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: 85.866 (ca. 39 normalsider)

Indhold

Indledning

Den regionale Ferrariforhandler har i næsten 70 år solgt dyre luksusbiler. Da det ikke er alle der har råd til disse dyre biler, består en del af deres forretningsmodel i at tilbyde finansiering til disse biler. Konkurrencen er skarp, da der er mange andre firmaer, der også tilbyder finansiering; firmaets proces til at indhente disse finansieringstilbud har ikke ændret sig siden firmaets opstart - og det kan mærkes. Der bliver tabt kunder til andre tilbud pga. af den langsommelige proces. Det vil firmaet gøre noget ved og vil gerne modernisere deres proces med et nyt IT-system.

For at kunne levere et godt system som Ferrari forhandleren vil være tilfreds med, har vi lavet en IT-forundersøgelse; denne er blevet brugt til at skabe en vision og en række krav. Rational Unified Process er blevet benyttet for at sikre en stabil og veldokumenteret arbejdsgang, hvor kunden også kan se løbende fremskridt. Visionen og kravene er blevet til use-cases, som er den primære ressource vi har brugt til at fremdrive projektet. Der er løbende blevet udarbejdet mange andre artefakter som f.eks. klassediagram, som har lagt ud med at være små, men som stabilt har vokset sig store gennem projektet. Andre artefakter som sekvensdiagrammer og aktivitetsdiagrammer har hjulpet med at holde overblik når vi skrev kode. Programmeringen er understøttet af veludvalgte designmønstre, mens vi har opholdt en striks lagdeling for at sikre eventuel fremtidig udvidelse til andre platforme.

Problemstilling

Ferrariforhandleren har naturligvis en række krav til systemet, de gerne vil have udarbejdet. Brugergrænsefladen skal være letforståelig og intuitiv, så både nye og gamle medarbejdere kan benytte systemet uden for meget oplæring. Brugergrænsefladen skal forblive responsiv, også når der foretages kald til diverse API'er. Oplysninger om kunder, sælgere biler og låneaftaler, skal gemmes i en database, og denne skal naturligvis overholde den nye persondatalov. Der skal være plads til, at systemet kan udvides til en webløsning på sigt. Der skal kunne eksporteres en fil på CSV formatet, som indeholder tilbagebetalingsplanen for et givet lån. Funktionen til fastsættelse af rentesats skal være af særlig høj kvalitet, fordi fejl i denne funktion kan medføre væsentlige omkostninger - enten i form af tabt forretning eller øget risiko.

It-forundersøgelse (Martin)

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 referencelinje 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[2] i inception-fasen arbejder vi netop ud fra firmaets nuværende problemstilling, hele formålet 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 dokumentanalyse 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[3] og et supplerende kravspecifikationsdokument[4]. 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 (Martin)

En af de første ting vi gjorde i projektet, var at opsætte vores udviklingsmiljø. 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 nyt repository til dette projekt. Vi har derefter oprettet branches, så vi alle har haft vores egen til at udvikle i,

[1] Professional it-forundersøgelse - grundlaget for brugerdrevet innovation af Keld Bødker, Finn Kensing & Jesper Simonsen, 2008 – kap 9.1
[2] Fremover forkortet UP
[3] Se Bilag 2
[4] Læs mere om krav under afsnittet Supplerende Kravspecifikation – side 8

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 ligget i vores Github repository.

Iterations- og faseplan (Sofie)

Som en del af project management har vi udarbejdet en iterations- og faseplan[5]. 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. I den faseplan vi har medtaget her i rapporten er construction og transition slet ikke nævnt, da vi, efterhånden som prpjektet skred frem, kunne se, at den ikke ville blive helt stabil i tide til at vi kunne overgå i construction[6].

Estimering (Martin)

Da vi startede projektet, var vi enige om, at vi ville prøve at bruge 3 punkts estimering til at finde en estimeret tid på alle de forskellige artefakter, vi kom til at lave. 3 punkts estimering er blot hvor vi finder tre forskellige estimater; Den absolut hurtigste tid vi mener vi kan klare opgaven på, den absolut langsomste tid vi mener vi kan klare opgaven på, og den mest sandsynlige tid vi mener vi kan klare opgaven på. Disse tider ligges så sammen og divideres med tre, dermed har vi en gen-

[5] Se Bilag 1
[6] Dette bliver uddybet senere

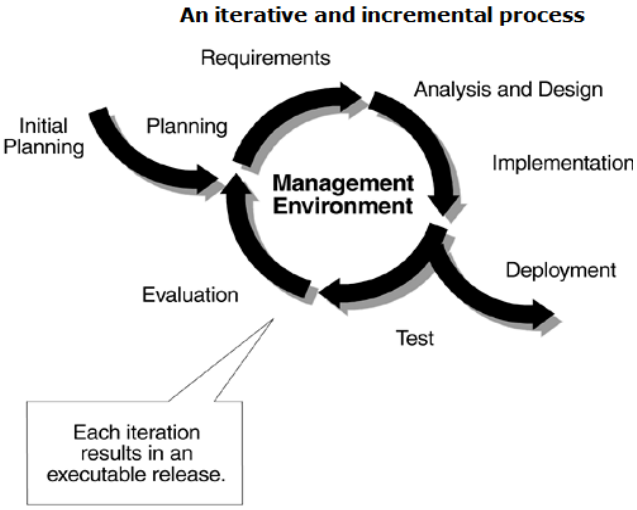
nemsnitstid. I praksis må vi erkende at dette blev tilsidesat, vi fandt det meget svært at give nogen meningsfulde gæt på hvor lang tid de forskellige opgaver ville tage. Vi besluttede i stedet at holde en log over hvor meget tid vi brugte på hver enkel opgave. Vi startede med at sige en iteration om ugen, men dette holdte slet ikke og vi var ude af første iteration og inception-fasen på fjerdedagen. Derfra planlagde vi ca. 5 arbejdsdage pr iteration. Vi begyndte også at starte hver dag med at lægge en slagplan for hvad vi skulle nå den pågældende dag for at holde os inden for tidsgrænsen. Dette viste sig at være en god måde for os at arbejde på, da det gjorde at vi nemt kunne holde overblik over hver dag og planlægge i små bidder hvad vi havde brug for til at nå de forskellige iterationer inden for tidsplanen. Vores mangelfulde estimering er også én af grundene til, at vi ikke er overgået til construction[7].

Review (Martin)

For at sikre et højt niveau på vores artefakter har vi benyttet os af at reviewe dem inden vi bruger dem til at arbejde videre i processen. Vi startede originalt med at aftale at lave reviews med en anden gruppe, ideen her var at vi kunne få et friskt perspektiv på vores aktiviteter. Det viste sig dog at være for ineffektivt, at vi skulle forstyrre hinandens arbejdsproces, hver gang vi fik udarbejdet et artefakt, hvilket skete ofte i de første iterationer, da vi havde stor fokus på design og kravindsamling. Vi gik derfor over til en intern form for review. En person fik ansvar for at udarbejde en artefakt, hvorefter én, eller hele gruppen for store artefakter, fik ansvar for at godkende artefaktet. Denne metode har fungeret ret godt og blev hurtigt bare en del af vores proces vi ikke tænkte videre over. Det skete selvfølgelig ofte at vi stadig måtte ind og lave ændringer i vores artefakter, men det mener vi mere er tilskrevet manglende erfaring end dårlige artefakter. Vi fik desuden nogle af vores artefakter reviewet af vores vejledere, i tilfælde af at vi efter review internt stadig havde tvivl om noget.

[7] Dette vil blive uddybet senere.

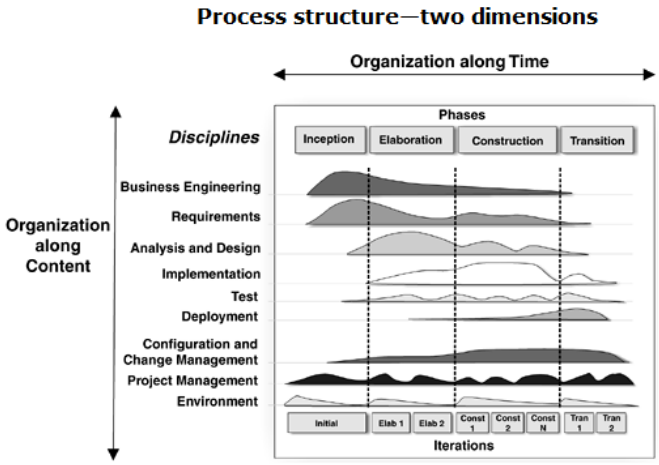
Rational Unified Process (Martin)



Klassisk softwareudvikling har benyttet sig af den udviklingsproces, 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 sig 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 tackle dem. Dette gør, at det ofte er dyrt at rette fejl, fordi de bliver opdaget så sent i udviklingsprocessen. UP arbejder i stedet inkrementel og iterativt[8]. Det er derfor muligt at angribe og reagere tidligt på risici. UP gør det også muligt at få brugerfeedback tidligt i - og flere gange i løbet af - 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. Det er især brugbart i forbindelse med større projekter hvor der er flere teams involveret, og man derfor har behov for en universel måde at kommunikere designkoncepter til hinanden på. 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. Vi bruger også UML's standardnotation i de fleste tilfælde i vores modeller, men dette vil blive uddybet i de forskellige kommende afsnit. Mange af disse modeller kan man frit vælge om man vil benytte sig af eller ej. UP er derfor meget konfigurerbar og fleksibel, hvilket gør, at vi kan benytte os af den i både store og små projekter.

[8] The Rational Unified Process: An Introduktion, Third Edition af Phillipe Kruchten, Addison Wesley 2003 – Kapitel 1.

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. megetbegrænset fokus på implementation, men til gengæld stor fokus på kravindsamling, og omvendt i Construction-fasen er der stor fokus på implementation og mindre fokus på kravindsamling. Det er dog vigtigt at forstå, at den stadig er der, blot at der ikke bruges så meget tid på den. Billedet ovenfor 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 UCs er identificeret, UC-1 er fuldt beskrevet, Domæne model påbegyndt, Visionsdokument godkendt, Iterationsplan for E1 er klar.
- **Elaboration - Lifecycle Architecture:**
  Alle UCs 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 UCs), 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[9]. 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 blive 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.

Fase 1: Inception (Sofie)

I inception-fasen lægger det primære fokus på indledende kravspecifikation. Det vil sige, at vi prøver at afgrænse projektet ved at udarbejde visionsdokument og identificere de mest centrale use cases. Vi bruger også tid på at analysere de risici der er forbundet med projektet og beregne omkostninger, med henblik på at vi kan vurdere, om projektet er værd at gå videre med. I vores tilfælde indeholder inception-fasen kun én iteration, som vi kalder I1[10].

Iteration I1 (Sofie)

I løbet af den første iteration vil vi få udarbejdet visionsdokumentet, identificere de centrale use-cases, udarbejde en formel beskrivelse af de to første use cases og udarbejde mock-ups. Desuden påbegynder vi vores iterationsplan, risikoanalyse, og domænemodel, hvilket er centrale aktiviteter i inception-fasen, da de vil hjælpe os til at vurdere, om projektet er værd at tage op og gå videre med. Desuden påbegynder vi et vagt udkast til et klassediagram, der vil give os overblik over den arkitektur vi vil bruge.

Visionsdokument (Sofie)

Vi har valgt at begynde med at udarbejde visionsdokumentet[11] 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 løsningsmu-

[9] The Rational Unified Process: An Introduktion, Third Edition af Phillipe Kruchten, Addison Wesley 2003 – Kapitel 4.
[10] Se vores milepæl for inception-fasen under afsnittet ”Rational Unified Process” – side 4
[11] Se Bilag 2 - Visionsdokumentet

lighederne for meget. Visionsdokumentet består af en beskrivelse af visionen, en interessentanalyse og en feature-liste.

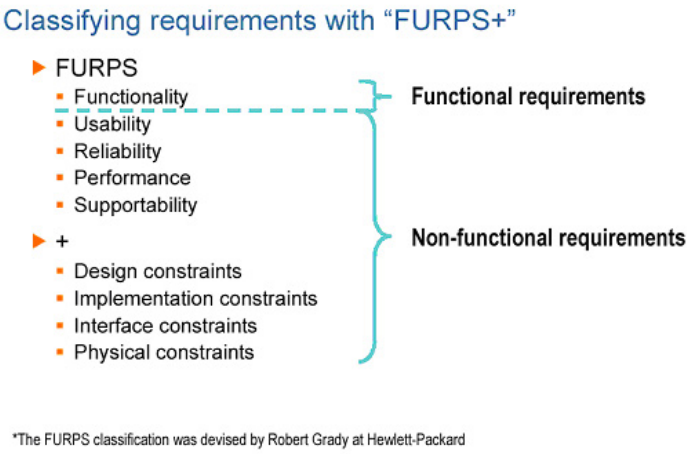
Visionen er en kortfattet fremtidsbeskrivelse, der formuleres på en måde der ikke specificerer teknologivalget, med mindre kunden har nogen helt specifikke krav hertil. Den sammenligner ikke det nye system med noget tidligere, men har fokus på fremtiden og løsningen af den nuværende problemstilling. 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 den 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 det 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)

Den supplerende kravspecifikation udarbejdes efter visionsdokumentet og sideløbende med use case-diagrammet[12]. Det er vigtigt at understrege dette kun er, som navnet netop indikerer, 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+[13] til at kategorisere de krav vi mener bør være i den supplerende kravspecifikation. Når alle kravene er samlet, bør de gøres målbare.



Figur 1 – FURPS+

Risiko Analyse(Shahnaz)

Ifølge ordbogen er en risiko muligheden er et negativt resultat, muligheden for skade, tab eller lignende[14]. I forbindelse med systemudvikling løber vi også ind i risici, som vi bør bruge noget energi på at identificere og vurdere tidligt i projektet. Jo tidligere risikoen bliver identificeret, jo bedre vil vi også være klædt på til at imødekomme den. Der skal kun medtages relevante risikoer fordi det øvrige forstyrrer analysen. Det vil sige, at vi ikke medtager risici, der er uden for vores kontrol. Hver identificeret risiko skal være afgørelig i en sådan grad, at kan man finde en løsning til den. Risikoen skal også have en sandsynlighed så vi kan determinere hvor vigtigt det er at finde løsning til den eller bare forhindre den før vi begynder at arbejde på systemet.

Vi har brugt risikoanalyse[15] til at finde ud at hvordan vi kan forhindre risici i projekten før vi begynder at arbejder med det. Men vi kunne ikke vurdere meget tal i den eller beregne de økonomiske risici da vi ikke kender til, hvad den reelle pris på et sådant system ville være. Vi har dog for-

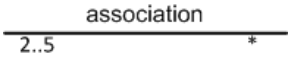
[12] IBM - https://www.ibm.com/developerworks/rational/library/3975.html
[13] Se *figur 1*.
[14] Den Danske Ordbog - https://ordnet.dk/ddo/ordbog?query=risiko
[15] Se bilag 10

søgt at angive sandsynligheden for de forskellige risici, vi har identificeret, hvilket hjælper os til at prioritere hvilke der skal imødekommes først, og hvordan.

Vi har brugt forskellige typer af proaktiv risikostyring, for eksempel monitoring, mitigation og management. Monitoring vil sige, at vi overvåger risikoen, og hvis vi kan se, at den bliver mere sandsynlig, vil vi gøre noget for at imødekomme den. Mitigation vil sige, at vi forsøger at begrænse konsekvensen af, at risikoen bliver realiseret. Vi har også brugt management, idet vi har forsøgt hele tiden at have de forskellige identificerede risici i tanke i vores projektplanlægning.

