UNIVERSITETET I OSLO
Institutt for informatikk

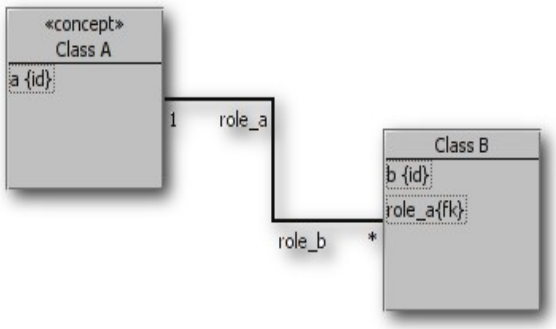
Grafisk editor for automatisk gruppering og degruppering av dataorienterte klassediagrammer

Masteroppgave

Øyvind Stegard

19. desember 2005

Grafisk editor for automatisk gruppering og degruppering av dataorienterte klassediagrammer



Øyvind Stegard <oyvinst@ifi.uio.no>

Sammendrag

Denne oppgaven omhandler utviklingen av et grafisk verktøy for automatisk gruppering- og degruppering av enkle dataorienterte UML klassediagrammer. Grupperingen vil resultere i klasser som kan brukes som utgangspunkt for et relasjonsdatabaseskjema for modellen. Verktøyet er tenkt brukt som et læremiddel i systemutviklingskurs. Som grunnlag for databehandlingen ligger en metamodell som beskriver de støttede UML konseptene. Verktøyet bruker XML som format ved persistering av modeller, og XSLT-2.0 benyttes for å gjøre grupperings-transformasjoner på disse modellene. Det grafiske verktøyet er implementert på Eclipse-plattformen.

Indekstermer

Datamodellering; gruppering; degruppering; relasjonsdatabase; optimal normalform; transformasjon; XML; XSLT; BCNF; UML; klassediagram; Eclipse; GEF; editor; dataorientert klassediagram; elementære utsagn; stereotyper; begreper; NIAM; ORM

Forord

Under utviklingen og skrivingen har jeg hele tiden hatt ukentlige møter med min veileder, Gerhard Skagetein. Jeg vil i den sammenheng rette en stor takk til ham for den innsikten og entusiasmen han har vist. Jeg vil også takke Jon Oldevik for gode tips i forbindelse med programvare han har skrevet [1], som en del av denne oppgaven omtaler.

Konvensjoner og språk

Jeg har valgt å skrive denne oppgaven på norsk, men har benyttet meg av engelsk i all kildekode, dvs. kommentarer, klassenavn, etc. Prototypens brukergrensesnitt er også 100% engelskspråklig, da jeg fant dette mest naturlig. Følgelig er også figurer tilknyttet applikasjonen hovedsakelig på engelsk. Jeg har generelt ingen sterke politiske oppfatninger angående bruken av engelske ord og uttrykk i en norskspråklig tekst som omhandler temaer innenfor informatikk. Jeg vil derfor skrive det som faller meg mest naturlig, dvs. bruke de engelske ordene der de er best kjent, og norsk der det finnes gode oversettelser.

Det refereres flere steder til «prototypen», «applikasjonen», «programmet» eller «editoren» i teksten. Med disse ordene sikter jeg alltid til prototypen som er laget i forbindelse med denne oppgaven, om ikke annet er nevnt.

Jeg vil generelt markere alle navn, ord, uttrykk og utdrag i forbindelse med kildekode og implementasjon `som dette`. Henvisninger til navn og ord brukt i figurer er **skrevet med tykk skrift.** Alle nøkkelord og sitater som forekommer i oppgaven, er *skrevet i kursiv.*

Innholdsliste

1 Innledning

1.1 Problemstilling

Hovedmålet med oppgaven er å lage en fungerende prototype av et grafisk verktøy for å jobbe med grupperings-operasjoner på dataorienterte UML klassediagrammer [2]. Slike diagrammer er en profil av UML klassediagrammer beskrevet i [3][1]. Ved hjelp av verktøyet lager brukeren en modell ved å legge inn klasser og definere assosiasjoner mellom disse. Klassene modelleres på et konseptuelt nivå (stereotyper [3][2]). Kun attributtet som er nødvendig for å identifisere en forekomst legges inn. Ytterligere egenskaper modelleres ved hjelp av assosiasjoner og roller til andre klasser. Fremgangsmåten har likheter med «Object Role Modeling» [4], [5]. Verktøyet skal kunne gruppere valgte klasser automatisk etter grupperingsalgoritmen som er beskrevet i [3], avsnitt 5.4. Resultatet skal bli UML-klasser (grupper av attributter) som vil kunne tilsvare tabelledefinisjonene (med kandidatnøkler) i en relasjonsdatabase på Boyce-Codd normalform [6]. Editoren skal visuelt presentere endringene som skjer ved gruppering/degruppering.

Målgruppen for bruk av prototypen er studenter som tar kurs i datamodellering og systemutvikling. På bakgrunn av dette bør brukergrensesnittet være enkelt, pedagogisk og fokusere på kjernen av problemstillingen. Studenter skal visuelt kunne observere hvordan gruppering fungerer på enkle modeller. Behovet for å støtte avanserte modelleringskonsepter er derfor mindre viktig i en slik sammenheng. Dette, sammen med behovet for å begrense oppgavens omfang, er med på å rettferdiggjøre de forenklingene som er gjort.

1.2 Rammer

En tidligere hovedoppgave skrevet av Eirik Meland [7], beskriver et modelleringsverktøy der modeller er representert med ORM/NIAM dialekten [5]. Verktøyet bruker XML [8] som intern datastruktur. Jeg har latt meg inspirere av denne oppgaven, da den behandler noen problemstillinger som er sentrale for min egen oppgave.

Prototypen som er laget bruker, i likhet med Melands verktøy, XML [8] som datastruktur for persistering av modelldata og ved utføring av transformasjoner. Jeg har valgt å basere meg på XSLT-2.0 [9] for å gjøre grupperingstransformasjonene. I denne sammenheng vil jeg se på egnetheten av XSLT-2.0 til dette formålet, samt gjøre noen sammenlikninger med den tidligere utgaven av språket, XSLT-1.0. Jeg vil også undersøke noen andre verktøy, språk og rammeverk for å gjøre modelltransformasjoner.

Eclipse-plattformen [10] er brukt for å realisere editoren og det grafiske brukergrensesnittet. I denne sammenheng vil jeg se på rammeverket Graphical Editing Framework (GEF) [11], som prototypen benytter seg av for grafisk presentasjon og brukerinteraksjon.

1 Kap. 5
2 Kap. 5, s. 122

1.3 Oversikt over oppgaven

Innledning

Her beskrives målet med oppgaven og problemstillingene som behandles.

Gruppering og degruppering av klasser

I dette kapitlet beskrives systematisk fremgangsmåtene som brukes ved gruppering av klasser i forskjellige tilfeller. Som en introduksjon til kapitlet vil jeg beskrive noen modelleringskonsepter som er sentrale for oppgaven.

Metamodell og hoveddatastruktur

I dette kapitellet beskrives metamodellen som ligger til grunn for modelleringen i applikasjonen. Deretter beskrives XML-formatet som benyttes for lagring og kjøring av transformasjoner på modeller.

Oppbygning av prototypen

Dette kapitlet handler om sentrale programkomponenter i den grafiske editoren som er laget og oppbygningen av prototypen som en helhet.

Modelltransformasjoner med XSLT

I dette kapitlet går jeg gjennom bruken av XSLT som transformasjonsspråk ved gruppering og degruppering av klasser.

Alternative verktøy og rammeverk for modelltransformasjoner

I dette kapitlet ser jeg på noen alternativer til språk og programvare som kan brukes for å gjøre modelltransformasjoner.

Diskusjon og fremtidige utvidelsesområder

I dette kapitlet beskrives mulige utvidelsesområder for prototypen som er laget.

Oppsummering

Her vil forsøke å oppsummere de viktigste temaene i oppgaven, samt se på gjennomføringen av oppgaven med hensyn på målet.

1.4 Forutsetninger

Det vil være en fordel for leseren med forkunnskaper om følgende tema:

- Datamodellering generelt. Kapittel 2.1 gir en liten introduksjon til de mest relevante konseptene for oppgaven.
- Kunnskap om sentrale begreper innenfor UML [2]og XML-relaterte teknologier[8].
- Generell kunnskap om programmeringsspråk er nødvendig for å kunne forstå hvordan prototypen er laget. Kjennskap til Java, Eclipse plattformen og XSLT er en fordel.

2 Gruppering og degruppering av klasser

I dette kapitlet vil jeg først ta for meg noen konsepter innenfor datamodellering som er sentrale for oppgaven. Dette vil være en introduksjon til hovedtemaet: gruppering og degruppering av klasser. Jeg vil hovedsakelig fokusere på problemstillingene som er relevante for metamodellen som beskrives i kapittel 3 og de modelleringskonseptene som støttes av prototypen.

2.1 Datamodellering

> *«In information system design, data modeling is the analysis and design of the information in the system, concentrating on the logical entities and the logical dependencies between these entities.» - [12] Wikipedia.org*

Når vi lager en datamodell er det fordi vi ønsker systematisk og nøyaktig å kunne beskrive en del av virkeligheten som vi ser på som interessant. Denne «delen» vi snakker om kaller vi gjerne for *interesseområdet [3]*. En datamodell vil alltid være en forenkling av den fulle og hele sannhet, i større eller mindre grad, derav ordet «modell», men den bør likevel dekke 100% av de begrepene som vi anser som relevante og nødvendige å kunne beskrive. I informatikken snakker vi gjerne om modellering i sammenheng med utforming og design av informasjonssystemer som skal kunne behandle og lagre data om fenomenene innenfor interesseområdet.

2.1.1 Begreper og representasjoner

Et *begrep* ("concept") er en type ting eller fenomen innenfor interesseområdet *[3]*[3]. Et typisk eksempel er begrepet «Person». Et begrep kan ha mange begrepsforekomster, i tilfellet «Person» konkrete personer som Per og Pål, far og mor, statsministeren og Stortingspresidenten. Hvis vi prøver å vise betydningen av ordet begrep som et spørsmål, kunne det kanskje vært «Hva er de sentrale tingene i vårt interesseområde som vi ønsker å beskrive i modellen ?». Eksempler på noen typiske begreper innenfor en modell av et betalingssystem kan være «Valuta», «Kunde», «Konto» og «Transaksjon». Begreper kan være generiske og abstrakte, som f.eks. «Objekt», «Tall» og «Ting». De kan også være svært spesifikke, som «Forgasserdel» i en bilmotor eller «Pengemarkedsfond» i et finanssystem. Hvilket nivå man ønsker å legge seg på er avhengig av målet med modellen og omfanget av interesseområdet, og er noen ganger en vanskelig balansegang.

Begrepsforekomster kan imidlertid ikke lagres i en datamaskin eller skrives ned på papir. Istedenfor lagrer vi forekomster av deres *representasjoner [3]*. Eksempelvis blir personer ofte representert ved hjelp av fødselsnumre, biler ved kjennetegn, bankkonti ved kontonumre, land ved en landkode og så videre. Når vi setter opp modellen bør vi bruke representasjonen som entydig peker ut de tilsvarende begrepsforekomstene. Slike representasjoner kaller vi indetifikatorer. Samfunnet har etablert en lang rekke representasjonssystemer som tilfredsstiller dette kravet, for eksempel finnes det (forhåpentligvis) ikke flere personer med samme fødselsnummer.

3 Kapittel 5.2

Imidlertid forekommer det ofte at representasjonen er entydig bare innenfor en bestemt kontekst eller i et bestemt navnerom, eksempelvis er kjennetegn på biler kun entydige innenfor hvert enkelt land (det er motorvognregistreringsmyndighetene ansvarlige for). Hvis interesseområdet omfatter flere land, må vi bruke en mer komplisert identifikator; nærliggende er sammensetningen av en landkode og kjennetegnet.

2.1.2 Assosiasjoner og elementære utsagn

> *«Utsagn som ikke kan gjøres kortere uten at vi mister meningsinnholdet, kaller vi elementære utsagn («elementary sentences»). I de aller fleste tilfeller vil dette være utsagn med to roller, såkalte binære utsagn («binary sentences»).*» - Skagestein [13]

Begreper er knyttet sammen med *assosiasjoner* som uttrykker hva begrepene har med hverandre å gjøre. Eksempelvis kan interesseområdet omfatte eiendomsforhold mellom personer og biler. I så fall trenger vi assosiasjonen «eier» eller «eiendomsforhold» mellom Person og Bil. En konkret forekomst av denne assosiasjonen kan være at «Person med fødselsnummer 12345678901 eier bil med kjennetegn DD-123455».

En assosiasjon kan knytte sammen et vilkårlig antall begreper. Imidlertid er vi mest interessert i de assosiasjonene som ikke kan plukkes fra hverandre i mindre assosiasjoner uten at meningsinnholdet går tapt. En slik assosiasjon med tilhørende begreper kalles et *elementært utsagn.* Det viser seg at de aller fleste elementære utsagn i en modell er binære [3], det vil si at assosiasjonen knytter sammen to begreper (eller knytter begrepet sammen med seg selv). Høyere ordens elementære utsagn kan alltid reduseres til et antall binære elementære utsagn ved å innføre nye begreper, se [3][4].

2.1.3 Modellering med elementære utsagn

Dette er den grunnleggende ideen bak ugrupperte datamodeller, nemlig *å bygge opp modellen utelukkende ved hjelp av elementære utsagn [3]*. Denne type datamodeller er sentrale i metoder som NIAM og ORM [5], men ideen forekom faktisk så tidlig som på slutten av 70-tallet i det eksperimentelle databasehåndteringssystemet CS4 [3][5],[14].

Fordelen med ugrupperte datamodeller er at vi etterpå kan gruppere dem på nøyaktig den måten vi vil. Den mest aktuelle formen for gruppering er gruppering til Boyce-Codd-normalform (BCNF) [6] som egner seg godt som grunnlag for utforming av en relasjonsdatabasestruktur. Også på andre måter kan denne formen virke nokså naturlig, ved at elementære utsagn som man føler hører sammen også grupperes sammen.

Ulempen med ugrupperte datamodeller er at de kan bli svært omfattende, nettopp fordi den basale byggesteinen (det elementære utsagnet) er så liten. Mange vil nok føle at dette ikke er den mest intuitive måten å lage modeller på. Derfor hadde det vært ideelt med et verktøy som gjorde det mulig å blande ugrupperte og grupperte utsagn i samme modell. Det er nettopp dette behovet som verktøyet i denne oppgaven skal dekke.

4 Kapittel 6.2.
5 Side 150.

2.1.4 Skranker

På modellen kan vi legge skranker av ulike typer, som for eksempel hvilke begrepsforekomster og representasjoner som er lovlige (eksempelvis kan en dato ikke ha et høyere dagnummer enn 31). Av størst interesse for denne oppgaven er multiplisitetene, som begrenser hvor mange forekomster av et begrep som kan knyttes til et annet begrep gjennom en assosiasjon. Eksempelvis kan vi gjennom en multiplisitetsskranke uttrykke at en bil skal være registrert i minst ett, og høyst ett (det vil si nøyaktig ett) land.

2.1.5 Modelleringsspråk

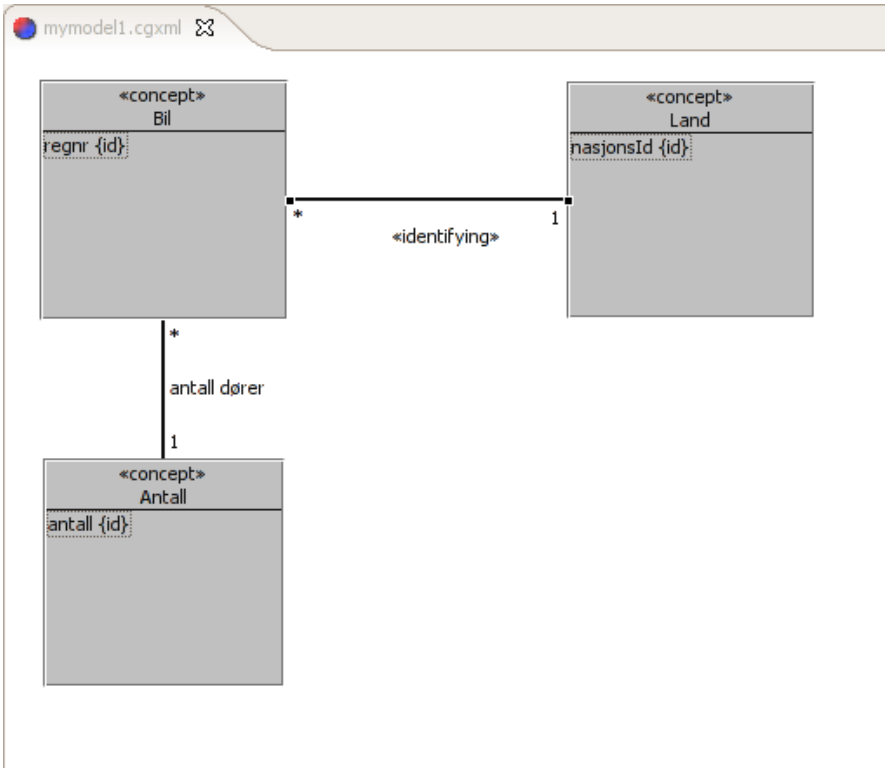
> *«Short for **U**nified **M**odeling **L**anguage, a general-purpose notational language for specifying and visualizing complex software, especially large, object-oriented projects.» [15] - Webopedia.com*

Datamodeller kan dokumenteres på mange ulike måter – en utbredt variant er de såkalte Entity-Relationship-diagrammene. Jeg skal imidlertid følge Skagestein*[3]* og bruke UML-klassediagrammer for dette formålet. Siden modellen er en ren datamodell[6], vil klassene i denne modellen imidlertid ikke ha metoder.

UML[2] er et rikt modelleringsspråk for å beskrive mange forskjellige aspekter ved et informasjonssystem.

6 Når man utvikler større systemer er det fordelaktig å kunne beskrive dynamiske aspekter som for eksempel objektinteraksjoner, lovlige tilstander, tilstandsoverganger og aktiviteter som foregår i systemet. UML er et omfattende språk som kan brukes for å beskrive alle disse egenskapene ved større programsystemer. Temaer rundt dette omtales ikke i denne oppgaven. Det er kun enkle UML klassediagrammer som er brukt.

2.1.6 Eksempel på modell med begreper



Figur 1: Eksempel på en ugruppert modell som bare består av begreper. Figuren er laget med prototypen.

Jeg har valgt å ta med et lite eksempel på modellering med elementære utsagn og begreper for å klargjøre dette ytterligere. Jeg antar her at hvert land i verden har et nummersystem for bilskilter, samt at alle disse systemene har den egenskapen at biler innenfor landet kan identifiseres ved hjelp av nummeret. Modellen i Figur 1 inneholder tre begreper: **«Bil»**, **«Land»** og **«Antall»**. **«Bil»** er *representert* ved attributtet **«regnr {id}»**, som er en unik tekststreng for alle biler innenfor ett enkelt land. La oss anta at modellen skal kunne lagre informasjon om biler fra hele verden. Da holder det sannsynligvis ikke å bare bruke bilens registreringsnummer som identifikator. Her kommer begrepet **«Land»** inn i bildet, representert ved attributtet **«nasjonsId {id}»**. **«Land»** er et ganske generelt begrep, som potensielt kunne vært knyttet til mange andre begreper innenfor samme modell. I vår modell er vi kun interessert i den uniken koden for hvert land, som sammen med et registreringsnummer lar oss identifisere en forekomst av en bil på verdensbasis. Assosiasjonen mellom biler og land er definert som **«identifying»** for begrepet **«Bil»**. Det betyr at en forekomst av en bil er avhengig av en forekomst av et land (og dermed at det eksisterer en **«nasjonsId»** for dette landet, i form av to tegn) for å kunne ble unikt identifisert med 100% sikkerhet[7]. Hvis vi skulle modellert det samme på den intuitive

7 I forhold relasjonsdatabase-teorien danner «regnr» «nasjonsId» en kandidatnøkkel for en tabell over biler.

måten ville vi sannsynligvis sagt at en bil hadde et ekstra attributt i sin klasse som inneholdt koden for landet den hørte til.

I eksemplet vårt ønsker vi også å lagre data om hvor mange dører en gitt bil har. For å oppnå dette introduserer vi begrepet **«Antall»**, igjen et generelt begrep. Vi må derfor bruke roller for å tydeligjøre nøyaktig hvorfor et **«Antall»** har en assosiasjon til **«Bil»**. Hvis man tenker seg om så kan begrepet **«Antall»** assosieres med begrepet **«Bil»** på et stort antall forskjellige måter. I vårt enkle eksempel konsentrerer vi oss kun om **«antall dører»**. Vi ser at **«antall dører»** er rollen **«Antall»** spiller overfor **«Bil»** i akkurat denne assosiasjonen. Om vi også hadde ønsket å modellere antall hestekrefter på motoren til en bil, så kunne vi opprettet enda en assosiasjonen mellom **«Bil»** og **«Antall»**, hvor vi lar **«Antall»** spille rollen **«antall hestekrefter».**

Hva multiplisitetene i assosiasjonene angår, ser vi at en bil kan ha én forekomst av **«Antall»** (følgelig ett heltall) som beskriver antall dører, mens et gitt heltall typisk vil være antallet dører for veldig mange biler. Det samme gjelder for assosiasjonen mellom **«Bil»** og **«Land»**[8].

