 (2) - Duration: 9 ms. Contains BlazorAppClientServer.Tests (2) which contains YdelseTests (2). Status: 2 Passed, 0 Failed, 0 Skipped.

Code Snippets:

```

namespace BlazorAppClientServer.Tests
{
    [TestFixture]
    public class OrdreToStringTests //Tester for at den returnerer de rigtige data
    {
        private Ordre _ordre;

        [SetUp]
        public void Setup()
        {
            // Oprettet en ny Ordre
            _ordre = new Ordre
            {
                OrdreId = 1,
                OrdreDate = DateTime.Now,
                Status = true
            };
        }

        [Test]
        public void Ordre_ToString_ShouldReturnCorrectStringFormat()
        {
            // Act
            var result = _ordre.ToString();

            // Assert
            Assert.AreEqual($"Id: {_ordre.Id}, OrdreDate: {_ordre.OrdreDate}, Status: {_ordre.Status}", result);
        }
    }
}

[TestFixture]
public class YdelseTests
{
    private Ydelse _ydelse;

    [SetUp]
    public void Setup()
    {
        // Initialisering af Ydelse objektet
        _ydelse = new Ydelse
        {
            YdelseId = 1,
            Navn = "Reparation",
            Pris = 500.0,
            Art = "Service",
            Timer = 2.5
        };
    }

    // Test af korrekt initialisering af Ydelse objektet
    [Test]
    public void Ydelse_ShouldBeInitializedCorrectly()
    {
        // Act
        var result = _ydelse;

        // Assert
        Assert.NotNull(result);
        Assert.AreEqual(1, result.YdelseId);
        Assert.AreEqual("Reparation", result.Navn);
        Assert.AreEqual(500.0, result.Pris);
        Assert.AreEqual("Service", result.Art);
        Assert.AreEqual(2.5, result.Timer);
    }

    // Test af ToString metoden
    [Test]
    public void Ydelse_ToString_ShouldReturnCorrectString()
    {
        // Act
        var result = _ydelse.ToString();

        // Assert
        Assert.AreEqual("Reparation", result);
    }
}

```

Så opdaterede jeg ForbindMærkeTilMekaniker, så man kan forbinde et scootermærke til en bestemt mekaniker, vha. mærkeID og mekanikerID:

```

@page "/forbind-mærke-mekaniker"
@using BlazorAppClientServer.Client.Services
@using BlazorAppClientServer.Shared.Models
@using BlazorAppClientServer.Client.Shared

@inject IMekanikerService MekanikerService
@inject IMærkeService MærkeService
@inject HttpClient Http

<Header />

<h3>Forbind Mærke til Mekaniker</h3>

<div>
    <label for="mekaniker">Vælg Mekaniker:</label>
    <select id="mekaniker" class="form-control dropdown-indicator" @bind="selectedMekanikerId">
        <option value="">-- Vælg --</option>
        @foreach (var mekaniker in mekanikere)
        {
            <option value="@mekaniker.MekanikerId">@mekaniker.Navn</option>
        }
    </select>
</div>

<div>
    <label for="maerke">Vælg Mærke:</label>
    <select id="maerke" class="form-control dropdown-indicator" @bind="selectedMaerkeId">
        <option value="">-- Vælg --</option>
        @foreach (var mærke in mærker)
        {

            <option value="@mærke.MærkeId">@mærke.Navn</option>
        }
    </select>
</div>

<button class="btn btn-primary mt-2" @onclick="ForbindMaerkeTilMekaniker">Forbind</button>

@if (errorMessage != null)
{
    <p style="color: red;">@errorMessage</p>
}

@if (successMessage != null)
{
    <p style="color: green;">@successMessage</p>
}

@code {
    private List<Mekaniker> mekanikere = new();
    private List<Mærke> mærker = new();

    private int? selectedMekanikerId;
    private int? selectedMaerkeId;
}

