Forsiden

Anders Looft, Thomas Nielsen,

Simon Lorentsen og Lasse Meilby.

//he15dmu-2s14

// FFS

//Førsteårsprøven

//Erhvervsakademi MidtVest

//Datamatiker

//Dato Juni 2015

//Vejleder: Anders W. Petersen, Flemming K. Jensen, Hans Iversen

Indholdsfortegnelse

[Opgave start 1](#_Toc420491904)

[Indledning 1](#_Toc420491905)

[Design patterns 2](#_Toc420491906)

[MVC pattern 2](#_Toc420491907)

[Anvendelse af singleton-pattern 2](#_Toc420491908)

[Hvorfor singleton? 2](#_Toc420491909)

[Observer pattern: 3](#_Toc420491910)

[Yderligere overvejelser omkring design patterns (mangler) 4](#_Toc420491911)

[Unified process 4](#_Toc420491912)

[Inception 4](#_Toc420491913)

[Elaboration: 5](#_Toc420491914)

[Construction 5](#_Toc420491915)

[Transition 5](#_Toc420491916)

[Dicipliner i UP: 6](#_Toc420491917)

[BPR 6](#_Toc420491918)

[UCD 6](#_Toc420491919)

[Reverse Engineering 7](#_Toc420491920)

[Forward Engineering 7](#_Toc420491921)

[Visionsdokument 8](#_Toc420491922)

[Vision: 8](#_Toc420491923)

[Interessentanalyse: 8](#_Toc420491924)

[Featureliste: 9](#_Toc420491925)

[Use case Diagram 10](#_Toc420491926)

[Domænemodel 11](#_Toc420491927)

[Argumentation for 3. normalform 12](#_Toc420491928)

[Endelige overvejelser 13](#_Toc420491929)

[Systemsekvensdiagram 15](#_Toc420491930)

[Aktivitetsdiagram 15](#_Toc420491931)

[Operationskontrakter 16](#_Toc420491932)

[Sekvensdiagram 17](#_Toc420491933)

[Klassediagram 21](#_Toc420491934)

[GRASP 22](#_Toc420491935)

[Kobling 22](#_Toc420491936)

[Samhørighed 23](#_Toc420491937)

[Test 23](#_Toc420491938)

[Dataordbog 24](#_Toc420491939)

[Årlig omkostning i procent (ÅOP): 25](#_Toc420491940)

[Intensionel definition 25](#_Toc420491941)

[Ekstensionel definition 25](#_Toc420491942)

[Konklusion 25](#_Toc420491943)

SKRIV BILAG HER (opdateres ikke, pga. sidetal)

# Opgave start

# Indledning

Vi har i dette projekt arbejdet iterativt over hver use case, og sågar over hele forløbet. Vi er gået ind i det med den indstilling, at vi skulle kunne dokumentere alt hvad vi implementere, og at vi ville nå så mange use cases som muligt. Opgaven lød på at systemet skulle have et letforståelig og intuitivt interface. Samt at feedback i brugergrænseflade skulle være hurtig. Vi har skulle designe og skabe en database, som kunne indeholde oplysninger omkring kunder, sælgere og biler samt aftaler. Alt i mens at personnumre på kunder skulle behandles med diskretion. En CSV-fil eksportering er også blevet implementeret, hvor en oversigt over lånetilbuddet samt tilbagebetalingsplan indgår.

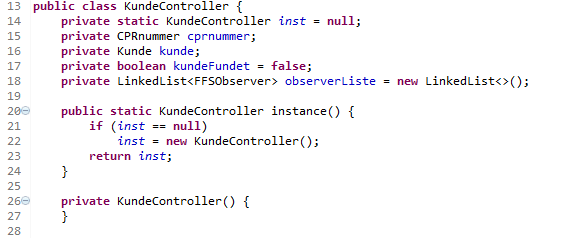
# Design patterns

## MVC pattern

Vi har i vores projekt valgt og adopdatere en af de mest brugte design patterns. MVC-pattern, også kaldet model, view controller. Dette har vi valgt og gøre da vi vidste allerede fra starten at dette design pattern ville passe fint til vores system, af den årsag at der er en masse kommunikation der skal behandles fra viewet og ned til modellen. Dette vil gør systemet nemmere og ændre på i fremtiden, eftersom at det kun er controlleren der ændre på systemets model. Eftersom at Observer pattern også er blevet en del af vores system, har vi valgt og have flere end 1 controller. Dette gør det mere sigende, hvem der skal behandle hvad, frem for vi lagde alt control mæssigt i en stor klasse.

## Singleton pattern

Hvorfor singleton?

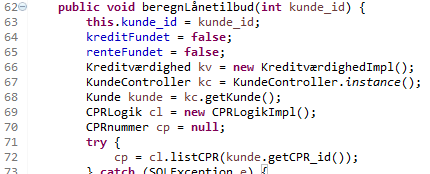
Vi har til alle vores controllere valgt at anvende singleton-pattern. Idéen opstod tidligt i udviklingsforløbet, da vi indså vi havde brug for kun at lave én instans af hver controller, og så nemt kunne genbruge denne fra flere logik implementationer. Et eksempel på implementationen finder man i KundeController klassen, som set nedenfor. 

Et af kendetegnene er den private contructor, der kun tillader klassen selv at oprette en instans. Constructorkaldet sker i instance() metoden, der, hvis ikke der allerede findes en instans, laver constructorkaldet, og returnerer dette. Hvis objektet allerede er blevet instantieret, returneres instansen.

Et eksempel på instantieringen finder vi i LånetilbudPanel, som vist nedenfor.



I KundePanel linje 43 findes vi præcis den samme linje, der også her forsøger at instantiere. Uanset hvad der får held med at oprette instansen først, har vi nu to klasser der begge har tilgang til den samme controller, og dermed den samme datakerne. Dette anvendes blandt andet i LånetilbudController, hvis vi får adgang til den allerede oprettede KundeController, og dermed let kan få returneret de instansvariabler der ligger gemt. I nedenstående tilfælde er det en instans af Kunde vi ønsker (se figur xx, kodelinje 68).



## Observer pattern

Observer pattern, også kaldet publisher-subscriber pattern, er et design mønster som overordnet går ud på at få viderebragt information om at der er sket ændringer. Måden det foregår på er ved er at man har en observer, som kan observerer på noget (subject), og derved få besked om ændringer. Rent praktisk sker dette i 3 faser:

1. Observeren tilmelder sig som hos subject.
2. Subject meddeler observeren om tilstandsændringer hver gang det sker.
3. Observeren framelder som som observer på subject.

I observer pattern findes der to forskellige teknikker til at oplyse om tilstandsændringer, push og pull. Vi har valgt at bruge push, som går ud på at subjectet både sender en reference til sig selv som parameter, og en oplysning om hvad der er sket. Hvor pull går ud på at observeren selv aktivt skal spørge om det er sket noget.