2.2 Gruppering av klasser

Det finnes flere typer grupperinger man kan gjøre når man modellerer [3][9]. I denne oppgaven er det snakk om en relasjonsdatabase-inspirert gruppering som gir klasser som tilsvarer databasetabeller på Boyce-Codd normalform [6].

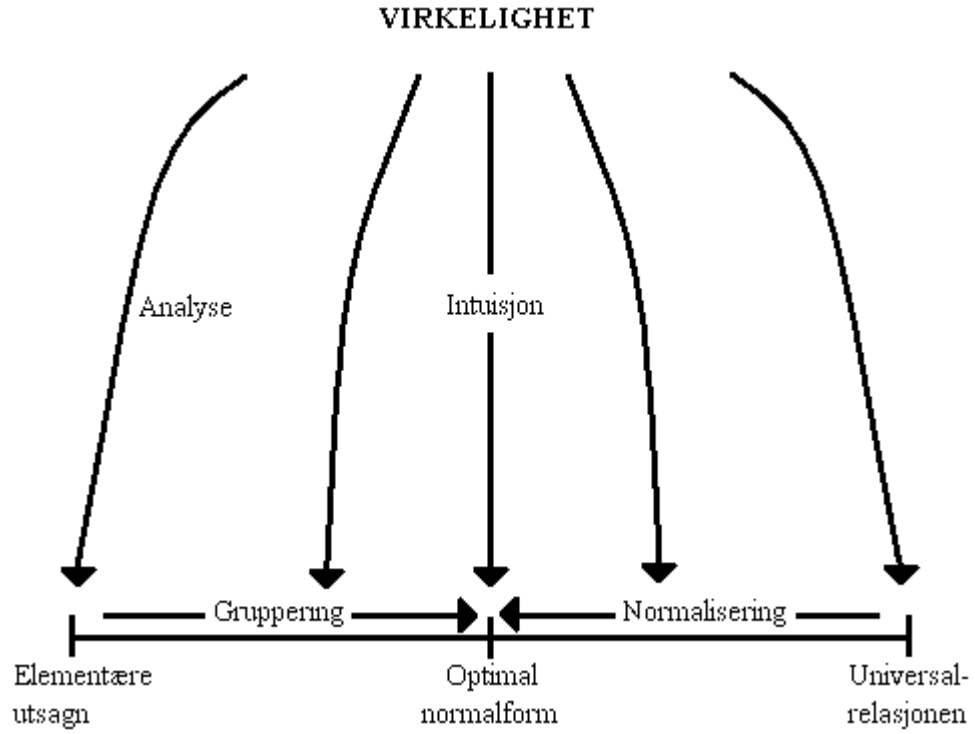
2.2.1 Målet

Målet med grupperingen er å få en mer håndterbar modell med færre små begrepsklasser (se avsnitt 2.1.3) og egenskaper ved klassene modellert som attributter. Man skal også kunne bruke den grupperte modellen til å lage et relasjonsdatabase-skjema. De grupperte klassene vil med andre ord tilsvare skjemaer for tabeller man kan inkludere i databasen. Settet av attributter i en gruppert klasse tilsvarer kolonnene i en tabell. De attributtene som er en del av klassens representasjon (identfikasjon) blir til sammen en kandidatnøkkel. Tabellene vil være på Boyce-Codd normalform [6],[16] (BCNF). I tillegg vil man kunne redusere antall grupper som blir med i databasen. Dette vil typisk være grupper som kun inneholder sin egen identifiserende representasjon etter gruppering, og omtales i avsnitt 2.2.7.

Fordi vi oppnår BCNF, så unngår vi å lagre redundante data av typene som kan forekomme i 1NF, 2NF og 3NF[16]. BCNF kort forklart betyr at attributtverdiene som ikke er en del av kandidatnøkkelen vil være bestemt av kandidatnøkkelen, i sin helhet, og ikke noe annet.

8 Etter nærmere ettertanke kan man kanskje påstå at et serienummer sammen med bil-merke er et bedre alternativ for å identifisere en forekomst av en bil globalt på. Det forandrer imidlertid ikke hovedpoenget med eksemplet, som forsøker å belyse et enkelt tilfelle av begrepsmessig modellering.

9 Kapittel 5.5.2.



Figur 2: Figuren viser to veier mot den optimale normalformen. Den er hentet fra [13]

Det finnes flere metoder man kan bruke for å komme frem til en normalisert databasestruktur [13]. Man kan bruke sin egen intuisjon, og sette opp relasjonene direkte. Deretter kan man bruke normaliseringsteori [3][10] for å bryte opp tabellene i mindre deler, skulle det være nødvendig. Den ekstreme siden av denne fremgangsmåten er å legge alle begreper i samme relasjon, som et utgangspunkt, for deretter å systematisk bryte ned denne i mindre tabeller. Når vi derimot jobber med modellering på det begrepsmessige planet, nærmer vi oss den normaliserte formen fra den motsatte siden. Vi går fra et sett av mange små begreper, assosiert med hverandre på forskjellige måter for å utrykke egenskaper, til en delmenge av de samme begrepene med assosiasjoner trukket inn som attributter etter grupperingsreglene. De grupperte klassene tilsvarer det normaliserte databaseskjemaet for modellen.

2.2.2 Gruppering og fremmednøkler

Følgende sitat fra [3] oppsummerer den grunnleggende grupperingsalgoritmen på en god måte:

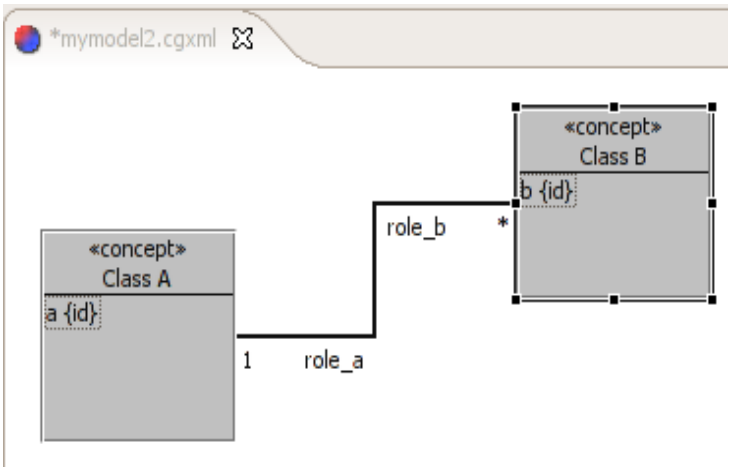
> *«For hver klasse (stereotypiet som begrep), genereres en fremmednøkkel for hver assosiasjon ut fra klassen der maksimumsmultiplisiteten er 1 på motsatt side. Klassen oppfyller nå ikke lenger kravet til begrepsstereotypien, og blir derfor til en vanlig klasse.» [3]*

Med fremmednøkler menes inngrupperte attributter som stammer fra andre klasser, og dermed «peker» på disse klassene. De er både *fremmede*, fordi de ikke ligger i klassen de

10 Kapittel 6.

kom fra, og de er *nøkler* fordi de identifiserer en forekomst av klassen de «peker» på. I grupperte klasser er alle fremmednøkler markert med **«{fk}»**.

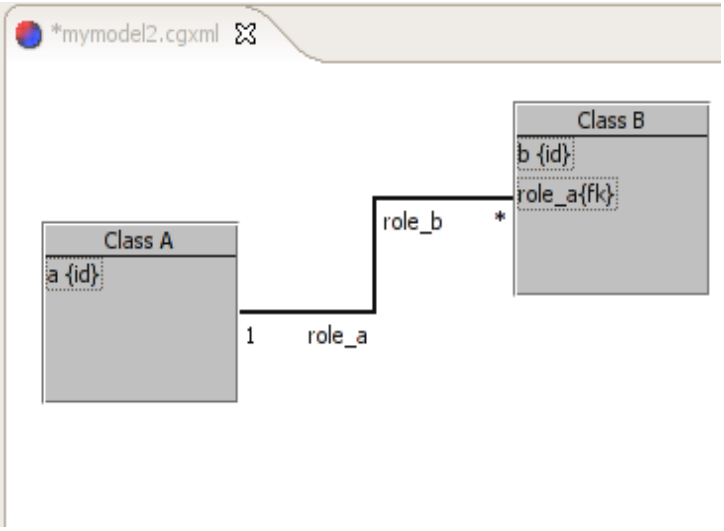
La meg å illustrere gruppering med et eksempel:



Figur 3: Eksempel på ugrupperte klasser

I figur 3 ser vi to begreper **«Class A»** og **«Class B»**. Videre kan vi observere at det eksisterer en én-til-mange assosiasjon mellom begrepene. **«Class A»** spiller rollen **«role_a»** overfor **«Class B»** i denne assosiasjonen. For **«Class B»** er multiplisiteten 1 mot motsatt side, som vil si at **«Class B»** maksimalt kan ha én forekomst av **«Class A»**. Det følger av relasjonsdatabaseteorien og BCNF at **«Class A»** sin representasjon må legges inn i en tabell som inneholder forekomster av **«Class B»**. Det kan godt være flere forekomster av **«Class A»** i denne tenkte tabellen, men da bare assosiert med forskjellige forekomster av **«Class B»**. Vi grupperer **«Class A»** sin representasjon, **«a{id}»** inn i **«Class B»,** som en fremmednøkkel. Når vi grupperer tar vi for oss ett begrep av gangen[11].

11 I prototypen markerer man de begrepene man ønsker gruppert eller degruppert i diagrammet. deretter utføres algoritmen på hver av de valgte klassene.



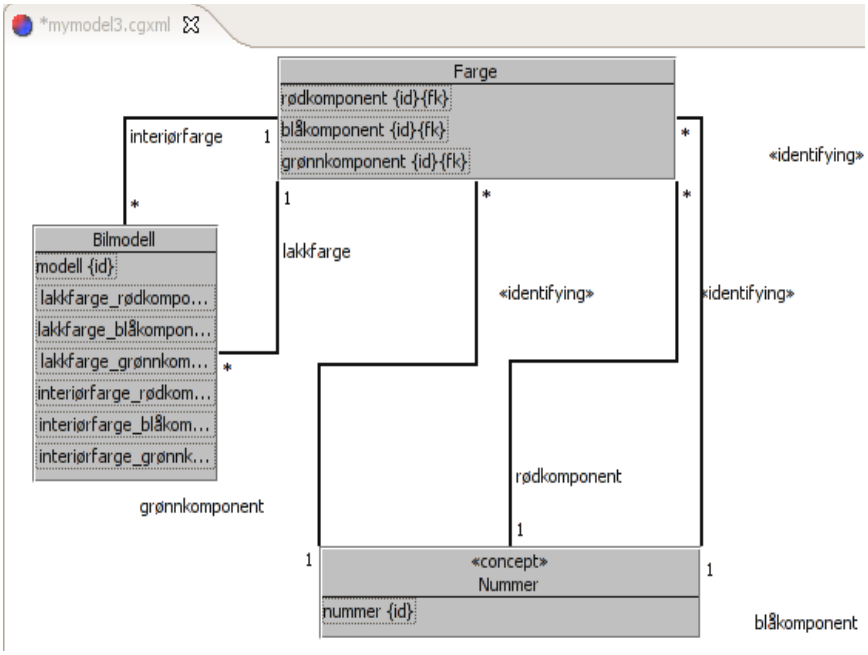
Figur 4: Eksempel på grupperte klasser

I figur 4 vises resultatet. Klassene er ikke lenger stereotypet som begreper, og vi ser at **«role_a»** har blitt navnet på *fremmednøkkelen* i **«Class B»** som representerer en forekomst av **«Class A»**. Alle attributter som et begrep «tilegner» seg ved gruppering blir i utgangspunktet fremmednøkler til andre begreper som det er assosiert med.

2.2.3 Navngivning

Attributter i en gruppert klasse trenger fornuftige navn. Navnet på et attributt bør tydelig fortelle hvilken egenskap det er snakk om. Vi bruker *rollenavnet* motparten i en assosiasjon spiller overfor klassen som skal grupperes, for å gi navn til inngrupperte attributter. Dersom ikke det er lagt på noen rollenavn bruker vi som en konvensjon klassenavnet med liten forbokstav. Det er likevel lurt å alltid sette på rollenavn, da dette tydeliggjør meningen med assosiasjonen[3]. Navngivning av fremmednøkler kan observeres i figur 4 og 6. I figur 4 har **«Class B»** fått inngruppert **«Class A»** sin representasjon **«a»** med navnet **«role_a»**. I figur 6 har **«Class B»** fått inngruppert **«Class A»** sin representasjon med navnet **«class a»**, fordi rollen ikke står eksplisitt spesifisert. Intuitivt sagt så har en forekomst av **«Class B»** en forekomst av **«Class A»** representert ved et attributt som naturlig nok er kalt **«class a».**

Dersom to klasser har flere assosiasjoner mellom seg, og dermed spiller flere forskjellige roller overfor hverandre, så kan det oppstå navnekonflikter på attributtnavn. Når grupperingen skal lages som en algoritme kan det være greit å ha en fast måte å skille navn på attributter som stammer fra samme begrep, men som kommer inn p.g.a. flere forskjellige assosiasjoner.



Figur 5: Eksempel på navngivning av inngrupperte attributter.

I prototypen løser jeg dette ved å bruke navnet attributtet har i sin opphavsklasse som suffiks på rolle- eller klassenavnet[12]. Figur 5 viser et eksempel på navngivning, der opphavsklassen[13] **«Farge»** representeres av tre attributter: **«rødkomponent», «blåkomponent» og «grønnkomponent»**. En **«Bilmodell»** har både en farge på interiøret og en farge på lakken. Dette vises ved at klassen har to assosiasjoner til **«Farge»**. Siden **«Farge»** representeres med flere attributter har disse attributtnavnene blitt suffikser i attributtnavnene i klassen **«Bilmodell»**. Da kan vi skille dem fra hverandre.

2.2.4 Andre multiplisitetskombinasjoner

Tilfellet i eksemplet i forrige avsnitt viser bare hva vi gjør ved assosiasjoner der multiplisiteten er 1..1 på den ene siden og 0..* på den andre. Jeg vil her se litt nærmere på noen andre vanlige kombinasjoner.

12 Klassenavnet med liten bokstav er det implisitte rollenavnet, om ikke rollen er eksplisitt spesifisert i en assosiasjon.

13 Klassen som er opphavet til farge-attributtene i klassen **«Bilmodell»**.

0..1 ↔ 1..1

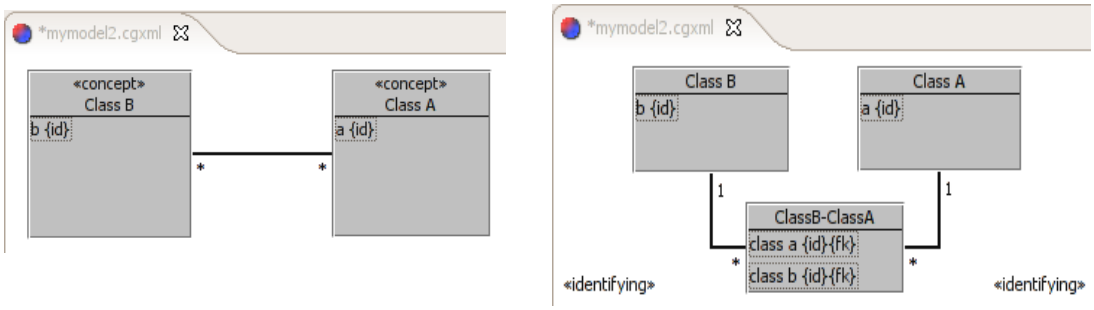
Figur 6: Eksempel på gruppering av en 0..1 ↔ 1..1 assosiasjon

I figur 6 ser vi at **«Class A»** enten har null eller én forekomst av **«Class B»**. Derimot *må* **« Class B»** ha en forekomst av **«Class A»**. Vi velger da å gruppere inn **«Class A»** sin representasjon i **«Class B»**. Høyre side av figur 6 viser de grupperte klassene.

1..1 ↔ 1..1

Dette tilfellet er litt verre å gjøre automatisk. Hvordan skal vi avgjøre hvilket begreps representasjon som skal grupperes inn i det andre ? Assosiasjonen sier at for hver forekomst av det ene begrepet så må det også eksistere en tilhørende forekomst av det andre, og motsatt. En mulighet er å genere fremmednøkler i begge klassene. Dette kan selvsagt gjøres algoritmisk, men det vil også introdusere redundans i den grupperte modellen [3]. Noen ganger lar situasjonen seg lett løse ved å bruke skjønn, mens andre ganger er det mer komplisert. Skagestein nevner et eksempel på dette, nemlig det monogame ekteskapet mellom en kvinne og en mann: skal fremmednøkkelen for Kvinne grupperes inn i Mann, eller omvendt ? Dette kan løses ved å danne begrepet «Ekteskap», og legge inn begge fremmednøklene der [3].

2.2.5 Begrepsdannelse ved mange-til-mange assosiasjoner



Figur 7: Eksempel på mange-til-mange assosiasjon

For at vi skal kunne representere mange-til-mange assosiasjoner i en relasjonsdatabase er det nødvendig å lage en tabell som kobler sammen forekomster. Vi må derfor «opprette en ny tabell» i form av en ny klasse i modellen. Den nye klassen skal da ha en én-til-mange assosiasjon til hver av klassene som kobles. Figur 7 viser nettopp dette. Klassen som er laget, **«ClassB-ClassA»**, vil ikke ha noen egen representasjon, men vil være avhengig av representasjonen til de to involverte begrepene som var grunnlaget for begrepsdannelsen[3][14].

2.2.6 Identifiserende assosiasjoner

Ser vi tilbake på figur 5, så ser vi at klassen **«Farge»** har tre *identifiserende* assosiasjoner til klassen **«Nummer»**, der **«Nummer»** spiller roller som verdien til de forskjellige fargekomponentene. **«Farge»** har ingen egen representasjon, men er avhengig av alle fargekomponentene for å kunne la seg identifisere. De identifiserende assosiasjonene er markert med **«identifying»**. Når vi grupperer klasser som har identfiserende assosiasjoner til andre klasser må vi passe på å markere nye attributter som både fremmednøkkel «**{fk}**» og at de er en del av den grupperte klassens representasjon «**{id}**».

Figur 5 viser også begrepet **«Nummer»** sin representasjon både finnes som fremmednøkler i **«Farge»** og fremmednøkler i **«Bilmodell».** De genererte fremmednøklene har selvsagt ikke samme navn lenger, men de peker alle tilbake på en forekomst av **«Nummer»**. Sagt på en annen måte har **«Nummer»** sin representasjon forplantet seg, via **«Farge»** til **«Bilmodell»**, i form av fremmednøkler [3][15].

2.2.7 Undertrykking

Når vi modellerer på det begrepsmessige plan vil vi kunne få mange småklasser, som etter gruppering ikke inneholder noe annet enn sin egen representasjon. Ser vi igjen på figur 5, og tenker at vi grupperer klassen **«Nummer»**, så vil denne bli en vanlig klasse med kun ett attributt: **«nummer{id}»**, som er selve tallet. Skulle vi direkte oversatt diagrammet i figur 5 til et databaseskjema, virker det ganske unødvendig å inkludere **«Nummer»** som en egen tabell. Da måtte vi duplisert et hvert tall som forekommer i en eller annen fargekomponent, i denne tabellen for å håndheve referanseintegritet. Denne situasjonen er langt fra optimal for dette tilfellet, og vi bør heller sløyfe hele nummerklassen når vi lager databaseskjemaet. Dette kalles *undertrykking*.

Det er likevel tilfeller der det kan være greit å beholde slike klasser i databasen. Dersom man ønsker å ha en skranke på hva som er lovlige forekomster for en klasse, så kan disse forekomstene, og ingen andre, legges inn i tabellen. Referanseintegriteten mot denne tabellen vil da sørge for at ingen ulovlige forekomster kan opprettes andre steder[3][16]. I alle tilfeller vil det være lurt å unngå å undertrykke klasser fra *selve modellen*. Det er bedre å gjøre undertrykkingen som et steg i omformingen til et databaseskjema, f.eks ved generering av SQL kode. Da unngår vi å redusere meningsinnholdet i modellen vår.

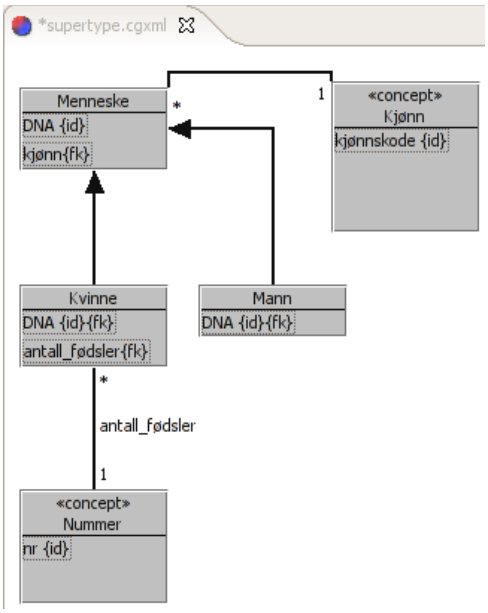
Når vi undertrykker klasser er det viktig at vi har brukt gode navn på rollene disse klassene har spilt overfor andre klasser. Dette for å lett kunne forstå hva attributter som

14 Side 129.
15 Side 136.
16 Side 142.

stammer fra undertrykte klasser egentlig inneholder [3][17], basert på navnet.

2.2.8 Underbegreper

Et underbegrep er et begrep som arver hele sin representasjon fra et annet begrep. Vi kan se på underbegrepet som en spesialisering, altså et begrep som forteller noe ekstra om superbegrepet, men ellers er helt likt. Begrepet som *arves fra* kan vi kalle superbegrepet. Snakker vi om begrepene som klasser kaller vi de gjerne subklasser og superklasser. Dette kjenner vi igjen fra klasser i objektorientert programmering. Prototype-applikasjonen støtter og lagrer informasjon om hvilke begreper som er undertyper av andre, med noen begrensninger. For eksempel er multippel arv ikke støttet; en klasse kan maksimalt ha én superklasse.



Figur 8: Eksempel på gruppering av underbegreper ved separasjon. Idéen til eksemplet er hentet fra [3].

Det finnes flere måter å gjøre en gruppering av underbegreper på. Man kan selvsagt alltid bruke intuisjon, men en datamaskin er ikke velegnet for intuisjonsbaserte beregninger. Vi trenger derfor noen veldefinerte fremgangsmåter for å kunne gjøre det systematisk [3]:

- Separasjon
- Absorpsjon
- Partisjonering

Separasjon

Når vi grupperer flere undertyper med samme supertype, så kan vi separere dem i hver sin tabell, sett fra databasens perspektiv. Undertypene vil ha nøyaktig samme representasjon

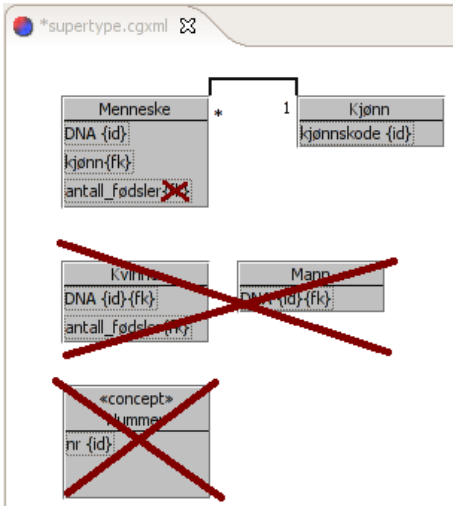
17 Kapittel 5.4.6

som supertypen det arves fra, men vi kan semantisk skille dem fra hverandre ved det faktum at de ligger i forskjellige tabeller. Dette er den eneste typen gruppering av underbegreper som støttes av prototypen[18], da den er enklest å gjennomføre. Vi trenger ikke opprette eller fjerne noen klasser, men kopierer bare hele superklassens representasjon inn som subklassens egen representasjon.

Figur 8 viser et eksempel på gruppering av to underbegreper **«Mann»** og **«Kvinne»**. Vi ser at begge arver supertypen **«Menneske»** sin representasjon **«DNA{id}»**[19] . Figure viser i tillegg at begrepet **«Menneske»** har en assosiasjon til **«Kjønn»**. I tilfellet ved seprarering er dette redundant informasjon da kjønnet også er gitt av hvilken av tabellene, **«Mann»** eller **«Kvinne»,** en forekomst av **«Menneske»** befinner seg i. Referanseintegritet fra undertypen til en forekomst i supertype-tabellen håndheves.

Absorpsjon

Med absorpsjon så kan vi tenke at supertypen «absorberer» undertypene sine attributter. En tabellen vil altså inneholde kolonner for supertypens attributter, samt kolonner for alle undertypenes attributter. Undertypene får ingen egne tabeller. For å vite noe om hvilken undertype en forekomst i supertype-tabellen faktisk er, blir det nødvendig å assosiere supertypen med et begrep som kan fortelle noe dette, som vil resultere i en diskriminerende attributt [3][20]. Det vil her kunne oppstå *nil*[21] i kolonner som stammer fra attributter fra andre undertyper enn den undertypen som en forekomst faktisk er av. Ved hjelp av det diskriminerende attributtet kan vi på forhånd vite i hvilke kolonner det skal være *nil*[3].



Figur 9: Eksempel på absorpsjon. Idéen til eksemplet er hentet fra [3].

Figur 9 viser et eksempel på absorpsjon der klassen **«Menneske»** har tatt opp

18 Se kapittel 5.5 for mer om hvilke begrensninger som gjelder for prototypen ved gruppering.
19 Bruken av «DNA» som identifikasjon er korrekt, men ikke veldig praktisk, da vi tenker at vi representerer et menneske vha. en enorm tekststreng kodet til å inneholde all DNA-informasjon. Dette har selvsagt ingen betydning for poenget med eksemplet.
20 Kapittel 5.4.4
21 Med «nil» menes en ikke-eksisterende verdi, også gjerne kalt *NULL*.

subklassenes attributter. I dette tilfellet var det bare subklassen **«Kvinne»** som hadde et ekstra attributt: **«antall_fødsler»**, som kom fra assosiasjonen med **«Nummer»**. Vi ser at det er tatt opp i **«Menneske»**. Vi diskriminerer forekomster av undertypene i supertypens tabell vha. attributtet **«kjønn»**. Dersom **«kjønn»** f.eks. er 'M', så vet vi at **«antall_fødsler»** *må* være nil i databasen.

Gruppering av underbegreper ved absorpsjon støttes ikke i prototypen.

Partisjonering

Jeg vil oppsummere partisjonering med et sitat fra Skagestein [3]:

> *«Vi oppretter – i likhet med fremgangsmåten ved separasjon – en tabell for superbegrepet og en tabell for hvert av underbegrepene. Vi oppretter fremmednøkler som før, men i tillegg føyer vi til i tabellene for underbegrepene alle fremmednøker som kan arves fra tabellen for superbegrepet.» - Skagestein [3]*[22].

Tabellene for subklassene vil altså inneholde alle attributter som tabellen for superklassen inneholder. Oppdateringer i databasen av superklassens attributter må også gjøres i alle subklassenes tabeller. Hensikten med dette er at all informasjon om en forekomst av et undergreper skal være tilgjengelig i underbegrepets tabell, og dette gjelder også nedarvede attributter [3][23].

Fordi vi dupliserer informasjon om supertypen i subtypenes tabeller, så trenger ikke nødvendigvis forekomster av subtypene eksistere i supertypens tabell. Følgende sitat fra Skagestein[3] oppsummerer problemstillingen:

> *«Det er et vurderingsspørsmål om forekomstene i undertabellen skal dupliseres i supertabellen, eller om vi skal sørge for at superbegrepet ikke inneholder noen forekomster som tilhører underbegrepene. I det første tilfellet får vi dobblertlagring av opplysninger. I det andre tilfellet kan vi risikere å måtte gå gjennom samtlige tabeller for å finne en forekomst, dersom vi ikke vet hvilket begrep forekomsten tilhører.» - Skagestein [3]*

Gruppering av sub/supertyper med partisjoneringsmetoden støttes ikke av prototypen.

2.3 Degruppering av klasser

For å systematisk degruppere en klasse trenger vi bare å fjerne alle attributter som er fremmednøkler. Dersom klasser har blitt undertrykket fra modellen vil det også kunne forekomme vanlige attributter som hverken er en del av klassens representasjon eller er fremmednøkler. Disse må også fjernes. Dersom klassen er en undertype av en annen klasse fjerner vi den identifiserende representasjonen som den har arvet. Etter dette blir klassen til et begrep igjen.

Klasser som har blitt undertrykt i grupperingsprosessen er det verre med[24]. Dersom man

22 Side 142.
23 Side 142.
24 Prototypen utfører ikke automatisk undertrykking, da dette ikke ville vært hensiktsmessig i tilfeller hvor brukeren ikke ønsket undertrykking for visse klasser.

har utført undertrykkingen *etter* konvertering til databaseskjema[25] vil man fortsatt ha de undertrykte begrepene i modellen. Men dersom disse også er slettet fra modellen vil man måtte gjenopprette disse ut fra hvilke ikke-identifiserende attributter en klasse har som ikke er fremmednøkler. Dette innebærer at man må gjette seg til hva et passende klassenavn for et undertrykt begrep som skal gjenopprettes skal være, samt egenskapene til assosiasjonen. Å forsøke å gjøre dette automatisk er lite hensiktsmessig. Undertrykker man klasser fra modellen så taper man informasjon, man mister opplysningene om hvilke begreper og representasjoner som ligger bak attributtene [3].

25 Man har altså bare droppet tabellene som naturlig er kandidater til undertrykking, fra database-skjemaet.

3 Metamodell og hoveddatastruktur

I dette kapitlet presenterer jeg metamodellen som ligger til grunn for applikasjonens hoveddatastrukturer og forretningslogikk.

3.1 Metamodell

> *«A meta-model is the collection of "concepts" (things, terms, etc.) within a certain domain and an attempt at describing the world around us for a particular purpose.» - [17] Wikipedia.org*
>
> *«A metamodel is a precise definition of the constructs and rules needed for creating semantic models.» - [18] metamodel.com*

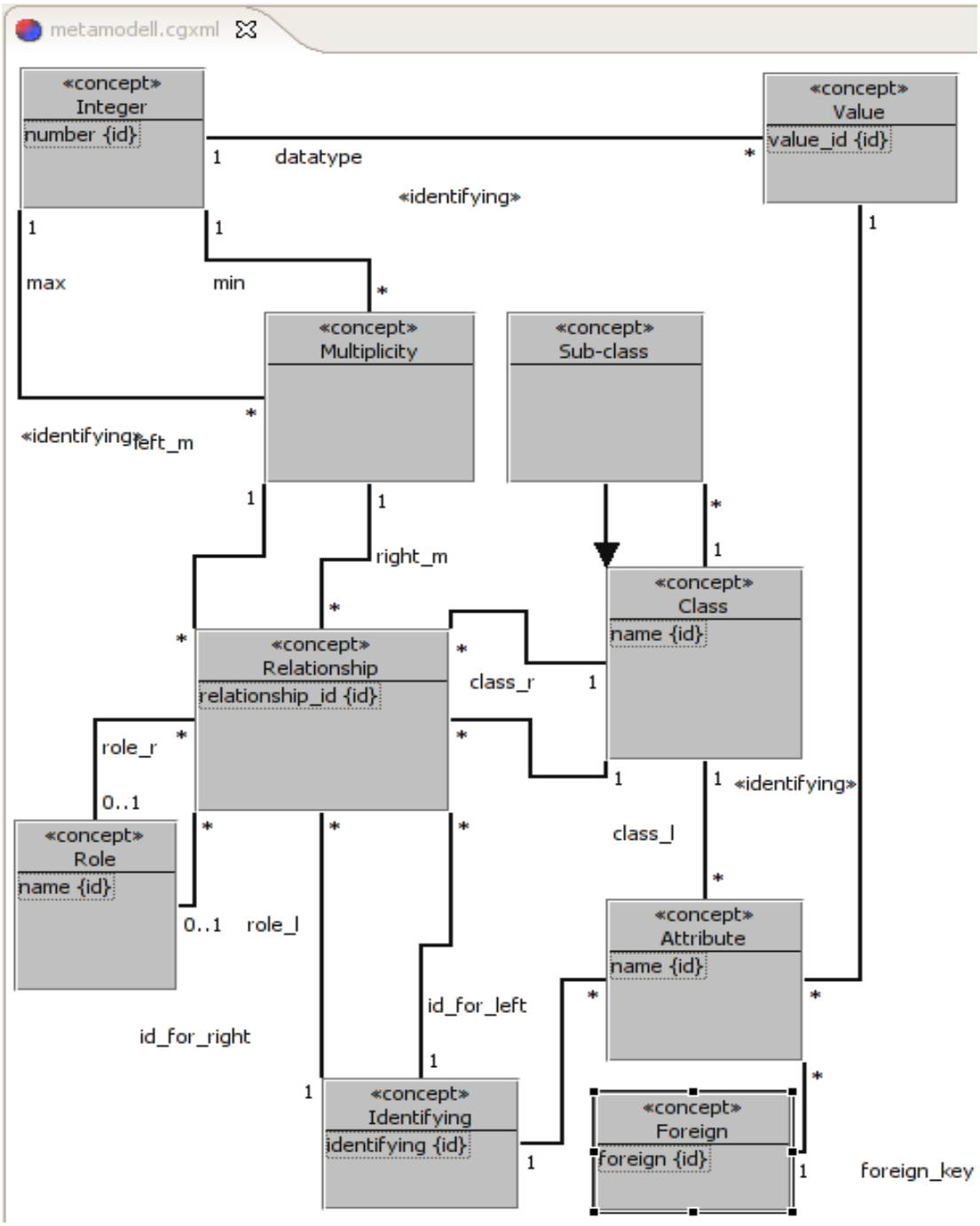
Når man lager et verktøy for modellering er det praktisk å ha en metamodell å forholde seg til. Ved hjelp av en slik modell får man et klart og tydelig avgrenset område for funksjonalitet. Metamodellens interesseområde er vanlige begreper som vi finner igjen i faktiske modellinstanser. Metamodellen forteller hvilke begreper, konstruksjoner og regler programmet må støtte ved manipulering og lagring av brukerdefinerte modeller. Sagt på en annen måte fungerer metamodellen som en spesifikasjon, og gir begrensningene for hvor rike modeller man kan lage. Eksempler på begreper i metamodellen for mitt program er «Class», «Relationship» og «Attribute».

I metamodellen er det bare inkludert de metabegrepene som er relevante for problemstillingen. Begrunnelsen for dette er at å implementere støtte for alle vanlige UML-konstruksjoner i klassediagrammer både ville tatt for lang tid, samt at det ville gått langt utover oppgavens problemområde. For å kunne utføre enkel gruppering av et klassediagram er det noen grunnleggende av metabegreper som er greie å ta med:

- Klasser (både stereotypier og grupperte klasser)
- Attributter, i rollene som identfiserende og/eller som fremmednøkler
- Datatyper (verdityper)
- Assosiasjoner mellom klassene
- Roller forbundet med assosiasjonene
- Multiplisiteter på begge sider av en assosiasjon
- Navn
- Underbegreper

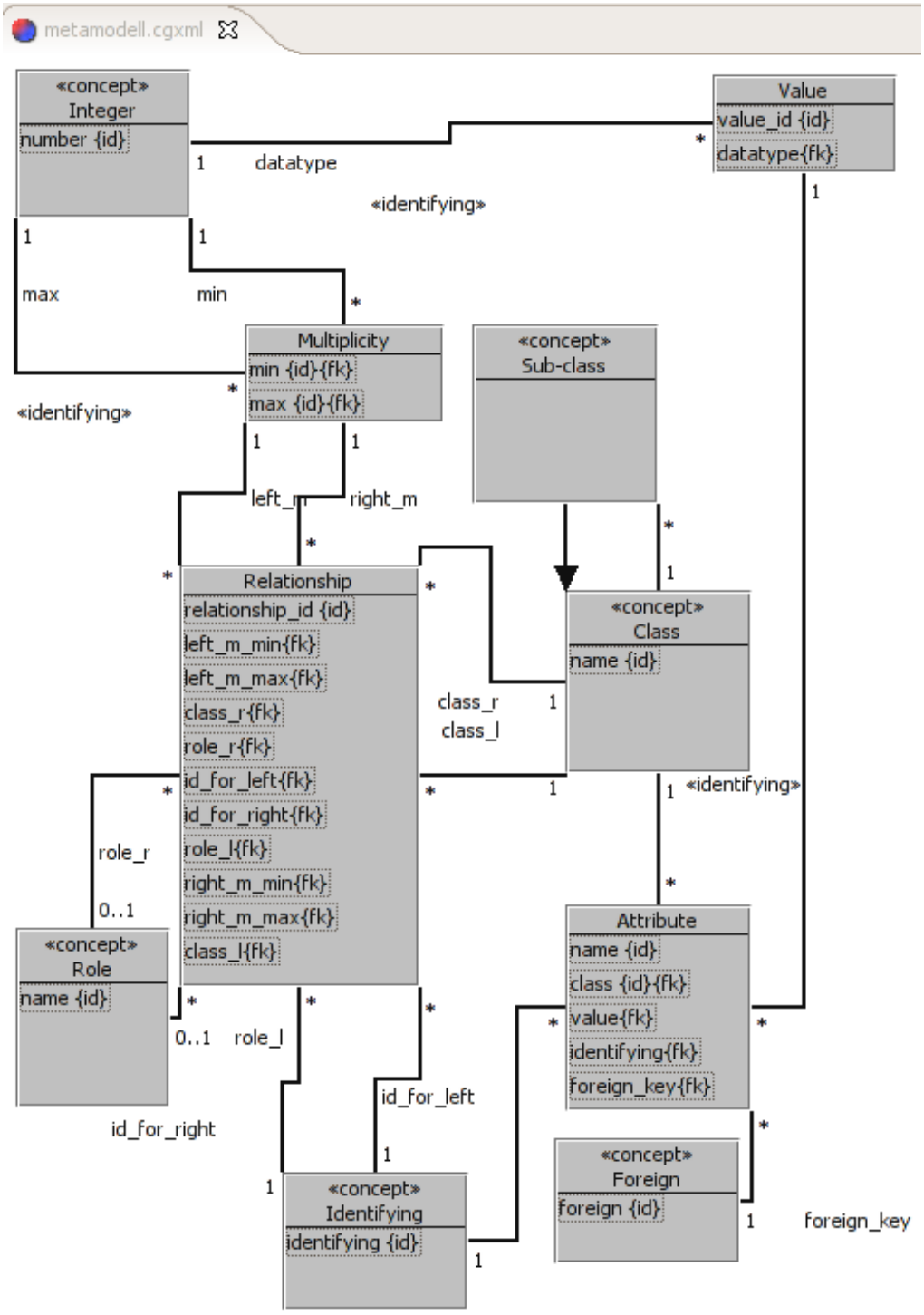
Disse begrepene kan man dele inn i to hovedkategorier: Primærobjekter og skranker på disse objektene. Primærobjektene i min metamodell er klasser. Skrankene er regler man kan knytte til klassene, for eksempel multiplisiteter på assosiasjoner mellom dem, verdidomenet for en verditype og underbegreper. Noen eksempler på vanlige begreper i UML klassediagrammer som jeg har utelatt er aggregering, komposisjon og metoder.

Hovedhensikten med applikasjonen er ikke å kunne modellere objekt-klasser som vil kunne inngå i et programsystem, men klasser som skal kunne brukes til å lage tabeller for en relasjonsdatabase. Det fokuseres derfor kun på statiske data.



Figur 10: Metamodell, ugruppert form. Figuren er laget med prototypen.

Figur 10 viser hvordan metamodellen ser ut i ugruppert form. Alle klasser er stereotypet som begreper. De inneholder kun attributter for å kunne identifisere en forekomst av seg selv, og ikke noe mer. Ytterligere egenskaper blir definert gjennom assosiasjoner og de tilhørende rollene til andre begreper.



Figur 11: Metamodell med klassene «Attribute», «Value», «Multiplicity» og «Relationship» gruppert. Figuren er laget med prototypen .

Figur 11 viser metamodellen med de grupperte klassene **«Relationship»**[26], **«Attribute»**, **«Multiplicity»** og **«Value»**. Her er ser man tydelig hvilken informasjon metamodellen inneholder for hver av de grupperte klassene. Vi ser for eksempel at en assosiasjon mellom to klasser (**«Relationship»**) blant annet har informasjon om hvilke klasser som inngår i assosiasjonen og maksimum- og minimums-multiplisiteten på høyre og venstre side. Dette kan vi se av de inngrupperte attributtene **«left_m_min»**, **«left_m_max»**, **«right_m_min»**, **«right_m_max»**, **«class_r»** og **«class_l»**. Ut fra dette kan vi også trekke konklusjonen at vi kun ønsker å støtte binære assosiasjoner. Som et annet eksempel ser vi, ut fra klassen **«Attribute»,** at applikasjonen må lagre informasjon om attributt-navn, datatype og om attributter er identifiserende og/eller fremmednøkler. Detaljene som metamodellen uttrykker finner man igjen i applikasjonens lagringsstrukturer og forretningslogikk.

Sett i forhold til en grafisk editor vil klasser i en modell typisk være forbundet med en container, nemlig et diagram. Dette anser jeg for å være såpass intuitivt at jeg har utelatt det fra metamodellen. En grafisk editor vil også trenge å bevare detaljerte data rundt visning av et klassediagram. Selv om ikke disse dataene har betydning for semantikken til en innlagt modell, så ville det være dumt om ikke editoren «husket» hvordan brukeren har plassert klasser og assosiasjoner visuelt i forhold til hverandre. Denne metamodellen omfatter ikke visualiseringsdirektiver, selv om applikasjonens datastrukturer inneholder slik informasjon.

Jeg har valgt å bruke prototypen for å lage figurer av klassediagrammer, f.eks. fig. 10 og 11. Dette prinsippet har jeg fulgt gjennom hele oppgaven.

I de påfølgende delene vil jeg beskrive hvert av de viktigste elementene i metamodellen.

26 Relationship er synonymt med assosiasjon.

3.1.1 «Class» elementet

Dette er den viktigste brikken i puslespillet. Uten klasser faller meningen med alle de andre elementene bort. Én eller flere klasser[27] må opprettes før man kan modellere noe som helst annet. En klasse i et UML diagram må ha et navn: **«name {id}»**. Dette er samtidig klassens identifiserende representasjon [28]. Det vil si at to klasser i en gitt modell-instans naturligvis ikke kan ha samme navn.

En klasse må kunne ha null, én eller flere attributter, avhengig av om den er gruppert eller ikke. Et begrep (ugruppert klasse) vil enten ikke ha noen attributter (den er da avhengig av identifiserende relasjoner), eller høyst én attributt som kan identifisere en forekomst[29]. Jeg forutsetter altså at man må modellere andre ønskede egenskaper som skal være del av et bergreps representasjon, som identifiserende assosiasjoner til andre begreper.

3.1.2 «Attribute» elementet

Til en attributt er det tilknyttet informasjon om den er identifiserende for klassen den befinner seg i **«identifying {fk}»**, om den er en fremmednøkkel, **«foreign{fk}»**, og hva slags datatype, **«value{fk}»**, (verdi-type) den har.

Datatypene må være atomiske, de kan ikke igjen bestå av andre typer eller være en sekvens av samme type (1. normalform i relasjonsdatabase-teorien). Jeg har valgt å ikke eksplisitt modellere inn de forskjellige datatypene, men heller si at datatypen er representert med et heltall. Dette heltallet ligger i **«Value»**-klassen. Det er opp til applikasjonslaget å bestemme hvilke datatyper som forbindes med de forskjellige verdiene dette tallet kan ha. Prototypen støtter kun verditypene «Text» og «Integer», da det er mindre interessant i forhold til problemstillingen å støtte et rikt spekter av verdier her.

Fra assosiasjonen mellom **«Class»** og **«Attribute»** ser vi at en klasse selvsagt kan inneholde flere attributter, mens et enkelt attributt kun kan eksistere sammen med én klasse. Datatypen til attributter som forekommer som fremmednøkler[30] i grupperte klasser vil naturligvis være den samme som attributtene de stammer fra.

3.1.3 «Relationship» elementet

For at gruppering av klasser i et diagram skal ha noen mening er man nødt til å kunne definere assosiasjoner mellom disse. En begrepsmessig klasse[31] blir gjort om til en ekte klasse ved at den får attributter fra andre klasser den har assosiasjoner til. **«Relationship»** elementet representerer assosiasjonene mellom forskjellige klasser i en modell. En

27 Når jeg snakker om *klasser* som opprettes, mener jeg begreper. Det er alltid snakk om at man gjør en begrepsmessig modellering i programmet før man eventuelt velger å gruppere noen klasser.

28 Med «identifiserende representasjon» menes den verdien (symbolet i form av en tekststreng eller et tall) som er unik for den enkelte klasse blant alle forekomster av klassen.

29 Det kan diskuteres om det er hensiktsmessig å begrense begreper til maksimalt å kunne inneholde én identifiserende attributt. Jeg har valgt å gjøre denne forenklingen både på grunn av enklere brukergrensesnitt, samt at ytterligere identifiserende attributter alltid kan modelleres som identifiserende assosiasjoner til andre begreper.

30 Markert med **«{fk}»** i grupperte klasser.

31 Begreper er markert med «concept» i klassediagrammer.

assosiasjon kan kun dannes mellom to klasser. Dette ser man tydelig ved at **«Relationship»** har to attributter forbundet med klasser: **«class_r»** og **«class_l»**. Dette er bare en forkortelse for «class_left» og «class_right». At jeg har valgt akkurat ordene «left» og «right» for å skille mellom klassene har ingen semantisk betydning. Dette gjelder også andre steder der to assosiasjoner forekommer mellom de samme klassene .

Figur 11 viser ut fra assosiasjonene mellom **«Relationship»** og **«Class»,** at en klasse kan inngå som én av partene i flere forskjellige assosiasjoner.

Vi må også lagre informasjon om multiplisiteter og roller på begge sider av en assosiasjon. Dette er representert i metamodellen som assosiasjonene fra **«Relationship»** til **«Multiplicity»** og **«Role»**. Figur 11 viser de inngrupperte attributtene som følger av disse assosiasjonene: **«right_m_min»**, **«right_m_max»**, **«left_m_min»**, **«left_m_max»**, **«role_l»** og **«role_r»**.

Til slutt må vi lagre informasjon om assosiasjonen er identifiserende for noen av klassene. Dette ser vi av de inngrupperte attributtene **«id_for_left»** (forkortelse for «identifying_for_left») og **«id_for_right»** (forkortelse for «identifying_for_right»). Verdiene til disse attributtene er gjensidig ekskluderende, og dette håndheves i applikasjonslaget.

3.1.4 «Multiplicity» elementet

Modeller må kunne inneholde data om både maksimums- og minimums-multiplisiteten på hver side av en assosiasjon. Dette er vitale data for å kunne gjøre en gruppering. **«Multiplicity»** klassen representerer multiplisiteten på én side av en assosiasjon, og inneholder attributtene **«max»** og **«min»**. **«Relationship»** klassen har to assosiasjoner med **«Multiplicity»** klassen, for venstre og høyre side.

3.1.5 «Value» elementet

Dette elementet er ment for å kunne modellere de mulig verditypene som attributter kan ha. Som tidligere nevnt støtter prototypen kun typene «Text» og «Integer», men for metamodellen er datatypen kun et heltall. Verdien av dette tallet tolkes og settes i applikasjonslaget.

3.1.6 «Sub-class» elementet

Jeg har valgt å ta med underbegreper i metamodellen. En subklasse vil arve den identifiserende representasjonen til sin superklasse («id-attributtet»). Jeg ønsket ikke å støtte multippel arv fra flere klasser, og har derfor satt en skranke, i form av en assosiasjon, på at en subklasse maksimalt kan ha én superklasse. En superklasse kan naturligvis ha flere subklasser. Metamodellen inneholder ingen ringskranker for å forhindre sykler i subklasse-treet, slik at det dannes en graf. Dette må håndteres på applikasjonsnivå.

3.1.7 «Role» elementet

«Role» elementet inneholder kun navnet, **«name»**, på rollen som spilles av én side i en assosiasjon. **«Relationship»** klassen har to slike roller, **«role_l»** og **«role_r»**, for

henholdsvis venstre og høyre side i en assosiasjon.

3.2 XML som datastruktur

> *«The **Extensible Markup Language (XML)** is a W3C-recommended general-purpose markup language for creating special-purpose markup languages, capable of describing many different kinds of data. It is a simplified subset of SGML. Its primary purpose is to facilitate the sharing of data across different systems, particularly systems connected via the Internet.»* [19] - Wikipedia.org, utdrag

Jeg har valgt å bruke XML [8] som hoveddatastruktur for å lagre modeller, samt ved utføring av transformasjoner på modellene[32]. Med «transformasjoner» menes i all hovedsak[33] gruppering og degruppering av klasser i modellen, jf. hovedproblemstillingen. XML egner seg godt som utvekslingsformat for data, og er også nødvendig for å kunne eksperimentere med XSLT [9] som transformasjonsspråk. Ved å bruke XML vil det være lettere å eksportere og importere modelldata til/fra andre XML-baserte modelleringsverktøy. Noen av disse verktøyene kommer jeg nærmere inn på i kaptittel 6.

XML-strukturen er naturligvis tett knyttet opp mot metamodellen. Den er laget for å fange opp de modellkonseptene som metamodellen inneholder. I tillegg til dette lagres visualiseringsdirektiver («view-data») i det samme XML dokumentet. Noe inspirasjon er hentet fra XML-formatet som beskrives i [7], kapittel 7. Melands XML-format beskriver ORM/NIAM [5] modeller, og er langt mer omfattende enn det som benyttes i denne oppgaven.

Da jeg utformet XML-strukturen hadde jeg noen prinsipper som jeg jobbet etter:

1. Den skulle være enkel å jobbe med som et innlest DOM [20] tre. Dette for å lette konverteringen frem og tilbake mellom applikasjonens objektmodell og XML.
2. Den skulle være semantisk sett så nær metamodellen som mulig. Her måtte det også tas praktiske hensyn, med tanke på prototypen som skal jobbe med formatet.
3. Den skulle være lett å bearbeide med XSLT.
4. Den måtte være referanse-basert, for å unngå en altfor verbal utforming med mye nøsting av elementer. Dette gjør det også lettere å mappe den til metamodellen og applikasjonens objektmodell.
5. Data for visning skulle lagres i det samme XML-formatet, men holdes strukturelt helt adskilt fra modelldataene.

32 Jeg ønsker å bemerke at prototypen av den grafiske editoren ikke bruker XML formatet internt, men en egen Java objektmodell. Det er først når en modell lagres eller transformeres at objektmodellen blir gjort om til XML. Dette kommer jeg nærmere tilbake til i kapittel 4.

33 Det gjøres også små transformasjoner på visningsdirektivene, som et ledd i eksperimenteringen med XSLT-2.0.

3.2.1 Hovedstruktur

```
<?xml version="1.0" encoding="UTF-8"?>
<classgun xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://folk.uio.no/ovyinst/classgun/xm
l/model.xsd">
  <model name="Simple model" id="diagram_2">
    <class id="class_20" ...>
       ....
    </class>
    <class ...>
    ...
    </class>
    <value id="value_20">
      ...
    </value>
    <value ...>
    ...
    </value>

    <relationship id="relationship_12">
    ...
    </relationship>
  </model>
  <views>
    <view default="yes" name="Standard view">
      <diagram title="Simple model">
        <class-view ...>...</classview>
        ...
        <relationship-view ...>...</relationship-view>
        ...
      </diagram>
    </view>
  </views>
</classgun>
```

Kildekode 1: Hovedstruktur for XML format (ikke velformet XML)

Kildekode-utdraget over viser hovedstrukturen. **`<classgun>`**[34] er rot-elementet for modell og visnings-data. Et Classgun XML-dokument kan maksimalt inneholde én modell-instans, men potensielt mange forskjellige visninger av denne. I praksis brukes bare ett «view», da applikasjonen foreløpig ikke støtter mer enn ett. Som utdraget viser er «view»-dataene strukturelt sett helt separert fra modelldataene.

3.2.2 Referanser og ID-verdier

Formatet benytter seg av identifikatorer (XML Schema «xs:ID»-typen) og referanser (XML Schema «xs:IDREF»-typen) for de sentralene elementene i en modell. Dermed kan XML skjemaet brukes til å håndheve referanse-integritet. ID-verdiene blir generert

34 «Classgun» er kodenavnet på prototypen som er benyttet under utviklingen. Det er en forkortelse for «Class-group-ungroup».

automatisk av applikasjonen når en modell lages med det grafiske grensesnittet. Først når en modell konverteres til XML første gangen blir ID'ene synlige. Objektmodellen i Java benytter seg internt av de samme ID-ene som XML-formatet.

Formatet for en ID-verdi er gitt av elementet den identifiserer. For eksempel har alle klasse-IDer følgende form: «class_X», der X er et heltall som inkrementeres. Heltallene for hver «ID-type» er uavhengige av hverandre. Sammen med type-prefikset sørger disse for unikhet gjennom hele dokumentet for alle identifiserbare elementer.

Steder i XML-strukturen der referanser til andre elementer forekommer kan lett identifiseres ved at det alltid brukes en attributt med navn «ref».

Jeg har laget et XML Schema[35] [21] som beskriver strukturen og tillatte verdier for attributter og elementer. Generelt er dette skjemaet ganske strengt med tanke på rekkefølge av elementer, antall tillatte forekomster osv. Dette er et valg jeg har gjort for å forenkle jobbingen mot XML-strukturen med XSLT og Java. Jeg vil videre se på de mest sentrale elementene i XML-strukturen sammen med utdrag fra XML-skjemaet.

3.2.3 <Model>

```
<xs:element name="model">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="class"/>
      <xs:element maxOccurs="unbounded" ref="value"/>
      <xs:element maxOccurs="unbounded" ref="relationship"/>
    </xs:sequence>
    <xs:attribute name="id" use="required" type="xs:ID"/>
    <xs:attribute name="name" use="optional"/>
  </xs:complexType>
</xs:elemen>
```

*Kildekode 2: XML Schema kode for **<model>**-elementet*

Elementet **<model>** inneholder alle data som er nødvendig å lagre i forhold til den gitte metamodellen. Det består av null eller flere forekomster av underelementene **<class>**, **<relationship>** og **<value>**. I tillegg har det en identifikator og et navn.

3.2.4 <Class>

Elementet **<class>** inneholder informasjon om én enkelt klasse. Dette inkluderer klassens representasjon og ekstra attributter, navn og intern id. I tillegg kan elementet inneholde en referanse til en super-type for klassen, via elementet <**supertype-ref**>. Vi ser av kildekodeutdrag 3 at XML-skjemaet begrenser antall supertyper til maksimalt én, for å håndheve denne begresningen fra metamodellen[36].

35 XML-skjemaet finnes som kildekode-vedlegg til oppgaven. Det er også tilgjengelig på følgende URL: http://folk.uio.no/oyvinst/classgun/xml/model.xsd

36 Multippel arv er ikke støttet.

```
  <xs:element name="class">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="supertype-ref" maxOccurs="1" minOccurs="0"/>
        <xs:element ref="representation"/>
        <xs:element ref="attributes"/>
      </xs:sequence>
      <xs:attribute name="concept" default="yes" type="yesNoType"/>
      <xs:attribute name="id" use="required" type="xs:ID"/>
    </xs:complexType>
  </xs:element>
----
  <class id="class_20" concept="yes">
    <name>Class A</name>
    <representation>
      <value-ref ref="value_20" name="a" fk="no" />
    </representation>
    <attributes />
  </class>
```

Kildekode 3: XML Schema kode og eksempelinstans for ***<class>*** *elementet*

En klasses navn, representert ved elementet **<name>,** er navnet som brukeren velger. Metamodellen uttrykker at en klasse er identifisert ved hjelp av sitt navn. I prototypen brukes imidlertid interne ID-er[37] for alle klasser, for å kunne bruke samme type ID-er på alle elementer som er nødvendige å identifisere i modellen. Dette er praktisk for å unngå at det oppstår en sammenblanding. Detaljer rundt dette kan leses i kapittel 4.5.

Attributtet **«concept»** forteller om en klasse er et begrep eller en gruppert klasse. Dette har XML-skjema-typen «yesNoType», som ikke er noe annet enn en egendefinert boolean type. Jeg henviser til kildekoden for XML-skjemaet for ytterligere detaljer.

Klassen består av to forskjellige underelementer som inneholder attributter, **<representation>** og **<attributes>**. Den semantiske forskjellen på disse er om attributter er med å å identifisere klassen eller ikke. Alle attributter som er en del av klassens totale identifikator (alle **«...{id}»** attributter) havner under elementet **<representation>**, resten havner under **<attributes>**.

3.2.5 <Relationship>

Dette elementet representerer en assosiasjon mellom to klasser. Det er en intern identifikator, samt to underelementer: **<source>** og **<target>**, se kildekode-utdrag 4. Grunnen til at jeg har valgt ordene «source» og «target» for å skille mellom de to involverte klassene, henger sammen med terminologien og tankegangen som er brukt i koden for det grafiske brukergrensesnittet. Dette kan leses mer om i kapittel 4.5.

37 Interne ID-er er usynlige for brukeren i selve applikasjonen.

```
  <xs:element name="relationship">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="source" maxOccurs="1" minOccurs="1"/>
        <xs:element ref="target" maxOccurs="1" minOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="id" use="required" type="xs:ID"/>
    </xs:complexType>
  </xs:element>
----
    <relationship id="relationship_12">
      <source identifying="no">
        <class-ref ref="class_20" />
        <role>role_a</role>
        <multiplicity min="1" max="1" />
      </source>
      <target identifying="no">
        <class-ref ref="class_21" />
        <role>role_b</role>
        <multiplicity min="0" max="*" />
      </target>
    </relationship>
```

Kildekode 4: XML schema kode og eksempel på ***<relationship>****-elementet*

Elementene **<source>** og **<target>** inneholder nøyaktig samme type informasjon, men skiller mellom de to involverte klassene. Attributtet «**identifying**» forekommer i begge elementene, og forteller om assosiasjonen er med på å identifisere den involverte klassen. **<class-ref>** referer til de klassene via «xs:ID»-attributtet «**ref**».

<role> inneholder rollenavnet, og kan godt ha tomt innhold, mens de to **<multiplicity>** elementene inneholder multiplisiteten for hver side av assosiasjonen, maksimum og minimum.

3.2.6 <Value>

```
  <xs:element name="value">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name" maxOccurs="1" minOccurs="1"/>
        <xs:element ref="datatype" maxOccurs="1" minOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="id" use="required" type="xs:ID"/>
    </xs:complexType>
  </xs:element>
----
    <value id="value_21">
      <name>b</name>
      <datatype>1</datatype>
    </value>
```

Kildekode 5: XML Schema kode og eksempel på ***<value>****-elementet*

Elementet **<value>** er ment å inneholde datatypen for forskjellige attributter. Her forekommer det også et avvik fra metamodellen ved at **<value>** inneholder et navn. Dette er navnet på opphavsattributtet. Det blir opprettet **<value>**-elementer for hvert begreps representasjons-attributt når en modell lages. Etter gruppering kan flere attributter referere til det samme **<value>**-elementet, dersom de opprinnelig stammer fra det samme begrepets representasjon.

3.2.7 <Representation> og <Attributes>

```
  <xs:element name="representation">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="value-ref" maxOccurs="unbounded"
                                    minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

Kildekode 6: XML Schema kode for elementet ***<representation>***

Eksempel på inneholdet i **<representation>**-elementet kan sees i kildekode-utdrag 6. Jeg tar ikke med schema-koden for <**attributes**>-elementet, da det strukturelle innholdet er det samme som for **<representation>**. En klasse som er et begrep vil aldri ha noen attributter under **<attributes>**, og null eller ett attributt under **<representation>**.

Både **<attributes>** og **<representation>** kan inneholde en sekvens av **<value-ref>**-elementer. **<value-ref>**-elementet peker på datatypen til attributtet, og inneholder i tillegg navnet som attributtet har i klassen. **<value-ref>**-elementet har også informasjon om det er en fremmednøkkel eller ikke.

3.2.8 <View> med underelementer

```
<view default="yes" name="Standard view">
  <diagram title="Simple model">
    <class-view ref="class_21">
      <location x="368" y="96" />
      <dimension width="100" height="85" />
    </class-view>
    <relationship-view ref="relationship_12">
      <target-anchor-gravity>2</target-anchor-gravity>
      <source-anchor-gravity>6</source-anchor-gravity>
    </relationship-view>
    <type-relationship-view>
      <supertype ref="class_24">
        <target-anchor-gravity>1</target-anchor-gravity>
      </supertype>
      <subtype ref="class_28">
        <source-anchor-gravity>4</source-anchor-gravity>
      </subtype>
    </type-relationship-view>
    ....
  </diagram>
</view>
```

*Kildekode 7: Eksempel-utdrag fra **<view>**-elementet med noen under-elementer*

<view>-elementet inneholder alle data som omhandler visning av klasser og assosiasjoner i et klassediagram. **<class-view>**, **<relationship-view>** og **<type-relationship-view>** representerer henholdsvis visning for klasser, assosiasjoner og subklasse/superklasse-koplinger.

For klasser lagres informasjon om hvor i diagrammet de befinner seg under elementet **<location>**, i form av to koordianter, **«x»** og **«y»**. I tillegg lagres klassens dimensjoner i elementet **<dimension>**. For assosiasjoner og sub/supertype forbindelser lagres informasjon om hvor forbindelseslinjene ender i hver av klassene. Jeg bruker ikke mer plass på å forklare visningsdirektivene her, men henviser til stedet til XML-skjemaet som er laget.

4 Oppbygning av prototypen

I dette kapitlet beskriver jeg oppbygningen av den grafiske editoren (prototypen) som er laget i forbindelse med oppgaven. Editoren er implementert i Java [22] som en *Eclipse* [10] plugin, og jeg kommer derfor inn på sentrale konsepter innenfor dette rammeverket. For å støtte visualisering og manipulering av grafiske klassediagrammer i Eclipse brukes rammeverket *Graphical Editing Framework (GEF)* [11]. Til slutt tar jeg for meg applikasjonsgrensesnittene for å utføre modelltransformasjor med XSLT og lagring (serialisering) og innlesning (de-serialisering) av modeller i XML formatet.

4.1 Mål med prototypen

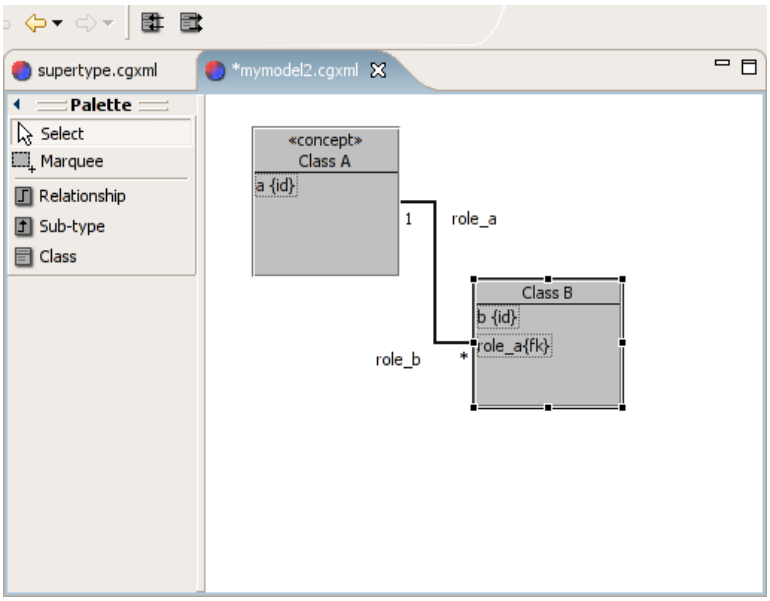
Jeg ønsket først og fremst å lage en prototype som var brukbar, og den skulle støtte modellering, visning og lagring av alle de konseptene som metamodellen omfatter. Dette med tanke på at studenter som tar systemutviklingskurs skulle kunne prøve ut konsepter innenfor begrepsmessig modellering og gruppering ved hjelp av editoren. Brukergrensesnittet skulle være intuitivt og lett å bruke, samt at det skulle være fokus på visningen av hva som skjer når man grupperer/degrupperer en klasse.

Prototypen er implementert på Eclipse-plattformen ved hjelp av GEF, og derfor har et delmål vært å utforske disse teknologiene. I den sammenheng har jeg også sett på ferdige rammeverk for utvikling av modelleringsverktøy. Dette omtales i kapittel 6.

Prototypen benytter seg av XML[8] for lagring av modeller og som datastruktur ved transformasjon av modeller. Dette valget vil gjøre det lettere å arbeide mot andre rammeverk med støtte for XML. Utvidelsesmuligheter for prototypen på dette området omtales i kapittel 7.

Valget av XML som lagringsformat har også gjort det mulig å benytte XSLT [9] som transformasjonsspråk. Prototypen bruker XSLT *versjon 2.0*. Dette har derfor gitt en mulighet for undersøke denne nye versjonen av XSLT og gjøre noen sammenlikninger med XSLT versjon 1.0 [23]. Disse tingene omtales i kapittel 5.

Kildekoden til hele prototypen finnes som et vedlegg til oppgaven.



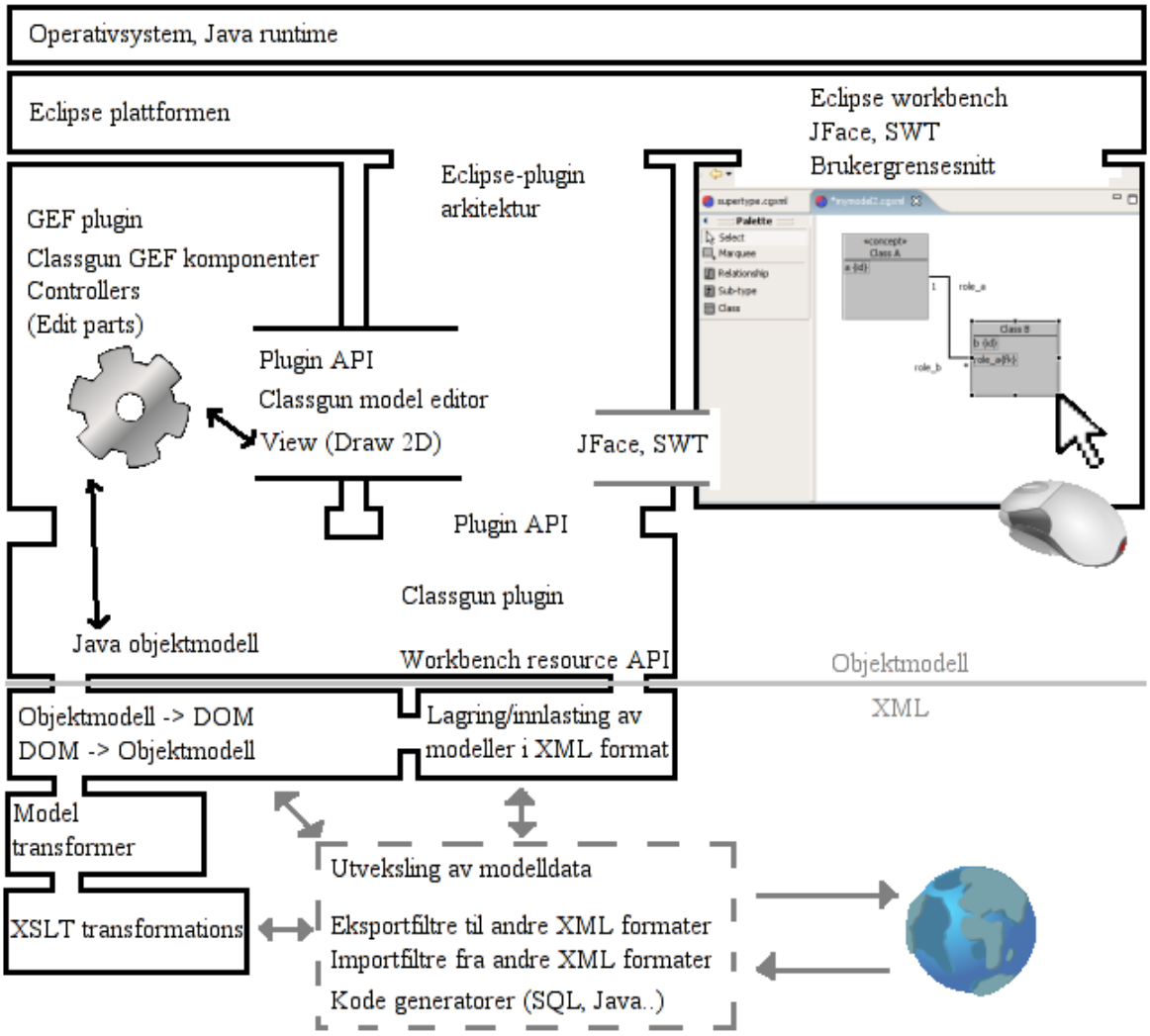
Figur 12: Skjermbilde fra Eclipse-editoren

4.2 Overordnet arkitektur

Prototypen er bygget rundt rammeverket GEF [11], som en plugin for Eclipse-plattformen. Grovt sagt er prototypen delt inn i to hoveddeler:

- Eclipse-plugin/GEF/Draw2D GUI kode
- Kode og grensesnitt for transformering og serialisering/de-serialisering av modeller på XML formatet.

Dette skillet er markert med en grå linje i figur 13.



Figur 13: Overordnet arkitektur

Figur 13 er en skisse over de viktigste komponentene i applikasjonen, og hvordan de forholder seg til hverandre. De viktigste komponentene som brukes for presentasjon og manipulering av modelldiagrammer, er i figuren market som **«Classgun model editor»**, **«Classgun plugin»** og **«Classgun GEF komponenter»**. Med «plugin» mener jeg plugin-koden som er skrevet i prototypen for at den skal kunne integreres i Eclipse. Med «editor»

menes koden for Eclipse-editoren som er laget. Med «GEF komponenter» menes de GEF komponentene som er laget for å understøtte editoren, brukergrensesnittet, bruker-input og visning.

Ut fra figur 13 ser vi de viktigste delene som er direkte relatert til Eclipse-plattformen i form av GEF komponenter og Eclipse plugin kode. Prototypen bruker en egen **Java objektmodell** som representerer brukerdefinerte modeller i minnet. Den omfatter grovt sett de samme konseptene som beskrives i metamodellen i kapittel 3, men inneholder også visningsdata. Objektmodellen er laget for å virke sammen med GEF, og blir manipulert av «GEF edit parts», som tilsvarer kontrollerne i MVC[24] arkitekturen til GEF. Disse kontrollerne er broen mellom objektmodellen og den grafiske visningen av modeller.

Den grå linjen på tvers av figuren markerer skillet mellom Eclipse/GEF/Draw2D avhengig kode, og koden som jobber med modeller på XML-formatet. **«Objektmodell -> DOM, DOM -> Objektmodell»** representerer kode som konverterer frem og tilbake mellom objektmodellen og DOM-trær [20]. XSLT-transformasjoner for gruppering og degruppering blir gjort direkte på disse DOM-trærene i Java. DOM-tre representasjonen brukes også som et mellomledd ved lagring og innlesning av XML[38].

[39]Boksen som er tegnet med en stiplet grå linje på figur 13 og inneholder **«Utveksling av modelldata»**, er tenkt fremtidig funksjonalitet. Dette omtales i kapittel 7.

All interaksjon med operativsystem og bruker går gjennom Eclipse-plattformen. Det viktigste i denne sammenheng er at lagring og innlesing av modeller i XML foregår gjennom Eclipse sitt ressurs-API, som en integrert del av editoren (**«Classgun model editor»**). Dette er markert i figuren med teksten **«Workbench resource API»**.

4.3 Design-mønstre

I dette avsnittet tar jeg for meg noen sentrale design-mønstre som er brukt i prototypen.

4.3.1 Model-View-Controller arkitekturen

MVC-arkitekturen er et prinsipp for å dele opp et programsystem i en datamodell, en logikkdel og en visningsdel. Denne separeringen sørger for lettere vedlikehold av en applikasjon fordi kodedelene for forretningslogikk, modell og visning blir relativt uavhengige av hverandre. GEF legger opp til at man bruker dette designet.

For å konkretisere og relatere dette til prototypen:

- «Model» i MVC-arkitekturen tilsvarer objektmodellen i Java. Denne modellen inneholder alle dataene som XML-formatet inneholder, men på en form som lettere kan brukes sammen med GEF.
- «Controller» i MVC-arkitekturen tilsvarer såkalte GEF «EditParts» som er laget i prototypen. Disse har ansvaret for å oppdatere modellen ved endringer i visning, og omvendt.
- «View» i MVC arkitekturen tilsvarer den grafiske Eclipse-editoren og Draw2D[40]

38 Serialisering og de-serialisering av XML gjøres alltid via DOM-tre-representasjonen.
39 Modellen er likevel helt uavhengig av GEF, rent kodemessig.
40 Draw2D er en sentral Eclipse plugin for tegning av alle slags figurer med SWT GUI biblioteket.

figurer som tilsammen danner det synlige klassediagrammet og rammene rundt. Eksempel på figurer i prototypen er «`ClassFigure`», «`RelationshipFigure`» og «`AttributesFigure`».

4.3.2 «Command»-mønsteret

«Command»-mønsteret kan oppsummeres med følgende sitat fra Grand [25]:

> *«Encapsulate commands in objects so that you can control their selection and sequencing, queue them, undo them, and otherwise manipulate them.» - M. Grand [25]*[41]

I GEF brukes kommando-mønsteret for all manipulering som skjer på diagrammer. Dette gjør bl.a. at man kan lage støtte for undo/redo. Prototypen støtter undo/redo på alle operasjoner som kan gjøres på et klassediagram, inkludert gruppering og degruppering.

For hver operasjon som brukeren utfører, f.eks. flytting eller sletting av en klasse i diagrammet, innkapsles informasjon om operasjonen i et kommando-objekt. Deretter eksekveres kommando-objektet og lagres i en stack-struktur. Hvis kommando-objektet støtter undo, kan vi be det utføre dette etter først å ha gjort en vellykket eksekvering. Dette innebærer at kommando-objektet er nødt til å lagre informasjon om tilstanden til objektene det skal manipulere, *før* selve manipulering skjer. For eksempel må en kommando som flytter en klasse i diagrammet lagre opplysningen om hvor klassen befant seg (x- og y-koordinatene) før flyttingen ble gjort.

Det er verdt å nevne at mye av applikasjonens forretningslogikk for manipulering av klassediagrammer ligger i kommando-klassene[42].

4.4 Teknologier og biblioteker

I dette avsnittet går jeg gjennom de viktigste Java-bibliotekene og rammeverkene som er benyttet i realiseringen av prototypen.

4.4.1 Eclipse

Eclipse er en åpen plattform og et utviklingsverktøy som gjør deg i stand til å lage mange typer applikasjoner[43]. Nøkkelord er utvidbarhet og plattform-uavhengighet. Noen tenker på Eclipse kun som en IDE[44] for utvikling av Java-programmer, men dette er ikke hele sannheten. Eclipse er riktignok skrevet i Java, men støtter eksempelvis utvikling av programmer i språkene C og C++ via plugins.

Jeg valgte å lage den grafiske editoren for klassediagrammer på Eclipse-plattformen. Dette fordi modellering er tett knyttet til systemutvikling og programmering. Eclipse er plattformuavhengig, og kjører like godt på Linux som på Windows. Det fantes også et bra bibliotek for å understøtte utvikling grafiske editorer: Graphical Editing Framework (GEF). For prototypens formål er dette et ypperlig rammeverk å bruke.

41 Side 277.

42 Finnes i pakken `no.uio.ifi.classgun.commands`

43 Eclipse-applikasjoner vil typisk være utviklingsverktøy og støttefunksjonalitet rundt dette.

44 Integrated Development Environment

Eclipse er basert på en såkalt «Rich Client Platform». Den følgende listen innholder hovedkomponentene og er hentet fra Wikipedia [26]:

- Core platform (boot Eclipse, run plugins)
- OSGi (a standard bundling framework)
- SWT (a portable widget toolkit)
- JFace (file buffers, text handling, text editors)
- The Eclipse Workbench (views, editors, perspectives, wizards)

Prototypen («Classgun») er implementert som en workbench komponent, nemlig en *editor*. Editorer i Eclipse lar oss åpne forskjellige typer prosjektressurser (les: filer) for redigering. Det enkleste eksemplet er en Eclipse editor som lar deg redigere tekstfiler. Prototypens *editor* åpner og lagere *datamodeller* i *XML* filer.

De fleste figurene i denne oppgaven er skjermbilder fra prototype-editoren. Figur 12 viser editoren (men ikke hele applikasjonsvinduet).

Under utviklingen har jeg brukt Eclipse versjon 3.1.0 og 3.1.1, med tilhørende standardbiblioteker.

Plugin-arkitekturen

Det er relativt lett å komme i gang med å lage plugins for Eclipse. Dokumentasjonen er god på de fleste områdene, og Eclipse kommer med ferdige maler man kan bruke for å opprette et skall til et plugin-prosjekt. I tillegg finnes det et helt sett av verktøy og hjelpemidler rettet spesielt mot utvikling av plugins *i* Eclipse *for* Eclipse [27].

```
<plugin>
   ......