```

```

private string? errorMessage;
private string? successMessage;

protected override async Task OnInitializedAsync()
{
    try
    {
        var fetchedMekanikere = (await MekanikerService.GetAllMekaniker()).ToList();
        mekanikere = fetchedMekanikere ?? new List<Mekaniker>(); // Sikrer, at mekanikere aldrig bliver null.

        var fetchedMaerker = (await MærkeService.GetAllMærker()).ToList();
        maerker = fetchedMaerker ?? new List<Mærke>(); // Sikrer, at maerker aldrig bliver null.
    }
    catch (Exception ex)
    {
        // Log fejl (valgfrit) og håndter tomme lister
        errorMessage = "Fejl ved indlæsning af data: " + ex.Message;
        mekanikere = new List<Mekaniker>();
        maerker = new List<Mærke>();
    }
}

private async Task ForbindMaerkeTilMekaniker()
{
    if (selectedMekanikerId == null || selectedMaerkeId == null)
    {
        errorMessage = "Vælg både en mekaniker og et mærke.";
        return;
    }

    var mekaniker = mekanikere.FirstOrDefault(m => m.MekanikerId == selectedMekanikerId);
    if (mekaniker != null)
    {
        // Fjern eksisterende mærker, hvis mekaniker skal have ét nyt mærke
        mekaniker.MærkeListe?.Clear();

        var mærke = maerker.FirstOrDefault(m => m.MærkeId == selectedMaerkeId);
        if (mærke != null)
        {
            mekaniker.MærkeListe ??= new List<Mærke>();
            mekaniker.MærkeListe.Add(mærke);

            var success = await MekanikerService.UpdateMekaniker(mekaniker);

            if (success)
            {
                successMessage = "Mærke forbundet til mekaniker.";
                errorMessage = null;
            }
            else
        }
    }
}

```

Til sidst læste jeg om GRASP i bogen UML and Patterns af Craig Larman, kapitel 17.

Sprint 5

Sprint 5 bestod for det meste af rapportskrivning. Derfor vil dette afsnit være kortere end det andre.

Validation (Rene)

Tak til Blazors framework, er validation for indtastede information meget simpelt at implementere. Når en bruger skal sende data til databasen, gør vi bruger af en validator, som tjekker deres indtastning imod specifikationerne i selve klasserne.

For et eksempel, hvis vi kigger på Ydelse klassen:

```
13 references
public int YdelseId { get; set; }

[Required(ErrorMessage = "Navnfeldet er påkrævet")]
[StringLength(50, MinimumLength = 3, ErrorMessage = "Navnet skal være mellem 3 og 50 tegn")]
19 references
public string Navn { get; set; }

[Required(ErrorMessage = "Prisfeltet er påkrævet")]
[Range(0, int.MaxValue, ErrorMessage = "Prisen skal være et positivt nummer")]
12 references
public double Pris { get; set; }

[Required(ErrorMessage = "Artfeltet er påkrævet")]
[StringLength(50)]
14 references
public string Art { get; set; }
14 references
public double? Timer { get; set; }
```

Her har jeg sat nogle begrænsninger for hvilke indtastninger som programmet vil acceptere. Hvis en bruger prøver at indtaste et navn for ydelsen, der er under 3 karakterer, vil den tilsvarende error besked vises i UI'et.

```
<div class="modal-body">
    <EditForm Model="selectedYdelse" OnValidSubmit="HandleFormSubmit">
        <ValidationSummary />
        <div class="mb-3">
            <label for="navn" class="form-label">Navn</label>
            <InputText @bind-Value="selectedYdelse.Navn" class="form-control" id="navn" />
        </div>
```

Det samme vil ske for pris feltet, hvis de prøver at sende et ikke positivt nummer, og art feltet

Jeg bruger også Required attributten, så brugeren ikke kan lade de tilsvarende felter stå blank. Siden at vi har sat Timer attributten til nullable, vil den ikke have dette attribut. Så det vil være muligt for brugeren at oprette et ydelse objekt med en timer attribut sat til null.

Rapportskrivning og final touch-ups (Karina)

Jeg stod for at skrive noget af rapporten, og satte mig ind i at gøre det på fulldtid. Under sprint 5 har jeg også testet om programmet/projektet kører som det skal, og vi er stødt på nogle problemer ang. Kunde Oversigt, linket til siden er på Blazor hjemmesiden, men vi mangler at putte siden ind og connecte det til linket, det er derfor vi får en fejl når vi prøver at loade siden Kunde Oversigt. Jeg har også sørget for at min database er korrekt, samt prøvet at finde ud af hvordan jeg får databasen i en .bacpac fil, og Blazor Scooterland-projektet i en zip-fil så jeg er klar til at aflevere fredag.

Jeg stødte på en fejl da vi mergede alle vores branches her til sidst, og blev nødt til at rette ForbindMærkeTilMekaniker til, ved at ændre på UpdateMekaniker i vores MekanikerService:

Før:

```
2 references
public async Task<bool> UpdateMekaniker(Mekaniker mekaniker)
{
    var response = await httpClient.PutAsJsonAsync("api/mechanikerapi" + Mekaniker.Id, mekaniker);
    return response.IsSuccessStatusCode;
}
```

Efter:

```
2 references
public async Task<bool> UpdateMekaniker(Mekaniker mekaniker)
{
    var response = await httpClient.PutAsJsonAsync("api/mechanikerapi", mekaniker);
    return response.IsSuccessStatusCode;
}
```

Rense koder (Amjad)

I Sprint 5 fokuserede jeg på at gennemgå og rense koderne og implementere verifikation til OrdreId for at sikre, at brugeren skal indtaste ordrenummeret korrekt, og at der tydeligt kommer en besked til kontordamen, der informerer hende om, at OrdreId skal indtastes.