Vi har i vores projekt anvendt ideen fra observer pattern om at kunne få besked om når der er sket ændringer. Vi har lavet vores egen FFSObserver som er et interface, med 1 enkelt metode som hedder update. Update tager 2 parametre, den første som er et Object det andet som er en string. Object bruger vi til at kunne finde ud af hvilken af vores konkrete observere der har givet besked. String parameteren bruger vi til at finde ud af hvilken metode der er blevet kaldt. På den måde ved vi hvilket subject ændringen er kommet fra, og kan derefter agere ud fra dette. Dette tillader os at foretage små opdateringer i programmet, selvom et enkelt subject kan have mange metoder. Det hjælper også til ikke at overskrive tekstfelter med det samme tekst.

For at kunne få det til at fungere, har vi så lavet 2 metoder i hver af vores controllere som som hedder tilmeldObserver, med en observer som parameter, og en notifyObserver med en string som parameter. TilmeldObserver tilføjer observeren til en liste, hvis den ikke allerede findes i den. NotifyObserver kører så listen igennem og kalder update på observerne, med den string som kommer fra notifyObservers.

Det sidste punkt med at framelde som observer har vi set bort fra, da vi kun arbejder med et frames og de samme panels, og ikke mener der skulle være noget behov for at skulle framelde sig.

## Yderligere overvejelser omkring design patterns (mangler)

# Unified process

Unified process (UP) er en iterativ udviklingsmetode som er inddelt i 4 faser. Inception, elaboration, construktion og transition. Hver af faserne har hver deres fokuspunkter, som vil blive beskrevet senere. Faserne køres sekventielt dvs man starter altid i inception og ender i transition. Der er nogle krav om hvad der skal være opfyldt for at man kan gå videre til den næste fase som vil være fasens milepæl. Hver fase kan inddeles i en til flere iterationer alt afhængigt at projektets kompleksitet og størrelse, hver iteration bliver så først planlagt, nedbrudt og udført, og før man starter næste iteration, vil man så lave iterationsplanen før man går videre.

Inception:  
Inception er den fase hvor man starter, og indebærer en opstart af projektet. Projektplanen bliver lavet og visionsdokumentet blive påbegyndt.

I Inception fasen startede vi vores første iteration, nemlig ”iteration 0”, hvor vi fik lavet en projektplan, og fik opsat et projekt i Git. Vi valgte fra starten af at bruge vores projektplan som et værktøj til at have et overblik over hvilke opgaver der skulle laves og inden for hvilken tidsramme. Vi ville ikke ligge vores fokus på hvor lang tid vi brugte på hver enkelt opgave og hvilke dage de blev lavet. Så datoerne for hver enkelt opgave er bare autogenereret. Milepælene er faktiske datoer hvor tingene var lavet. Milepælen for fasen er placeret lige før den næste fase.

I iteration 1 valgte vi at placerer vores BPR analyse da det var oplagt at bruge den til at danne os et godt overblik over det nuværende proces, og hvad problemerne ved den var.  
Vi fik så startet visionsdokument, lavet et use case diagram og lavet casual use cases samt en enkelt fully dressed use case. Vi vurderede at vores fokus skulle ligge på vores UC7 ”Udarbejd lånetilbud” eftersom det var den primære funktion i vores system. UC1 og UC2 var også 2 use cases som vi gerne ville ligge fokus på. Til sidst fik vi lavet en faseplan og en iterationsplan for iteration 2.

Milepæl for inception:

* De fleste usecases er identificerede
* Centrale use-cases er formelt beskrevet
* Udkast til vision er færdig
* Domænemodel er påbegyndt
* Dataordbog påbegyndt

## Elaboration:

I elaboration går fokus mere på analyse og design i forhold til requirements som ligger hovedsageligt i inception. Det er også her implementationen starter.

Vores elaboration startede med vores iteration 2, hvor vi gik i gang med at udarbejde flere fully dressed usecases og fik lavet casual use cases til. Og fik startet noget design med klassediagrammer og sekvensdiagrammer, og her gik vores vurdering på at vi godt kunne nå at implementerer UC1 og 2 i før vi startede på UC7. Så startede vi ud med UC2 opret kunde, og fik lavet diverse diagrammer til den. Og fik den implementeret. I forbindelse med implementationen har vi testet dele af programmet.  
Grunden til at vi ifølge UP ikke var gået over i construcrion fasen var at vi ikke havde fået lavet en supplerende kravsspecifikation, som er en milepæl for at elaboration kunne være gennemført. På trods af denne mangel havde vi været i construction fasen.

Milepæl:

* Alle usecases er identificeret
* De fleste usecases er formelt beskrevet
* Vision er stabil/færdig
* Arkitekturen er velbeskrevet
  + Beskrivelse af ikke-funktionelle krav
  + Eksekverbar prototype
* Projektplan med overblik over iterationer

## Construction

Indebærer implementation af resten af systemet med en tilhørende brugervejledning

Milepæl:

* Implementation er færdig
* Systemet består systemtest
* Systemet er stabilt til release
* Brugervejledning er udarbejdet

## Transition

Indebærer overdragelsen af softwaren.

Milepæl:

* Systemet er leveret og i drift
* Systemet består accepttests
* Brugerne er tilfredse

## Dicipliner i UP:

Business modelling:  
Her har vi placeret vores BPR og vores domæne model.

Requirements:  
Her finder vi artefakter som omhandler krav til programmet, så som visionsdokument, use cases, operations kontrakter, systemsekvensdiagrammer, aktivitetsdiagrammer og dataordbog.

Design:  
I denne disciplin, finder vi klassediagrammer og sekvensdiagrammer samt vores datamodel.

Implementation:  
Indebærer alt den kode vi har implementeret i eclipse.

Tests:  
Her ligger vores testsuite.

Deployment:  
Her ligger vores program og en brugervejledning.

