|  |
| --- |
| Førsteårsprøven |
| Ferrari Financing System |
| FFS |

|  |
| --- |
| Sofie Blom Jensen, Shahnaz Yahyavi, Martin Tang Olesen  Vejledere: Flemming Koch Jensen, Hans Iversen  Hold: he18dmu-2s17 - Datamatikeruddannelsen  Udannelsesinstitution: Erhvervsakademi Midtvest  01-06-2018  Antal tegn: ? |

Indhold

[Indledning 1](#_Toc515354257)

[Problemstilling 1](#_Toc515354258)

[It-forundersøgelse (Martin) 1](#_Toc515354259)

[Project management 1](#_Toc515354260)

[Udviklingsmiljø 1](#_Toc515354261)

[Iterations- og faseplan (Sofie) 2](#_Toc515354262)

[Estimering 2](#_Toc515354263)

[Fase 1: Inception 3](#_Toc515354264)

[Iteration I1 3](#_Toc515354265)

[Visionsdokument (Sofie) (review Martin) 3](#_Toc515354266)

[Supplerende kravspecifikation (Martin Review Shahnaz) 4](#_Toc515354267)

[Risiko Analyse(Shahnaz) 4](#_Toc515354268)

[Domænemodel (Sofie, review Martin) 5](#_Toc515354269)

[Use case diagram (Sofie, review Martin) 6](#_Toc515354270)

[Use cases (Martin, review Sofie) 8](#_Toc515354271)

[Klassediagram – trelagsarkitektur (Shahnaz review Sofie) 10](#_Toc515354272)

[Mock-ups(Shahnaz review Sofie) 12](#_Toc515354273)

[Fase 2: Elaboration 13](#_Toc515354274)

[Iteration E1 13](#_Toc515354275)

[Klassediagram 13](#_Toc515354276)

[Systemsekvensdiagrammer (Shahnaz review Sofie) 13](#_Toc515354277)

[Operationskontrakter (Martin Review Shahnaz) 13](#_Toc515354278)

[Dataordbog(Shahnaz) 14](#_Toc515354279)

[Data Model (Shahnaz review Sofie) 14](#_Toc515354280)

[Normalisering (Shahnaz review Sofie) 15](#_Toc515354281)

[Sekvensdiagrammer(Sofie) 17](#_Toc515354282)

[facade-controller (Facade patten) 19](#_Toc515354283)

[Singleton pattern(Sofie) 19](#_Toc515354284)

[Tråde(Shahnaz) 20](#_Toc515354285)

[Observer pattern(Sofie) 21](#_Toc515354286)

[Unit-test (Martin review Sofie) 22](#_Toc515354287)

[Iteration E2 24](#_Toc515354288)

[GRASP (Grundlæggende designprincipper for objektorienteret design) (Shahnaz) 24](#_Toc515354289)

[Datamodel 25](#_Toc515354290)

[Klassediagram (Shahnaz) 25](#_Toc515354291)

[Aktivitetsdiagram (Shahnaz Review Sofie) 26](#_Toc515354292)

[Algoritme 27](#_Toc515354293)

[Tests (Martin review Sofie) 27](#_Toc515354294)

[Exception (Martin) 29](#_Toc515354295)

[Rational Unified Process(Martin) 30](#_Toc515354296)

[CSV File(Shahnaz) 32](#_Toc515354297)

[Konklusion 34](#_Toc515354298)

[Litteraturliste 35](#_Toc515354299)

[Bilag 36](#_Toc515354300)

[Bilag 1 – Iterations- og faseplan 36](#_Toc515354301)

[Bilag 2 – Visionsdokumentet 37](#_Toc515354302)

[Visionen 37](#_Toc515354303)

[Interessentanalyse 37](#_Toc515354304)

[Feature-liste 37](#_Toc515354305)

[Bilag 3 – Use case diagram, første udkast 38](#_Toc515354306)

[Bilag 4 – Use case 1 39](#_Toc515354307)

[Bilag 5 – Use case 2 40](#_Toc515354308)

[Bilag 6 – Use case 3 41](#_Toc515354309)

[Bilag 7 – Domænemodel 42](#_Toc515354310)

[Bilag 8 - Mock-ups 43](#_Toc515354311)

[Bilag 9 – Datamodel 45](#_Toc515354312)

[Bilag 10 - Risikoanalyse 46](#_Toc515354313)

[Bilag 11 - Testsuite 47](#_Toc515354314)

[Bilag 12 - Klassediagram 48](#_Toc515354315)

# Indledning

## Problemstilling

# It-forundersøgelse (Martin review Shahnaz)

En It-forundersøgelse der benytter MUST principperne er inddelt i 4 faser; forberedelsesfasen,

fokuseringsfasen, fordybelsesfasen og fornyelsesfasen. For at holde styr på de forskellige faser og hvilke artefakter der kommer ud af dem, benytter man referencelinjeplanlægning 1 , dette gør det muligt at overvåge og regulere processen. For at kunne arbejde med denne form for planlægning nedsættes to grupper, en projektgruppe og en styregruppe. Projektgruppen står for udarbejdelse af de forskellige artefakter mens styregruppen står for godkendelse af disse artefakter. Hver gang en reference linje mødes af projektgruppen vurderes artefakterne, bliver disse dømt ok kan den næste fase påbegyndes. Da vi ikke har haft mulighed for at lave en reel it-forundersøgelse hos virksomheden selv, har vi ikke gjort brug af dette værktøj. Vi forventer at denne form for proces er blevet benyttet som forarbejde til vores case. I Rationel Unified Process, fremover forkortet UP, laver vi i inception fasen arbejder vi netop ud fra firmaets nuværende problemstilling, hele forrmålet med inception er netop at afklare om det giver mening, både for kunden og os, at lave dette it-system. Vi startede med at lave dokument analyse på den udleverede case, ud fra denne kunne vi udlede en masse krav og ønsker. Disse Krav og ønsker blev så formuleret i et visionsdokument og et supplerende krav specifikations dokument. Mock-ups blev også tegnet så vi kunne få feedback fra kunden omkring brugervenlighed, disse blev også brugt til at sikre den ønskede funktionalitet var med i programmet fra et tidligt stadie.

# Project management

## Udviklingsmiljø

En af de første ting vi gjorde i projektet var at opsætte vores udviklings miljø. Vi var enige om at vi ville benytte github som vores platform til at sammenflette vores individuelle kode. Vi havde i gruppen allerede en organisation fra et tidligere projekt og oprettede derfor blot et ny repository til dette projekt. Vi har derefter oprette branches så vi alle haft vores egen til at udvikle i, mens masteren så er samlingspunkt for kode der allerede er implementeret. Projektet er programmeret i Java og vi har dertil brugt Eclipse som vores IDE, dertil har vi oprettet vores projekt og integreret de udleverede API’er. Som database har vi brugt Microsoft SQL server, dette har vi installeret på alle vores maskiner og vi har så haft en query fil som vi alle har benyttet til at oprette den specifikke database. Query filen har sammen med alle vores andre artefakter også lagt i vores github repository

## Iterations- og faseplan (Sofie)

Som en del af project management har vi udarbejdet en iterations- og faseplan. Denne er med til at sikre, at vi holder et overblik over de kommende iterationer samt de milepæle der skal nås for hver fase. Den udarbejdes i starten af inception fasen, da den givetvis er et grundlæggende led i project management. Den er dog på dette tidspunkt endnu ikke helt stabil, da vi har behov for at opdatere den løbende. Vores plan indeholder desuden en detaljeret opremsning af de aktiviteter, vi gennemfører den enkelte dag i hver iteration. For hver dag har vi valgt at starte med at se på vores plan for dagen, og vi afslutter desuden dagen med at opdatere planen, samt at føre en log over hvad vi har nået. Det hjælper os til at overholde planen samt reflektere over vores arbejdsproces. Iterations- og faseplanen bør være stabil inden vi går i construction-fasen.

## Estimering

# Fase 1: Inception

## Iteration I1

### Visionsdokument (Sofie) (review Martin)

Vi har valgt at begynde med at udarbejde visionsdokumentet[[1]](#footnote-2) som en af de første artefakter i den første iteration. Visionsdokumentet er fordelagtigt at arbejde med i starten af projektet, da man får indskrænket problemdomænet og formålet med projektet uden dog at begrænse løsningsmulighederne for meget. Visionsdokumentet består af en beskrivelse af visionen, en interessentanalyse og en feature-liste.

Visionen er en kortfattet fremtidsbeskrivelse, der formuleres på en måde der ikke specificerer teknologivalget, med mindre kunden har nogen helt specifikke krav hertil. Den sammenligner ikke det nye system med noget tidligere, men har fokus på fremtiden og løsningen af den nuværende problemstilling. I vores tilfælde er der altså fokus på, at systemet skal samle alle skridt der tages i forbindelse med afgivelse af lånetilbud, når Ferrari skal sælge en bil med finansiering. Visionsdokumentet tager også højde for, at Ferrari er interesseret i et intuitivt brugerinterface uden forsinkelser, en effektivisering af virksomhedens nuværende proces og en minimering af tab af salg, hvilket de har oplevet som en konsekvens af den nuværende arbejdsgang. Visionen indeholder også dele af vores it-forundersøgelse, da vi har fået en case udleveret er vi nødsaget til at bruge elementerne fra case. Normalt vil mange af disse punkter have dukket op igennem netop denne analyse

Interessentanalysen er en analyse af alle der har interesse i løsningen, vi når frem til. Der kan være tale om brugere, kunder, lovgivere, ejere af understøttende systemer og så videre. Med andre ord: interessenter der anvender eller betaler for produktet, tilbyder services til os som udviklere eller på anden måde regulerer problemdomænet. Interessentanalysen tager højde for sådanne interessenters konkrete ønsker, krav og behov. Vores interessentanalyse tager således udgangspunkt i forhandlerens kunder, dem der er interesseret i at købe en bil, bilsælgerne, kontorassistenter og økonomimedarbejdere samt salgschefen og den bank, forretningen giver finansieringstilbud i samarbejde med. Vi henviser til Bilag 2 for en mere detaljeret gennemgang af interessentanalysen og de enkelte interessenters interesser og behov.

