UNIVERSITETET I OSLO
Institutt for informatikk

Et verktøy for å editere datamodeller i flere views

Masteroppgave
60 Studiepoeng

Asbjørn Eide

1. mai 2007

Forord

Jeg vil takke min veileder Gerhard Skagestein for hans innsikt og hjelp på veien i løpet av masterstudiet. Hans entusiasme for arbeidet og utviklingen av verktøyet har vært en stor inspirasjon. Jeg vil også takke Øyvind Stegard som skrev prototypen som jeg har videreutviklet. Han har vært til god hjelp med å få meg i gang med programmeringen. Hans struktur og kommentering i kildekoden gjorde det raskere å sette seg inn i rammeverket.

Innholdsfortegnelse

1. Innledning

1.1. Problemstillingen

Masteroppgaven dreier seg om utforming og implementasjon av et UML utviklingsverktøy som utnytter et Model/View konsept hvor viewet kan se helt annerledes ut enn modellen.
Som eksempel skal det utvikles et verktøy som har funksjonalitet til å gruppere UML klassediagrammer både delvis og helt. I tillegg har jeg noen delproblemstillinger om valg av grupperingsalgoritme for grupperingsfunksjonalitet i verktøyet, hvordan behandle assosiasjonsklasser og bruk av filformater for å sikre kompatibilitet mot andre verktøy.

1.2. Målgruppe

Hovedmålgruppen for verktøyet som utvikles er studenter som skal lære seg systemutvikling og modellering, og overgangen fra datamodell til relasjonsdatabase[1]. Verktøyet er spesiellt rettet mot studenter som tar kurset INF1050 – Systemutvikling ved Institutt for Informasjonrmatikk på Blindern, da verktøyet er tenkt brukt som en del av læreplanen. Også for kurset INF1300 der man arbeider med den NIAM/ORM-inspirerte profilen av UML-klassediagrammer [Skagestein 2005] kan verktøyet være interessant.

En annen målgruppe for denne masteroppgaven er personer som har generell interesse for systemutvikling og modellering, og personer som har interesse for design av brukergrensesnitt.

En tredje målgruppe er utviklere og brukere av IFI UML Total. IFI UML Total er en pakke med Open Source UML verktøy. Mange av verktøyene er resultater av studenters masteroppgaver og skal dekke behov ved UML modellering.
Det tas opp en del problemstillinger i denne oppgaven som er veldig nærliggende deres utfordringer.

1.3. Rammer

Verktøyet som utvikles baserer seg på en prototype utviklet av Øyvind Stegard som en del av hans masteroppgave i 2005 [Stegard 2005]. Det er skrevet i Java med Eclipse og GEF[1] og benytter seg av XML og XSLT for fillagring og gruppering. Verktøyet arbeider på dataorienterte UML-klassediagrammer inspirert av NIAM/ORM.

1.4. Forutsetninger

Forkunnskaper som er fordelaktig når man leser denne oppgaven, er generell kunnskap om:

- systemutvikling
- modellering, spesifikt UML. Teori om gruppering finnes i kapittel 2.4 og 2.5
- programmering ved de mer tekniske aspektene av løsningen, se kapittel 3 og 5

1.5. Språk og terminologi

Jeg benytter norsk som hovedspråk i denne oppgaven. Det er også innslag av engelske uttrykk der jeg føler det beskriver ting best og der det ikke finnes gode norske oversettelser.

I noen sammenhenger bruker jeg flere uttrykk for å beskrive det samme. Dette gjelder selve UML-editoren jeg utvikler som refereres som ”verktøyet”, ”applikasjonen”.
Der jeg snakker om ”dagens verktøy” snakker jeg om det verktøyet som er bygd på protoypen.
Når jeg benytter meg av begrepet ”prototypen” refererer jeg her til Øyvind Stegard sin prototype.

Jeg skriver ”*vi* bruker..” og ”*vi* må..” flere steder i oppgaven. Dette har blitt brukt intuitivt og gjenspeiler en del av de diskusjonene jeg har hatt med min veileder der vi har drøftet ulike problemstillinger.

[1] De forskjellige teknologiene utdypes senere i oppgaven

I beskrivelsene av skillet mellom modelldata og viewdata referer jeg til objekter og figurer. Objekter tilhører modellen og figurer tilhører viewet.

Når vi snakker om gruppering vil jeg referere til ”Begreper” og ”Klasser”. Begreper blir klasser ved å gruppere inn assosiasjoner, men vi referer fortsatt til det samme elementet i modellen.

1.6. Oversikt over oppgaven

Kapittel 1. Innledning

Innledning med beskrivelse av målgruppe, rammer og forutsetninger for masteroppgaven.

Kapittel 2. Krav

Generelle krav til verktøyet

Kapittel 3. Skillet Modell - View

Teori og mulige løsninger på problemstillinger rundt dette

Kapittel 4. Grupperingsalgoritme

Drøfting av algoritmer for gruppering av UML-klassediagram og metode brukt i verktøyet

Kapittel 5. Prototypen

Beskrivelse av dagens prototype og løsninger der

Kapittel 6. Rapport – Empirisk undersøkelse

Rapport fra brukerundersøkelse foretatt i gruppetime med studenter i INF1050

Kapittel 7. Framtidige utbedringer

Videre utbedringer og utvidet funksjonalitet som er ønsket og ligger utenom kravspesifikasjonen til verktøyet

Kapittel 8. Konklusjon

Oppsummering av masteroppgaven med drøfting av prosess og erfaring

Kapittel 9. Litteraturliste

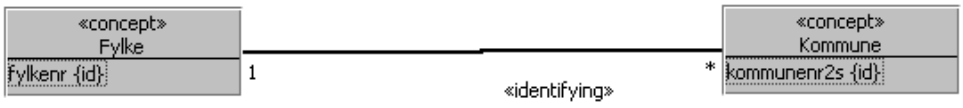
Liste over kilder brukt i oppgaven.

2. Krav til verktøyet

I dette kapitlet tar jeg for meg de kravene som UML-verktøyet skal oppfylle. For å beskrive de forskjellige problemstillingene som oppstår i oppgaven, benytter jeg figurer laget med dagens verktøy dersom ikke annet er spesifisert. De er eksportert til bilde ved hjelp av funksjonalitet i verktøyet

2.1. Om datamodeller

Med en datamodell prøver vi å beskrive vår oppfatning av en del av virkeligheten, kalt vårt interesseområde. Dette gjør vi ved å beskrive virkeligheten med begreper og assosiasjoner mellom begrepene ved konseptualisering [Skagestein 2005]. I dataorienterte klassediagrammer konsentrer vi oss kun om identifiserende attributter, dvs. de som er nødvendig for å kunne identifisere en forekomst av et begrep. For å kunne identifisere en forekomst må vi ha en representasjoner av begrepet. Som eksempel kan vi nevne fødselsnr som representasjon på en person. Dette attributtet vil unikt identifisere en forekomst av begrepet ”Person” og kalles derfor en *identifikator*. I noen tilfeller kan identifikatoren bestå av flere identifiserende attributter. Assosiasjoner kan også bidra til identifisering.
La oss se på begrepet Kommune. En kommune identifiseres med et 4-sifret kommunenr. De to første sifrene er identiske med fylkenummeret til det fylket kommunen ligger i, mens de to siste sifrene beskriver kommunen entydig innen et fylke. Det vil si at det kan finnes andre kommuner som identifiseres av disse to sifrene så lenge de ligger i ulike fylker.
I stedet for å bruke 4-sifret kommunenr kan vi lage en identifiserende assosiasjon mellom Kommune og Fylke, og benytte et 2-sifret kommunenr kalt kommunenr2s, se figur 1. Identifiserende assosiasjoner merkes med stereotypiet **«***identifying***»**.



Figur 1: Kommune identifiseres ned Fylke (ugruppert)

Ved å se på multiplisitetene til assosiasjon kan vi se at et fylke kan ha innholde flere kommuner, men en kommune kan kun ligge i et fylke. Med andre ord kan vi si at det bare ett, og bare ett fylke som kan hjelpe oss med å identifisere en kommune.

2.2. En modell, flere views

Når vi arbeider med UML, er vi stort sett vant til å arbeide direkte på modellen. Tenk å kunne ha mange måter å se på modellen og kunne gjøre forandringer på den i flere forskjellige vinduer. Ideen er at det kun finnes én modell, en modell på det vi kan kalle kanonisk form, se kap 2.3. Den kan ligge urørt mens vi arbeider i de forskjellige viewene. I tillegg vil vi i et view kunne oppdatere modellen. En slik oppdatering vil da få konsekvenser for alle de andre viewene.

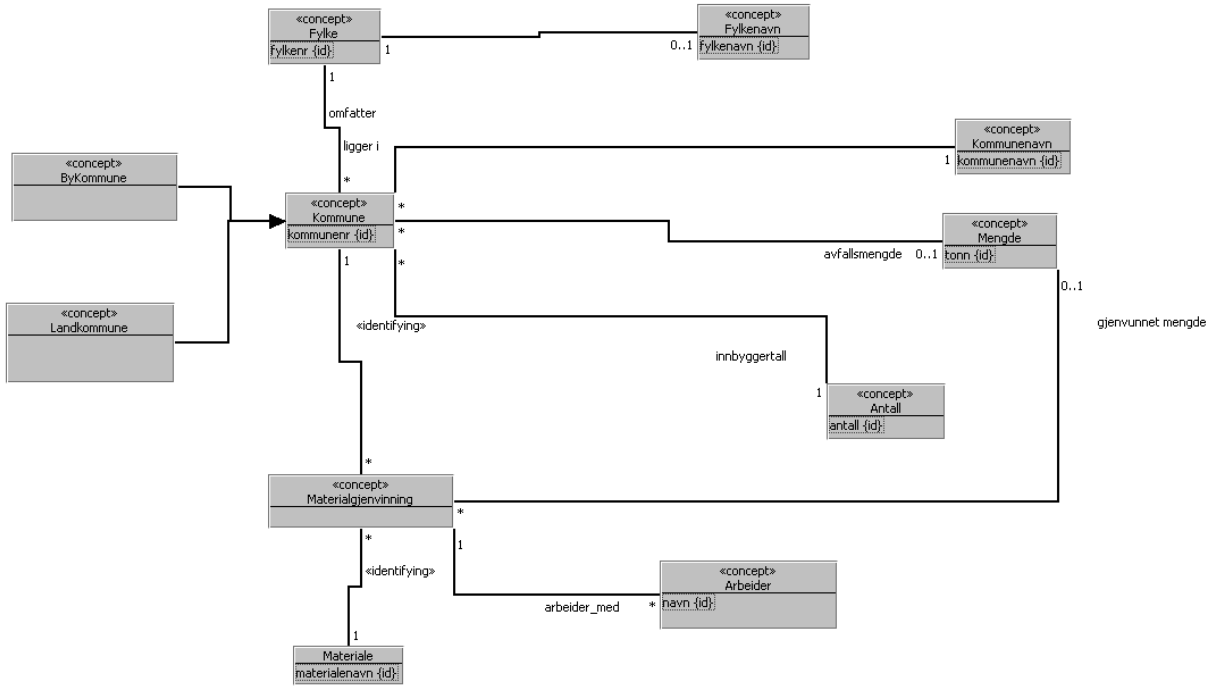
Dette er en velkjent programvarearkitektur. Et eksempel er 3-lags arkitekturen som benyttes f.eks. ved websider. Rådata (modell) lagres i en database mens en server framstiller dataene som hentes fram på skjermen (view) hos en klient. Webserver henter data ut av databasen og behandler dataene med f.eks. PHP-kode som lager HTML-sider som deretter kan tolkes og vises hos klienten. Det er da mulig å presentere de samme dataene på flere forskjellige måter ved å lage forskjellige HTML-sider. Et annet eksempel er mulighetene for å sette opp views mot en database: I en relasjonsdatabase kan vi ved hjelp av SQL-kommandoen CREATE VIEW skaffe oss ulike views mot den samme databasen.

Verktøyet skal altså kunne vise frem modellen på flere måter. Dette vil gjøre at brukeren kan se på modellen i forskjellige vinduer og med forskjellige framstillinger. Vi kan tenke oss at brukeren for eksempel vil ha et view for en ugruppert variant av modellen, se kapittel 2.4.4, et view for en gruppert variant og et view som representer modellen slik den skal modelleres i en relasjonsdatabase, se kapittel 2.5. For å få til dette må jeg skille modelldata og viewdata. Dette skal jeg komme tilbake til i kapittel 3.

2.3. Modell på kanonisk form

Basismodellen som skal være utgangspunkt for alle viewene er en modell på kanonisk form. En datamodell på kanonisk form består kun av klasser stereotypiet som begreper, binære en-til-mange og en-til-en asssosiasjoner med roller og multiplisiteter, generaliseringer, identifiserende attributter og assosiasjoner, samt eventuelle skranker. Alt dette er velkjente konstruksjonselementer i UML-klassediagrammer – se for eksempel The unified modeling reference manual [Rumbaugh, Jacobson & Booch 2004], med unntak av begreps-stereotypien. En klasse stereotypiet som begrep skal ikke innholde andre attributter enn de som bidrar til å identifisere forekomster av klassen. Denne stereoptypien er sentral i den NIAM/ORM-inspirerte UML-klassediagramprofilen som er foreslått av Skagestein [Skagestein 2005] og som denne oppgaven bygger på[2]. Siden "vanlige" attributter da ikke kan forekomme i modellen, må de erstattes med assosiasjoner til andre klasser stereotypiet som begreper. Vi sier også at modellen er *ugruppert*, siden attributtene ikke er gruppert inn i en klasse.

Disse modellelementene er tilstrekkelige for å bygge en komplett datamodell, om enn på et noe lavt abstraksjonsnivå. Figur 2 viser et eksempel på en modell på kanonisk form.

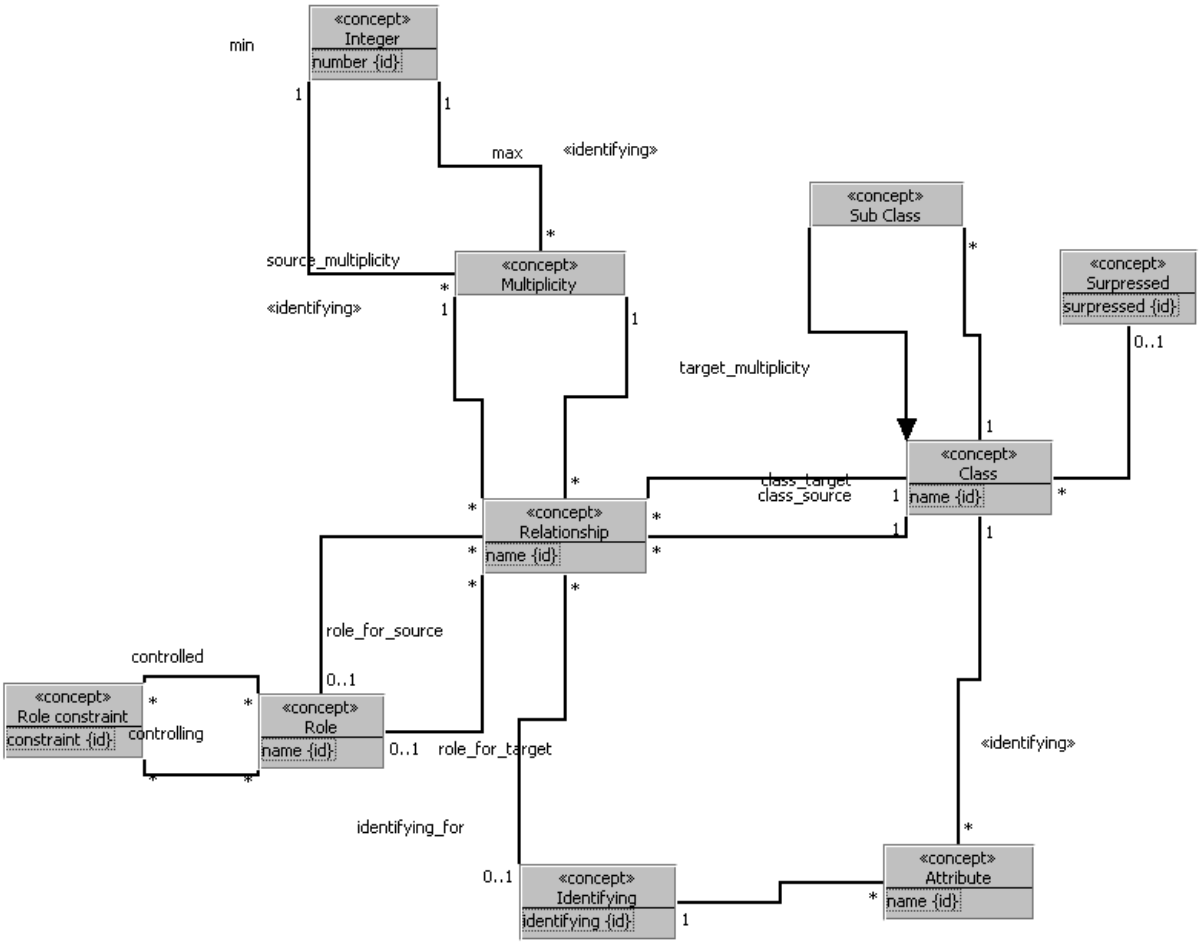


Figur 2: Testmodell - en modell på kanonisk form

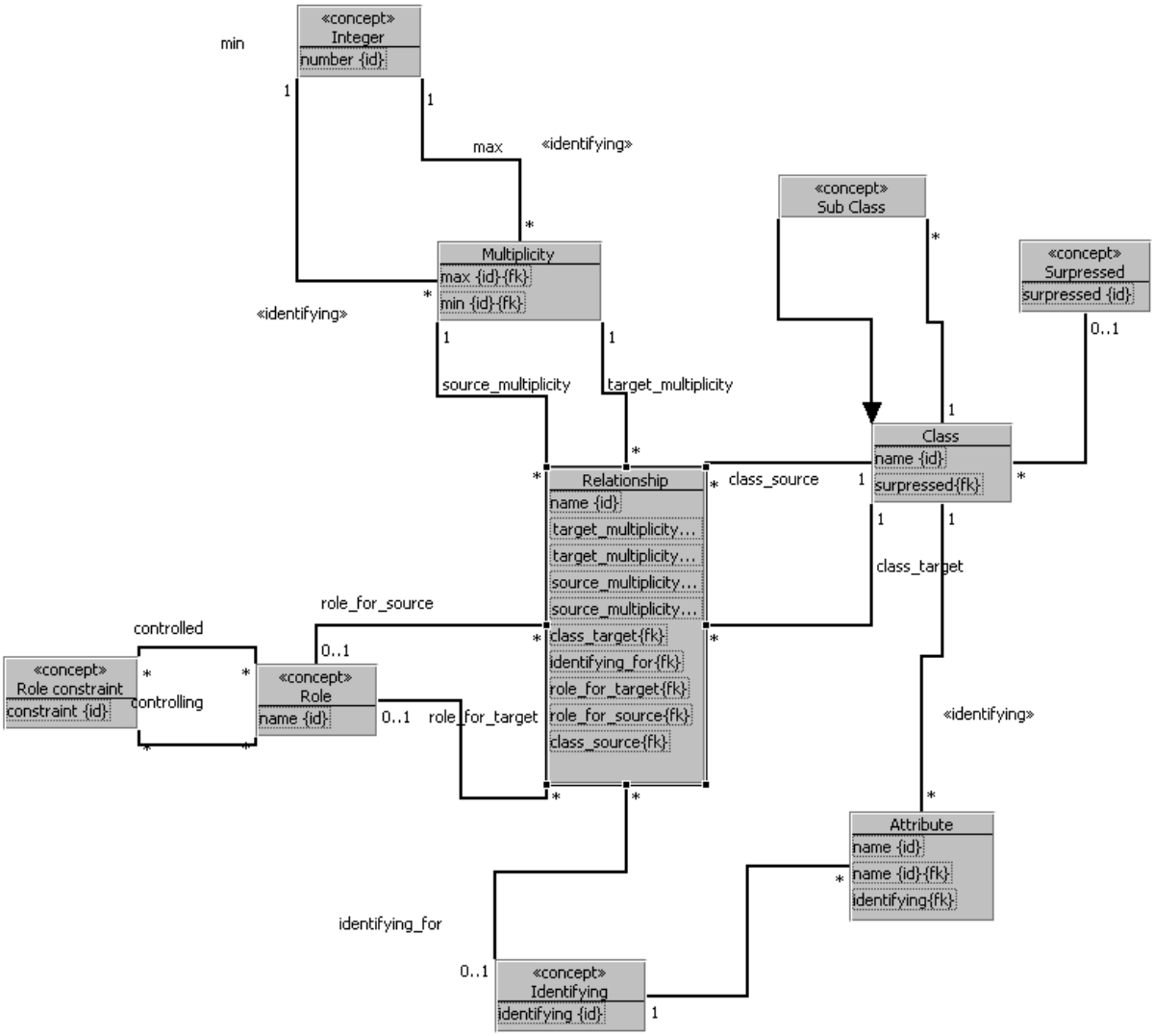
[2] Stereotypien er identisk med det som kalles begreper (NOLOTS) i NIAM/ORM-modelleringsspråket.

Testmodellen i figur 2 vil være utgangspunkt for å eksemplifisere bruk av modell på kanonisk form gjennom hele oppgaven. Den skal beskrive interesseområdet innenfor materialgjenvinning innenfor kommuner. Den beskriver hvilke mengder og hvilke materialer det er snakk om og hvem som jobber med det.

For den kanoniske formen kan vi sette opp en metamodell som vist i figur 2.



Figur 3: Metamodell for modell på kanonisk form (ugruppert)



Figur 4: Metamodell (gruppert) for modell på kanonisk form

Grupperer vi denne modellen ved hjelp av de grupperingsprinsippene som vil bli beskrevet senere (se kapittel 2.4.2) får vi et vanlig klassediagram som vist i figur 4. Utfra denne modellen ser vi egenskapene assosiasjonene, klassene og attributtene kan ha.

Class

En klasse kan ha et navn. Den kan være både en superklasse og en subklasse. Surpressed angir om klassen er undertrykt i modellen.

Relationship

En assosiasjon kan ha et navn. Den er nødt til å ha en source class og en target class for å eksistere, den kan kun ha en av hver som tilfredstiller kravet om binære assosiasjoner. Assosiasjonen kan ha en minimums- og maksimumsmultiplisietet på hver side. I tillegg kan den ha roller for hver av sidene. Det finnes også et attributt som angir om assosiasjonen er identifying eller ikke.

Attribute

Attribute har et navn. Den har også et attributt som forteller om den er identifiserende for sin Class.

2.4. Ulike views

Vi skal i dette avsnittet se på noen abstraksjonsprinsipper over den kanoniske modellen. Disse abstraksjonsprinsippene er grunnlaget for å lage views som gjør det lettere for brukerne både å lese og forstå og å manipulere modellen.

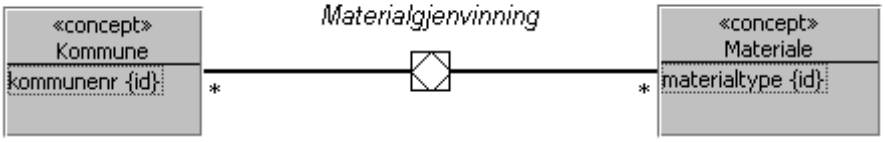
2.4.1. Binære mange-til-mange assosiasjoner og assosiasjoner av høyere orden

Figur 5: Modell med mange-til-mange-assosiasjon på kanonisk form

Vi vil ofte komme over situasjoner der vi får en mange-til-mange assosiasjon mellom to eller flere klasser. I det gjennomgående eksemplet om avfallshåndtering [Skagestein 2005] finner vi forholdet mellom Kommune og Materiale der et Materiale kan gjenvinnes i mange kommuner og en kommune kan gjenvinne mange materialer.

Figur 5 viser oss denne modellen på kanonisk form (et utsnitt av testmodellen i figur 1). Vi beskriver her gjenvinning av materialer i flere kommuner. Begrepet Materialgjenvinning gir

oss informasjon om hvilket materiale som er gjenvunnet i hvilken kommune. Vi ser at Materialgjenvinning ikke selv har noen identifiserende attributter, men har derimot to identifiserende assosiasjoner som identifiserer hver forekomst av begrepet. Når vi har denne type situasjon med en to identifiserende en-til-mange assosiasjoner kan vi også se på dette på en annen måte. Vi kan lage en abstraksjon der vi kun ønsker å se på forholdet Kommune – Materiale uten å gå via Materialgjenvinning. En kommune kan gjenvinne mange materialer. Et materiale kan gjenvinnes i mange kommuner. Dette gir oss en modell som vist i figur 6 – legg merke til multiplisitetene.



Figur 6: Navngitt mange-til-mange assosiasjon

Et koblingspunkt som det vi ser midt på assosiasjonen i figur 6 kan brukes til å knytte enda flere klasser til assosiasjonen, slik at vi får en assosiasjon av høyere orden[3].. Det tillates ikke å benytte et slikt koblingspunkt på en-til-mange-assosiasjoner; fordi dette ville åpne for flere lite meningsfylte modellkonstruksjoner

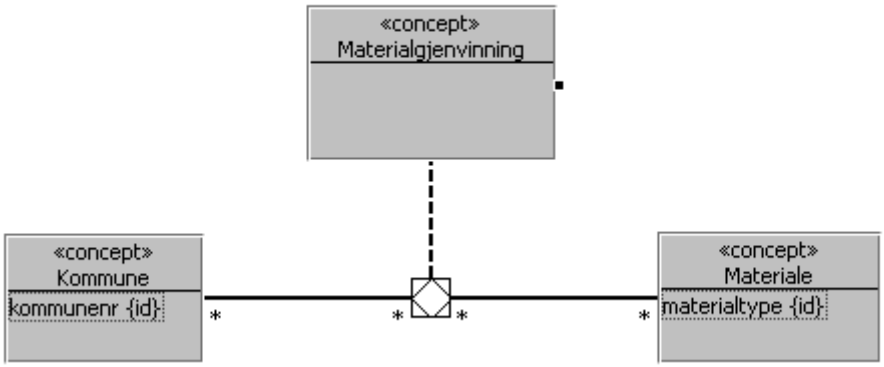
En slik assosiasjon er ikke tillatt i den kanoniske modellen og vil derfor kun eksistere i et view. Det som er viktig å få fram her, er at i vår underliggende modell på kanonisk form har vi fortsatt begrepet Materialgjenvinning.

Ofte ønsker vi å knytte ytterligere assosiasjoner til en mange-til-mange-assosiasjon. I UML-klassediagrammer kan dette ikke gjøres direkte, istedenfor oppretter vi en såkalt *assosiasjonsklasse* tilknyttet mange-til-mange-assosiasjonen, se figur 7. Ytterligere assosiasjoner kan nå knyttes til assosiasjonsklassen – et eksempel på dette er vist i figur 8. I NIAM/ORM er innføringen av en assosiasjonsklasse kjent som en *begrepsdannelse*. Assosiasjonsklasse-konstruksjonen kan transformeres over til den kanoniske formen som er

[3] Ternær-, kvarternær assosiasjon etc...

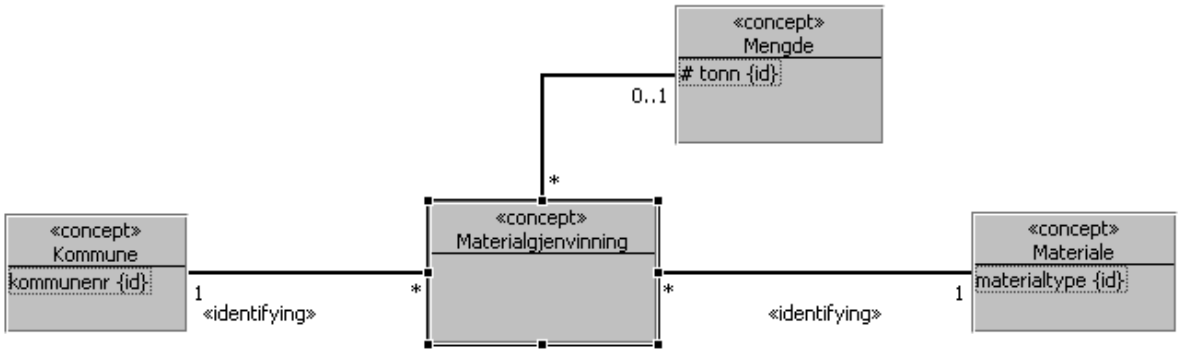
vist i figur 6 ved hjelp av en velkjent transformasjon, se for eksempel [Skagestein 2005] side 131.

I forbindelse med modelleringsverktøyet betrakter vi assosiasjonsklasse-kontruksjonen i figur 7 som et abstrahert view over den kanoniske formen som er vist i figur 5. Forskjellen fra konstruksjonen med en mange-til-mange-assosiasjon som i figur 6, er at assosiasjonsklassen vises eksplisitt slik at vi kan knytte ytterligere assosiasjoner til den. Dessuten kan vi istedenfor å navngi assosiasjonen navngi assosiasjonsklassen. Modelleringsverktøyet skal på en enkel måte kunne veksle mellom de tre konstruksjonene som er vist i figurene 5, 6 og 7.

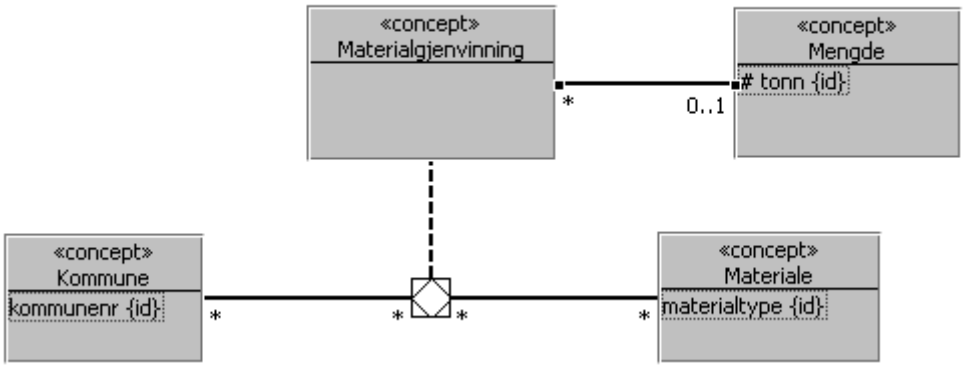


Figur 7: Mange-til-mange assosiasjon med assosiasjonsklasse

Figur 8 viser et utsnitt av testmodellen på kanonisk form. I figur 9 ser vi den samme modellen i et view basert på en assosiasjonsklasse. Assosiasjonsklassen i figur 9 er nøvending fordi vi ikke kan knytte assosiasjonen til mengde direkte til mange-til-mange-assosiasjonen.



Figur 8: Utsnitt av testmodellen på kanonisk form



Figur 9: View basert på assosiasjonsklasse

Situasjonen som er beskrevet over baserer seg på at assosiasjonsklassen Materialgjenvinning i viewet eksisterer som en vanlig klasse med to identifiserende en-til-mange assosiasjoner i den kanoniske modellen.

Hvis vi ønsker å opprette en mange-til-mange relasjon i et view, må vi sørge for at vi har nok opplysninger til å kunne generere den underliggende modellen på kanonisk form. Figur 6 har

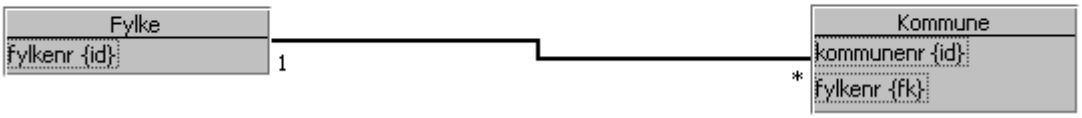
ikke noen assosiasjonsklasse, og det er derfor nødvendig å navngi mange-til-mange-assosiasjonen for å kunne navngi klassen mellom de to identifiserende assosiasjonene i den underliggende modellen. Modelleringsverktøyet vil derfor kreve navngivning av mange-til-mange assosiasjoner, eller rettere sagt en navngivning av koblingspunktet. Dette navnet vil gi navnet på klassen som skal ligge i den kanoniske modellen. For å gjøre det enkelt for brukeren, kan vi tillate at brukeren unnlater å angi et navn: istedenfor kan vi generere et default navn basert på navnene på de klassene som knyttes sammen av assosiasjonen, i dette tilfellet ”kommune_materiale”. Dette navnet kan brukeren endre senere for å gi et mer beskrivende navn hvis han skulle ønske det.

Dette gjør at vi nå kan fritt bytte view mellom figur 5, 6 og 7 gitt at begrepet mellom de to identifiserende assosiasjonene i modellen på kanonisk form ikke har noen assosiasjoner mot noen andre begreper. Hvis begrepet har andre assosiasjoner vil vi kun vise det i henhold til figur 8 og 9.

2.4.2. Gruppering

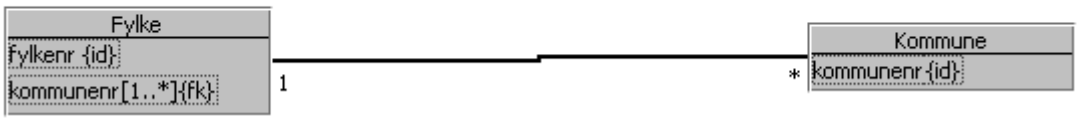
Gruppering innebærer å opprette et attributt i en klasse som representasjon for en assosiasjon ut fra denne klassen. Et slikt attributt kalles ofte for en fremmednøkkel – en term som er hentet fra relasjonsdatabaseteorien. I en forekomst av klassen vil verdien i dette attributtet være den samme som verdien i det identifiserende attributtet i klasseforekomsten på motsatt side av assosasjonen. Dersom identifikatoren i klassen på motsatt side er sammensatt, dvs. at denne klassen har flere identifiserende attributter og/eller assosiasjonr, må fremmednøkkelen også være sammensatt, altså inneholde like mange attributter.

Dersom assosiasjonen har maksimumsmultiplisiteten 1 på motsatt side, vil fremmednøkkelen bli et enkelt attributt. Dersom maksimumsmultiplisiteten er større en 1, (som regel *), vil fremmednøkkelen ta form av en "collection" av enkle attributter. I modelleringsfasen kan vi tenke oss at modelleringsverktøyet kan gi oss views basert på begge varianter. Så snart vi benytter gruppering for å lage datastrukturer for bestemte realiseringsplattformer, som relasjonsdatabaser eller objektorienterte strukturer, vil dette begrense hvilke grupperingsvarianter vi kan benytte, se kapittel 2.5.



Figur 10: Kommune identifiseres med Fylke (gruppert)

Begrepene Fylke og Kommune viser hvordan vi kan konseputalisere forholdet mellom dem er. Etter gruppering vil Kommune inneholde ”fylkenr” som fremmednøkkel. Men vi kan også tenke oss en annen måte å vise dette på i et view. Vi kan også tenke oss at Fylke har en liste over ”sine” kommuner, altså kommuner som ligger i fylket. En slik liste kalles en ”collection”. Figur 11 viser hvordan dette kan løses for vår situasjon.



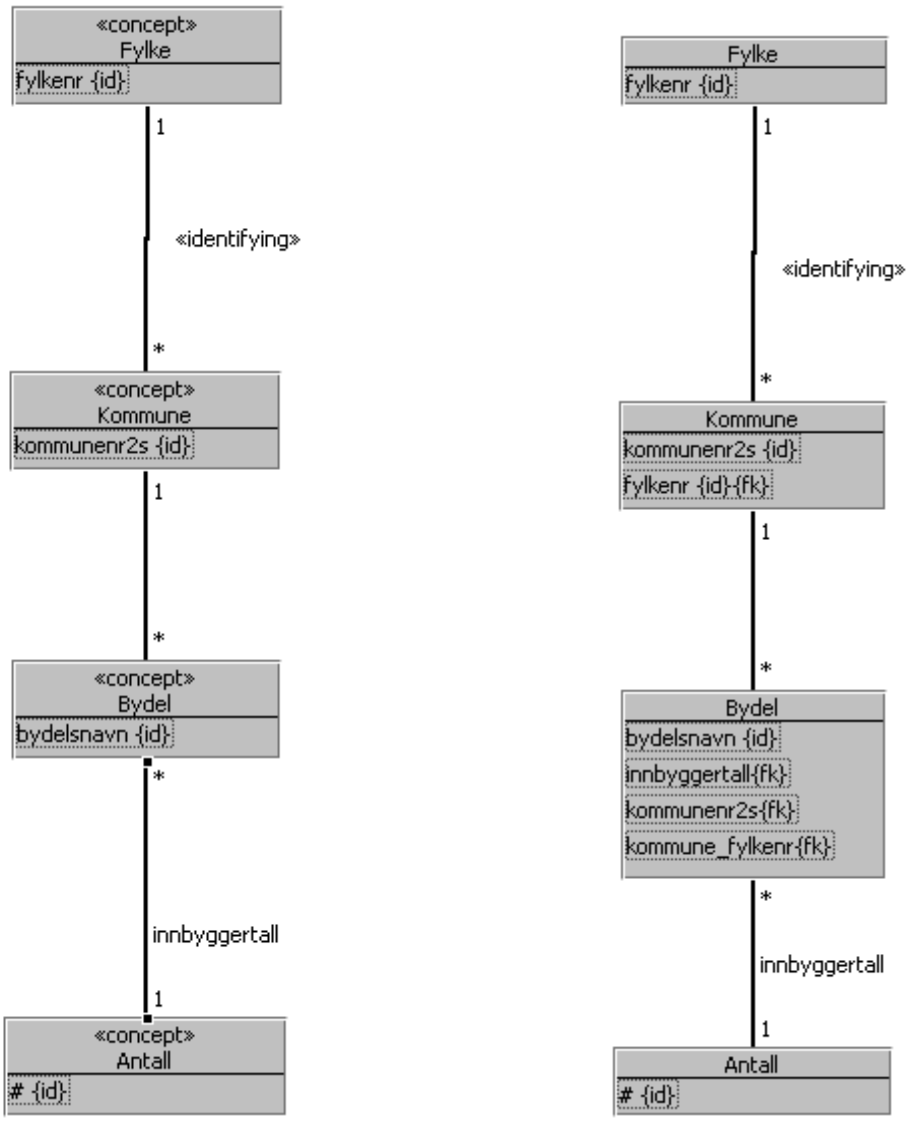
Figur 11: Fylke med en collection av kommuner

Sammensatte identifikatorer

Hvis man ved i et begrep går fra en enkelt identifikator til en sammensatt identifikator vil dette ha konsekvenser for den referende begrepet. Da må man opprette et fremmednøkkelattributt pr identifiserende attributt i klassen den referer til.

Forplanting av fremmednøkler

Hvis en klasse har en fremmednøkkel som inngår i klassen identifikator vil denne kunne forplante seg til andre klasser som har en assosiasjon til denne klassen. Da må vi bevege oss rekursivt videre i trestrukturen for å kunne genere ”riktig” fremmednøkkel i bunnen av treet. Figur 12 viser en slik situasjon.

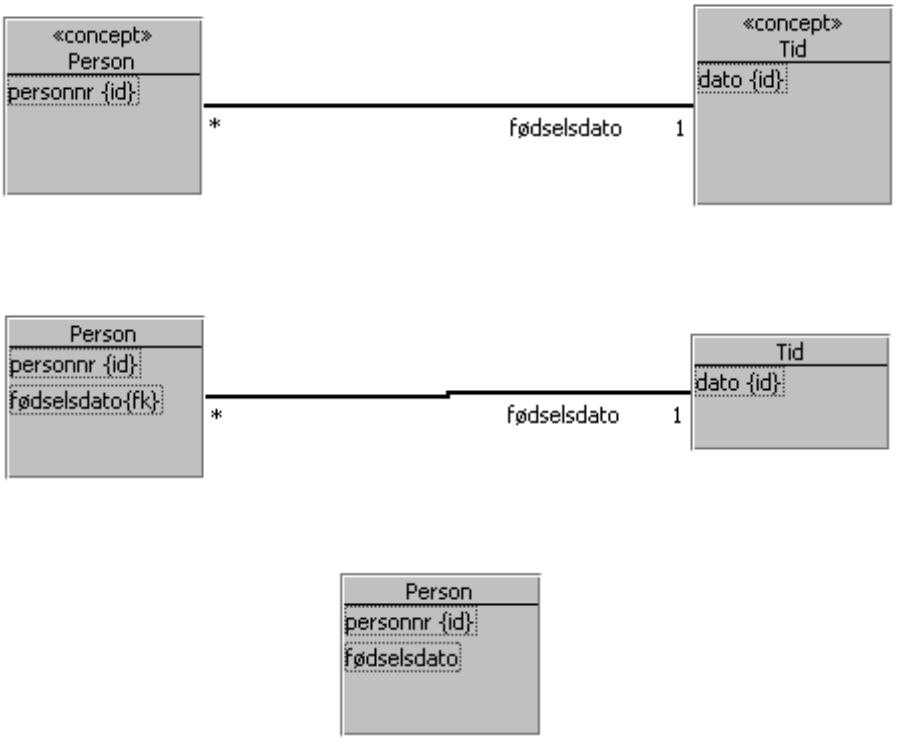


Figur 12: Fremmednøkler ved forplanting (ugruppert og gruppert)

Undertrykking

Ved gruppering av en datamodell på kanonisk form vil det som regel oppstå en mengde klasser som ikke inneholder andre attributter enn de identifiserende – dette er klasser der det ikke er blitt generert noen fremmednøkler. Det dreier seg ofte om klasser for mål, beløp, tidspunkt og antall. Verdiene som finnes i forekomstene av disse klassene vil også finnes som verdier i fremmednøkler i andre klasser, og vi kan derfor vurdere å undertrykke disse klassene etter grupperingen. Ved å undertrykke en klasse vil fremmednøkler generert fra denne klassen gjøres om til vanlige attributter. Et eksempel er en modell som omfatter en assosiasjon

mellom klassene Person og Tidspunkt, der Tidspunkt spiller rollen som fødselsdag. I den grupperte modellen er det unødvendig med en egen klasse Tidspunkt for alle mulige forekomster av en fødselsdag. Framgangsmåten for denne undertrykking blir beskrevet i figur 13.



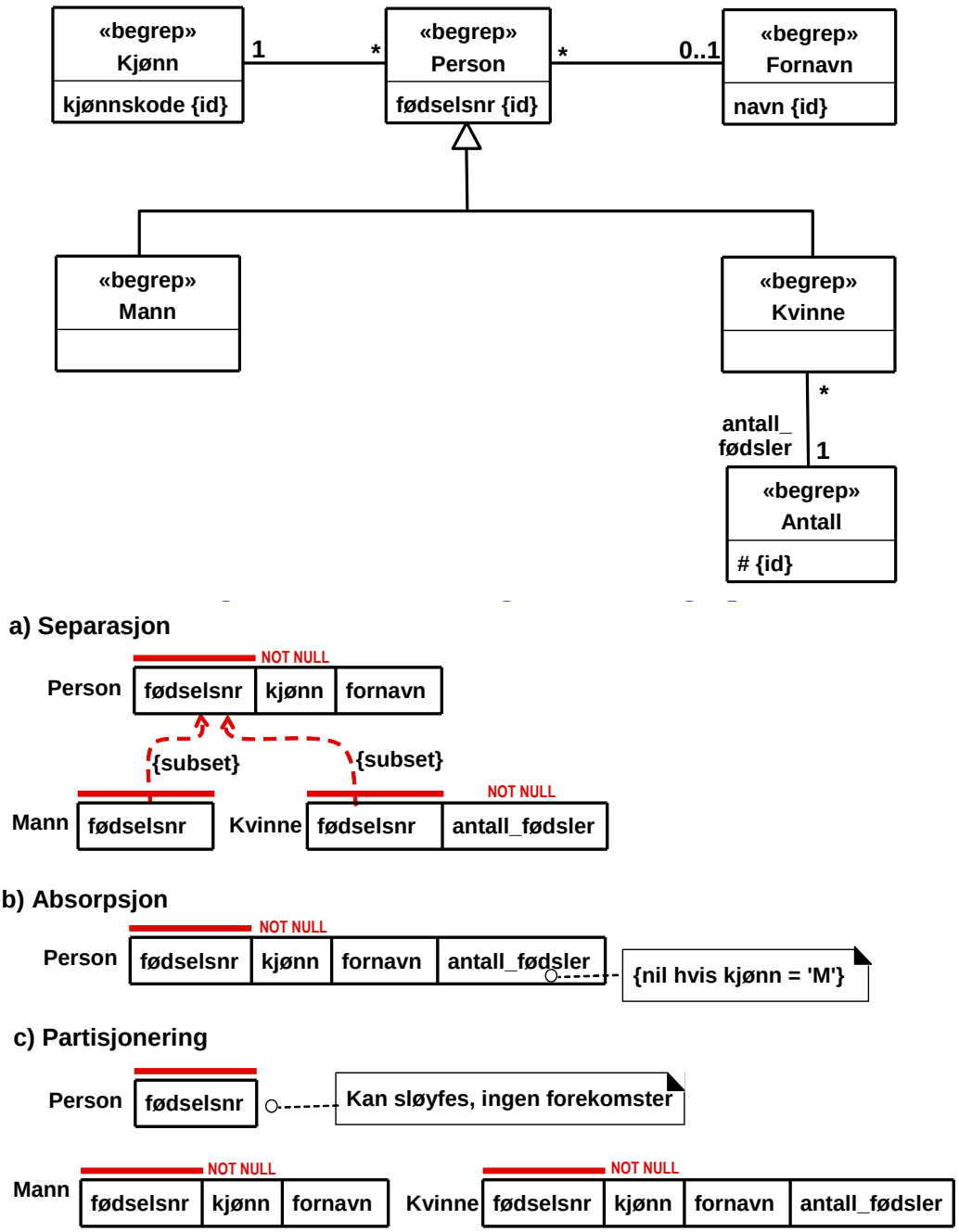
Figur 13: Gruppering og undertrykking

I verktøyet skal en klasse kunne undertrykkes. Når en klasse undertrykkes settes et attributt i beskrivelsen av klassen til verdien ”supressed” som dermed indikerer klassens status som undertrykt. Vi sletter ikke klassen fra modellen. Slik kan vi opprettholde fremmednøklene som er konsekvens av gruppering av den undertrykte klassen som ”vanlige” attributter. Disse attributtene vil da ikke ha {fk} bak attributtnavnet, og klassen og dens assosiasjoner vil ikke være synlig i noen views.

Gruppering av begreper med underbegreper (generalisering)

Underbegreper kan ikke håndteres av relasjonsdatabaser og må derfor erstattes med andre konstruksjoner under grupperingen. Det tre forskjellige måter å håndtere disse på.

Teksten og figuren som beskriver disse metodene er hentet fra [Skagestein 2005] side 140-142.



Figur 14: Håndtering av underbegreper

Separasjon

Vi oppretter en tabell for superbegrepet og en tabell for hvert av underbegrepene – se figur 13a. Underbegrepene må få med seg den identifiserende representasjonen som de arver fra superbegrepet. Vi får da en referanseintegritet fra primærnøkkelen i tabellene som er oppstått fra underbegrepene til primærnøkkelen i tabellen som er oppstått fra superbegrepet.
Eller kan vi merke oss at i dette eksemplet er tabellen Mann egentlig overflødig og kan sløyfes.

Absorpsjon

Vi generaliserer ved å slå underbegrepene sammen med superbegrepet – se figur 13b. Dermed vil homogenitetsregelen ikke lenger være oppfylt, og vi kan ikke ut fra databasestrukturen se hvilke begreper de enkelte forekomster tilhører. Dette må vi istedenfor få vite ut fra verdiene i fremmednøkkelen som oppstår ut fra assosiasjonen mot Kjønn, som er en såkalt *diskriminerende assosiasjon*. Vi kan da kontrollere at vi har nil i de fremmednøklene som er irrelevante for hver enkelt begrepstype.

I eksemplet her er det bare personer som har verdien ”K” i kjønn som kan ha en verdi i antall_fødsler. Legg merke til at vi får to typer nil: Én type som konsekvens av absorpsjonen – denne typen kan forekomme i antall_fødsler – , og en annen type som konsekvens av minimumsmultiplisitet lik 0 – denne typen kan forekomme i fornavn. Den første typen kan ikke bli erstattet av en verdi så lenge undertypedefinisjonen ikke endrer seg, det kan derimot den andre typen.

Partisjonering

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 fremmednøkler som kan arves fra tabellen for superbegrepet. En forekomst i en tabell for et underbegrep skal nå ha verdier i alle fremmednøkler, også de nedarvede. Deretter sørger vi for at disse nedarvede fremmednøklene blir fylt med verdier som kopieres fra tabellen for superbegrepet – se figur 14c. Hensikten med dette er at vi skal kunne hente ut alle opplysninger tilknyttet en forekomst av et

underbegrep fra en eneste tabell, uten å måtte komplettere med opplysninger tilknyttet superbegrepet.

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 dobbeltlagring av opplysninger. I det andre tilfelle kan vi risikere å måtte gå gjennom samtlige tabeller for å finne en forekomst, dersom vi ikke ved om hvilket begrep forekomsten tilhører.

Hvis vi velger den andre løsningen, og underbegrepene er uttømmende i forhold til superbegrepet, får tabellen for superbegrepet ingen forekomster og kan sløyfes! Dette er tilfellet for eksemplet i figur 14.

Skranker

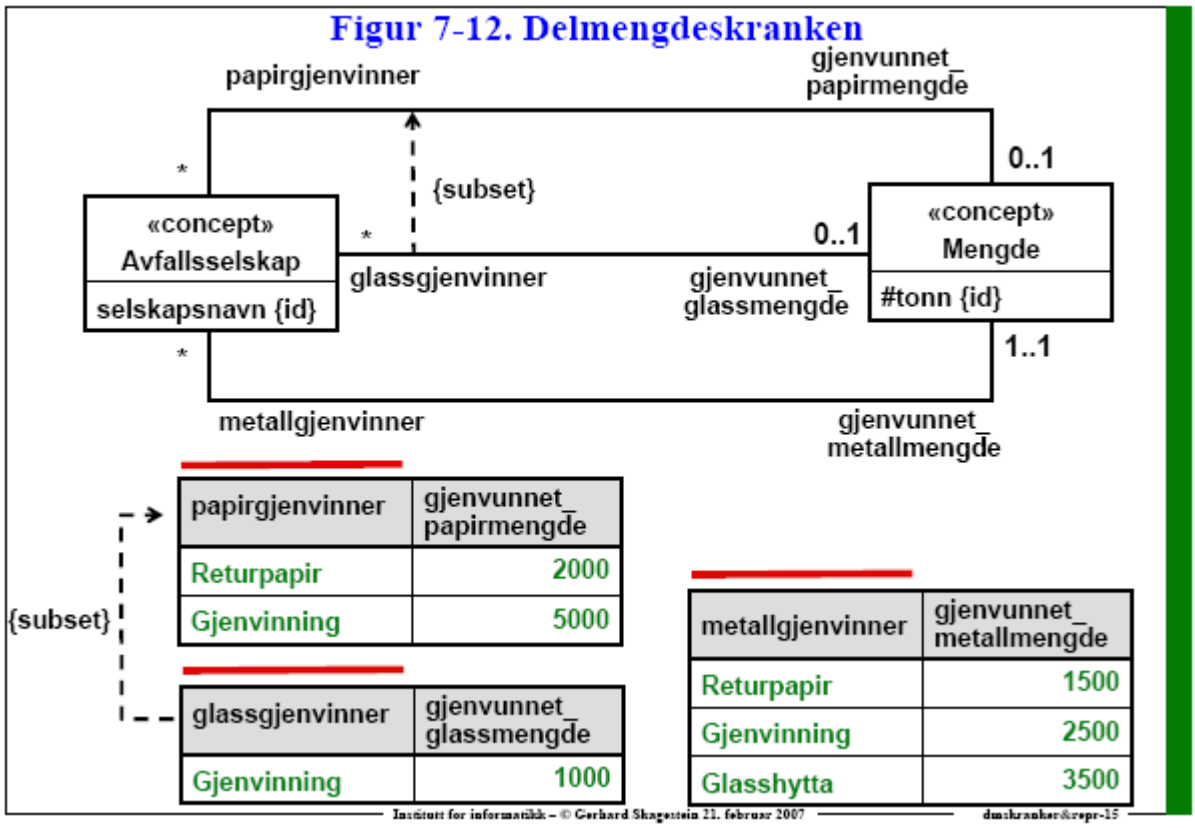
Det finnes mange typer skranker som beskriver de regler som gjelder for interesseområdet i virkeligheten. Jeg vil her gjøre en avgrensning på hvilke skranker som verktøyet håndterer. De to som håndteres er entydighetsskranken og påkrevd rolle.

Entydighetsskranken er den viktigste skranken da den uttrykker at alle forekomster i en rolle må være forskjellige fra hverandre, mao. alle forekomster skal være unike. Dette blir gitt med de identifiserende attributtene til et begrep.

En påkrevd rolle forteller oss at enhver forekomst av et begrep i modellen må den hverfall finnes som forekomst i den påkrevde rollen. Eksempel på dette er forholdet mellom Kommune og Fylke i testmodellen vår. I henhold til minimumsmultiplisiteten på Fylke sin side som er 1, gir dette at en kommune må være knyttet til et fylke. Dette gir at en kommune har rollen ”kommune” i tabellen.

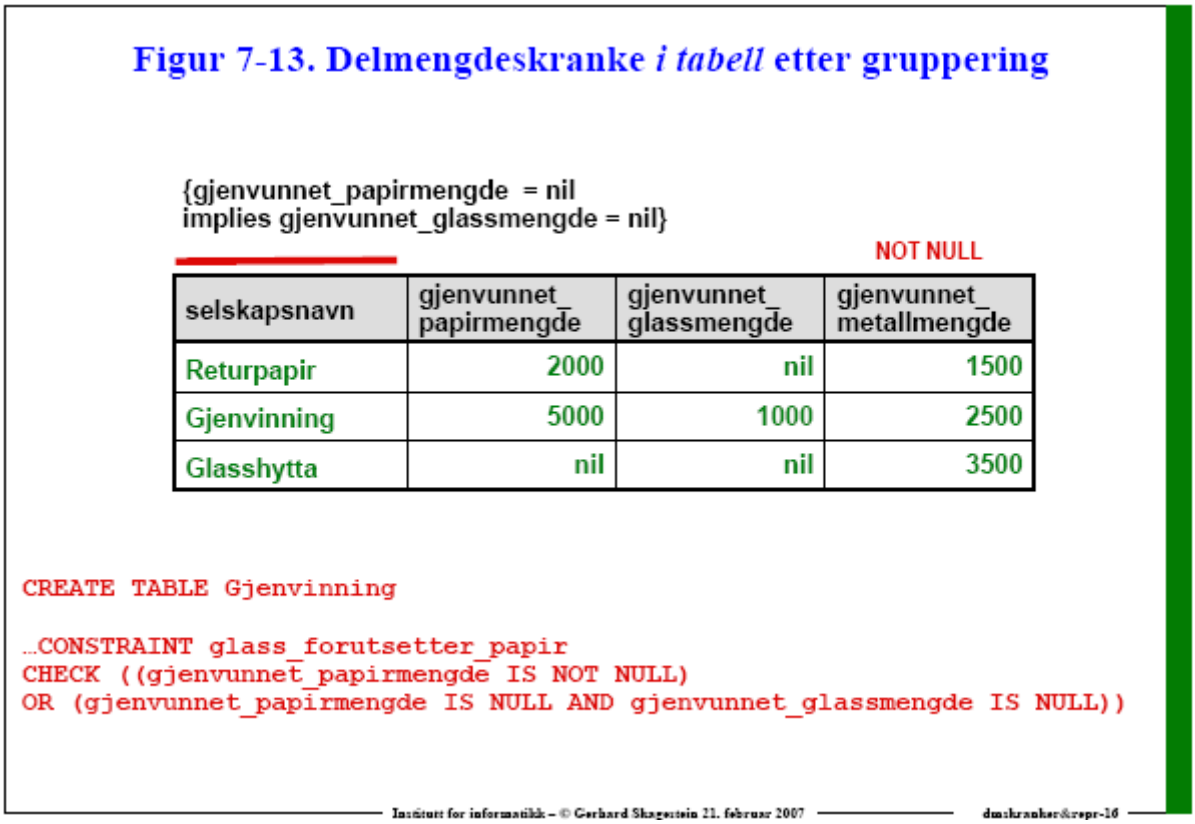
For å gå over til en relasjonsdatabase må vi kunne representere skrankene der. Dette gjøres bl.a. ved å sette skrankene NULL og NOT NULL på attributtene. NULL gir at attributtet ikke må ha en verdi og NOT NULL gir at attributtet må ha en verdi. Identifiserende attributter eller primærnøkler som de kalles i en relasjonsdatabase må ha verdi for å gi mening. I en relasjonsdatabase vil disse derfor settes til NOT NULL. Fremmednøklene kan derimot være NULL eller NOT NULL. For å finne hvilken skranke vi skal sette må vi se på

minimumsmultiplisiteten på motsatt side av assosiasjonen. Hvis den er 1 er det gitt at fremmednøkkel må ha en verdi og settes til NOT NULL. Er den derimot 0 betyr det at fremmednøkkelen kan, men må ikke ha en verdi og settes derfor til NULL.



Figur 15: Delmengdeskranke på ugruppert modell [Skagestein 2007]

Skranker vises i UML-profilen vår på denne formen *{skranke}*. Vi ser her at man har skranken subset. Det den sier er at for at Gjenvinning i tabellen glassgjenvinner, må Gjenvinning finnes i tabellen papirgjenvinner.



Figur 16: Delmengdeskranke i tabell etter gruppering [Skagestein 2007]

Etter gruppering kan vi uttrykke det på denne måten {gjenvunnet_papirmengde = nil implies gjenvunnet_glassmengde = nil}. Med andre ord vil det si at hvis Gjenvinning ikke har verdi for gjenvunnet_papirmengde, kan den heller ikke ha verdi for gjenvunnet_glassmengde som for Glasshytta i eksempelet over.

Forskjeller mellom ugruppert og gruppert modell

Jeg vil nå gå igjennom de forskjellige problemstillingene ved gruppert/ugruppert modell og hvorfor jeg velger å plassere de forskjellige elementene i modelldata eller viewdata.

«Concept»:

En ugruppert klasse merkes med stereotypien «concept». En gruppert klasse oppfyller ikke lenger kravet til begrepsstereotypien og er dermed ikke lenger et begrep, men en vanlig klasse. I modellen på kanonisk form tillater vi kun begreper. Dette gjør at vi må lagre informasjon om en klasse er gruppert eller ikke i viewet.

Fremmednøkler:

Ved gruppering dannes fremmednøkler. En fremmednøkkel er et resultat av assosiasjonen til en annen klasse. Dermed må en fremmednøkkel være en atributt til viewet og ikke eksistere i modellen. Hvis vi så skulle velge å undertrykke klassen som fremmednøkkelen referer til vil den bli gjort til et vanlig attributt i viewet.

2.4.3. Andre abstraksjoner

Et view kan inneholde noen eller alle klasser som finnes i modellen på den kanoniske form. Tenkt funksjonalitet er her at når bruker oppretter et nytt view, vil han velge hvilke klasser som skal være med. Denne listen vil være tilgjengelig ved forespørsel av bruker og han kan til enhver tid endre sitt utvalg av klasser i viewet.

2.4.4. Viewet – Kombinerte abstraksjoner

En bruker skal ikke være bundet til én abstraksjon pr. view. Med dette mener jeg at en bruker skal stå fritt til å benytte en kombinasjon av de abstraksjonene beskrevet i kapittel 2.3.1, 2.3.2 og 2.3.3.

Fordeler ved å benytte view:

- Modellen kan forbli som den er, samtidig som vi kan representere forskjellige views til forskjellig bruk og brukere.
- Vi kan illustrere veien fra konseptualisert datamodell til relasjonsdatabase.
- Vi kan lage forskjellige varianter av modellen som er klare for overgang til relasjonsdatabase.

Alle disse fordelene kan være til god nytte spesielt for målgruppen vår (se kapittel 1.2). Dette vil kunne øke deres innsikt i til allment brukte modelleringskonsepter, hva gruppering er, hvorfor vi gjør det og generelt forstå veien fra datamodell til relasjonsdatabase.

2.5. Gruppering av modell på kanonisk form til relasjonsdatabasestruktur

"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."

– Gerhard Skagestein [Skagestein 2005], side 135

Gruppering av assosiasjoner

For å kunne gå fra en datamodell til en relasjonsdatabasestruktur må vi erstatte assosiasjonene mellom klassene med fremmednøkler i klassene. Vi må altså gjennomføre en gruppering.

En fremmednøkkel er, som vi har sett, et attributt som identifiserer en annen klasse på grunnlag av assosiasjonen mellom dem. Når vi grupperer til en relasjonsdatabasestruktur, er et av kravene til denne fremmednøkkelen er at den identifiserer kun én forekomst av en klasse, altså må maksimumsmultiplisiteten på motsatt side av assosiasjonen være lik 1.

Den mest vanlige assosiasjoner vi møter i modeller på kanonisk form, er en-til-mange, vist i diagrammene ved multiplisitetene 1..1 og 0..*, på kortform som 1 på ene siden av assosiasjonen og * på den andre. Disse grupperes ved at vi generer fremmednøkler der maksimumsmultiplisiteten er 1 på motsatt side[4]. Ved en-til-en assiasjoner må brukeren selv ofte ta stilling til på hvilken side fremmednøkkelen skal genereres. (Dersom den ene minimumsmultiplisiteten er 0 og den andre 1, er det nærliggende å generere fremmednøkkelen der minimumsmultiplisiteten er 1 på motsatt side.) Det må også være mulig å få generert en fremmednøkkel i begge klasser. I så fall har vi en redundans i datastrukturen, og applikasjonene må sikre at fremmednøkkelverdiene til enhver tid stemmer overens. (Hvis Per er gift med Tove, må Tove være gift med Per, ikke med Pål).

[4] Se figur 10.

2.6. Oppdatering av modell - funksjonalitetskrav

Noen endringer som bruker gjør i et view vil ha konsekvenser for vår modell på kanonisk form. En slik endring i et view vil dermed få konsekvenser for alle viewene på modellen i henhold til kravene om én modell – flere views. Spesifisert er det disse endringene som gjør at vår modell må oppdateres:

Opprette klasser:
Alle klasser eksisterer i modellen. Opprettelse av klasser må derfor skje i modellen.

Endre navn på klasser:
Endring av navn på en klasse må skje på modell. Dette kan få konsekvenser for navngivningen av fremmednøkler under gruppering.

Slette klasser:
Alle klasser eksisterer i modellen. Sletting av klasser skjer derfor i modellen. Konsekvenser av sletting er at fremmednøkler generert fra denne klassen også må slettes.

Undertrykke en klasse:
Når en klasse undertrykkes, vil den ikke slettes fra modellen og vil ikke være synlig fra noen views. Hvis en annen klasse har fremmednøkler fra en undertrykt klasse vil disse fremmednøklene se ut og oppføre seg som vanlige attributter, dvs. uten {fk} bak attributtnavnet.

Endre navn på attributter:
Endring av navn på et attributt må skje i modell. Dette kan få konsekvenser for navngivningen av fremmednøkler under gruppering.

Opprette og slette assosiasjoner:
Assosiasjonene er en del av modellen. Ved sletting vil det kunne få konsekvens for genererte fremmednøkler og disse fremmednøklene skal slettes.

Endre navn på assosiasjoner:
Vi må kunne endre navn på assosiasjoner. Grunnen til navngivningen er at vi har krav om navn på mange-til-mange assosiasjoner[5]. Fører til konsekvenser i views ved at assosiasjonsklasse til mange-til-mange assosiasjonen må skifte navn.

Endre av multiplisitet på assosiasjoner:
Multiplisiteter er en del av modellen og endringer vil få konsekvenser for gruppering.

Endre roller på assosiasjoner:
Rollen en klasse har ovenfor en annen er med på navngivning av fremmednøkler og vil få konsekvenser i views.

2.7. Brukergrensesnitt og feilhåndtering

"Creating a bad GUI is really, really easy. Creating a "good" GUI is really, really hard." –
Eric M. Burke [Burke 2004]

I denne oppgaven er hovedfokus på mulighetene for å fremstille en modell i flere forskjellige views. Da må vi heller ikke glemme måten vi vil presentere disse viewene mot brukeren. Brukergrensesnittet (GUI) er ikke bare brukerens vindu til modellen og funksjonaliteten, men er ofte det som bestemmer inntrykket brukeren sitter igjen med etter bruk av verktøyet. Det går an å lage programmer med masse og god funksjonalitet, men som aldri vil slå an fordi det er for vanskelig, tungvint og/eller komplisert å bruke pga. av for dårlig brukergrensesnitt. Det er viktig å at brukeren ikke føler seg "dum" ved at han ikke får ting til. Arbeidslivet kan være stressende nok om man ikke må krangle med et program også, noe jeg kan relatere til i min egen arbeidssituasjon.

Et viktig utgangspunkt for designet er å sette seg i inn i brukerens tankegang og arbeidsprosess. Hindringer i arbeidsflyt kan være grunn til å ikke bruke programmet.

[5] Ref. Kapittel 2.2

Brukeren skal få følelsen av å ha kontroll på verktøyet. Det gjøres ved å gi bruker oversikt i over hva som skjer i verktøyet og at man får respons på hva man gjør. Denne responsen kan være at brukeren ser forandringene som skjer når en trykker på en knapp. I verktøyet der grupperingen er så sentral er det viktig at brukeren enkelt ser forandringene på klassene han den trykker på ”Grupper”-knappen. Som et pedagogisk verktøy er forståelsen av grupperingsfunksjonaliteten essensiell.
Gode tilbakemeldinger gir brukeren oversikt. Hvis brukeren prøver å gjøre noe som ikke er tillatt er det viktig at brukeren forstår hvorfor han ikke får lov til å utføre operasjonen, i stedet for å bare ”gråe” ut en knapp eller bare la bruker trykke og ingenting skjer. Disse tilbakemeldingene skal også være så korte og konsise som mulige. Når et vindu med et error-ikon popper opp er det ofte kjapt å trykke på OK knappen hvis en lang og ofte uforståelig feilmelding skulle dukke opp. Det går også an å benytte seg av andre tilbakemeldingskanaler som lyd. Skal man benytte seg av dette synes jeg det er viktig å ha muligheten til å skru av tilbakemeldinger i form av lyd. Lyd kan være en effektiv måte å få oppmerksomheten til brukeren, men kan også være kilde til irritasjon. Jeg har ikke tall på de gangene jeg har hatt høyt volum på og en MSN melding med påfølgelgende pip har poppet opp.

En annen viktig faktor for arbeidsflyten er responstiden til programmet. Lange lastinger og trege operasjoner kan føre til at brukeren legger det vekk. I noen tilfeller må man bare godta at ting tar tid f.eks. ved lange matematiske utregninger, krevende databasespørringer etc. Her kommer vi igjen til temaet om feedback til brukeren. Programmet bør her fortelle brukeren: ”Her står du nå”. Typiske og gode løsninger på dette er bruk av ”progressbar” og angivelse av estimert tid til fullførelse av operasjonen. For å gi brukeren kontroll er det også viktig å kunne la brukeren stoppe prosesser. Kanskje brukeren angrer seg i det han ser at operasjonen han ønsker å gjøre vil ta en halvtime? Dette kan løses ved å ha en avbryt knapp som kan redde en fra å drepe programmet og kanskje miste ikke-lagret informasjon.
Programmer trenger ikke være perfekte, så lenge de ikke stopper opp og forhinderer at man kan arbeide videre med dem.
Feil må kunne tilgis både fra brukeren og programmet. Ved å innkapsle alle handlinger vil man kunne benytte ”undo/redo” funksjonalitet. Dette vil gi muligheten til å angre hvis man f.eks. har gjort noe feil. Programmet kan også gjøre feil. Med god feilhåndtering kan man unngå et ”kræsj” ved at programmet må avsluttes og man kanskje mister verdifull ikke-lagret

data. Skulle derimot uhellet være ute (kanskje datamaskinen henger seg pga. noe utenom programmets rekkevidde) er autolagring en smart funksjonalitet. Automatisk lagring ved bestemte tidsintervaller kan gjøre at brukeren slipper å begynne arbeidet helt på nytt.

Kanskje det viktigste med å få et godt brukerdesign er å gjøre det intuitivt. Bruker skal enkelt forstå hvordan han kan benytte seg av verktøyet uten å gå gjennom en haug med dokumentasjon. Dette kan gjøres på flere områder. Lett tilgjengelighet til basis funksjonalitet og plasseringen i grensesnittet hjelper. Utseende på elementer og knapper hjelper også brukeren å se hvordan han skal benytte seg av programmet. Ved å la klassene se like ut som den NIAM/ORM- inspirerte UML-profilen vil dette være til hjelp for forståelsen av verktøyet. Man bør også benytte seg av standarder for brukergrensesnitt. De fleste GUI baserte programmer baserer seg på godt etablerte standarder. Eksempler på dette er å ha en filmeny øverst og en toolbar under med lett tilgjengelige knapper. Under kommer hovedvinduet der man arbeider. Underst kommer gjerne en statuslinje.
Andre standariserte funksjoner som å kunne printe ved å klikke CTRL-P på tastaturet er ting som også bør legges inn.

2.8. Open Source

" Open source describes the principles and methodologies to promote open access to the production and design process for various goods, products, resources and technical conclusions or advice." [Wiki – Open Source]

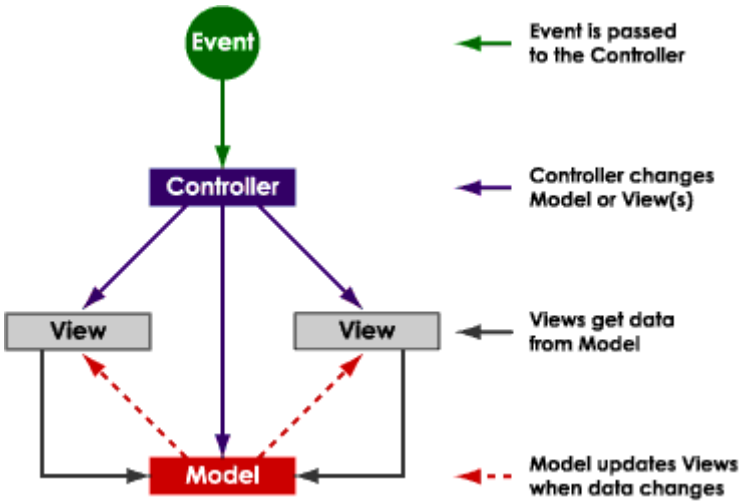
Verktøyet skal være Open Source. Det vil si at både kildekoden og kompilert applikasjon skal være gratis og fritt tilgjengelig. Da verktøyet er resultat av studenters masterstudier vil tiden det blir oppdatert og videreutviklet hovedsaklig være begrenset til studentenes studietid. Ved å frigi kildekoden vil det være mulig for andre utviklere å vedlikeholde, videreutvikle og ikke minst forbedre verktøyet. Ved å gjøre verktøyet fritt tilgjengelig (f.eks. distribusjon på Internett) kan man også øke brukermassen. Desto flere brukere man har, desto tidligere vil man kunne oppdage feil. Dette vil bidra til å kvalitetssikre applikasjonen.

En av de beste tingene ved Open Source nettverket er all kunnskapen som finnes. Community'et er bygget opp av folk som på frivillig basis kommer sammen og utvikler på hobbybasis. Drøfting av metoder og løsninger bidra til et bedre produkt.

Det er dog ikke bare positive sider ved Open Source miljøet. "Jo flere kokker, jo mere søl" er et kjent uttrykk og det kan også brukes i denne sammenhengen, spesielt med har sine preferanser om hvordan ting skal gjøres og utvikles.

3. Skillet Modell - View

3.1. Modell – View – Kontroller (MVC) konseptet



Figur 17: Presentasjon av MVC konseptet [eNode 2002]

By the end of this chapter, you should be mumbling the MVC mantra to yourself at the grocery store—"the model manages data and logic, the view creates the interface, and the controller processes user input." [Moock 2004]

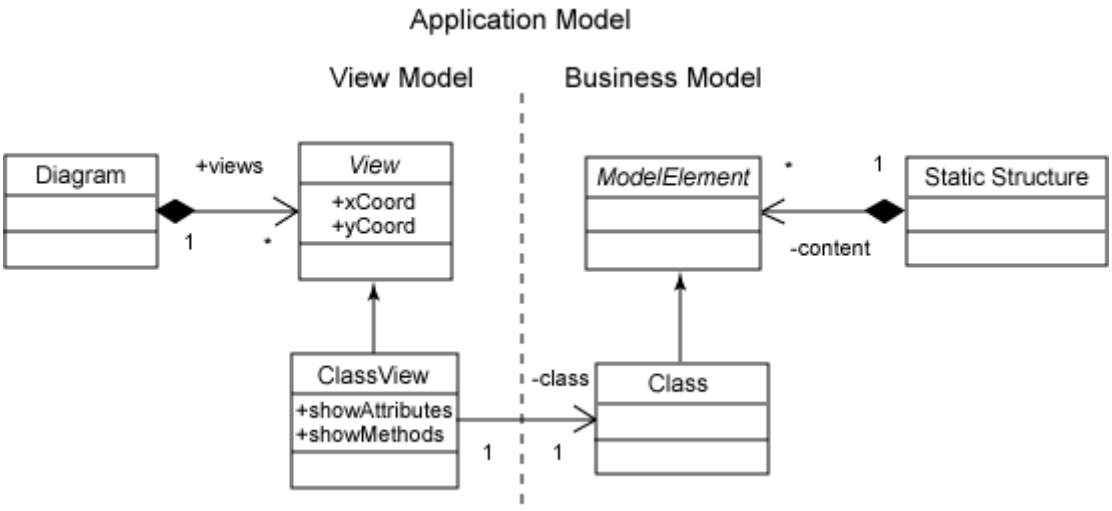
Modell-View-Kontroller konseptet (heretter kalt MVC) er en arkitektur som brukes i grafiske editorer[GEF Wiki]. Den ble for første gang tatt i bruk med systemet SmallTalk rundt 1980 etter at Trygve Reenskaug introduserte konseptet. [Model 2001]

MVC deler applikasjonen i tre deler:

- Modell – Applikasjonens data og funksjonalitet
- View – Presentasjonen av data til brukeren (GUI)
- Kontroller – Kontrollerer strøm av input fra bruker (tastatur, mus etc.)

Figur 17 over viser forholdet Modell – View - Kontroller og reglene for kommunikasjon. Modellen kan gi begrenset informasjon til viewet, f.eks. ”Nå har en forandring skjedd i modellen. Oppdater view” uten å si detaljert hva som er forandret.
Viewet må da se på modellen og oppdatere seg deretter. Viewet ber her om å få beskjed fra kontrolleren hvis en bestemt hendelse skjer. Denne prosessen kalles Notify/Subscribe mekanismen.
Linjen View - Model viser til at viewet kan kalle direkte på modellens funksjoner. Linjen View – Controller kommer av at viewet kan kalle på et begrenset utvalg av funksjoner i kontrolleren.

Denne ansvarsfordelingen gjør at modellen (data og logikk) ikke trenger å ta hensyn til brukergrensesnitt og viewet trenger ikke ta hensyn til logikk eller prosessering av input. Vi kan si mer direkte at modellen ikke skal vite noe om viewet, noe som også innebærer at den ikke inneholder noen referanser til view.



Figur 18: Beskrivelse av skille modell – view [ibm]

Fordeler med MVC [Moock 2004]:

- Én modell kan representeres til brukeren på flere måter (flere views)

- Views kan lages, forandres eller fjernes uten å påvirke dataene (modellen)
- Håndtering av brukerinput kan enkelt forandres
- Et view kan brukes for flere modeller
- Man kan arbeide uavhengig på de tre områdene som er fordel hvis man er flere utviklere med forskjellige arbeidsområder
- Hjelper utvikleren med å fokusere på et område av applikasjonen av gangen

Et verktøy som utnytter MVC konseptet er en UML editor utviklet av [Inteks]. I en systemutviklingsprosess skal gjerne brukeren inn og se på UML diagrammer for å godkjenne retning og gjerne komme med innspill underveis i utviklingen. Dette krever at brukeren forstår hva diagrammene viser, noe som kan kreve endel forkunnskaper. For å forenkle kompleksiteten av diagrammene ønsket Inteks å lage to views utifra samme underliggende modell i deres UML editor. En rettet mot utviklerne og en rettet mot brukerne.

MVC konseptet brukes ikke bare som hovedarkitektur i et system, men kan også eksistere på mange nivåer og mange dimensjoner i en applikasjon.

3.2. Kontrolleren

All brukerinput skal gå gjennom kontrolleren. En forandring kan skje i view ved en brukerinteraksjon (en bruker trykker på en knapp). Kontrolleren vil da kalle på metoder som vil sørge for en eventuell forandring i modellen. I en aktiv modell vil forandringen i modellen reflekteres i view, mens i en passiv kan kontrolleren bestemme når view skal oppdateres.

Generelt har vi en kontroller for en tenkt handling, eksempelvis et use-case. En handling i vår applikasjon er, for eksempel, å gruppere et view. Kontrolleren vil da håndtere alt som med å trigge de forskjellige handlingene som må skje for at grupperingen skal gjennomføres.

3.3. Metode i verktøyet

For å benytte seg av skille mellom modell og view må vi definere hvilke data som tilhører hva. I verktøyet tenker vi oss denne løsningen:

Modelldata :

- Klassene
- De identifiserende attributtene
- Binære assosiasjoner med deres attributter (multiplisitet, roller, identifying)

Viewdata:

- Visning av klasse – skal en klasse være synlig eller skjult i viewet?
- Gruppert/ugruppert klasse – skal en klasse vises som gruppert eller ugruppert?
- Visning av assosiasjonsklasser – hvilken måte skal vi se på assosiasjonsklasser?
- Fremmednøkler

Generelt kan vi si: Hvis et element har en egenskap som kan være ulik i forskjellige view vil dette tilhøre viewdata. Og i motsatt fall er det data som felles for alle viewene vil dette tilhøre modelldata.

3.3.1. Teknologi

Eclipse

Eclipse er et open-source, plattformuavhengig rammeverk skrevet i Java for å utvikle ”rik-klient” applikasjoner. Med ”rik-klient” mener vi her klienter som kjører applikasjoner som er avhengig av Eclipse for å kjøre. Samtidig er Eclipse også et community og prosjekt i seg selv som stadig utvikles og utvides. Med sin gode støtte for plug-ins er det i dag mange Eclipse prosjekter som igjen kan bygge på hverandre (plug-ins kan bruke andre plug-ins).

GEF

GEF er en Eclipse plug-in som er tilrettelagt for utvikling av grafiske editorer. Den baserer seg på MVC konseptet og har mye funksjonalitet for å håndtere skillet mellom modell og view.

Dette open source rammeverket tilbyr et grafisk editormiljø som legger til rette mye GUI funksjonalitet, spesielt med hensyn til modellering. Det finnes mye innebygget logikk som er spesielt rettet mot utvikling av grafiske editorer.

Draw2d

Draw2d følger med GEF rammeverket. Det er et lite bibliotek med figurer som brukes i verktøyet. Alle figurene i et view er tegnet opp av Draw2d.

Knytningen av modell, view og kontroller defineres med disse objektene i GEF:

- Modell: Modellobjekter
- View: Figurer
- Controller: EditParts

Modellobjekter

Alt som finnes i modellen vår vil ha sine egne modellobjekter. Mer spesifikt vil alle klasser, assosiasjoner og deres attributter være modellobjekter.

Figurer

Hvert modellobjekt vil ha en tilhørende figur. Figurene i et view settes sammen ved hjelp av en trestruktur. Øverst i treet finner vi som oftest selve diagrammet, rettere sagt rammen for alle klasse- og assosiasjonsfigurer. Disse er da barn av diagrammet. Et barn kan igjen ha barn osv som illustrert i figuren under.

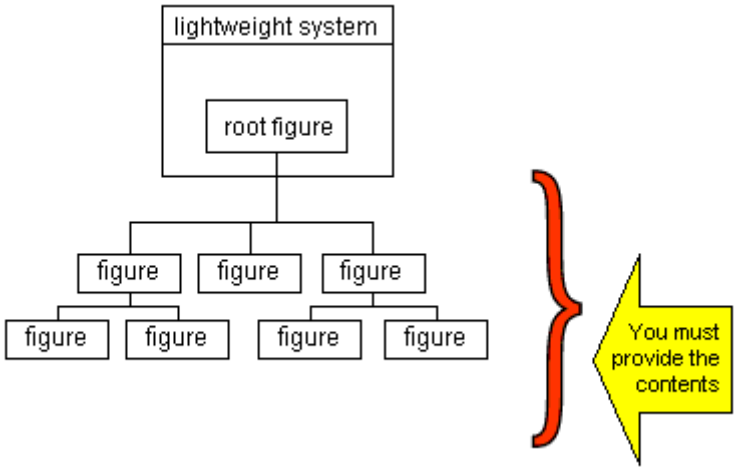
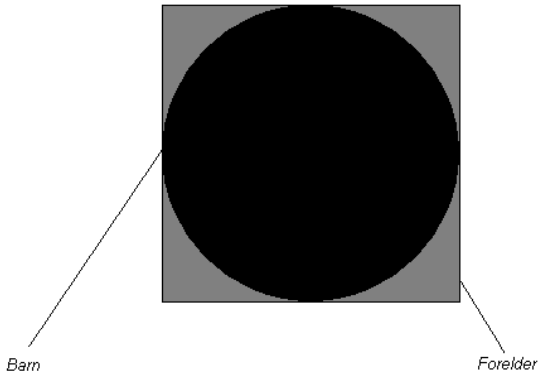


Figure 19: Draw2d sin trestruktur [Draw2d Figur]

En figur kan ha en layoutmanager som kontroller størrelsen som f.eks. maksimumsbredde, foretrukket høyde etc. Layoutmanageren kan også gjøre slik at et barn holder seg innenfor sin foreldres grense. Eksempelvis kan vi ha et rektangel som forelder og en sirkel som barn. Da kan vi sette layoutmanageren til å sette størrelsen til sirkelen slik at den akkurat passer inn i rektangelet.



Figur 20: Forelder og barn med layout manager

Editparts

Hvert modellobjekt som skal vises i view vil ha en editpart. Den inneholder logikk som gjør at modell og view blir opprettholdt ved endringer. En editpart har kobling til både modellobjektet og figuren som skal være representasjonen av den i viewet. Den lytter til

modellens ”state”, og ved endringer av sitt modellobjekt vet den hvordan den skal oppdatere view i henhold til disse endringene.

Command

Alle endringer som foretas på modellen foregår gjennom en ”Command”. En slik kommando er oppdelt i flere steg. Først undersøkes det om kommandoen skal få lov til å utføres, eksempelvis om et Class element som skal grupperes er et begrep (ugruppert). Så innkapsles informasjon om handlingen. Ved å gjøre dette, vil vi ha muligheten til å lagre tilstanden til diagrammet før handlingen eksekveres. Etter eksekveringen sendes det ut en event om hva som er forandret og de involverte EditParts sørger for at view oppdaterer seg slik det skal. Kommandoen har også tatt vare på en kopi av tilstanden til objektet som forandres slik som det var før eksekvering. Dette gjør at brukeren har mulighet til å angre (Undo) handlingen. Modellen går da tilbake til tilstanden den hadde før eksekvering.

3.3.2. Views

Vi har to ulike typer abstraksjoner over modellenen vår:

- Binære mange-til-mange assosiasjoner og assosiasjoner av høyere orden, se kap 2.4.1
- Fremmednøkler, se kap 2.4.2

Utenom disse må også all informasjon om figurene, som størrelse og plassering, lagres i viewet.

Navngivning av views

En bruker kan arbeide med flere kanoniske modeller med flere views. Derfor er det viktig at brukeren kan få oversikt over hvilket view på hvilken modell han til enhver tid arbeider med. Her vil jeg bruke navngivning av modell og view til å identifisere hver enkelt view.
Eksempel: ”View_1 – Modell_1”, ”View_2 – Modell_1” osv.
Modell refererer her til navnet på modell på kanonisk form.
Dette identifiserende navnet vil stå på de forskjellige view arkfanene.
Det må derfor være mulig å endre navn både på modell og view i verktøyet.

3.3.3. Sub-modeller som views

"Identifying the appropriate presentation for the application will greatly facilitate the subsequent windows being developed since they will have a common framework to reside in. On the other hand, if you do not define the presentation model early in the design of your GUI, late changes to the look and feel of the application will be much more costly and time-consuming because nearly every window may be affected." James Hobart [Hobart 1995]

I GEF defineres objektklasssene som modellen og figurklassene som viewet. En måte å benytte seg av dette på er å la modell på kanonisk form lagres i objektklassene og resten i figurklassene. Dette gir noen utfordringer i forhold til gruppering av modellen vi ser i viewet. Hvis vi velger å følge GEF helt og holdent vil grupperingsalgoritmen arbeide med både modell og view, noe jeg ikke ser som en god løsning.

Derimot ser jeg en annen løsning som ser på modell – view skillet i fra en litt annen vinkel. Modellen på kanonisk form er basismodell. Når en bruker ønsker å opprette et nytt view fra modellen på kanonisk form vil det dannes en kopi av objektklassene (klasser og assosiasjoner) i modellen på kanonisk form internt i verktøyet. Vi kan se på sub-modellen som et view. Da kan vi si at vi har disse tre lagene: Modellen (modellen på kanonisk form), viewet (sub-modellen) og presentasjonen som er det brukeren ser.
Vi arbeider teknisk i løsningen på flere modeller, men dette er kun en praktisk måte å gjøre det på. Dette vil ikke ha noen innvirkning på hva bruker oppfatter. Utifra brukerens synsvinkel arbeider han/hun fortsatt på views direkte mot modellen på kanonisk form.

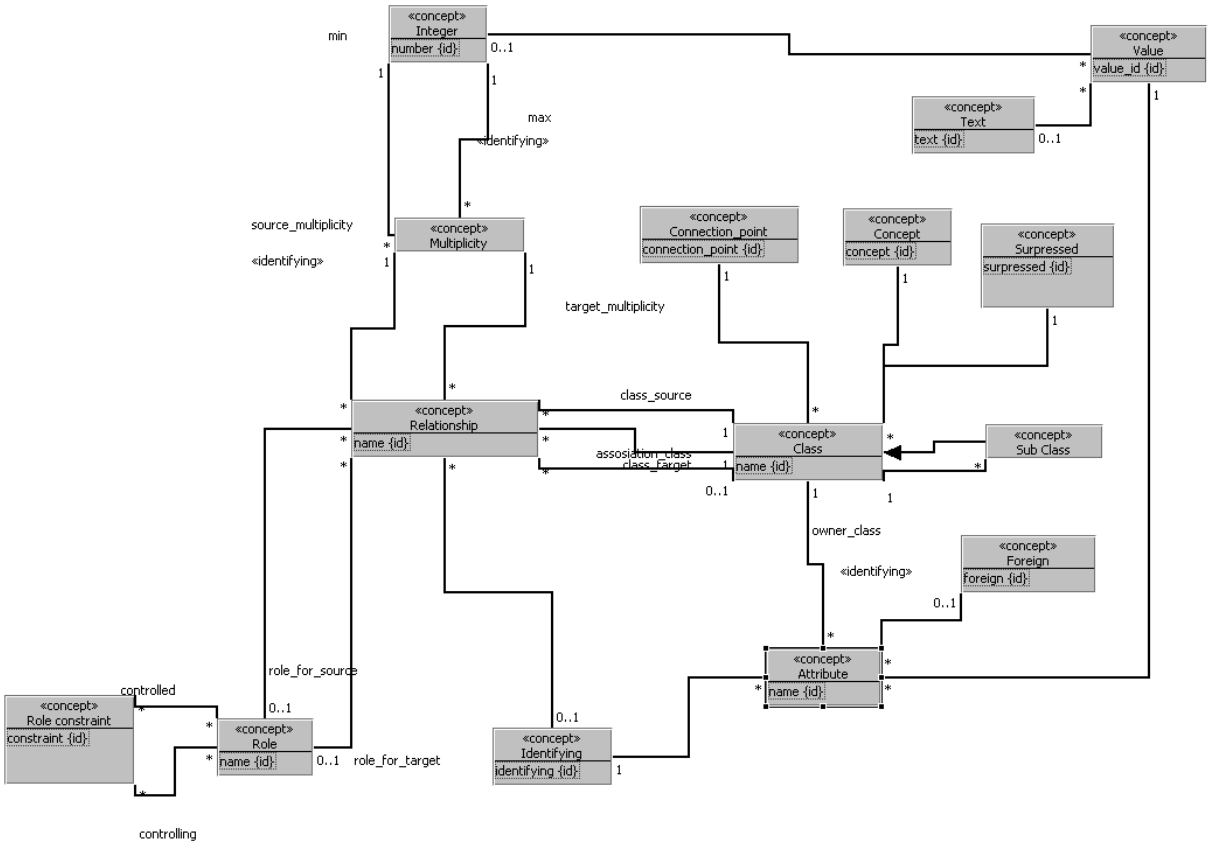
"Whether your model is split into two parts, or even multiple resources, does not matter to GEF. The term model is used to refer to the whole application model. An object on screen may correspond to multiple objects in the model. GEF is designed to allow the developer to handle such mappings easily." - Randy Hudson [Hudson]

Hver gang bruker gjør en operasjon som krever oppdatering i modellen vil ikke bare submodellen oppdateres, men også modellen på kanonisk form. Dette gjør at vi fortsatt oppfyller kravene om en modell og flere views.

En stor fordel med å gjøre det på denne måten, er at jeg kan utnytte den logikken som allerede ligger i dagens verktøy, bl.a. ligger grupperingsalgoritmen til rette for å arbeide på modellen (i dette tilfellet sub-modellen).

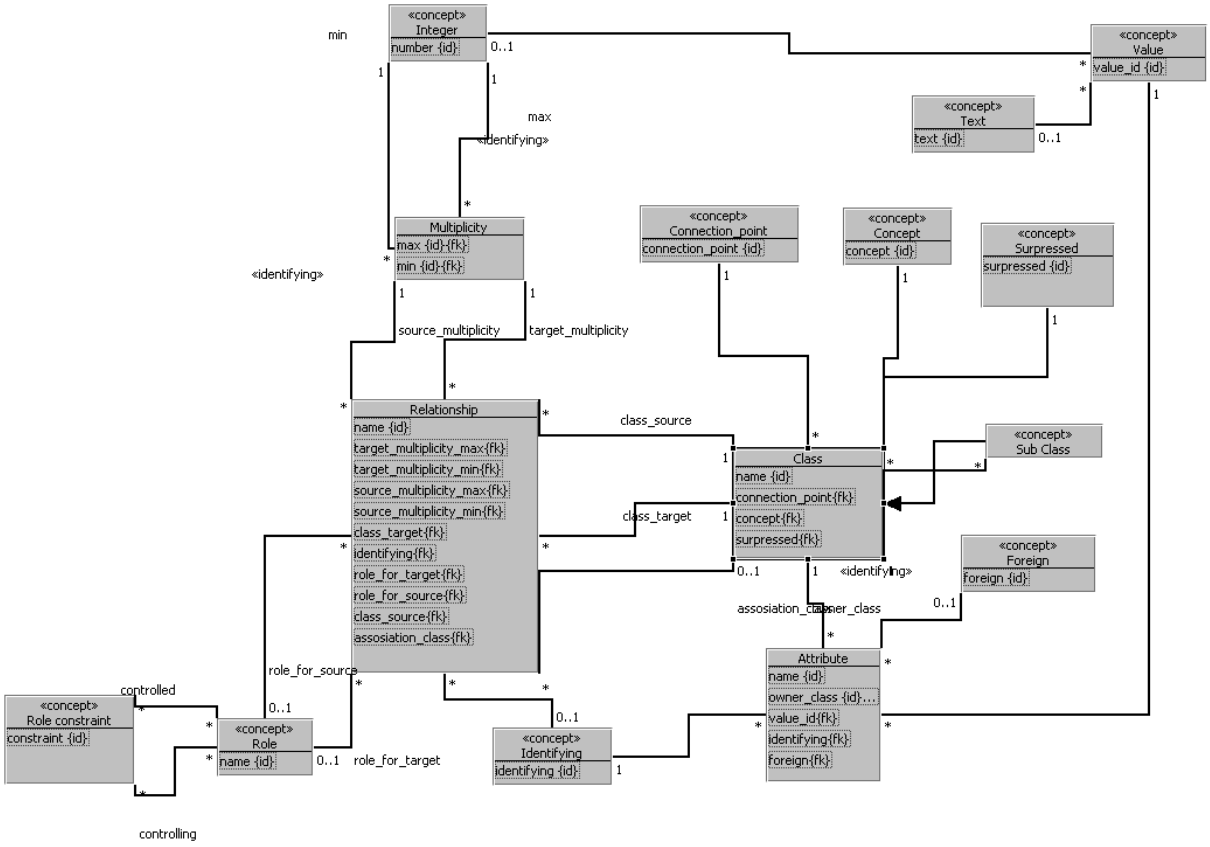
3.4. Datamodell i verktøyet

3.4.1. Objektstruktur



Figur 21: Metamodellen for verktøyet (ugruppert)

Figur 21 viser datamodellen som skal benyttes i verktøyet. Ved å gruppere den ser vi hvilke objektklasser vi trenger i verktøyet for å arbeide med modeller.



Figur 22: Metamodellen gruppert

Den grupperte modellen gir oss strukturen på sub-modellene vi arbeider med i verktøyet. Hver av de grupperte klassene tilsvarer en objektklasse. De ugrupperte klassene undertrykkes og fremmednøklene de har generert gjøres om til vanlige attributter. Navnene på objektklassene og variablene kan variere noe fra metamodellen til applikasjonen, men er laget slik her for å beskrive modellen.

Class

Class er hovedelementet i modellen vår. Den har en intern id som er unik i hver av sub-modellene, inkludert modellen på kanonisk form. Grunnen til dette er at vi da kan ha samme navn på klassene i forskjellige modeller uten at det vil skape problemer. Dette gjør det også

mulig at et begrep kan ha forskjellig navn i forskjellige views, men det skal legges logikk i applikasjonen slik at dette ikke skal gå an. Alle klassene er bundet til modell på kanonisk form. Denne interne id'en er ikke modellert i metamodellen fordi det holder metamodellen mer oversiktlig. Skulle vi hatt intern id som eneste id attributt måtte vi hatt med et begrep Navn som alle med attributt navn referer til. Class inneholder også de boolske attributtene connection_point og concept. Connection_point variabelen forteller om en klasse opptrer som et koblingspunkt eller en klasse. Variabelen concept forteller oss om Class er gruppert eller ikke. I tillegg har Class en liste som holder orden på alle attributter den har, alle assosiasjoner den har og eventuell superklasse og sub-klasser den måtte ha. Disse listene er med for å lettere kunne arbeide med sub-modellene.

Relationship

Relationship inneholder informasjon om forbindelsene i sub-modellene. Den har som Class en intern unik id og muligheten for å ha samme navn i forskjellige sub-modeller. Det brukes tre typer assosiasjoner i verktøyet. Vanlige assosiasjoner, sub-type generaliseringer og assosiasjons-klasse forbindelser.

Forbindelsen må ha en source- og en targetklasse for å eksistere og referanser til disse lagres i de respektive variablene class_source og class_target. Disse klassene kan ha roller i forbindelsen som lagres i role_for_target og role_for_source.

En assosiasjon kan være identifying. Den boolske variablen identifying lagrer informasjon om dette. Vanlige assosiasjoner har også multiplisiteter som holdes orden på av to referanser til

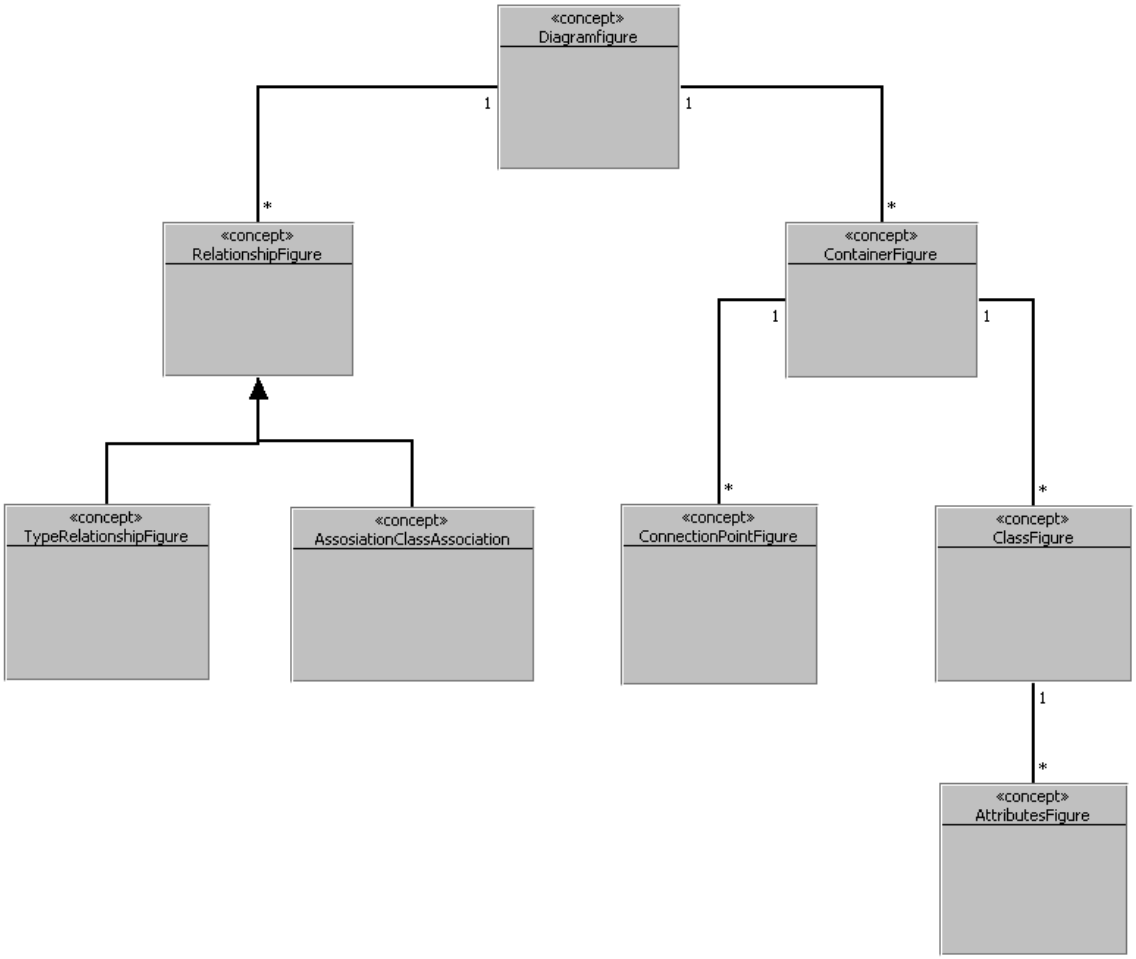
Attribute

Et attributt har også en intern id og et navn. Den har også en referanse til sin "eier"-klasse og en referanse til Value som inneholder informasjon om datatypen til attributtet. Den har to boolske variabler som forteller om attributtet er identifying og/eller fremmednøkkel.

Multiplicity

Multiplicity inneholder informasjon om maksimum og minimumsmultiplisiteten til en assosiasjon. For å unngå å blande heltall og tekst, *, bruker vi den høyeste integerverdien vi kan lagre som maksimumsmultiplisitet.

3.4.2. Presentasjonsmodellen



Figur 23: Presentasjonsmodellen med alle figurklassene

I figur 23 ser vi hierarkiet som presentasjonen mot brukeren er bygget opp av. Figurene har ingen intern id. EditPart'en (kontrolleren) holder styr på hvilke figurer som tilhører hvilke modellobjekter. På toppen av treet har vi roten – DiagramFiguren. Under den kommer Containerfiguren. Denne figuren er ikke synlig i seg selv, men inneholder synlige barn. Basert på om en klasse skal vises som et koblingspunkt eller en klasse settes barnefiguren øverst i Container figuren. En Class figur har også sitt eget barn – Attributes figuren. Denne figuren viser alle attributtene til klassen, både identifiserende, fremmednøkler og vanlige attributter.

På venstre side av modellen ser vi Relationship figuren. Denne figuren brukes til vanlige assosiasjoner og inneholder alle ord for roller, multiplisiteter, identifying, navn og ankerpunkter for å koble sammen klassene og assosiasjonene. TypeRelationship og AssociationClassConnection figuren er sub-typer av RelationshipFiguren. Disse arver alle egenskapene til forelderen, men ser forskjellige ut.

3.5. Filformat i verktøyet

XML-formatet[W3C XML] benyttes til lagring av verktøyets data til fil. XML er et veletablert format for både lagring og utveksling av data. Når vi arbeider på så enkle modeller som vi gjør med dette verktøyet, er det også oversiktlig. Med tanke på import og eksport vil det være enklere å konvertere filene til andre formater. Det har også vært en tankegang å kunne arbeide direkte på filene uten å gå gjennom verktøyet f.eks. ved å gruppere med XSLT fra kommandolinje.
Jeg vil i dette kapitlet vise en XML-eksempelfil. Så vil jeg vise deler av XSD skjemaet som er definisjonen til XML filen. Jeg vil ta for meg de viktigste elementene her.

```
<?xml version="1.0" encoding="UTF-8" ?>
<classgun xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="../../../../../classgun/xml/nyxml/model1.xsd">
 <models>
  <model name="Canonic model" id="diagram_1" default=”yes”>
   <class id="class_5" concept="yes" connection_point="no" surpressed="no">
    <name>Class A</name>
    <representation>
     <value-ref ref="value_5" name="a" fk="no"/>
    </representation>
    <attributes/>
   </class>
   <class id="class_6" concept="no" connection_point="no" surpressed="no">
    <name>Class B</name>
    <representation>
     <value-ref ref="value_6" name="b" fk="no"/>
    </representation>
    <attributes>
     <value-ref ref="value_9" name="role_a" fk="yes"/>
    </attributes>
   </class>
   <value id="value_5">
    <name>a</name>
    <datatype>1</datatype>
   </value>
   <value id="value_6">
    <name>b</name>
    <datatype>1</datatype>
   </value>
   <value id="value_9">
    <name>role_a</name>
    <datatype>1</datatype>
   </value>
   <relationship id="relationship_3">
    <source identifying="no">
     <class-ref ref="class_5"/>
     <role>role_a</role>
     <multiplicity min="1" max="1"/>
    </source>
    <target identifying="no">
     <class-ref ref="class_6"/>
     <role>role_b</role>
     <multiplicity min="0" max="*"/>
    </target>
   </relationship>
  </model>
 </models>
 <views>
    ...
 </views>
</classgun>
```

Figur 24: Modelldelen

```
<views>
  <view default="yes" name="Standard view">
    <diagram title="Simple model">
      <class-view ref="class_5">
        <location x="10" y="100" />
        <dimension width="100" height="100" />
      </class-view>
      <class-view ref="class_6">
        <location x="400" y="100" />
        <dimension width="100" height="55" />
      </class-view>
      <relationship-view ref="relationship_3">
        <target-anchor-gravity>2</target-anchor-gravity>
        <source-anchor-gravity>6</source-anchor-gravity>
      </relationship-view>
    </diagram>
  </view>
</views>
```

Figur 25: Viewdelen

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <!-- This is the XML Schema for Classgun model XML documents.
       The standard extension for Classgun XML files is '.cgxml'.-->

  <!-- Root element and children. -->
  <xs:element name="classgun">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="models" maxOccurs="1" minOccurs="1"/>
        <xs:element ref="views" maxOccurs="1" minOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <!-- Root element for all models -->
  <xs:element name="models">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="model" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
       <xs:attribute name="default" use="required" type="yesNoType"/>
    </xs:complexType>
  </xs:element>

<!—Root element for all views -->
  <xs:element name="views">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="view" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

Figur 26: Hovedelementene

Her ser vi hovedelementene i XML-filen. Først har vi definisjonen på XML-filen.

<Classgun>

Dette elementet er toppnivået i XML-filen og omkranser alle dataene.

<Models>

Dette elementet gjør at vi kan ha flere modeller (sub-modeller) lagret i en fil.

<Views>

Dette elementet gjør at vi kan ha flere views i filen vår.

```
<!-- Root element for model data. -->
<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="default" use="required" type="yesNoType"/>
    <xs:attribute name="id" use="required" type="xs:ID"/>
    <xs:attribute name="name" use="optional"/>
  </xs:complexType>
</xs:element>
```

Figur 27: Rotelementet for modell data

<Model>

Model elementet inneholder class, value og relationshipelementer. Vi ser at en model kan være satt til å være default som definerer modellen på kanonisk form. I tillegg ser vi at modellen må ha en id og kan ha et navn.

```
<!-- Root element for a class -->
  <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="connection_point" default="no" type="yesNoType"/>
      <xs:attribute name="surpressed" default="no" type="yesNoType"/>
      <xs:attribute name="id" use="required" type="xs:ID"/>
    </xs:complexType>
  </xs:element>
```

Figur 28: Rotelementet for en klasse

<Class>

Model-elementet inneholder name, supertype-referanse, representasjon og attributter. Name er en referanse til datatypen String. Vi ser at en klasse kan være ”concept”**,** ”connection_point” og har en påkrevet id.

```
<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:attribute name="name" use="optional"/>
  </xs:complexType>
</xs:element>
```

Figur 29: Rotelementet for en forbindelse

<Relationship>

Relationship består av en source og en target klasse. Den har også en påkrevet id og mulighet for et navn.

```
<xs:element name="view">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="diagram"/>
    </xs:sequence>
    <xs:attribute name="default" use="required" type="yesNoType"/>
    <xs:attribute name="name" use="required"/>
  </xs:complexType>
</xs:element>
```

Figur 30: Rotelementet for et view

<View>

View inneholder et diagram. Viewet kan settes til default og man kan tenke seg at dette er det viewet som åpnes automatisk ved åpning av fil. Det kan også ha et navn.

```
<xs:element name="diagram">
   <xs:complexType>
     <xs:sequence>
       <xs:element ref="class-view"minOccurs="0" maxOccurs="unbounded" />
       <xs:element ref="relationship-view" maxOccurs="unbounded"
minOccurs="0"/>
       <xs:element ref="type-relationship-view" maxOccurs="unbounded"
minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="title" use="optional"/>
   </xs:complexType>
  </xs:element>
```

Figur 31: Rotelementet for et diagram

<Diagram>

Diagrammet inneholder class-view, relationship-view og type-relationship-view. Alle disse inneholder informasjon om hvordan de respektive elementene skal vises fram (beliggenhet og størrelse).

En eller to filer for modell og views

For å holde modell og view fra hverandre kunne man tenke seg å lagre to filer per diagram. En fil som inneholder all modelldata og en fil som inneholder alle data om hvordan modellen skal vises mot bruker.

I denne sammenhengen vil denne splittelsen gi denne arkitekturen:

Modell – Kontroller – View:

Modelfil – Verktøy - Viewfil

Dette vil kreve en referanse mellom filene slik at verktøyet kan vite hvilken viewfil som tilhører hvilken modell. For å følge MVC konseptet er det viktig at modellfilen ikke blir avhengig av viewfilen. Jeg ser for meg disse løsninger på dette:

- Bruk av navnestruktur på filnavn for å se koblingen. F.eks. Modell1.mod.cgxml og Modell1.view.cgxml
- Viewfilen innholder navnet på modellfilen
- Bruker må selv gi beskjed til verktøyet hvilke filer som skal brukes.

Ved oppretting av ny modell, åpning av modellfil der viewfil ikke finnes og en eventuell import av et annet filformat må det da opprettes en ny fil ved lagring for å kunne ta vare på data om viewene.

Bruk av to filer vil stille krav til at brukeren må håndtere to filer for å ta vare på både view og modell. Jeg tror dette vil kunne gi problemer eksempelvis ved kopiering, flytting av filer etc. Selv om vi skiller mellom modell og view i verktøyet, vil ikke brukeren tenke på skillet og få følelsen av å arbeide rett på modellen. Jeg velger derfor å beholde modell- og viewdataene i én fil. Hvis man ønsker å eksportere filen til andre editorer finnes det gode alternativer.

Eksport til andre editorer

Flere av de mer kjente UML editorene (Rational Rose, Microsoft Visio) støtter et format for utveksling av metamodeller kalt XMI (XML Metadata Interchange) [OMG XMI]. XMI tilbyr en måte å lagre en UML modell i XML.

Det er tre muligheter ved eksport med hensyn til hva vi faktisk legger i eksportfilen. Den ene er å eksportere modellen. Den andre er å eksportere et view uten informasjon om layout, og det tredje er å eksportere et view med layout. XMI har støtte for viewinformasjon i sitt skjema, men jeg vil i første omgang konsentrere meg om eksport uten layoutinformasjon.

Import fra andre editorer

Med støtte for XMI vil det også være mulig å importere modeller fra andre UML-editorer. Det vil allikevel være begrensninger på hva som er mulig å importere. Vårt verktøy håndterer kun klassediagrammer, så man vil kun plukke ut klasser, assosiasjoner og attributter fra en slik modell. I første omgang vil heller ikke viewdata fra XMI filen håndteres. I stedet vil verktøyet etter import opprette den importerte modellen og la det være opp til brukeren hva han videre vil gjøre med modellen. Dette vil kunne være å opprette et view, velge klasser fra modellen og presentere dem i et view. For å forenkle strukturering i view vil man kunne benytte seg av layoutmangeren (se kapittel 3.6)

3.6. Automatisert generering av Views

Diagrammet kan bli rotete og uoversiktlige hvis klasser og assosiasjoner overlapper hverandre i viewet. Derfor kan det være nyttig med en layoutmanager som kan sortere figurene på en måte som gir en god oversikt uten overlapping der det er mulig. Spesielt ved import av en modell kan dette være nyttig. Også etter kjøring av funksjoner som kan gjøre forandringer på modell eller view uten å ta hensyn til hvordan viewet blir seende ut kan dette være et behov. Eksempel på dette er endring av et assosiasjonklasseview. Når det foretas en begrepsdannelse på en mange-til-mange assosiasjon, vil det nye assosiasjonsklasseområdet være utvidet og kan kollidere med andre figurer.