Project management:  
Her findes vores projektplan

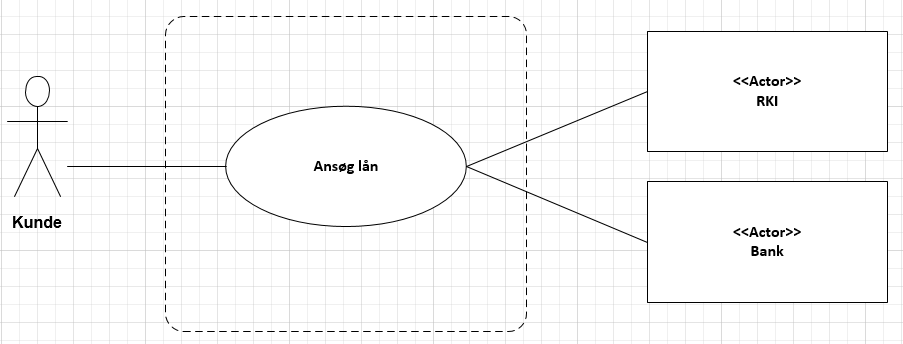
Environment:

Her ligger filer som opgavebeskrivelse, opslagsværker og diverse hjælpeværktøjer

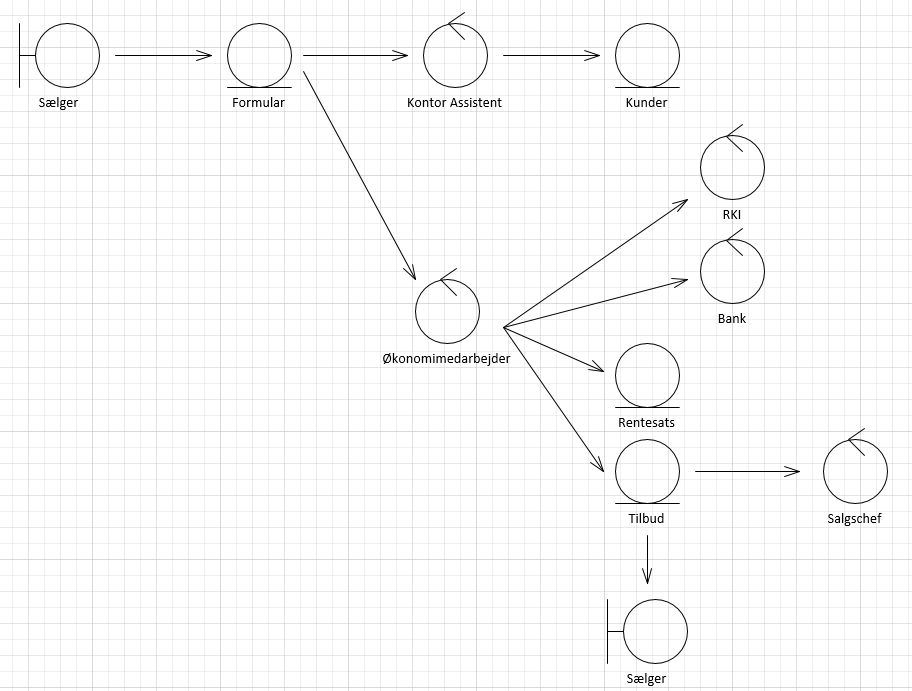
# BPR

## UCD

Vi har her lavet et Use Case Diagram for virksomhedens perspektiv. Dette giver os en idé om, at RKI og Bank spiller en stor rolle, når en kunde skal ansøge om et lån. UCD for både Reverse og Forward ser således ud, da dette ikke burde ændres ved implementationen af vores nye system.



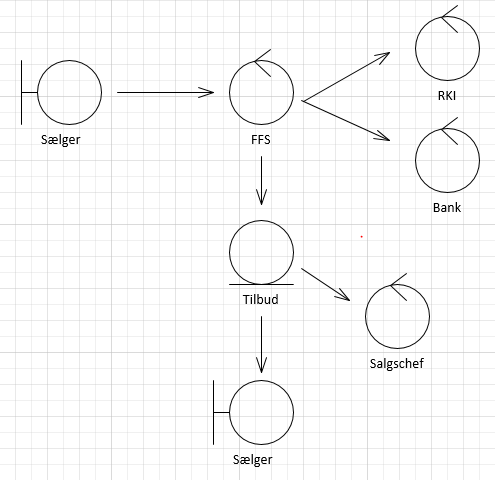
## Reverse Engineering

Vi har ved hjælp af Reverse Engineering, kunne identificere problem domænet, hvilket har givet os en forståelse for, hvordan det fremtidige system skal se ud. Object modellen viser at Økonomimedarbejderen har ekstrem meget ansvar i det nuværende system, da Økonomimedarbejderen skal igennem en masse processer før han/hun kan give lånetilbuddet videre til sælgeren, eller salgschefen, afhængig af lånetilbuddets størrelse, og om kunden har voldt problemer før.

## Forward Engineering

Vi har her ved hjælp af Forward Engineering, kunne fjerne store problemer ved det nuværende system. Objekt modellen nedenfor viser en tydelig forbedring i det fremtidige system. Dette udtrykkes ved at systemet(FFS) er kommet ind i billedet. Ved hjælp af FFS er det mulig of fjerne 2 control-objekter (Økonomimedarbejder og Kontor Assistent). Entity-objektet Formular ligger nu i FFS.

FFS står nu for at hente informationer fra RKI og bank. FFS sætter hermed også Rentesatsen, så der ikke længere er brug for en medarbejder til at gøre det. Dette systematisere processen betydeligt, hvilket realisere et hurtigere og mere effektivt system.



# Visionsdokument

## Vision:

Vi forestiller os et system der er baseret på at realisere en kundes drømme uden ventetid. Ferrari Financing System (FFS) har en automatiseret forbindelse til RKI og banken, og kan udregne det bedst mulige lånetilbud baseret på hver enkelt kunde inden for få minutter. Med FFS kan kunden altså få sin drømmebil med hjem den selv samme dag. Derudover vil FFS gøre det muligt for sælgeren at have flere salg med den automatiserede proces. Med vores system vil kunden trygt kunne udlevere sine personlige informationer, da systemet håndterer personlige informationer med diskretion.

## Interessentanalyse:

Kunden er interesseret i at hans/hendes oplysninger bliver gemt fortroligt. Samt at processen ikke tager for lang tid. Endvidere at han/hun kan få det bedst mulige tilbud.

Sælgeren er interesseret i et system der giver hurtig og korrekt respons, for at kunne lave flere salg.

Salgschefen er interesseret i at kunne modtage de lånetilbud der skal godkendes elektronisk. Endvidere er han interesseret i at forøge salget, og sikre høj kvalitet ved udregning af tilbud.

RKI har interesse i at personlige data bliver behandlet med diskretion, og sikkerheden til deres system er vedligeholdt.

Banken har interesse i at processen forløber hurtigt, og sikkerheden til deres system er vedligeholdt.

Ejeren har interesse i at der kan laves lånetilbud på kort tid, med høj kvalitet. Dette kan øge salget.

Datatilsynet har interesse i at personfølsomme oplysninger behandles korrekt og sikkert, i forhold til de gældende regler og love.

## Featureliste:

Lånetilbud

* Oprettelse
* Kreditvurdering
* Udregning af rentesats
* Godkendelse
* Eksportering
* CSV-fil

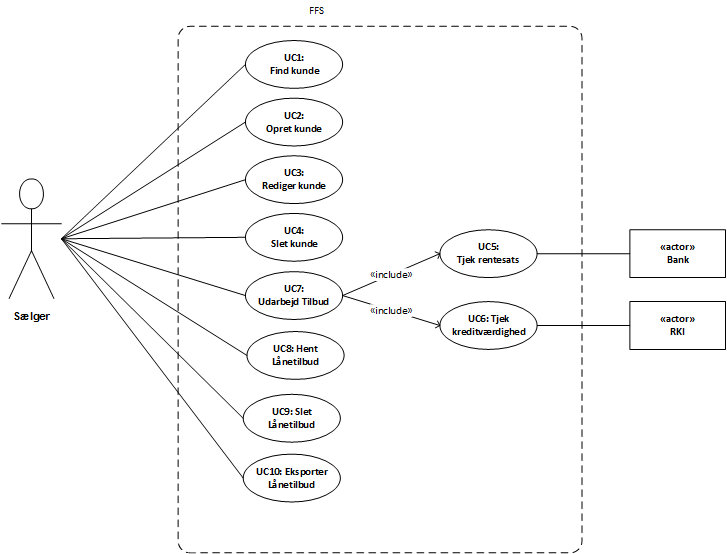
Kundehåndtering

Persistering af data

Om visions dokumentet

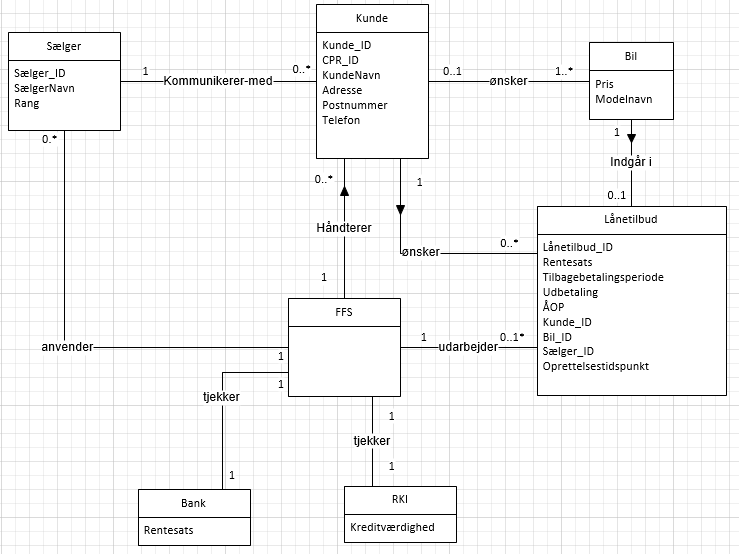
Visions dokumentet er et dokument som indeholder en vision, selve visions teksten, er beskrivelsen som skal sælge systemet til kunde, og overbevise kunden om at systemet er uundværligt, så derfor er det vigtigt at fokuserer på at visionen ikke er en kedelig opremsning. Ud over visions teksten indeholder dokumentet også en interessent analyse, som nævner hvem der kan have interesse i systemet og hvad de har interesse i. til Sidst er der en featureliste, som viser funktionerne i systemet i en rækkefølge med de vigtigste features øverst.

# Use case Diagram

Et use case diagram (UCD) er en artefakt, som man anvender til at få et overblik over alle use cases (UC) som er i systemet. Ligeledes giver det et overblik over hvem der har med de forskellige use cases at gøre altså aktørerne. Der findes både primære – og sekundære aktører. Den primære aktør vises altid til venstre i et UCD, og er den som er anvender den enkelte use case. Den sekundære aktør vises altid til højre i et UCD og kan for eksempel være et udefrakommende system som ens eget system interagerer med for at kunne udfører en given use case.   
i et use case diagram arbejdes der med 2 forskellige slags use cases, konkrete - og abstrakte use cases, hvor den konkrete er den som bliver startet af en aktør, og den abstrakte som startes af en anden use case. 

I vores UCD over FFS har vi en primær aktør som er sælger, fordi det er ham der anvender vores system og ham der interagerer med hver enkelt use case, som kan ses ved at der er en streg fra sælger til hver enkelt use case. Alle de use cases som er i forbindelse med sælgeren er derfor konkrete use cases og de to sidste UC5 og UC6 er abstrakte use cases. Udover at være en abstrakt use case har UC5 også en sekundær aktør RKI if UC6 har Bank som sekundær aktør.

# Domænemodel



Domænemodellen for det samlede system, har vi lavet for og give et klart overblik over problemdomænet. Vi ser tydeligt at FFS skal håntere, udarbejde og anvende en masse forskellige ting. Domænemodellen giver os her et overblik over hvordan strukturen vil komme til og se ud. Og hvordan et lånetilbud egentlig bliver oprettet.



Vi har valgt og vise domænemodellen for UC7, da det er en af de centrale use cases for vores system.

I denne use-case ser vi hvordan en sælger har kontakt med 1 eller flere kunder som ønsker et lånetilbud. Sælger anvender derefter systemet (FFS), til at udarbejde et lånetilbud til kunden. Systemet har kontakt til RKI og bank som returnerer henholdsvis Kreditværdighed og Rentesats til systemet.

# Argumentation for 3. normalform

Figur xx: Tidligt udkast af datamodel



Tidligt i forløbet så vores datamodel således ud. Dette var første udkast, og primært brugt til et overblik over hvilke værdier der skulle være i databasen. Vi udvidede hurtigt modellen til at inkludere information omkring sælger og bil, med dertilhørende værdier. Disse to blev koblet på lånetilbud med en foreign key/primary key relation. Dette tillod os at have tabeller med sælgere og biler oprettet, og blot linke dem på når et nyt lånetilbud blev lavet. Modellen levede stadig ikke op til de 3 normalformer. 1. normalform er bestået, da alle attributter kun har én værdi.

Det første problem lå i bynavn og postnummer, da bynavnet er fuldstændig afhængigt af postnummeret. Disse to blev derfor sat i en tabel for sig selv, og så lod vi postnummeret i kundetabellen referere til postnummeret i postnummer tabellen. Således havde vi altid både postnummer og bynavn tilgængeligt, såfremt vi kendte postnummeret.

Efter at vi havde flyttet pris ud i en tabel sammen med de nye attributter kunne modellen bestå 2. normalform, da alle attributter nu var fuld afhængige af den primary key der ligger i den tilsvarende tabel.

Til sidst valgte vi at lade CPRnummer ligge i en tabel for sig selv, men en dertilsvarende CPR\_id, der refererede tilbage til kunde tabellen. Det kan argumenteres at datamodellen allerede her opfyldte den 3. og sidste normalform, da der ikke direkte var nogle transitive afhængigheder. Det blev dog besluttet at flytte CPRnummeret ud, da dette evt. i en fremtidig revision kunne øge sikkerheden. Den opdaterede datamodel kan ses på figur xx nedenfor.

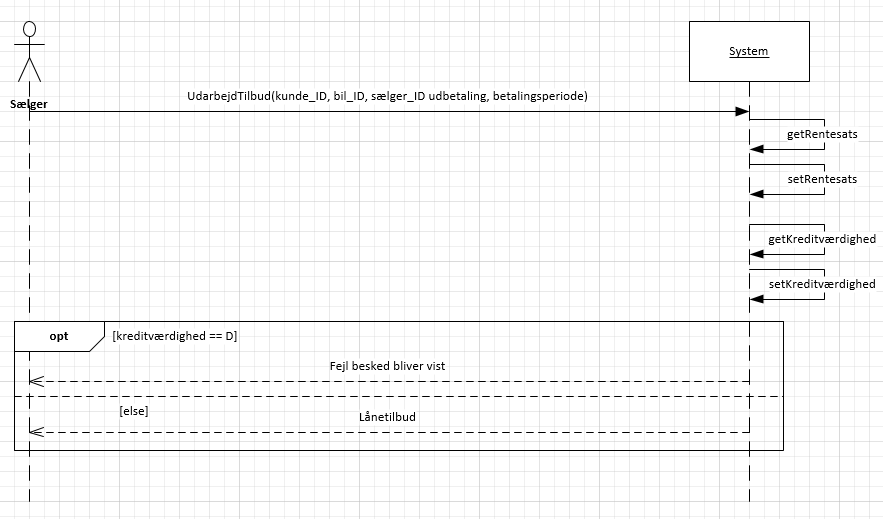
Endelige overvejelser

Databasen har et begrænset omfang, og der er helt klart plads til forbedringer. Vi har foretaget nogle valg der måske går imod visse standarder, for at holde det simpelt. Bil tabellen indeholde kun attributterne pris og modelnavn. I et virkeligt system ville dette ikke være tilstrækkeligt, da der ikke er plads til fx tilvalg eller tilbud.

Ligeledes findes der i sælgertabellen en attribut; beløbsgrænse. Denne *burde* være i en tabel for sig selv, forbundet til sælgertabellen med primary-/foreignkey relation via rang. Vi har valgt at begrænse vores system til ganske få sælgere, og derfor tillod de enkelte gentagne værdier i samme kolonne der nu måtte være. Det betyder selvfølgelig at det ikke overholder normalformerne til fulde. 

Figur xx: Datamodel

# Systemsekvensdiagram



Systemsekvensdiagrammet for UC7 – Udarbejdtilbud, går i sin store helhed ud på at systemet skal have: en kunde, bil, sælger et beløb samt en betalingsperiode, for og kunne fortsætte processen. Vi har vi dette tidelige stadie klar over, at en rentesats og en kreditværdighed, skulle findes før et tilbud kunne beregnes. Metode kaldet UdarbejdTilbud sender de nødvendige parameter med, og herefter starter systemet med og finde rentesatsen den given dag, og finde kreditværdigheden på kunden, som blev sendt med som parameter. Systemet tjekker om kunden har kreditværdigheden: D. Hvilket resultere i en fejl besked, da systemet ved allerede her at kunden ikke vil få lånet godkendt. Hvis kunden har kreditværdigheden A, B eller C, fortsætter systemet med beregningerne, og sender så tilsidst et lånetilbud tilbage til sælgeren.

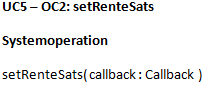
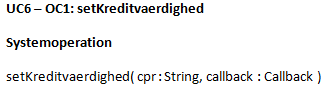
# Aktivitetsdiagram

I et aktivitetsdiagram tager vi en use-case og visualisere for os selv hvordan processen forløber. I et aktivitetsdiagram har man den primære aktør til venstre og systemet til højre som så er opdelt med en linje ned i midten.



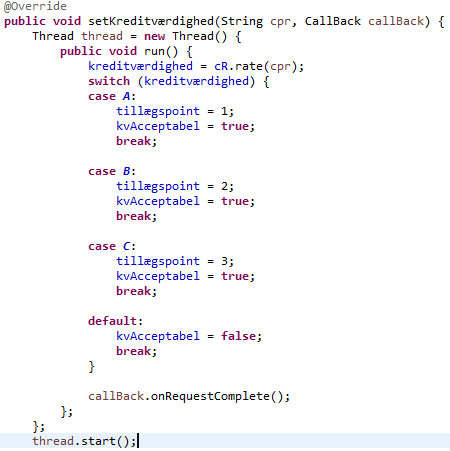
Man starter fra den sorte prik og så bevæger man sig ellers bare med pilene. Den roterede firkant som deler pilen i to, kaldes "decision". Her kigger man på to muligheder, altså en if/else. Efter man har fundet ud af hvilken vej der er den rigtige, fortsætter man så igennem ind til man når den sorteprik med en cirkel omkring, hvilket er slutningen på aktiviten.

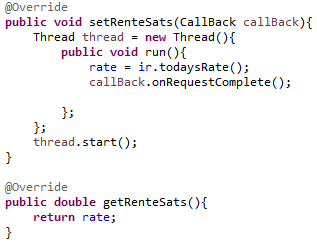
# Operationskontrakter

Ved operationskontrakterne har vi valgt og fokusere på centrale dele af systemet. Ved Usecase diagrammet blev der vist, en tydelig forbindelse imellem UC5+UC6 til UC7. I eksemplet ovenover ser vi vores første operationskontrakt: setKreditvaerdighed. Hvis vi ser på parameterne på denne metode(og får OC2), er der klart at en parameter springer i øjene: Callback.

Ideen med Callback parameteren var, at vi allerede her vist at disse to metoder(setKreditvaerdighed, setRenteSats) skulle køres i tråde. Her ser i koden til setKreditvaerdighed:

  
Callback parameteren tillader os og kalde en metode som hedder onRequestComplete(). Efter at tråden er blevet startet, bliver der oprettet forbindelse til RKI, hvilket så resulterer i en kreditværdighed. Dette bliver så tjekket igennem switch sætningen, hvor efter callback kalder onRequestComplete() på sig selv. Dette gør at systemet ved præcis hvornår metoden setKreditværdighed() er færdig. Ellers skulle systemet sidde og pinge i et fast interval, for og finde ud af hvornår den er færdig.

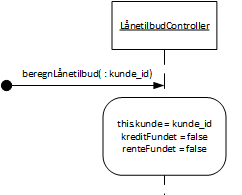


Ovenfor ser vi metoden setRenteSats(). Ideen er den samme som for setKrediværdighed(). Dette gør vores 2 tråde meget mere effektive, eftersom at systemet kan fokuser på andre opgaver, indtil at onRequestComplete() bliver kaldt.

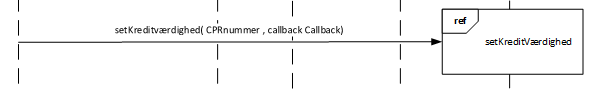
Operationkontrakterne kan ses i deres helhed i bilagene.

# Sekvensdiagram

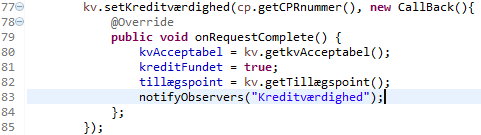
Sekvensdiagrammet for UC7 viser forløbet fra kaldet i LånetilbudPanel, indtil de to usecases (UC5 og UC6). Metoden den viser ligger i LånetilbudController, linje 62-96. Diagrammet er i sig selv rimelig selvforklarende, men der er alligevel flere elementer af interesse. Nedenstående eksempel viser konsistensen mellem diagram og kode, hvor metodekaldet resulterer i variabelerklæringen.

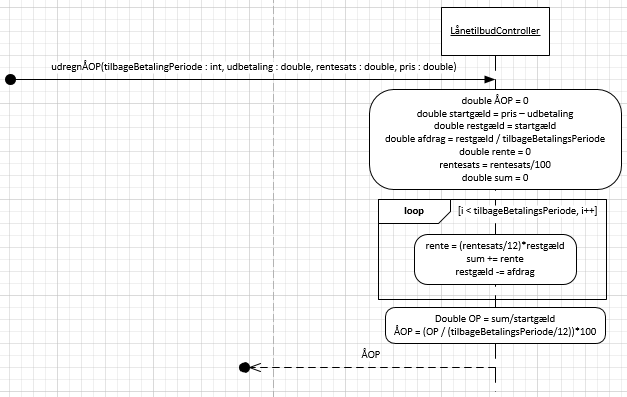
Videre i diagrammet ligger der en reference til setKreditværdighed, som er videre beskrevet i sekvensdiagrammet for UC6. Noteringen ser således ud:

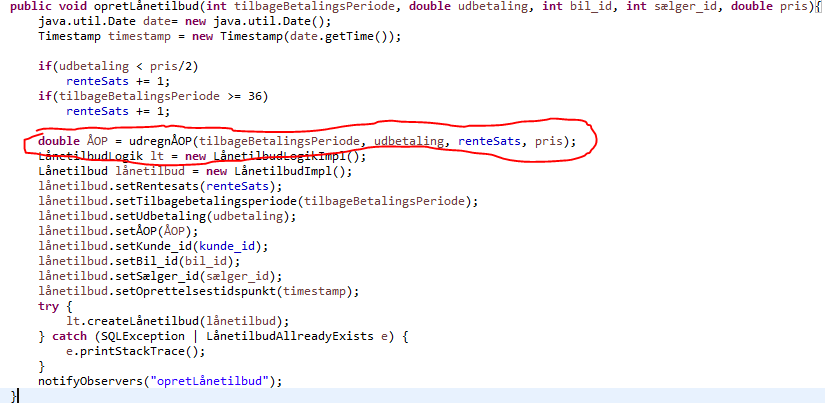


I koden kommer det til udtryk med kaldet setKreditværdighed på kv, der er en instans af KreditværdighedImpl.

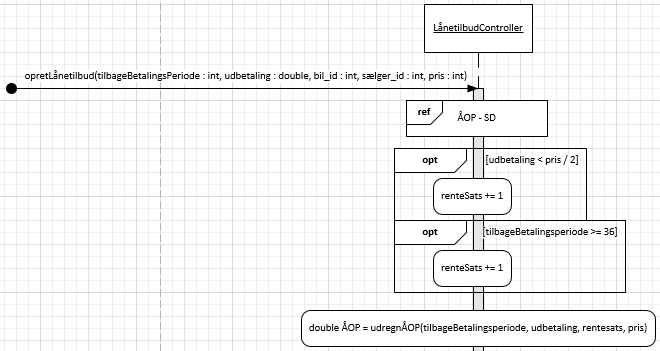


De to reference sætninger oplyser om de ændringer der er sket, når deres tråde er kørt færdig, og data behandles videre derfra. Sekvensdiagrammet for UC7 slutter derfor med de to referencekald, hvilket forklarer der ingen returkald eller anden form for afrunding ligger bagefter.

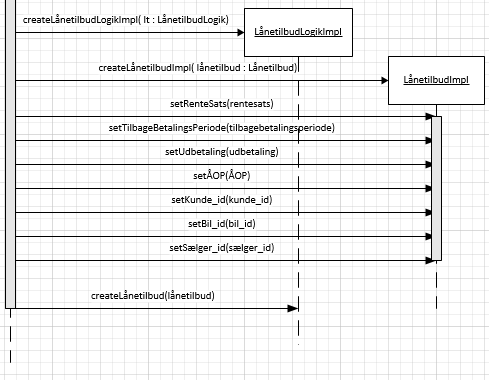


Sekvensdiagrammet for udregnÅOP viser primært hvordan vi kommer frem til ÅOP. Der bliver sat en masse instanser i starten til en masser forskellige værdier. Men det interresante sker i loopet. I loopet finder vi summen, som vi bruger til og udregne OP(Omkostnings procent). Dette gør vi ved kører en for løkke i gennem, for og finde frem til summen. For løkken kører en gang for hver måned tilbageBetalingsPeriode er blevet sat til. Vi ender så ud med og få OP, som vi skal bruge i formlen til ÅOP. LånetilbudController returnerer så til sidst ÅOP til den der har kaldt metoden. 

I dette eksempel kan vi se(markeret med rødt) at udregnÅOP bliver kald når et lånetilbud skal oprettes. Det er også vigtig og bemærke at, rentesatsen der bliver parameter i udregnÅOP, er den daglige rentesats fra banken. Denne rentesats bliver nemlig altid beregnet, før man kalder opretlånetilbud. Dette kan ses i sekvensdiagrammet for beregnLånetilbud.

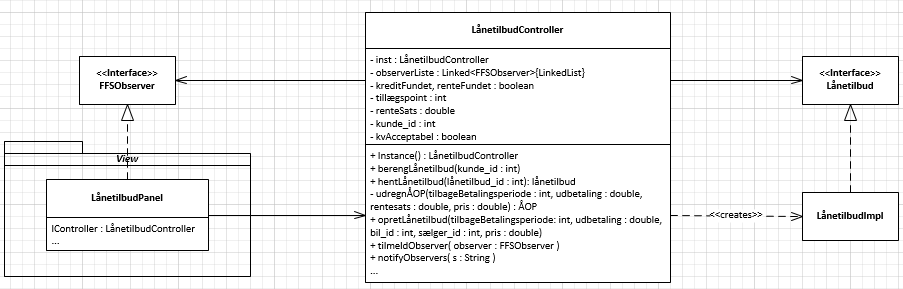


Sekvensdiagrammet for opretLånetilbuddet, ser vi den tydelige ref, vi nævte før. Går vi længere ned af livslinjen ser vi også at vi tager højde for at hvis udbetalingen er mindre en 50% af den absolute pris, så skal der ligges +1 procentpoint til rentesatsen. Dette gælder også hvis at tilbageBetalingsperioden er mere en 35 måneder. Efter vi så har den rigtige rentesats, bliver udregnÅOP kaldt.