Feature-listen er et element i visionsdokumentet, der skaber overblik over systemets funktioner. Det er således en liste over alt hvad systemet ud fra problemdomænet skal kunne udføre overordnet set, og er med til at danne grundlag for den videre udviklingsproces. Vores feature-liste indeholder de funktioner, der er behov for i forbindelse med afgivelse af lånetilbud hos bilforhandleren. Vi har valgt at medtage kreditværdighedstjek hos RKI, indhentning af aktuel rentesats hos banken, beregning af rentesats ud fra givne oplysninger fra banken og RKI, registrering af et lånetilbud til en kunde samt eksport af et lånetilbud med en tilbagebetalingsplan.

### Supplerende kravspecifikation (Martin Review Shahnaz)



Figur 1 – FURPS+

Den supplerende kravspecifikation udarbejdes efter visionsdokumentet og sideløbende med use case-diagrammet[[2]](#footnote-3). Det er vigtigt at understrege dette kun er, som navnet netop indikere, supplerende, alle krav der kan tilskrives specifikke use cases bør ikke optræde her, hvis denne regel ikke overholdes, kan vi ende med at gøre systemet mere omfattende end nødvendigt. Vi bruger FURPS+[[3]](#footnote-4) til at kategorisere de krav vi mener bør være i den supplerende kravspecifikation. Når alle kravene er samlet, bør de gøres målbare.

### Risiko Analyse(Shahnaz)

Der skal medtage kun relevante risikoer fordi det øvrige udstyring af analysen. Risiko skal være afgørelige til at kan man finde en læsning til den. Risiko skal også have en sandsynlighed til at kan regne med hvor meget vigtigt til at finde løsning til den eller bare forhindre den før at begynder arbejde på systemet.

Vi har brugt risiko analyse til at finde ud at hvordan kan enlig forhindre risici i projekten før vi begynder og arbejder med den. Men vi kunne ikke vurdere meget tal i den og beregner noget økonomiske risici fordi vi kunne ikke arbejde med rigtig prise på systemet. Man kan godt vores risiko analyse i bilag 10.

### Domænemodel (Sofie, review Martin)



Figur 2 - Domænemodelnotation

I den første iteration har vi valgt at udarbejde et første udkast til domænemodellen[[4]](#footnote-5). Domænemodellen er en måde at visualisere og analysere problemdomænet på et forholdsvis tidligt stadie i projektet. Modellen viser relationerne mellem forskellige koncepter i problemdomænet, og vi udarbejder den ud fra problemformuleringen og de use case beskrivelser vi har på nuværende tidspunkt. Vi tegner domænemodellen så den ligner UML-klassediagrammer, dog med en del variationer i notation[[5]](#footnote-6). Kort sagt er notationen opbygget således at kasserne, der i et klassediagram ville repræsentere klasser, nu snarere repræsenterer koncepter, der ikke nødvendigvis skal afspejles som klasser i det færdige system. Dog kan de tjene som inspiration når vi begynder at udarbejde klassediagrammer, datamodeller osv. Pilene viser association mellem de forskellige koncepter, samt om der er tale om multiplicitet.



Figur 3 – Domænemodel

Vores første udkast til en domænemodel ses i *Figur 3*. I vores domænemodel har vi valgt at medtage Salgschef, Bank, RKI, Kunde, Bilsælger og Lånetilbud som koncepter i problemdomænet. Her er det sælgeren der kommer til at interagere med systemet som primær aktør, derfor har han associationer til alle andre koncepter i problemdomænet.

Sælgerens relation til både banken og RKI består i, at han indhenter oplysninger fra dem, der er nødvendige for at udarbejde det endelige lånetilbud. Derfor viser modellen også, at der kan være flere sælgere, men der er altid kun tale om én bank og ét RKI register. Her ved vi at sælgeren henter rentesatsen fra banken i forbindelse med beregningen af lånetilbuddet, og han henter kundens kreditvurdering fra RKI.

Sælgerens relation til salgschefen er, at han arbejder under ham. Ud fra problemformuleringen ved vi, at der kun er én salgschef, men flere sælgere.

Bilsælgerens relation til det enkelte lånetilbud er, at det er ham der opretter og afgiver det. Han kan naturligvis oprette mange lånetilbud, da de vil være unikke for den enkelte kunde. Til gengæld vil det kun være én sælger der har tilknytning til hvert tilbud. Dog kan flere sælgere have fat i tilbuddet eftersom hver kunde kan interagere med flere sælgere.

Associationspilen mellem kunden og sælgeren går begge veje, da de har kontakt med hinanden og begge kan tage initiativ til denne kontakt. En sælger kan traditionelt sagtens have kontakt med flere forskellige kunder. Kunden kan også have kontakt med flere sælgere, men vi forstår ud fra problemformuleringen, at der er behov for en kontrol af, at den enkelte kunde kun modtager ét lånetilbud, ligegyldigt hvor mange sælgere han har været i kontakt med i forbindelse med købet. Dette er en af de centrale ting, som vores kunde, den regionale Ferrari-forhandler, har bedt om at få optimeret ved implementationen af dette system, hvorfor det er vigtigt for os at understrege det i vores domænemodel.

### Use case diagram (Sofie, review Martin)

Use case-diagrammet giver overblik over funktionelle krav, hvorfor den er fordelagtig at begynde på i den første iteration. Den bør være stabil inden overgang til construction-fasen, da det er en del af milepælen der skal nås indenfor elaboration. På nuværende tidspunkt behøver den dog ikke være stabil eller færdiggjort, da vi ikke har identificeret alle use cases endnu. Den anvendes blot som et værktøj til os som udviklere til at holde overblik over de use cases vi har identificeret på nuværende tidspunkt.

Use case-diagrammet viser den primære aktør, der interagerer med systemet, og de understøttende aktører, der bidrager til opfyldelse af målet med hver af de identificerede use cases. En mere detaljeret formel beskrivelse af de enkelte use cases og deres formål findes under afsnittet ”Use cases”.

Identifikation af use cases er en del af Elementary Business Process (EBP), hvilket repræsenterer en virksomheds aktiviteter på det mest elementære niveau. En use case udgør således som udgangspunkt en EBP, idet det kan defineres som én opgave, der udføres af én bestemt person i forbindelse med én form for forretningshændelse. Denne opgave vil føre til en værdi og nogen data som er målbare for virksomheden. En sådan EBP består så af flere trin, der tilsammen vil skabe denne værdi. Det kan derfor være svært at identificere use cases, da der kan opstå tvivl om hvilket niveau de bliver defineret ud fra. For eksempel: er ”opret låneaftale” en passende use case? Eller består den af for mange trin? Eller: ”Start Programmet”, vil det være en passende use case, eller består den af for små skridt. Man bør altså have fokus på hvor mange skridt en use case vil indeholde, og hvilken værdi den vil skabe. Men som sagt giver use case-diagrammet blot et overblik over de identificerede use cases, og afspejler således ikke disse detaljer[[6]](#footnote-7). De er blot nødvendige i forbindelse med identifikationen af use cases.









**Kommunikation**

****

**System boundary**

Figur 4 – Use case-diagramnotation

Vores use case diagram er forholdsvis begrænset i omfang, og vi har kun på nuværende tidspunkt få primære aktører at fokusere på, nemlig sælgeren og salgschefen, og derfor har vi valgt kun at lave ét diagram. Hvis der var mange aktører med mange use cases til hver, ville det være fordelagtigt at dele diagrammet op, således at der var ét diagram pr. aktør. I vores tilfælde giver det dog mest mening at beholde de identificerede use cases i ét diagram, som vi herefter kan opdatere og udvide efterhånden som processen skrider frem i de kommende iterationer. Notationen vi har benyttet i forbindelse med udarbejdelse af use case-diagrammet ses i *figur 4*. i Bilag 3 ses vores første udkast til et use case-diagram. I udarbejdelsen af use case diagrammet har vi desuden haft fokus på at de forskellige use cases og aktører alle sammen eksisterer i domænemodellen. Det er endnu en måde at sikre sig, at de use cases vi identificerer, er passende og forbliver indenfor projektets scope. Som det ses, er alle aktørerne, salgschef, sælger, bank og RKI repræsenteret som koncepter i domænemodellen. Desuden passer alle use case-titlerne til forholdet mellem de forskellige koncepter i domænemodellen; for eksempel er ord som ”kreditvurdering” og ”rentesats” kendte både i domænemodellen og use case-diagrammet. Det er desuden værd at bemærke, at alle use cases i det første use case-diagram er repræsenteret som Concrete use cases, idet de startes af en aktør og realiserer aktørens ønskede mål med casen[[7]](#footnote-8). Vores use case diagram vil blive nærmere beskrevet i elaboration afsnittet, da det som sagt er i denne fase den skal være stabil.

### Use cases (Martin, review Sofie)

Use cases spiller en stor rolle i UP, det er dem der driver projektet fremad og det er ud fra vores use cases vi vælger det næste skridt. Ved hjælp af use case-diagrammet fik vi hurtigt udvalgt de første use cases vi vil tackle. En god hovedregel i UP er at tackle den use case med størst risici involveret først, netop fordi et af de primære formål med inception fasen er at afdække om vi overhovedet kan lave dette system. Vi udvalgte hurtigt 3 use cases, som vi i fællesskab var helt enige om var kernen i dette system, og dermed her de største risici ligger. Den store use case som vi har valgt at kalde ”FFS-UC3 - Udregn lånetilbud”[[8]](#footnote-9), indeholder alle de tre trusler vi har identificeret som værende de store; vi har derfor valgt at tackle de 3 trusler som værende hver deres use case. Den første use case, kaldet ”FFS-UC1 - Tjek kreditværdighed”[[9]](#footnote-10), blev så til en underfunktion af UC3. Det samme gør sig gældende for ”FFS-UC2 - Indhent aktuel rentesats”[[10]](#footnote-11). Vi har valgt at beskrive disse use cases formelt; vi kunne også have brugt et uformelt use case format, men vi mener, at eftersom dette er kernen af systemet, vil det hjælpe fremadreddet, at de er beskrevet på det formelle niveau. Det formelle format er netop godt, fordi de funktionelle krav bliver beskrevet på en formel og utvetydig måde. Målet er at få beskrevet hovedscenariet og alle variationer af dette.

