

Juni 2015

Førsteårsprøve

FerrariFinanceSystem

Datamatiker – Erhvervs Akademi MidtVest

Vejledere:

Hans Iversen

Anders Westfall Reinholdt Petersen

Flemming Koch Jensen

Antal tegn: 79.290 tegn

Louise Niemann Hansen

Henrik Broe Henriksen

Enok Nørager Mikkelsen

Karsten Heino Karlsen

HE15DMU-2S14

Indhold

Indledning	3
Unified Process (Louise)	4
Hvad er UP?	4
Unified Process i et lille projekt	4
Projekt Styring	5
GitHub (Enok)	5
Faserne (Louise).....	6
Business Process Reengineering(BPR) (Henrik).....	7
BPR Modeller	8
Use Case Diagram.....	8
Objektmodel.....	8
Inception (forberedelsen) (Louise).....	10
Visionsdokument (Enok).....	10
Vision	10
Interessent analyse.....	11
Featureliste.....	13
Supplerende Kravsspecifikationer	13
Elaboration (etableringen)(Louise).....	14
Use Case Model (Louise)	14
Domænemodel (Karsten)	2
Dataordbog (Karsten)	2
Systemsekvensdiagram (Henrik)	3
Operationskontrakter (Henrik).....	4
Aktivitetsdiagram (Henrik)	5
Sekvensdiagram (Enok)	6
Klassediagram (Enok)	8
Samspil mellem Sekvensdiagram & Klassediagram.....	9
Brugergrænseflade (Louise)	9
Logisk arkitektur (Karsten).....	10
Teknologi	11
Trådprogrammering (Karsten).....	11
GRASP (Louise).....	13
Databaser (Enok)	16
IT Sikkerhed (Enok).....	18

Design Patterns (Henrik).....	19
Introduktion til Design Patterns	19
Brug af Design Patterns	19
Typer af Design Patterns	19
Kvalitetssikring.....	22
Test (Karsten).....	22
Testparadigmer (Karsten).....	24
TDD – Test Driven Development (Karsten).....	27
Whitebox test (Louise)	29
Reviews (Louise)	29
Construction (konstruktionen) (Louise).....	31
Transition (overdragelsen)(Louise).....	31
Konklusion	32
Litteraturliste.....	33

Indledning

Ferrari Finance System vil være vores løsning, på denne måneds Case.

Igennem udformningen af projektet vil vi benytte os af Unified Process som systemudviklingsmetode.

Dertil vil vi benytte os af versioneringssystemet Git.

Vi er bestående af et Team på fire mennesker, der vil arbejdet sig gennem de normale faser i en iterativ, inkrementerende udviklingsmetode.

Vi vil starte med Business Project Reengineering, hvor vil identificerer virksomhedens nuværende fremgang. Derefter vil vi gå i en kort Inception fase, hvor vi vil identificere, de handlinger, som det nye system skal kunne håndterer, samt udvikling af visionsdokumentet.

Derefter vil vi overgå til Elaboration, der formentlig vil blive vores længste fase, her vil vi sørge for at få dokumentering i orden til implementering. Dette gøres vha. artefakterne fra OOA og OOD, hvor vi benytter os af UML notation.

Når al dokumentation af koden er på plads, vil vi overgå til Construction, der formentligt ikke er nogen ret stor fase i dette projekt, grundet Casens størrelse.

Transition vil vi formentligt ikke komme i, ved mindre vi vælger at anmærke aflevering samt eksamination som Transition, da der ellers ikke er så meget overlevering til kunden.

Vi vil sideløbende med dokumentering overveje, hvilke designmønstre, der vil være hensigtsmæssigt for Ferrari Finance System og endvidere pakkestrukturen og arkitekturen bag dette. Casen indebærer desuden anvendelsen af concurrency, og vi vil overveje hvordan vi udnytter Javas muligheder mest fornuftigt.

Det er planen, at kvalitetssikring skal forløbe sideløbende med udvikling, således at både test af systemet samt reviews indgår som en naturlig del af projektet.

Der bør løbende være en verificering af de fremstillede dokumenter.

De modeller, der benyttes i rapporten, vil desuden være placeret som bilag i deres fuld længde.

Unified Process (Louise)

Hvad er UP?

Unified Process er en systemudviklingsprocess, der er udviklet i slut '90erne af Ivar Jacobsen, Grady Booch og James Rumbaugh.

UP anvender UML notation, og er desuden brugt og omtalt af bl.a. Craig Larman.

UP er særligt velegnet i meget store projekter, der enten har mange mennesker involveret, eller strækker sig over længere tid.

Der er tale om en iterativ udviklingsmetode, hvilket vil sige, at man bevæger sig i cirkler, og hele tiden holder kontakt med kunden og sørger for, at kravene er up to date.

Programmet udvikles på denne måde i mindre steps (inkrementerende), hvilket giver udviklerne mulighed for, at udnytte den viden de opnår igennem projektet.

På denne måde undgår man tidlige problemstillinger, som er kendt fra vandfaldsmodellen, når denne anvendes på større projekter, der strækker sig over længere tid.

Unified Process i et lille projekt

Her er projekter som vores, ikke ret velegnede til at kører UP, da planen med bl.a. projektplan og faser kan falde lidt til jorden. Når man sidder i et så lille team, at det er muligt at opretholde god kommunikation med alle i gruppen.

Hvis vi kigger på projektets opsætning, er der flere steder, hvor vi i stedet ville have haft gavn af nogle af principperne fra Scrum, og fx har vi de dage vi har været på skolen, næsten dagligt haft et lille morgenmøde, hvor vi har snakket om, hvad vi lavede den foregående dag, og hvad dagen skal forløbe med.

Desuden ville et Scrumboard formentlig have bragt mere værdi i vores projekt, end vores projektplan gjorde, da vi har manglet fleksibiliteten i projektplanen, og derfor ofte fået rykket el. utsat små opgaver. Vi har desuden overset et par opgaver, der ville være husket bedre med en TODO, så de kunne være udført når gruppen samlet set var klar til det.

En overordnet fejl vi har begået, der har spændt ben for vores projekt, er at glemme at uddele roller. Noget af det der tilfører megen værdi til UP er rollefordeling, og at folk holder sig i disse roller, mens de er i den.

Dette ville have hindret os i, konstant at udvide produktet.

Projekt Styring

Vi har som tidligere nævnt ikke fået ført projektplanen godt med. Til gengæld er det lykkedes os at bruge risiko analysen godt.

Vi har haft en del fravær i gruppen, men en kombination af en god politik i gruppen, samt en forudseende risikoanalyse, lykkedes det os, at overkomme disse, og have et overblik over, hvad der skulle ske i denne situation. Desuden har analysen haft en forebyggende effekt, da vi har fået snakket mulige grundet og været åbne omkring dette..

Risiko	Sandsynlig (SS)	Konsekvens (K)	Prioritering (SS * K)	RMM	Revideret SS	Revideret konsekvens
Hardwareproblemer	1	2 timer	2			
Tab af data	2	2 timer	4			
Uloversensstemmelser	4	4 timer	12			
Tab af motivation	3	6 timer	18			
11+ dages fravær	1	24 timer	30	Arbejder hjemmefra. Det er både gruppens og den syges opgave at være med til at kommunikere opgaver. Begrenset deltag.	1	4
Overordnet tab af motivation	1	60 timer	60	Kommunikation om situationen og løsning	1	6
5 - 10 dages fravær	3	24 timer	78	Arbejder hjemmefra. Det er både gruppens og den syges opgave at være med til at kommunikere opgaver.	3	4
Udfordringer i	4	20 timer	80	Udarbejd prototyper af	1	20

GitHub (Enok)

Github er et webbaseret Git repository versioneringshåndterings værktøj, som man bruger til at håndtere kildekode og evt. dokumenter til sit projekt. Det giver en mulighed for let at udgive forskellige versioner af sit program, efterhånden som projektet skrider frem. Desuden hjælper det en til at lade flere personer arbejde på den samme kode, og lade dem ændre i koden så at sige live.

Der arbejdes i branches, som der commites til, og pulles fra. Og når man ønsker at lave en ny version, så oprettes en ny branche, som man så kan arbejde videre i. Men den gamle beholdes ofte, så man kan se, hvor meget reel t blev udført osv.

I vores projekt har vi brugt en branche strategi hvor vi har kørt vores branches i forhold til implementering, men set i bagklogskabens klare lys, vil vi måske have fået mere ud af at have en branche for hver eneste iteration. Dette ville have gjort at vi virkelig kunne se udviklingen i vores artefakter. Hvilket, ud fra et undervisningssynspunkt, ville have været bedre.

Fordele ved Git i forhold til almindelig deling af filer:

I forhold til almindelig fil deling, som fx Fronter, eller Dropbox, giver GitHub os nogle gode fordele. For det første er Git ikke bare lavet til fildeling, men specifikt lavet til deling af code, samt det at man kan udgive releases/udgivelser af sit program igennem hele forløbet.

Desuden giver det muligheden for at man kan arbejde i forskellige branches og bruge forks som begge dele giver gode fordele i forhold til at side flere i den samme kode.

Derudover er det værd at nævne ang. Git er at det holder styr på forskellige version af filer. Dvs. at så frem at 2 personer ændre i den samme fil, og begge forsøger at commite dem til GitHub, vil

Git lave en merge konflikt hos den der forsøgte at lægge filen op sidste. Og det er så brugerens opgave at forsøge at løse denne merge konflikt, men ofte giver GitHub et forslag til løsningen.

En sidste ting der er værd at nævne, er at i forhold til vort projekt, så har vi arbejdet med Eclipse igennem det meste af projektet, og den understøtter også GitHub, hvilket betyder at vi kan merge, committe og pull direkte i Eclipse.

Det har dog ikke været uden besvær at vi har brugt Git. Hvis vi ser på de fejl meddelser vi har haft igennem projektet, så har vi været fejl listen rigtig godt igennem. GitHub har dog givet os meget mere værdi ud fra de funktioner der har været i Git i forhold til det besvær vi til tider har haft med det.

Mappe struktur

Det at UP har været så stor en del af vort projekt, gjorde at vi også valgte at opbygge af mappe struktur på Git skulle være efter UP discipliner. Dette gjorde at vi klart viste hvor hvilke filer og artefakter osv. skulle ligge.

Under de enkelte discipliner blev der så oprettet under-mapper til de enkelte artefakter, således at det ikke kom til at ligge hulter til bulter med alt for mange filer i en mappe.

Alternative versioneringssystemer

Af alternative værkstøjer kan nævnes: CodePlane, BitBucket, Source Forge og GitLab.

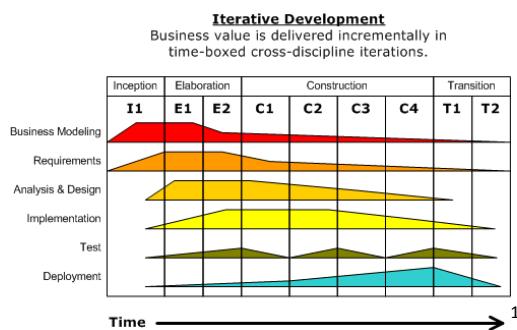
Grunden til at vi har valgt GitHub er mest at det er det vi har brugt i undervisningen.

Faserne (Louise)

Unified Process er inddelt i fire overordnede faser: Inception, Elaboration, Construction og Transition.

Faserne er det øverste led i projektets plan. Der er forskel på, hvor længe man er i de forskellige faser. I hver fase er desuden delt ind i en eller flere Iterationer, som det er afbilledet på nedenstående billede. Der er forskelligt fra projekt til projekt, hvor mange iterationer der ligger i hver fase. I vores projekt har der fx været én Iteration i Inception (Hvilket er meget typisk) og derefter fem iterationer i Elaboration, da vi simpelthen ikke har været klar til at gå i Construction, grundet manglende dokumentation.

Foruden faserne, vil der foreligge noget business modelling i starten afprojektet. Det er dog ikke sikkert, at det er udviklingsteamet, der sidder med denne opgaver. I nogle tilfælde, vil der være tale om en opgave, der tager stilling til, om der skal foretages en Reengineering, før projektet er hyret ind til at udvikle det nye system.



¹ http://da.wikipedia.org/wiki/Unified_Process#/media/File:Development-iterative.png

Business Process Reengineering(BPR) (Henrik)

BPR indebærer ændringer i struktur og processer i en eksisterende forretningsgang. Hele det eksisterende informationsteknologiske og organisatoriske system kan blive ændret, i forbindelse med BPR.

Definition:

- *En forretningsproces(business process) kan defineres som faste skridt, der bliver fulgt for at udføre en given opgave i en virksomhed. BPR kan derfor defineres som radikal omstrukturering af en eksisterende forretningsproces, med henblik på at opnå store forbedringer indenfor f.eks. omkostningsniveau eller hastighed.*

BPR kan med fordel anvendes i virksomheder, der har et ønske om at effektivisere, hele eller dele af, sin nuværende forretningsgang.

BPR undersøger om den/de nuværende processer i virksomheden er forældede/ubrugelige, og fremkommer med forslag til komplette nye processer til implementation ved at starte forfra. For at gøre dette, skal man se bort fra nuværende processor, og fokusere fuldstændigt på nye.

Dette kan medføre at:

- flere jobfunktioner bliver lagt sammen til en.
- den enkelte medarbejder får mere ansvar til at tage beslutninger.
- processen bliver udført i mere logisk rækkefølge og det sted i virksomheden, der giver mest mening.
- kunden har kun én kontakt i virksomheden.

BPR processen starter med, at modellere den eksisterende proces vha. Use Case Model og Objekt Model. Dette kaldes også "Reverse Business Engineering". Derefter modelleres den fremtidige proces, også kaldet "Forward Business Engineering".

Under BPR kan man f.eks. have fokus på følgende:

- **Kunder**, man laver processer med fokus på kundeservice for at nedbringe klager eller behandlingstid.
- **Hastighed**, man identificerer nøgle processer, og fokuserer på at nedbringe tiden, disse tager at udføre.
- **Kvalitet**, fokuserer på at frembringe ensartet kvalitet, med kvalitetskontrol indbygget i de enkelte processor, frem for en enkelt afdeling.

Forventede resultater for virksomheden, efter succesfuld gennemførsel af BPR kan være:

- reducering af omkostninger.
- bedre kvalitet.
- effektiviserede arbejdsgange.

De drastiske ændringer ved gennemførsel af BPR, kan for visse medarbejdere betyde tab af job eller nye arbejdsfunktioner.

BPR Modeller

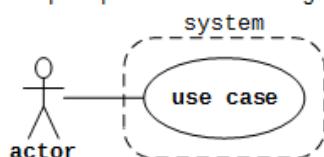
I det følgende vil jeg beskrive hvilken dynamisk og statisk model, der er brugt i BPR i denne opgave. Jeg vil kort gennemgå dem og give et simpelt eksempel til hvert. Hvordan vi har brugt disse modeller i vores opgave, følger efter gennemgangen.

Den dynamisk model(også kaldet proces model) vi har brugt hedder Use Case Diagram.

Use Case Diagram

Et use case diagram anvendes til beskrivelsen af funktionaliteten i det nye system. Diagrammet bruges til at vise forbindelsen mellem en bruger (en person eller et andet system, også kaldet Actor) og en specifik use case i det aktuelle system.

Eksempel på use case diagram:



En use case beskriver en funktionalitet i virksomheden, der repræsenterer et betydningsfuldt arbejde for virksomhed og actor. Et hurtigt eksempel kan være "opret ordre".

Et use case diagram skal suppleres med en tekstuel beskrivelse, af hver enkelt identificeret use case.

Den **statiske model**(også kaldet struktur model) vi har brugt hedder objekt model.

Objektmodel

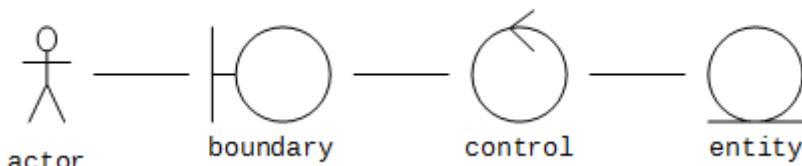
Dette benyttes til at give et overblik over ting internt i virksomheden, som skal indgå i realiseringen af en use case. Dette kan være personer, formularer, andre systemer osv. I BPR sammenhæng benyttes ofte Ivar Jacobson's notation med boundary-, control- og entity-objekter.

Boundary-objekt er et objekt, der interagerer med en eller flere actors(en person eller et andet system). Det kan f.eks. repræsentere en ansat i virksomheden, eller en formular, der bliver udleveret/videresendt. Actors kan kun kommunikere igennem boundary-objekter. Boundary-objekter kan kommunikere direkte med andre boundary-objekter, actors og control-objekter.

Control-objekt er et udførende objekt. Dette kan repræsentere personer, afdelinger eller andre systemer i virksomheden. Det kontrollerer adgangen til de forskellige entity-objekter. Control-objekter kan kommunikere med andre control-objekter, boundary-objekter og entity-objekter.

Entity-objekt er et objekt der opbevarer data/oplysninger. Eksempler kan være records i database, blanketter mv. Entity-objekter kan kun kommunikere med control-objekter.

Eksempel på objektmodel:

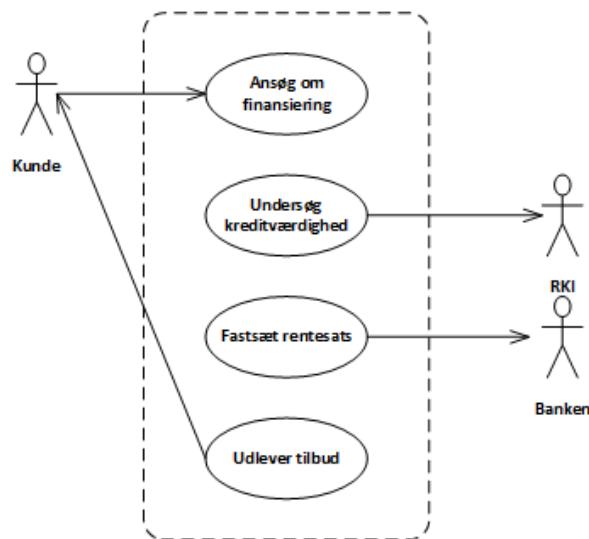


Brug af BPR modeller i opgaven

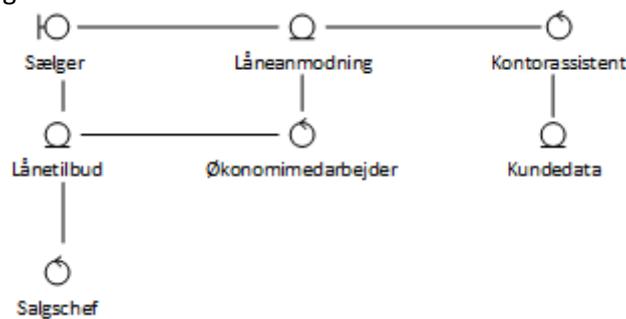
Vores opgave begyndte med en modellering, af det eksisterende system hos den regionale Ferrari-forhandler.

Til dette arbejde benyttede vi use case diagram og objekt model.

Vores use case diagram, viser virksomheden set udefra som det ser ud før reengineering, med de eksterne actors, der er behov for. Pilene er benyttet, som et alternativt værktøj, der skaber overblik over handlingernes udførelse.



Vores objektmodel viser tingene internt i virksomheden:



Der efter opstillede vi et use case diagram, over de identificerede use cases i det kommende system, som vi kunne forestille os, det skulle se ud.

Inception (forberedelsen) (Louise)

Inception den første fase. Denne fase er desuden også den klart korteste. I denne fase fastslås det, hvad meningen er med projektet, der udarbejdes en business case, Use Cases identificeres, risiko adresseres og man estimerer på, hvad det vil koste at lave.

Når man forlader Inception-fasen bør der altså foreligge:

- Et visionsdokument
- En Use Case model (med en liste over de aktører og Use Cases, der kan identificeres på så tidligt et tidspunkt)
- En påbegyndt Dataordbog
- En Business Case (indeholdende lidt om forretningen, et succes kriterie og en prognose af den finansielle.)
- En risikoanalyse
- En projektplan

Selve Inception fasen forløb ret planmæssigt i vores projekt og strakte sig over lidt over en dag. Vi havde dog glemt risikoanalysen, der derfor tilkom ret sent i forløbet.

Man kunne have argumenteret for, at udarbejdelse af vores primære Use Case på Fully dressed niveau skulle ligge i denne fase, samt udvikling af prototyper, på de ukendte områder, men grundet projektets størrelse, valgte vi at lægge dette i Elaboration.

Visionsdokument (Enok)

Visionsdokumentet er udgangspunktet i nye system. Det er bestående af:

- Vision: En kort tekst som beskriver idéen og visionen for det software vi overvejer at udvikle.
- Interessent Analyse: En liste over interesser og hvad de hver især er interesseret i, i forhold til vort kommende program.
- Featureliste: En liste over de features som vort program skal have.

Vision

Er en kortfattet tekst der beskriver fremtidsvisionen for det nye stykke software, på en måde så vi ikke sammenligner det nye system med det gamle og ikke ligger os fast på nogen bestemt form for teknologi.

Desuden skal visionen meget gerne afbillede de interesser som vore interesserter har.

I vores vision har vi forsøgt både at lave en inspirerende tekst, som vil kunne motivere alle i projektet til at lave et super godt program, men også afspejle det firma som køber programmet, nemlig Ferrari som netop har et helt bestemt brand, som vi gerne så afspejlet i programmet.

Vi forestiller os et FerrariFinanceSystem

Vi forestiller os et FerrariFinanceSystem, som afspejler det brand som Ferrari er. Selve programmet skal understøtte det faktum, at Ferrari er et High-end produkt. Systemet skal være i stand til korrekt og sikkert at behandle oplysninger. Systemet skal være med til at skabe en unik følelse omkring købet af kundens nye Ferrari, der yderligere forstærkes i og med, at vi effektivt gennemfører låneforløbet, således at kunden er i stand til at køre væk i sin nye bil, den samme dag. Systemet skal være intuitivt, og derved være med til at give sælgeren mere frihed og overskud til at give en professionel betjening af kunden.

Interessent analyse

Er en liste over de interessenter der har interesse i det kommende stykke software.

Eksempler kan fx være Direktør/Direktion som køber softwaren, Brugere af programmer, SKAT eller andre myndigheder, der fx kan stille krav til databeskyttelse eller har interesse i at moms og skatter m.m. bliver beregnet korrekt.

Grunden til at man analysere disse interessenter er netop for at være sikker på at man tager hensyn til så mange af deres interesser så muligt. Således at man ikke pludselig opdager at programmet mangler en funktionalitet eller egenskab som en central interessent forespørger.

Interessentanalyse

Direktør:

- At salgsteamet for solgt mest muligt.
- At låneanmodning bliver korrekt registreret med det samme.
- At renten er korrekt beregnet.
- At data ikke går tabt i processen.
- At kontakt med bank/RKI forløber hurtigt og uden tab af data.
- At låneanmodning fra samme kunde ikke bliver behandlet flere gange.

Sælger:

- At jeg sælger mest!
- At låneanmodning kan godkendes hurtigst muligt.
- At låneanmodning fra samme kunde ikke bliver behandlet flere gange.
- At låneanmodning bliver korrekt registreret med det samme.
- Lettest mulig arbejdsgang

Kunden:

- At få svar på låneanmodning hurtigst muligt.
- At renten er korrekt beregnet.
- At oplysningerne bliver opbevaret sikkert.

Bank:

- Korrekt brug af deres service.

RKI:

- Korrekt brug af deres service.

Datatilsynet:

- At data opbevares i overensstemmelse med persondataloven.

Forslag til ændringer

- Optælle af emodning om låneanmodning

I forhold til vores kommende program har vi lavet analysen ovenfor.

I analysen indgår både interne interessenter, så som brugeren og de der skal betale for programmet (ofte en direktør eller direktion), samt eksterne interessenter, både kunden, men også de firmaer hvor vi benytter en service de har stillet til rådighed. I vores tilfælde er det Banken og RKI.

Endelig kan der også være lovgivere eller andre myndigheder, som vi bør tage hensyn til i udformningen af programmet. I vores tilfælde Datatilsynet.

"Vi forestiller os et FerrariFinanceSystem, som afspejler det brand som Ferrari er. Selve programmet skal understøtte det faktum, at Ferrari er et High-end produkt. Systemet skal være i stand til korrekt og sikkert at behandle oplysninger. Systemet skal være med til at skabe en unik følelse omkring købet af kundens nye Ferrari, der yderligere forstærkes i og med, at vi effektivt gennemfører låneforløbet, således at kunden er i stand til at kører væk i sin nye bil, den samme dag. Systemet skal være intuitivt, og derved være med til at give sælgeren mere frihed og overskud til at give en professionel betjening af kunden."

Direktør:

- At salgsteamet for solgt mest muligt.
- At låneanmodning bliver korrekt registreret med det samme.
- At renten er korrekt beregnet.
- At data ikke går tabt i processen.
- At kontakt med bank/RKI forløber hurtigt og uden tab af data.
- At låneanmodning fra samme kunde ikke bliver behandlet flere gange.

Sælger:

- At jeg **sælger mest!**
- At låneanmodning kan godkendes hurtigst muligt.
- At låneanmodning fra samme kunde ikke bliver behandlet flere gange.
- At låneanmodning bliver korrekt registreret med det samme.
- Lettest mulig arbejdsgang

Kunden:

- At få svar på låneanmodning hurtigst muligt.
- At renten er korrekt beregnet.
- At oplysningerne bliver opbevaret sikkert.

Bank:

- Korrekt brug af deres service.

RKI:

- Korrekt brug af deres service.

Datatilsynet:

- At data opbevares i overensstemmelse med persondataloven.

Igennem ovenstående tabel forsøger vi at vise det samspil, der er mellem visionen og interessentanalyse. Når vi står med det endelige program, ved afslutningen af projektet, bør vi være i stand til at holde det op imod hele visionsdokumentet, og se det afspejlet både i visionen, interessentanalysen og featurelisten.

Featureliste

Er en kort tekst der beskriver de funktioner, som det kommende program skal have, men på en sådan måde at vi heller ikke her ligger os fast på bestemte metoder eller teknologi, med mindre at det direkte er beskrevet i kravene til vort kommende program.

Funktioner:

- Korrekt brug af denne service.

Dekompatibilitet:

- At data opbevares i overensstemmelse med persondatabes.

Featureliste

- Oprettelse af anmodning om lånetilbud
- Håndtering af kunde
- Oprettelse af tilbud
 - Forespørgsels om kreditværdighed
 - Forespørgsels om rentesats
 - Fastsættelse af rentesats for hvert tilbud
- Eksport af lånetilbud og afdragsordning
 - CSV format

I forhold til vores feature liste, havde vi én bestemt ting, som var blevet pålagt af krav til programmet, nemlig at det skulle kunne eksportere lånetilbud m.m. til en CSV fil.

Så ud over de centrale ting i programmet, måtte dette krav også med i feature listen.

Supplerende Kravsspecifikationer

Når man har lavet visionsdokumentet, og især interessent analysen, kan det være særligt opagt at man også kigger på Supplerende Kravsspecifikationer.

Supplerende Kravsspecifikationer er de ikke funktionelle krav som der kan stilles til programmet. Disse kategoriseres ofte i 4 kategorier:

- **Reliability:** Hvor pålidelig er systemet. Det kan fx være krav til hvor meget man forventer systemet skal være til rådighed. Et krav her kunne fx være 99,9% op-tid på systemet.
- **Availability:** Hvor let systemet er at bruge, fx for 1. gangs brugere. Et krav her kunne være at en første gangs bruger skal kunne foretage et salg på max. 5 minutter.
- **Maintainability:** Hvor let er vedligeholde. Der kan fx være krav til tredeling i software arkitekturen, eller til at nye moduler let kan installeres ind i det nye program.
- **Performance:** Hvad programmet skal kunne klare. Et typisk krav her kan være er mængden af beregninger pr sekund, eller hvor langtid en backup session fx højest må vare.

Grunden til at det er logisk at lave denne liste efter visionsdokumentet, er fordi at de krav man opstiller her ofte er udledt af de interesser som en interessent har.

Hvad er fælles for alle de krav der opstilles under Supplerende Kravsspecifikationer, er at de er målbare. Dvs. at når vi er færdige med vort software, kan vi teste det og sammenligne resultatet med vore krav her og sikre os at alle de ikke-funktionelle krav er opfyldt.

Elaboration (etableringen)(Louise)

Elaboration er den mest kritiske fase. Det er i denne fase man beslutter, om man ønsker at færdiggøre projektet, eller om der er nogle problematiske områder, der gør at vi vælger at droppe projektet (dette kan både være grundet en høj risiko, der er forbundet med projektet, eller at projektets færdige omkostninger ville overstige det budget, der er estimeret i Inception). I denne fase skal al beskrivelsen foreligge, desuden skal den mest væsentlige handling i programmet implementeres.

Når man forlader Elaboration, og bevæger sig ind i Construction, skal man være sikker på, at projektet føres til ende.

Når man forlader denne fase bør der foreligge:

- En Use Case model, der mindst er 80 % færdig
- De andre krav bør være beskrevne, her i blandt de ikke funktionelle krav.
- En fuld beskrivelse af Softwarets arkitektur
- Prototyper af nødvendige områder

Vores gruppe forlod aldrig Elaboration, da vi manglede indtil flere af de ting, vi burde have før dette skete. Fx var vores Use Cases ikke 80 % beskrevet før i den aller sidste uge, det samme gør dig gældende for krav.

Use Case Model (Louise)

Use Cases er artificer, der har indflydelse på mange andre artificer. Use Case modellen er opfundet af Ivar Jacobson, der også er en af grundlæggerne af UP, i 1986. Heraf grunden til at Use Cases spiller en meget centrale rolle i UP.

Omkring 10 % af de mest kritiske Use Cases bør være beskrevet allerede i Inception, og alle Use Cases bør være identificeret og i kort form, allerede inden visionen skrives.

Når man identificerer Use Cases bør man først kigge på systemet, og derefter hvem der er aktører her er.

I vores tilfælde kunne man spørge sig selv, hvorfor det er sælgeren, og ikke kunden der er primær aktør. Svaret på dette skal findes i systemet. Det er vores sælger, der er bruger af systemet, og selve salget til kunden er derfor ikke et mål i denne forstand. Sælgeren har til gengæld et mål, der hedder at sælge biler til kunderne, og i forbindelse med dette låne penge ud. Til dette skal han oprette et lån, som må siges at være vores mest centrale Use Case.

Der er tre forskellige tests, der kan udføres på en Use Case, for at se, om den opfylder de basale krav for at være en Use Case.

- **Boss Test**

Use Casen skal være en handling, som chefen ville være tilfreds med, at hans ansatte udførte hele dagen.

- **EBP Test (Elemental Business Process)**

Ligesom med Boss Testen skal Use Casen tilfører virksomheden en eller anden form for værdi. Det kan fx være at godkende låneanmodninger.

- **Size Test**

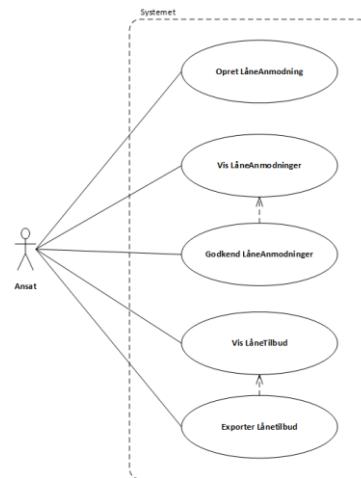
En Fully dressed Use Case vil som regelt være af en hvis størrelse, så hvis vi er i besiddelse af en Use Case, der kun indeholder et enkelt trin, vil den ikke bestå en Size Test.

- **Undtagelser**

Der kan være tilfælde, hvor en Use Case får lov at forblive en Use Case, på trods af, at den ikke består en test.

Der er både Use Case Diagrammer, Brief Use Cases, Casual Use Cases og Fully Dressed Use Cases.

Use Case Diagrammet er ikke en artifikat, der kan stå alene, men kan bruges i en kontekst, til at skabe overblik over sammenspillet mellem aktører og handlinger. Desuden bruges de som inputs til Use Case Teksten.



Aktøren vil indgå i Use Case teksten og de identificerede Use Cases vil være overskrifter på de nedskrevne Use Cases.

Brief Use Cases er det første, der skrives ned, og er egentlig bare en kort, tekstuel beskrivelse af forløbet, uden nogle afvigelser.

Anmod om Låneanmodning

Sælgeren anmoder systemet om at oprette et nyt lånetilbud. Systemet opretter et lånetilbud. Sælger angiver CPR-nummer. Sælger validerer information med kunden. Sælger angiver ønskede biltypen. Sælger angiver kundens ønskede udbetaling. Sælger angiver ønskede afdragsperiode

En **Casual Use Case** er på samme måde kort, men med afvigelser, og desuden mere struktureret opsat.

ITS - UI: Anmod om Låneanmodning

- Sælgeren er logget ind i systemet og klar til at betjene en ny kunde.
Der er en kunde til stede, der ønsker at optage et lån.

Sælgeren anmoder systemet om at oprette et nyt lånetilbud
Systemet opretter et lånetilbud
Sælger angiver CPR-nummer
Sælger validerer information med kunden
Sælger angiver ønskede biltypen
Sælger angiver kundens ønskede udbetaling
Sælger angiver ønskede afdragsperiode

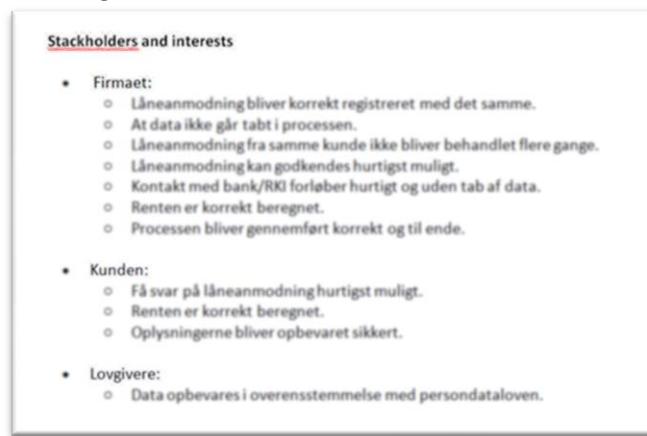
Hvis kunden ikke eksisterer i databasen oprettes kunden med navn, adresse, CPR-nummer, telefonnummer og e-mail.
Hvis der er kommentarer om tidligere problemer med kunden, afsluttes forløbet.
Hvis kunden får en speciel rabat, angiver sælgeren den nye salgspris.
Hvis kundens ønskede lånesum overstiger det sælgeren er autoriseret til at godkende, får sælgeren en notifikation om, at han ikke selv er bemyndiget til at udstede lånet.

En **Fully Dressed Use Case** er meget længere, og indeholder alt, der kan identificeres i forbindelse med handlingen.