   <!-- Editor -->
   <extension point="org.eclipse.ui.editors">
      <editor
            class="no.uio.ifi.classgun.editor.ModelEditor"
            default="true"
            extensions="cgxml"
            icon="icons/classgun.png"
            id="no.uio.ifi.classgun.editor.ModelEditor"
            contributorClass="no.uio.ifi.classgun.editor.ModelEditorActi
onBarContributor"
            name="Classgun Model Editor"/>
   </extension>
</plugin>
```

Kildekode 8: Eclipse plugin spesifikasjon

Kildekode-utdrag 8 viser hvor enkelt det er å spesifisere en utvidelse i Eclipse. Her ser vi hvordan modelleditoren er definert, med bl.a. informasjon om Java-klassen som

inneholder implementasjonen, hva slags filendelser editoren støtter og hvilket ikon som skal symbolisere editoren. En Eclipse plugin kan bestå av mange typer utvidelser («extension points»), f.eks. ekstra verktøylinjer, visninger og editorer.

I tillegg til koden over må man definere hovedklassen for Eclipse-plugin'en («ClassgunPlugin»[45]). Den brukes bl.a. til håndtering av livssyklusen for en plugin på Eclipse-plattformen.

Standard Widget Toolkit (SWT) og JFace

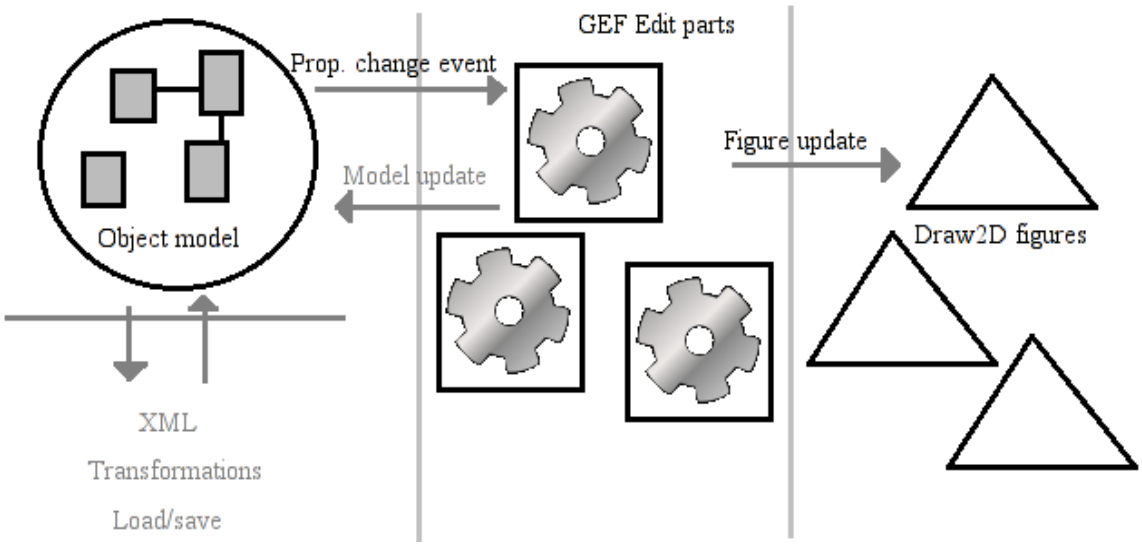
Standard Widget Toolkit (SWT) [28] er GUI-biblioteket som brukes av Eclipse, og abstraherer plattform-spesifikke toolkits, som Win32 (Windows) og GTK2 (Linux/Unix). Dersom man skal lage plugins for Eclipse er det nærmest en nødvendighet å kjenne til dette biblioteket.

JFace er et rammeverk/bibliotek i Eclipse som tilbyr GUI-komponenter på et høyere nivå enn SWT, og er selv bygget på SWT. Det inneholder en del ferdiglagde komponenter som dekker typiske behov for plugins.

4.4.2 Graphical Editing Framework (GEF)

GEF [11] er et rammeverk for å lage grafiske editorer i Eclipse. Det er bygget rundt en MVC arkitektur, der modellen er logisk separert fra visning/manipulering. Noen av de mest sentrale begrepene i GEF er:

- Edit parts
- Modellobjekter
- Figurer



Figur 14: GEF arkitekturen

Edit parts spiller rollen som kontrollerene (MVC). En edit part er en del av et grafisk

45 Klassen finnes i Java-pakken `no.uio.ifi.classgun.ClassgunPlugin`

diagram i GEF som kan manipuleres. Når endringer gjøres i diagrammet, f.eks. ved at brukeren oppretter en ny klasse, så må disse endringene utføres på objektmodellen. Det må lages et nytt modellobjekt for klassen, samtidig som det må opprettes en figur. Slike aktiviteter er koordinert av edit parts. Edit parts lytter også til endringer som skjer i modellen, og oppdaterer de tilhørende figurene.

Objektmodellen er en trestruktur hvis vi ser bort fra forbindelser mellom klassene. Roten i treet er diagrammet, og barna er klassene. Man lager en GEF edit part for hvert av elementene i objektmodellen som kan editeres på en eller annen måte. Disse edit partene vil også danne en trestruktur som likner på strukturen i objektmodellen (f.eks. korresponderer en «DiagramEditPart» til et «DiagramElement» og en «ClassEditPart» til et «ClassElement» i kildekoden til prototypen). GEF inneholder spesiell funksjonalitet for å jobbe med *forbindelser* mellom figurer, kalt «connections».

Som et utgangspunkt for utviklingen av prototypen med GEF brukte jeg et eksempel. Dette finnes på [29] http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html.

For ytterligere detaljer henviser jeg til GEF prosjektets hjemmeside [11] og kildekoden for prototypen.

4.4.3 Draw2D

Draw2D er et bibliotek i form av en Eclipse plugin. Det følger med GEF, og håndterer alt som har med layout og figurtegning å gjøre. Figurene er bygget opp som et hierarki, og SWT brukes for å tegne disse til skjerm. Jeg har brukt Draw2D i prototypen bl.a. for tegning av figurer for klasser, assosiasjoner osv. Mer detaljer om Draw2D finnes i dokumentasjonen som følger med GEF, samt på GEF-prosjektets hjemmeside [11].

4.4.4 JDOM

Jeg har valgt å bruke JDOM [30] API'et for behandling av XML-dokumenter. Jeg synes dette API'et er både lett å bruke, og samtidig kraftig nok til å dekke alle behov. Koden blir mindre verbal enn f.eks. ved bruk av W3C-DOM [31] API'et.

4.4.5 Saxon XSLT processor

For å kunne kjøre transformasjoner på XML modeller med XSLT-*2.0* brukes Saxon[32]. Saxon er en XSLT-prosessor implementert i Java, og gratisversjonen støtter XSLT 2.0, men uten mulighet for Schema[21]-betinget prosessering. Dette har liten betydninge for hovedproblemstilligen med gruppering og degruppering. Jeg brukte versjon Saxon-B 8.6.1 under utviklingen, og denne er inkludert i plugin-arkivet til prototypen.

4.5 Objektmodellen

Objektmodellen er såpass sentral i prototypen at jeg har valgt å beskrive noen deler av den her. Med «objektmodellen» mener jeg Java-klassene som representerer brukerdefinerte modeller i minnet. Det er objektinstanser av disse klassene som GEF oppretter og

manipulerer når brukeren bygger opp en modell. Dette kan observeres i figur 14. Containeren for modellobjekter er diagrammet, representert ved klassen «DiagramElement». Diagrammet inneholder modellens klasser, «ClassElement», og forbindelser mellom disse klassene. Det er to typer forbindelser: «RelationshipElement» og «TypeRelationshipElement» for henholdsvis assosiasjoner og koblinger mellom sub/superklasser.

Ordbruken i koden for objektmodellen henspeiler at den er ment å brukes for å representere en grafisk modell. Det skilles bl.a. mellom hva som er «source» og «target» i en connection, hvor en connection «sitter fast» på en klasse-figur, osv. Disse dataene må også kunne lagres i XML-formatet, og derfor finner man igjen den samme ordbruken der. En vesentlig forskjell mellom XML-formatet og objektmodellen er at visningsdirektiver strukturelt separeres fra modelldataene i XML-formatet.

4.5.1 Oppdatering av visning

Når vi utfører en gruppering av klasser i et diagram, så vil den underliggende objektmodellen vanligvis være forandret etter at grupperingen er utført. Et av målene med prototypen var at man skulle kunne observere endringene ved gruppering/degruppering visuelt og øyeblikkelig. GEF edit parts «lytter» etter endringer i egenskapene til modellobjektene de er koblet med. Dette er implementert ved hjelp av Java PropertyChangeSupport[33]. Når et modell-objekt endres, f.eks. av en GEF kommando, vil objektet sende ut et event om dette i form av kall på en callback-funksjon. Registrerte edit parts som lytter på det aktuelle objektet vil oppdatere tilhørende figurer, om nødvendig. Ved gruppering og degruppering vil selve diagram-objektet[46] sende ut et event om at klasser har forandret seg, og dermed vil alle de tilhørende edit part'ene oppdatere visningen av klassene.

4.5.2 Oppdatering av objektmodell

Som nevnt i avsnitt 4.3.2 brukes kommando-mønstret i GEF ved manipulering av modellen som ligger til grunn for visningen. For å illustrere kort hva dette innebærer vil jeg bruke et eksempel der vi legger inn en ny klasse i et diagram:

- Kommandoen «CreateClassCommand» opprettes, og får en peker til diagramobjektet i objektmodellen.
- Kommandoen eksekveres og lagres. Den oppretter da et nytt modellobjekt for klassen, og legger dette objektet til diagram-objektet.
- Diagramobjektet sender ut et event om at klassekonfigurasjonen har forandret seg.
- Dette plukkes opp av GEF edit part'en som lytter til diagram-objektet, og denne vil da be sine barn[47] om å oppdatere seg. Siden objektmodellen har fått et nytt klasse-objekt, vil det bli opprettet en ny edit part instans for å håndtere det.
- Figurene oppfriskes.
- Kommando-objektet har lagret en peker til det nye klasseobjektet som ble

46 Det finnes kun ett diagramobjekt per åpne editor-instans.
47 Barna er i dette tilfellet alle edit parts som håndterer klasser (`ClassEditPart`).

opprettet. Ved undo vil kommando-objektet kunne fjerne dette fra diagrammet igjen.

4.5.3 Generering av ID for modell elementer

Viktige elementer fra metamodellen (omtalt i kapittel 3.1) har en egen intern ID, både i XML-formatet og objektmodellen. ID'ene genereres på grunnlag av et prefiks for typen, samt et heltall som inkrementeres. Når en modell lastes inn av brukeren, oppdateres ID generatoren[48] med de ID-verdiene som allerede er brukt opp. Slik unngår man konflikter når påfølgende nye ID-er skal genereres.

4.5.4 Konvertering til og fra XML-DOM

Konvertering til og fra XML-formatet gjøres ved hjelp av JDOM og skjer hver gang en modell lagres, lastes inn eller transformeres med XSLT. Tar man transformering eller lagring som et eksempel, så vil først objektmodellen konverteres til et JDOM DOM-tre. Deretter utføres enten XSLT-transformasjoner direkte på dette DOM-treet, ellers så blir det serialisert og lagret. Koden som er laget for å utføre konverteringene befinner seg i Java-pakken `no.uio.ifi.classgun.model.dom`.

4.6 Arkitektur for lagring i XML og eksekvering av transformasjoner

4.6.1 Lagring

Det er laget et enkelt og generisk grensesnitt for serialisering og de-serialisering av objektmodellen. Prototypen inneholder to implementasjoner av dette grensesnittet: En for XML-serialisering og en for binær serialisering[49].

4.6.2 Transformering

Ved utføring av XSLT-transformasjoner på modeller, gjøres dette direkte på DOM-treet som lages av objektmodellen. Det er unødvendig å sende inn XML-dataene som en serialisert strøm når man både har datastrukturen i minnet fra før, og transformatoren er tilgjengelig i samme applikasjon.

Et viktig poeng å nevne når jeg snakker om «modelltransformasjoner», er at jeg mener transformasjoner som resulterer i en *forandret modell av samme type*. Det vil si at et XSLT- stilark som utfører en operasjon på en modell, ikke kan forandre *typen* eller *strukturen* til XML-dokumentet, bare modellens innhold. Dette passer godt med hovedformålet, som er gruppering og degruppering. En konsekvens av dette er at grensesnittet[50] som er laget for kjøring av transformasjoner, ikke er egnet til eksportering eller importering av modeller i andre formater. For dette formålet bør det lages nye grensesnitt, se kapittel 7.

48 Finnes i kildekoden som klassen `no.uio.ifi.classgun.model.util.ModelElementIdUtil`

49 Java objekt-serialisering/de-serialisering

50 Finnes i klassen `no.uio.ifi.classgun.model.transform.ModelTransformer`

Når man skal gruppere gitte klasser i editoren, må man først markere disse klassene visuelt. Det vil si at vi er nødt til å overføre informasjon om hvilke klasser som er valgt, til XSLT-stilarket som utfører grupperingen. Siden vi i objektmodellen bruker nøyaktig de samme ID'ene på modellelementer som i XML-formatet, så holder det at vi sender med en liste som inneholder ID'ene til de valgte klassene. Grensesnittet for kjøring av transformasjoner spesifiserer at man kan sende med en generisk parameter av typen `Object`, sammen med selve objektmodellen[51]. Ved gruppering og degruppering brukes denne parameteren, og er da av typen `String[]` (en liste av klasse-ID'er). For degrupperings-transformasjonen gjelder det samme, kun de valgte klassene berøres ved eksekvering.

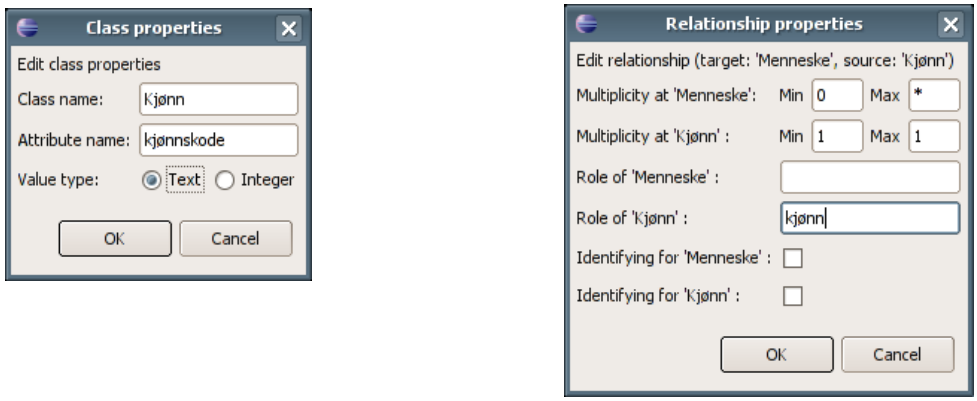
4.7 Figurer

Jeg har holdt meg på et enkelt nivå hva utseende på figurer angår. De viktigste figurene er klassefiguren og figurene for å tegne assosiasjoner og sub-type koblinger. Jeg henviser til kildekoden for ytterligere detaljer[52].

4.8 Brukerinteraksjon

Som nevnt tidligere har det alltid vært et mål å gjøre bruken av editoren så enkel som mulig. Det er temaene rundt dataorientert modellering og gruppering som er viktig, og som skal være i fokus.

De fleste operasjoner på et diagram utføres vha. musepekeren. Man kan intuitivt dra klasser rundt i et diagram, endre størrelse osv. Alle assosiasjoner og subklasse-koplinger kan dras mellom forskjellige klasser. Man vil ikke kunne fullføre operasjoner som bryter med skrankene gitt av metamodellen, f.eks. ved å forsøke å opprette en subklasse-kobling som gjør at en klasse får mer enn én superklasse. Kommando-objektene blir spurt om de kan eksekvere i en gitt situasjon, og det er på dette stadiet at slike kontroller utføres.

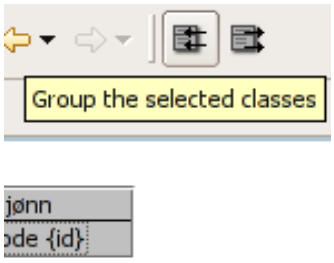


Figur 15: Dialogboks for redigering av egenskapene til en (ugruppert) klasse (t.v.), og dialogboks for redigering av egenskapene til en assosiasjon (t.h.).

Endring av egenskaper[53] til klasser (f.eks. navn) og assosiasjoner gjøres enkelt ved å

51 Objektmodellen vil naturligvis konverteres til et XML DOM-tre før transformasjonsprosessen starter.
52 Figurene finnes i pakken `no.uio.ifi.classgun.figures`
53 Man kan ikke endre allerede grupperte klasser. Da må man først degruppere klassen.

markere objektet i diagrammet, og deretter høyreklikke og velge *«Edit properties»*. Dialogboksene som dukker opp ved redigering av klasser eller assosiasjoner, vises i figur 15.



Figur 16: Knapper for å kjøre gruppering og degruppering

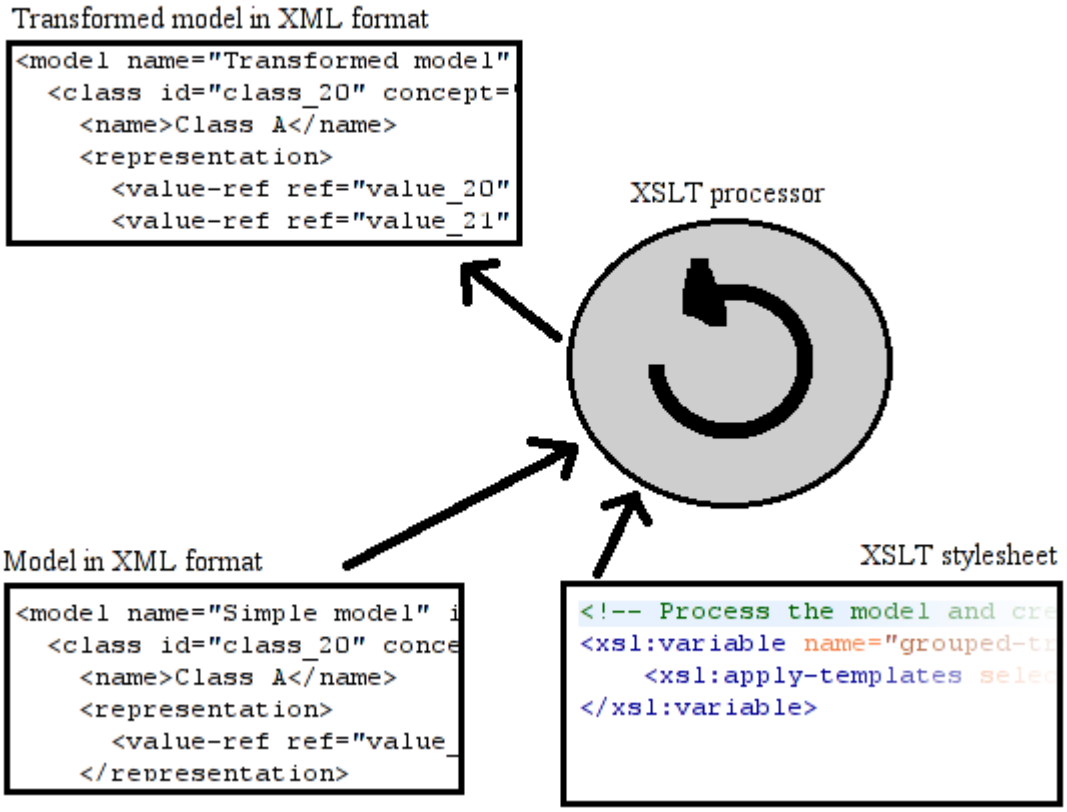
Siden undo/redo er støttet på alle operasjoner vil man lett kunne rulle tilbake om man har gjørt noen uheldige endringer i diagrammet. Dette er også er med på å lette bruken av verktøyet.

4.9 Oversikt de viktigste pakkene

- `no.uio.ifi.classgun.model.*`
  Inneholder all kode relatert til objektmodellen, XML, transformering og lagring.
- `no.uio.ifi.classgun.commands`
  Inneholder alle kommandoklassene som gjør endringer på objektmodellen.
- `no.uio.ifi.classgun.edit`
  Inneholder alle GEF edit parts.

5 Modelltransformasjoner med XSLT

I dette kapitlet ser jeg på hvordan gruppering og degruppering av klasser utføres ved hjelp av XSLT. Jeg går gjennom grupperingsprinsippene omtalt i kapittel 2. Jeg vil også se på noen fordeler som XSLT versjon 2.0 har sammenliknet med XSLT versjon 1.0.



Figur 17: Prinsipp-skisse av en modelltransformasjon med et XSLT stilark

5.1 XSLT som transformasjonsspråk

Prototypen er fra starten i utviklingsprosessen bygget opp med tanke på at XML skulle brukes som lagringsformat (se kapittel 3.2). I denne sammmenheng så jeg muligheter for å kunne eksperimentere med XSLT[9] 2.0 som transformasjonsspråk for modeller. I tillegg så jeg for meg muligheter for å bruke XSLT til å konvertere verktøyets egne XML-modellformat til andre XML-formater og ved kodegenerering (f.eks.SQL[54]). Når det gjelder visning av modeller vil også XSLT kunne brukes til å eksportere til f.eks. SVG[34][55] diagrammer. Dette omtales i kapittel 7.

Alt som er relatert til transformasjoner av modeller i verktøyet behandles på «XML nivå» med XSLT. Dette gir et skille i applikasjonen mellom denne delen og delene som handler om presentasjon og grafisk manipulering av modellene, se kapittel 4. Hvis man ønsker å utvide verktøyet med andre typer transformasjoner av modellene, så kan dette gjøres uten å måtte lage mye ny Java-kode. Samtidig kan den eksisterende XSLT-implementasjonen for gruppering og degruppering forandres og forbedres uten å måtte rekompilere applikasjonen.

5.2 Teknologier

I de påfølgende avsnittene vil jeg kort introdusere språkene XSLT og XPath.

5.2.1 XSLT

XSLT er en del av XSL [35] språkfamilien, som er et sett av W3C anbefalte standarder for å definere transformasjoner og presentasjoner av XML dokumenter [35][36]. XSLT er selve *transformasjonsspråket*.

XSLT transformasjoner[56] er basert på tankegangen om at man har et *kildedokument* og et *resultat*. Det vil si at vi ikke forandrer det originale kildedokumentet som danner grunnlaget for transformasjonen, men i stedet *produserer* vi et nytt. Dersom vi relaterer dette til oppgaven betyr det at vi sender inn en ugruppert modell som et XML DOM-tre[20], og får et *nytt* tre tilbake med den grupperte utgaven. Dette har innvirkning på måten man programmerer i XSLT.

XSLT er et deklarativt språk[37] tilpasset for å bearbeide hierarkisk strukturerte input-data[57]. Man spesifiserer hvordan man ønsker at resultatet skal bli for et gitt XML-dokument ved å lage maler med regler for hvordan de forskjellige delene (nodene) av et kildedokument skal behandles. Disse malene kalles *template rules*.

Når en transformasjon utføres vil XSLT-prosessoren forsøke å finne templates i XSLT-

54 Jeg nevner her SQL («Structured Query Language») som et eksempel på et språk med støtte for DDL («Data Definition Language») konstruksjoner

55 SVG er et standardisert XML format for å beskrive vektorgrafikk. SVG står for «Scalable Vector Graphics», og er en W3C «Recommendation».

56 Ordet «transformasjoner» er en innbakt del av akronymet «XSLT», men jeg sier likevel «XSLT transformasjoner», selv om dette blir «smør på flesk». Dette for å unngå forvirring.

57 XML-data

stilarket som kan matche elementer (noder) i kildedokumentet, basert på kriterier gitt for hver definerte template[58]. Når en template matcher en node vil reglene laget for denne eksekveres. Slik jobber man seg gjennom kildetreet av XML-elementer (noder), og produserer samtidig ny output. Nodene i input-treet trenger ikke nødvendigvis å prosesseres sekvensielt. Dette avhenger hvordan en template er laget og hvilke data fra input-treet den trenger for å produsere et resultat.

Kilden for transformasjoner vil alltid være et XML-dokument, men det samme gjelder *ikke* for resultatet. Det går for eksempel an å transformere XML-dokumenter om til ren tekst. I denne oppgaven vil imidlertid resultatet alltid være et XML dokument av samme type som kildedokumentet[59].

Reglene (templates) for hvordan et XML dokument skal transformeres ligger i *stilarkene* som vi bruker i forbindelse med en transformasjon. Stilarkene er i seg selv XML-dokumenter. De kan inneholde en blanding av XSL-direktiver, tekst og andre XML-elementer. Rekkefølgen på *templates i et stilark* er av underordnet betydning[60], da det som «driver» transformasjonen er matchingen av nodene fra input-treet.

Jeg bruker versjon 2.0 av XSLT for å gjøre transformasjoner. Denne versjonen er i skrivende stund enda ikke deklarert av W3C som en ferdig anbefaling («Recommendation»), men foreligger som en kanditat til dette («Candidate Recommendation»). Versjon 2.0 byr på mange forbedringer i forhold til 1.0. Se kapittel 5.3.

5.2.2 XPath

XPath [38] er også et språk innenfor XSL-familien. Det er integrert i XSLT og brukes primært for å *adressere* noder (f.eks. attributter og elementer) i et XML dokument. Jeg nevnte i forrige avsnitt at man i XSLT har templates som *matcher* elementer i kildedokumentet. XPath brukes blant annet for å uttrykke kriterier for hvilke nodetyper en template kan håndtere. Språket inneholder i tillegg en rekke nyttefunksjoner for strengmanipulering, aritmetikk, evaluering av logiske uttrykk m.m.

5.2.3 XSLT prosessor

Det finnes en rekke konkrete implementasjoner av XSLT-1.0. Situasjonen er ikke den samme for XSLT-2.0. Dette har naturligvis sammenheng med at spesifikasjon av XSLT-2.0 enda ikke har fått status som en endelig W3C anbefaling. Som nevnt i avsnitt 4.4.5 bruker prototypen Saxon-prosessoren[32]. Saxon støtter hele XSLT-2.0 anbefalingen, slik den er i skrivende stund.

5.3 XSLT-2.0 vs XSLT-1.0

En av grunnene til at jeg valgte å bruke XSLT-2.0, var for å kunne utforske og dra nytte av

58 Dersom ingen templates i et stilark matcher en gitt input-node, vil innebygde standard-templates benyttes.

59 Jeg vil bemerke at med et «XML dokument» mener jeg her et XML dokument som er representert som et DOM-tre.

60 Dersom flere templates matcher en gitt node, vil konflikten løses etter presedensregler spesifisert i språket.

noen av forbedringene som er gjort i forhold til versjon 1.0. Mange av disse ligger i XPath-2.0, som er en viktig del av XSLT-2.0. Fordi Saxon allerede støtter hele 2.0-spesifikasjonen, har jeg hatt full frihet til å prøve ut ny funksjonalitet i utviklingsfasen.

Her er noen av de viktigste forbedringene i XSLT-2.0 og XPath 2.0 [39]:

1. XSLT-2.0 støtter ekte temporære trær, som man kan lagre i variabler og adressere noder i. I XSLT-1.0 bruker man begrepet «result tree fragment», som er en veldig begrenset utgave av slike trær. Et temporært tre i XSLT-2.0 er alltid definert som en sekvens av noder.
2. XSLT-2.0 gir muligheten til å lage flere output-dokumenter i løpet av én enkelt transformasjon.
3. Man kan lage egendefinerte funksjoner i stilarkene. I XSLT-1.0 kan man lage noe som kalles «named templates»[61], dvs. at man gir et navn til en template som man senere kan kalle som en funksjon. Fordelen med å ha ekte funksjoner er at disse *kan kalles direkte fra XPath-uttrykk,* noe som ikke er tilfellet for «named templates».
4. XSLT-2.0 støtter prosessering med validering i henhold til et XML Schema[62].
5. Evaluering av et Xpath-2.0 uttrykk vil alltid resultere i en sekvens. Denne datatypen kan være en sekvens av noder eller en sekvens av primitive datatyper, f.eks. strenger. *I XPath-1.0 kan man ikke arbeide med (ekte) sekvenser av primitive datatyper.*
6. XPath 2.0 har over 100 standardfunksjoner, sammenliknet med 27 for XPath 1.0.
7. XPath 2.0 er sterkt typet, og man kan bruke typer definerte i XML Schema-dokumenter. Man kan teste instanser opp mot typer.
8. XPath 2.0 har fått en rikere syntaks blant annet i forbindelse med behandling av sekvenser og betingelsesuttrykk.

Den viktigste forbedringen som jeg bruker, er muligheten til å lage egne funksjoner (1). Dette utnytter jeg i stor grad i forbindelse med koden for gruppering av klasser. Ser man dette sammen med muligheten for å sende med temporære trær som funksjonsparametre (3), har man en krafig kombinasjon.

I XSLT-koden for gruppering og degruppering brukes en stilark-parameter som inneholder identifikasjonen til klasser som er valgt i diagrammet. Dette er en (ekte) sekvens av XPath-strenger (5) som mappes direkte fra Java-typen `java.lang.String[]` av XSLT-prosessoren.

Jeg har i tillegg utnyttet noe av den berikede syntaksen som XPath-2.0 tilbyr (8), sammenliknet med Xpath-1.0.

Punkt (2) i listen over er interessant i forbindelse kodegenerering basert på en modell. Ved slike transformasjoner får man gjerne bruk for å lage flere output-filer. Se kapittel 7.

---

61 En rekke implementasjoner av XSLT-1.0/XPath-1.0 tilbyr «extensions», altså XPath-funksjoner som går utover den definerte standarden. Mange av implementasjonene gir også muligheten til å programmere egne funksjoner, men dette må vanligvis gjøres i språket som XSLT-prosessoren selv er implementert i.

62 Gratisversjonen av Saxon som benyttes i prototypen, støtter ikke dette. Dersom man ønsker denne funksjonaliteten må man bruke den kommersielle utgaven.

5.4 Generelt

Når vi velger å gruppere eller degruppere klasser, vil klassenes interne ID'er sendes med som en parameter til XSLT-stilarket. Ut fra denne listen med identifikatorer vet vi fra XSLT-koden hvilke klasser vi skal utføre operasjoner på. Øvrige klasser vil ikke bli berørt. Assosiasjoner mellom klasser forandres aldri av XSLT-koden, og elementene som representerer disse dataene[63] vil derfor bli kopiert direkte til resultattreet. Det samme gjelder for verdityper.

Implementasjonen av gruppering og degruppering finnes i hvert sitt stilark:

- `group.xsl`
- `ungroup.xsl`

Disse stilarkene ligger sammen med Java-kildekoden under pakken `no.uio.ifi.classgun.model.transform`

Det finnes også et tredje stilark: `views.xsl`. Dette inneholder kun kode for å håndtere en enkel transformasjon av visningsdirektivene for klasser og omtales i kapittel 5.6.

5.5 Grupping og degruppering

5.5.1 Gruppering

Når vi skal gruppere en klasse er vi først og fremst avhengige av å se på assosiasjonene til andre klasser og multiplisiteten på disse. Som algoritmen beskrevet i kapittel 2 sier, er multiplisiteten avgjørende for om vi velger å gruppere inn representasjonen til motparten i en assosiasjon som fremmednøkler.

Initialisering

Datastrukturer i form av temporære trær og XSLT key->node[64] mappings settes opp basert på input-treet. Dette gjøres bare for å på en behagelig måte kunne hente ut relevante klasser, assosiasjoner og verdier basert på referanser til disse. Samtidig demonstrerer det bruken av temporære trær i XSLT-2.0.

Parameteren som inneholder identifikasjonen til valgte klasser i et diagram initialiseres til den tomme sekvens. Dersom ingen klasser er valgt, vil den produserte output-modellen alltid bli lik input-modellen.

Overordnet fremgangsmåte

Listen under gir en oversikt over fremgangsmåten som brukes ved gruppering:

1. For hver klasse som er definert i modellen, sjekkes det om klassen er blant de som er valgt for gruppering.
2. For hver av klassene som skal grupperes sjekkes det om klassen

63 <relationship>

64 XSLT inneholder funksjonalitet for å sette opp mapping-tabeller for XML-elementer basert på gitte verdier. Etter man har satt opp en slik tabell, der ID-strengene mappes til nodene i dette tilfellet, kan man bruke en funksjon for å hente ut noder basert på disse strengene.

kan grupperes. Klassene som ikke kan grupperes, kopieres direkte til resultattreet[65].

3. Hvis klassen kan grupperes sjekkes det først om klassen har en referanse til en superklasse. Hvis dette er tilfellet henter vi klassens representasjon fra superklassen. Dette blir da klassens eneste representasjon. Fremmednøklene blir gitt samme navn i superklassen.
4. Hvis klassen ikke har en superklasse, så kopieres klassens nåværende representasjonen først (kan være tom). Deretter letes det etter fremmednøkler fra identifiserende assosiasjoner.
5. Til slutt letes det etter fremmednøkler fra assosiasjoner som ikke er identifiserende for klassen. Disse legges inn som en del av klassens attributter, og blir også fremmednøkler. Fremmednøklene blir navngitt etter reglene fra grupperingsalgoritmen.
6. Verdityper (<value>) og assosiasjoner (<relationship>) forandres ikke av grupperingen og blir kopiert direkte til resultattreet.

Jeg vil referere til punkter i listen over som tall i parenteser, i de påfølgende avsnittene.

Finne fremmednøklene fra assosiasjoner

Når vi leter etter fremmednøkler (4,5) er vi nødt til å skille mellom de fremmednøklene som blir en del av en klassses representasjon og de som bare blir vanlige fremmednøkler[66].

For dette laget jeg en parametrisert XPath funksjon: `cg:get-attributes-from-relationships(cid,identifying)`[67].

Denne funksjonen tar to parametre: ID'en til klassen som den skal finne attributter *for* (`cid`), og en parameter som forteller om den bare skal se på assosiasjoner som er identifiserende for klassen (`identifying`). I denne funksjonen gjøres sentrale deler av algoritmen beskrevet i kapittel 2. Resultatet av funksjonen blir en sekvens av `<value-ref>` noder for attributtene som ble funnet.

I kildekode-utdrag 9 vises hoveddelen av funksjonen. Vi går gjennom hver assosiasjon som klassen har og ser på multiplisitetene i begge ender. Basert på dette avgjør vi om vi skal gruppere inn attributtene fra motparten som fremmednøkler etter de gjeldende regler. Dersom kriteriene for å gruppere inn fremmednøklene fra en assosiasjon er oppfylt, kalles funksjonen `cg:get-relationship-peer-class-attributes(cid, role)`. Denne sørger for å hente motpartens representasjon, samtidig som den sørger for korrekt navngivning i henhold til reglene fra grupperingsalgoritmen.

65 Kriteriet for å avgjøre om en klasse kan grupperes baseres på om klassen er gruppert fra før.
66 I XML-strukturen ligger identifiserende fremmednøkler under <representation>, mens vanlige fremmednøkler ligger under <attributes>, se kapittel 3.2.
67 «cg» er XML-navnerommet som inneholder alle egendefinerte funksjoner for «Classgun».

```
<xsl:for-each select="cg:get-class-relationships($cid)">
 <!-- Context '//relationship' -->
 <xsl:choose>
  <xsl:when test="source/class-ref/@ref = $cid
                  and
                  source/@identifying = $idtest-string">
   <!-- Class is "source" -->
    <xsl:if test="source/multiplicity/@max = '*'
                  and
                  (some $max in ('0','1') satisfies
                   ($max = target/multiplicity/@max))">
     <xsl:sequence select="
            cg:get-relationship-peer-class-attributes(
                        target/class-ref/@ref, target/role)"/>
    </xsl:if>
    <xsl:if test="source/multiplicity/@min = '0'
                  and source/multiplicity/@max = '1'
                  and target/multiplicity/@min = '1'
                  and target/multiplicity/@max = '1'">
     <xsl:sequence select="
                cg:get-relationship-peer-class-attributes(
                       target/class-ref/@ref, target/role)"/>
    </xsl:if>
  </xsl:when>
  <xsl:otherwise>
   <!-- Class is "target" -->
   <xsl:if test="target/class-ref/@ref =
                 $cid and target/@identifying = $idtest-string">
    <xsl:if test="target/multiplicity/@max = '*'
                  and (some $max in ('0','1') satisfies
                       ($max = source/multiplicity/@max))">
     <xsl:sequence select="
                cg:get-relationship-peer-class-attributes(
                       source/class-ref/@ref, source/role)"/>
    </xsl:if>
    <xsl:if test="target/multiplicity/@min = '0'
                  and target/multiplicity/@max = '1'
                  and source/multiplicity/@min = '1'
                  and source/multiplicity/@max = '1'">
     <xsl:sequence select="cg:get-relationship-peer-class-attributes(
                         source/class-ref/@ref, source/role)"/>
    </xsl:if>
   </xsl:if>
  </xsl:otherwise>
 </xsl:choose>
</xsl:for-each>
```

Kildekode 9: Hovedinnholdet av funksjonen `cg:get-attributes-from-relationships(cid, identifying)`. Koden er formatert for å passe til en smalere visning.

Fra kildekode-utdrag 9 kan vi se bruk av endel XSLT-2.0-spesifikke konstruksjoner. Vi har kall på egendefinerte funksjoner og bruk av ny XPath-2.0 syntaks: «`some $val in (sequence) satisfies (condition)`». XSLT-kode kan fort bli ganske verbal. Bruk av egendefinerte funksjoner har gjort koden noe mer kompakt, samtidig som den blir

lettere å lese.

Representasjon for subklasser

Subklasser arver superklassens representasjon som én eller flere fremmednøkler (3). Dersom subklasse-hierarkiet har flere nivåer, rekurserer koden opp til toppklassen og bruker denne klassens representasjon. Representasjon ved superklasse har alltid presedens over andre representasjoner. Koden støtter kun gruppering av underklasser etter *separasjonsprinsippet*, beskrevet i kapittel 2.

Navngiving av fremmednøkler

```
<xsl:for-each select="cg:get-class-representation($cid)
                      /representation/*">
 <xsl:element name="value-ref">
  <xsl:attribute name="ref" select="@ref"/>
  <xsl:attribute name="name">
   <xsl:choose>
    <xsl:when test="$role != ''">
     <xsl:value-of select="$role"/>
    </xsl:when>
    <xsl:otherwise>
     <xsl:value-of select="if ($peerClassName != '')
                           then lower-case($peerClassName)
                           else @name"/>
    </xsl:otherwise>
   </xsl:choose>
  </xsl:attribute>
  <xsl:attribute name="fk" select="'yes'"/>
 </xsl:element>
</xsl:for-each>
```

Kildekode 10: En del av koden for navngiving av fremmednøkler

Kildekode-utdrag 10 viser eksempel på navngiving av fremmednøkkel (3, 5), og er en del av funksjonen `cg:get-relationship-peer-class-attributes(cid, role)`. Dersom et rollenavn ikke er angitt for assosiasjonen brukes klassenavnet med liten forbokstav, som en konvensjon.

Begrepsdannelse ved mange-til-mange assosiasjoner

Dette håndteres ikke automatisk av den nåværende XSLT-koden for gruppering. Jeg vil derfor skissere opp følgende løsningsforslag:

1. For hver assosiasjon vi itererer over for en gitt klasse, sjekker vi om maksimumsmultiplisiteten er '*' på begge sider. Hvis dette er tilfellet må vi gjøre en begrepsdannelse, ellers fortsetter grupperingen på vanlig måte.

   Resten av punktene antar at vi håndterer en mange-til-mange assosiasjon.

2. Navnet på den nye klassen kan vi lage automatisk ved å sette sammen navnene på de to klassene i mange-til-mange assosiasjonen. Klassen må også ha en intern ID. Vi må derfor gå gjennom alle eksisterende klassers ID'er for å finne et «ledig ID-heltall».

3. Den opprinnelige mange-til-mange assosiasjonen mellom klassene blir fjernet, altså blir den ikke kopiert til resultattreet.

4. To nye en-til-mange assosiasjoner opprettes fra begge de involverte klassene til den *nye* klassen. Det vil måtte lages interne ID'er for disse assosiasjonene, på samme måte som for den nye klassen. Assosiasjonene blir begge identifiserende for den nye klassen, og klassen får ingen egen representasjon.

5. Den nye klassen grupperes ved å sette sammen representasjonene til de to opprinnelige klassene og legge inn dette som fremmednøkler i klassens representasjon. Fremmednøklene får navn etter reglene for navngiving, og baseres på rollene fra den opprinnelige mange-til-mange assosiasjonen.

6. Det vil måtte opprettes nye visningsdirektiver for de to nye assosiasjonene og den nye klassen. I tillegg må visningsdirektivene for den opprinnelige mange-til-mange-assosiasjonen slettes. Fordi transformasjonen ved gruppering foregår i to steg (der modelldata behandles først, se kapittel 5.6) kan visningdirektivene for de nye elementene genereres etter at hele modellen er ferdig prosessert.

7. Når det gjelder verdiene for de nye visningdirektivene foreslår jeg å bruke noen fornuftige standardverdier. Dette vil uansett ikke forhindre at den nye klassen kan komme til å dukke opp over andre klasser i diagrammet, eller på andre ugunstige steder. For enkelhets skyld overlates det til brukeren å plassere nye klasser på ønsket sted. Det er ikke trivielt å forsøke å finne en optimal plassering automatisk, og heller ikke hensiktsmessig.

Før dette eventuelt blir implementert i prototypen, må man gjøre begrepsdannelsen «manuelt» når man modellerer, ved å selv bryte opp mange-til-mange assosiasjoner.

Håndtering av 1..1 ↔ 1..1 assosiasjoner

Jeg har valgt å ikke gjøre dette automatisk i XSLT. Det er bedre at brukeren selv gjør en skjønnsmessig vurdering basert på meningsinnholdet i assosiasjonen og den øvrige konteksten. Dersom man hadde introdusert fremmednøkler i begge de involverte begrepene ville dette danne opphav for redundans i databasen, slik det omtales i avsnitt 2.2.4. Dersom man likevel hadde ønsket å alltid gruppere slike assosiasjoner automatisk ville antakelig det beste vært å bruke begrepsdannelse. Sett fra et XSLT-ståsted ville da fremgangsmåten fått likheter med den beskrevet i forrige avsnitt.

5.5.2 Degruppering