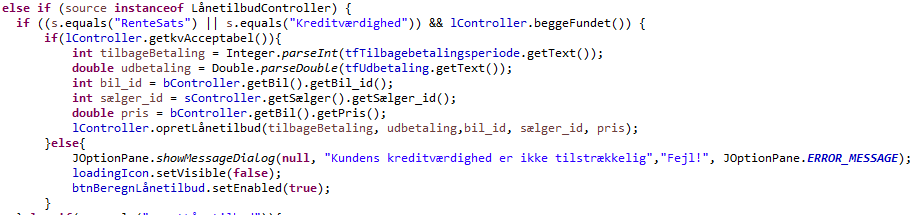


Senere på livslinjen bliver der lavet 2 objekter: LånetilbudLogikImpl og LånetilbudImpl. Herefter bliver alle de nødvendige instanser for og lave et låetilbud sat på LånetilbudImpl. Efter dette bliver der så sendt et lånetilbud afsted til LånetilbudLogikImpl, som så ligger lånetilbuddet ind i databasen. Sekvensdiagrammet for opretLånetilbud kan ses i dets helhed i bilagene.

# Klassediagram



Vi har valgt og fremhæve klassediagrammet for UC7, hvor vi ser hvordan vores LånetilbudController fungerer. Klassediagrammet giver os et overblik over hvad en LånetilbudController skal kunne og hvem der skal hjælpe den med og løse problemer. Viser illustrere en FFSObserver, hvilket giver en klar indikation om at observer pattern skal anvendes. Ved og asosiere LånetilbudController med interfaces FFSObserver, gør vi det muligt at observere om der sker noget i de andres observere. Dette gør det så muligt for LånetilbudController og beregne et lånetilbud. 

Vi ser her update metoden, som ligger i vores LånetilbudPanel. Denne metode bliver overrided fra FFSObserver interfacet. Dette gør så at når LånetilbudController klader notifyObservers() bliver metoden update kaldt i LånetilbudPanel. Metoden update gør selvfølgelig en masse forskellige ting, ud fra hvem det er der har kaldt den. 

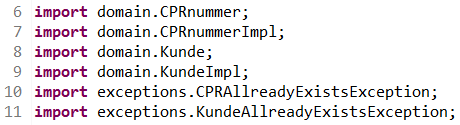
Dette stykke af koden fokusere vi på LånetilbudControlleren, og hvad der sker hvis den kalder notifyObservers(). I den første if sætning, tjekker vi om notifyObservers er blevet kaldt med ”RenteSats” eller ”Kreditværdighed”, og tjekke bagefter om både RenteSats og Kreditværdigheden er fundet, da dette er en betingelse for at kunne fortsætte processen. Herefter finder vi ud af om kreditværdigheden er acceptabel, hvis den er fortsætter vi. Derefter bliver en masse variabler instansieret, og herefter kalder update metoden videre på LånetilbudController, hvor opretLånetilbud bliver kaldt, med de værdier den har fået fra de andre controllerere.

I vores beregnLånetilbud metode som jo egentlig kalder update metoden i dette tilfælde, vil vi komme ind på i vores sekvensdiagram.

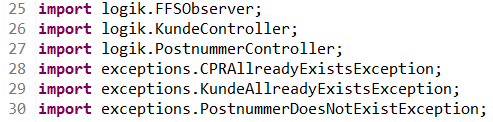
Vores update metoden kan ses i LånetilbudPanel klassen(view pakken) fra linje 340.

# GRASP

Et gennemgående koncept i udviklingen af vores system har været evnen til at kunne videreudvikle og/eller genbruge væsentlige dele af programmet. Vi har blandt andet valgt at anvende interfaces til de fleste klasser, for at gøre det lettere at udskifte/opdatere de dertilhørende klasser. Desuden har vi oprettet flere controllere, der hver især fungerer som information experts. Til funktionalitet på lånetilbud findes der derfor en lånetilbud controller, der håndterer logikken (der derved fjerner alt logik fra viewet) for lånetilbud. Vi har fulgt dette system igennem, men den eneste undtagelse af CPR numre. Logikken til dette ligger i kunde controlleren, da vi mente en CPR controller ville blive for lille, og alligevel være koblet til kunde controlleren. Et kig på importlisten for kundeController viser også hvilke forbindelser denne har til andre klasser



KundeController anvendes primært af KundePanel, der ligesom controlleren har begrænset sit import til de relevante klasser. Ud over en længere række GUI imports ser listen således ud:



Postnummer er ligesom CPR tæt knyttet til kunde, men der ligger alligevel lidt logik for sig selv, derfor har denne fået sin egen controller. Dette gør det ikke bare muligt at udskifte dele af eller hele controlleren, men også nemt at finde frem til den præcise metode man har brug for.

## Kobling

I vores system har vi selvfølgelig gået efter og få den mindst mulig kobling imellem objekter, dov stadig haft i hoved at for lav kobling kan føre til forvirring i det samlede billed af systemet. Da en vis kobling imellem objekter aldrig kan undgås har vi, som nævnt tideligere, gjort benytte af interfaces og controllers. Dette har selvfølgelig noget og gøre med hvilke design patterns vi har valgt og gå ud fra. Ved brug af disse interfaces har det lykkedes os, og skabe dependencies, hvilket er den svageste kobling, fremfor nedarvning som er den stærkeste. Man kan vel sige at vi har gået efter ikke at bruge nedarvning. Dette har dov været nødvendigt i vores tests. Det er også blevet brugt i vores GUI kode, men her har nedarvningen været af Jpanels osv.

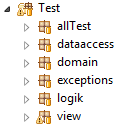
## Samhørighed

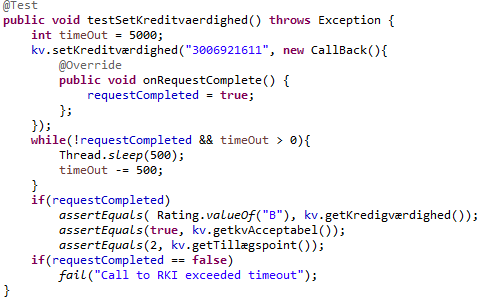
Som tidligere nævnt har en høj samhørighed været i højsædet under udviklingen. Vi har så vidt muligt begrænset hver klasse til det relevante scope. Således kan der trækkes en lige linje fra KundePanel til KundeController, via KundeLogik til Kunde i domain.

For at mindske koblingen ligger der både metoder til at oprette kunde og finde kunde i kundeController. Disse metoder har som sådan ikke noget med hinanden at gøre, og for at øge samhørigheden kunne disse være lagt i klasser for sig selv. Vi har dog valgt at lægge dem sammen, dels for at undgå for små controller klasser, dels for at have en høj samhørighed ved controllere.