Domænemodel (Sofie)

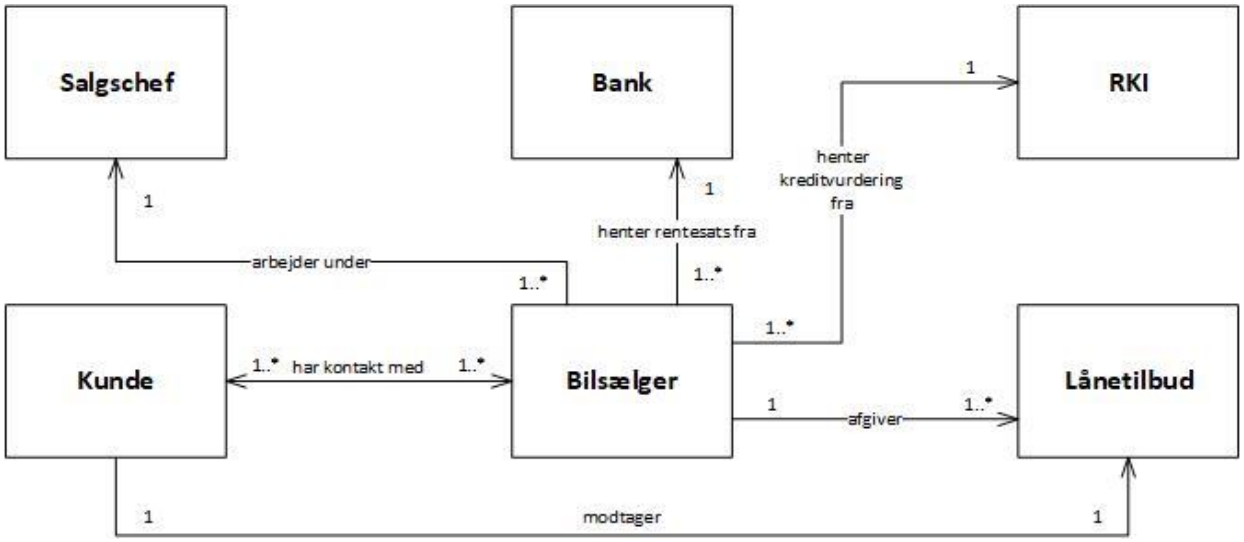
I den første iteration har vi valgt at udarbejde et første udkast til domænemodellen[16]. Domænemodellen er en måde at visualisere og analysere problemdomænet på et forholdsvis tidligt stadie i projektet. Modellen viser relationerne mellem forskellige koncepter i problemdomænet, og vi udarbejder den 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[17]. 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 2 - Domænemodelnotation

Vores første udkast til en domænemodel ses i *Figur 3*[18]. 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.

[16] Se den endelig domænemodel i bilag 4
[17] Se *figur 2 - domænemodelnotation*
[18] Se den endelige domænemodel i bilag 7



Figur 3 – Domænemodel

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.

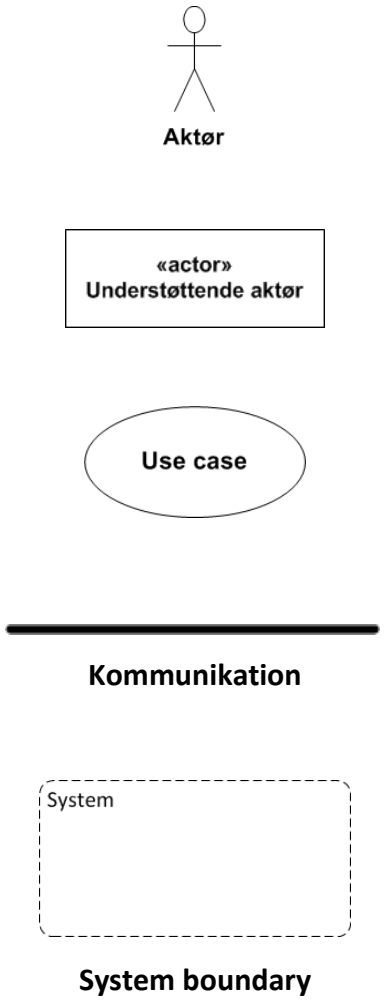
Associationspilen mellem kunden og sælgeren går begge veje, da de har kontakt med hinanden og begge kan tage initiativ til denne kontakt. En sælger kan traditionelt sagtens have kontakt med flere forskellige kunder. Kunden kan også have kontakt med flere sælgere, men vi forstår ud fra problemformuleringen, 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.

I den endelige domænemodel er der desuden en association tilføjet mellem salgschef og lånetilbud. Dette skyldes, at vi efter et ekstra møde med vores kunde fik specificeret, at vores system også skal tage hensyn til, at et lånetilbud skal kunne godkendes af salgschefen, i tilfælde af, at bilsælgeren, der opretter tilbuddet, ikke har ret til at godkende tilbuddet på baggrund af bilens pris. Dette har vi dog ikke vidst da vi udarbejdet det første udkast, hvorfor vi var nødt til at opdatere vores domænemodel. Det var imidlertid vigtigt at gøre, da det havde en indvirkning på det endelige antal af use cases vi fik identificeret.

Use case diagram (Sofie)

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”.



Figur 4 – Use case-diagramnotation

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 nogle 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[19]. De er blot nødvendige i forbindelse med identifikationen af use cases.

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*[20].

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 use case-diagrammet er repræsenteret som Concrete use cases, idet de startes af en aktør og realiserer aktørens ønskede mål med casen[21].

Use cases (Martin)

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 in-

[19] Se afsnittet Use cases for flere detaljer – side 13
[20] i Bilag 3 ses vores endelige use case-diagram.
[21] I modsætning til Abstract use cases, der ses som en underfunktion til andre use cases og ikke kan stå alene.

volveret 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”[22], 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”[23], blev så til en underfunktion af UC3. Det samme gør sig gældende for ”FFS-UC2 - Indhent aktuel rentesats”[24]. 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

[22] Se bilag 4.
[23] Se bilag 5.
[24] Se bilag 6.

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)

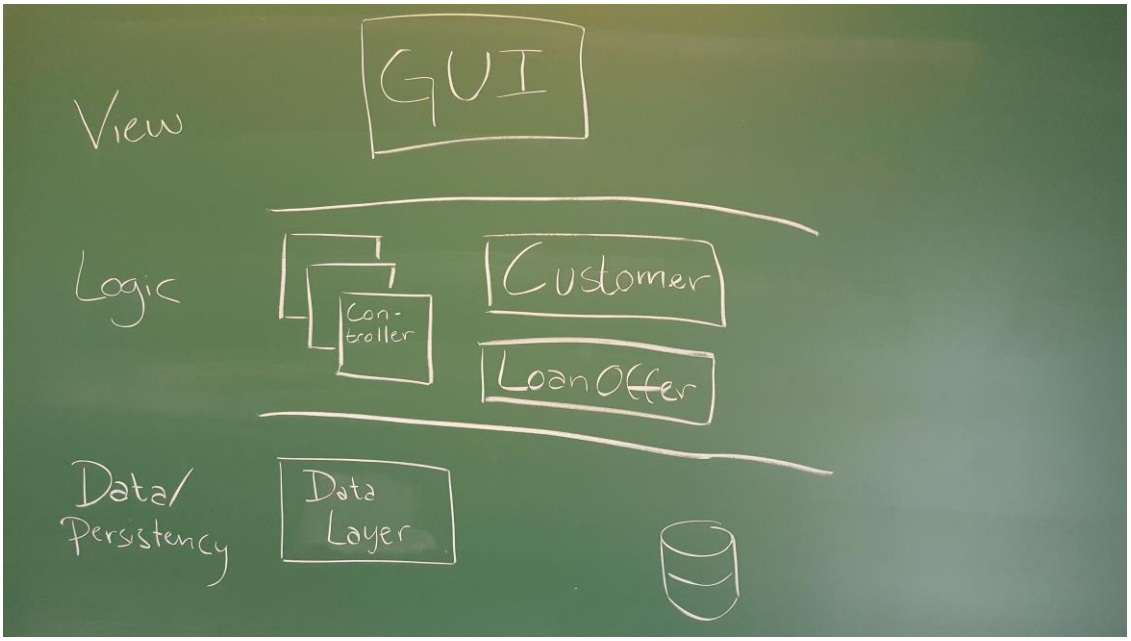
Trelagsarkitektur er en model til beskrivelse af de blokke et program er opbygget af. Modellen anvendes blandt andet fordi det gør det nemmere at overskue kommunikationen mellem lagene og kommunikationen med brugerne og med andre IT-systemer. Vi bruger trelagsarkitektur til at holde struktur i vores 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[25] er lagene defineret på følgende måde:

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.

[25] Se *figur 5*.



Figur 5 – trelagsarkitekturen som vi tegner den

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.

Mock-ups (Shahnaz)

Mock-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 projektet 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 use case 1-3 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.

Konklusion på inception-fasen (Sofie)

Efter udarbejdelsen af alle de ovennævnte artefakter, er vi ved at være klar til at vurdere, om projektet er værd at gå videre med. Det har været muligt for os at få overblik over en fordelagtig arki-

tektur - trelagsarkitekturen. Det er en arkitektur vi allerede kender til, og derfor har vi et nogenlunde overblik over projektet. De risici, vi har analyseret os frem til[26], udgør en meget begrænset fare for os, og vi vurderer, at de kan imødekommes i tilfælde af at de skulle blive aktuelle, især i kraft af vores vejledere. Vi har desuden fået vores mock-ups godkendt, hvilket fortæller os, at vi er på rette spor, og giver os en motivation for at gå videre med projektet. Vi kan nu trygt gå videre til elaboration-fasen.

Fase 2: Elaboration (Sofie)

I elaboration-fasen lægger vi mere fokus på at få uddybet de krav, vi har identificeret i den foregående fase. Det vil sige, at vi får flere af vores use cases formelt beskrevet og vi udarbejder operationskontrakter og systemsekvensdiagrammer, der giver overblik over brugerens interaktion med systemet. På baggrund af disse artefakter, kan vi udarbejde mere kodenære modeller som for eksempel sekvens- og klassediagrammer, hvilket vi derefter kan implementere de beskrevne use cases ud fra. Vi starter med de use cases der er størst risiko forbundet med. Elaboration-fasen skal give os et dybere indblik i projektet og eliminere alle risici, så vi kan gå i construction-fasen.

Iteration E1 (Sofie)

I denne første iteration af elaboration-fasen vil vi udarbejde udkast til klassediagram samt systemsekvendiagrammer, hvilket lægger et grundlag til at vi lige så stille kan begynde at implementere nogle af vores use-cases. Vi vil desuden i denne iteration have meget fokus på vores datamodel og normalisering i forbindelse dermed. Vi ønsker at implementere de 2 første use cases, og påbegynde j-unit tests for at sikre os, at vores program bliver systematisk testet. Desuden vil vi begynde at se på forskellige patterns, der kan bruges i forbindelse med implementationen af vores use cases.

Klassediagram (Shahnaz)

Klassediagram, der er et design diagram, bruges som visualisering af et statisk perspektiv til kode og implementering af software. Det kan med fordel benyttes til at visualisere relationerne mellem softwareklasser, og den har en statiske fokus på programs statiske struktur, den kan også identifi-

[26] Se risikoanalysen bilag 10

cere afhængigheder mellem klasserne, når man bruger GRASP principper[27] kan man se en visualisering af de principper i et klassediagram.

Det kan også vise et rigtig godt overblik over den enkelte klasse og detaljerne (metoder og attributter i den) og hvad klassen egentlig laver. For eksempel viser det om en klasse er abstrakt eller ej, eller en metode er static eller ej.

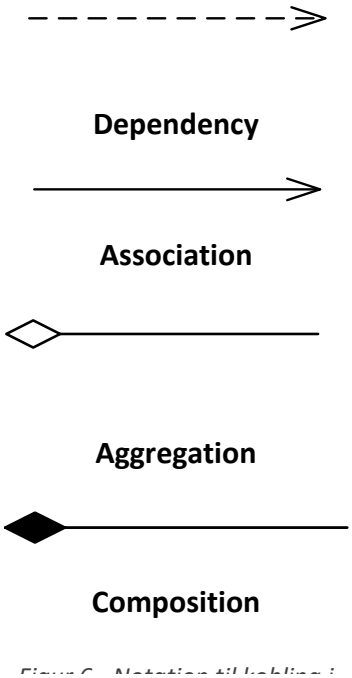
Klassediagram har forskellige måder at illustrere relationer mellem klasser: **Dependency** benyttes når man ønsker at indikere at en klasse er afhængig af en anden klasse når den har en lokal variabel, bliver kaldt eller skabt, er parameter eller returværdi, eller er superklasse eller interface, den skaber den løseste kobling så man bruger den normalt i klassediagram en hel del.

Association[28] angiver, at den anden klasse er attribut i den første klasse. Den er en stærkere kobling end dependency.

Aggregation angiver at den anden klasse kan ændre nemt på attributter til første klasse (anden klasse har en set-konstruktor). Det angiver, at de to klasser indgår som en del af et whole/part-forhold til hinanden. Den er stærkere kobling end association.

Composition de vikker som aggregation men hvor den første klasse har eksklusiv adgang til den anden klasse (anden klasse har en copy-kostruktor), den er stærkere kobling end aggregation, men forveksles ofte.

Vi har brugte klasse diagram til at vise vores struktur i koden og vise hvordan vores softwareklasser er afhængige af hinanden, og vi viser også attributter og metoder til hver klasse[29].



Figur 6 - Notation til kobling i

I view-laget har vi placeret vores GUI-klasse, som har ansvaret for at modtage input og præsentere output til brugeren. Denne klasse har association ned til logik-laget, da det er logikken der

[27] Se afsnittet "GRASP" – side 30
[28] Se notation i *figur 6 - notation til kobling i klassediagram*
[29] Se det endelige klassediagram i Bilag 16

bestemmer, hvad der skal præsenteres for brugeren. På nuværende tidspunkt har vi kun én klasse i view-laget, men dette bliver opdateret senere, som det kan læses i det endelige klassediagram.

Alle vores view klasser skal bruge facade interface som association til at få kontakt med logik lag[30]. I logik lag vi har vores dataklasser (kunde, bil, …) og vi har calculator som har alle matematiske algoritmer[31] som skal bruges til at regne lånetilbud ud. Alle vores dataklasser har dependency med FFS-controller, og calculator har association med FFS-controller. Det skyldes, at det er controlleren, der har ansvaret for at kommunikere kommandoer videre fra view-laget til logikken. Vores lånetilbud klasse bruger de klasser der repræsenterer kunde, bil og sælgeren som attribut og har set -konstruktør til dem så de har Aggregation til lånetilbud, da der er tale om et whole/part forhold. Til sidst i data lag kan men se vores Datalayer klasse, som står for al kommunikation med databasen. Der er således også association mellem datalayer og controller, da controlleren sender opgaver videre til datalayer ved hjælp af en instansvariabel.

Vi har opdateret klassediagrammet flere gange i løbet af projektet, og derfor vil der være flere afsnit om disse opdateringer senere.

Systemsekvensdiagrammer (Shahnaz)

Systemsekvensdiagrammer viser hvordan en bruger kommunikerer med systemet i forhold til et konkret use case scenarie. 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 hvordan vores system kommunikerer med banken til at hente renteset og RKI til at vurdere kreditværdighed til en kunde og den 3. SSD handler om hvordan bilsælger som er primære aktør i vores system kommunikerer med systemet giver det nogle oplysninger og får udregnet lånetilbuddet til kunden og til sidste bekræfter informationer til[32].

[30] Læs mere om façade-mønsteret under afsnittet ”Facade pattern” – side 27
[31] Læs mere om algoritme under afsnittet ”Algoritme” – side 35
[32] Se vores systemsekvensdiagrammer i bilag 15.