Det er vigtigt at vi har et **unikt id** til alle vores use cases, da vi laver mange andre artefakter ud fra disse, og det er vigtigt at holde sporbarhed hele vejen tilbage til den konkrete use case. **Titlen**, der står efter vores unikke id, prøver vi så vidt muligt at holde som en beskrivende kommando, der indfanger use casens formål - den holdes på bydeform. **Afgrænsningen** af use casen fortæller hvad der berøres af use casen, i mindre systemer som dette vil afgrænsningen meget naturligt være systemet selv; dette er også tilfældet for alle vores use cases. **Niveauet** indikerer use casens placering i hierarkiet, den typiske og mest brugte her er "brugermål", men vores UC1 og UC2 er underfunktioner til UC3 og har derfor fået niveauet ”underfunktion” i stedet. Den **primære aktør** beskriver den person, der interagerer med systemet i den konkrete use case; som nævnt i afsnittet ovenfor bestræber man sig på at det kun er én bruger pr. use case. Det er vigtigt at bemærke, at vi forsøger at beskrive rollen på aktøren så præcist som muligt, dvs. betegnelsen ”bilsælger”, som er vores aktør, er væsentligt bedre end blot at kalde ham en ”bruger”.

**Interesser og interessenter** beskrives også i de enkelte use cases. Her trækker vi en rød tråd tilbage til vores visionsdokument og udvælger de interessenter, som har interesse i den konkrete use case. **Forudsætninger** dækker over krav der skal være opfyldt før use casens udførelse kan gå i gang. Det kan være andre use cases som skal være afviklet først, som vi ser i vores UC3. Det er vigtigt at man ikke nævner trivielle forudsætninger, da det ikke har nogen værdi for os som udviklere – eksempelvis ”computeren er tændt” eller ”brugeren er logget ind” hvis de ting siger sig selv. **Succesgaranti** dækker over de forhold der skal være opfyldte ved use casens afslutning. De gælder primært for hovedscenariet, men alternative scenariers succesgarantier kan også beskrives. disse succesgarantier bør opfylde alle vores interessenters interesser. For vores use cases var det ret simpelt og ligefrem at beskrive vores succesgarantier, da de bare er opfyldt når den funktion er fuldendt korrekt.

**Hovedscenariet** beskriver det oftest forekommende scenarie. Det beskrives uden variation og skal som hovedregel opfylde alle interessenternes interesser. På grund af størrelsen af vores UC1 og UC2 er der ikke mange trin i disse, men hovedscenarier kan ofte blive på mange trin i use cases der er mere omfangsrige.

**Varianter** er hvor man beskriver alternative scenarier, der kan give en variation fra hovedscenariet - både successcenarier men også fejlhåndteringsscenarier. Disse varianter består af 2 trin. Første trin er betingelsen, her beskrives betingelsen samt hvordan den opdages. Trin 2 er håndteringen. Her beskrives de trin, der skal tages i denne variation, hvilket kan lede til en fortsættelse til hovedscenariet eller specifikt vælge at afslutte use casen. De primære varianter, vi har brugt i vores UC1 og UC2, er fejlhåndteringer.

**Teknologier og dataformer** dækker over eventuelle krav der kan være stillet fra kunden. Vi bør som hovedregel prøve at begrænse disse mest muligt for ikke at ligge os fast på teknologivalg så tidligt i processen. Krav her kan i værste tilfælde blive en trussel for os senere, som vi så skal håndtere og dermed gøre systemet dyrere for kunden. I vores tilfælde er vi blevet stillet visse krav til brug af visse API'er, derfor er disse medtaget i vores UC1 og UC2.

**Ikke-funktionelle krav**, som hører til use casen, eller som afdækkes mens use casen udføres, tilføjes her; de bør dog også blive tilføjet til den supplerende kravspecifikation senere. Et af de ikke funktionelle krav, som går igen i både UC1 og UC2, er kravet om at vores kald til RKI og bank ikke må påvirke brugbarheden af vores brugergrænseflade, mens de foretages.

**Hyppighed** beskriver hvor tit en use case forekommer. I vores tilfælde er det ikke noget, der reelt vil ske hurtigt efter hinanden - vi har dog skrevet ofte fordi vi mener vores system skal kunne understøtte at man laver disse kald inden for kort tid (for eksempel mange gange på én dag).

Det sidste punkt på den formelle use case er **diverse**, hvor man beskriver det, som ikke passer ind under de andre punkter. Vi kan også tilføje kendte eller eventuelt uløste problemstillinger, man har opdaget undervejs. Vi har ikke haft noget til diverse posten i nogle af vores use cases og den er derfor ikke taget med på den formelle beskrivelse af use casen.

### Klassediagram – trelagsarkitektur (Shahnaz review Sofie)

Trelagsarkitektur er en model til beskrivelse af de blokke et program er opbygget af. Modellen anvendes blandt andet fordi det gør det nemmere at overskue kommunikationen mellem lagene og kommunikationen med brugerne og med andre IT-systemer. Vi bruger trelagsarkitektur til at holde struktur i vores system, og vi sætter klasser i klassediagram på plads til at vi kan selv se hver opgave til den enkelte klasse i vores system. Udover det bruger man interfaces og trelagsarkitektur til at gøre vores system mere brugervenlig til andre udviklere i fremtid at vedligeholde og modificere. I vores trelagsarkitektur[[11]](#footnote-12) er lagene defineret på følgende måde:

Database

**View-laget** er det øverste lag, der håndterer modtagelse og præsentation af data til brugeren af systemet. Dette lag er kendetegnet ved at være ”tæt” på brugeren af programmet. Her findes på nuværende tidspunkt kun vores GUI-klasse.

**Logic-laget** er det midterste lag, der håndterer udvekslingen af data mellem præsentationslaget og datalaget. Det indeholder vores forretningslogik, og det bør have så få koblinger som muligt, både opad og nedad, da det således er lettere at vedligeholde og beholde, da det senere skal kunne benyttes på en web-platform.

**Data-laget** er det nederste lag, der opbevarer og håndterer data. Dette lag er også kendetegnet ved at være ”tæt” på computeren. Her findes vores database, samt vores klasse til oversættelse mellem databasen og forretningslogikken. Denne klasse vil nødvendigvis have en del kobling begge veje, og derfor vil den blive nødt til at blive skiftet ud, hvis man ønsker at flytte over til en anden form for database.

Vores tegning af arkitekturen, som ses i *figur 5* er på nuværende tidspunkt stadig meget mangelfuld, da den ikke indeholder alle de klasser vi kommer til at have i det færdige system. Dog hjælper den os til at få overblik over vores system og reflektere over den valgte arkitektur.



Figur 5 – trelagsarkitekturen som vi tegner den

### Mock-ups(Shahnaz review Sofie)

Mocke-ups er vigtige værktøjer til at kommunikere prototypes brugbarhed og funktionalitet til kunderne, hvilket giver et præcis visuelt supplement til verbaliserede ideer og designs. Vi laver mock-ups til at få et overblik over, hvordan systemets user interface skal se ud senere. Mock-ups er også meget nyttige til at vise en frem gørende skridt i projekten til kunden. Det er desuden en håndgribelig ting vi kan tage med og vise kunden, hvilket vil hjælpe dem til at kommunikere hvad de helt konkret har behov for og forventer af systemet. I vores tilfælde har vi lavet en interview med vores vejleder, som har hjulpet os til at se hvordan vores system i fremtiden skal være. Vi har udarbejdet mock-ups om ”opret lånetilbud” som den er anmoder om at giver oplysninger til en kunde og udregner lånetilbud og bekræfter oplysninger. Vores Mock-ups kan ses i Bilag 8.

## Fase 2: Elaboration

## Iteration E1

### Klassediagram

### Systemsekvensdiagrammer (Shahnaz review Sofie)

Systemsekvensdiagrammer viser hvordan en bruger kommunikerer med systemet i forhold til en konkrete use case scenarier. Den fokuserer også på hvad systemets respons er i forbindelse med bestemte stimuli fra brugeren. I udarbejdelsen af systemsekvensdiagrammer kigger vi på systemet ligesom en black box, og vi viser alle de funktioner som kommer fra en use case over systemet. Systemsekvensdiagrammer er virkningsfulde til at analysere systemet, og de viser hvad systemet egentlig gør. De er også meget gode til at illustrere rollerne(aktør) i systemet. Vi bruger systemsekvensdiagrammer til at identificere systemoperationer. Vi har lavet SSD1 og SSD2 som handler om at hvordan vores system kommenekere med banken til at henter renteset og RKI til at vurdere kreditværdighed til en kunde og den 3. SSD handler om at hvordan bilsælger som er primære aktør i vores system kommunikere med systemet giver det nogle oplysninger og får udregnet låne tilbuddet til kunden og til sidste bekræfter informationer til.

### Operationskontrakter (Martin Review Shahnaz)

Operationskontrakter[[12]](#footnote-13) bruges i UP når vi føler vi har brugt for en mere detaljeret beskrivelse af hvordan systemet opfører sig. Der er fokus på de ændringer der bliver foretaget. Operationskontrakter[[13]](#footnote-14) skal skabe overblik over hvad der sker og ikke hvordan det sker. En OC beskrives formelt og har et specifikt format som benyttes. Vi starter med et unikt id og et navn, det unikke id bruges i tilfælde af navneændringer senere. Navnet består af systemoperationens betegnelse minus eventuelle parametre. Vores OC’er er navngivet, FFS-OC1[[14]](#footnote-15) og FFS-OC2[[15]](#footnote-16), så vi kører videre med formatet fra vores formelle use cases. Systemoperationen der er tale om skal angives med eventuelle parametre, som vist i OC1. Krydsreferencer viser alle de use cases hvor systemoperationen anvendes, vores OC’er udspringer fra FFS-UC1 og FFS-UC2, disse er derfor listet på hver deres OC, udover det bruger FFS-UC3 begge disse systemoperationer og er derfor også listet. Forudsætninger angiver alle de forudsætninger der skal være imødekommet før operationen kan kaldes, de skrives i nutid, én linje pr. forudsætning for at bevare overblik. Det er ikke altid der er nogen forudsætninger, som i vores OC2. Slutbetingelser angiver alle de betingelser som skal være opfyldt inden operationen bliver afsluttet. Det er værd at notere at disse betingelser kan med god mening benyttes til testformål. Slutbetingelser skrives i førdatid, der anvendes dot-notation til reference af attributter.