Én af de vigtige ting, der skal identificeres i forbindelse med skrivning af Fully Dressed Use Case, er interesserter, og hvad deres interesser er.

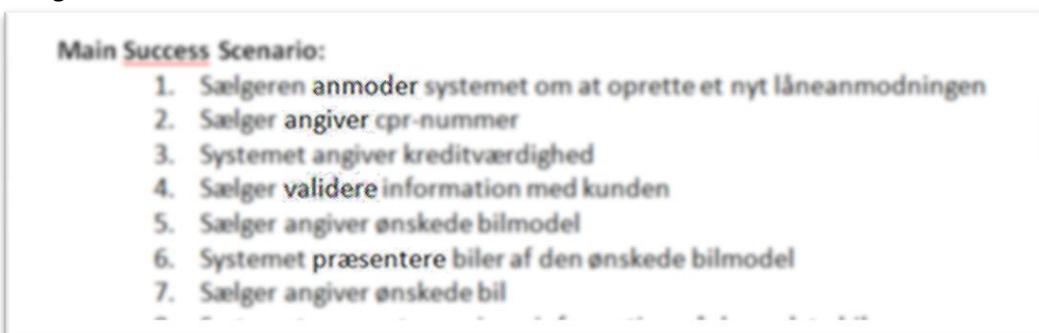
Her er det ikke kun de åbenlyse, som firmaet og kunden, der skal identificeres, men også udenforstående interesserter, så som lovgivere, skat, banken osv.

I denne Use Case kan vi se, at lovgiveren har en interesse, da der er en lovgivning, der skal overholdes. Havde der nu været tale om et salg, så havde SKAT været en interessent.



Ligesom når vi skriver vores Vision, er det vigtigt at overveje sprogbruget i en Use Case. Der bør lægges vægt på, at ordene er så neutrale som muligt, og at ordvalget ikke låser sig fast på en specifik fremgangsmåde. Dette valg bør nemlig være op til designerne i en senere fase i forløbet.

Derfor har vi brugt ord som "anmoder" og "præsenterer" i stedet for fx "Sælgeren trykker på "Opret låneanmodning""

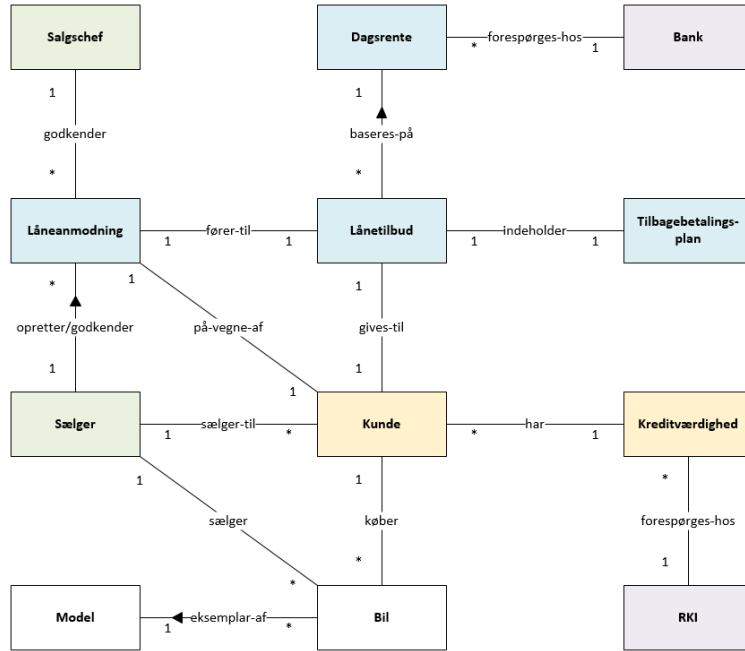


Use Cases bør desuden skrives uden diverse UI specifikke oplysninger. Endnu en grund til, at der står "anmoder om at oprette en ny låneanmodning" i stedet for "Trykker på opret ny låneanmodning", dette giver en langt bredere mulighed for nytænkning. Vi bør desuden anse systemet for en Blackbox, og i vores beskrivelser være så uspecifikke som muligt.

Skrivning af Use Cases bør færdiggøres i Elaboration, og der bør kun tilstøde meget lidt arbejde m. Use Cases i Construction fasen.

Domænemodel (Karsten)

Domænemodellen er en visualisering af domænet og viser relationer mellem domænets koncepter (aktører og artefakter) i et sporbart forhold til use case-beskrivelserne. Relationerne vises som beskrivende associationer med multipliciteter. Den udarbejdes på baggrund af en analyse af problemdomænet og visualiseres med et UML-klassediagram. Figur 1 viser vores domænemodel.



Figur 1: Domænemodel

Dataordbog (Karsten)

Med dataordbogen søger vi at beskrive samtlige væsentlige termer i problemdomænet. En fyldestgørende dataordbog indeholder gode definitioner og eksempler på domæneterminologi, og kan således styrke dels den interne kommunikation blandt os som udviklere—hvis vi læser den—andels kommunikationen eksternt med kunden og andre interesserter.

Gode definitioner dækker så vidt muligt en intentionel så vel som en ekstensionel definition, og henviser i øvrigt til relatedede termer. Er der særlige valideringsregler vedrørende en term, indgår de også her.

Den intentionelle definition er en generel beskrivelse af termen som koncept. Hvor det er muligt, suppleres den med en **ekstensionel definition** bestående af konkrete eksempler på det som konceptet dækker over.

Det typiske format på en definition af en term er:

- Term
 - Intentionel definition
 - Evt. valideringsregel
 - Ekstensionel definition

- Henvisning til relaterede termer

Tabel 1 viser starten på vores dataordbog. Resten kan ses i bilag A. Her har vi suppleret de danske termer med de tilsvarende engelske da vi har holdt vores kodebase på engelsk.

Term	Engelsk betegnelse
Afdrag Den del af den faste, månedlige ydelse som går til afbetaling af restgælden efter betaling af renter. Se: Ydelse, Restgæld	Repayment / <i>tilbagebetalingsplan:</i> Principal paid
Ansat En ansat i virksomheden med en given rolle. Se: Salgschef, Sælger	Employee
Anseelse En kundes anseelse. Angiver hvorvidt der tidligere har været problemer med kunden. Mulige værdier: God, dårlig	Standing
Basispris Den vejledende pris som en bil sælges for. Se: Salgspris	Base price
Bil En given bil som er til salg.	Car
Bruger En ansat som er oprettet som bruger af systemet med brugernavn og adgangskode.	User
CPR Personnummer i det danske CPR-register. Personfølsom data der behandles med særlig forsigtighed. Format: NNNNNNNNNN Fx: 140659XXXX	CPR

Tabel 1: Fragment af dataordbog

Systemsekvensdiagram (Henrik)

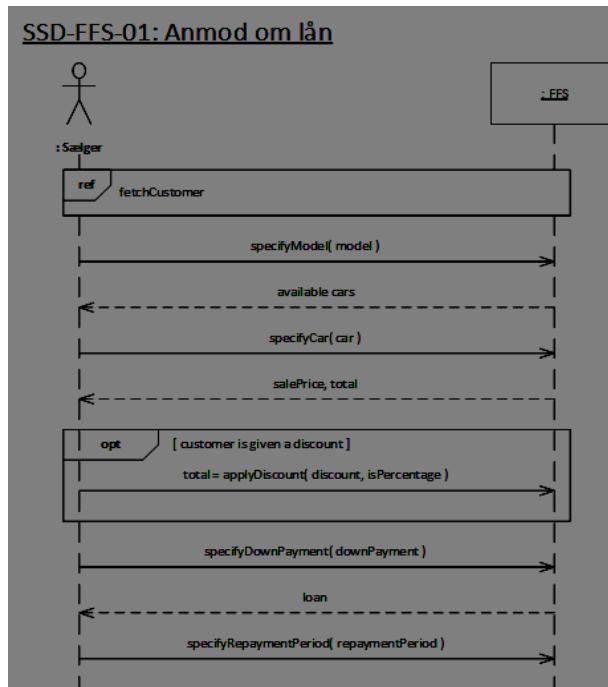
Systemsekvensdiagram anvendes til at vise kommunikationen i mellem en actor og systemet vha. systemoperationer og svar fra systemet.

Det skal læses oppefra og ned følgende livslinjen. Der kan benyttes tre forskellige frames til at vise **valgfri**(opt), **alternativ**(alt) eller **gentagne**(loop) metodekald. Det er opfyldelsen af guard-udtrykket i hver frame der afgør, hvad der skal ske. Som man kan se i nedenstående diagram, kører vi f.eks. kun opt-frame(applyDiscount), hvis vi giver kunden rabat(guard udtrykket).

For mere avanceret kommunikation, kan de forskellige frames nestes inde i hinanden. Den inderste frame afsluttes altid først.

Vi har, i vores opgave, lavet Systemsekvensdiagram til Use Case FFS-01 og FFS-02.

Diagrammet til FFS-01 ser ud som følgende:



Operationskontrakter (Henrik)

Operationskontrakter kan bruges som et tillæg til systemsekvensdiagrammet. Det udspecifierer Use Casen yderligere og beskriver systemoperationer mere detaljeret.

Operationskontrakter benytter en skabelon der følges, med fokus på hvad der sker, ikke hvorfor eller hvordan.

Skabelonen indeholder følgende:

Id	Gør den let at identificere, selv hvis systemoperationen ændrer navn. F.eks OC-02 som i nedenstående.
Navn	Navnet på den systemoperation, den beskriver. I nedenstående skal vi angive fornavn.
Operation	Navnet på den systemoperation operationskontrakten omhandler, komplet med parametre. Disse bliver ofte identificeret, når man laver systemsekvensdiagrammet, som de input actor foretager til systemet(blackbox). I nedenstående er firstName vor parameter.
Krydsreferencer	Navnene på de use cases denne operationskontrakt har med at gøre. OC-02 hører til FFS-01.
Preconditions	Forudsætninger for tilstanden i systemet, før udførelsen af den beskrevne operationskontrakt. F.eks. skal vi i nedenstående OC-02 have oprettet en instans af kunde inden vi kan påbegynde låneproceduren.

Postconditions	Beskriver de ændringer i tilstanden af domænet, der er tilstede, når systemoperationen er gennemført succesfuldt.
----------------	---

I vores opgave har vi f.eks. følgende operationskontrakter:

FFS-OC-02: specifyFirstName

Systemoperation

specifyFirstName(firstName)

Krydsreferencer

FFS-01: Anmod om lån

Forudsætninger

En instans c af Customer eksisterer.

Slutbetingelser

Hvis værdien af firstName er gyldig, blev c.firstName sat til firstName; ellers blev en fejlbesked præsenteret.

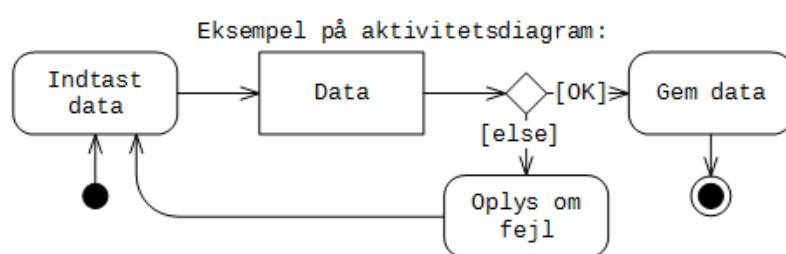
Aktivitetsdiagram (Henrik)

Dette diagram anvendes til at vise rækkefælgen af de forskellige processer(aktiviteter).

Det viser hele flowet i gennem en Use Case fra start til slut, med de forskellige decision og merge veje, der kan opstå i gennemløbet.

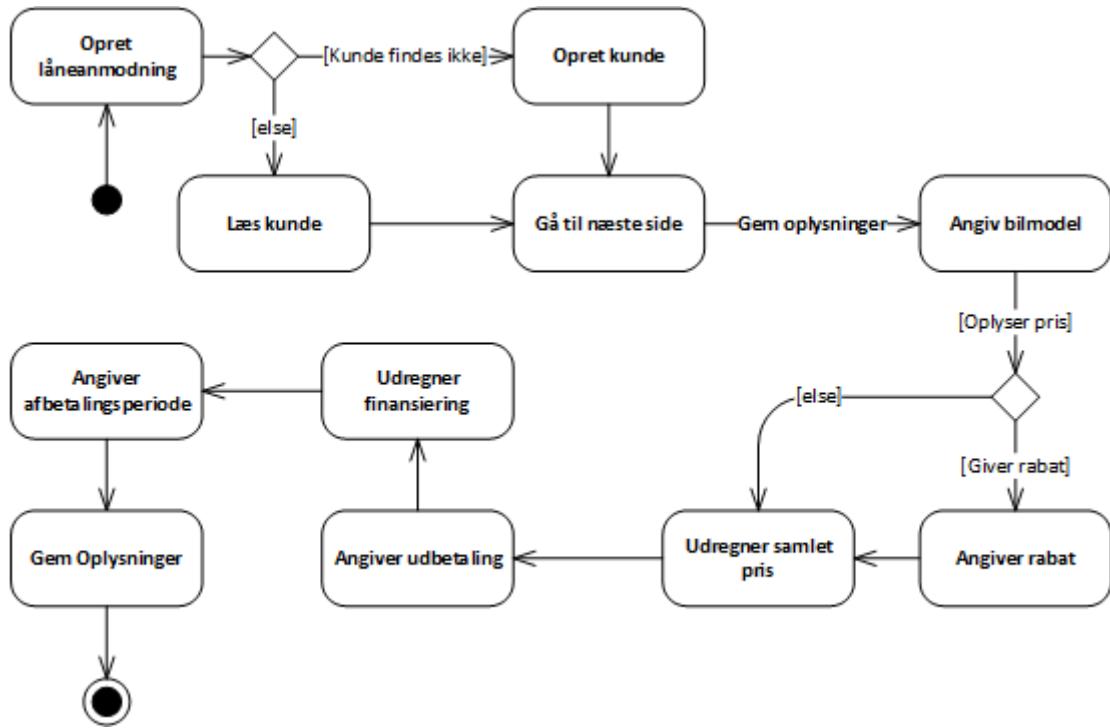
De kan vha. "fork" og "join" notation bruges til detaljeret at vise, om der er parallelle flows i løbet af processen. Endvidere kan "rake" benyttes til at angive en kompleks action, der kan udspecificeres i en delaktivitet.

Det kan også indeholde forskellige signaler så som time, send og accept.



I opgaven har vi lavet aktivitetsdiagrammer til FFS-01.

FFS-01 ser ud som følgende:

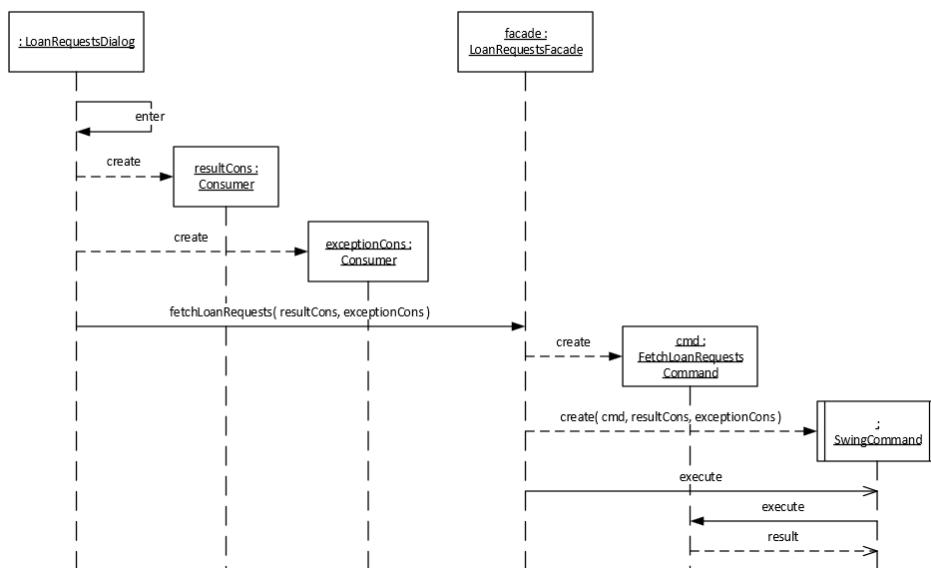


Sekvensdiagram (Enok)

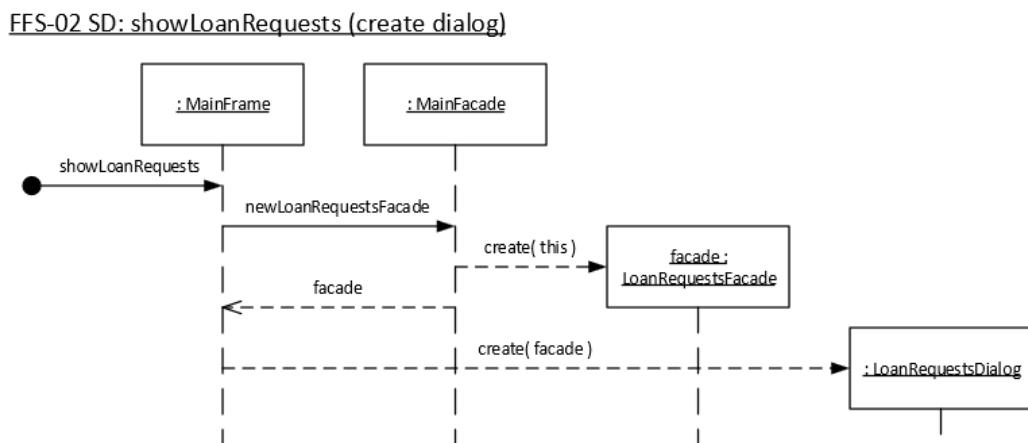
Sekvensdiagrammet er et værktøj til software design. Her kigger man især på kommunikationen mellem softwareobjekter, samt på programmets adfærd under udførelse. Desuden er et sekvensdiagram med til at identificerer hvilke objekter der har ansvaret for konkrete opgaver.

Sekvensdiagrammet bør være opbygget på en sådan måde at den læses fra venstre til højre, og oppe fra og ned.

FFS-02 SD: showLoanRequests (execute command)



I her er et eksempel på et af vores Sekvensdiagrammer, som illustrerer hvad der sker når vi beder systemet om at vise os alle låneanmodninger (Use Case FFS-02):



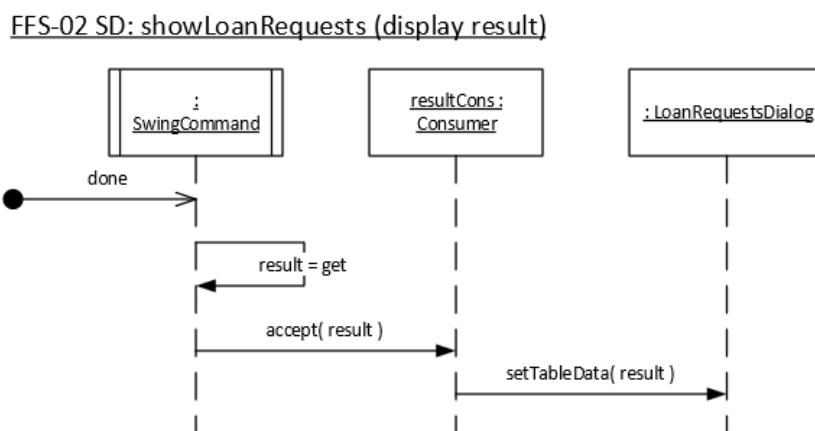
Ved at følge metode kaldene ned efter, kan vi følge den kommunikation der sker mellem programnets objekter og se, hvem der laver hvilke objekter hvornår og i hvilken rækkefølge. Dette gør at når vi endelig sidder med programmeringen så vil vi let kunne se hvilke metoder objekternes klasse skal have. Samtidig kan vi også kontrollere at vi holder vores tredeling, hvis vi samtidig er lidt opmærksom på, hvor hvilke klasser er placeret i vores software arkitektur. Hvis vi fx ser at DataAccess objekter kalder metoder på logik, så bør vi måske gøre et eller andet i vores design.

Men tilbage til sekvensdiagrammet, for at lette overblikket er det vi ser på det foregående diagram, kun oprettelsen af en facade og der efter et nyt vindue, resten af diagrammet har vi valgt at placere i et andet diagram:

Dette diagram viser kommunikationen under udførelsen af selve hentning af data(hvilket vores FetchLoanRequest Command står for).

Igen diagrammet her er en del af en større sammenhæng, men for at fremme overblikket er det taget ud, for at diagrammet ikke skal blive FOR forvirrende med alt for mange objekter og metode kald osv.

Og for at komme helt i mål har vi det sidste diagram i Use Casen:



Diagrammet her visser kommunikationen fra at swing command, som er vores tråd, får dataen og sender den retur til dialogframen.

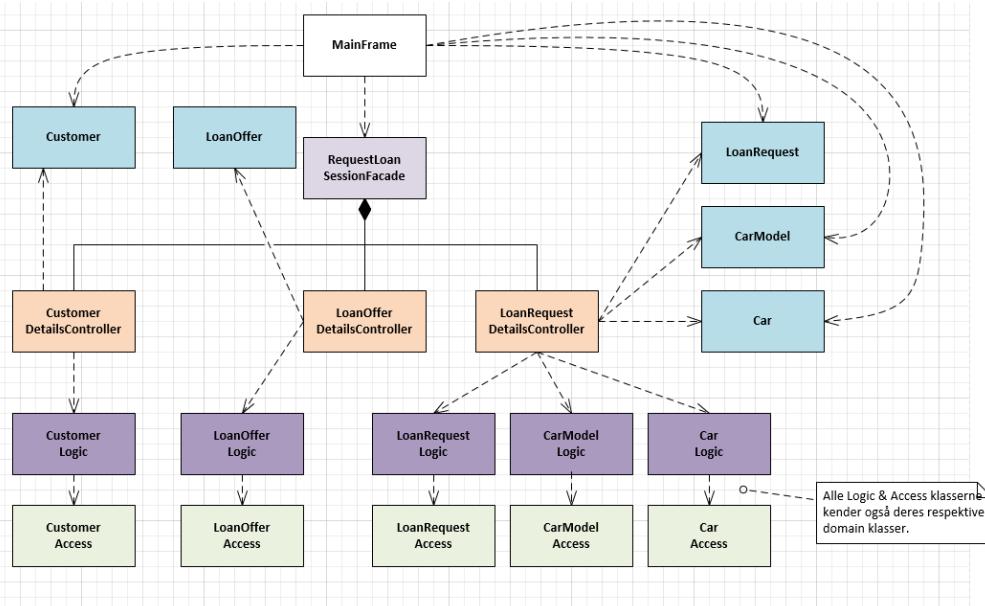
Som før fortalt så giver sekvensdiagrammer fordele når vi endelig sider og skal kode vores program, man bør dog altid huske på at sekvensdiagrammet, lige som en del af de andre artefakter, er dynamiske, og vi kan ændre ting igennem hele projektet ind til vort software er endelig færdig. Så det er ikke sådan at designet er sat i sten så snart vi har lavet det første sekvensdiagram.

Klassediagram (Enok)

Hvor SD diagrammet beskrev de dynamiske adfærd under kørsel af vort program, ud fra objekter, beskriver klassediagrammet den statiske struktur der findes i programmet ved at visualisere relationerne mellem softwareklasser.

Ofte vil et klassediagram tage et formål og beskrive det, for at fremme overblik ved kun at beskrive de detaljer, der er relevant for netop det formål.

I vores diagram har vi dog forsøgt at få plads til de fleste klasser ved at bruge farve notation som gerne skulle fremme overblikket tilstrækkelig til at man vil kunne bevare overblikket.



Farve kode:

Hvid(kun MainFrame): Repræsenterer User Interface klasser

Lys lilla: SessionFacade klasse

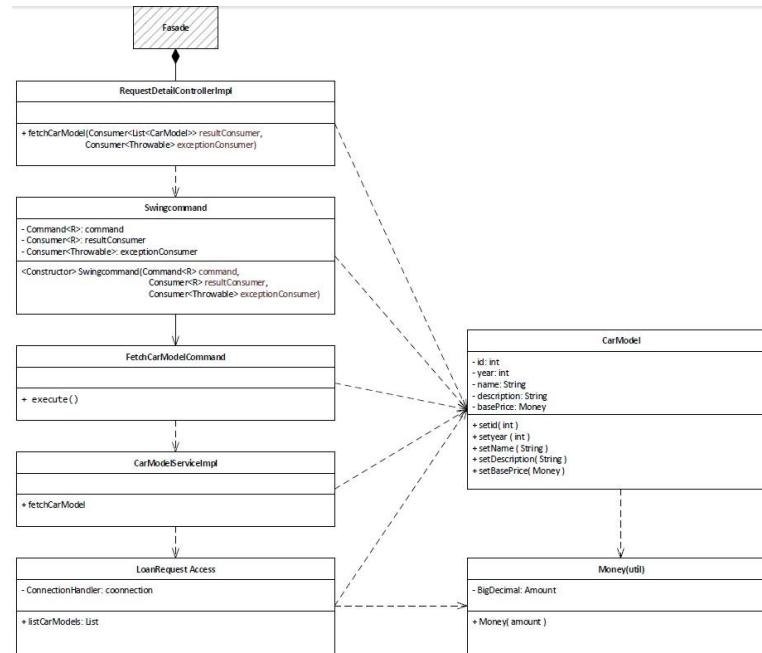
Lyserød: Controller klasser

Lilla: Logic klasser

Sandfarvet: DataAccess

Vi har brugt dette diagram til at forklare forholdet imellem vores UI og vores controllere. Hvor vi har en sessionFacade der fordeler alle metode kald fra UI til de rigtige klasser. Derudover har vi brugt klassediagrammet til at sikre tredeling i software designet. Således at kommunikationen kun går en vej, nemlig fra toppen til bunden.

Diagrammet oven for er til for at vise relationerne mellem vores klasser.



Klassediagrammet ovenfor viser kommunikations flowet under udførelse af fetchCarModels som bliver hentet for at vi kan vælge hvilken model vi vil sælge.

Samspil mellem Sekvensdiagram & Klassediagram

I sekvensdiagrammet arbejder vi med objekter, disse objekter er enten afledt af klasser eller Javas indbyggede klasser og metoder. Dvs. at når vi laver klassediagrammet kan vi bruge vores sekvensdiagram. De fleste objekter, som ikke kommer fra en Java klasse, skal have deres egen klasse i klassediagrammet.

Brugergrænseflade (Louise)

Brugergrænsefladen er en essentiel del af programmet, og kan dog være noget vanskelig at udarbejde i en god format, der tilfredsstiller alle parter. Derfor er løbende brugertests en vigtig del af udviklingen.

Fremstilling af modeller til dette, bør udvikles hurtigt, da man må påregne, at de forkastes om laves om af flere gange, efter samtalér med brugeren.

Det mest vellykkede er at have flere forskellige udkast, og ændre efter brugerens input, og herefter udfører testen igen.

Brugervenlighed er en meget vigtig faktor i vores system, og har en positiv påvirkning på flere af de ikke funktionelle krav, både pålidelighed og ydeevne. Til gengæld modarbejder det sikkerhed, da vi ikke ønsker den øgede tilgængelighed. Derfor er brugervenligheden vejet op modsikkerheden, og udgør en konstant balance. Fx kan man ved at fjerne log ind øge brugervenlighed og ydeevne, men der bør overvejes, hvad der ligger til grund for at have en log ind.

Brugervenlighed øger succesgarantien i vores program, da manglende brugervenlighed vil øge brugerens modstand til programmet.

I vores tilfælde har vi valgt at lave mockups manuelt, for at speede processen op, sikre flest mulige inputs og give maksimal mulighed overblik og hurtigt at kunne ændre, efter utallige ændringer blev den endelige mockup rentegnet i Visio.

Logisk arkitektur (Karsten)

Softwarearkitekturen udgør sættet af vigtige beslutninger om systemets organisering, valget af de strukturelle elementer og deres interfaces, og deres adfærd i form af kommunikationen mellem dem. [Larman, kap. 13]

Vi har valgt en lagdelt arkitektur med udgangspunkt i trelagsmodellen der overordnet består af tre lag, hhv. brugerflade, logik, og data access. Vores arkitektur er *relaxed* da et lag kan kalde ned til ethvert lag under sig—en udbredt arkitektur i informationssystemer.

Lagdeling af arkitekturen er en anvendelse af Layers Pattern og har en række fordele [Larman, kap. 13]:

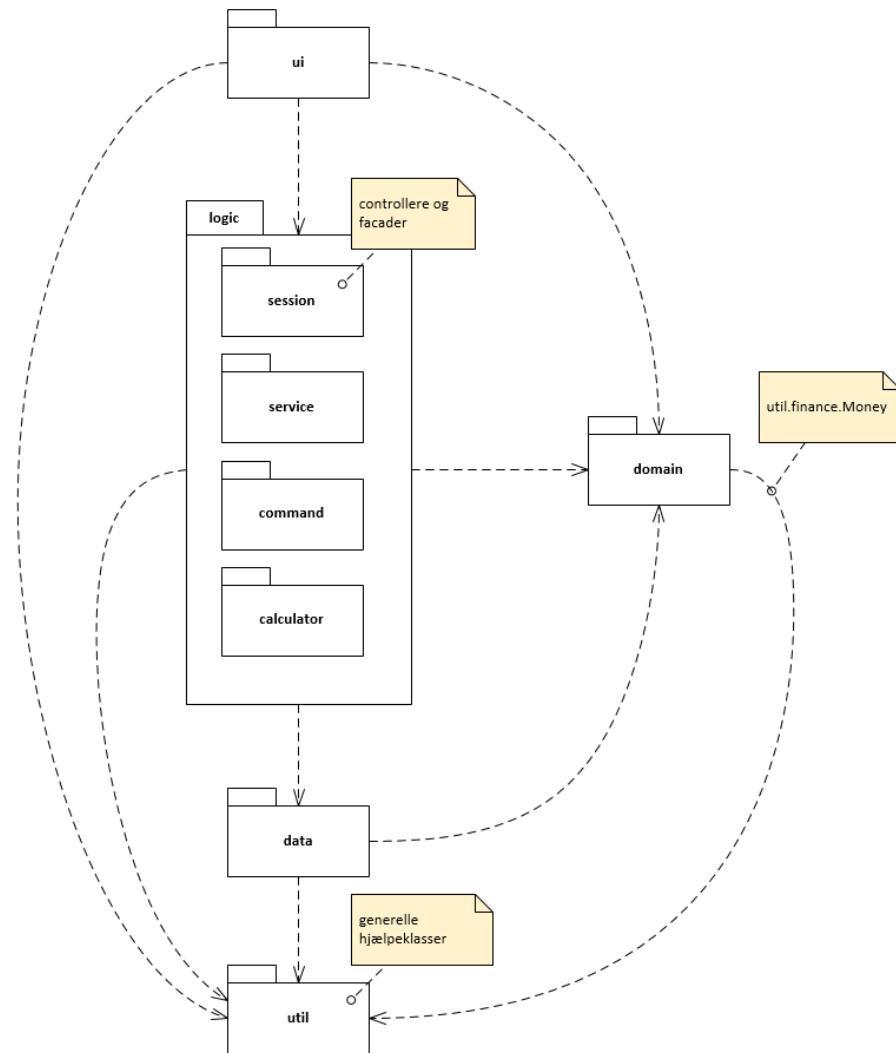
- Separation of concerns
- Adskillelse af high level fra low level services
- Adskillelse af applikationsspecifikke fra generelle services
- Reduceret kobling
- Øget potentialet for genbrug
- Øget gennemskuelighed

Det enkelte lags niveau afspejler dets stabilitet i forhold til forandringer. De nederste lag bør være de mest stabile da forandringer her vil propagere op igennem hele arkitekturen. Modsat er brugerfladen øverst fri fra koblinger fra de øvrige lag, og kan med større lethed udsættes for forandringer eller helt skiftes ud efter behov.

Pakkediagrammet i figur 2 viser vores logiske arkitektur. Entitetklasser har vi valgt at trække ud af logiklaget til en separat pakke: domain. Denne pakke er speciel da vi i OO typisk vælger at lade brugerflade og data access kende entiteterne frem for at gøre disse lag dataagnostiske. Dermed skaber vi en kobling som er i strid med kravet om udelukkende nedadgående kommunikation i arkitekturen. Det er en undtagelse vi kan leve med idet vi holder entiteterne som simple dataobjekter.

I logiklaget findes applikationens forretningsregler og service-klasser til entiteterne i domænet. Brugerfladen (ui) indeholder ”dumme” vinduer og paneler som blot er tynde klienter med ansvar for at præsentere output og deletere input videre ned til logiklaget. Dermed ingen logik i brugerfladen, jf. Model-View-separation. Endelig har datalaget ansvar for at oversætte forespørgsler fra serviceklasser i logik til CRUD-operationer i databasen.

Allernederst har vi en util-pakke med diverse generelle hjælpeklasser, fx en Money-klasse til repræsentation af pengebeløb.



Figur 2: Logisk arkitektur

Teknologi

Trådprogrammering (Karsten)

En proces afvikles af operativsystemet (eller den virtuelle maskine; Java Virtual Machine i vores tilfælde) i sin egen *execution context* der indeholder processens tilstand under afvikling. Den enkelte proces kan udnytte moderne processorers muligheder for parallelisme (concurrency) ved at uddeletere arbejdsopgaver til tråde—"miniprocesser"—som alle deler adgang til processens execution context.

Udfordringer i concurrency

Fordi trådene deler samme tilstand og afvikles parallelt, medfører brugen af concurrency nogle problemer: Hvis flere tråde arbejder på samme del af tilstanden, kan vi opleve *thread interference*; at deres operationer interfererer og overskriver hinanden med uventede resultater til følge.

Et andet problem er *memory inconsistency* hvor flere tråde ser forskellig data som burde være ens i den samme tilstand. Til håndtering af dette arbejder vi med *happens-before*-garantier. Når en happens-before-garanti er givet, er vi forsikret om at ændringer i tilstanden udført af én operation er synlige for en efterfølgende operation. Det er fx tilfældet når vi kalder Thread.start—effekten af alle operationer inden kaldet er synlige for alle operationer i den nye tråd der startes. [JT]

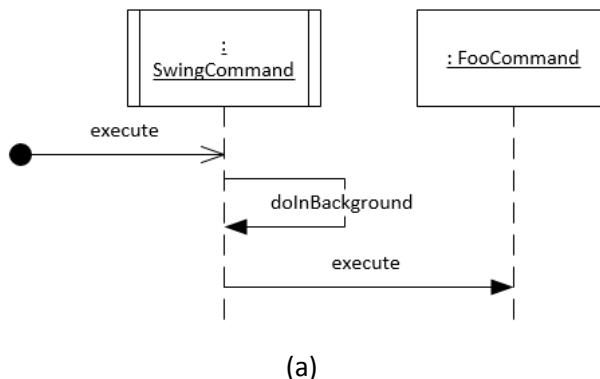
Synkronisering

For at undgå thread interference og give happens-before-garantier kan vi indføre synkronisering på metoder og statements. Synkronisering medfører at en intrinsic lock—intern lås—sikrer at kun én tråd kan afvikle en given metode eller statement ad gangen.

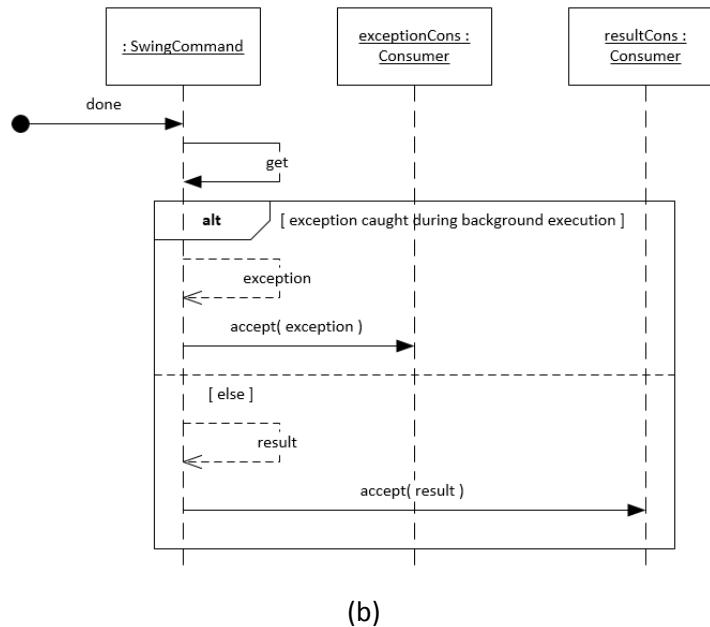
Synkronisering er en løsning på de første problemer; desværre indfører brugen af locks nye problemer vi må være opmærksomme på. En af de mest kendte er deadlock: en tilstand vi kan risikere hvor afviklingen af to eller flere tråde går i stå fordi de alle venter på hinanden. Omvendt beskriver den mere sjældne livelock en situation hvor trådene ikke går i stå, men konstant giver plads til hinanden uden at nogen af dem får udført deres opgave. Et andet fænomen er starvation hvor vi risikerer at en tråd må vente meget længe på adgang til en resurse fordi den optages af en eller flere andre ”grådige” tråde. [JT]

SwingWorker

Da vi i dette projekt arbejder med en Swing GUI, har vi valgt at gøre brug af javax.swing.SwingWorker til håndtering af vores tråde. [JT2] Det særlige ved SwingWorker er dens doInBackground- og done-metoder. Når vi kalder execute på en SwingWorker, oprettes automatisk en ny tråd (SwingWorker gør brug af Executors internt). I den nye tråd kaldes doInBackground som udfører vores tidskrævende operation. Når doInBackground har returneret, kaldes done automatisk i EDT (Event Dispatch Thread); tråden hvor alle Swing events behandles. Vi kan således komme helt uden om manuel synkronisering, blot vi holder vores doInBackground-operationer indkapslet som isolerede operationer uden brug af delt tilstand. SD-diagrammet i figur 3 viser hvordan vi i EDT starter udførelse af en kommando (a) og håndterer resultatet fra den efterfølgende (b).



(a)



(b)

Figur 3: Udførelse af FooCommand med SwingCommand og Consumer

Er det en event i brugerfladen der starter udførelse af FooCommand, instantierer brugerfladen to Consumer-objekter (functional interface); den ene til behandling af det forventede resultat, den anden til behandling af evt. exception. Brugerfladen sender disse consumers med kommandokaldet til controller (via facade), og controller opretter en instans af SwingCommand (vores nedarvning af SwingWorker) der modtager consumer-instanserne. Således kan SwingCommand—ved kald til `done`—undersøge om et resultat kom tilbage fra `FooCommand.execute`, eller der blev fanget en exception under udførelsen, og kalde `accept` på den tilsvarende consumer.

GRASP (Louise)

GRASP, også kaldet "**General Responsibility Assignment Software Patterns or Principles**" kan betragtes som et hjælpemiddel til at lærer "Responsibility-driven Design".

Når vi taler om Responsibility-driven Design (RDD), tænker vi på hvilke objekter, der har ansvar for hvad.

Der er 2 typer ansvar:

- Doing
- Knowing

GRASP er bestående af 9 grundlæggende mønstre. Der diskutes endvidere, om der er tale om mønstre, eller om de reelt er gået hen og blevet principper, men dette er ikke vigtigt.

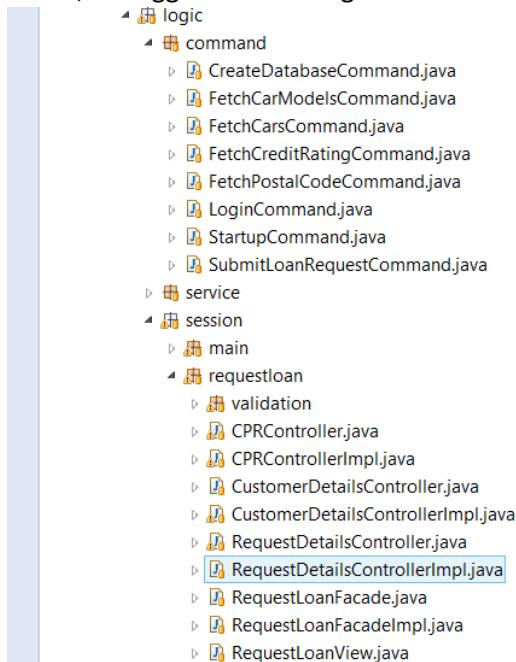
Når man taler om mønstre, er der tale om noget velkendt. Det er altså tale om noget der er gentaget igennem lang tid. Jo mere velkendt det er, jo bedre.

Et godt mønster har altså et sigende navn, en velkendt problemstilling, med en tilhørende løsning på dette problem.

De 9 GRASP mønstre er svære at sætte ind i individuelle bokse, da de langt hen ad vejen overlapper hinanden. Fx går "High Cohesion" hånd i hånd med "Low Coupling". Desuden kan "Low Cohesion" fx stamme fra en "Bloated Controller".

Når vi kigger i vores projekt, vil man meget hurtigt bemærke, at der er en overvældende mængde klasser. Dette er netop for at sikre, at hver klasse kun er ansvarlig for et konkret område. På denne måde, har vi sikret, at der er High Cohesion, og Low Coupling i programmet. Dette gør det både lettere at overskue og vedligeholde, da det gør den enkelte klasse meget lille.

Hvis vi forestiller os, at de Controllere, der ligger i session.logic



Hvis vi forestiller os, at vi i stedet for disse Controllere, havde lavet én stor Controller, der skulle tage sig af delegering af alt i FFS-01, så havde vi haft en klasse, der havde indeholdt mange hundrede SLOC.

Dette er hvad vi ville kalde en "Bloated Controller", klassen ville på denne måde have alt for mange ansvarsområder, og komme til at lide under både Low Cohesion og High Coupling.

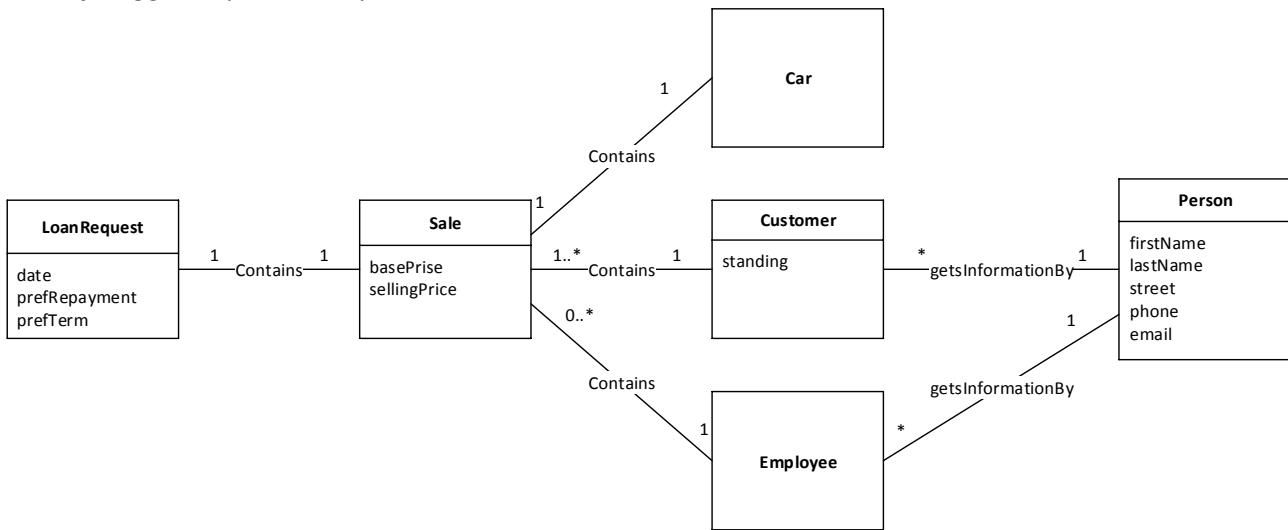
En anden måde at gøre vores Controllere til en Bloated Controller, ville være at lade Controllerne udfører opgaverne selv, i stedet for at delegerere. På denne måde ville klassen igen have et alt for stort ansvarsområde, og herved igen lide under Low Cohesion.

Information Experts

Sammenhørighed er ikke noget, der kan vurderes alene i et program, man vil som tidligere set logisk kigge på koblingen, men eksperter er endnu en del, som vi overvejer sammen med. Disse kalder vi "Information Experts"

Pointen m. Information Experts er, logisk at kigge på, hvilken klasse, der bør være ansvarlig for hvilken information.

Vi kan fx kigge lidt på LoanRequest.



Rent principielt, kunne vi godt komme al information, der kommer til at berører **LoanRequest**-klassen ind i **LoanRequest**, men dette vil skabe et par problemer, som vi allerede har snakket om:

- Low Cohesion
- High Coupling

Dette ville fører til en meget stor og uoverskuelig klasse, der desuden ville være svær at vedligeholde, og blive påvirket ofte, hvis der skal laves om i programmet.

Dette leder os til Information Experts.

I vores eksempel, kan vi se, at **LoanRequest** har kun ansvar for at kende til en dato, en ønske afdragsperiode og en ønsket tilbagebetalings sum. De resterende informationer er mere logiske at lægge i et salg. Heraf vores salgs-klasse.

I den er der igen kun få ting, der behøver være kendt i salg. Det samme gør sig gældende for de resterende 4 klasser.

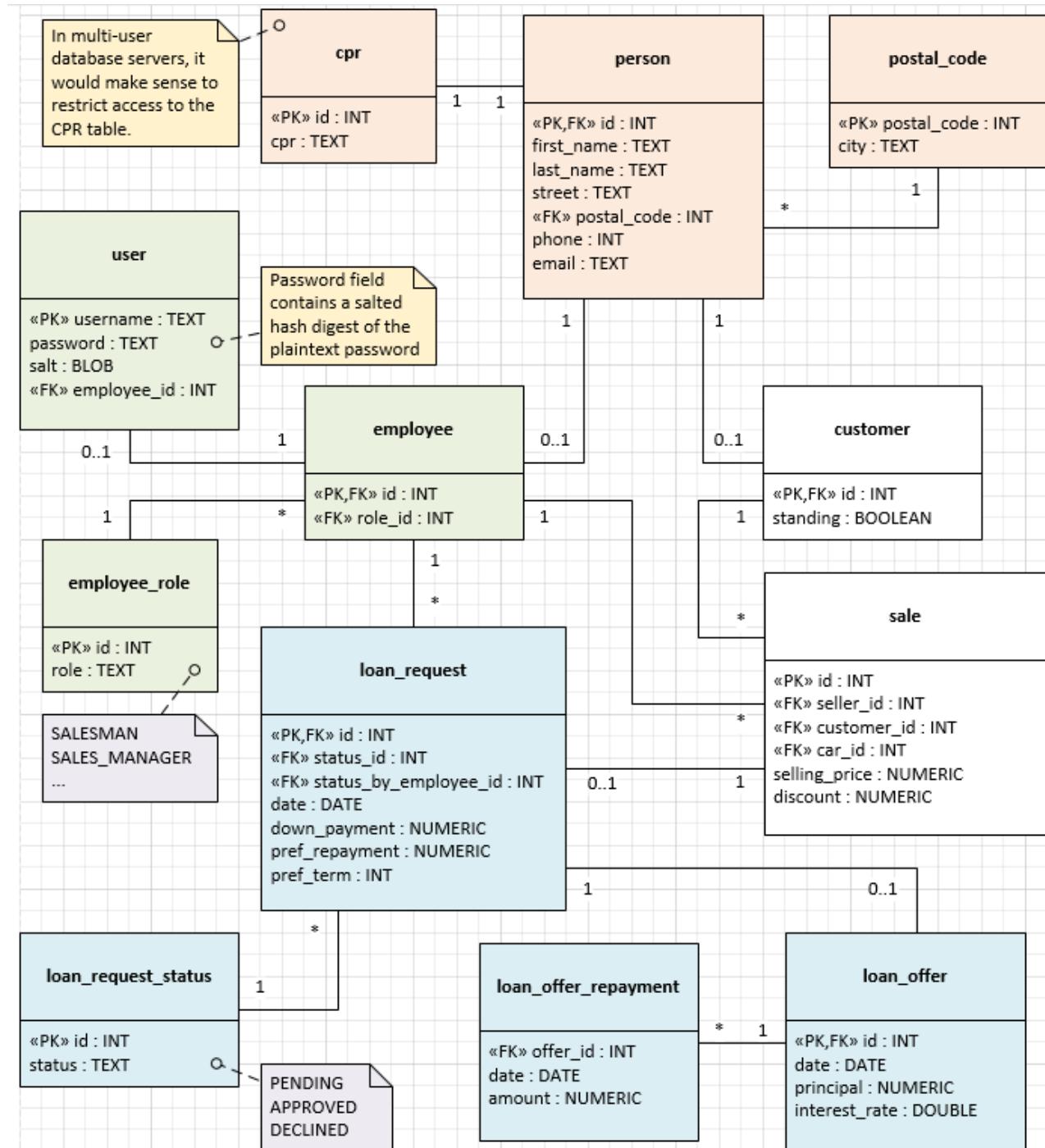
Der er endnu flere klasser, der indeholder små mængder information, der logisk set bør ligge ved dem.

Dette hjælper til, at det er let, ud fra klassens navn, at identificere, hvad der vil ligge i den pågældende klasse.

Databaser (Enok)

For at gemme data i et program bruger man databaser. Databasen er en struktureret samling af digital data gemt som bits og bytes.

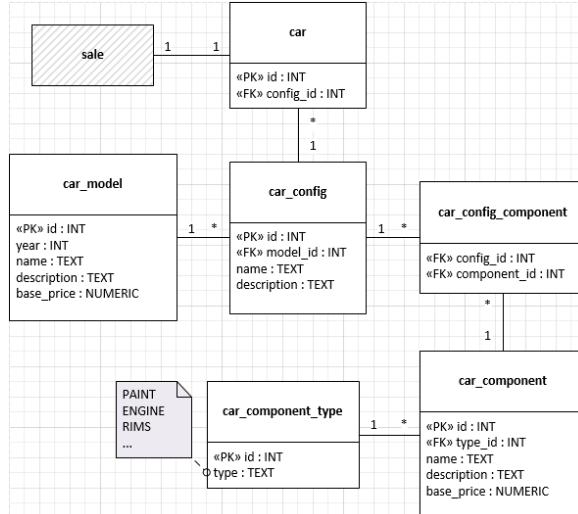
Databasemodel:



For at lette selve processen med at designe databaser, laver man en database model. Modellen består af de forskellige tabeller, samt de attributter som hvert tabel indeholder.

Derudover viser modellen også relationen mellem de forskellige tabeller med multipliciteter samt foreign keys osv.

Normalt vil man sige at vi har for mange tabeller i vort diagram da vi mennesker kun kan overskue 8 +/- 2, men ved hjælp af farve kodning har vi gjort det meget lettere at holde overblikket.



Alternativt vil man tage de enkelte områder og tage dem ud i diagrammer for dem selv. Som vi har gjort her med vores "Bil" diagram.

En ting at notere her er især tabellen car_config_component, som kun består af foreign keys. Dette betyder at tabellen er sat ind imellem 2 andre tabeller for at realisere en mange-til-mange relation.

Normal former:

For at gøre strukturen i databaserne og derved også søgbarheden bedre, har man nogle normalformer, som man bruger til at Normalisere databaser. Disse normalformer sikre at vi ikke bruger for meget data i vores databaser, ved fx gengivelser, men sørger også for at vi let kan finde rundt imellem linkede tabeller, og at vores program hurtigt og effektivt vil kunne søge efter data i vores database tabeller.

1. Normalform:

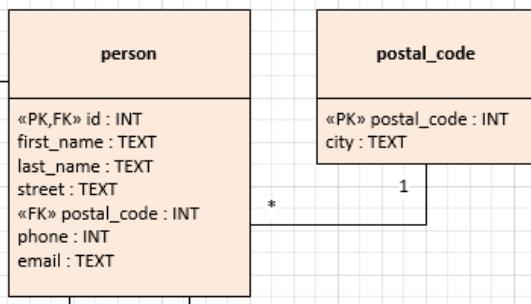
- Ingen felter med mere end 1 værdi.
Dvs. man må ikke gemme både postnummer og by i samme felt, men skal splittes ud i 2.
- Fordel data i særskilte tabeller med en slags information i hver.
- Ingen gentagelser. Så frem der er gentagelser bør den gentagne data tages ud og sættes i en selvstændig tabel, som så forbindes med en fremmednøgle/foreign key.

2. Normalform:

- Intet felt/attribut i en tabel må være afhæng af en del af primærnøglen.
Dette kan ske hvis man har primærnøgler der består af 2 felter/attributetter.

3. Normalform:

- Intet felt må afhænge af andre felter end primær nøglen.
For at overholde denne normal form, bliver vi fx, nød til at tage by ud af Person tabellen, og placere den i en tabel for sig selv. For by navnet afhænger nemlig ikke af primærnøglen i Person tabellen, men af postnummeret.



Desuden skal det bemærkes at for at 2. og 3. normalform kan være opfyldt skal de(n) foregående normalform(er) være opfyldt.

IT Sikkerhed (Enok)

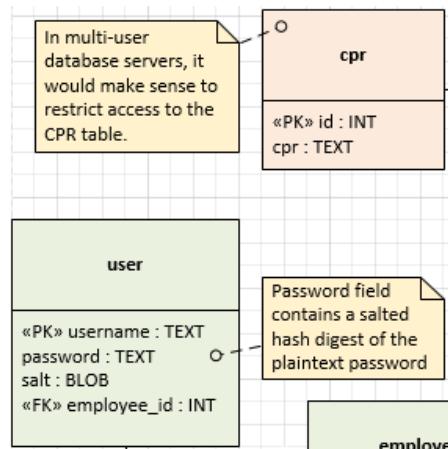
Når man taler om IT sikkerhed er der en bestemt ting man altid bør tage højde for når man laver et stykke software: Hvor sikker er den information du gemmer?

I forhold til vort projekt er der en central information som er særdeles følsom for både kunder og ansatte: CPR-nummeret. Dette kan give en evt. informations tyv mulighed for at foretage identitets tyveri og derved skade vore kunder og ansatte. Det er derfor meget vigtig at vi tager højde for dette denne risiko når vi laver vores system.

Måden vi har valgt at sikre CPR data på er ved at placere den i en tabel for sig selv. Dette gør at man kan begrænse adgangen til netop denne tabel, så det kun er særdeles betroede medarbejdere der kan få adgang til den. Derved begrænser vi adgangen til den følsomme data.

Alternativt bør man, som minimum, sørge for at CPR-nummeret ikke bruges som primær nøgle i en person tabel. Således at det ikke er CPR-nummeret der bruges til at hægte personer i Person tabellen sammen med andre elementer i andre tabeller.

Udover CPR-nummeret kan der også være andre data man bør behandle på en bestemt måde. Fx passwords. Et password kan give direkte adgang til vort system så frem det bliver stjålet fra databasen, så kan være en god idé at beskytte det.



Vores eksempel på databeskyttelse i vores database model:

Til slut skal det bemærkes at vi i Danmark har et Datatilsyn som opstiller regler for hvordan personfølsom data skal behandles og opbevares, og disse regler skal vort program og software selvfølgelig leve op til.

Design Patterns (Henrik)

Introduktion til Design Patterns

Design patterns er en beskrivelse af den bedste løsning, til et ofte gentaget problem, i software programmering. Et design pattern er ikke et færdigt design, der direkte kan omskrives til kode. Det er mere en beskrivelse eller skabelon til løsningen, som kan benyttes i mange forskellige situationer.

Et design pattern viser den bedste tilgang til løsningen af et problem, som er gennemtestet mange gange af erfarne udviklere/ programmører. Det skal forstås som den bedste løsning på de generelle problemer, man kan støde på som udvikler/programmør.

Hele begrebet design patterns opstod i 1994, da de fire forfattere

Erich Gamma, Richard Helm, Ralph Johnson og John Vlissides udgav en bog med titlen "Design Patterns – Elements of Reusable Object-Oriented Software". Disse fire forfattere kendes tilsammen som Gang of Four(GoF).

Brug af Design Patterns

Design patterns kan gøre udviklingsprocessen hurtigere ved at levere gennemtestede løsninger. Effektiv software design kræver, at man overvejer problemer, der måske ikke opstår før senere i implementationen. Ved at "genbruge" kendte design patterns, kan man undgå småtning, der kan give store problemer senere. Det øger desuden læsbarheden for andre programmører, der kender det anvendte Design Pattern i forvejen.

Da Design Patterns ofte er udviklet over lang tid, er den struktur de tilbyder, som regel den bedste og mest effektive måde at løse bestemte problemer på.

Typen af Design Patterns

Ifølge ovennævnte bog, er der 23 design patterns der kan inddeltes i tre hovedkategorier. Der er siden kommet mange flere til. I det følgende vil jeg inddеле de 23 design patterns i de tre hovedkategorier, og give en kort beskrivelse af hver.

Creational design patterns handler om instantiering af objekter.

Det indeholder følgende Design Patterns:

- Abstract factory bruges til at give en klient et sæt af relaterede eller afhængige objekter.
- Builder skaber komplekse objekter.
- Factory method kan erstatte konstruktorer og derved gøre objekt konstruktionen abstrakt.
- Prototype instantierer nye objekter ved at kopiere eksisterende objekter.
- Singleton sikrer at der kun eksisterer en instans af en bestemt klasse.

Structural design patterns handler om klasse og objekt composition.

Det indeholder følgende Design Patterns:

- Adapter skaber forbindelse mellem to inkompatible typer.
- Bridge adskiller abstrakte elementer i en klasse fra implementationen. Muliggør udskiftning af implementationen uden at ændre i det abstrakte.
- Composite skaber rekursive træ strukturer af sammenhængende objekter.
- Decorator kan øge eller ændre funktionaliteten i et objekt under kørsel.
- Facade giver et simpelt interface til et mere komplekst objektsystem.
- Flyweight reducerer ressource forbrug for komplekse modeller.
- Proxy giver et erstatningsobjekt med reference til et underliggende objekt.

Behavioral design patterns handler om kommunikation mellem klasser og objekter.

Det indeholder følgende Design Patterns:

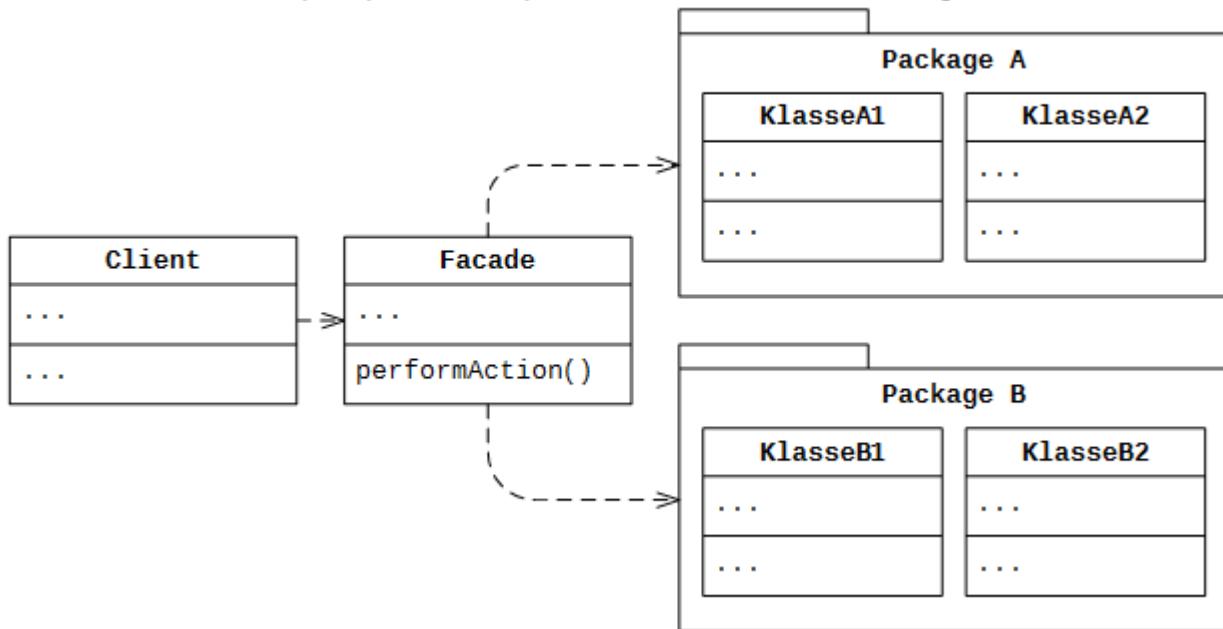
- Chain of responsibility giver et container objekt, der modtager alle requests og afgør hvilket objekt, der skal varetage den.
- Command lagrer requests i et command objekt, inklusiv kald med alle påkrævede parametre.
- Interpreter håndterer grammatik.
- Iterator giver mulighed for at gennemløbe elementerne i en collection.
- Mediator reducerer kobling mellem klasser der kommunikerer sammen.
- Memento gemmer den aktuelle tilstand i et objekt og kan genskabe det senere.
- Observer definerer et link mellem et objekt(subject) og dens afhængige objekter(observers). Dette giver observers besked, så snart tilstanden i subject objektet ændrer sig.
- State ændrer adfærdten af et objekt, når dets tilstand ændres.
- Strategy indeholder lignende algoritmer liggende i hver deres subklasse, den aktuelle der skal køres afgøres ved run-time ud fra krav.
- Template method definerer skridt i en algoritme og tillader hvert skridt at blive ændret uafhængigt.
- Visitor adskiller komplekst datastruktur fra den funktionalitet, der ønskes udført på det.

I forbindelse med vores opgave har vi benyttet tre Design Patterns, som jeg vil beskrive nærmere i det kommende.

Facade Pattern giver et simpelt interface til et mere komplekst objektsystem.

Det er ideelt, når man arbejder med et stort antal af afhængige klasser, eller klasser der kræver brug af mange metoder. Facade Pattern bruger en enkelt klasse, med forenklede metoder der bruges af klienten. Det delegerer de forskellige kald ud til metoder i eksisterende klasser i systemet.

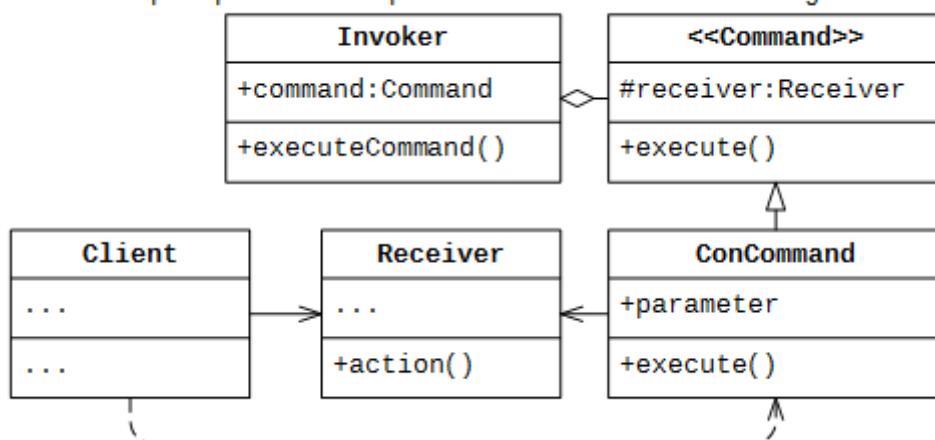
Eksempel på facade pattern vist med klassediagram:



Command pattern lagrer requests i et command objekt, inklusiv kald med alle påkrævede parametre.

Det indeholder ikke funktionaliteten der skal udføres, men al information der skal til for at udføre et kald. Funktionaliteten gemmes i et receiver objekt. Dette sørger for, at der er en løs kobling, mellem det objekt hvor hændelsen opstår, og det objekt der skal udføre funktionen. For at kontrollere hvornår kaldet udføres bruger man en "invoker klasse".

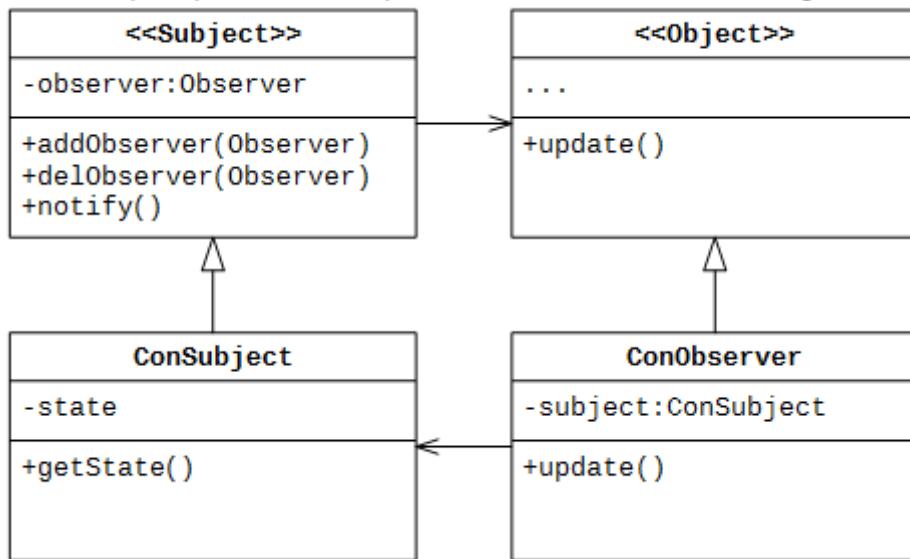
Eksempel på command pattern vist med klassediagram:



Observer pattern definerer et link mellem et objekt(subject) og dens afhængige objekter(observers). Dette giver observers besked, så snart tilstanden i subject objektet ændrer sig.

Dette pattern giver et abstrakt forhold mellem objekter. Observer tilmelder sig hos subject for at kunne modtage besked, når tilstanden i subject ændrer sig. Subject underretter de tilmeldte observers, når en ændring opstår. Observer kan altid framlede sig disse opdateringer.

Eksempel på observer pattern vist med klassediagram:



Kvalitetssikring

Test (Karsten)

Formålet med tests er at identificere fejl i koden (defects)—og de findes selvom vi kan føle os nok så overbeviste om at den er fejlfri.

"A defect is the algorithmic cause of a failure: some code logic that is incorrectly implemented" [Christensen, s. 16]

Slipper fejl i koden forbi os, risikerer vi at vores program fejler under brug (failure).

"A failure is a situation in which the behavior of the executing software deviates from what is expected"
 [Christensen, s. 16]

Det kan være en farlig situation for vores image som udviklere hvis programmet fejler ude hos kunden, og det skader indtrykket af vores produkt. Dertil kommer at fejlretning bliver betydeligt vanskeligere at foretage når først produktet er hos kunden efter release. Det kan fx være en udfordring at skulle reproducere en fejl alene baseret på brugerobservationer hos kunden. Vi ønsker derfor at opdage fejl så hurtigt som muligt i udviklingsprocessen. Jo hurtigere de opdages, des færre fejlretninger skal vi foretage, og vi undgår helst at ende med en stor kodelinje som viser sig afhængig af fejlagtig funktionalitet på et dybere niveau.

Hvad er en test?

Testen er et værdifuldt værktøj og kilde til høj moral idet den giver tillidsvækkende feedback til os som udviklere. Vi kan skrive ny kode eller refaktorisere eksisterende kode og straks—hvis testen er automatiseret—få bekræftet at vores kode fungerer efter hensigten.

"Testing is the process of executing software in order to find failures" [Christensen, s. 15]

Vores tests beviser ikke at vores program er fejlfrit (en fuldkommen og udtømmende test er urealistisk), men gode tests reducerer risikoen for failures i takt med at defects identificeres og rettes. Formår vi at lave omfattende tests på alle centrale dele af vores program, kan vi hvile i forsikringen om at vi leverer et produkt af høj kvalitet—i hvert fald hvad løsningens korrekthed angår.

Test cases

Når vi tester, arbejder vi med konkrete test cases. En test case udgør et specifikt, afgrænset scenarie hvor en testenhed kontrolleres i forhold til et forventet output som respons på et konkret sæt af input.

"A test case is a definition of input values and expected output values for a unit under test" [Christensen, s. 16]

En testenhed (unit) er i bred forstand en del af et system som vi i testsammenhæng betragter som en hel, atomisk enhed. Hvad en unit i en given test case dækker over, afhænger af testens omfang og kan spænde fra et lille metodefragment, over subsystemer og helt til det komplette system.

"A unit under test is some part of a system that we consider to be a whole" [Christensen, s. 17]

Der vil typisk være flere test cases rettet mod en given unit. Samlingen af de relaterede test cases udgør en test suite som søger at definere en dækkende test.

"A test suite is a set of test cases" [Christensen, s. 16]

Hvordan vi tester

Det er muligt—men sjældent udført i praksis—at teste koden manuelt. I den manuelle test er det en person som styrer gennemgangen af en test case. Input angives manuelt til programmet hvorefter output verificeres. Det er en tidskrævende form for test, men kan af og til være nødvendig, fx ved brugertest.

"Manual testing is a process in which suites of test cases are executed and verified manually by humans"
[Christensen, s. 17]

Mere anvendelig er den automatiserede test hvor der skrives egentlig testkode hvis ansvar det er at udføre test cases og verificere at faktisk output stemmer overens med det forventede.

"The test code is the source code that defines test cases for the production code" [Christensen, s. 19]

Automatiseret test tillader dermed en nem og hurtig udførelse af test suites, og derfor ønsker vi naturligvis at automatisere vores tests så vidt muligt.

"Automated testing is a process in which test suites are executed and verified automatically by computer programs" [Christensen, s. 18]

Vi får dermed to sæt af kode: produktionskode og testkode. De to sæt holdes skarpt adskilt da testkode ikke er en del af produktet. Den separate testkode er dog afhængig af produktionskoden i det omfang at de enkelte test cases er koblet til produktionskoden. Vigtigt er det at produktionskoden intet kender til testkoden.

