Fleksibel styring af SKATs inddrivelsesprocesser med regelmotorer: Evaluering af Frameworket Drools

Jannik Lind Andreasen

Kongens Lyngby 2012
IMM-B.Eng-2010-86

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-B.Eng-2010-86

Forord

Denne afhandling er udarbejdet ved Institut for Informatik og Matematisk Modellering på Danmarks Tekniske Universitet i opfyldelse af kravene for erhvervelse af en B.Sc. i Informatik.

Afhandlingen beskæftiger sig med implementering af forretningslogik med regelmotorer.

Afhandlingen består af et leveret produkt, i form af en prototype, samt en rapport til dokumentation af dette.

Jeg vil gerne takke min vejleder på DTU, Ekkart Kindler, samt min vejleder i KMD A/S, Tage Kiilsholm Hansen, for deres tid og vejledning under projektforløbet.

Jeg vedkender mig, at indholdet i denne afhandling er forfattet af mig, medmindre andet er angivet.

Lyngby, 20-Januar-2012

Jannik Lind Andreasen

Indhold

Kapitel 1

Indledning

I 2005 blev inddrivelsesopgaven i det offentlige samlet under SKAT, og Et Fælles Inddrivelsessystem, EFI, blev derfor etableret. EFI har til formål at samle alle offentlige fordringer og restancer i ét system. Dette skal sikre hver enkelt borgers retssikkerhed gennem ensartet praksis og samtidig effektivisere arbejdsgangen for SKATs sagsbehandlere ved at lade disse fokusere på de arbejdsgange, som kræver faglige vurderinger, mens EFI automatiserer de processer, som er standardiserede og rutinepræget.

KMD A/S fik til ansvar at udvikle og implementere løsningen. Under analysefasen af løsningen blev det diskuteret om en regelmotor skulle anvendes til at implementere de mange og til tider komplekse forretningsregler som EFI rummer. Regelmotoren der blev foreslået var Drools Expert, da denne af KMD A/S, samt flere betragtes som værende de facto standard inden for regelmotorer.

Idéen om at implementere forretningslogik i EFI med Drools Expert blev dog valgt fra, til fordel for en almindelig Java-implementering. Dette skyldtes til dels, at udviklerne skulle læres op i ny teknologi inden udviklingen kunne påbegyndes, men vigtigst af alt, at det altid indebærer en vis risiko at arbejde med nye og ukendte teknologier.

Denne risiko ønskede SKAT ikke at løbe, da EFI var klassificeret som et højrisikoprojekt, hvor mange penge var involveret og risikoen for ikke at nå i mål var

rimelig, grundet løsningens kompleksitet.

KMD A/S er dog meget interesserede i en analyse af Drools Expert med et konkret eksempel på hvordan det kan anvendes.

Dette projekt vil tage udgangspunkt i selveste Skat EFI. Det vil blive vurderet hvad der med fordel kunne være implementeret med Drools Expert. Der stiles undervejs efter en så fleksibel, samt skalerbar løsning som muligt.

Til at demonstrere løsningen i drift vil der blive udarbejdet et webinterface, der skal fungere som en portal, hvor en sagsbehandler kan oprette og administrere kunder. Det skal endvidere være muligt, at generere hændelser, der omfatter kunderne, som var de fra eksterne systemer.

Arbejdet struktureres som følgende. Først vil forretningsmodellen blive fastslået. På baggrund af denne vil en analyse med henblik på selve softwareudviklingen blive foretaget. I denne vil der blive taget stilling til, hvilke koncepter der egner sig i en regelmotor. Der udarbejdes tidligt en prototype, der har til formål at give indblik i de tilgængelige teknologier, og hvilken indflydelse disse har på et design af den endelige prototype. Med denne viden, vil et endeligt design af prototypen blive udarbejdet. Efter implementeringen af denne prototype, vil en kvalitetssikring redegøre for, om den er egnet til den mængde kunder som SKAT ønsker systemet skal gå i drift med.

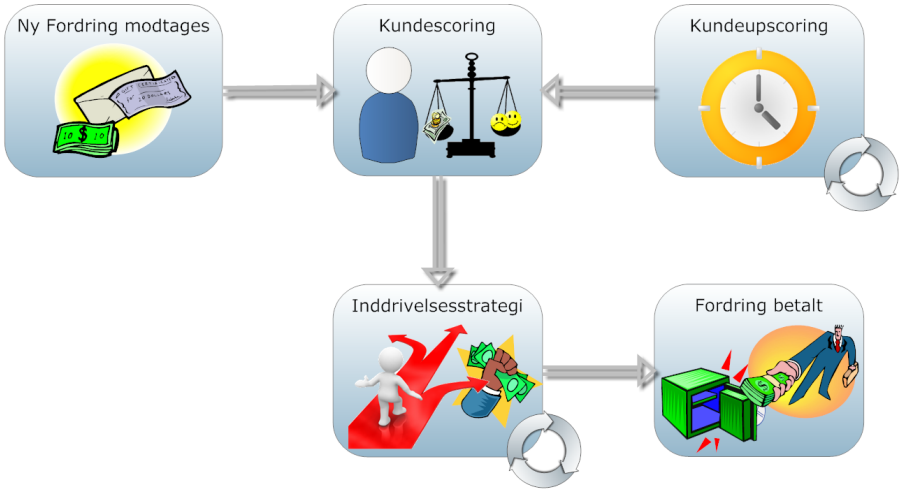
Kapitel 2

Forretningsmæssig analyse

Før at noget som helst kan udvikles, er det vigtigt at have en forståelse for forretningsmodellen og konceptet bag denne. SKAT ønskede at samle inddrivelsen af alle offentlige fordringer og restancer i ét system. Dette havde til formål at sikre hver enkelt borgers retssikkerhed gennem ensartet praksis. Samtidig skulle arbejdsgangen for SKATs sagsbehandlere effektiviseres, så mængden af standardiserede og rutineprægede opgaver blev mindsket.

SKAT havde udtænkt en løsning, hvor konceptet bestod i, at alle kunder bliver scoret ud fra deres sociale samt økonomiske forhold. Baseret på denne score vælges der en strategi for inddrivelsen, som formodes at være optimal i kundens aktuelle situation. Scoringen samt den valgte inddrivelsesstrategi skal ske automatisk. En sagsbehandler kommer således kun på en kundes sag, når der skal tages faglige beslutninger som systemet ikke kan træffe i sagen.

I de følgende underafsnit vil ovenstående forretningsmodel blive nærmere beskrevet.



Figur 2.1: SKATs forretningsmodel. Inddrivelsesstrategien vælges på baggrund af kundens betalingsevne og -vilje.

2.1 Forretningsmodel

SKATs forretningsmodel for inddrivelse af fordringer og restancer tager i høj grad hensyn til borgernes aktuelle situation. SKAT ønsker ikke at presse borgerne voldsomt og har derfor vedtaget en strategi, der skal sikre dette.

Når en ny fordring modtages, vurderes kunden med hensyn til dennes betalingsevne og -vilje. Denne vurdering vil blive nærmere forklaret i afsnit 2.2. På baggrund af kundevurderingen anbefales den inddrivelsesstrategi, som bør iværksættes overfor kunden. En inddrivelsesstrategi er en plan for hvordan en given kunde bør behandles. Inddrivelsesstrategier bliver nærmere beskrevet i afsnit 2.3. For hele tiden at anvende den mest hensigtsmæssige inddrivelsesstrategi overfor kunden, foretages der en scoring af kunden med faste intervaller, så længe kunden har fordringer til inddrivelse.

Figur 2.1 illustrerer denne forretningsmodel.

I de følgende underafsnit vil begreberne blive yderligere beskrevet.

2.2 Scoring af kunde

Udgangspunktet for inddrivelsen er scoringen af kunden. En kundes score er et udtryk for dennes betalingsevne og -vilje, med det formål at anbefale en inddrivelsesstrategi som kunden bør behandles efter.

Scoringen af kunden sker som det første, når en ny fordring modtages, og derefter med faste intervaller så længe kunden har fordringer til inddrivelse.

Herunder listes nogle af de parametre, der indgår i udregningen af score:

- Betalingsevne
  - Kundens nettoindkomst
  - Kundens ansvar i forhold til forsørgerpligt
  - Kundens stilling, dvs. pensionist, lønmodtager, kontanthjælpsmodtager
- Betalingsvilje
  - Har kunden tidligere skyldt penge
  - Har kunden tidligere nægtet eller nægter stadig at betale
  - Har kunden ikke svaret på henvendelser
- Eventuel evaluering af fordringstype
- Eventuel evaluering af fordringens størrelse

I den eksisterende løsning foretages scoreudregningen ikke af EFI selv, men af et system hos SKAT ved navn DebitorMotor Inddrivelse, også kaldet DMI. DMI er et system med adgang til informationer vedrørende kundens økonomiske forhold. DMI anvendes ydermere af SKAT til at håndtere betalingsordninger, og har derfor historik vedrørende tidligere betalingsordninger, samt forløbet af disse.

På baggrund af ovenstående informationer, udbyder DMI en score af kunden. Resultatet af scoreudregningen leveres som en simpel talværdi. Denne talværdi fortolkes derefter af EFI til en anbefalet inddrivelsesstrategi for kunden.

Da DMI ikke er tilgængeligt for dette projekt, vil udregningen af kundescoren blive integreret med fortolkningen af scoren. Udregning vil her ske alene på

baggrund af kundens betalingsevne samt -vilje. Disse to parametre vil blive indtastet af en sagsbehandler ved oprettelse af kunden i sagsbehandlerportalen. Denne løsning er valgt, da det for eksemplets skyld er mere intuitivt, at indtaste konkrete værdier vedrørende en kunde end en form for variabel, internt i systemet.

I afsnit 2.5, der omhandler sportyper, bliver det nærmere beskrevet hvordan betalingsevne samt -vilje omsættes til en anbefalet inddrivelsesstrategi.

2.3 Den samlede inddrivelsesstrategi

SKATs samlede inddrivelsesstrategi koncentrerer sig om at gøre inddrivelsen, så effektiv som muligt, i hver enkelt kundesag. Dette opnås ved, blandt flere mulige inddrivelsesstrategier, at vælge netop den, der passer bedst til den givne kundes situation.

SKAT har derfor introduceret følgende koncepter: Hændelse, Spor, Indsats og Aktivitet. Hændelser er drivkraften i den samlede inddrivelsesstrategi. Det er disse, der kommer med besked når noget nyt sker i en kundes sag. Dette kunne f.eks. være en besked om, at en kunde har betalt alle sine fordringer. Omvendt kunne det også være, at en kunde ikke har reageret på en udsendt rykker indenfor den fastsatte tidsfrist. Hændelser bliver nærmere beskrevet i afsnit 2.3.1. Sporet er, et udtryk for den valgte inddrivelsesstrategi for den givne kunde. Det består af én eller flere indsatser, hvor hver indsats er en konkret foranstaltning, som kan udføres over for en kunde, for at inddrive én eller flere fordringer. Indsatsen består af én eller flere aktiviteter og reagerer på hændelser. Indsatsen vil starte aktiviteterne baseret på de modtagne hændelser, samt hvilken tilstand indsatsen befinder sig i. Spor samt indsatser vil blive nærmere beskrevet i henholdsvis afsnit 2.3.2 og afsnit 2.3.3. Aktiviteter er tiltag som skaber fremdrift i indsatsen overfor en kunde. En aktivitet kan ikke afbrydes når den først er startet og reagerer heller ikke på hændelser. En aktivitet vil altid føre til et tilstandsskifte i indsatsen, der startede den. Aktiviteter bliver nærmere beskrevet i afsnit 2.3.4. Når inddrivelsen er påbegyndt vil kunden altid være placeret på et spor og være omfattet af mindst én indsats.

Set i et tidsmæssigt perspektiv, vil indsatser kunne vare alt fra dage til år. Et spor består som sagt af en eller flere indsatser, og vil derfor også vare alt fra dage til år. Aktiviteter vil oftest vare få sekunder.

2.1 Eksempel En kunde placeres på et spor. Dette spor starter som det første indsatsen Betalingsrykker. Indsatsen Betalingsrykker indeholder en regel der dikterer, at hvis betalingsrykkeren er sendt (en tilstand), og der modtages besked om, at kunden har betalt de fordringer som er omfattet af rykkeren (en hændelse), så skal indsatsen stoppes (en aktivitet).

2.3.1 Hændelser

Hændelser er drivkraften i den samlede inddrivelsesstrategi. Det er disse hændelser som sporene og indsatserne abonnerer på og det er derfor disse, der indirekte sætter aktiviteterne i gang. Set i et forretningsmæssigt perspektiv er en hændelse, en information, der vedrører en specifik kunde og formidler, at noget er sket i relation til kundens sag. En hændelse vil derfor altid have en kunde tilknyttet.

I eksempel 2.1 blev det beskrevet hvordan en modtaget hændelse om, at kunden har betalt sine fordringer medfører, at indsatsen Betalingsrykker stopper.

Der skelnes mellem tre former for hændelser; systemgenererede hændelser, udefrakommende hændelser og menneskeskabte hændelser. De systemgenererede hændelser kan være hændelser udløst af tid eller af aktiviteter, som ønsker at publicere en besked via en hændelse. Udefrakommende hændelser kommer fra andre systemer, der informerer om ændringer i en kundes sag. Dette kunne være en hændelse om, at kunden har misligholdt en betalingsordning. De menneskeskabte hændelser vil komme fra sagsbehandleren, når denne foretager sporskifte eller ændringer i indsatsplanen på et spor.

Alle former for mulige hændelser vil dog blive modtaget og behandlet på samme måde.

Eksempler på hændelser kunne være følgende:

- Nettoindkomst ændret – en hændelse, der orienterer om, at kundens forhold er ændret. Dette kan resultere i, at en ny strategi overfor kunden igangsættes.
- Frist udløbet – en hændelse, der oprettes når en frist opsat i forbindelse med en indsats udløber.
- Sporændring – en hændelse, der oprettes når en sagsbehandler manuelt igangsætter en ny inddrivelsesstrategi over for en kunde.

Hændelser kan altså opstå automatisk når relevante informationer vedrørende en kunde ændres, når en tidsfrist opsat af systemet udløber, eller de kan opstå manuelt ved menneskelig interaktion.

2.3.2 Spor

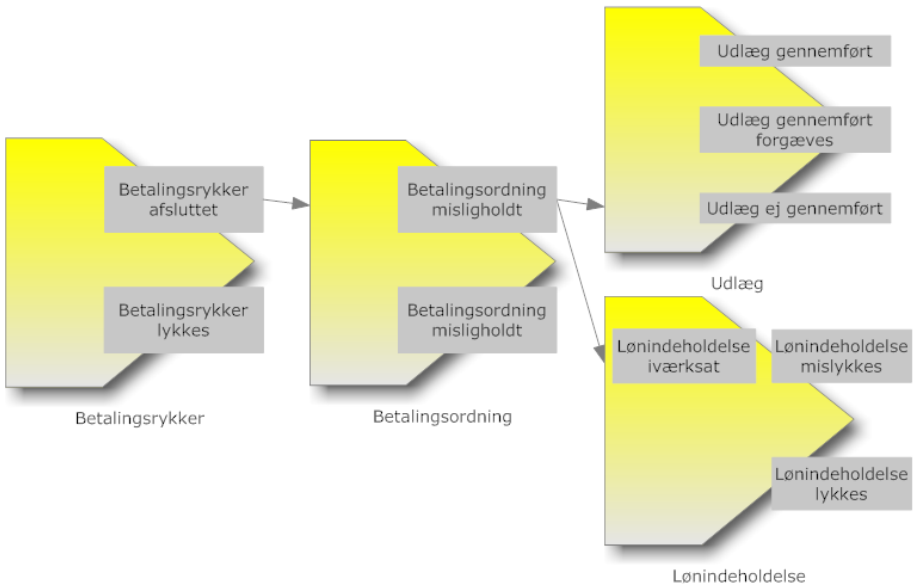
Spor er et SKAT-term, der dækker over deres inddrivelsesstrategier. Sporene anvendes som skabeloner for inddrivelsen af fordringer hos kunderne. Når en fordring er modtaget og kunden er blevet scoret, anbefales der en inddrivelsesstrategi. Når denne inddrivelsesstrategi iværksættes, er det synonym med udtrykket; "Kunden placeres på sporet". En kunde kan kun være på ét spor ad gangen.

Et spor består af en indsatsplan. Denne indsatsplan beskriver hvilke indsatser, der skal anvendes overfor kunden og i hvilket forløb. Indsatserne kan planlægges serielt eller parallelt, afhængig af hvordan SKAT ønsker, at forløbet skal foregå overfor kunden. Det gælder, at det ikke er muligt at have cykliske afhængigheder mellem indsatser i en indsatsplan. Det er ligeledes ikke muligt, at kunden kan være omfattet af flere indsatser af samme indsatstype samtidigt.

Et eksempel på en indsatsplan kan ses i figur 2.2. Notationen der anvendes, er den samme som anvendes internt i SKAT. Indsatsplanen definerer her følgende: Hvis ikke en betalingsrykker fører til betaling af de omfattende fordringer, skal en betalingsordning iværksættes. Misligholdes denne betalingsordning, skal inddrivelsen eskaleres over for kunden, ved at iværksætte udlæg og lønindeholdelse parallelt. Fører betalingsordningen derimod til, at kundens fordring er betalt, vil det betyde, at betalingsordningen slutter uden, at nye indsatser bliver startet. Sporet er dermed nået til ende.

Kunden kan til én hver tid vælge at betale alle sine fordringer. I et sådant tilfælde vil en hændelse om dette, blive modtaget af indsatserne på sporet, og disse vil herefter afslutte. Bemærk at indsatsplanen ikke tager stilling til, hvad der skal ske hvis både udlæg og lønindeholdelse ikke medfører, at alle kundens fordringer er betalt. Her vil det dog altid gælde, at indsatsen Sagsbehandling tilføjes på de spor, der er kørt til ende uden at have inddrevet alle kundens fordringer. Dette lader en sagsbehandler tage en faglig vurdering om hvad der nu skal ske, og dermed bygge videre på indsatsplanen på sporet, eller vælge at anbefale en helt ny inddrivelsesstrategi.

Den anbefalede inddrivelsesstrategi overfor en kunde kan, som sagt, ændres. Dette kan være resultatet af en ny scoring af kunden, eller det kan være en sagsbehandler, der manuelt har anmodet om dette via sagsbehandlerportalen.



Figur 2.2: Eksempel på en indsatsplan, hvor iværksættelsen af indsatser bestemmes af forløbet af de foregående.

Ved en ny anbefalet inddrivelsesstrategi skal der foretages et sporskifte. SKAT har vedtaget, at et sporskifte skal udføres ved først at lukke alle aktive indsatser ned på det gamle spor, hvorefter det nye spor startes op. Hvis en sagsbehandler derimod ligger nye indsatser ind på kundens nuværende inddrivelsesstrategi, vil de eksisterende indsatser fortsætte som før.

Sagsbehandleren modellerer indsatsplanen ved at vælge hvilke indsatsers udgangstilstande, der skal medføre opstart af nye indsatser. Udgangstilstandene bliver nærmere beskrevet i afsnit 2.3.3, men er et udtryk for de indsatstilstande som kan forbindes til andre indsatser i indsatsplanen. Formålet med at definere nogle af indsatstilstandene som udgangstilstande er at begrænse antallet af indsatstilstande, som kan forbindes i indsatsplanen, og dermed gøre det lettere for sagsbehandleren, når indsatsplanen skal planlægges.

Når flere indsatser er forbundet til samme udgangstilstand på en anden indsats, vil disse indsatser altid blive startet parallelt. Omvendt når en indsats venter på flere udgangstilstande, da vil denne indsats starte op når blot en af udgangstilstandene er nået.

I den oprindelige EFI løsning findes der sportyper for både lønmodtagere, selvstændige og virksomheder. I denne prototype indgår alene sportyperne for løn-

modtagere. Disse forklares i afsnit 2.5.

2.3.3 Indsatser

Indsatser er konkrete foranstaltninger, der kan udføres overfor en kunde for at inddrive én eller flere fordringer. En indsats består af tilstande og aktiviteter. Tilstandene beskriver hvor i forløbet indsatsen er nået til, og aktiviteterne skaber fremdrift i indsatsen overfor en kunde. Aktiviteter beskrives nærmere i afsnit 2.3.4. Indsatser reagerer på hændelser, og definerer ved hvilke hændelser en aktivitet skal starte. Det vil altid gælde, at kun én aktivitet, i hver indsats, starter ved modtagelse af en hændelse. Den startede aktivitet vil efter udførsel altid føre til et tilstandsskifte i indsatsen, der startede den. Hvilken tilstand der skiftes til, afhænger af aktivitetens slutresultat.

Sammenkoblingen af tilstand, aktivitet og hændelse i indsatsen gør, at indsatsen reagerer på ændringer vedrørende kunden, baseret på den tilstand indsatsen befinder sig i, på det givne tidspunkt hændelsen modtages.

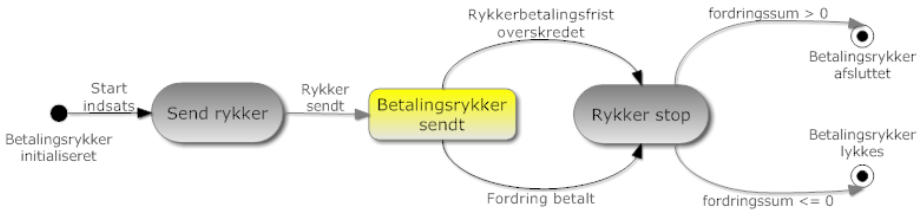
Nogle af tilstandene i en indsats vil være udgangstilstande. Udgangstilstande er tilstande i indsatsen hvorfra der kan forbindes til andre indsatser. Ordet udgangstilstand indikerer en anelse, at det er en tilstand der kun opnås i indsatsen ved afslutning. Dette er dog ikke tilfældet. En udgangstilstand kan sagtens opnås i en igangværende indsats. For at skelne mellem de udgangstilstande, der er resultatet af en afsluttet indsats, og de udgangstilstande, der kan opnås i en igangværende indsats. Er de udgangstilstande, der kan opnås i en igangværende indsats altid afbildet til venstre for de udgangstilstande, der kun opnås i en afsluttet indsats. Eksempler på udgangstilstande for igangværende indsatser kan ses i indsatstyperne Lønindeholdelse og Kreditoplysningsbureau, der beskrives i henholdsvis afsnit 2.6.3 og afsnit 2.6.4. Til trods for denne tvetydighed anvendes ordet udgangstilstand her, da det er et begreb, som anvendes internt i SKAT.

Det er sporet hvorpå indsatsen ligger, der sikrer, at indsatsen starter op ifølge indsatsplanen.

Under inddrivelsen vil en kunde altid være omfattet af mindst én indsats.

Et eksempel på en indsats modelleret vha. den notation, der anvendes internt i SKAT kan ses i figur 2.3. Indsatsen er Betalingsrykker.

Indsatsmodellen illustrerer hvorledes et antal aktiviteter (grå ellipser), hændelser (sorte pile) og tilstande (gule rektangler samt "final state noder") kobles sammen i en indsats. Aktivitetens slutresultat (grå pile) viser sammenhængen



Figur 2.3: Indsatsmodellen for indsatsen Betalingsrykker

mellem aktivitetens slutresultat og hvilken tilstand, der skiftes til efter udførsel af aktiviteten.

Eksemplet for betalingsrykker viser, at der som det første sendes en rykker til kunden når indsatsen starter. Herefter venter indsatsen i tilstanden "Betalingsrykker sendt" indtil der modtages en hændelse om, at enten betalingsfristen er overskredet, fordringen er betalt eller, at indsatsen skal stoppes.

Eksemplet indeholder alle tre former for hændelser; "Rykkerbetalingsfrist overskredet" er en systemgenereret hændelse styret af tid. "Fordring betalt" er en hændelse modtaget af et udefrakommende system, mens "Stop indsats" er en hændelse genereret af en menneskelig aktør. I dette tilfælde en sagsbehandler.

I den oprindelige EFI løsning findes der 12 indsatser. De indsatser som vil indgå i prototypen er; Betalingsrykker, Betalingsordning, Lønindeholdelse, Kreditoplysningsbureau, Udlæg, Henstand og Manuel sagsbehandling.

Disse bliver nærmere beskrevet i afsnit 2.6.

2.3.4 Aktiviteter

Aktiviteter er beskrivelsen af de arbejdsindsatser, som udføres i forbindelse med en indsats overfor en kunde. Dvs. at de er tiltag som skaber fremdrift i indsatsen overfor en kunde. Følgende er gældende for en aktivitet:

- Den starter kun når dens startbetingelser er opfyldt.
- Den reagerer ikke på hændelser.
- Den kan ikke afbrydes efter start.
- Den udføres altid indenfor kort tid.

- Den vil altid medføre et tilstandsskifte i indsatsen, der startede den.

At aktiviteter ikke reagerer på hændelser medfører også, at i de tilfælde hvor der f.eks. ønskes en faglig vurdering fra en sagsbehandler, er det ikke tilladt for aktiviteten at vente herpå. I stedet løses dette ved at bryde aktiviteten op i to; Den første kunne f.eks. være: "Book sagsbehandler-opgave." Denne kan så ligge en opgave i en kø. Dernæst afslutter aktiviteten og indsatsen vil vente på en hændelse, eksempelvis: "Sagsbehandler har truffet afgørelse." Herefter kan indsatsen starte samme aktivitet som før, eller en anden, som skal foretage næste handling.

Startbetingelserne eksisterer for at sikre at gældende love og regler overholdes. Aktiviteten udføres kun når alle dens startbetingelserne er opfyldt. Dette gælder uanset om det er sagsbehandleren eller systemet, der forsøger at starte aktiviteten.

Nogle eksempler kunne være at lovgivningen fastslog, at der ikke måtte foretages lønindeholdelse af en kontanthjælpsmodtager. Aktiviteten "Lønindehold Kunde" kunne derfor have som startbetingelse, at kunden ikke måtte være kontanthjælpsmodtager. Det andet kunne være, at SKAT kun ønskede at lønindeholde kunder over en vis beløbsgrænse, så indgår dette også som en del af startbetingelserne. Startbetingelserne er dermed også en understøttelse af den samlede inddrivelsesstrategi.

Når en indsats ved modtagelse af en hændelse forsøger at starte en aktivitet evalueres startbetingelserne. Hvis de ikke er opfyldt, gennemføres aktiviteten ikke. Man kan således konkludere, at indsatsen har ansvaret for hvornår aktiviteten bør starte, mens aktiviteten har ansvar for at validere om den må starte. Som eksempel herpå henvises til eksempel 2.2 angående lønindeholdelse af kontanthjælpsmodtager.

2.2 Eksempel En kunde, der er kontanthjælpsmodtager har ikke betalt sine parkeringsbøder. Kundens fordring indberettes derfor til SKAT. Kunden scores til et givent spor, hvor der udsendes en rykker som det første. Kunden reagerer ikke herpå, hvilket medfører at betalingsrykkeren afslutter uden, at fordringen er betalt. Indsatsplanen for sporet definerer, at lønindeholdelse bør iværksættes overfor kunden. Startbetingelsen i første aktivitet i indsatsen Lønindeholdelse, er dog ikke opfyldt, da denne skal sikre, at kunden ikke er kontanthjælpsmodtager. Indsatsen Lønindeholdelse kan dermed ikke starte op.

Alle aktiviteter slutter med ét resultat. Dette slutresultat vil blive evalueret af indsatsen, der på baggrund af dette skifter tilstand.

2.4 Scenarier

Essensen i SKATs samlede inddrivelsesstrategi er at automatisere så mange rutineprægede arbejdsgange som muligt. Det forventes derfor, at størstedelen af fordringsinddrivelserne vil foregå helt automatisk, altså uden indblanding fra en sagsbehandler. Der vil være tilfælde, hvor kundens forhold ændres i en sådan grad, at den igangværende inddrivelsesstrategi overfor kunden bør ændres til en mere hensigtsmæssig inddrivelsesstrategi. Her forventes det, at der i størstedelen af tilfældene automatisk skiftes til den nye inddrivelsesstrategi.

Da et automatiseret system altid vil basere beslutningerne på et fastlagt regelsæt, i en eller anden form, medfører dette, at begrebet "Skøn under regel" sommetider vil forekomme. Dvs. at SKAT ikke har foretaget det skøn, som de efter forvaltningsloven har pligt til. I disse tilfælde skal det være muligt for en sagsbehandle, at foretage et skøn af kundens forhold og derefter planlægge en inddrivelsesstrategi, der passer hertil.

Af ovenstående kan der udledes tre mulige scenarier;

- Fuldautomatisk – En inddrivelsesstrategi vælges automatisk og fuldføres til ende.
- Fuldautomatisk med ændring – En inddrivelesstrategi vælges automatisk. Herefter ændres kundens forhold en eller flere gange, hvorefter en ny og mere hensigtsmæssig inddrivelsesstrategi vælges og iværksættes automatisk. Denne fortsætter indtil hele fordringen er inddrevet.
- Sagsbehandler manuelt – En sagsbehandler bestemmer strategien.

Ovenstående tre scenarier vil blive forklaret nærmere i de følgende tre underafsnit.

De tre scenarier ønskes opfyldt i prototypen, og vil blive anvendt som acceptancetest af løsningen.

2.4.1 Fuldautomatisk inddrivelse

Dette scenarie er et eksempel på en kundesag, der behandles uden indgriben fra sagsbehandler.

Jens er lønmodtager og lever hvert år i sommermånederne, i sit sommerhus. Der er 110 km. mellem sommerhuset og til Jens' arbejde. Jens har derfor i sin forskudsopgørelse under befordring anført, at han 90 dage om året kører 110 km. til sit arbejde. Dette har udløst et befordringsfradrag på 13.851 kr.

Befordringsfradraget skal dog udregnes ud fra den sædvanlige bopæl, også selvom man i perioder bor et andet sted. Jens' sædvanlige bopæl ligger under 24 km. fra arbejdet og han er derfor ikke berettiget til befordringsfradraget. Hans restance udgør derfor:

Befordringsfradrag til Københavns kommune 13.851 kr.

Jens har betalingsevne, men der ligger oplysninger hos DMI om, at han tidligere har misligholdt betalingsordninger.

Jens bliver derfor scoret til Spor 5 – "Modspiller med betalingsevne, lille restance". Sporet er illustreret i figur 2.10.

Når sporet iværksættes vil indsatsen Betalingsrykker starte op, og Jens vil efter kort tid modtage en rykker, hvori det anføres, at fordringen nu er til inddrivelse hos SKAT.

Efter 14 dage er fristen for rykkeren overskredet og Jens har endnu ikke henvendt sig eller betalt sit udestående. Indsatsen Lønindeholdelse starter derfor op og der udsendes et varsel herom.

Ved varslets udløb er der stadig ingen reaktion fra Jens, og lønindeholdelse iværksættes. Når lønindeholdelse er iværksat sendes et varsel til Jens med en frist om henvendelse indenfor 4 uger, ellers vil han blive indberettet til et kreditoplysningsbureau.

Jens retter ikke henvendelse og han bliver derfor indberettet. Lønindeholdesen fortsætter uændret til restancen er betalt, og Jens vil herefter blive slettet fra kreditoplysningsbureauet.

2.4.2 Fuldautomatisk ændret inddrivelse

Dette scenarie er et eksempel på en kundesag, der behandles uden indgriben fra sagsbehandler, men hvor strategien ændres undervejs.

Klaus er uddannet og arbejder som murer. Han er ansat i et mindre murerfirma, hvilket vil sige, at han er almindelig lønmodtager. Klaus har nu afdraget på en

tidligere skattegæld på 17.000 kr. i 14 måneder, hvor han betaler 500 kr. om måneden. Hans nuværende restance er derfor på 10.000 kr. Dengang fordringen blev modtaget blev han sporet til Spor 3 – "Medspiller med betalingsevne, lille restance". Sporet er illustreret i figur 2.8.

Da sporet dengang blev iværksat, reagerede Klaus ikke på den udsendte betalingsrykker. Han reagerede dog på det udsendte forslag om betalingsordning, hvor han skulle afdrage de 17.000 kr. af 34 rater, hvilket vil sige 500 kr. om måneden. Han har lige siden indbetalt de 500 kr. hver måned.

Inden Klaus når at betale 15. rate af hans nuværende restance, modtages der endnu en fordring. Denne er på 22.000 kr. og skyldes, at Klaus har glemt at ændre sin selvangivelse. Hans samlede restance er nu på 32.000 kr, hvilket resulterer i, at han denne gang bliver scoret til Spor 4 – "Medspiller med betalingsevne, stor restance". Sporet er illustreret i figur 2.9.

Når sporet iværksættes vil en betalingsrykker med den nuværende restance blive udsendt. Reagerer Klaus ikke på denne, vil det blive forsøgt at gennemføre udlæg samtidig med, at en ny betalingsordning iværksættes. Hvis udlægget bliver gennemført, men det resulterende beløb af udlægget ikke fører til betaling af restancen, vil Klaus blive indberettet til et kreditoplysningsbureau. Ligeledes vil en lønindeholdelse blive iværksat, hvis Klaus misligholder betalingsordningen.

Klaus reagerer dog allerede på betalingsrykkeren, hvor han vælger at betale hele restancen, hvilket betyder at inddrivelsen er færdiggjort.

2.4.3 Sagsbehandler bestemmer inddrivelsesstrategien

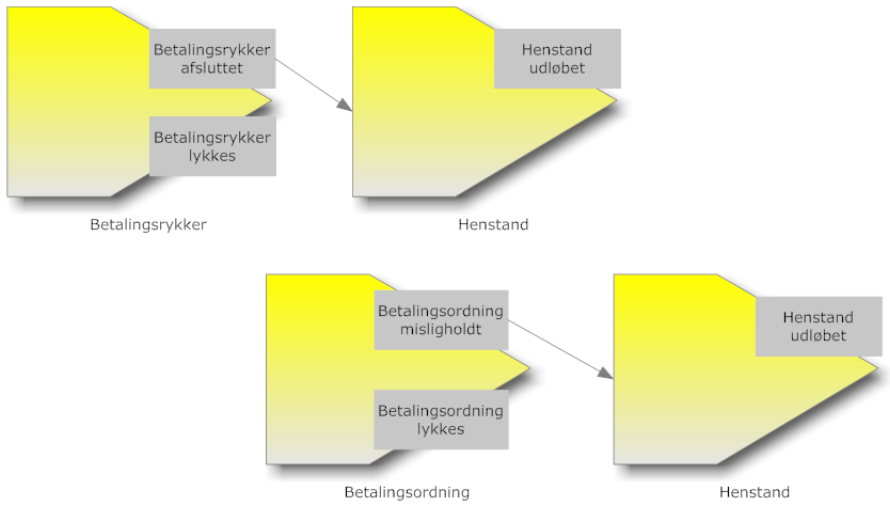
Dette scenarie er et eksempel på en kundesag, der behandles med indgriben fra sagsbehandler.

Lone er pensionist og har fået boligsikring beregnet og udbetalt på et forkert grundlag. Hun skal derfor tilbagebetale for meget udbetalt boligsikring. Hendes restance udgør:

Boligsikring til Herlev Kommune 3.000 kr.

Lone har ingen betalingsevne. Hun har ligeledes ingen aktiver som bankindestående, fast ejendom eller bil.

Lone bliver derfor scoret til Spor 1 – "Ingen betalingsevne, lille restance". Sporet er illustreret i figur 2.6.



Figur 2.4: Spor 1 – Ingen betalingsevne, lille restance (Betalingsordning og Henstand er tilføjet manuelt)

Når sporet iværksættes vil indsatsen Betalingsrykker starte op, og Lone vil efter kort tid modtage en rykker, hvori det anføres, at fordringen nu er til inddrivelse hos SKAT. Hun informeres samtidig om, at hun kan vælge at indgå en frivillig betalingsordning, idet hun finder dette muligt. I modsat fald vil hun få bevilget henstand, da hendes årlige indkomst ligger under grænsen for inddrivelse.

Efter 14 dage er fristen for rykkeren overskredet og Lone har endnu ikke henvendt sig eller betalt sit udestående. Derfor starter indsatsen Henstand op, hvor der er bevilget henstand til Lone i 1 år.

Efter 2 måneder henvender Lone sig. Hun vil gerne have gælden ud af verden og indvilger derfor frivilligt i at indgå en betalingsordning, hvor hun afdrager 200 kr. om måneden.

Sagsbehandleren trækker derfor indsatsen Betalingsordning ind på sporet og opretter betalingsordningen. Betalingsordningen efterfølges af en ny Henstand indsats, idet Lone ikke har en reel betalingsevne.

Sporet er nu manuelt ændret af sagsbehandleren og ser ud som vist i figur 2.4.

Betalingsordningen fortsætter uændret til restancen er betalt.

De to nye indsatser Betalingsordning og Henstand har ingen relation til de gamle

indsatser Betalingsrykker og Henstand udover, at de er på samme spor.

Betalingsordning og den eventuelle efterfølgende Henstand er, som tidligere nævnt, oprettet af en sagsbehandler. SKATs notation for modellering af spor, illustrerer dog ikke denne detalje, og figur 2.4 illustrerer derfor egentligt, at indsatserne Betalingsrykker og Betalingsordning er startet parallelt, idet det heller ikke er muligt at se den aktuelle tilstand for en indsats. Dette er dog ikke tilfældet her, da betalingsordningen er tilføjet senere end betalingsrykkeren.

2.5 Sportyper

Det er muligt for en sagsbehandler selv, at modellere et spor for en kunde helt fra bunden af, dvs. at oprette indsatsplanen ved selv at tilføje indsatser og skabe forbindelser herimellem. Det vil dog oftest være tilfældet, at der automatisk foretages en score af kunden og ud fra denne, anbefales en inddrivelsesstrategi. Når en inddrivelsesstrategi anbefales, er det i virkeligheden et udtryk for, at en eksisterende skabelon for hvordan sporet skal forløbe, anvendes. Denne sporskabelon er synonym med en sportype. En sportype beskriver derfor ligeledes hvilke indsatser, der skal iværksættes over for kunden og i hvilken rækkefølge.

I den oprindelige EFI-løsning findes der sportyper for både lønmodtagere, selvstændige og virksomheder. I denne prototype indgår alene sportyperne for lønmodtagere.

Sportyperne for lønmodtagere er defineret ud fra et scoringstræ. Dette scoringstræ er illustreret i figur 2.5.

Som det ses i scoringstræet findes der seks forskellige spor. Der skelnes ikke mellem medspiller og modspiller, når kunden ingen betalingsevne har, da det for inddrivelsen er uden betydning om kunden vil eller ej, når evnen ikke er til det.

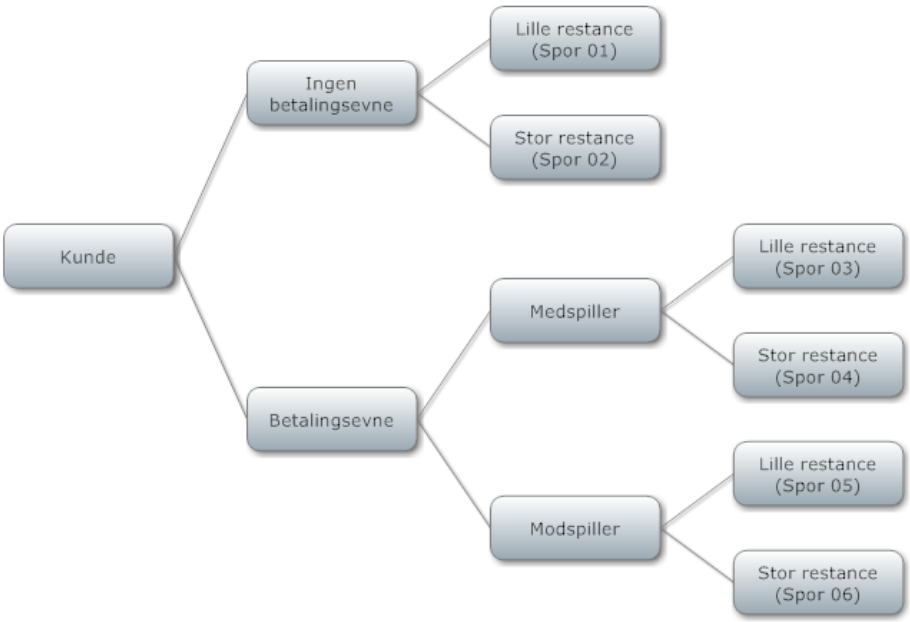
Sporene er navngivet nedenfor:

Spor 1 Ingen betalingsevne, lille restance.

Spor 2 Ingen betalingsevne, stor restance.

Spor 3 Medspiller med betalingsevne, lille restance.

Spor 4 Medspiller med betalingsevne, stor restance.



Figur 2.5: Sportype scoringstræ

Spor 5 Modspiller med betalingsevne, lille restance.

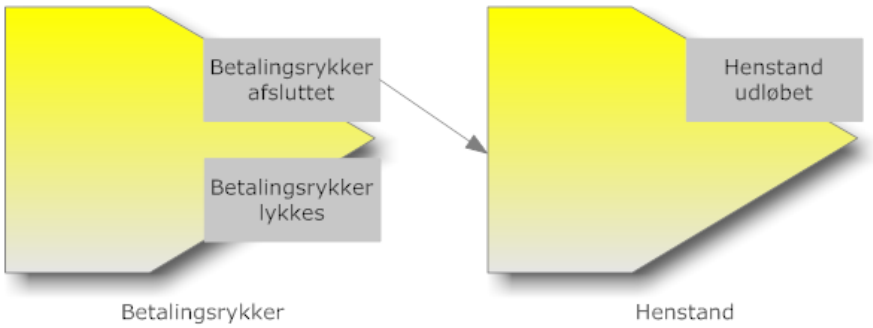
Spor 6 Modspiller med betalingsevne, stor restance.

Hvert enkelt spor bliver nærmere beskrevet i de følgende underafsnit.

2.5.1 Spor 1 - Ingen betalingsevne, lille restance

Denne inddrivelsesstrategi anvendes til kunden, der vurderes til ingen betalingsevne at have. Samtidig er summen af kundens samlede fordringer ikke større end, at det tillades at vente på ændringer i kundens forhold, der muliggør, at fordringerne kan betales.

Der sendes som det første en betalingsrykker ud til kunden. Hvis kunden ikke reagerer på denne, bevilges der automatisk henstand til kunden. Når henstandsperioden udløber vil det være op til en sagsbehandler at vurdere den fremtidige inddrivelsesstrategi overfor kunden.



Figur 2.6: Spor 1 – Ingen betalingsevne, lille restance.

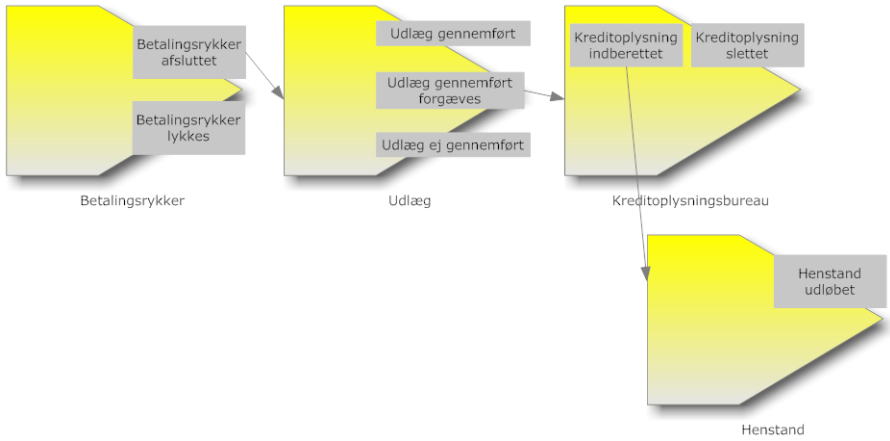
I figur 2.6 illustreres modelleringen af sporet som det vises i sagsbehandlerportalen.

2.5.2 Spor 2 - Ingen betalingsevne, stor restance

Denne inddrivelsesstrategi anvendes til kunden, der vurderes til ingen betalingsevne at have. Samtidig er summen af kundens samlede fordringer af en størrelse, hvor det vurderes, at der med det samme skal gøres noget aktivt for at inddrive fordringerne.

Der sendes som det første en betalingsrykker ud til kunden. Hvis kunden ikke reagerer på denne, vil det forsøges at foretage udlæg i kundens aktiver. Det er dog op til sagsbehandleren i hvert enkelt tilfælde at vurdere om en udlægsforretning skal gennemføres eller ej. Gennemføres udlægsforretningen uden, at det fører til betaling af kundens fordringer, vil kunden blive indberettet hos et kreditoplysningsbureau. Dette har til formål, at forebygge yderligere gældstiftelse hos kunden. Når kunden er indberettet til kreditoplysningsbureauet, bevilges der henstand, da der i øjeblikket ikke er mere at komme efter. Når henstandsperioden udløber vil det være op til en sagsbehandler at vurdere den fremtidige inddrivelsesstrategi overfor kunden.

I figur 2.7 illustreres modelleringen af sporet som det vises i sagsbehandlerportalen.



Figur 2.7: Spor 2 – Ingen betalingsevne, stor restance

2.5.3 Spor 3 - Medspiller med betalingsevne, lille restance

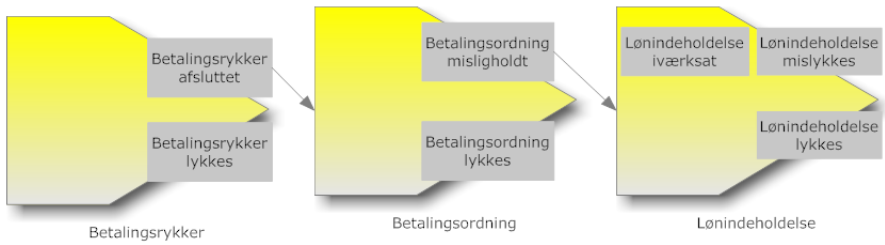
Denne inddrivelsesstrategi anvendes til kunden, der vurderes til at have viljen samt evnen til at betale sine fordringer. Samtidig er summen af kundens samlede fordringer ikke større end hvad, der kan inddrives på almindelig vis.

Der sendes som det første en betalingsrykker ud til kunden. Hvis kunden ikke reagerer på denne, iværksættes en betalingsordning, der er beregnet ud fra kundens aktuelle betalingsevne. Det forventes at betalingsordningen vil føre til, at kundens fordringer bliver betalt. Er dette ikke tilfældet vil en lønindeholdelse blive iværksat. Denne forhøjer den trækprocent som anvendes til at udregne kundens A-skat og anvender det indeholdte beløb til at afdrage kundens fordringer.

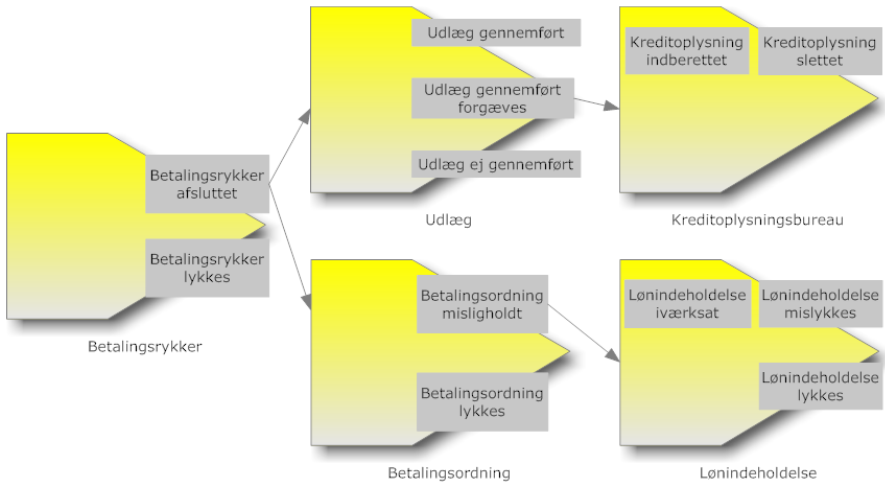
I figur 2.8 illustreres modelleringen af sporet som det vises i sagsbehandlerportalen.

2.5.4 Spor 4 - Medspiller med betalingsevne, stor restance

Denne inddrivelsesstrategi anvendes til kunden, der vurderes til at have viljen samt evnen til at betale sine fordringer. Samtidig er summen af kundens samlede fordringer af en størrelse, hvor det vurderes, at en afdragelsesordning alene ikke er tilstrækkelig.



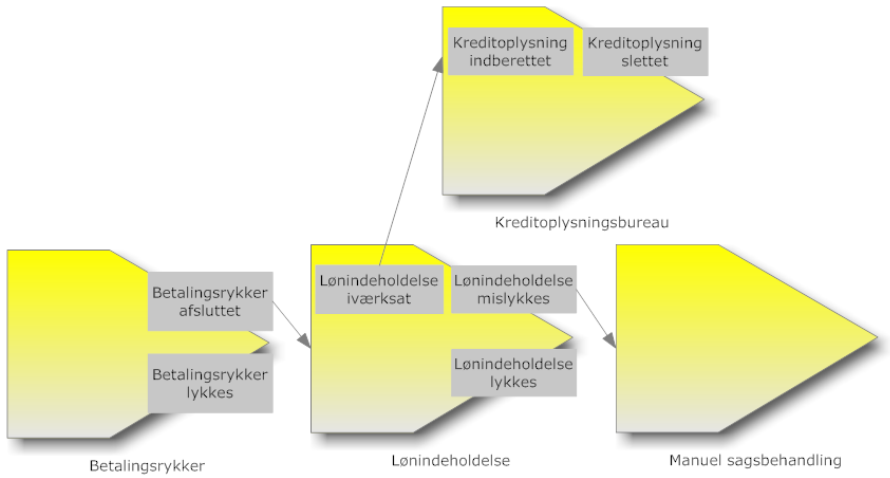
Figur 2.8: Spor 3 – Medspiller med betalingsevne, lille restance



Figur 2.9: Spor 4 – Medspiller med betalingsevne, stor restance

Der sendes som det første en betalingsrykker ud til kunden. Hvis kunden ikke reagerer på denne iværksættes en betalingsordning samt udlæg parallelt. Foretages udlægsforretningen forgæves vil kunden blive indberettet hos et kreditoplysningsbureau. Ligeledes iværksættes lønindeholdelse, hvis betalingsordningen misligholdes.

I figur 2.9 illustreres modelleringen af sporet som det vises i sagsbehandlerportalen.



Figur 2.10: Spor 5 – Modspiller med betalingsevne, lille restance

2.5.5 Spor 5 - Modspiller med betalingsevne, lille restance

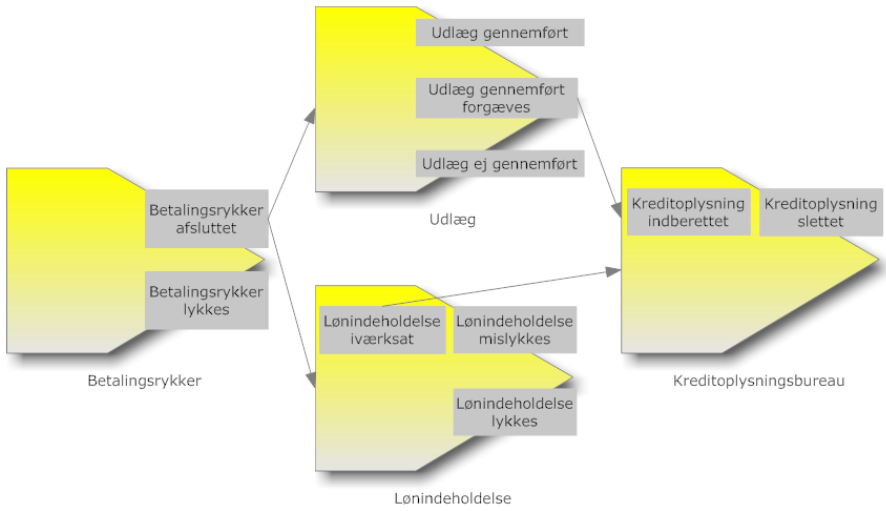
Denne inddrivelsesstrategi anvendes til kunden, der vurderes til at have evnen, men ikke viljen til at betale sine fordringer. Samtidig er summen af kundens samlede fordringer ikke større end, at en afdragelsesordning alene er tilstrækkelig. Fordringerne bliver dog afdraget i form af lønindeholdelse, da det ikke forventes, at kunden frivilligt indvilger i at betale.

Der sendes som det første en betalingsrykker ud til kunden. Hvis kunden ikke reagerer på denne iværksættes lønindeholdelsen. Når denne er iværksat indberettes kunden til et kreditoplysningsbureau. Mislykkes lønindeholdelsen følges sagen op af en sagsbehandler.

I figur 2.10 illustreres modelleringen af sporet som det vises i sagsbehandlerportalen.

2.5.6 Spor 6 - Modspiller med betalingsevne, stor restance

Denne inddrivelsesstrategi anvendes til kunden, der vurderes til at have evnen, men ikke viljen til at betale sine fordringer. Samtidig er summen af kundens samlede fordringer af en størrelse, hvor det vurderes, at en afdragelsesordning alene ikke er tilstrækkelig. Derfor foretages der lønindeholdelse og udlæg parallelt. Både ved iværksættelse af lønindeholdelse, samt ved et forgæves gennemført



Figur 2.11: Spor 6 – Modspiller med betalingsevne, stor restance

udlæg, vil kunden blive indberettet til et kreditoplysningsbureau. Derfra slettes indberetningen først, når indsatsen Kreditoplysningsbureau modtager en hændelse om, at fordringerne er betalt.

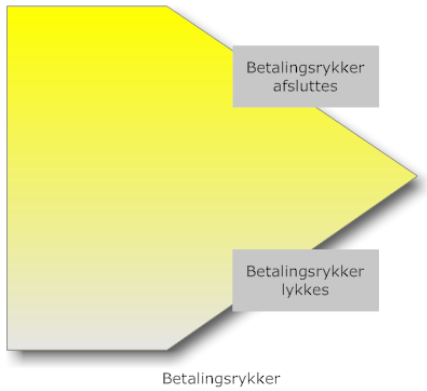
I figur 2.11 illustreres modelleringen af sporet som det vises i sagsbehandlerportalen.

2.6 Indsatstyper

En indsatstype er en skabelon til en specifik foranstaltning. Den beskriver således hvilke aktiviteter, der skal startes overfor kunden og i hvilke tilfælde.

Det gælder, at kunden ikke kan være omfattet af flere indsatser af samme indsatstype samtidigt. Dette bliver dog håndteret i sporafviklingen.

I den oprindelige EFI-løsning findes der 12 indsatstyper. De indsatstyper som vil indgå i prototypen er; Betalingsrykker, Betalingsordning, Lønindeholdelse, Kreditoplysningsbureau, Udlæg, Henstand og Manuel sagsbehandling. Nogle af indsatstyperne vil være simplificeret her.



Figur 2.12: Indsatsen Betalingsrykker med udgangstilstandene; ”Betalingsrykker afsluttet” og "Betalingsrykker lykkes"

2.6.1 Betalingsrykker

Indsatsen Betalingsrykker sender en rykker til kunden og overvåger rykkerfristen. Indsatsen afsluttes senest når rykkerfristen er overskredet. Herefter fortsætter sporet til næste indsats.

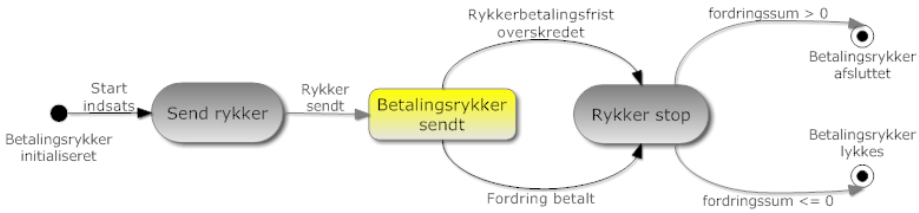
I figur 2.12 illustreres hvordan indsatsen Betalingsrykker vises i sagsbehandlerportalen.

Betalingsrykker har følgende udgangstilstande, som kan forbindes til næste indsats i et spor;

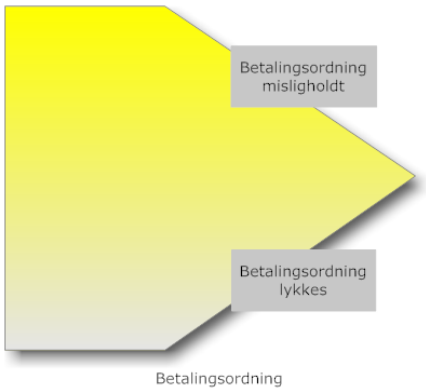
- Betalingsrykker afsluttet
- Betalingsrykker lykkes

Når indsatsen afslutter i tilstanden ”Betalingsrykker lykkes”, vil kunden have afdraget alle sine fordringer omfattet af betalingsrykkeren. Lykkes det ikke at indkræve fordringerne i indsatsen, inden fristen for rykkeren udløber, vil indsatsen afslutte i tilstanden ”Betalingsrykker afsluttet”.

Betalingsrykker reagerer ikke på nye fordringer eller ændringer i kundens betalingsevne, når den først er startet. Indsatsmodellen for Betalingsrykker er vist i figur 2.13.



Figur 2.13: Indsatsmodel for indsatsen Betalingsrykker



Figur 2.14: Indsatsen Betalingsordning med udgangstilstandene; "Betalingsordning misligholdt" og "Betalingsordning lykkes"

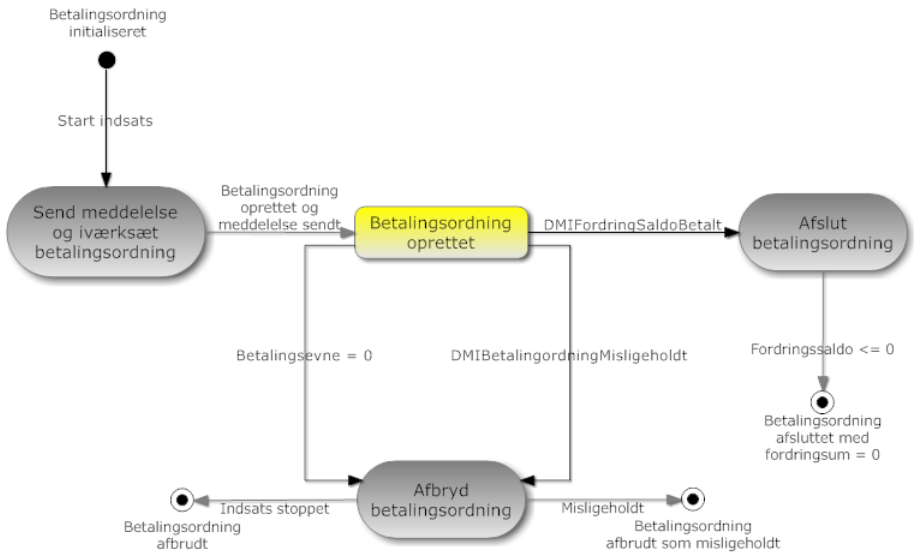
2.6.2 Betalingsordning

Indsatsen betalingsordning, er en aftale med kunden om at afdrage fordringen fordelt over et antal rater.

Når indsatsen Betalingsordning starter, sendes en meddelelse herom til kunden, sammen med en plan over betalingsordningen. Denne plan laves ikke af EFI selv, men af en eksisterende komponent i SKATs infrastruktur, kaldet DMI. DMI er en forkortelse af DebitorMotor til Inddrivelse. DMI kan udregne en plan for tilbagebetaling af fordringerne og sørger selv for månedligt at indkræve raten. Sker det at betalingsordningen ifølge DMI bliver misligholdt, vil DMI udsende en hændelse herom.

I figur 2.14 illustreres hvordan indsatsen Betalingsordning vises i sagsbehandlerportalen.

Betalingsordning har følgende udgangstilstande, som kan forbindes til næste



Figur 2.15: Indsatsmodel for indsatsen Betalingsordning

indsats i et spor;

- Betalingsordning misligholdt
- Betalingsordning lykkes

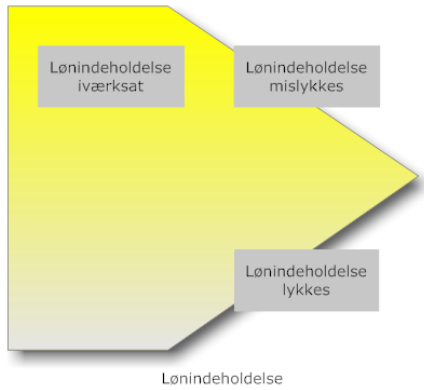
Når indsatsen afslutter i tilstanden "Betalingsordning lykkes", da vil kunden have afdraget alle sine fordringer omfattet af betalingsordningen med succes. Lykkes det ikke at indkræve fordringerne i indsatsen, vil denne afslutte i tilstanden "Betalingsordning misligholdt".

Betalingsordningen reagerer på ændringer i kundens betalingsevne, men vil ikke reagere på nye fordringer, når den først er startet.

Indsatsmodellen for betalingsordning er vist i figur 2.15.

2.6.3 Lønindeholdelse

Indsatsen Lønindeholdelse forhøjer den trækprocent som anvendes til at udregne kundens A-skat og anvender det indeholdte beløb til at afdrage kundens fordringer.



Figur 2.16: Indsatsen Lønindeholdelse med udgangstilstandene; "Lønindeholdelse iværksat", "Lønindeholdelse mislykkes" og "Lønindeholdelse lykkes". Udgangstilstanden "Lønindeholdelse iværksat" kan opnås inden indsatsen afslutter og er derfor afbildet til venstre for de to andre udgangstilstande, der først kan opnås når indsatsen er afsluttet.

Indsatsen overvåger kundens betalingsevne og igangsætter automatisk lønindeholdelse, når der er betalingsevne. Lønindeholdelsesprocenten reguleres automatisk, hvis kundens betalingsevne varigt ændres, ligesom den kan sættes til 0 i en periode, hvis kunden mister betalingsevnen. Når kunden har betalingsevne varsles der om lønindeholdelse med en frist til at betale inden lønindeholdelsen iværksættes.

I figur 2.16 illustreres hvordan indsatsen Lønindeholdelse vises i sagsbehandlerportalen.

Lønindeholdelsen har følgende udgangstilstande som kan forbindes til næste indsats i et spor;

- Lønindeholdelse iværksat
- Lønindeholdelse mislykkes
- Lønindeholdelse lykkes

Indsatsen har udgangstilstanden "Lønindeholdelse iværksat" for at sikre, at sporet kan sætte indsatser i gang, der venter på, at indsatsen Lønindeholdelse starter op, men ikke behøver vente på, at den afslutter.

Når indsatsen afslutter i tilstanden "Lønindeholdelse lykkes", da vil kunden have betalt alle sine fordringer omfattet af lønindeholdelsen med succes. Lykkes det ikke at indkræve fordringerne i indsatsen, vil denne afslutte i tilstanden "Lønindeholdelse mislykkes".

Lønindeholdelsen reagerer på ændringer i kundens betalingsevne, men vil ikke reagere på nye fordringer, når den først er startet.

Indsatsmodellen for lønindeholdelse er vist i figur 2.17.

2.6.4 Kreditoplysningsbureau

Indsatsen Kreditoplysningsbureau anvendes når SKAT ønsker at indberette en kunde som dårlig betaler. Varslet om indberetning kan alene få kunden til at betale sine fordringer, hvor selve indberetningen forebygger, at kunden stifter yderligere gæld.

Når indsatsen Kreditoplysningsbureau starter, sendes der et varsel til kunden herom. Indsatsen vil herefter overvåge om kunden betaler inden fristen. Sker dette ikke, vil indsatsen indberette kunden til et kreditoplysningsbureau, f.eks. Experians RKI register.

I figur 2.18 illustreres hvordan indsatsen Kreditoplysningsbureau vises i sagsbehandlerportalen.

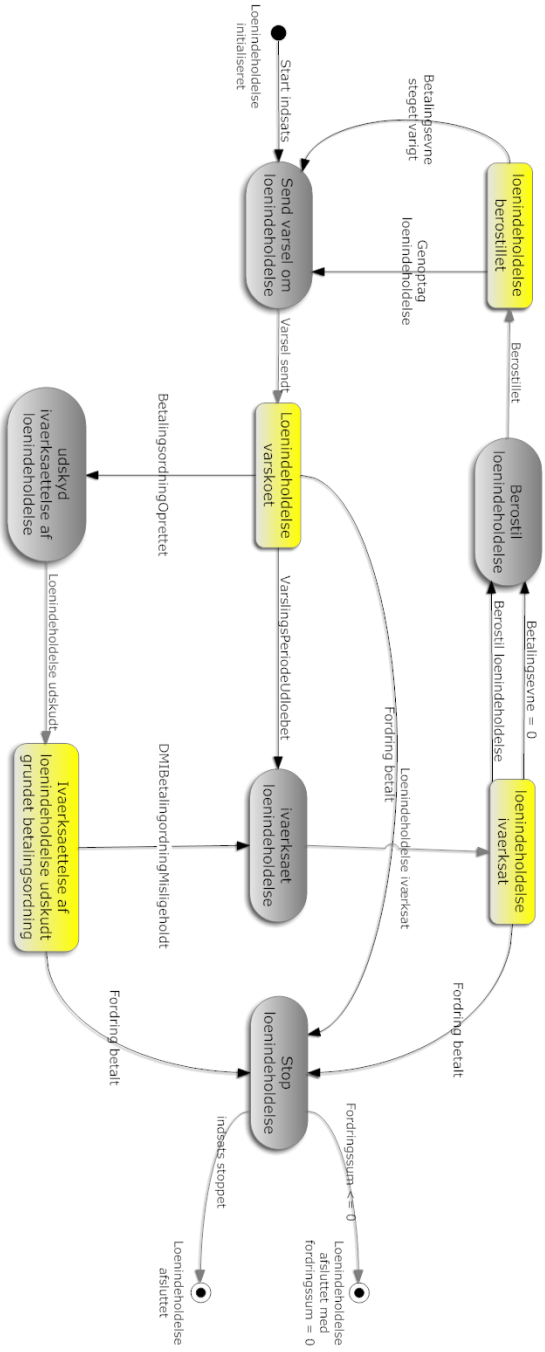
Indsatsen Kreditoplysningsbureau har følgende udgangstilstande som kan forbindes til næste indsats i et spor;

- Kreditoplysning indberettet
- Kreditoplysning slettet

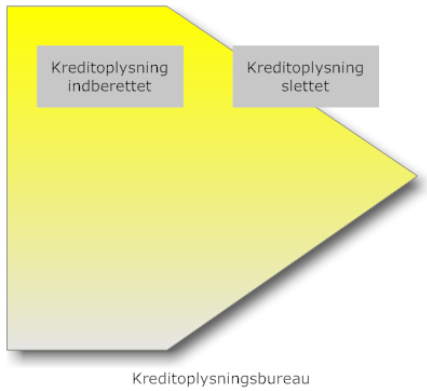
Indsatsen har udgangstilstanden "Kreditoplysning indberettet" for at sikre, at sporet kan sætte indsatser i gang, som venter på, at indsatsen Kreditoplysningsbureau starter op, men ikke behøver vente på, at den afslutter. Et eksempel på dette blev set i figur 2.7.

Indsatsen afslutter først idet en hændelse om, at kunden har betalt sine fordringer modtages. Herefter vil kunden blive slettet fra kreditoplysningsbureauet. Indsatsen afslutter altid i tilstanden "Kreditoplysning slettet".

Indsatsmodellen for kreditoplysningsbureau er vist i figur 2.19.



Figur 2.17: Indsatsmodel for indsatsen Lønindeholdelse



Figur 2.18: Indsatsen Kreditoplysningsbureau med udgangstilstandene; "Kreditoplysning indberettet" og "Kreditoplysning slettet". Udgangstilstanden "Kreditoplysning indberettet" kan opnås inden indsatsen afslutter og er derfor afbildet til venstre for udgangstilstanden "Kreditoplysning slettet", der først kan opnås når indsatsen er afsluttet.

2.6.5 Udlæg

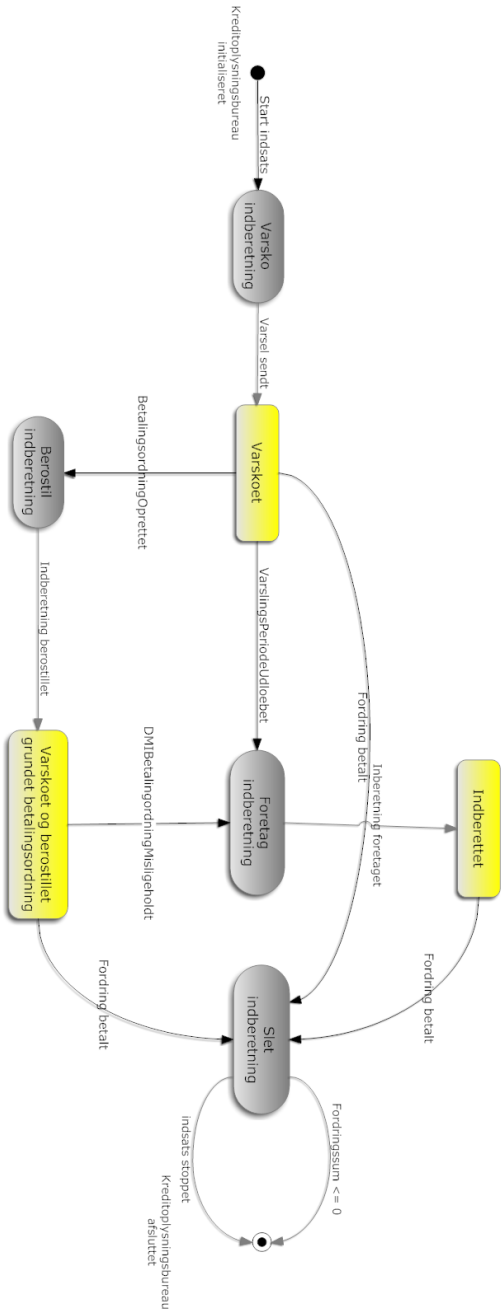
Indsatsen Udlæg anvendes når SKAT ønsker at foretage udlæg i kundens aktiver, og sælge disse aktiver, så kundens fordringer kan betales. Det kan også være rede penge indestående på en konto, der foretages udlæg i.

Når indsatsen Udlæg starter, bookes der en sagsbehandler til at foretage udlægsforretningen. Sagsbehandleren udfører opgaven ved at planlægge møde mellem pantefoged, kunde og sagsbehandler selv. Når sagsbehandleren i samarbejde med pantefogeden har fundet frem til hvilket beløb udlægget mod kunden vil afstedkomme, da vil sagsbehandleren indtaste dette i EFI.

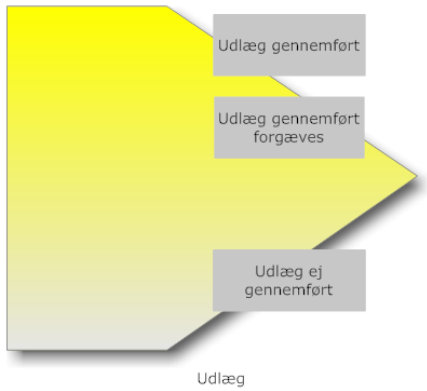
I figur 2.20 illustreres hvordan indsatsen Udlæg vises i sagsbehandlerportalen.

Indsatsen Udlæg har følgende udgangstilstande som kan forbindes til næste indsats i et spor;

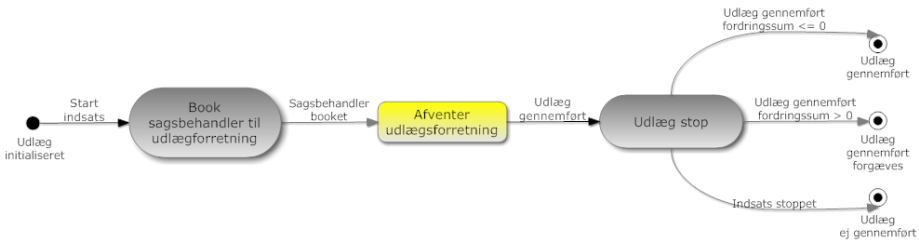
- Udlæg gennemført
- Udlæg gennemført forgæves
- Udlæg ej gennemført



Figur 2.19: Indsatsmodel for indsatsen Kreditoplysningsbureau



Figur 2.20: Indsatsen Udlæg med udgangstilstandene; "Udlæg gennemført", "Udlæg gennemført forgæves" og "Udlæg ej gennemført"

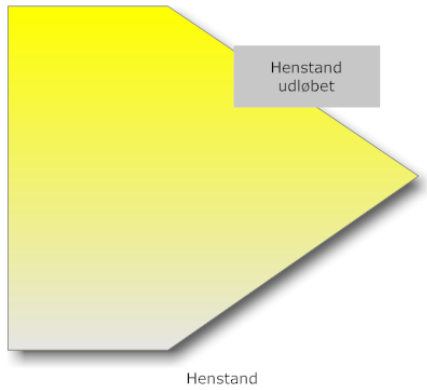


Figur 2.21: Indsatsmodel for indsatsen Udlæg

Når indsatsen afslutter i tilstanden "Udlæg gennemført"", da vil udlægget i kundens aktiver have udløst et beløb stort nok til at betale kundens fordringer. Afslutter indsatsen derimod i tilstanden "Udlæg gennemført forgæves", da kunne beløbet ikke dække kundens fordringer. Indsatsen afslutter i tilstanden "Udlæg ej gennemført" i de tilfælde, hvor sagsbehandleren tager en beslutning om, at der ikke skal foretages udlæg i kundens aktiver.

Udlæg reagerer ikke på nye fordringer eller ændringer i kundens betalingsevne, når den først er startet.

Indsatsmodellen for Udlæg er vist i figur 2.21.



Figur 2.22: Indsatsen Henstand med udgangstilstanden; "Henstand udløbet"

2.6.6 Henstand

Indsatsen Henstand anvendes af sagsbehandleren, når en kunde af sociale eller andre forhold, ikke kan betale sine fordringer. Henstand gives i en periode og indsatsen afslutter automatisk når perioden udløber.

I figur 2.22 illustreres hvordan indsatsen Henstand vises i sagsbehandlerportalen.

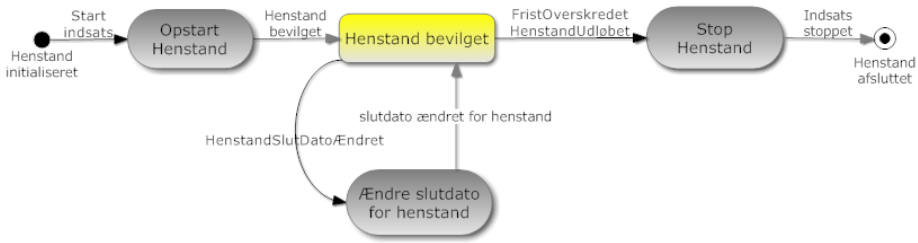
Indsatsen Henstand har følgende udgangstilstande som kan forbindes til næste indsats i et spor;

- Henstand udløbet

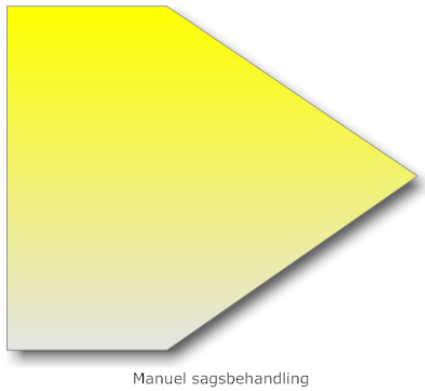
Indsatsen afslutter altid i tilstanden "Henstand udløbet". Herefter vil det oftest være op til en sagsbehandler at lave en vurdering om hvad der nu skal ske.

Indsatsen Henstand reagerer ikke på nye fordringer eller ændringer i kundens betalingsevne, når den først er startet. Dette skyldes, at henstand er bevilget af ikke økonomiske hensyn.

Indsatsmodellen for henstand er vist i figur 2.23.



Figur 2.23: Indsatsmodel for indsatsen Henstand



Figur 2.24: Indsatsen Manuel sagsbehandling.

2.6.7 Manuel sagsbehandling

Manuel sagsbehandling anvendes når der ønskes en sagsbehandlers vurdering af den videre inddrivelsesstrategi overfor kunden.

Indsatsen booker en sagsbehandler til at vurdere den aktuelle kundesag. Sagsbehandleren udfører opgaven ved at manipulere kundens spor. Dette kan ske ved at indsætte en ny indsats, f.eks. Henstand.

I figur 2.24 illustreres hvordan indsatsen Manuel sagsbehandling vises i sagsbehandlerportalen.

Indsatsen har ingen udgangstilstand. Dette skyldes at formålet med Manuel sagsbehandling er, at få en sagsbehandler til at foretage en faglig vurdering om hvad der skal ske videre i forløbet, og derefter manipulere sporet herefter. Indsatsen Manuel sagsbehandling bliver oftest aktuel, når sporet når til ende uden at have medført, at kunden har betalt sit udestående.

Figur 2.25: Indsatsmodel for indsatsen Manuel sagsbehandling

Indsatsen reagerer derfor heller ikke på nye fordringer eller ændringer i betalingsevnen, da sagsbehandleren altid vil foretage en samlet vurdering af kunden når inddrivelsesstrategien fastlægges.

Indsatsmodellen for Manuel sagsbehandling er vist i figur 2.25.

Kapitel 3

Software analyse

For at kunne udarbejde en løsning, der understøtter det forretningsmæssige koncept, foretages der en analyse med henblik på softwareudviklingen. Denne analyse skal tydeliggøre de nævnte begreber og deres relation til hinanden. En domænemodel vil derfor blive udarbejdet. Der vil også blive taget stilling til hvilke koncepter, der med fordel kan implementeres med en regelmotor, samt hvilke former for datasikkerhed, der er nødvendigt.

3.1 Domænemodel

Formålet med domænemodellen er at analysere de forretningsmæssige koncepter og nå frem til en afklaring af de forskellige begrebers karakteristika og indbyrdes relationer til hinanden.

Det forretningsmæssige koncept foreskriver, at der kun reageres på hændelser, for at mindske rutineprægede tjek af kundeforhold. Hver hændelse vedrører en specifik kunde. Denne kunde har en fordring at betale, samt en betalingsevne og -vilje. Betalingsevnen og -viljen er henholdsvis udtryk for, om kunden kan og vil betale sin fordring. Disse vil danne grundlag for en scoring af kunden. Scoringen af kunden omfatter, at der anbefales en sportype til kunden som denne

skal behandles efter. Sportypen er en skabelon for den valgte inddrivelsesstrategi. Den definerer hvilke indsatser der skal iværksættes over for kunden, og i hvilken rækkefølge. Skabelonen er generel for alle kunder, hvor samme sportype anbefales.

Det er muligt for en sagsbehandler i hver enkel kundesag, at modificere hvordan inddrivelsesstrategien skal forløbe overfor kunden. Med andre ord er det muligt at ændre sporet. Derfor er det vigtigt, at skelne mellem sportype og spor. Sportypen er, som tidligere beskrevet, blot en skabelon for hvordan inddrivelsesstrategien bør forløbe, hvor sporet er en konkret instans, der er unik for hver enkel kunde. Kundens aktuelle spor kan derfor ændres uden indflydelse på andre kundesager.

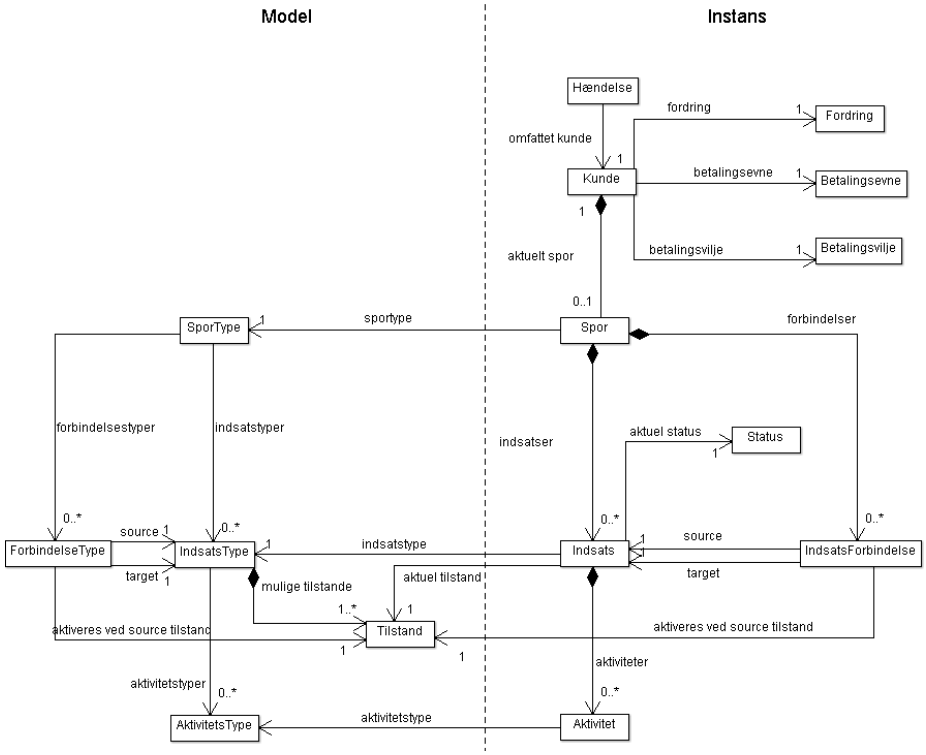
Sporet består af en række indsatser samt eventuelle forbindelser imellem disse. Forbindelserne udspringer fra en indsats' tilstand og ender ved en anden indsats' tilstand.

Figur 3.1 illustrerer domænemodellen.

I domænemodellen ses der på modelsiden de konceptuelle begreber. Disse er Sportype, Indsatstype samt Aktivitetstype. Disse er henholdsvis skabeloner for hvordan Spor, Indsatser og Aktiviteter skal udføres. De er generelle og gælder for alle kunder. Når der foretages en scoring af en kunde og en anbefalet inddrivelsesstrategi vælges, vil den tilsvarende sportype kopieres til et spor, der er specifik for kunden. Dette er en nødvendighed, idet det skal være muligt for sagsbehandleren at ændre spor vedrørende en enkel kunde uden, at det har indflydelse på de andre kunder i systemet.

På instanssiden er det derfor konkrete instanser, der drejer sig om en specifik kundesag under køretid. De holder konkret information omkring kunden, og hvor langt kunden er nået i forløbet.

Da et spor som sagt kan ændres af en sagsbehandler, gælder det for sportyper, at disse vil blive kopieret til en konkret instans af et spor til kunden ved anvendelse. Dette er dog ikke tilfældet for indsatstyper, idet det ikke er muligt, for en sagsbehandler, at ændre i disse. Derfor vil der blot oprettes en instans af en indsats, der beskriver hvilken tilstand denne indsats er i. Indsatstypen beskriver således hvordan forløbet skal udføres, hvor indsatsen beskriver hvor i indsatsforløbet den givne kunde er. For at beskrive indsatsens livscyklus, har instansen af indsatsen også en status. Denne beskriver således om indsatsen er startet, igangværende eller afsluttet. Denne status skal ikke forveksles med tilstanden, som beskriver mere detaljeret hvor i forløbet en igangværende indsats er, mht. indsatstypen.



Figur 3.1: Domænemodel for SKAT EFI

I afsnit 3.1.1 vil det blive nærmere beskrevet hvorfor modellen for spor, er blevet som den er.

3.1.1 Spormodellering

SKAT har udtrykt et ønske om, at det skal være så nemt som muligt for en sagsbehandler at modellere sporene via sagsbehandlerportalen. Det ønskes, at sporene skal kunne modelleres ved blot at forbinde en af udgangstilstandene fra indsats A, til indsats B. Hvis indsats A når i den tilstand som indsats B ventede på, da er det tilladt at starte indsats B.

Det er beskrevet i afsnit 2.3.2, hvordan en indsats der afslutter i en af dens udgangstilstande vil resultere i, at to indsatser herefter opstarter parallelt, hvis de begge var forbundet til denne udgangstilstand.

Det er endvidere beskrevet hvornår en indsats, der afventer flere udgangstilstande vil starte op. Dette vil ske når blot den ene af de to udgangstilstande, som afventes, er opnået. Det er således ikke begge tilstande, der behøver at være opnået for, at opstart vil ske.

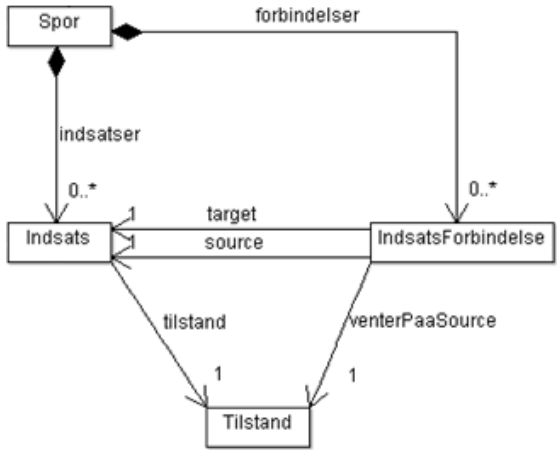
Et eksempel på dette kan ses i figur 2.11, der illustrerer spor 6. Her vil Betalingsrykker resultere i, at både Udlæg og Lønindeholdelse starter, når denne afslutter. Kreditoplysningsbureau vil ligeledes starte op, idet lønindeholdelsen iværksættes eller udlæg er forsøgt gennemført, men forgæves.

Domænemodellen for spormodellering, i figur 3.2, vil derfor være tilstrækkeligt til at dække ovenstående scenarier.

I domænemodellen for spormodellering ses det, at sporet består af et antal indsatser, samt forbindelser herimellem. Forbindelsen tænkes aktiveret når tilstanden i indsatsen, hvor forbindelsen udspringer fra (Source), er den samme som den tilstand forbindelsen venter på. Når en forbindelse er aktiv, kan indsatsen den ender ved (Target) starte.

3.2 Forretningsregler

Alle regler, der har en forretningsmæssig værdi skal evalueres i en regelmotor. Dermed kvalificeres hele modelleringen af den automatiserede inddrivelse til



Figur 3.2: Domænemodel for spormodellering

dette. De involverede koncepter er regler til scoring af kunde, samt regler til spor-, indsats- og aktivitetsafvikling. Alle forretningsregler vil følge formatet: "When something happens and if something is true, do something."Oversat til dansk: "Hvis der sker noget og noget er sandt, så gør noget."Reglerne skal selvfølgelig kende formen af de informationer, de bliver givet. Beskrivelsen af denne form kaldes i Drools-terminologi en ”Fact Model”. Denne ”Fact Model” er blot endnu et udtryk for domænemodellen. Udtrykket har til formål at understrege, at domænemodellen danner grundlag for de fakta, der evalueres i reglerne. For at sikre en vis orden i reglerne i takt med, at disse skalerer, opdeles reglerne logisk i funktionsområder:

- Scoringsregler – regler der ud fra kundens fordring, betalingsevne samt -vilje definerer hvilken sportype kunden skal behandles efter.
- Sporregler – regler der styrer afviklingen af spor.
- Indsatsregler – regler der styrer afviklingen af indsatser.
- Aktivitetsregler – regler der angiver om en aktivitet må udføres.

Reglerne vil blive navngivet således, at det ud fra navnet alene, er muligt at identificere reglens logiske funktionsområde, idet dette gør det lettere at få et hurtigt overblik.

3.3 Datapersistering

Informationer der relaterer til en kunde, skal under hele inddrivelsesprocessen persisteres, og skal til en hver tid kunne afspejle kundens aktuelle tilstand.

SKAT anslår, at EFI går i drift med 600.000 kunder og at antallet af transaktioner vil være mindre end 40 per sekund. En modtaget hændelse persisteres i sin egen transaktion, og de aktiviteter, der måtte blive udført som resultat af hændelsen udføres hver i deres egen transaktion. Antallet af hændelser per kunde er svagt stigende, hvilket vil øge antallet af transaktioner med tiden. Derfor skal det sikres, at den valgte løsning til at persistere data er skalerbar i takt med, at antallet af kunder samt hændelser, der omhandler disse kunder vokser.

For at understøtte, at flere kan arbejde med dataene samtidig, skal der indføres samtidighedskontrol. Det anslås, at to sagsbehandlere sjældent arbejder på samme kundesag samtidigt. Desuden anslås det, at det sjældent sker, at en sagsbehandler arbejder på en kundesag i samme øjeblik som systemet gør. Dette vil betyde, at der ikke er mange transaktioner, der skal afbrydes/rulles tilbage. Endvidere vil administrationstiden når en sagsbehandler eksempelvis ændrer en kundes spor, betyde forholdsvis lange transaktioner.

Når ovenstående tages i betragtning, skønnes en optimistisk tilgang til samtidighedskontrol at være den optimale løsning, til at sikre korrekte resultater ved samtidige operationer. Desuden skønnes det, at performance samt skalerbarhed vil være højere end ved en pessimistisk tilgang, da det ikke er nødvendigt at låse de dataressourcer som transaktionerne påvirker. Da ingen ressourcer bliver låst forhindres deadlocks også.

I tilfælde af, at en aktivitet forsøges udført i en transaktion og denne fejler, da vil alle ændringer som blev foretaget i transaktionen blive rullet tilbage. Herefter vil en sagsbehandleropgave blive booket vedrørende den kunde, aktiviteten omhandlede. Sagsbehandleren har så mulighed for at kontrollere om de ønskede informationer vedrørende kundens sag er til rådighed. Aktiviteten vil således ikke automatisk blive forsøgt udført igen. Det er altså en sagsbehandlers opgave at forsøge igen.

3.4 Sagsbehandlerportal

Der er et behov for en portal, hvor igennem en sagsbehandler kan administrere kundesager. Det skal være muligt for sagsbehandleren at udføre følgende opga-

ver:

- Oprette kunder
- Ændre informationer for eksisterende kunder
- Ændre sporet for eksisterende kunder
- Se historik over kundesager
- Ændre eksisterende sportyper

Det skal endvidere være muligt, at genere hændelser, der normalt ville komme fra eksterne systemer. Dette er for prototypens skyld, idet det gør det muligt at simulere hændelser udefra. Dette gør det lettere at demonstrere prototypen.

Kapitel 4

Teknologier

Dette projekt vil blive implementeret i Java. En af fordelene ved udvikling i Java, er de mange tredjepartsbiblioteker, der kan anvendes, da de oftest fører til en højere produktivitet i udviklingen.

De tredjepartsbiblioteker, som anvendes her er; Maven, Spring, Hibernate, JUnit, Mockito, Quartz og som det mest væsentlige Drools. Hvad disse tredjepartsbiblioteker er og hvilke ansvarsområder de hver især har, vil blive beskrevet i de følgende underafsnit.

Det er en fordel at have en idé om de funktionaliteter som tredjepartsbibliotekerne, der anvendes her, giver. Dette skyldes, at de kan have indflydelse på valget af design. Især Drools og Hibernate har indflydelse på valg af design i dette projekt. De resterede bliver relevante i implementeringen.

Af vedligeholdsmæssige årsager er der her opstillet nogle få, men væsentlige krav, som disse tredjepartsbiblioteker har måttet opfylde. Som det første har de skulle have en høj grad af udbredelse i industrien, da det afspejler en vis form for kvalitet af biblioteket. For det andet har det været væsentligt, at de er frigivet under Open-Source licenser, da dette mindsker afhængigheden til andre selskaber.

4.1 Drools

Drools er en Open Source platform til integration af regelbaseret forretningslogik, og består af følgende dele;

Drools Guvnor Business Rule Management System, BRMS, der hjælper med at skrive og holde styr på regler, samt versioner af disse.

Drools Expert Regelmotor der implementerer og udvider Charles Forgy's Rete algoritme.

Drools Flow Business Process Management System, BPMS, der hjælper med at modellere og holde styr på processer, samt versioner af disse. Drools Flow anvender Business Process Model and Notation standarden 2.0, BPMN 2.0, til at beskrive processerne.

Drools Fusion Event Stream Processing, ESP, framework med support for Complex Event Processing, CEP.

Drools Planner Framework der håndterer NP hårde problemer, hvad angår planlægning, og kan eksempelvis anvendes ved knapsack lignende problemer.

De Frameworks, der vil blive anvendt her er især Drools Expert og Drools Fusion.

Drools Flow vil ikke blive anvendt her, til trods for, at det umiddelbart synes at være det ideelle værktøj til netop denne problemstilling. Drools Flow er designet til at modellere statiske processer, hvor de samme trin forventes at skulle udføres hver gang. Dette er ikke tilfældet her, idet sagsbehandleren, i hver enkelt kundesag har frihed til at vælge i hvilken rækkefølge indsatserne på sporet skal afvikles.

4.1.1 Drools Expert

Drools Expert er en implementeret regelmotor, og er derfor ideel til at håndtere komplekse regler. Drools Expert giver mulighed for at beskrive hvad, der skal gøres og ikke hvordan det skal gøres. Forstået på den måde, at der blot beskrives hvad, der på det givne tidspunkt skal være opfyldt for, at noget skal ske. Drools Expert tager sig af opgaven at finde ud af konsekvensen, af de fakta der findes på det givne tidspunkt. Regler vil derfor kunne tilføjes, ændres samt fjernes uafhængigt af hinanden, hvilket giver en god fleksibilitet.

```
#
# Kunde uden betalingsevne og lille restance scores til spor 1
#
rule "Kunde Score - Ingen betalingsevne, lille restance"
    when
        $kunde : Kunde( betalingsevne == 0,
                        fordring <= 1000)
    then
        LOG.info("Sportype 'INGEN_BETALINGSEVNE_LILLE_RESTANCE' anbefalet");
        modify( $kunde ) {
            setSenesteScoringsAnbefaling( INGEN_BETALINGSEVNE_LILLE_RESTANCE ),
        }
end
```

Figur 4.1: Eksempel på en regel skrevet i Drools Expert

Drools Expert anvender Rete-algoritmen og det er matematisk bevist, at regler eksekveret vha. Rete-algoritmen er hurtigere end de tilsvarende regler implementeret vha. traditionelle if-else sætninger (kilde; "JBoss Drools Business Rules"af Paul Browne, side 18). Rete-algoritmen har dog den ulempe, at den skal eksekveres i samme tråd. Dette er en væsentlig faktor, der skal tages hensyn til under designet. Desuden kræver det ekstra memory, da Rete-netværket også skal ligge i memory, hvilket også vil give en begrænsning på antallet af samtidige objekter i working memory.

At have forretningslogikken samlet i regelfilerne og selve dataene samlet i databasen, giver en god separation mellem logik og data, hvilket gør det endelige system mere vedligeholdelsesvenligt.

Regelmotoren Drools Expert vil blive anvendt til at evaluere alle forretningsregler i løsningen. De primære er de logiske funktionsområder; Scoring af kunde samt Spor-, Indsats- og Aktivitetsafvikling.

Et eksempel på hvordan en regel skrives, kan ses i figur 4.1. Reglen i eksemplet scorer alle kunder med en betalingsevne der er lig med 0 kr. og hvor deres fordring er mindre eller lig med 1000 kr. til den sportype der anbefales i dette tilfælde. Ønskes det at implementere en lignende regel der anbefaler en anden sportype, kan denne tilføjes uden ændringer i de eksisterende regler. Dette betyder, at det er nemt og fleksibelt at udvide reglerne. Det er således nemt at vedligeholde reglerne, og det vil ikke være en større byrde at implementere nye regler ved lovtilføjelser. Lovændringer kan naturligvis medføre, at eksisterende regler skal ændres.

Det er muligt at skrive reglerne i et domænespecifikt sprog. Oversat til engelsk; Domain Specific Language, DSL.

```
declare KundeUpscoringHaendelse
    @role( event )
end

#
# Kunde uden betalingsevne og lille restance scores til spor 1
#
rule "Kunde Score - Ingen betalingsevne, lille restance"
    when
        KundeUpscoringHaendelse($kunde : kunde) from entry-point "SkatEfiEventListener"
        Kunde( this == $kunde,
               betalingsevne == 0,
               fordring <= 1000)
    then
        LOG.info("Sportype 'INGEN_BETALINGSEVNE_LILLE_RESTANCE' anbefalet");
        modify( $kunde ) {
            setSenesteScoringsAnbefaling( INGEN_BETALINGSEVNE_LILLE_RESTANCE ),
        }
end
```

Figur 4.2: Eksempel på en regel skrevet i Drools Expert og med Drools Fusion ESP

4.1.2 Drools Fusion

Drools Fusion vil blive anvendt til at håndtere hændelser i systemet. Med Drools Fusion ESP vil regelmotoren blive utrolig effektiv, da kun de regler som abonnerer på en bestemt hændelse vil blive aktiveret, når netop denne type hændelse modtages. Hændelser der modtages og som ikke er relevante, på det givne tidspunkt, vil dermed blive ignoreret. Dette gør at regelsættet kan skalere uden væsentlig indflydelse på performance. Desuden giver det mulighed for, at hver enkelt regel ikke skal kende til hændelser, som ikke vedrører denne.

Et eksempel kunne være at reglerne for scoring af en kunde kun skal afvikles, når der modtages en hændelse om netop dette. Her vil eksemplet i figur 4.1 skulle ændres til hvad der illustreres i figur 4.2.

I eksemplet ses det, at reglen fra figur 4.1 har gennemgået en lille ændring til figur 4.2. Ændringen der er foretaget er, at KundeUpscoringHaendelsen er indført og, at det nu kun er kunden som hændelsen omfatter, der vil blive scoret på ny. Dette har den betydning af reglen kun bliver evalueret når en hændelse af typen KundeUpscoringHaendelse bliver modtaget, hvilket er en god optimering.

Det planlægges at anvende Drools Fusion i alle regler, hvor det er muligt. Reglerne for indsatsafvikling vil have stor fordel af Drools Fusion. Hvorimod reglerne for sporafvikling ikke vil kunne drage den store fordel, da sporet skal evalueres hver gang der er sket fremskridt i en indsats. Det sikres dog i Drools Expert, at reglerne kun evalueres når de variabler de tjekker på, ændres. Dette er takket

```
rule "Betalingsrykker ikke betalt inden rykker frist"
    when
        $h : RykkerSendtInternHaendelse()
        $indsats : Indsats( id == $h.indsatsId, tilstand == IndsatsTilstandEnum.BETALINGSRYKKER_SENDT )
        not(FordringBetaltHaendelse(kundeId == $indsats.kundeId, this after [-*, 30d] $h)
            from entry-point "SkatEfiEventListener")
    then
        modify($indsats) {
            setTilstand(IndsatsTilstandEnum.BETALINGSRYKKER_AFSLUTTET);
        }
end
```

Figur 4.3: Illustrerer Drools Fusion syntaks til CEP hvor tid mellem hændelser tages i betragtning

være Rete-algoritmen.

Drools Fusions tilbyder også funktionalitet til Complex Event Processing, CEP. Dette vil som udgangspunkt blive anvendt til at holde styr på de modtagende eller ikke modtagende hændelsers forbindelse til hinanden, samt forbindelse til tiden.

Et eksempel på hvordan CEP kan anvendes kan ses i figur 4.3. Eksemplet definerer en regel der sikrer, at betalingsrykkeren afsluttes, hvis der for 30 dage siden er modtaget en hændelse om, at rykkeren er sendt, men der endnu ikke er modtaget en hændelse om, at fordringen er betalt. Drools Fusion sikrer, at reglen bliver evalueret når de 30 dage er gået. Drools Expert evaluerer datagrundlaget for reglen. Er de ønskede data tilstede, og i den ønskede tilstand, vil reglen blive aktiveret.

4.2 Hibernate

Hibernate er et Open Source Framework, som mapper fra en objektorienteret domænemodel til en traditionel relationel database.

Hibernate håndterer således størstedelen af de problematikker, der ligger i at konvertere Java-objekter til og fra en relationel database. Dette inkluderer også konvertering fra Java-datatyper til SQL-datatyper. Hibernate tilbyder ydermere sit eget SQL lignende sprog kaldet Hibernate Query Language, HQL. Dette tillader at skrive SQL lignende forespørgsler, der matcher mod mappede Java-objekter.

Hibernate tilbyder også forskellig funktionalitet til at sikre datakonsistens. Hvilken der anvendes, tages der stilling til i designet. Hvordan datakonsistens implementeres med funktionalitet fra Hibernate beskrives i afsnit 6.3.

4.3 Spring

Spring Framework er et Open Source Framework, der giver mulighed for en fuldstændig abstraktion og indkapsling af implementeringsspecifikke teknologier. Dette opnås ved transparent at give mulighed for at administrere applikationens komponenter. F.eks. kan en komponent B, som en anden komponent A afhænger af, uden videre erstattes af komponenten C, og komponenten A vil ikke bemærke noget.

Især Spring Frameworkets Inversion-of-Control container, IoC, er nyttig. Herfra vil begrebet Dependency-Injection blive anvendt. Dette betyder at afhængigheder bliver givet i Runtime og, at disse således ikke skal være givet ved compile time. Det er således ikke nødvendigt, at have en central kode, der skal sikre at systemet bliver sat op til at være i en bestemt tilstand ved start, da Spring tager sig af dette.

Dette giver en adskillelse mellem kode og konfiguration, der især er god i forbindelse med tests. Dette skyldes, at konfigurationen kan ændres således, at automatiserede tests kan køre uden, at en applikationsserver er sat i drift. Dette gør det nemmere at sikre en god kvalitet af produktet, idet tests kan køres oftere og hurtigere under udviklingen. Til tests vil Spring primært blive anvendt i integrationstests samt acceptance-tests.

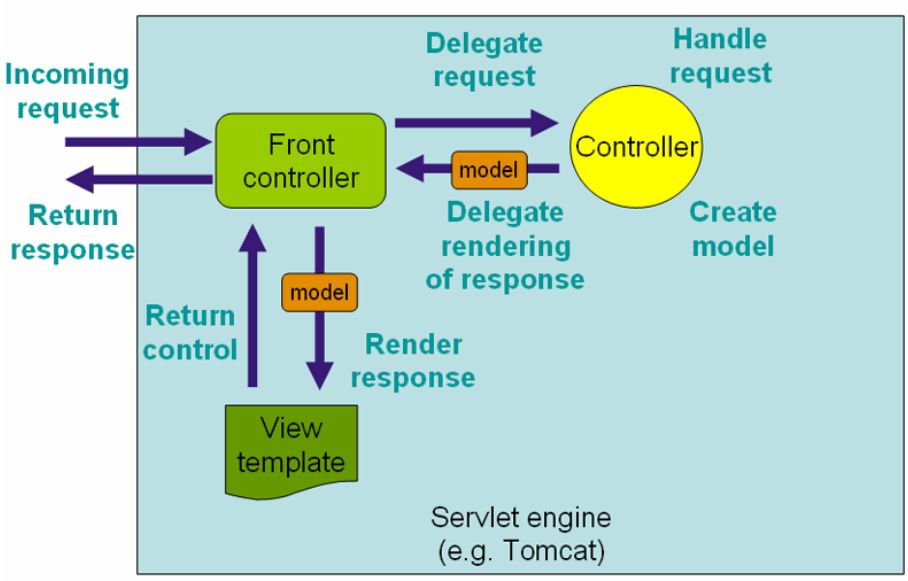
Springs Data Access Framework vil blive anvendt til at integrere Hibernate i Spring. Spring vil automatisk stå for at åbne og lukke forbindelsen til databasen. Til at sikre datakonsistens vil også Springs Transaction Management Framework blive anvendt.

Spring Frameworket er valgt, da dets styrker i forhold til konfiguration, godt modulært design og testfaciliteter er vigtige med henblik på en robust og vedligeholdelsesvenlig løsning.

4.3.1 Spring MVC

Spring MVC er et Framework, der gør det nemt at designe og implementere en webapplikation ud fra designmønstret Model-View-Controller.

Når et request modtages bliver dette uddelegeret til den korrekte Controller. Denne Controller behandler informationerne i requestet og udfører de nødvendige operationer gennem servicelaget, der tager sig af forretningslogikken. Når Controlleren har udført arbejdet, returneres en Model. Denne Model indeholder



Figur 4.4: Workflow for modtagelse og behandling af requests i Spring MVC. Billedet er kopieret herfra: `http://static.springsource.org/spring/docs/2.5.6/reference/mvc.html`

de data, som er nødvendige for at generere den HTML side som klienten vil få vist i sin browser. Til at generere siderne anvendes der her Java Server Pages, JSP, med Jave Standard Tag Library, JSTL. Det er dog også muligt at anvende andre teknologier sammen med Spring MVC til at generere HTML-siderne med.

I figur 4.4 er det grafisk illustreret hvordan Spring MVC behandler et request.

Spring MVC vil her blive anvendt i sagsbehandlerportalen. Spring MVC sikrer, at objekterne i domænemodellen kan anvendes direkte som datagrundlag for de genererede HTML-sider.

4.4 Quartz

Quartz Scheduler er et Open Source Framework, der giver mulighed for, at konfigurere et batchjob til at starte efter et givent interval eller på et givent tidspunkt.

Quartz er valgt, da det integreres nemt i Spring.

4.5 Mockito

Mockito er et Open Source Framework, der gør det muligt at lave en Mock af et objekt. Dvs. at simulere et objekt ved at efterligne den måde det rigtige objekt opfører sig på.

Det er muligt at definere hvad bestemte metoder på Mocken skal returnere, ud fra hvad disse bliver kaldt med. Det er endvidere muligt at se hvor mange gange en metode er kaldt på Mocken, samt med hvilke parametre metoden er kaldt med.

Mockito er valgt da, det er et meget simpelt Framework, der gør det nemt og hurtigt at skrive tests. Det er muligt at integrere Mockito i Spring, men her er det valgt fra, da en adskillelse mellem integrationstest, og test af en enkelt komponent er at foretrække.

Mockito anvendes primært under tests af reglerne, da det er muligt at teste hvilke metoder, der er kaldt på de objekter, der er givet til regelmotoren.

4.6 Maven

Maven er et Open Source værktøj, der anvendes til at styre et projekts afhængighed af andre projekter. Det styrer således det samledes projekts afhængighed af tredjepartsbiblioteker, men også projektets underprojekters interne afhængighed af hinanden.

Maven anvendes også undervejs i udviklingsprocessen til at kvalitetssikre systemet, idet de skrevne JUnit tests bliver eksekveret ved hver enkelt byg af systemet med Maven.

4.7 MySQL

MySQL er en Open Source relationel database, der kan håndtere mange samtidige brugere.

I det oprindelige EFI, anvendes en Oracle database, men det vil være muligt at ændre projektet til at anvende denne også, da der blot kræves simpel Spring konfiguration.

I databasen vil alle kundedata blive persisteret.

4.8 JBoss AS

JBoss Applikations Server, JBoss AS, er en Open Source Java EE baseret applikationsserver. Denne er valgt, da teknologierne Drools og Hibernate også er under udvikling af JBoss og Redhat, og det må derfor formodes, at disse teknologier vil være optimale at anvende her.

I det oprindelige EFI anvendes Oracle Weblogic Application Server. Denne løsning designes dog ikke således, at den er låst til JBoss AS. Det vil være muligt at flytte løsningen til en anden applikationsserver, ved at ændre produktion konfigurationen i Spring.

JBoss AS, vil blive anvendt til at afvikle det endelige system på.

Kapitel 5

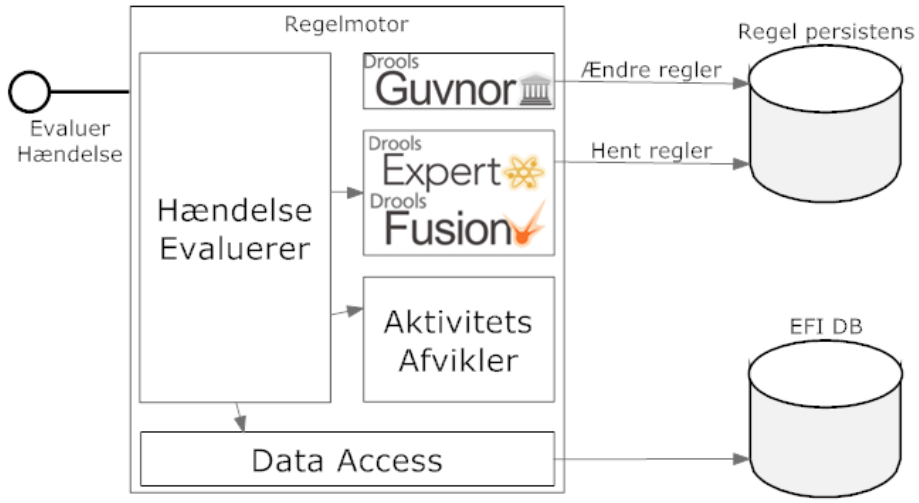
Software design

Da meget af teknologien, og især Drools-platformen var ukendt fra begyndelsen, startede teknologieksperimenter og studier sideløbende med kravafklaringen og analysen. Af dette skulle fremkomme en prototype, der havde til formål at afsløre om der var noget, der skulle tages særligt hensyn til i designfasen. Prototypen afslørede, at regelmotoren Drools Expert ikke rigtigt er designet til at evaluere regler for et stort antal af objekter samtidigt. Den er nærmere designet til at håndtere et stort antal regler over få objekter ad gangen. Derfor er det valgt at designe løsningen, så den anvender Drools Expert på en tilstandsløs måde for hver enkel kundesag. I afsnit 5.1 kan der læses mere om denne udforskning af Drools.

De efterfølgende afsnit omhandler designet af den prototype, der vil resultere i det endelige produkt.

5.1 Første prototype

I den første prototype skulle teknologien i Drools-platformen udforskes. Det blev forsøgt at udnytte alle elementer heri til fulde. Guvnor blev anvendt som editor til at skrive reglerne. CEP-funktionaliteten i Drools Fusion blev forsøgt anvendt



Figur 5.1: Systemdiagram over første prototype

til at overvåge og evaluere tidsbaserede hændelser. Eksempelvis når en rykker er udsendt til en kunde, skal systemet overvåge, at kunden reagerer på denne hændelse inden fristen for rykkeren udløber.

Da formålet med prototypen var at finde eventuelle begrænsninger ved regelmotoren Drools Expert, der skulle tages højde for i designfasen, blev prototypen designet således, at regelmotoren havde ansvar for alt undtagen at persistere data.

Prototypen fungerede således, at når en hændelse blev modtaget kom denne direkte ind i regelmotoren med Drools Fusion ESP. Regelmotoren skulle så selv hente relevant data vedrørende kunden frem. Dette kunne ske ved et opslag på et unikt id for kunden. Det er dog valgt her, at hele kundeobjektet skal være påsat hændelsen ved modtagelse. Alt hvad der blev indsat i regelmotorens working memory blev med Listeners opfanget og persisteret. På denne måde gik ingen hændelser tabt. Herefter udføres de aktuelle operationer vedrørende kunden som hændelsen igangsatte. Resultatet blev efterfølgende persisteret.

Drools Expert er den implementerede regelmotor, og holder reglerne vedrørende scoring af kunde, spor-, indsats- og aktivitetsafvikling.

Figur 5.1 illustrerer strukturen af første prototype i et systemdiagram.

Data persisteres ved at implementere en Listener, der kan indgå i Drools Expert.

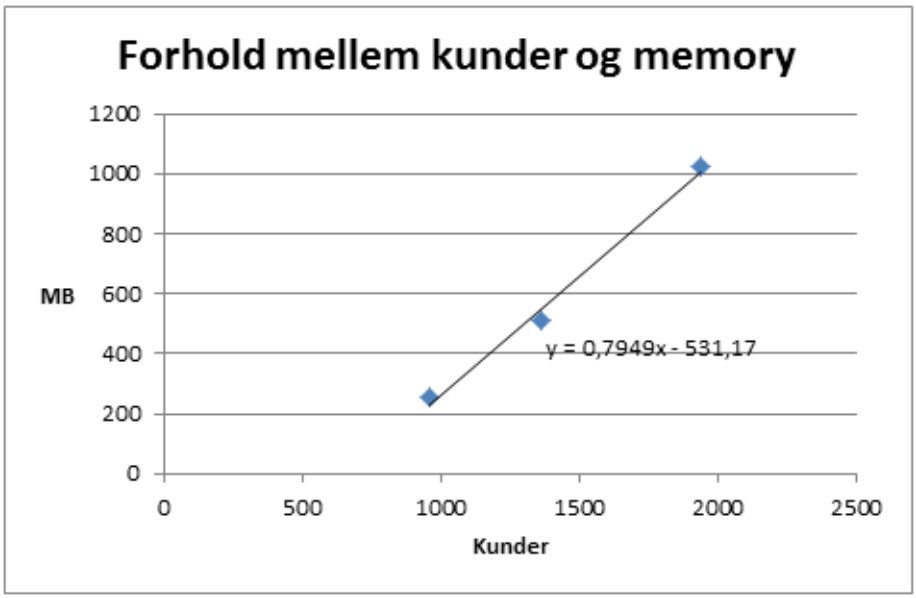
Denne bliver kaldt en WorkingMemoryEventListener. Følgende tre metoder implementers i listeneren; objectInserted(ObjectInsertedEvent event), objectUpdated(ObjectUpdatedEvent event) og objectRetracted(ObjectRetractedEvent event). Hver gang noget indsættes eller opdateres i Working memory formodes det, at dette skal persisteres.

Idéen var, at alle kunder kunne behandles samtidig i regelmotoren. Når en hændelse blev modtaget og nogle kundedata blev relevante, da ville disse data blive hentet og lagt i memory. Når dataene ikke længere var relevante mere ville de blive fjernet fra memory. Det er en nødvendighed at kunne overvåge tidsperioden mellem de modtagende hændelser eller mangel på samme. Med andre ord, hvis en forventet hændelse ikke modtages skal dette også kunne håndteres. Det viste sig dog, at for at anvende Drools Fusions CEP funktionalitet til dette, var det nødvendigt at have alle tidligere hændelser i Working memory, for at kunne sammenligne dem med de nyligt indkomne. Dette giver en væsentlig begrænsning for hvor mange kunder, der kan håndteres af systemet, idet det formodentligt vil kræve meget memory.

Et eksempel på dette er når indsatsen Betalingsrykker udsender en rykker. Hvis ikke indsatsen modtager en hændelse om, at rykkeren er betalt inden for et givet tidsinterval, da skal indsatsen Betalingsrykker afslutte, så ventende indsatser kan starte. Et eksempel på hvordan syntaksen i reglerne så ud da CEP blev anvendt, blev vist i figur 4.3. Her definerede reglen, at hvis der eksisterede en hændelse om, at rykkeren var sendt. Samtidig med, at der indenfor 30 dage endnu ikke var modtaget en hændelse om, at fordringen var betalt, så skulle indsatsen Betalingsrykker afslutte.

En test af løsningen, hvor tidligere hændelser lå i Working memory, blev udført for at se hvor meget memory der kræves i forhold til antallet af kunder. Testen virkede ved løbende at oprette nye kunder samt generere hændelser for eksisterende kunder. For at resultatet af testen er acceptabelt, skal behovet for memory være under 32 GB, når 600.000 kunder håndteres.

I figur 5.2 ses et diagram med de resulterende målinger fra testen. I diagrammet ses en lineær tendens i forholdet mellem antal af kunder og krævet memory. Dvs. der kræves næsten 470 GB. memory af systemet for at kunne håndtere 600.000 kunder. Dette er langt over det acceptable niveau. Desuden må det siges at være dårlig skalerbarhed, da hver enkelt kunde kræver næsten 1 MB af systemet. For hver ettusinde nye kunder, skal der således tilføjes næsten 1 GB memory. Meget af memoryen blev anvendt til Rete-netværket, der var meget kostbart, hvad angår memory. Det blev også bemærket, at det tog tid at oprette Rete-netværket, men herefter blev reglerne til gengæld eksekveret hurtigt. Ved at anvende samme KnowledgeBase bruges der kun tid på at oprette Rete-netværket ved opstart.



Figur 5.2: Forhold mellem antallet af kunder og krævet memory.

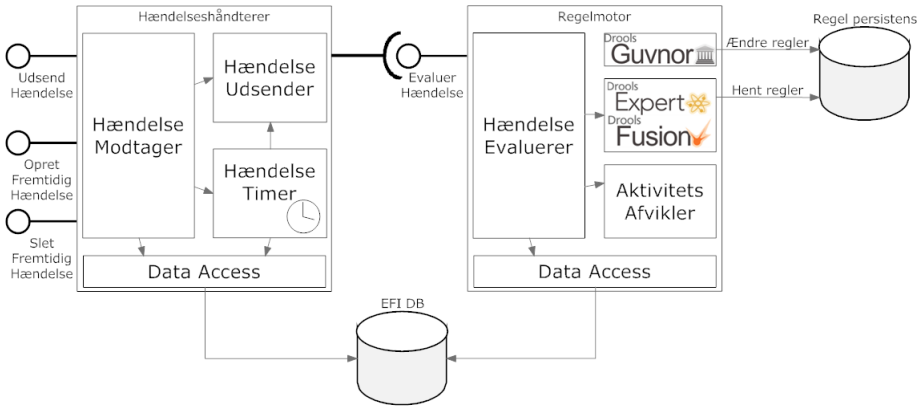
Da det var klart at ikke alle hændelser kunne ligge i Working memory, måtte en løsning med en ekstern håndtering af hændelser udtænkes. Hvordan denne løsning blev, kan der læses om i afsnit 5.2.

5.2 Anden prototype

Anden prototype måtte udbedre de problemer, der var i designet af Første prototype. Problemet var især manglende skalerbarhed, da det var nødvendigt at have alle hændelser i working memory.

Grunden til, at hændelserne var nødsaget til at ligge i working memory, var som tidligere nævn problemet med at overvåge tidsfrister. Løsningen i denne prototype er derfor at indføre en komponent, der skal stå for at håndtere hændelserne. Hændelserne er dermed ikke mere nødsaget til at ligge i regelmotorens memory. Hver hændelse vil sammen med informationen om den vedrørende kunde blive givet videre til regelmotoren ved modtagelse. Regelmotoren har nu alene til ansvar at evaluere hændelsen og returnere resultatet.

Som eksempel vil en indsats, der har brug for at blive overvåget, vedrørende



Figur 5.3: Systemdiagram over Anden prototype.

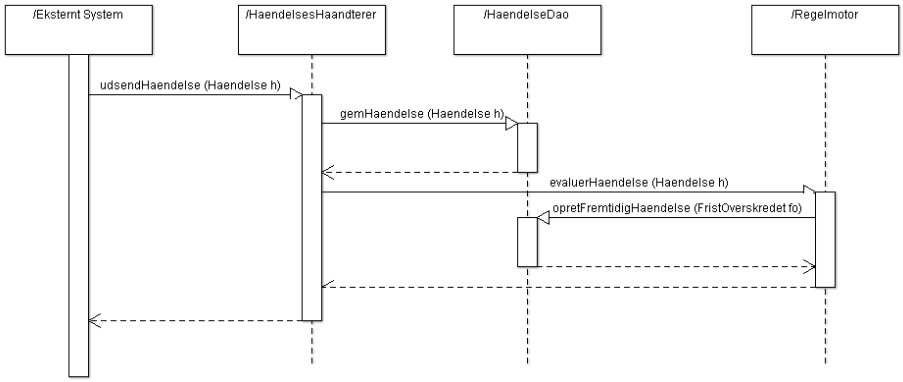
en tidsfrist, selv oprette en fremtidig hændelse. En fremtidig hændelse er en hændelse med et udførelsestidspunkt, der ligger frem i tiden, eksempelvis om 30 dage. Et batchjob har så til ansvar at få disse hændelser udsendt på det ønskede udførelsestidspunkt. Modtages en hændelse vedrørende en kundehenvendelse, vel at mærke inden tidsfristen, altså inden udsendelsen af den oprettede fremtidige hændelse finder sted, vil den oprettede fremtidige hændelse blive ignoreret når den modtages.

Det er muligt at slette hændelser, hvis disse ikke har nået deres udførelsestidspunkt, og derfor endnu ikke er markeret som udført. Hændelser som er udført vil ikke være mulige at slette af hensyn til sporbarhed.

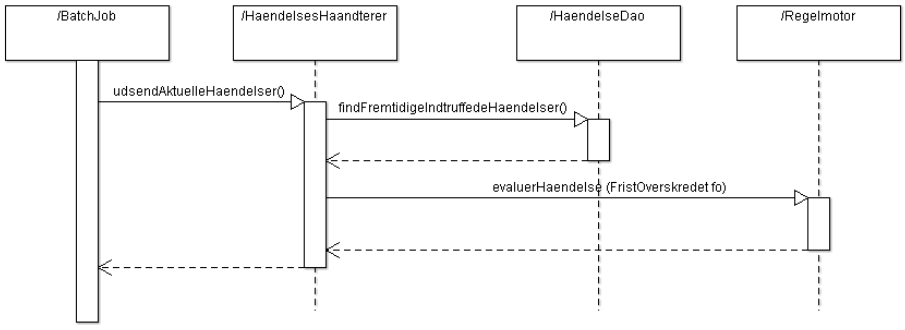
Systemdiagrammet, i figur 5.3, illustrerer strukturen af Anden prototype. Til forskel fra systemdiagrammet over Første prototype vist i figur 5.1, er der nu tilføjet en ekstra komponent. Denne har til ansvar at håndtere hændelser samt at persistere disse. Komponenten kaldes Hændelseshåndteren.

Til trods for, at Drools Fusions CEP funktionalitet ikke bliver anvendt, er Drools Fusion stadig en del af regelmotoren. Dette skyldes, at Drools Fusions ESP funktionalitet stadig vil blive anvendt til at gøre reglerne mere effektive. Hvordan reglerne effektiviseres med ESP, blev beskrevet i afsnit 4.1.2.

I sekvensdiagrammet, i figur 5.4, illustreres et eksempel på hvordan en hændelse, der modtages fra et eksternt system fører til, at en fremtidig hændelse oprettes. Denne fremtidige hændelse udsendes efterfølgende, når udførselstidspunktet er nået. Det er et batchjob der med et fast tidsinterval sikrer, at hændelserne bliver udsendt på det rette tidspunkt. Dette illustreres i sekvensdiagrammet i figur 5.5.



Figur 5.4: Sekvensdiagram der viser håndtering af hændelse modtaget fra et eksternt system.



Figur 5.5: Sekvensdiagram der viser udsendelse af hændelser med batchjob.

Når hændelser modtages af Hændelseshåndtereren, vil Hændelseshåndtereren persistere hændelsen og derefter give denne videre til regelmotoren til evaluering. Regelmotoren vil evaluere hændelsen og foretage eventuelle ændringer på kundeobjektet samt kundens spor og indsatser. I tilfældet i figur 5.4 fører hændelsen til at en fremtidig hændelse oprettes. Den fremtidige hændelse er af typen FristOverskredet. Denne type fremtidig hændelse bliver eksempelvis oprettet, når en betalingsrykker udsendes til en kunde. Indsatsen Betalingsrykker opretter hændelsen med det formål at få besked, når rykkerfristen er udløbet. I dette tilfælde udløber rykkerfristen. Med andre ord, hændelsen om, at rykkerfristen er overskredet udsendes, inden kunden har rettet henvendelse. Besked om at rykkerfristen er overskredet udsendes når batchjobbet udsender de hændelser, der tidligere er oprettet som fremtidige, men nu er aktuelle. Hvordan batchjobbet forløber blev illustreret i figur 5.5.

Regelmotoren samt Hændelseshåndtereren beskrives nærmere i de to følgende afsnit.

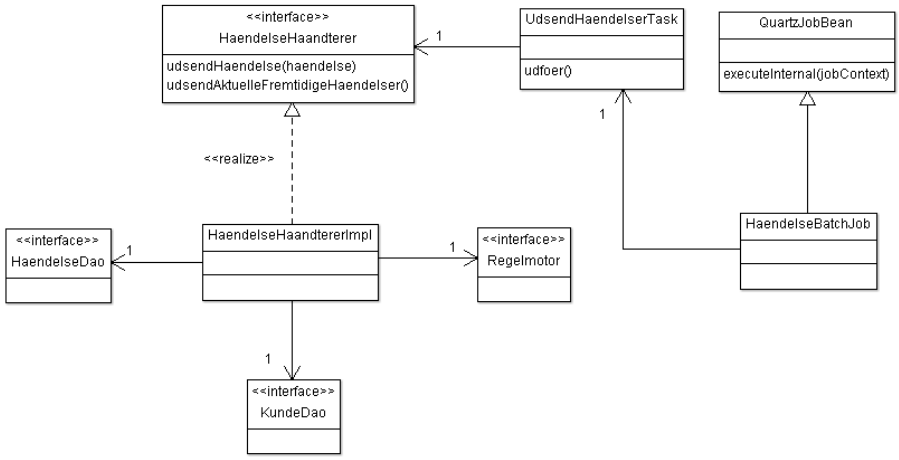
5.2.1 Hændelseshåndtereren

Hændelseshåndtereren kommer til at modtage hændelser fra enten eksterne systemer eller sagsbehandlerportalen. Der skelnes dog ikke mellem disse hændelser. De modtagne hændelser skal persisteres af hensyn til sporbarhed, derefter sendes de til evaluering i regelmotoren. Resultatet som regelmotoren leverer, vil herefter også blive persisteret.

Hændelseshåndtereren kommer desuden til at bestå af et batchjob, der skal sikre, at de fremtidige hændelser, der er oprettet af indsatserne, vil blive udsendt med faste intervaller. Eksempelvis om natten klokken 02.00 på alle hverdage. Frameworket Quartz vil blive anvendt til at skedulere batchjobbet. Quartz blev omtalt i afsnit 4.4.

Klassediagrammet i figur 5.6 viser hvordan hændelseshåndteringsløsningen er struktureret.

HaendelseBatchJob kommer til at nedarve fra QuartzJobBean. Metoden executeInternal skal blot implementeres, da denne vil blive kaldt af Quartz Frameworket, når det er tid til at starte batchjobbet. Batchjobbet vil derefter udføre opgaven at udsende hændelser. Dette sker blot ved at kalde metoden udsendAktuelleFremtidigeHaendelser. Denne metode sikrer, at alle hændelser, der er oprettet som fremtidige hændelser, men nu er aktuelle, vil blive udsendt til evaluering af regelmotoren.



Figur 5.6: Struktur af hændelseshåndteringsløsningen.

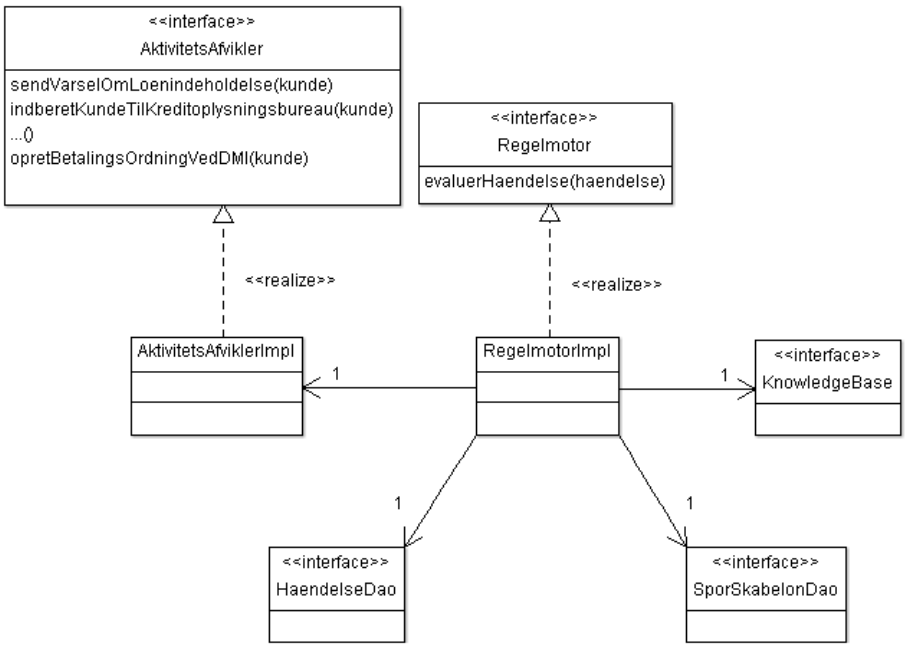
Ved udsendelse af hændelser skelnes der ikke mellem aktuelle og fremtidige hændelser. Regelmotoren vil derfor heller ikke kunne skelne mellem disse.

5.2.2 Regelmotoren

Regelmotoren skal som bekendt implementeres med Drools platformen. Helt nøjagtigt er det Drools Expert samt Drools Fusion, der anvendes. Drools anvendes ved, at der opbygges en central base, hvor alle reglerne har hjemme. Denne base kaldes for KnowledgeBase i Drools' terminologi.

Ud fra en instans af denne KnowledgeBase oprettes en KnowledgeSession. Denne session kan både være stateless eller stateful. Der skal dog anvendes en StatefulKnowledgeSession her, da en StatelessKnowledgeSession ikke kan anvendes i forbindelse med Drools Fusion. StatelessKnowledgeSession er dog blot en wrapper udenom StatefulKnowledgeSession, der sørger for at rydde op efter hver eksekvering af reglerne, ved at fjerne alle indsatte objekter fra working memory. På samme måde skal der ryddes op efter hver eksekvering af reglerne her, så regelmotoren i dette design også ender med at være stateless.

Den instantierede session kan herefter fodres med facts. Facts er et begreb i Drools' terminologi, der blot er et synonym til objekterne i domænemodellen. Når disse facts ligger i Working memory aktiveres reglerne. De regler hvor betingelserne er opfyldt, vil resultere i, at en handling udføres. Alle facts, der er at



Figur 5.7: Struktur af regelmotorløsningen.

finde på instans-siden i domænemodellen, illustreret i figur 3.1, kan være tilstede i Working memory.

Klassediagrammet, i figur 5.7 viser hvordan regelmotoren er struktureret. RegelmotorenImpl har først og fremmest kendskab til KnowledgeBase, der kommer fra Drools Expert Frameworket. Dette er den egentlige regelmotor. Det er denne der har kendskab til alle reglerne, og det er her at alle facts indsættes og reglerne herefter evalueres. RegelmotorImpl er således en wrapper der sikrer, at KnowledgeBase har de nødvendige ressourcer som AktivitetsAfvikleren og Data Access Objekterne til rådighed. Den sikrer også, at alle facts vedrørende kunden bliver indsat i den egentlige regelmotor inden evaluering af reglerne.

Når reglerne udføres vil det være muligt at kalde AktivitetsAfvikleren. Denne har til ansvar at udføre de aktiviteter der måtte omhandle en kunde, eksempelvis at sende et brev med varsel om lønindeholdelse ud til en kunde. Al funktionalitet, der vedrører aktiviteter vil være implementeret i, eller være muligt at kalde igennem AktivitetsAfvikleren.

Reglerne vil også have adgang til et API, hvorigennem de kan oprette fremtidige

hændelser. Dette vil især gælde for de implementerede indsatser.

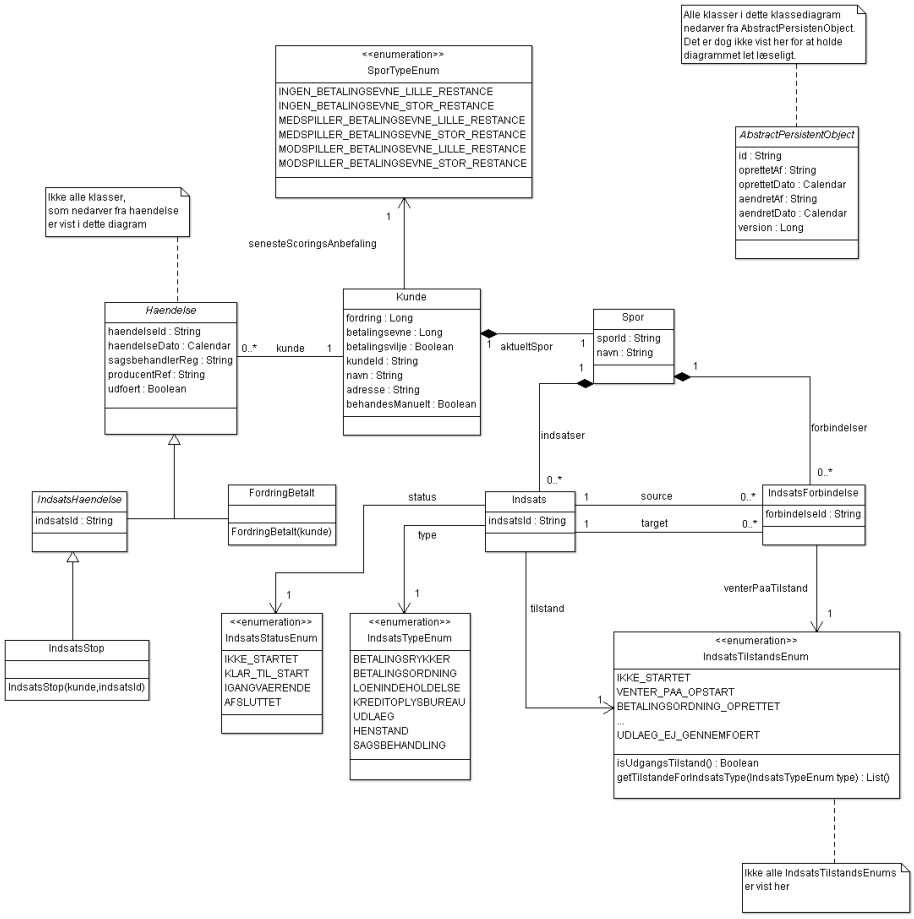
5.2.3 Model

Da Hibernate anvendes til at mappe Java objekter til og fra databasen, er det nødvendigt at indføre nogle tovejs afhængigheder i domænemodellen, hvis Hibernates Cascade funktionalitet ønskes anvendt. Eksempelvis kender et spor de indsatser, der ligger herpå, og hver enkelt indsats ved hvilket spor denne er på. Når dette er opfyldt er det muligt at anvende Hibernates Cascade funktionalitet, der gør det nemt at persistere objekter hele vejen igennem. Eksempelvis hvis en kunde hentes ud af databasen, og der sker fremgang i dennes spor ved at ændre tilstand i en af dets indsatser. Når denne ændring skal persisteres sker det ved blot at persistere kundeobjektet og alle ændringer til afhængige objekter vil blive cascaded. Med andre ord vil ændringerne blive ført helt igennem. Hvordan Cascade bliver indført uden at medføre unødvendige opdateringer, der koster performance, eller uønskede sletninger, bliver beskrevet i afsnit 6.2.

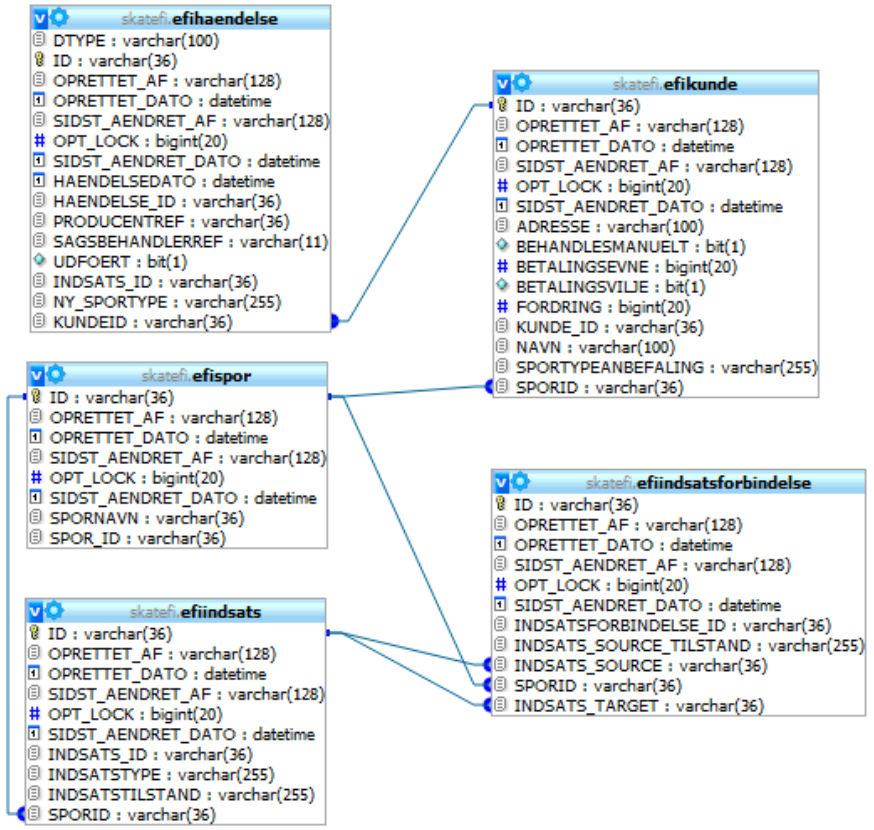
I figur 5.8 illustreres et klassediagram udarbejdet ud fra den tidligere viste domænemodel i figur 3.1. I klassediagrammet ses det, at der er i forhold til den tidligere viste domænemodel, er indført tovejs afhængigheder imellem; Kunde og Hændelse, Kunde og Spor, Spor og Indsats, Spor og IndsatsForbindelse samt Indsats og IndsatsForbindelse. Dette skyldes, som sagt, at det er en nødvendighed for at anvende Hibernates Cascade funktion.

I klassediagrammet er der indført Enums for SporType, IndsatsStatus, IndsatsTilstand samt IndsatsType. For de førstnævnte tre, er dette en god løsning, da de vil blive anvendt på en måde, hvor de ofte ændres efter instantiering. Dette er dog ikke tilfældet for indsatstypen. I denne prototype er det valgt at bruge en IndsatsTypeEnum til at beskrive hvilken type, indsatsinstansen er. Dette skyldes, at ingen endnu har deres egne karaktertræk. I en færdig løsning ville det sandsynligvis skulle ændres til, at hver indsats nedarver fra indsatsklassen i stedet, da dette muliggør, at hver indsats kan få sine egne egenskaber. Præcis som det er gældende for hændelserne.

Idet det er forsøgt at holde klassediagrammet så simpelt som muligt, vises det ikke direkte herpå, at alle objekter, der skal persisteres, nedarver fra klassen AbstractPersistentObject. Dette gør de for at skabe et fælles grundlag, der kan anvendes af Hibernate til persistering. Samtidig har det den funktion, at skabe sporbarhed, da det er muligt at holde styr på hvad eller hvem, der har ændret data, samt hvornår dataene blev ændret. AbstractPersistentObject klassen bliver nærmere beskrevet i afsnit 6.3.



Figur 5.8: Klassediagram, der illustrerer relevante kundeinformationer, som bliver indsat i regelmotoren til evaluering.



Figur 5.9: E/R model for kundeinformationerne. Fremmednøglen eksisterer på den tabel, hvor den blå bobbel sidder.

5.2.4 Datapersistering

Hibernate vil blive anvendt til at mappe Java-objekter til og fra den relationelle database. E/R modellen vil derfor under implementering blive genereret derudfra. Hibernate skal dog via annotationer i koden, forklares hvordan den skal forstå relationerne, objekterne imellem.

Derfor er der i figur 5.9 udarbejdet et traditionelt Entity/Relationship diagram, E/R diagram. Dette er udarbejdet på baggrund af klassediagrammet i figur 5.8.

Alle entiteter i E/R diagrammet har attributterne; id, opt_lock, oprettet_af,

oprettet_dato, sidst_aendret_af, sidst_aendret_dato.

Id og opt_lock anvendes af Hibernate til at styre samtidighedskontrollen med en optimistisk tilgang.

Oprettet_af, oprettet_dato, sidst_aendret_af, sidst_aendret_dato er for at øge sporbarheden. På denne måde kan man se hvem eller hvad der har oprettet eller ændret dataene. Historie over dette skal sikres af databasen, hvor databasetrickers ønskes anvendt til at kopiere data over i en anden tabel ved ændring.

For hændelsen oprettes der en fremmednøgle til kunden, således at der ikke kan persisteres en hændelse for ikke eksisterende kunder.

Ligeledes kan indsatserne samt indsatsforbindelserne også kun trækkes ind på et eksisterende spor. Derfor oprettes der også fremmednøgler her.

Indsatsforbindelserne er også kun nyttige hvis de forbinder to eksisterende indsatser, så her skal der også være en fremmednøgle.

Hvordan E/R modellen tilvejebringes via Hibernate annotationer beskrives i afsnit 6.2.

Der vil blive oprettet Data Access Objekter, DAO, for hver entitet i E/R diagrammet. Disse har til formål at udstille et interface, der giver nem adgang til data i databasen, uden at andre enheder i systemet har kendskab til databasespecifikke detaljer.

5.2.5 Sagsbehandlerportal

Sagsbehandlerportalen skal kunne understøtte den ønskede funktionalitet beskrevet i afsnit 3.4. Den overordnede funktionalitet er oprettelse af kunder, ændring af kunder samt den aktuelle inddrivelsesstrategi for hver enkelt kunde. Det skal også være muligt at udsende hændelser.

Sagsbehandlerportalen designes ud fra designmønstret Model-View-Controller. Spring MVC vil blive anvendt til at sikre denne opdeling mellem domænemodellen og brugergrænsefladen. Spring MVC blev beskrevet i afsnit 4.3.1.

Brugergrænsefladen skal have en side for hver ønsket funktionalitet, hvor det er muligt for sagsbehandleren at udføre opgaven. Controlleren binder domænemodellen og brugergrænsefladen sammen. Der udarbejdes derfor også en Controller for hver af siderne.

Sammenhæng mellem side og controller ses i tabellen nedenfor:

Side:	Controller:
KundeOpretPage	KundeOpretController
KundeAendrePage	KundeAendreController
KundeSporAendrePage	KundeSporAendreController
KundeOverblikPage	KundeOverblikController
UdsendHaendelsePage	UdsendHaendelseController

I afsnit 6.6 bliver det beskrevet hvordan en Spring MVC Controller skaber sammenhæng mellem et domæneobjekt og brugergrænsefladen.

Sagsbehandlerportalen vil have kendskab til Hændelseshåndtereren, hvor den udsender hændelser igennem. Den vil også have kendskab til alle DAO'er, så det er muligt at hente og ændre data.

5.3 Forretningsregler

I den forretningsmæssige analyse er der opstillet nogle forretningsregler, der underbygger forretningskonceptet. Disse regler skal omsættes til regler, en regelmotor kan evaluere.

For Scoring af kunde er det lige til, idet disse allerede er på en sådan form, at de kan implementeres med det samme. For spor- samt indsatsafvikling, vil der i de følgende underafsnit blive defineret nogle konceptuelle regler, der skal lette implementeringen af reglerne senere.

5.3.1 Sporreglerne

Sporene består, som tidligere beskrevet, af indsatser samt indsatsforbindelser. Indsatsforbindelserne har en indsats i hver sin ende. Der hvor forbindelsen udspringer fra, er indsatsen source. Der hvor forbindelsen ender, er indsatsen target. I afsnit 3.1.1 er det defineret, hvornår en indsats må starte. For repetitionens skyld er det herunder også listet hvornår en indsats må starte:

- Hvis en indsats ikke venter på andre indsatser, kan den starte.

- Hvis en indsats venter på andre indsatser, da må denne indsats først starte, idet én af disse indsatser når til en tilstand som indsatsen, der ventede, ønskede.

I reglerne defineres en indsats der ikke venter på andre indsatser ved, at der ikke eksisterer nogen indsatsforbindelser, hvori den pågældende indsats er target.

I reglerne kan en indsats, der venter på andre indsatser, starte, når det gælder, at der for én af sourceindsatserne i alle de indsatsforbindelser, hvor indsatsen er target, er opnået én af de ønskede udgangstilstande, der blev ventet på.

Hvordan de to ovenstående konceptuelle regler implementeres, bliver illustreret i afsnit 6.4.1.

5.3.2 Indsatsreglerne

Indsatsmodellerne bliver brugt internt i SKAT, og fungerer som et fælles grundlag mellem tekniske og ikke-tekniske personer, til at forstå hvordan en indsats skal forløbe. Derfor har det været nærliggende at definere en måde at skrive reglerne på således, at det så vidt muligt ligger op af den i forvejen kendte model. Indsatsmodellerne omsættes derfor næsten direkte til indsatsregler efter følgende to principper:

- Hver hændelse som indsatsen kan reagere på, defineres som en regel, der abonnerer på hændelsen samt sikrer, at indsatsen er i netop den tilstand hvor hændelsen er relevant.

- Når en regel aktiveres og handlingen udføres, da kaldes den tilsvarende metode i aktivitetsafvikleren. Herefter skiftes der tilstand i indsatsen på baggrund af dette.

Den følgende tabel indeholder eksempler på de nødvendige regler for at dække en indsatsmodel. Indsatsmodellen der forsøges omsat til regler, er indsatsmodellen for betalingsrykker illustreret i figur 2.13.

Regelnavn:	Aktiveres når:	Handling:
Start betalingsrykker	Indsatsen har fået tilladelse til start af sporafvikleren.	Rykker udsendes til kunde og en fremtidig rykkerfristhændelse oprettes.
Rykkerfrist overskredet	Indsatsen befinder sig i tilstanden "Rykker sendt" og modtager en hændelse om, at fristen er overskredet.	Stop indsatsen og skift til udgangstilstanden: Betalingsrykker Afsluttet.
Stop betalingsrykker	Indsatsen befinder sig i tilstanden "Rykker sendt" og modtager en hændelse om, at indsatsen skal stoppe.	Stop indsatsen og skift til udgangstilstanden: Betalingsrykker Afsluttet.
Fordring betalt	Indsatsen befinder sig i tilstanden "Rykker sendt" og modtager en hændelse om, at fordringen er betalt.	Stop indsatsen og skift til udgangstilstanden: Betalingsrykker Lykkes.

I tabellen ses det hvordan der er udarbejdet regler for hver hændelse som indsatsen kan reagere på. Ved modtagelse af hændelsen sikres det, at indsatsen er i en tilstand, hvor hændelsen har relevans.

Kolonnen Handling i tabellen repræsenterer aktiviteterne og deres slutresultater.

Hvordan de konceptuelle indsatsregler implementeres bliver illustreret i afsnit 6.4.2.

Kapitel 6

Software implementering

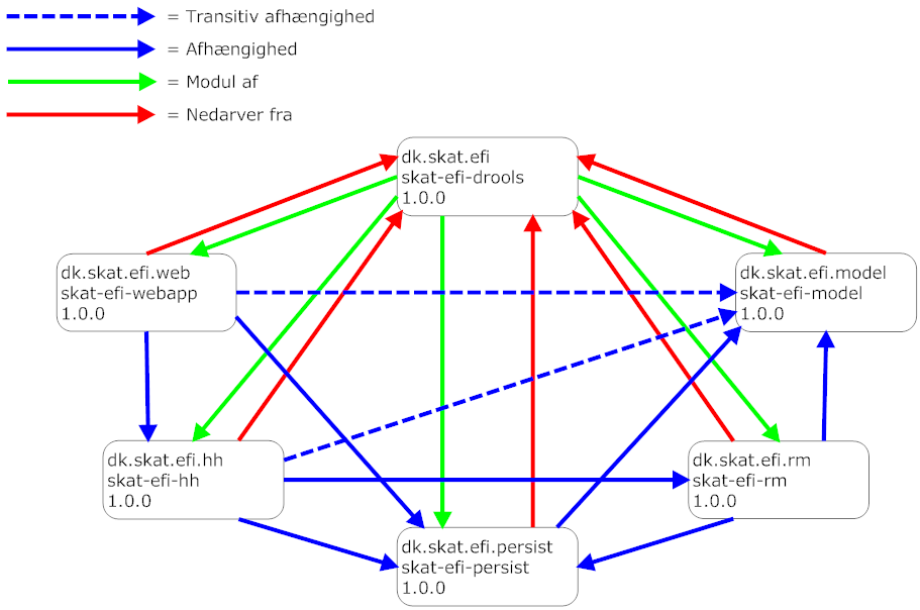
Anden prototype er ligesom Første prototype implementeret i Java, da både det oprindelige EFI, samt Drools Expert også er implementeret i Java. Maven er anvendt til at opdele projektet i moduler, der hver især har deres ansvarsområde. Et Maven Multi-module project vil blive anvendt til at samle modulerne til én enhed.

Et passende antal moduler er blevet oprettet ud fra systemdiagrammet i figur 5.3. Disse moduler samt deres relation til hinanden vises i figur 6.1.

Som nævnt foroven har hvert enkelt projekt sit eget ansvarsområde. Herunder følger en kort beskrivelse af disse ansvarsområder. Hvert enkelt projekt vil dog blive nærmere beskrevet i de følgende underafsnit.

skat-efi-drools Maven Multi-module projekt, der samler modulerne til én enhed. Samtidig er projektet med til at sikre, at alle moduler anvender samme version af deres afhængigheder.

skat-efi-model Maven module, der definerer domæneobjekterne, som anvendes som fælles datagrundlag for alle modulerne, nogle mere afhængige end andre.



Figur 6.1: Illustration af hvordan projektet er opdelt i Maven-moduler, med hver deres ansvarsområder.

skat-efi-persist Maven module, der har til ansvar at persistere data. Der udstilles et API til hændelseshåndtereren og et til regelmotoren. I forhold til systemdiagrammet er dette data access laget.

skat-efi-rm Maven module, der instantierer regelmotoren ud fra de skrevne forretningsregler. I forhold til systemdiagrammet er dette hele regelmotorkomponenten inkl. aktivitetsafvikleren.

skat-efi-hh Maven module, der instantierer hændelseshåndtereren, som modtager hændelser og videregiver disse til regelmotoren, til evaluering. Modulet har også til ansvar at instantiere et batchjob, der sikrer at fremtidige hændelser bliver udsendt med faste intervaller. I forhold til systemdiagrammet er dette hele hændelseshåndterer-komponenten.

skat-efi-webapp Maven module, der konfigurerer en webapplikation, der skal fungere som sagsbehandlerportal, hvor sagsbehandleren kan ændre en kundes informationer.

6.1 Afhængighed

Projektet bliver bygget som et Maven Multi-module projekt. Til dette oprettes en Super POM. POM står for Project Object Model. Det er en fil, der holder alle konfigurationer for et enkelt projekt. I dette tilfælde laves en generel POM, der kaldes Super POM. Denne sikrer, at alle modulerne i dette Multi-modul benytter samme konfiguration, hvilket vil sige samme version af afhængigheder udefra, såvel som interne afhængigheder. Dette mindsker risikoen for uventede fejl, både under udvikling af løsningen og når denne går i produktion. Det vil gøre vedligeholdelsen af systemet nemmere, da versionerne styres et centralt sted.

I figur 6.2 vises en forsimplet udgave af Super POM'en, der skal sikre ovenstående. I Super POM'en ses der hvilke moduler, der er byggesten i projektet. Det ses også hvordan bygningen af projektet defineres et centralt sted. I dette eksempel er det med et plug-in defineret, at alt Java-kode skal kompileres til Java 6. Super POM'en i eksemplet er en forsimplet udgave. I den rigtige er der bl.a. også plug-ins til at genere code-coverage rapporter, der på en grafisk måde hjælper under udviklingen til at give et overblik over, hvilken kode der er dækket af unit tests. Det ses samtidig hvordan afhængigheden til andre projekter defineres et centralt sted. Her er det Drools platformen, der bliver defineret en afhængighed af.

Når et modul anvender Super POM'en til at sikre, at de korrekte versioner bliver anvendt, da ser det ud som i figur 6.3. Det er et forsimplet eksempel

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>dk.skat.efi</groupId>
    <artifactId>skat-efi-drools</artifactId>
    <version>1.0.0</version>
    <packaging>pom</packaging>
    <name>Maven-Multi-Module SKAT EFI Drools</name>
    <modules>
        <module>skat-efi-model</module>
        <module>skat-efi-persist</module>
        <module>skat-efi-rm</module>
        <module>skat-efi-hh</module>
        <module>skat-efi-webapp</module>
    </modules>
    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <version>2.0.2</version>
                    <configuration>
                        <source>1.6</source>
                        <target>1.6</target>
                    </configuration>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>
    <dependencyManagement>
        <dependencies>
            <!-- Dependencies for Drools -->
            <dependency>
                <groupId>org.drools</groupId>
                <artifactId>drools-core</artifactId>
                <version>${drools.version}</version>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <properties>
        <drools.version>5.3.0.Final</drools.version>
    </properties>
</project>
```

Figur 6.2: Forsimplet eksempel på den Super POM, der sikrer at alle moduler anvender samme version af afhængigheder.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>dk.skat.efi.rm</groupId>
    <artifactId>skat-efi-rm</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>
    <name>Maven-Module SKAT EFI Regelmotor</name>

    <parent>
        <groupId>dk.skat.efi</groupId>
        <artifactId>skat-efi-drools</artifactId>
        <version>1.0.0</version>
    </parent>

    <dependencies>
        <!-- Dependencies for Drools -->
        <dependency>
            <groupId>org.drools</groupId>
            <artifactId>drools-core</artifactId>
        </dependency>
    </dependencies>

</project>
```

Figur 6.3: Forsimplet eksempel på Child POM, der arver fra Super POM.

på POM filen i projektet skat-efi-rm, der instantierer regelmotoren. I POM'en ses det hvordan denne angiver Super POM'en som sin 'parent'. Dette medfører at POM'en nedarver egenskaberne fra Super POM'en. Derfor er det i POM'en heller ikke defineret, hvilken version af drools-core, der anvendes. Dette skyldtes som sagt, at versionen arves fra Super POM'en. Build-delen arves ligeledes fra Super POM'en.

Denne fremgangsmåde vil blive anvendt i alle modulerne tilhørende dette projekt. Alle afhængighederne vil ikke blive listet her, da disse kan ses i den originale Super POM i koden. Det vil dog primært være afhængigheder til de tidligere nævnte teknologier i afsnit 4.

```
@Entity
@Table(name = "EFIKUNDE")
public class Kunde extends AbstractPersistentObject {

    @Column(name = "KUNDE_ID", length = 36, nullable = false)
    private String kundeId;

    @Column(name = "NAVN", length = 100, nullable = false)
    private String navn;

    @OneToOne(cascade=CascadeType.ALL)
    @JoinColumn(name="SPORID")
    private Spor aktueltSpor;

    @Column(name = "SPORTYPEANBEFALING", nullable = true)
    @Enumerated(EnumType.STRING)
    private SportTypeEnum senesteScoringsAnbefaling;

    @OneToMany(mappedBy="kunde",cascade={CascadeType.ALL}, orphanRemoval=true)
    private List<Haendelse> haendelser = new ArrayList<Haendelse>();

    public Kunde() {
        super();
    }

    /*
     * Getters og Setters
     */

}
```

Figur 6.4: Forsimplet eksempel på hvordan en kunde vha. notationer muliggør brug af Hibernate.

6.2 Model

Klassediagrammet, vist i figur 5.8, er blevet omsat til almindelige Java Beans med accessors, getter-metoder, og mutators, setter-metoder. Hibernate anvendes til, at mappe disse Java objekter til og fra den relationelle database, MySQL. Derfor er der lavet nogle annotationer i koden, der skal gøre dette muligt for Hibernate. Annotationer er skrevet, så de tilvejebringer E/R modellen i figur 5.9. I figur 6.4 kan en forsimplet udgave af Kundeobjektet ses. Der illustreres hvordan en fremmednøgle til et andet objekt opnås med annotationer.

Det ses hvordan der laves en One-To-One association mellem kunden og kundens spor. Dette medfører, at der laves en fremmednøgle i databasen. Kunden kan således ikke være tilknyttet et spor, der ikke ligger i databasen, eller ikke bliver indsat i databasen samtidig med kunden.

I domænemodellen i analysefasen var det kun tiltænkt, at hændelsen havde kendskab til kunden. Kunden har i implementeringen fået kendskab til de hændelser, der omhandler denne. Alle hændelserne, der har kendskab til den givne kunde, bliver derfor lagt ind i listen vha. Hibernate. Dette giver den praktiske fordel, at Cascade og OrphanRemoval kan anvendes til automatisk at fjerne alle hændelser vedrørende en kunde, når denne kunde slettes. Cascade er her sat til CascadeType.ALL. Dette betyder at både Create, Update og Delete statements, CRUD, skal videregives til både sporet samt hændelserne. Det omvendte er ikke tilfældet i klasserne Spor og Haendelse, da en sletning af en hændelse eksempelvis ikke må resultere i sletning af en kunde. Det er værd at bemærke, at der ikke oprettes en fremmednøgle til hændelserne, da det er hændelserne, som er afhængig af kunden. Resten af felterne mappes blot til en kolonne i databasetabellen for en kunde.

Det bemærkes desuden at klassen nedarver fra AbstractPersistentObject. Denne klasse nedarver alle klasser, hvor objekter skal persisteres, fra. Klassen beskrives nærmere i afsnit 6.3.

I figur 5.9, der illustrerede E/R diagrammet blev der i tabellen for hændelser set en attribut ved navn DTYPE. Dette skyldes, at denne tabel er en samlet tabel over alle hændelser, der er nedarvet fra Java klassen Haendelse. Det er en måde hvorpå at Hibernate sikrer, at den rigtige Java-klasse returneres ved en Hent-operation. Klassen Haendelse sikrer ved Hibernate mapping, at alle haendelser vil have en kunde påsat, da kunden ikke må være null herpå.

6.3 Datapersistering

Da Hibernate anvendes til at persistere data, slippes der for at skrive triviel JDBC kode. Det kræver således ikke mange linjer kode at implementere et passende servicelag, til adgang af data. Datalaget bliver mindre indviklet og nemmere at forstå, hvilket medfører at fokus kan holdes på forretningslogikken.

Hibernate gør desuden systemet mere vedligeholdbart, da det er nemmere at implementere ændringer. Det sikres også, at systemet ikke gøres afhængig af en specifik database, da Hibernate tager sig af low-level SQL statements. Hvilken database Hibernate skal arbejde med, defineres i Spring konfigurationen.

I modellen er der oprettet en klasse ved navn AbstractPersistentObject. Denne klasse sikrer, at alle objekter, der skal persisteres, har et fælles grundlag, som Hibernate kan arbejde med.

```java
@MappedSuperclass
public abstract class AbstractPersistentObject implements Serializable {
    @Id
    @Column(name = "ID", length = 36, nullable = false)
    protected String id = UUID.randomUUID().toString();

    @Version
    @Column(name = "OPT_LOCK")
    protected Long optLockVersion = null;

    @Column(name = "OPRETTET_AF", nullable = true, length = 128, insertable = true, updatable = false)
    protected String createdBy;

    @Column(name = "SIDST_AENDRET_AF", nullable = true, length = 128, insertable = false, updatable = true)
    protected String modifiedBy;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "OPRETTET_DATO", nullable = false, insertable = true, updatable = false)
    protected Calendar createdTimestamp = Calendar.getInstance();

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "SIDST_AENDRET_DATO", nullable = true, insertable = false, updatable = true)
    protected Calendar updatedTimestamp;

    @Override
    public boolean equals(Object o) {
        boolean result;

        if (this == o) {
            return true;
        }
        if (!(o instanceof AbstractPersistentObject)) {
            return false;
        }
        String id1 = getId();
        String id2 = ((AbstractPersistentObject) o).getId();
        result = id1.equals(id2);

        return result;
    }

    @Override
    public int hashCode() {
        return id.hashCode();
    }
}
```

Figur 6.5: AbstractPersistentObject klassen alle objekter til persistering nedarver fra.

En forsimplet udgave af klassen er vist i figur 6.5.

Klassen er en abstract super klasse, der sikrer, at et en UUID primærnøgle genereres ved oprettelse af objektet, og altså ikke ved indsættelse i databasen. Dette gør det muligt at definere equals og hashCode metoderne baseret på UUID'et. Hibernate informeres om, at UUID'et skal være primærnøgle i databasen ved at annotere id med annotationen @Id.

I afsnit 3.3, blev det besluttet at have en optimistisk tilgang til samtidighedskontrol, dette sikres hurtigt med Hibernate annotationen @Version. Hibernate tæller denne variabel op hver gang en update statement, der ændrer objektet, udføres i databasen. Sker det derfor, at to brugere samtidigt henter samme objekt ud fra databasen, for dernæst at ændre objektets tilstand, da vil den første

brugers commit til databasen blive udført, hvorimod den anden brugers commit vil medføre, at dennes igangværende transaktion vil blive afbrudt og rulle ændringer som transaktionen lavede, tilbage.

AbstractPersistentObject sikrer også, at der er en vis form for sporbarhed på de objekter, der persisteres. Dette sker ved, at to fælles timestamp, for de persisterede objekter defineres. Det ene instantieres kun ved oprettelse af objektet. Det andet ændres hver gang en update statement, der ændrer objektet udføres i databasen. Det vil således også være muligt at angive referencer for hvem, der henholdsvis har oprettet eller har ændret objektet.

Historien forestilles gemt med databasetrickers, der skal kopiere de gamle data over i en anden tabel inden ændring.

6.3.1 Data Access Objects

For at spare et yderligere antal linjer af kode, er der oprettet en abstract klasse ved navn GenericDaoImpl, der implementerer interfacet GenericDao. GenericDaoImpl indeholder alle standardmetoder, der anvendes i forbindelse med at persistere objekterne.

I figur 6.6 vises GenericDaoImpl, der sikrer at kun ekstra funktionalitet skal implementeres i DAO'erne.

Data Access Objekter, der ønsker at implementere ekstra funktionalitet, eksempelvis KundeDaoImpl, der giver mulighed for at lave et opslag på en kunde ud fra dennes CPR-nummer, anvender GenericDaoImpl til standardoperationer, og implementer således kun den funktionalitet, der gør denne DAO unik.

I figur 6.7 er eksemplet med KundeDaoImpl vist.

Det er værd at bemærke, at forespørgslen ikke er skrevet i traditionel SQL, men i stedet Hibernates eget sprog, HQL. HQL anvendes for at sikre, at koden ikke bliver afhængig af en bestemt database.

Fordelen ved HQL er også, at det er muligt at give Java-objekter med som argumenter i forespørgslen, så vil Hibernate selv sørge for at sammenligne dette med objekter i databasen. Eksemplet i figur 6.7. illustrerer ikke dette, men et sådan eksempel kan ses i DAO'en for hændelser i koden, hvor alle hændelser vedrørende en kunde hentes ud.

```
@Transactional(propagation = Propagation.MANDATORY)
public abstract class GenericDaoImpl <T extends Serializable> implements GenericDao<T> {

    @Autowired
    protected HibernateTemplate hibernateTemplate;

    protected Class<T> type;

    public GenericDaoImpl(Class<T> type) {
        this.type = type;
    }

    public void saveOrUpdate(T object) {
        this.hibernateTemplate.saveOrUpdate(object);
    }

    public void delete(T object) {
        hibernateTemplate.delete(object);
    }

    public T findById(String id) {
        return this.hibernateTemplate.get(type, id);
    }

    public List<T> findAll() {
        return this.hibernateTemplate.loadAll(type);
    }

    public void flush() {
        this.hibernateTemplate.flush();
    }

}
```

Figur 6.6: GenericDaoImpl implementerer standard funktionalitet som alle data access objekter anvender.

```
public class KundeDaoImpl extends GenericDaoImpl<Kunde> implements KundeDao {

    public KundeDaoImpl(Class<Kunde> type) {
        super(type);
    }

    @SuppressWarnings("unchecked")
    @Override
    public Kunde findByKundeId(String kundeId) {
        Kunde result = null;

        String queryString = "from Kunde as k where k.kundeId = ?";
        List<Kunde> kunder = this.hibernateTemplate.find(queryString, kundeId);
        if (!kunder.isEmpty()) {
            result = kunder.get(0);
        }

        return result;
    }

}
```

Figur 6.7: KundeDaoImpl implementerer yderligere funktionalitet end hvad, der findes i GenericDaoImpl.

6.3.2 Spring opsætning

For at adskille konfiguration og kode bedst muligt, anvendes Spring til konfiguration af miljøet i hele projektet. Konfigurationen af Hibernate og forbindelsen til databasen sker derfor også i Spring. Datalaget konfigureres i filen skat-efi-persist.xml.

Her oprettes en DataSource, der står for forbindelsen til den fysiske database. Med denne samt en liste over alle objekterne i domænemodellen, oprettes der en SessionFactory. Denne står for at oprette hver enkelt session, som anvendes til at sende og modtage beskeder til og fra databasen.

HibernateTemplated, der blev set anvendt i GenericDaoImpl instantieres med denne SessionFactory. Desuden oprettes også en TransactionManager, til at styre transaktionerne. Denne konfigureres, så transaktioner kan startes i koden vha. annotationer. Se eksempelvis i toppen af GenericDaoImpl. Her defineres det, at der skal være en åben transaktion tilgængelig før, at metoderne må kaldes. Det defineres med annotationen @Transactional, hvor 'propagation' sættes til 'mandatory'. Dette betyder som sagt, at en transaktion skal være tilstede.

Transaktioner anvendes for at sikre, at data i databasen forbliver konsistent, da alt hvad der sker indenfor en transaktion kan rulles tilbage, hvis en fejl opstår.

```
<bean id="skat-efi-persist-kundeDao" class="dk.skat.efi.persist.dao.KundeDaoImpl">
    <constructor-arg>
        <value>dk.skat.efi.model.rm.Kunde</value>
    </constructor-arg>
</bean>
```

Figur 6.8: Eksempel på hvordan en DAO oprettes i Spring konfigurationen.

I figur 6.8 er det illustreret hvordan en DAO instans oprettes i Spring konfigurationen.

De steder i koden, hvor KundeDAO'en anvendes bliver den injected vha. af Springs IoC Container ved opstart af systemet. Annotationen @Autowired skal dog anvendes, for at give containeren besked.

6.4 Regelmotor

Forretningslogikken implementeres med Drools. I Drools samles alle de regler, der udgør forretningslogikken i et objekt kaldet KnowledgeBase. Ved instantiering af KnowledgeBase-objektet genereres Rete-netværket. Det tager tid at strukturere Rete-netværket, men det har ingen indflydelse på performance, da KnowledgeBase objektet anvendes som en singleton. Objektet oprettes altså kun ved applikationens opstart. For at gøre konfigurationen af KnowledgeBase lettere tilgængelig, er der oprettet en Factory-klasse kaldet KnowledgeBaseFactoryBean. Denne gør det muligt, at oprette en KnowledgeBase fra Spring konfigurationen ved blot at specificere hvilke regler, den skal instantieres med.

I figur 6.9 vises et udsnit af Spring konfigurationen for regelmotoren, der viser hvordan KnowledgeBase oprettes med KnowledgeBaseFactoryBean.

Den instans af KnowledgeBase som KnowledgeBaseFactoryBean opretter, ud fra de givne regler, vil være konfigureret til at være i en tilstand, hvor den kan håndtere hændelser gennem en Stream. Dette er vigtigt for, at Drools Fusion kan anvendes til at effektivisere reglerne.

Hvordan KnowledgeBase instantieres og konfigureres kan ses i figur 6.10, der viser et kodeudsnit, fra KnowledgeBaseFactoryBean. Det er muligt at konfigurere KnowledgeBase yderligere, men default opsætningen dækker resten af behovet her.

Reglerne skrives i et sprog kaldet Drools Rule Language, DRL. Dette indikeres af filtypen.

```
<bean id="skat-efi-rm-knowledgeBase" class="dk.skat.efi.rm.KnowledgeBaseFactoryBean">
    <constructor-arg>
        <map>
            <entry key="classpath:rules/scoringsregler.drl" value="DRL" />
            <entry key="classpath:rules/sporregler.drl" value="DRL" />
            <entry key="classpath:rules/indsatser/betalingsordning.drl" value="DRL" />
            <entry key="classpath:rules/indsatser/betalingsrykker.drl" value="DRL" />
            <entry key="classpath:rules/indsatser/henstand.drl" value="DRL" />
            <entry key="classpath:rules/indsatser/kreditoplysningsbureau.drl" value="DRL" />
            <entry key="classpath:rules/indsatser/loenindeholdelse.drl" value="DRL" />
            <entry key="classpath:rules/indsatser/sagsbehandling.drl" value="DRL" />
            <entry key="classpath:rules/indsatser/udlaeg.drl" value="DRL" />
        </map>
    </constructor-arg>
</bean>
```

Figur 6.9: KnowledgeBaseFactoryBean anvendes til at oprette en KnowledgeBase med de angivne regler.

```
public class KnowledgeBaseFactoryBean implements FactoryBean<KnowledgeBase> {

    private KnowledgeBase knowledgeBase;

    public KnowledgeBaseFactoryBean(Map<Resource, ResourceType> resourceMap) throws IOException {
        KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

        for (Entry<Resource, ResourceType> entry : resourceMap.entrySet()) {
            kbuilder.add(ResourceFactory.newInputStreamResource(entry.getKey()
                    .getInputStream()), entry.getValue());
        }

        if (kbuilder.hasErrors()) {
            throw new RuntimeException(kbuilder.getErrors().toString());
        }

        KnowledgeBaseConfiguration config = KnowledgeBaseFactory.newKnowledgeBaseConfiguration();
        config.setOption(EventProcessingOption.STREAM);

        knowledgeBase = KnowledgeBaseFactory.newKnowledgeBase(config);
        knowledgeBase.addKnowledgePackages(kbuilder.getKnowledgePackages());

    }

    @Override
    public KnowledgeBase getObject() throws Exception {
        return this.knowledgeBase;
    }

    @Override
    public Class<KnowledgeBase> getObjectType() {
        return KnowledgeBase.class;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }

}
```

Figur 6.10: KnowledgeBaseFactoryBean har til ansvar at oprette KnowledgeBase ud fra de givne regler i mappet.

Hvordan reglerne er skrevet til at afvikle spor og indsatser, vil blive forklaret i de følgende to afsnit. Reglerne for scoring af kunde vil ikke blive nærmere beskrevet her, da eksemplet som blev gennemgået i afsnit 4.1.1 næsten er identisk med den endelige implementering. Implementeringen dækker blot over flere tilfælde.

6.4.1 Sporreglerne

Sporene består, som tidligere beskrevet, af indsatser samt indsatsforbindelser. Indsatsforbindelserne har en indsats i hver sin ende. Den ene indsats er source. Den anden er target.

I afsnit 5.3.1 blev de konceptuelle regler, for hvornår en indsats må starte, defineret. For repetitionens skyld er disse listet herunder:

- Hvis en indsats ikke venter på andre indsatser, kan den starte.
  - I reglerne defineres en indsats der ikke venter på andre indsatser ved, at der ikke eksisterer nogen indsatsforbindelser hvori den pågældende indsats er target.
- Hvis en indsats venter på andre indsatser, da må denne indsats først starte, idet én af disse indsatser, når til en tilstand som indsatsen, der ventede, ønskede.
  - I reglerne kan en indsats, der venter på andre indsatser starte, når det gælder, at der for én af sourceindsatserne i alle de indsatsforbindelser, hvor indsatsen er target, er opnået én af de ønskede udgangstilstande, der blev ventet på.

Reglerne der er implementeret i Drools Rule Language, DRL, er blot en oversættelse til teknisk syntakst fra ovenstående to konceptuelle regler. I figur 6.11 vises implementeringen for sporafviklingen:

Reglen "Opstart indsatser der ikke afventer" markerer, at alle de indsatser, hvor det gælder, at der ikke findes en indsatsforbindelse, hvori denne er target, må starte.

Reglen "Opstart indsatser der afventer udgangstilstand" markerer, at alle de indsatser, hvor det gælder, at der findes en indsatsforbindelse, hvor indsatsen er target og det samtidig gælder, at source indsatsen er nået til den ønskede udgangstilstand.

```
#
# Reglen opstarter alle indsatser, der ikke afventer en udgangstilstand fra en anden indsats
#
rule "Opstart indsatser der ikke afventer"
    when
        $indsats : Indsats( tilstand == IndsatsTilstandEnum.IKKE_STARTET )
        not IndsatsForbindelse( targetIndsats == $indsats )
    then
        LOG.info("Indsats " +$indsats.getIndsatsType()+ " er sat til at starte");
        modify($indsats) {
            setTilstand( IndsatsTilstandEnum.VENTER_PAA_OPSTART )
        }
end

#
# Reglen opstarter alle indsatser, der afventer en udgangstilstand fra en anden indsats og hvor denne er opfyldt
#
rule "Opstart indsatser der afventer udgangstilstand"
    when
        $srcIndsats : Indsats( $srcTilstand : tilstand )
        $trgtIndsats : Indsats( tilstand == IndsatsTilstandEnum.IKKE_STARTET )
        $forbindelse : IndsatsForbindelse ( sourceIndsats == $srcIndsats, sourceTilstand == $srcTilstand,
                                            targetIndsats == $trgtIndsats)
    then
        LOG.info("Indsats " + $trgtIndsats.getIndsatsType() + " er sat til at starte");
        modify($trgtIndsats) {
            setTilstand( IndsatsTilstandEnum.VENTER_PAA_OPSTART )
        }
end
```

Figur 6.11: Regler for sporafvikling implementeret i Drools Rule Language, DRL.

6.4.2 Indsatsreglerne

Indsatsmodellerne bliver brugt til at udarbejde de nødvendige regler for indsatserne. Oversættelsen fra indsatsmodel til regler blev beskrevet i afsnit 5.3.2. Det er dog grundlæggende følgende to principper der anvendes:

- Hver hændelse som indsatsen kan reagere på, defineres som en regel, der abonnerer på hændelsen samt sikrer, at indsatsen er i netop den tilstand hvor hændelsen er relevant.
- Når en regel aktiveres og handlingen udføres, da kaldes den tilsvarende metode i aktivitetsafvikleren. Herefter skiftes der tilstand i indsatsen på baggrund af dette.

Et eksempel på hvordan en sådan regel implementeres kan ses i figur 6.12. Reglen findes i indsatsen Betalingsrykker.

Først angives det, at klassen FristOverskredetHaendelse skal behandles som en hændelse. Dernæst defineres den regel som abonnerer på hændelsen. Her er hændelsen kun relevant hvis indsatsen Betalingsrykker har udsendt rykkeren.

Modtages hændelsen om, at fristen er overskredet, da afsluttes indsatsen Betalingsrykker.

```
declare FristOverskredetHaendelse
    @role( event )
end
#
# Rykkerbetalingsfrist overskredet, afslut indsats i udgangstilstand "Betalingsrykker aflsluttet"
#
rule "Betalingsrykker ikke betalt inden rykker frist"
    when
        $h : FristOverskredetHaendelse() from entry-point "SkatEfiEventListener"
        $indsats : Indsats( indsatsId == $h.indsatsId,
                            tilstand == IndsatsTilstandEnum.BETALINGSRYKKER_SENDT )
    then
        LOG.info("Rykkerbetalingsfrist overskredet - Indsats betalingsrykker afsluttet");
        modify($indsats) {
            setTilstand(IndsatsTilstandEnum.BETALINGSRYKKER_AFSLUTTET)
        }
end
```

Figur 6.12: Eksempel på indsatsregel implementeret med Drools Expert.

```
#
# Betalingsrykker initialiseres og rykker sendes til kunde
#
rule "Start indsats - Betalingsrykker"
    when
        $indsats : Indsats( indsatsType == IndsatsTypeEnum.BETALINGSRYKKER,
                            tilstand == IndsatsStatusEnum.KLAR_TIL_START)
    then
        LOG.info("Indsats betalingsrykker oprettet og rykker sendt til kunde");

        aktivitetsAfvikler.sendRykkerTilKunde($indsats.getSpor().getKunde());

        Kunde kunde = $indsats.getSpor().getKunde();
        String indsatsId = $indsats.getIndsatsId();
        FristOverskredetHaendelse fristHaendelse = new FristOverskredetHaendelse(kunde, indsatsId);
        fristHaendelse.setProducentReference("RM-Rykker");
        haendelseDao.opretFremtidigHaendelse(fristHaendelse, 14); // udsend om 14 dage

        modify($indsats) {
            setTilstand(IndsatsTilstandEnum.BETALINGSRYKKER_SENDT)
        }
end
```

Figur 6.13: Eksempel på en regel der starter en indsats.

Alle indsatser definerer desuden også en regel om hvornår denne skal starte. Som oftest er betingelsen blot, at der er givet tilladelse fra sporafvikleren. Det er dog uden komplikationer at definere yderligere betingelser.

Et eksempel på en regel, der starter en indsats og udfører første aktivitet i indsatsen, igen taget fra betalingsrykker, er vist i figur 6.13.

Reglen aktiveres, hvis en indsats af typen betalingsrykker er i den fase af sin livscyklus, hvor den er klar til at blive startet. Med andre ord, den har fået tilladelse fra sporafvikleren til at starte. Når reglen aktiveres, sendes en betalingsrykker ud til kunden og indsatsen opretter en fremtidig hændelse, der vil blive udsendt når fristen for rykkeren udløber.

```
public interface HaendelsesHaandterer {

    /**
     * Haendelsen udsendes direkte til evaluering ved regelmotor
     * derefter persisteres resultatet
     * @param haendelse
     */
    void udsendHaendelse(Haendelse haendelse);

    /**
     * Alle foer fremtidige haendelser, der nu er aktuelle udsendes
     */
    void udsendAktuelleFremtidigeHaendelser();

}
```

Figur 6.14: Hændelseshåndtereren har til ansvar at videregive hændelser til regelmotoren samt persistere resultatet heraf.

6.5 Hændelseshåndtereren

Hændelseshåndtereren står for at modtage og persistere hændelser. Herefter videregives disse til evaluering af regelmotoren. Resultatet fra regelmotoren bliver ligeledes persisteret.

HaendelseDao samt KundeDao bliver anvendt til at persistere hændelserne samt resultatet fra regelmotoren. KundeDao alene, er nok til at persistere hele resultatet, da denne DAO som bekendt cascader alle CRUD SQL statements.

I figur 6.14 er interfacet for Hændelseshåndtereren vist.

Metoden udsendHaendelse kaldes af eksterne systemer eller fra sagsbehandlerportalen. Metoden udsendAktuelleFremtidigeHaendelser kaldes derimod af det batchjob, der sikrer, at de oprettede fremtidige hændelser udsendes ved faste intervaller.

Batchjobbet implementeres med Frameworket Quartz Scheduler. Klassen der har til ansvar, at jobbet bliver udført, er HaendelseBatchJob. Denne nedarver fra klassen QuartzJobBean, der er at finde i frameworket. HaendelseBatchJob implementerer blot metoden executeInternal, der kaldes af frameworket, når det er tid til at udføre jobbet. Derefter kaldes metoden udsendAktuelleFremtidigeHaendelser og klassen HaendelsesHaandtererImpl vil derefter sikre, at alle hændelser sendes til evaluering i regelmotoren, hvorefter resultatet persisteres.

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

   <bean id="udsendHaendelserTask" class="dk.skat.efi.hh.jobs.UdsendHaendelserTask" />

   <bean name="haendelsesBatchJob"
            class="org.springframework.scheduling.quartz.JobDetailBean">
        <property name="jobClass" value="dk.skat.efi.hh.jobs.HaendelsesBatchJob" />
        <property name="jobDataAsMap">
            <map>
                <entry key="udsendHaendelserTask" value-ref="udsendHaendelserTask" />
            </map>
        </property>
    </bean>

    <!-- Cron Trigger -->
    <bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerBean">
        <property name="jobDetail" ref="haendelsesBatchJob" />
        <property name="cronExpression" value="0/5 * * * * ?" />
    </bean>

    <bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
        <property name="jobDetails">
            <list>
                <ref bean="haendelsesBatchJob" />
            </list>
        </property>

        <property name="triggers">
            <list>
                <ref bean="cronTrigger" />
            </list>
        </property>
    </bean>

</beans>
```

Figur 6.15: Batchjobbet konfigureres i Spring. En Cron-Expression anvendes til at angive tidspunktet for udførelse.

Batchjobbet oprettes og konfigureres i Spring. Hvilke tidspunkter batchjobbet skal udføres på, er angivet ved en Cron-Expression. På samme måde som det anvendes i Unix-miljøet.

Hvordan denne Spring konfigurationen ser ud for batchjobbet kan ses i figur 6.15.

Cron-Expression er valgt, da det er en standardmetode at angive tidspunkter for udførsel af batchjobs i Unix-miljøet. Selve batchjobbet oprettes og Frameworket konfigureres til, at give batchjobbet besked, når triggeren udløses.

```
<form:form method="POST" commandName="kunde" action="kundeOpret.htm">
    <form:errors path="*" cssClass="errorblock" element="div" />
    <table>
        <tr>
            <td>CPR :</td>
            <td><form:input path="kundeId" /></td>
            <td><form:errors path="kundeId" cssClass="error" /></td>
        </tr>
        <tr>
            <td>Navn :</td>
            <td><form:input path="navn" /></td>
            <td><form:errors path="navn" cssClass="error" /></td>
        </tr>
        <tr>
            <td colspan="3"><input type="submit" name="opret" value="Opret Kunde"/></td>
        </tr>
    </table>
</form:form>
```

Figur 6.16: Udsnit af JSP side, hvori der anvendes JSTL.

6.6 Sagsbehandlerportal

Sagsbehandlerportalen implementeres med Spring MVC, der blev beskrevet i afsnit 4.3.1. Frameworket er med til at sikre en opdeling mellem domæneobjekterne og brugergrænsefladen.

Brugergrænsefladen er en webapplikation og skal derfor tilgås via en browser. Til at generere HTML-siderne anvendes der JSP samt JSTL. Et eksempel på hvordan JSP og JSTL anvendes til at lave en udvidet HTML-form, kan ses i figur 6.16. En HTML-form anvendes når det skal være muligt for klienten at sende data af sted sammen med et request. HTML-formen blev før omtalt som udvidet. Dette skyldes, at den med JSTL får nogle ekstra egenskaber. I figur 6.16 ses det at formen har fået en attribut ved navn "commandName". Denne attribut er ikke at finde i en normal HTML-form. Den sikrer at når et request, fra denne form, sendes afsted, vil den kunne hentes frem som en attribut med nøglen "kunde".

I figur 6.16 ses det også hvordan 'kundeId' og 'navn' matcher de felter fra Kundeklassen, der blev vist i figur 6.4. Dette er så Spring MVC kan parse data fra formen til det ønskede domæneobjekt.

Når et request fra denne form sendes, vil det ramme en URL der slutter med "/kundeOpret.htm". I Spring MVC sikres det, at requestet behandles af den rigtige controller ved at mappe requestet til en bestemt controller. Et eksempel på dette ses i figur 6.17, hvor første linje i koden sikrer, at metoden bliver kaldt når et request der slutter med "kundeOpret.html" modtages. Dette sker med annotationen @RequestMapping.

```
@RequestMapping(value = "/kundeOpret.htm", method = RequestMethod.POST)
public String processSubmitKundeOpret( @ModelAttribute("kunde") Kunde kunde,
        BindingResult result, SessionStatus status, ModelMap model) {

    kundeValidator.validate(kunde, result);

    if (result.hasErrors()) { // validation failed
        return "KundeOpretPage";
    } else {
        try {
            portalService.opretKunde(kunde);
            status.setComplete(); // success
        } catch(Exception e) { // failed
            result.reject("","Fejl - Kunden blev ikke oprettet! - " +e.getMessage());
        }
        return "KundeOpretPage";
    }

}
```

Figur 6.17: Udsnit af KundeOpretController, der validerer og opretter kunden via servicelaget.

Kundeobjektet hentes ud med nøglen "kunde", som blev omtalt tidligere. Når annotationen @ModelAttribute anvendes, forsøges det automatisk at parse formen til et domæneobjekt. Er formen korrekt lykkes det. I metoden bliver kundeobjektet valideret. Dette sker med et objekt, der implementerer Interfacet Validator fra Spring Frameworket. Dette gør det muligt at udskrive en passende fejlmeddelelse, hvis indtastet data ikke overholder det ønskede format. Et eksempel på en sådan valideringsklasse kan ses i figur 6.18. Et eksempel på en fejlmeddelelse kan ses i figur 8.2.

Hvis valideringen af kundeobjektet lykkes vil det blive gemt via servicelaget og status på requestet markeres som udført med succes. Derefter returneres et response med siden KundeOpretPage, hvor sagsbehandleren igen kan oprette en ny kunde.

Fejler valideringen vil der også blive returneret et response med siden KundeOpretPage, men nu med passende fejlmeddelelser ud fra hvert enkelt inputfelt.

Valideringsklassen i figur 6.18 illustrerer hvordan Spring Frameworket anvendes til validering. I eksemplet kontrolleres det kun, at de indtastede data ikke er tomme eller kun 'whitespace'. I det virkelige eksempel kontrolleres der også for om data kan parses til en numerisk værdi, hvis det skal anvendes således.

Sagsbehandlerportalens grafiske brugergrænseflade bliver demonstreret i afsnit 8.2.

```
@Component("portal-kundeValidator")
public class KundeValidator implements Validator{

    @Override
    public boolean supports(Class clazz) {
        return Kunde.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "kundeId", "", "CPR nummer skal være udfyldt!");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "navn", "", "Navn skal være udfyldt!");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "adresse", "", "Adresse skal være udfyldt!");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "betalingsevne", "", "Betalingsevne skal være udfyldt!");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "betalingsvilje", "", "Betalingsvilje skal være udfyldt!");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "fordring", "", "Fordring skal være udfyldt!");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "behandlesManuelt", "", "BehandlesManuelt skal være udfyldt!");
    }

}
```

Figur 6.18: Udsnit af KundeValidator, der anvendes af Spring Frameworket til validering af Kundeobjektet.

Kapitel 7

Test

For at sikre, at den leverede prototype er af en vis kvalitet, er der under hele udviklingsprocessen blevet udarbejdet tests, der har til formål at sikre dette. Det væsentligste har været at sikre, at prototypen understøtter de tre typer af scenarier, der hver blev beskrevet i afsnit 2.4.

Den anvendte teststrategi har været en blanding mellem top-down og bottom-up. Dette skal forstås på den måde, at der relativt tidligt i udviklingsprocessen blev udarbejdet tests til at dække den overordnede funktionalitet, der kræves af de tre scenarier. Disse scenarier er dækket med integrationstests. Integrationstestene har her til formål at afprøve alle enhederne samlet i den komplette prototype, således er både HændelsesHåndtereren, Regelmotoren samt Persisteringslaget dækket af disse tests.

Herefter blev enhederne udarbejdet samt testet med unittests. Unittestene har her til formål at afprøve den specifikke enhed isoleret fra andre enheder. Alle enheder i prototypen med undtagelse af webapplikationen er dækket af disse unittest. Webapplikationen er dog dækket af manuelle Black-box-tests, der udføres ved interaktion med sagsbehandlerportalen i en af integrationstestene.

JUnit Frameworket anvendes både ved opsætning af integrationstest samt enhedstest. Ved integrationstest konfigureres hele miljøet med Spring. Der anvendes derfor også en rigtig database, så prototypen bliver testet op imod denne.

I databasen ligger sportyperne samt tre kunder. Ved enhedstest derimod er det alene den enhed der afprøves, som bliver instantieret. Enheden isoleres fra andre enheder ved at anvende Mockito Frameworket til at 'mocke' disse enheder. Mockito blev beskrevet i afsnit 4.5.

Da Maven anvendes til at bygge projektet, vil alle tests blive udført ved hver enkelt byg af projektet, hvilket sikrer, at fejl i projektet vil blive opdaget tidligt, hvis disse er dækket af tests.

Ovenstående har været en naturlig del af udviklingsprocessen for at sikre, at den skrevne kode opfylder den ønskede funktionalitet. Alle enhedstest samt integrationstest, herunder de tre scenarier, eksekveres uden fejl i prototypen. De er at finde i hvert enkelt Maven projekt i mapperne "src/test/java". Systemet består af lidt under 100 klasser og testes af lidt over 30 testklasser. Testene dækker tilsammen 73 procent af koden. Procententallet er fundet med Maven-plugin'et Cobertura, der ud fra de skrevne unittests udregner dækningen af koden. Procenttallet er her acceptabelt, idet så meget som muligt af logikken i systemet bliver testet.

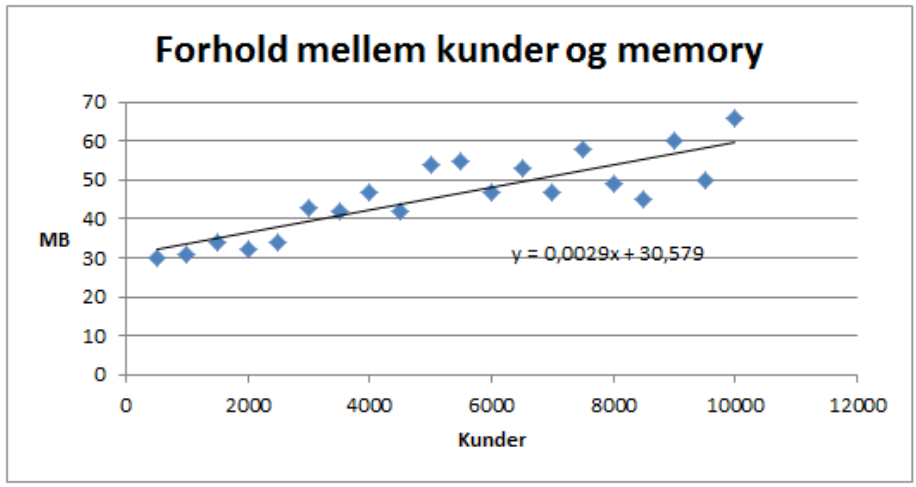
Det var tiltænkt, at EFI skulle kunne håndtere omkring 600.000 kunder. Derfor har det været vigtigt, at løsningen implementeret med en regelmotor også kunne opfylde dette krav. Første prototype, som blev beskrevet i afsnit 5.1, kunne ikke håndtere den mængde kunder. I afsnit 7.1 bliver Anden prototype testet for at kunne redegøre for om denne opfylder kravet.

7.1 Test af performance

For at teste om prototypen kan opfylde kravet om håndtering af 600.000 kunder, udføres der en test, som har til formål at give en forestilling om, hvor mange kunder systemet kan håndtere.

Testen der udføres er en test lignende den, som blev udført på Første prototype, der er beskrevet i afsnit 5.1. Testen opretter løbende nye kunder, samt genererer tilfældige hændelser for de eksisterende kunder. Der oprettes ca. 10 kunder og genereres ca. 20 hændelser per sekund.

Memory var en af de begrænsende faktorer i Første prototype, så denne test skal klarlægge, hvor meget memory der kræves for at håndtere den ønskede mængde kunder i Anden prototype. Testen er foretaget med 10.000 kunder. Resultatet af målinger vedrørende memory i forhold til antallet af kunder ses i figur 7.1.



Figur 7.1: Forhold mellem antallet af kunder og krævet memory.

Som det ses i figur 7.1 er den anvendte mængde af memory betragteligt mindre end i Første prototype. For at håndtere de 600.000 kunder kræves der nu lidt under 2 GB. I de målinger der blev foretaget, var det stort set kun ved Hibernates indbyggede Cache, at mængden af anvendt memory var stigende, så det formodes, at størstedelen af disse 2 GB vil blive anvendt af Hibernate. Dette øger performance i tilfælde af, at samme kunde ofte anvendes flere gange. Dette er dog ikke tilfældet ved denne test, da kunderne udvælges tilfældigt. Hibernates cache kan derfor med fordel deaktiveres.

Designet af systemet sikrer desuden, at der er mulighed for at skalere både vertikalt og horisontalt. Det er altså muligt at give flere ressourcer til en enkelt maskine, og det er muligt at sætte flere maskiner op med hver sin instans af regelmotoren, idet denne anvendes på en tilstandsløs måde. En sådan skalering kan være en nødvendighed, hvis det ønskes at øge 'throughput'.

Kapitel 8

Produktet

Produktet er en prototype, der viser hvordan det er muligt at implementere et hændelsesbaseret system, hvor hver hændelse bliver evalueret ud fra regler i en regelmotor.

Prototypen styrer automatisk inddrivelsesforløbet overfor en kunde, når denne har et udestående til SKAT. Til administration af prototypen, er der lavet en webapplikation ved navn Sagsbehandlerportalen. Via denne er det muligt for en sagsbehandler at oprette nye kunder, samt ændre den anbefalede inddrivelsesstrategi for hver enkelt eksisterende kunde. Hvordan det er tænkt, at en sagsbehandler skal navigere rundt på siden, vil blive beskrevet i afsnit 8.2.

Hvad der skal til for at få en demo til afprøvning op at køre, bliver beskrevet i afsnit 8.1.

8.1 Installation

Produktet er en webapplikation. Denne kan deployes på en hvilken som helst applikationsserver, med små justeringer i Spring konfigurationen. Webapplikationen er pakket i en Webapplication ARchive, WAR, fil. Denne er at finde i

roden af CD'en ved navn skat-efi-webapp.war. For at gøre det lettere at komme i gang med afprøvning af demoen, er der udarbejdet en demo, der hurtigt kan installeres med nogle enkelte trin. Denne er dog kun en demo, og vil ikke fungere i produktion, da den anvender en in-memory database. Demoen startes i en letvægts-applikationsserver ved navn Jetty. Følgende trin skal udføres for at afprøve demoen:

1. Kopier filen SkatEfiDrools-in-memory-db.zip fra roden af CD'en til din lokale maskine.
2. Udpak zip filen i en hvilken som helst folder.
3. Naviger til roden af den udpakkede folder og dobbeltklik på start.bat (Windows) eller start.sh (Linux).
4. Når serveren kører, kopier følgende link til din browser: `http://127.0.0.1:8080/skat-efi-webapp/`

For at køre demoen kræves det, at Java er installeret på den lokale maskine samt, at JAVA_HOME er sat til at pege derpå. Det kræves endvidere, at en browser er tilgængelig på maskinen.

8.2 Rundvisning af Sagsbehandlerportalen

I sagsbehandlerportalen er det muligt for en sagsbehandler at oprette nye kunder, ændre informationer for en eksisterende kunde, udsende hændelser vedrørende en kunde samt danne sig et overblik over en aktuel kundesag. Det er endvidere muligt at ændre eksisterende sportyper. Hvordan dette udføres, bliver beskrevet i de følgende underafsnit.

8.2.1 Opret kunde

En kunde oprettes i sagsbehandlerportalen ved at vælge "Opret Kunde" i menuen. Herefter kommer skærmbilledet, som er illustreret i figur 8.1, frem. I skærmbilledet indtastes informationer om kunden, hvorefter der klikkes på "Opret Kunde".

Når der er klikket på "Opret Kunde" vil de indtastede informationer blive valideret. Ved fejl i validering udskrives passende fejlmeddelelser som vist i figur 8.2. En sådan validering bliver anvendt på alle input via sagsbehandlerportalen.

Figur 8.1: Skærmbillede for "Opret Kunde"dialogen i sagsbehandlerportalen.

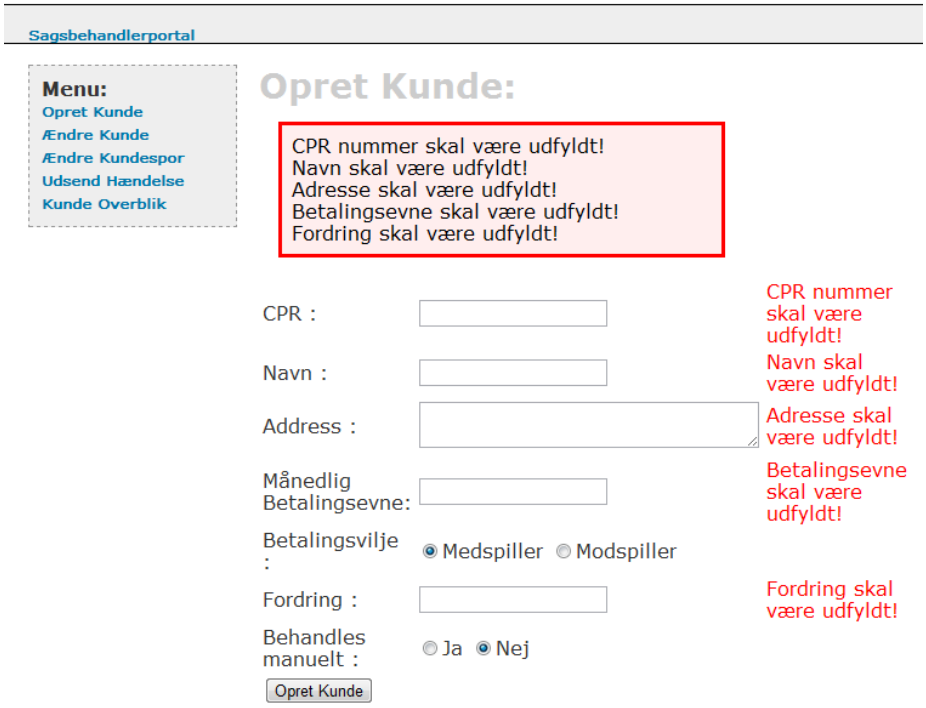
Lykkes det at validere de indtastede kundedata, bliver en kunde med de indtastede data oprettet, og der gives besked herom. En sådan besked er illustreret i figur 8.3.

8.2.2 Ændre kunde

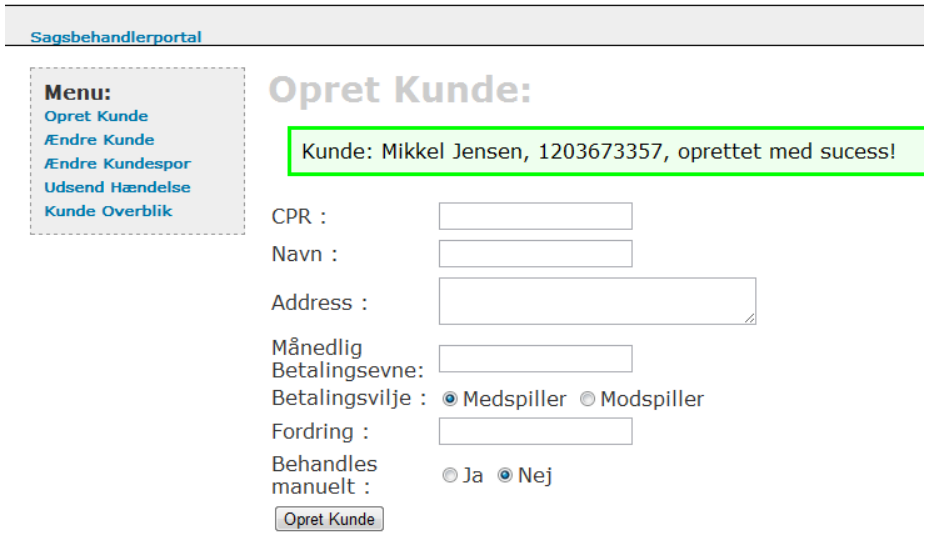
Informationer vedrørende en kunde ændres i sagsbehandlerportalen ved at klikke på "Ændre kunde" i menuen. Herefter er det muligt, via en Drop-down menu, at vælge den kunde, hvor det ønskes at ændre informationer. Et eksempel på en sådan dialog er illustreret i figur 8.4.

Alle steder hvor sagsbehandlingen vedrører en specifik kunde, vil en sådan Drop-down menu blive anvendt til at vælge kunden. Dette er selvfølgelig ikke designet for en endelig løsning. I en færdig løsning ville en dialog til søgning efter kunder være mere passende.

Når kunden er valgt i dialogen, som er illustreret i figur 8.4, ses en ny dialog. Denne dialog indeholder kundens nuværende informationer, og det er herfra



Figur 8.2: Skærmbillede for "Opret Kunde"dialogen i sagsbehandlerportalen, hvor validering er fejlet.



Figur 8.3: Skærmbillede for "Opret Kunde"dialogen i sagsbehandlerportalen, efter kunde er oprettet.

Figur 8.4: Skærmbillede for "Ændre Kunde"dialogen i sagsbehandlerportalen, hvor kunden vælges.

Figur 8.5: Skærmbillede for "Ændre Kunde"dialogen i sagsbehandlerportalen, hvor en specifik kunde er valgt.

muligt at ændre disse informationer. Et eksempel på denne dialog ses i figur 8.5.

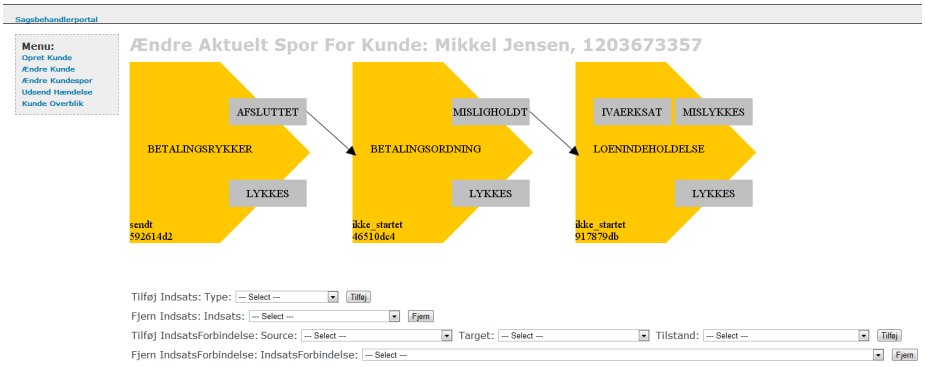
De indtastede informationer skal selvfølgelig også overholde kravet for validering, ellers vises der ligeledes passende fejlmeddelelser. Lykkes det at ændre kundens informationer gives der besked herom, præcis som ved oprettelse af en kunde.

8.2.3 Ændre kundespor

En kundes spor ændres i sagsbehandlerportalen ved at klikke på "Ændre Kundespor " i menuen. Herefter vælges den kunde, hvor sporet ønskes ændret. Når kunden er valgt, fremkommer den dialog, hvor det er muligt at ændre sporet. Dialogen er illustreret i figur 8.6.

Det er muligt at tilføje en indsats ved blot at vælge, hvilken indsatstype, der ønskes i Drop-down menuen. Når indsatstypen er valgt klikkes der på "Tilføj". Det er ligeledes muligt at fjerne en indsats. Her vælges indsatsen, hvor der efter trykkes på "Fjern".

Når det ønskes, at en indsats skal vente på resultatet af en anden indsats, tilføjes en indsatsforbindelse. Den indsats, hvor resultatet afventes, skal vælges i Drop-down menuen Source. Den indsats der afventer resultatet, skal vælges i Drop-down menuen Target. I Drop-down menuen Tilstand vælges den tilstand som indsatsen, der afventer ønsker, at den igangværende indsats afslutter i. Herefter klikkes der på "Tilføj".



Figur 8.6: Skærmbillede for "Ændre Kundespor"dialogen i sagsbehandlerportalen, hvor en specifik kunde er valgt.

Ønskes det at fjerne en indsatsforbindelse gælder samme procedure som ved fjernelse af en indsats.

Et eksempel på en tilføjet indsats samt indsatsforbindelse er illustreret i figur 8.7. Her er der siden figur 8.6 blevet tilføjet en indsats af typen Kreditoplysningsbureau. Hvorefter det er defineret med en indsatsforbindelse, at kunden skal indberettes til kreditoplysningsbureauet i tilfælde af, at kunden misligholder betalingsordningen.

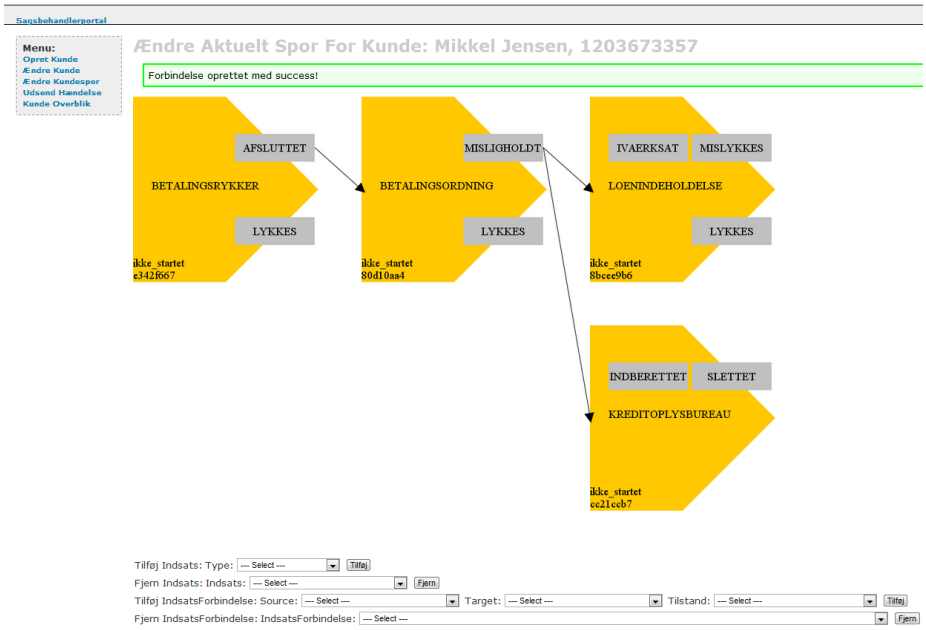
Samme fremgangsmåde gør sig gældende, når det ønskes at ændre en eksisterende sportype. Her trykkes der blot på "Ændre Sporskabelon"og en sportype vælges, på samme måde som en kunde før blev valgt i Drop-down menuen.

8.2.4 Udsend hændelse

Størstedelen af hændelser vil normalt blive modtaget fra eksterne systemer. Der er dog enkelte hændelser, der også er tiltænkt udsendt af en sagsbehandler via sagsbehandlerportalen.

I prototypen er det muligt gennem sagsbehandlerportalen at udsende alle former for hændelser. Dette skyldes, at det giver mulighed for bedre at simulere hvordan systemet vil fungere i drift.

Når en hændelse vedrørende en kunde ønskes udsendt, klikkes der på "Udsend Hændelse" i menuen. Herefter vælges kunden. Når kunden er valgt, vises dialogen



Figur 8.7: Skærmbillede for "Ændre Kundespor"dialogen i sagsbehandlerportalen, hvor en indsatsforbindelse er blevet tilføjet.

Figur 8.8: Skærmbillede for "Udsend Hændelse"dialogen i sagsbehandlerportalen, hvor en specifik kunde er valgt.

illustreret i figur 8.8.

Det er forskelligt hvilke informationer der udsendes sammen med en hændelse, men disse udfyldes altid inden der trykkes "Udsend". I tilfælde af, at en hændelse ikke er forsynet med de krævede informationer, ved tryk på "Udsend", da vil en passende fejlmeddelelse vises på skærmen.

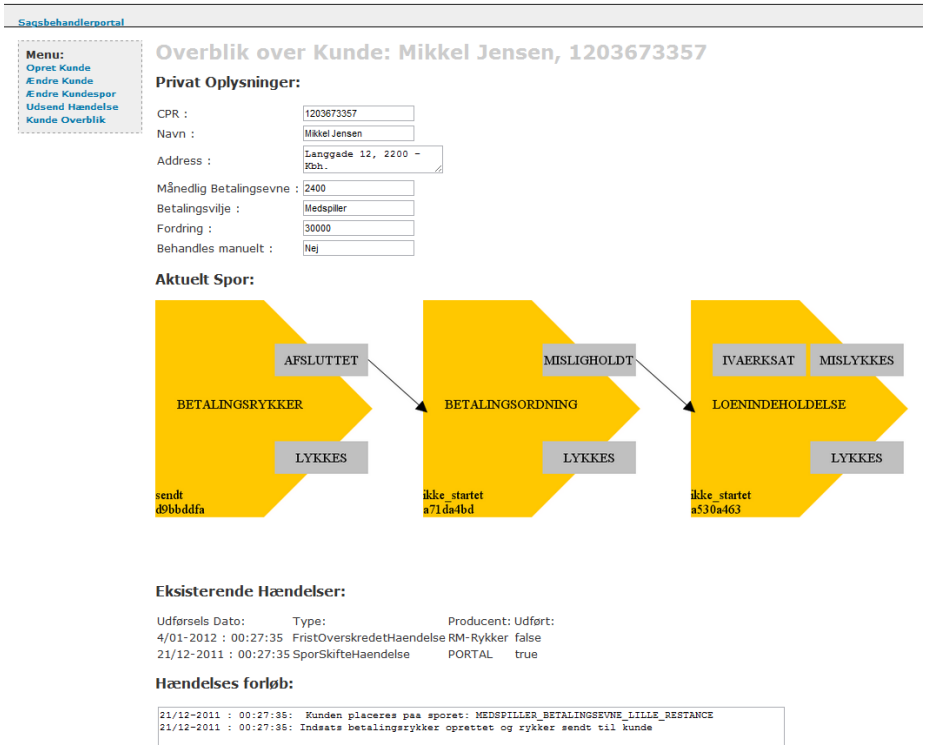
8.2.5 Overblik over kundesag

I sagsbehandlerportalen er det muligt at få et samlet overblik over kundens sagsforløb. Dette fås ved at klikke på "Kunde Overblik". Herefter vælges den kunde, der ønskes overblik over. Når kunden er valgt bliver dialogen illustreret i figur 8.9 vist.

I dialogen "Kunde overblik" er det ikke muligt at ændre data. Dialogen har udelukkende til formål at give et samlet overblik over kundens sag.

Her vises kundens privatoplysninger samt kundens aktuelle spor, der kunne ændres i henholdsvis dialogerne "Ændre Kunde" og "Ændre Kundespor". I Kundens aktuelle spor er det muligt at aflæse, hvilken tilstand indsatserne er i, på nuværende tidspunkt.

Der vises desuden hvilke hændelser, der er modtaget og behandlet vedrørende kunden, samt hvilke fremtidige hændelser, som senere vil blive udsendt vedrø-



Figur 8.9: Skærmbillede for "Kunde Overblik"dialogen i sagsbehandlerportalen, hvor en specifik kunde er valgt.

rende kunden, og de igangværende indsatser, der omhandler kunden.

I boksen Hændelsesforløb gives en detaljeret beskrivelse af hele forløbet i den specifikke kundesag. Eksempelvis en dato for hvornår en rykker er udsendt til kunden.

Kapitel 9

Konklusion

En prototype, der har til formål at vise hvordan regelbaseret forretningslogik, kan implementeres med en regelmotor, er blevet udarbejdet. Regelmotoren i prototypen er implementeret med Drools Expert. Undervejs i forløbet har Drools Expert vist både gode og knap så gode sider.

Blandt de gode kan nævnes, at Drools Expert har et stort community omkring sig, og er veldokumenteret. Det har derfor været forholdsvis nemt at sætte sig ind i og komme i gang med. Reglerne skrives i en naturlig og let læselig syntaks, og hvordan selve regelmotoren evaluerer reglerne har været meget intuitivt. Selve udviklingsmiljøet har også fungeret udmærket, og med tilstrækkelig funktionalitet til at debugge reglerne med.

Blandt de knap så gode kan nævnes, at Drools Expert til tider virker en smule ustabilt. Dette har resulteret i, at et versionsskifte til tider har været eneste udvej. Udviklingen startede således med version 5.1.0, men til slut endte den med at være 5.3.0.Final. I skrivende stund er version 5.4.0.Beta1 udkommet. Der kommer således ofte fejlretninger, hvilket er positivt. Det vil dog stadig blive kategoriseret som udvikling på kanten. Der findes en kommerciel udgave af hele Drools-platformen ved navn JBoss BRMS, men om denne er mere stabil, vides ikke.

Blandt de tanker og meninger, der har indfundet sig ved udvikling af denne

prototype med Drools, men som menes at gælde for regelmotorer generelt, kan det nævnes, at regelmotorerne syntes at være designet til at evaluere mange komplekse regler baseret på 'få' fakta. Dermed er den ikke designet til mange data på én gang. Det er dog generelt muligt, at hente data udefra, når der er behov for det. Dette vil dog medføre, at reglerne evalueres oftere end nødvendigt.

Grundet størrelsen af dette projekt, virker hele strukturen omkring en regelmotor, som at skyde gråspurve med kanoner. I projekter hvor forretningsreglerne ofte ændres eller udvides til at bestå af flere komplekse regler, syntes krudtet dog ikke spildt. Dette skyldes, at det her vil være lettere at indføre ændringer, i forhold til en traditionel implementering.

I fremtidige projekter kan en regelmotor derfor med fordel anvendes, hvor forretningsreglerne hyppigt ændres, da det her vil være dyrt at vedligeholde dette i traditionel kode. Det er dog vigtigt fra start, at vurdere om ændringer i reglerne også vil medføre ændringer i domænemodellen. Her vil der nemlig alligevel skulle ændres i koden, og effekten er derfor måske ikke så stor som ønsket. Det er også vigtigt at holde sig for øje, at denne løsning kun kan anvendes, idet det er muligt at isolere data. Forstået på den måde, at hver kundesag er uafhængig af de andre kundesager.

Det blev også konstateret, at regelmotoren anvender en del ressourcer. Dette udelukker en anvendelse af regelmotorer, hvor dette er en begrænsende faktor.

En regelmotor kan give mange fordele, hvis den anvendes i det rigtige tilfælde, men kan ligeledes føre til ulemper i de forkerte. Derfor skal disse vurderes i hvert enkelt tilfælde.

Kapitel 10

Referencer

Business Process Management Systems - Strategy and Implementation. 2005, af James Chang.

Business Process Management - Practical Guidelines to Successful Implementations. 2008, af John Jeston og Johan Nelis.

How to Build a Business Rules Engine - Extending Application Functionality Through Metadata Engineering. 2004, af Malcolm Chisholm.

JBoss Drools Business Rules 2009, af Paul Browne.

Drools JBoss Rules 5.0 - Developer's Guide. 2009, af Michal Bali.

Drools Expert User Guide Hentet 23. December, 2011, fra: `http://docs.jboss.org/drools/release/5.3.0.Final/drools-expert-docs/html_single/index.html`

Drools Fusion User Guide Hentet 23. December, 2011, fra: `http://docs.jboss.org/drools/release/5.3.0.Final/drools-fusion-docs/html_single/index.html`

Guvnor Manual Hentet 23. December, 2011, fra: `http://docs.jboss.org/drools/release/5.3.0.Final/drools-guvnor-docs/html_single/index.html`

Hibernate Reference Documentation Hentet 23. December, 2011, fra: `http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html_single/`

Spring Documentation Hentet 23. December, 2011, fra: `http://www.springsource.org/documentation`

Mockito Framework Hentet 23. December, 2011, fra: `http://code.google.com/p/mockito/`

Quartz Documentation Hentet 23. December, 2011, fra: `http://quartz-scheduler.org/documentation`