### Dataordbog(Shahnaz)

Data ordbog beskriver alle væsentlige termer og forkortelser i problemdomænet. Den er også giver eksempler på dem. På den måde systemudvikler kan forstå forretning sprog og forstå bedre hvad henleder systemet om. Den beskriver en general beskrivelse om en koncept og beskriver også en konkrete eksempler på den.

### Data Model (Shahnaz review Sofie)

Data modellering er en proces, der bruges til at definere og analysere datakrav, der er nødvendige for at understøtte forretningsprocesserne inden for rammerne af tilsvarende informationssystemer i organisationer som men kan finde dem.

Når man analyserer præcise hvilken koncepter man har i sit system, laver en domænemodel og bagefter laver en datamodel er de koncepter som kan alle objekter i systemet arbejde med.

Man har bruger dokument analyse til at kommer med noget koncepter til at forstå hvad er enlige vores verden i systemet skal være, men nogle er de koncepter skal men bruges til at forbinder de use caser sammen med noget oplysninger som vi analysere ud fra data model.

Data modellering kan udføres under forskellige typer projekter og i flere faser af projekter.

Datamodellerne er fremadskridende. Det vil sige, skal en datamodel betragtes som et levende dokument, som vil ændre sig som gennem systemudvikling af et system gennem tid.

Vi har brugt datamodel til at finde vores entities og alle attributter som den entities har, og vi også vi fundet ud af det at hvilke de oplysninger som vi har endelig en entities eller bare en attribut som hører under en entities. Vores datamodel findes i bilag 9

### Normalisering (Shahnaz review Sofie)

Normalisering af en database er en teknik, som sikrer, at rettelser i databasen kan foretages med mindst mulig indflydelse på det oprindelige system. Målet er at minimere redundant data. Det vil sige at samme oplysning er gemt flere steder. Med normalisering bliver det lettere at foretage rettelser i databasen (så skal man ideelt kun rette det ét sted). Lidt en balancegang i forhold til performance, men det skal være velovervejede designbeslutninger, og ikke dovenskab der skal være styrende for, om man vælger at gå på kompromis med normalformerne. Normalisering er godt i teorien, og er absolut en god tommelfingerregel.

**

Figur 6 – datamodellen efter 1. normalform er opfyldt

Problemerne med redundans kan deles op i to grundargumenter: **Performance** bliver påvirket idet dataudtræk kan blive langsomme, da den redundante data medfører, at der bliver brugt unødvendige ressourcer (IO, memory, netværk og cpu). **Maintainability** påvirkes eftersom den samme opdatering skal foretages mange steder.

Udarbejdelsen af vores database efter de tre normalformer udformede sig således:

**1. normalform:** I vores første udkast til modellen indeholdt tabellen alle de attributter som ses i *figur 5*. Denne database opfylder første normalform, da denne bare er nok til at oprette en database. I denne normalform øges risikoen dog for uregelmæssigheder, når data skal opdateres, da de samme data findes i flere tabeller. Vi har i vores løsning til første normalform altså dataredundans, da vi ikke opretter flere tabeller, men blot tilføjer flere og flere rækker i den eksisterende tabel.

**2. normalform**: Ifølge anden normalform skal tabellen først opfylde alle krav til første normalform. Hvis tabellen har en sammensat nøgle, skal alle felter, der ikke indgår i nøglen, afhænge af den samlede nøgle. Alle non-prime attributter skal være fuldt funktionelt afhængige af primærnøglen. Det vil sige, at der bliver ingen partiel funktionel afhængighed. Hvis der kun findes en enkelt primær nøgle, er tabellen allerede i anden normalform. Hvis der derimod er en sammensæt primærnøgle kan den stadig bringes i anden normalform ved at splitte tabellen op i separate tabeller. I vores database skal alle attributter være fuldt afhængige af primærnøglen (fremover forkortet som PK), som i det første udkast er id (det vil sige id på lånetilbuddet). Derfor opretter vi flere tabeller. Vi opretter en tabel til henholdsvis kunde, bil og bilsælger, hvor vi benytter id som PK i hver. Hvad hver tabel indeholder ses i *figur 6*.

**

Figur 7 – datamodellen efter 2. normalform er opfyldt

**3. normalform**: Ifølge tredje normalform skal tabellen først opfylde alle krav til anden normalform. Der må derudover ikke findes felter udenfor PK som er indbyrdes afhængige. Ingen transitive funktionelle afhængigheder mellem non-prime attributter. Der må ikke være nogen attributter udover PK, der er unikke. Vi kan ikke opfylde denne normalform, da kundens cpr-nummer er unikt, men vi bruger id som PK. Det er af hensyn til persondatasikkerhed et krav, at cpr-nummeret ikke benyttes som en nøgle. Vi er derfor tvunget til at benytte en anden nøgle som er unik. Vi har derfor valgt at bruge kundens telefonnummer som PK, da vi må formode, at det også vil være unikt.

Vi har lavet mange ændringer i datamodelen og normal former i hver iteration og men kan godt se den sidste datamodel i bilæg 9.

### Sekvensdiagrammer(Sofie)





**Et objekt med en lifeline**

****

**Metodekald**

****

**Returkald**

****

****

**Found/lost-kald**



**Execution specification**

*Figur 8 - sekvensdiagramnotation*

Et sekvensdiagram er et designdiagram, der er meget kodenært. Det kan benyttes til at visualisere dele af vores kode der er sekventielle, ved at vise kommunikationen mellem softwareobjekter. Det hjælper os med at determinere hvilke objekter, der bør have ansvar for hvilke konkrete opgaver i en sekvens. Vi har i denne iteration valgt at udarbejde to sekvensdiagrammer, ét for henholdsvis UC1 og UC2. Den notation, vi har benyttet, er et finde i figur 8. Som det fremgår af figuren, bruger vi en kasse med en lifeline til at symbolisere et objekt med en type og en ’levetid’. Vores lifeline er med til at fortælle noget om hvornår et objekt skabes og hvornår det ikke længere bruges. I vores diagrammer[[16]](#footnote-17), er alle objekter dog eksisterende gennem hele sekvensen, derfor er deres lifelines samme længde. Diagrammet viser ved hjælp af pile hvornår metoder kaldes og returnerer, samt eventuelle parameter- og returtyper. Den vej pilen peger, indikerer hvilket objekt der indeholder metoden, og således får kontrollen. Kontrollen symboliseres ved en execution specification, der desuden indikerer for os, hvor mange metoder der på et givent tidspunkt findes på kald-stakken, altså hvor mange metoder der er i gang med at blive udført. Desuden markerer vi lost- og found-kald i vores diagram. Dette er med til at indikere, at en beskeds oprindelse er ukendt. I sekvensdiagrammer er det desuden muligt at illustrere iteration og selektion ved hjælp af forskellige frames, men da vi ikke har haft behov for at benytte denne notation, vil vi heller ikke gennemgå det i dette afsnit.

I vores første sekvensdiagram[[17]](#footnote-18) starter vi med et found-kald, der fortæller os hvilken metode der starter sekvensen, i dette tilfælde ”setCreditRating”, der tager et objekt af typen Customer som parameter. Dette objekt, samt det controller-objekt vi kalder metoden på, har således hver deres lifeline i diagrammet. Controller-objektet får altså kontrollen, og starter således med at anmode om instansen af RKI-klassen, der er en singletonklasse[[18]](#footnote-19). Den giver herefter, vha. et nyt metodekald, kontrollen videre til instansen, der får ansvaret for at slå kundens kreditvurdering op. Det skal altså kommunikere både med Customer-objektet og instansen af CreditRator klassen, der også er en singleton. Først kaldes getCPR-metoden, der er en indkapslingsmetode, der benyttes til at få adgang til Customer-objektets instansvariabel cpr, der er en String. RKI-objektet kalder derefter rate-metoden med denne String som parameter, og giver den returnerede Rating videre til opbevaring i Customer-objektets datakerne. Herefter returnerer alle de aktive metoder tilbage, og kaldstakken reduceres helt ned igen. Dette sekvensdiagram illustrerer på denne måde for os, hvilke data vi har behov for, hvor vi kan få dem fra og hvornår de er nødvendige for os i løbet af sekvensen. Hvis dette sekvensdiagram er udformet rigtigt, bør det gøre det nemt for os at skrive den kode, der skal udføre operationerne illustreret i diagrammet. Det ville også gøre det nemt at overlade det til et helt andet team at programmere operationerne, da de har en udførlig visuel beskrivelse af sekvensen.

### facade-controller (Facade patten)

### Singleton pattern(Sofie)

I vores implementation har vi valgt at benytte det design pattern der hedder ’Singleton’. Dette pattern bruges når man kun ønsker at tillade, at der bliver lavet én instans af en bestemt klasse. Til gengæld skal denne instans også være tilgængelig udefra. Vi bruger singleton pattern i forbindelse med vores klasser ”RKI” og ”Bank”[[19]](#footnote-20). Vi har valgt at bruge dette pattern, fordi det må formodes, at forretningen kun har forbindelse til én bank, og ét RKI register. Det er derfor logisk at gøre de klasser, der repræsenterer disse domæneobjekter, til singletons.

Der indgår flere elementer i implementationen af en singletonklasse. For at være sikker på, at der ikke kan laves flere instanser af klassen, vil vi først sørge for at give constructoren ’private’ som access modifier. På den måde er det kun klassen selv, der får lov til at invoke constructoren. Som sagt ønsker vi dog samtidig, at den ene instans, der findes af klassen, også er tilgængelig. Her indfører vi derfor en instance-metode, der skal returnere denne instans, som naturligvis vil være public. Metoden her vil altså først tjekke, om der er lavet en instans af klassen i forvejen. Hvis der ikke er, så invoker den constructoren og returnerer en ny instans. Hvis instansen allerede findes, vil den selvsagt blot returnere den. Instance-metoden gøres static, således at vi kan kalde den på klassen, frem for en instans af klassen. Instansen vil altså blive skabt, første gang instance-metoden kaldes. Vi gemmer instansen som en privat instansvariabel til klassen.