Fakturaer

Opret Ny Faktura

- Ordred er påkrævet

Ordre ID:

Ordred er påkrævet

Opret Faktura

- Faktura #: 1 - Ordre: 1 Vis Detaljer Slet
- Faktura #: 2 - Ordre: 1 Vis Detaljer Slet

Jeg lavede også nogle forbedringer ved at tilføje en printknap ud for detaljerne på hver faktura, som gør det muligt at vise fakturaoplysningerne foran kunden eller kontordamen.

Faktura #: 2

| Faktura Detaljer | | | |
|------------------|-------------|--|--|
| Faktura ID: | 2 | | |
| Ordre ID: | 1 | | |
| Kunde Navn: | Jack Kundte | | |
| Mekaniker: | Ukendt | | |
| Dato: | 11/12/2024 | | |
| Status: | Fuldført | | |

Ydelser i Ordren

| Navn | Pris (dkk) | Mængde | Total Pris (dkk) |
|-------------|------------|--------|------------------|
| Lej Scooter | 200 | 2 | 400 |
| Lej Scooter | 200 | 2 | 400 |

Luk Detaljer Print

I slutningen af Sprint afsluttede jeg arbejdet med rapporten og dokumenterede, hvad jeg lavede i hvert Sprint, og alt forløb korrekt, så jeg var klar til levering.

Visuelle justeringer og fejlfinding (Thomas)

Vi havde som gruppe næsten færdiggjort al funktionalitet i programmet. Der var kun småting tilbage, UI og fejlrettelser. Jeg brugte tiden på fejlfinding, ved at teste programmet og de forskellige processer. Grundet den valgte arkitektur af systemet, kunne vi nemmere finde fejl, samt lave redigeringer i hinandens kode på tværs af udviklerteamet.

Jeg lavede flere ændringer på CSS og vores sidebar(navigation menu). Dette var for at skabe en mere ensartet stil igennem hele systemet, så de forskellige sider ikke afveg for meget fra hinanden.

Perspektivering

Vi har ikke haft nået alt i vores 5 sprints, såsom Login-systemet i en razor. Men vi lavede dog et Backup System med WinForms, hvor man kan browse igennem Folder Paths, indsætte et USB og eksportere filer/data til USB driven, så vi har en backup af vores Scooterland projekt.

Da nogle af os var syge et par gange, de gange vi mødtes på skolen, har vi måske ikke været så produktive, som vi gerne ville have haft været, hvorfaf 2 af os fra gruppen, har haft arbejde

ved siden af, og andre gange har nogle af os været for dovne eller for stressede til at skrive eller programmere lige den ene dag ud af de 7 dage i ugen vi havde til hvert sprint. Vi kunne derfor nok godt have forbedret vores kommunikation over disse problemer, så resten af gruppen ville have det nemmere med at arbejde rundt om dem.

Konklusion

Alt i alt har vi som gruppe konstrueret et Blazor program der vil kunne fuldføre opbevaring af data samt visning af relevante data. Selve virksomhedens design er meget simpel, men som et internt program for brug af en lille virksomhed, vil det ikke være så vigtigt for den at være visuel imponerende. I stedet har vi fokuseret på at gøre den brugervenlig og simpel.

Litteraturliste

GoF-mønstre, kapitel 3, 4 og 5: [Design-Patterns-Mentorship/Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides-Design Patterns – Elements of Reusable Object-Oriented Software -Addison-Wesley Professional \(1994\).pdf at master · TushaarGVS/Design-Patterns-Mentorship](#)

GoF-mønstre, Geeks for geeks: [Gang of Four \(GOF\) Design Patterns - GeeksforGeeks](#)

GRASP-mønstre, kapitel 17 og 18: [Applying-uml-and-patterns-3rd-edition/Applying UML and Patterns 3rd Edition.pdf at master · AraNaldinho/Applying-uml-and-patterns-3rd-edition](#)

SCRUM: De udleverede powerpoints og artikler vi fik på 1., og 2. semester af Henrik.