Operationskontrakter (Martin Review Shahnaz)

Operationskontrakter[33] bruges i UP når vi føler vi har brug for en mere detaljeret beskrivelse af hvordan systemet opfører sig. Der er fokus på de ændringer der bliver foretaget. Operationskontrakter[34] 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[35] er navngivet, FFS-OC1[36] og FFS-OC2[37], 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, der skal være opfyldt, som i tilfældet med 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 kan systemudvikleren forstå forretningens sprog og forstå bedre hvad handler systemet om. Den beskriver en general beskrivelse om en koncept og giver også konkrete eksempler på det. Vi har udarbejdet en dataordbog[38], der indeholder nogle begreber fra problemformuleringen, vi havde behov for at definere, før vi med sikkerhed kunne gå vide-

[33] Larman, Applying UML and Patterns 3rd edition. Kapitel 11
[34] Fremover forkortes til OC
[35] Se bilag 13
[36] Fremover kaldt OC1
[37] Fremover kaldt OC2
[38] Se bilag 18

re i projektet. Mange begræber var i forvejen godt beskrevet i den problemformulering vi fik udleveret, og derfor havde vi ikke behov for en særligt omfattende dataordbog.

Data Model (Shahnaz)

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.

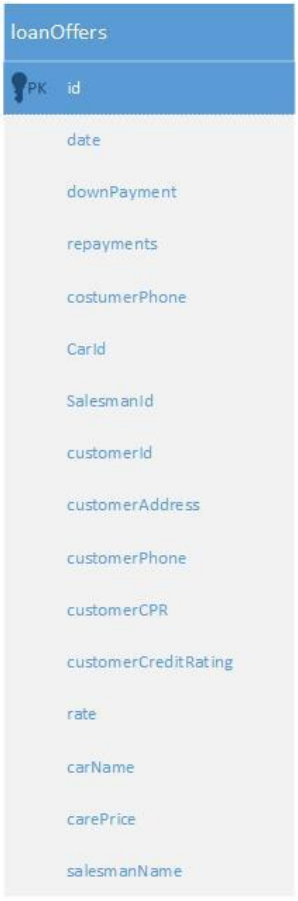
Når man analyserer præcise hvilken koncepter man har i sit system, laver en domænemodel og bagefter laver en datamodel er de koncepter som kan alle objekter i systemet arbejde med.

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.



Figur 6 – datamodellen efter 1. normalform er opfyldt

Normalisering (Shahnaz)

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 per-

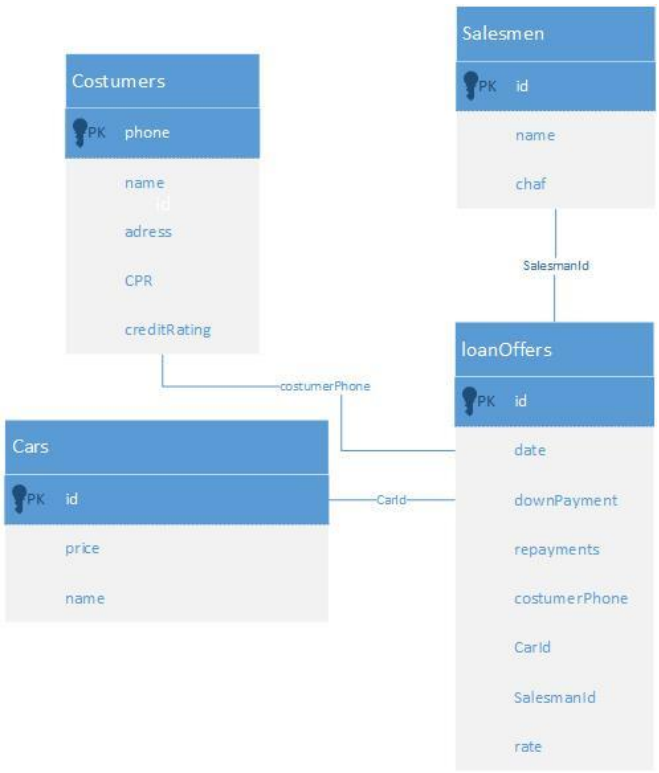
formance, men det skal være velovervejede designbeslutninger, og ikke dovenskab der skal være styrende for, om man vælger at gå på kompromis med normalformerne. Normalisering er godt i teorien, og er absolut en god tommelfingerregel.

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

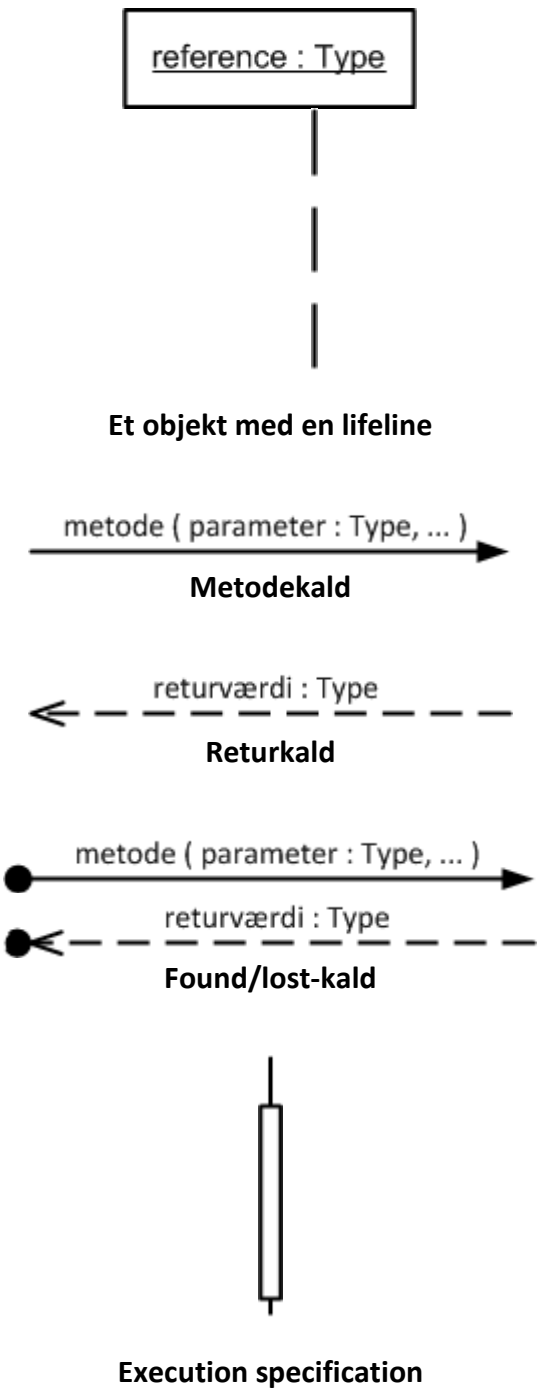


Figur 7 – datamodellen efter 2. normalform er opfyldt

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*.

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 en anden nøgle 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, den endelige datamodel kan ses i bilag 9.



Figur 8 - sekvensdiagramnotation

Sekvensdiagrammer(Sofie)

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 kon-

krete 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[39], 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[40] 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[41]. 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 se-

[39] Se Bilag 15
[40] Se Bilag 15
[41] Læs om singleton-mønsteret under afsnittet "Singleton pattern" – side 27

kvensen. 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) (Martin)

Et af kravene til vores program er, at vi i fremtiden skal kunne skifte til en webplatform. Vi har derfor valgt at benytte et facadepattern[42] som kommunikation mellem vores view og logic pakker. Vores brugergrænseflade skal kommunikere oplysninger fra klasserne i forretningslogikken. Hvis GUI-klassen skal kommunikere med alle disse klasser direkte, bliver det meget kompliceret at implementere og vedligeholde vores system. Facade pattern løsner koblingen mellem vores GUI og de klasser, som den benytter fra logic pakken, ved at indføre dette facadeobjekt, som kender klasserne i logic pakken. Det vil sige, GUI kender kun til facaden, og hver gang den har brug for at kalde en metode bliver de kaldt på facade, som så kender klasserne og sender kaldet videre til den klasse der skal løse problemet.

Vi overvejede også at implementere mediator pattern[43] da de to ligger meget tæt op af hinanden, men i vores system er der kun kommunikation mellem og GUI og vores klasser i logic pakken, ikke ret meget kommunikation mellem vores klasser i logic pakken. Vores logik-klasser har heller ikke behov for at GUI-klassen skal udføre noget for dem. Derfor mener vi det ligger tættere på et facade pattern end et mediator pattern. For at gøre det endnu lettere for os at skifte til en webplatform, har vi valgt at lave et interface til vores facadepattern, dermed har vi sikret at vi får implementeret alle de metoder vi har behov for, samt sikret os at vi kan udskifte GUI til f.eks. webplatform relativt nemt, da vi har en kontrakt at følge og dermed ikke skal bekymre os om hvad der foregår i logic pakken.

Singleton pattern(Sofie)

Singleton pattern 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 har overvejet at

[42] http://www.docjava.dk/patterns/facade/facade.htm
[43] http://www.docjava.dk/patterns/mediator/mediator.htm

benytte singletonmønsteret i flere situationer i forbindelse med implementationen af vores system.

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[44]. 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 valgte at bruge singletonmønsteret i forbindelse med vores DataLayer klasse. Denne klasse har forbindelse med databasen, og vi ønsker at undgå, at flere klasser opretter instanser af klassen, og tilgår den på samme tid. Serveren er en begrænset ressource, forstået på den måde, at hvis der er for mange åbne forbindelser samtidig, vil den ikke kunne holde til det. I vores program er der ikke så mange forbindelser, at det er et problem, men vi har fået at vide, at man i fremtiden ønsker at overføre programmet til en webplatform. I så fald, er der risiko for, at der vil blive åbnet mange forbindelser, hvilket vil belaste vores server. Hvis vores datalayer klasse er en singleton, har vi mulighed for kun at oprette én forbindelse til databaseserveren, hvilket vil løse dette problem. Vi kan imidlertid ikke lukke forbindelsen igen, eftersom vi ikke kan være sikre på hvornår den bruges for sidste gang, forbindelsen vil blive lukket når programmet terminerer.

[44] Med andre ord: en dårlig løsning på et designproblem.

Unit-test (Martin)

Unit-tests forsøger at teste så små dele af vores kode som muligt[45]. 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[46], 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 (Sofie)

I denne iteration har vil vi implementere use case 3, der beskæftiger sig mest med tråde. Vi vil udarbejde et aktivitetsdiagram, der visualiserer trådene i vores program, samt nogle forskellige design patterns i den forbindelse. Desuden vil vi implementere den algoritme, der skal udregne lånetilbud, og opdatere vores klassediagram efter implementationen.

---

[45] The Rational Unified Process: An Introduktion, Third Edition af Phillipe Kruchten, Addison Wesley 2003 – Kapitel 12
[46] Se bilag 10

GRASP (Grundlæggende designprincipper for objektorienteret design) (Shahnaz)

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

Information Expert, handler om hvor den nødvendige information skal findes. Metoder placeres i de klasser hvor de nødvendige attributter naturligt hører til, til at mindske kobling og øge samhørighed. I vores program befinder alle set og get metoder sig i de klasser, som har de attributter, som skal get eller settes.

Creator, handler om hvilket objekt skal skabe et nyt objekt, det objekt som har den stærkeste kobling til det nye objekt, eller det objekt som besidder information om det nye objekts initialiseringsdata, eller aggregerende objekter skaber aggregater (den situation skaber stærkere kobling som er uhensigtsmæssig). I vores program skaber vores FFSController et objekt fra datalayer, fordi den klasse er en facade controller og skal forbindelse til datalaget, så har stærkere kobling til det. Og vores GUI klasse skaber kunde- og bil- og sælgerobjekter fordi den indeholder initialiseringsdata til dem.

Controller, handler om hvilket objekt skal modtage input fra GUI, objektet, som repræsenterer en facade for det samlede system[47], eller objektet, som repræsenterer det konkrete use case-scenarie(session-controller). Systemsekvensdiagrammer hjælper rigtig meget til vi kan finde ud af hvor vi skal bruge en controller. Controllerne delegerer også objektet. I vores program bruger vi en FFSController som repræsenterer vores samlede system og videredelegerer arbejdet.

Kobling, betegner afhængigheder mellem klasser[48], man ønsker at begrænse kobling, men samarbejde mellem objekter forudsætter altid en vis kobling som ikke kan undgås. Et system som har meget kobling er meget svært til at overskue, genbruge og vedligeholde.

[47] Læs om vores facadecontroller under afsnittet Facade Pattern(facadecontroller) - side 27
[48] Læs om kobling i vores system under afsnittet Klassediagram - side 19

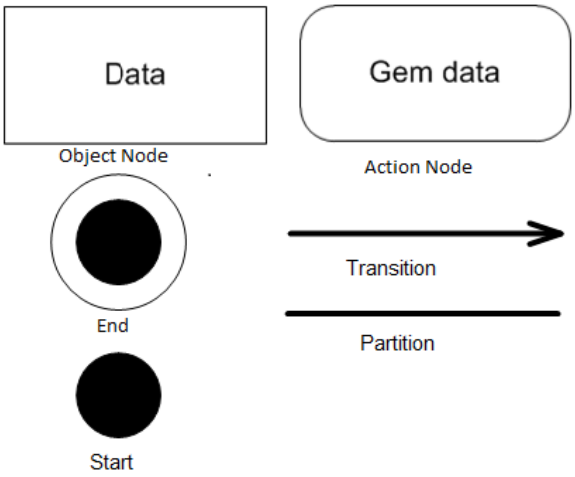
Samhørighed, betegner det funktionelle slægtskab for indholdet af en klasse. Man ønsker altid høj samhørighed, men der er umuligt at sætte alle metoder i en klasse så skaber på den måde den stærkeste kobling.

Aktivitetsdiagram (Shahnaz)

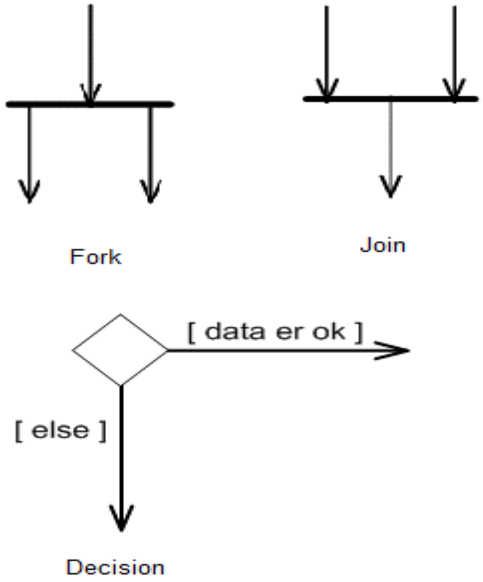
Når man vil visualisere en proces, kan man bruge aktivitetsdiagrammer. Man kan bruge dem til at visualisere alle dele i vores arbejdsproces.

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[49] 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 9 - Aktivitetsdiagramnotation



Figur 10 - notation vedrørende parallelitet

[49] Se al notation i *figur 9*

Man kan vise to eller flere aktiviteter som arbejder samtidig med Fork[50] 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[51].

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[52], 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[53] kommer med dagens rentesats og UC1[54] 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.

Tråde(Shahnaz)

Enkelttrådede programmer er en sekventiel proces som begynder fra et sted i et program og efter at have kørt al nødvendige kode færdig på et tidspunkt giver svar til sidst; men nogen gange i et program forventes at to eller mange stykker kode kører på samme tid og giver mange forskellige resultater, derfor bliver en programmør nødsaget til at bruge en måde som kan gøre det.

Tråde giver os den mulighed for at køre mange stykker kode sammen og giver forskellige resultater. Et enkelt-trådet programs tilstand består af objektsystemets tilstand, alle lokale variables tilstande, og det sted programmet er kommet til i sin udførelse; men en tråds tilstand består af alle lokale variables tilstande i tråden, samt det sted som tråden kommet til i sin udførelse. Så hvis vi gerne vil vide et flertrådet programs tilstand skal vi kende alle trådenes tilstande og alle objektsystemets tilstande.

Man har to måder at lave tråde i Java på: Den første er at nedarve (extende) den abstrakte klasse Thread som har en Run metode. Denne metode udfører al den kode vi giver den, efter vi har

[50] Se al notation vedrørende parallelitet i *figur 10*
[51] Se aktivitetsdiagrammet, bilag 17
[52] Se bilag 6
[53] Se bilag 5
[54] Se bilag 4

kaldt trådens Start-metode. Start-metoden skaber tråden ved at returnere samtidig med, at den begynder at køre koden i Run-metoden, som vi har overrided.

Den anden måde er, at man kan implementere interfacet Runnable, som har en abstract Run metode. Vi laver så et Thread objekt, som tager en Runnable klasse parameter og giver den et objekt fra vores klasse. På den måde kan man nøjes med extends Thread i sit program og ikke flere klasser. I vores program vi var nødt til at bruge den anden mode, hvor vi implementerer interfacet Runnable, fordi vi var nødt til bruger observer patten[55] og skulle extende klassen observerble i vores program og kunne ikke extende to klasser.

Det var et krav til vores program, at vi indhenter oplysninger om en kundes kreditværdighed fra RKI og rentesatsen fra banken samtidig med, at kunden kunne indtaste andre oplysninger, der var nødvendige til udregning af et lånetilbud. Derfor har vi implementeret tråde, som også kan ses i vores aktivitetsdiagram[56].

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.

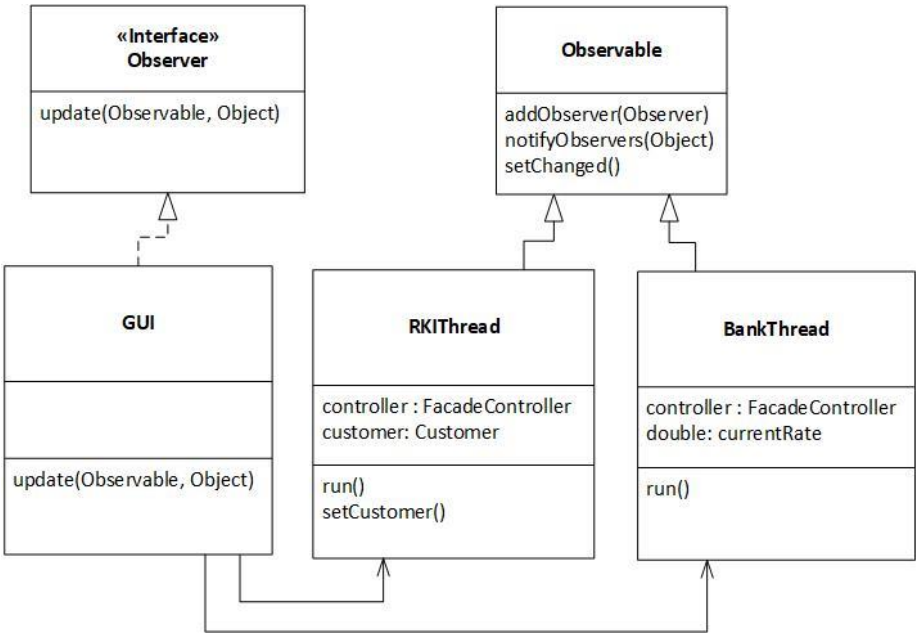
Observer-pattern fungerer således: der er en eller flere observere samt et subject. Observeren tilmelder sig som observer hos subject, med det formål at få besked, hver gang der sker en tilstandsændring hos subject. Subject vil så give besked hver gang der sker tilstandsændringer, og senere kan observeren 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.

[55] Læs om vores implementation af observer pattern under afsnittet Observer Pattern - side 32
[56] Se Bilag 17

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[57]. 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



Figur 11 - Observer pattern - diagram over implementation

[57] Se *figur 11 - Observer pattern - diagram over implementation*. Kun de af Javas metoder, som vi har benyttet, er repræsenteret i diagrammet.

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.

Klassediagram - opdateret (Shahnaz)

I dette trin af klassediagram har vi tilføjet to klasser som hedder RKIThread og BankThread til klassediagrammet. Vi bruger dem til at skabe tråde[58] i programmet; vi tilføjer dem i view-laget og tilføjer også metoder til GUI som skaber tråde til programmet. Alle disse opdateringer er blevet indført i vores klassediagram.

Algoritme (Martin)

En central del af vores system er at beregne lån. Til det formål skal vi bruge en rente, renten grundlag for en stor del af virksomhedens indtjening på de lån der oprettes. Vi har ud fra virksomhedens krav udarbejdet netop denne algoritme.

Algoritmen ligger i Calculator-klassen. Vi mente ikke, at nogle af de på den tid eksisterende klasser kunne indeholde algoritmen. Den fik derfor sin egen klasse, som senere blev udvidet med flere beregninger. For at beregne en kundes personlige rente skal vi bruge en række parametre: kundens kreditvurdering fra RKI, den nuværende bank rente, udbetaling på lånet, lånets løbetid og bilens pris. Alle disse parametre bliver sendt med når calcInterestRate-metoden bliver kaldt. I calculator klassen har vi valgt at lave tre private metoder som alle beregner deres egen del af renten og returnere den.

intRateFromRating-metoden udregner hvor meget der skal pålægges renten afhængig af hvad en kundes kreditvurdering er: 3% tillægges hvis de har fået en C vurdering, 2% tillægges hvis de har fået en B vurdering, 1 % tillægges hvis de har fået en A vurdering og hvis de har fået D som er den dårligste vil vi slet ikke handle med dem. Vi har håndteret dette med simple if/else statements

[58] Læs om tråde under afsnittet ”Tråde” – side 32

til A, B og C; else dækker så over dem vi ikke vil handle med og kaster i stedet en af vores exceptions[59]. Metoden returnerer så den rentesats, der skal pålægges.

intRateFromDP-metoden udregner, om der skal påligges 1% mere til renten. I tilfælde af kunden har valgt en udbetaling der er mindre end 50% af bilens værdi, vil den så returnere 1%, der skal pålægges; hvis ikke returnerer vi blot 0. Her har vi igen benyttet en if/else statement .

intRateFromMonths-metoden tjekker om lånet løber over længere end 3 år. Hvis dette er tilfældet, skal der pålægges 1% yderligere. Vi benytter her igen en if/else statement, som blot tjekker, om antallet af måneder er over 36; hvis ja så returnerer vi 1% som skal lægges til, hvis ikke returnerer vi blot 0.

Disse tre værdier bliver så lagt sammen med den rente, der er indhentet fra banken, og resultatet returneres.

Tests (Martin)

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[60]. 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 me-

```
@Test
public void calcInterestRatingATest() throws PoorCreditRatingException {
    Calculator intcalc = new Calculator();
    double currentRate = 5.0;
    Rating rate = Rating.A;
    int downPayment = 250000;
    int numberOfMonts = 60;
    int carPrice = 1000000;

    assertEquals(8.0, intcalc.calcInterestRate(rate, currentRate, downPayment, numberOfMonts, carPrice), 0);
}
```

Figur 12

tode.

I *figur 12* 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 -

[59] Læs om exceptions under afsnittet Exceptions – side 38
[60] The Rational Unified Process: An Introduktion, Third Edition af Phillipe Kruchten, Addison Wesley 2003 – Kapitel 12

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.

Det er værd at notere, at der er yderligere kommentarer, som ikke er vist på vores lille kode

```
@Test(expected = PoorCreditRatingException.class)
public void calcInterestRatingDTest() throws PoorCreditRatingException {
    Calculator intcalc = new Calculator();
    double currentRate = 5.0;
    Rating rate = Rating.D;
    int downPayment = 250000;
    int numberOfMonts = 60;
    int carPrice = 1000000;

    assertEquals(10.0, intcalc.calcInterestRate(rate, currentRate, downPayment, numberOfMonts, carPrice), 0);
}
```

Figur 13

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[61]**.** Dette er lidt interessant i forhold til vores j-unit test. Vi har måtte tilføje en ekstra linje[62]. 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.

Da vi ikke laver metode kald direkte fra GUI til klassen som beregner renten, men går igennem

```
@Test
public void calcInterestRatingATest() throws PoorCreditRatingException {
    FacadeController cont = new FFSController();
    double currentRate = 5.0;
    Rating rate = Rating.A;
    int downPayment = 250000;
    int numberOfMonts = 60;
    int carPrice = 1000000;

    assertEquals(8.0, cont.calculateInterestRate(rate, currentRate, downPayment, numberOfMonts, carPrice), 0);
}
```

Figur 14

[61] Se afsnit om Exceptions – side 38
[62] Se *figur 13*

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[63].

Exceptions (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 sikkerhedsforanstaltning i vores system, så bilsælgere ikke ved en fejl kommer til at oprette lån til disse kunder. Vi har derfor valgt at lave vores egen exception klasse, som nedarver fra superklassen Exception. Vores exception tager en String som parameter til constructoren, som vi vil bruge til at give en fejlmeddelelse med. Denne Constructor er også defineret i superklassen Exception, og vi kan derfor kalde den videre igennem vores egen constructor. Vi kunne godt have valgt ikke at give nogen parametre med, da vi kun bruger den ét specifikt sted, men vi fandt at det er var god idé 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.

Iteration E3 + E4 (Sofie)

I disse iterationer vil vi implementere salgschefens mulighed for at godkende lånetilbud, samt eksport af tilbagebetalingsplan til CSV-fil, med andre ord: use case 4 og 5. Vi vil ud fra implementationen også opdateret klasse- og sekvensdiagrammer. Disse iterationer er slået sammen under ét afsnit, eftersom vi har brugt meget tid på implementation og mindre tid på analyse. Det er naturligt i UP, at man efterhånden som projektet skrider frem bruger mere og mere tid på implementation. Der har derfor ikke været meget at skrive i rapporten om disse iterationer. Til gengæld henviser vi til vores omfattende kommentarer i koden, der dokumenterer vores implementation.

[63] Se *figur 14*

CSV File(Shahnaz)

Til at vi kan printe tilbagebetalingsplanen, skal vi implementere en csv-fil, hvilket senere kan åbnes i eksempelvis Excel. I Calculator klassen har vi implementeret en metode, som kan udregne alle detaljerne til den tilbagebetalingsplan. Den returnerer et 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(OUT_DEBT), renten pr. måned, som regnet ud fra ÅOP, og afdrag pr. termin minus rente(INSTALL) (se i figur 15) og senere vi har brugte den til at vise de detaljer i csv file.

```
    HashMap<LoanPlanComponent, ArrayList<Double>> comps = new HashMap<LoanPlanComponent, ArrayList<Double>>();

    // primo restgæld, startende med hovedstol
    double outsDebt = loanOffer.getCar().getPrice() - loanOffer.getDownPayment();
    // renten pr måned regnet ud fra ÅOP
    double rateMonth = calcMonthlyInterestRate(loanOffer.getAnnualCost());

    // indeholder rente pr. termin i kr.
    ArrayList<Double> rate = new ArrayList<Double>();

    // indeholder afdrag pr termin minus rente
    ArrayList<Double> install = new ArrayList<Double>();

    // indeholder ultimo restgæld
    ArrayList<Double> outDebt = new ArrayList<Double>();

    for (int i = 0; i < loanOffer.getNumberOfMonths(); i++) {

        // udregn renten pr termin i kr. (rente * primo restgæld)
        rate.add((rateMonth / 100) * outsDebt);

        // udregn afdrag pr termin uden rente (ydelse - rente pr. termin i kr.)
        install.add(loanOffer.getRepayments() - rate.get(i));

        // træk afdrag fra primo restgæld og sæt det til at være ultimo restgæld
        outsDebt = outsDebt - install.get(i);

        // tilføj den nuværende ultimo restgæld til array
        outDebt.add(outsDebt);
    }

    comps.put(LoanPlanComponent.RATE, rate);
    comps.put(LoanPlanComponent.INSTALL, install);
    comps.put(LoanPlanComponent.OUT_DEBT, outDebt);

    return comps;
}
```

Figur 15 - 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.

Use cases (Shahnaz)

Vi har lavet uformelle use case til use case 4 og 5[64] som vi har beskrevet kort sekvens af handlinger og interaktioner i vores use case

I **use case 4** beskriver vi hvordan chef sælger godkender et lånetilbud, og hvordan det lånetilbud bliver gemt som et bekræftet lånetilbud i databasen, hvis tilbuddet bliver godkendt.

I **use case 5** beskriver vi hvordan man eksporterer en tilbagebetalingsplan fra det lånetilbud som bliver godkendt, så systemet præsenterer i den use case de lånetilbud for bilsælgeren og han kan så printe dem ud i CSV-formatet[65].

Use case 4 og 5 er ikke de centrale use cases i projektet. Hvis vi lavede nogen fejl i dem eller vi kom til at misforstå dem, giver de ikke lige så stor omkostning til vores projekt som nogle af de andre use-cases. Derfor vi har valgt at arbejde med dem til slutning af projektet. Man kan se i vores iterationsplan, at vi udarbejdede use case 5 to gange, fordi vi tænkte, at vi skulle eksportere detaljerne for et lånetilbud, frem for en bogstavelig tilbagebetalingsplan.

Mock-ups - opdateret (Shahnaz)

I denne iteration har vi lavet en del af mock-ups[66] som viser user interface som handler om vindue af programmet, som viser alle lånetilbud, der er blevet godkendt. Sælgeren kan så vælge at se detaljerne for én af dem, og herefter udskrive dem til en CSV-fil. Brugerfladen for use case 4 og 5 ligner hinanden meget, og derfor har vi kun udarbejdet én mock-up for de to use cases samlet.

Test (Sofie)

Det at teste har i disse iterationer været mere manuelle, vi har ikke brugt j-unit, men mere kørt vores programmer med coverage og manuelt set at systemet kører al koden og fungerer hen-

[64] Se Bilag 12
[65] Se afsnittet CSV file – side 38
[66] Se Bilag 8.

sigtsmæssigt. Vi har dog ikke haft lige så meget fokus på testing i disse iterationer, som i de andre. Testing var især hensigtsmæssigt i forbindelse med vores matematiske udregninger i implementationen af algoritmen, og vi henviser til det tidligere afsnit om tests.

Sekvens- og klassediagram - opdateret (Shahnaz)

Vi har lavet mange vigtige ændringer i klasse diagram i den her iteration, vi har havet RKI og Banken som singleton men vi besluttede til at lave dem bare almindelige klasser, eftersom de ved nærmere eftertanke ikke løste nogen af vores designproblemer, og derfor var overflødige. Vi har også i første omgang lavet vores Datalayer klasse som almindelig klasse, men vi har ændret den til at bliver singleton[67] og selvfølgelig vi har tilføjet nogle metoder til den, så den var up to date med koden. Generelt har vi opdateret klassediagrammet sideløbende med at vi har kodet, eftersom klassediagrammet er en hjælp for os til at visualisere programmets klasser[68].

Vi har også lavet sekvensdiagram SD4[69] til use case 4 som viser metoden til godkendelse af lånetilbud. Det virker således: (approveLoan(loan: LoanOffer)) Facade controller kalder approveLoan(loan: LoanOffer) metoden, og den metode kalder updateLoanOfferbyId som opdaterer en attribut i databasen, som hedder isApproved til at bliver true og viser, at det lånetilbud bliver godkendt.

Konklusion på elaboration-fasen (Sofie)

Vi har haft flere iterationer igennem elaboration-fasen, der har resulteret i meget dybdeborende analyse og implementation. Vi har dog ikke på noget tidspunkt fået elimineret alle vores identificerede risici. Vi har ikke været i stand til at vurdere præcis hvor lang tid de forskellige aktiviteter tager, og derfor har vi ikke kunnet være fuldstændig sikre på, at vi kunne overholde deadline. Desuden er vores projektplan blevet opdateret flere gange, men ikke blevet helt fuldført. Det er primært disse fakta, der gør at vi ikke er gået over i construction-fasen. Vi har dog efterhånden haft større og større fokus på implementation og design, mens analyse og modellering er trådt i baggrunden, hvilket er et af kendetegnene ved at man nærmer sig construction-fasen, og projektet

[67] Læs argumentation for brug af singleton under afsnittet Singleton - side 27
[68] Se den nyeste version af klassediagrammet i Bilag 16.
[69] Se sekvensdiagrammer i Bilag 15.

fremskrider på en god måde. Vi har dog ikke i nogen af iterationerne undladet analyse og kravspecifikation, da det ville betyde, at vi brugte vandfaldsmodellen frem for UP. Dog har vi i slutningen af vores sidste iteration fået udarbejdet et fuldt ud eksekverbart produkt, der kan overdrages til kunden.

Konklusion

Dette projekt har haft til formål at løse nogle af den regionale Ferrariforhandlers nuværende problemer i forbindelse med deres nuværende arbejdsgang mht. oprettelse og afgivelse af finansieringstilbud. Implementationen af vores system bør effektivisere denne arbejdsgang på flere måder.

Vi har indført et enkeltbrugersystem, hvilket betyder, at mange medarbejderes arbejdsbyrde lettes markant. Systemets algoritme står for udregning af lånetilbud, og er også i stand til at gemme dem vha. en relationel sql-database. Ved hjælp af implementation af et flertrådet system, har vi muliggjort et responsivt og flydende bruger interface, der gør oplæringen af nye medarbejdere, og implementationen af systemet nem og effektiv for virksomheden. Vores system er opbygget med en velvalgt og gennemtænkt arkitektur, således at en udvidelse til en webplatform i fremtiden bør være simpel at indføre. Desuden har vi muliggjort en eksport af lånetilbuddenes tilbagebetalingsplaner til en fil i CSV-format, som kan åbnes i ekstern software, som for eksempel Excel eller lignende. vi har, med virksomhedens forretning in mente, haft stort fokus på at skabe en pålidelig algoritme, der udregner lånetilbuddene med stor nøjagtighed. Implementationen af vores system vil med sikkerhed effektivisere Ferrariforhandlerens proces og bør, alt andet lige, skabe værdi og indtjening for vores kunde.

Vi har i udarbejdelsen af vores projekt taget udgangspunkt i UP, hvilket er en iterativ udviklingsproces. Den er feksibel, og har gjort det muligt for os at placere vores fokus der, hvor det har været nødvendigt i hver iteration. Vi har tidligt i processen haft fokus på analyse af virksomhedens opbygning og arbejdsgang, samt kravindsamling; vi har for eksempel udarbejdet et visionsdokument, som også er blevet godkendt, samt domænemodel, operationskontrakter og lignende. Hele vores projekt har desuden været drevet af use cases baseret på de oplysninger vi har fået fra kunden. Dette har hjulpet os til at indskrænke projektets scope, så vi har kunnet fokusere vores energi de rigtige steder.

Vi har udarbejdet artefakter, der hjælper os som udviklere, og gør vedligehold og senere udvikling af systemet let og mindsker konflikter; i den forbindelse har vi benyttet UML-diagrammer, der er baseret på et universelt modelleringssprog. Klassediagram, aktivitetsdiagram og sekvensdiagrammer har hjulpet os til hele projektet igennem at holde overblik, samt at være konsekvent med vores valgte arkitektur, nemlig trelagsmodellen. Her har vi også haft GRASP (grundlæggende designprincipper for objektorienteret design) i tanke, og prioriteret at følge disse principper i så vid udstrækning som muligt. Vores implementation bygger desuden på velvalgte design patterns; her kan nævnes Observer pattern, Singleton pattern og Facade pattern. Disse mønstre har afhjulpet mange af vores designkonflikter, og desuden muliggjort et flertrådet program, hvilket sikrer en god brugeroplevelse. Vi har løbende testet vores program med automatiserede tests i form af en testsuite, hvilket har gjort at vi hele tiden har haft styr på vores program. Vi har samtidig reviewet vores arbejde, både internt og med andre grupper, hvilket sikrer, at alle vores artefakter er forståelige udadtil. Vi har løbende haft møder med vores vejledere, for således at sikre, at vi har været på rette spor, og har kundens interesser og behov som højeste prioritet.

Litteraturliste

- The Rational Unified Process: An Introduktion, Third Edition af Phillipe Kruchten, Addison Wesley 2003

- Professional it-forundersøgelse - grundlaget for brugerdrevet innovation af Keld Bødker, Finn Kensing & Jesper Simonsen, 2008

- IBM - https://www.ibm.com/developerworks/rational/library/3975.html

- Larman, Applying UML and Patterns 3rd edition.

- www.docjava.dk - *(C) 1999-2014, Flemming K. Jensen*

Bilag

Bilag 1 – Iterations- og faseplan

Fase	**Inception**	**Elaboration**			
Iteration	*I0 30/4 - 3/5*	*E1 3/5 - 10/5*	*E2 14/5 - 18/5*	*E3 21/5 - 25/5*	*E4 28/5 - 1/6*
Plan	Opsæt udviklingsmiljø	Lave ikke funktionelle krav (FURPS)	Operationskontrakt til UC3	Opdater FURPS	Udarbejd endelig FURPS
	Påbegynd fase- og iterationsplan	Identificer alle use cases.	Aktivitetsdiagram for UC3	Opdater domænemodel	Udarbejd endelig domænemodel
	Påbegynd visionsdokument	Påbegynd datamodel	Opdatere klassediagram	Opdater risikoanalyse	færdiggør risikoanalyse
	Identificer use cases	Formel beskriv UC3 – opret lånetilbud	Implementation af UC3	Opdater datamodel	færdiggør datamodel
	Påbegynd risikoanalyse	Systemsekvensdiagram for UC1-3	Uformel beskrivelse af yderligere use cases	OC til UC4-5	OC til UC6
	Påbegynd domænemodel	Operationskontrakt for UC1 og UC2	Opdater datamodel	Opdater CD	færdiggør CD
	Formel beskrivelse af UC1-2	Sekvens diagram for UC1 og UC2	Unit test af UC3	implementation af UC 4-5	implementation af UC6
	Udarbejd mock-ups	Opdatere klassediagram		diverse tests af UC4-5	Tests (System-, Accept-, Integrations-)
	Review i samarbejde med gruppe 2	Implementere UC1 og UC2.		færdiggør use case-diagram (alle UC's identificeret)	
	Påbegynd dataordbog	Kørende GUI (proof of concept)			
		Unit test af UC1 og UC2			
		Opdater dataorbog			
Milepæl	*De fleste UC's er identificeret*	Alle UC's er identificeret			
	UC-1 er fuldt beskrevet	UC1-3 er fuldt beskrevet			
	Domæne model påbegyndt	Arkitekturen er stabil og beskrevet			
	Visionsdokument godkendt	Eksekverbart produkt er udarbejdet (use cases implementeret)			
	Iterationsplan for E1 er klar				

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.

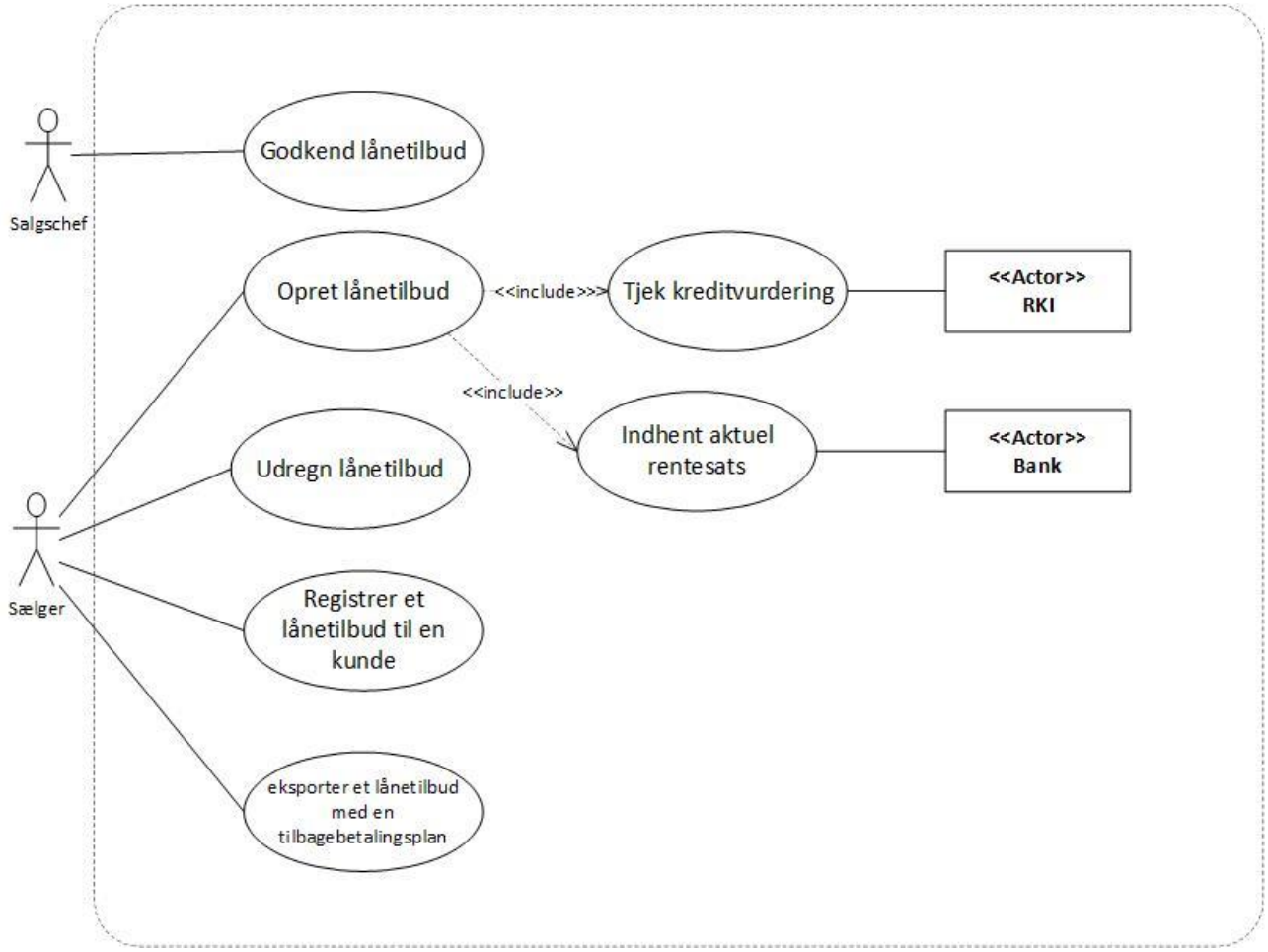
Interessentanalyse

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



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

3. Sælgeren starter processen forfra
4. Sælgeren genstarter systemet

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

2. 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

5. Bilsælgeren starter processen forfra
6. Bilsælgeren genstarter systemet

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

3. 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.

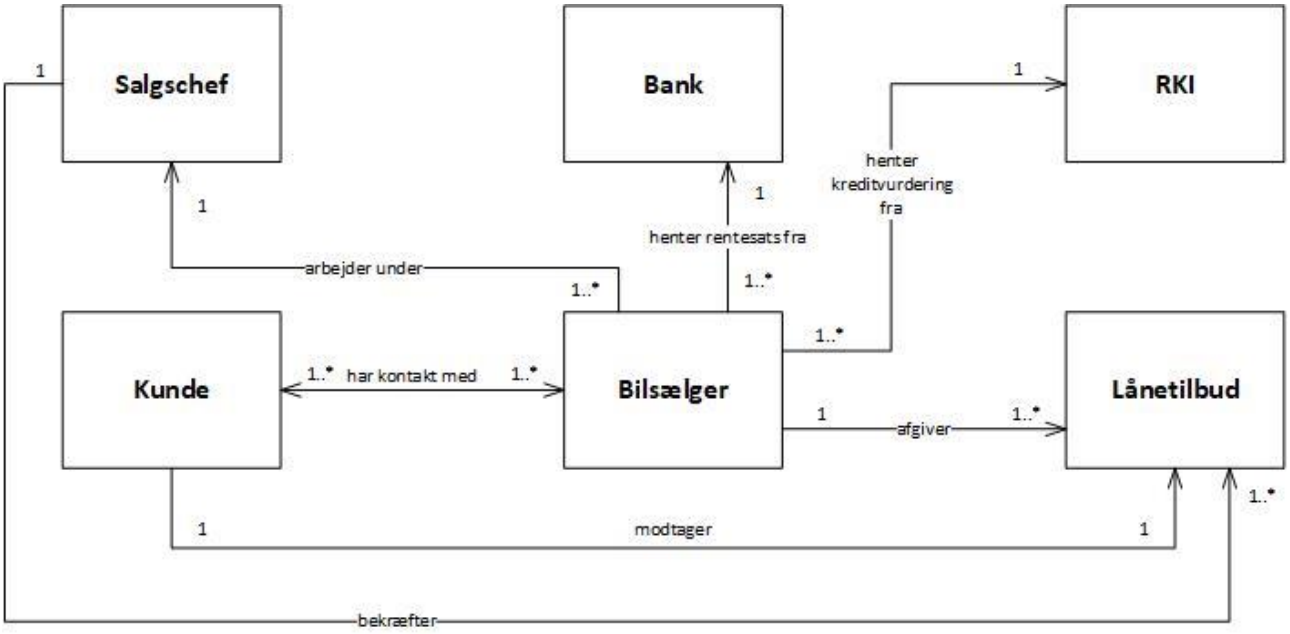
1. Bilsælgeren afbryder processen og prøver igen.
   1a. Bilsælgeren kontakter RKI eller Bank med henblik på fejlfinding
   1b. Bilsælgeren tilkalder support for lokale fejl