"The production code is the code that defines the behavior implementing the software's requirements"
[Christensen, s. 18]

Automatiserede tests har desuden den sidegevinst at motivere til udførelse af regressionstest ofte, så vi hele tiden sikrer at nye tilføjelser til og ændringer i produktionskoden ikke får uventede, negative konsekvenser.

"Regression testing is the repeated execution of test suites to ensure they still pass and the system does not fail after a modification" [Christensen, s. 17]

Test framework

Når vi implementerer vores test cases i testkoden, er det naturligt at benytte et test framework. Frameworket består af en mængde af fælles funktionalitet til udførelse af test. Det øger produktiviteten at vi ikke selv skal udarbejde de fundamentale dele af testkoden, og ikke mindst bidrager brugen af et framework til at holde risikoen for defects i testkoden på et minimum. Andre nyttige funktioner er typisk muligheden for opstilling af statistikker over udførte tests.

Udbuddet af test frameworks er stort. Udbredt til Java er frameworket JUnit som vi benytter i dette projekt. Centralt i test frameworket er assert-mekanismen som sammenligner forventet med faktisk værdi og melder fejl hvis vores antagelse ikke holder. Vi kan supplere frameworkets standardbesked med vores egen for en mere detaljeret tilbagemelding.

Testparadigmer (Karsten)

For at sikre kvaliteten af vores program mest muligt kan vi teste efter særlige testparadigmer. Testparadigmerne beskriver hver især en bestemt tilgang til løsning af testopgaven på en måde så testen vurderes tilstrækkelig. Vi kan ikke teste udtymmende da det vil kræve et urealistisk stort antal test cases. Der er derfor behov for kriterier for tilstrækkelighed der lader os begrænse antallet af test cases og samtidig sikre at dækningen af dem er fyldestgørende.

Blackbox

Blackbox-test, også kendt som functional testing, beskriver en tilgang til testenheden som et lukket system. Her kender vi i principippet intet til hvordan systemet fungerer; kun hvad dets funktion er. Vi tester systemet på grundlag af forventninger og specifikationer, og kan således opstille en liste af inputs hvortil der skal svare et bestemt output. Her er kriteriet for tilstrækkelighed at hver enkelt delfunktion er testet mindst én gang. Kombinationer af input som behandles forskelligt tester hver deres delfunktion.

Blackbox-test er et alternativ til whitebox-test, eller structural testing, som omvendt tilgår testenheden som et åbent system. Her kan anvendelsen af kodemetrikker til automatiseret analyse på en række

forskellige parametre bl.a. denne grundlag for udarbejdelsen af dækkende tests samt forbedring af produktionskoden gennem refaktorisering.

Der findes forskellige testteknikker til opdeling af testforløbet i bestemte aktiviteter. I blackbox-test findes en vifte af teknikker som hver især dækker systemet på forskellig vis. Hver teknik afspejler en specifik måde at drive testen på.

Datadrevet test

I datadrevet test er målet at identificere de sammenhænge mellem data som tilsammen vil teste alle dele af systemet. Med udgangspunkt i projektets case har vi i kraft af forretningsreglen vedrørende renteberegning en beskrivelse af sådan sammenhænge. Der gælder følgende:

- Kreditværdighed skal være A, B eller C
- Kreditværdighed A tillægger +1 procentpoint
- Kreditværdighed B tillægger +2 procentpoint
- Kreditværdighed C tillægger +3 procentpoint
- Udbetalingsprocenten skal være på mindst 20 %
- En udbetalingsprocent under 50 % tillægger +1 procentpoint
- En løbetid på mere end 36 mdr. tillægger +1 procentpoint

Hertil føjer vi følgende begrænsninger for frasortering af værdier som ikke giver mening i forbindelse med beregningen [Vinje 5.4.2]:

- Løbetiden skal være minimum 1 md.
- Udbetalingsprocenten skal være under 100 %
- Dagsrenten skal være $> -\infty$ og $< \infty$ (*en mere restriktiv begrænsning vil helt sikkert være fornuftig*)

Med denne behandlingsregel defineret benytter vi nu en fast fremgangsmåde for datadrevet test:

1. Identificér det gyldige og det ugyldige værdiområde
2. Opdel de to områder i ækvivalensklasser
3. Beskriv grænseværdier

Værdiområder

Tabel 2 viser input til renteberegningen og deres gyldige værdiområder.

<u>Input</u>	<u>Type</u>	<u>Gyldigt værdiområde</u>	<u>Særlige, ugyldige værdier</u>
Dagsrenten	double	$] -\infty, \infty [$	Double.NaN
Kreditværdighed	Rating enum	$\{A, B, C\}$	null
Udbetalingsprocent	double	$[0.20, 1[$	Double.NaN
Løbetid	int	$]0, Integer.MAX_VALUE]$	

Tabel 2: Input og værdiområder for renteberegning

Ækvivalensklasser

Tabel 3 viser de identificerede ækvivalensklasser. Vi vælger at lade ækvivalensklasser med samme ugyldige input sammenfatte til én. Dermed får vi reduceret i alt $4 \cdot 5 \cdot 5 \cdot 3 = 300$ klasser til blot 21.

Klasse#	Dagsrente	Kreditværdighed	Udbetalingspct.	Løbetid / mdr.
1	1.00%	A	50.00%	37
2				36
3				0
4			20.00%	37
5				36
6			19.99%	37
7			100.00%	37
8			Nan	37
9		B	50.00%	37
10				36
11			20.00%	37
12				36
13		C	50.00%	37
14				36
15			20.00%	37
16				36
17		D	50.00%	37
18		null	50.00%	37
19	Nan	A	50.00%	37
20	POS_INFY	A	50.00%	37
21	NEG_INFY	A	50.00%	37

Tabel 3: Identificerede ækvivalensklasser

Grænseværdianalyse

"Behovet for test af grænseværdier bygger på erfaringer. Der begås oftere fejl i grænseområderne end andre steder" [Vinje, s. 224]

Visse grænser afspejles allerede i ækvivalensklasserne. Øvrige grænseværdier vi må supplere vores test suite med, er:

- En udbetalingsprocent på 49.99 %
- En udbetalingsprocent på 99.99 %
- En løbetid på 1 md.

Test cases

Vi lader os inspirere direkte af ækvivalensklasserne når vi skal definere vores test cases. Således ser TC-01 ud som i tabel 4. De øvrige kan findes i bilag B.

TC#	INPUT	OUTPUT	
1	Dagsrente	1.00%	Rente 3.00%
	Kreditværdighed	A	
	Udbetalingspct.	50.00%	
	Løbetid	37	

Tabel 4: TC-01

Knap halvdelen af de i alt 24 test cases drejer sig om ugyldigt input, helt som ventet.

"Det er en god tommelfingerregel, at halvdelen af alle testcases skal beskæftige sig med det forbudte, det usandsynlige og det umulige" [Vinje, s. 222]

TDD – Test Driven Development (Karsten)

Til implementering af vores test cases og udvikling af produktionskoden til renteberegning anvendte vi Test Driven Development (TDD). TDD er en agil praksis relateret til Test-First Programming-koncepter fra Extreme Programming (XP). Her handler det om at skrive unit testen først for derefter at implementere koden der skal testes. Denne fremgangsmåde øger testbarheden af kildekoden helt naturligt idet den implementeres med testen for øje. Effekten af dette er bl.a. øget tillid til implementationen i kraft af grundige tests, og det letter bekymringer om refaktorisering. Desuden sørger metoden for at fokus fastholdes under implementering så kodens funktionalitet begrænses til at opfylde de definerede test cases—og ikke mere end det. Feature eller scope creep er et kendt problem i branchen, og her kan TDD være en løsning.

Særlige værdier i TDD (og XP i øvrigt) er 1) software produceres ved at skrive kode, 2) tag små skridt, og 3) hold fokus. Ved anvendelsen af TDD opnår vi en række fordele:

- Ren kode der virker, opbygget ad små skridt og refaktoriseret undervejs.
- En struktureret proces for hvordan vi kan starte implementering for et problem. Vi kan anvende konkrete Green/Red Bar-mønstre der sætter os i gang frem for at arbejde ud fra en mavefornemmelse.
- Hurtig feedback er et boost til tillid og moral. Hurtige, små iterationer nedbryder problemet i små, overkommelige skridt som er hurtigt løst. Hver green bar er en succes der bringer os et skridt tættere på at have løst opgaven, og får vi red bar, ved vi at fejlen næsten med garanti må ligge i den nye kode tilføjet i den aktuelle iteration.
- Stærkt fokus på pålidelig software. Med vores tests kan vi have tillid til programmet selv efter omfattende refaktorisering. Det gør op med "if it ain't broke, don't fix it"-mentaliteten hvor softwareudvikling styres af frygten for at introducere defects.
- Uafhængigt af øvrig dokumentation udgør vores test cases i sig selv detaljeret, up-to-date dokumentation af hvordan vores klasser anvendes (især med Evident Data-mønstret).

Figur 4 viser det typiske workflow i TDD.

[Christensen, s. 14] beskriver rytmen i TDD:

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

Skåret ud i pap er workflowet:

1. Tilføj test case til testliste
2. Kør test suite
Hvis red bar
3. Ændr kildekode
4. Gå til 2.

Hvis green bar

3. Tjek for koderedundans
Hvis koderedundans
4. Refaktoriser kildekode
5. Gå til 2.

Hvis ingen redundans

4. Implementer ny test case
5. Gå til 2.

Figur 4: TDD workflow

Fake It ('Til You Make It)

Et Green Bar-mønster der anvendes til hurtig progression når test suites fejler, er Fake It. Her implementerer vi blot den simplest mulige løsning for at bestå en test case, fx ved returnering af en konstant med den forventede værdi eller en null-reference.

Forløbet i figur 4 efter red bar bliver da:

Hvis red bar

3. Ændr kildekode
4. Tilføj ny test case til testliste
5. Gå til 2.

Her anvender vi Triangulation og tilføjer en ny test case til testlisten som skal sikre at den naive løsning senere fejler. Fake It hjælper os således til at tage små skridt og holde fokus på at komme hurtigst muligt videre.

Triangulation

Et Green Bar-mønster der ligger i tæt relation til Fake It er Triangulation. Med Triangulation kan vi opbygge abstraktion gradvist vha. adskillige datapunkter. Således kan skiftevis brug af Fake It og Triangulation bidrage til gradvis konstruktion af en kompleks algoritme i takt med at anvendelsen af Fake It resulterer i nye test cases med nye datapunkter.

One Step Test

One Step Test er et eksempel på et Red Bar-mønster som anvendes når test suiten lykkes. Udgangspunktet her er at gå til implementering af den test case som umiddelbart virker nemmest. Der er fokus på hurtige iterationer, og implementering af test og løsning bør kun tage ganske få minutter. Begynder en implementering at være længere, kan Fake It træde til som en måde at komme videre på. Ved test og udvikling af løsningen til renteberegning var One Step Test udgangspunktet for implementering af de test cases der blot forventer en Exception på ugyldigt input.

Andre relevante testmønstre

- Obvious Implementation
 - Test er en afvejning af cost vs. benefit. Der opstår typisk situationer hvor de enkelte skridt i Fake It og Triangulation synes så små at det bedre kan betale sig at implementere en åbenlys løsning ad én omgang.
- Evident Tests
 - Vi forsøger så vidt muligt at undgå defects i vores tests ved at holde testkoden åbenbar og så simpel som muligt. Ideelt set består testkoden udelukkende af assignments og assertions; løkker og andre komplekse kontrolstrukturer undgås. [Christensen, s. 23]
- Evident Data
 - Vi viser hensigten med datapunkterne og tydeliggør hvad der er forventet og faktisk output. Det dokumenterer testen i sig selv så vel som enheden der testes.

Whitebox test (Louise)

I modsætning til Blackbox test, er Whitebox test en test, der er baseret på det fysiske system, og ud fra dette udarbejdes der testdata.

Det er normalt programmører og teknikkere, der udfører whitebox-test og komponent test.

Der bliver talt om 3 typer test:

- Statement Coverage - Hvor det gælder om, at sætte sig et mål (typisk er målet 100 %) om hvor stor en del af koden, man ønsker at afdække.
- Decision Coverage - Også kaldet "Branche Coverage". Dækker forgreninger i systemet. Der er ikke tale om almindelige IF sætninger (målet her bør også være 100 %)
- Path Coverage - Gælder om at afdække alle mulige veje til det samme mål. Her kan det være svært at opnå 100 %, men dette bør stadig være målet.

Vi taler desuden også om Condition Coverage, der afdækker tests af programmet i forskellige stadier.

Reviews (Louise)

En af de ting, som vi med fordel kunne have fokuseret kraftigere på, er reviews.

Reviews er vigtige til udførsel af tidlig test.

Når man udfører reviews, sætter en række mennesker sig sammen i en gruppe og gennemgår sammen produktet.

Dette er dog lykkedes os at gennemfører en form for reviews af mange af vores modeller, i form af samlet gennemgang af de enkelte modeller. Derfor har vi fået en del af bonus af både erfaringsudveksling, etablering af standarder og bedre og mere ensartet planlægning.

Vi kunne dog formentligt have haft megen gavn af den strukturerede review, indlagt allerede fra start.

Hvis man har et ønske om, at udfører tidlige tests kan der desuden bruges:

- Prototyper
- Tænk højt test

Et sted hvor reviews desuden giver god mening er i forståelse af forskellige termer. Det kan være svært på egen hånd, at identificerer de ord, der ikke er enighed omkring. Men når en review-gruppe sidder samlet, vil det hurtigt vise sig, hvilke ord der falder udenfor.

Reviews hjælper til:

- 0 - fejls udvikling
- Opfyldelse af krav
- Opretholdelse af standarder
- Ensartethed

Når vi taler om reviews, så taler vi endvidere også om Verificering og Validering, der ikke bør forveksles.

Definitioner:

- **Verificering** - Når udvikleren gennemgår data- og klasser modeller, for at sikre, at det er dokumenteret på den korrekte måde.
- **Validering** - Når brugeren gennemgår systemet, fx vha. prototyper, for at sikre, at det er de korrekte handlinger der udføres, vha. virkelighedstro scenarier.

Ud fra disse to definitioner, kan vi konkludere, at vi har verificeret dele af vores system, men at der er en overvældende mangel på validering.

Der findes flere typer test, der kunne være udført. En af de typer er Walk-Throught formålet her er at se, om programmet fungerer i virkelige scenarier. I vores ville dette være at lege sælger, og derefter gennemfører fx en oprettelse af en låneanmodning. Herved kunne vi se om programmet er i stand til at håndterer dette.

Vi er her nødt til at have brugt nogen tid på forberedelse og identificere, hvilke fejl, der kunne opstå.

Denne type test er procedureorienteret. Dvs. at det skal udføres i den rækkefølge programmet skal afvikles. Denne type reviews kan udføres på alle dynamiske modeller. Dvs. sekvensdiagram, prototyper el. use cases.

Man kan også udfører Play-Throught, som på mange måder minder om Walk-Throught, med den forskel, at det i denne type er mennesker der "spiller" de forskellige roller i systemet.

Vi lavede den fatale fejl i vores projekt, at vi ikke fik skrevet reviews på projektplanen som en selvstændig opgave, ergo havde vi ikke den nødvendige tid til udførslen af dem.

Construction (konstruktionen) (Louise)

Construction er en meget stor fase, dette behøver dog ikke afspejle at der ligger kalendermæssigt meget tid i den. Når man træder ind i denne fase er alt beskrevet, og der skal blot kodes (og selvfølgelig holde alle beskrivelser up to date).

I denne fase kommer der ofte mange ekstra mennesker på projektet. Da det her både er muligt at sætte mindre erfarne programmører på eller benytte sig af outsourcing.

I denne fase ligger al det sidste kode, samt en masse test. Det gælder om hurtigst muligt at få et funktionelt program til verden, så der kan påbegyndes test af dette.

Når man forlader denne fase, skal der tages stilling til, om systemet er klar til en release. og om kunden er klar til at få programmet i Transition

I et så lille projekt som vores, er construction ikke en fase, man behøver at gå i. Da vi meget sent i forløbet har alt fuldt beskrevet.

Transition (overdragelsen)(Louise)

Transition er den sidste fase. Formålet med denne fase er, at få produktet ud til brugeren. Når dette sker vil der oftest tilstøde problemer el. lign, der skal omkodes, laves ny release el. færdiggøres nogle features.

I denne fase vil der ofte ligge en el. anden form for beta test, så man ved, at programmet stemmer overens med kundens forventninger. Desuden vil man sørge for oplæring af fremtidige brugere, og at få kørt programmet ud til alle fremtidige brugere.

I nogle projekter vil dette bare anmærke starten på et nyt projekt, Da der blot startes på udvikling af næste version.

I vores projekt har vi ikke haft en Transition fase, og det tætteste vi kommer på, må siges enten at være aflevering af projektet el. selve eksamen.

Konklusion

Vi har i den seneste måned arbejdet med Ferrari Finance System, der har været vores projekt Case.

Vi har arbejdet, som et firemandsteam under systemudviklingsmetoden Unified Process. Denne udviklingsmetode har voldt os nogle problemer, både i form af opretholdelse af projektplanen, samt dokumentering af udførslen af de forskellige opgaver.

Hvilket til dels kan forklares med, at UP ikke primært henvender sig små grupper. Når det er sagt, ville vi formentligt have fået mere gavn af UP, hvis gruppen havde indeholdt over mindst et medlem, der havde haft en større rutine.

Sammen med den manglende rutine har vi lidt under manglende delegering af roller, der foruden en meget anarkistisk tilgang i gruppen, har bidraget til, at der al for længe er sket udvidelser og tiltag i udviklingen.

Git og vores Teamstørrelse har endvidere også bidraget, at det har været svært at opretholde en brugbar projektplan, både grundet en god kommunikation internt i gruppen, og pga. en følelse af overblik vha. Git.

Vores Business Project Reengineering gik relativt godt, her agerede vi alle Business-Process Analyst. Den samme ro gjorde sig gældende i Inception-fasen, der endte som en 2-dages iteration med udvikling af Visionsdokumentet og identificering af Use Cases.

I Elaboration opstod problemerne, da det var i denne fase, vi ikke fik specifiseret rollerne, og herved oplevede at designbeslutninger blev truffet, hver gang vi fik et godt overblik over systemet. Således lykkedes det os, at have en relativ lille indsigt i det samlede system, samt en konstant opgave i at verificerer dokumentation og udvide.

Den ovenstående Elaboration fase resulterede i, at det simpelthen ikke lykkedes os, at gå i Construction, da vi aldrig nåede at opfylde det essentielle krav for at gå i Construction "Al kode er dokumenteret"

Som forventet var Transition ikke en fase vi stilede efter.

Vi oplevede desuden, at det manglende overblik gjorde, at vi ikke fik udført den mængde test, som vi havde ønsket og tænkt i starten af projektet.

Når vi ser bort fra den overordnede mangel på intern struktur, er det lykkedes os, at udarbejde et fornuftigt Ferrari Finance System, hvor der er lagt stor vægt på arkitektur, lagdeling og overholdelse af GRAPS patterns.

Dette har ført til en række overskuelige, små klasser af Information Experts og et par meget overskuelige Controllere.

Desuden har grundige overvejelser omkring Design Patterns ført til valget af Facade Pattern og Command Pattern. Dette bidraget til en øget maintainability, da disse patterns er nogle af de meget velkendte. Desuden er Facade Pattern yderst velegnet til applikationer, der er opbygget efter Use Cases.

Det er desuden lykkedes os at løse trådproblemet på en måde så vi udnytter Javas indbyggede trådhåndtering mest muligt.

Litteraturliste

Bøger

- The Rational Unified Process an Introduction - Philippe Kruchten
- Applying UML and Patterns - Craig Larman
- Software Tests - Poul Staal Vinje
- Kap. II om Test Driven Development - Christensen

Links

- http://en.wikipedia.org/wiki/Unified_Process
- http://en.wikipedia.org/wiki/Waterfall_model
- http://www.kim-andersen.dk/database_normalisering/mysql_anden_normalform_database.htm
- <https://docs.oracle.com/javase/tutorial/essential/concurrency/>
- <http://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html>

Bilag

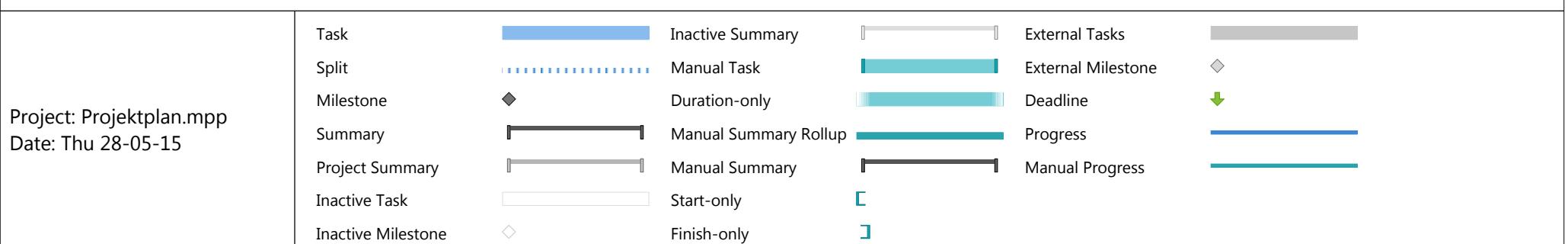
Udgangspunkter for vores Eksamens Projekt 2. sem.:

- Vi holder som udgangspunkt fri onsdage, dog bør man forvente at der arbejdes derhjemme.
- Er man forhindret, giver man besked via mail til alle i gruppen:
 - o Louise Niemann Hansen (Niemannlouise@gmail.com)
 - o Henrik Broe Henriksen (henrik.broe@gmail.com)
 - o Enok Nørager Mikkelsen (enokmik@live.dk)
 - o Karsten Heino Karlsen (k.heino.karlsen@gmail.com)
- Deadline for afbud er kl. 7:30, ellers er der buksevand.
- Som udgangspunkt mødes vi kl. 9.00 og arbejder til 15.00
- Uden for skolen foregår arbejdskommunikere som udgangspunkt over skype, i nødstilfælde over mail.

Brug af Git-hub:

- Vi laver tidsregistering i en og samme branch.
- Vi opretter en "v0.00" branch nu, og burger den ind til vi mener at vi er klar til næste version.
- Vi burger en notation hvor vi kører versions nummer op hver gang vi har implementeret en use case. Dvs. når Use Case 3 er implementeret vil vi gå fra v0.2 til v0.3.

ID	Task Mode	Task Name	Duration	Start	Finish	Pred.	Resource Initials	Baseline Duration	Work
1		Hele Projektet	243,58 hrs?	Mon 27-04-15	Thu 28-05-15			0 hrs?	152,25 ...
2		BPR	3,5 hrs	Mon 27-04-15	Mon 27-04-15	9		3 hrs?	14 hrs
3		0. Iteration	3,5 hrs	Mon 27-04-15	Mon 27-04-15			0 hrs?	14 hrs
4	?	Identifier Use Cases	45 mins	Mon 27-04-15	Mon 27-04-15		E;H;K;L	30 mins?	3 hrs
5	?	Optegn Use Case Diagram	15 mins	Mon 27-04-15	Mon 27-04-15	4	E;H;K;L	30 mins?	1 hr
6	?	Tegn Objektmodeller	2,5 hrs	Mon 27-04-15	Mon 27-04-15	5	E;H;K;L	2 hrs?	10 hrs
7		Inception	5,5 hrs	Mon 27-04-15	Mon 27-04-15			0 hrs?	8 hrs
8		1. Iteration	5,5 hrs	Mon 27-04-15	Mon 27-04-15			27 hrs?	8 hrs
9	?	Fastlæg grupperegler	30 mins	Mon 27-04-15	Mon 27-04-15		E;H;K;L	1 hr?	2 hrs
10		Visionsdokument	1,25 hrs	Mon 27-04-15	Mon 27-04-15	2		3 hrs?	5 hrs
11	?	Interessantanalyse	15 mins	Mon 27-04-15	Mon 27-04-15		E;H;K;L	30 mins?	1 hr
12	?	Supplerende krav specifikation	30 mins	Mon 27-04-15	Mon 27-04-15	11	E;H;K;L	30 mins?	2 hrs
13	?	Featureliste	15 mins	Mon 27-04-15	Mon 27-04-15	12	E;H;K;L	1 hr?	1 hr
14	?	Vision	15 mins	Mon 27-04-15	Mon 27-04-15	13	E;H;K;L	1 hr?	1 hr
15	?	Opstart Dataordbog	15 mins	Mon 27-04-15	Mon 27-04-15	10	E;H;K;L	1 hr?	1 hr
16		Elaboration	196,93 hrs?	Mon 27-04-15	Fri 22-05-15			0 hrs?	130,25 ...
17		2. Iteration	52,15 hrs?	Mon 27-04-15	Sun 03-05-15			0 hrs?	49 hrs
18		OOA	2,5 hrs	Tue 28-04-15	Tue 28-04-15			5 hrs?	10 hrs
19		Use cases	1,75 hrs	Tue 28-04-15	Tue 28-04-15			4 hrs?	7 hrs
20	?	Optegn Use Case Diagram	30 mins	Tue 28-04-15	Tue 28-04-15		E;H;K;L	30 mins?	2 hrs
21	?	Skriv Casual Use Cases	30 mins	Tue 28-04-15	Tue 28-04-15	20	E;H;K;L	2 hrs?	2 hrs
22	?	Fully dressed på den mest centrale	45 mins	Tue 28-04-15	Tue 28-04-15	21	E;H;K;L	1,5 hrs?	3 hrs

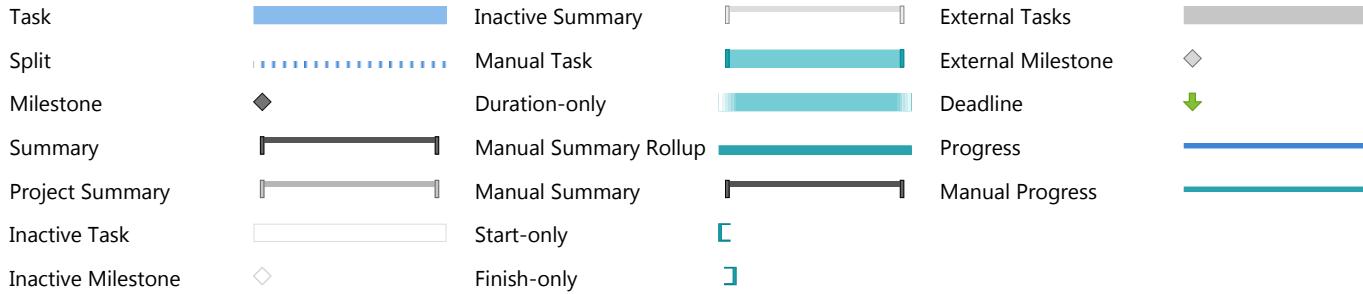


ID	Task Mode	Task Name	Duration	Start	Finish	Pred.	Resource Initials	Baseline Duration	Work
23	👤	Udarbejd domænemodel	45 mins	Tue 28-04-15	Tue 28-04-15	19	E;H;K;L	1 hr?	3 hrs
24	👤	Risikoanalyse	75 mins	Tue 28-04-15	Tue 28-04-15	18	E;H;K;L	2 hrs?	5 hrs
25	👤	Mock ups	1,75 hrs	Tue 28-04-15	Tue 28-04-15	24		5 hrs?	7 hrs
26	👤	Afklar forventninger til Udseende	75 mins	Tue 28-04-15	Tue 28-04-15		E;H;K;L	4 hrs?	5 hrs
27	👤	Rentegn Mock ups	30 mins	Tue 28-04-15	Tue 28-04-15	26	E;H;K;L	1 hr?	2 hrs
28	👤	Arkitektur	46,48 hrs	Tue 28-04-15	Sun 03-05-15	25		0 hrs?	9 hrs
29	📅	SystemSekvensdiagram	1 hr	Sat 02-05-15	Sat 02-05-15		H	1 hr?	1 hr
30	👤	SekvensDiagram	1,5 hrs	Tue 28-04-15	Wed 29-04-15		L	1 hr?	1,5 hrs
31	👤	KlasseDiagram	0,5 hrs	Wed 29-04-15	Wed 29-04-15	30	L	1 hr?	0,5 hrs
32	📅	Operations Kontrakt	3 hrs	Sat 02-05-15	Sat 02-05-15	29	H	1 hr?	3 hrs
33	👤	Aktivitets Diagram	1 hr	Wed 29-04-15	Wed 29-04-15	31	L	1 hr?	1 hr
34	📅	Pakkediagram	2 hrs	Sun 03-05-15	Sun 03-05-15		K	4 hrs?	2 hrs
35	👤	Prototyper	6 hrs	Tue 28-04-15	Wed 29-04-15	25		0 hrs?	8,5 hrs
36	👤	CSV	2,5 hrs	Tue 28-04-15	Wed 29-04-15		K	5 hrs?	2,5 hrs
37	👤	Sqlite	6 hrs	Tue 28-04-15	Wed 29-04-15		E	5 hrs?	6 hrs
38	👤	Opdatering af Projektplan	1,5 hrs	Tue 28-04-15	Wed 29-04-15	26	K	1,5 hrs?	1,5 hrs
39	📅	Projektplan for 2. Iteration	2 hrs?	Thu 30-04-15	Thu 30-04-15	8	E;L	2 hrs?	4 hrs
40	👤	1. Fællessamling på klassen	1 hr?	Mon 27-04-15	Tue 28-04-15	8	E;H;K;L	8 hrs?	4 hrs
41	👤	3. Iteration	16,75 hrs?	Mon 27-04-15	Wed 29-04-15	8		0 hrs?	47 hrs
42	👤	Opfølgning & Evaluering af 1. Interation	1 hr	Mon 27-04-15	Tue 28-04-15		E;H;K;L	0,5 hrs?	4 hrs
43	👤	Gennemgang af hjemmearbejde(diagrammer etc)	1 hr	Tue 28-04-15	Tue 28-04-15	42	E;H;K;L	1 hr?	4 hrs

Project: Projektplan.mpp Date: Thu 28-05-15	Task		Inactive Summary		External Tasks	
	Split		Manual Task		External Milestone	
	Milestone		Duration-only		Deadline	
	Summary		Manual Summary Rollup		Progress	
	Project Summary		Manual Summary		Manual Progress	
	Inactive Task		Start-only			
	Inactive Milestone		Finish-only			

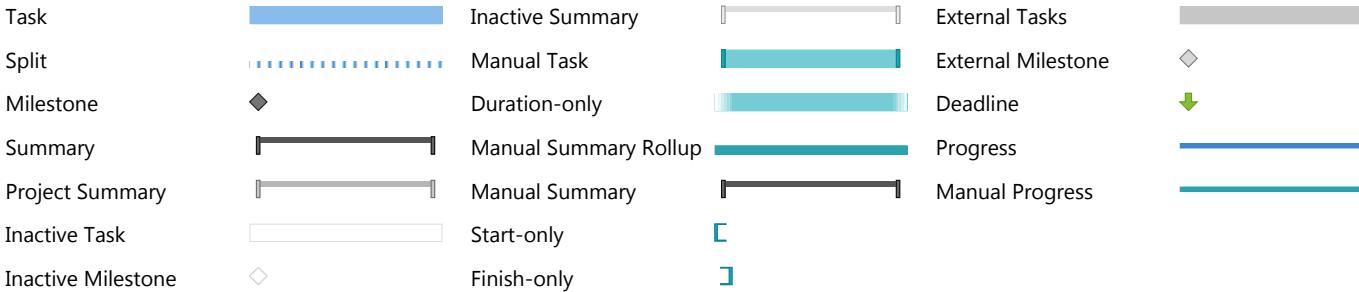
ID	Task Mode	Task Name	Duration	Start	Finish	Pred.	Resource Initials	Baseline Duration	Work
44	⌚➡️	Opfølgning på BPR: Objektmodel	30 mins	Tue 28-04-15	Tue 28-04-15	43	E;H;K;L	1 hr?	2 hrs
45	⌚➡️	Test Suite	1 hr	Mon 27-04-15	Tue 28-04-15		L	2 hrs?	1 hr
46	⌚➡️	Revider SSD-FFS-01	4 hrs	Mon 27-04-15	Tue 28-04-15		K	2 hrs?	4 hrs
47	⌚➡️	Udarbejd FFS-OC-01..13	1,75 hrs	Tue 28-04-15	Tue 28-04-15	46	K	2 hrs?	1,75 hrs
48	⌚➡️	Bruger Test	5 hrs?	Tue 28-04-15	Wed 29-04-15	45		0 hrs?	20 hrs
49	⌚➡️	Udarbejde Bruger Test	2 hrs	Tue 28-04-15	Tue 28-04-15		E;L;H;K	2 hrs?	8 hrs
50	⌚➡️	Afhold & Dokumenter Bruger Test	2 hrs?	Tue 28-04-15	Tue 28-04-15	49	E;K;L;H	2 hrs?	8 hrs
51	⌚➡️	Reflektion over Bruger test	1 hr?	Tue 28-04-15	Wed 29-04-15	50	E;H;K;L	2 hrs?	4 hrs
52	⌚➡️	Implementation	3,5 hrs?	Wed 29-04-15	Wed 29-04-15	48		0 hrs?	3 hrs
53	⌚➡️	Opsætning af Eclips projekt	1,5 hrs	Wed 29-04-15	Wed 29-04-15			1 hr?	0 hrs
54	⌚➡️	Opsætning på vores computere.	2 hrs?	Wed 29-04-15	Wed 29-04-15	53		2 hrs?	0 hrs
55	⌚➡️	Domain klasser oprettes og laves	3 hrs	Wed 29-04-15	Wed 29-04-15		L	2 hrs?	3 hrs
56	⌚➡️	FFS - 02	7,25 hrs?	Wed 29-04-15	Wed 29-04-15	52		0 hrs?	7,25 hrs
57	⌚➡️	Casual Use Case	15 mins	Wed 29-04-15	Wed 29-04-15		E	15 mins?	0,25 hrs
58	⌚➡️	Fully dressed Use Case	7 hrs?	Wed 29-04-15	Wed 29-04-15	57	E	30 mins?	7 hrs
59	⌚➡️	4. iteration	133,7 hrs?	Wed 29-04-15	Mon 18-05-15	41		0 hrs?	34,25 hrs
60	⌚➡️	Tjek af milepæl B	3 hrs?	Wed 29-04-15	Wed 29-04-15			3 hrs?	0 hrs
61	⌚➡️	FFS: 01	6,5 hrs?	Wed 29-04-15	Wed 29-04-15			0 hrs?	6,5 hrs
62	⌚➡️	GUI	6,5 hrs?	Wed 29-04-15	Wed 29-04-15			0 hrs?	6,5 hrs
63	⌚➡️	Login GUI	2 hrs	Wed 29-04-15	Wed 29-04-15		E	2 hrs?	2 hrs
64	⌚➡️	Menu GUI	1,5 hrs	Wed 29-04-15	Wed 29-04-15	63	E	2 hrs?	1,5 hrs
65	⌚➡️	Kunde GUI	3 hrs	Wed 29-04-15	Wed 29-04-15	64	E	2 hrs?	3 hrs