HTML og CSS:

<https://www.w3schools.com/html/default.asp>

WindowsForms, tutorial 1, 2 og 3:

<https://learn.microsoft.com/en-us/visualstudio/ide/create-csharp-winform-visual-studio?toc=%2Fvisualstudio%2Fget-started%2Fcsharp%2Ftoc.json&bc=%2Fvisualstudio%2Fget-started%2Fcsharp%2Fbreadcrumb%2Ftoc.json&view=vs-2022>

Systemsekvensdiagram (SSD):

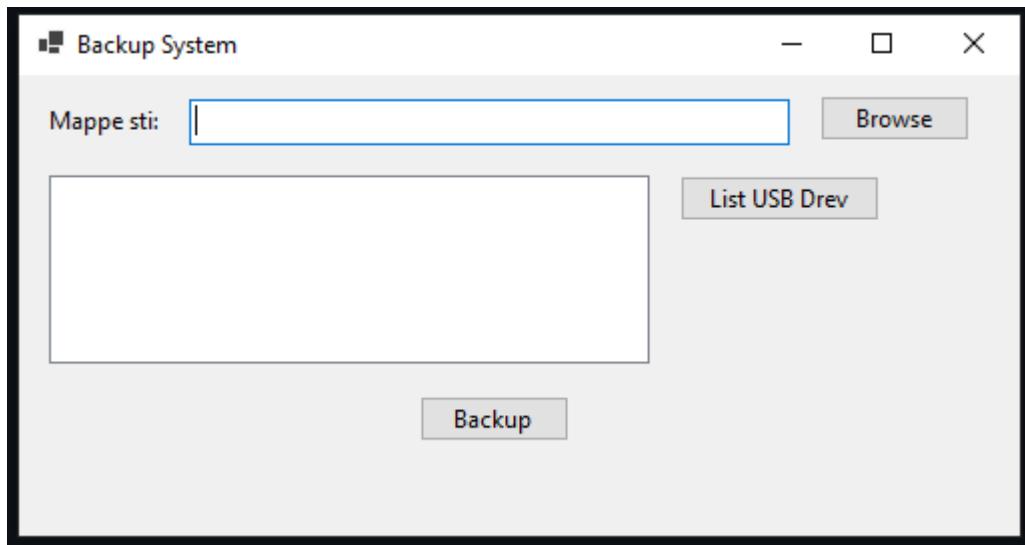
<https://www.edrawsoft.com/article/uml-system-sequence-diagram.html>

Use Case: Applying UML and Patterns-By Craig Larman-ISBN: 0-13-148906-2-Chapter 6.

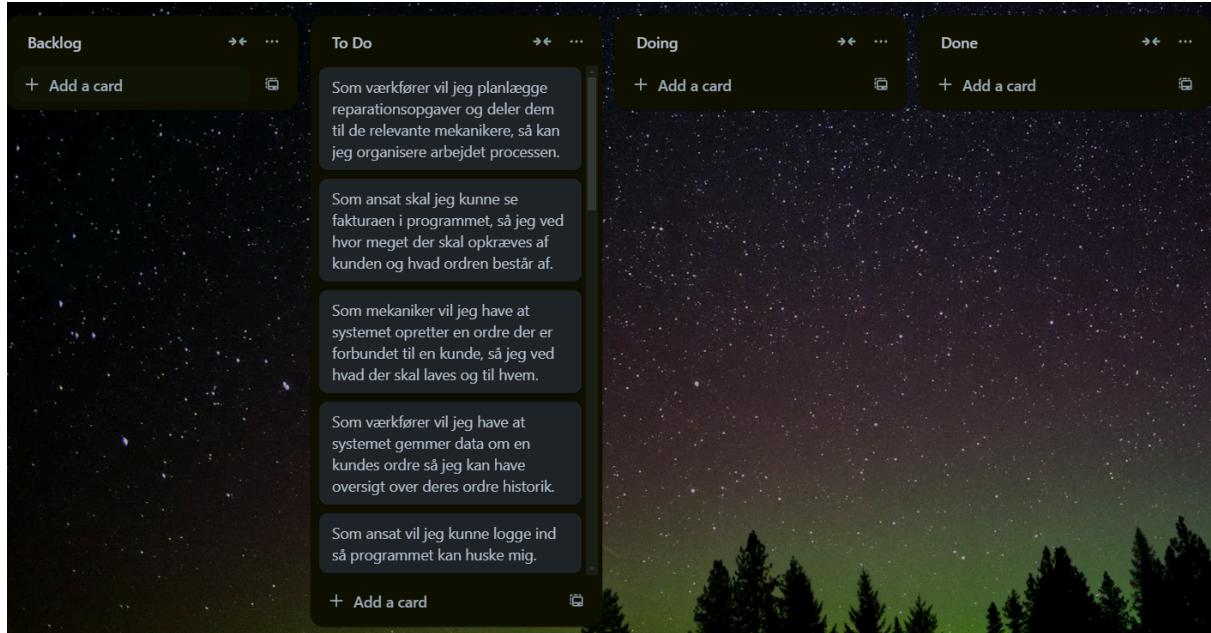
Formartering for Blazor: <https://blazorhelpwebsite.com/ViewBlogPost/34>

Bilag:

Backup program:



User stories. Sprint 0:



Skitse over Blazor, Sprint 0:

