NTNU
Det skapende universitet

Regelbaserte Ekspertsystemer

Veien til en smidigere forretningsdrift

Marit Hegle

Master i datateknikk
Oppgaven levert: Juni 2007
Hovedveileder: Harald Rønneberg, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Oppgavetekst

Hensikten med denne oppgaven er å gi en innføring i regelbaserte ekspertsystem og hvordan man kan bruke disse for å oppnå en smidigere forretningsdrift, oppgaven er delt i følgende to deler:

DEL I:
Først skal det utføres en studie av regelbaserte ekspertsystem. Denne skal inkludere deres oppbygning, bruksområder og fordeler.

DEL II:
Deretter skal det velges ut en av aktørene på markedet. Det skal gis en innføring i dette produktets elementer og egenskaper. Så skal det implementers et case, caset skal utarbeides i samarbeid mellom student og Abeo AS. Implementeringen av caset skal se på hvor velegnet denne aktøren er til å la bruker innføre caset og hvilke verktøy og muligheter som er tilrettelagt.

Oppgaven gitt: 19. januar 2007
Hovedveileder: Harald Rønneberg, IDI

Sammendrag

Denne oppgaven omhandler regelbaserte ekspertsystemer og en implementering av et slikt system. Hensikten med oppgaven er å gi en innsikt i hvordan slike system kan brukes til å oppnå en smidigere forretningsdrift ved å forenkle interne prosesser.

Et ekspertsystem er et kunnskapsbasert system som kan gi råd eller fatte beslutninger i et smalt definert område på nært nivå med en menneskelig ekspert. I det virkelige liv bruker eksperter praktisk erfaring og forståelse av problemet for å finne snarveier til en løsning, dette kan være både tommelfingerregler og heuristiske metoder. Ekspertsystem gir oss mulighet til å benytte de samme metodene i et datasystem.

Et av de viktigste elementene for å oppnå en smidigere forretningsdrift og for å kunne følge markedets svingninger, er å endre regler og standarder innad i bedriften så ofte som nødvendig. Tradisjonelle metoder for å endre regler, spesielt i store og omfattende systemer, krever store ressurser. Ved bruk av et håndteringssystem for forretningsregler, kan denne prosessen forenkles betraktelig.

Oppgaven vil gi en innsikt i de regelbaserte ekspertsystemenes historie, bruksområder og fordeler. For å få til dette har jeg også tatt med litt informasjon om generelle ekspertsystem.

Jeg har implementert en løsning ved bruk av regelmotoren ILOG Rules for .NET. Løsningen er en validering av ordrer ved bruk av .NET og XML. Dette er en regelmotor som lar deg enkelt skrive regler og styre flyten av disse ved hjelp av sitt Rule Studio. I kombinasjon med XML og Visual Studio tilbys en helhetlig løsning, som lar seg integrere med blant annet BizTalk Server og Microsoft Office.

Forord

Denne hovedoppgaven er gjennomført i tiende semester ved Sivilingeniørstudiet ved institutt for datateknikk og informasjonsvitenskap ved NTNU. Oppgaven er skrevet i samarbeid med Abeo AS, der Abeo har stilt med ressurser i form av veiledning, arbeidsplass, case og nødvendig programvare og maskinvare.

Jeg ønsker å rette en stor takk til min veileder i Abeo, Sigurd Ringbakken. Han har vært til stor hjelp med både valg av case for implementering, framdrift i oppgaven og generell veiledning. Jeg ønsker også å takke Jan Fredrik Øveråsen ved Abeos osloavdeling, han har kommet med en rekke nyttige innspill med sin ekspertise innen regelmotorer. En spesiell takk til Lionel Mace ved ILOGs kontor i Frankrike. Han stilte til et møte med meg i Oslo hvor vi diskuterte ILOG Rules for .NET som produkt samt av vi gjennomgikk relevansen av mitt case, han har også vært svært behjelpelig på å svare på mine spørsmå på e-post. En takk skal også rettes til Abeo AS som betalte opphold og kursavgift i forbindelse med et 5 dagers kurs i Oslo for å lære meg .NET, noe som var til stor hjelp i implementeringen av case. Takk til veileder ved NTNU, Harald Rønneberg, for nyttig input angående struktur på oppgaven. Helt til slutt ønsker jeg å rette en takk til min bror, Ole Andreas, for nyttige innspill med siste finpuss på rapporten.

Alle kilder som er referert til via en nettside, er sist aksessert 14.06.2007

Oppgavebeskrivelse

Hensikten med denne oppgaven er å gi en innføring i regelbaserte ekspertsystem og hvordan man kan bruke disse for å oppnå en smidigere forretningsdrift, oppgaven er delt i følgende to deler:

DEL I:

Først skal det utføres en studie av regelbaserte ekspertsystem. Denne skal inkludere deres oppbygning, bruksområder og fordeler.

DEL II:

Deretter skal det velges ut en av aktørene på markedet. Det skal gis en innføring i dette produktets elementer og egenskaper. Så skal det implementers et case, caset skal utarbeides i samarbeid mellom student og Abeo AS. Implementeringen av caset skal se på hvor velegnet denne aktøren er til å la bruker innføre caset og hvilke verktøy og muligheter som er tilrettelagt.

Leserveiledning

Den resterende delen av rapporten er bygd opp som følger:

Kapittel 1 Kapitlet starter med en innføring av hva et ekspertsystem er. Deretter gås det nærmere inn på de regelbaserte ekspertystemene og hvordan de er bygd opp, fordeler med et slikt system og utviklingen fra den spede start og fram til i dag.

Kapittel 2 Beskrivelse av det regelbaserte ekspertsystemet ILOG Rules for .NET, systemet jeg har brukt til å implementere et ordresystem i kapittel 4.

Kapittel 3 I kapittel 3 kommer en beskrivelse av caset som ble implementert, hvordan gjennomføringen foregikk og hvilke resultat som foreligger. Kapitlet avsluttes med en evaluering av funn, både fra implementeringen og literaturstudiet i de to foregående kapitlene.

Kapittel 4 Her kommer konklusjon og forslag til videre arbeid.

Innhold

Figurer

Kapittel 1

Ekspertsystem

Dette kapitlet vil hovedsakelig gi en innføring i de regelbaserte ekspertsystemene. For å få en helhetlig oversikt over deres historie samt bruksområder, blir det også gått en del innpå ekspertsystem generelt og hvordan disse skiller seg fra konvensjonelle system.

Det vil bli gått nærmere inn på strategier, problemer og fordeler ved bruk av regelbaserte ekspertsystem.

1.1 Innføring i Ekspertsystem

Et ekspertsystem er et kunnskapsbasert system som kan gi råd eller fatte beslutninger i et smalt definert område på nært nivå med en menneskelig ekspert. Det blir definert som "system av programvare som simulerer, så nært som mulig, utputten av et svært kunnskapsrikt og erfarent menneske som fungerer i et problemløsende modus i et spesifikt problemområde" [7]. Det finnes to slike typer system, den ene varianten fatter beslutninger. Dette er først og fremst brukt ved kontroller av prosesser og applikasjoner, i tillegg til system som fungerer som støtte til beslutninger. Den andre typen ekspertsystem gir råd, men de fatter ikke selv beslutningene. Her skal det nevnes at det er svært sjelden et ekspertsystem brukes alene til å fatte beslutninger, det er mer vanlig at det brukes i kombinasjon med andre produkter for å bistå forretningsbrukerne [8].

Ekspertsystem er bygd for å brukes i et smalt, spesialisert domene. Deres oppgave er å etterligne resoneringen til en menneskelig ekspert og dermed også komme fram til de samme løsningene. Følgelig vil det være naturlig

å måle kvaliteten på systemet i forhold til prestasjonen et menneske ville hatt i den samme gitte situasjon. Det er derfor svært viktig at systemet har en meget høy kvalitet på oppgavene det utfører. Fra en brukers ståsted er det ofte ikke av mye verdi med et svar som ikke er korrekt, uansett hvor raskt systemet produserte svaret. Likevel må man ikke minske viktigheten av hastigheten til systemet, mange av dagens ekspertsystemer er kritiske sanntidssystemer brukt innen sykehus, militæret og kjernekraftverk. I disse systemene er det høye krav til hastigheten på systemet, på toppen av kravene om korrekt produserte løsninger.

I det virkelige liv bruker eksperter praktisk erfaring og forståelse av problemet for å finne snarveier til en løsning, dette kan være både tommelfingerregler og heuristiske metoder. Sistnevnte er en metode som gir resultater uten at en vet om det skyldes tilfeldigheter. Selv om man ved bruk av heuristikk ikke alltid kan være like sikker på om resultatet er 100 % korrekt, vil svaret likevel være det man antar er den beste løsningen ut ifra de gitte data. I likhet med deres menneskelige motparter, bør ekspertsystem benytte seg av heuristikk for å veilede tankegangen og dermed redusere søkerommet for en løsning [10].

En helt unik egenskap ved ekspertsystem er deres mulighet til å forklare hvordan de kom fram til løsningen sin. Dette åpner for at systemet kan se over sin egen tankegang og dermed rettferdiggjøre sine svar. Det er varierende grad av nødvendighet for forklaring i de ulike systemer, avhengig av hvem som bruker systemet og til hvilket formål det blir brukt. Noen ganger brukes systemet av eksperter slik at de konklusjoner som blir nådd er selvforklarende for brukeren, da kan en enkel sporing av kjørte regler være nok. Ekspertsystem brukt til å bistå i beslutningstaking krever en grundigere oppfølging og forklaring, dette grunner i at en gal beslutning kan bli svært kostbar for det firmaet som bruker disse beslutningene. Følgelig må man vurdere krav til forklaringsgrad ulikt for hvert system.

En forklaring foregår ved at de reglene som er kjørt i løpet av en sesjon blir sporet. Dette er naturlig nok en forenklet forklaring, en sekvens av kjørte regler kan ikke fullstendig rettferdiggjøre konklusjonene som blir nådd. En virkelig eller menneskelig forklaring er enda ikke mulig fordi det krever en grunnleggende forståelse av domenet. Ekspertsystemets kunnskap er begrenset til de regler som ligger inne i kunnskapsbasen, utover dette kan det ingenting. En mulig løsning er å legge til noen fundamentale prinsipper slik at hver regel får en tilhørende forklarende tekst. Dette er trolig så nært man kan komme en forklaring.

Aksept av ekspertsystem i kommersielle applikasjoner har ikke gått hurtig, dette skyldes hovedsakelig at man ikke har stor nok tiltro til påliteligheten

og kvaliteten til systemet [13]. I det virkelige liv stoler vi på eksperters uttaleleser, selv om de kun er mennesker og vi vet at de kan ta feil. På samme måten bør man også ta høyde for at ekspertsystem iblant kan ta feil, de er tross alt bygd på menneskelig kunnskap.

I alle bedrifter finnes det regler som man må forholde seg til, dette være seg store etablerte firma eller små nyetablerte. Regler kan omfatte alt som omhandler bedriftens drift og daglige virksomhet. Dette kan være alt fra at kunder må være fylt 18 år før de får kjøpe alkohol, at kunder må ha handlet varer over et visst beløp før de mottar et bonuskort eller noe annet som omhandler en avgjørelse eller beslutning som må tas. Reglene utarbeides på grunnlag av framgangsmåter, gjeldende praksis og prosedyrer som finnes i organisasjonen.

Ekspertsystemet skiller mellom den deklarative kunnskapen og koden som kontrollerer kjøringen av reglene. Kunnskapsbasen inneholder reglene, den deklarative kunnskapen, systemet består av. Det er svært essensielt at kvaliteten på innholdet i denne basen er høyt. Dersom det ikke ligger god nok informasjon inne i systemet vil det heller ikke ha mulighet til å produsere gode løsninger, jamfør det kjente "Garbage in - garbage out", søppel inn - søppel ut.

Separasjon av kunnskap og kontroll gir to store fordeler. Det garanterer at systemet kan respondere på en mer fleksibel måte, i tillegg til at utvikleren kan legge det meste av sitt fokus på spesifikasjonen av den deklarative kunnskapen. Dermed kan utviklingen av en effektiv algoritme for søk overlates til regelmotoren. Fordelene med dette er flere. For det første går utviklingen glattere når utviklerne ikke behøver å få tak i ekspertenes kunnskap i tillegg til å samtidlig sette sammen denne kunnskapen med resten av systemet i en trinnvis prosess. For det andre kan eksperter ha problemer med å legge fram all deres kunnskap på strak arm, informasjonen kommer lettere fram dersom de har mulighet til å diskutere hvordan de har løst tidligere problemer. Ved bruk av et regelbasert ekspertsystem åpnes muligheten for utvikler og ekspert til å samarbeide om å få fram viktig informasjon og gradvis, i usammenhengende deler, legge den inn i kunnskapsbasen etter hvert som den kommer frem.

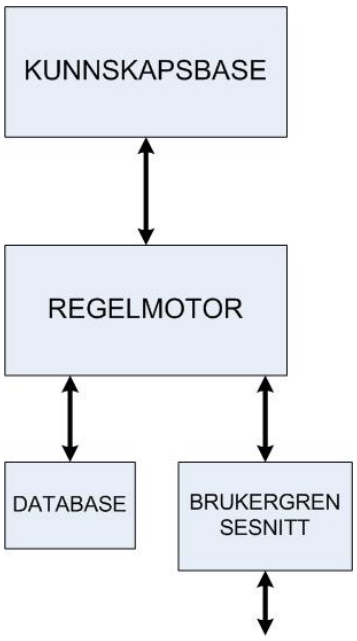
Utviklerne får en del nye utfordringer i forbindelse med innføringen av et regelbasert ekspertsystem. I tillegg til å være ansvarlig for å kode plattformen som implementerer de varige elementene i applikasjonen, må de nå i tillegg samarbeide med forretningsbrukerne. Utviklerne må også skrive og teste forretningsregler, dette er spesielt viktig i tidlige faser av utviklingen av applikasjonen, før forretningsbrukerne tar over med å fullføre arbeidet

med å legge inn reglene i systemet. Når dette er gjort, må så utvikleren til slutt integrere forretningslogikken som er implementert i forretningsreglene med resten av applikasjonen. Dette vil typisk inkludere å samle inn regler basert på kontekst, samle inn data som reglene skal prosessere, invokere en lokal eller fjern regelmotor og bruke resultatene fra dataprosesseringen til å fortsette prosessering med fastsatt logikk.

Et slikt regelbasert system egner seg best når kunnskapen som brukes naturlig forekommer i en form som er passende for regler, når programmets kontroll er ekstremt kompleks eller når et program er forventet å bli betydelig modifisert over en lang tidsperiode [8].

Et godt håndteringssystem for forretningsregler bør tillate applikasjoner å ta i bruk en tjenesteorientert arkitektur (SOA) [3]. Derfor bør regelmotoren presentere seg selv som en tjeneste til applikasjoner, mens applikasjonene selv bør kunne brukes som tjenester.

1.2 Arkitektur



Figur 1.1: Komponenter i et ekspertsystem [13]

Den viktigste arkitekturiske egenskapen til et ekspertsystem er at den skiller fakta fra implementasjonen. Kunnskapsbasen er lageret hvor kunnskapen i ekspertsystemet ligger lagret, denne blir gjerne referert til som regelbasen

Databasen fungerer som en global database av symboler som representerer fakta og påstander om problemet. Fakta er instanser av objekt, disse kan representere enten fysiske objekter eller fakta som er relatert til domenet, eller de kan representere konseptuelle objekter. Et konseptuelt objekt er for eksempel mål relatert til en problemløsningsstrategi. Dersom man skal håndtere regler kan det virke som om det eneste fornuftige alternativet er å lagre de i en type sentral database. Det vil gi tydelige fordeler med hensyn til vedlikehold når man lagrer reglene i et lag separat fra både applikasjoner og databaser som finnes i organisasjonen.

I tillegg til kunnskapsbasen og databasen, inneholder et regelbasert ekspertsystem ytterlige tre komponenter: regelmotoren, et sett av forklarende fasiliteter og et grensesnittet mot brukeren. De forklarende fasiliteter legger til rette for at en bruker skal kunne spørre systemet hvordan et bestemt mål er nådd. Kort fortalt er det regelmotoren som håndterer tankegangen hvor ekspertsystemet når en løsning. Den linker reglene i kunnskapsbasen med fakta den finner i databassen. Regelmotoren vil bli gjennomgått i nærmere detalj i kapittel 1.4.

Regelmotoren er en algoritme som dynamisk kontrollerer systemet når det søker i kunnskapsbasen. I et regelbasert ekspertsystem er kontroll basert på en jevnlig revaluering av tilstanden til dataene, ikke av en statisk kontrollstruktur til programmet. Derfor sier vi at beregningene i et produksjonssystem er datadrevne istedenfor instruksjonsdrevne. Følgelig er den eneste muligheten for regler å kommunisere seg imellom ved hjelp av data. Alle regler har et navn, likevel kan ikke reglene referere andre regler, den eneste grunnen til at de har et navn er for å hjelpe programmereren eller andre for å lettere få oversikt.

Regelbaserte systemer kommer i to varianter, de enkle grupperer alle reglene sammen i ett enkelt sett og undersøker alle samtidig. Den andre typen deler reglene inn i delsett, arrangerer disse delsettene inn i trær og undersøker de så i henhold til en bestemt søkestrategi. I slike tilfeller er det svært fordelaktig å bruke et strukturert regelsystem hvor man kan dele inn reglene i grupper ved hjelp av en trestruktur. Da behøver systemet kun å gå igjennom de regler som er relevant for problemet det skal løse.

Data kan legges til, endres eller slettes i databasen. Dataelementer blir ofte lagt til som et resultat av en lang løsningsslutning, mens en endring øker eller korrigerer eksisterende kunnskap. Grunner til at fakta blir slettet inkluderer

at de kan ha blitt utdatert, dette kalles "glemming med en hensikt", når de er for gammel til å være av en reell interesse. Fakta kan også slettes ved søppeltømming, dette er relevant dersom et faktum kun er et resultat ev en mellomberegning. Vi kan også slette fakta dersom en annen fakta inneholder den samme informasjonen slik at de er overflødige.

1.2.1 Implementeringsvalg

Når en bedrift har tatt beslutningen med å innføre et håndteringssystem for forretningsregler finnes det flere muligheter for implementasjonen av dette. Det vil hovedsakelig si å enten bygge systemet fra bunnen, kjøpe en helt ferdig løsning eller å kjøpe en løsning som man spesialtilpasser til bedriftens egne behov.

Bygge fra Bunnen

Denne løsningen er attraktiv ettersom den tillater leverandørene å lage løsninger med konkurransedyktige verktøy. Disse verktøyene vil da bli helt perfekt tilpasset hele systemet, samarbeidet mellom alle delene går glatt og det tilbys konsistente vindu. Ulempene er flere, blant annet vil utviklingen av brukergrensesnitt representere 50 % [1] av den totale utviklingstiden og vil aldri bli helt fullført. For hver eneste nye utgave som slippes, vil brukerne stadig etterspørre forbedringer av verktøyene, nytt design og nye tjenester. I tillegg vil den grafiske layout og effektive regelmotorer avhenge av ekstrem kompleks ekspertise og algoritmer, noe de alle fleste bedrifter ikke har i hus og dermed må kjøpe inn / leie. Av disse grunner vil mange bedrift bli fristet av å kjøpe en ferdig fungerende løsning eller eventuelt tilpasse komponenter som de kan koble til sine kjerneapplikasjoner for å redusere innsats og risiko.

Kjøpe en ferdig implementert løsning

Denne løsningen unngår kostnadene ved utvikling og samtidig risikoen for egenskaper ved løsningen som ikke er innen kjernevirkskomheten til bedriften. Ulempen er at du kan ende opp med å betale mye penger for et system som

ikke lar seg integrere fullstendig med annen programvare i bedriften, samt at du ikke får alle de verktøy du ønsker.

Kjøpe og deretter spesialtilpasse løsningen

Denne tredje løsningen er avhengig av høynivås komponenter som bringer inn de nødvendige verktøyene - klart til å brukes men likevel mulig å tilpasse, og som er fullstendig åpne for å tillate en full integrasjon med de andre delene av systemet. Dette alternativet kombinerer kreativitet med fleksible komponenter som kan personaliseres for å legge til økt markedsverdi som utgjør forskjellen for sluttbruker og mot konkurransen. Det kombineres også produktivitet med komponenter som er nær den endelige løsningen og som forsikrer en begrenset utviklingskostnad og risiko. Det bevares den konkurransemessige fordelen for prising og egenskaper, mens samtidig blir det en komplett integrasjon med de andre komponentene i systemet.

1.3 Regelbaserte ekspertsystem

Den mest populære typen av ekspertsystemer er de regelbaserte [10]. Et stort antall av disse har blitt bygd og vellykket tatt i bruk innen områder som medisin, geologi, ingeniørvitenskap, forretning, militært og i energibransjen. Slike system brukes ofte til å implementere en del av et stort system og de er svært velegnet til å bruke i kombinasjon med menneskelig ekspertise.

Håndtering av forretningsregler, såkalt Business Rules Management, er mye brukt innen regelbaserte ekspertsystem. Dette vil si den praktiske kunst å implementere system ved å bruke regler som basis for å uttrykke bedriftens standarder og policy. Et slikt system lar utviklerne opprette et sett av forretningsregler som et separat, håndterbart prosjekt utenfor applikasjonskoden. Et slikt system har følgende egenskaper og ansvarsområder [3]:

- Lagre og vedlikeholde et lager av forretningsregler som representerer policy og prosedyrer til en organisasjon
- Holde reglene separat fra "rørleggingen" som er nødvendig for å få moderne distribuerte datasystem til å arbeide sammen

- Integrere med organisasjonens applikasjoner, slik at reglene kan brukes for å fatte beslutninger ved å bruke ordinære data.

- Forme regler til uavhengige lenkbare datasett og utføre inferensen innenfor og over slike regelsett

- Tillate brukere og beslutningstagere å opprette, forstå og vedlikeholde regler og policy uten at det krever mye opplæring

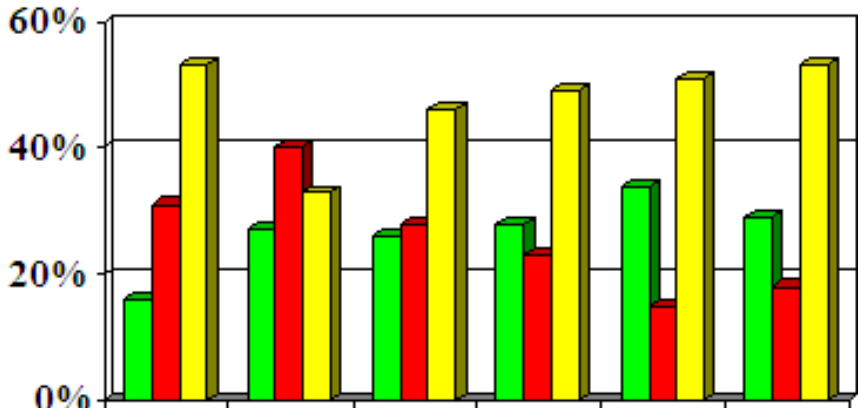
- Automatisere og fasilitere forretningsprosesser

- Opprette intelligente applikasjoner som samhandler med brukerne ved hjelp av en naturlig, forståelsesfull og logisk dialog.

1.3.1 Hvorfor bruke et regelbasert ekspertsystem?

I dagens raskt endrende marked er det en viktig differensieringsfaktor for en bedrift å tilpasse seg markedet. Den bedriften i en bransje som viser seg mest tilpasningsdyktig vil ha en stor fordel når det gjelder å tilby det kunden ønsker. I løpet av de siste årene har flere og flere store bedrifter innsett de begrensninger bruk av tradisjonell programvare setter på å minske responstiden til å få utført nødvendige endringer. Dette skyldes hovedsakelig at kjernekunnskap og data som reguleres av offentlige instanser blir lukket inne i flere systemer via bruk av svært tekniske språk. Denne informasjonen blir dermed vanskelig, om enn ikke umulig, å nå for eksperter og de som ønsker å fatte beslutninger basert på den informasjonen som ligger inne i systemet.

Ifølge forskning utført av The Standish Group, mislykkes omtrent 66 % av alle store prosjekter i USA, dette er tall som rapporten hevder er like gjeldende for prosjekter i andre deler av verden. Dette kan skyldes kanselleringer, overskridelse av budsjetter eller levering av programvare som aldri blir satt i produksjon. Den samme undersøkelsen ble foretatt både i 1994 og i 2004, ved den siste av disse hadde tallet av regelrette fiaskoer gått ned fra 31 % til 18 %.



	1994	1996	1998	2000	2002	2004
Succeeded	16%	27%	26%	28%	34%	29%
Failed	31%	40%	28%	23%	15%	18%
Challenged	53%	33%	46%	49%	51%	53%

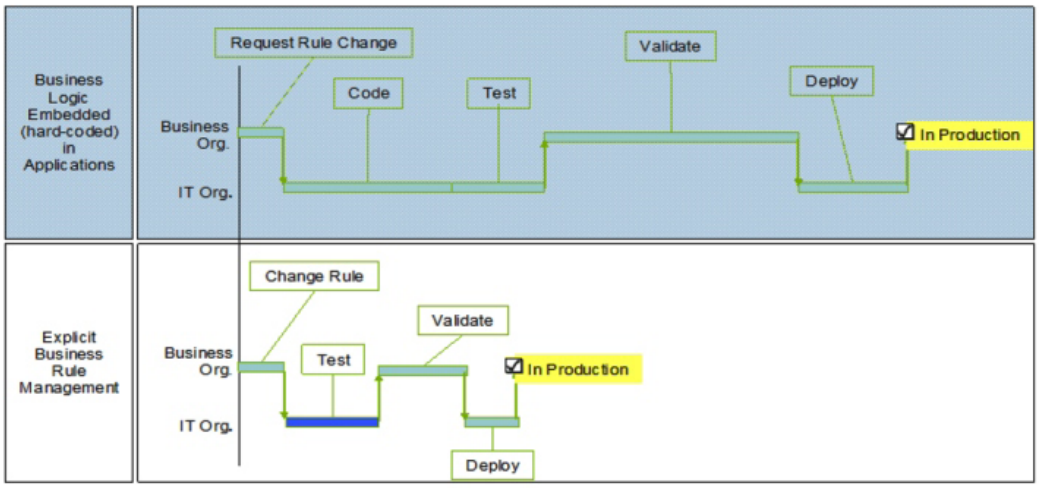
Figur 1.2: Feilede prosjekter

Selv om det er en betydelig nedgang er tallet urovekkende høyt. Grunner til disse ble i hovedsak relatert til lite involvering av brukerne, ingen klar redegjørelse av krav, lite eierforhold til prosjektet, uklar visjon og objektiv og for lite planlegging. Dette peker mot en utviklersentrert kultur blant mange it-bedrifter. Alt for ofte blir det krevd at brukerne av systemet må kunne utviklernes språk, i dag vil dette typisk si UML. Prosjektteamene må utvikle språk som baseres på enkle konseptuelle modeller av domenet, skrevet i enkle termer som både utviklere og brukere forstår. Innføring av et håndteringssystem for forretningsregler vil ikke kunne løse alle problemer knyttet til dette, men det vil være en meget god start. Ideelt burde reglene deles inn i moduler som innkapsles i individuelle objekter, inkludert noe som kalles tavleobjekter, som er synlige for alle objekter som har registrert en interesse i dem. Slike tavler innkapsler globale eller en organisasjonelle policyer, mens regelsett tilhører spesifikke klasser som lagres innenfor disse objektene. Et slik system bør minke utviklingskostnader og drastisk forkorte både utviklings- og vedlikeholdssykler.

Det er estimert at 90 % [3] av kostnader i forbindelse med IT går til vedlikehold av eksisterende system. Dette er en av grunnene til at objektorientert og komponentbasert utvikling er så attraktiv som den er, når implementasjo-

nen av en datastruktur eller funksjon endrer seg, vil ikke disse endringene propagere til de andre objektene. Dermed blir det heller ikke nødvendig å utføre omfattende tester på hele systemet, men kun på den delen man har endret. På denne måten vil vedlikeholdet begrense seg til de aktuelle objekter eller komponenter. Denne fordelen vil ikke gjelde ved endringer i forretningsreglene dersom de spres rundt omkring i systemet. Dersom grensesnittet da endres, vil endringene propagere og vedlikehold vil bli svært kostbart. For å unngå dette, må vi separere definisjon av bedriftens policy fra implementasjonen. Dette er noe et håndteringssystem for forretningsregler støtter.

Ved å trekke ut reglene og legge deres definering og modifisering tilgjengelig for brukerne i bedriften, vil tidsforløpet fra et krav blir fremmet til det blir satt i produksjon, gjerne drastisk redusert. Figur 1.3viser hvor innsparingen kan komme.



Figur 1.3: Mulig besparelse [2]

Dersom brukerne på forretningssiden i en bedrift ønsker endringer i en applikasjon, sendte de før et ønske til IT-avdelingen, som kanskje ikke skjønte helt hvilke endringer som ble ønsket. Dette kunne være starten på en rekke telefonsamtaler eller møter for å få klarhet i krav. I løpet av den tiden en bruker i organisasjonen før brukte på utvikle et ønske om en endring i en regel, kan han nå selv endre regelen. Det er dermed ikke behov for at IT-avdelingen skal gjøre noe koding i forbindelse med dette, da brukeren gjør det selv. Organisasjonen vil bruke mindre tid på å validere utførte endringer ettersom de blir utført dirkekte. Når reglene er eksternalisert, er det kun de endringer som er utført på reglene som trenger å bli satt i anvendelse, dette bør gå raskere enn å sette hele komponenter i anvendelse.

Dagens praksis med utvikling av programvare hemmer rask leveranse av nye løsninger, selv små endringer i eksisterende system kan ta for lang tid. I tillegg øker stadig konkurransen slik at bedrifters policy og regler som angår de automatiserte prosesser må være tilbøyelige for hurtige endringer. Dette kan være endringer drevet av ny produktutvikling, behov for å tilby individuell tilpasning og ønske om å hurtig forbedre forretningsprosesser som angår ulike kundegrupper. Ved å personalisere tjenester, innhold og interaksjonsmåter, basert på prosesstyper og kundekarakteristikk, kan en organisasjons forretningsprosesser øke verdien betraktelig. Naturlig dialog og klart uttrykte regler klargjør mening og avhengighet mellom regler og policy. I regulerte industrier, for eksempel farmasøytisk og finans, vil regler for styring og forskrifter endre seg utenfor bedriftens kontroll. Ved å separere disse reglene fra applikasjonskoden blir det lettere å endre de når det blir nødvendig. Dette er spesielt viktig når miljøet har flere typer valuta, er multinasjonalt og multikulturelt. [3].

Følgende indikatorer kan påvise et behov for et håndteringssystem for forretningsregler [9].

- Policy defineres av eksterne byrå
- Variasjoner mellom organisasjonelle enheter: Geografi, hierarki eller forretningsfunksjon
- Objekter som kan ha ulike tilstander, for eksempel ordrestatus.
- Spesialisering av forretningsobjekter: Kundetype, forretningshendelser, produkter
- Automatiserte system. Forretningslogikk innlemmet og gjemt innen eksisterende system
- Definerte nivå og grenser for policy: Alder, kvalifiseringskriterier, sikkerhetssjekk
- Betingelser som er linket til tid: Åpningstider, startdato, ferier
- Kvalitetsmanual: Hvem gjør hva, autoriseringsnivå
- Signifikante diskriminatorer: Forgreiningspunkt i prosesser, gjentagende oppførselsmønstre
- informasjonsbetingelser: Tillatte spenn av tillatte verdier, objekter og beslutninger som må kombineres eller ekskludere hverandre.
- Definisjoner, derivasjoner eller beregninger: lagringsflyktig spesialisering av forretningsobjekter, proprietære algoritmer, definisjoner av re-

lasjoner

- Aktiviteter relatert til bestemte hendelser eller omgivelser: Nyttår, utløsende hendelser, betingelsesbestemte prosedyrer.

Dersom en bedrift kjenner seg igjen i noen av disse indikatorene, bør den vurdere å anskaffe seg et håndteringssystem for forretningsregler.

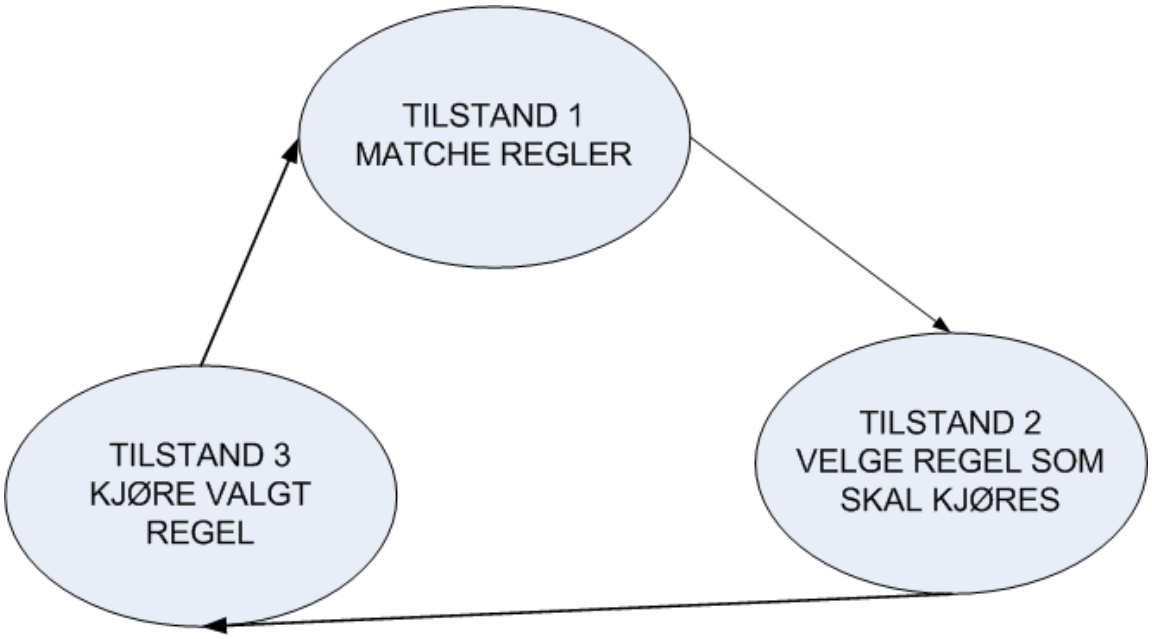
1.4 Regelmotor

Når du initierer en konsultasjon med et ekspertsystem, er det regelmotoren som setter i gang et søk etter kunnskap i kunnskapsbasen for å se om den kan tilfredsstille forespørselen. I motsetning til en komponent hos en regel, er ikke en regelmotor bygd for å løse et bestemt problem. Istedenfor tilbyr den et felles sett av muligheter for å definere, lagre og anvende reglene.

Stien regelmotoren vil følge gjennom et søk er ikke lagt på forhånd, i motsetning til i konvensjonelle systemer. Gjennomkjøringen vil avhenge av hvilke mål vi ønsker å oppnå samt hvilken informasjon vi gir inn til systemet. Selve inferensen i regelmotoren baserer seg på den velkjente logikken Modus Ponens. Den antar at hvis du deklarerer A + B THEN C, og at dersom du i en gitt situasjon finner at A og B er sann, har du lov til å anta at også C er sann.

1.4.1 Regelmotorens tilstander

En regelmotor kan betraktes som en endelig tilstandsmaskin med en syklus som består av tre tilstander: Matche regler, velge regel og deretter kjøre den valgte regelen, se figur 1.4



Figur 1.4: Tilstander til regelmotoren

I den første tilstanden finner regelmotoren alle regler som tilfredsstiller innholdet i databasen, dette i henhold til regelmotors algoritme for sammenligning. Den bruker spesielle søkestrategier for å velge hvilke regler som skal prosesseres. Dette kalles lenking og blir tatt opp i kapittel 1.4.4 Lenking. Disse strategiene kontrollerer hvordan regelmotoren finner de regler som er egnet, hvilke regler som velges og hvordan regler skal prosesseres.

Dette leder oss over til den andre tilstanden i syklusen. Alle regler som ble funnet i tilstand 1 overføres til tilstand 2, det må da velges ut hvilken av reglene som skal kjøres. Disse reglene kalles et konfliktsett. Her finnes det flere mulige implementasjoner for å velge hvilken regel som skal kjøres, disse blir gjennomgått under Konfliktløsning i kapittel 1.4.2.

Etter at regelmotoren har kommet fram til hvilken regel som skal kjøres og kjørt den, returnerer den så tilbake til tilstand 1 hvor den igjen prøver å matche regler med innholdet i databasen. Når en regel blir kjørt, hender det svært ofte at innholdet i databasen blir endret. Dermed er det ofte slik at andre regler vil tilfredsstille betingelsene i regelmotoren neste gang den starter på tilstand 1.

Noen systemer har en innebygd kompilator som transformerer reglene slik at det blir lettere å lage mer effektive aksesseringsmetoder for å gjenkjenne

hvilke regler som er velegnet på et bestemt stadium i prosesseringen.

1.4.2 Konfliktløsning

Se for deg at databasen inneholder 'solen skinner' og følgende regler ligger inne i kunnskapsbasen:
Regel1: IF det snør
THEN handling er hold deg innendørs

Regel2: IF solen skinner
THEN handling er å gå ut på ski

Regel3: IF solen skinner
THEN handling er hold deg innendørs

Vi vil vi her ha en konflikt mellom regel 2 og regel 3 dersom vi har en regelmotor som prøver å trekke en konklusjon ut ifra de data som er gitt, dette kalles en datadreven regelmotor. Begge disse reglene vil bli kjørt ettersom de har en sann betingelse, konklusjonen som nås avhenger av hvilken av de to reglene som blir kjørt sist. På grunn av muligheter for at kjøring av en regel kan medføre endringer i databasen og følgelig kjøring av andre regler, vil kun én regel kjøres av gangen. Konfliktløsningen omhandler hvilke regler som skal kjøres i et slikt tilfelle som eksemplet nevnt ovenfor.

En mulig løsning er å stoppe eksekveringen av regler når vi har nådd det målet vi ønsker. I tilfellet ovenfor ville vi ha stoppet når objektet handling får en verdi. Dette medfører at rekkefølgen reglene legges inn i systemet på er av meget stor betydning for resultatet, i og med at det er den første regelen som blir kjørt sin handling som blir stående som konklusjon. Etterfølgende reglers handlinger blir aldri utført.

En annen mulighet er å legge til prioritet for hver enkelt regel. Dette vil ikke være en god løsning dersom antall regler overstiger 100 [10], da dette vil bli svært tidkrevende og uoversiktlig. Prioriteten bestemmes enten ut ifra rekkefølgen reglene legges inn i systemet på, eller ved å manuelt legge til prioritet til de regler man ønsker skal kjøres først.

Man kan også kjøre regler i henhold til hvilken av reglene som er mest spesifikke, også kalt 'lengste matchende strategi'. Den baseres på antagelsen om at en spesifikk regel prosesserer mer informasjon enn en mer generell regel.

Det vil si at dersom vi har A og B i vår database, og vi har to regler som har A i sin betingelse, vil den av de to reglene som i tillegg har B i sin betingelse bli kjørt.

Å velge den regelen som bruker de data som mest nylig er lagt inn i systemet er også en mulighet, denne løsningen forutsetter at hvert faktum som ligger inne i databasen har et tidsstempel festet til seg. Dette er en implementasjon som kan være særs nyttig i sanntidssystemer hvor informasjon ofte blir oppdatert.

Metaregler

Alle de ovennevnte konfliktløsningsmetodene er forholdsvis enkle å implementere og gir oss en tilfredsstillende løsning på problemet. Dersom systemet blir veldig stort, blir det derimot vanskelig å håndtere det store antall regler og ekspertsystemet må selv ta litt ansvar for å ordne opp. Dette kan gjøres ved bruk av metakunnskap, kunnskap om kunnskapen. Metakunnskap omhandler bruk og kontroll av domenekunnskap i et ekspertsystem [10]. I regelbaserte ekspertsystem representeres denne kunnskapen som metaregler. En mataregel determinerer en strategi for bruk av oppgavespesifikke regler i ekspertsystemet.

Den som implementerer systemet overfører kunnskap fra domeneeksperten over til systemet og lærer da hvordan problemspesifikke regler er brukt. Dermed skaper han gradvis en ny kunnskap om oppførselen til ekspertsystemet. Denne nye metakunnskapen er stort sett domeneuavhengig. Et eksempel på en slik uavhengig regel: “en regel som er lagt inn av en ekspert har høyere prioritet enn en regel som er lagt inn av en nybegynner“

Noen ekspertsystem tilbyr en egen regelmotor for metaregler. Ulempen er at de fleste systemer ikke har mulighet til å skille mellom en regel og en metaregel. Derfor bør en metaregel ha høyest prioritet i en eksisterende kunnskapsbase. Når den blir kjørt, legger en metaregel inn noe viktig informasjon inn i databasen som kan endre prioriteten blant andre regler.

1.4.3 Regler

Det finnes uttallige definisjoner av en forretningsregel, jeg velger å bruke denne dersom noe annet ikke blir spesifisert: “En kompakt uttalelse angående et aspekt hos en bedrift som kan uttrykkes i termer som kan direkte relateres

til forretningen, ved å bruke et enkelt utvetydig språk som er tilgjengelig for alle interesserte parter: bedriftseiere, forretningsanalytikere, tekniske arkitekter og videre". Videre antas det at disse reglene innlemmes i et regime som kan lenke de sammen, en nøkkelkomponent i et hvilket som helst håndteringssystem for forretningsregler [9]. Med andre ord er en regel ikke en beskrivelse av en prosess eller prosessering. De er heller en definering av betingelser under hvor en prosess blir utført eller nye betingelser som vil eksistere etter at en prosess har blitt fullført. På den måten kan man si at forretningsregler definerer hva som må være tilfellet istedenfor hvordan det blir til.

Kunnskap om kausale forhold er vanligvis lagret på formen "IF A THEN B". I motsetning til if/then uttrykk som man har i konvensjonelle språk som for eksempel Java, er regelspråkene ofte deklarative eller ekvivalent ikke prosedyreorientert. Det betyr at rekkefølgen reglene skrives i, ikke nødvendigvis spiller noen rolle for utfallet av programmet. Her må det nevnes at det kan ha stor påvirkning på resultatet hvilken rekkefølge reglene kjøres i, slik som nevnt i kapitlet som angår konfliktløsning, 1.3.2.

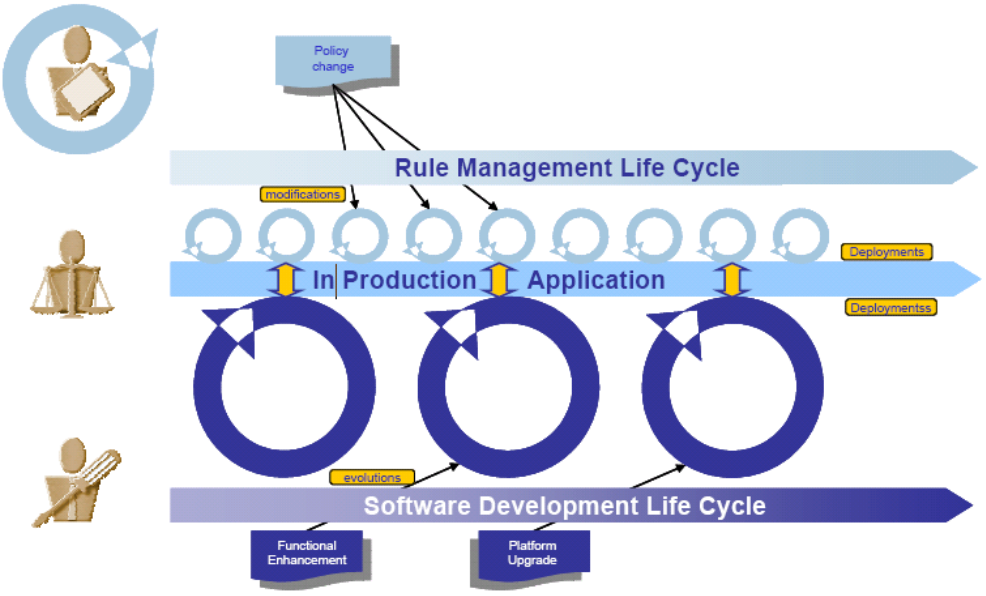
Reglene arbeider på kunnskap om entiteter eller objekter. Regler, prosedyrer og objekter er de vanligste måtene å representere kunnskap på i håndteringssystem for forretningsregler. Reglene er en restriksjon i den forstand at en forretningsregel legger ned hva som må og ikke må være saken. Ved et hvilket som helst stadium, skal det være mulig å bestemme at betingelsen som er implisert av betingelsen er sann i et logisk perspektiv, hvis ikke behøves forebyggende tiltak.

Ifølge Morgan [9], bør forretningsreglene kun interessere seg for de betingelser som henvender seg til en definert tilstand. Det vil si at dersom du skriver et uttrykk i en regel, bør du unngå fristelsen til å legge til ekstra informasjon som går utenfor denne grensen. Den ekstra informasjonen bør istedenfor dekkes et annet sted, ved å duplisere den i en regel øker du kun faren for potensielle feil. Nærmere bestemt, en regel skal bestemme hva som skal være tilfellet og ikke:

- Hvem som invokerer regelen. Dette beskrives i use case eller en prosessbeskrivelse
- Når regelen eksekveres. Dette vil ofte bli beskrevet i en forretningsevent, use case eller prosessbeskrivelse
- Hvor regelen eksekveres. Dette vil defineres i designet
- Hvordan regelen skal implementeres. Dette vil bli definert i designet

Livssyklus for en forretningsregel

Å tillate en ekspert å utnytte kunnskapen sin ved å enkelt legge til eller endre regler i systemet er en essensiell del av et håndteringssystem for forretningsregler. For å få utnyttet bedriftens ressurser foreslår ILOG, organisasjonen bak regelmotoren jeg har benyttet i caset, at man deler inn utviklingen av programvaren og forretningsreglene i to ulike livssykluser. Dermed kan utviklingen av disse to delene foregå uavhengig av hverandre og dermed føre til mer effektiv utvikling. Dette tillater at de som skriver reglene kan operere uavhengig av utviklingssyklusen til programvaren. Dette leder til parallellitet.



Figur 1.5: Livssykluser for regler og programvareutvikling [5]

Dette er illustrert i 1.5. Den nederste halvdelen viser livssyklusen til programvaren, en ny versjon av denne drives av behov for store endringer. Den er også avhengig av slipp av eksterne produkter.

Den øverste delen viser den samme syklusen for forretningsreglene. Disse reglene har som regel behov for å bli endret langt oftere enn programvaren, endringene drives av endringer i organisasjonens policy. Forandringer implementert i livssyklusen til reglene krever mindre, mer fokuserte sykler av skriving og testing. Tidsrammen for å komplettere en syklus avhenger av kravene til applikasjonen, men det kan være alt fra et par timer til flere måneder [5].

Formelen "Fortjeneste = inntekt - kostnader" er ikke en regel, men heller en prosedyre eller funksjon for å regne ut fortjeneste, eventuelt et utsagn som forteller hva en fortjeneste er. En regel i et regelbasert system er ikke prosedyreorientert, de fastslår hva som er sant men ikke hvordan man skal beregne hva det er.

En viktig egenskap med regler er inferensen. Dersom man kommer med de to følgende fakta: Innholdet i denne termosen er kaffe, og at denne kaffen er varm - vil du automatisk vite at innholdet i denne termosen er varmt. Dette vet du selv om ingen har fortalt deg det, denne konklusjonen sluttet du selv. Av denne årsak må et håndteringssystem for forretningsregler støtte slik logisk slutning/inferens.

Oppbygning av regler

I et regelbasert system er kunnskapen representert som et sett av IF - THEN produksjonsregler, og data er representert som et sett av fakta om nåværende situasjon. En regel hevder at dersom betingelsen i regelen, altså IF-delen, er sann, så kan man slutte at også handlingsdelen, THEN-delen, av regelen er sann. Dette er i henhold til tidligere nevnte modus ponens.

Regler består av to deler: en betingelse og en konklusjon/handling, gjerne kalt hhv. forgjenger og konsekvent. Den generelle syntaks er som følger:
IF <forgjenger>
THEN <konsekvent>

En regel kan bestå av flere både betingelser og handlinger. Disse delene av en regel bindes sammen ved hjelp av konjunksjon (AND) og disjunksjon (OR), eventuelt en kombinasjon av disse. Det anbefales likevel å unngå å bruke både konjunksjon og disjunksjon i den samme regelen [10].
IF < forgjenger 1>
AND < forgjenger 2>
.
.
.
THEN < konsekvent 1>
AND < konsekvent 2>
.
.
Betingelsen i en regel innehar to deler, et objekt og dets verdi. Et eksempel

på dette kan være at objektet er antall deltagere og at verdien er 5. Objektet og verdien knyttes sammen ved hjelp av operatorene er / er ikke. Operatoren tildeler en symbolsk verdi til objektet. Ekspertsystem kan i tillegg bruke matematiske operatorer til å definere et objekt som numerisk og tildele det en numerisk verdi:
IF 'antall deltagere' < 10
THEN 'kurs er avlyst'

I likhet med en regels betingelse, knyttes en handlings objekt og verdi sammen ved bruk av operatorer. Regler kan brukes til å representere relasjoner, råd, direktiv og strategier (1).

Selve inferensen forekommer mellom betingelsen og handlingen i en regel. Dette fører til at vi har to typer regler: Definisjonsregler og heuristiske regler. Førstnevnte er regler hvor inferensen etablerer et forhold mellom termer, mens ved den siste typen baseres inferensen på ufullstendige bevis.
Definisjonsregel:
IF hjemby = Bergen
THEN hjemland = Norge

Heuristisk regel:
IF hjemby = Bergen
THEN vær = regn

Her ser vi klart at i den første regelen er det en selvfølge at dersom du bor i Bergen, så bor du også i Norge. Denne regelen er sann fordi et forhold mellom klasse - subklasse er definert, denne regelen vil være sann for ethvert individ som bor i Bergen. I den andre regelen derimot, er slutningen mer uklar. Selv om du bor i Bergen, behøver det ikke å regne. For å være helt sikker må du selv sjekke været. Det man kan si, er at dersom man bor i Bergen, og ikke har mer kunnskap enn dette, kan man anta at det regner. De fleste regelbaserte systemer er satt sammen av mange definisjonsregler og noen få heuristiske regler.

Noe som er tillatt i enkelte systemer, er å legge inn en ELSE i definisjonen av en regel. Dette må gjøre med stor forsiktighet, og bare i regler som har to mulige konklusjoner. Årsaken til det er at når man legger inn ELSE i regelen, vil den alltid bli kjørt. Dersom betingelsen er sann vil THEN bli kjørt mens dersom betingelsen ikke er sann vil ELSE bli kjørt. Denne typen regler kan legge inn et uønsket prosedyrepreg til de regelsettene de er en del av. I tillegg kan den relative plasseringen til en regel som inneholder en ELSE dramatisk

påvirke resultatet til en kunnskapsbase.

1.4.4 Lenking

Selv om kunnskapen ligger lagret et eller annet egnet sted inne i en datamaskin, må vi ha fasiliteter for å navigere gjennom den og for å manipulere den. Hvis ikke oppnår vi ingenting med å ha den lagret. Inferens er en prosess hvor man trekker gyldige konklusjoner ut ifra visse gitte premisser. Regelmotoren sammenligner hver regel lagret i kunnskapsbasen med fakta som ligger i databasen. Når en IF-betingelse matcher et faktum, kjøres regelen og handlingsdelen av regelen blir utført. Når en regel kjøres, kan den endre de fakta som ligger i databasen ved å legge til, slette eller endre. Matching av reglers betingelser med fakta i basen produserer en inferenslenke. En slik lenke indikerer hvordan et ekspertsystem bruker regler for å nå en konklusjon. Nedenfor følger et eksempel for å vise dette:
Innhold i database ved start: A, B, C

Regel 1: IF D er sant
THEN E er sant

Regel 2: IF B er sant
AND C er sant
THEN F er sant

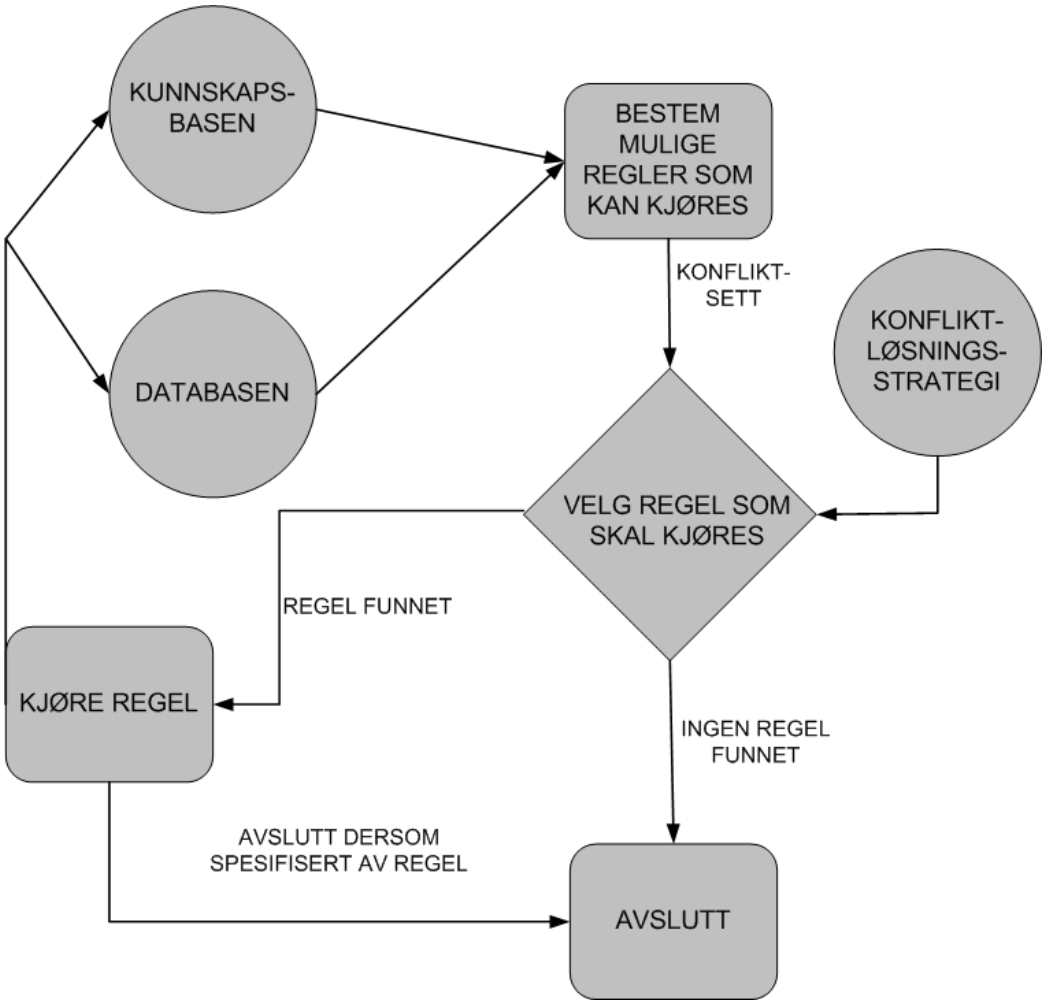
Regel 3: IF F er sant
THEN D er sant

Forklaring: Når programmet begynner å kjøre, inneholder databasen A, B og C. Det sjekker så betingelsen i regel 1 og finner at den ikke er overens med databasen. Så blir betingelsen i regel 2 kontrollert, den er sann og dermed kjøres handlingen i regel 2. Etter regel 2 er kjørt, inneholder databasen A, B, C og F. Programmet starter så på toppen igjen og prøver regel 1. Den vil heller ikke denne gang bli kjørt. Regel 2 kjøres ikke da den allerede er kjørt. Regel 3 kjøres og legger til D i databasen. Dermed blir også regel 1 tilslutt kjørt. Ekspertsystemet kan vise inferenslenken for å forklare hvordan den nådde sin konklusjon, i dette tilfellet E. Regelmotoren må bestemme når regler må kjøres. Det er to hovedmåter regler kan kjøres på, den ene kalles framlengs lenking mens den andre heter baklengs lenking.

Framlengs lenking

Eksemplet vist ovenfor bruker framlengs lenking. Et annet vanlig navn på denne metoden er datadreven slutning. Dette navnet kommer av at man bruker de data man har tilgjengelig når man starter systemet og stadig legger til nye data etter hvert som regler blir kjørt. For hver gang regelmotoren kjører gjennom systemet blir kun den øverste regelen med en gyldig betingelse kjørt. Hver regel kan kjøres kun en gang. Systemet stopper når det ikke er flere regler som kan kjøres.

Figur 1.6 viser gangen i et framlengs lenket system.



Figur 1.6: Framlengs lenket system

Denne metoden samler inn så mye informasjon som overhodet mulig og trekker så en konklusjon. Dette medfører at unødig mange regler kan bli kjørt, regler som ikke har noe å gjøre med den konklusjonen vi ønsker å komme fram til. Regelbaserte ekspertsystem kan ha flere hundre, i tilfeller også flere tusen, regler i sin kunnskapsbase. Dermed blir det svært vanskelig å implementere framlengs lenking effektivt. Dette fordi svært få endringer blir utført på fakta som ligger i arbeidsminnet for hver sykel. Dette problemet kan unngås ved bruk av algoritmen Rete. Det er en algoritme som er svært effektiv for å bestemme relevansen til reglene, gitt bestemte data. Rete er et nettverk som modifiserer seg selv etter kjøring av hver regel, på denne måten behøver ikke unødvendige regler å kjøres. Man ser besparelsene ved bruk av Rete tydeligere dess flere regler som finnes inne i systemet. Rete gjennomgås i detalj i kapittel 1.8.

Matching av betingelsen kan ikke ha noen sideeffekter eller endre på tilstander i databasen. Derfor fungerer betingelsesdelen av regelen som en read-only av minnet.

En slik lenking er mest velegnet når vi har flere likeverdige måltilstander og en enkelt startsituasjon. Det blir brukt heuristisk informasjon for å veilede søket fra start til slutt. Denne heuristiske informasjonen kan også inkludere ulike nivå av kontroll, slik at reglene i et forlengs lenket system bærer både kunnskap om domenet og om kontrollen. På denne måten kan man oppnå en høy grad av fleksibilitet og effektivitet på kostnad av ren representasjon av domenekunnskap [8].

Baklengs lenking

Dersom man har som mål å komme fram til kun et lite antall fakta, er ofte en baklengs lenking et bedre alternativ. Dette er en målrettet tankegang. Man starter her med et mål, en konklusjon, som man ønsker å komme fram til. Deretter ser regelmotoren på hvilke regler som kan lede fram til dette målet.

Framgangsmåten som blir benyttet ved denne målrettede metoden er som følger: Regelmotoren ser etter en regel som har det ønskede mål som THEN-del. Dersom denne regelens IF-del finnes i databasen, er målet nådd. Dette er derimot ofte ikke tilfellet, regelmotoren må gjennom flere regler for å nå fram til å bevise sitt mål. Dermed blir betingelsen i regelen satt som et nytt submål som man ønsker å bevise. Dette submålet blir satt som det nye målet, mens det originale målet blir satt til side. Gjerne kalt å bli stakket. Eksempel: Innhold i database ved start: A, B, C Vi ønsker å bevise E.

Regel 1: IF D er sant
THEN E er sant

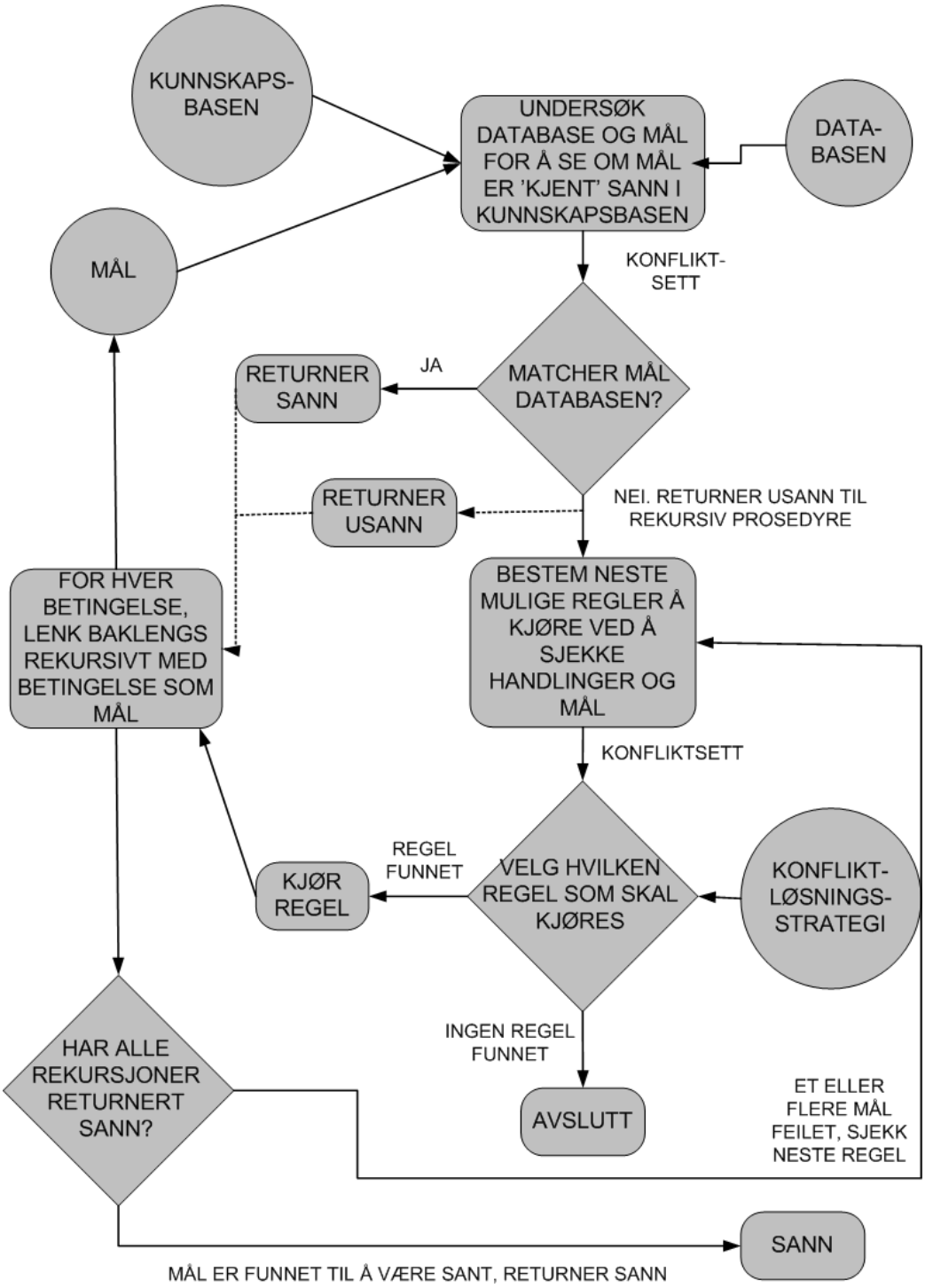
Regel 2: IF B er sant
AND C er sant
THEN F er sant

Regel 3: IF F er sant
THEN D er sant

Forklaring: Regelmotoren går igjennom reglene for å se om en av de har en handling som er lik målet, i vårt tilfelle E. Dette er oppfylt i regel 1. Hadde vi nå hatt D i vår database, ville jobben nå vært ferdig. Dette er ikke tilfellet. Dermed blir vårt opprinnelige mål, E, satt til side og vi ønsker nå å bevise D. Regelmotoren søker gjennom reglene og finner at regel 3 tilfredsstiller D som mål. Heller ikke denne gang blir betingelsen oppfylt og følgelig blir F satt som det nye målet vi ønsker å bevise. Regel 2 beviser F, betingelsene i regel 2 er tilfredsstilt i vår database. Dermed er F lagt til i databasen. I neste runde går vi tilbake til regel 3 og legger til D i databasen. Dermed har vi kommet tilbake til vårt opprinnelige mål og bevist at E er sann.

Dersom vi i eksemplet nevnt ovenfor hadde hatt et delmål øverst på stakken, og regelmotoren søker gjennom reglene for å finne en regel som har dette delmålet som konkludent, men ikke finner en gyldig regel, genererer systemet automatisk et spørsmål til brukeren hvor han blir bedt om å skrive inn en verdi. Regelmotoren spør kun om informasjon som hjelper til med å løse svaret, ingen unødvendige spørsmål blir stilt.

Figur 1.7 viser gangen i et baklengs lenket system.



Figur 1.7: Baklengs lenket system

Baklengs lenking er mest egnet når det er et enkelt mål vi ønsker å oppnå, for eksempel ved stilling av en diagnose, spesielt dersom vi har så mye tilgjengelig informasjon at det er svært uhensiktsmessig å gå igjennom all. En av de største ulempene ved denne strategien er at den har ikke mulighet til å sortere bort implikasjonene til de fakta hvor signifikansen ikke er kjent. Det er nemlig ikke alltid mulig å vite hvilke fakta som er viktige i en gitt setting. Problemer som involverer design og planlegging kan kreve komplekse tankeganger for å bestemme når en plan eller en design er tilstrekkelig godt nok. Noen problemer har ikke engang et mål, i slike tilfeller gir det ikke mening å kjøre kun de av reglene som bidrar til å nå det ikke eksisterende målet.

Regelmotoren har flere plikter ved baklengs lenking: Den må gjenkjenne mål som kan nås umiddelbart, utvide mål som ikke er umiddelbart oppnåelige til enklere delmål, utføre en nødvendig handling for å nå primitive mål (for eksempel spørre bruker eller utføre kjente prosedyrer), følge opp løsninger til submål slik at de støtter de originale mål og å vedlikeholde en trestruktur over mål for å tillate forklaring av beslutningstakingsprosessen. Dette medfører at mye kontroll må bygges inn i regelmotoren, og dette medfører en nedgang i effektivitet.

Framlengs eller Baklengs lenking?

Vi ser en tydelig forskjeller på forlengs og baklengs lenking. Ved forlengs lenking kjenner vi alle data i begynnelsen av gjennomkjøringen og brukeren blir aldri bedt om å legge inn mer data. I baklengs lenking, er målet allerede satt opp og de eneste data som brukes er de som er nødvendig for å støtte den direkte tankegang for å bevise målet. Brukeren kan bli spurt om å legge inn informasjon som ikke allerede ligger inne i systemet.

Valget av forlengs eller baklengs lenking er ikke alltid like enkelt. De fleste systemer i dag benytter baklengs. En måte for å finne ut hvilken metode som er mest egnet er å studere hvordan en domeneekspert løser et problem i det gitte domenet. Dersom han i forkant må samle inn en mengde informasjon og deretter prøver å trekke de konklusjoner som er mulig ut ifra de gitte data, er den datadrevne modellen mest egnet. Denne modellen er også mest egnet dersom man, selv om man jobber for å komme fram til kun ett enkelt mål, må kjøre bortimot alle reglene. Dersom man i det tilfellet da bruker den målrettede strategien, brukes et stort antall ressurser på å finne ut hvilke regler som skal kjøres når, selv om nesten alle kommer til å bli kjørt uansett. Har han derimot har en hypotese han ønsker å teste, og forsøker å finne fakta

for å bygge under denne tesen, bør man velge den målrettede regelmotoren.

Andre implementasjoner

Ser man for seg hvordan en ekspert løser et problem i den virkelige verden, ser man fort at det er avhengig av problemets natur. Har man mulighet til å velge strategi som passer til hvert enkelt problem får man et mye sterkere hjelpemiddel. Derfor kombinerer noen regelmotorer de to ovennevnte modellene. Den kombinerte modellen antar at baklengs lenking brukes som søketeknikk, men dersom man får behov for framlengs lenking prosesserer regelmotoren kunnskapsbasen på den datadrevne strategien. Når dette ikke lengre er nødvendig, går den tilbake til baklengs. Denne modellen kan sørge for å eliminere noen av ulempene ved baklengs lenking.

Mange regelmotorer lar deg midlertidig overstyre baklengs lenking ved å definere en framlengs egenskap, denne krever å bli betraktet øyeblikkelig under visse bestemte betingelser. Dette kan gjøres ved å legge til en merkelapp FORWARD på denne måten:
IF...
THEN...
FORWARD: YES

Kunnskapsbasen blir da behandlet på vanlig baklengs vis. Når et av premissene til en regel merket FORWARD: YES blir kjent, blir denne regelen kjørt og da prøver regelmotoren på framlengs vis å evaluere regelen. Dersom regelen ikke kan bli evaluert fordi en eller flere av dens premisser enda ikke er kjent, blir regelen lagt til side og tatt opp igjen når et av disse premissene så blir kjent. Selv om disse reglene gir en framlengs strategi, vil det å kjøre de ikke si det samme som dersom vi hadde brukt en framlengs lenket regelmotor. Da hadde alle regler blitt behandlet på samme måte. Reglene får nå en egenskap ved hjelp av merkelappen. Denne egenskapen gir regelmotoren prosedyreorientert informasjon om når disse reglene skal tas i betraktning. På denne måten får disse reglene en prosedyrekvalitet som er annerledes enn framgangsmåten en framlengs lenket regelmotor benytter.

En annen mulig implementasjon, uten å bruke merkelapper, er å gå ut ifra at vi har et baklengs lenket system men behandle alle regler som potensielle framlengs lenkede. Det foregår på følgende måte: Så fort en attributt blir tildelt en verdi, leter regelmotoren gjennom alle regler på jakt etter om denne attributt er en av deres premisser. Hvis den er det, prøver den å kjøre disse reglene på den framlengs lenkede metoden beskrevet i forrige avsnitt.

Disse metodene åpner for muligheten til å skrive ut informasjon til brukeren, noe som ofte ikke er mulig i et vanlig baklengs lenket system fordi handlingsdelen av regelen består da av å vise fram en tekst som ikke bidrar til å komme fram til målet.

1.5 Historie

De første ekspertsystem skilte ikke kunnskapen fra implementasjonen, men blandet fakta inn sammen med resten av systemet. Der ble fakta, regler, data og prosedyrer lagret i den samme kunnskapsbasen. Den første diskusjonen om håndteringssystem for forretningsregler oppsto i miljøet rundt databaser sent på åttitallet, i et magasin kalt "The Database Newsletter". Første gang begrepet ble brukt var så tidlig som i 1984, i en artikkel i "Datamation". De første implementasjoner av regler i databaser var mer begrenset enn de vi kjenner til i dag. Der var regler lagret som prosedyrer skrevet i SQL. Etter hvert ble det brukt aktive databaser som benyttet triggere, dette var if/then-regler som medførte oppdateringer avhengig av hvilke verdier som ble lagt inn i databasen.

Regelbaserte systemer inkorporerer ideer fra mange ulike kilder. Startpunktet, ifølge [4], var beslutningstabeller og kompilatorer. Denne teknologien dukket opp for omtrent 40 år siden, og tilbød en representasjon av beslutningslogikk for prosessering av transaksjoner og generering av rapporter. Oppføringer i tabellen definerte if/then-regler som kjørte sekvensielt på gjeldende inputtdata. Effekten på omgivelsene var øyeblikkelige ettersom det ikke var noen database som inneholdt fakta, ingen arbeidsminne. De eneste oppføringene i kunnskapsbasen var reglene, som representerte enkle boolske betingelser. Begrensningene ved beslutningstabellene var, når man ser tilbake på dem, betydelige. Store regeltabeller er særs komplekse, den fastsatte rekkefølgen til evalueringen av reglene viste seg utilfredsstillende og ikke i stand til å beskrive komplekse symbolske mønster.

Allerede tidlig på 70 tallet ble grunnlaget til de moderne regelbaserte ekspertsystemene lagt av Alleen Newell og Herbert Simon. Deres modell baserer seg på at mennesker løser problemer ved å bruke sin kunnskap på et gitt problem [11]. Denne kunnskapen representere de som et sett av produksjonsregler som ble lagret i et langtidsminne, kunnskapsbasen. De problemspesifikke fakta ble lagret i korttidsminne, databasen.

Arbeidere ved universitetet Carnegie-Mellon var de første til å bygge et regel-

basert system med tusenvis av regler og til å utvikle effektive kompilatorer og oversettere. Et slikt system, kalt XCON, var det første ekspertsystemet til å tjene en fortjeneste på flere titalls millioner kroner. Det generelle regelbaserte programmeringssystemet kjent som OPS, som ble brukt i XCON, har siden blitt brukt til flere andre regelbaserte applikasjoner.

Arbeidere ved Stanford utviklet noe som er kjent som MYCIN-familien innen regelbaserte ekspertsystem. MYCIN var det første systemet hvis ekspertise ble anerkjent av eksperter. Det var i stand til å utføre subjektiv resonering på ekspertnivå, dette selv med usikre data og kunnskap. Det forklarte også resonneringen sin ved hjelp av naturlig språk.

PROLOG var det første universelle logikkbaserte programmeringsspråk. PROLOG er hovedsakelig et regelbasert system som bruker lagrede fakta og regler til å slutte løsninger. Det var designet for å bevise teorem, men har vist seg svært attraktivt for et mye bredere spekter av oppgaver innen kunstig intelligens.

1.6 Konvensjonelle system i forhold til ekspertsystem

Konvensjonelle dataprogrammer benytter seg av algoritmer når de prosesserer data. På denne måten følges en bestemt stegvis rekkefølge under beregningene og samme svar oppnås hver gang programmet kjøres med samme innputt. Feil oppstår ikke, unntatt når programmereren har kodet feil. Ekspertsystem følger ikke en slik forhåndsbestemt rekkefølge, de tillater en unøyaktig tankegang og kan håndtere ufullstendige data.

En annen viktig forskjell mellom disse to systemene, er at ekspertsystemet skiller kunnskapen fra prosesseringen. I et konvensjonelt system vil dette si at regelmotoren og kunnskapsbasen er slått sammen. Det konvensjonelle systemet er en miks av kunnskap og kontrollstrukturen for å prosessere denne kunnskapen. Dermed vil en endring i systemet påvirke både kunnskapen og prosesseringen. Dette medfølger en mye mer omfattende prosess for å gjennomføre selv små endringer. Det vil også bli mye mer komplisert å forstå koden. Ved bruk av et ekspertsystem kan en ekspert enkelt legge inn regler i kunnskapsbasen - uten at dette har noe påvirkning på andre deler av systemet. Hver enkelt regel legger inn mer kunnskap i systemet og gjør det så til et smartere system.

I ekspertsystemenes tidlige dager ble det lagt stor vekt på forskjellene mellom dem og de konvensjonelle datasystemene. Differensieringen var spesielt tydelig da ekspertsystemene enda var på forskningsstadiet. Nå, da ekspertsystemene har kommet mer inn i det kommersielle miljøet, prøver man istedenfor å ta fordel av noen av de konvensjonelle systemenes konsepter ved å se på likhetene mellom disse systemene. Både ekspertsystem og konvensjonell programvare deler de samme mål om produksjon av programvare som holder høy kvalitet. Dette målet påvirker utviklingsprosessen. I tillegg må man huske at mye at programvaren i ekspertsystemene er konvensjonell programvare, dette inkluderer brukergrensesnittet og regelmotoren [13]. Konsepter som allerede er blitt tatt i bruk er abstraksjon og modularitet. Ved utvikling av ekspertsystemer har man mye å hente ved å se nærmere på hvordan de konvensjonelle systemene blir utviklet, dette fordi de har allerede har lagt ettertrykkelig fokus på emner som prosjektledelse, forsikring om høy kvalitet på programvaren, verifikasjon og validering. Regelmotoren er bygd opp på samme måte som konvensjonell programvare, ved hjelp av algoritmer. Dermed kan man benytte de samme metodene for å måle kvaliteten til programvarene.

ekspertsystem	*konvensjonellesystem*
Representere og manipulere kunnskap	Representere og manipulere data
Heuristisk løsning	Algoritmisk løsning
Symbolsk informasjon	Numerisk informasjon
Vage krav	Veldefinerte krav
Tilnærmede svar	Eksakte svar
Kontroll og data separat	Kontroll og data blandet

I denne tabellen, [13], er det en kort oppsummering over de største forskjellene mellom de konvensjonelle systemene og ekspertsystem.

1.7 Involvering av brukerne

Vi behøver måter å skrive regler på som er forståelige for brukerne. I en perfekt verden ville dette vært et rent naturlig språk, men dette er ikke mulig. Det menneskelige språket er for komplekst til at en datamaskin kan forstå det fullt ut. For å overkomme dette problemet finnes det 4 mulige løsninger [3]:

- Sørge for at brukerne av systemet lærer seg språk som datamaskinene forstår, dette kan være Java eller UML. Språket kan være tekstbasert eller grafisk, så lenge datamaskinen forstår det.

- Komme fram til et nytt språk som er svært likt naturlig språk, men som datamaskinen forstår
- Tilby et brukervennlig grensesnitt som genererer regler på en måte som er naturlig for brukeren
- Innsnevre bruken slik at vi kun behøver et subsett av det naturlige språket for å uttrykke domenet

Det første av disse 4 forslagene vil nok svært sjelden la seg vellykket gjennomføre. I praksis er det likevel slik det foregår svært ofte i dag. De tre siste forslagene krever konstruksjon av et vokabular eller en domeneontologi: en modell av hva som blir diskutert og hvilke konsept som brukes, i tillegg til relasjonen mellom dem. Det viser seg at dette er mye av tankegangen bak UML. Dette behøver ikke være negativt. Det finnes store variasjoner av UML, som går fra å være svært Javaorientert til å bruke tilnærminger som er språkuavhengige.

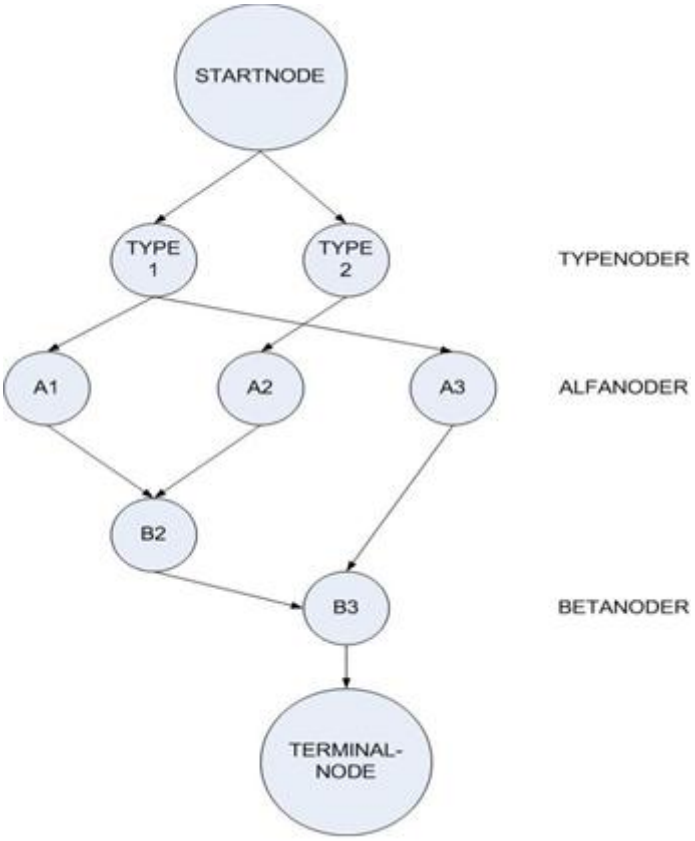
Brukerne er de som er eksperter i domenet som ekspertsystemet opererer i, derfor må de aktivt være med i utviklingen, håndteringen og optimaliseringen av de automatiserte prosessene. Dessverre er det ofte slik at de verktøyene som brukes i hver fase av en prosess livssykel -definisjon, implementering, overvåking og analyse oppmuntrer ikke disse brukerne til å delta. Tvert imot, slike verktøy er ofte designet for mennesker med teknisk bakgrunn.

1.8 RETE

Algoritmen kalt Rete er en effektiv algoritme for mønstergjenkjenning ved bruk i regelbaserte produksjonssystem. Den ble først publisert i 1974 av Dr. Charles L. Forgy, hvorpå han ved to senere anledninger har videreutviklet algoritmen, henholdsvis i 1979 og i 1982. Rete har blitt grunnlaget for en rekke populære ekspertsystem, blant andre Blaze Advisor, ILOG, regelmotoren i Microsofts BizTalk Server og OPS5. Hovedtanken bak Rete er at den ofrer minne for å oppnå en høyere hastighet.

En naiv implementasjon av et ekspertsystem kan finne på å sjekke hver enkelt regel opp imot de kjente fakta i kunnskapsbasen, kjøre regelen hvis det er nødvendig, og deretter gå videre til neste regel. Denne framgangsmåten er meget ueffektiv og kjører svært sakte. Rete derimot, legger grunnlaget for en mer effektiv implementasjon. Algoritmen benytter seg kun av venstresiden, betingelsene, i reglene. Tanken er å filtrere data etter hvert som de forflytter

seg nedover i nettverket.



Figur 1.8: Et Rete [13]

Systemer som er basert på Rete bygger et nettverk av noder, en asyklisk graf. Innputt til Rete består av en rotnode, Alfanoder som tar inn én innputt og Betanoder som tar inn to innputter.

Hver node, med unntak av roten, svarer til et mønster som forekommer på venstresiden av en regel. Stien fra rotnoden til en løvnode definerer en komplett venstreside av en regel. Hver node har et minne av de fakta som tilfredsstiller mønsteret til nodene i stien fra roten og til og med denne noden. Denne informasjonen er en relasjon som representerer de mulige verdier av variabler som forekommer i mønstrene i stien. Terminalnoder brukes til å indikere at en enkelt regel har fått treff på alle sine betingelser. Da vil denne regelen bli kjørt, med andre ord vil handlingsdelen av regelen nå bli utført.

Rotnodens barn er typenoder. Alle objekter som kommer inn til startnoden

går øyeblikkelig ned til typenodene. Meningen bak typenodene er å passe på at regelmotoren ikke utfører mer arbeid enn nødvendig. Derfor skal regelmotoren kun videresende de objekter som tilfredsstiller en nodes objekttype. Det finnes en typenode for hver objekttype som kan forekomme. Dette gir oss en fordel fordi man slipper å evaluere hver enkelt node mot hvert eneste objekt.

Fra typenoden går objektene videre til en Alfanode. For hver regel og for hver av dens mønster lager vi en Alfanode. Disse nodene brukes til å evaluere litterære betingelser, et eksempel på en slik betingelse er: " mor.fornavn == Kari". Innputt til hver node er et sett av fakta, utputt er de fakta som tilfredsstiller testen. Når en regel har flere litterære betingelser for en enkelt type objekt, linkes de sammen. Hver Alfanodene er assosiert med en relasjon, kalt Alfa-minnet, hvis kolonner er navngitt etter variabler som forekommer i nodens mønster. Disse minnene lagrer samlinger av elementer som treffer hver betingelse i hver node i en gitt gren i grafen. Et eksempel på dette er dersom mønsteret til en node er som følgende: "(er-mor-til ?x ?y)", så vil relasjonen ha "x" og "y" som kolonner, se tabellen i eksemplet nedenfor.

For hver regel Rn, dersom vi kaller de tilhørende Alfanoder for An(1), An(2), ..., An(i), konstruerer vi Betanodene Bn(2), Bn(3), ..., Bn(i). Bn(2) har som sin venstre innputt An(1) og som sin høyre innputt har den An(2), se figur 1.8. Deretter er det slik at for hver node, Bn(j) får sin venstre innputt fra Betanode Bn(j-1) og høyreinnputt fra Alfanode An(j).

På hver Betanode lagrer vi en relasjon, Betaminnet, som er sammenslåingen til de relasjoner som er assosiert med dens venstre og høyre innputt. De er koblet sammen ved kolonnene som er navngitt ved de variabler som forekommer i begge relasjonene. De to innputtene behøver ikke være av samme type.
Eksempel: An(1)

X	Y
Anne	3
Ole	4

og An(2):

X	Z
Anne	Marit
Berit	Geir

medfører Bn(2):

X	Y	Z
Anne	3	Marit

Rete reduserer eller eliminerer visse typer redundans ved hjelp av nodedeling. Den lagrer partielle treff når den utfører Join i Betanodene ved kjøring av systemet, dette medfører at systemet unngår en total revaluering av alle fakta hver gang endringer gjøres i arbeidsminnet. Systemet behøver kun å evaluere endringer. I tillegg legges det til rette for effektiv fjerning av elementer i minnet når fakta fjernes fra arbeidsminnet. Når et faktum legges til eller fjernes fra arbeidsminnet, vil et token som representerer faktumet og operasjonen komme inn til rotnoden. En kopi av tokenet vil flytte seg nedover til Alfanodene hvor en SELECT blir utført, den velger da ut kun de tokens som er av dens type.

1.8.1 Eksempel ved bruk av Rete

Ved bruk av Rete kan rekkefølgen på elementene en regel er bygd opp på, ha stor betydning. Dette lar seg best vise ved hjelp av et lite eksempel:

*REGEL*1	*REGEL*2
(finn-treff ?x ?y ?z ?w)	(element ?x)
(element ?x)	(element ?y)
(element ?y)	(element ?z)
(element ?z)	(element ?w)
(element ?w)	(finn-treff ?x ?y ?z ?w)

Både regel 1 og regel 2 bruker 5 Alfanoder og 4 Betanoder. Forskjellen ligger i antall elementer som blir lagret.

Minnet for noder ved regel 1:

NODE	*PATTERN*	*MINNE*
A1	(finn-treff ?x ?y ?z ?w)	F1
A2	(element ?x)	F2 F3 F4 F5 F6 F7 F8
A3	(element ?y)	F2 F3 F4 F5 F6 F7 F8
A4	(element ?z)	F2 F3 F4 F5 F6 F7 F8
A5	(element ?w)	F2 F3 F4 F5 F6 F7 F8
B2	A1 & A2	F1
B3	B2 & A3	F1
B4	B3 & A4	F1
B5	B4 & A5	F1

Ved denne regelen har vi totalt 1+7+7+7+7+1+1+1+1=33 elementer

Minnet for noder ved regel 2:

NODE	*PATTERN*	*MINNE*
A1	(element ?x)	F2 F3 F4 F5 F6 F7 F8
A2	(element ?y)	F2 F3 F4 F5 F6 F7 F8
A3	(element ?z)	F2 F3 F4 F5 F6 F7 F8
A4	(element ?w)	F2 F3 F4 F5 F6 F7 F8
A5	(finn-treff ?x ?y ?z ?w)	F1
B2	A1 & A2	[F2 F2] [F2 F3] ... [F8 F8]
B3	B2 & A3	[F2 F2 F2] ... [F8 F8 F8]
B4	B3 & A4	[F2 F2 F2 F2] ... [F8 F8 F8 F8]
B5	B4 & A5	F1

Ved denne regelen har vi totalt
7+7+7+7+1+(7*7)+(7*7*7)+(7*7*7*7)+1 = 2823 elementer

Forklaring: Dette viser hvor mye minne man kan spare når Alfonoder joines i Betanodene, da må man ta hensyn til dette ved oppbygging av reglene. Dersom vi hadde hatt begge disse reglene i det samme systemet, ville Rete gjenbrukt Alfanodene fra regel 1 i regel 2. Hadde elementene i betingelsene i regel 2 hatt samme rekkefølge som i regel 1, kunne Rete brukt de samme Betanodene. I dette tilfellet kommer de i en annen rekkefølge, og Rete vil derfor konstruere et helt nytt sett med Betanoder. Hadde det vært flere regler i dette systemet, kunne det vært tilfellet at noen av Betanodene kunne brukes om igjen mens andre måtte legges til fra nytt. Dette viser at man kan få mindre eller større, raskere eller tregere nettverk dersom man endrer den interne rekkefølgen i reglene.

Ved bruk av Rete får man en effektiv mekanisme for å finne og kjøre subsett av reglene som er relevante til et bestemt datasett, spesielt når kun noen få av et stort sett av regler er forventet å kjøre. Da bygges en optimalisert nettverksstruktur av reglene, og forplanter objektene gjennom denne strukturen. Dette tillater regelmotoren å effektivt identifisere regler som skal kjøres, uavhengig av rekkefølgen disse reglene har blitt definert i, og også unngå reprosesseringen av dupliserte betingelser.

Som tidligere nevnt, den retebaserte eksekveringen oppnår effektivitet ved å konstruere optimaliserte nettverk og effektivt håndtere forplantningen av objekter gjennom nettverket. Objekter som behøver en revaluering på grunn av handlingsdelen av tidligere regler, spores og reprosesseres.

1.9 Sekvensiell utføring

I motsetning til retebasert eksekvering, vil en sekvensiell basert utføring eksekvere reglene i den rekkefølgen de ble lagt inn eller basert på en eksplisitt definert rekkefølge. I dette tilfellet blir det ikke sporet endringer på objektene for å identifisere ytterligere regler som skal kjøres. Den enkle metoden for eksekvering, sammen med tiden spart på å unngå den ekstra tiden det tar å bygge og vedlikeholde retenettverket og i tillegg forplante objektene, står for den større hastigheten til den sekvensielle utføringen. Sekvensiell modus er spesielt verdifullt for applikasjoner som blir kjørt i grupper, hvor beslutningstaking kan modulariseres slik at all data som er nødvendig for å eksekvere et sett av regler er tilgjengelig når steget initieres.

Tilfeller der hvor sekvensiell utføring kan produsere en meget god ytelse uten å ofre viktig fleksibilitet er når man har et stort antall objekter, spesielt hvor objektmodellen er dyp og hvor man behøver betydelig navigering i modellen. Det vil si når regler krever evaluering av subobjekt, noen ganger også flere nivå av subobjekt. Denne utføringen er også svært gunstig når man ikke krever en revaluering av reglene, slik at en kjøring av en regel ikke trigger en revaluering av tidligere evaluerte regler. Det bør da ikke være mange betingelser som deles på tvers av flere regler. Et annet tilfelle hvor sekvensiell utføring er egnet, er når en betydelig andel av reglene i et regelsett behøver å kjøres for hver komplette evaluering eller når det er store volum hvor det er strenge tidsbegrensninger.

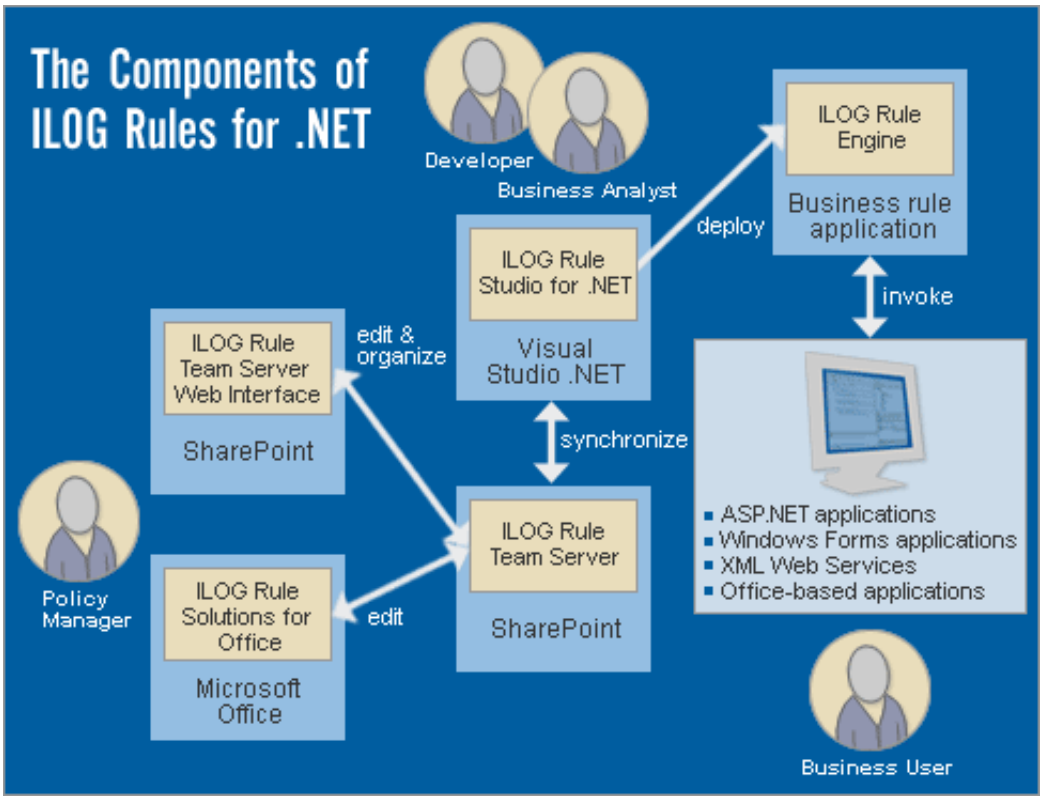
Kapittel 2

ILOG Rules for .NET

ILOG Rules for .NET er et håndteringssytem for forretningsregler som ble brukt til å implementere caset i denne oppgaven. Dette kapitlet går nærmere inn på hvilke komponenter dette systemet består av og hvordan man bruker det.

Til slutt i kapitlet kommer en kort nøytral evaluering av ILOG Rules for .NET i forhold til andre aktører på markedet.

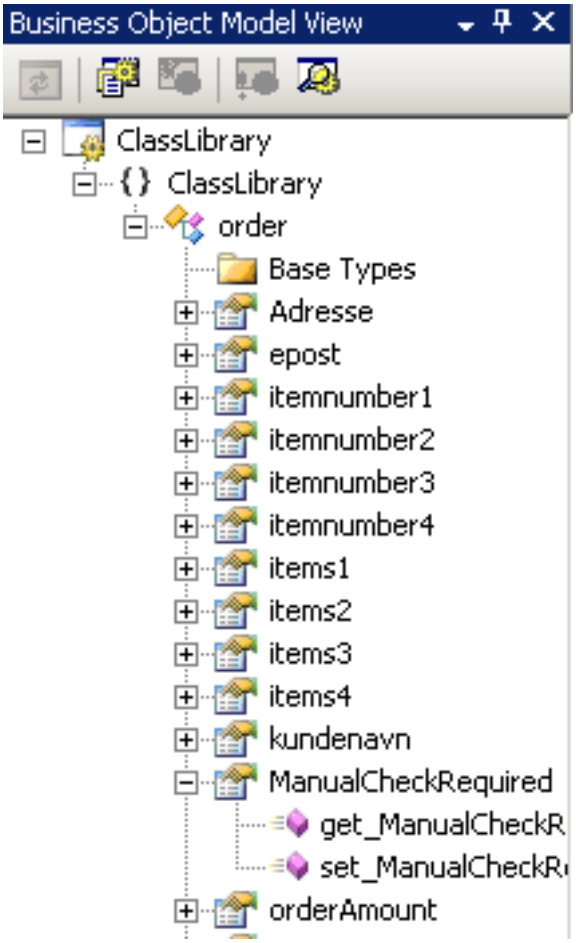
2.1 Komponenter



Figur 2.1: ILOG Rules for .NETs elementer

ILOG Rules for .NET mapper forretningsobjektmodeller til ord og setninger som er kjente for de som skal bruke systemet når det er ferdig implementert. En forretningsobjektmodell (Business Objekt Model, BOM) er språket som brukes i den organisasjonen som reglene påvirker. En liten tommelfingerregel er at i de fleste tilfeller består forretningsobjektmodellen av de substantivene som brukes i reglene. Tidlig i utviklingsprosessen må systemutviklerne og forretningsbrukerne bli enige om en slik forretningsobjektmodell, før denne modellen kan knyttes til klasser i .NET eller XML-skjema. Denne koblingen kan være ganske komplisert og må utføres av en systemutvikler som behersker .NET.

Et verktøy kalt Business Object Modell View, figur 2.2, tilbys for å kunne visualisere forretningsobjektmodellen.



Figur 2.2: Business Object Model View

I de aller fleste tilfeller kan opprettelse av forretningsobjektmodellen forenkles betraktelig ved å importere allerede eksisterende .NET-klasser ved å bruke en egen veiviser i modellen. Når en klasse først er importert, vil den vises i modelloversikten og kan endres slik at den er i overensstemmelse med den domenespesifikke verbaliseringen som ble spesifisert i definisjonen av forretningsobjektmodellen.

Fordi, som nevnt under 3.1 Regelmotor, rekkefølgen reglene legges inn i systemet på kan ha utslag på resultatet, tilbyr ILOG at vi styrer rekkefølgen reglene kjøres i. Dette gjøres ved et grafisk grensesnitt som styrer flyten av reglene.

ILOG har samarbeidet med Microsoft om å lage en tilleggsfunksjon i Visual Studio slik at det gjør skriving og håndtering av forretningsregler like lett som å skrive ordinær kode i Visual Studio. Gjennom Visual Studio kommer også Rules for .NET med muligheter for bruk innenfor et Windows Workflow-miljø. I tillegg kan utviklerne blant annet utnytte verktøy som re-factoring og Windows Forms i kombinasjon for å implementere regelapplikasjoner hurtigere.

2.1.1 Rule Studio

ILOG Rule Studio for .NET er et sett av plug-ins som i sin helhet er integrert i Visual Studio, noe som tillater brukere å fokusere på å lære forretningsregelteknologier istedenfor et nytt utviklingsmiljø. Dette er miljøet for forretningsregelapplikasjoner, stedet hvor man skriver applikasjonskoden. Brukerne kan så definere vokabularet til reglene ved å kommentere modellene i Rule Studio. Via Rule Studio kan man også teste reglene ved å kjøre de gjennom en instans av regelmotoren.

Forretningsobjektmodellen blir definert i Rule Studio, domenet hvor reglene har sin gyldighet. Denne modellen må være assosiert med et Visual Studio .NET Class Library-prosjekt. Rule Studio brukes for å legge til dette biblioteket. Modellen kan inneholde typer som er definert innenfor prosjektet, i tillegg til typer som refereres av prosjektet. Referansene kan være andre prosjekt innenfor den samme løsningen. I tillegg kan modellen inkludere XML-baserte klasser ved å generere .NET-klasser fra XML-skjema. Dette gjøres ved å bruke verktøyet XmlSchemaCodeGenerator som er inkludert i ILOG Rules for .NET.

2.1.2 ILOG Rule Team Server for SharePoint

Forretningsregelapplikasjoner må designes slik at de setter forretningsbrukerne i stand til å vedlikeholde og håndtere forretningsregler gjennom hele deres livssyklus. ILOG støtter dette ved å tilby Rule Team Server for SharePoint, et dokumentsentrert lager for forretningsregler -bygd på topp av Microsoft Windows SharePoint Services. Det har en nettbasert regeleditor, bygd opp av deler som er konfigurerbare komponenter i Windows SharePoint Services. Disse delene kan brukes til å tilby et personalisert brukergrensesnitt for forretningsbrukerne. Via regeleditoren kan regler endres til og med fra eksterne kilder.

Forretningsreglene organiseres i lageret som bibliotek, mapper og dokument, såkalte RuleDocs. Dette lagret fungerer som et langvarig oppbevaringssted for reglene, som gjør det enkelt for brukerne å finne reglene som de skal jobbe med. Et RuleDoc er et Microsoft Office dokument som inneholder forretningsregler.

Brukergrupper som jobber på store forretningsregelapplikasjoner må ofte samarbeide med hverandre. De kan jobbe sammen på forretningsreglene på en mer effektiv måte ved å utnytte samarbeidsstøtten Windows SharePoint Services tilbyr, dette inkluderer en oversikt over hvilke brukere som er til stede, diskusjonsgrupper og oppgavelister.

Tilgang og tillatelser til forretningsreglene i lageret håndteres ved å bruke en innebygd Windows SharePoint Services tilgangskontroll. Muligheten til å legge til, slette eller endre reglene avhenger av rollen brukeren som ønsker å modifisere regelen har. En administrator kan definere brukere og tilegne dem til grupper som pålegger de tilgang til dokumenter.

Ved å bruke filversjonsmulighetene som tilbys, kan brukerne håndtere ulike versjoner av dokumenter for revisjon og tilbakerulling. Det er også et annet viktig aspekt ved å samarbeide som ikke må glemmes, nemlig muligheten til å låse dokumenter. Muligheten til fillåsing i Windows SharePoint Services gjør at ILOG støtter at flere brukere håndterer reglene samtidig. Figur 2.3 viser hvor lett det er å holde applikasjonen synkronisert med de gjeldene reglene.

Specify the actions to perform (right-click):

!	Rule Name	Rule Studio	Team Server	Action
	Price.BaseRate	Unchanged	Unchanged	None
	QualifyFor.LongTermDiscount	Unchanged	Unchanged	None
	Price.LongTermDiscount	Unchanged	Modified	Synchronize
	QualifyFor.DefaultDiscount	Unchanged	Unchanged	None
	Price.DefaultDiscount	Unchanged	Unchanged	None
	Price.Coverage	Unchanged	Unchanged	None

Figur 2.3: Synkronisering av regler i Rule Studio

Når utviklerne overlater systemet til forretningsbrukerne, publiseres reglene til miljøet hvor reglene kan skrives via Rule Team Server for SharePoint. I dette miljøet kan brukerne endre reglene ved å benytte et web-basert grensesnitt eller det Microsoft Office-baserte grensesnittet ILOG Rule Solutions for Office.

Det nettbaserte grensesnittet til Team Server er brukerens grensesnitt til

Team Server. Ved å bruke dette grensesnittet oppretter de, endrer og organiserer reglene og beslutningstabellene som er lagret ved å bruke Rule Team Server. Delene grensesnittet består av inkluderer en nettleser for RuleDocs, selve regeleditoren og en editor for reglenes egenskaper. Brukerne kan endre reglene og forretningstabellene.

Alle regler blir utgitt til SharePoint som dokumenter i format av XML.

2.1.3 ILOG Rule Solutions for office

ILOG Rule Solutions for Office lar brukerne endre forretningsreglene ved å benytte kjente applikasjoner fra Microsoft Office. Det øker produktiviteten og reduserer læringskurven ved å la brukerne endre og håndtere reglene med verktøy de er vant til å bruke. For tiden støtter ILOG integrasjon med Microsoft Office Word 2003 og Microsoft Office Excel 2003. Ved hjelp av dette verktøyet kan brukerne skrive og håndtere forretningsregler i RuleDocs ved å bruke en regeleditor som er tilbudt inne i Word og Excel, mens de utnytter samarbeidstrekkene til SharePoint gjennom et delt Workspace vindu. Når du bruker Microsoft Office Word eller Excel som regeleditor, gir det brukerne muligheten til å skrive regler selv når de ikke er tilkoblet bedriftens nettverk.

Dokumentene som inneholder forretningsreglene og beslutningstabellene, er XML-dokumenter, såkalte RuleDocs. Dette betyr at brukerne kan ta fordel av egenskapene til Office for å se på og håndtere XML-filer, de vil benytte Microsoft Word for forretningsreglene og Microsoft Excel for beslutningstabellene.

2.1.4 Regelmotoren

Forretningsregelapplikasjoner invokerer reglene ved å kalle regelmotoren. ILOGs regelmotor tilbyr både RETE og sekvensiell utføring. De anbefaler [5] å bruke RETE når man skal fatte beslutninger som omhandler inferens, dette kan være beregninger eller kliniske beslutningsanbefalinger. For enklere beslutninger, slik som validering, er det best å bruke sekvensiell utføring.

Regelmotoren er en naturlig .NET-assembler og tilbyr et API for å innkapsle regelmotoren inn i en .NET-applikasjon ved å bruke et hvilket som helst programmeringsspråk som retter seg mot Microsofts .NET-rammeverk. Regelmotoren avdekker også et API som kan brukes til å spore eksekvering av regler og i tillegg logge hendelsene i regelmotoren. En arkitekt kan bruke regel-

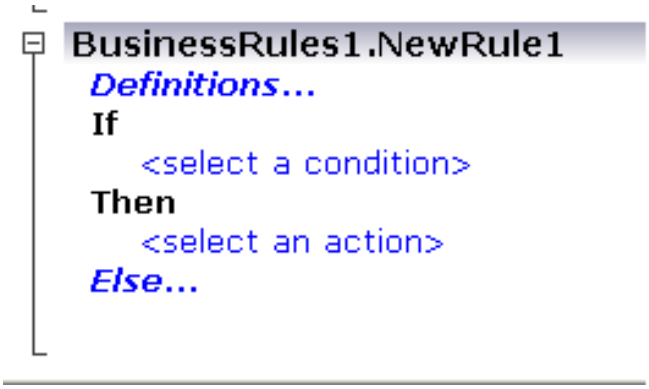
motoren til ulike .NET-applikasjoner på klientsiden (WinForms og Office-baserte applikasjoner) eller på serversiden(ASP.NET, Windows Services og XML Web Services).

De fleste .NET regelapplikasjoner er bygd ved å bruke en lagvis modellkomponentarkitektur. Fra en arkitekts perspektiv, passer ILOGs regelmotor og dens tilhørende forretningsregler inn i Microsofts foreskrevne applikasjonsarkitektur.

2.2 Oppretting av regler

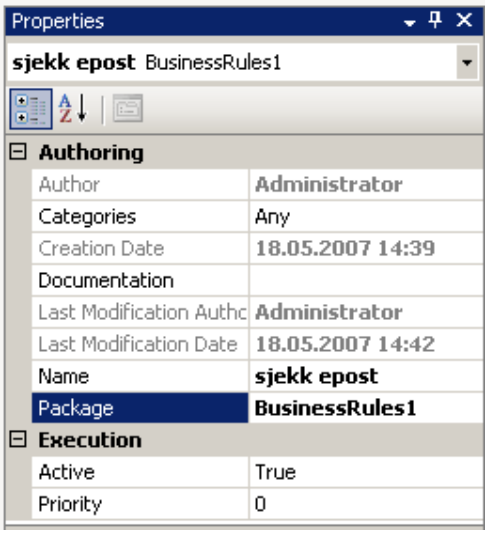
Det finnes flere ulike definisjoner av en forretningsregel, ILOG har kommet opp med sin egen som de benytter: "En hvilken som helst påstand på formen 'IF kriterium THEN handling', så lenge den er i interesse for bedriften".

ILOG Rule Studio for .NET tilbyr en pek-og-klikk regeleditor og integrerte verktøy som forenkler oppgaven med å skrive og håndtere reglene. Figur 2.4 viser utgangspunktet man har for å skrive en forretningsregel i ILOG.



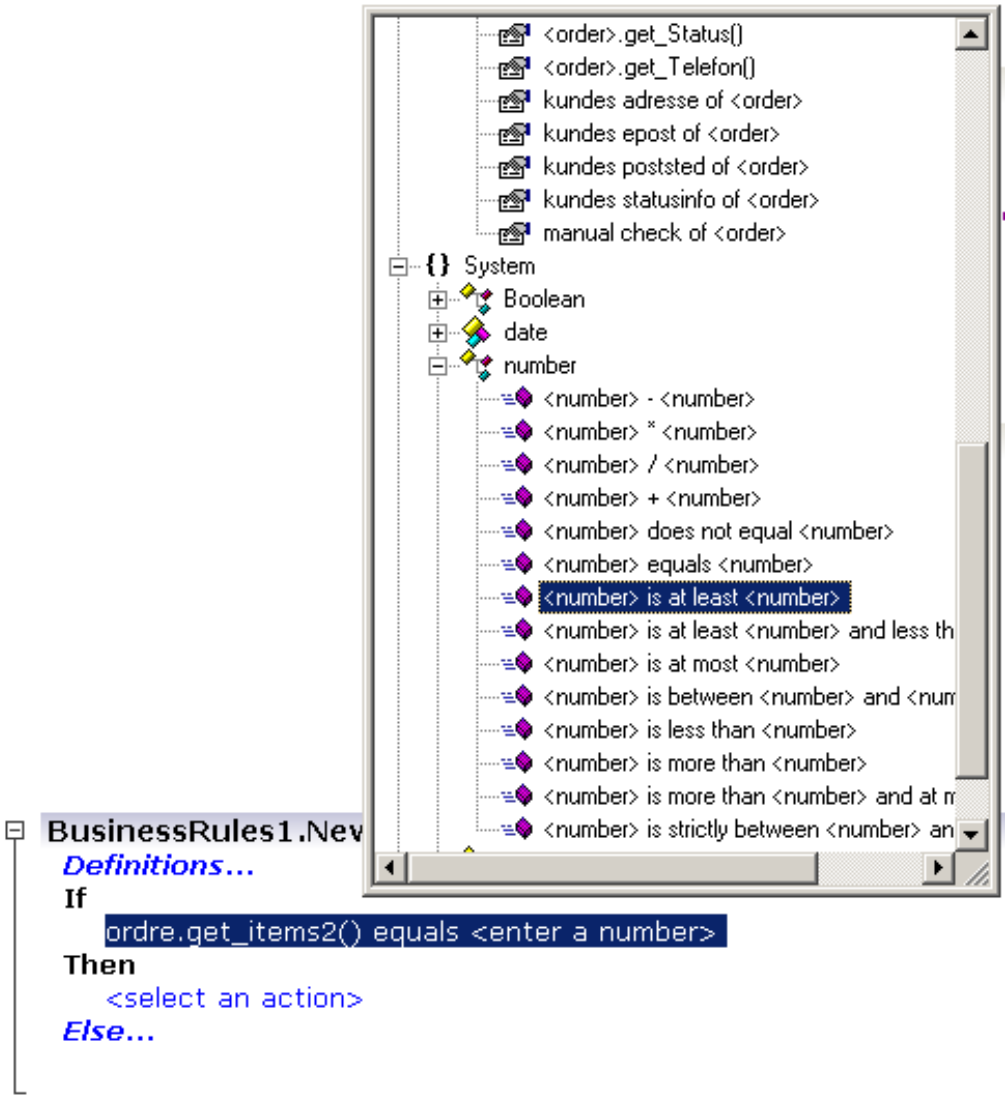
Figur 2.4: Utgangspunkt for regelskriving

Det er et eget egenskapsvindu knyttet til hver regel, når man skal endre de vanligste egenskaper til en regel, dette kan for eksempel være navn eller prioritet, gjøres dette enkelt ved å skrive inn ønsket verdi i dette vinduet. Se figur 2.5.



Figur 2.5: Egenskapsvindu for regel

For å gjøre det enklere å skrive syntaktisk korrekte regler, kommer det en rullegardinmeny når man skal opprette eller endre en regel. Der vises en oversikt over hvilke verdier man har mulighet til å sette inn i en aktuell del av regelen. Dette vises i Figur 2.6.



Figur 2.6: Rullegardinmeny for å skrive en regel

Etter hvert som man fyller inn delene i regelen, vises dette i vinduet. De til enhver tid syntaktisk lovlige input i en regel vises i rullegardinmenyen. Rule Studio tilbyr også verktøy for å organisere og søke i regler, og også for å holde reglene synkronisert med lageret av regler i ILOG Rule Team Server for SharePoint.

2.3 Integrasjon mellom BizTalk og ILOG

Microsoft BizTalk har en innebygd regelmotor, men ofte er det nødvendig å bruke Rules for .NET istedenfor. Den innebygde regelmotoren fungerer godt dersom man har prosjekter uten behov for et stort antall regler, dette sparer store kostnader og er samtidig svært enkelt og oversiktlig. Dersom dette ikke er tilfellet og man har behov for mange regler, håndtere beslutningsintensive prosesser eller ønsker å tillate brukerne å håndtere deres egne regler, er det en god ide å bruke Rules for .NET i kombinasjon med BizTalk.

En integrasjon mellom BizTalk og Rules for .NET består av å kalle ILOGs regelmotor direkte fra BizTalks Orchestrations. Jeg skal ikke gå nærmere inn i Microsofts Biztalk, da dette blir på kanten av hva som er relevant for denne oppgaven.

2.4 Uavhengig evaluering av ILOG i forhold til andre aktører på markedet

Forrester er et uavhengig firma som utfører analyser innen ulike teknologier, de opplevde en meget stor etterspørsel fra sine klienter angående regelmotorer, de besluttet derfor å utføre en inngående evaluering av de største aktørene på markedet. Da jeg kun har tatt for meg en av regelmotorene på markedet, ILOG Rules for .NET, synes jeg det er interessant å se hvor den ligger an i forhold til andre produkter. I tillegg synes jeg denne artikkelen er god å ha med som et referansepunkt for videre lesing for brukere som er interessert i kvaliteten til andre produkter på markedet. I denne testen ble både ILOG JRules og ILOG Rules for .NET testet. ILOG JRules er et annet produkt fra ILOG, det er et mer etablert håndteringssystem for forretningsregler basert på Java istedenfor .NET. De ble evaluert hver for seg fordi de er to ifølge Forrester er svært forskjellige produkter. Jeg vil i dette kapitlet stort sett konsentrere meg om resultater angående ILOG Rules for .NET, men der det er relevant vil også andre leverandører og produkter bli trukket inn. Noe som er verdt å nevne er at det er brukt en tidlig versjon av Rules for .NET i denne testen, det har nå kommet en helt ny versjon på markedet.

Gjennomføringen av denne evalueringen gikk ut på at hver regelmotor ble bedømt etter 174 kriterier. Disse kriteriene ble oppdelt i 3 båser: Hva de tilbyr i dag, strategien deres og tilstedeværelse i markedet.

Hva som tilbys i dag: En leverandørs plassering på den vertikale aksen i

figur 2.7 indikerer styrken til produktets egenskaper. I dette inngår plattformens arkitektur, utvikling, utviklingsverktøy, verktøy for forretningsbrukere, håndteringsmuligheter for livssyklusene til forretningsreglene, anvendelse og administrasjon og interoperabilitet.

Strategi: En leverandørs plassering på den horisontale aksen i figur 2.7 indikerer styrken til produktet og bedriftsstrategien, i tillegg til produktets kostnader og finansielle styrke.

Tilstedeværelse i markedet: En leverandørs størrelse på prikk i figur 2.7 indikerer dens tilstedeværelse i markedet. Dette måler Forrester i antall ansatte, antall partnere, størrelse på installert base og implementerings- og opplæringstjenester som tilbys kundene.

Rent praktisk ble testen gjennomført ved å kombinere fire kilder til informasjon. Hver leverandør fikk et møte med en rekke analytikere og det ble utført scenariobaserte tester. I tillegg ble leverandørene utspurt angående strategier og kostnader, dette er informasjon som ble kontrollert ved hjelp av offentlige kilder når det var mulig. Leverandørene fikk også mulighet til å vise fram en demonstrasjon av produktets egenskaper, dette omhandlet spesielt adminstrerings- og håndteringsverktøy. Til slutt ble eksisterende kunder av leverandørene rådspurt for å validere hva Forrester hadde kommet fram til.

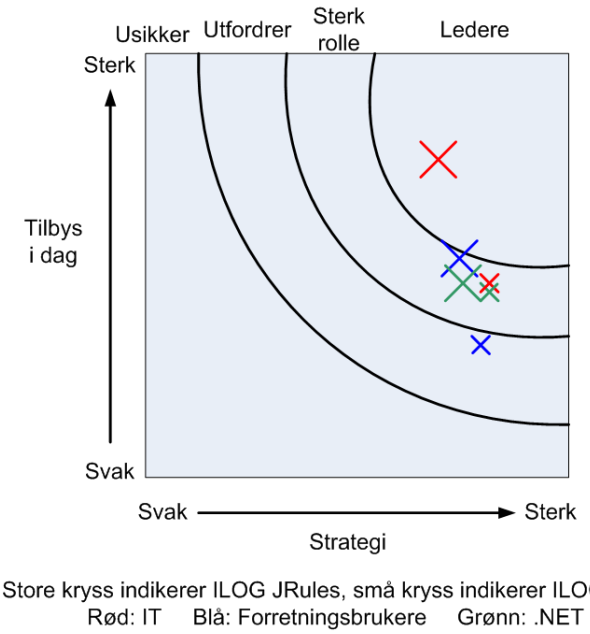
Først ble det testet hvilket produkt som gir IT kontroll over utvikling og anvendelse. I denne kategorien kom ILOG Rules for .NET i den nederste kategorien sammen med 5 andre produkter, selv om ILOG Rules for .NET fokuserer på denne brukergruppen. Dette skyldes i stor grad at ILOG Rules enda er i tidlige versjoner med begrensede egenskaper. ILOG Rules har kun en brøkdel av ILOG JRules sine egenskaper, dette gjør seg spesielt synlig innen områdene administrasjon og anvendelse. Der ILOG Rules gjør det meget bra, er innen bruk av Microsoft Visual Studio .NET, Visual SourceSafe, SharePoint Portal Services og Office. IlOG planlegger å legge inn flere egenskaper i Rules for .NET.

Neste runde med tester omhandlet produktenes egenskaper for forretningsbrukere. Verktøy for forretningsbrukere er planlagt i neste versjon av ILOG Rules, men pr. dags dato har ikke ILOG Rules noe å stille opp med mot sine konkurrenter på dette området.

ILOG Rules gjør det bedre under testen for brukere av .NET. Noe som er verdt å merke seg, er at ILOGs JRules, altså versjonen for Java, scorer bedre i .NET-testen enn ILOG Rules som er bygd for .NET. Grunnen til dette er at JRules har et mye større spekter av verktøy og interoperabilitetspoeng med Microsoft .NET. Ved slipp av neste versjon av ILOG Rules vil disse poengene

endre seg og Rules vil score bedre enn JRules for brukere av .NET.

Som en oppsummering kan man si at ILOG Rules for .NET har en lang vei å gå før den kan konkurrere på linje med andre markedsledende regelmotorer. Brukere som benytter seg av .NET kan, ifølge Forrester, betrakte ILOG Rules som et godt alternativ. Man bør også se på Rules som et tegn på ILOGs engasjement til å tilpasse og integrere plattformer i framtidige produkter. Rules introduserer interessante ideer for å utvikle engasjement blant brukere som ikke er programmerere ved å bruke Microsoft Office-applikasjoner.



Figur 2.7: ILOG Rules for .NET i forhold til ILOG JRules [12]

Kapittel 3

Implementering

3.1 Case

En ordre kommer inn til vårt system på XML-format. Ordren har følgende felter som omhandler kundeinformasjon:

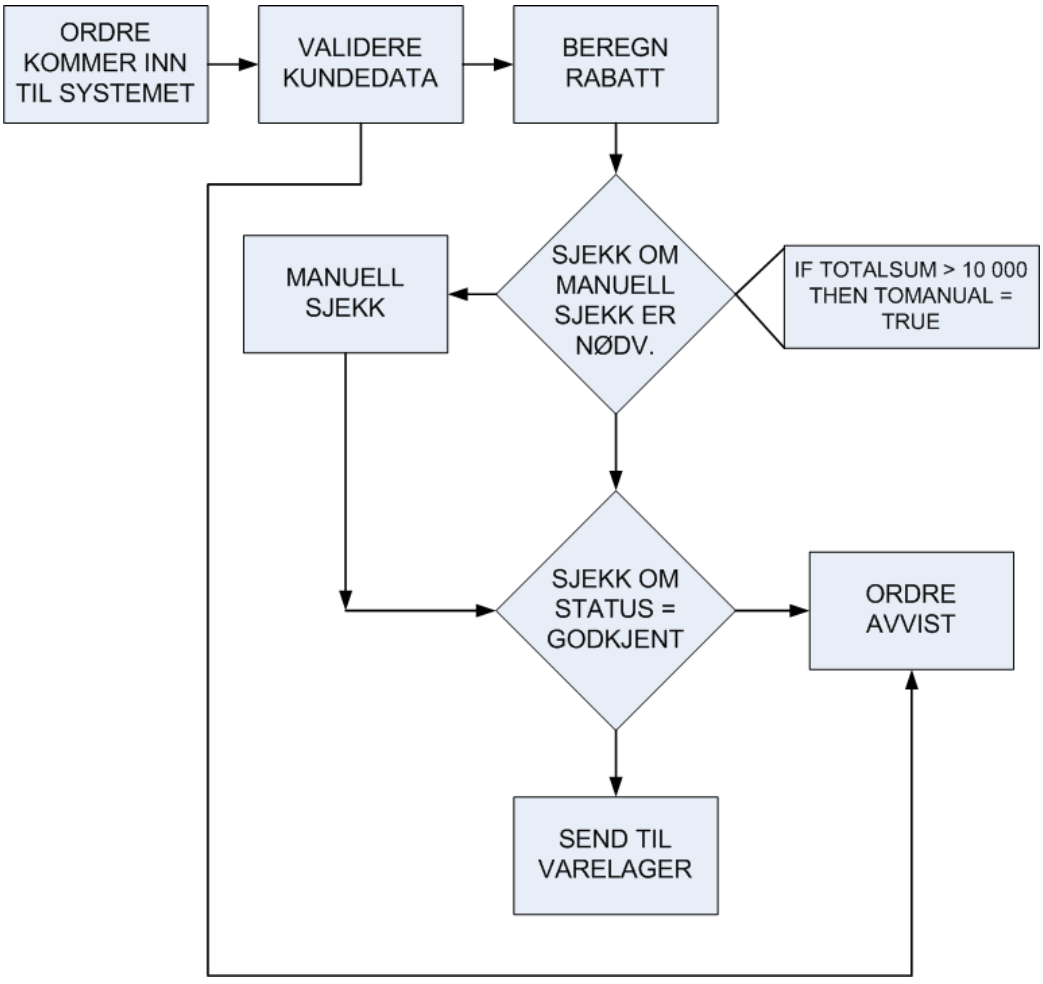
- Navn
- Adresse
- Postnummer
- Sted
- Telefonnummer
- E-post
- Dato

I tillegg til feltene som omhandler personalia, består hver ordre av minst en ordrelinje. Hver ordrelinje inneholder følgende felter:

- Varenummer
- Antall
- Pris pr. stk.
- Sum

Til slutt i ordren er det et felt for totalsum ordre.

Når ordren kommer inn til systemet, må ordren først valideres. Dette betyr at man må kontrollere at alle felt er korrekt utfylt. Denne kontrollen sjekker at det er fylt inn gyldige verdier. I regelmotoren skal det opprettes et eget regelsett som omhandler evalueringen. I XML-filen blir feltet <status> satt til godkjent eller avvist, avhengig av utfallet av denne valideringen. Dersom ordren kommer gjennom valideringen vil ordren gjennomgå en beregning av rabatt. Denne rabatten er avhengig av størrelsen på ordrebeløpet. Deretter testes det om totalbeløpet er over en viss sum. I dette tilfellet vil feltet <tomanual> bli satt til true og ordren blir sendt til manuell behandling. I motsatt tilfelle blir ordrens status satt til godkjent og sendt til ordrekontoret for fullføring.



Figur 3.1: Behandling av ordren

3.2 Gjennomføring

3.2.1 Valg og begrensning

Man har flere valg for å bestemme hvordan en ordre skal komme inn til systemet, jeg har valgt å la ordren komme i format av enXML-fil. En annen løsning som ble vurdert var bruk av database. I sistnevnte tilfelle ville kunden fylt ut et skjema på nett og de innfylte data ville bli lagt i en database. Regelmotoren hadde så hentet ut de nødvendige data fra databasen når det var behov for de. Av flere årsaker ble denne løsningen ikke fulgt videre opp. En av disse er at ved et system slik som det jeg har implementert, får man ofte behov for å endre de data som skal uveksles. I en XML-fil lar dette seg lett gjøre i en enkel teksteditor. Bruker man derimot en database, blir spørringene gjerne komplekse og tidkrevende, selv for små endringer. I tillegg er XML et uavhengig format som støttes av de fleste systemer. I konsultasjon med Jan Fredrik Øveråsen ved Abeo AS i Oslo, ble vi i fellesskap enig om at den beste løsningen for mitt program vil være å benytte XML.

Utfylling av skjema / grafisk brukergrensesnitt, generering av XML fra utfylt skjema og oversending av skjema inn til regelmotoren er ikke tatt hensyn til i min implementasjon. Dette ble sett på som mindre viktig i forbindelse med denne oppgaven, da det interessante var å se på implementering og bruk av valgt system. Det ble derfor, av omfangshensyn, prioritert behandlingen av ordren fra det punktet når den kommer inn til systemet. Av denne grunn genererte jeg flere XML-skjema manuelt for å teste at systemet oppførte seg slik det skulle.

Jeg valgte i utgangspunktet sekvensiell utføring fordi det er mest velegnet når systemet skal brukes til å validere data, dette ifølge ILOG selv. Etter hvert som oppgaven kom i gang valgte jeg å også benytte Rete, for å se om det ble en betydelig endring i kjøretid.

ILOG Rules for .NET tilbyr to elementer som jeg ikke brukte i implementeringen av ordresystemet, det er Rule Team Server for SharePoint og Rule Solutions for Office. Sistnevnte ble ikke installert fordi jeg ikke har Microsoft Office på maskinen jeg brukte til å implementere løsningen. Rule Solutions for Office er mest velegnet for erfarne brukere, da det gjør skriving av regler mer effektivt. Det tilfører ingen nye aspekt til implementeringen og jeg valgte derfor ikke å installere Office og dermed heller ikke Rule Solutions for Office. RuleTeamServer for SharePoint ble ikke benyttet av flere årsaker, blant annet er SharePoint ikke installert på maskinen som ble brukt. I tillegg så

jeg ikke på det som nødvendig for å gjennomføre en full implementering av ordresystemet.

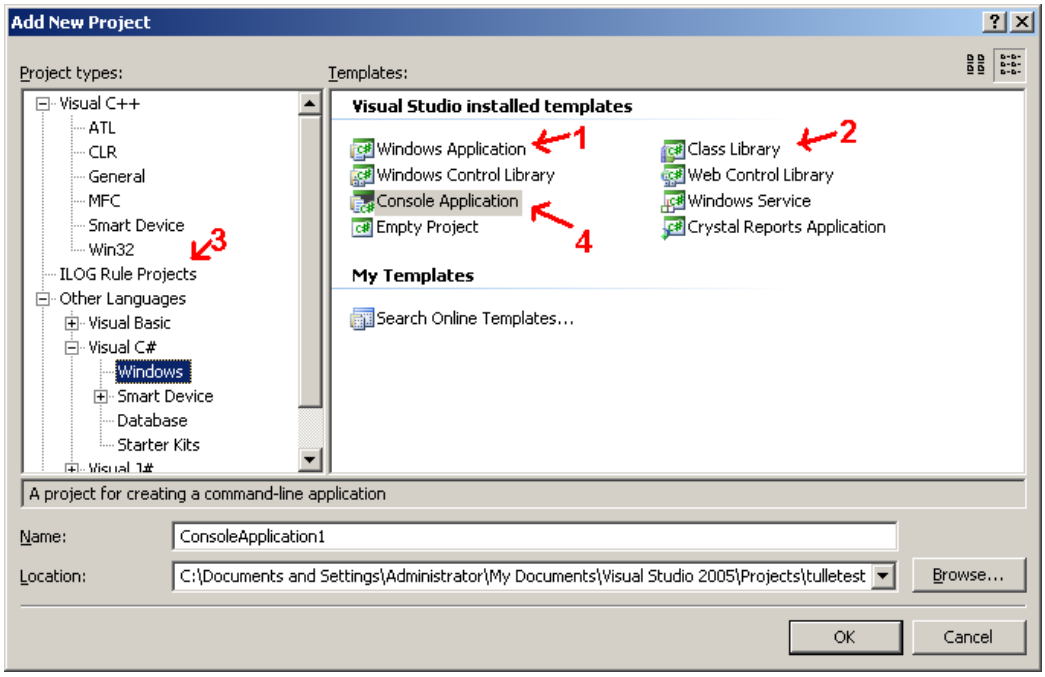
For å vise flere sider av systemets egenskaper, bestemte jeg meg for å benytte både en beslutningstabell og ordinære regler for å implementere løsningen min. En beslutningstabell er mest velegnet i tilfeller der beregninger, valg eller handling kan få mange forskjellige utfall avhengig av verdien på objektet vi ønsker å evaluere. I mitt case er de fleste av dataene som skal evalueres enten/eller, slik at de ikke er egnet for bruk i en beslutningstabell. Jeg valgte derfor å implementere utregning av rabatt i beslutningstabellen. Det ga meg muligheten til å legge inn ulike rabattprosenter for ulike beløpsintervall som totalbeløpet av ordren kan ha.

Oppgavebeskrivelsen åpnet for mange ulike muligheter for validering og godkjenning av ordren som kommer inn til regelmotoren. Jeg valgte å først gå igjennom feltene som omhandlet informasjon om kunden. Der testet jeg om obligatoriske felter er tomme, om de inneholder nødvendige tegn og at numeriske verdier er innenfor gyldige intervall. Dersom et av kriteriene feiler, blir ordrens status satt til ugyldig. I tillegg til at ordren blir satt til ugyldig, har jeg også valgt å innføre en variabel tilknyttet hver ordre. Denne ordren har jeg kalt statusordreinfo og er av type string. Når et av kriteriene feiler, blir dette feltet oppdatert med en verdi som forklarer hvorfor testen feilet. Når eksekveringen av regler er fullført, blir det utført en test i Program.cs. Testen sjekker statusen til ordren, dersom ordren feilet valideringen vil statusordreinfo skrives ut. Dermed får brukeren vite hvorfor valideringen av ordren feilet. Kontroll av regelflyten ga meg muligheten til å stoppe eksekveringen av reglene, slik at dersom ordrens status blir satt til ugyldig vil ingen flere regler bli kjørt. Følgende tester ble utført på kundedata:
Navn: ikke tomt
Adresse: ikke tomt
Postnummer: et tall mellom 0 og 9999
Telefonnummer: et tall mellom 100 og 99999999
e-post: ikke tomt, samt at felt må inneholde en @

Se vedlegg for å se noen av testene som ble kjørt for å validere kundedata.

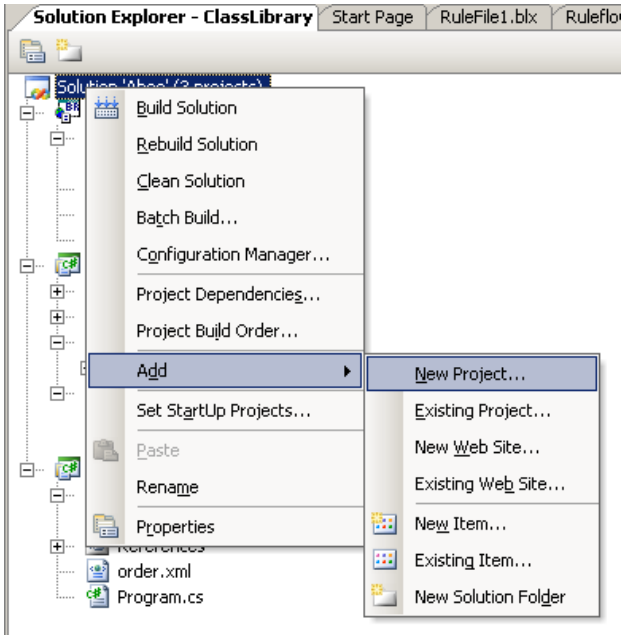
3.2.2 Implementeringen

Det første som må gjøres er å opprette en ny løsning, i Visual Studio kalles dette en Solution. Når du oppretter en ny løsning, er dette av typen Windows Application. Detter er punkt 1 i figur 3.2.



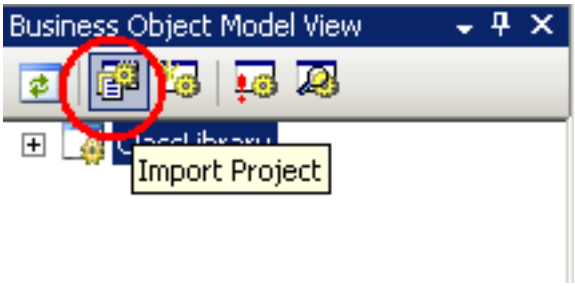
Figur 3.2: Oversikt over hvilke prosjekter som kan legges til

En av de grunnleggende tankene bak Rules for .NET er bruken av forretningsobjektmodellen, Business Object Model. Opprettelse av modellen er forholdsvis enkelt. Først må man legge til et prosjekt av type bibliotek til løsningen. For å legge til et nytt prosjekt til løsningen høyreklikker man på løsningen i løsningsutforskeren, Solution Explorer, og får opp figur 3.3:



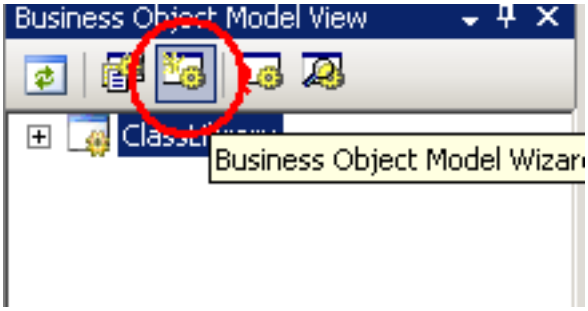
Figur 3.3: Legge til et nytt prosjekt til løsningen

Når man følger denne veiviseren kommer man nok en gang til vinduet i figur 3.2, hvor man kan velge hvilken type nytt prosjekt man ønsker å legge til løsningen sin. Denne gang velger man et prosjekt av type Class Library, mitt bibliotek kalte jeg ClassLibrary. Deretter må dette biblioteket importeres i forretningsobjektmodellen, da klikker man på "import project", figur 3.4, i Business Object Model View og får opp en liste over hvilke prosjekt man kan importere.



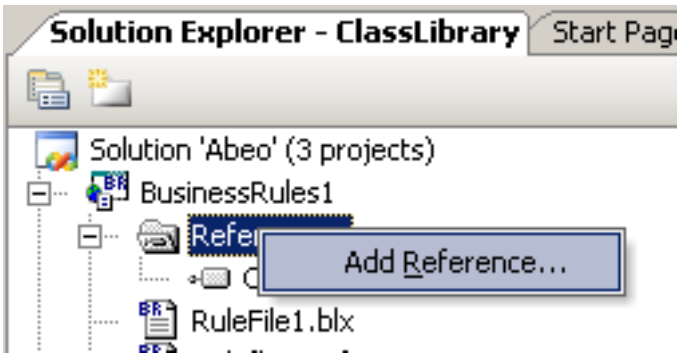
Figur 3.4: Importere biblioteket

Man velger da biblioteket. Nå når biblioteket er importert, må man benytte forretningsmodellveiviseren, Business Object Model Wizard, for å importere klassene til biblioteket. Denne vises i figur 3.5



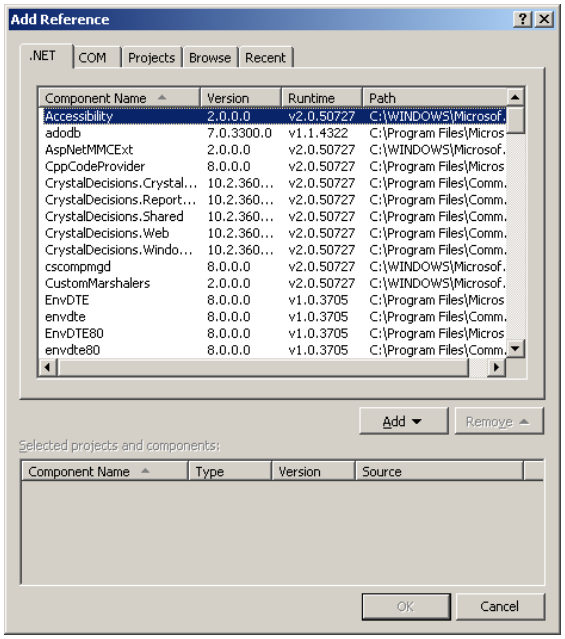
Figur 3.5: Importere klassene

Nå når grunnlaget er lagt, må man definere forretningslogikken. Det er her reglene kommer inn i bildet. Da velger man å legge til et nytt prosjekt, et rule project. I min implementering heter det BusinessRules1. Se figur 3.2. Når man legger til et nytt prosjekt, må man referere til biblioteket for å få tak i informasjonen som ligger lagret der. Da høyreklikker man på "references" under BusinessRules1 slik som dette:



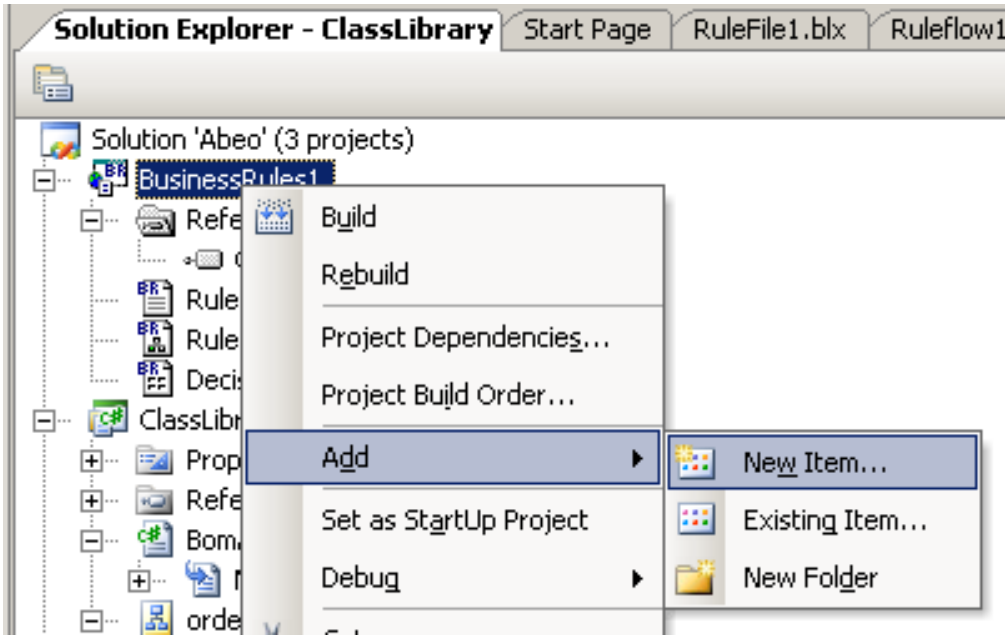
Figur 3.6: Legge til en referanse

...og får opp følgende vindu:



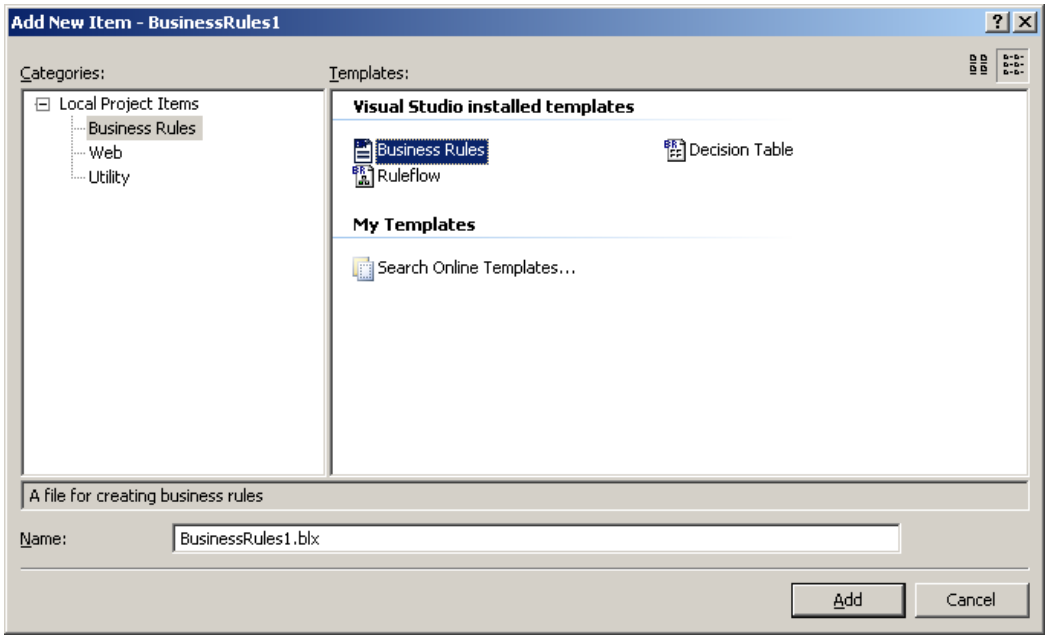
Figur 3.7: Oversikt over hva man kan referere til

Da velger man fanen 'Projects' i figur 3.8 og deretter ClassLibrary. Neste steg på veien er å opprette et nytt item av type Business Rule, jeg har valgt å kalle det RuleFile1.blx.



Figur 3.8: Legge til enheter

Da kommer man til dette vinduet:



Figur 3.9: Oversikt over hva som kan legges til BusinessRules1

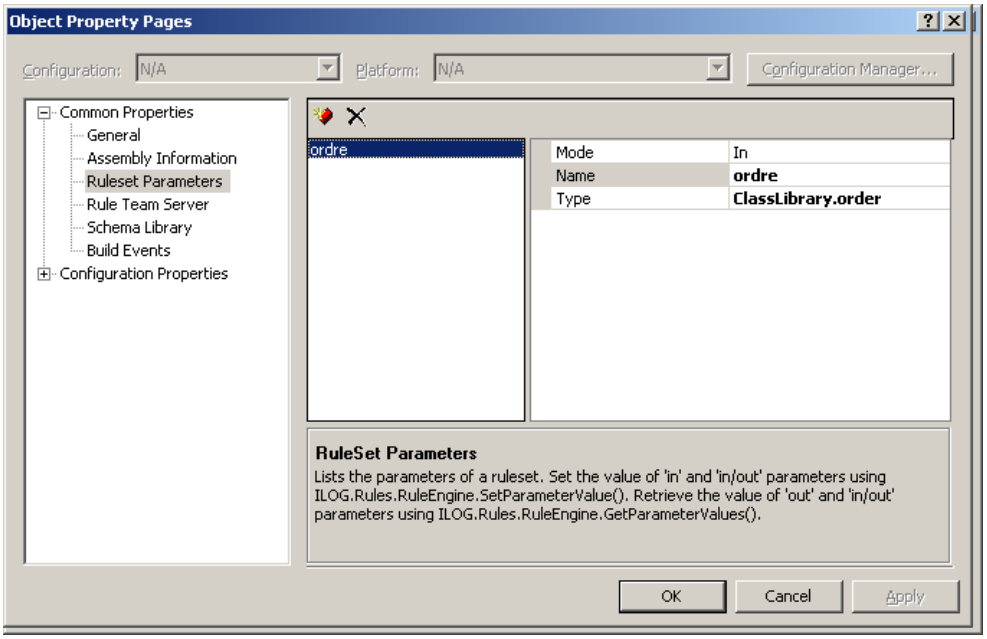
Her har man valget mellom å legge til en regelfil, en beslutningstabell eller en regelflyt. I første omgang nøyer jeg meg med å legge inn regelfilen. Så er det bare å sette i gang med å legge inn reglene.

I tillegg til å implementere forretningslogikken, må man også legge inn applikasjonslogikken. Det er den delen av programmet som utfører oppgaver som behandler interaksjon med brukeren og en del av XML'en. Applikasjonslogikken inkluderer det som innkapsler regelmotoren og kjører reglene. Da legger man til enda et prosjekt, av type Console Application. Denne har jeg valgt å kalle ConsoleApplication1. Her må det også legges til referanser. I tillegg til å legge til referanse til biblioteket, må man legge til en referanse til forretningsreglene, BusinessRules1. Det må også refereres til .NET komponentene ILOG.Rules og ILOG.Rules.RuleEngine. Disse ligger under .NET-fanen i figur 3.7.

Neste steg er å håndtere XML-delen av implementeringen. Først skrev jeg en helt enkel XML-fil som er et eksempel på en XML-fil som kan komme inn til systemet. Deretter måtte jeg fjerne hele Class1.cs fra klassebiblioteket, dette er en klasse som blir autogenerert ved oppretting av biblioteket. Den fjernede klassen erstattet jeg med en klasse jeg kalte order.xsd. Deretter gikk jeg tilbake til løsningsveiviseren og trykket på egenskapene til order.xsd. Så kommer den viktige egenskapen til ILOG som gjør det velegnet for bruk av XML: I egenskapsoversikten er det noe som heter 'Custom Tool', der skrev jeg inn 'XmlSchemaCodeGenerator'. Tilbake i løsningsveiviseren må man høyreklikke å order.xsd og velge "Run Custom Tool", dermed blir det generert en binding mellom XML-skjemaet mitt og regelmotoren.

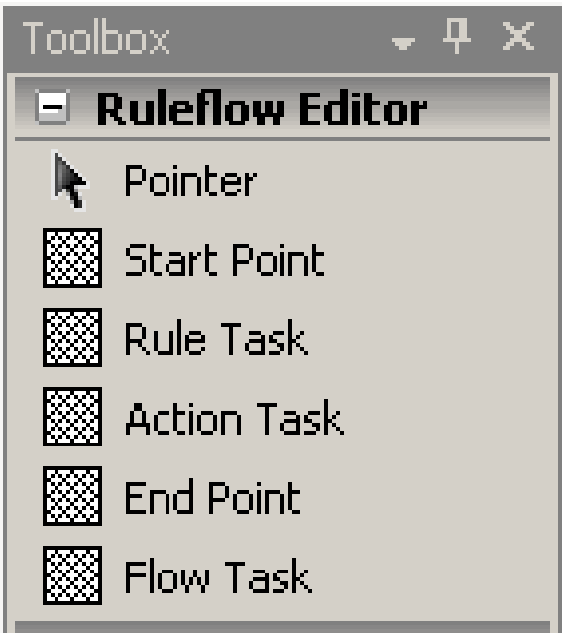
Nå som XML-delen av implementeringen er fullført, gjenstår det å skrive Program.cs. Her leses ordren som kommer inn til systemet fra kunden på XML-format, ordren bindes til en variabel av type order. Programmet mitt skriver så ut informasjonen som ligger i ordren. Deretter opprettes regelmotor, regelsettet lastes og settes før reglene kjøres.

Skriving av forretningsregler var forholdsvis rett fram, men først måtte jeg sette et regelsettparameter på følgende måte: høyreklikke på BusinessRules1 og fikk opp følgende vindu:



Figur 3.10: Ruleset parameter

Jeg opprettet et nytt parameter som jeg kalte ordre, av type ClassLibrary.order. Dermed fikk jeg opprettet variabelen ordre som jeg brukte til å skrive forretningsreglene. Etter fullføringen av reglene, opprettet jeg en ny ruleflow, kalt ruleflow1, for å styre flyten av reglene. En regelflyt opprettes på samme måte som en regelfil.



Figur 3.11: Ruleflow editor

Denne verktøyboksen lar deg opprette elementer som brukes i regelflyten. Man trykker på et av verktøyene og drar de over i editoren. Man starter med å dra over et 'Start Point' og et 'End Point'. Deretter kan man legge inn hva man ønsker av Rule Task og Action Tasks. Etter at man har lagt til en Rule Task, dobbelklikker man på den og får opp en oversikt over hvilke regler som finnes i BusinessRules1. Da kan man velge mellom en eller flere regler som skal bli kjørt når programmet når den aktuelle regelflyten. Man kan også legge til Action Tasks. I en Action Task kan man velge hvilke handlinger som skal utføres når regelflyten når den aktuelle Action Task.

For å bestemme flyten i programmet, kobler man sammen startpunktet, regeloppgaver, handlingsoppgaver og endepunktet. Det gjøres ved å holde inn shift mens man trykker på de to punktene som skal kobles sammen. I tillegg til å koble sammen flere objekter, kan man sette betingelser på koblingene. Trykker man på en kobling, får man opp en betingelse som kan settes for at denne flyten skal velges. Skal dette gjøres, må det alltid gå en kobling ut fra objektet som ikke har en betingelse knyttet til seg, en såkalt else. Noe jeg merket meg da jeg skulle kompilere programmet, var at når jeg satte betingelser på koblingene, fikk jeg alltid kompileringsfeil. Da viste det seg at for at programmet skal kunne kompilere og kjøres, med betingelser på koblingene, måtte følgende setning være med i program.cs:

Engine.Ruleflow = engine.RuleSet.Ruleflows[“BusinessRules1”, “Ruleflow1”];
Når dette var gjort, var det mulig å styre regelflyten uten å få kompileringsfeil.

Et nyttig lite verktøy, av estetiske årsaker, som finnes i regelflyteditoren, er ’Perform Layout’. Dette får man opp ved å høyreklikke i editoren, dette verktøyet flytter om på alle objektene i regelflyten og plasserer de oversiktlig i forhold til hverandre.

Selve koden ligger vedlagt i Appendix.

3.3 Resultat

3.3.1 Ekspertsystem

Den samlede ytelsen til et regelbasert ekspertsystem avhenger av tre faktorer: antall regler som faktisk kjøres, antall forretningsobjekter som reglene må evaluere og hvilken eksekveringsmodus som er valgt. Noe som har blitt tydelig, er at den relative ytelsen til en standard retemodus forbedres etter hvert som flere regler legges til regelsettet. Slik er det fordi dette gjerne medfører at prosenten av regler som må kjøres går ned. Den relative ytelsen til en sekvensiell modus forbedres etter hvert som flere objekter legges til, dette fordi en oppretting av et retenettverk og håndteringsaktiviteter assosiert med Retes inferens blir mer komplekst og tidskonsumerende. I tillegg vil en økning av antall objekter føre til at kostnaden av sporing av endringer i objektene og egenskapsverdier som kan trigge kjøring av enda flere regler, øke signifikant. I den sekvensielle modusen vil reglene kun bli eksekvert i forhold til rekkefølgen de har i regelsettet, eventuelt basert på en eksplisitt prioritet/rekkefølge.

Samtaler jeg har hatt med en bruker som har vært med på å bruke regelmotor i store offentlige prosjekter, har støttet opp under det jeg har funnet ut, nettopp at bruk av en regelmotor kan være svært nyttig for visse typer bedrifter og organisasjoner. Han viste til et prosjekt i det offentlige som har pågått i mange år uten at produktet har blitt ferdig utviklet. Grunner til dette var i hovedsak at data og krav endret seg til stadighet, før man fikk implementert disse nye endringene ble stadig nye endringer ønsket. Dermed kom man ingen vei og kostnadene ved utvikling økte uten at man kom noe nærmere et fungerende produkt. Til slutt innså man at man ikke kom noen vei og tenkte derfor i helt nye baner. Et regelbasert ekspertsystem ble så innført, på relativt kort tid, med stor suksess, både økonomisk og kvalitetsmessig.

Jeg har fått bekreftet at en regelmotor er svært velegnet til å validere en ordre som kommer inn til systemet vårt. Dette var et sentralt spørsmål da oppgaven ble gitt, det var ønskelig å se på hvor egnet en regelmotor er til å validere data. Bruk av regelflyt for å segmentere og kontrollere eksekveringen av regler viser seg som en svært god teknikk for å skape mer vedlikeholdbare prosjekter og i tillegg øke ytelsen.

Ekspertsystem vil i mange tilfeller være mer egnet enn konvensjonelle system når informasjonen man har ikke er fullstendig. I slike tilfeller vil ikke de konvensjonelle systemene kunne komme fram til noe svar/løsning, mens ekspertsystem vil kunne komme fram til i alle fall en delvis løsning. Slik heuristikk

er en nyttig egenskap ved ekspertsystem som bør utnyttes, dermed kan man oppnå svar og resultater selv om de ikke nødvendigvis er helt fulkomne.

Ekspertsystem er særs egnet i tilfeller hvor informasjonen ofte blir endret, gjerne av eksterne parter. Skulle man i slike tilfeller benyttet konvensjonelle system, ville man vært nødt til å gå inn i koden og skiftet ut de aktuelle parametrene manuelt. En slik inngripen i koden hadde medført omfattende testing for å forsikre seg om at programmet fortsatt fungerer som det skal. Når det er snakk om 100 + parameter som kan endre seg, og gjerne forholdsvis ofte, ser man nytten av å benytte en regelmotor. Man kan hurtigere tilpasse seg markedets svingninger og tilby det kunden ønsker.

En stor fordel med bruk av regelmotor er at det sparer it-avdelingen for mye arbeid. I et konvensjonelt system er det it-avdelingens ansvar å legge inn informasjon og data som brukerne selv kan legge inn i regelbaserte system. Dette er tid som utviklerne istedenfor kunne brukt til å effektivisere og optimalisere algoritmer og andre deler av systemet for å forbedre ytelsen.

3.3.2 ILOG Rules for .NET

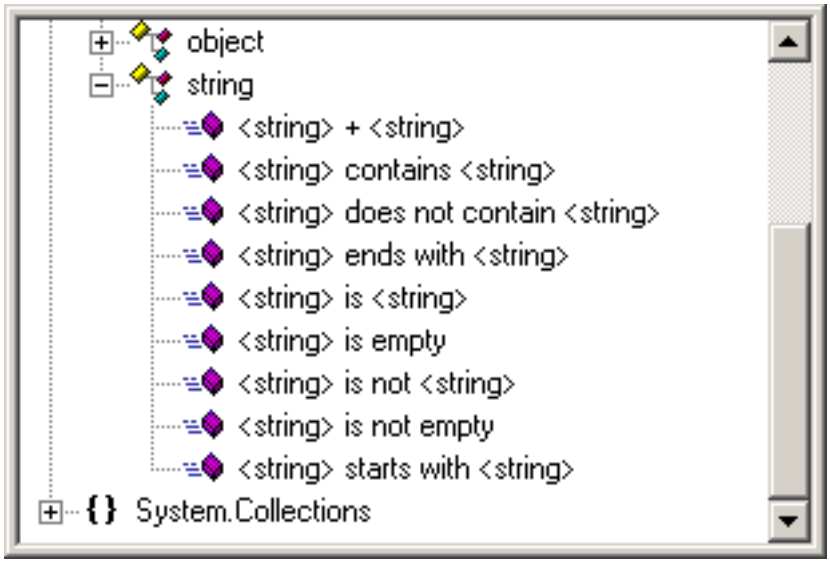
Inntrykket jeg sitter igjen med etter å ha prøvd ut ILOG Rules for .NET er stort sett positivt. Det var veldig greit å skrive reglene når man først fikk satt opp rammeverket. Grensesnittet for å skrive regler er brukervennlig, og jeg vil tro at forretningsbrukerne ikke vil ha noe problem med å legge inn eller endre regler etter hvert som det blir behov for det, selv om de ikke har noe kjennskap til programmering eller andre lignende konsepter.

For å få installert programvaren, var det en forutsetning at man har Windows Server 2003 samt Visual Studio. Jeg hadde verken Visual Studio eller Windows Server 2003 på min maskin da jeg startet på denne oppgaven, og lastet derfor ned en gratis prøveversjon av VS for å bruke under denne evalueringen. Når jeg så skulle installere Rules for .NET, medførte dette uforståelige feilmeldinger. Jeg hadde tett kontakt med ILOGs kundeservice, men de forsto heller ikke hvorfor programmet feilet. Det var først da jeg fikk tak i den profesjonelle utgaven av VS at programmet lot seg installere. Dette er noe som burde stått bedre forklart under hva som kreves for å installere ILOG. Det kan nevnes at ILOG ellers kommer med greie spesifiseringer av hva som kreves for å installere de ulike komponenter av Rules.

Jeg er heller ikke helt fornøyd med ILOGs brukerveiledninger når det gjelder å koble regelmotoren opp mot XML-skjema. Det står lite informasjon om dette på nett eller i brukerveiledninger, noe som medførte unødvendig mye

prøving og feiling for å få det til å fungere slik det skulle. Når man først har fått koblingene til å fungere, egner XML seg meget godt til å kombinere med ILOG.

Noe annet som jeg synes legger en begrensing på implementeringen ved bruk av regelmotor, er testene som gjøres under valideringen.



Figur 3.12: Sammenligningsmuligheter for strenger

Ønsker man for eksempel å teste strenger, har man kun disse valgene man ser i figur 3.12. I et konvensjonelt system har man større frihet når man utfører testene sine, når det gjelder strenger, tall og objekter.

Kompatibilitet med andre produkter

ILOG er en gullsertifisert partner av Microsoft og det første håndteringssystemet for forretningsregler som fullt ut utnytter Microsofts .NET rammeverk. Det er også først ute med å integrere Microsoft Office, Visual Studio, Windows SharePoint og Microsoft BizTalk Server. Som det går ganske tydelig fram her, er det Microsoft som er partneren ILOG Rules for .NET retter seg mot. Det er derfor et godt valg å benytte ILOG dersom man benytter seg av Microsofts teknologi i bedriften.

Rete og Sekvensiell utføring

Ved bruk av ILOG Rules for .NET er det kun ved bruk av RuleFlow man kan velge om eksekveringen skal foregå sekvensielt eller ved Rete. Jeg gikk inn i filen RuleFlow1 og endret manuelt alle ruletasks til å bruke Rete utføring. Deretter testet jeg hvor lang tid systemet brukte på eksekveringen første gang samt påfølgende eksekveringer. Jeg endret så alle ruletasks til sekvensiell utføring og testet de samme responstider som ved Rete. Denne testingen utførte jeg flere ganger for å få et mest mulig realistisk resultat.

Gjennomsnittlig responstid første kjøring Rete : 19,5s
Gjennomsnittlig responstid første kjøring sekv : 24,9s

Gjennomsnittlig responstid etterfølgende kjøringer Rete : 14,7s
Gjennomsnittlig responstid etterfølgende kjøringer sekv : 17,3s

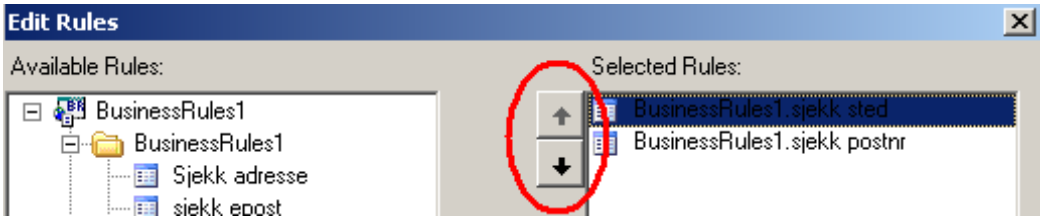
Dette viser tydelig at ved implementeringen min av et ordresystem, er det bruk av Rete som gir en mest effektiv eksekvering. Den brukte gjennomsnittlig 5,4 sekunder mindre på første gjennomkjøring. Det er 20 % forbedring i forhold til ved sekvensiell kjøring. Ved påfølgende gjennomføringer, altså etter at første gjennomkjøring er fullført, brukte Rete 2,6 sekunder kortere tid. Det er 15 % bedre enn sekvensiell kjøring.

Konfliktløsning

Jeg ønsket å teste hvordan ILOG håndterte konfliktløsning, som nevnt under kapittel 1.4.2. ILOG har ikke imponert når det gjelder kvalitet på skriftlig materiale om sine produkter. I dokumenter jeg fant på nett, var ingenting nevnt om hvordan ILOG Rules for .NET behandler dette området. Det første jeg prøvde, var å legge inn to regler som hadde samme betingelse men to forskjellige handlinger. Disse to reglene la jeg inn i samme ruletask i Ruleflow1. Dette resulterte, ved kjøring, i at den første regelens handling ble overskrevet av den andre. Ved videre testing la jeg til en prioritet, det lot seg gjøre ved å skrive inn ønsket verdi i prioritetsfeltet, se figur 2.5. Den ene av de to reglene i konflikten fikk prioritet 1 mens den andre fortsatt hadde standard prioritet 0. Dette ga ingen utslag ved kjøring av systemet. Resultatet forble det samme som i den første testen.

Videre testing gjorde så utslag. Når jeg gikk inn i den aktuelle ruletask i Ruleflow1, endret jeg manuelt rekkefølgen på reglene. Dette lot seg gjøre ved

å markere en regel og deretter trykke på en pil for å prioritere regelen enten opp eller ned. Se figur 3.13



Figur 3.13: Endre prioritet for regler

Dette ga utslag i den form av at jeg fikk muligheten til å bestemme hvilken av disse to reglene som skal kjøres først. Dermed kontrollerte jeg i og for seg hvilken regels handling som skulle bli stående som konklusjon. Ytterligere testing resulterte i at jeg fikk bruk for prioritetsegenskapen i 2.5. For å få brukt denne, måtte jeg kommentere ut hele filen som styrte regelflyten, RuleFlow1. Da ble prioriteten brukt til å bestemme rekkefølgen reglene skulle kjøres i, på samme måte og med samme resultat som ved bruk av prioritetsegenskapen i figur 3.13

I Rules for .NET finnes det ingen måte å stoppe regelmotoren på når et objekt har blitt bundet. Den eneste muligheten til å implementere denne mekanismen på, er ved å ha kun de regler som kan komme i et konfliktsett i en egen rule task i regelflyten. I rule task må man da sette "Exit Criteria" til å være Rule Instance. På denne måten vil kun en regel bli eksekvert inne i denne rule task.

Jeg fant ingen informasjon om mulighet for å implementere strategien hvor man kjører den regelen som har flest treff i sin betingelse. I min korrespondanse med ILOGs kundeservice spurte jeg om dette, de svarte da at dette var et konsept de ikke engang har hørt om.

3.4 Evaluering av resultat

Innføring av en regelmotor kan, med vekt på kan, være en god investering for en bedrift. Den bør imidlertid tenke seg godt om før den går til denne investeringen. Dersom du har en situasjon som kan beskrives ved hjelp av kun få regler behøver du ikke å gå igjennom anstrengelsen og kostnadene med å anskaffe en regelmotor. Har du derimot en betydelig mengde med forretningslogikk som må implementeres, kan det være vel verdt å undersøke mulighetene for å gjøre det motsatte.

Kombinasjonen av at et ekspertsystem både kan forklare sin egen tankegang og at det håndterer usikkerhet, medfører at regelbaserte system kan tilby flere konklusjoner, rangert etter konfidens. Ulempen med disse to egenskapene er at de øker kostnadene ved anskaffelse av et system betraktelig. Det kan da også bli mer komplisert å skrive reglene. Innebygde forklarende fasiliteter er nyttige når man skal utføre en feilsøking, men dette er sjelden nødvendig for de spørringer brukerne utfører. For å få nyttige forklaring av systemets tankegang må brukene som oftest gjøre det selv for hånd.

Med tanke på at de konvensjonelle systemene alltid følger en bestemt stegvis rekkefølge under beregningene og at feil ikke oppnås, kan man spekulere i om konvensjonelle systemer er bedre enn ekspertsystemene, med tanke på at de produserer kun korrekte svar. Dersom man til enhver tid har den nødvendige fullstendig korrekt informasjon vil svaret være ja. Dette er derimot ofte ikke tilfellet, og i slike tilfeller blir det da i de konvensjonelle systemene produsert gale svar eller ikke noe svar i det hele tatt. Ekspertsystemene tar høyde for slike ukomplette data og kan, selv med svært ufullstendige data, komme fram til en noenlunde fornuftlig løsning ved bruk av heuristikk. Ulempen med dette er alt vi mennesker ser på som innlysende, må inkorporeres i systemet. Systemet har heller ingen mulighet til å lære av erfaring.

Rete eller sekvensiell?

Testene under kapittel 3.3.2 viste at Rete kjørte relativt mye raskere enn den sekvensielle utføringen. Her må det nevnes at det er en mulighet for at disse resultatene er delvis misvisende da tidtakingen var noe unøyaktig. Det ble brukt en stoppeklokke til å måle tiden fra start til slutt i eksekveringen. Den manuelle tidtakingen blir aldri helt nøyaktig, men det ble ingen store avvik under målingene. Alle målingene til første gjennomkjøring ved Rete var lavere enn de tilhørende sekvensielle, det samme gjelder de påfølgende gjennomkjøringene. Medianen til gjennomsnittlig kjøring ved Rete var 13,

mens den var 17 hos den sekvensielle. Dette tyder på, med høyde for delvis unøyaktighet i tidtakingen, at Rete eksekverte raskere enn den sekvensielle. Mitt testresultat strider imot det meste jeg har lest, de fleste artikler og bøker hevder at den sekvensielle utføringen ofte har en raskere eksekvering enn Rete. Men som tidligere nevnt, dette avhenger av systemet.

Det er vanskelig å fastslå om Rete eller sekvensiell utføring vil føre til best resultater, det er fordeler og ulemper med begge to. I tillegg kommer mye an på systemet som skal implementeres. Ved en sekvensiell utføring legges en større vekt på den som skriver reglene, han må sørge for at hvilken som helst regel som påvirker andre regler blir utført i korrekt rekkefølge. Utbyttet er ofte større hastighet ved eksekveringen.

Ytelse ved bruk av Rete er ofte meget god. Likevel, dersom man har en situasjon med et høyt volum av situasjoner av batch-jobber [1], er ofte kravet til gjennomstrømning svært høyt, og det behøver ikke være behov for retebasert inferens [6]. I de fleste store batch-jobber, er det ganske tydelig om en gitt beslutningstakingsprosess eller steg skal utføres i sekvensiell modus eller i standard Rete. Dersom hver regel utfører en diskret evaluering og gjør en diskret beslutning, har du en god kandidat for en sekvensiell utføring.

Det er en klar avveining mellom sekvensiell eller Retemodus. Dersom du har 500 regler, og i gjennomsnitt 50 vil kjøres, vil kostnaden av å kjøre 450 regler unødvendig, noe som kan være tilfellet ved en sekvensiell utføring, være ganske nær kostnaden av å opprette et retenettverk og spore endringene slik at du kun behøver å kjøre de 50 reglene som må kjøres [6]

Konfliktløsningen

Det kan betviles om den måten ILOG Rules for .NET håndterer konflikter på kan kalles konfliktløsning. Som mine tester, under kapittel 3.3.2 Konfliktløsning viste, var det to måter å løse eventuelle konflikter på å. Den ene gikk ut på å bestemme hvem av reglene som skulle kjøres først, mens den andre brukte en rule task for å la kun en av reglene kjøre. Jeg vil ikke kalle dette ekte konfliktløsning, ettersom de bare dekker over problemet. I det første tilfellet, kan den første regelens handling bli overskrevet av den andres handling. Det er derfor mer en regelflyt enn en konfliktløsning.

I det andre tilfellet, så blir resultatet avhengig av hvilken regel man legger først inn i rule task. Dette er en løsning jeg ikke vil anbefale, det er ofte veldig

[1] En batch-jobb er en serie av eksekveringer som blir utført etter hverandre, uten menneskelig interaksjon

mange regler i et system, og hver regel kan være med i flere konfliktsett. Skal man legge inn alle konfliktsettene i en egen rule task i regelflyten, blir dette alt for tidkrevende i tillegg til at det blir uoversiktlig.

Validering av data

Det er diskuterbart om en validering hører hjemme under et håndteringssystem for forretningsregler. Bør ordren valideres før den kommer inn til regelmotoren? Et argument som peker i retning av dette er at man har flere valg når det gjelder parametrene man ønsker å validere. Ved bruk av regelmotoren har man et begrenset antall muligheter, i forhold til for eksempel konvensjonell bruk av Java, for å sammenligne og vurdere data. Se figur 3.12. Dette kan medføre at man behøver flere regler for å uttrykke en betingelse som kunne vært skrevet som en enkelt linje i et konvensjonelt datasystem. I tillegg kan det være vanskelig å uttrykke sekvenser og kompliserte løkker.

Result fra Forrester

Resultatene fra undersøkelsene til Forrester Research, som ble presentert i kapittel 2.4, er delvis misvisende. Jeg har prøvd å finne ut hvilken versjon av Rules for .NET de brukte i evalueringen, men det eneste jeg fant var at de skrev “first release“. Det kan tyde på at de brukte versjon 1, altså en mye tidligere utgave enn versjon 3 som nå finnes på markedet. En tidlig versjon av et program har ofte mer begrenset utvalg av verktøy og tjenester, slik at dersom denne testen hadde blitt utført i dag hadde nok resultatene vært en del annerledes.

Kapittel 4

Konklusjon

4.1 Konklusjon

Et regelbasert ekspertsystem er bygd opp av en kunnskapsbase som inneholder reglene, en regelmotor, en database som inneholder fakta og et brukergrensesnitt. I tillegg finnes det sett av fasiliteter som forklarer hvordan konklusjoner ble nådd. Et slikt system er særs velegnet for bedrifter som har policy som defineres av eksterne enheter, bruker mye data som endrer seg over tid eller har objekter som kan ha ulike tilstander.

Utbyttet man kan ha av å bruke et regelbasert ekspertsystem er mange, den viktigste er at bedriften raskere klarer å få utført endringer og dermed hurtigere tilpasse seg et marked i stadig endring. Utgifter til vedlikehold av systemet går ned og it-avdelingen slipper å bruke ressurser på å utføre endringer som forretningsbrukerne nå selv kan endre.

Aktøren ILOG Rules for .NET er en forholdsvis ny aktør på markedet, et godt valg for en bedrift som baserer seg på Microsofts teknologi. Integrerer blant annet Microsoft Office, Visual Studio, Windows SharePoint og BizTalk Server.

Rules for .NET er velegnet til å innføre valgt case, validering av ordre i XML-format. Regelmotoren lar deg bestemme både rekkefølge av regler som skal kjøres og ulike valg ved kjøring, men den mangler noen verktøy for å håndtere eventuelle konflikter mellom reglene.

4.2 Videre arbeid

Det er flere aspekt som kunne være interessant å se på ved et videre arbeid med denne oppgaven. Et ville være og utvidet implementeringen til å omfatte brukeren som plasserer ordren. Dette ville innebære å lage et grafisk brukergrensesnitt hvor brukeren legger inn sin ordre, generering av xml fra grensesnittet og deretter oversende xml fra brukeren og inn til systemet. Dette ville gjort eksemplet mer helhetlig og virkelighetsnært. En videreføring av oppgaven kunne også inkludere bruk av Rule Solutions for Office og Rule Team Server for SharePoint.

En annen ting som ville være interessant, er å legge inn forholdsvis mange regler, 100+. Noen av disse reglene kunne omhandlet en dypere validering av kundedataene. Deretter utføre en omfattende testing av responstiden til systemet når man bruker Rete i forhold til en sekvensiell utføring.

Noe jeg også synes kunne være interessant, er å evaluere Blaze Advisor, en av de største aktørene på markedet innen regelmotorer. Da ville jeg gått fram på samme måte som jeg gjorde med ILOG i denne oppgaven og implementert det samme caset. Deretter ville jeg sammenlignet disse to aktørene med hensyn til brukervennlighet, kvalitet på tilgjengelige verktøy, kompatibilitet med andre programmer og se på hvem av de som lar meg implementere caset på en best og enklest mulig måte.

Bibliografi

[1] Martin Ader. Ilog components, for business process management solutions. \http://www.wngs.com/downloads/ilogbml.pdf.

[2] Peter Abrahams for Bloor Research. Jrules 4.6 from ilog. 2004.

[3] Ian Graham. Service oriented business rules management systems. \http://www.giaever.com/sosiologi/KM.htm.

[4] Frederick Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28(9), 1985.

[5] ILOG. Ilog rules for .net for architects and developers. \http://www.ilog.com/products/rulesnet/whitepapers/index.cfm.

[6] Fair Isaac. High-volume batch processing with blaze advisor. \http://www.fairisaac.com/NR/rdonlyres/9489A6B1-A829-4D1A-ACCE-F137DA67266B/0/HighVolumeBatchProcessingwithBlazeAug2004WP.pdf.

[7] N.E. Lane. Global issues in evaluation of expert systems. *IEEE Computer Society Press*, 1986.

[8] Elaine Kant Lee Brownston, Robert Farrell and Nancy Martin. *Programming Expert Systems In OPS5.* Addison-Wesley, 1985.

[9] Tony Morgan. *Business Rules and Information Systems.* Addison-Wesley, 2002.

[10] Michael Negnevitsky. *Artificial Intelligence: A Guide to Intelligent Systems.* Addison-Wesley, 2004.

[11] Allen Newell and Herbert Simon. *Human Problem Solving.* Prentice-Hall, 1972.

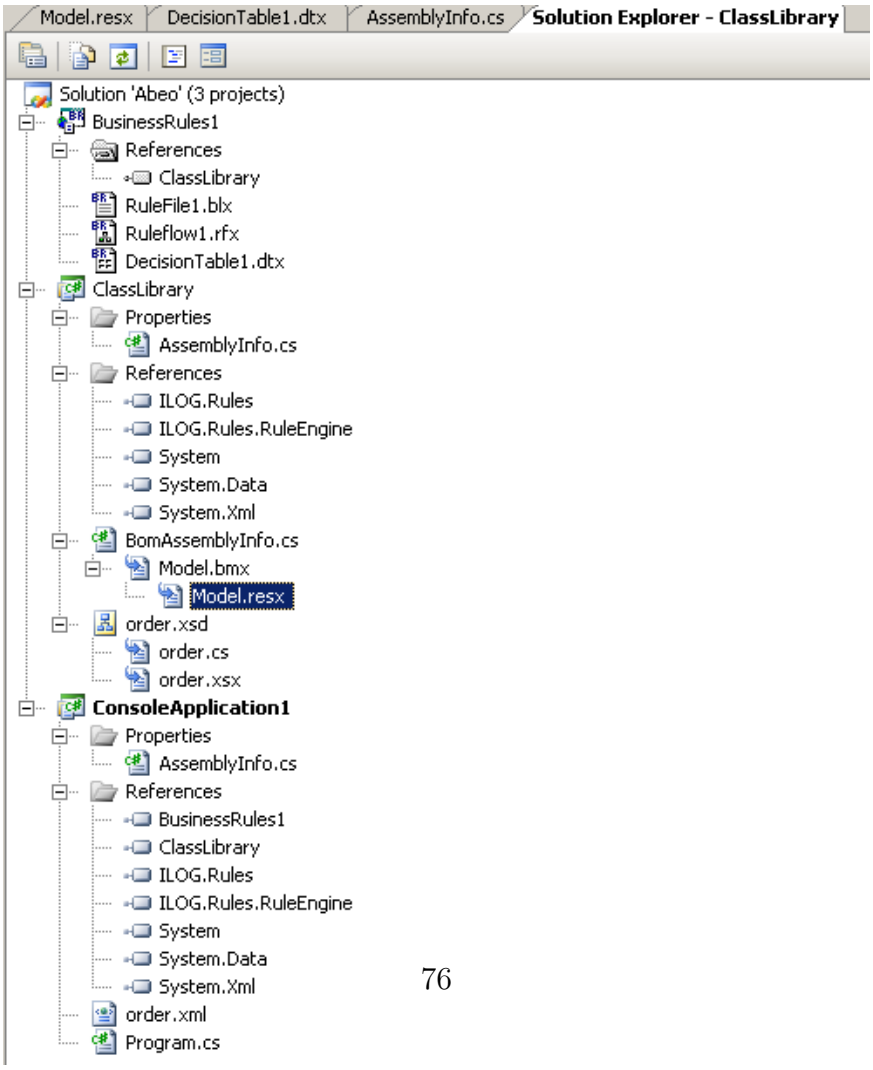
[12] John Rymer. Business rules platforms, q1 2006. *Forester Research*, 2006.

[13] Suzanne Smith and Abraham Kandel. *Verification and Validation of Rule-Based Expert Systems*. CRC Press, 1993.

Tillegg A

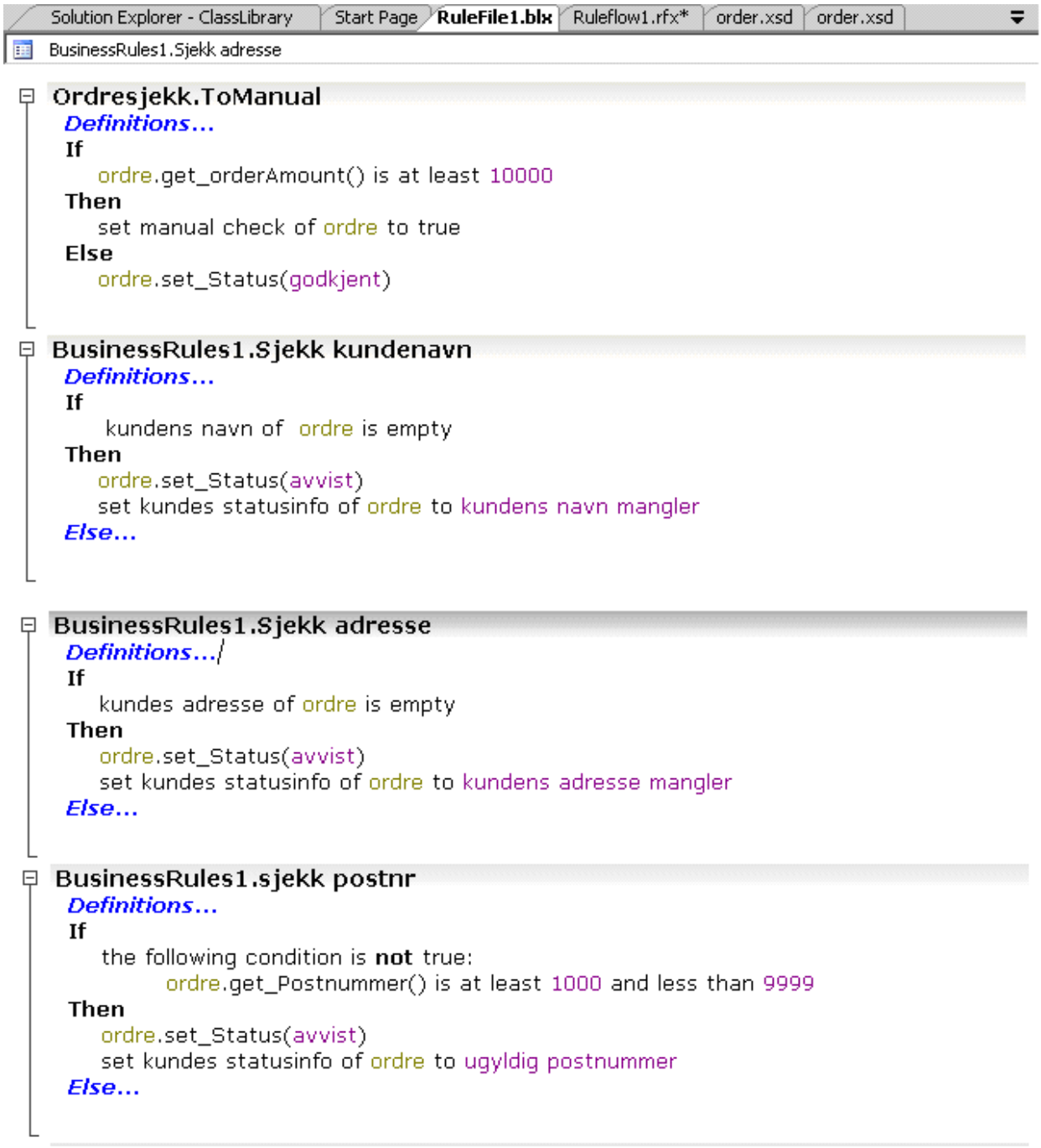
Appendix

A.1 Solution Explorer



Figur A.1: SolutionExplorer

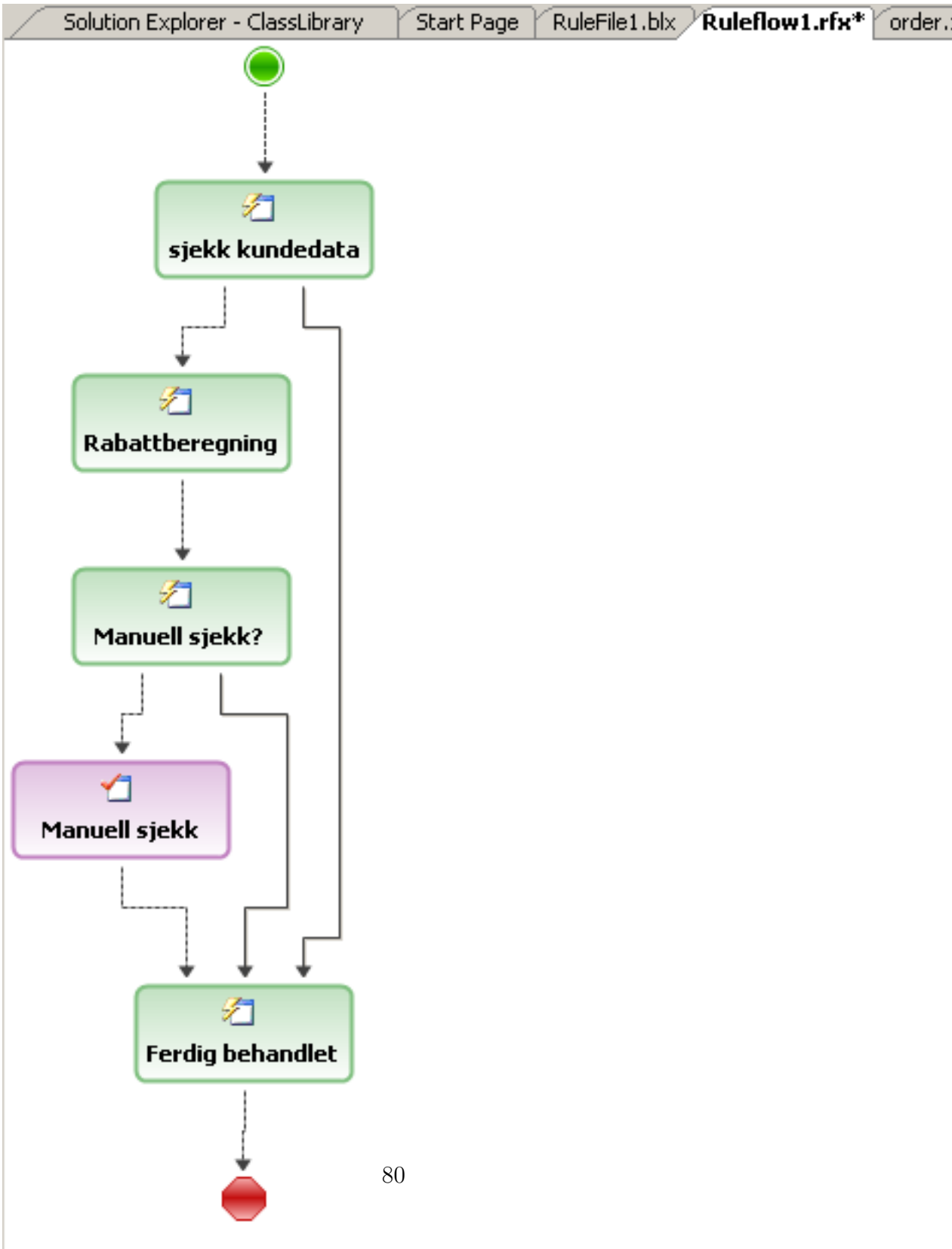
A.2 RuleFile1.blx



Figur A.2: RuleFile1 del1

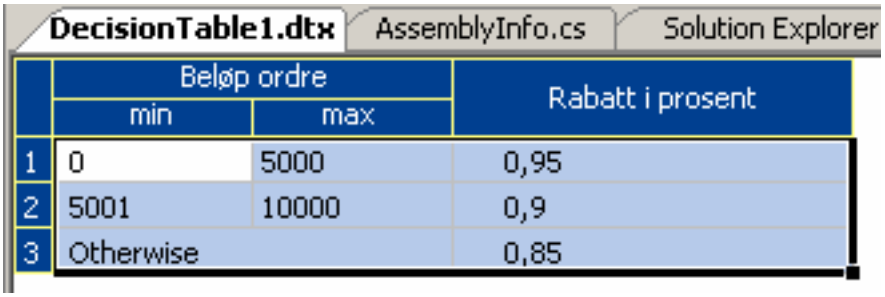
Figur A.3: RuleFile1 del2

A.3 RuleFlow1.rfx



Figur A.4: ruleFlow1

A.4 DecisionTable1.dtx



	Beløp ordre		Rabatt i prosent
	min	max	
1	0	5000	0,95
2	5001	10000	0,9
3	Otherwise		0,85

Figur A.5: DecisionTable1

A.5 order.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns="http://www.ilog.com/rules"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.ilog.com/rules">
    <xs:element name="order">
        <xs:complexType>
            <xs:sequence>
              <xs:element name="orderId" type="xs:int"/>
              <xs:element name="orderDate" type="xs:date"/>
              <xs:element name="statusinfo" type="xs:string"/>
              <xs:element name="kundenavn" type="xs:string" />
              <xs:element name="Adresse" type="xs:string"/>
              <xs:element name="Postnummer" type="xs:int"/>
              <xs:element name="Sted" type="xs:string"/>
              <xs:element name="Telefon" type="xs:int"/>
              <xs:element name="epost" type="xs:string"/>
              <xs:element name="itemnumber1" type="xs:int"/>
              <xs:element name="items1" type="xs:int"/>
              <xs:element name="pricepritem1" type="xs:int"/>
              <xs:element name="priceitem1" type="xs:int"/>
              <xs:element name="itemnumber2" type="xs:int"/>
              <xs:element name="items2" type="xs:int"/>
              <xs:element name="pricepritem2" type="xs:int"/>
              <xs:element name="priceitem2" type="xs:int"/>
              <xs:element name="itemnumber3" type="xs:int"/>
```

```
            <xs:element name="items3" type="xs:int"/>
            <xs:element name="pricepritem3" type="xs:int"/>
            <xs:element name="priceitem3" type="xs:int"/>
            <xs:element name="itemnumber4" type="xs:int"/>
            <xs:element name="items4" type="xs:int"/>
            <xs:element name="pricepritem4" type="xs:int"/>
            <xs:element name="priceitem4" type="xs:int"/>
            <xs:element name="Status" type="xs:string" />
            <xs:element name="orderAmount" type="xs:double"/>
            <xs:element name="ManualCheckRequired" type="xs:boolean"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
</xs:schema>
```

A.6 order.cs

```
//------------------------------------------------------------------------------
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:2.0.50727.42
//
//     Changes to this file may cause incorrect behavior and will be los
//     the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------

namespace ClassLibrary {


    /// <remarks/>
    [System.CodeDom.Compiler.GeneratedCodeAttribute("System.Xml", "2.0.5
    [System.SerializableAttribute()]
    [System.Diagnostics.DebuggerStepThroughAttribute()]
    [System.ComponentModel.DesignerCategoryAttribute("code")]
    [System.Xml.Serialization.XmlTypeAttribute(AnonymousType=true)]
    [System.Xml.Serialization.XmlRootAttribute(Namespace="http://www.ilo
    com/rules",
IsNullable=false)]
    public partial class order {

        private int orderIdField;
```

```
        private System.DateTime orderDateField;

        private string statusinfoField;

        private string kundenavnField;

        private string adresseField;

        private int postnummerField;

        private string stedField;

        private int telefonField;

        private string epostField;

        private int itemnumber1Field;

        private int items1Field;
         private int pricepritem1Field;

        private int priceitem1Field;

        private int itemnumber2Field;

        private int items2Field;

        private int pricepritem2Field;

        private int priceitem2Field;

        private int itemnumber3Field;

        private int items3Field;

        private int pricepritem3Field;

        private int priceitem3Field;

        private int itemnumber4Field;

        private int items4Field;
```

```
        private int pricepritem4Field;

        private int priceitem4Field;

        private string statusField;

        private double orderAmountField;

        private bool manualCheckRequiredField;

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int orderId {
 get {
                return this.orderIdField;
            }
            set {
                this.orderIdField = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified, DataType="date")]
        public System.DateTime orderDate {
            get {
                return this.orderDateField;
            }
            set {
                this.orderDateField = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public string statusinfo {
            get {
                return this.statusinfoField;
```

```
            }
            set {
                this.statusinfoField = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public string kundenavn {
            get {
                return this.kundenavnField;
            }
            set {
                this.kundenavnField = value;
            }
        }

        /// <remarks/>
[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public string Adresse {
            get {
                return this.adresseField;
            }
            set {
                this.adresseField = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int Postnummer {
            get {
                return this.postnummerField;
            }
            set {
                this.postnummerField = value;
            }
        }
```

```
        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public string Sted {
            get {
                return this.stedField;
            }
            set {
                this.stedField = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int Telefon {
            get {
                return this.telefonField;
            }
            set {
                this.telefonField = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public string epost {
get {
                return this.epostField;
            }
            set {
                this.epostField = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
```

```
XmlSchemaForm.Unqualified)]
        public int itemnumber1 {
            get {
                return this.itemnumber1Field;
            }
            set {
                this.itemnumber1Field = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int items1 {
            get {
                return this.items1Field;
            }
            set {
                this.items1Field = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int pricepritem1 {
            get {
                return this.pricepritem1Field;
            }
            set {
                this.pricepritem1Field = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
public int priceitem1 {
            get {
                return this.priceitem1Field;
```

```
            }
            set {
                this.priceitem1Field = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int itemnumber2 {
            get {
                return this.itemnumber2Field;
            }
            set {
                this.itemnumber2Field = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int items2 {
            get {
                return this.items2Field;
            }
            set {
                this.items2Field = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int pricepritem2 {
            get {
                return this.pricepritem2Field;
            }
            set {
                this.pricepritem2Field = value;
            }
```

```
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int priceitem2 {
            get {
                return this.priceitem2Field;
            }
            set {
                this.priceitem2Field = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int itemnumber3 {
            get {
                return this.itemnumber3Field;
            }
            set {
                this.itemnumber3Field = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int items3 {
            get {
                return this.items3Field;
            }
            set {
                this.items3Field = value;
            }
        }

        /// <remarks/>
```

```
[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int pricepritem3 {
            get {
                return this.pricepritem3Field;
            }
            set {
                this.pricepritem3Field = value;
            }
        }

        /// <remarks/>
[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int priceitem3 {
            get {
                return this.priceitem3Field;
            }
            set {
                this.priceitem3Field = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int itemnumber4 {
            get {
                return this.itemnumber4Field;
            }
            set {
                this.itemnumber4Field = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int items4 {
            get {
                return this.items4Field;
```

```
            }
            set {
                this.items4Field = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int pricepritem4 {
            get {
                return this.pricepritem4Field;
            }
            set {
                this.pricepritem4Field = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public int priceitem4 {
            get {
 return this.priceitem4Field;
            }
            set {
                this.priceitem4Field = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public string Status {
            get {
                return this.statusField;
            }
            set {
                this.statusField = value;
            }
```

```
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public double orderAmount {
            get {
                return this.orderAmountField;
            }
            set {
                this.orderAmountField = value;
            }
        }

        /// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.Schema.
XmlSchemaForm.Unqualified)]
        public bool ManualCheckRequired {
            get {
                return this.manualCheckRequiredField;
            }
            set {
                this.manualCheckRequiredField = value;
            }
        }
    }
}
```

A.7 order.xsx

```
<?xml version="1.0" encoding="utf-8"?>
<!--This file is auto-generated by the XML Schema Designer. It holds layout
information for components on the designer surface.-->
<XSDDesignerLayout Style="LeftRight" layoutVersion="2" viewPortLeft="0"
viewPortTop="0" zoom="100">
    <order_XmlElement left="317" top="254" width="5292" height="2831"
    selected="0" zOrder="1" index="0" expanded="1" /> </XSDDesignerLayout>
```

A.8 order.xml

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <p:order xmlns:p="http://www.ilog.com/rules" xmlns:xsi="http://www.w3.
/2001/XMLSchema-instance">
  <orderId>0</orderId>
  <orderDate>2001-01-01</orderDate>
  <statusinfo />
  <kundenavn>Per</kundenavn>
  <Adresse>Karisvingen 5</Adresse>
  <Postnummer>7000</Postnummer>
  <Sted>Trondheim</Sted>
  <Telefon>22334455</Telefon>
  <epost>ola@gmail.com</epost>
  <itemnumber1>23</itemnumber1>
  <items1>3</items1>
  <pricepritem1>250</pricepritem1>
  <priceitem1>750</priceitem1>
  <itemnumber2>28</itemnumber2>
  <items2>2</items2>
  <pricepritem2>500</pricepritem2>
  <priceitem2>1000</priceitem2>
  <itemnumber3>45</itemnumber3>
  <items3>1</items3>
  <pricepritem3>50</pricepritem3>
  <priceitem3>50</priceitem3>
  <itemnumber4>40</itemnumber4>
  <items4>5</items4>
  <pricepritem4>15</pricepritem4>
  <priceitem4>75</priceitem4>
  <Status>mottatt</Status>
  <orderAmount>1875</orderAmount>
  <ManualCheckRequired>false</ManualCheckRequired>
  </p:order>
```

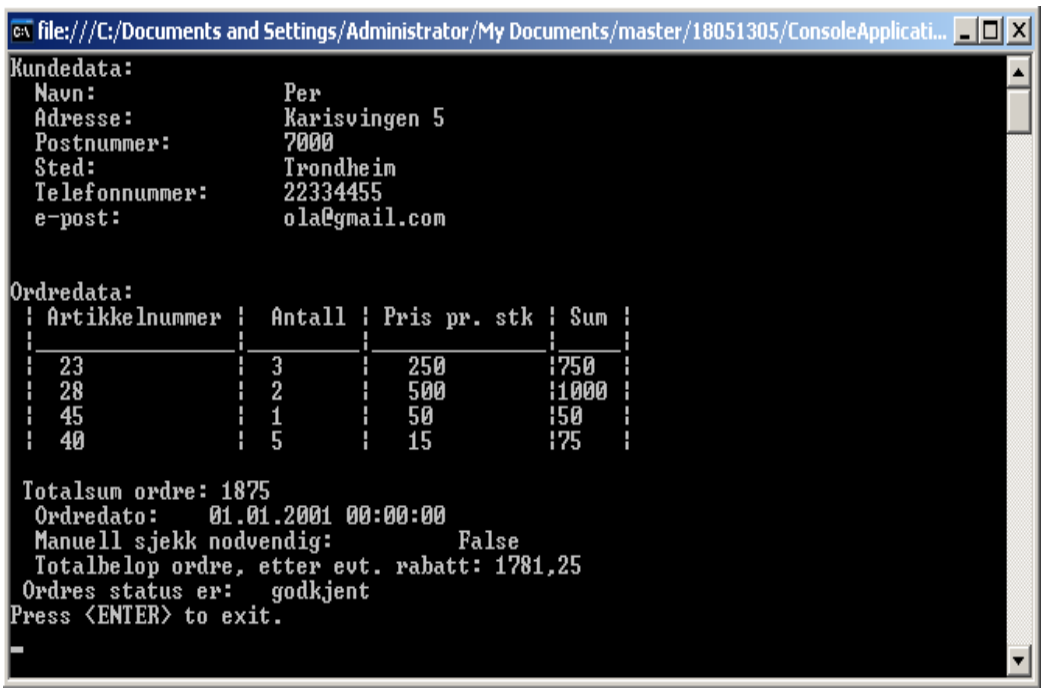
A.9 Program.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Xml.Serialization;
using ClassLibrary;
using ILOG.Rules;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {

            //Leser et xml-objekt og binder det til variabelen order
            XmlSerializer serializer = new XmlSerializer(typeof(order));
            XmlTextReader reader = null;
            order order = null;
            try
            {
                reader = new XmlTextReader(@"..\..\order.xml");
                order = (order)serializer.Deserialize(reader);
            }
            finally
            {
                if (reader != null)
                    reader.Close();
            }
            // Skriver ut verdiene til ordren
            Console.WriteLine("Kundedata:");
            Console.WriteLine("  Navn:                {0}", order.kundenavn);
            Console.WriteLine("  Adresse:             {0}", order.Adresse);
            Console.WriteLine("  Postnummer:          {0}", order.Postnummer);
            Console.WriteLine("  Sted:                {0}", order.Sted);
            Console.WriteLine("  Telefonnummer:       {0}", order.Telefon);
            Console.WriteLine("  e-post:              {0}", order.epost);
            Console.WriteLine();
            Console.WriteLine();
            Console.WriteLine("Ordredata:");
            Console.WriteLine(" | Artikkelnummer |  Antall | Pris pr. stk |
```

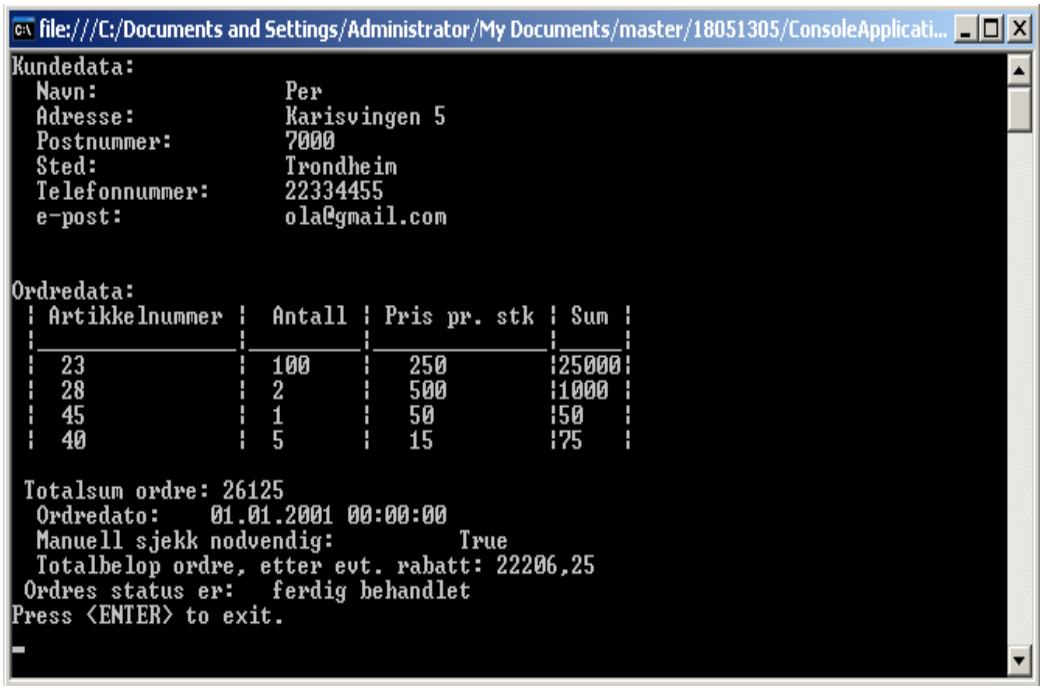
```
Sum |  ");
            Console.WriteLine(" |----------------|---------|----------
            ----|-----| ");
            Console.WriteLine(" |  {0}            |  {1}
            | {2}        |{3}  | ", order.itemnumber1, order.items1,
               order.pricepritem1, order.priceitem1);
            Console.WriteLine(" |  {0}            |  {1}
            | {2}        |{3} | ", order.itemnumber2, order.items2,
              order.pricepritem2, order.priceitem2);
            Console.WriteLine(" |  {0}            |  {1}
            | {2}         |{3}   | ", order.itemnumber3, order.items3,
             order.pricepritem3, order.priceitem3);
            Console.WriteLine(" |  {0}            |  {1}
            | {2}         |{3}   | ", order.itemnumber4, order.items4,
             order.pricepritem4, order.priceitem4);
            Console.WriteLine();
            Console.WriteLine(" Totalsum ordre: {0}", order.orderAmount)
            Console.WriteLine("  Ordredato:    {0}", order.orderDate);


            //Oppretter regelmotoren og kjører den
            RuleEngine engine = new RuleEngine();
               engine.Assert(order);
            // Laster og setter opp regelsettet
            engine.RuleSet = new BusinessRules1();
            //   Kjører reglene");
            //Setter parameterverdier. Dette må gjøres for å styre
            //flyten i kjøring av reglene
            engine.SetParameterValue("ordre", order);


            // Setter regelflyten
            engine.Ruleflow = engine.RuleSet.Ruleflows["BusinessRules1",
"Ruleflow1"];
            //Kjører regelmotoren....
            engine.Execute();

            if(order.Status == "avvist"){
                Console.WriteLine("Ordre ble avvist grunnet {0}",
                order.statusinfo);
            }
            else{
                Console.WriteLine("  Manuell sjekk nødvendig:
```

```
{0}",
order.ManualCheckRequired);
                Console.WriteLine("  Totalbeløp ordre, etter evt. rabatt:
{0}", order.orderAmount);
                Console.WriteLine(" Ordres status er:    {0}", order.Status);
            }
            Console.WriteLine("Press <ENTER> to exit.");
            Console.In.ReadLine();
        }
    }
}
```

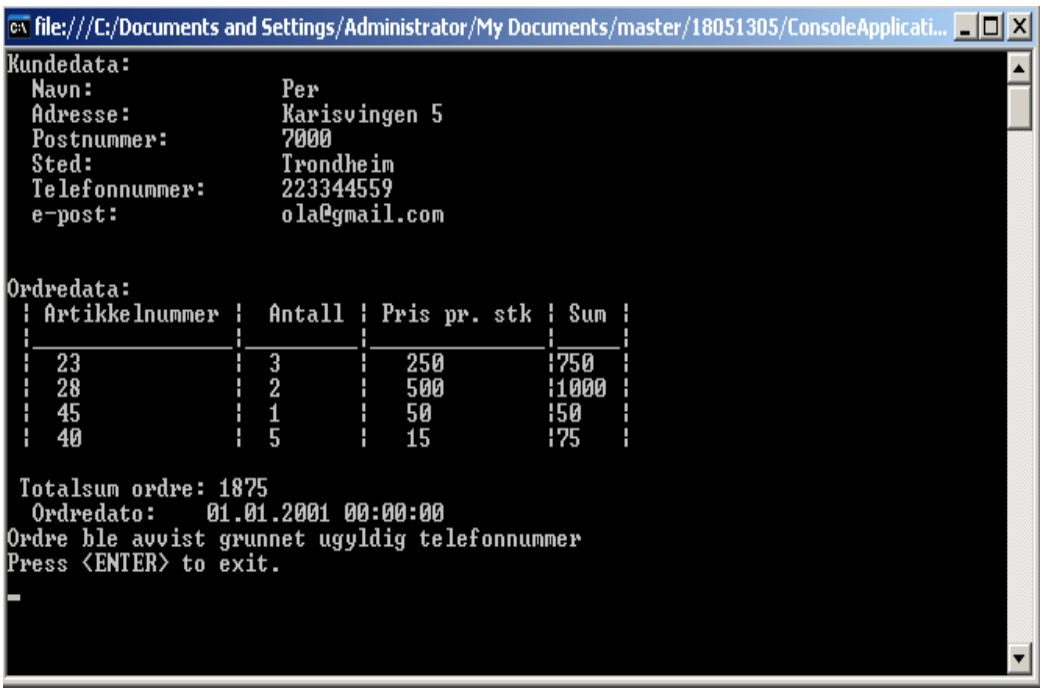
A.10 Output fra tester



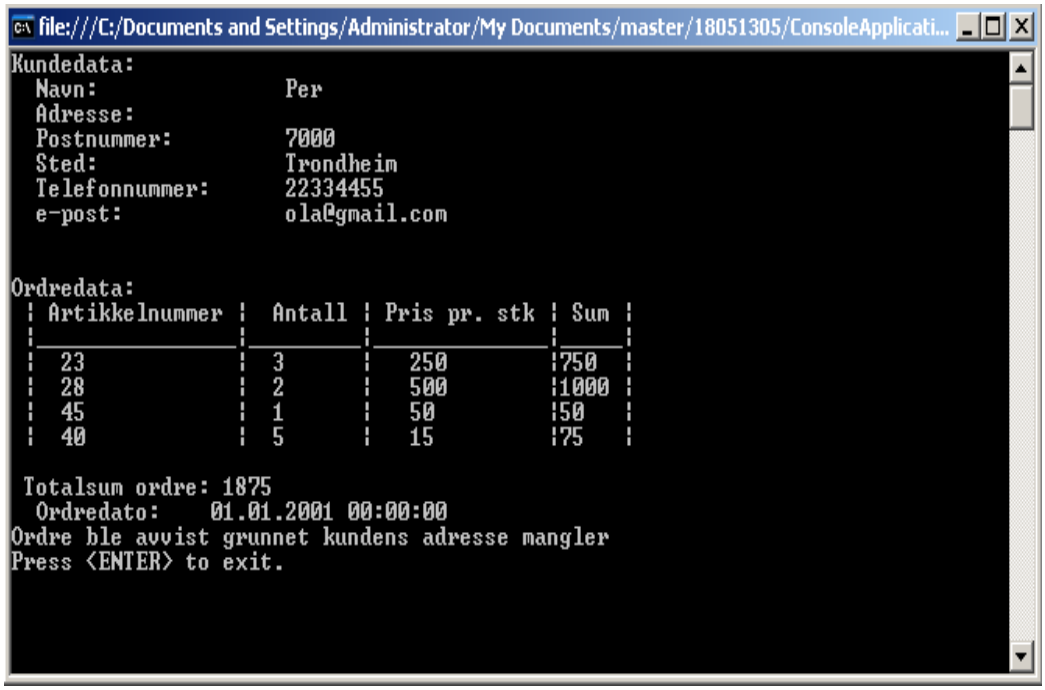
Figur A.6: Output fra Program.cs



Figur A.7: Output med beløp over manuellsjekkgrense



Figur A.8: Ouput når telefonnummer er fylt inn feil



Figur A.9: Output når adressefelt mangler