Hvor godt og anbefalelsesværdigt det er, at benytte singletonmønsteret er meget omdiskuteret, og det er også blevet kaldt for et anti-pattern[[20]](#footnote-21). For eksempel kan en singleton være svær at håndtere i forbindelse med test-scenarier. Det skyldes, at instansen af klassen ikke reelt kan slettes, eftersom klassen selv altid vil holde fast i instansen, og dermed ikke giver plads til, at garbage collectoren kan komme og gøre sit arbejde. Hvis en singleton indeholder meget information, vil det således skabe problemer, i det tilfælde at man har behov for at ”nulstille” tilstanden på den i forbindelse med hver test. Vi har i vores implementation ikke gemt meget information i vores singletonklasser. Klasserne har ikke andre instansvariable end den der er nødvendig for en singleton, og funktionaliteten er også begrænset til få metoder. Derfor har vi ikke behov for at nulstille deres tilstand i forbindelse med vores tests. Vi undgår på den måde mange af de problemer, singletonmønsteret giver, og derfor kan vi benytte det med god samvittighed.

### Tråde(Shahnaz)

Enkelte- trådede programmer, der er en sekventielle proces som begynder fra et sted i et program og efter at kører alle nødvendige kode færdig på en tids punkt og giver svær til sidste, men nogle gange i et program forventes at to eller mange stykker kode kør på samme tid og giver mange forskellige resultater, derfor en programmør nøde til bruger en måde som kan gøre det.

Tråde giver os den mulighed som kan kører mange stykker programmer sammen og giver forskellige resultater. En enkelt-trådet programs tilstand består af objektsystemets tilstand, alle lokale variables tilstande, og det sted programmet er kommet til i sin udførelse., men en tråde tilstand består af alle lokale variable tilstand i tråden og det sted som tråden kommet til, i sin udførelse, så hvis vi vil gerne vad et flere trådet programs tilstand skal vad alle trådenes tilstand og alle objektsystemets tilstande.

Men har to måde at lave tråde i Java, den første at arver (extends) abstrakt klasse Thread som har en Run metode og skriver alle de koder som de vil sin tråde gør i den, og bagefter opkalder på metoden Start på en instans fra vores objekt i Main metode, og den anden måde men kan arver fra interface Runnable som har en abstract Run metode og i main metode laver en Thread objekt som tager en Runnable klasse parameter og giver den et objekt fra vores klasse.

på den måde men kan kun extends Thread i sit program og ikke flere klasser. I vores program vi var nåd til at bruge den anden mode, som det aver interface Runnable, fordi vi var nødt til bruger observer patten (læs mere i afsnit observer patten) og skulle extends klasse observerble i vores program og kunne ikke extends to klasse, så vi har brugt til at implementere interface Runnable.

Vi ville i vores program samtidig til at vi følger oplysninger til vores beregne lånetilbud kommer svær fra IKR (kundes kreditværdighed) og banken(renteset) (se Aktivitet diagram til use case 3 bilæg ??), så vi har brugt tråde til at implementere den stykker af programmet.

### Observer pattern(Sofie)

I forbindelse med vores implementation af tråde har vi benyttet os af observer pattern. Det er et krav til systemet, at vi skal kunne hente oplysninger fra banken og RKI-registret uden at brugergrænsefladen kommer til at hænge - den skal med andre ord kunne tage imod output fra logikken, samtidig med den tager imod input fra brugeren. Problemet er, at de oplysninger vi får fra trådene, skal præsenteres for brugeren umiddelbart efter de er hentet ind. Vi har derfor valgt at gøre brugergrænsefladen - vores GUI-klasse - til observer på trådene. På den måde er vi ikke afhængige af, hvornår trådene lever eller dør - brugeren får oplysningerne med det samme de er fundet, selv hvis tråden ikke er færdig med at køre endnu.

Observer-pattern fungerer således: der er en eller flere observere samt et subject. Observeren tilmelder sig som observer hos subject, med det formål at få besked, hver gang der sker en tilstandsændring hos subject. Subject vil så give besked hver gang der sker tilstandsændringer, og senere kan observeren framelde sig, hvis den ikke længere har behov for at blive oplyst om subjects tilstandsændringer. Observeren tager altså initiativ ved at melde sig til, men det er subject der tager initiativ til at underrette sine observers.



Figur 9 - Observer pattern - diagram over implementation

Et subject har således brug for en addObserver-metode, vha. hvilken observeren kan tilmelde sig. Omvendt skal observeren have en update-metode, som subject kan kalde for at underrette om tilstandsændringer. I den forbindelse vil den sende en reference til sig selv med, så observeren kan skelne mellem hvilket subject der er blevet ændret, i tilfælde af, at observeren har tilmeldt sig flere steder. Her kan man anvende enten push eller pull. Ved pull vil subject kun oplyse sine observers om, at der er sket en ændring, men det er observerens opgave selv at indhente, eller trække (deraf navnet pull) de konkrete ændringer. Push betyder derimod, at subject ikke blot skal oplyse at der er sket end ændring, men også fortælle, eller skubbe (deraf navnet push) observeren hvilken ændring der er sket. Som det sidste vil subject have en deleteObserver-metode, hvor en observer kan framelde sig igen.

Observer mønsteret er understøttet i Java, og det er denne understøttelse vi har valgt at bruge i vores implementation[[21]](#footnote-22). Vi lader altså vores tråde nedarve fra Javas Observable-klasse, hvilket vil give os metoder som setChanged og notifyObservers. Vi bruger den notify-metode, der tager en parameter, da vi ønsker at anvende push. Vores klasse sender altså både en notifikation om ændringen, og den konkrete ændring der er sket, til observeren.

På den anden side implementerer vores GUI-klasse Observer interfacet. Dette interface har den update-metode, som Observables notify-metoder kalder, når der sker en ændring. Vores implementation tjekker således først, hvilket subject der har ændret sig, hvorefter den tager imod ændringen og præsenterer den for brugeren igennem brugergrænsefladen.

Fordi vi har valgt at bruge Javas understøttelse af Observer-mønsteret, bliver vores tråd-klasser ikke mere omfangsrige, da de nødvendige metoder er at finde i observable-klassen, som vi nedarver fra.

### Unit-test (Martin review Sofie)

Unit-tests forsøger at teste så små dele af vores kode som muligt[[22]](#footnote-23). Målet er at fange eventuelle fejl og uhensigtsmæssig opførsel så tidligt i systemets implementation som muligt. Unit-tests er automatiserede, og derfor kan de gentages på et hvilket som helst tidspunkt. Hvis vi blot testede manuelt, ville vi have en tendens til ikke at teste så ofte som vi burde; med de automatiserede unit-tests er alle vores tests dog kun et par klik væk, de bør derfor køres ofte. Vi har brugt JUnit4, da det er et udbredt værktøj med god dokumentation samt integrering i Eclipse.

Vores første unit-tests består i at teste vores eksterne kald til bank og RKI. Et af problemerne, vi løb ind i, var, at kaldet til banken returnerer en rente som varierer. Dette gør det svært at unit teste, da vi netop forsøger at sammenligne et forventet resultat med det reelle resultat. I dette tilfælde er der brugt en af de testmetoder, vi fik stillet til rådighed i stedet. Testen af kaldet til RKI er et godt eksempel på, hvordan vi gerne vil have en unit-test til at se ud: metoden leverer det samme resultat hver gang, og det er den samme metode, vi bruger i det reelle system. Vi har valgt at placere vores unit-tests i en pakke for sig selv - vi kunne også have lavet en ny mappe og lagt ved siden af vores source-mappe, i denne mappe ville vi så forsøge at kopiere vores eksisterende mappe system, men i stedet ligge test-klasserne her. Denne metode vil være mere effektiv hvis vi skulle overdrage vores program, da vi så kunne ekskludere vores test mappe fra buildpath. Da vi blot skal overdrage vores kode, følte vi, at en test pakke var rigeligt. I testpakken ligger testklasserne, som vi laver unit-tests for, disse klasser kan køres individuelt. Sammen med testklasserne ligger vores testsuite[[23]](#footnote-24), som kan køre tests fra flere klasser. Vores er sat til at køre alle tests hver gang. Dette er netop styrken ved unit-tests: selv hvis vi laver noget, vi ikke mener har indflydelse på en anden del af koden, vil en eksekvering af vores test-suite opdage det, hvis vi tager fejl.

## Iteration E2

### GRASP (Grundlæggende designprincipper for objektorienteret design) (Shahnaz)

En systemudvikler eller en programmør ønsker til at skab en software, der er let at forstå og vedligeholde, så den software skal have færre fejl, højere udviklingstempo og lettere vedligeholdelse. Men kan realisere dette i en softwaremodel som ligner problemdomænet mest muligt, Dvs. men skal holde den software mist muligt tæt på problemdomænet. De også ønsker til at skabe softwarekomponenter som kan genbruges i andre programmer, til at kan bruge en hurtigere og billigere udvikling, som kan realiseres dette ved at begrænse afhængigheder mellem de forskellige programmoduler, så de funktionalitet, som har forbindelse med hindenden bør placeres sammen, og konkret implementering (logik lag i program) bør afkobles fra brug (view lag) i program.

**Information Expert,** handler om at, hvor den nødvendige information skal findes. Metoder placeres i de klasser hvor de nødvendige attributter naturligt hører til, til at give mindsker kobling og øger samhørighed, fx i vores program alle set og get metoder befinder sig i klasser som har de attributter som skal get eller setes. **Creator,** handler om hvilket objekt skal skabe et nyt objekt, det objekt som har den stærkeste kobling til det nye objekt, eller den objekt som besidder information om det nye objekter intialiseringsdata, eller aggregerende objekter skaber aggregater (den situation skaber stærkere kobling som vi kan ikke lide den), fx i vores program vores FFScontoroler skaber en objekt fra datalayer, fordi den klasse er en faced controller og skal forbinder data pakke, så har stærkere kobling til det. Og vores GUI klasse skaber kunde og bil og sælgeren objekter fordi den skal intialiseringsdata til dem. **Controller,** handler om hvilket objekt skal modtage input fra GUI, objektet, som repræsenterer en facade for det samlede system (facade controller) (læs i afsnit facade patten), eller objektet, som repræsenterer det konkrete use case-scenariet(session-controller) , system sekvens diagrammer hjælper rigtig meget til vi kan finder ud af det hvor vi skal bruge en controller, controllerne delegerer også objektet. Fx i vores program vi bruger en FFS controller som repræsenterer vores samlede system og delegerer arbejdet. **Kobling**,betegner afhængigheder mellem klasser(læs i afsnit klasse diagram ), men ønsker at begrænse kobling , men samarbejde mellem objekter for udsætter altid en vis kobling som ikke kan undgås, den system som har meget kobling er meget svært til at overskue, genbruge og vedligeholde. **Samhørighed,** betegner det funktionelle slægtskab for indholdet af en klasse, man ønsker altid høj samhørighed, men der er umuligt at sætter alle metoder i en klasse så skaber på den måde den stærkeste kobling.

### Datamodel

[[24]](#footnote-25)

### Klassediagram (Shahnaz)

Klassediagram, der er en design diagram, som visualisering af et statisk perspektiv til kode og implementering er software. Den kan rigtig godt visualisere relationerne mellem softwareklasser, og den har en statiske fokus på programs statiske struktur, den kan også identificere afhængigheder mellem klasserne, når men bruger GRASP principper (læs i afsnit ??) kan men rigtig godt se en visualisering af den principper i klasse diagram.

Den kan også vise en rigtig godt overblik over en enkelte klasse og detaljerne (metoder og attributter i den) og hvad er endelig klassen laver for eksempel viser den klasse abstrakt eller ej, eller en metode er statik eller ej.

klasse diagram har forskellige forbindelser til hinanden :Dependency er sker når indikerer at en klasse er afhængig af en anden klasse når har en lokal variabel, bliver kaldt eller skabt, er parameter eller returværdi, eller er superklasse eller interface, den skaber den løseste kobling så men normalt bruger den i klasse diagram så meget, Association angiver at den anden klasse er attribut i den første klasse. Den er en stærkere kobling en dependency. Aggregation angiver at den anden klasse kan ændre nemt på attributter til første klasse (anden klasse har en set-konstruktør), den er stærkere kobling til association. Composition de vikker som aggregation men hvor den første klasse har inklusive adgang til den anden klasse (anden klasse har en copy-kostruktør), den er stærkere kobling end aggregation.

Vi har brugte klasse diagram til at viser vores struktur i kode og viser hvordan vores software klasse afhængig i hinanden, og vi viser også attributter og metoder til hver klasse.

Vi har i view lag vi har som har association med GUI klasse fordi GUI klasse bruger dem til at opfylde tekstfelt er som skal vise nogle bankens rente og RKI til hver kunde. Og alle vores view klasser skal bruge facade interface som association til at kan kontakter med logik lag. I logik lag vi har vores data klasser (kunde, bil, …) og vi har calculator som har alle matematik algoritmer som skal bruges til at regne rente ud. Alle vores data klasser har dependency med FFS- controller, og calculator har association med FFS-controller. Vores låntilbud klasse bruger kunde, bil og sælgeren klasse som attribut og har set -konstruktør til dem så de har Aggregation til låntilbud, og til sidst i data lag kan men se vores DatalyerImp klasse som alle metoder i forbindelser med database er der.

Vi opdateret klasse diagram mange gange gennem iterationer, og men kan se vores klasse diagram i bilæg??.

### Aktivitetsdiagram (Shahnaz Review Sofie)

Et billede, der indeholder skærmbillede

Beskrivelse, der er oprettet med meget høj sikkerhed

Figur 7 - Aktivitetsdiagramnotation

Når man vil visualisere en proces, kan man bruge aktivitetsdiagrammer. Man kan bruge dem til at visualisere alle dele i vores arbejdsproces.

For eksempel kan vi, mens vi stadig befinder os i begyndelsen af projektet, udarbejde aktivitetsdiagram ud fra en use case, for at lægge fokus på hvordan det nuværende system fungerer og hvordan vi kan udvikle det. Processen er meget kompleks, gør det det nemmere for os at danne overblik over arbejdsgangen.

Man bruger Action[[25]](#footnote-26) til at angive en handling, og Transition til at vise flowet mellem handlinger og/eller objekter. Start og End angiver aktivitetens begyndelse og slutning, Object Node angiver objekter eller data og benyttes til at vise data flowdiagram og Partition deler diagrammet mellem forskellige aktører.



Figur 8 - notation vedrørende parallelitet

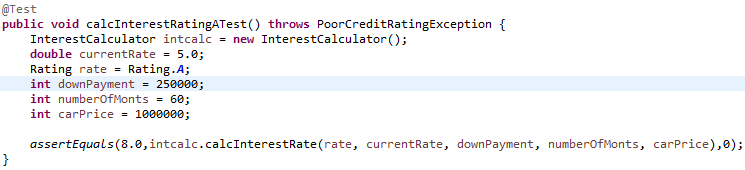
Men kan vise to eller flere aktiviteter som arbejder samtidig med Fork[[26]](#footnote-27) som opdeling af aktiviteter som afvikles samtidigt og Join dem alle når de færdig med arbejde. Ud over det kan man bruge Decision når man står i en situation hvor processen kan gå to veje. For eksempel viser vi i vores Aktivitets diagram, at hvis der ikke kommer svar fra RKI eller banken, skal bilsælger enten begynde forfra eller terminere processen[[27]](#footnote-28).

Aktivitetsdiagrammer er rigtig gode til at vise parallelle aktiviteter i forhold til Sekvensdiagram, derfor vi har brugt den til at vise vores aktiviteter i UC3[[28]](#footnote-29), der har tråde med som en del af casens udførelse. I forhold til vores UC3, AD (Aktivitetsdiagram)17 begynder med at bilsælgeren anmoder om et lånetilbud, og AD viser at på samme tid som UC2[[29]](#footnote-30) kommer med dagens rentesats og UC1[[30]](#footnote-31) sætter kreditværdighed til en kunde, angiver bilsælgeren lånets oplysninger til systemet. Når systemet får alle informationer som skal bruges, kan bilsælgeren bekræfte dem, hvorefter systemet til sidst vil gemme dem i databasen.

### Algoritme

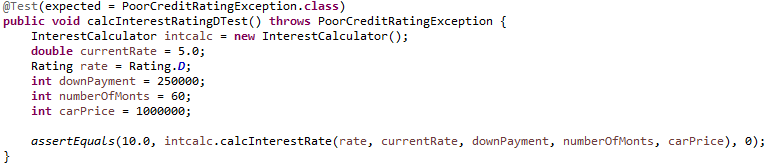
### Tests (Martin review Sofie)

Vi har i iteration E2 udvidet vores test suite yderligere. Både med flere unit-tests, men vi har også bevæget os et niveau op og laver nu integrationstest[[31]](#footnote-32). Integrationstest tester komponenter af systemet, og hvordan flere klasser kommunikerer med hinanden. I vores tilfælde er de tests, vi har lavet, designet til at sikre, at vores GUI-klasse får de korrekte informationer tilbage, når den beder om at få en rentesats beregnet. Vi startede med at lave unit-tests for vores renteberegnings metode.



Figur ?????????

I *figur ????????* ses et eksempel på, hvordan vi opbygger vores test cases. Metodenavnet indikerer, at det, der testes, er når rating er A. Dette ses bedre når man kigger på hele testklassen, og den næste så tester med en B rating, og alle andre variable uændret. De næste seks linjer er opbygning - her har vi brugt variabelnavne, som er så sigende som muligt for at skabe overblik, og for at gøre det lettere for andre at forstå vores kode uden for meget forklaring. Netop af samme grund har vi lavet disse variable i stedet for bare at sætte dem direkte ind i metoden. Sidste linje er vores assertEquals, som sammenligner vores forventede output med det output vi får ved det aktuelle input. Til sidst får assertEquals-metoden også delta som parameter, den maksimale forskel, der må være mellem resultaterne, for at de kan betragtes som værende ens. Forventet output er beregnet på baggrund af vores udleverede case.



Figur ??

Det er værd at notere, at der er yderligere kommentarer, som ikke er vist på vores lille kode screenshot. Da vi ikke vil oprette lån til folk som har fået den dårligste kreditvurdering, har vi lavet en form for sikkerhed i vores program, så når man prøver at oprette et lån til en kunde med denne kreditvurdering, bliver der kastet en af vores egne hjemmelavede exceptions[[32]](#footnote-33)**.** Dette er lidt interessant i forhold til vores j-unit test. Vi har måtte tilføje en ekstra linje[[33]](#footnote-34). Denne linje gør, at denne test kun vil gå igennem, hvis denne exception kastes; kastes den ikke, melder testen fejl, ligegyldigt om det, der står assertEquals ellers passer sammen.



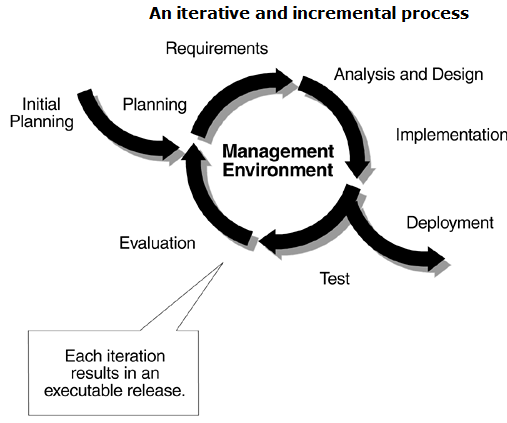
Figur ?!?

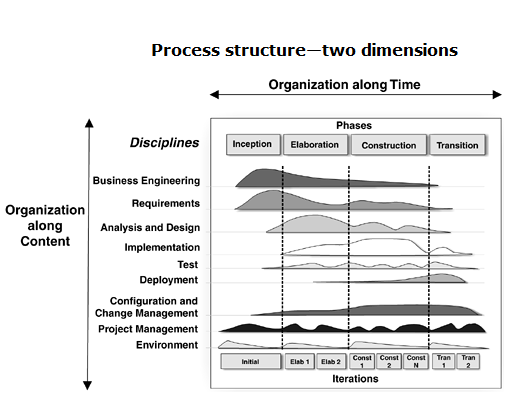
Da vi ikke laver metode kald direkte fra GUI til klassen som beregner renten, men går igennem vores facadecontroller, har vi lavet en test, som benytter sig af netop denne facadecontroller i stedet for beregnerklassen. Metoden, som bliver kaldt i facadekontrolleren, sender den således blot videre til beregner klassen; den er derfor praktisk talt magen til og tager samme parametre[[34]](#footnote-35).