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

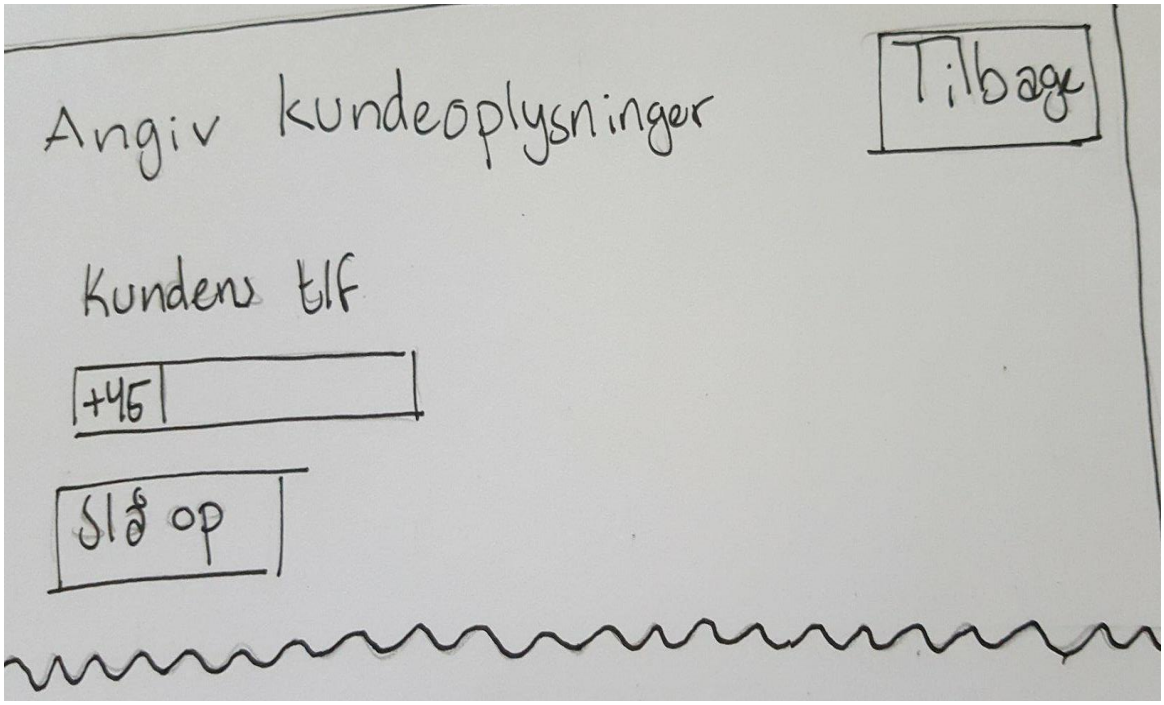
Hyppighed: Ofte

Bilag 7 – Domænemodel

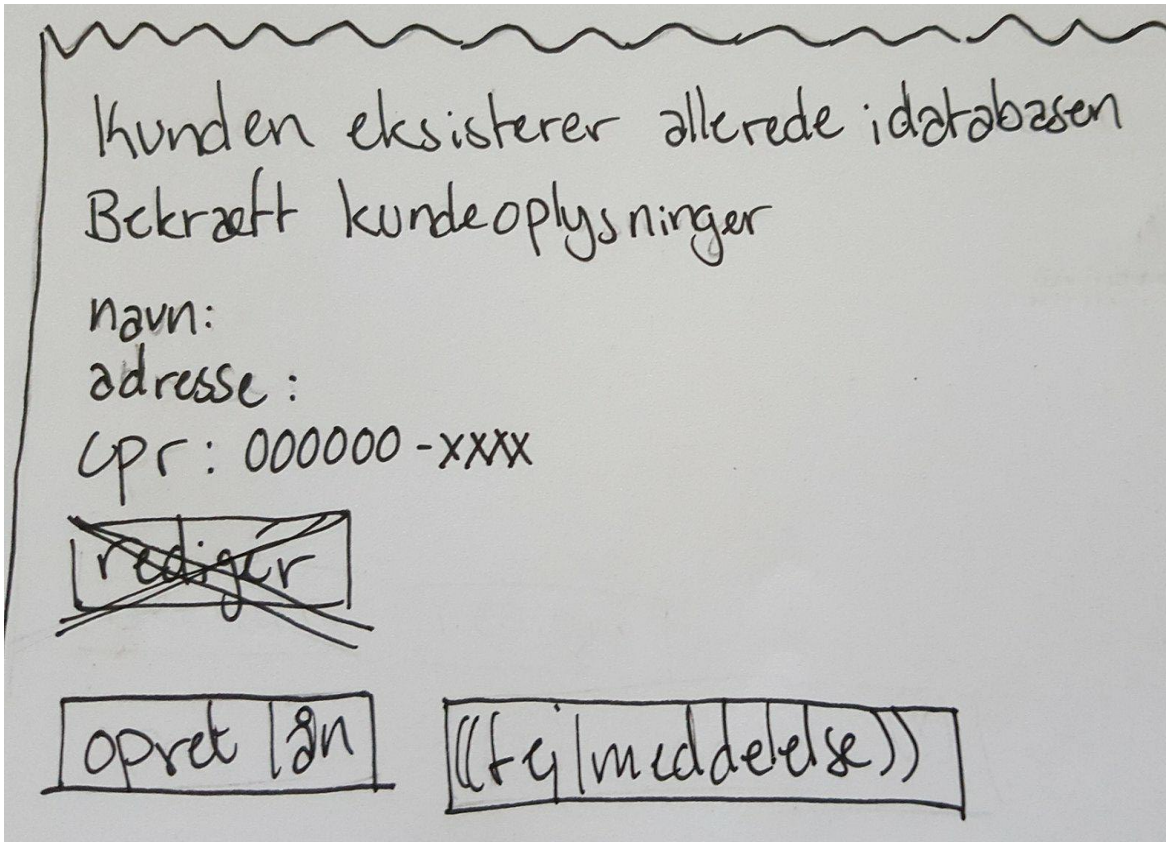


Bilag 8 - Mock-ups

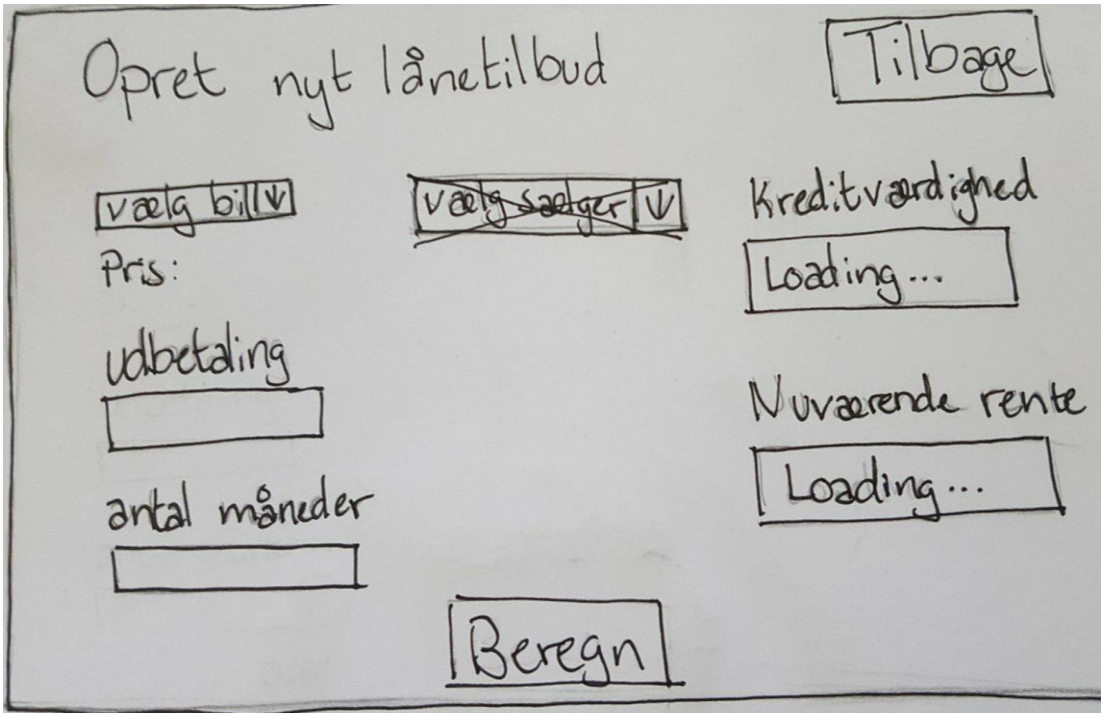
Mock-up 1 - UC1-3: Slå en kunde op i databasen



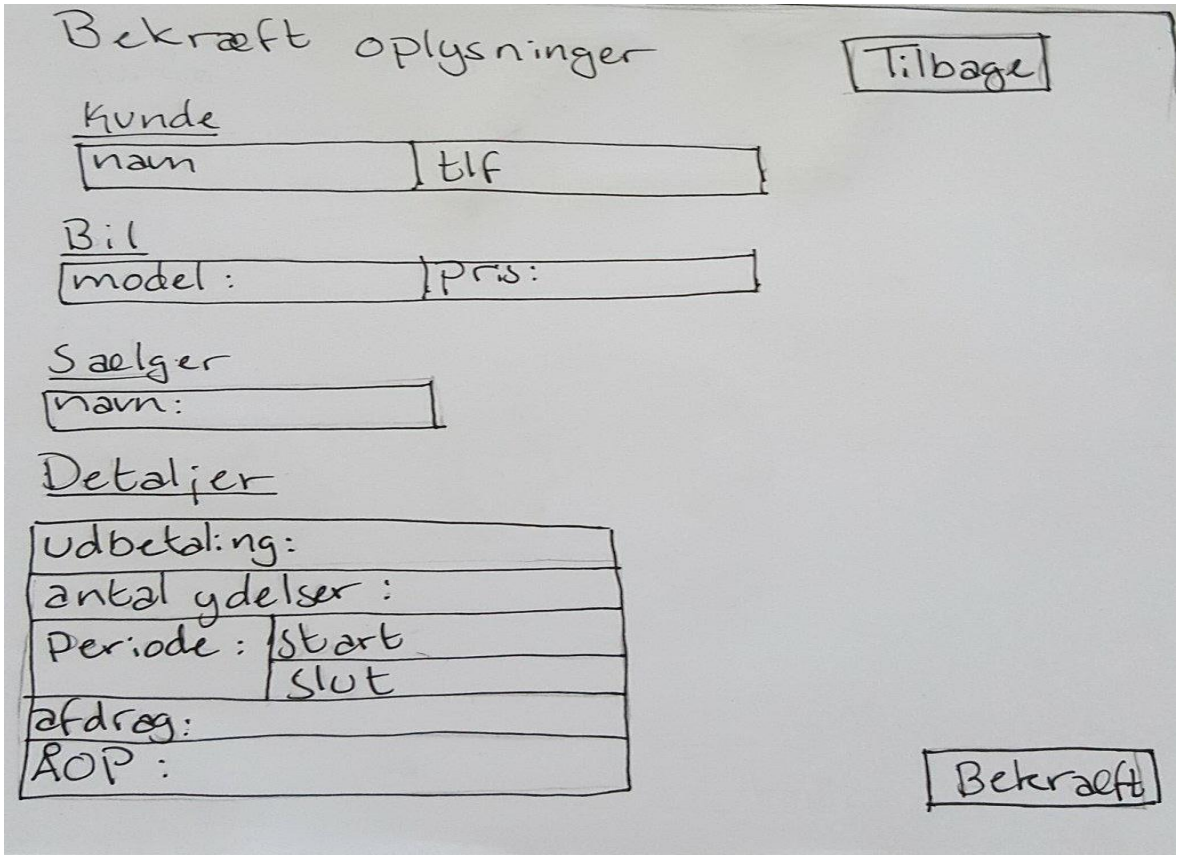
Mock-Up 2 - UC1-3: Vælg kunden der skal oprettes en lånetilbud for



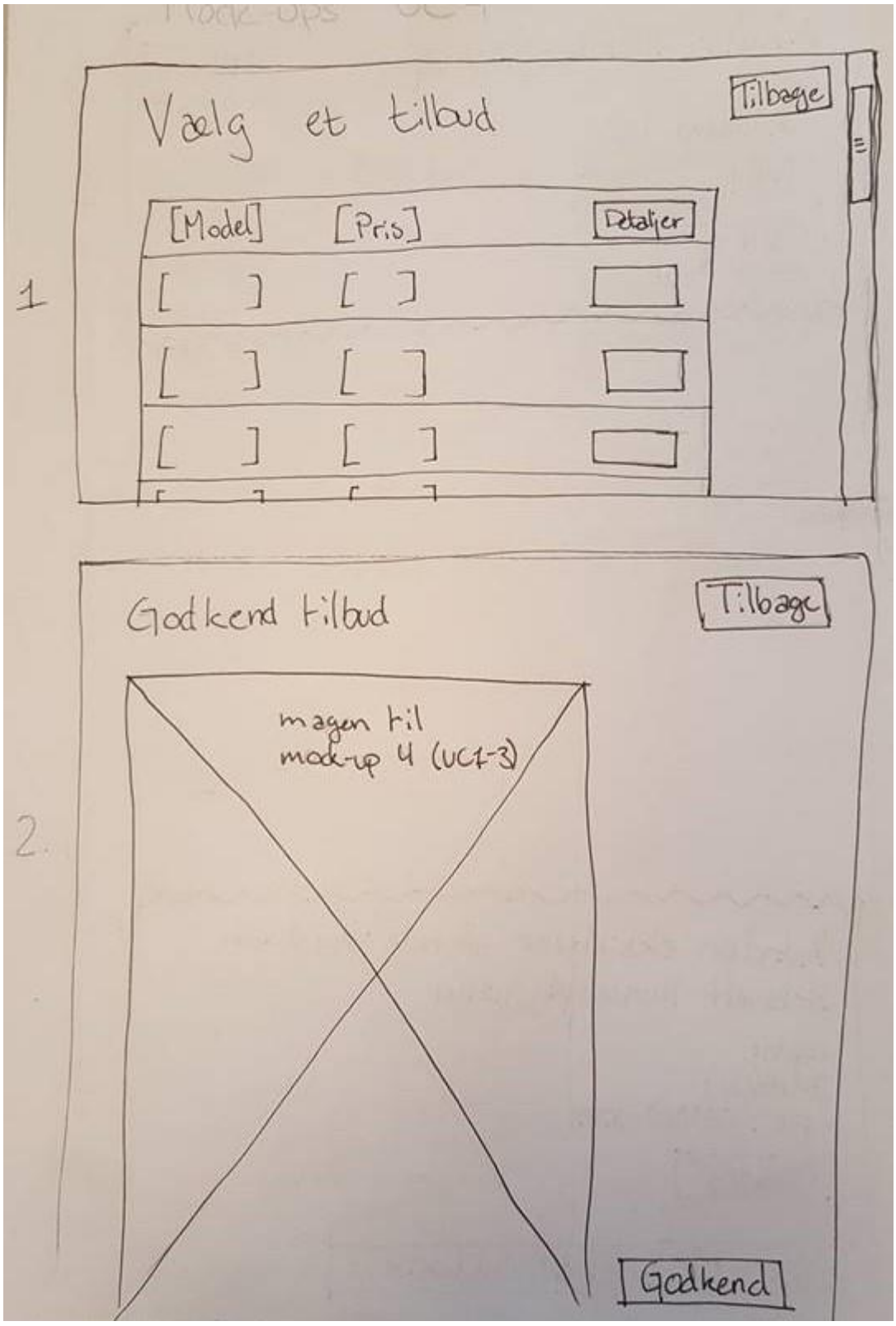
Mock-up 3 - UC1-3: Angiv oplysninger for et nyt lånetilbud:



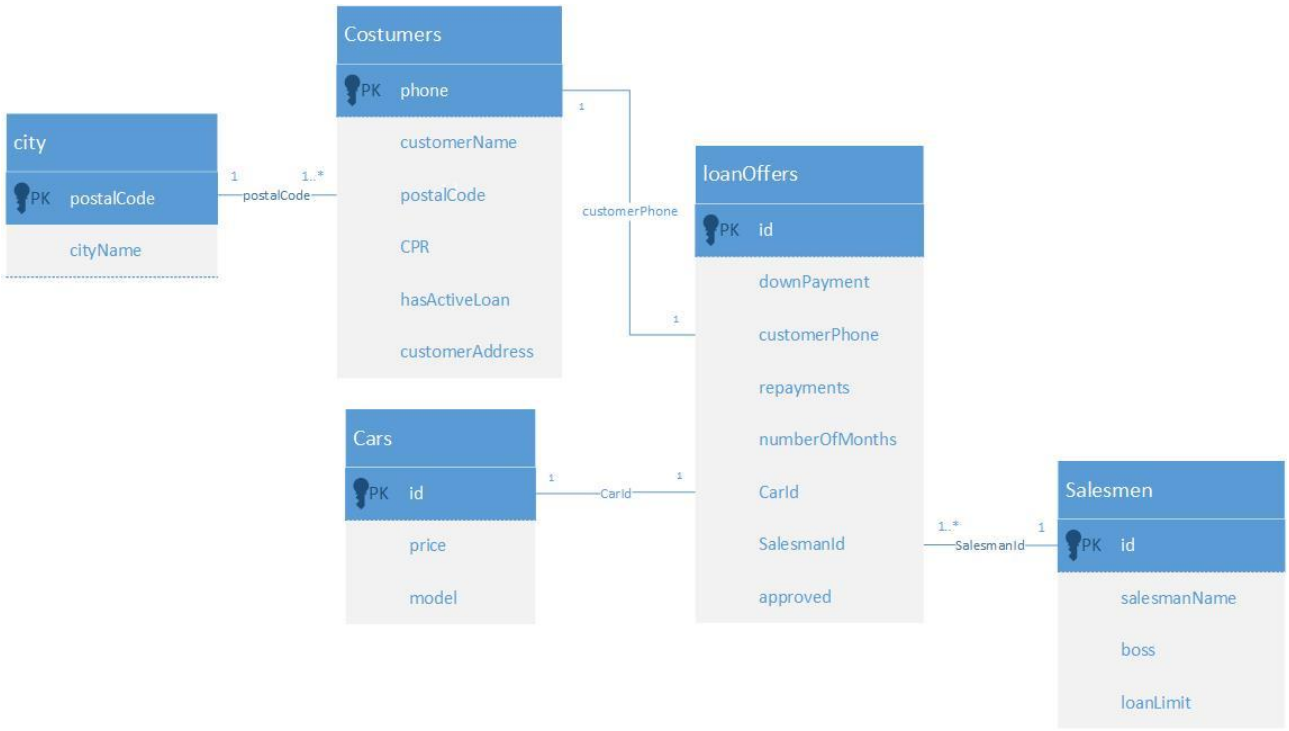
Mock-up 4 - UC1-3: Bekræft oplysninger som bil sælger gemmer i databasen



Mock-up1-2 - UC4: Vælg og godkend tilbud



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:

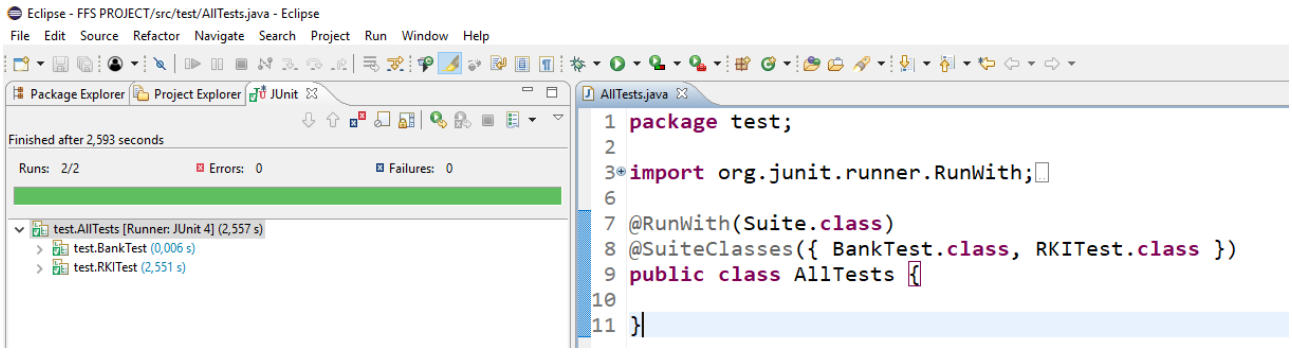
  vi kan ikke lave noget ved sygdom så bare lader den være.

- 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 - use case 4 + 5 (uformelle)

FFS-UC4: Godkend lånetilbud

Der findes et ikke godkendt lånetilbud i databasen. Salgschefen er parat til at godkende et nyt lånetilbud.

Salgschefen navigerer ind til ”Godkend lånetilbud”. Systemet præsenterer de lånetilbud for salgschefen, der endnu ikke er godkendt. Salgschefen gennemser lånetilbuddet og angiver om han ønsker at godkende det nu eller lade det ligge. Hvis tilbuddet godkendes, markerer systemet tilbuddet som værende godkendt i databasen. Hvis tilbuddet ikke godkendes, ændres dets tilstand i databasen ikke.

FFS-UC5: Eksporter lånetilbud

Der findes et godkendt en tilbagebetaling plan til en lånetilbud i databasen. Bilsælgeren er parat til at eksportere tilbagebetaling planen.

Bilsælgeren navigerer ind til ”eksporter lånetilbud”. Systemet præsenterer de lånetilbud for bilsælgeren, der er godkendt. Bilsælgeren vælger det lånetilbud han gerne vil eksportere tilbagebetaling plan til. Systemet præsenterer detaljer for det valgte lånetilbud for bilsælgeren. Bilsælgeren vælger ”eksporter tilbagebetaling plan”. Systemet udskriver tilbagebetaling planen som en CSV-fil. Systemet præsenterer bilsælgeren med en bekræftelse om at filen er blevet eksporteret.

FFS-OC1: SetCreditRating

Systemoperation

setCreditRating(customer)

Krydsreferencer

FFS-UC1 - tjek kreditværdighed

FFS-UC3 - Udregn lånetilbud

Forudsætninger

Instansen customer af Customer eksisterer.

Slutbetingelser

customer.creditRating er blevet sat.

FFS-OC2: getCurrentRate

Systemoperation

getCurrentRate()

Krydsreferencer

FFS-UC2 - Indhent aktuel rentesats

FFS-UC3 – Udregn lånetilbud

Forudsætninger

-

Slutbetingelser

currentRate er oplyst

FFS-OC3 udregnLånetilbud

Systemopration

calculateLoanOffers(customer, date, downPayments, repayments, customerPhone, carId, salesmanId)

Krydsreferencer

FFS-UC2

FFS-UC1

FFS-UC3

Forudsætninger

customer er oprettet i databasen

Slutbetingelser

En instans af loanOffer l er blevet skabt.

l.date er blevet sat til date.

l.downPayments er blevet sat til downPayments.

l.repayments er blevet sat til repayments.

l.customerPhone er blevet sat til customerPhone.

l.carId er blevet sat til carId.

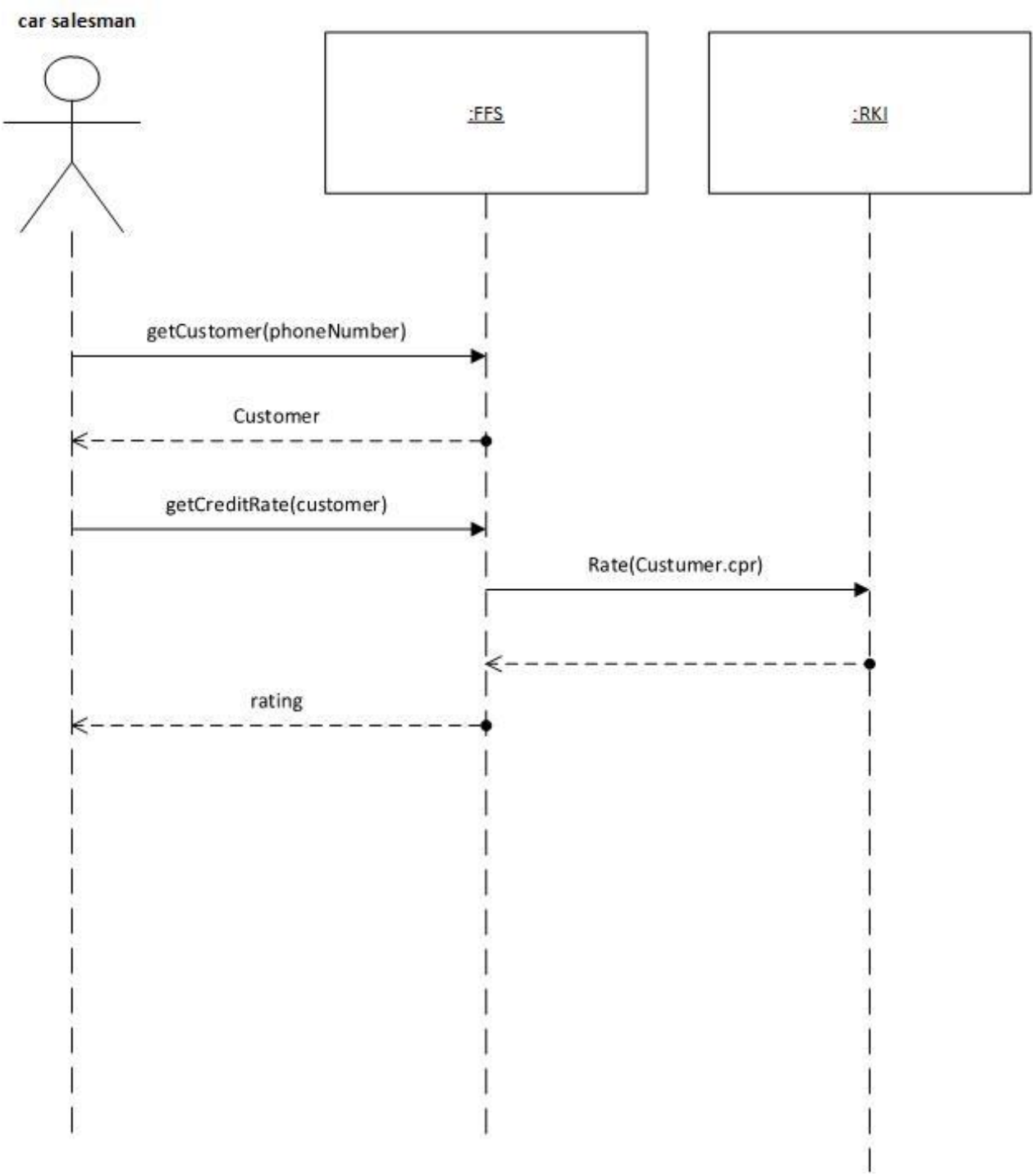
l.salesmanId er blevet sat til salesmanId.

customer.hasActiveLoan er blevet sat true

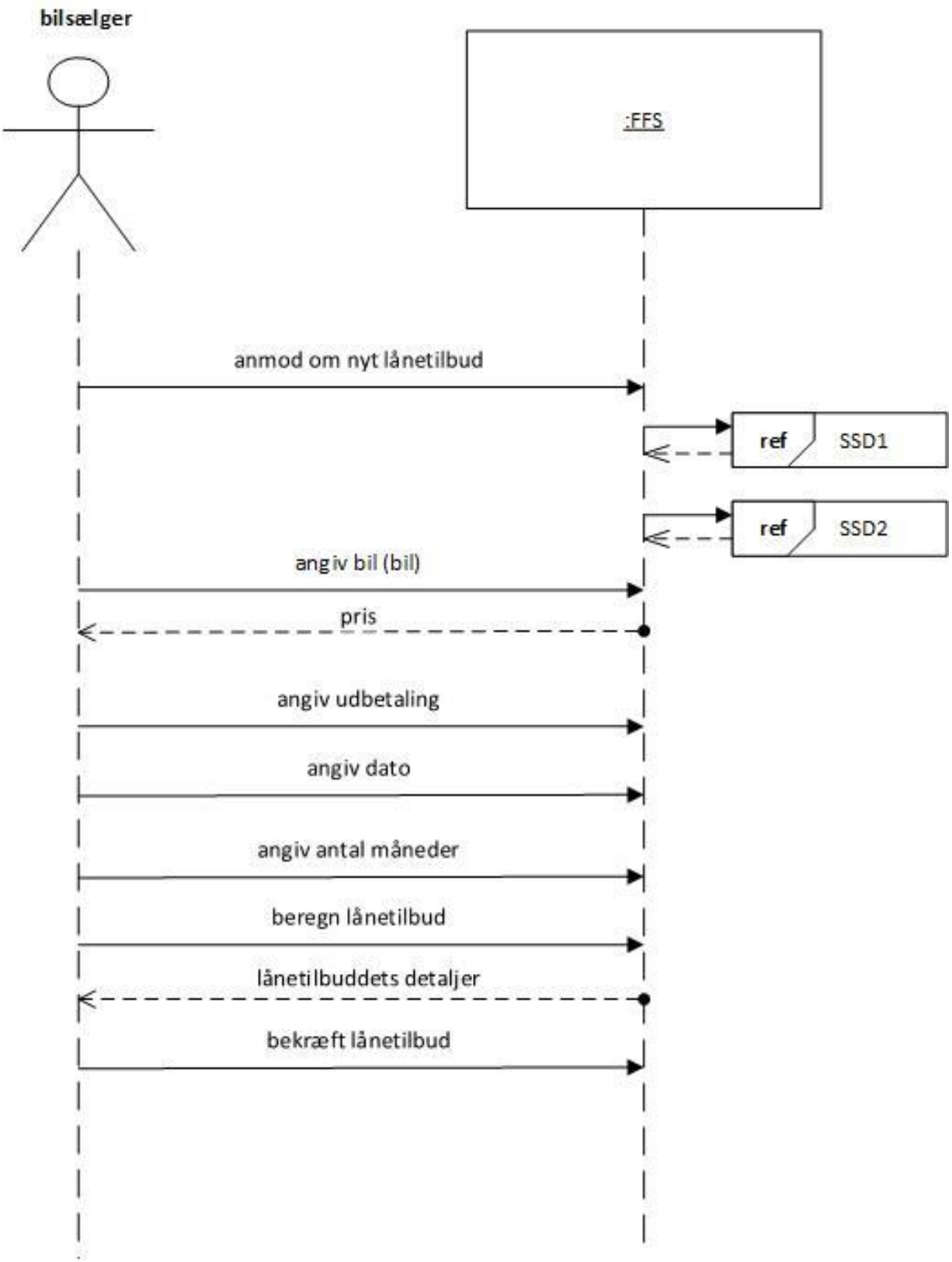
Bilag 13 - Operationskontrakter

Bilag 14 - Systemsekvensdiagram

SSD - UC1

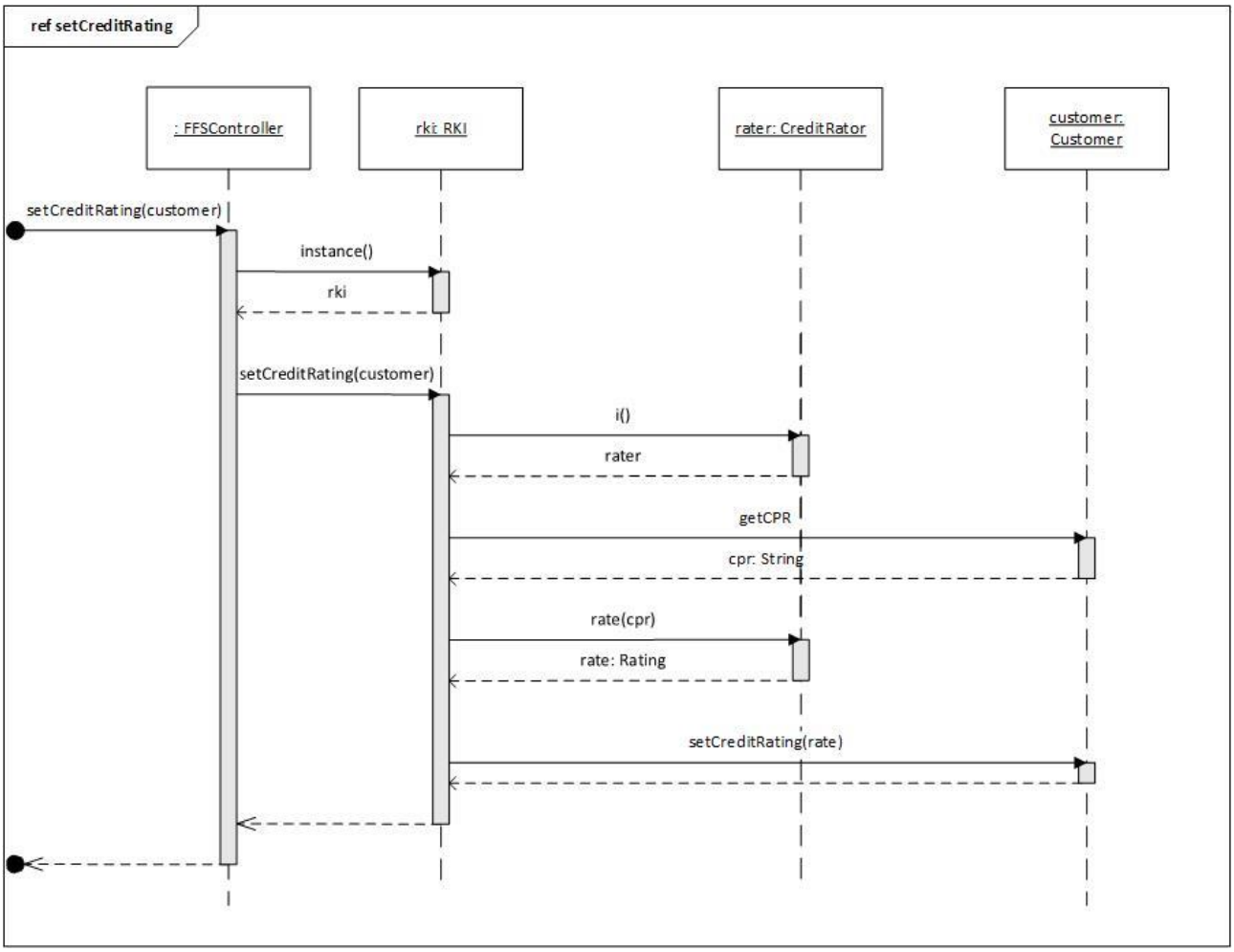


SSD - UC3

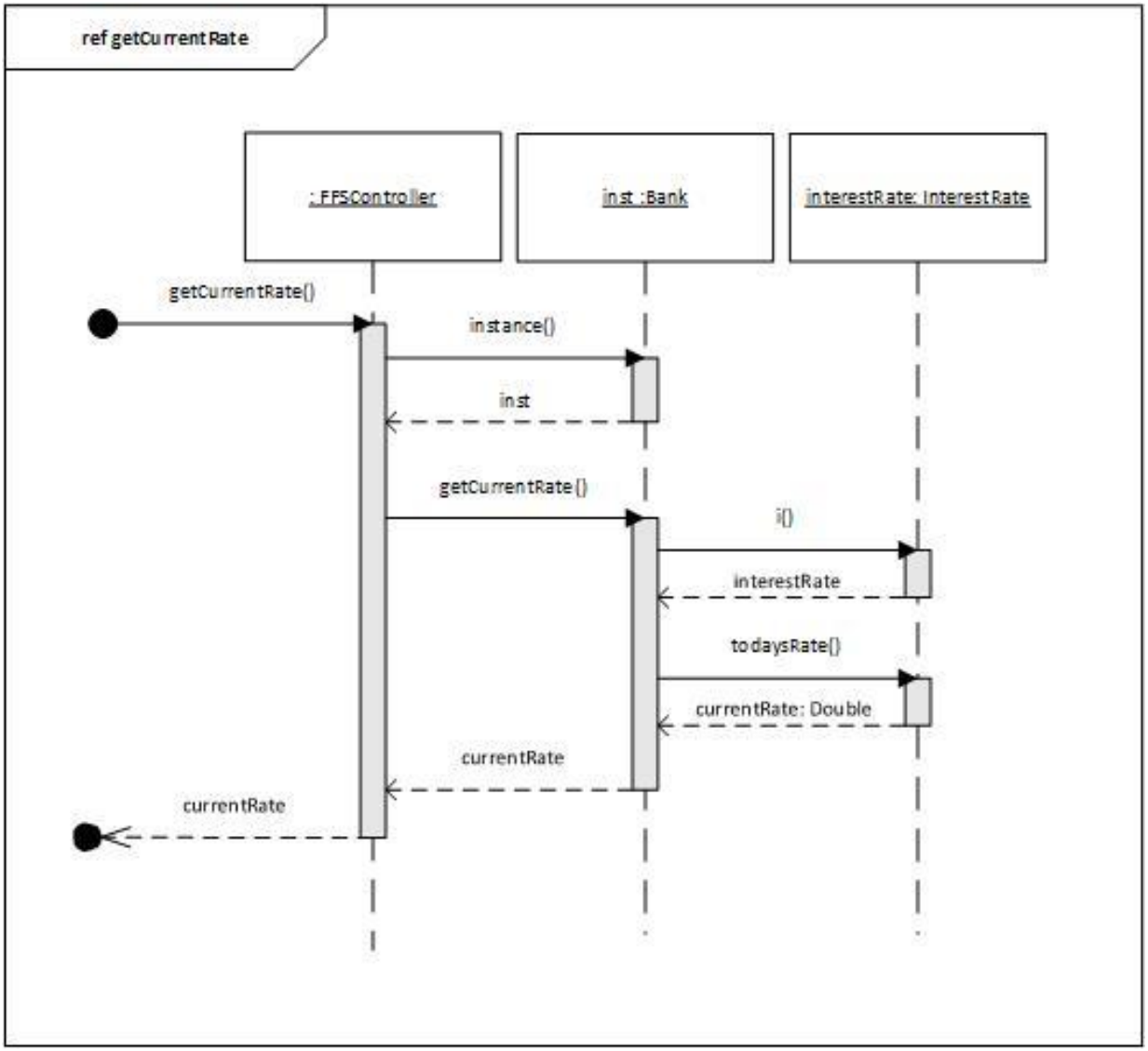

bilsælger
:FFS
anmod om nyt lånetilbud
ref
SSD1
ref
SSD2
angiv bil (bil)
pris
angiv udbetaling
angiv dato
angiv antal måneder
beregn lånetilbud
lånetilbuddets detaljer
bekræft lånetilbud

Bilag 15 - Sekvensdiagrammer

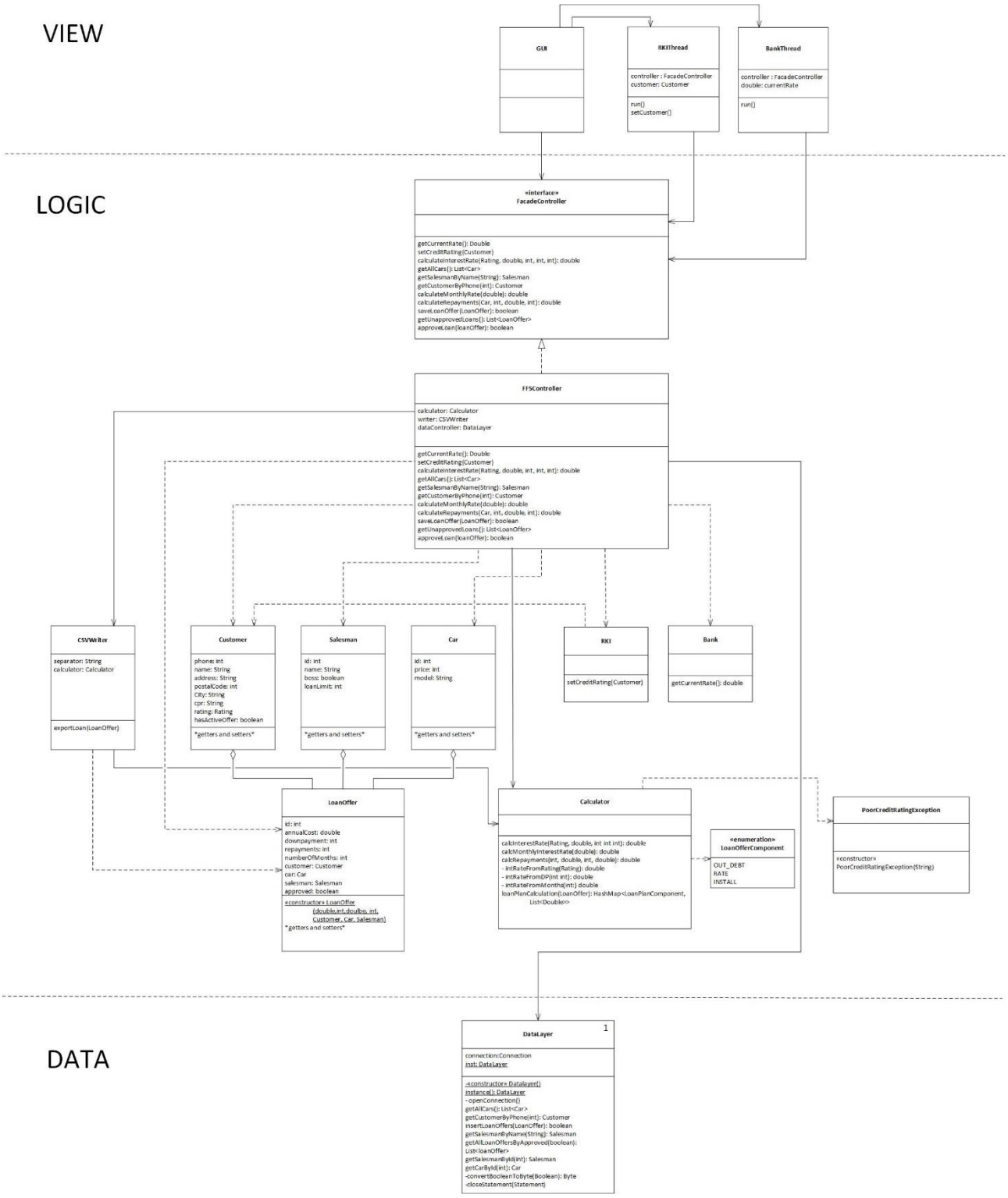
SD1 - UC1



SD2 - UC2

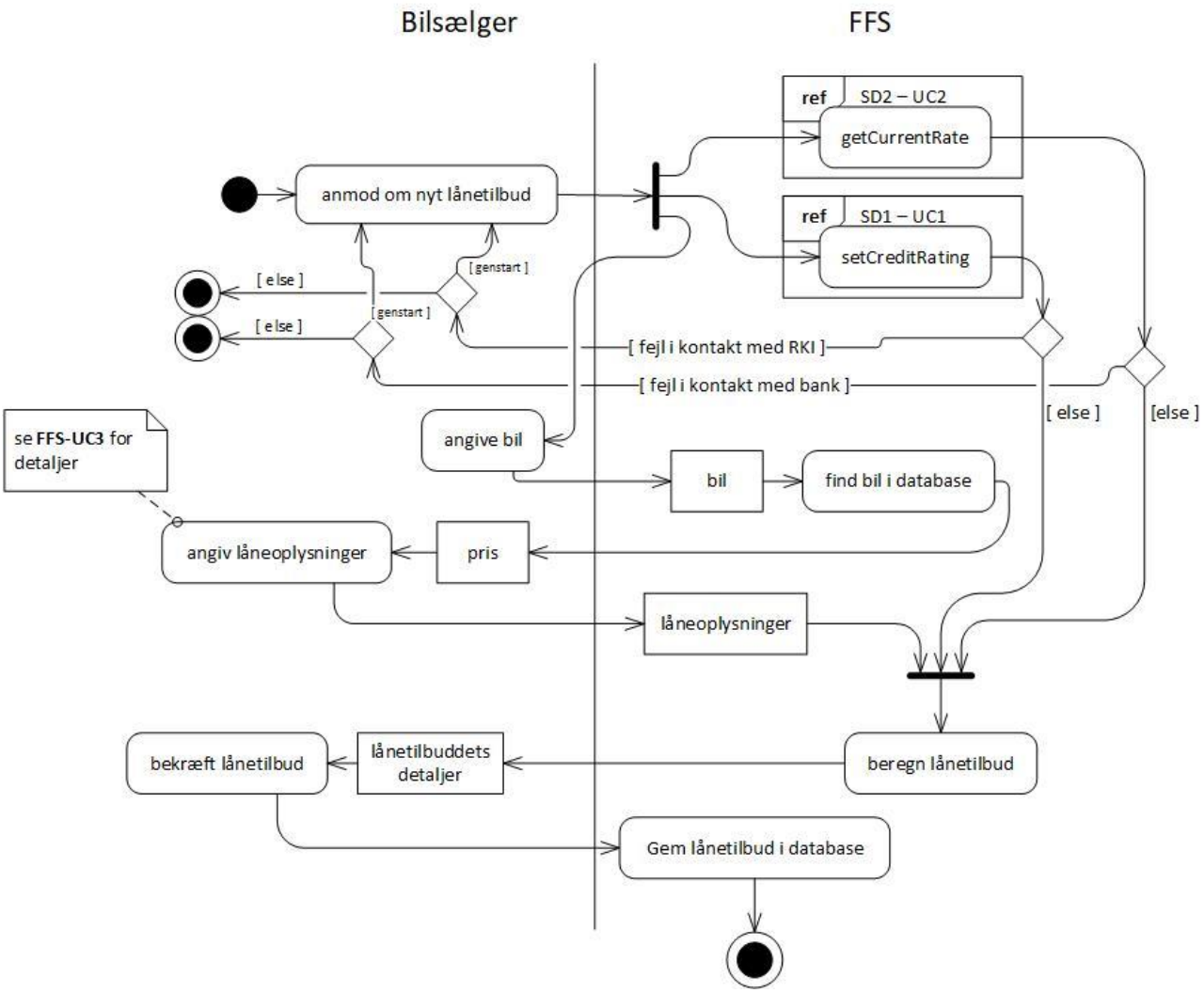

ref getCurrentRate
: FFSController
inst :Bank
interestRate: InterestRate
getCurrentRate()
instance()
inst
getCurrentRate()
i()
interestRate
todaysRate()
currentRate: Double
currentRate
currentRate

Bilag 16 - Klassediagram



Bilag 17 - Aktivitetsdiagram

AD - UC3



Bilag 18 – Data ordbog

Låneanmodning: En kunde anmoder forhandling om finansiering til at købe en Ferrari.

Økonomimedarbejder: En økonomimedarbejder undersøger kundens kreditværdighed hos RKI.

Kreditværdighed: i vores system findes kreditværdighederne A, B, C og D, disse tildeles kunden afhængig af hvor gode betalere de er. Algoritmen benytter disse til at udregne lånetilbud. Renten på kundens tilbud vil afhænge af deres kreditværdighed.

RKI: Expedias RKI-register sikrer danske virksomheder mod økonomiske tab og dårlige betalere. Registret består bl.a. af en database, hvori virksomheder kan registrere og søge efter dårlige betalere. Registret kan både bruges forebyggende ved indgåelse af købsaftaler eller opfølgende, hvor virksomheder adviseres, hvis en eksisterende kunde registreres som dårlig betaler. Ferrari har opstillet krav til hvilken kreditværdighed de forventer af deres kunder, og hvordan denne påvirker lånetilbud.

Aktuelle rentesats: den rentesats som den er ændrer sig hver dag og satsen er afhængig af banken.

ÅOP: Årlig omkostning i procent, er et udtryk for alle årlige reelle omkostninger (udtrykt i procent) i forbindelse med et lån og skal oplyses overfor forbruger i alle kreditforhold.

Udbetaling: det beløb kunden udbetaler inden begyndelsen af et lånetilbud.