Project: Projektplan.mpp
Date: Thu 28-05-15



ID	Task Mode	Task Name	Duration	Start	Finish	Pred.	Resource Initials	Baseline Duration	Work
66	⌚	Bil GUI	2 hrs?	Wed 29-04-15	Wed 29-04-15			2 hrs?	0 hrs
67	⌚	DataAccess	3 hrs?	Wed 29-04-15	Wed 29-04-15			4 hrs?	0 hrs
68	⌚	DataAccess	15 mins?	Wed 29-04-15	Wed 29-04-15			15 mins?	0 hrs
69	⌚	CustomerDataAccess	3 hrs?	Wed 29-04-15	Wed 29-04-15			3 hrs?	0 hrs
70	⌚	LoanRequestDataAccess	3 hrs?	Wed 29-04-15	Wed 29-04-15			3 hrs?	0 hrs
71	⌚	CarModelDataAccess	3 hrs?	Wed 29-04-15	Wed 29-04-15			3 hrs?	0 hrs
72	⌚	CarDataAccess	3 hrs?	Wed 29-04-15	Wed 29-04-15			3 hrs?	0 hrs
73	⌚	Logic	4 hrs?	Wed 29-04-15	Wed 29-04-15			8 hrs?	0 hrs
74	⌚	Controller-klasser	4 hrs?	Wed 29-04-15	Wed 29-04-15			0 hrs?	0 hrs
75	⌚	CustomerController	30 mins	Wed 29-04-15	Wed 29-04-15			1 hr?	0 hrs
76	⌚	CustomerControllerImpl	3 hrs?	Wed 29-04-15	Wed 29-04-15			3 hrs?	0 hrs
77	⌚	CarController	1 hr	Wed 29-04-15	Wed 29-04-15			1 hr?	0 hrs
78	⌚	CarControllerImpl	4 hrs?	Wed 29-04-15	Wed 29-04-15			4 hrs?	0 hrs
79	⌚	StartController	30 mins	Wed 29-04-15	Wed 29-04-15			30 mins?	0 hrs
80	⌚	StartControllerImpl	3 hrs?	Wed 29-04-15	Wed 29-04-15			3 hrs?	0 hrs
81	⌚	Logic-klasser	3 hrs?	Wed 29-04-15	Wed 29-04-15			0 hrs?	0 hrs
82	⌚	CustomerLogic	1 hr?	Wed 29-04-15	Wed 29-04-15			1 hr?	0 hrs
83	⌚	CustomerLogicImpl	3 hrs?	Wed 29-04-15	Wed 29-04-15			3 hrs?	0 hrs
84	⌚	LoanRequestLogic	1 hr?	Wed 29-04-15	Wed 29-04-15			1 hr?	0 hrs
85	⌚	LoanRequestLogicImpl	3 hrs?	Wed 29-04-15	Wed 29-04-15			3 hrs?	0 hrs
86	⌚	CarModelLogic	1 hr?	Wed 29-04-15	Wed 29-04-15			1 hr?	0 hrs
87	⌚	CarModelLogicImpl	3 hrs?	Wed 29-04-15	Wed 29-04-15			3 hrs?	0 hrs

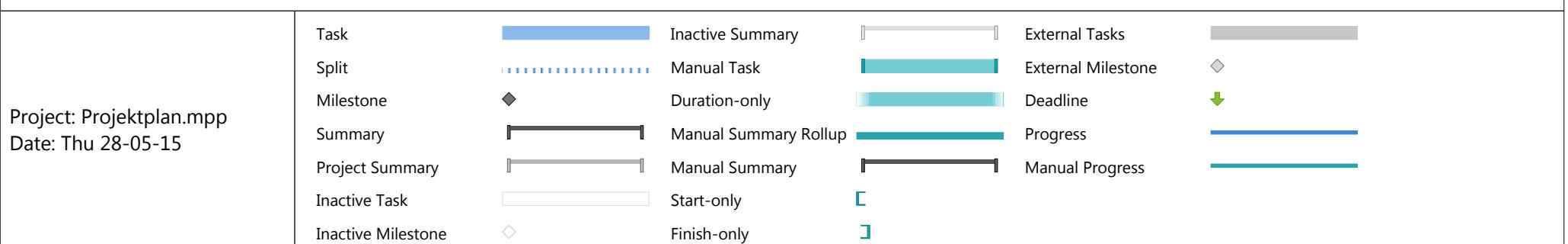
Project: Projektplan.mpp
Date: Thu 28-05-15



ID	Task Mode	Task Name	Duration	Start	Finish	Pred.	Resource Initials	Baseline Duration	Work
88		CarLogic	1 hr?	Wed 29-04-15	Wed 29-04-15			1 hr?	0 hrs
89		CarLogicImpl	3 hrs?	Wed 29-04-15	Wed 29-04-15			3 hrs?	0 hrs
90		Exceptions	2 hrs?	Wed 29-04-15	Wed 29-04-15			0 hrs?	0 hrs
91		Identificér Exceptions og tilhørende beskeder	2 hrs?	Wed 29-04-15	Wed 29-04-15			2 hrs?	0 hrs
92		Test	2 hrs?	Wed 29-04-15	Wed 29-04-15			0 hrs?	0 hrs
93		Opdatér TestSuite	2 hrs?	Wed 29-04-15	Wed 29-04-15			2 hrs?	0 hrs
94		Test I Junit	2 hrs?	Wed 29-04-15	Wed 29-04-15			2 hrs?	0 hrs
95	📅	1. Use Case er færdig implementeret	0 hrs	Mon 18-05-15	Mon 18-05-15			0 hrs?	0 hrs
96		FFS - 02	1 hr?	Wed 29-04-15	Wed 29-04-15			0 hrs?	7,75 hrs
97		Casual Use Case	15 mins	Wed 29-04-15	Wed 29-04-15	H		30 mins?	0,25 hrs
98		Fully Dressed Use Case	30 mins	Wed 29-04-15	Wed 29-04-15	H		30 mins?	0,5 hrs
99		Klassediagram	1 hr?	Wed 29-04-15	Wed 29-04-15	K		1 hr?	1 hr
100		Sekvensdiagram	1 hr	Wed 29-04-15	Wed 29-04-15	H		1 hr?	1 hr
101		Systemsekvensdiagram	1 hr	Wed 29-04-15	Wed 29-04-15	H		1 hr?	1 hr
102		Mockups	0,5 hrs?	Wed 29-04-15	Wed 29-04-15			30 mins?	1,5 hrs
103	📍	Fremstilling i gruppen	15 mins	Wed 29-04-15	Wed 29-04-15	E;H;K;L		30 mins?	1 hr
104		Renskrivning	30 mins?	Wed 29-04-15	Wed 29-04-15	L		30 hrs?	0,5 hrs
105		Operations Kontrakt	30 mins?	Wed 29-04-15	Wed 29-04-15	L		30 mins?	0,5 hrs
106		Aktivitets Diagram	1 hr	Wed 29-04-15	Wed 29-04-15	H		30 mins?	1 hr
107		Pakkediagram (ændres)	30 mins	Wed 29-04-15	Wed 29-04-15			15 mins?	0 hrs
108		Testsuite	1 hr?	Wed 29-04-15	Wed 29-04-15	H		1 hr?	1 hr
109		FFS - 03	2 hrs?	Wed 29-04-15	Wed 29-04-15			0 hrs?	20 hrs

Project: Projektplan.mpp Date: Thu 28-05-15	Task		Inactive Summary		External Tasks	
	Split		Manual Task		External Milestone	
	Milestone		Duration-only		Deadline	
	Summary		Manual Summary Rollup		Progress	
	Project Summary		Manual Summary		Manual Progress	
	Inactive Task		Start-only			
	Inactive Milestone		Finish-only			

ID	Task Mode	Task Name	Duration	Start	Finish	Pred.	Resource Initials	Baseline Duration	Work
110	👤	Klassediagram	1 hr?	Wed 29-04-15	Wed 29-04-15	E		1 hr?	1 hr
111	👤	Sekvensdiagram	1,5 hrs?	Wed 29-04-15	Wed 29-04-15	E		1,5 hrs?	1,5 hrs
112		Systemsekvensdiagram	1 hr?	Wed 29-04-15	Wed 29-04-15	L		1 hr?	1 hr
113		Mockups	1 hr?	Wed 29-04-15 Wed 29-04-15				3 hrs?	2 hrs
114	👤	Fremstilling I gruppen	15 mins	Wed 29-04-15	Wed 29-04-15	E;H;K;L		30 mins?	1 hr
115		Renskrivning	1 hr?	Wed 29-04-15	Wed 29-04-15	L		1 hr?	1 hr
116		Operations Kontrakt	1 hr	Wed 29-04-15	Wed 29-04-15	L		3 hrs?	1 hr
117		Aktivitets Diagram	1 hr?	Wed 29-04-15	Wed 29-04-15	K		1 hr?	1 hr
118	👤	Pakkediagram (ændres)	1 hr?	Wed 29-04-15	Wed 29-04-15	E;H;K;L		1 hr?	4 hrs
119		Testsuite	30 mins?	Wed 29-04-15	Wed 29-04-15	H		30 mins?	0,5 hrs
120	👤	Bruger Test af 2 og 3	2 hrs?	Wed 29-04-15	Wed 29-04-15	E;H;K;L		2 hrs?	8 hrs
121		5. iteration	7 hrs?	Mon 27-04-15 Tue 28-04-15				0 hrs?	0 hrs
122		FFS - 02	7 hrs?	Mon 27-04-15 Tue 28-04-15				0 hrs?	0 hrs
123		Implementation	7 hrs?	Mon 27-04-15	Tue 28-04-15			0 hrs?	0 hrs
124		FFS - 03	7 hrs?	Mon 27-04-15 Tue 28-04-15				0 hrs?	0 hrs
125		Implementation	7 hrs?	Mon 27-04-15	Tue 28-04-15			0 hrs?	0 hrs
126		FFS - 04	7 hrs?	Mon 27-04-15 Tue 28-04-15				0 hrs?	0 hrs
127		Beskriv	7 hrs?	Mon 27-04-15	Tue 28-04-15			0 hrs?	0 hrs
128		FFS - 05	7 hrs?	Mon 27-04-15 Tue 28-04-15				0 hrs?	0 hrs
129		Beskriv	7 hrs?	Mon 27-04-15	Tue 28-04-15			0 hrs?	0 hrs
130	📅	Eleboration er nu gennemført	0 hrs	Fri 22-05-15	Fri 22-05-15			0 hrs?	0 hrs
131		Construction	7 hrs?	Mon 27-04-15 Tue 28-04-15				0 hrs?	0 hrs

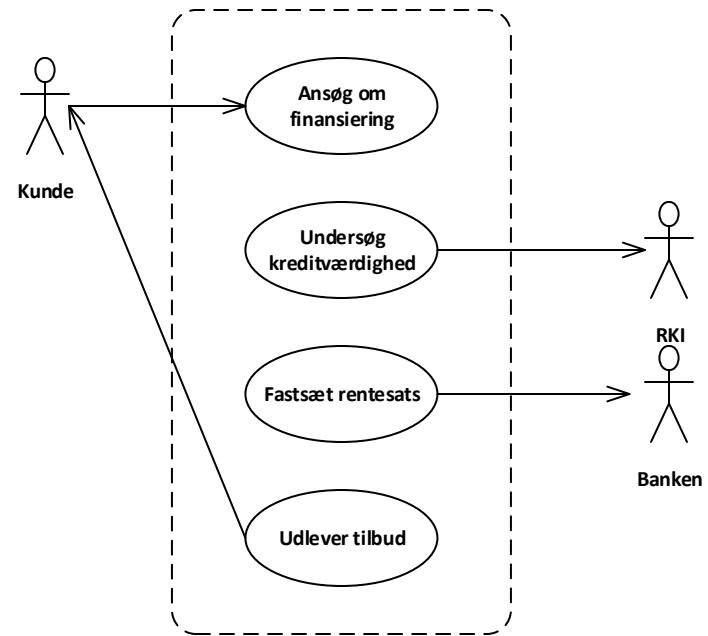


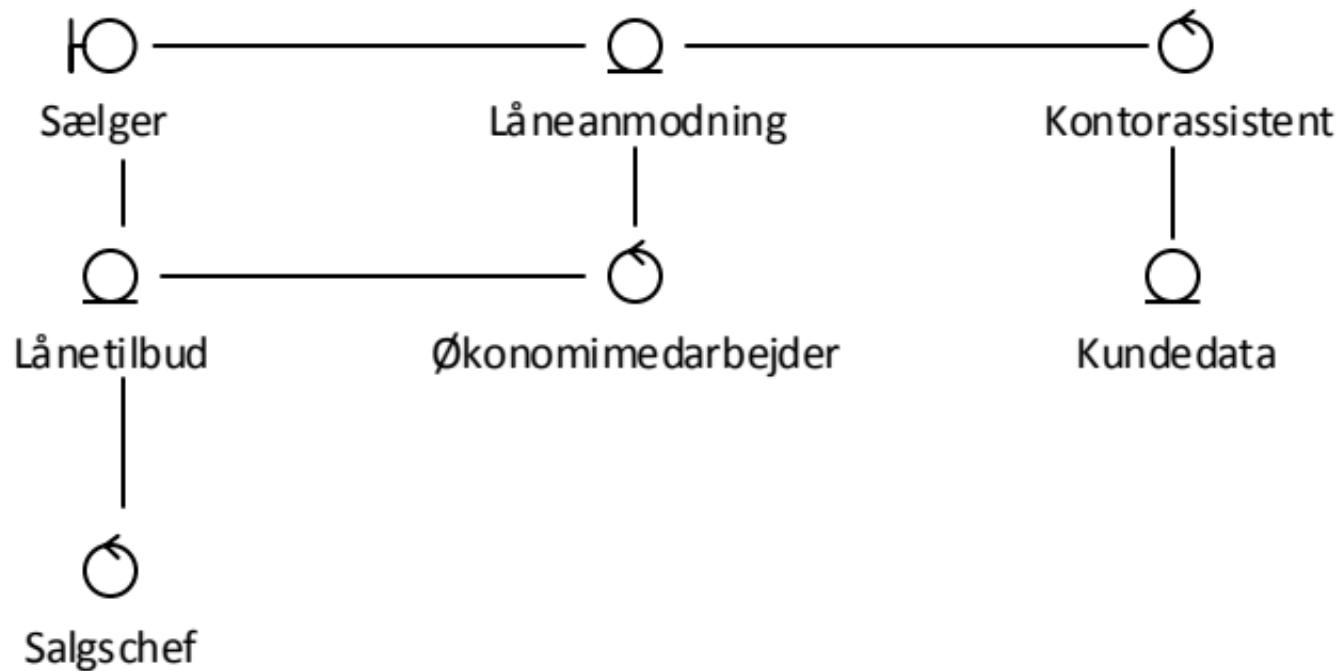
ID	Task Mode	Task Name	Duration	Start	Finish	Pred.	Resource Initials	Baseline Duration	Work
132		6. Iteration	7 hrs?	Mon 27-04-15	Tue 28-04-15			0 hrs?	0 hrs
133		FFS -04	7 hrs?	Mon 27-04-15	Tue 28-04-15			0 hrs?	0 hrs
134		Implementation	7 hrs?	Mon 27-04-15	Tue 28-04-15			0 hrs?	0 hrs
135		FFS - 05	7 hrs?	Mon 27-04-15	Tue 28-04-15			0 hrs?	0 hrs
136		Implementation	7 hrs?	Mon 27-04-15	Tue 28-04-15			0 hrs?	0 hrs
137		Test	7 hrs?	Mon 27-04-15	Tue 28-04-15			0 hrs?	0 hrs
138		7. iteration	7 hrs?	Mon 27-04-15	Tue 28-04-15			0 hrs?	0 hrs
139		Implementation	7 hrs?	Mon 27-04-15	Tue 28-04-15			0 hrs?	0 hrs
140		Test	7 hrs?	Mon 27-04-15	Tue 28-04-15			0 hrs?	0 hrs
141		Construction er færdig	0 hrs	Mon 27-04-15	Mon 27-04-15			0 hrs?	0 hrs
142		Transition	20,65 hrs?	Wed 27-05-15	Thu 28-05-15			0 hrs?	0 hrs
143		Eksportering og aflevering	20,65 hrs?	Wed 27-05-15	Thu 28-05-15			0 hrs?	0 hrs
144		Projektet er færdigt	0 hrs	Thu 28-05-15	Thu 28-05-15			0 hrs?	0 hrs

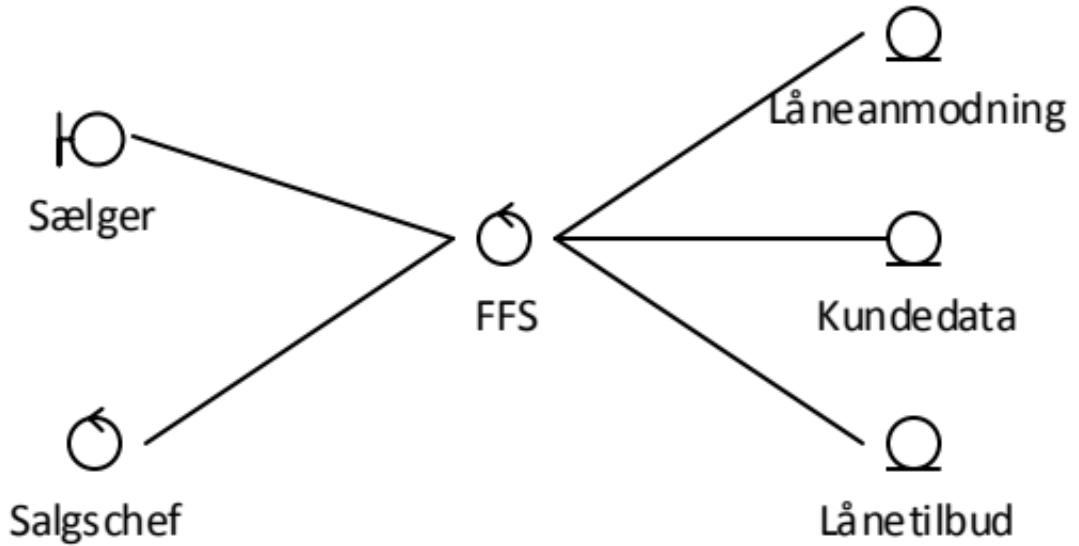
Project: Projektplan.mpp Date: Thu 28-05-15	Task		Inactive Summary		External Tasks	
	Split		Manual Task		External Milestone	
	Milestone		Duration-only		Deadline	
	Summary		Manual Summary Rollup		Progress	
	Project Summary		Manual Summary		Manual Progress	
	Inactive Task		Start-only			
	Inactive Milestone		Finish-only			

Risikoanalyse

Risiko	Sandsynlig (SS)	Konsekvens (K)	Prioritering($SS * K$)	RMM	Revideret SS	Revideret konsekvens
Hardwareproblemer	1	2 timer	2			
Tab af data	2	2 timer	4			
Uoverensstemmelser	4	4 timer	12			
Tab af motivation	3	6 timer	18			
11+ dages fravær	1	24 timer	30	Arbejder hjemmefra. Det er både gruppens og den syges opgave at være med til at kommunikere opgaver. Begrænset deltag.	1	4
Overordnet tab af motivation	1	60 timer	60	Kommunikation om situationen og løsning	1	6
5 - 10 dages fravær	3	24 timer	78	Arbejder hjemmefra. Det er både gruppens og den syges opgave at være med til at kommunikere opgaver.	3	4
Udfordringer i forbindelse m. implementering	4	20 timer	80	Udarbejd prototyper af mulige problemområder	1	20
Op til 5 dages fravær	5	30 timer	150	Regnet fra samlet tid	5	0
Manglende evne til begrænsning	4	40 timer	160	Burde regnes fra den samlede tid. Vi når ikke at færdigimplementere hele programmet.	4	40







Visionsdokument.

Vision

Vi forestiller os et FerrariFinanceSystem, som afspejler det brand som Ferrari er. Selve programmet skal understøtte det faktum, at Ferrari er et High-end produkt. Systemet skal være i stand til korrekt og sikkert at behandle oplysninger. Systemet skal være med til at skabe en unik følelse omkring købet af kundens nye Ferrari, der yderligere forstærkes i og med, at vi effektivt gennemfører låneforløbet, således at kunden er i stand til at kører væk i sin nye bil, den samme dag. Systemet skal være intuitivt, og derved være med til at give sælgeren mere frihed og overskud til at give en professionel betjening af kunden.

Interessentanalyse

Direktør:

- At salgsteamet for solgt mest muligt.
- At låneanmodning bliver korrekt registreret med det samme.
- At renten er korrekt beregnet.
- At data ikke går tabt i processen.
- At kontakt med bank/RKI forløber hurtigt og uden tab af data.
- At låneanmodning fra samme kunde ikke bliver behandlet flere gange.

Sælger:

- At jeg sælger mest!
- At låneanmodning kan godkendes hurtigst muligt.
- At låneanmodning fra samme kunde ikke bliver behandlet flere gange.
- At låneanmodning bliver korrekt registreret med det samme.
- Lettest mulig arbejdsgang

Kunden:

- At få svar på låneanmodning hurtigst muligt.
- At renten er korrekt beregnet.
- At oplysningerne bliver opbevaret sikkert.

Bank:

- Korrekt brug af deres service.

RKI:

- Korrekt brug af deres service.

Datatilsynet:

- At data opbevares i overensstemmelse med persondataloven.

Featureliste

- Oprettelse af anmodning om lånetilbud
- Håndtering af kunde
- Oprettelse af tilbud
 - Forespørgsels om kreditværdighed
 - Forespørgsels om rentesats
 - Fastsættelse af rentesats for hvert tilbud
- Eksport af lånetilbud og afdragsordning
 - CSV format

Supplerende kravspecifikation

- **Usability:**

80 % af de ansatte i firmaet, skal kunne løse deres opgaver vha. det nye system, uden hjælp, inden for fem minutter, den første gang de prøver det.

- **Reliability:**

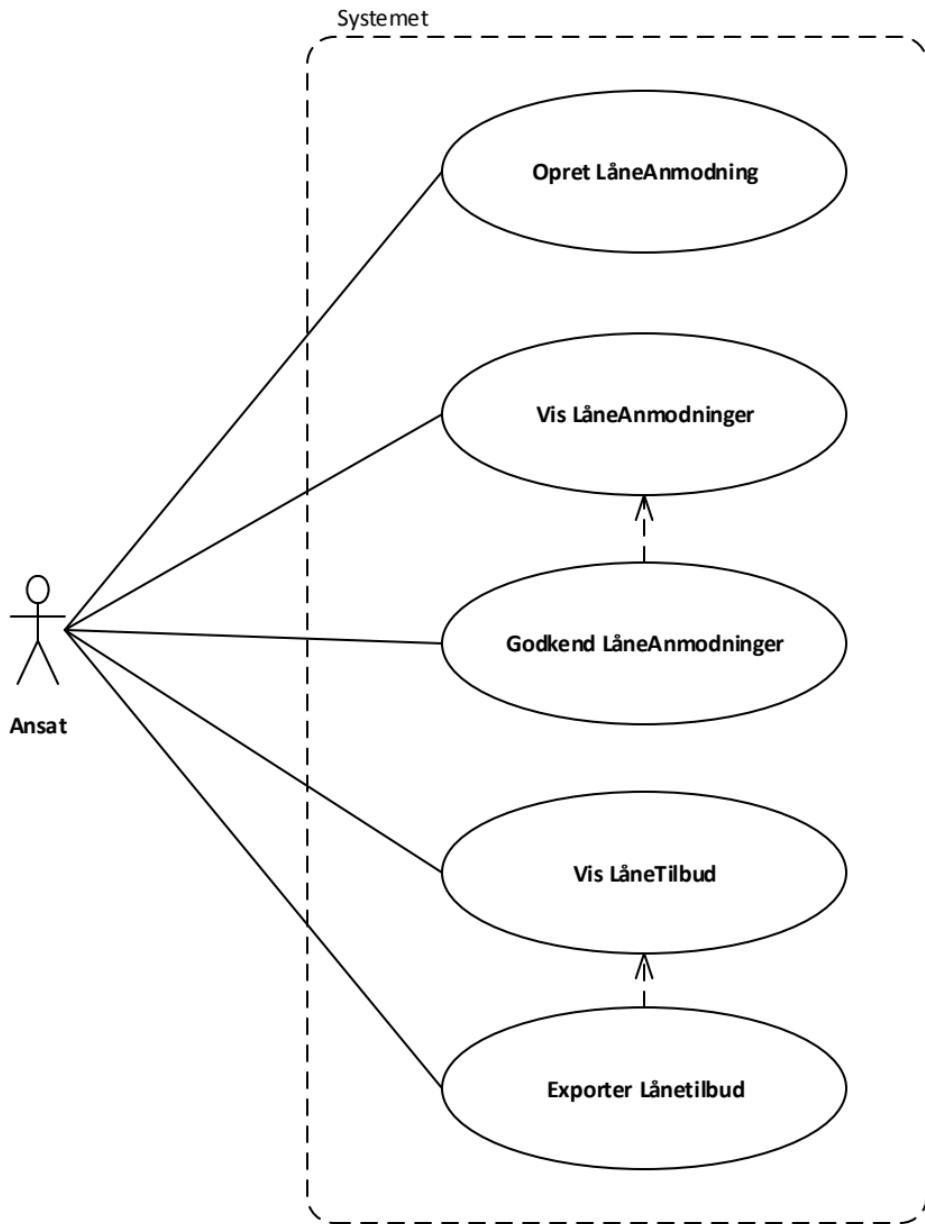
Systemet må højst være utilgængeligt 1,5 % af arbejdstiden.

- **Performance:**

Oprettelse af én låneanmodning må højst tage 45 sekunder, forudsat at systemet er operationelt og forespørgsel til Bank og RKI forløber problemfrit.

- **Supportability:**

Systemet skal være forberedt for flytning til web platform, så dette kan gennemføres let og hurtigt.



Use Case 0(Sub-funktion)

Casual Use Case

FFS - 00: Login

- *Sælgeren er ikke logget ind.*
Sælgeren ønsker at tilgå systemet.

Sælgeren angiver brugernavn
Sælgeren angiver adgangskode
Sælgeren anmoder systemet om at logge ind.

Hvis brugernavnet ikke findes, præsentere systemet en fejlmeldelse, for sælgeren.

Hvis adgangskoden ikke stemmer overens med brugernavn, præsentere systemet en fejlmeldelse, for sælgeren.

Use Case 1

Casual Use Case

Ffs - 01: Anmod om Låneanmodning

- *Sælgeren er logget ind i systemet og klar til at betjene en ny kunde.
Der er en kunde til stede, der ønsker at optage et lån.*

Sælgeren anmoder systemet om at oprette et nyt lånetilbud

Systemet opretter en lånetilbud

Sælger angiver cpr-nummer

Sælger validere information med kunden

Sælger angiver ønskede biltype

Sælger angiver kundens ønskede udbetaling

Sælger angiver ønskede afdragsperiode

Hvis kunden ikke eksisterer i databasen oprettes kunden med navn, adresse, cpr-nummer, telefonnummer og e-mail.

Hvis der er kommentarer om tidligere problemer med kunden, afsluttes forløbet.

Hvis kunden får en speciel rabat, angiver sælgeren den nye salgspris.

Hvis kundens ønskede lånesum overstiger det sælgeren er autoriseret til at godkende, får sælgeren en notifikation om, at han ikke selv er bemyndiget til at udstede lånet.

Fully Dressed Use Case

Ffs - 01: Anmod om Lånetilbud

Scope: FerrariFinansSystem

Level: User Goal

Primary Actor: Sælger

Stackholders and interests

- Firmaet:
 - Låneanmodning bliver korrekt registreret med det samme.
 - At data ikke går tabt i processen.
 - Låneanmodning fra samme kunde ikke bliver behandlet flere gange.
 - Låneanmodning kan godkendes hurtigst muligt.
 - Kontakt med bank/RKI forløber hurtigt og uden tab af data.
 - Renten er korrekt beregnet.
 - Processen bliver gennemført korrekt og til ende.
- Kunden:
 - Få svar på låneanmodning hurtigst muligt.
 - Renten er korrekt beregnet.
 - Oplysningerne bliver opbevaret sikkert.
- Lovgivere:
 - Data opbevares i overensstemmelse med persondataloven.

Pre-conditions: Sælgeren er identificeret i systemet og klar til at betjene en ny kunde. Der er en kunde til stede, der ønsker at optage et lån.

Success Guarantee (post-condition): Der er genereret et korrekt lånetilbud til kunden.

Main Success Scenario:

1. Sælgeren anmelder systemet om at oprette en ny låneanmodning
2. Sælger angiver cpr-nummer
3. Systemet angiver kreditværdighed
4. Sælger validere information med kunden
5. Sælger angiver ønskede bilmodel
6. Systemet præsentere biler af den ønskede bilmodel
7. Sælger angiver ønskede bil
8. Systemet præsentere pris og information på den valgte bil
9. Sælger angiver kundens ønskede udbetaling
10. Systemet præsentere lånebeløbet
11. Sælger angiver ønskede afdragsperiode
12. Sælger beder systemet gemme låneanmodningen
13. Systemet angiver at låneanmodningen er gemt

Extension:

2a. Hvis der er kommentarer om tidligere problemer med kunden

1. Afslutter hovedscenariet

2b. Hvis kunden ikke eksisterer i databasen

1. Sælger angiver navn, adresse, postnummer, telefonnummer, e-mail
2. Fortsæt til hovedscenariet pkt. 3.

3a. Hvis kreditværdigheden er D

1. Afslutte hovedscenariet

4a. Hvis kundens informationer har ændret sig

2. Sælgeren angiver korrekt information.
3. Forsætter til hovedscenariet pkt. 5

8a. Hvis sælgeren ønsker at give rabat

1. Sælger angiver rabatbeløbet
2. Systemet præsentere ny pris
3. Forsætter til hovedscenariet pkt. 9

9a. Udbetaling er under 20% af købsprisen

1. Systemet angiver at udbetalings beløbet er for lille.
2. Sælger informere kunden

9aa. Hvis Kunden ønsker ny udbetalingsbeløb

- Forsætter til hovedscenariet pkt. 9

9ab. Hvis Kunden ikke ønsker højere udbetalingsbeløb

- Afslutte Hovedscenariet.

10a. Lånebeløbet oversiger sælgers beløbsgrænse.

1. Systemet angiver at lånebeløbet oversiger sælgers beløbsgrænse.
2. Forsætter til hovedscenariet pkt. 11.

12a. Hvis systemet ikke svare tilbage indenfor 30 sek.

1. Systemet angiver en fejl-meddeles.
2. Fortsætter til hovedscenariet pkt. 12.

6a. Hvis kundens ønskede lånesum overstiger det sælgeren er autoriseret til at godkende

1. Sælgeren får en notifikation om, at han ikke selv er bemyndiget til at udstede lånet.
2. Fortsæt til hovedscenariet pkt. 7. m. gennemførsel af pkt. 11 i samarbejde med sælger af højere autoritet

Special Requirements:

- Oprettelse af én låneanmodning må højst tage 45 sekunder, forudsat at systemet er operationelt og forespørgsel til Bank og RKI forløber problemfrit.
80 % af de ansatte i firmaet, skal kunne løse deres opgaver vha. det nye system, uden hjælp, indenfor fem minutter, den første gang de prøver det.

Technology and Data Variations List

- Ingen

Frequency of Occurrence

- Use Casen udføres 2 gange om dagen

Use Case 2

Casual Use Case

FFS-02: Vis låneanmodning

- FFS-01 er gennemført korrekt og låneanmodninger ligger klar til visning.

Sælgeren anmoder systemet om, at få vist alle eksisterende låneanmodninger.
Systemet viser eksisterende låneanmodninger.

Hvis låneanmodningen ikke eksisterer i systemet gennemføres FFS-01.

Fully Dressed Use Case

FFS-02: Vis låneanmodning

Scope: FerrariFinanceSystem

Level: User Goal

Primary Actor: Sælger/Salgschef

Stackholders and interests

- **Firmaet:**
 - Låneanmodning er gemt korrekt i systemet, så evt. flere forespørgsler, fra samme kunde til forskellige sælgere, ikke bliver behandlet flere gange.
 - Låneanmodning bliver vist indeholdende korrekt data.
- **Kunden:**
 - At låneanmodningen er gemt korrekt, så det ikke skal oprettes igen.
 - Oplysningerne bliver opbevaret sikkert.
- **Lovgivere:**
 - Data opbevares i overensstemmelse med persondataloven.

Pre-conditions: Låneanmodninger er gemt korrekt i systemet og ligger klar til visning.

Success Guarantee (post-condition): Sælgeren får vist de låneanmodninger, han har oprettet. Hvis Salgschefen er logget på, får han vist samtlige låneanmodninger i systemet.

Main Success Scenario:

1. Sælgeren anmoder systemet om, at få vist sine eksisterende låneanmodninger.
2. Systemet viser låneanmodninger.

Extensions:

- 2a. Hvis systemet viser forkerte låneanmodninger:
 1. Sælger validerer, at han er logget korrekt ind.
 2. Fortsæt hovedscenariet fra pkt. 1.

Special Requirements:

- Visning af liste over låneanmodninger må højest tage 15 sekunder, forudsat at systemet er operationelt.

Technology and Data Variations List

- Ingen

Frequency of Occurrence

- Use Casen udføres to gange om dagen.

Use Case 3

Casual Use Case

FFS-03: Godkend låneanmodninger

- FFS-01 er gennemført korrekt, og låneanmodning er gemt korrekt.

Sælgeren anmoder systemet om, at acceptere låneanmodning.

Systemet viser aktuelle låneanmodninger.

Sælger vælger låneanmodning og accepterer.

Systemet gemmer nyt lånetilbud og afdragsordning.

Systemet præsenterer endeligt lånetilbud og afdragsordning.

Hvis låneanmodningen ikke eksisterer i systemet gennemføres FFS-01.

Fully Dressed Use Case

FFS-03: Godkend låneanmodninger

Scope: FerrariFinanceSystem

Level: User Goal

Primary Actor: Sælger eller Salgschef

Stackholders and interests

- Firmaet:
 - Lånetilbud inkl. afdragsordning er gemt korrekt i systemet, så evt. flere forespørgsler, fra samme kunde til forskellige sælgere, ikke bliver behandlet flere gange.
 - Lånetilbud inkl. afdragsordning bliver eksporteret hurtigt indeholdende korrekt data.
- Kunden:
 - Få sit eget lånetilbud inkl. afdragsordning til godkendelse.
 - Renten og afdragene er korrekt beregnet.
 - Oplysningerne bliver opbevaret sikkert.
- Lovgivere:
 - Data opbevares i overensstemmelse med persondataloven.

Pre-conditions: FFS-02 er gennemført korrekt.

Success Guarantee (post-condition): Lånetilbud og afdragsordning er gemt korrekt i systemet. Og præsenteret for sælger.

Main Success Scenario:

1. Sælger vælger en låneanmodning.
2. Systemet præsentere informationer om låneanmodningen.
3. Sælger anmoder systemet om at godkende låneanmodning.
4. Systemet forespørger efter aktuel dagsrente.
5. Systemet beregner rentesats.
6. Systemet genererer færdigt lånetilbud med afdragsordning.
7. Systemet gemmer lånetilbud og afdragsordning.
8. Systemet præsenteret endeligt lånetilbud og afdragsordning.

Extensions:

4a. Hvis systemet ikke svare tilbage indenfor 30 sek.

1. Systemet præsentere en fejlmeldelse.
2. Fortsætter fra hovedscenariet pkt. 4 eller afslutter hovedscenariet.

Special Requirements:

- Oprettelse af ét lånetilbud inkl. afdragsordning må højest tage 2 minutter.

Technology and Data Variations List

- Ingen.

Frequency of Occurrence

- Use Casen udføres 2 gange om dagen.

Use Case 4

Casual Use Case

FFS-04: Vis lånetilbud

- FFS-03 er gennemført korrekt og lånetilbud ligger klar til visning.

Sælgeren anmoder systemet om, at få vist alle eksisterende lånetilbud.
Systemet viser eksisterende lånetilbud.

Fully Dressed Use Case

FFS-02: Vis lånetilbud

Scope: FerrariFinanceSystem

Level: User Goal

Primary Actor: Sælger/Salgschef

Stackholders and interests

- Firmaet:
 - Lånetilbud er gemt korrekt i systemet, så evt. flere forespørgsler, fra forskellige sælgere, ikke bliver behandlet flere gange.
 - Lånetilbud bliver vist indeholdende korrekt data.
- Kunden:
 - At lånetilbud er gemt korrekt, så det ikke skal oprettes igen.
 - Oplysningerne bliver opbevaret sikkert.
- Lovgivere:
 - Data opbevares i overensstemmelse med persondataloven.

Pre-conditions: Lånetilbud er gemt korrekt i systemet og ligger klar til visning.

Success Guarantee (post-condition): Sælgeren får vist de lånetilbud, han har oprettet. Hvis Salgschefen er logget på, får han vist samtlige lånetilbud i systemet.

Main Success Scenario:

1. Sælgeren anmoder systemet om, at få vist sine eksisterende lånetilbud.
2. Systemet viser lånetilbud.

Extensions:

- 2a. Hvis systemet viser forkerte lånetilbud:
 1. Sælger validerer, at han er logget korrekt ind.
 2. Fortsæt hovedscenariet fra pkt. 1.

Special Requirements:

- Visning af liste over lånetilbud må højst tage 15 sekunder.

Technology and Data Variations List

- Ingen

Frequency of Occurrence

- Use Casen udføres to gange om dagen.

Use Case 5

Casual Use Case

FFS-02: Eksporter lånetilbud & afdragsordning

- FFS-04 er gennemført korrekt og lånetilbud inkl. afdragsordning ligger klar til eksportering.

Sælgeren anmoder systemet om, at præsentere et eksisterende lånetilbud.

Systemet præsentere lånetilbudget.

Sælger anmoder systemet om at eksportere det præsenterede lånetilbud inkl. afdragsordning.

Systemet eksporterer lånetilbud inkl. afdragsordning.

Fully Dressed Use Case

FFS-02: Eksporter lånetilbud & afdragsordning

Scope: FerrariFinanceSystem

Level: User Goal

Primary Actor: Sælger

Stackholders and interests

- Firmaet:
 - Lånetilbud er gemt korrekt i systemet, så evt. flere forespørgsler, fra samme kunde til forskellige sælgere, ikke bliver behandlet flere gange.
 - Lånetilbud bliver vist indeholdende korrekt data.
- Kunden:
 - At lånetilbud er gemt korrekt, så det ikke skal oprettes igen.
 - Oplysningerne bliver opbevaret sikkert.
- Lovgivere:
 - Data opbevares i overensstemmelse med persondataloven.

Pre-conditions: Lånetilbud er gemt korrekt i systemet og ligger klar til at blive eksporteret.

Success Guarantee (post-condition): Sælgeren får vist de lånetilbud, han har oprettet. Hvis Salgschefen er logget på, får han vist samtlige lånetilbud i systemet.

Main Success Scenario:

1. Sælgeren anmoder systemet om, at præsentere et eksisterende lånetilbud.
2. Systemet præsentere lånetilbudget.

3. Sælger anmoder systemet om at eksportere det præsenterede lånetilbud inkl. afdragsordning.
4. Systemet eksporterer lånetilbud inkl. afdragsordning.

Extensions:

- 2a. Hvis systemet viser forkerte lånetilbud:
 1. Sælger validerer, at han er logget korrekt ind.
 2. Fortsæt hovedscenariet fra pkt. 1.

Special Requirements:

- Eksportering af lånetilbud må højest tage 30 sekunder.

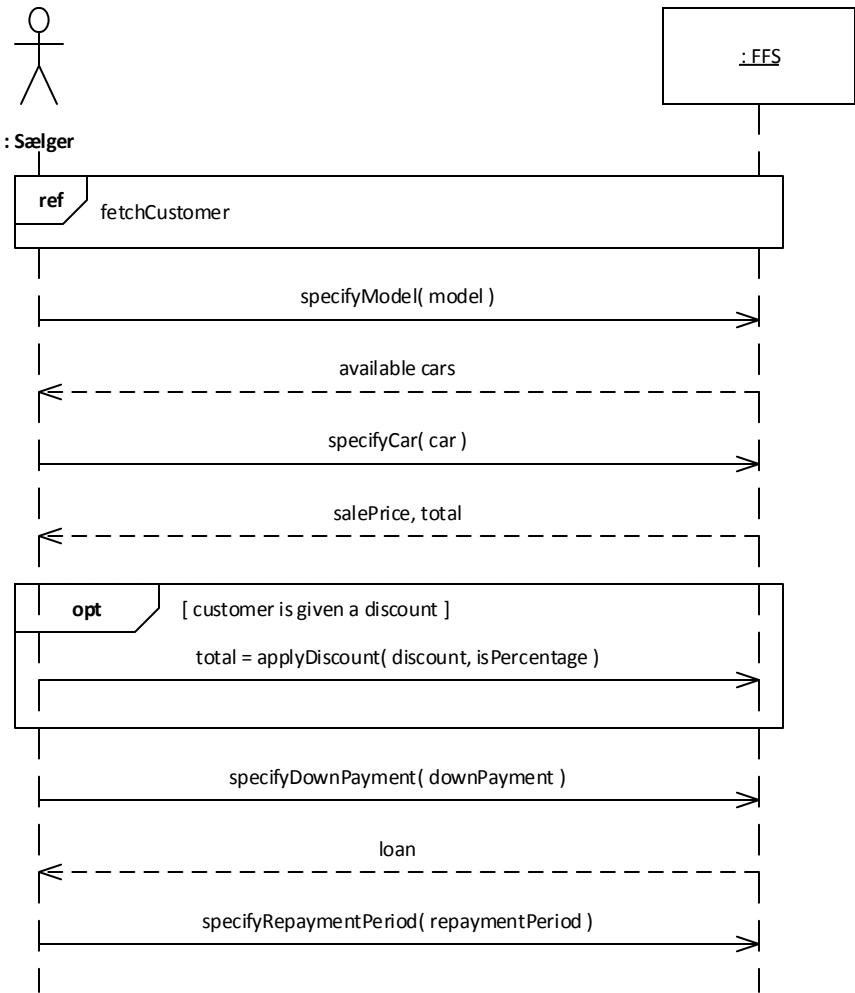
Technology and Data Variations List

- Ingen

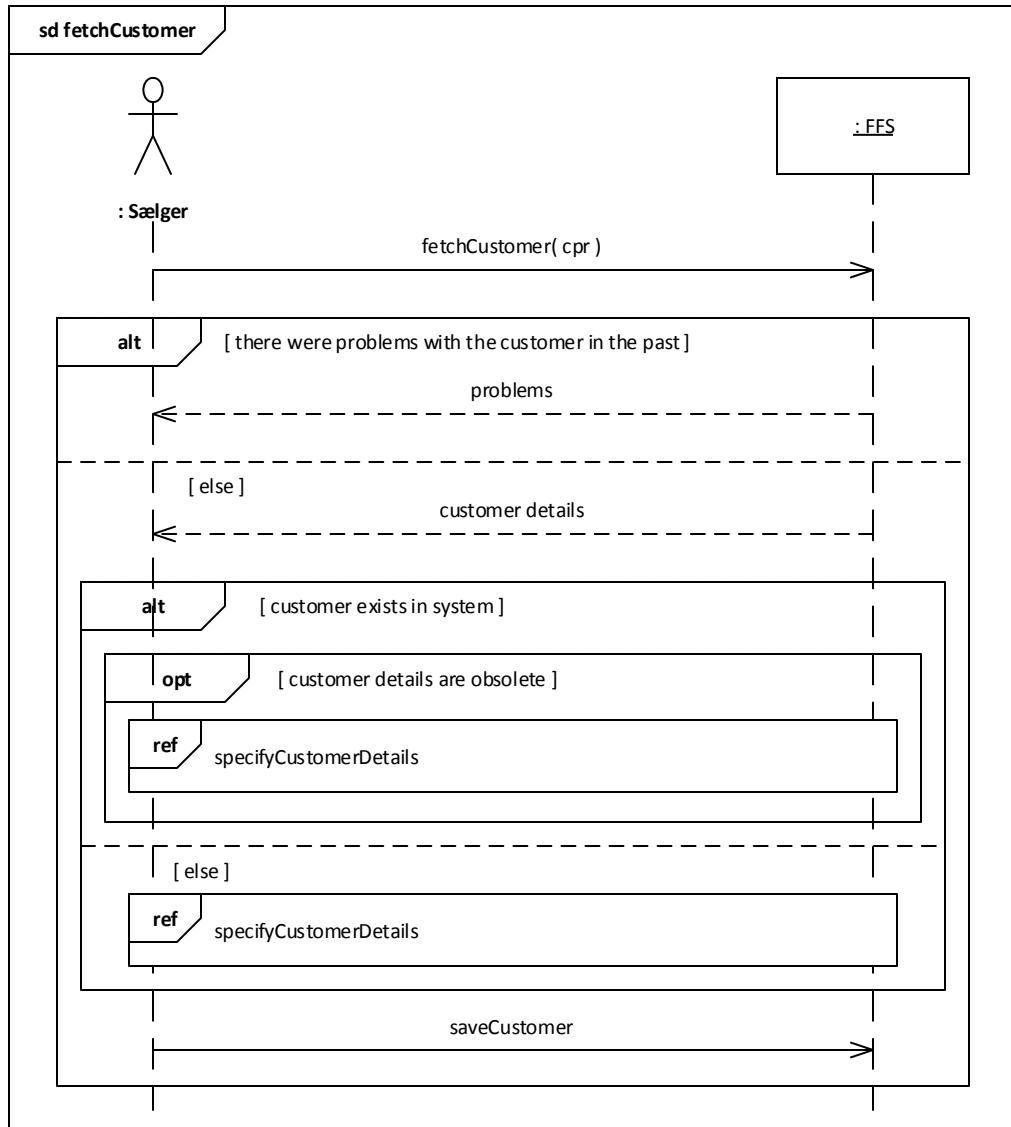
Frequency of Occurrence

- Use Casen udføres to gange om dagen.

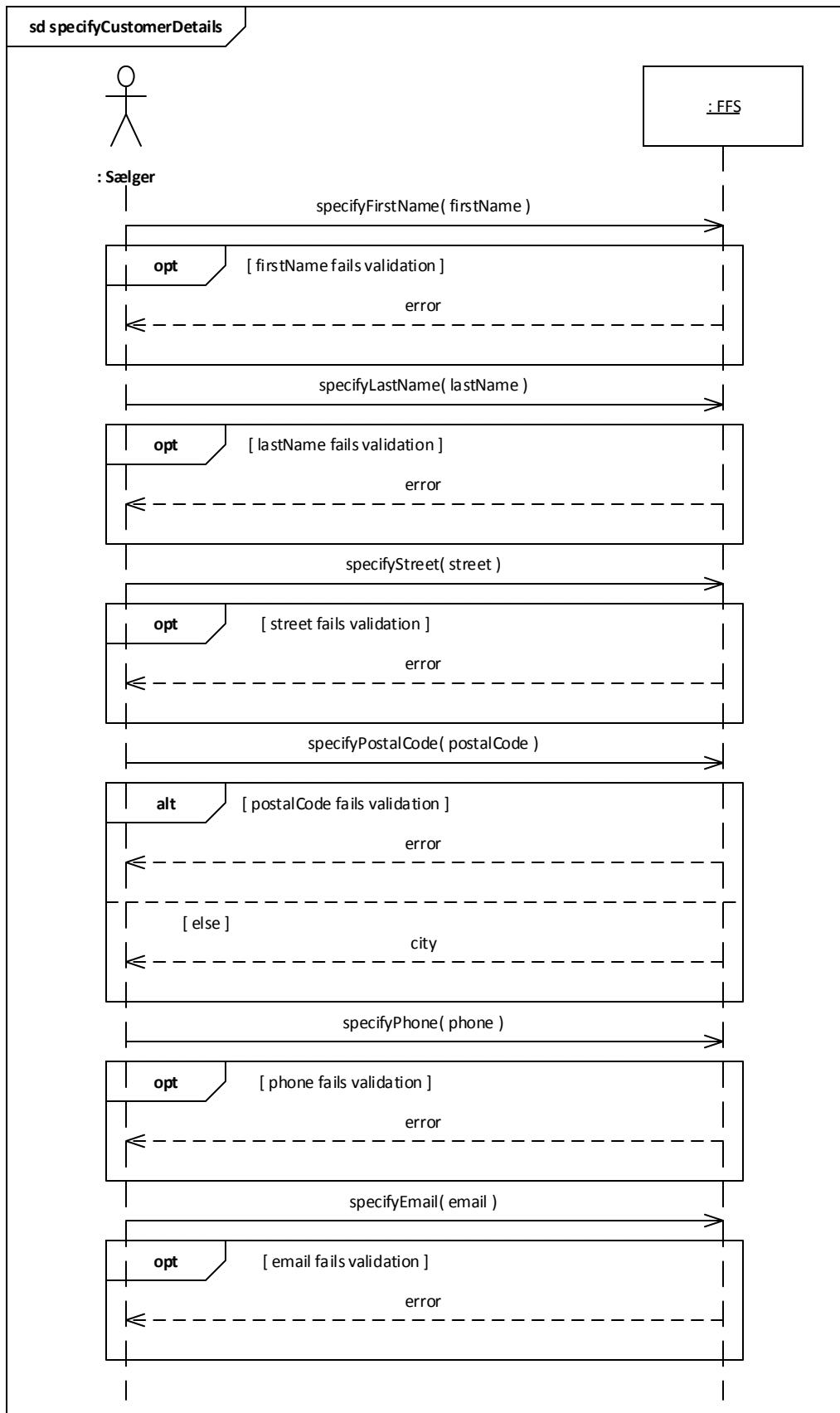
SSD-FFS-01: Anmod om lån



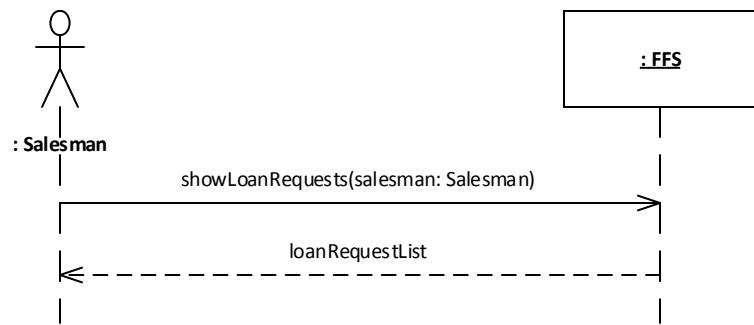
SSD-FFS-01: Anmod om lån (fetchCustomer)



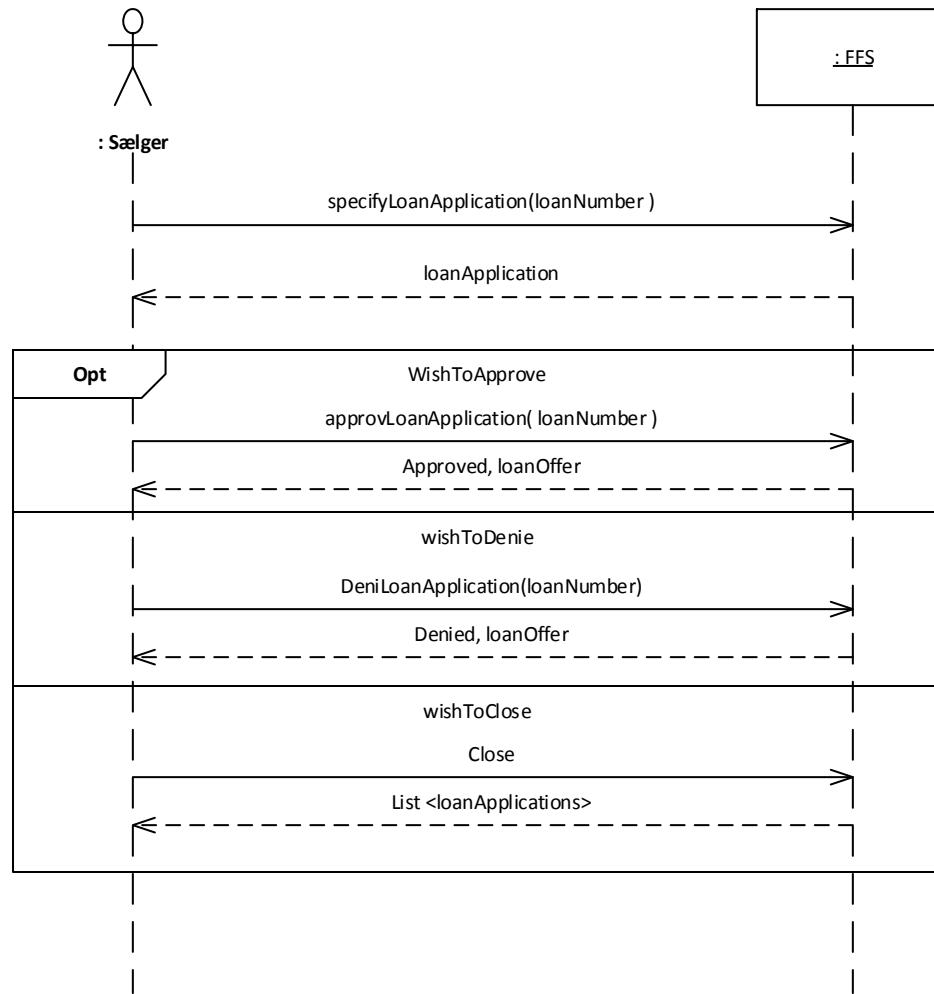
SSD-FFS-01: Anmod om lån (specifyCustomerDetails)



Systemsekvensdiagram FFS-02: Vis låneanmodning



SSD-FFS-02: Godkend låneanmodning



FFS-OC-01: fetchCustomer

Systemoperation

fetchCustomer(cpr)

Krydsreferencer

FFS-01: Anmod om lån

Forudsætninger

Ingen.

Slutbetingelser

Hvis cpr matchede en kundes CPR-nr., og der er registreret problemer med kunden, blev problemerne præsenteret; ellers:

En instans c af Customer blev skabt.

Hvis cpr matchede en kundes CPR-nr., blev

- c.firstName sat til kundens fornavn
- c.lastName sat til kundens efternavn
- c.street sat til kundens adresse
- c.postalCode sat til kundens postnummer
- c.phone sat til kundens telefonnummer
- c.email sat til kundens email-adresse

De nævnte kundedata blev præsenteret.

FFS-OC-02: specifyFirstName

Systemoperation

`specifyFirstName(firstName)`

Krydsreferencer

FFS-01: Anmod om lån

Forudsætninger

En instans c af Customer eksisterer.

Slutbetingelser

Hvis værdien af firstName er gyldig, blev c.firstName sat til firstName; ellers blev en fejlbesked præsenteret.

FFS-OC-03: specifyLastName

Systemoperation

specifyLastName(lastName)

Krydsreferencer

FFS-01: Anmod om lån

Forudsætninger

En instans c af Customer eksisterer.

Slutbetingelser

Hvis værdien af lastName er gyldig, blev c.lastName sat til lastName; ellers blev en fejlbesked præsenteret.

FFS-OC-04: specifyStreet

Systemoperation

`specifyStreet(street)`

Krydsreferencer

FFS-01: Anmod om lån

Forudsætninger

En instans c af Customer eksisterer.

Slutbetingelser

Hvis værdien af street er gyldig, blev c.street sat til street; ellers blev en fejlbesked præsenteret.

FFS-OC-05: specifyPostalCode

Systemoperation

`specifyPostalCode(postalCode)`

Krydsreferencer

FFS-01: Anmod om lån

Forudsætninger

En instans c af Customer eksisterer.

Slutbetingelser

Hvis værdien af postalCode er gyldig, blev

- c.postalCode sat til postalCode
 - den tilsvarende by præsenteret
- ellers blev en fejlbesked præsenteret.

FFS-OC-06: specifyPhone

Systemoperation

`specifyPhone(phone)`

Krydsreferencer

FFS-01: Anmod om lån

Forudsætninger

En instans c af Customer eksisterer.

Slutbetingelser

Hvis værdien af phone er gyldig, blev c.phone sat til phone; ellers blev en fejlbesked præsenteret.

FFS-OC-07: specifyEmail

Systemoperation

specifyEmail(email)

Krydsreferencer

FFS-01: Anmod om lån

Forudsætninger

En instans c af Customer eksisterer.

Slutbetingelser

Hvis værdien af email er gyldig, blev c.email sat til email; ellers blev en fejlbesked præsenteret.

FFS-OC-08: saveCustomer

Systemoperation

saveCustomer

Krydsreferencer

FFS-01: Anmod om lån

Forudsætninger

En instans c af Customer eksisterer.

Slutbetingelser

Ingen.

FFS-OC-09: specifyModel

Systemoperation

`specifyModel(model)`

Krydsreferencer

FFS-01: Anmod om lån

Forudsætninger

CarModel model eksisterer.

Slutbetingelser

Alle instanser c af Car hvor c.model = model blev præsenteret.

FFS-OC-10: specifyCar

Systemoperation

`specifyCar(car)`

Krydsreferencer

FFS-01: Anmod om lån

Forudsætninger

En instans lr af LoanRequest eksisterer.

Car car eksisterer.

Slutbetingelser

lr blev associeret med car.

car.salePrice blev præsenteret.

lr.total blev præsenteret.

FFS-OC-11: applyDiscount

Systemoperation

applyDiscount(discount, isPercentage)

Krydsreferencer

FFS-01: Anmod om lån

Forudsætninger

En instans lr af LoanRequest eksisterer.

Slutbetingelser

En instans d af Discount blev skabt.

d.discount blev sat til discount.

d.isPercentage blev sat til isPercentage.

lr blev associeret med d.

lr.total blev præsenteret.

FFS-OC-12: specifyDownPayment

Systemoperation

`specifyDownPayment(downPayment)`

Krydsreferencer

FFS-01: Anmod om lån

Forudsætninger

En instans Ir af LoanRequest eksisterer.

Slutbetingelser

Ir.downPayment blev sat til downPayment.

Ir.loan blev præsenteret.

FFS-OC-13: specifyRepaymentPeriod

Systemoperation

specifyRepaymentPeriod(repaymentPeriod)

Krydsreferencer

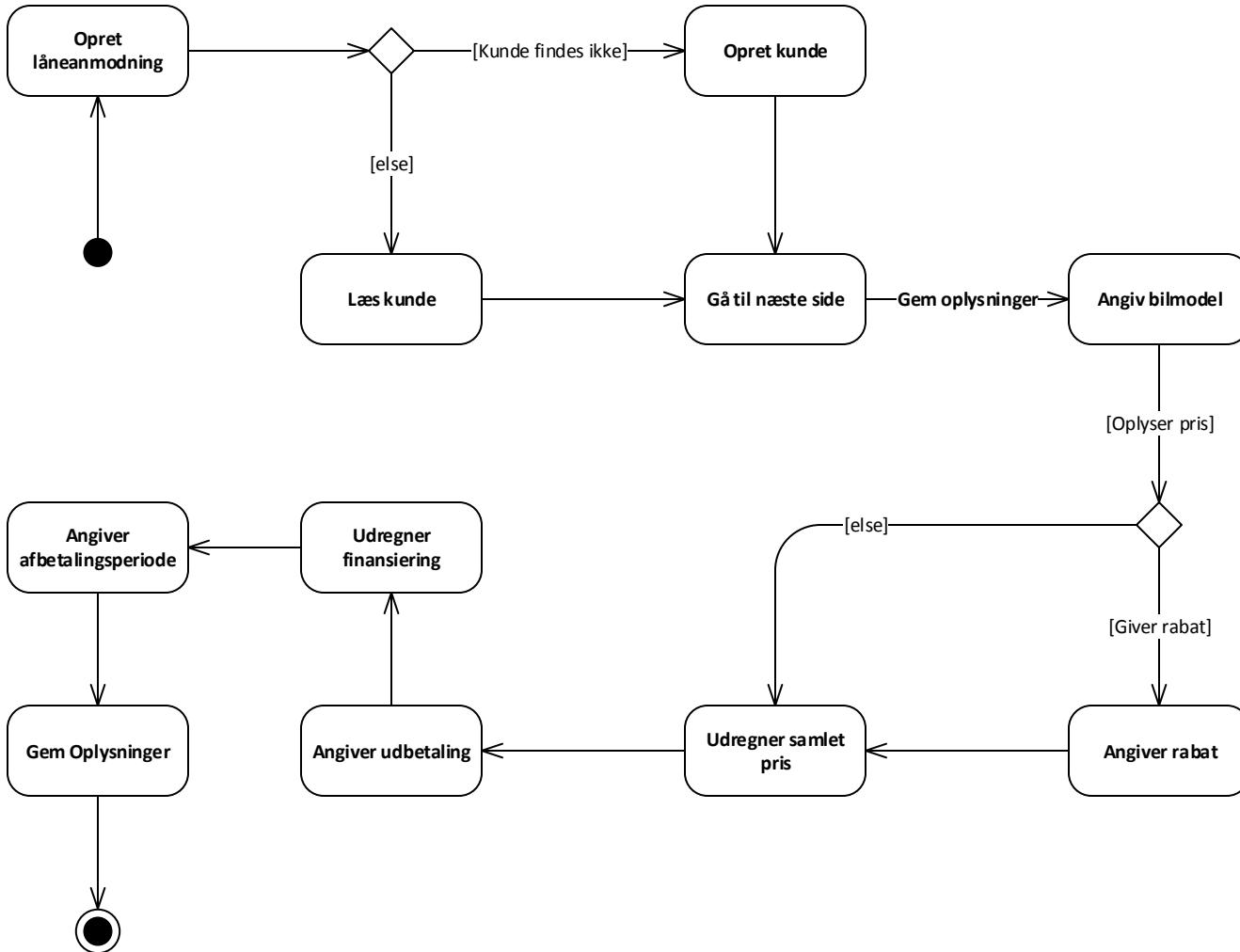
FFS-01: Anmod om lån

Forudsætninger

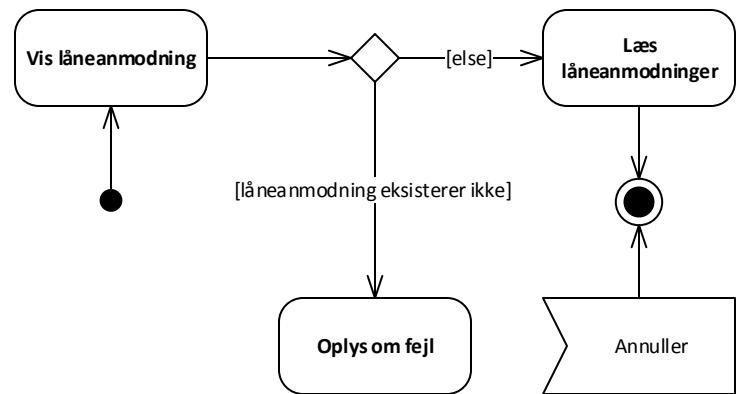
En instans lr af LoanRequest eksisterer.

Slutbetingelser

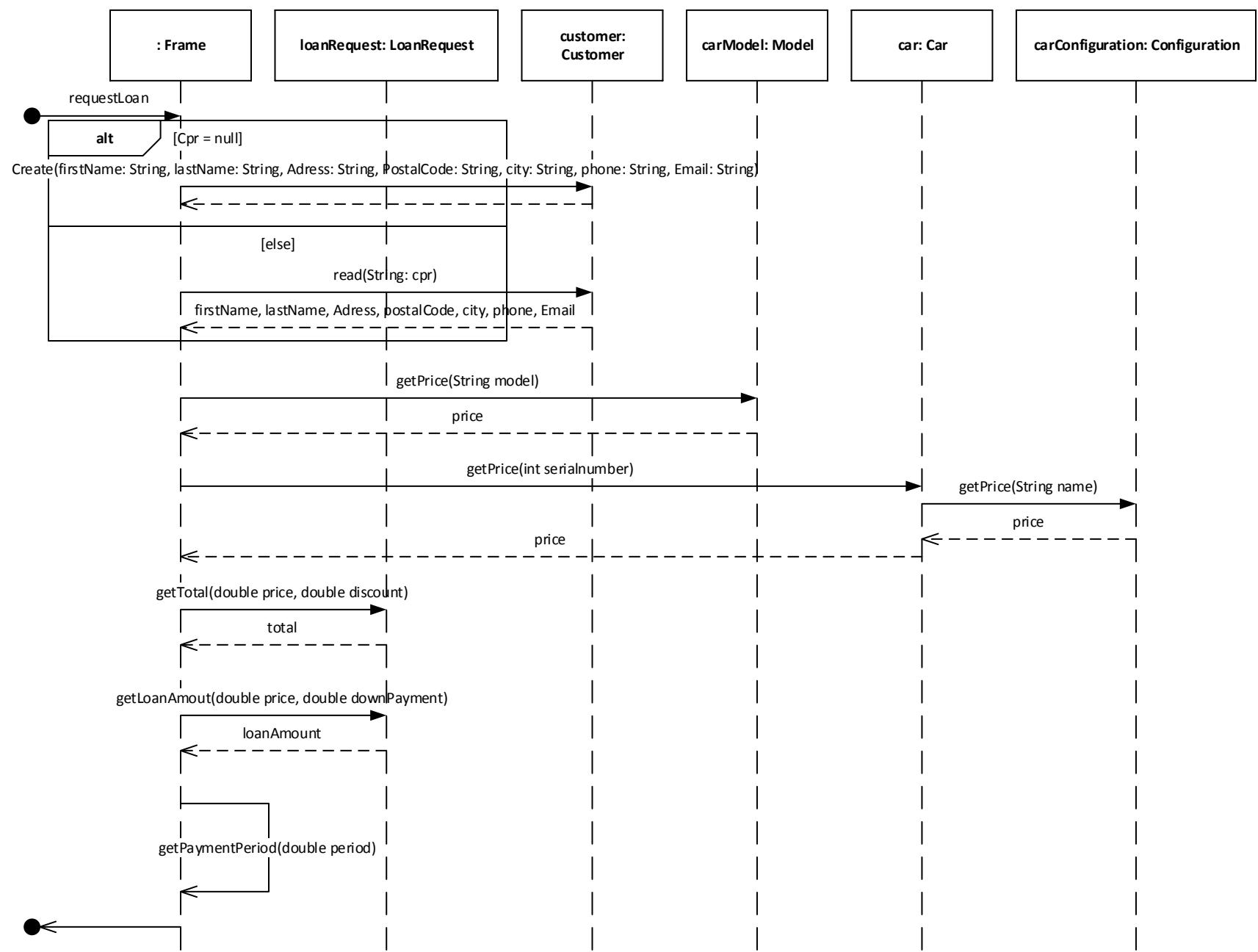
lr.repaymentPeriod blev sat til repaymentPeriod.

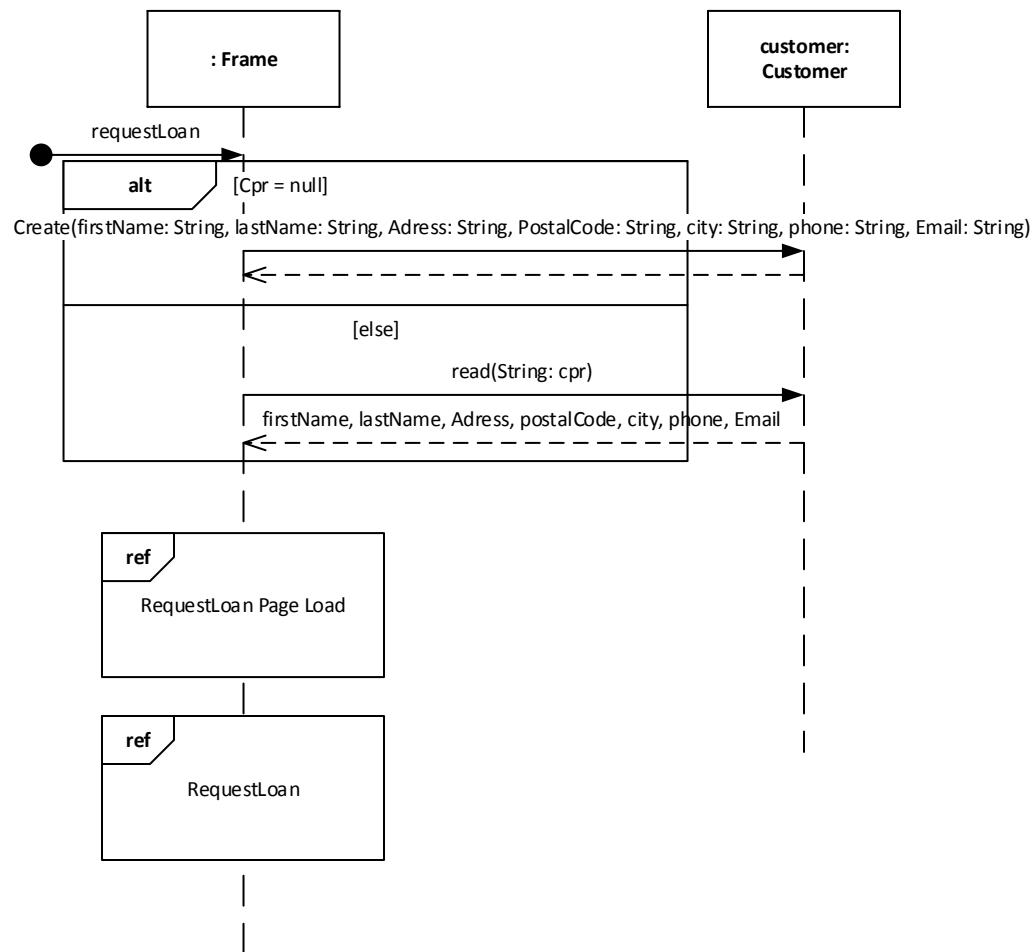


Aktivitetsdiagram FFS-02: Vis låneanmodning

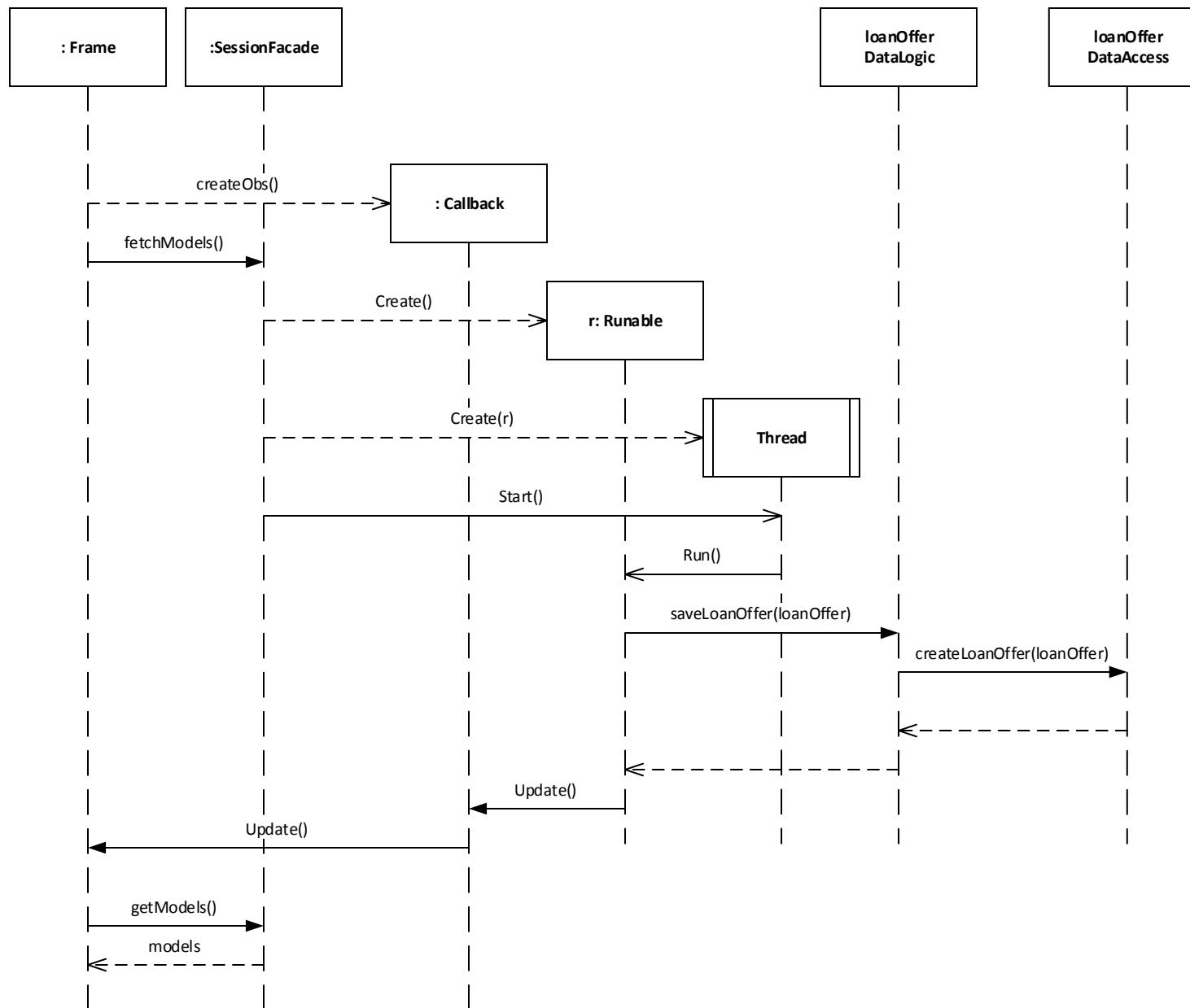


Opret låneanmodning

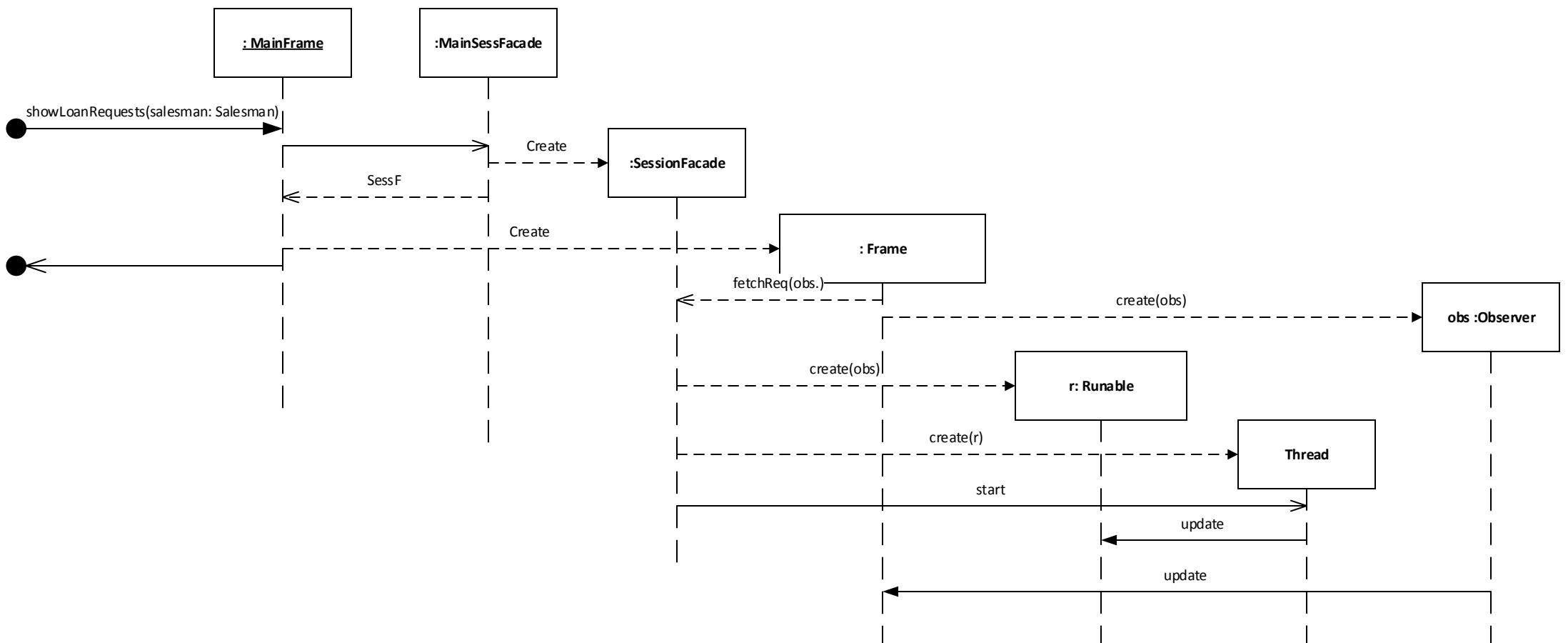




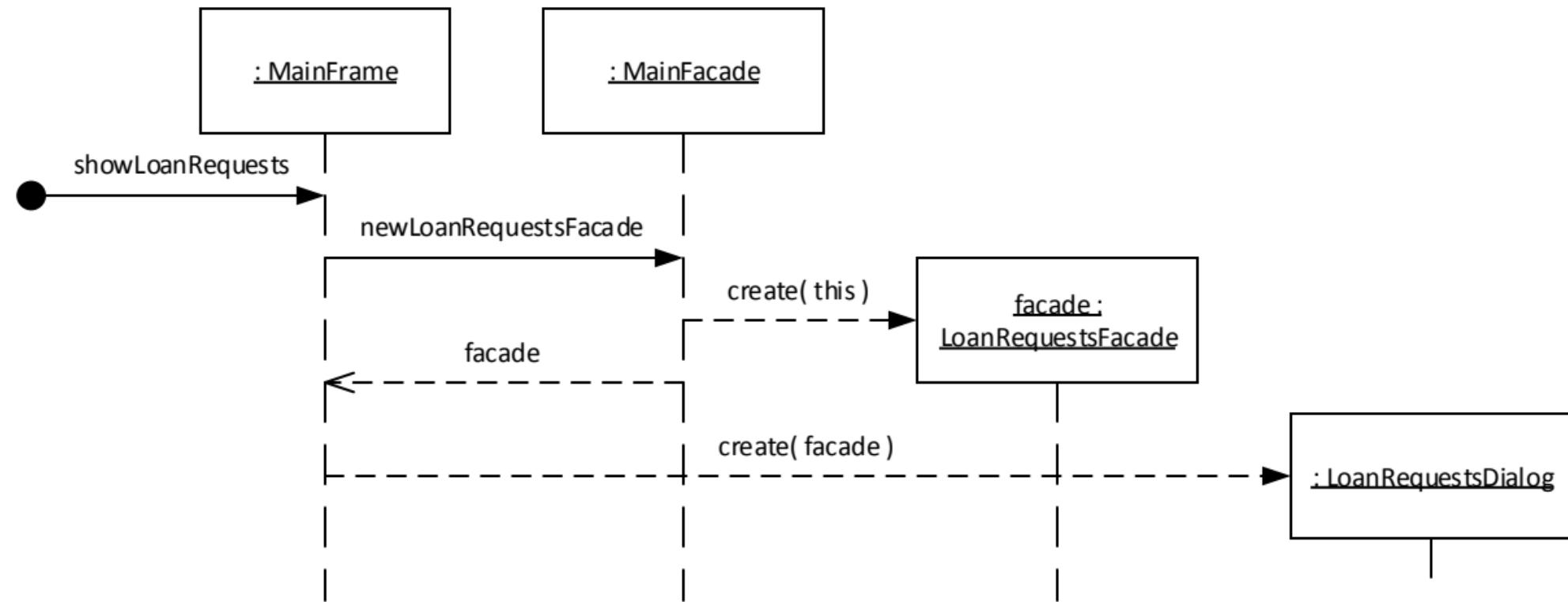
SD FFS 01



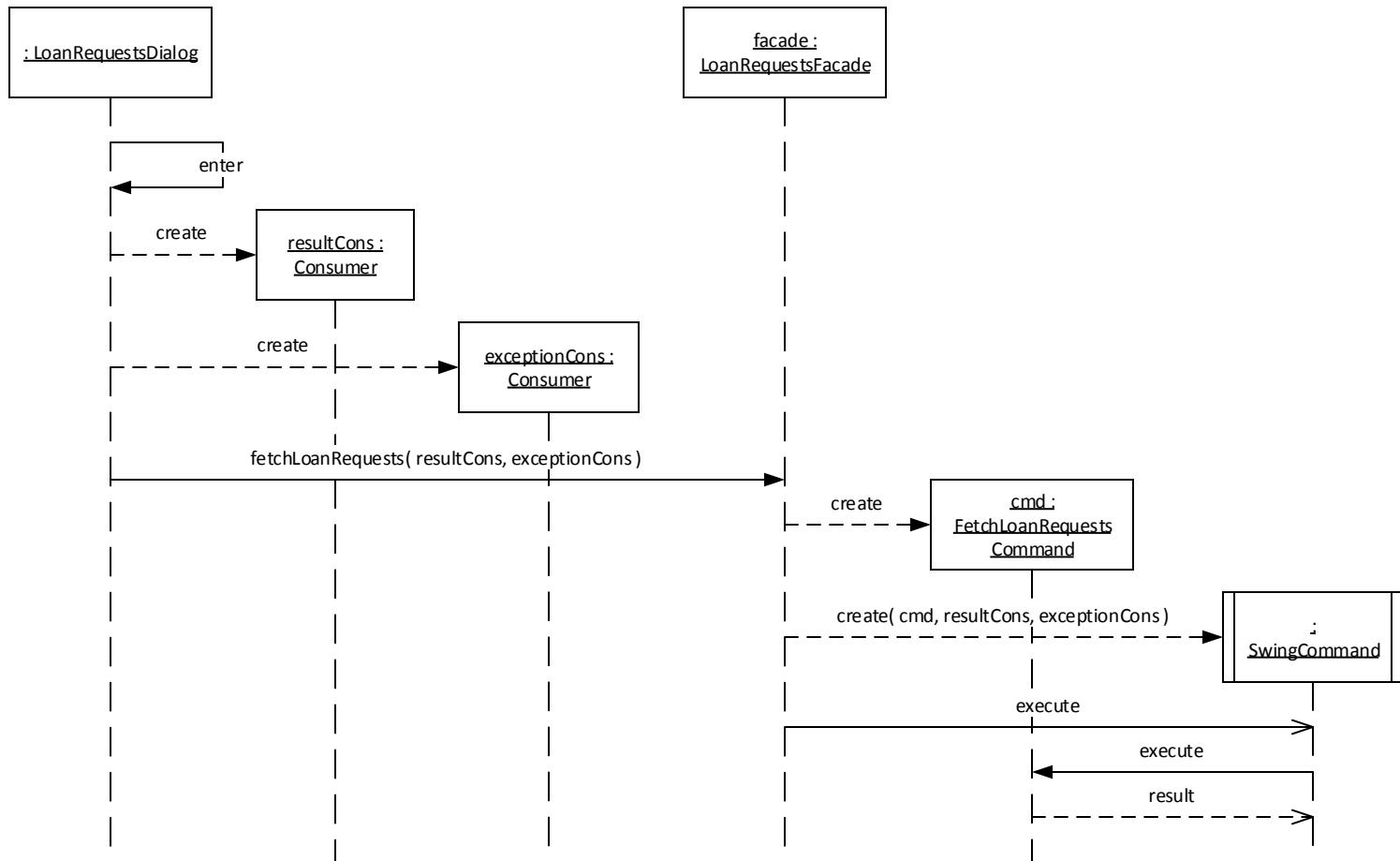
Sekvensdiagram FFS-02: Vis låneanmodninger
Early draft!



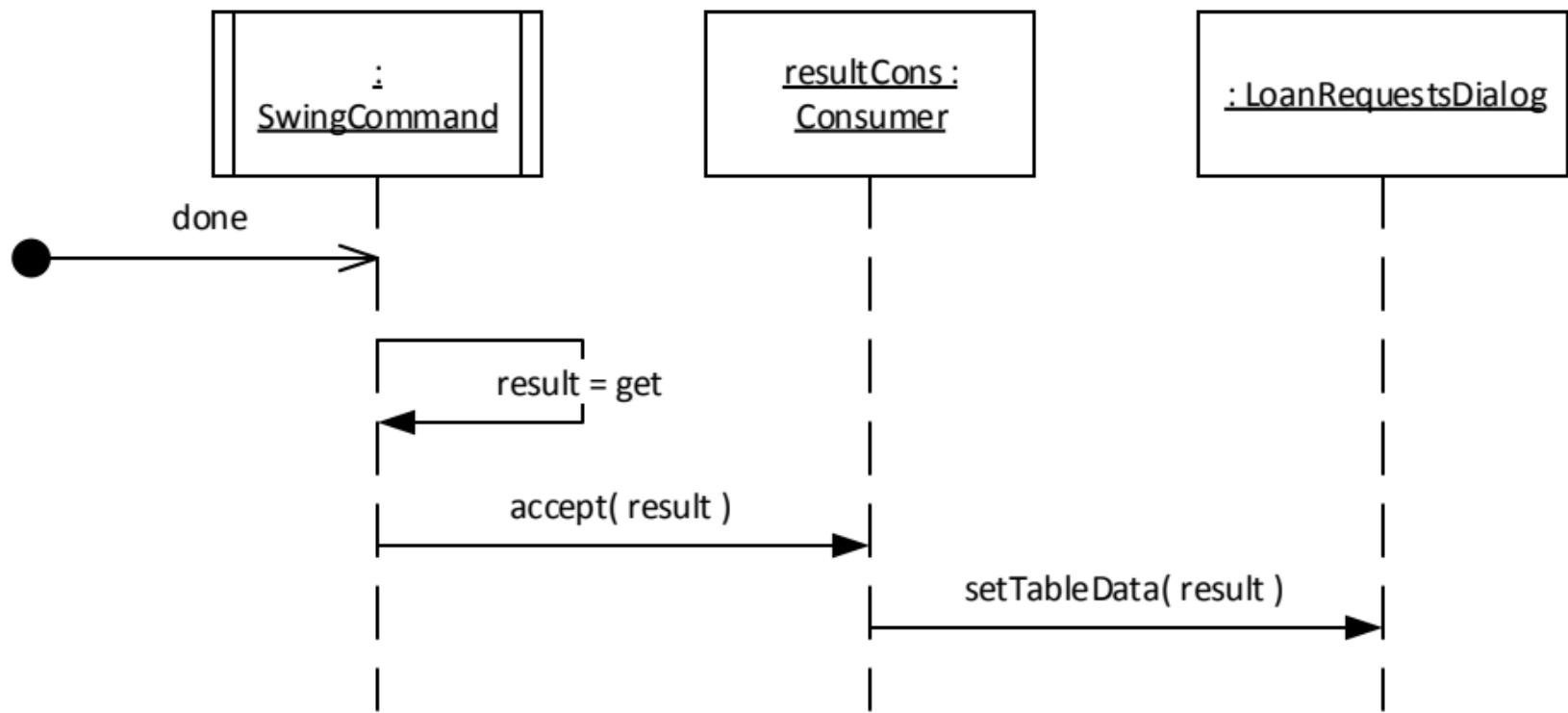
FFS-02 SD: showLoanRequests (create dialog)

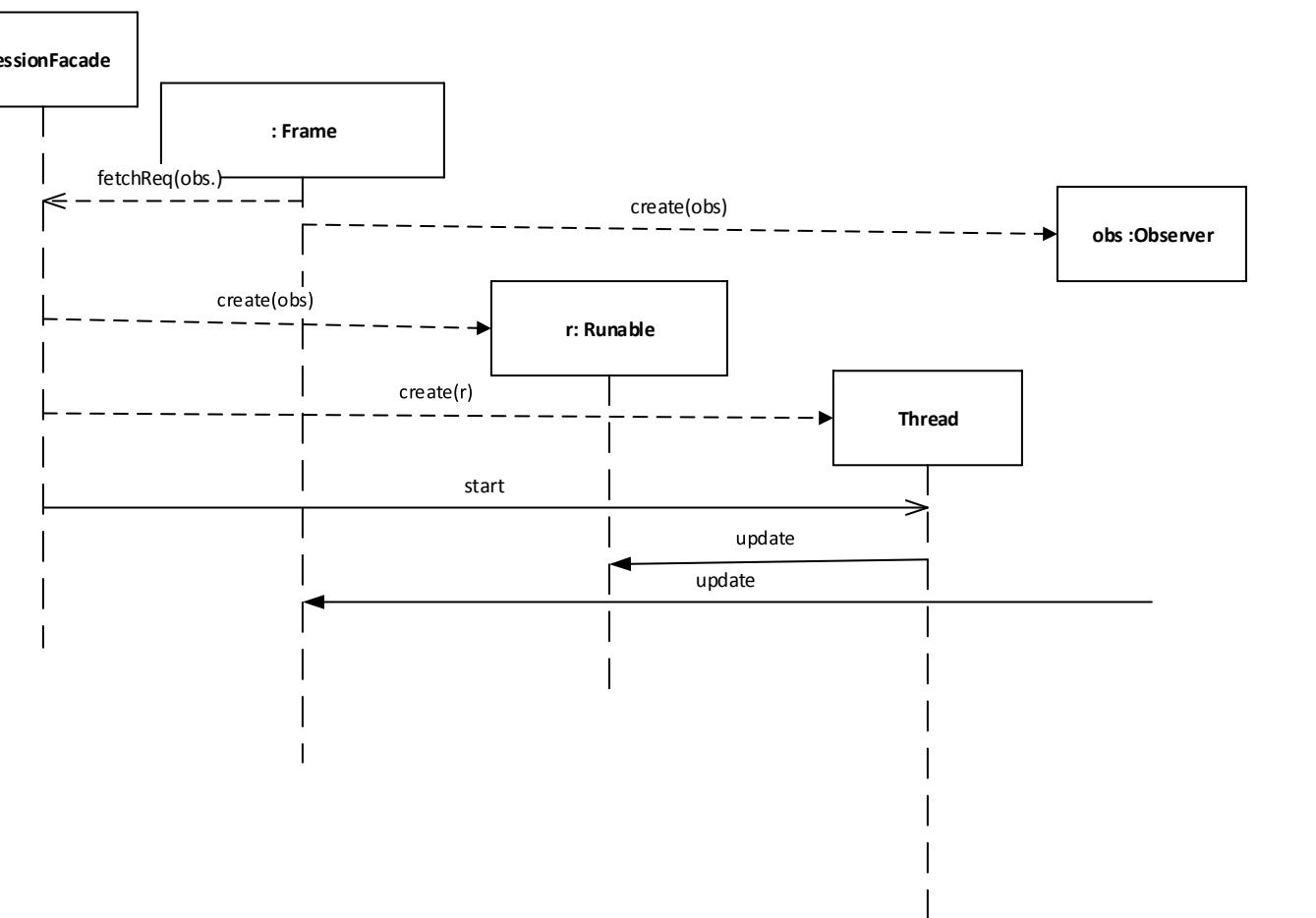
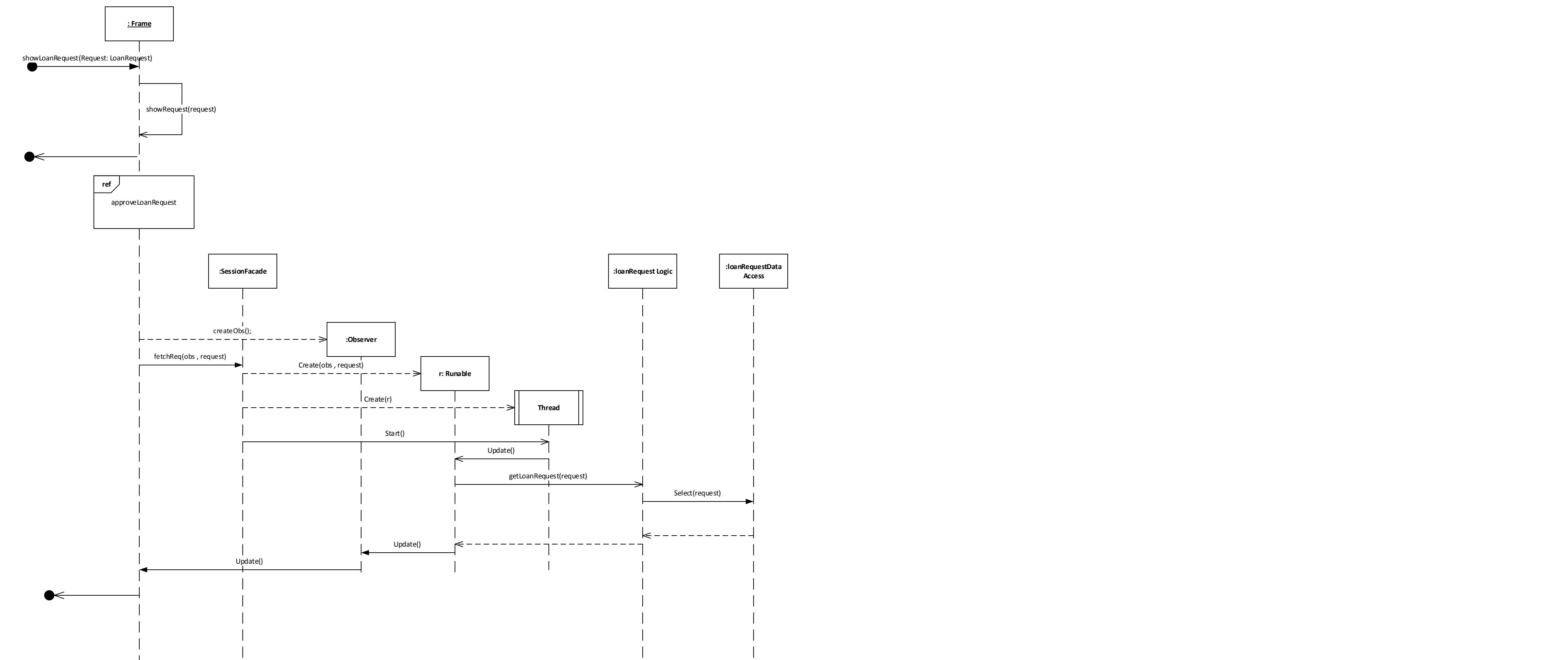


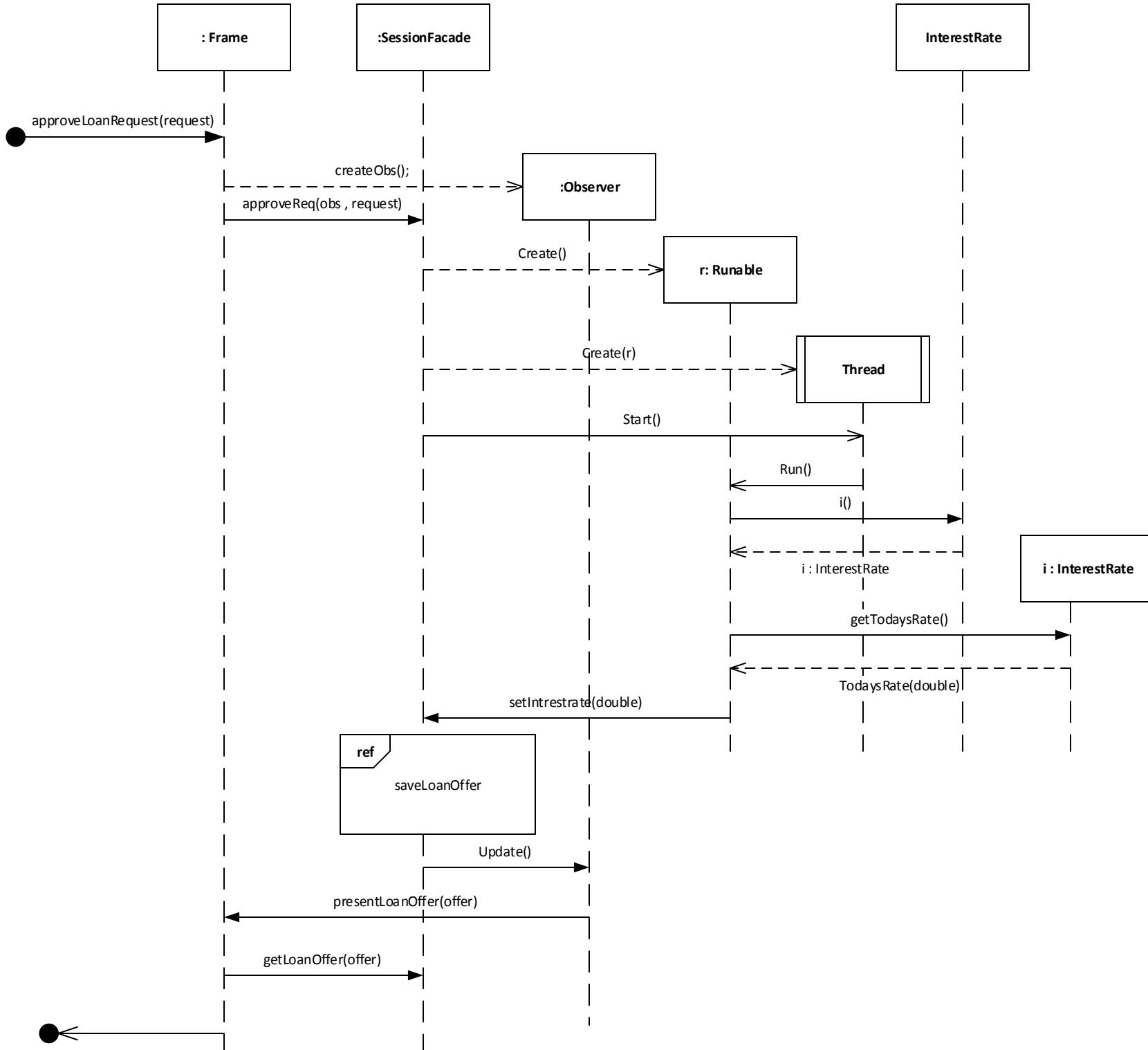
FFS-02 SD: showLoanRequests (execute command)



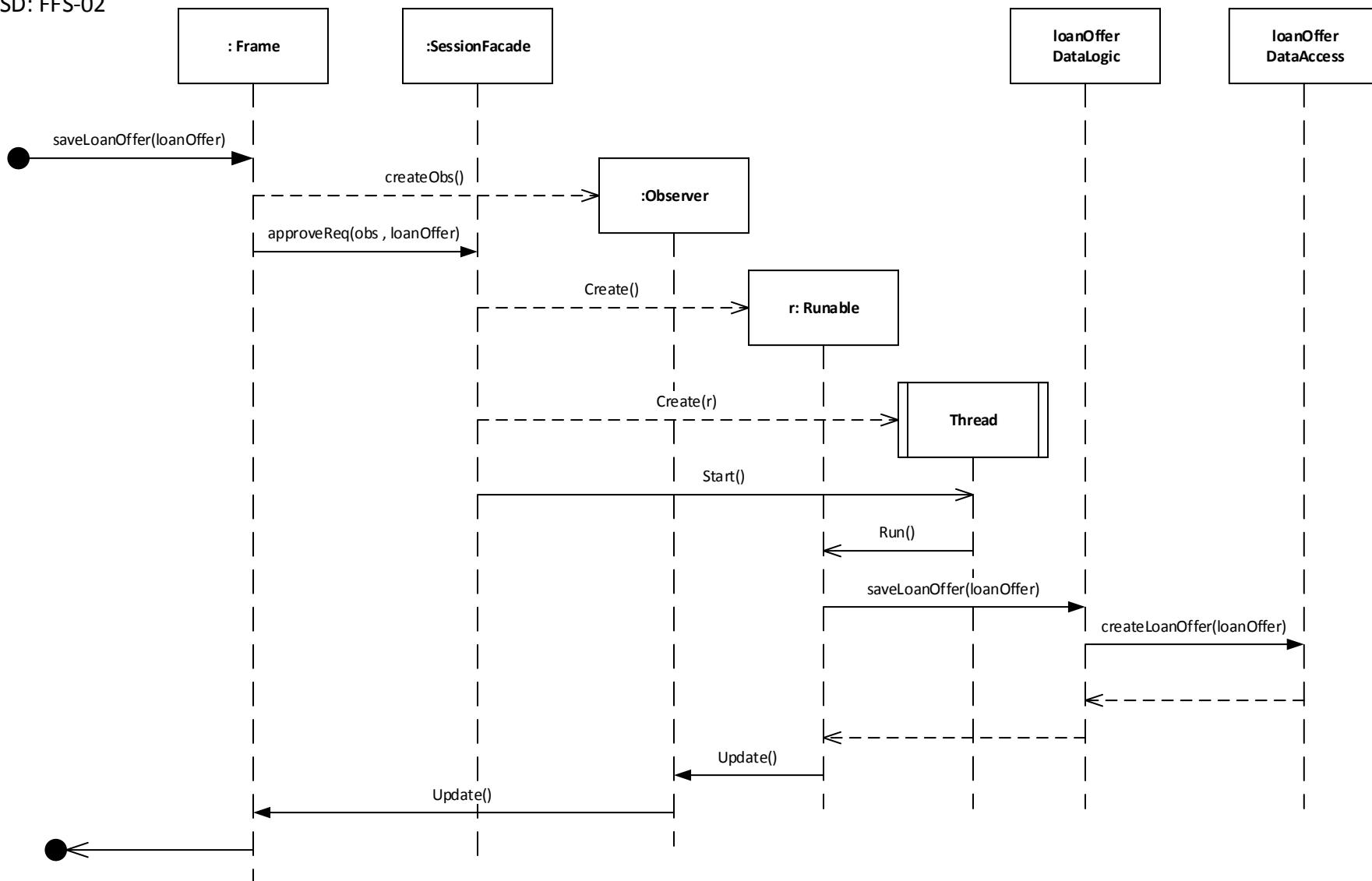
FFS-02 SD: showLoanRequests (display result)



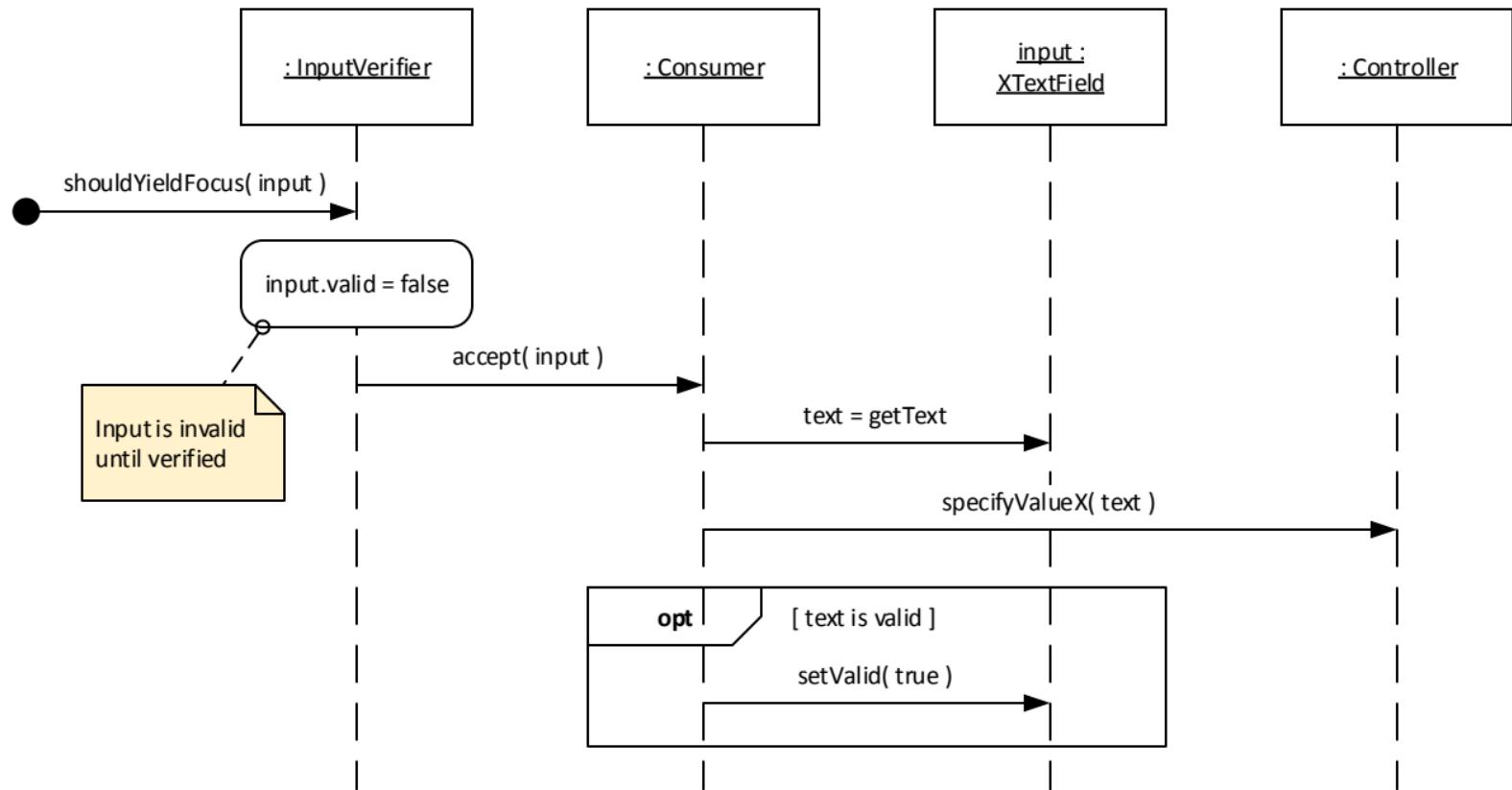


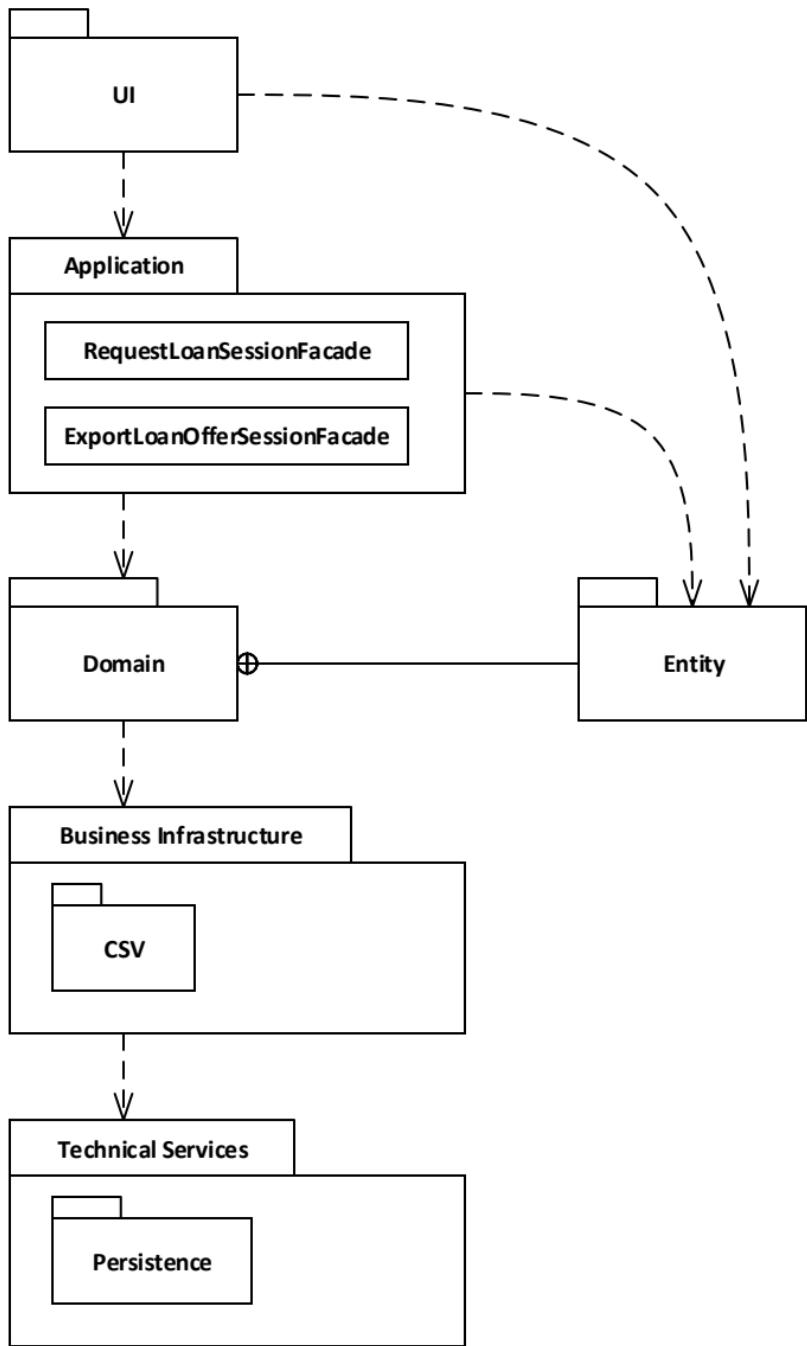


SD: FFS-02

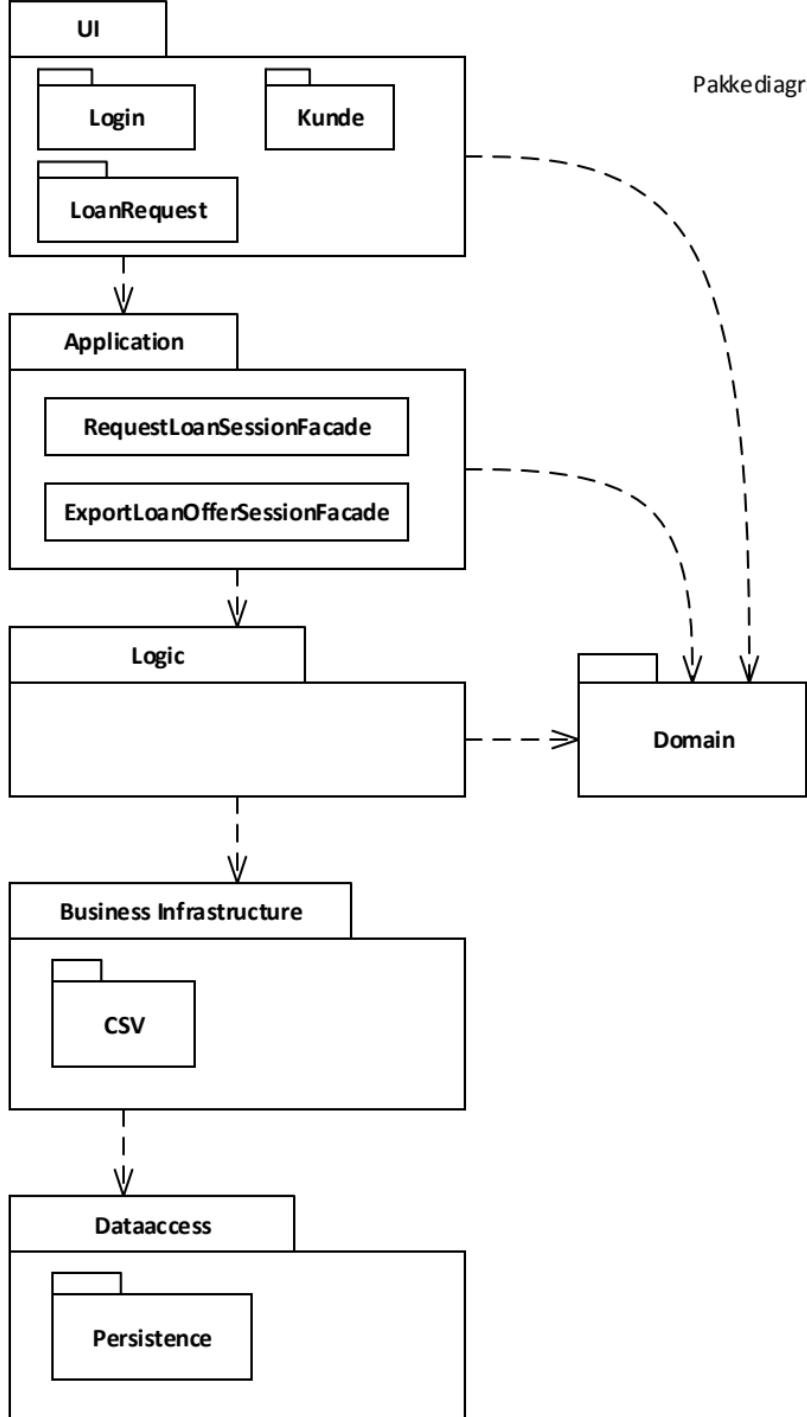


SD: Transfer of text input from UI to Controller using InputVerifier

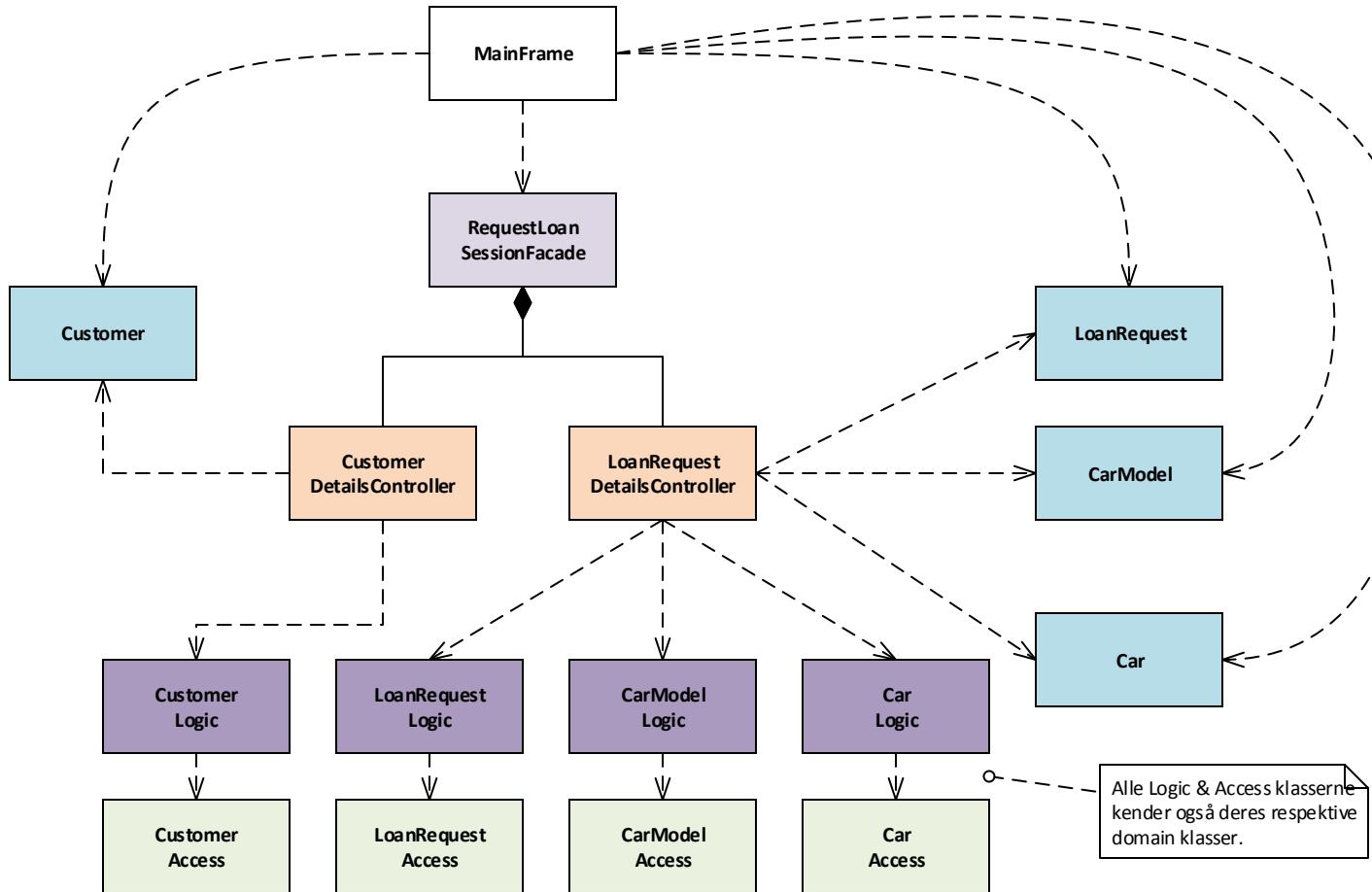




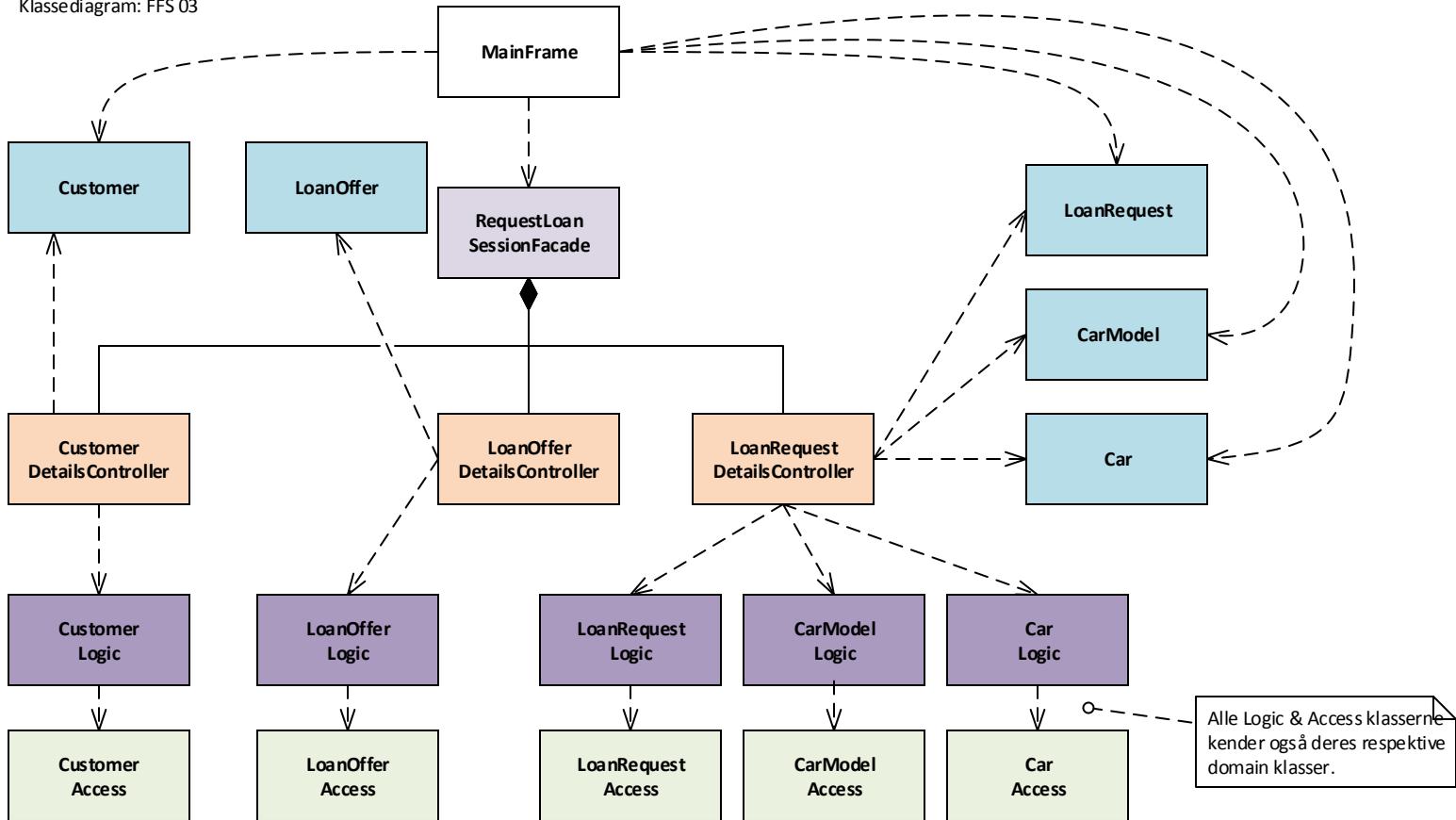
Pakkediagram



Klassediagram: FFS 01



Klassediagram: FFS 03



1

FF

Ferrari ✓ 90°

Name: [Name]

Adres: [Address]

[Logo]

UC 1:

Logo [Logo] FF [ID-label] [ID-label]

CPR: [CPR]

Bemerk: [Bemerk]

Name: [First] [Last]

Adres: [Adres]

Postnr.: [Postnr.] By [By]

T/F: [T/F]

Email: [Email]

[Multi] [Naar]

Logo [Logo] [ID-label] [ID-label]

Model: [Drop-Down]

Pn's: [Pn's]

Rabat: [Rabat]

Total: [Total]

Ud Betalning: [Ud Betalning]

Finansiering: [Finansiering]
[Text]

Afbetalingssperiode: [Afbetalingssperiode]

[Gem]





Ferrari Finance System



Brugernavn:

Adgangskode:



Ferrari Finance System

Bruger: Jens Pedersen



Menu

CPR-nr.:

Kreditværdighed:

Navn:

<i>Fornavn</i>	<i>Efternavn</i>
----------------	------------------

Adresse:

Nr./etage/dør:

Postnr.:

 By:

Telefon:

Email:

Nulstil

Næste

Ferrari Finance System



Bruger: Jens Pedersen

Menu



Model:

Pris:

Rabat:

Total:

Udbetaling:

Finansering

Afdragsperiode:

Model specifikation:



Ferrari Finance System

Bruger: Jens Pedersen

Menu



CPR-nr.: KV:

Navn: Fornavn Efternavn

Adresse:

Nr./etage/dør:

Postnr.: By:

Telefon:

Email:

Nulstil

Næste



Ferrari Finance System

Bruger: Jens Pedersen

Menu



CPR-nr.:

Kreditvurdering

X

Ferrari Finance System



Grundet problemer med denne
kunde tidligere, er salget afbrudt!

OK

VisAnmodninger

X

Løbenummer	Dato	Kunde	Sælger

Vis

Luk

X

Låneanmodning

Louise Niemann Hansen (B)
150790-xxxx

Adresse
Fløvej 41
7330 Brande

Kontakt
41662262
Niemannlouise@gmail.com

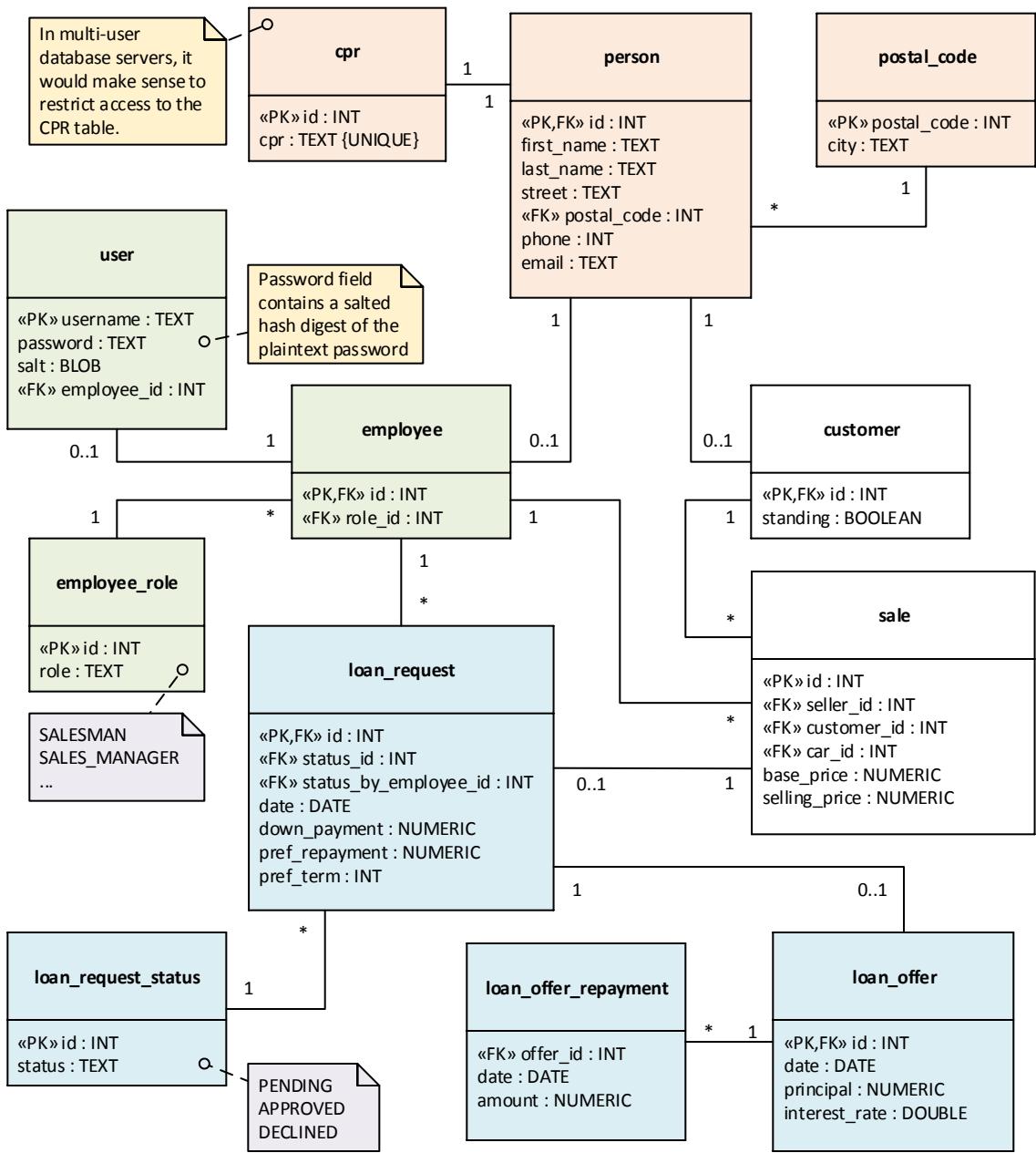
Bil
Ferrari 480 GTB
856.200 kr

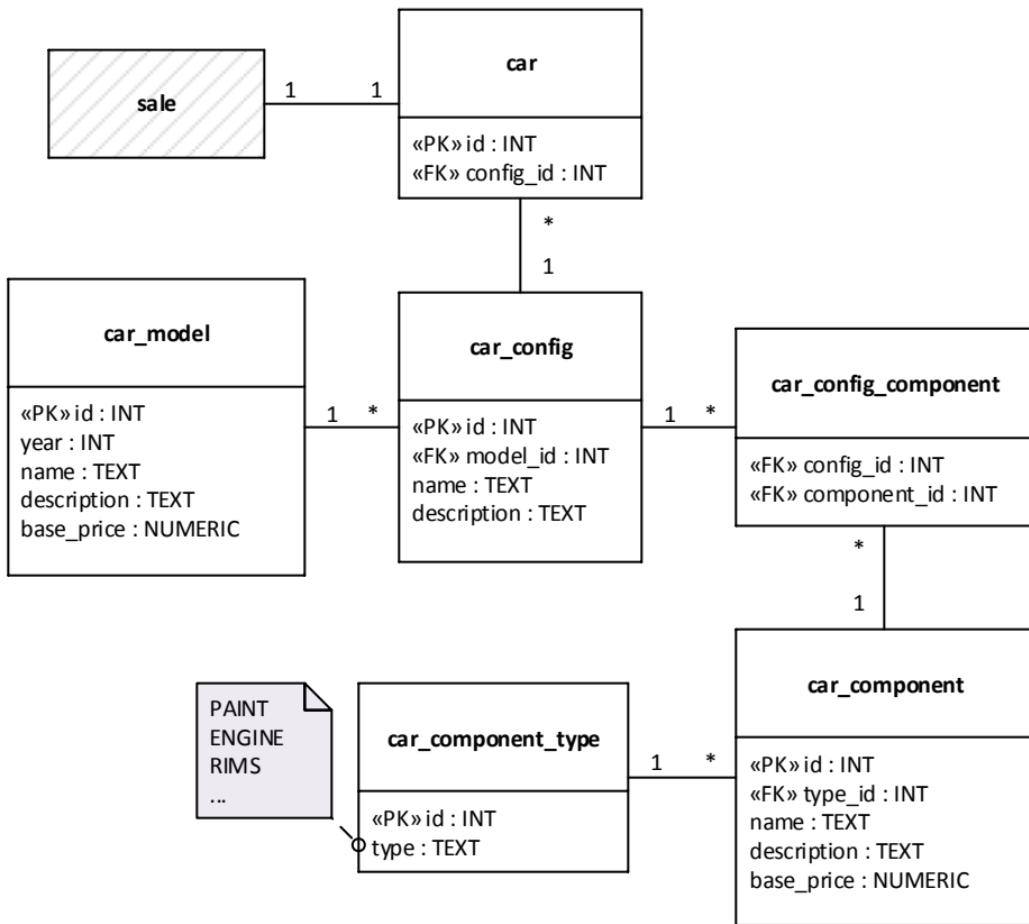
Lån
Udb. 18.000 kr.
Lånes: 838.200 kr.
Afb. P. 72 md.

Afvis

Godkend

Luk





Testsuite OC1 og OC2

<u>Gør sig gældende:</u>	<u>Input</u>	<u>Output</u>
Cpr		
• Cpr på 9 cifre	012345678	"Skal indeholde 10 cifre"
• Cpr på 11 cifre	01234567890	"Skal indeholde 10 cifre"
• Cpr indeholdende andet end tal 0 - 9	012345678a9	"Må kun indeholde tal"
• Cpr på 10 cifre, bestående af tallene 0 - 9	0123456789	0123456789
Kreditværdighed		
• Kreditværdigheden er A		A
• Kreditværdigheden er B		B
• Kreditværdigheden er C		C
• Kreditværdigheden er D		D
Fornavn		
• Fornavn indeholdende tal el. tegn	L0u1se	"Må kun indeholde bogstaver"
• Fornavn kun bestående af bogstaver	Louise	Louise
Efternavn		
• Efternavn indeholdende tal el. tegn	N1emann	"Må kun indeholde bogstaver"
• Efternavn kun bestående af bogstaver	Niemann	Niemann
Gadenavn		
• Gadenavn bestående af tal el. tegn	N0rr3gade	"Må kun indeholde bogstaver og tegn"
• Gadenavn kun bestående af bogstaver	Nørregade	Nørregade
Gadenummer		
• Gadenummer på u. 10 cifre	123	123
Postnummer		
• Postnummer u. 1000	0900	"Postnummer mellem 1000 og

		9990"
• Postnummer på mindre end 4 cifre	123	"Postnummer mellem 1000 og 9990"
• Postnumre på over 9999	10213	"Postnummer mellem 1000 og 9990"
• Postnummer indeholdende bogstaver el. tegn	75oo	"Postnummer mellem 1000 og 9990"
• Postnummer korrekt	7330	7330
Bynavn		
• Bynavn indeholdende tal el. tegn	Ko1d1ng	"Bynavn må kun bestå af bogstaver"
• Bynavn kun indeholdende bogstaver	Kolding	Kolding
Telefonnummer		
• Telefonnummer på mindre end 8 cifre	1234567	"Telefonnummer skal bestå af 8 cifre"
• Telefonnummer på mere end 8 cifre	123456789	"Telefonnummer skal bestå af 8 cifre"
• Telefonnummer indeholdende bogstaver el. tegn	12345678	12345678
E-mail		
• E-mail u. "@"	Niemannlouise@gmail.com	"E-mail skal indeholde "@" og ".+"
• E-mail u. "."	Niemannlouise@gmailcom	"E-mail skal indeholde "@" og ".+"
• E-mail korrekt m. "@" og "."	Niemannlouise@gmail.com	Niemannlouise@gmail.com
Finansiering		
• Udbetaling er under 20 % af bilens pris	17.000kr. På model til 853.200	"Udbetalingen er for lille"
• Udbetaling er over 20 % af bilens pris	18.000kr. På model til 853.200	Intet

TestSuite AnnuityCalculator

TC#	INPUT	OUTPUT		
1	Beløb Rente Løbetid Termin	1.000,00 10,000% 2 1	Ydelse Afdrag Renter Gestgæld	506,26 497,93 8,33 502,07
2	Beløb Rente Løbetid Termin	1.000,00 10,000% 2 2	Ydelse Afdrag Renter Gestgæld	506,26 502,07 4,18 0,00
15	Beløb Rente Løbetid Termin	1.000,00 10,000% 10 1	Ydelse Afdrag Renter Gestgæld	104,64 96,31 8,33 903,69
16	Beløb Rente Løbetid Termin	1.000,00 10,000% 10 10	Ydelse Afdrag Renter Gestgæld	104,64 103,78 0,86 0,00
3	Beløb Rente Løbetid Termin	0,01 100,000% 2 1	Ydelse Afdrag Renter Gestgæld	0,01 0,00 0,00 0,01
4	Beløb Rente Løbetid Termin	0,01 100,000% 2 2	Ydelse Afdrag Renter Gestgæld	0,01 0,01 0,00 0,00
5	Beløb Rente Løbetid Termin	1.000,00 0,001% 2 1	Ydelse Afdrag Renter Gestgæld	500,00 500,00 0,00 500,00
6	Beløb Rente Løbetid Termin	1.000,00 0,001% 2 2	Ydelse Afdrag Renter Gestgæld	500,00 500,00 0,00 0,00
7	Beløb Rente Løbetid Termin	1.000,00 10,000% 1 1	Ydelse Afdrag Renter Gestgæld	1.008,33 1.000,00 8,33 0,00
8	Beløb Rente Løbetid Termin	0,0949 10,000% 2 1	Exception: Ugyldigt argument: Beløb skal være >= 0.10	
9	Beløb Rente Løbetid Termin	1.000,00 0,000% 2 1	Exception: Ugyldigt argument: Rente skal være > 0	
10	Beløb	1.000,00	Exception: Ugyldigt argument:	

	Rente	10,000%	Løbetid skal være > 0
	Løbetid	0	
	Termin	1	
11	Beløb	1.000,00	Exception: Ugyldigt argument:
	Rente	10,000%	Termin skal være > 0
	Løbetid	2	
	Termin	0	
12	Beløb	1.000,00	Exception: Ugyldigt argument:
	Rente	POS_INF	Rente skal være < POS_INF
	Løbetid	2	
	Termin	1	
13	Beløb	1.000,00	Exception: Ugyldigt argument:
	Rente	NaN	Rente kan ikke være NaN
	Løbetid	2	
	Termin	1	
14	Beløb	1.000,00	Exception: Ugyldigt argument:
	Rente	10,000%	Termin skal være <= løbetid
	Løbetid	1	
	Termin	2	
17	Beløb	null	Exception: Ugyldigt argument:
	Rente	10,000%	Beløb kan ikke være null
	Løbetid	2	
	Termin	1	

TC#	INPUT		OUTPUT	Test Suite Int	
					TC#
1	Dagsrente	1,00%	Rente	3,00%	13
	Kreditværdighed	A			
	Udbetalingspct.	50,00%			
	Løbetid	37			
2	Dagsrente	1,00%	Rente	2,00%	14
	Kreditværdighed	A			
	Udbetalingspct.	50,00%			
	Løbetid	36			
3	Dagsrente	1,00%	Rente	4,00%	15
	Kreditværdighed	A			
	Udbetalingspct.	20,00%			
	Løbetid	37			
4	Dagsrente	1,00%	Rente	3,00%	16
	Kreditværdighed	A			
	Udbetalingspct.	20,00%			
	Løbetid	36			
5	Dagsrente	1,00%	Rente	4,00%	17
	Kreditværdighed	B			
	Udbetalingspct.	50,00%			
	Løbetid	37			
6	Dagsrente	1,00%	Rente	3,00%	18
	Kreditværdighed	B			
	Udbetalingspct.	50,00%			
	Løbetid	36			
7	Dagsrente	1,00%	Rente	5,00%	19
	Kreditværdighed	B			
	Udbetalingspct.	20,00%			
	Løbetid	37			
8	Dagsrente	1,00%	Rente	4,00%	20
	Kreditværdighed	B			
	Udbetalingspct.	20,00%			
	Løbetid	36			
9	Dagsrente	1,00%	Rente	5,00%	21
	Kreditværdighed	C			
	Udbetalingspct.	50,00%			
	Løbetid	37			
10	Dagsrente	1,00%	Rente	4,00%	22
	Kreditværdighed	C			
	Udbetalingspct.	50,00%			
	Løbetid	36			
11	Dagsrente	1,00%	Rente	6,00%	23
	Kreditværdighed	C			
	Udbetalingspct.	20,00%			
	Løbetid	37			
12	Dagsrente	1,00%	Rente	5,00%	24

Kreditværdighed	C	
Udbetalingspct.	20,00%	
Løbetid	36	

erestRateCalculator

INPUT	OUTPUT
Dagsrente	1,00%
Kreditværdighed	A
Udbetalingspct.	49,99%
Løbetid	37
Dagsrente	1,00%
Kreditværdighed	A
Udbetalingspct.	99,99%
Løbetid	37
Dagsrente	1,00%
Kreditværdighed	A
Udbetalingspct.	50,00%
Løbetid	1
Dagsrente	1,00% Exception: Ugyldigt argument: A Udbetalingspct. skal være >= 20 %
Kreditværdighed	19,99%
Udbetalingspct.	37
Dagsrente	1,00% Exception: Ugyldigt argument: A Løbetid skal være > 0
Kreditværdighed	50,00%
Udbetalingspct.	0
Dagsrente	1,00% Exception: Ugyldigt argument: A Udbetalingspct. skal være < 100 %
Kreditværdighed	100,00%
Udbetalingspct.	37
Dagsrente	1,00% Exception: Ugyldigt argument: D Kreditværdighed skal være A, B eller C
Kreditværdighed	50,00%
Udbetalingspct.	37
Løbetid	Dagsrente
Dagsrente	Nan Exception: Ugyldigt argument: A Dagsrente kan ikke være Nan
Kreditværdighed	50,00%
Udbetalingspct.	37
Løbetid	Dagsrente
Dagsrente	POS_INFTY Exception: Ugyldigt argument: A Dagsrente kan ikke være POS_INFTY
Kreditværdighed	50,00%
Udbetalingspct.	37
Løbetid	Dagsrente
Dagsrente	NEG_INFTY Exception: Ugyldigt argument: A Dagsrente kan ikke være NEG_INFTY
Kreditværdighed	50,00%
Udbetalingspct.	37
Løbetid	Dagsrente
Dagsrente	1,00% Exception: Ugyldigt argument: A Udbetalingspct. kan ikke være NaN
Kreditværdighed	Nan
Udbetalingspct.	37
Løbetid	Dagsrente
Dagsrente	1,00% Exception: Null pointer:

Kreditværdighed	null Kreditværdighed kan ikke være null
Udbetalingspct.	50,00%
Løbetid	37