```
<xsl:template match="representation">
 <xsl:param name="is-subclass"/>
 <xsl:copy>
  <xsl:for-each select="*">
   <xsl:if test="@fk = 'no' and $is-subclass = false()">
    <xsl:copy-of select="."/>
   </xsl:if>
  </xsl:for-each>
 </xsl:copy>
</xsl:template>
```

Kildekode 11: XSLT kode for å fjerne fremmednøkler fra en klasse sin representasjon

Degruppering av en klasse er forholdvis enkelt. Kildekode-utdrag 11 viser en del av koden for å degruppere klasser, og vi ser her hvordan vi fjerner fremmednøkler fra en klasses representasjon. Dersom klassen er en subklasse, vil all representasjon fjernes, da den identifiseres ved hjelp av sin superklasse. Alle fremmednøkler som ikke er en del av klassens representasjon blir fjernet.

5.6 Transformasjon av visningsdirektiver (view)

Som et «proof-of-concept» utføres det en enkel transformasjon av visningsdirektivene i XML-formatet ved gruppering og degruppering. Alle klasser som er valgt i diagrammet ved kjøring av en transformasjon, vil bli affektert av dette. Det blir beregnet en ny høyde på klassefigurene basert på bl.a. antall attributter i klassene. Poenget med dette er først og fremst at dersom klasser får gruppert inn flere attributter enn figurene har plass til å vise, vil de automatisk vokse seg store nok.

I denne sammenheng vil det være naturlig å spørre seg hvorfor ikke dette gjøres i presentasjonskoden på Java-nivå. Det har ingen annen årsak enn at jeg ønsket å demonstrere noe av det man kan gjøre med XSLT-2.0, sett i forhold til XSLT-1.0.

Det er naturligvis slik at om vi skal beregne ny høyde for klasser som en del av en transformasjon, må det skje etter at klassene er ferdig grupperte eller degrupperte; det er dette som vil kunne forandre antall attributter. Vi ønsker ikke å blande XSLT-kode relatert til visning med selve (de)grupperingskoden, og har derfor separert ut dette i et eget stilark. Logisk sett vil det være nødvendig å gjøre transformasjonen i to iterasjoner:

- Gjør gruppering eller degruppering
- Juster visningdirektiver etterpå, med resultattreet fra gruppering/degruppering som input.

Dette må i XSLT-1.0 løses ved å kjede flere XSLT-stilark sammen. Men i XSLT-2.0 slipper man å gjøre det slik. I stedet gjøres gruppering/degruppering alltid først. Deretter lagres resultatet av dette som et temporært tre i en variabel. Det temporære treet sendes videre til templates i stilarket for visning. Dette stilarket produserere transformasjonens endelige output ved å kopiere ut alle modelldata, og deretter justere visningdirektivene basert på disse modelldataene. Sett utenfra er alt bare én enkelt transformasjon, men internt foregår det altså i to steg. I XML formatet ligger visningdirektivene *etter* modelldataene, og er *strukturelt separert*.

6 Alternative verktøy og rammeverk for modelltransformasjoner

Jeg har sett på noen rammeverk som kan brukes til å gjøre modelltransformasjoner i forbindelse med utviklingen av prototypen. Meningen var å undersøke om det lot seg gjøre å bruke noen av disse rammeverkene for å gjøre gruppering- og degrupperings-tranformasjoner av modeller. Oppgavens tidsrammer og problemstilling må naturligvis tas i betraktning.

6.1 OMG XMI

XML Metadata Interchange (XMI) [40] er en standard fra Object Management Group [41] for utveksling av metadata i XML. Formatet har tett tilknytning til OMG MetaObject Facility (MOF) [42], som er en standard for å definere metamodeller[68]. XMI brukes ofte som utvekslingsformat for UML modeller. Det finnes mange modelleringsverktøy som kan eksportere og importere modeller i dette formatet. I lys av dette, samtidig som oppgavens prototype også lagrer modeller i XML, er dette en interessant teknologi. Jeg kommer inn på muligheten for å implementere støtte for eksport og import av dette formatet i kapittel 7.3.

6.2 UMT-QVT

UML Model Transformation Tool (UMT) [43][1] er utviklet av Jon Oldevik ved SINTEF i Oslo. Det er et verktøy for å jobbe med kodegenering og transformasjoner basert på UML modeller representert ved XMI. I startfasen så jeg på om det var mulig å bruke dette verktøyet som en del av applikasjonen. Denne delen ville da blitt modelltransformasjonslaget, mens visning fortsatt ville vært basert på Eclipse og GEF.

UMT er et selvstendig verktøy, der fokus ligger på generering av kode ut fra modeller (SQL, Java, etc.). For å kunne utnytte UMT ville jeg sannsynligvis måttet bruke XMI for modellpersistens i min egen prototype. UMT bruker internt en egen lettere utgave av XMI formatet («XMI Light»). Det ville vært nødvendig å lage en UMT-transformasjonsimplementasjon for gruppering og degruppering av klassediagrammer som jobbet på dette formatet. I UMT er transformasjonsimplementasjoner basert på XSLT eller Java. Implementasjonen for gruppering og degruppering ville vært avhengig av UMT's egne «XMI Light»-format for modeller. Tar vi i betraktning at verktøyet ikke passer til å brukes som en komponent i andre applikasjoner, ville vært lite hensiktsmessig å forsøke å bruke det som en del av en Eclipse plugin. Den samme konklusjonen kom jeg også frem til i fellesskap med utvikleren av verktøyet, Jon Oldevik. For å holde fokus på problemstillingen, valgte jeg å ikke basere meg på UMT.

6.3 Modellrammeverk i Eclipse

6.3.1 Eclipse Modeling Framework (EMF)

EMF [44] er et rammeverk for å understøtte utvikling av applikasjoner som jobber på

68 Som et eksempel er metamodellen for UML definert ved hjelp av MOF.

datamodeller. Rammeverket kan generere kodeskjellet i Java for visning og manipulering av modeller spesifisert med XMI.

EMF består av tre fundamentale deler [44]:

- EMF: Metamodell (Ecore) for å beskrive modeller, samt runtime støtte for kommunisering av endringer på modeller[69] og persistering ved hjelp av XMI.
- EMF.Edit: Generiske klasser for å understøtte utviklingen av editorer for EMF modeller.
- EMF.Codegen: Støtte for å generere editor-kode for en EMF modell.

Det går frem av egenskapene over at det kunne vært interessant å lage en editor for gruppering og degruppering ved hjelp av EMF. Jeg valgte å bygge opp prototype-editoren fra bunnen av med min egen objektmodell, fordi jeg mener dette er en bedre måte å lære seg utvikling av GEF-baserte Eclipse plugins på. Jeg hadde i tillegg allerede laget min egen metamodell med et tilhørende XML-format. Jeg hadde ingen forkunnskaper om plugin-utvikling i Eclipse, EMF eller XMI da jeg begynte utviklingen. Derfor regnet jeg med at det var lettere å komme i mål og få en fungerende prototype ved å bruke et eget opplegg.

6.3.2 UML2

UML2 er en EMF-basert implementasjon av UML-2.0 metamodellen for Eclipse [45]. Målet til UML2-prosjektet er å kunne tilby en gjenbrukbar metamodell for utvikling av modelleringsverktøy, samt støttefunksjonalitet relatert til dette. Jeg oppdaget UML2-prosjektet sent i utviklingsfasen og har derfor ikke sett noe nærmere på hvordan jeg kunne brukt dette.

6.3.3 Atlas Transformation Language (ATL)

ATL er et språk for å uttrykke modelltransformasjoner i henhold til OMG-MDA [46] fremgsmåten [47]. Språket er bl.a. basert på et prinsipp om at transformasjoner i seg selv er modeller. Det er en blanding av deklarative og imperative språkkonstruksjoner. En ATL transformasjonsmodell er spesifisert som et sett av transformasjonsregler [47].

Det er implementert en editor i Eclipse for den tekstlige syntaksen til språket, og det er implementert en prototype av transformasjonsmotoren i Java. Jeg installerte ATL i Eclipse for å prøve det ut, og kom frem til at det sannsynligvis hadde blitt for omfattende å lære seg dette språket sammen med relaterte teknologier, hvis jeg skulle komme i mål med prototypen.

6.3.4 IBM Model Transformation Framework (MTF)

MTF er et språk for å definere mappings (transformasjoner) mellom forskjellige Eclipse EMF modeller. Det er beskrevet slik på prosjektets hjemmeside [48]:

> *«The Model Transformation Framework (MTF) is a set of tools that helps*

69 I prototypen benyttes Java PropertyChange-support. Dette gjør at eksterne entiteter (GEF) kan «lytte» til endringer i objektmodellen. Interessenter registrerer seg på modellobjekter og får beskjed om endringer via callback-metoder.

> *developers make comparisons, check consistency, and implement transformations between Eclipse Modeling Framework (EMF) models. The framework also supports persistence of a record of what was mapped to what by the transformation; this record can be used to support round-tripping, reconciliation of changes, or display of the results to a user.» [48]*

Jeg ble tipset om MTF først på et senere stadium i utviklingsfasen og hadde derfor ikke nok tid til å sette meg inn i det. Dersom jeg hadde bestemt meg for å bruke EMF, så hadde MTF-alternativet for implementasjon av gruppering og degruppering vært en mer realistisk valgmulighet, da MTF impliserer bruk av EMF.

7 Utvidelsesmuligheter og diskusjon

I løpet av utviklingen har jeg fått mange idéer til hvordan prototypen kan forbedres og utvides på forskjellige områder. I dette kapitlet vil jeg ta for meg noen av dem.

7.1 Modelleringsmessige forbedringer

7.1.1 Forbedringer av grupperingsalgoritmen

Ved gruppering av et sett med begreper der det forekommer identifiserende assosiasjoner, vil rekkefølgen på grupperingen få betydning. Ved identifiserende assosiasjoner vil grupperte klasser kunne få en sammensatt identifikator. *Hele* denne identifikastoren må naturligvis brukes dersom denne klassens representasjon dukker opp som fremmednøkler i andre klasser som grupperes. Det vil si at fremmednøkler kan forplante seg gjennom hele modellen [3]. Dette tas ikke hensyn til i grupperingskoden, og ville vært en interessant forbedring å implementere.

For å omgå dette problemet må man sørge for å gruppere klasser med identifiserende assosiasjoner først, i den rekkefølge som gjør at man får meg seg alle fremmednøkler videre. Heldigvis støtter prototypen «undo» og «redo» på alle operasjoner, slik at om det gjøres feil vil man kunne gå tilbake stegvis.

Som nevnt i kapittel 5 gjøres det ikke begrepsdannelse ved mange-til-mange assosiasjoner. Det kunne vært greit å implementere dette i henhold til løsningsforslaget som er gitt, hvertfall som en opsjon.

7.1.2 Undertrykking og view

Det hadde vært interessant å legge inn et ekstra visningsdirektiv som kunne fortelle noe om en klasses synlighet i et diagram. En opsjon ved gruppering kunne vært å automatisk *skjule* klasser som blir gjenstand for undertrykking etter grupperingen, uten at de slettes fra modellen. I transformasjonsfasen som behandler visningdirektivene kunne man satt dette «flagget» for klasser som har oppfylt gitte kriterier for undertrykking.

7.1.3 Håndtering av flere views

I XML-formatet er det lagt opp til at man skal kunne legge inn flere forskjellige views for den sammen modellen. En åpenbar utvidelsesmulighet er derfor å implementere støtte for dette i applikasjonslaget.

7.1.4 XML-formatet

Bruk av schema

Det er laget et XML Schema for å beskrive strukturen og skrankene for XML-formatet som brukes. Dessverre utnyttes ikke dette til validering ved innlasting av lagrede modeller. Dette bør gjøres før koden konverterer XML dataene (DOM-treet) om til objektmodellen, slik at man kan fange opp feil på et tidlig tidspunkt. Ellers kan man risikere å laste inn

modelldata som bryter skrankene dersom man manuelt har «flikket på» XML filene eller gjort eksterne transformasjoner. Dette er en liten utvidelse som kan gjøres med en beskjeden mengde kode, og som vil gi mange fordeler.

Skille ut views i egen dokumenttype

XSLT-2.0 har støtte for å generere flere output-dokumenter i løpet av en enkelt transformasjon. Man kan også jobbe med flere input-dokumenter, som i XSLT-1.0. Det ville vært mulig å fjerne visningsdirektivene fra XML dokumenttypen som inneholder modeller. Disse direktivene kunne vært lagt i egne XML-filer, med en egen dokumenttype.

7.1.5 Klasser med assosiasjoner til seg selv

Jeg rakk ikke å implementere støtte for å kunne lage assosiasjoner som går mellom samme klasse. Dette vil kreve noe arbeid i kode som håndterer og tegner forbindelser mellom klasser, samt forretningslogikken i sammenheng med dette.

7.2 Utvidelser til Eclipse editoren

7.2.1 Formaliserte interfacer for figurer og objektmodell

Det hadde vært en fordel å lage et mer formalisert grensesnitt for de forskjellige figurene, slik at man ikke jobbet direkte mot implementasjonen. Det samme gjelder for objektmodellen. Dette ville resultet i enda klarere skillelinjer i MVC-arkitekturen som brukes i forbindelse med GEF.

7.2.2 Skille ut visningdirektiver fra objektmodellen

Det ville vært en fordel å skille ut alle data relatert til visning i egne klasser. Da hadde man fått en enda «renere» objektmodell.

7.2.3 Eclipse-view for hiearkisk visning av modell-elementer

Som en konkret utvidelse kan man lage et Eclipse workbench-view for å vise diagramobjektene i et ekstra panel ved siden av editoren. Brukeren skal kunne velge klasser på i diagrammet på vanlig måte og ved å klikke på objekter i dette panelet.

7.2.4 Forretningsregler i editoren

Når man jobber med et klassediagram i editoren vil det typisk være at man har gruppert noen klasser, samtidig som andre klasser har forblitt begreper. Slik prototypen er i dag er grupperte klasser «låst» for redigering. Hvis man ønsker å endre representasjonsattributt eller navn må klassen først degrupperes. Men man får likevel lov til å slette assosiasjoner som går til grupperte klasser. Dette betyr at de gruppere klassene bør få fremmednøkler fra slettede assosiasjoner satt til å være vanlige attributter, fordi det ikke lenger er mulig å håndheve referanseintegritet. I editoren vil fortsatt slike attributter være markert med «{fk}».

7.2.5 Konkret

Det er laget en «wizard» for å opprette nye modell-filer i et Eclipse prosjekt. Man velger et navn på filen og i hvilken katalogen den skal ligge. Deretter startes det opp en editor-instans med denne filen som kilde. Normal lagring av filen («Save») støttes, men ikke «Save as». Hvis man vil gi filen et nytt navn, eller lagre modellen med et annet filnavn, må man altså gjøre dette manuelt ved å kopiere filen.

7.3 Datautveksling

7.3.1 Generisk interface for transformering til virkårlig output-format

Det hadde vært en fordel med formalisert støtte for å kunne eksportere modelldata til et vilkårlig annet format basert på tenkte implementasjoner av et «ModelExporter»-interface .

7.3.2 Eksportering og importering av XMI

Siden prototypen selv ikke baserer seg på XMI-formatet for persistering av modeller, ville det vært en stor fordel å kunne importere/eksportere dette formatet. Da transformasjoner er basert på XSLT vil kan kunne utføre impoteringen/eksporteringen ved hjelp av dette språket. Ved importering av et UML klassediagram vil man måtte fjerne alle elementer som ikke prototypens metamodell er i stand til å beskrive.

7.3.3 Eksportering til SVG

Som et ledd i visualiseringen av modeller kunne det vært interessant å lage en «ModelExporter» implementasjon som genererer SVG filer [34]. Dette kan implementeres med XSLT.

7.3.4 Kodegenerering

På dette punktet kan det gjøres mange interessante utvidelser. Jeg ser for meg at man lager et «CodeGenerator» interface, og implementasjoner for f.eks. SQL. Her kan man dra nytte av muligheter i XSLT-2.0 for å lage flere output-dokumenter i én transformasjon.Man bør ha anledning til å spesifisere klasser som man ikke ønsker skal være med under generering av kode. For eksempel kunne det vært en opsjon for å automatisk undertrykke klasser når SQL kode for et relasjonsdatabaseskjema genereres. Dette kan igjen implementeres med parametre til XSLT-stilark(ene) som brukes.

7.4 Brukerinteraksjon og feilmeldinger

Det bør lages mer informative feilmeldinger når brukeren manipulerer et klassediagram og gjør ting som bryter med forretningsregler. Dette er implementert på noen områder, men man vil blant annet ikke få noen feilmelding dersom man prøver å redigere egenskaper til en gruppert klasse. Det eneste som skjer i dette tilfellet er at den vanlige dialogboksen unnlater å vise seg.

Når dette er sagt, så er editoren i utgangspunktet laget slik at det ikke skal gå an å gjøre

operasjoner som bryter med forretningsreglene. Man kan forsøke seg, men musepekeren vil få et «forbudt operasjon»-symbol over seg, og dermed avverges situasjonen.

7.4.1 Brukergruppen

I problemstillingen nevner jeg at en av målgruppene for bruk av editoren er studenter som tar kurs i systemutvikling. På grunn av oppgavens omfang og korte gjennomføringsperiode ville brukertester ikke være fornuftig ressursbruk på dette stadiet av en slik verktøysimplementasjon. Dersom prototypen videreutvikles med tanke på å bli brukt i slike sammenhenger, kan det være en fordel å gjøre noen brukertester for å finne områder som trenger endringer/forbedringer.

7.5 Omfang av kildekoden

Følgende er en utskrift fra programmet **`sloccount`**, versjon 2.26:

```
Creating filelist for src
Categorizing files.
Finding a working MD5 command....
Found a working MD5 command.
Computing results.


SLOC    Directory       SLOC-by-Language (Sorted)
4501    src             java=4501


Totals grouped by language (dominant language first):
java:          4501 (100.00%)




Total Physical Source Lines of     Code (SLOC)                = 4,501
Development Effort Estimate, Person-Years (Person-Months) = 0.97 (11.65)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                          = 0.53 (6.35)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 1.83
Total Estimated Cost to Develop                            = $ 131,104
 (average salary = $56,286/year, overhead = 2.40).
SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
SLOCCount is Open Source Software/Free Software, licensed under the GNU
GPL.
SLOCCount comes with ABSOLUTELY NO WARRANTY, and you are welcome to
redistribute it under certain conditions as specified by the GNU GPL
license;
see the documentation for details.
Please credit this data as "generated using David A. Wheeler's
'SLOCCount'."
```

*Kommandoutskrift 1: Utskrift fra kjøring av programmet **`sloccount`**. Som eneste parameter til kommandoen var rotkatalogen til kildekoden.*

Programmet teller antall fysiske linjer med kildekode («***S***ource ***L***ines ***O***f ***C***ode») og ignorerer kommentarer og blanke linjer. Ut fra kjøringen ser man at det er ca. 4500 linjer med Java-kode.

```
$ wc -l *.xsl *.xsd
  296 group.xsl
   86 ungroup.xsl
   77 views.xsl
  184 model.xsd
  643 total
```

Kommandoutskrift 2: Antall linjer

Kommandoutskrift 2 viser at det er rundt 650 linjer med kode for XSLT og XML Schema.

8 Oppsummering og konklusjon

Målet med oppgaven var å lage en fungerende prototype av et verktøy for gruppering og degruppering av enkle klassediagrammer. Meningen med verktøyet var at det skulle kunne brukes av studenter for å lære om prinsippene rundt modellering med elementære utsagn og hvordan slike modeller automatisk kan om gjøres til klasser tilsvarende et databaseskjema.

Editoren gir brukeren full frihet til å manipulere klassediagrammer innenfor de gitte rammene. Fordi jeg har brukt et kraftig og godt rammeverk for å lage grafiske editorer, har jeg fått med mye nyttig funksjonalitet på kjøpet. Man får en umiddelbar visuell tilbakemelding på alle operasjoner. I tillegg kan man utføre selektiv gruppering/degruppering av klasser og angre eller omgjøre alle handlinger. Fordi prototypen er implementert på Eclipse-plattformen er den både lett å installere og kan kjøres på alle operativsystemer som Eclipse støtter.

Det er ikke utført brukertesting av prototypen, noe som vil være nødvendig hvis den eventuelt skal tas i bruk. Dette kan hjelpe til med å kartlegge områder som trenger forbedringer innenfor brukerinteraksjon. Jeg vil likevel påstå at jeg har lykkes i å lage et godt utgangspunkt for en enkel og intuitiv editor, som fokuserer på problemstillingen.

Ser vi samlet på de ovenfornevnte egenskapene mener jeg at målene for prototypen er oppfylt innenfor den gitte tidsrammen..

Gjennom oppgaven har jeg gått gjennom teorien som ligger bak grupperingsprinsippene, og rammene jeg har satt for funksjonalitet. Metamodellen har fungert som en spesifikasjon både for hva editoren skal støtte og for implementasjonen av XML-formatet og transformasjonene. Jeg har fått brukt XSLT-2.0 som transformasjonsspråk og har i denne sammenheng fått prøvd ut endel av den nye funksjonaliteten som tilbys, sammenliknet med XSLT-1.0. Jeg vil påstå at valget av XML som primært lagringsformat for modeller har gjort mulighetene for fremtidige utvidelser enklere. Ved å videreutvikle prototypens grensesnitt for modelloperasjoner vil man kunne legge til rette for kodegenerering og utveksling av modelldata i andre formater. Implementasjonen av dette vil da kunne gjøres i XSLT og vil kreve minimalt av endringer i applikasjonslaget.

Jeg har vurdert noen alternative rammeverk for å gjøre modelltransformasjoner. Det er liten tvil om at disse rammeverkene har mye å by på av interessant funksjonalitet for utvikling av modelleringsapplikasjoner. I startfasen undersøkte jeg blant annet verktøyet UMT-QVT's egnethet som modelltransformasjonskomponent i en Eclipse plugin. Både ATL og MTF var ukjent territorium, og har samtidig en læringsterskel som er høyere enn hva jeg kunne tillate meg for å komme i mål innen tidsfristen. Jeg mener å ha gjort et riktig valg ved å satse på teknologier som jeg visste jeg ville komme i mål med, i stedet for å bevege meg inn på ville veier. Jeg har implementert den mest nødvendige funksjonaliteten for modelltransformasjoner ved hjelp av XSLT, og jeg mener jeg har fått frem de viktigste poengene rundt dette i oppgaven. Jeg har også foreslått løsningsforslag på steder der mangler forekommer. Til slutt har jeg kommet med en rekke forslag til utvidelsesmuligheter.

Implementasjonen av det grafiske brukergrensesnittet har tatt en vesentlig del av den totale utviklingstiden. Men uten et skikkelig brukergrensesnitt ville aldri prototypen blitt en

realitet. Jeg så på det som en nødvendighet å komme i mål med dette, selv om det gikk ut over tiden jeg opprinnelig hadde stipulert å bruke. Jeg har også fått frem hva jeg mener har vært mest interessant rundt bruken av GEF og Eclipse.

Jeg har vist at man er i stand til å lage alle klassediagrammer som er brukt i figurer i denne oppgaven, ved hjelp av prototypen. Dette inkluderer metamodellen og klassediagrammene som illustrerer de forskjellige grupperingsprinsippene. Jeg vil derfor konkludere med at den bør være god nok som utgangspunkt for utvikling av et læringsverktøy som behandler de samme temaene.

Bibliografi

[1] Oldevik, J., *UML Model Transformation Tool - Overview and guide documentation,* http://umt-qvt.sourceforge.net/docs/UMT_documentation_v08.pdf, 2004

[2] http://www.uml.org/, *Unified Modeling Language*

[3] Skagestein, G., *Systemutvikling - fra kjernen og ut, fra skallet og inn, 2. utgave,* Høyskoleforlaget, 2005, ISBN 82-7634-671-5

[4] Halpin, T., *UML Data Models From An ORM Perspective*, Journal of Conceptual Modelling , 1998

[5] Halpin, T., *Object Role Modelling (ORM/NIAM),* 1998, ISBN 3-540-64453-9

[6] http://en.wikipedia.org/wiki/Boyce-Codd_Normal_Form#Boyce-Codd_normal_form, *Database normalization*

[7] Meland, E., *Et flerspåklig datamodelleringsverktøy med XML som modellrepresentasjon*, hovedfagsoppgave ved UiO, 2004

[8] http://www.w3.org/TR/REC-xml/, *Extensible Markup Language*

[9] http://www.w3.org/TR/2005/CR-xslt20-20051103/, *XSL Transformations Version 2.0*

[10] http://www.eclipse.org/, *Eclipse Integrated Development Environment*

[11] http://www.eclipse.org/gef, *Graphical Editing Framework*

[12] http://en.wikipedia.org/wiki/Data_modeling, *Data modeling*

[13] Skagestein, G., *Dataorientert Systemutvikling,* Universitetsforlaget, 1996 ISBN 82-00-22476-7

[14] Janning M., Nachmens S. & Berild S., *Introduktion till associative databaser och CS4-systemet*, Lund : Studentlitteratur, 1981, ISBN 91-44-17111-03-88598-019-30-86238-012-x

[15] http://www.webopedia.com/TERM/U/UML.html, *What is UML ?*

[16] Garcia-Molina H., Ullman, Jeffrey D., Widom, J., *Database Systems - The Complete* Book, Prentice Hall, 2002, ISBN 0-13-098043-9

[17] http://en.wikipedia.org/wiki/Meta-Modeling, *Meta-Modeling*

[18] http://www.metamodel.com/staticpages/index.php?page=20021010231056977, *What is metamodeling, and what is it good for ?*

[19] http://en.wikipedia.org/wiki/Xml, *XML*

[20] http://www.w3.org/DOM/, *Document Object Model (DOM)*

[21] http://www.w3.org/XML/Schema, *W3C XML Schema*

[22] http://java.sun.com/, *Java Technology*

[23] http://www.w3.org/TR/xslt, *XSL Transformations (XSLT)*

[24] http://ootips.org/mvc-pattern.html, *(ootips) Model-View-Controller*

[25] Grand M., *Patterns in Java: A catalog of reusable design patterns illustrated with UML,* John Wiley & Sons, Inc., 1998

[26] http://en.wikipedia.org/wiki/Eclipse_(software), *Eclipse (software)*

[27] http://www.eclipse.org/pde/, *Plugin Development Environment*

[28] http://www.eclipse.org/swt/, *SWT: The Standard Widget Toolkit*

[29] http://www.eclipse.org/articles/Article-GEF-diagram-editor/shape.html, *A Shape Diagram Editor*

[30] http://www.jdom.org/, *JDOM*

[31] http://java.sun.com/j2se/1.4.2/docs/api/org/w3c/dom/package-summary.html, *org.w3c.dom (Java 2 Platform SE 1.4.2)*

[32] http://saxon.sourceforge.net/, *Saxon XSLT 2.0 processor*

[33] http://java.sun.com/docs/books/tutorial/javabeans/properties/bound.html, *Bound Properties*

[34] http://www.w3.org/TR/SVG/, *Scalable Vector Graphics (SVG) 1.1 Specification*

[35] http://www.w3.org/Style/XSL/, *The Extensible Stylesheet Language Family (XSL)*

[36] http://www.w3.org/, *World Wide Web Consortium*

[37] http://en.wikipedia.org/wiki/Declarative_programming, *Declarative programming*

[38] http://www.w3.org/TR/xpath20/, *W3C XML Path Language 2.0*

[39] Fitzgerald, M., *Learning XSLT,* O'Reilly & Associates*,* 2004
ISBN 0-596-00327-7

[40] http://www.omg.org/technology/documents/formal/xmi.htm, *XML Metadata Interchange*

[41] http://omg.org/, *Object Management Group*

[42] http://www.omg.org/mof/, *OMG's MetaObject Facility*

[43] http://umt-qvt.sourceforge.net/, *UMT-QVT Homepage*

[44] http://www.eclipse.org/emf, *Eclipse Modelling Framework*

[45] http://www.eclipse.org/uml2/, *The Eclipse UML2 project*

[46] http://www.omg.org/mda, *OMG Model Driven Architecture*

[47] http://www.sciences.univ-nantes.fr/lina/atl/atlProject/presentation/, *ATL - The ATL project*

[48] http://www.alphaworks.ibm.com/tech/mtf, *Model Transformation Framework*