# Test

Vi har i vores project selvfølgelig testet vores system undervejs i processen. Dette har vi gjort på en systematisk måde, så hvis vi senere i processen skulle ændre noget i koden, skulle testene stadig kunne køre. Vi har valgt og fokusere på centrale dele af systemet, og lave test suites til disse. Grunden til at hele system ikke bliver testet er, at man kan ende med og dobbel teste bestemte dele af systemet, og vi ikke har set det nødvendigt og gøre. Vores test har hjulpet os til en bedre forståelse for dele af vores kode, samt også gjort det muligt for os, at identificere fejl tideligere. De centrale dele vi har fokuseret på er UC5, UC6 og UC7.

Sådan ser vores test folder ud. Vi har delt det op på denne måde, da vi har alle domain test klasser i domain packagen osv. Dette gør det muligt for os og test hvad der nu skulle ligge vores domain package i vores src folder. I hver package har vi også en TestSuite for alle klasser i den package. De bliver så alle sammen samlet i allTest, som er en TestSuite til alle vores tests.



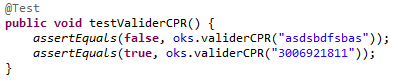
Her er koden for vores KreditvaerdighedTest klasse. Dette er den eneste metode i køre i denne klasse. Dette skyldes at der kun skal testets på en ting i vores Kreditvaerdighed klasse. Om den gør det rigtige ud fra hvilken kreditværdighed kunden nu har. Der bliver instansieret 2 private variabler i denne klasse også:

kv som er af typen Kreditværdighed, samt en boolean kaldet requestcompleted.

Vi også at det er nødvendigt for os at lave en variable i metoden: timeOut. Dette skyldes at hvis vi kørte metoden uden vores timeOut, ville metoden simpelthen fejle, da vores kreditvaerdighed metode der bliver kørt, er en Thread. Og denne Thread eksistere da vi både skal kunne hente kreditværdighed og rentesats. Uden og skulle vente på at den anden blev færdig. Vi tæller så timeOut ned i metoden så længe requestCompleted er false, og timeOut er størrere end 0. Vores onRequestComplete() er en metode der bliver overrided fra Kreditværdighed klassen.

Vi forvente så i vores assertEquals, at kreditværdigheden er ”B”, at kunden er acceptabel for lån, og at der bliver lagt 2 tillægspoint til vores tillægspoints variable. Hvis dette fejler, vil der blive skrevet en fejl besked ud.

Testen for RenteSats er for sin vis den samme. Metoderne til disse to klasser har hjulpet os med og forstå hvordan vores threads virker, og ikke mindst hvordan vores CallBack variable virker.

Vi har selvfølgelig også test på vores view. Her har vi fokuseret på og teste vores valider metoder, og om de virkelig kan valider input fra brugeren. Klassen tester alt fra CPRnummer, til navn på kunden. Denne klasse kan ses i sin helhed i view pakken i vores Test folder.

Test suites for systemet vil være og finde i bilagene.

# Dataordbog

I vores dataordbog finder man definitioner og eksempler på domæneterminologi. Væsentlige termer bliver beskrevet i dette afsnit, det samme med forkortelser. Der er to forskellige definitioner i dataordbogen. Først er der intensionel definition som giver en meget generel beskrivelse af det givne koncept. Den anden definition er den ekstentionelle definition, som giver et mere konkret eksempel på konceptet. I forhold til objekt orienteret programmering, vil klasser være lig med den intensionelle definiton og objekter vil være lig med den ekstentionelle definition.

## Årlig omkostning i procent (ÅOP):

### Intensionel definition

ÅOP er hvad navnet ligger op til, årlig omkostning i procent. Gebyrer, renter og andre udgifter bliver lagt sammen her så man får et nemt og overskueligt overblik over hvor meget man kommer til at skulle betale i procent om året af sit lån. Der er mange ting som påvirker ÅOP, om det er den givne rente eller stiftelsesgebyr eller andre afgifter variere fra långiver til långiver.

### Ekstensionel definition

Ved et lånetilbud vil man altid udregne ÅOP så kunden let kan se hvad han eller hun kommer til at skulle betale i procent om året. Det er med til at gøre det lettere for kunder at finde det rigtige lånetilbud til dem, da det ikke altid er en lavrente der et det billigste.

I vores låneaftale afdrager man med et fast beløb hver måned indtil hovedstolen er blevet betalt tilbage, der skal selvfølgelig betales renter af det resterende beløb hver måned ud over afdraget.

# Konklusion

Vi har hele projektet igennem, forsøgt og arbejde så iterativt som muligt. Der har været steder i projektet hvor vi har afviget lidt fra den normale iterative tilgang til konstruktioner af systemer. Dette har primært været diagrammer, som ikke altid blev lavet i den rigtige rækkefølge. Dette kunne resultere i at vi måtte sidde og rette vores diagrammer igennem mange gange, før vi kom frem til det endelige resultat.

Den iterative måde og arbejde har dog også bragt gode ting med sig. At vi har kunne fokusere hundred procent på og implementere 1 use case af gangen, har givet os nogle fordele, eftersom vi har arbejdet i en gruppe på fire. Det har givet os den fordel at vi kunne uddele opgaver imellem os, og være sikker på at alle havde den samme tilgangsvinkel til use casen.

Allerede tidligt i projektets forløb, besluttede vi at holde os til opgavebeskrivelsen. Som i kan se i vores usecase diagram, har vi use cases som ikke har noget med opgavebeskrivelsen at gøre. Men vi følte at alle vores use cases, var en del af vores ”drømme” system. Men eftersom at hver use case kræver en masse arbejde og implementerer, valgte vi de mest essentielle.

Vores brugergrænseflade har vi lavet så simpel som vi synes vi kunne. Hvis vi havde flere kompetencer inden for design af brugergrænseflader, mht farver og ligende, ville vi have lavet en mere spændende brugergrænseflade.

Vi har kunne konkludere i vores projekt, at få en database til at opfylde 3 normalform er nemmere sagt end gjort. Især når den bliver større og større. Vi opdagede flere gange at vores database ikke levede op til 3 normalform, og måtte så igang med at rette i databasen, hvilket altid vil resultere i en masse kode der også skal rettes.

Der blev stillet et krav i opgaven omkring en CSV-fil. Dette problem gik vi forholdsvis hurtig til, for at se om det var en funktion vi kunne implementere senere i projektet uden at skulle ændre meget på systemet. Da dette ikke var tilfældet valgte vi og implementere denne funktion til sidst.

Vi kan til slut sige, havde vi haft længere tid, havde systemet blevet væsentlig større hvor alle vores use cases havde blevet implementeret. Men eftersom kunden ikke bad om mere, er systemet blevet et lille, men effektivt lånesystem.

Bilag