### Exception (Martin)

Som et led i vores udregning af kundens personlige rente bruger vi en kreditvurdering og da vi ikke vil oprette lån til folk der er kendt som værende dårlige betalere, har vi brug for en form for sikkerheds foranstaltning i vores system så bilsælgere ikke ved fejl kommer til at oprette lån til disse kunder. Vi har derfor valgt at lave vores egen exception klasse. Vores exception tager en string som parameter , vi vil bruge denne string til at give en fejlmeddelelse med. Vi kunne godt have valgt ikke at give nogle parametre med da vi kun bruger den et specifikt sted, men vi at det er en god ide i forhold til at skulle udvide programmet at kunne sende beskeden med. Vi vil bruge vores exception til at vise fejlmeddelelsen til bilsælgeren i GUI'en så han ikke kan oprette lån til de forkerte personer.

### Rational Unified Process(Martin)

klassisk softwareudvikling har benyttet sig af den udviklings proces vi kalder for Vandfaldsmodellen. Når man udvikler software efter denne model går man først videre fra en disciplin når man er helt færdig med den, dvs. man bevæger søg først videre til design disciplinen når kravindsamling er 100% færdig, først til implementation når design er 100% færdig osv. Den primære fare ved dette er at vi skubber risici foran os, i stedet for at takle dem, dette gør at det ofte er dyrt at rette fejl fordi de bliver opdaget så sent i udviklingsprocessen. Rational Unified Process ,fremover forkortet UP, arbejder i stedet ikrementel og iterativt[[35]](#footnote-36). Det er derfor muligt at angribe og reagere tidligt på risici. UP gør det også muligt at få brugerfeedback tidligt i processen, dette er et rigtig godt værktøj til at afdække krav der ikke er lige umiddelbart synlige. Vi har også bedre mulighed for at dokumentere fremgang i projektet overfor kunden. i UP gør vi brug af mange visuelle modeller til at hjælpe udviklere med at forstå, specificere, bygge og dokumentere hvordan systemet er struktureret og opfører sig. Derfor er det vigtigt man benytter sig af et standard modellerings sprog, som f.eks. Unified Modelling Laungage (UML), så kommunikationen mellem udviklere bliver utvetydig og præcis. Mange af disse modeller kan man frit vælge om man vil benytte sig af eller ej, UP er derfor meget konfigurerbar, hvilket gør at vi kan benytte os af den i både store og små projekter.

UP er inddelt i 4 faser; Inception, Elaboration, Construction og Transition. Når man siger UP er iterativt er det fordi vi i hver fase kommer igennem hver eneste disciplin. Det er dog ikke alle faser der har lige meget fokus på de individuelle discipliner. I krav inception-fasen er der f.eks. meget lidt fokus på implementation, men til gengæld stor fokus på kravindsamling, og omvendt i Construction-fasen er der stor fokus på implementation og mindre fokus kravindsamling, men det er vigtigt at forstå at den stadig er der blot at der ikke bruges så meget tid på den. Billedet til højre viser hvordan et typisk UP drevet projekt kunne se ud. De forskellige faser afsluttes af milepæle. Hver milepæl har flere mål der skal være opfyldt for at man kan sige man har afsluttet en fase. Vi har i vores projekt startet med at lave et udkast til hvordan vores milepæle skulle se ud for hver enkelt fase.

**Inception - Lifecycle Objective:** De fleste UC's er identificeret, UC-1 er fuldt beskrevet, Domæne model påbegyndt, Visionsdokument godkendt, Iterationsplan for E1 er klar.

**Elaboration - Lifecycle Architecture:** Alle UC's er identificeret, UC1-3 er fuldt beskrevet, Arkitekturen er stabil og beskrevet, Eksekverbart produkt er udarbejdet

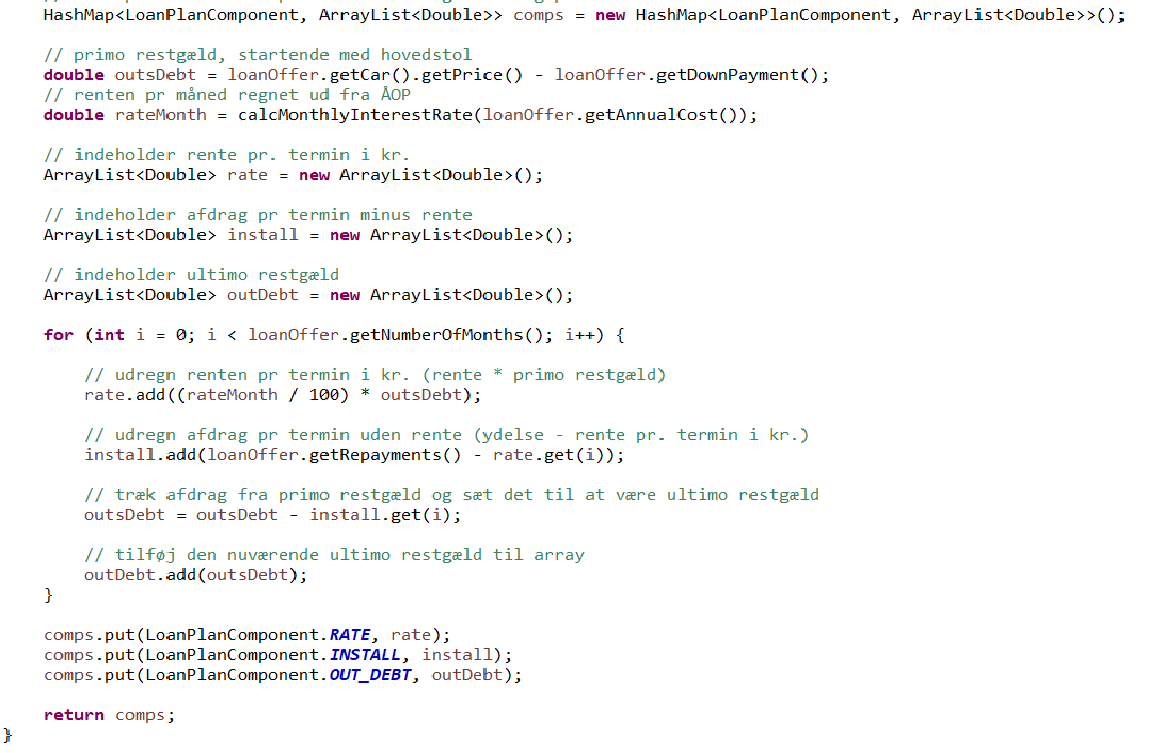
**Construction - Initial Operational Capability:** Systemet er fuldt implementeret (alle UC's) , Systemet består alle systemtests, Interessenter er parat til systemets udrulning

**Transition - Product Release:** Systemet er leveret og i drift, Systemet består accepttest, Brugerne er tilfredse.

Nogle af disse faser har også andre beslutninger og hensyn at tage inden de afsluttes. Når man når milepælen i Inception, er det et godt tidspunkt at tage en beslutning om at gå videre eller forkaste projektet[[36]](#footnote-37), det samme kan siges om milepælen for Elaboration, yderligere er det vigtigt at man kan afgøre hvor lang tid en Construction fase vil tage og man kan sætte præcise datoer for resten af projektet. Når Construction fasen er komplet står man med et projekt som er klar til at bliver overlevet til kunden. I Transition fasen er hovedspørgsmålet om brugeren er tilfreds, er kunden ikke tilfreds kan det være vi skal igennem en cyklus mere for at levere et tilfredsstillende produkt.

### CSV File(Shahnaz)

Til at vi kan print tilbagebetaling planen, skal vi implementere en csv fil(EXCEL file).I Calculator klasse har vi implementeret en metode som kan ud regne alle detaljerne til den tilbagebetaling plan den returnere en hash map, som en Enum Klasse som hedder LoanPlanComponent, der har key i den hash map og hver er dem har en arrayListe som har alle udregninger i, som der består af ultimo restgæld(outDebt), renten pr. måned, som regnet ud fra ÅOP, og afdrag pr. termin minus rente(install) (se i bilde??) og senere vi har brugte den til at vise de detaljer i csv file.



Figur ?? metode til at skab en csv file

Java.io-pakken indeholder klasser til systemindgang og -udgang via datastrømme, sterilisering og filsystemet, og praktiske metoder til rådighed i java.io-pakken.Vi har brugt writer og bufferwriter klasser til at saber en csv file, Der en lille forskel mellem en writer og en buffer writer, writer åbner file hver gange vi kaller på den og skrive noget i den men bufferwriter gemmer dem i en buffer og skriver dem på en gang, når vi kaller på metoden close og vil lukke filen.

# Konklusion

# Litteraturliste

# Bilag

## Bilag 1 – Iterations- og faseplan

## Bilag 2 – Visionsdokumentet

### Visionen

Systemet samler alle skridt der tages i forbindelse med afgivelse af lånetilbud ved køb af virksomhedens produkter(Ferrari). Systemet skal have et intuitivt brugerinterface som reagerer uden forsinkelse. Systemet bidrager til virksomhedens drift ved at effektivisere processen ved afgivelse af lånetilbud. Det skal kunne minimere tab af salg grundet bortkomne formularer, og kunne tilgås fra alle steder, af hensyn til salgschefens forretningsrejser.

### Interessentanalyse

Regional Ferrari-forhandler: Interesseret i at øge virksomhedens salg. Interesseret i muligheden for at godkende låneplaner uden at være fysisk tilstede i virksomheden.

Kunde: Kunden har interesse i at få en hurtig og korrekt udmelding om lån kan godkendes.

Bilsælger: Bilsælgeren har interesse i at vide om kunden allerede er registreret, for at undgå at der gives tilbud til samme kunde flere gange. Han har interesse i at processen bliver nemmere og hurtigere, da det gør det muligt at fokusere på kundekontakt og salg.

Kontorassistent: Kontorassistentens arbejdsbyrde lettes ved implementering af systemet.

Økonomimedarbejder: Økonomimedarbejderen har interesse i at processen bliver effektiviseret, da det gør hans arbejde lettere og hurtigere at udføre.

Salgschef: Salgschefen er interesseret i at øge virksomhedens salg ved en mere effektiv proces.

Banken: Banken er interesseret i virksomhedens finansielle situation, der forventes forbedret ved implementering af systemet. Banken øger også sit eget salg, når virksomhedens salg forøges.

### Feature-liste

Effektiv kommunikation med brugeren gennem en grafisk brugergrænseflade

Kreditværdighedstjek hos RKI

Indhentning af aktuel rentesats hos banken

Beregning af rentesats ud fra givne oplysninger

Registrering af et lånetilbud til en kunde

Eksport af et lånetilbud med en tilbagebetalingsplan

## Bilag 3 – Use case diagram, første udkast



## Bilag 4 – Use case 1

**FFS-UC1 - tjek kreditværdighed**

**Afgrænsning**: FFS

**Niveau**: Underfunnktion

**Primær aktør**: Bilsælger

**Interessenter og interesser:**

Kunde er interesseret i at få en god kreditvurdering.

Sælger skal kende kreditværdighed for at afgive tilbud.

Forretning og bank er interesseret i at kunden har en god kreditværdighed før de afgiver lånetilbud.

**Forudsætninger**: Kunden skal være oprettet og kunne slås op i systemet.

**Succesgaranti**: Sælgeren har fået en kreditvurdering tilbage for kunden.

**Hovedscenarie**:

1. Sælger angiver hvilken kunde han vil finde kreditværdigheden for.
2. Systemet præsenterer den pågældende kunde for sælgeren
3. Sælger anmoder systemet om at hente kreditvurdering for den pågældende kunde
4. Systemet indhenter kreditvurdering fra RKI.
5. Systemet præsenterer den pågældende kundes kreditværdighed til sælgeren.

**Varianter**:

\*a. systemet fejler på et givent tidspunkt

1. Sælgeren starter processen forfra
2. Sælgeren genstarter systemet

2a. Der kommer ikke noget svar / en fejl fra RKI

1. Sælger starter processen forfra

**Teknologier og dataformer:**

Systemet samarbejder med RKI’s API’er

**Ikke funktionelle krav:**

Brugerinterfacet må ikke blive påvirket af at der afventes svar fra RKI

**Hyppighed:** Ofte

## Bilag 5 – Use case 2

**FFS-UC2 – indhent aktuel rentesats**

**Afgrænsning**: FFS

**Niveau**: Underfunktion

**Primær aktør**: Bilsælger

**Interessenter og interesser:**

Kunden er interesseret i at få den korrekte rentesats oplyst.

Sælger er interesseret i at få en rentesats oplyst så han kan gennemføre salget.

**Forudsætninger**:

**Succesgaranti:** Den aktuelle rentesats er blevet korrekt oplyst til sælgeren

**Hovedscenarie**:

1. Sælger anmoder systemet om at oplyse bankens aktuelle rentesats.
2. Systemet indhenter rentesatsen fra banken.
3. Systemet præsenterer den aktuelle rentesats til sælgeren.

**Varianter**:

\*a. systemet fejler på et givent tidspunkt

1. Sælgeren starter processen forfra
2. Sælgeren genstarter systemet

2a. Der kommer ikke noget svar / en fejl fra banken

1. Sælger starter processen forfra

**Teknologier og dataformer:**

Systemet samarbejder med bankens API’er

**Ikke funktionelle krav:**

Brugerinterfacet må ikke blive påvirket af at der afventes svar fra banken

**Hyppighed:** Ofte

## Bilag 6 – Use case 3

**FFS-UC3 – Udregn lånetilbud**

**Primær aktør**: Bilsælger

**Interessenter og interesser:**

Kunden er interesseret i at få den korrekte rentesats oplyst.

Bilsælger er interesseret i at få en rentesats oplyst så han kan gennemføre salget.

**Forudsætninger**:

Kunden er slået op i databasen.

Kunden har ikke et aktivt lånetilbud.

**Succesgaranti:** Lånetilbuddet er blevet korrekt udregnet og gemt i databasen.

**Hovedscenarie**:

1. Bilsælger anmoder systemet om at udregne et lånetilbud.
2. Systemet starter udførelsen af FFS-UC1 og FFS-UC2 asynkront.
3. Bilsælger angiver bil der skal udregnes lånetilbud for.
4. Systemet præsenterer pris for bilsælgeren.
5. Bilsælger angiver udbetaling.
6. Bilsælger angiver start dato for lånet.
7. Bilsælger angiver antal måneder lånet skal løbe over.
8. Bilsælger anmoder systemet om at beregne lånetilbuddet.
9. Systemet beregner kundens rentesats.
10. Systemet præsenterer alle lånetilbuddets detaljer for bilsælgeren.
11. Bilsælgeren bekræfter oplysningerne.
12. Systemet gemmer lånetilbuddet i databasen.

**Varianter**:

\*a. systemet fejler på et givent tidspunkt

1. Bilsælgeren starter processen forfra
2. Bilsælgeren genstarter systemet

8a. FFS-UC1 og FFS-UC2 er ikke færdige med udførelse inden bilsælger anmoder om beregning

1. Bilsælgeren venter til FFS-UC1 og FFS-UC2 er færdige med udførelse.

8b. FFS-UC1 og/eller FFS-UC2 får ikke svar tilbage fra RKI eller bank.

1. Bilsælgeren afbryder processen og prøver igen.

1a. Bilsælgeren kontakter RKI eller Bank med henblik på fejlfinding

1b. Bilsælgeren tilkalder support for lokale fejl

**Teknologier og dataformer:**

Systemet samarbejder med RKI og bankens API’er

**Ikke funktionelle krav:**

Brugerinterfacet må ikke blive påvirket af at der afventes svar fra RKI eller bank

**Hyppighed:** Ofte

## Bilag 7 – Domænemodel



## Et billede, der indeholder tekst, whiteboard Beskrivelse, der er oprettet med meget høj sikkerhedBilag 8 - Mock-ups

Slå en kunde op i databasen

Et billede, der indeholder tekst, whiteboard

Beskrivelse, der er oprettet med meget høj sikkerhed

Vælg kunden der skal oprettes en lånetilbud for

Angiv oplysninger for et nyt lånetilbud:

Et billede, der indeholder tekst, whiteboard

Beskrivelse, der er oprettet med meget høj sikkerhed

Bekræft oplysninger som bil sælger gemmer i databasen

## Bilag 9 – Datamodel



## Bilag 10 - Risikoanalyse

* Vores viden er ikke tilstrækkelig til at vi kan udføre opgaven. Lav risiko, overvåg (monitor)

I modekommersen:

Vi har vejlederen, så vi kan bruger dem til at imod kommer den risiko.

* Sygdom eller lignende medfører højt mandefald, da der ikke er mange der arbejder på projektet. Lav risiko, overvåg (monitor)

I modekommersen:

vi kan ikke lave noget ved sygdom så bare lader den være.

* Opgaven kan ikke færdiggøres før deadline. Medium, begræns sandsynlighed (forebyggelse, mitigation), planlæg med risikoen i tanke (management)

I modekommersen:

Med en god planlægning kan godt forhindre den.

## Bilag 11 - Testsuite



## Bilag 12 - Klassediagram

1. Se Bilag 2 - Visionsdokumentet [↑](#footnote-ref-2)
2. https://www.ibm.com/developerworks/rational/library/3975.html [↑](#footnote-ref-3)
3. Se *figur 1*. [↑](#footnote-ref-4)
4. Se den endelig domænemodel i bilag 4 [↑](#footnote-ref-5)
5. Se illustration 1 - domænemodelnotation [↑](#footnote-ref-6)
6. Se afsnittet Use cases for flere detaljer. [↑](#footnote-ref-7)
7. I modsætning til Abstract use cases, der ses som en underfunktion til andre use cases og ikke kan stå alene. Dette vil dog blive revideret senere. [↑](#footnote-ref-8)
8. Se bilag 6. [↑](#footnote-ref-9)
9. Se bilag 4. [↑](#footnote-ref-10)
10. Se bilag 5. [↑](#footnote-ref-11)
11. Se *figur 5*. [↑](#footnote-ref-12)
12. Larman, Applying UML and Patterns 3rd edition. Kapitel 11 [↑](#footnote-ref-13)
13. Fremover forkortes til OC [↑](#footnote-ref-14)
14. Fremover kaldt OC1 [↑](#footnote-ref-15)
15. Fremover kaldt OC2 [↑](#footnote-ref-16)
16. Se Bilag ?????? og Bilag ???????? [↑](#footnote-ref-17)
17. Se Bilag ??????? [↑](#footnote-ref-18)
18. Læs om singleton-mønsteret under afsnittet ”Singleton pattern” side ?????????? [↑](#footnote-ref-19)
19. Se klassediagrammet, bilag ????????????? [↑](#footnote-ref-20)
20. Med andre ord: en dårlig løsning på et designproblem. [↑](#footnote-ref-21)
21. Se *figur 9 - Observer pattern - diagram over implementation*. Kun de af Javas metoder, som vi har benyttet, er repræsenteret i diagrammet. [↑](#footnote-ref-22)
22. The Rational Unified Process: An Introduktion, Third Edition af Phillipe Kruchten, Addison Wesley 2003 – Kapitel 12 [↑](#footnote-ref-23)
23. Se bilag 10 [↑](#footnote-ref-24)
24. Se endelig datamodel i Bilag?????????? [↑](#footnote-ref-25)
25. Se al notation i *figur 7* [↑](#footnote-ref-26)
26. Se al notation vedrørende parallelitet i *figur 8* [↑](#footnote-ref-27)
27. Se aktivitetsdiagrammet, bilag ??????????????? [↑](#footnote-ref-28)
28. Se bilag?????????? [↑](#footnote-ref-29)
29. Se bilag ??????????? [↑](#footnote-ref-30)
30. Se bilag ??????????? [↑](#footnote-ref-31)
31. The Rational Unified Process: An Introduktion, Third Edition af Phillipe Kruchten, Addison Wesley 2003 – Kapitel 12 [↑](#footnote-ref-32)
32. Se afsnit om Exceptions side ????????????? [↑](#footnote-ref-33)
33. Se *figur ??* [↑](#footnote-ref-34)
34. Se *figur ?!?* [↑](#footnote-ref-35)
35. The Rational Unified Process: An Introduktion, Third Edition af Phillipe Kruchten, Addison Wesley 2003 – Kapitel 1. [↑](#footnote-ref-36)
36. The Rational Unified Process: An Introduktion, Third Edition af Phillipe Kruchten, Addison Wesley 2003 – Kapitel 4. [↑](#footnote-ref-37)