Universitetet i Oslo
Institutt for informatikk

Et flerspråklig data-modelleringsverktøy med XML som modellrepresentasjon

Eirik Meland

hovedfagsoppgave

16. februar 2004

Forord

Jeg vil rette en stor takk til min veileder, Gerhard Skagestein, som har vært til uvurderlig hjelp og gjort arbeidet med oppgaven spennende og lærerikt.

Jeg vil også takke Svein Håkon Henriksen, Mathias Meisfjordskar, Harald Meland, Espen Myhre, Martin Tostrup Setek, Kenneth Svee og Asbjørn Reglund Thorsen som har bidratt med faglig og moralsk støtte.

Innhold

Figurer

Kapittel 1

Innledning

1.1 Mål for oppgaven

Målet for denne oppgaven er å lage et Open Source datamodelleringsverktøy med et nøytralt fundament for alle mulige datamodelleringsspråk; et nøytralt datamodelleringsverktøy som kan fungere som brobygger mellom de forskjellige datamodelleringsspråkene. Med dette ønsker jeg å flytte modelleringsdebatten fra «visualiseringen av modellen» til den egentlige «modellen».

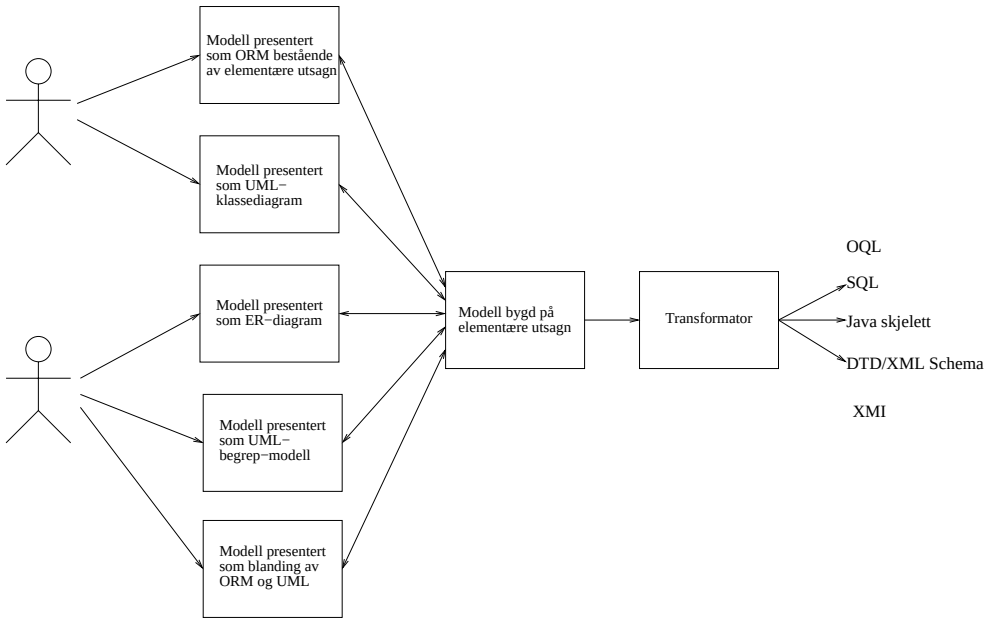
Verktøyet vil bruke XML (*Extensible Markup Language*) som intern modellrepresentasjon, og XSLT (*Extensible Stylesheet Language Transformations*) til bl.a. gruppering. Det vil bli undersøkt hvilke åpne standarder i kjølvannet av XML som kan brukes og hvilke konsekvenser dette gir.

XML[1] er designet for å gjøre det lett å utveksle strukturerte dokumenter over internett. XML blir mye brukt som format for utveksling og lagring av data. Jeg vil i tillegg bruke XML som intern datastruktur.

XSLT[2] er et «XML-språk» for å transformere et XML-format til et annet format, f.eks. HTML eller et annet XML-format. XSLT bruker XPath (*XML Path Language*)[3] for å navigere i XML-strukturen.

Under modelleringen skal det være mulig å få presentert modellen i ulike grafiske dialekter som f.eks. ER (*Entity-Relationhip*), ORM (*Object Role Modeling*) eller UML (*Unified Modeling Language*). Det skal også være mulig å blande disse for å kunne dra nytte av styrkene til de ulike teknikkene, f.eks. ved å gruppere et gitt ORM-begrep til en

[1] W3C recommendation[4]
[2] W3C recommendation[3, 8]
[3] W3C recommendation[2, 7]



Figur 1.1: Oversikt

UML klasse med attributter, eller blande ORM inn i UML for å kunne visualisere skranker som i UML bare kan uttrykkes ved hjelp av OCL (*Object Constraint Language*). Det kan også være ønskelig å vise en ORM modell som ER/UML for å gjøre den mer tilgjengelig for en bruker. Dette krever et verktøy som har støtte for flere modelleringsspråk.

Verktøyet skal kunne utvides til f.eks. å foreskrive databasedefinisjoner som SQL-kode utifra modellen, eller generere et Java-kode skjelett.

Jeg forutsetter at leseren har kjennskap til generell databaseteori, ORM/NIAM, og er kjent med anvendelsen av UML og ER. Jeg vil ikke bruke nevneverdig plass på å forklare UML, DTD (*Document Type Definition*) eller XSLT.

1.2 Oppbygning av oppgaven

Innledning
Beskrivelse av problemstilling.

Overordnet arkitektur
Dette kapittelet gir først en oversikt over de sentrale delene i arkitekturen: prosjekt, modell og presentasjon.

Arkitekturen blir så grundigere forklart ved hjelp av mønster og programvarelag.

Elementære utsagn blir introdusert som kandidat til en nøytral modell.

Plattformen blir diskutert og definert.

Elementære utsagn
Elementære utsagn blir definert og plassert i forhold til ORM.

Gjennomgang av funksjon og ORM-notasjon til de ulike bestanddelene til elementære utsagn.

Skranker
Skrankene blir forklart og kategorisert med hensyn på funksjon.

Skranker og elementære utsagn vil utgjøre den nøytrale modellen.

Manipulering av elementære utsagn
Ulike manipuleringer av elementære utsagn for å kunne presentere modellen på et høyere abstraksjonsnivå.

ORM metamodell
Nøyaktig beskrivelse av ORM ved hjelp av en metamodell, for å være istand til å lage en XML-mal.

Metamodellen vil bli sammenlignet Terry Halpins metamodell for ORM.

XML-representasjon av modellen
Gruppering av metamodellen til en XML-mal for den nøytrale modellen.

Visualiseringsdirektiver i XML
Gjennomgang av direktivene som trengs for å kunne visualisere de ulike presentasjonene av modellen.

Manipulering med XSLT
Algoritmer for identifisering og gruppering av elementære utsagn.

Identifiseringsalgoritmen påvirker modellen.

Grupperingsalgoritmen løfter data fra modellnivå til presentasjonsnivå.

Realisering av «Niamderthal»

Deler av realiseringen er forklart ved hjelp av sekvensdiagrammer.

Omfang av kildekoden.

Diskusjon

Er XML og XSLT egnet til dette formålet?

Erfaringer fra utviklingen av verktøyet, og forslag til utvidelser.

Vedlegg (separat)

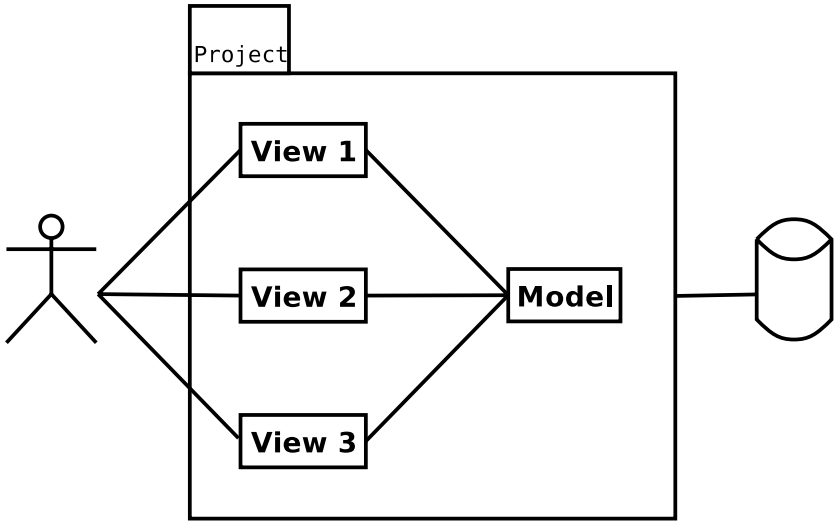
Kildekoden til det realiserte verktøyet.

Kapittel 2

Overordnet arkitektur

Dette kapittelet gir en oversikt over den valgte plattformen og de mest sentrale klassene i arkitekturen.

2.1 Prinsippskisse av arkitekturen



Figur 2.1: Prinsippskisse

Figur 2.1 (i UML-lignende notasjon) viser et meget forenklet bilde av arkitekturen for verktøyet. Det er delt inn i tre hoveddeler: **Project** (prosjekt) som inneholder et **Model**-objekt (modell) og en eller flere **View**-objekter (presentasjoner). **Project** har det overordnede ansvaret for **Model** og **View**, og tar seg av både lesing og lagring av disse.

Hvert **View** vet hvilken **Model** det tilhører, men **Model** kjenner ikke til noen **View**. Internstrukturen i modellen holdes hemmelig for presentasjonene, og modelldataene kan bare aksesseres gjennom API'et som modellen tilbyr. Dette resulterer i en «løs kobling» mellom presentasjon og modell, og gir muligheten til å erstatte et **Model** med en annen klasse som implementerer det samme API'et uten at dette påvirker resten av arkitekturen.

Brukeren manipulerer modellen gjennom en eller flere presentasjoner. Endringer i **Model** må sendes til **Project**. Derfra blir endringene sendt videre til **Project** sine **View**-objekter.

Modellen kan vises frem som f.eks. ORM, ER eller UML. Blandinger er også mulig, men de vil kun ha en grunntype. Hver av disse fremstillingene har sine egne figur-objekter og sin egen lerret-meny.

2.2 En nøytral modell

> En modell er en representasjon av noe, der visse egenskaper, som er viktige for det formål representasjonen skal brukes til, er fremhevet, mens øvrige egenskaper utelates.[23]

I verktøyet skilles det mellom selve modellen og presentasjonen av modellen. Dette skillet er et av grunnprinsippene i arkitekturen.[1]

Jeg vil prøve å forklare forskjellen mellom modell og presentasjon ved hjelp av et eksempel:

> I en spørreundersøkelse er det samlet inn data om hvor mange timer folk ser på TV hver dag. Det er flere måter å se på disse dataene:
>
> - individuelle tall i tabellform
> - kakediagram
> - histogram
> - osv

Alle disse vinklingene forholder seg til de samme dataene, og kan derfor betraktes som forskjellige presentasjoner av dataene (modellen).

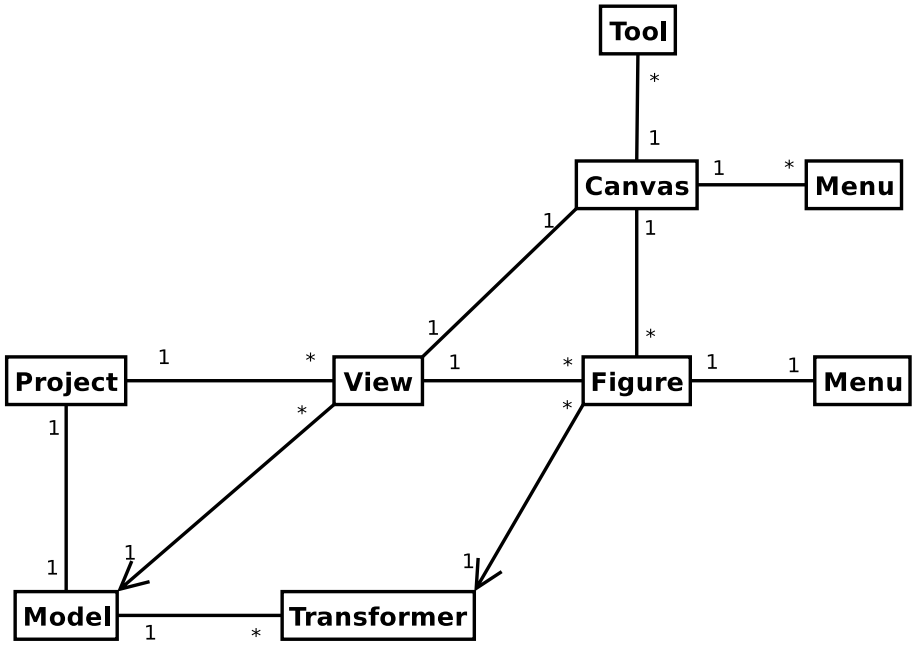
Ved å skille mellom modell og presentasjon slipper man å lagre redundante data i presentasjonene, og man får et fleksibelt design som gjør det mulig å koble på flere presentasjoner.

[1] Dette prinsippet er kjent fra MVC-mønsteret (*Model View Controller*), se kapittel 2.4.2 på side 9.

En nøytral modell skal være gjeldende for alle typer datamodelleringsteknikker. Jeg bygger på antagelsen om at en modell bygd opp av «elementære utsagn»[29] kan fungere som en slik nøytral modell (mer om «elementære utsagn» i kapittel 3 på side 17).

2.3 En mer detaljert oversikt over arkitekturen

Denne oversikten over arkitekturen viser de viktigste klassene.



Figur 2.2: Oversikt over arkitektur

Canvas deler livsløp med **View**. Det er likevel ønskelig å ha objektene fra disse klassene adskilt siden de har forskjellige ansvarsområder (mer om dette i kapittel 2.4 på neste side). Alle figurene (**Figure**-objektene) tegner seg selv på **Canvas**.

Canvas har en **Menu**. Denne menyen kommer opp når brukeren høyreklikker på bakgrunnen (**Canvas**), og er den eneste måten å tegne opp figurer på tomt lerret. **Canvas** holder greie på hvilken modelleringsteknikk som brukes og bruker tilsvarende **Menu**. Det finnes en bakgrunnsmeny for hver teknikk.

Tool bestemmer hvordan «mouse events» skal bli behandlet. **Canvas** kan skifte ut **Tool** under kjøring, men kan bare ha et om gangen. I utgangspunktet er det *Select-Tool* som er koblet til **Canvas**. Dette brukes til å velge figurer og flytte forhåndsvalgte figurer.

Hvis brukeren ønsker å koble sammen to figurer blir **Tool** skiftet ut med et «opprettelses»-**Tool** som tar imot «mouse events» fra brukeren for å få informasjon om hvilke figurer som skal kobles sammen, og hvor "koblingen" skal plasseres. Etter opprettelsen skifter lerretet tilbake til *Select-Tool*.

Figure står egentlig for mange forskjellige figurer som er sortert inn i forskjellige pakker etter hvilket modelleringsspråk de tilhører. Når jeg omtaler **Figure** mener jeg alle de ulike figur-klassene.

Figure inneholder en kobling til **Canvas**, **View** og **Transfomer**. Hver figur har en popup-meny. Denne menyen skiller seg fra lerretmenyen ved at den inneholder valgmuligheter man har fra figuren, og kan tilrettelegges pr. figurinstans, f.eks for å gi brukeren mulighet til å gjennomføre en begrepsdannelse av en assosiasjon som oppfyller kravene for å danne et begrep (se kapittel 5.1 på side 29).

Transformer tar seg av alle spørringene fra figuren til modellen, og er et abstraksjonslag mellom **Model** og **Figure** (mer om dette i kapittel 2.4).

2.4 Objektenes ansvarsområder

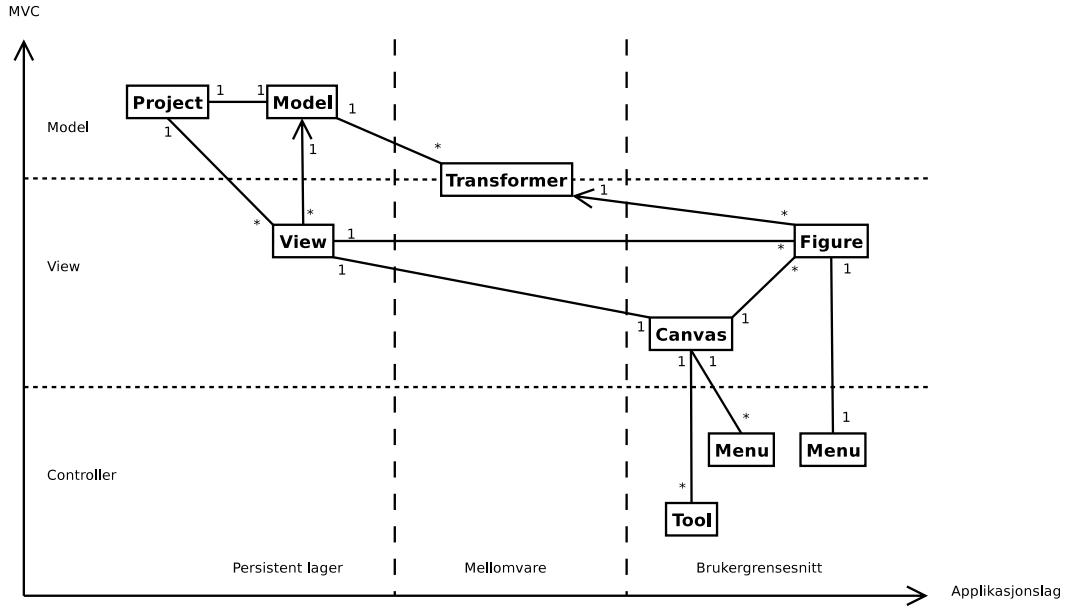
Jeg vil her dele inn arkitekturen på to ulike måter for å definere objektenes ansvarsområder.

2.4.1 Programvarelag

Horisontalt deler figur 2.3 på neste side arkitekturen inn i programvarelag:

Persistent lager består av **Project**, **Model** og **View**, og inneholder den interne XML-datastrukturen.

Det persistente lageret tilbyr sammen med mellomvarelaget et API for å aksessere den interne datastrukturen. Dette gir et modulært design, der «persistent lager» (eller måten dataene lagres på) kan byttes ut uten at brukergrensesnittet blir påvirket; en løs kobling mellom «persistent lager» og «brukergrensesnitt».



Figur 2.3: Inndeling av arkitektur

Mellomvarelag er et abstraksjonslag for å løfte modellen opp til et høyere nivå. Alle spørringer fra **Figure** til **Model** går gjennom denne mellomvaren, **Transformer**. I noen tilfeller må det gjøres noe med dataene for å løfte dem opp på ønsket nivå før de sendes videre, i andre tilfeller blir dataene sendt videre uten noen form for behandling. Jeg har allikevel valgt å ha en **Transformer** pr figur for å få en konsistent arkitektur.

Brukergrensesnitt er den delen av verktøyet som brukeren kan se og ha direkte interaksjon med. Klassene som befinner seg i denne gruppen er **Canvas** (lerretet), **Figure** (egentlig flere klasser som tilsvarer de ulike modellfigurene), **Menu** (lerret-meny og figur-menyene) og **Tool**.

Denne delen av arkitekturen lagrer ingenting, men gjør forespørsler mot «persistent lager» for å hente ut data ved hjelp av API'et som tilbys av **View** og **Transformer**.

2.4.2 MVC

Vertikalt deler figur 2.3 arkitekturen inn i MVC-mønsteret.

MVC-mønsteret[25] stammer fra Smalltalk-80, og er et mønster som skiller modell (Model), presentasjon (View), og kontroll (Control).

Model inneholder de data som er felles for alle presentasjoner og som trengs for å skape et sett med semantisk ekvivalente diagrammer; dvs et sett med diagrammer som har samme mening, men nødvendigvis ikke samme layout[2].

Hvis modellen endres må alle presentasjonene oppdateres. MVC bruker *observer*-mønsteret[25] til å håndtere denne en-til-mange avhengigheten. Jeg har valgt å spre MVC's Model i to klasser: **Model** som inneholder den interne datastrukturen, og **Project** som implementerer observer-mønsteret sammen med **View**. Ved opprettelse registrerer presentasjonen seg som "abonnent" i **Project**. Når modellen skal fortelle presentasjonene at den har endret seg, sender den beskjed til **Project**, som kommuniserer med alle sin abonnenter. Dette resulterer i at modellen er uavhengig av alle presentasjoner.

View er fremstillingen av dataene på skjermen. Klassene **Canvas** og **Figure** tilhører denne gruppen.

Controller definerer hvordan brukergrensesnittet reagerer på brukerinput. **Tool** og menyene utgjør Controller og utgjør brukerens interaksjonsgrensesnitt for å manipulere modellen og presentasjonen.

(Transformer) er plassert midt imellom Model og View. Dette er fordi den kjenner til modellens interne datastruktur, men inneholder ingen modell-data.

2.5 Oversikt over presentasjonsdelen

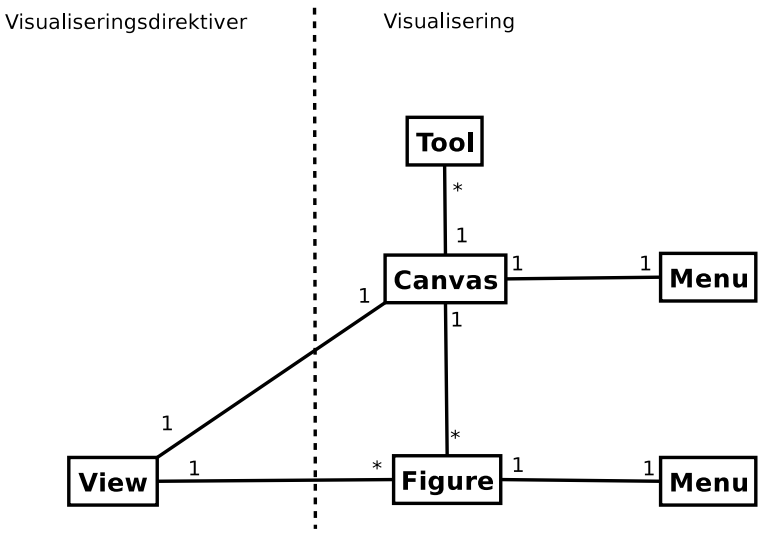
Visualiseringen blir tegnet opp på bakgrunn av visualiseringsdirektivene.

Hvert element i modellen har et tilsvarende element i visualiseringsdirektivene.

2.5.1 Visualisering

Visualiseringen består av et **Canvas**-objekt og flere **Figure**-objekter.

[2]Dette forutsetter at layout ikke er av semantisk betydning i diagrammet.



Figur 2.4: Presentasjon

Ved opptegning sender **Canvas** beskjed til alle sine **Figure**-objekter om at de skal tegne seg opp. Hvert **Figure**-objekt tegner så seg selv på **Canvas**.

Figurene er bygd opp ved hjelp av *composite* mønsteret, som har følgende oppgave:

> Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.[25]

Dette medfører at lerretet kan behandle alle figurobjektene likt. Figurer som er sammensatt av flere andre figurer, tar ansvaret for å tegne opp sine "barn".

2.5.2 Visualiseringsdirektiver

Visualiseringsdirektivene blir lagret sammen med modellen, og har som oppgave å "huske" hva som skal bli tegnet, hvor det skal bli tegnet, og i noen tilfeller også hvordan det skal bli tegnet.

Ved opprettelse av en ny figur er det mulig å automatisk plassere figuren i de presentasjonene hvor plassering ikke eksplisitt blir oppgitt. Algoritmen for plassering kan da plassere figuren f.eks. slik at færrest mulig linjer krysser hverandre. Siden den nye figuren bare får eksplisitt plassering i den aktive presentasjonen, vil den automatiske

plasseringen bli gjort i de inaktive presentasjonene. Brukeren vet ikke hvor den nye figuren ble plassert i disse diagrammene.

En slik plasseringsalgoritme er altfor omfattende som en del av denne oppgaven. Jeg plasserer derfor alle figurer som ikke eksplisitt får plassering under opprettelse (inaktive presentasjoner) i «origo» på lerretet, og lar brukeren flytte dem til ønsket sted. I tillegg til å være en enkel løsning, har dette også den fordelen at brukeren har full kontroll over plasseringene.

View inneholder alle visualiseringsdirektivene lagret som XML. **Figure**-objektene inneholder kun en ID, og hver gang en figur skal tegnes opp, spør den **View** om hvor den skal tegne seg selv opp.

Hvis modellen må løftes opp på et høyere abstraksjonsnivå enn den befinner seg på, er dette spesifisert i visualiseringsdirektivene. Figuren spør transformatoren om modelldataene den trenger, og transformatoren løfter dataene til ønsket abstraksjonsnivå.

Visualiseringsdirektivene vil bli gjennomgått i kapittel 8 på side 66.

2.6 Åpne standarder

Jeg vil her beskrive de åpne standardene som er premisser for oppgaven.

2.6.1 XML

I november 1996 ga *W3C* ut det første «Working Draft» for XML (*Extensible Markup Language*)[1]. Der spesifiserer de målene med XML slik:

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.

10. Terseness is of minimal importance.

XML er hierarkisk oppbygde tekstfiler som bruker nøstede start- og stopp-markeringer for å beskrive innhold.

XML skiller seg fra andre markup-språk som f.eks. HTML ved at HTML brukes for å beskrive layout, mens XML brukes for å beskrive data og trenger ikke ha noe med layout å gjøre.

En av fordelene med XML er at det skrives i ren tekst[32]. Dette bidrar til å gjøre utveksling av data mellom ulike plattformer enklere.

XML parser

For å kunne gjøre nytte av XML, må verktøyet kunne lese inn et XML-dokument, lage en intern datastruktur og lagre dataene som XML igjen. For dette formålet kreves en XML-parser.

Det finnes to dominerende kategorier av XML-parsere, SAX og XML-DOM.

- **SAX**

  > Simple API for XML. A standard interface for event-based XML parsing.[13]

  SAX (*Simple API for XML*) er «event-driven». Dette betyr at den benytter seg av «events» ved parsering av dokumentet for å fortelle f.eks. at et nytt element er startet eller sluttet. Programmeren må spesifisere hva han ønsker å gjøre ved de forskjellige «events».

  Fordelen med denne typen parsering er at man ikke trenger å ha hele dokumentet i minnet samtidig. Man kan under parsering plukke ut de deler man synes er interessante og utelate resten. SAX er et API for parsering av XML, og manipulering av data og skriving til fil er definert utenfor ansvarsområdet.

- **XML-DOM**

  > The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page.[5]

DOM (*Document Object Model*) er «tree-driven». Dette betyr at hele dokumentet leses inn i minnet som en trestruktur. DOM har støtte for å manipulere dette treet og «serialisere» det til en streng for f.eks å kunne skrive det til fil.

Flere XSLT-prosessorer støtter også DOM-tre som input (mer om XSLT i kapittel 9 på side 74).

Jeg har valgt å bruke XML-DOM fordi den tilbyr en datastruktur med tilhørende API for lesing, lagring og manipulering.

Velformet eller validerbar XML

Ved å definere de ulike elementenes rolle i en formell «mal», *Document Type Definition* (DTD) e.l., kan dokumenter kontrolleres med hensyn på om elementene forekommer på lovlige steder. Hvis et dokument oppfyller malen er det validert (*valid*).

Uten mal er det kun mulig å kontrollere om dokumentet oppfyller kravene for velformethet (*wellformed*), eller syntaktisk riktig XML.

Hvis et dokument er validert, impliserer dette at det er velformet.

Jeg har valgt å bruke validerbar XML fordi en mal kan være nyttig under utviklingen for å kontrollere "håndskrevne" test-data. I tillegg kan malen brukes som dokumentasjon av XML-strukturen.

XML-mal

Det finnes flere standarder for å skrive maler for XML-dokumenter. De to mulighetene jeg har vurdert er XSD (*XML Schema Definition*)[17] og DTD (*Document Type Definition*)[10].

Selv om XSD er nyere og gir flere muligheter, har jeg valgt å bruke DTD på grunn av bedre verktøy- og biblioteksstøtte[3].

2.6.2 XSLT og XPath

XSLT

XSLT er et eget språk for å behandle XML.

> The term stylesheet reflects the fact that one of the important roles of XSLT is to add styling information to an XML

[3]XSD er f.eks. ikke er fullt støttet i Saxon

> source document, by transforming it into a document consisting of XSL formatting objects (see [XSL]), or into another presentation-oriented format such as HTML, XHTML, or SVG. However, XSLT is used for a wide range of XML-to-XML transformation tasks, not exclusively for formatting and presentation applications.[8]

Det er store forandringer fra XSLT 1.0 til 2.0. En av grunnene er at XSLT 1.0 hadde mange barnesykdommer som er forsøkt ryddet opp i. Eksempelvis var det i XSLT 1.0 ikke mulig å gjøre spørringer i datatypen «result tree» (resultater). Dette gjorde at mange XSLT 1.0 implementasjoner brøt standarden og tilbød egne utvidelser for å løse dette problemet.

I XSLT 2.0 er alt sekvenser (*sequence*), og det er dermed mulig å foreta spørringer i resultater (som også er en sekvens). Det er også mulig å definere egne funksjoner som kan kalles fra XPath-uttrykk.

XSLT 2.0 og XPath 2.0 tilbyr ny funksjonalitet som kan gjøre stilarkene lettere å programmere og vedlikeholde. Jeg ønsker derfor å bruke XSLT 2.0 selv om denne standarden ikke er ferdig[4].

XPATH

> The primary purpose of XPath is to address the nodes of [XML 1.0] trees. XPath gets its name from its use of a path notation for navigating through the hierarchical structure of an XML document. XPath uses a compact, non-XML syntax to facilitate use of XPath within URIs and XML attribute values.[7]

XPath (*XML Path Language*) er et språk for å adressere de ulike delene i et XML-dokument. Det er integrert i XSLT, men kan også bli brukt fra Java for å navigere i et DOM-tre.

2.7 Programmeringsspråk

Jeg har valgt å bruke Java som programmeringsspråk for å kunne bruke Saxon. Saxon er den eneste XSLT-prosessoren som har begynt å implementere XSLT 2.0, og er skrevet i Java.

[4]XSLT 2.0 har i skrivende stund status som «Working Draft»

2.7.1 Relevante biblioteker

Jeg har funnet følgende java-biblioteker som kan være relevante for oppgaven (alfabetisk rekkefølge):

Dom4J [9] er en XML-DOM parser som har mer funksjonalitet enn W3Cs DOM[5]. Saxon støtter DOM4J-trær som input.

Log4j [11] er et bibliotek for å skrive logger. Biblioteket er raskt, konfigurerbart, og mye brukt i Java-miljøet.

Kapittel 3

Elementære utsagn

Jeg har tidligere forutsatt at elementære utsagn er fellesnevneren for alle datamodelleringsspråk, og derfor egnet som nøytral modell.

Dette kapittelet gir en innføring i elementære utsagn og delene som elementære utsagn er bygd opp av.

ORM er bygd på ideen om elementære utsagn og er derfor et naturlig valg for visualisering av disse.

3.1 ORM

ORM (*Object Role Model*) stammer fra NIAM (*Natural language Information Analysis Method*), som ble utviklet av en forskningsgruppe i Nederland.

NIAM bygger på følgende prinsipper (jf. [33]):

Naturlig språk
: Siden naturlig språk er den mest fundamentale kommunikasjonen mellom mennesker, må modellen kunne uttrykkes som naturlig språk, slik at den blir forstått av de som virkelig forstår interesseområdet. «Elementære utsagn» er en formalisert delmengde av naturlig språk.

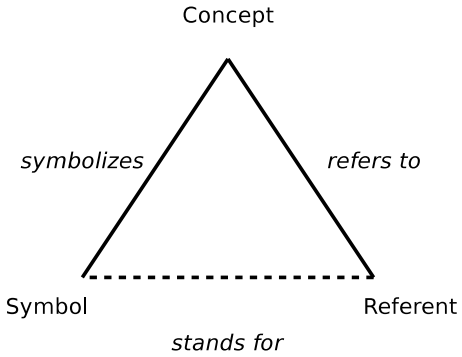
100% regelen
: Interesseområde (*Universe of Discourse*) er det utsnittet av virkeligheten man ønsker å beskrive ved hjelp av modellen.

  Det er mulig å lage en modell av vårt interesseområde som er nøyaktig nok til å tjene som konseptuelt skjema (i tre-skjema ar-

kitekturen) i en implementasjon av informasjonssystemet for vårt interesseområde.

Dette er viktig for at det skal være mulig å bruke elementære utsagn som en nøytral modell.

Ogdens trekant
Forholdet mellom de tre essensielle hjørnesteinene i all modellering (virkelighet, begrep og representasjon) er illustrert ved hjelp av Ogdens trekant[34].



Figur 3.1: Ogdens trekant

1. Ting, gjenstand eller fenomen i interesseområde som vi ønsker å referere til (nederst til høyre).
2. Forestillingen eller begrepet vi bruker for å referere til tingen, gjenstanden eller fenomenet (øverst).
3. Symbol vi bruker for å representere forestillingen eller begrepet (nederst til venstre)

3.2 Elementære utsagn

Utsagn er begreper som spiller roller overfor hverandre.

Et «elementært utsagn» er et utsagn som ikke kan deles uten at informasjon går tapt i henhold til interesseområdet. Et utsagn kan være elementært i et interesseområde, men ikke elementært i et annet. [33]

- **Personen** med *fornavn* 'Kari' fikk **stillingen** med *stillingsbetegnelse* 'avdelingsingeniør' på **dag** med *dato* '1. juni 2003'.

Utsagnet er elementært hvis interesseområdet er Karis karriere, men hvis interesseområdet bare innbefatter nåværende stilling, er utsagnet ikke elementært. Det kan da deles opp i to utsagn uten å miste informasjon:

- **Personen** med *fornavn* 'Kari' har **stillingen** med *stillingsbetegnelse* 'avdelingsingeniør'.
- **Personen** med *fornavn* 'Kari' fikk sin nåværende stilling på **dag** med *dato* '1. juni 2003'.

[1]

3.3 Oppbygning av utsagn

Jeg vil her gå igjennom de ulike delene som elementære utsagn består av.

3.3.1 Begreper og representasjoner

Et begrep er forestillingen om en ting/gjenstand/fenomen i interesseområdet. Begrepene er skrevet med uthevet skrift i de elementære utsagnene i kapittel 3.2 på forrige side: **Person**, **stilling** og **dag**. Begreper kalles også `nolot` (**non-lexical object type**), og trenger en representasjon for å kunne lagres.

I ORM kalles et begrep for «Entity» og tegnes som en heltrukket oval med begrepets navn midt i ovalen.

Alle begrepene i eksempelet har en representasjon:

- **Person** → *fornavn*
- **stilling** → *stillingsbeskrivelse*
- **dag** → *dato*

Assosiasjoner mellom begreper og en representasjoner kalles «broer».

Representasjoner kalles også `lot` (**lexical object type**), og må knyttes til et begrep for at de skal gi noen mening i modellen.

[1] Mer utfyllende informasjon om NIAM/ORM/«elementære utsagn» finnes i [36, 31, 40].

Ogdens trekant illustrerer at for å referere til en virkelig ting/gjenstand/fenomen ved hjelp av et «symbol», må man gå veien om begrep. Det er akkurat det som skjer når et begrep får en representasjon. Linjen til venstre i trekanten er en bro.

Det finnes flere måter et begrep kan få sin representasjon på:

1. **Representasjon ved arv**
2. **Representasjon ved perfekt bro**
3. **Representasjon ved entydig utsagn**
4. **Sammensatt representasjon**

Disse representasjonsformene vil bli gjennomgått i sin helhet i kapittel 9.1 på side 74.

I ORM kalles en representasjon for «Value» og tegnes som en stiplet oval med representasjonens navn i midten.

Et begreps representasjon kan også skrives under begrepsnavnet (i begrepet) som representasjonsnavn omsluttet av parenteser. Denne notasjonen er gjensidig utelukkende med førstnevnte notasjon.

3.3.2 Verdi

Verdier er noterbare forekomster av representasjoner. De elementære utsagnene i forrige avsnitt inneholder flere verdier (skrevet i anførselstegn): 'Kari', 'avdelingsingeniør' og '1. juni 2003'.

Jeg vil hovedsakelig oppholde meg på modelleringsnivået og derfor ikke berøre verdier i noen stor grad.

3.3.3 Assosiasjoner og roller

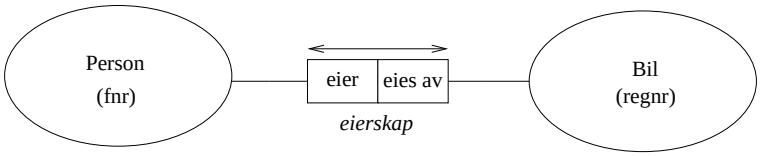
Begreper og representasjoner bindes sammen ved hjelp av assosiasjoner, hvis meningsinnhold er beskrevet ved hjelp av roller.

- **Person** *eier* **Bil**
- **Bil** *eies av* **Person**

Tilsammen beskriver disse to rollene (skrevet i kursiv) assosiasjonen mellom begrepene **Person** og **Bil**.

I det første utsagnet er *eier* rollen som **Person** spiller overfor **Bil**. «Korollen» til **Person** er rollen som **Bil** spiller overfor **Person**; *eies av*.

For at ORM skal kunne fungere som en nøytral modell tillater jeg at et predikatnavn kan bindes til en assosiasjon for å beskrive hele assosiasjonen. Eksempel på predikatnavn til assosiasjonen i eksempelet kan være «eierskap». Predikatnavnet skrives i kursiv under eller over assosiasjonen som vist i figur 3.2.



Figur 3.2: Utsagn

I ORM kalles en assosiasjon for «relationship», og en rolle for «role».

Kapittel 4

Skranker

Skranker er restriksjoner på hvilke forekomster av elementære utsagn og erstatninger av forekomster som er lovlige i elementære utsagn.

Dette kapittelet inneholder kategorisering og forklaring av de ulike skrankene.

4.1 Notasjon og kategorisering

Som grafiske symboler for skranker bruker jeg notasjon fra «Fra virkelighet til datamodell»[38] og «Datatorientert systemutvikling»[36]. Denne notasjonen bruker matematiske symboler istedenfor engelskspråklige forkortelser, og bruker samme symbol for ulike varianter av skranken.

I ORM kalles skranker for «constraints». De kan deles inn på følgende måte:

Kategori			Skrankenavn
Rolle		Intern	Unique Internal (*UI*) Frequency Internal (*FI*) Mandatory Simple Internal (*MSI*)
		Ekstern	Unique External (*UE*) Frequency External (*FE*) Mandatory Simple External (*MSE*)
	Mengde	Ikke kommutativ	SubSet (*SS*)
		Kommutativ	eXclusion (*X*) Equality (*EQ*)
Ring	Genererende		Intransitive (*IT*) Antisymmetric (*AN*) Irreflexive (*IR*)
	Regulerende		Symmetric (*SY*) Reflexive (*RE*) Transitive (*TR*)
Verdi			Value (*V*) Default Value (*DV*)
Subtype			SubType (*ST*)

For at en skranke skal være gyldig kan den bare beskranke elementer med støtte i en algebra, f.eks. slik at elementene kan testes på verdilikhet.

4.2 Rolleskranker

Rolleskranker finnes i to varianter: interne og eksterne.

En intern skranke er en skranke som legger restriksjoner på verdier i roller i samme assosiasjon.

En ekstern skranke legger restriksjoner på verdier i roller som befinner seg i forskjellige assosiasjoner. Denne varianten tegnes som skrankens symbol omgitt av en heltrukket sirkel med stiplede linjer knyttet til rollene den beskranker.

4.2.1 Entydighet

Denne skranken betyr at kombinasjonen av verdiene i rollene som skranken spenner skal være unik for alle forekomster.

Intern entydighetsskranke

(ORM: Unique Internal)
Skranken blir tegnet som en strek med pil i begge ender over rolle-

ne den spenner. Hvis rollene den spenner ikke befinner seg ved siden av hverandre, utelater man streken over de rollene som befinner seg mellom starten og slutten på skranken, men som ikke er en del av skranken (se figur 8.3 på side 71).

Ekstern entydighetsskranke

(ORM: Unique External)
Denne skranken bruker samme symbol som den interne varianten, men har en sirkel rundt pilen og stiplede linjer fra sirkelen til alle rollene som er med i skranken. Notasjonen er tidligere brukt i [36].

4.2.2 Frekvens

(ORM: Frequency Constraint)
Denne skranken beskriver hvor mange ganger en forekomst kan forekomme i en assosiasjon, og kan funksjonelt ses på som en generalisering av entydighetsskranken; en entydighetsskranke er en frekvensskranke der $frekvens = 1$.

Frekvensen oppgis som en liste med heltall og/eller intervaller.

Siden denne skranken betraktes som en generalisering av entydighetsskranken er det naturlig at også denne skranken kan ha en intern (*FI*) og en ekstern (*FE*) variant.

Frekvensskranken tegnes likt som entydighetsskranken med frekvensen oppgitt midt på streken.

4.2.3 Påbudt rolle

(ORM: Mandatory Simple Constraint)
Den interne varianten (*MSI*) av denne skranken tegnes som en hake ($\bigvee$) på rollestreken ved siden av rollen. Skranken plegger rollen som er beskranket å være utfyllt for alle forekomster.

Den eksterne varianten (*MSE*) betyr at en eller flere av de beskrankede rollene må være utfylt.

4.3 Mengeskranker

Mengdeskranker skiller seg fra rolleskranker ved at de angår to grupper av roller istedenfor en. Mengdeskrankene kan deles inn i to kategorier: kommutative (**likhet** og **ulikhet**) og ikke kommutative (**delmengde**).

4.3.1 Likhet

(ORM: Equality)
Denne skranken betyr at verdiene i forekomstene i to rollegrupper skal være de samme. Skranken tegnes som et likhetstegn ($=$).

4.3.2 Ulikhet

(ORM: eXclusion)
Denne skranken er det motsatte av sistnevnte skranke. Verdiene i forekomstene i to rollegrupper skal være disjunkte. Skranken tegnes som et ulikhetstegn ($\neq$).

4.3.3 Delmengde

(ORM: Subset)
Denne skranken sier at forekomstene i en rollegruppe skal være en delmengde av forekomstene i en annen rollegruppe.

I ORM tegnes den som en pil fra supermengde til delmengden. Jeg har valgt å følge ifi-standarden og bruke delmengdetegn ($\subseteq$). Ettersom skranken ikke er kommutativ er $\subseteq$-tegnets retning avgjørende i visualiseringen.

4.4 Ringskranker

Ringskrankene har alle sitt opphav i relasjonsalgebra. De kan deles inn i to grupper.

Regulerende ringskranker nekter forekomster som er i konflikt med skranken å bli lagt inn i databasen.

Asyklisk
En relasjon R på en mengde S er asyklisk hviss for alle $x_1, ..., x_n \in S$, $x_1 R x_2 \ldots x_{n-1} R x_n \Rightarrow \neg x_n R x_1$
(Relasjonen inneholder ingen sykler.)

Antisymmetrisk
En relasjon R på en mengde S er antisymmetrisk hviss for alle $x, y \in S$, (xRy og yRx) $\Rightarrow x = y$.
(Distinkte elementer er ikke relatert til hverandre.)

Intransitiv
En relasjon R på en mengde S er intransitiv hviss $x, y \in S$, (xRy og yRz) $\Rightarrow \neg xRz$

Irrefleksiv
En relasjon R på en mengde S er irrefleksiv hviss for alle $x \in S$, $\neg xRx$.
(Ingen elementer er relatert til seg selv.)

Genererende ringskranker legger til ekstra forekomster for å opprettholde ringskranken ved innlegging av data. Sletting kan håndteres på to måter. Den kan slette «ekstra data», eller nekte sletting med mindre alle relaterte forekomster blir slettet i samme transaksjon.

Symmetrisk
En relasjon R på en mengde S er symmetrisk hviss $xRy \Rightarrow yRx$, for alle $x, y \in S$.

Transitiv
En relasjon R på en mengde S er transitiv hviss for alle $x, y, z \in S$, (xRy og yRz) $\Rightarrow xRz$.

Refleksiv
En relasjon R på en mengde S er refleksiv hviss xRx for alle $x \in S$.
(Alle elementer er relatert til seg selv.)

(jf. [14, 36, 26])

4.5 Verdiskranke

(ORM: Value Constraint)
Denne skranken kan oppgis på samme måte som frekvens, men tekstlige verdier er også tillatt.

Skranken tegnes som en mengde med verdier ved siden av representasjonen den beskranker (se 6.1 på side 40).

4.6 Standardverdiskranke

(ORM: Default Value)
Denne skranken oppgis som en streng, og forteller hva som er standardverdien for en representasjon.

4.7 Subtypeskranke

Subtyper og supertyper er det som i UML kalles «spesialisering» og «generalisering». Et begrep kan være subtype av et annet begrep. Hvis A er subtype av B, så er B supertypen til A.

Subtype-skranken skiller seg fra de andre skrankene ved at den krever et eget språk for å uttrykke subtypedefinisjoner. En subtypedefinisjon forteller hva som må oppfylles for at en forekomst skal tilhøre en bestemt subtype.

Subtypedefinisjonene i Halpins metamodell kan deles inn i tre kategorier:

1. Hver A er en subtype av B (supertype) der B spiller [rollenavn] overfor C

   - **each** Subtype **is an** ObjectType **that** is a «subtype of» **some** ObjectType

   «subtype of» er rollen som ObjectType spiller overfor seg selv. Rollen fungerer her som «diskriminator» for subtypen «Subtype».

   Denne kategorien omhandler assosiasjoner.

2. Hver A er en subtype av B (supertype) der B spiller [rollenavn] overfor C som er en av verdiene i mengden {'x', 'y', ...}

   - **each** ValueType **is an** ObjectType **that** is of OTKind 'VT'
     (OTKind fungerer her som diskriminator for ValueType)
   - **each** SetComparisionConstraint **is a** Constraint **that** is of ConstraintType **in** {'SS', 'EQ', 'X'}
     (ConstraintType fungerer her som diskriminator for SetComparision-Constraint.)

   Denne kategorien omhandler assosiasjoner og verdier og kan betraktes som en spesifisering av kategori 1.

3. Hver A er en subtype av B (supertype) som ikke er C

   - **each** UnnestedEntityType **is an** EntityType **that is not a** NestedEntityType

   Denne kategorien sier at alt som ikke er av subtype X er av subtype Y, og er en disjunkt skranke.

Skranken tegnes som en pil fra subtype til supertype. Subtypedefinisjonene samles i en firkant som kan plasseres som regel i et av hjørnene i diagrammet (se f.eks. figur 6.1 på side 40 eller figur 6.3 på side 49).

Kapittel 5

Manipulering av elementære utsagn

Elementære utsagn kan presenteres på forskjellige måter.

Jeg vil i dette kapittelet forklare hvordan binære elementære utsagn kan manipuleres for å komme frem til noen slike presentasjoner.

5.1 Begrepsdannelse

(ORM: Objectified relationship)
En begrepsdannelse er å danne et begrep av en assosiasjon. Dette er vanlig i dagligtalen:

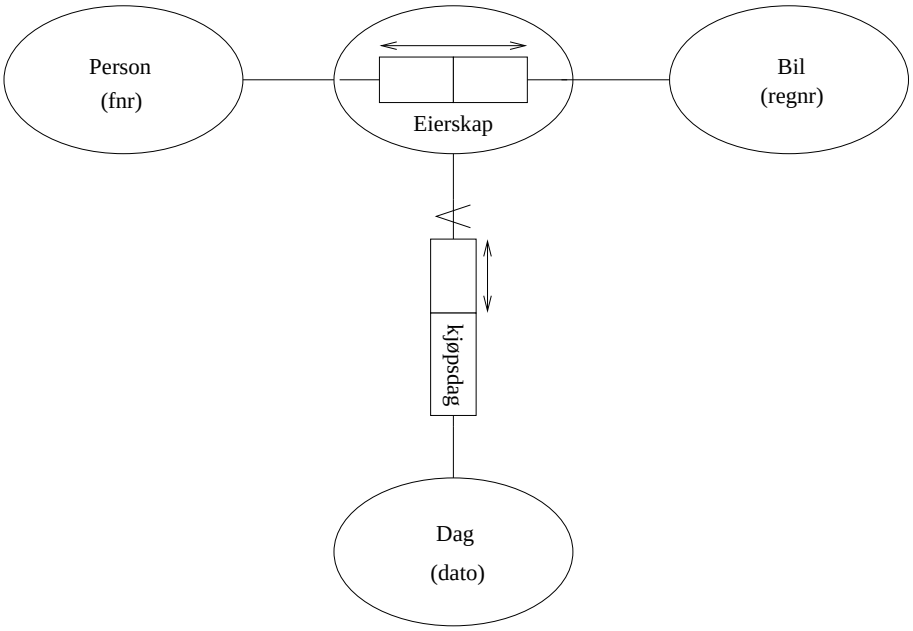
> "Per *eier* en Volvo. **Eierskapet** ble inngått 2003-01-01."

Rollen fra første setningen (*eier*), blir gjort om til et begrep i den andre setningen (**Eierskap**). Figur 5.1 på neste side viser en grafisk notasjon av eksempelet.

Begrepsdannelser er vanlig i modelleringsspråk, og finnes i både ER, UML og ORM/«elementære utsagn». Begrepsdannelser brukes bl.a. for å kunne knytte opplysninger til en assosiasjon.

I ORM tegnes en begrepsdannelse som en mange-til-mange assosiasjon med et begrep rundt. Begrepsnavnet skrives under eller over assosiasjonen. Under modelleringen behandles den som et begrep.

Hvis man tillater assosiasjoner som ikke er mange-til-mange å danne begreper, betyr dette at man tillater ikke-elementære utsagn i modellen. Det finnes dog tilfeller der det kan virke hensiktsmessig å tillate dette[29].



Figur 5.1: Begrepsdannelse

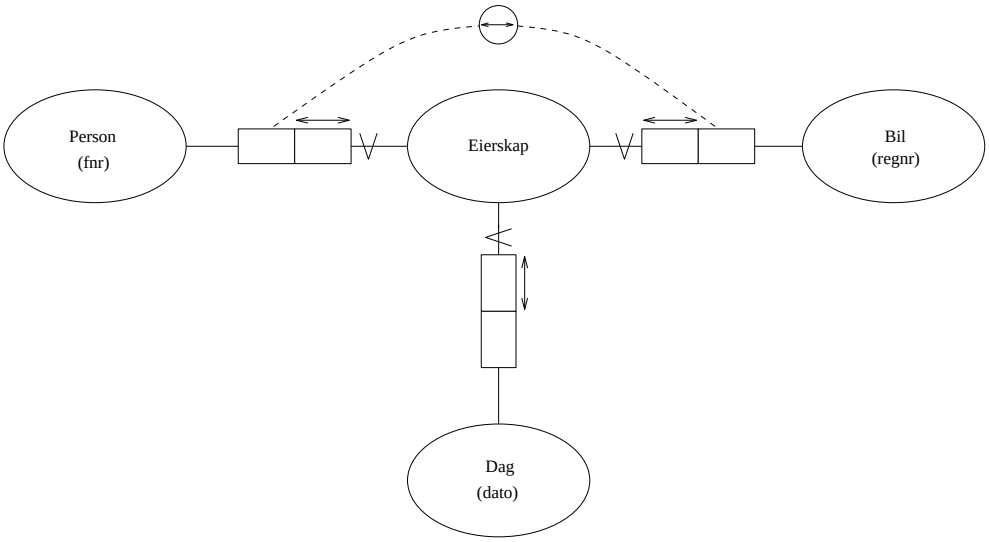
En begrepsdannelse kan skrives om til binære elementære utsagn. Begrepsdannelsen blir da et eget begrep (se figur 5.2 på neste side).

Figur 5.1 og figur 5.2 på neste side er semantisk ekvivalente, og figur 5.1 kan derfor betraktes som en presentasjon av figur 5.2.

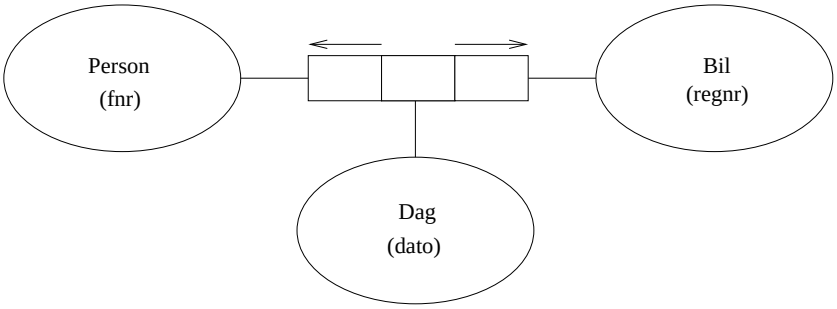
5.2 N-ære utsagn

En n-ær assosiasjon er en assosiasjon med n-roller. Hvis alle rollene som er tilknyttet en begrepsdannelse er påkrevde er det mulig å skrive den om til en n-ær assosiasjon (se figur 8.3 på side 71) uten at semantikken blir påvirket (ved hjelp av sterk gruppering, se kapittel 5.3 på side 32).

Hvis man også tillater omskriving av roller som ikke er påkrevde vil en slik gruppering resultere i informasjonstap (svak gruppering, se kapittel 5.3 på side 32). Det vil da ikke finnes en 1:1 avbildning mellom de binære og det n-ære utsagnet, og det vil ikke være mulig å skille påkrevde fra ikke-påkrevde roller; den grupperte presentasjonen vil ikke lenger være semantisk ekvivalent med utganspunktet.



Figur 5.2: Begrepsdannelse som binære utsagn



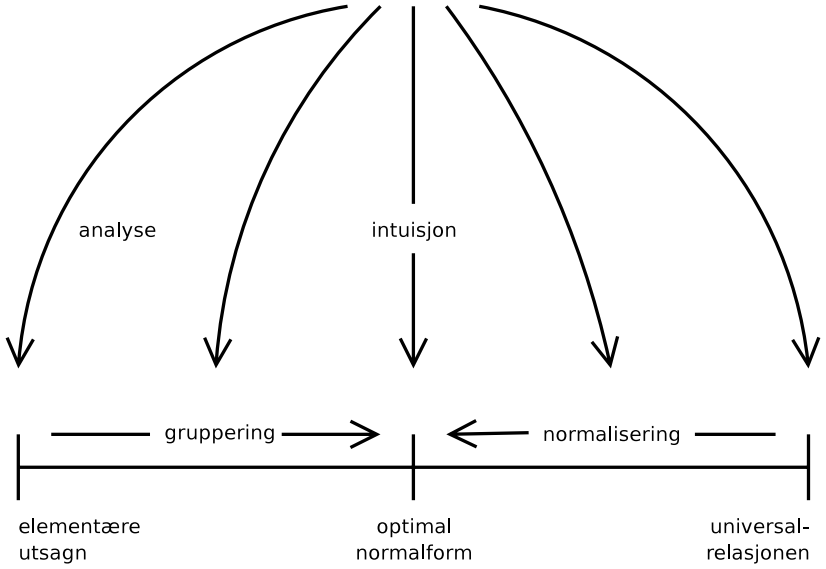
Figur 5.3: Ternært utsagn

5.3 Gruppering

Dette avsnittet omhandler gruppering fra elementære utsagn til grupper bestående av ikke-elementære utsagn, på den «optimale normalformen».

Optimal normalform innebærer at alle grupper er på Boyce-Codd-normalform[22] og at antall grupper er minst mulig. Presentasjoner på denne formen oppleves av mange brukere som oversiktlig og behagelig, kanskje spesielt fordi den har færrest mulig grupper uten at noen av gruppene inneholder repeterende attributtgrupper (1NF).

Det finnes flere måter å komme frem til den optimale normalformen. Dette er illustrert i figur 5.4[36, side 147].



Figur 5.4: Ulike veier til grupper

Pilen til høyre i figur 5.4 illustrerer normalisering. Dette innebærer å samle alle attributter i en gruppe (universalrelasjonen), og normalisere denne gruppen, dvs benytte normaliseringsteori for å splitte den opp i flere mindre grupper.

De tre pilene i midten illustrerer «intuisjon», som kan sammenlignes med ER's måte å modellere. Man danner grupper og primærnøkler under modelleringen. Når man har kommet frem til en gruppert modell er det mulig å kontrollere modellen og justere den ved hjelp av gruppering eller normalisering for å komme frem til den optimale normalformen.

«Analyse» innebærer å beskrive interesseområdet ved hjelp av elementære utsagn. «Gruppering» er å transformere de elementære utsagnene til grupper på den optimale normalformen.

Det finnes to typer gruppering:

Svak gruppering
: Gruppering til grupper som tillater attributter uten verdi.

Sterk gruppering
: Gruppering til grupper som ikke tillater attributter uten verdi.

5.3.1 Algoritme for gruppering

Målet her å gruppere til en presentasjon på optimal normalform. Dette er imidlertid kun en av mange mulige presentasjoner. Algoritmen er basert på algoritmen beskrevet i "Dataorientert systemutvikling"[36, side 235] som grupperer til relasjoner i en relasjonsdatabase. For mitt formål er det imidlertid ikke nødvendig at alle begreper er representerbare.

Det finnes flere forutsetninger som må innfris før det er mulig å gruppere:

1. Ved svak gruppering må alle assosiasjoner som har interne entydighetsskranker som spenner over mer enn en rolle, ha et predikatnavn eller være gjenstand for begrepsdannelse. Ikke-påkrevd en-til-en assosiasjoner må ha et predikatnavn.

   Ved sterk gruppering må alle ikke-påbudte assosiasjoner ha et predikatnavn.

   De nevnte assosiasjonene gir opphav til nye grupper, og for at grupperingen skal gå automatisk, uten interaksjon fra brukeren, må de ha et navn. En assosiasjon kan få navn ved hjelp av predikatnavn eller ved hjelp av begrepsdannelse.

   Det er mulig å forbigå dette kravet ved at applikasjonen genererer navn der dette mangler. Ingen navnekonvensjon kan dog erstatte brukerens forståelse av modellen. Det er derfor å foretrekke at brukeren navngir disse assosiasjonene.

2. Modellen må ikke inneholde noen synonyme brotyper.

   En synonym bro er en bro der entydighetsskranken står på representasjonssiden av assosiasjonen.

Dette problemet kan unngås ved å nekte brukeren å tegne denne konstruksjonen. Hvis dette viser seg å være for strengt kan problemet håndteres bak kulissene ved å la synonyme broer være en del av presentasjon, og forekomme i modellen som den foreslåtte omskrivingen av synonyme broer i [36, side 236]. Dette innebærer at det blir lagd et nytt begrep bak kulissene. Begrepet vil senere bli gjenstand for gruppering og det må derfor genereres et naturlig navn ved opprettelse.

3. Unære utsagn må være omformet til binære utsagn med et boolsk begrep.

   Dette kan også håndteres ved at unære utsagn bare er en presentasjon og blir lagret i modellen som binære utsagn.

Navning under gruppering

Under grupperingen får attributter og grupper navn.

- Gruppene får navn etter begrepet/assosiasjonen som var opphavet.
- Attributter som inngår i identifikator får navn etter betegnelsen på representasjonen.
- Resterende attributter tar navn fra korollen. Hvis dette rollenavnet ikke er utfylt, brukes begreps-/representasjons-navnet fra det inngrupperte begrepet/representasjonen.

Det finnes to konvensjoner for navngivning av roller under modelleringen (jf. [36]): verbaler eller substantiver. Verbalene kan brukes til å danne setninger som ligger nært opp til naturlig språk under modelleringen. Substativene ligger nærmest de attributtnavnene det er naturlig å velge i et databaseskjema, og vil også være informasjonsbærende etter gruppering.

Siste punkt i listen over forutsetter at substantivkonvensjonen blir brukt.

5.3.2 Gruppering

Grupperingen blir utført for et begrep om gangen, og hvert begrep gir opphav til en ny gruppe. For å unngå repeterende attributtgrupper

(1NF), grupperer vi bare inn begreper og representasjoner som er entydige sett fra begrepet som grupperes.

Alle koroller som grupperes inn i den nye gruppen blir fremmednøkler/pekere til identifikator i gruppen som har opphav i begrepet som spiller korollen.

Alt som identifiserer begrepet, dvs påbudte en-til-en assosiasjoner og påbudte en-til-mange assosiasjoner som knyttes sammen av en ekstern entydighetsskranke på korollene, grupperes inn og blir kandidater til identifikator i den nye gruppen.

Hvis begrepet som grupperes allerede har en representasjon, vil denne bli brukt som identifikator for gruppen. Hvis ikke blir en av kandidatene brukt som identifikator. De resterende kandidatene blir "kandidatnøkler".

Påbudt en-til-mange assosiasjoner der korollene ikke er beskranket av noen ekstern entydighetsskranke, grupperes inn på den entydige siden.

Mange-til-mange assosiasjonene som ikke er blitt gjenstand for begrepsdannelse danner egne grupper, og tar navn fra predikatnavnet.

Ikke-påbudt en-til-en assosiasjoner kan grupperes inn på valgfri side eller danne en egen gruppe. Jeg foretrekker det siste for å styre unna symmetribrudd. Dette gjør også at disse assosiasjonen behandles likt for svak og sterk gruppering. Den nye gruppen får navn fra predikatnavnet.

Alt som til nå er beskrevet er felles for svak og sterk gruppering. Resten av algoritmen er spesifikk for den valgte grupperingsformen.

Svak gruppering

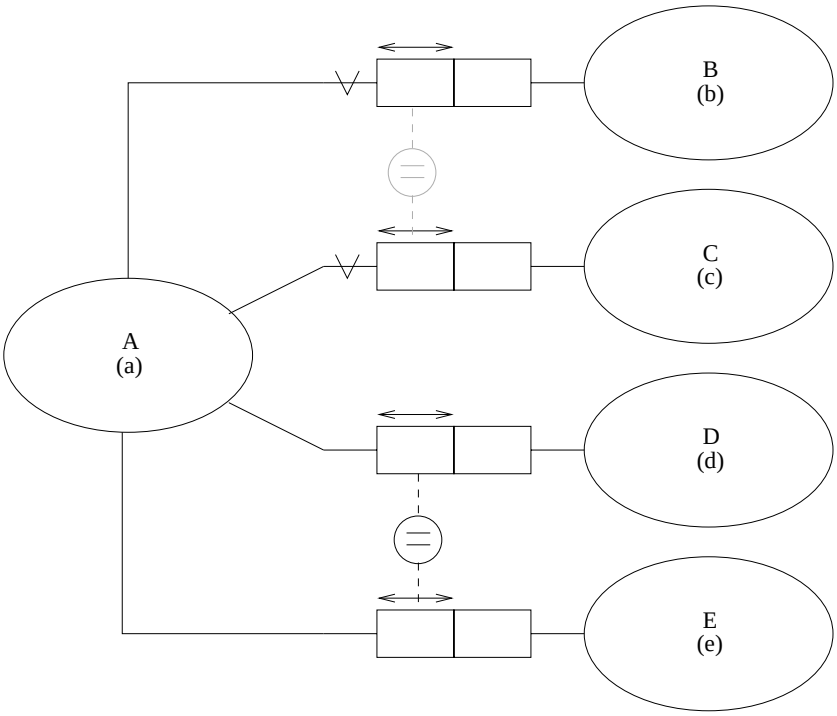
Ikke-påbudte en-til-mange assosiasjoner grupperes inn og blir attributter som kan være uten verdi.

Sterk gruppering

Sterk gruppering oppnås ved å gruppere inn attributter med hensyn på likhet. Dette innebærer at man lager en gruppe for hver likhetsskranke som befinner seg mellom de entydige rollene som er tilknyttet begrepet som grupperes.

Det er en implisert en likhetsskranke mellom mellom alle påkrevde roller som er tilknyttet begrepet som grupperes. Dette er vist i figur 5.5 på neste side. Den grå likhetsskranken er implisert.

Sterk gruppering av denne figuren vil resultere i to grupper: [a, b, c] og [a, d, e] (identifikator er understreket).



Figur 5.5: Implisert likhetsskranke

Assosiasjonene som ikke blir gruppert inn noen steder danner egne grupper.

5.3.3 Subtype

Det er mulig å behandle subtyper på tre forskjellige måter (jf. [36]):

Separasjon
: Subtype og supertype grupperes hver for seg. Identifikator i subtype-gruppen blir en fremmednøkkel/peker til identifikator i supertype-gruppen. Dette gir to ulike grupper, der subtype-gruppen består av en fremmednøkkel/peker til supertype-gruppen, og eventuelle andre attributter som grupperes inn i subtype-gruppen.

Absorpsjon
: Slå sammen subtype og supertype, og tillat attributter uten verdi i gruppen. Denne er med andre ord uaktuell i forbindelse med sterk gruppering.

Partisjonering

Grupper supertypen og subtypen hver for seg. La så subtype-gruppen arve alle attributtene fra supertype-gruppen. Dette vil resultere i to ulike grupper med en ulikhetsskranke mellom identifikatorene.

5.3.4 Undertrykking

Siste steg av grupperingen er undertrykking. Både grupper og attributter kan undertrykkes.

Undertrykking av grupper

Ofte vil man sitte igjen med grupper som ikke inneholder noe annet en identifikatoren. Disse kan undertrykkes. I noen tilfeller kan det dog være ønskelig å beholde en slik gruppe fordi den kan fungere som en dynamisk verdiskranke, pga referanseintegritet mellom denne gruppen og de refererende gruppene.

I andre tilfeller gir det ingen mening å beskranke verdiene som refererer til denne gruppen. Eksempler på dette kan være **lengde**, **vekt**, **beløp** og **dag**.

Undertrykking av attributter

Hvis det finnes verdilikhet mellom to eller flere attributter i samme gruppe, skyldes dette at det finnes ekvivalente stier i modellen. De overflødige attributtene kan da undertrykkes.

For å finne ut at det er verdilikhet mellom attributter kreves det at de ekvivalente stiene noteres i modellen. Jeg anser dette som utenfor rammene til denne oppgaven, og overlater undertrykking av attributter til brukeren.

Kapittel 6

ORM metamodell

For å lage en XML-mal for å uttrykke ORM ved hjelp av XML, er det helt nødvendig å ha en presis beskrivelse av ORM, dvs en metamodell.

> *A metamodel is a precise definition of the constructs and rules needed for creating semantic models.[20]*

Metamodellen som ligger til grunn for applikasjonen, er inspirert av Terry Halpins metamodell publisert i InConcept[27].

Metamodellen er fordelt på tre figurer.

6.1 En nøytral modell bestående av binære elementære utsagn

Jeg har valgt å bruke kun binære elementære utsagn i modellen. Dette gjør at metamodellen kan forenkles.

Begrepsdannelser kan skrives som binære elementære utsagn (se kapittel 5.1 på side 29). Jeg har derfor valgt å se på begrepsdannelser utelukkende som en presentasjon av disse binære elementære utsagnene. Begrepsdannelser er derfor ikke omtalt i metamodellen.

Alle ikke-binære elementære utsagn kan også skrives om til semantisk ekvivalente binære elementære utsagn[36, side 81], og kan derfor betraktes som presentasjoner av binære elementære utsagn. Konsekvensen av dette er at metamodellen slipper å forholde seg til n-ære utsagn.

6.2 «Begreper og representasjoner»

Figur 6.1 på neste side tar for seg **begreper**, **representasjoner** og **subtyper**.

`ObjectType` representeres ved hjelp av et navn, og er en «generalisering» av begrep (`EntityType`) og representasjon (`ValueType`). Dette impliserer at begreper og representasjoner har felles "navnerom"; et begrep kan ikke ha samme navn som en representasjon. [1]

`OTKind` fungerer som diskriminator for `ObjectType` og forteller om `ObjectType` er av typen `EntityType` eller `ValueType`. Dette er uttrykt i subtypedefinisjonene i firkanten nederst til venstre.

Et begrep kan kun ha en representasjon, men begrepets representasjon kan være en kombinasjon av flere representasjoner. Dette er uttrykt ved hjelp av assosiasjonen med predikatnavn `Representation`.

`ValueType` er tilknyttet `DataType` som forteller hvilken datatype representasjonen har: tekstlig eller numerisk.

`EntityType` og `ValueType` har begge en assosiasjon knyttet til seg selv som beskriver at et begrep kan være subtype av et annet begrep (`EntitySubtype`), og at en representasjon kan være subtype av en annen representasjon (`ValueSubtype`).

Selv om tradisjonell ORM ikke tillater representasjoner som er subtyper av representasjoner, har jeg tatt med dette for å åpne for en fremtidig utvidelse.

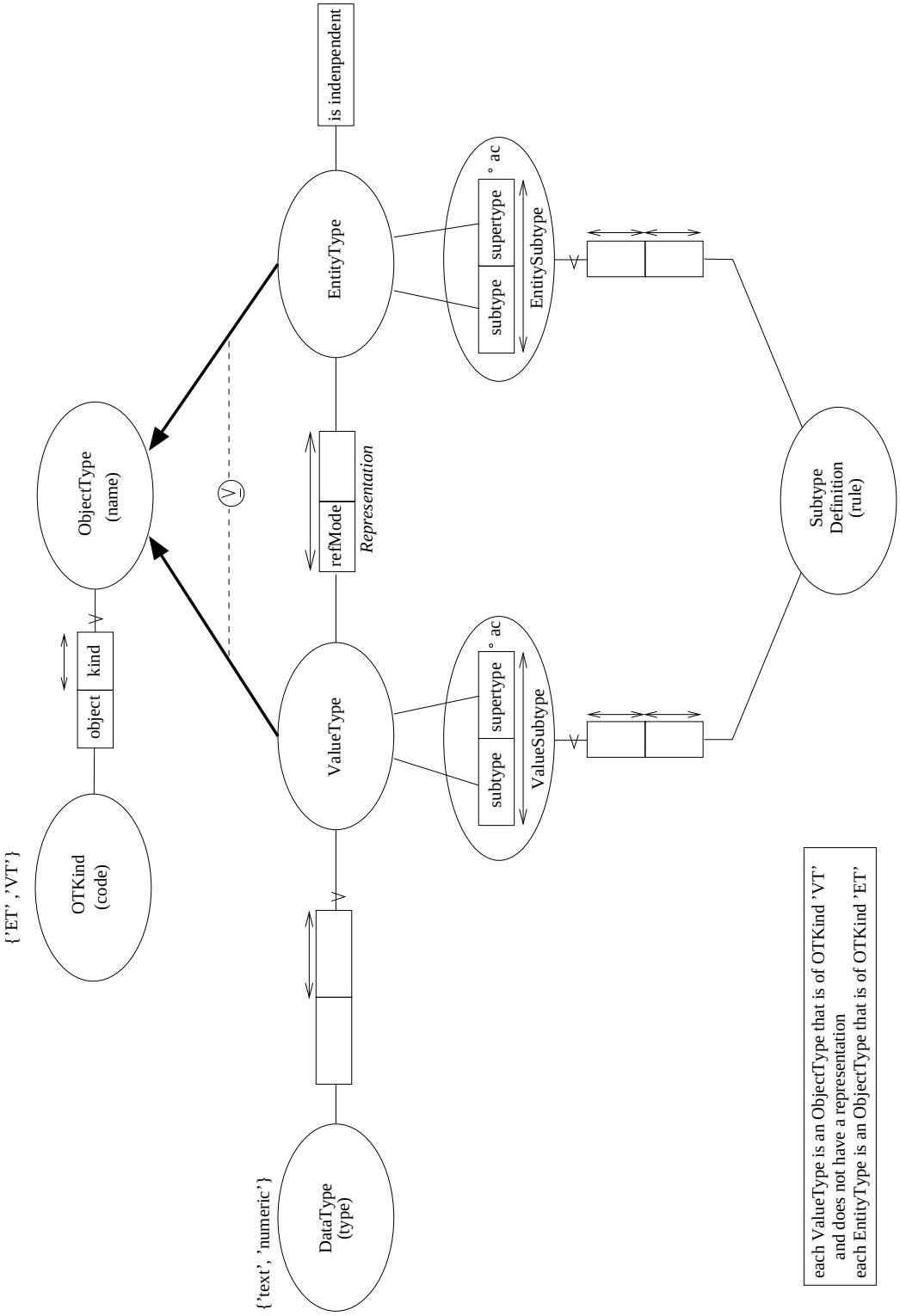
`is independent` er en unær assosiasjon tilknyttet `EntityType`. Den brukes for å uttrykke at et begrep som etter gruppering består av kun identifikator, ikke skal undertrykkes, men beholdes for å pålegge referanseintegritet/delmengdeskranke mellom gruppen som begrepet er opphav til og refererende attributter. I relasjonsdatabasekontekst betyr dette at selv om begrepet blir en relasjon med kun primærnøkkel skal ikke relasjonen undertrykkes.

Subtype assosiasjonene er beskranket med **asykliske** ringskranker for å unngå sykler i subtype-grafen. Multippel arv er kun lov hvis supertypene, direkte eller indirekte, har felles absolutt supertype. Dette er ikke uttrykt i figuren. [2]

`SubtypeDefinition` er representert av `rule`. Denne figuren har to eksempler på slike regler i firkanten nede til venstre. Siden det ikke

[1] Av praktiske årsaker avviker representasjonen til ObjectType i realiseringen fra metamodellen (se kapittel 7.2.1 på side 55 og kapittel 7.2.3 på side 57)

[2] Flere skranker mangler grafisk notasjon og er derfor utelatt fra den grafiske fremstillingen av metamodellen.



Figur 6.1: Begreper og representasjoner

gir noen mening med flere like regler, må reglene være unike innenfor modellen.

Det er også en "skranke" på subtype-strekene til `ET` og `VT` ($\overline{\vee}$). Dette er en gjensidig utelukkende skranke som sier at `ObjectType` må være enten `EntityType` eller `ValueType`. Dette er avledet fra subtypedefinisjonen og verdiskranken på diskriminatoren, og er redundant informasjon. "Skranken" er kun tatt med for å tydeliggjøre modellen.

$\overline{\vee}$-symbolet er forøvrig en kombinasjon av $\vee$ og $\neq$. Jeg betrakter det derfor som en del av presentasjonen og vil ikke omtale det i metamodellen.

6.3 «Assosiasjoner»

Figur 6.2 på neste side omhandler assosiasjoner og roller, og deres forhold til begreper og representasjoner.

`Role` kan ikke bli representert ved hjelp av rollenavn fordi det er tillatt med roller uten navn og rollenavn som ikke er unike for modellen. `Role` blir derfor representert ved hjelp av `nr`, og kan ha et `RoleName`, som må være unikt innen i assosiasjonen. `Role` må være tilknyttet en `ObjectType` (fra figur 6.1 på forrige side), og er enten første eller andre rolle i en `relationship`, men aldri begge deler.

`PredicateName` brukes for å beskrive hele assosiasjonen. Dette er en utvidelse av ORM som er tatt med for å få en nøytral modell.

6.4 «Skranker»

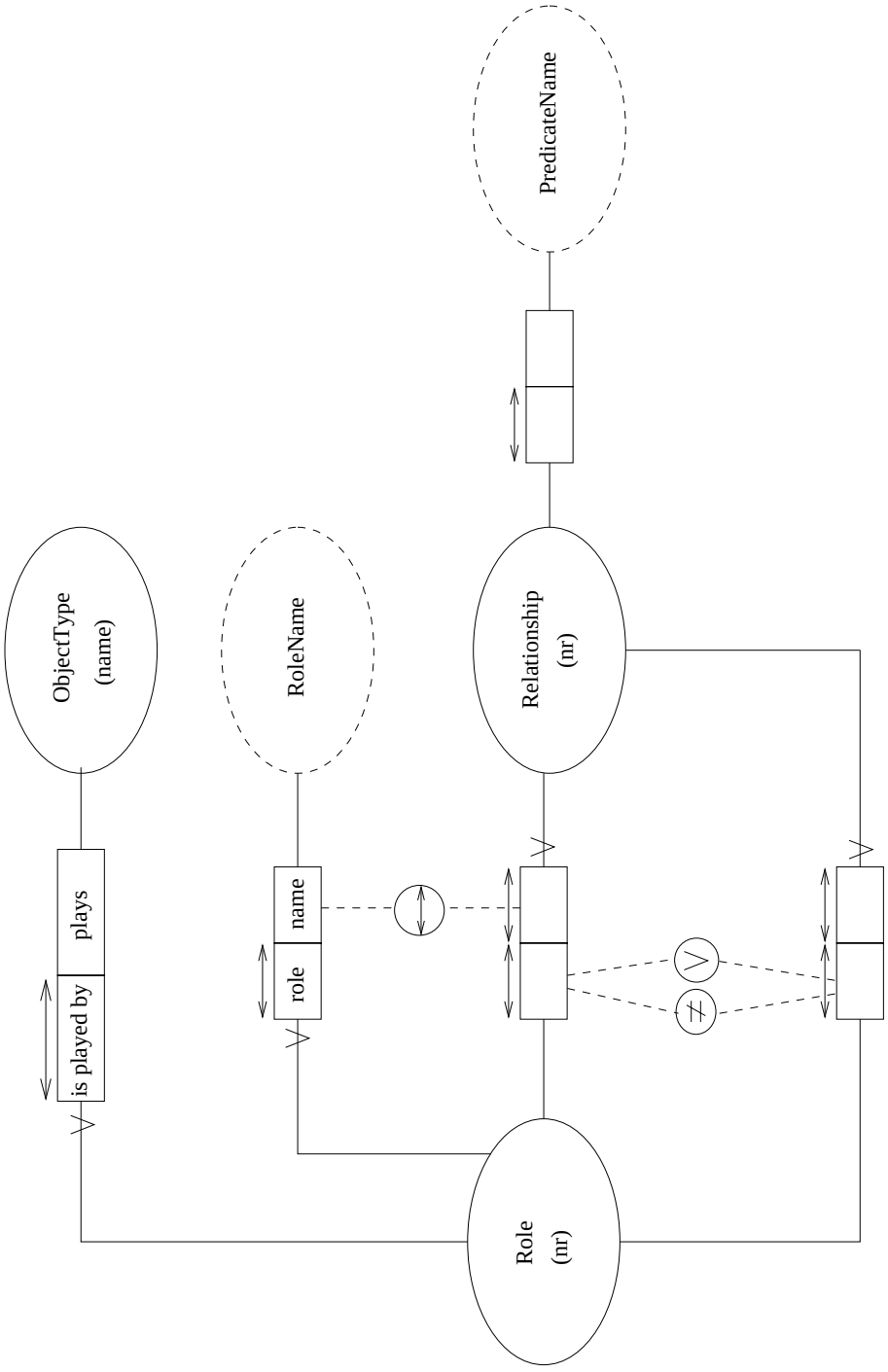
Figur 6.3 på side 49 inneholder en kategorisering av skrankene, og deres tilknytning til roller, assosiasjoner, representasjoner og begreper.

`Constraint` er generaliseringen av alle typer skranker og representeres ved hjelp av `nr`.

Subtypedefinisjonen sier at `CKind` er diskriminator for `Constraint` og forteller hvilken subtype de ulike skrankene tilhører.

`RoleConstraint` er en subtype av `Constraint` og er knyttet til en `RoleGroup`, som kan inneholde en eller flere (0 gir ingen mening) `Role` (hentet fra figur 6.2 på neste side). Subtypedefinisjonen forteller at følgende skranker er inneholdt i denne subtypen: intern entydighet, ekstern entydighet, intern påbudt rolle og ekstern påbudt rolle.

Jeg har tidligere sett på frekvensskranken funksjonsmessig som en generalisering av entydighetsskranken. I metamodellen vil dette lede



Figur 6.2: Assosiasjoner

til et nytt begrep, **UniqueConstraint** som blir en subtype av **Frequency**. Det nye begrepet vil ikke være tilknyttet noen andre begreper. For å unngå at det lages en subtype som ikke er med i noen utsagn, har jeg i metamodellen valgt å se på frekvensskranken som en spesialisering av entydighetsskranken; en frekvensskranke er en entydighetsskranke der frekvensen kan spesifiseres.

`Frequency` blir da en spesialisering av `RoleConstraint` og har, i tillegg til kobling mot `RoleGroup`, en mange-til-mange assosiasjon mot `ValueNumSpec`. `ValueNumSpec` blir brukt til å angi en verdi, et intervall med nedre og valgfri øvre grense, eller en kombinasjon av disse. Den nedre grensen må alltid være oppgitt og må være 1 eller høyere (det gir ingen mening i å si at noe skal forekomme 0 eller et negativt antall ganger). Dette er imidlertid ikke vist i figuren.

Mengdeskrankene (`SetConstraint`) er binære operatorer, i motsetning til skrankene nevnt tidligere som er unære, og er derfor tilknyttet enda en `RoleGroup`.

Antall roller i `RoleGroup` fra `RoleConstraint` må være likt antall roller i `RoleGroup` fra `SetConstraints`. Dette er ikke vist i figuren.

`SubType` er en subtype av `Constraint` og fremstilles bare delvis i figure 6.3 på side 49 fordi den allerede er beskrevet i figur 6.1 på side 40.

Det finnes to typer verdiskranker (`ValueConstraints`): Verdiskranken (`ValueConstraint`) og Standardverdi-skranken (`DefaultValue`)

`ValueConstraint` kan være knyttet til en eller flere `ValueSpec`. `ValueSpec` blir brukt til å angi verdier, intervaller eller kombinasjoner av disse. Verdiene kan være både numeriske og tekstlige.

`DefaultValue` tillater kun en verdi (tekstlig eller numerisk).

En ringskranke (`RingConstraint`) er knyttet til en assosiasjon[3] (`Relationship` fra figur 6.2 på forrige side). Ringskranker kan kun knyttes til assosiasjoner som er fra og til samme begrep. Dette er ikke vist i figuren.

6.5 Andre metamodeller

Jeg har funnet to andre metamodeller for ORM. Begge er laget av Terry Halpin. Den ene finnes i “Conceptual Schema & Relational Database Design”[31]. Den andre er publisert i InConcept[27].

Min metamodell er inspirert av sistnevnte. “Object-Role Modeling:

[3]Dette skyldes at det bare finnes binære assosiasjoner

an overview"[28] gir en innføring i notasjonen som brukes i denne modellen.

Jeg har tilpasset Halpins modell der jeg mener den er unødvendig kompleks for mitt tilfelle, og rettet ting der jeg mener den inneholder feil.

6.5.1 «Begreper, representasjoner og assosiasjoner»

Jeg vil her sammenligne min metamodell figur 6.1 på side 40 med Halpins tilsvarende metamodell figur 6.4 på side 50.

Begrepsdannelse

Siden jeg ikke tillater begrepsdannelser i modellen, er assosiasjonen mellom `EntityType` og `Relationship`, selve begrepsdannelses-assosiasjonen, i denne figuren fjernet. Dette medfører at subtypedefinisjonen og subtypene til `EntityType` (`NestedEntityType` og `UnnestedEntityType`) også kan fjernes.

Fordi begrepsdannelser ikke finnes i modellen, men er konstruksjoner som kan vises som begrepsdannelser i visualiseringen (se figur 5.2 på side 31), vil det være mulig med begrepsdannelser som er sub- og supertyper i presentasjonen. Jeg vil ikke gjøre noe for å begrense disse mulighetene, men det kan diskuteres hvor nyttige disse konstruksjonene er.

Subtype

Ifølge Halpins metamodell kan representasjoner være både subtyper og supertyper. Dette er antagelig ikke tilsiktet. Jeg har allikevel beholdt denne muligheten for fremtidige utvidelser.

Det skal imidlertid ikke være lov å blande representasjoner og begreper i subtyper. For å fjerne denne muligheten har jeg flyttet subtypeassosiasjonen fra `ObjectType` ned til `EntityType` og `ValueType`. Representasjoner får dermed en assosiasjon til seg selv, og begreper får en assosiasjon til seg selv.

For å unngå at det finnes sykler i subtype-grafene har jeg lagt til en asyklisk ringskranke på hver av disse assosiasjonene. Som nevnt tidligere er fremdeles ikke alle skranker for subtype-assosiasjonen visualisert i modellen.

is independent

Figur 6.4 på side 50 har knyttet denne unære assosiasjonen til `ObjectType`, noe som impliserer at det er mulig å la en `ValueType` ha dette attributtet.

Jeg har flyttet dette «is independent» til `EntityType` i min metamodell.

Description - Modell eller presentasjon

`Description` kan sammenlignes med *Note* i UML. Det kan diskuteres om den tilhører modellen eller presentasjonen. Hvis man sier at modellen bare inneholder det som trengs for å lage en semantisk lik modell, og at semantikken er begrenset av det applikasjonen «forstår», så vil `Description` tilhøre presentasjonen.

På den andre siden kan det være naturlig å inkludere `Description` i modellen hvis beskrivelsen inneholder semantikk som applikasjonen ikke er i stand til å forstå.

Jeg har valgt å følge den første retningslinjen. Grensene for hva som er semantikk er dermed bestemt av hva som er implementert i applikasjonen. Dette medfører at `Description` tilhører presentasjonen og kan fjernes fra metamodellen.

Relationship

For at `DerivedRelationship` eller `VirtualFactType` (som ikke er tatt med i Halpins metamodell) skal gi noen mening for verktøyet, må jeg ha et eget språk for å definere regler for disse. Uten dette språket blir `DerivedRelationship` og `VirtualFactType` en del av presentasjonen, og vil dermed ikke kunne brukes til noe under gruppering til f.eks. relasjonsdatabaseskjema. I mangel av dette språket har jeg valgt å utelate både `DerivedRelationship` og `VirtualFactType`.

Ettersom det nå ikke finnes noen subtyper av `Relationship`, kan diskriminatoren `is derived` og den tilhørende subtypedefinisjonen fjernes.

Alle assosiasjonene som knytter `Relationship` til resten av modellen i denne figuren er nå fjernet. Jeg har derfor flyttet `Relationship` til neste figur.

DataType

ORM tillater to datatyper for representasjoner (`ValueType`): Numeriske og ikke-numeriske. Halpin har utelatt dette attributtet for representasjoner i sin metamodell selv om han bruker numerisk datatype i figur 6.6 på side 52.

6.5.2 «Navning»

I figur 6.5 på side 51 har jeg også fjernet mye fordi jeg kun tillater binære utsagn i min metamodell figur 6.2 på side 42.

Assosiasjon

`Relationship` er i denne figuren knyttet til `RelationshipReading` og `Arity`. Assosiasjonen mot `Arity` er avledet, og kan finnes ved å telle antall roller et gitt `Relationship` er knyttet mot. Siden alle assosiasjonene i modellen har to roller har jeg valgt å gjerne `Arity`.

`RelationshipReading` er avledet av `PredicateText` og brukes for å lage naturlig språk av n-ære utsagn. Dette er setninger med "hull" i, f.eks.

- ... kjøpte ... på dato ...

Min metamodell bruker kun binære assosiasjoner, og jeg har derfor fjernet `PredicateText` og `RelationshipReading`.

Roller

I Halpins modell er det tillatt med n-ære assosiasjoner, og for at `RelationshipReading` skal gi mening, må rollene ha en bestemt rekkefølge.

Jeg har valgt å uttrykke at en `Relationship` består av to roller ved hjelp av to assosiasjoner, som begge er påbudt fra `Relationship` til `Role`. Jeg kan dermed fjerne `Position`.

PredicateText

Jeg er av den oppfatning at det ikke er tillatt med assosiasjoner mellom to representasjoner. Jeg stiller meg derfor tvilende til at representasjonen `PredicateText` er tilknyttet en unær assosiasjon. `PredicateText` er imidlertid fjernet fra min metamodell.

6.5.3 «Skranker»

Halpins metamodell figur 6.6 på side 52 vil her bli sammenlignet med min metamodell figur 6.3 på side 49.

Frekvensskranke

Halpins metamodell tillater tekstlige verdier og intervaller for frekvensskranken, noe som ikke gir mening. I min metamodell tillater jeg kun numeriske verdier og intervaller for frekvensskranken.

Verdiskranke

Verdiskranken i Halpins metamodell er knyttet til `ObjectType`. Jeg mener dette er galt, og at den skal være knyttet til `ValueType`.

Primærnøkkel

`UC` (entydighetsskranke) er en subtype av `RoleConstraint`. Den har en unær assosiasjon (`is primary`) som beskriver at denne entydighetsskranken skal gi opphav til en primærnøkkel.

Jeg synes det er galt å bringe inn primærnøkler under modelleringen, og har derfor ikke tatt den med i min metamodell.

Mengdeskranker

Disse skrankene har i tillegg til rollene arvet fra `RoleConstraint`, et `ArgLength` attributt som forteller hvordan rollene skal deles inn i to grupper.

Jeg har valgt å gjøre dette ved å la mengdeskrankene ha en egen assosiasjon til `RoleGroup`.

Ringskranker

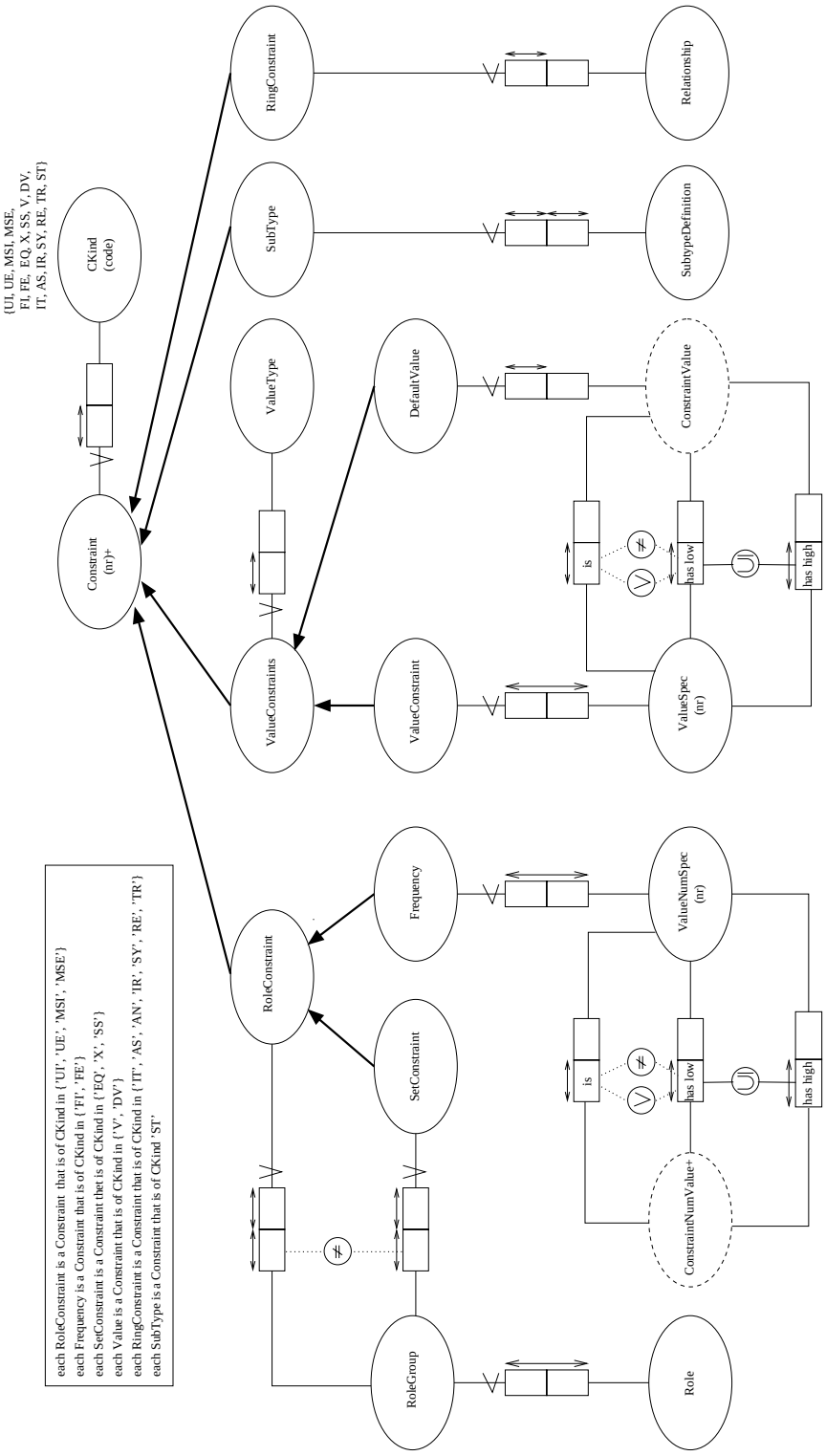
Ringskrankene blir i denne figuren behandlet som rolleskranker. Dette skyldes at Halpin tillater n-ære assosiasjoner. Hvis man da har en assosiasjon med 4 roller, tilknyttet to begreper, er det mulig å ha to forskjellige ringskranker på de to rolleparene.

Siden jeg kun tillater binære utsagn, kan ringskrankene knyttes til en assosiasjon istedenfor rollepar.

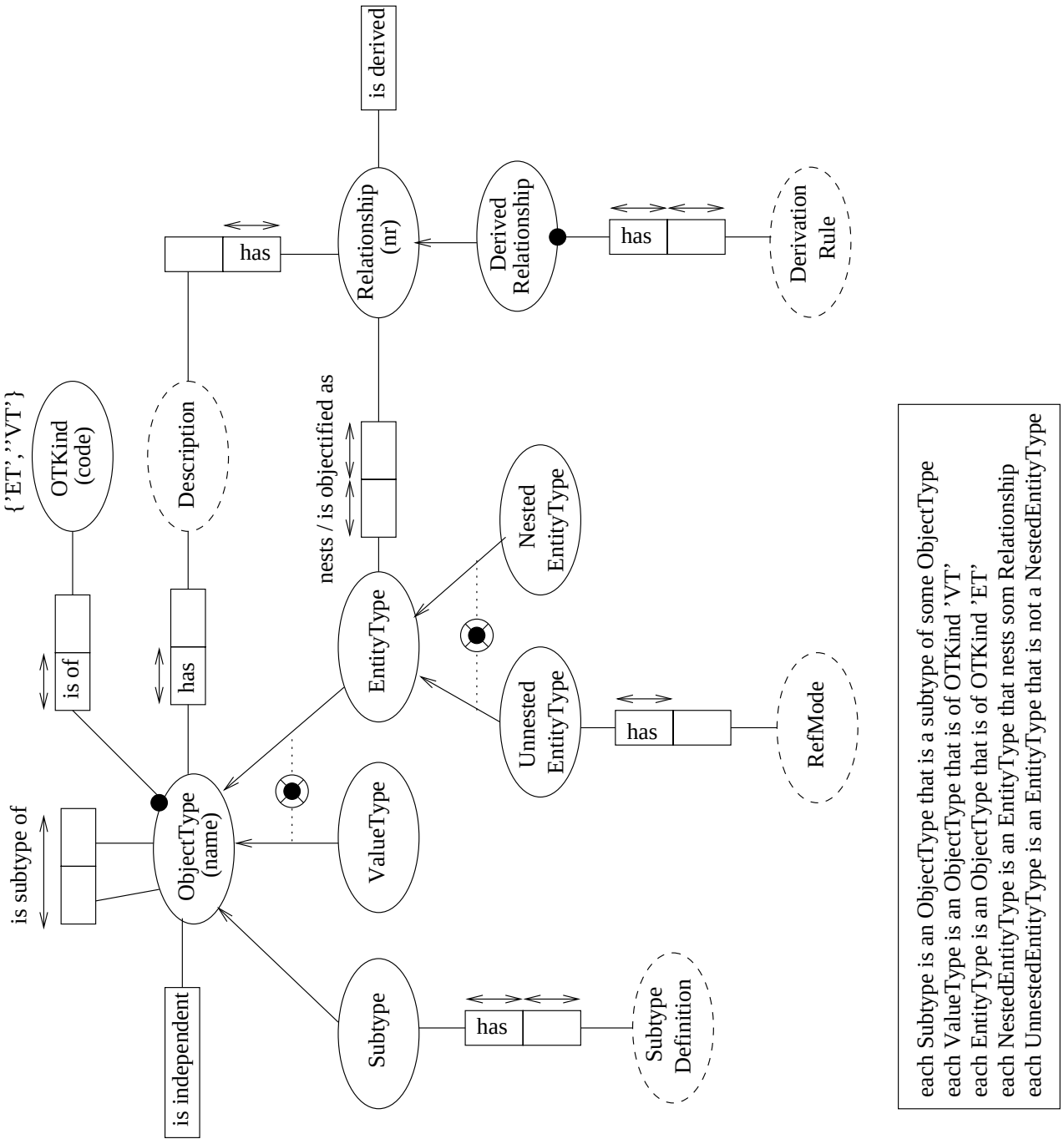
Halpin har tatt med asymmetrisk-ringskranke. Dette er en kombinasjon av antisymmetrisk- og irrefleksiv-ringskranke, og er derfor fjernet fra min metamodell

Mandatory Disjunctive

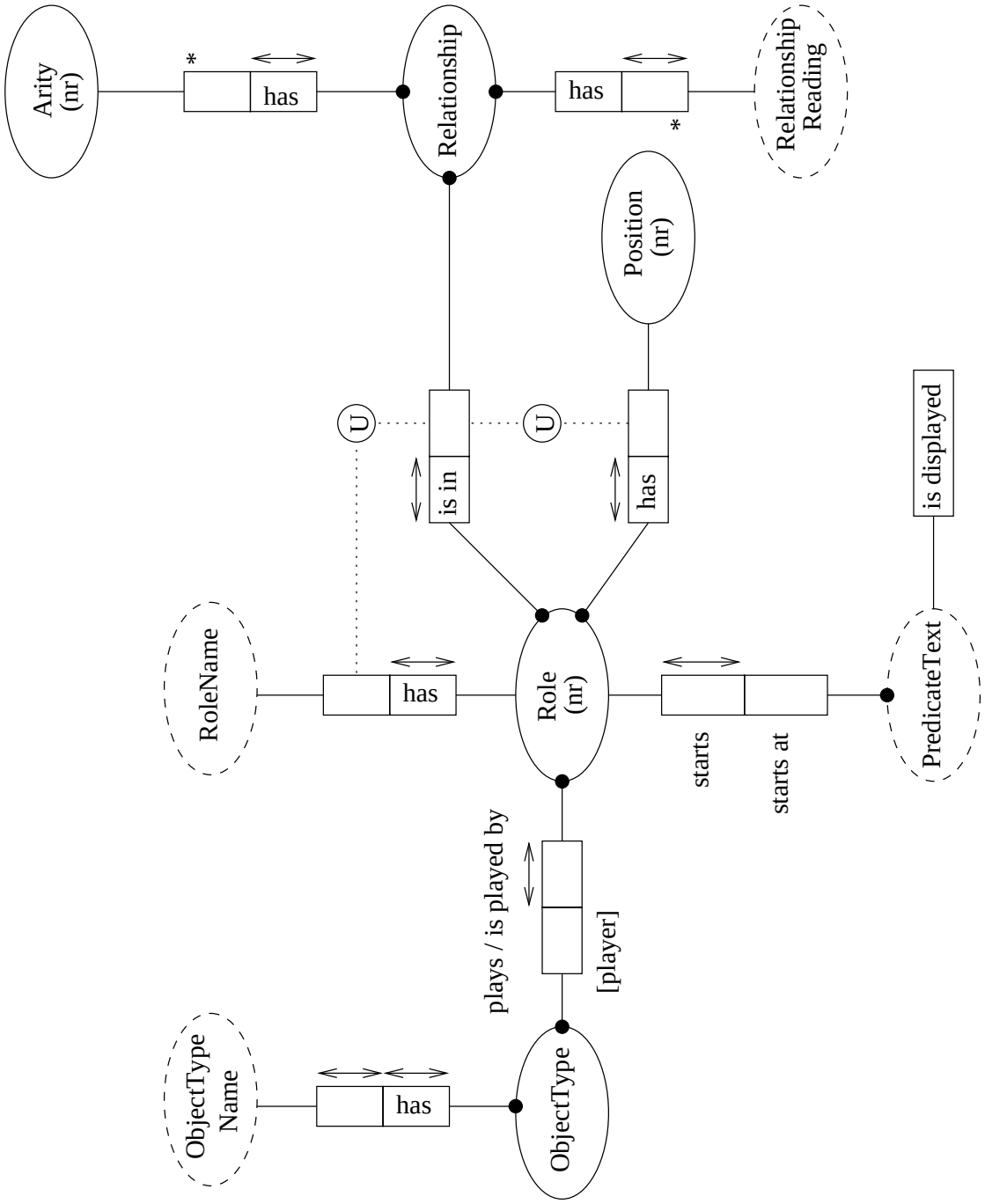
Terry Halpin har tatt med Mandatory Disjunctive ($\overline{\vee}$) i metamodellen. Dette er egentlig en skranke satt sammen av ulikhet ($\neq$) og påbudt rolle ($\vee$). Den tilhører derfor presentasjonen og er utelatt fra min metamodell.



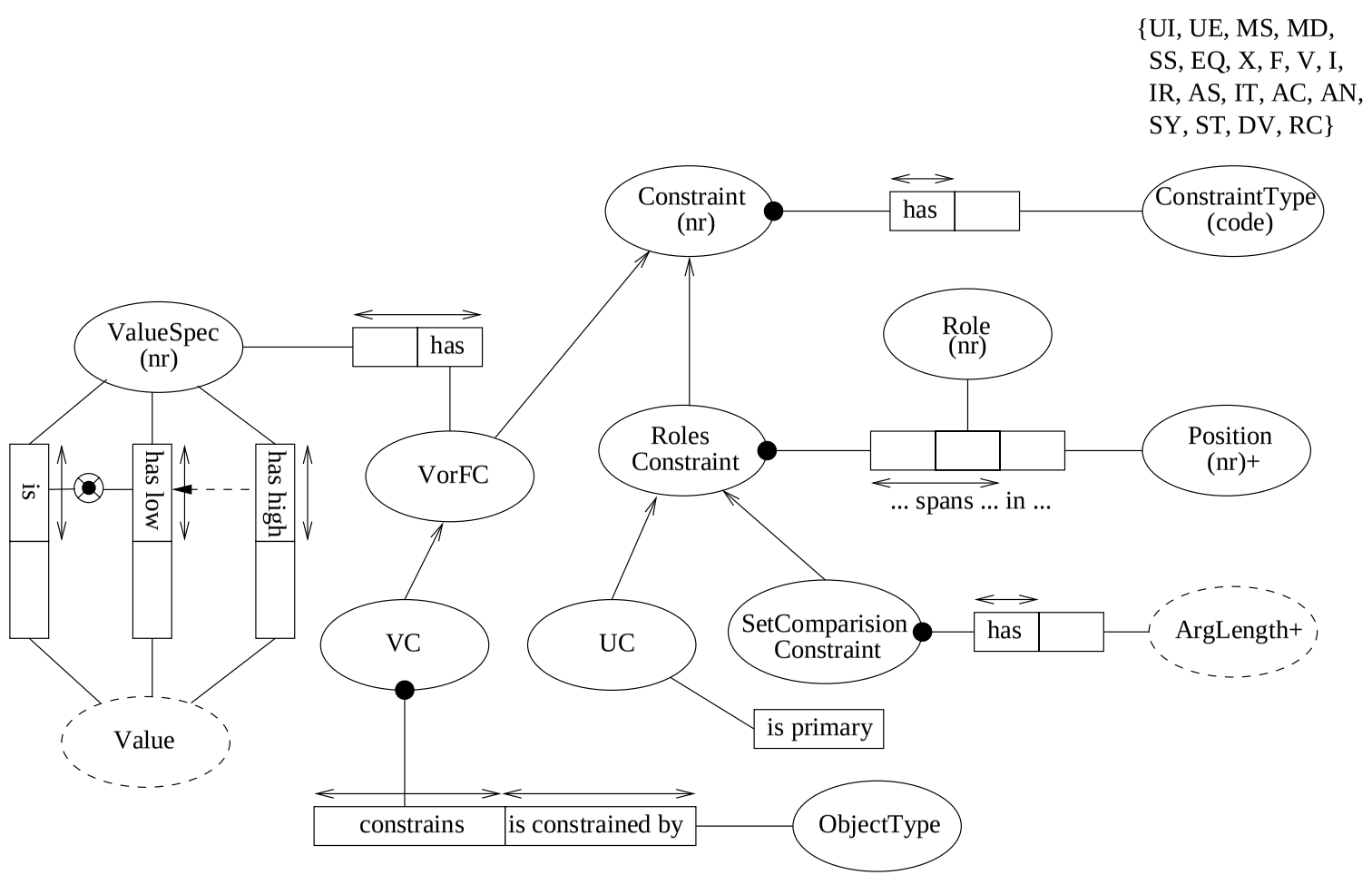
Figur 6.3: Skranker



Figur 6.4: Begreper, representasjoner og assosiasjoner



Figur 6.5: Navning av begreper, representasjoner, assosiasjoner og roller



Figur 6.6: Skranker

each SetComparisionConstraint is a Constraint that is of ConstraintType in {'SS', 'EQ', 'X'}
each UC is a Constraint that is of ConstraintType in {'UI', 'UE'}
each VC is a Constraint that is of ConstraintType 'V'
each VorFC is a Constraint that is of ConstraintType in {'V', 'F'}
each RolesConstraint is a Constraint that is of ConstraintType in
{'UI', 'UE', 'MS', 'MD', 'SS', 'EQ', 'X', 'F', '
'I', 'IR', 'AS', 'IT', 'AC', 'AN', 'SY', 'RC', 'DV'}

Kapittel 7

XML-representasjon av modellen

Dette kapittelet omhandler gruppering av metamodellen til en XML-mal.

Jeg vil oppgi både DTD og eksempel på bruk av DTD'en for de ulike elementene.

7.1 Gruppering av metamodell til XML

"Object Role Modeling and XML-Schema"[30] omhandler gruppering fra ORM til XML. Metoden grupperer til XML Schema, og bygger på endel skjønn. Jeg ønsker, i tillegg til å oversette fra ORM til DTD, å tilrettelegge mest mulig for applikasjonen og har derfor valgt å gruppere uavhengig av denne metoden.

DTD tilbyr datatypen ID for å representere et element ved hjelp av en unik verdi, og IDREF og IDREFS for å referere til en eller flere elementer som er representert ved hjelp av en slik ID.

IDREF og IDREFS kan peke på respektive en og vilkårlig mange ID'er. Baksiden av medaljen er at disse pekerene ikke er typede, og kan peke på alle mulige ID'er uavhengig av hvilket element som inneholder ID'en. Dette resulterer i at en rolle som bare skal kunne peke på et begrep eller en representasjon, i realiteten kan peke på f.eks. en annen rolle.

7.1.1 Navnekonvensjon for ID'er

For å øke lesbarheten i XML-koden ønsker jeg å dele ID'ene inn i ulike navnerom. XML støtter navnerom, men bare i elementnavn, ikke i attributter[15]. Jeg ønsker derfor å "simulere" navnerom ved å bruke elementnavnet som prefiks for alle ID'er, etterfulgt av separatoren '_' og et generert nummer.

I utgangspunktet hadde jeg tenkt at den numeriske delen av ID'ene skulle være en stigende sekvens av heltall, men siden dette valget kun var estetisk begrunnet[1] og ville medført unødvendig mye programkode, ble et finkornet *timestamp* valgt til dette formålet. ID'ene vil da se slik ut: *elementnavn_timestamp*

7.1.2 Overordnet struktur

Alle XML-dokumenter må ha et rot-element. Siden denne strukturen skal beskrive ORM er det naturlig å kalle dette elementet for `orm`.

Det er et poeng for meg å skille mellom modell og presentasjon, og for å markere dette kunne jeg valgt å skille de ulike delene ut i hver sin fil (kanskje til og med en fil for hver presentasjon). Dette ville imidlertid skapt ekstra programkode for filbehandling. Siden jeg kun er ute etter å skape et «logisk» skille, nøyer jeg meg med å legge dem inn i `orm`-dokumentet som adskilte elementer.

Den overordnede strukturen blir da slik:

Overordnet struktur

```
<orm>
  <model>
    ..
  </model>
  <view>
    ..
  </view>
  <view>
    ..
  </view>
</orm>
```

`orm` tilsvarer her **Project** fra prinsippskissen (se kapittel 2.1 på side 5). Den inneholder en modell og en eller flere presentasjoner.

DTD for `orm`

[1] Noe XML tar "avstand" fra (punkt 10 på side 13).

```
<!ELEMENT orm ( model, view+ ) >
```

7.2 Modell

Innholdet i modell er delt inn under tre elementer:

`objects` inneholder begreper og representasjoner.

`relationships` inneholder assosiasjoner.

`constraints` inneholder skranker.

Jeg har valgt å dele elementene inn i kategorier på denne måten for å lettere kunne adressere de ulike delene ved hjelp av XPath og XSLT. De tre kategori-elementene vil da fungere som bokmerker.

DTD for `model`

```
<!ELEMENT model ( objects, relationships, constraints ) >
```

7.2.1 Representasjon

I metamodellen blir representasjoner representert ved hjelp av et navn. Dette betyr at hvis en representasjon skifter navn under modelleringen, må alle referanser til representasjonen oppdateres. Dette vil lede til unødvendig mye programkode. Jeg har derfor valgt å avvike fra metamodellen og løse dette ved å la representasjoner representeres ved hjelp av en ID i henhold til ID-navngivningskonvensjonen. Unike representasjonsnavn kan da ikke håndheves av DTD, fordi ID er den eneste datatypen som håndhever unikhet, og det er kun tillatt med en ID pr element. Denne unikheten må derfor håndheves på applikasjonsnivået.

ORM-metamodellen tilsier at representasjonenes verdier skal kunne være numeriske eller ikke numeriske (tekstlige). Dette beskrives ved hjelp av attributtet `datatype`, som kan ha to forskjellige verdier: *numeric* eller *text*, der *text* er standard (*default*). Denne løsningen gjør det enkelt å utvide med nye datatyper, f.eks. *date*, *timestamp*, *boolean* osv.

DTD for value

```
<!ELEMENT value EMPTY >
<!ATTLIST value
             id       ID             #REQUIRED
             name     CDATA          #REQUIRED
             datatype (numeric|text) "text" >
```

Eksempel

```
<value id       = "value_1074610026.817683"
       name     = "height"
       datatype = "numeric"/>
```

7.2.2 Assosiasjoner og roller

Rolle-elementet er kalt `role`, og identifiseres ved hjelp av en ID i henhold til konvensjonen. `role` knyttes til en representasjon eller et begrep ved hjelp av `object-ref`, og kan ha et navn (`name`). I metamodellen er navnet unikt pr assosiasjon. Dette kan ikke håndheves i DTD'en og må derfor gjøres på applikasjonsnivå.

Det er ønskelig at det bare skal være mulig å referere til begreper og representasjoner fra `role`. Som nevnt tidligere er dette ikke mulig å håndheve ved hjelp av en DTD, og må også håndheves i applikasjonen.

Assosiasjonselementet er kalt `relationship`. En assosiasjon inneholder to roller, som skissert i metamodellen, og representeres ved hjelp av en generert ID i tråd med konvensjonen.

Hvis den interne datastrukturen hadde vært en relasjonsdatabase istedenfor XML, ville det vært naturlig å identifisere en assosiasjon ved hjelp av en eller begge rolle-ID'ene. Dette er ikke like aktuelt i XML som ikke har typede pekere og ikke kan håndheve referanseintegritet og unikhet for samme element.

DTD for relationship *og* role (relationships *inkludert*)

```
<!ELEMENT relationships ( relationship* ) >

<!ELEMENT relationship (role,role) >
<!ATTLIST relationship
             id ID #REQUIRED >

<!ELEMENT  role EMPTY >
<!ATTLIST  role
             id         ID    #REQUIRED
             object-ref IDREF #REQUIRED
             name       CDATA #IMPLIED >
```

Eksempel

```
<relationships>
  <relationship id="relationship_1074611332.692673">
    <role id         = "role_1074611338.8361809"
          object-ref = "entity_1074611344.8865039"
          name       = "employer"/>
    <role id         = "role_1074611352.1894951"
          object-ref = "entity_1074611357.6818531"
          name       = "employee"/>
  </relationship>
</relationships>
```

7.2.3 Begrep og representasjon

Begreps-elementet heter `entity`, og representeres i metamodellen ved hjelp av et navn. Jeg har her bestemt meg for å avvike fra metamodellen på samme vis som med `value`. Dette betyr at `entity` representeres ved hjelp av en ID i tråd med navngivningskonvensjonen, og kan i tillegg ha et navn.

Hvis et begrep har en representasjon, må dette lagres i modellen. Jeg vil forklare hvordan jeg har valgt å gjøre dette ved hjelp av et eksempel på en representasjon ved perfekt bro:

Eksempel på representasjon ved perfekt bro

```
<entity id="entity_1074611785.4943449" name="A">
   <representation>
      <value-ref ref="value_1074611791.2858181"/>
      <criterion>
         <relationship-ref ref="relationship_1074611797.2358379"/>
         <mc-ref ref="mc_1074611802.9501071"/>
         <uc-ref ref="uc_1074611808.1362641"/>
         <uc-ref ref="uc_1074611813.4551339"/>
      </criterion>
   </representation>
</entity>
```

Et begrep kan ifølge metamodellen bare representeres ved hjelp av representasjoner. For å gjøre det lettere for applikasjonen å oppdatere "indirekte" representasjon under modelleringen, har jeg åpnet for at begreper kan representeres av andre begreper.

Eventuell `representation` legges inn under `entity`. Elementet inneholder referanser til alle begrepene og representasjonene (`entity-ref` og `value-ref`) som er med på å representere begrepet.

`criterion` inneholder kriteriene som ligger til grunn for at de nevnte begrepene/representasjonene skal utgjøre representasjon for begrepet. Dette er et avvik fra metamodellen og er tatt med for å gjøre det lettere for applikasjonen å se hvilke endringer som vil påvirke begrepets representasjon.

Kriteriene for representasjonen i eksempelet inneholder elementene `mc-ref` og `uc-ref` som er referanser til respektive «påbudt rolle»-skranke og entydighetsskranke. Disse skrankene vil bli gjennomgått i kapittel 7.2.4 på neste side.

Jeg har valgt å bruke referanser istedenfor kopiering/flytting av elementer for å unngå oppdateringsproblemer hvis noen av de refererte elementene forandres. Det kan virke mer logisk å flytte originalen der dette er mulig for å indikere at elementet ikke skal vises lenger, men dette er presentasjonens ansvar.

Siden pekerene ikke er typede, kunne jeg brukt et `ref`-element for å peke på alle mulige elementer. Dette ville imidlertid ha senket lesbarheten betraktelig.

DTD for `entity (objects` *inkludert*`)`

```
<!ELEMENT objects ( entity | value )* >

<!ELEMENT entity ( representation? ) >
<!ATTLIST entity
            id   ID    #REQUIRED
            name CDATA #REQUIRED >

<!ELEMENT representation ( ( value-ref | entity-ref )+,
                           criterion ) >

<!ELEMENT value-ref EMPTY >
<!ATTLIST value-ref
            ref IDREF #REQUIRED >

<!ELEMENT entity-ref EMPTY >
<!ATTLIST entity-ref
            ref IDREF #REQUIRED >

<!ELEMENT criterion ( relationship-ref | uc-ref | mc-ref )+ >

<!ELEMENT relationship-ref EMPTY >
<!ATTLIST relationship-ref
            ref IDREF #REQUIRED >

<!ELEMENT  mc-ref EMPTY >
<!ATTLIST  mc-ref
            ref IDREF #REQUIRED >
```

```
<!ELEMENT  uc-ref EMPTY >
<!ATTLIST  uc-ref
             ref IDREF #REQUIRED >
```

7.2.4 Skranker

Jeg vil her gå igjennom de ulike skrankene gruppevis.

Alle skranker representeres ved hjelp av en ID i tråd med konvensjonen.

Rolleskranker

Jeg følger her metamodellen og skiller ikke mellom interne og eksterne skranker. Dette må derfor overlates til applikasjonen.

Påbudt rolle har fått navnet `mc` (*Mandatory Constraint*) og entydighetsskranken har fått navnet `uc` (*Unique Constraint*).

Jeg har valgt å bruke attributtet `role-refs` av type IDREFS for å peke på rollen/rollene skrankene gjelder[2].

DTD for `uc` *og* `mc`

```
<!ELEMENT  uc EMPTY >
<!ATTLIST  uc
             id  ID #REQUIRED
             role-refs IDREFS >

<!ELEMENT  mc EMPTY >
<!ATTLIST  mc
             id  ID #REQUIRED
             role-refs IDREFS >
```

Eksempel

```
<mc id        = "uc_1074614851.430728"
    role-refs = "role_1074614862.1221609
                 role_1074614867.5371931
                 role_1074614874.994899"/>
```

[2]Dessverre gjør dette at jeg pr dags dato blir nødt til å bruke strengbehandling i XSLT for å skille de forskjellige ID'ene. Dette skyldes imidlertid at XSLT-funskjonen for å lage en sekvens av IDREFS ikke er implementert i Saxon enda.

Frekvens

Denne skranken har fått navnet `fc` (*Frequency Constraint*). Frekvensen kan spesifiseres som et eller flere heltall, et eller flere intervaller eller en kombinasjon av disse.

Hvis jeg hadde valgt XML Schema til å definere XML-malen hadde det vært mulig å si at `fc-num` skulle være et ikke-negativt heltall. Dette er ikke mulig i DTD og må derfor håndteres på applikasjonsnivå.

DTD for `fc`

```
<!ELEMENT  fc ( fc-num | fc-interval )+ >
<!ATTLIST  fc
             id        ID     #REQUIRED
             role-refs IDREFS #REQUIRED >

<!ELEMENT  fc-num EMPTY >
<!ATTLIST  fc-num
             value CDATA #REQUIRED >

<!ELEMENT  fc-interval EMPTY >
<!ATTLIST  fc-interval
             start CDATA #REQUIRED
             stop  CDATA #IMPLIED >
```

Eksempel

```
<fc id        = "fc_1075413839.187187"
    role-refs = "role_1074615235.41852">
  <fc-num value="1"/>
  <fc-interval start="5" stop="10"/>
</fc>
```

Mengdeskranker

Skrankene i denne gruppen inkluderer `ssc` (*Subset Constraint*), `eqc` (*Equality Constraint*) og `xc` (*Exclusion Constraint*).

Alle disse skrankene har attributtene `to-role-refs` og `from-role-refs` som brukes til å referere til to rollegrupper. Selv om det bare er `ssc` som ikke er kommutativ og trenger retningen som `to` og `from` impliserer, har jeg gjort det på denne måten for å få en konsistent løsning.

DTD

```
<!ELEMENT ssc EMPTY >
<!ATTLIST ssc
            id ID #REQUIRED
```

```
            to-role-refs  IDREFS #REQUIRED
            from-role-ref IDREFS #REQUIRED >

<!ELEMENT eqc EMPTY >
<!ATTLIST eqc
            id ID #REQUIRED
            to-role-refs  IDREFS #REQUIRED
            from-role-ref IDREFS #REQUIRED >

<!ELEMENT xc EMPTY >
<!ATTLIST xc
            id ID #REQUIRED
            to-role-refs  IDREFS #REQUIRED
            from-role-ref IDREFS #REQUIRED >
```

Eksempel

```
<ssc id             = "ssc_1075414014.547344"
     to-role-ref    = "role_1075414022.006377
                       role_1075414027.4384439"
     from-role-refs = "role_1075414033.002702
                       role_1075414038.903682"/>
<eqc id             = "eqc_1075414045.71509"
     to-role-refs   = "role_1075414051.268748"
     from-role-refs = "role_1075414057.4042771"/>
```

Ringskranker

Med en modell bestående av bare binære utsagn gjelder ringskranker assosiasjoner istedenfor roller[3].

Istedenfor `role-refs` vil ringskrankene få `relationship-ref` av typen IDREF.

Skrankene som befinner seg i denne gruppen er:

- `itc` (*Intransitive Constraint*)
- `asc` (*Antisymmetric Constraint*)
- `irc` (*Irreflexive Constraint*)
- `syc` (*Symmetric Constraint*)
- `rec` (*Reflexive Constraint*)

[3]For å være helt nøyaktig gjelder de bare assosiasjoner som går fra og til samme begrep.

- `trc` (*Transitive Constraint*)

DTD

```
<!ELEMENT itc EMPTY >
<!ATTLIST itc
            id               ID    #REQUIRED
            relationship-ref IDREF #REQUIRED >
```

DTD'en for de andre ringskrankene blir tilsvarende.

Eksempel

```
<itc id               = "itc_1074616473.6953559"
     relationship-ref = "relationship_1074616486.3412049"/>
```

Verdi skranker

I denne gruppen er det bare to typer skranker: `dvc` (*Default Value Constraint*) og `vc` (*Value Constraint*).

Default Value Constraint tillater kun en verdi. *Value Constraint* tillater verdier, intervaller og kombinasjoner av disse.

DTD

```
<!ELEMENT vc ( vc-value | vc-interval )+ >
<!ATTLIST vc
            id        ID    #REQUIRED
            value-ref IDREF #REQUIRED >

<!ELEMENT vc-value EMPTY >
<!ATTLIST vc-value
            value CDATA #REQUIRED >

<!ELEMENT vc-interval EMPTY >
<!ATTLIST vc-interval
            start CDATA #REQUIRED
            stop  CDATA #IMPLIED >

<!ELEMENT dvc ( vc-value ) >
<!ATTLIST dvc
            id        ID    #REQUIRED
            value-ref IDREF #REQUIRED >
```

Eksempel

```
<vc id        = "vc_1074616932.9701741"
    value-ref = "value_1074616944.8915441">
```

```
  <vc-value value="ugift"/>
  <vc-value value="gift"/>
  <vc-value value="fraskilt"/>
  <vc-value value="enke"/>
</vc>

<dfc id        = "dvc_1074616961.000145"
     value-ref = "value_1074616944.8915441">
  <vc-value value="ugift"/>
</dfc>
```

Eksempelet viser skrankene for en representasjon som bare kan være 'ugift', 'gift', 'fraskilt' eller 'enke'. 'ugift' er standardverdi og vil bli tatt for gitt hvis ingen verdi blir oppgitt.

Subtype

Det er mulig å la subtypedefinisjonen være en streng inne i subtype-elementet, men denne strengen vil ikke være til stor hjelp for verktøyet. For at verktøyet skal kunne gjøre bruk av den, må denne strengen være formalisert og strukturert. [4]

En subtypeskranke har en ID, en referanse til supertypen og en referanse til subtypen.

Overfladisk skisse av subtype-elementet

```
<stc id="stc_1076002050.1582389"
     subtype-ref="entity_1076002057.9632421"
     supertype-ref"entity_1076002063.7692299">
  ..
</stc>
```

Denne skissen forteller hva som er supertype og hva som er subtype, men forteller ingenting om hvorfor (hva som er diskriminator).

Her er et forslag for hvordan subtype-språket kan realiseres ved hjelp av en DTD:

DTD

```
<!ELEMENT stc ( stc-relationship-constraint |
                stc-disjunct-constraint ) >
<!ATTLIST stc
```

[4] Det er mulig å parsere strengen og dele den opp i strukturen som er foreslått, men dette gjør at brukeren blir ansvarlig for at strengen er syntaktisk korrekt. I tillegg krever det en relativt tung parser.

```
            id            ID    #REQUIRED
            subtype-ref   IDREF #REQUIRED
            supertype-ref IDREF #REQUIRED >

<!ELEMENT stc-relationship-constraint ( stc-value-constraint )* >
<!ATTLIST stc-relationship-constraint
            relationship-ref IDREF #REQUIRED >

<!ELEMENT stc-value-constraint EMPTY >
<!ATTLIST stc-value-constraint
            value CDATA #REQUIRED >

<!ELEMENT stc-disjunct-constraint EMPTY >
<!ATTLIST stc-disjunct-constraint
            subtype-ref IDREF #REQUIRED >
```

Subtypedefinisjonene fra eksemplene i de ulike kategoriene (kapittel 4.7 på side 27) blir da seende slik ut:

Eksempel på kategori 1

```
<stc id        = "stc_1075749781.8493581"
     subtype   = "Subtype"
     supertype = "ObjectType">
  <stc-relationship-constraint
       relationship-ref="relationship_1075749788.9936399"/>
</stc>
```

Eksempel på kategori 2

```
<stc id        = "stc_1075749798.53332"
     subtype   = "ValueType"
     supertype = "ObjectType">
  <stc-relationship-constraint
       relationship-ref="relationship_1075749804.4290199"/>
    <stc-value-constraint value="VT"/>
  </stc-relatinoship-constraint>
</stc>

<stc id        = "stc_1075749811.0122571"
     subtype   = "SetComparisionConstraint"
     supertype = "Constraint">
  <stc-relationship-constraint
       relationship-ref="relationship_1075749815.816412">
    <stc-value-constraint value="SS"/>
    <stc-value-constraint value="EQ"/>
    <stc-value-constraint value="X"/>
  </stc-relationship-constraint>
</stc>
```

Eksempel på kategori 3

```
<stc id="stc_1075750021.5556171"
     subtype="UnnestedEntityType"
     supertype="EntityType">
  <stc-disjunct-constraint
       subtype-ref="stc_1075750012.014303"/>
</stc>
```

Det hadde her også vært mulig å referere til begrepet (subtypen) som er disjunkt med subtypen som defineres. Jeg har imidlertid valgt å referere til subtypeskranken for å gjøre ting enklere for applikasjonen.

I eksemplene er begreps-ID'ene byttet ut med begrepsnavn. Dette er gjort for å gjøre det lettere å se sammenhengen mellom de tekstlige subtypedefinisjonene og subtypedefinisjonene uttrykt ved hjelp av XML.

Som nevnt i kapittel 6.2 på side 39 er multippel arv kun tillatt hvis supertypene, direkte eller indirekte, har felles rot-supertype. Dette kan ikke håndheves av en DTD og må håndteres på applikasjonsnivå.

7.2.5 Samlende element for alle skranker

Nå er alle skrankene definert og det er mulig å definere det omkransende elementet.

DTD for constraints

```
<!ELEMENT constraints ( mc  | uc  | fc  |
                        ssc | eqc | xc  |
                        itc | asc | irc |
                        syc | rec | trc |
                        vc  | dvc | stc )* >
```

Kapittel 8

Visualiseringsdirektiver i XML

En visualisering beskrives ved hjelp av visualiseringsdirektiver. Direktivene inneholder kun informasjon om hvordan modellen skal visualiseres, og ikke noe som forandrer semantikken til modellen.

Direktivene havner under `view`-elementet (se kapittel 7.1.2 på side 54) i XML-strukturen.

Jeg vil i dette kapittelet beskrive visualiseringsdirektivene som trengs for å visualisere en modell med og uten manipuleringene fra kapittel 5 på side 29.

8.1 Plassering

For at et element fra modellen skal kunne visualiseres trenger det en plassering. Hvert direktiv refererer til en eller flere modell-elementer ved hjelp av et `-ref`-element.

Visualiseringsdirektiver for å plassere elementer

```
<view>
  <entity-ref ref = "entity_1076849603.468184"
              x   = "10"
              y   = "20"/>
  <value-ref ref = "value_1076849609.256264"
             x   = "100"
             y   = "20"/>
  <relationship-ref ref = "relationship_1076849614.3715291"
                    x   = "50"
                    y   = "20"/>
</view>
```

I visualiseringsdirektivene kan man enten eksplisitt uttrykke hvordan hver enkelt figur skal vises, eller man kan beskrive det «konseptuelt», og la applikasjonen trekke slutninger selv. I arkitekturen jeg bruker har jeg lagt opp til sistnevnte løsning, der figurens transformatorobjekt er ansvarlig for å løfte dataene fra modell-nivå til presentasjonsnivået.

Det blir dermed overlatt til verktøyet hvordan linjene fra rollene til begrepet og representasjonen skal tegnes.

8.2 Rollenes rekkefølge

Endring av rollenes rekkefølgen i en assosiasjon er ikke noe som påvirker semantikken til modellen og skal derfor håndteres i presentasjonen.

For å oppnå en konsistent løsning er rekkefølgen på rollene i alle assosiasjoner beskrevet i visualiseringsdirektivene, selv om den er lik rekkefølgen i modellen.

Siden alle assosiasjoner er binære holder det å beskrive hvilken rolle som er først, eller hvilken rolle som er sist. Jeg har imidlertid valgt å ta med begge rollene i beskrivelsen.

Visualiseringsdirektiver for å beskrive rollerekkefølge

```
<relationship-ref
  ref="relationship_1075289616.3421111"> <role-ref
  ref="role_1075289630.9135571"/> <role-ref
  ref="role_1075289644.2954619"/> </relationship-ref>
```

Det er ønskelig å kunne tegne både horisontalt og vertikalt orienterte assosiasjoner. I visualiseringsdirektivene blir dette uttrykt i `relationship`-elementet ved hjelp av `orientation`-attributtet som kan være *vertical* eller *horizontal*, der sistnevnte er standardverdi.

8.3 Representasjon

Hvis en representasjon identifiserer et begrep kan den grupperes inn i begrepet (og skrives i parenteser under begrepsnavnet). Assosiasjonen mellom representasjonen og begrepet, med tilhørende skranker, blir da

undertrykt og skal ikke vises eksplisitt. Representasjonen i sin orginale form blir også undertrykt, og blir istedenfor tegnet som navn omgitt av parenteser under begrepsnavnet.

Visualiseringsdirektiver for representasjon ved perfekt bro

```
<entity-ref ref="entity_"/>
<value-ref ref="value_" hidden="true"/>
<relationship-ref ref="relationship_" hidden="true"/>
<uc-ref ref="uc_" hidden="true"/>
<uc-ref ref="uc_" hidden="true"/>
<mc-ref ref="mc_" hidden="true"/>
```

`hidden`-attributtet brukes for å beskrive at elementets figur er undertrykt i sin originale form. Elementene ble undertrykt da brukeren valgte å gruppere representasjonen inn i begrepet.

Begrensninger med denne løsningen oppstår hvis et element blir undertrykt av mer enn en abstraksjon, og brukeren "oppløser" en av abstraksjonene. Elementet vil da bli synlig igjen, selv om det egentlig er undertrykt av en annen abstraksjon. Dette kan løses ved å la `hidden`-attributtet inneholde et heltall som teller antall abstraksjoner det er undertrykt av, istedenfor **true/false**. Når heltallet synker ned til 0 skal figuren som elementet representerer vises igjen. Jeg har valgt å gå for **true/false** løsningen fordi jeg ikke ser for meg at et element kan bli undertrykt av mer enn en abstraksjon.

Hvis dette viser seg å være galt, krever det ikke store forandringer for å skfite til tellere istedenfor **true/false**.

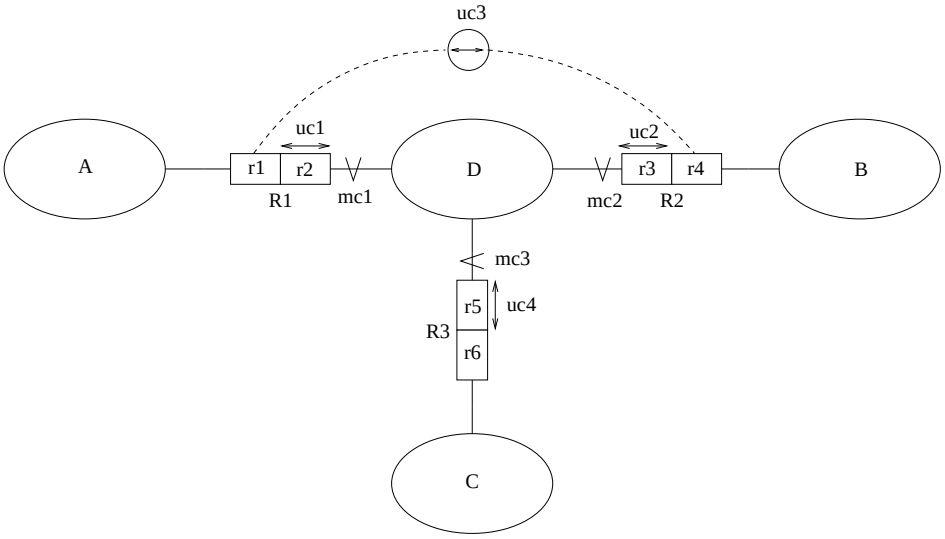
Merk at det ikke blir markert i visualiseringsdirektivene at representasjonen identifiserer begrepet. Dette er semantisk informasjon, og tilhører derfor modellen.

8.4 Begrepsdannelse

Figur 8.1 på neste side viser hvordan en begrepsdannelse blir uttrykt i modellen.

Alle roller og skranker er navngitt for at det skal bli lettere å se hva som skjer med dem ved omskriving.

Modellen inneholder ingen føringer på hvordan dette skal presenteres, og visualiseringsdirektivenes oppgave er å markere hvilket begrep som skal visualiseres som en begrepsdannelse. I dette tilfellet vil dette være begrepet **D**.



Figur 8.1: Begrepsdannelse uttrykt i modellen

Visualiseringsdirektiver for begrepsdannelse

```
<view>
  <entity-ref ref="D">
    <objectified>
      <role-ref ref="r1"/>
      <role-ref ref="r4"/>
    </objectified>
  </entity-ref>
  <relationship-ref ref="R1" hidden="true"/>
  <relationship-ref ref="r2" hidden="true"/>
  <uc-ref ref="uc1" hidden="true"/>
  <uc-ref ref="uc2" hidden="true"/>
  <uc-ref ref="uc3" hidden="true"/>
  <mc-ref ref="mc1" hidden="true"/>
  <mc-ref ref="mc2" hidden="true"/>
</view>
```

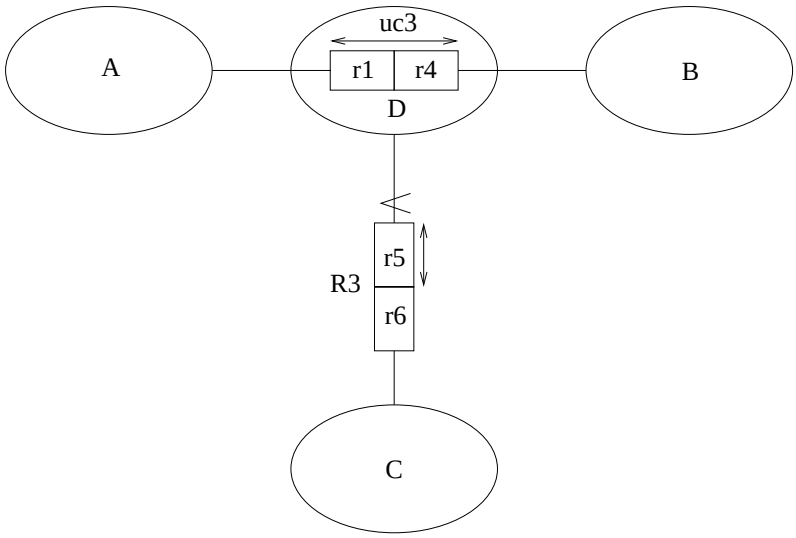
I visualiseringsdirektivene har jeg for enkelhets skyld utelatt koordinatene, og brukt ID'ene fra figuren istedenfor ID'er i tråd med navngivningskonvensjonen. Dette vil også bli gjort i de kommende eksemplene.

En visualisering som tar i bruk disse visualiseringsdirektivene er vist i figur 8.2 på neste side.

`objectified`-elementet inneholder rekkefølgen på rollene i den virtuelle assosiasjonen som har blitt gjenstand for begrepsdannelse.

For at notasjonen ikke skal være tvetydig har jeg valgt å ikke tillate omskriving av begreper til begrepsdannelser hvis det er mulig å gjøre

dette på mer enn en måte.



Figur 8.2: Visualisert som begrepsdannelse

Selv om **uc3** fremdeles vises er den undertrykt i visualiseringsdirektivene. Dette er fordi den er undertrykt i sin originale form. Den har gått fra å være en selvstendig figur til å bli en del av begrepsdannelsen. Dette betyr at opptegningsansvaret er flyttet fra lerretet til begrepsdannelse-figuren.

Ved å beskrive en begrepsdannelse på denne måten vil begrepet og begrepsdannelsen **D** dele koordinater. Dette betyr at hvis man ønsker å se på begrepet **D** som en begrepsdannelse, vil begrepsdannelsen overta koordinaten til begrepet **D**, og vice versa.

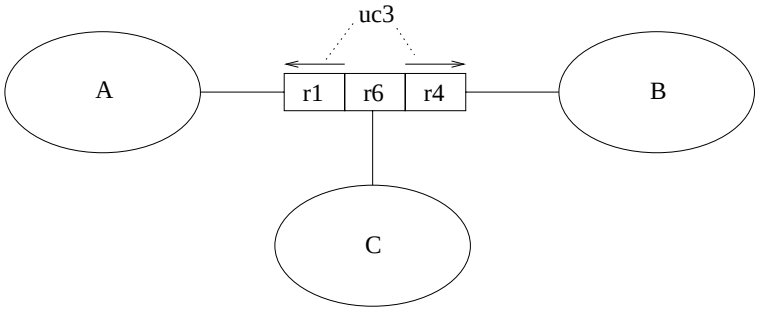
8.5 N-ære utsagn

Figur 8.1 på forrige side kan også skrives som til det ternære utsagnet vist i figur 8.3 på neste side.

Transformatoren bruker sterk gruppering for å lage den ternære assosiasjonen. N-ære utsagn kan bare dannes hvis alle tilknyttede roller blir gruppert sammen til en gruppe. `n-ary`-elementet inneholder rekkefølgen på rollene.

Visualiseringsdirektiver for n-ært utsagn

```
<entity-ref ref="D">
  <n-ary>
```

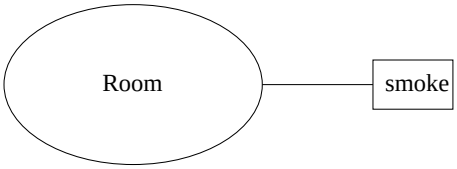


Figur 8.3: Visualisert som ternært utsagn

```
    <role-ref ref="r1"/>
    <role-ref ref="r4"/>
    <role-ref ref="r6"/>
  </n-ary>
</entity-ref>
<entity-ref ref="A"/>
<entity-ref ref="B"/>
<entity-ref ref="C"/>
<uc-ref ref="uc1" hidden="true"/>
<uc-ref ref="uc2" hidden="true"/>
<uc-ref ref="uc3" hidden="true"/>
<uc-ref ref="uc4" hidden="true"/>
<mc-ref ref="mc1" hidden="true"/>
<mc-ref ref="mc2" hidden="true"/>
<mc-ref ref="mc3" hidden="true"/>
```

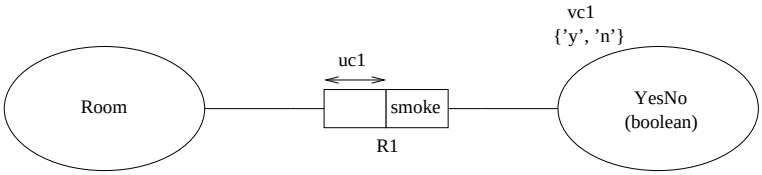
Selv om **uc3** fremdeles vises, er den undertrykt i sin originale form, og opptegningsansvaret er flyttet fra lerretet til n-ære-asssosiasjon-figuren.

8.6 Unære utsagn



Figur 8.4: Unært utsagn

Unære utsagn (figur 8.4) blir uttrykt ved hjelp av binære utsagn i modellen (figur 8.5 på neste side).



Figur 8.5: Modell

Visualiseringsdirektiver for unært utsagn

```
<view>
  <entity-ref ref="Room"/>
  <entity-ref ref="YesNo" hidden="true"/>
  <relationship-ref ref="R1">
    <unary>
  </relationship-ref>
  <uc-ref ref="uc1" hidden="true"/>
  <vc-ref ref="vc1" hidden="true"/>
</view>
```

Den perfekte broen mellom **YesNo** og *boolean* er utelatt fra visualiseringsdirektivene for enkelhets skyld.

Det er her assosiasjonen som blir abstrahert. `uc1` og **YesNo** med tilhørende verdiskranke blir undertrykt. Transformator-objektet tar seg av abstraheringen.

8.7 Grupper

Visualiseringsdirektiver for gruppering

```
<entity-ref ref="entity_1076853336.872483">
  <grouped method="strong">
    <group name="Person">
      ..
      ..
    </group>
  </grouped>
</entity-ref>
..
..
```

Visualiseringsdirektivene beskriver hvilken type gruppering transformatorobjektet skal bruke for å danne gruppen. `grouped`-elementet forteller hvilken grupperingsforms som er benyttet, og inneholder en eller flere grupper.

Hvert `group`-element inneholder rekkefølgen på attributtene.

8.8 Fra visualiseringsdirektiv til visualisering

For at et element skal kunne visualiseres må det ha en ID. Denne ID'en brukes av figuren for å få informasjon fra modellen (via transformatoren) om semantikken og informasjon fra presentasjonen om plassering og eventuelt abstraksjon.

Hvis brukeren tegner en ternær assosiasjon, vil det bak kulissene bli dannet en modell som vist i figur 5.2 på side 31, og visualiseringsdirektiver som vist i 8.5 på side 70. Den ternære assosiasjonen vil identifiseres av ID'en til begrepet som er senter for grupperingen (D). Rollene vil ha ekte rolle ID'er.

Ved opptegning spør figuren transformatoren om hvilke roller den har, og presentasjonen om rekkefølgen på disse rollene. Abstraheringen resulterer i at assosiasjonen i tillegg til å være ansvarlig for opptegning av rollene, vil bli ansvarlig for opptegningen av de interne entydighetsskrankene for denne virtuelle assosiasjonen[1].

[1]Dette er et eksempel på bruk av composite mønsteret[25].

Kapittel 9

Manipulering med XSLT

Transformator-objektene bruker XSLT for å løfte data fra modellnivå til presentasjonsnivå.

I dette kapittelet vil jeg gå igjennom noen av disse XSLT-algoritmene.

9.1 Identifisering

Identifisering går ut på å finne en representasjon for et begrep.

Vi kan nå uttrykke mer presist hvordan et begrep kan bli representert:

Representasjon ved arv
: Hvis et begrep er en subtype av et annet begrep vil det arve representasjon av supertypen. Denne typen representasjon er den eneste representasjonsform som har høyere presedens enn de andre.

Representasjon ved perfekt bro
: En perfekt bro er en påbudt en-til-en assosiasjon mellom et begrep og en representasjon. Dette er den vanligste formen for representasjon.

Representasjon ved entydig utsagn
: Et begrep A har en påbudt en-til-en assosiasjon til et begrep B. A blir da representert av begrepet B. Hvis B har en representasjon, vil A indirekte bli representert ved denne representasjonen.

Sammensatt representasjon
: Med sammensatt representasjon menes begreper som represen-

teres av kombinasjonen av flere representasjoner og fullt-/delvis-informasjonsbærende utsagn. Begrepet vil da bli indirekte representert av en eller flere begreper i tillegg til eventuelle representasjoner.

I mange tilfeller kan representasjonen finnes automatisk. Dette gjelder i de tilfeller der det bare finnes et alternativ, og der en representasjons kandidat overstyrer en eller flere andre kandidater (representasjon ved arv har høyere presedens enn de tre andre representasjonsformene). Hvis det finnes flere alternativer kan man enten legge inn konvensjoner i applikasjonen som gjør at enkelte representasjoner blir foretrukket fremfor andre, eller overlate valget til brukeren.

En konvensjon kan ikke garantere gode representasjoner, siden verktøyet ikke har noen forståelse av modellen og ikke vet hva som f.eks. vil være en stabil representasjon. Jeg vil derfor utelate identifiseringskonvensjoner fra verktøyet, og nøye meg med å implementere presedens.

9.1.1 Overordnet algoritme

Den overodnede algoritmen for identifisering er skissert i figur 9.1 på side 85.

Identifiseringsalgoritmen tar begreps-ID'en til begrepet som skal identifiseres som input.

Det første som blir gjort er å sjekke om input er gyldig. Hvis denne testen mislykkes, returneres et `error`-element med beskrivelse av hva som gikk galt, og algoritmen avsluttes. Dette er utelatt fra flytdiagrammet av plasshensyn.

«Let etter kandidater til representasjon» innbærer leting etter representasjon blant de fire ulike representasjonsformene.

9.1.2 Arv

Subtyper arver representasjon fra supertypen, og begreper som er subtyper kan ikke ha annen representasjon enn den arvede.

Denne delen av algoritmen begynner med å lete etter subtypeskranker som inneholder det aktuelle begrepet som subtype.

Hvis det finnes en supertype, returneres et svar på følgende format som en «kandidat til representasjon»:

Returverdi

```
<identified-by-supertype>
  <entity-ref ... />
  <criterion>
    <stc-ref ... />
  </criterion>
</identified-by-supertype>
```

Innholdet i `identified-by-supertype` er på samme format som innholdet i `representation`. Hvis denne kandidaten ender opp som representasjon for det aktuelle begrepet, blir rot-elementet i svaret erstattet med `representation`, og lagt inn under begrepet. Det samme gjelder for svarene som blir returnert fra de andre representasjonsmåtene.

Hvis begrepet ikke har noen supertype returneres ingenting.

Jeg har valgt å benytte meg av en snarvei for å finne denne typen representasjon. Istedenfor å rekursere oppover i subtype-hierarkiet for å finne representasjonen til den absolutte supertypen, plukker jeg bare opp den første supertypen jeg treffer, og lar dette begrepet være representasjon for det aktuelle begrepet . Dette medfører at applikasjonen må sjekke om subtype-hierarkiet er gyldig, og rekursere oppover for å finne den egentlige representasjonen (representasjonen til den absolutte supertypen). Under modelleringen kan det være en fordel at kun begynnelsen på "resonnementet" ligger lagret istedenfor "svaret". Dette medfører at forandringer automatisk blir plukket opp. På den andre siden vil dette være en tidkrevende prosess.

Noe lignende blir gjort for «sammensatt representasjon» og «Representasjon ved entydig utsagn» (se kapittel 9.1.4 på neste side og 9.1.3).

9.1.3 Perfekte broer og representasjon ved entydig utsagn

Algoritmisk sett er det ikke mye som skiller leting etter «perfekte broer» fra leting etter «representasjon ved entydige utsagn». Forskjellen ligger i om det er en representasjon eller et begrep på den andre siden av en-til-en assosiasjonen. Jeg har derfor valgt å slå sammen disse to algoritmene.

Algoritmen itererer over alle assosiasjonen som er tilknyttet begrepet. Hver assosiasjon blir delt inn i: «rolle» og «korolle» («korollen» er rollen som ikke er tilknyttet begrepet).

For hver av disse assosiasjonene blir følgende sjekket:

- det finnes 1 påbudt rolle og 2 entydighetsskranker tilknyttet assosiasjonen.
  - «rollen» er påbudt.
  - det er en entydighetsskranke knyttet til «rollen».
  - det er en entydighetsskranke knyttet til «korollen».

For hver assosiasjon som faller inn under disse kriteriene blir det lagd et element som gjenspeiler om det er funnet en «perfekt bro» eller en «representasjon ved entydig utsagn». Inne i dette elementet blir det lagt referanser til representasjonen og kriteriene for representasjonen.

Dataene blir returnert på dette formatet:

Returverdi

```
<[perfect-bridge|identifying-association]>
  <[value-ref|entity-ref] ../>
  <criterion>
    <relationship-ref ../>
    <mc-ref ../>
    <uc-ref ../>
    <uc-ref ../>
  </criterion>
</[perfect-bridge|identifying-association]>
```

9.1.4 Sammensatt representasjon

De interessante representasjonsformene er her fullt- og delvis-informasjonsbærende utsagn, og begreper som representeres av en sammensetning av to eller flere representasjoner.

For å finne disse representasjonsformene, samles alle assosiasjoner som er tilknyttet begrepet.

Alle `uc`- og `mc`-skranker som kun refererer til en eller flere av rollene i de valgte assosiasjonen blir plukket ut.

Vi har nå en gruppe med assosiasjoner og en gruppe med skranker. De eksterne entydighetsskrankene fra skranke-gruppen blir brukt til å danne nye grupper, som skal inneholde den eksterne skranken brukt for å gruppere, alle assosiasjonene som har roller som blir beskranket av den eksterne entydighetsskranken, og alle andre skrankene fra skranke-gruppen som kun refererer til roller som befinner seg i denne nye gruppen.

Hver assosiasjonen i den nye gruppen deles opp i «rolle» og «korolle». Alle «roller» må ha en «påbudt rolle»-skranke og en entydighetsskranke i gruppen, for at gruppen skal plukkes ut. [1]

Nå gjenstår det bare å finne alle elementene tilknyttet «korollene» i hver utplukkede gruppe, og formatere dataene på ønsket måte.

Formatet for `composite-identification` er det samme som for bl.a. `perfect-bridge`, men med et annet rot-element.

Returverdi

```
<composite-identification>
  <[entity|value]-ref ... />
  <[entity|value]-ref ... />
  <criterion>
    <relationship-ref ... />
    <relationship-ref ... />
    <[mc|uc]-ref ... />
    ..
    ..
    <[mc|uc]-ref ... />
  </criterion>
</composite-identification>
```

9.2 Gruppering

Det er her snakk om flere algoritmer som veves sammen:

- Svak gruppering
  - Svak gruppering av begrep
  - Svak gruppering av assosiasjon
- Sterk gruppering
  - Sterk gruppering av begrep
  - Sterk gruppering av assosiasjon
- Identifisering av begrep (beskrevet i kapittel 9.1 på side 74)

Jeg vil her skissere opp deler av XSLT-skriptet for «svak gruppering av begrep» og forklare hva som skal gjøres i de andre algoritmene.

[1] Vi vet at «korollen» er del av en ekstern entydighetsskranke, siden det var den som var kriteriet for gruppen.

9.2.1 Svak gruppering

Denne typen gruppering brukes bl.a. til å lage UML-klasser eller ER-entiteter av elementære utsagn.

Den overordnede algoritmen finner alt som skal være gjenstand for svak gruppering. Selve grupperingen overlates til de underordnede algoritmene: «Svak gruppering av begrep» og «Svak gruppering av assosiasjon».

Det blir først sjekket om alle forutsetningene for gruppering er innfridd. Hvis det finnes en eller flere mange-til-mange eller ikke-påbudt en-til-en assosiasjoner som ikke har predikatnavn, returneres en opplysende feilmelding og algoritmen termineres. [2]

Hvis alle forutsetningene er innfridd fortsetter algoritmen. For hvert begrep kjøres «Svak gruppering av begrep»-algoritmen.

Alle en-til-en assosiasjoner som ikke er påbudt i noen av endene, og mange-til-mange assosiasjonene sendes videre til «Svak gruppering av assosiasjon».

Svak gruppering av begrep

Jeg har her delt opp algoritmen i flere punkter:

1. **Forberedelse**
   Finn alle assosiasjoner som er tilknyttet begrepet, og lag grupper av dem ved hjelp av entydighetsskranker tilknyttet «korollen». Assosiasjoner som ikke har noen entydighetsskranke tilknyttet «korollen», danner egne grupper.

   Ekskluder grupper som består av en mange-til-mange assosiasjon eller en ikke-påkrevd en-til-en assosiasjon. Disse grupperes av «svak gruppering av assosiasjoner».

   Ekskluder assosiasjoner der den «rollen» ikke er beskranket av en entydighetsskranke.

   De gjenværende gruppene lagres i variabelen *remaining-groups*.

2. **Finn representasjon**
   Identifiseringsalgoritmen blir kjørt for begreper som ikke har representasjon.

[2] Mange-til-mange assosiasjoner som har blitt gjenstand for begrepsdannelse vil ikke lenger være uttrykt som mange-til-mange assosiasjoner i modellen.

Hvis algoritmen gir et entydig svar, vil svaret blir brukt til representasjon.

remaining-groups = *remaining-groups* − *«representasjon»*

3. **Finn identifikator**
   Begreper som nå har en representasjon setter representasjonen til å være identifikator for gruppen.

   remaining-groups = *remaining-groups* − *«identifikator»*

4. **Finn nøkler som ikke kan inneholde attributter uten verdi**
   Finn alle gruppene i *remaining-groups* der alle «roller» er påbudte og alle «korollene» er beskranket med en entydighetsskranke. Gruppene brukes til å danne kandidatnøkler som ikke tillater attributter uten verdi i gruppen.

   remaining-groups = *remaining-groups* − *«gruppene funnet»*

5. **Finn nøkler som tillater attributter uten verdi**
   Finn alle gruppene i groups-remaining der alle «korollene» er beskranket av entydighetsskranke (intern eller ekstern). Siden alle assosiasjonene som har påbudte «tilknyttede roller» er fjernet fra groups-remaining, vil disse gruppene bli nøkler som kan inneholde attributter uten verdi.

   remaining-groups = *remaining-groups* − *«gruppene funnet»*

6. **Finn attributter som må ha en verdi**
   Nå skal *remaining-groups* bestå av grupper som kun inneholder en assosiasjon som enten er påbudt en-til-mange eller ikke-påbudt en-til-mange.

   Finn alle grupper/assosiasjoner i *remaining-groups* der den «rollen» er påbudt. Dette blir attributter som må ha en verdi.

   remaining-groups = *remaining-groups* − *«gruppene funnet»*

7. **Finn attributter som ikke må å ha en verdi**
   Gruppene som nå er igjen inneholder en assosiasjon som er en-til-mange (der «rollen» er entydig). Alle disse gruppene blir til attributter som ikke må ha en verdi.

Som beskrevet tidligere finnes det flere ulike måter å behandle subtyper under gruppering (se kapittel 5.3.3 på side 36). Gruppering av subtyper ved separasjon implementeres ved at algoritmen sørger for

at identifikator for subtype-gruppen blir en peker/fremmednøkkel til identifikatoren i supertype-gruppen. Dette forutsetter at supertypen grupperes før subtypen.

XSLT for Svak gruppering av begrep

Grupperingen løses ved hjelp av *Divide and Conquer*[18, 19]. Alle problemstillinger deles opp i mindre biter som blir skilt ut i egne funksjoner. Disse funksjonene blir igjen delt opp i enda mindre biter osv. osv.

Dette gjør hver enkelt problemstilling mer oversiktlig og algoritmen blir lettere å vedlikeholde.

Jeg vil her skissere opp den overordnede algoritmen for gruppering av begrep. Den tar seg av henting og formatering av informasjon.

Istedenfor å lage f.eks. en `get-attributes`-funksjon har jeg laget `get-not-null-attributes` og `get-null-attributes`. Dette er gjort for å ende opp med funksjoner som også kan brukes av algoritmen for sterk gruppering.

orm-group:group-entity

```
<xsl:function name="orm-group:group-entity">
  <xsl:param name="eid">

  <!-- get connected relationships -->
  <xsl:variable name="relationships"
                select="orm-util:get-connected-relationships($eid)"/>

  <!-- group connected relationships by uniqueness-constraints -->
  <xsl:variable name="remaining-groups"
        select="orm-util:group-relationships-with-constraints($eid,
                                                              $relationships)"/>

  <xsl:variable name="return-value">
    <!-- find primary key -->
    <xsl:variable name="primary"
                  select="orm-group:get-primary($eid,
                                                $remaining-groups)"/>
    <!-- output primary key -->
    <xsl:element name="primary">
      <xsl:copy-of select="$primary"/>
    </xsl:element>

    <!-- exclude primary-key from remaining-groups -->
    <xsl:variable name="remaining-groups"
                  select="orm-group:exclude-groups($remaining-groups,
                                                   $primary)"/>
```

```
    <!-- find not-null-keys -->
    <xsl:variable name="not-null-keys"
                  select="orm-group:get-not-null-keys($eid,
                                                      $remaining-groups)"/>
    <!-- exclude not-null-keys from remaining-groups -->
    <xsl:variable name="remaining-groups"
                  select="orm-group:exclude-groups($remaining-groups,
                                                   $not-null-keys)"/>

    <!-- find null-keys -->
    <xsl:variable name="null-keys"
                  select="orm-group:get-null-keys($eid,
                                                  $remaining-groups)"/>
    <!-- exclude null-keys from remaining-groups -->
    <xsl:variable name="remaining-groups"
                  select="orm-group:exclude-groups($remaining-groups,
                                                   $null-keys)"/>

    <!-- output keys -->
    <xsl:element name="keys">
      <xsl:copy-of select="$not-null-keys"/>
      <xsl:copy-of selecT="$null-keys"/>
    </xsl:element>

    <!-- find not-null attributes -->
    <xsl:variable name="not-null-attributes"
                  select="orm-group:get-not-null-attributes($eid,
                                                            $remaining-groups)"/>
    <!-- exclude not-null-attributes from remaining-groups -->
    <xsl:variable name="remaining-groups"
                  select="orm-group:exclude-groups($remaining-groups,
                                                   $not-null-attributes)"/>

    <!-- find null attributes -->
    <xsl:variable name="null-attributes"
                  select="orm-group:get-null-attributes($eid,
                                                        $remaining-groups)"/>

    <!-- output attributes -->
    <xsl:element name="attributes">
      <xsl:copy-of select="$not-null-attributes"/>
      <xsl:copy-of select="$null-attributes"/>
    </xsl:element>
  </xsl:variable>
  <xsl:sequence select="$return-value"/>
</xsl:function>
```

`exclude-groups`-funksjonen er veldig sentral. Den har to input-

parameter: gjenværende grupper, og grupper som skal eksluderes. Den returnerer en liste over *«gjenværende grupper» - «grupper som skal eksluderes»*.

Algoritmen itererer over alle gjenværende grupper og sjekker om hver enkelt gruppe finnes i listen over grupper som skal bli ekskludert. Gruppene som ikke finnes der blir samlet og returnert.

To grupper som har like mange `relationship-ref`, der hver `relationship-ref` finnes i begge gruppene blir betraktet som like grupper.

orm-group:exclude-groups

```
<xsl:function name="orm-group:exclude-groups">
  <xsl:param name="groups"/>
  <xsl:param name="groups-to-be-excluded"/>

  <xsl:variable name="return-value">
    <xsl:for-each select="$groups">
      <xsl:if test="
         every $g in $groups-to-be-excluded/*
         satisfies ( not( count(current()//relationship-ref)
                          =
                          count($g//relationship-ref) )
                     or
                     some $r in $g//relationship-ref
                     satisfies not(current()//relationship-ref[@ref = $r/@ref]))
                   )">
        <xsl:copy-of select="current()"/>
      </xsl:if>
    </xsl:for-each>
  </xsl:variable>
  <xsl:sequence select="$return-value"/>
</xsl:function>
```

Øvrige funksjoner er utelatt av plasshensyn.

Svak gruppering av assosiasjon

Ikke-påbudte en-til-en assosiasjoner og mange-til-mange assosiasjoner blir her til egne grupper. Gruppen får navn fra predikatnavnet.

9.2.2 Sterk gruppering

Denne typen gruppering tillater ikke attributter uten verdier i gruppene, og blir brukt bl.a. for å gruppere til n-ære assosiasjoner i visualiseringen.

Alle begreper blir sendt videre til «Sterk gruppering av begrep».

Assosiasjoner som ikke er påbudt i noen ende og ikke blir beskranket av noen likhetsskranke, blir sendt videre til «Sterk gruppering av assosiasjoner».

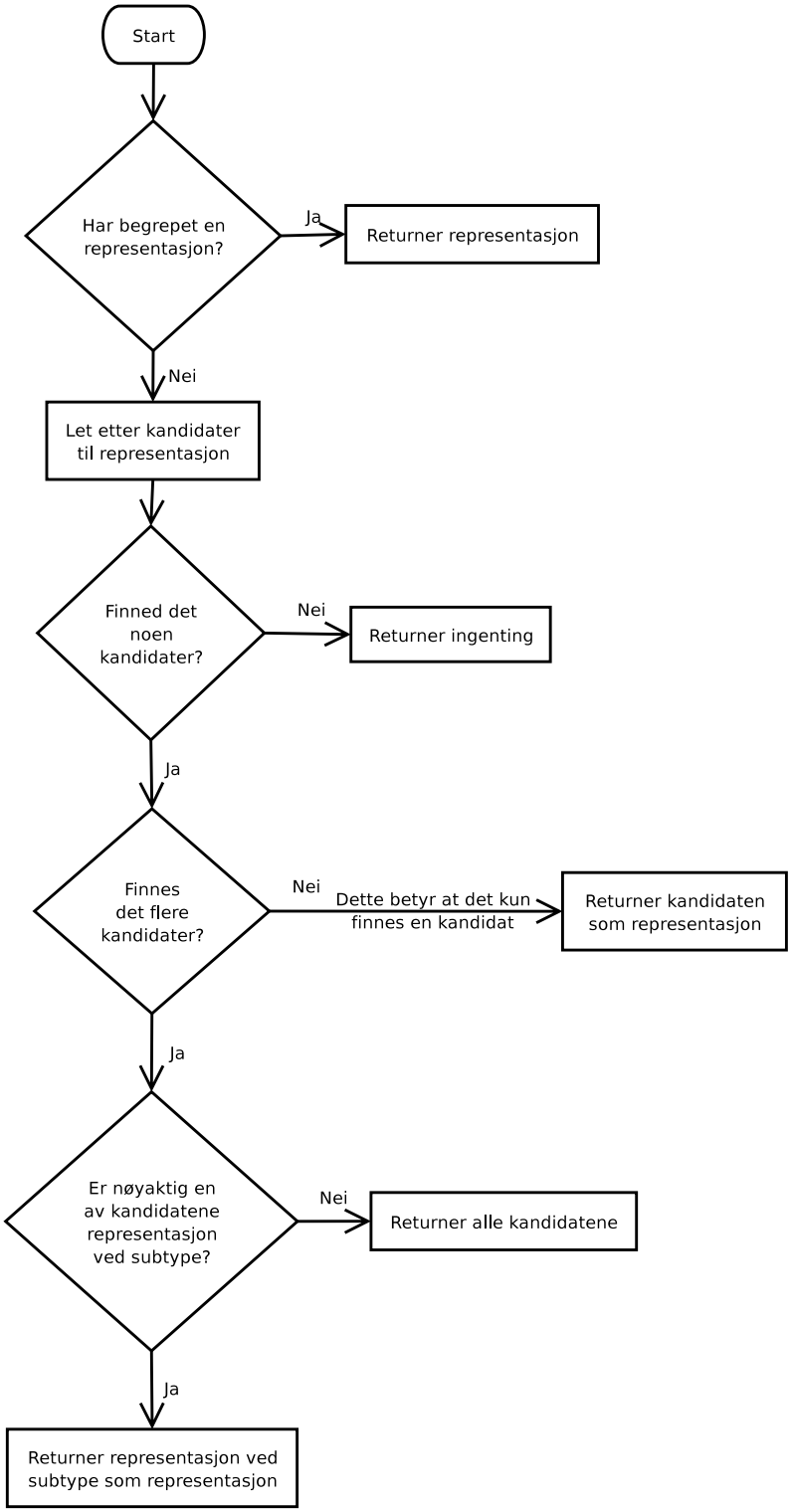
Sterk gruppering av begrep

Algoritmen for sterk gruppering av begrep er veldig lik algoritmen for svak gruppering av begrep, men vil hoppe over punkt 5 og 7 på side 80.

Dette betyr at XSLT-algoritmen ikke vil ta i bruk `get-null-keys` og `get-null-attributes`.

Sterk gruppering av assosiasjon

Alle assosiasjonene som kommer inn her blir egne grupper. Entydighetskranken avgjør hva som blir identifikator.



Figur 9.1: Oversikt over identifiseringsalgoritme

Kapittel 10

Realisering av «Niamderthal»

Verktøyet har fått arbeidstittelen «Niamderthal», og spiller på at den interne modellrepresentasjonen består av kun «binære elementære utsagn», som i tidlig NIAM.

Dette kapittelet omhandler realiseringen av «Niamderthal»; hvordan utviklingen har foregått og hvordan arkitekturen er realisert.

10.1 Utviklingsforløp

Vedlegget inneholder versjon 0.4 av Niamderthal. De to første versjonene var prototyper og ble laget for å få en forståelse av problemområdet. Versjon 0.3 ble programmert helt fra bunn uten XML som modellrepresentasjon. Siste versjon (v0.4) er en endring av v0.3 der XML er tatt i bruk som intern modellrepresentasjon.

På grunn av dette utviklingsforløpet finnes det endel kode som ikke er i bruk idag (f.eks. `no.uio.ifi.Niamderthal.uml`). Dette er kode som virket i v0.3, men som ikke er flyttet over på den nye arkitekturen. Jeg regner med å få mye gratis av koden som ligger «brakk» ved eventuell videre utvikling.

10.2 Realisering av arkitektur

Applikasjonen er delt inn i mange forskjellige pakker og underpakker. Det er tenkt at f.eks. `no.uio.ifi.Niamderthal.common.figures` er en underpakke av `no.uio.ifi.Niamderthal.common`, selv om det ikke finnes noe slektskap mellom disse pakkene i Java.

Figurene er delt inn i ulike pakker: `no.uio.ifi.Niamderthal.orm.figures`, `no.uio.ifi.Niamderthal.uml.figures` og `no.uio.ifi.Niamderthal.common.figures`. Sistnevnte pakke inneholder både kodebaser som er felles for alle symboler (`no.uio.ifi.Niamderthal.common.figures.Symbol`), og symboler som er felles for alle teknikker (`no.uio.ifi.Niamderthal.common.figures.Note`). De andre pakkene inneholder symboler for hver sine datamodelleringsspråk.

For å få et generisk lerret som kan håndtere alle typer symboler, må figurene ha et kjent API som lerretet benytter seg av. Jeg har implementert dette ved hjelp av arv. Alle symboler arver direkte eller indirekte fra `no.uio.ifi.Niamderthal.common.figures.Symbol`, som innholder alle metoder som blir brukt av lerretet.

Jeg har i realiseringen valgt å slå sammen **Figure** med tilhørende **Meny**. Selv om de har forskjellige ansvarsområder, ser jeg dette som praktisk siden de deler livsløp.

Da jeg hadde programmert de essensielle delene av arkitekturen og det som gjenstod kunne betraktes som håndverk, stoppet jeg utviklingen. Under skrivingen av rapporten ble jeg imidlertid i flere tilfeller "opplyst", og kom frem til bedre løsninger enn de jeg allerede allerede hadde programmert.

Dette betyr at realiseringen ikke er en 100% realisering av det som har blitt beskrevet tidligere. På grunn av tidsmangel ble ikke de nye løsningene implementert, og realiseringen avviker derfor noe fra teksten.

10.2.1 Opprettelse av figur

Den eneste muligheten for å opprette en ny figur fra tomt lerret er ved hjelp av lerret-menyen. Figur 10.1 på side 91 viser funksjonskallene som blir gjort bak kulissene.

createFigure-funksjonen er ment som en generalisering av alle funksjonene som skaper figurer, og er ikke en ekte funksjon. Ved å bytte ut "Figure" med ønsket figurnavn får man de ekte funksjonene: *createEntity*, *createValue*, osv.

Det første som skjer er at bakgrunnsmenyen spør lerretet om å få en peker til fasaden for modelleringsspråket som er brukt som grunntype i presentasjonen. **Facade** er et mønster med følgende formål:

> Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.[25]

Sekvensdiagrammet illustrerer dette ved at **BgMenu** kun har et kall mot **Facade**. Fasaden tar seg av alle "lavnivå" funksjonskallene.

getTransformer-funksjonen i **Model** returnerer en transformator for en figur spesifisert med ID. Hvis det ikke finnes noen transformator for denne figuren, blir en ny transformator laget og returnert.

Hvis det allerede finnes en transformator for denne figuren, returneres denne. Dette skjer når det finnes flere presentasjoner som visualiserer det samme modell-elementet. Alle visualiseringene av elementet vil da dele ett transformator-objekt.

«every»-stereotypen til **View**-objektet illustrerer at *createFigure* blir gjort for alle **View**.

Det siste som skjer før *createFigure* i **Facade** returnerer, er at den nyopprettede figuren får en plassering (koordinat) i presentasjonen den ble opprettet i. Dette er den eneste presentasjonen der brukeren har spesifisert hvor figuren skal stå. I de andre presentasjonene blir den plassert i origo.

Når lerretet ikke lenger er tomt, vil det også være mulig å opprette nye figurer ved hjelp av de ulike figur-menyene. Når brukeren velger å opprette en figur fra figur-menyen vil det bli lagd et passende opprettelses-**tool** som mottar alle *mouseevents*. Etter at opprettelses-**tool** har fått all den informasjonen det trenger (hva som skal kobles sammen og hvor den nye figuren skal være) sendes det beskjed til fasaden. Herfra er dataflyten den samme som for oppretting av figur ved tomt lerret.

10.2.2 Sletting av figur

En figur kan bli slettet på to måter:

- **sletting av figur fra presentasjon** innebærer av figuren fremdeles kan finnes i andre presentasjoner.
- **sletting av figur fra modell** innebærer at den også må slettes fra alle presentasjoner.

Sletting av figur fra presentasjon

Dette er en operasjon som bare kan gjøres fra figur-menyen (**FigureMenu** i figur 10.2 på side 92).

deregister-kallet fra **current:View** til **Figure** er en del av *observer*-mønsteret[25] som implementeres mellom figurene og presentasjonen for å håndtere en-til-mange forholdet imellom dem.

Figure itererer over alle tilknyttede figurer og sletter de som er "mindreverdige". Med dette mener jeg at f.eks. sletting av et begrep kan medføre at en en eller flere assosiasjon blir slettet, sletting av en assosiasjon kan medføre at en eller flere skranker blir slettet, men sletting av en assosiasjon vil ikke medføre at et begrep blir slettet. Dette er fordi et begrep har høyere "rang" enn en assosiasjon. Figurene er hierarkisk inndelt slik at det ikke vil forekomme sykler i slette-grafen.

connected:Figure illustrerer de figurene som blir slettet fordi de har lavere "rang".

Egentlig burde **Facade** vært brukt til denne oppgaven også, men pga tidsmangel er ikke dette gjennomført.

Sletting av figur fra modell

I figur 10.3 på side 93 får **Facade**-objektet mindre å gjøre. Dette skyldes at sletting fra modell medfører sletting fra presentasjon. Dette er **Model** sitt ansvar.

other:Figure illustrerer alle figurer som må slettes fordi de er "mindreverdige".

10.3 Omfang av kildekoden

For å gi et inntrykk av kildekodens omfang har jeg tatt med en kjøreutskrift av `sloccount`, som er et program som brukes for å telle "fysiske" kildekodelinjer (linjer som ikke er kommentarer eller blanke). Utskriften sier at det finnes nesten 8.700 fysiske Java-kodelinjer.

sloccount

```
SLOC    Directory       SLOC-by-Language (Sorted)
8709    no              java=8696,lisp=13
0       CVS             (none)
0       top_dir         (none)


Totals grouped by language (dominant language first):
java:          8696 (99.85%)
lisp:            13 (0.15%)



Total Physical Source Lines of Code (SLOC)                = 8,709
Development Effort Estimate, Person-Years (Person-Months) = 1.94 (23.29)
```

```
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                        = 0.69 (8.27)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 2.82
Total Estimated Cost to Develop                          = $ 262,185
 (average salary = $56,286/year, overhead = 2.40).
SLOCCount is Open Source Software/Free Software, licensed under the FSF GPL.
Please credit this data as "generated using 'SLOCCount' by David A. Wheeler."
```

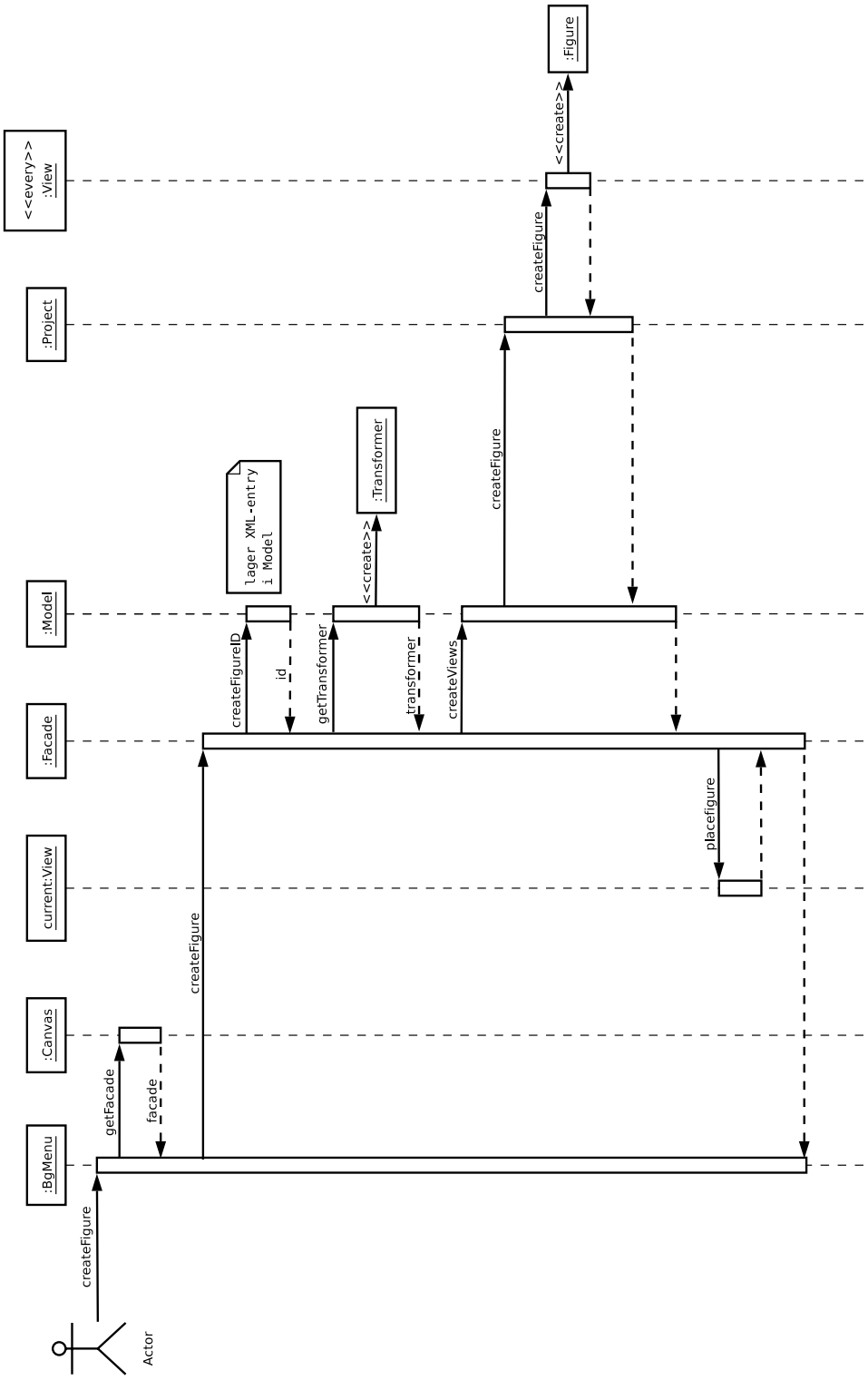
Lisp-koden som er nevnt i utskriften er konfigurasjon av Emacs' java-modus, og tilhører ikke applikasjonen.

I tillegg til Java-koden finnes det endel XSLT-kode som ikke er kommet med i beregningene fordi `sloccount` ikke kjenner XSLT.

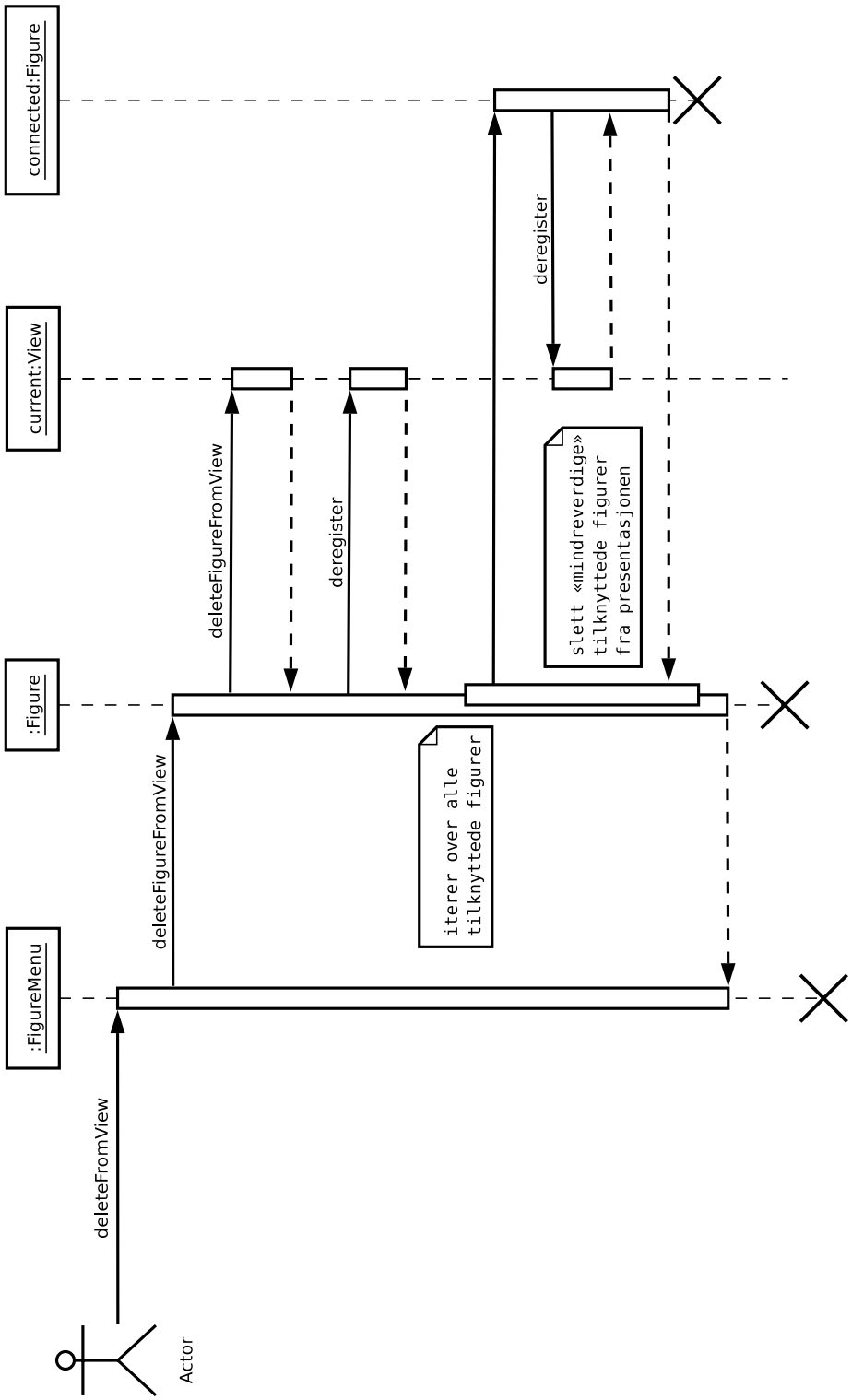
Det finnes ca 1400 linjer XSLT-kode (inkludert blanke linjer og kommentarer).

```
$ find . -name \*.xslt|xargs wc -l
..
..
 1417 total
```

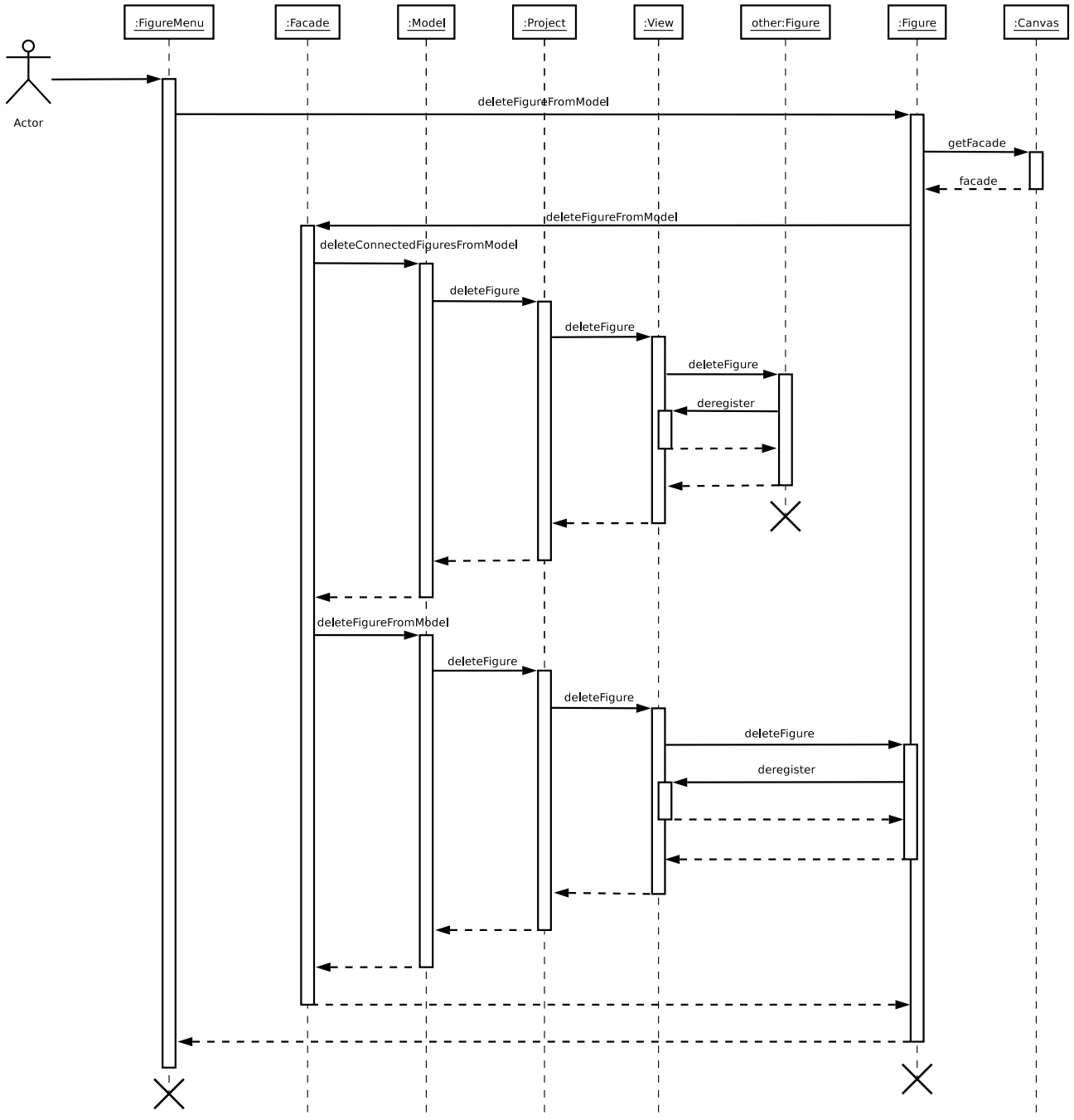
Tilsammen blir dette omtrent 10.000 linjer kildekode. Denne summen er imidlertid bare et overslag, siden blanke linjer og kommentarer er tatt med i antall XSLT-kodelinjer, men ikke tatt med i antall Java-kodelinjer.



Figur 10.1: Opprettelse av ny figur



Figur 10.2: Sletting av figur fra presentasjon



Figur 10.3: Sletting av figur fra modell

Kapittel 11

Diskusjon

Målet for oppgaven var å lage en datamodelleringsverktøy med en nøytral modellrepresentasjon i XML, der XSLT skulle bli brukt for manipulering av modellrepresentasjonen.

Selv om verktøyet ikke er fullendt, vil jeg påstå at jeg har lykkes med dette.

Jeg vil gå igjennom plattformen og beskrive mine erfaringer med de ulike bitene fra utviklingen.

11.1 Metamodell

Ved å begrense ORM-metamodellen til å bare tillate binære utsagn, ble den både mer kompakt og mer intuitiv enn Halpins metamodell[27].

Disse begrensningene har dog ikke utelukkende positive konsekvenser.

11.1.1 Ringskranker på begrepsdannelser

Jeg har kategorisert ringskranker som «assosiasjons-skranker» og i XML gitt disse skrankene mulighet til å referere til en assosiasjon. Denne løsningen gjør det umulig å legge en ringskranke på en assosiasjon i en begrepsdannelse. Dette skyldes at med en gang assosiasjonen blir gjenstand for begrepsdannelse, så blir den i modellen splittet opp til konstruksjonen vist i figur 5.2 på side 31. I denne konstruksjonen finnes det ikke en assosiasjon som går til og fra samme begrep. Problemet kan løses ved å la ringskranker referere til to roller istedenfor en assosiasjon.

11.1.2 ORMML

Starlab har brukt Halpins metamodell fra "Conceptual Schema & Relational Database Design"[31] til å lage ORMML[21] som er en XML-struktur (definert i XSD) for å uttryke ORM. Dessverre var denne metamodellen for enkel for mitt bruk, og ORMML kunne derfor ikke brukes.

11.2 XML

Modellrepresentasjonen i XML ble lagd på grunnlag av metamodellen, med noen tilpasninger mot applikasjonen.

11.2.1 XML-mal

Idag finnes det en felles DTD for modell og presentasjon. For å unngå kollisjoner i navnerommet kunne modell og presentasjon med fordel vært spesifisert i hver sin DTD. På denne måten kunne `model`-elementene tilhørt et annet navnerom enn `view`-elementene.

11.2.2 Mislykket forenkling av XML-modellen

Under utvikling av DTD fant jeg på en "smart" løsning for å øke lesbarheten av XML-modellen. Pga en metamodell med kun binære assosiasjoner var det mulig å uttrykke intern entydighet og intern påkrevd rolle som attributter i rollene. Dette resulterte i en mindre "ordrik" modell og økt lesbarhet. Ulempen med denne løsningen var at de to nevnte skrankene kunne uttrykkes på to forskjellige måter. Den ene måten som attributt i `role` og de andre måten som et eget element under `constraints`.

I ettertid fant jeg ut at dette ble et stort problem. Alle interne påbudte roller og interne entydighets-skranker var uten ID, og alle figurobjekter trengte en ID for å kunne gjøre spørringer mot presentasjon og modell.

Denne løsningen ble forkastet til fordel for den opprinnelige løsningen som er mer ordrik, men konsistent.

11.2.3 XML vs relasjonsdatabase

Å sammenligne XML med en relasjonsdatabase er ikke en rettferdig sammenligning: XML er et format å strukturere data, en relasjonsdatabase innebærer et databasehåndteringssystem (RDBMS) som kan håndtere transaksjoner og håndheve integritetsskranker.

Jeg vil allikevel sammenligne disse to siden jeg i dette tilfellet betrakter relasjonsdatabase som et reelt alternativ til XML.

XML tilbyr kun datatypen ID for å håndheve unikhet. Denne datatypen kan bare forekomme en gang pr element.

For å peke på ID'er tilbyr XML datatypene IDREF og IDREFS som peker på respektive en og flere ID'er. Det er ikke mulig å si hvor mange ID'er en IDREFS får lov til å peke på. Pekerene er ikke typede, og det er dermed ikke mulig å spesifisere at man ønsker at en rolle kun skal kunne peke på en representasjon eller et begrep.

Mangelen av typede pekere gjør at veldig mye av det som ville vært et relasjonsdatabasehåndteringssystems ansvar må håndteres i applikasjonen.

Ved implementering av skranker hadde det vært ønskelig å kunne deklarere hva som var lovlig "plassering" av skranken (se kapittel 4 på side 22). Siden dette ikke kunne håndteres av XML, ble jeg nødt til å kontrollere dette på applikasjonsnivå. I en relasjonsdatabase kunne "deklarasjonene" av skranker vært lagt inn som *triggere*.

Skranker som gjør andre skranker overflødige kan heller ikke håndteres av DTD/XML, og må også flyttes opp på applikasjonsnivå. Dette er også mulig å håndtere ved hjelp av *triggere* i en relasjonsdatabase.

Jeg kunne imidlertid ha unngått mange av disse problemene ved å lage et XSLT-bibliotek for å håndtere typede-pekere o.l. på "XML-nivå". Dette ville imidlertid tatt lang tid å programmere og ført til tidkrevende kontrollering av XML-koden i applikasjonen.

På den andre siden er XML et portabelt format som gir gode muligheter til eksportering til andre filformater f.eks. SVG, Dia og XMI.

XML gjør det også lett å lage testdata, siden XML-koden lagres i filer. Editering av data i en relasjonsdatabase ville vært mye mer tungvint.

På grunn av manglende håndtering av skrankeintegritet i XML er en relasjonsdatabase å foretrekke fremfor XML. Arkitekturen som er brukt tillater imidlertid å skifte ut det persistente lageret uten at det skulle påvirke de andre programvarelagene i noen nevneverdig grad.

Jeg har ikke fått tid til å undersøke om en XML-database ville løst noen av problemene jeg har funnet ved bruken av XML.

11.3 XSLT

Før jeg endte opp med å bruke XSLT 2.0, prøvde jeg å skrive identifiseringsalgoritmen i XSLT 1.0. Dette ble veldig komplisert og veldig vanskelig å vedlikeholde.

Dagens «Working Draft» av XSLT (XSLT 2.0) har like mye "linjestøy" som XSLT 1.0, men byr på muligheten til å lage funksjoner som kan kalles fra XPath-uttrykk. I tillegg er alt i XSLT 2.0 av datatypen sekvens (inkludert resultater).

XSLT 2.0 kan programmeres funksjonelt. Dette impliserer at det ikke skulle være noen begrensninger i språket.

Ved å bruke XSLT til manipulering av modellrepresentasjonen blir det et klart skille mellom de delene av applikasjonen som har ansvar for opptegning, og de som har ansvar for manipulering av de elementære utsagnene.

På den andre siden kreves det kunnskap om både XSLT og Java for å kunne videreutvikle verktøyet.

XSLT 2.0 er i skrivende stund ikke en «W3C recommendation». Det kan ses på som negativt å bruke denne funksjonaliteten før den har stabilisert seg og blitt en standard. Jeg ha imidlertid bare hatt positive opplevelser med XSLT 2.0 i forhold til XSLT 1.0

11.4 Biblioteker

11.4.1 JHotDraw

JHotDraw (se kapittel 2.7.1 på side 16) så ut som et rammeverk som kunne komme til nytte. Dessverre var dokumentasjonen så mangelfull at jeg estimerte at det ville ta mindre tid på å skrive alt fra bunnen enn å lære seg JHotDraw.

11.4.2 SVG og Batik

SVG[6] er en XML-grammatikk for å beskrive vektorbasert 2D grafikk. Selv om dette er en «W3C recommendation» finnes det fremdeles ikke mange implementasjoner av dette.

Jeg har undersøkt om SVG[6] kunne brukes for opptegning av figurene i applikasjonen. Dette er det mulig å gjøre ved hjelp av Batik[16], en SVG-motor skrevet i Java.

SVG inneholdt fremdeles mange "barnesykdommer", f.eks. var det ikke mulig å finne ut hvor bred en tekst ville bli. Det var heller ikke mulig å la en strek ha et endepunkt i hver sin gruppe (noe som ville vært aktuelt for f.eks. en strek mellom en rolle og et begrep).

Jeg fant ut at SVG ville gjort arkitekturen unødvendig kompleks. Alle symboler ville bestått av to «komponenter»: en SVG-del og en Java-del (som ville vært avhengig av SVG-delen) for å håndtere forretningreglene. På grunn av mangler ved SVG ville komponentene fått overlappende ansvarsområder.

Siden SVG var det eneste argumentet for å bruke Batik, ble heller ikke Batik tatt i bruk.

Selv om SVG ble forkastet som intern rendrings-motor ser jeg nytten av å kunne eksportere et diagram til SVG.

Under utviklingen laget jeg et JavaScript som gjorde det mulig å se på og endre layout til et diagram ved hjelp av en nettleser som støttet SVG. På grunn av problemer med SVG-implementasjonen i nettleseren ble dette aldri fullført.

11.5 Programmeringsmiljø

Programmeringsmiljøet mitt har bestått av Emacs med JDEE (java-modus), Sun's javakompilator og jikes (javakompilator).

Emacs med JDEE kan kalles en lettvekts IDE (`Integrated Developer Environment`), men mangler gode refaktoreringsmuligheter. Dette bærer dessverre kildekoden preg av idag.

To forskjellige javakompilatorer ble brukt. `jikes` ble brukt fordi den er veldig rask, og `javac` (Sun's javakompilator) ble brukt fordi den gir gode feilmeldinger.

11.6 Figur API

Jeg har i min realisering av arkitekturen brukt `no.uio.ifi.Niamderthal.common.figures.Symbol`, som supertype for alle figur-klasser. Siden Java ikke tilbyr multippel arv mister alle figur-objektene muligheten til å subtype noe annet enn denne klassen.

Dette kan løses ved å la `no.uio.ifi.Niamderthal.common.figures.Symbol` være et «interface» (istedenfor en klasse) som alle figur-klassene implementerer. For enkelhets skyld kan jeg legge til `no.uio.ifi.Niamderthal.common.figure.SymbolImpl` som er en

klasse som implementerte dette grensesnittet. Denne klassen kan bli subklasset av klasser som ikke trenger subklassings-muligheten til noe annet.

11.7 Alternativ arkitektur

Slik arkitekturen er nå må figurene via et mellomvarelag, **Transformer**, når de skal gjøre spørringer mot modellen.

En figur spør presentasjonen for å få vite sin plassering, men spør sin transformator for å f.eks. få vite hvilke "barn" den har. Dette impliserer at figurene vet om skillet mellom modell og presentasjon.

Ved å la **Transformer** være et mellomvarelag for både modellen og presentasjonen, ville ikke figurene trengt å vite hva som tilhører presentasjon og hva som tilhører modellen, eller at det i det hele tatt eksisterte et slikt skille. Figurene ville da gjort alle spørringene sine mot ett objekt, og dette objektet ville fått ansvaret for å delegere spørringene videre. Jeg tror dette ville resultert i en "renere" implementasjon av de ulike modellkonstruksjonene/-abstraksjonene.

11.8 Forbedringer og utvidelser

11.8.1 Lag

For å gjøre modeller mer oversiktlige hadde det vært greit å kunne dele den opp i flere lag. En spennende utvidelse av dette hadde vært å ha de ulike "delmodellene" i hvert sitt lag, for så å kunne se på assosiasjonene som fantes mellom disse lagene. Dette ville vært assosiasjoner som knyttet sammen komponentene i modellen, og ville vært egnet som oversikt over modellen.

11.8.2 Ark

For å gjøre en modell egnet for utskrift hadde det vært greit å kunne spre den utover flere ark.

11.8.3 Undo

Et verktøy er ikke egnet for brukere før det har en angre-funksjon. Denne funksjonen kan implementeres ved hjelp av Command-mønsteret[25].

11.8.4 Behandling av skranker under gruppering

Algoritmen for gruppering som er beskrevet i denne oppgaven tar kun hensyn til skrankene som er med på å bestemme utfallet av grupperingen. De resterende skrankene blir utelatt.

For å få en 1:1 avbildning mellom de elementære utsagnene og gruppene må den overordnede grupperingsalgoritmen utvides.

Bibliografi

[1] Extensible Markup Language (XML). http://www.w3.org/TR/WD-xml-961114.html (sist lest 2004-02-07), 1996.

[2] XML Path Language (XPath) version 1.0. http://www.w3.org/TR/xpath (sist lest 2004-02-07), 1999.

[3] XSL Transformations (XSLT) version 1.0. http://www.w3.org/TR/xslt (sist lest 2004-02-07), 1999.

[4] Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/REC-xml (sist lest 2004-02-07), 2000.

[5] All DOM level 2 recommendations. http://www.w3.org/DOM/DOMTR#dom2 (sist lest 2004-02-07), 2000/2003.

[6] Scalable Vector Graphics (SVG) 1.1 Specification. http://www.w3.org/TR/SVG11/ (sist lest 2004-02-07), 2003.

[7] XPath requirements version 2.0. http://www.w3.org/TR/xpath20req (sist lest 2004-02-07), 2003.

[8] XSL Transformations (XSLT) version 2.0. http://www.w3.org/TR/xslt20 (sist lest 2004-02-07), 2003.

[9] Dom4J. http://dom4j.org, sist lest 2004-02-06.

[10] DTD tutorial. http://www.w3schools.com/dtd/, sist lest 2004-02-06.

[11] Log4j. http://logging.apache.org/log4j/docs/, sist lest 2004-02-06.

[12] Sun's online java tutorial. http://java.sun.com/docs/books/tutorial/, sist lest 2004-02-06.

[13] http://www.posc.org/ebiz/resources/glossaryMain.html, sist lest 2004-02-07.

[14] http://mathworld.wolfram.com/Relation.html, sist lest 2004-02-07.

[15] http://www.w3.org/TR/REC-xml-names/#Conformance, sist lest 2004-02-07.

[16] Batik. http://xml.apache.org/batik/, sist lest 2004-02-07.

[17] XML Schema. http://www.w3.org/XML/Schema, sist lest 2004-02-07.

[18] http://www.csc.liv.ac.uk/ ped/teachadmin/algor/d_and_c.html, sist lest 2004-02-12.

[19] http://www.nist.gov/dads/HTML/divideconqr.html, sist lest 2004-02-12.

[20] http://www.metamodel.com/staticpages/index.php?page=20021010231056977, sist lest 2004-02-15.

[21] Jan Demey, Mustafa Jarrar og Robert Meersman. A Markup Language for ORM Business Rules. Rapport, STAR Lab, 2002.

[22] Ramez Elmasri og Shamkant B. Navathe. *Fundamentals of database systems*. Benjamin-Cummings Publishing Co., Inc., andre utgave, 1994. ISBN 0-8053-01753-8.

[23] Dagfinn Føllesdal, Lars Walløe og Jon Elster. *Dataorientert systemutvikling*. Universitetsforlaget, 1990.

[24] Martin Fowler og Kendall Scott. *UML distilled (2nd ed.): a brief guide to the standard object modeling language*. Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 0-201-65783-X.

[25] Erich Gamma, Richard Helm, Ralph Johnson og John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[26] Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction*. Addison-Wesley Longman Publishing Co., Inc., 1994. ISBN 0201600447.

[27] Dr. Terry Halpin. An ORM metamodel. *Journal of Conceptual Modeling*, 16, october 2000.

[28] Terry Halpin. Object-role modeling: an overview. http://www.orm.net/pdf/ORMwhitePaper.pdf (sist lest 2004-02-15).

[29] Terry Halpin. What is an elementary fact? http://www.orm.net/pdf/ElemFact.pdf, sist lest 2004-02-15.

[30] Terry Halpin, Linda Bird og Andrew Goodchild. Object Role Modelling and XML-Schema. 2000.

[31] Terry A. Halpin. *Conceptual Schema & Relational Database Design*. Prentice Hall Australia, 1995.

[32] Brian W. Kernighan og Rob Pike. *The Practice of Programming*. Addison-Wesley, tredje utgave, 1999.

[33] Ragnar Normann og Gerhard Skagestein. Revival of the elementary sentence - or the dark side of UML class diagrams. 2003.

[34] C. K. Ogden og I. A. Richards. The meanind of meaning: a study of the influence of language upon thought and of science of symbolism. *Routledge & Kegan Paul*, London 1949.

[35] Matthew Robinson, Pavel Vorobiev, Pavel A. Vorobiev og David Anderson. *Swing, Second Edition*. Manning Publications Company, andre utgave, 2000. ISBN 193011088X.

[36] Gerhard Skagestein. *Dataorientert systemutvikling*. Universitetsforlaget, 1996.

[37] Gerhard Skagestein. *Systemutvikling - fra kjernen og ut, fra skallet og inn*. Høyskoleforlaget, 2002.

[38] Gerhard Skagestein og Arild Thorvaldsen. *Fra virkelighet til datamodell*. Universitetsforlaget, 1986.

[39] Kim Topley. *Core Swing: Advanced Programming*. Prentice Hall PTR, 1999. ISBN 0130832928.

[40] J. J. V. R. Wintraeken. *Informatie-analyse volgens NIAM in theorie en praktijk*. Academic Service, Schoonhoven, 2002.

[41] John Zukowski. *Definitive Guide to Swing for Java 2, Second Edition*. APress, andre utgave, 2000. ISBN 189311578X.