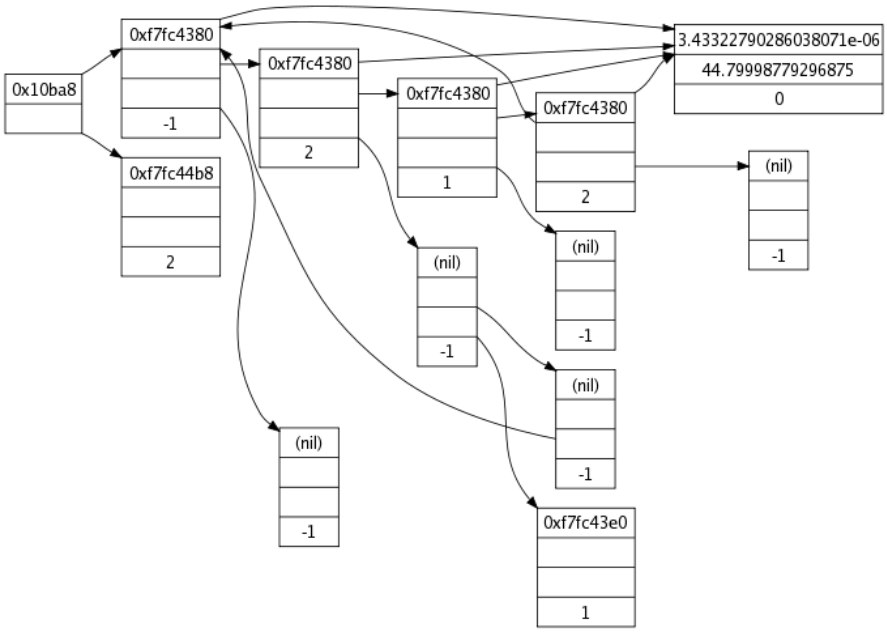
For ikke å ta vekk kontrollen fra brukeren skal en slik layoutmanager ikke arbeide konstant på viewet. Det vil si at bruker står fritt til å flytte på figur og utføre funksjoner på modell og view uten å bli hindret. Tanken er derimot å ha en knapp som bruker kan trykke på hvis han vil ha hjelp til å rydde opp i viewet. Resultatet kan brukeren bruke som utgangspunkt i den videre modelleringen.

GEF har som tidligere sagt egne layoutmanagere. Disse kan kobles direkte på figurene. Rotfiguren er i vårt tilfelle diagramfiguren. Alle elementer i diagrammet er dermed avkom fra denne. Ved å sette nødvendige skranker for hvordan barna til diagramfiguren skal opptre overfor hverandre (bl.a. ikke overlappe hverandre), vil jeg prøve å kunne få til en automatisk sortering.

Graphviz

En annen layoutmanager jeg har sett brukt sammen med GEF er Graphviz[Graphviz][Metropolis]. Denne skal være kraftigere enn GEF sine egne ”layout managere” og gi penere layouts. Dette er et frittstående opensource verktøy som kan lage layout basert på et eget modellformat. Det skal være mulig å integrere med Java, men som de selv sier på hjemmesiden kan dette være ganske komplisert [Graphviz].
En alternativ måte å løse det på er å eksportere modell- og viewdataene fra Classgun til formatet Graphviz bruker, så utføre en transformasjon på disse dataene i henhold til typen

layout jeg ønsker (circular etc) for deretter å transformere dataene tilbake til Classgun. Graphviz har kommandolinjebaserte verktøy som kan benyttes til dette.



Figur 32: Eksempel på layout generet av Graphviz [Graphviz]

Det trengs erfaring med de forskjellige layout løsningene i verktøyet for å finne den beste og peneste løsning, det vil si den som gir det mest oversiktlige viewet etter sortering.

Hvordan håndtere skjulte klasser ved automatisk generering av view

Skjulte klasser skal logisk nok ikke kunne vises i et view. Men skal de ha sin plass i viewet slik at de kan vises fram uten å måtte restrukture viewet? Hvis vi lager plasser til skjulte klasser vil vi kunne få hull i viewet ved at layout manageren tar hensyn til den skjulte figuren. Dette synes jeg ikke vil se pent ut, og man kan heller velge å kalle på layoutmanager for å restrukturere viewet etter at skjulte klasser blir synlige om man skulle ønske det.

4. Grupperingsalgoritmer

I dette kapitlet vil jeg ta for meg to grupperingsalgoritmer. Jeg ser på det som er essensielt for automatisk gruppering og ikke gå i detalj på grupperingsmetode. Alle algoritmene som starter med samme ugrupperte modell vil føre til den samme grupperte modellen. For generell grupperingsteori henviser jeg til kap 2.3. Her skal vi finne en god måte å komme til denne grupperte modellen med en automatisert algoritme. Etter å ha gått gjennom de to algoritmene vil jeg gå gjennom valgt løsning i verktøyet.

4.1. NIAM Suitens algoritme

Ragnar Normann og Jan Erik Ressem har skrevet et kompendium til informatikkkurset IN113 (som det het i sin tid) der de beskriver NIAM som modellspråk og gruppering av NIAM[6] diagrammer. Metoden de beskriver er brukt i modelleringsverktøyet NIAM Siuten. NIAM er både en metode og en teknikk for å finne og beskrive strukturen på informasjon i UoD (interesseområdet) [Normann og Ressem 1994]. I kompendiet beskrives hvordan en NIAM-modell ser ut og hvordan vi går fra NIAM modellen til en relasjonsdatabase. NIAM-metodene til Normann kan oversettes til å gjelde generelt for gruppering på UML-klassediagrammer i ORM/NIAM dialekten. Til hjelp med å forstå sammenhengen mellom UML og NIAM har jeg brukt ”A comparison of UML and ORM for data modelling” [Halpin & Bloesch 1998]. Jeg vil ta for meg det som er spesifikt for gruppering ved bruk av denne algoritmen og ikke gå inn på detaljer om NIAM-språket.

Det stilles tre krav til modellen for å at den skal kunne realiseres som en relasjonsdatabase:

1. Diagrammet må ikke ha ternære assosiasjoner der vi ikke har foretatt en begrepsdannelse.
2. Alle begreper må være refererbare, det vil si at de må ha en identifikator.

[6] Nijssen’s Informasjonrmation Analysis Methodology

3. Diagrammet må ikke inneholde noen synonyme begreper. Et eksempel på et synonymt begrep er kjælenavn på amerikanske presidenter. Noen har hatt to, men ingen har hatt samme kjælenavn. I vår sammenheng vil det si at det ikke skal være mulig å identifisere et og samme begrep med forskjellige representasjoner.

Framgangsmåte ved gruppering

Trinn 1: Finn (velg) primærnøkler. Først lages det en liste over alle begreper og en liste som inneholder navnet på de relasjonsskjema vi har som utgangspunkt. Det er nøyaktig ett relasjonsskjema (tabell i databasen) for hvert begrep i diagrammet. Så lages en liste over alle relasjonsskjemaene med tilhørende primærnøkler.
Trinn 2: Gruppering av binære en-til-mange og en-til-en assosiasjoner.
Trinn 3: Undertrykking. Noen relasjonsskjema vil ikke få noen nye attributter i løpet av grupperingen. Relasjonsskjemaet vil da bare ha det identifiserende attributtet. Hvis et annet skjema har en fremmednøkkel fra denne klassen vil den kunne undertrykkes og fremmednøkkelen i den andre klassen gjøres om til et vanlig attributt.
Trinn 4: Realisering av skranker
Trinn 5: Substitusjon. Alle fremmednøkkelattributter i klassene erstattes med hver sin kopi av det tilsvarende identiserende attributtet. Her kan det være nødvendig å justere noen av attributtnavnene for å sikre at alle attributter har forskjellige navn.

Navngivning:

Den tradisjonelle måten å navngi attributter (fremmednøkler) i NIAM miljøet er å bruke navnet på begrepet og rollen på motsatt side av assosiasjonen. Dette er ikke en del av grupperingsalgoritmen, så det er opp til hver enkelt å lage passende attributtnavn.

Undertrykking:

Et referanseskjema kan fjernes fra databaseskjemaet hvis, og bare hvis, det finnes et annet relasjonsskjema som har en fremmednøkkel som referer det.

4.2. Modellators algoritme

Bostrøm har i sitt arbeid med å utvikle modelleditoren Modellator skrevet om overgang fra datamodeller til relasjonsdatabaser og problemstilling rundt dette. Han beskriver han også metoder for automatisk gruppering og hvordan dette er implementert i Modellator [Bostrøm 1993].

Han snakker her om at en fremmednøkkel eies av en primærnøkkel, noe som gjenspeiles i hans bruk av en såkalt placeholder (se nedenfor)

Framgangsmåte ved gruppering
I Modellator genereres det løpende fremmednøkler. Det vil si at for hver gang det skjer en forandring et sted i modellen vil fremmednøklene som blir påvirket oppdateres.

Navngivning:
Det vil generes unike navn på fremmednøklene automatisk, men det vil være mulig å endre disse navnene før en overgang til en relasjonsdatabase.

Problemstillinger ved automatisk generering av fremmednøkler
Genering av fremmednøkler der identifikatorer ikke er definert enda
Her foreslår Bostrøm å bruke en placeholder. Det vil si at det genereres en ”foreløpig fremmednøkkel” som fungerer som en ”plassholder” for den virkelige fremmednøkkelen som skal komme når identifikator på ”eiersiden” er blitt etablert. Eksempel på navngivning av en foreløpig fremmednøkkel er <Kunde.id>.

Entitetisering ved multiplisitene 0..1:0..1
Ved forekomster av slike multiplisiteter skal det være mulig å automatisk entitetisere dem, dvs. å gjøre dem om til ”uproblematiske” multiplisiteter ved begrepsdannelse.

4.3. Drøfting

Begge algoritmene tar for seg problemstillinger ved overgang fra modell til relasjonsdatabase. Niam Suitens algoritme går gjennom hele modellen ved gruppering. Modellator sin algoritme

lar oss derimot gruppere én og én klasse uten å måtte gå gjennom hele modellen først. Siden et av kravene til gruppering i verktøyet er at vi skal kunne gruppere både delvis og helt, velger jeg å benytte Bostrøms algoritme som basis for hvordan automatisk gruppering vil bli løst i verktøyet. Dessuten ser jeg på hans måte å benytte seg av placeholdere som en god løsning på problemstillinger som dukker opp i løpet av grupperingsprosessen.

4.4. Verktøyets algoritme

Verktøyet skal kunne gruppere et klassediagram både delvis og helt. Derfor må vi kunne gruppere ved å ta utgangspunkt i en bestemt klasse og uten å gå gjennom diagrammet som helhet ved gruppering. Bostrøms algoritme støtter dette og gir retningslinjer for at vi rekursivt kan bevege oss utover ”trestrukturen” for å nå toppklassen/begrepet. Selv om vi nå har muligheten til å lage et verktøy som kan gjøre mesteparten av grupperingen for oss, er det ikke nødvendigvis det vi er ute etter. Dette verktøyet skal være pedagogisk, og vi ønsker derfor ikke at verktøyet uten videre skal gi løsningene på modeller med forplantning av fremmednøkler. Vi vil heller at verktøyet skal gi en tilbakemelding om at her er det noe brukeren må se nærmere på. Her utnytter vi Bostrøms placeholder strategi, se nærmere forklaringer lenger ned. Samtidig vil vi ha god nytte av en automatisk algoritme som kan gruppere hele modellen på kanonisk med så lite brukerinput som mulig ved realisering, dvs. å få generert SQL-kode for overgang til relasjonsdatabase. Dette løser jeg ved å si at gruppering vha. ”Group” knappene (altså gruppering i view) brukes den pedagogiske varianten med placeholder. Skal man derimot generere SQL kode er algoritmen helt automatisk og rekursiv for å kunne gå gjennom hele modellen på kanonisk form.

4.4.1. Håndtering av spesielle multiplisiter

0..1:0..1

Fremmednøkkel kan generes på ene siden, valgfritt hvilken side. Bruker vil her få muligheten til å velge hvilken side det skal genereres fremmednøkkel.

1..1:1..1

Kan håndteres som 0..1:1..1., men skal helst unngås på modelleringsnivå

*1..1:1..**
Slike multiplisiteter bør unngås på modelleringsnivå. Håndteres ikke av verktøyet.

4.4.2. Navngivning av fremmednøkler

Navngivning vil variere etter hva slags assosiasjoner vi har og hva slags identifikasjon klassen eller begrepet har på motsatt side. Under viser de forskjellige situasjonene vi kan komme i og hvordan navngivning skjer i prioritert rekkefølge.

Ved ”vanlige” assosiasjoner og kun et identifiserende attributt på motsatt side:

1. Hvis ingen identifiserende attributt på motsatt side, settes placeholder. Eksempel på navngivning: <ClassA.id>
2. Rollenavn med små bokstaver
3. Hvis det ikke eksisterer rollenavn brukes klassenavn med små bokstaver

Ved ”vanlige” assosiasjoner og mer enn et identifiserende attributt på motsatt side:

1. Hvis ingen identifiserende attributt på motsatt side, settes placeholder. Eksempel på navngivning: <ClassA.id>
2. Rollenavn med små bokstaver og attributtnavn
3. Hvis det ikke eksisterer rollenavn brukes klassenavn med små bokstaver og attributtnavn

Ved identifiserende assosiasjoner:

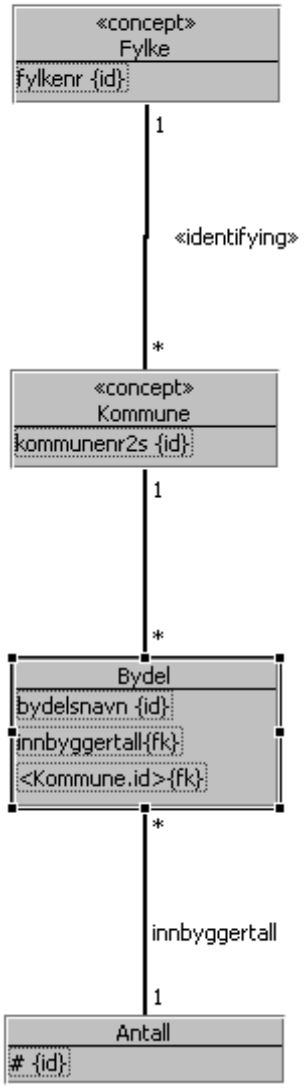
1. Hvis ingen identifiserende attributt på motsatt side, settes placeholder. Eksempel på navngivning: <ClassA.id>
2. Klassenavn med små bokstaver og identifiserende attributtnavn på motsatt side og {id}

Ved sammensatt identifikator på motsatt side:

1. Rollenavn med små bokstaver og identifiserende attributtnavn på motsatt side
2. Klassenavn med små bokstaver og identifiserende attributtnavn på motsatt side

Ved forplanting dvs. hvis mottsatt side er ugruppert og har en identifiserende assosiasjon:

1. Placeholder settes. Eksempel på navngivning: <ClassA.id>



Figur 33: Bruk av placeholder ved forplanting

Fremmednøklene blir alltid markert med **{fk}** etter attributtnavnet. Det skal være mulig å endre navn på en fremmednøkkel, så lenge ikke det nye navnet er likt noen av de andre attributtene i modellen.

4.4.3. Degruppering

Degruppering av en klasse skjer ved å fjerne alle fremmednøkler, dvs. å slette alle attributter som er definert som fremmednøkler. Klassen får igjen stereotypi «concept» og regnes som et begrep.

4.5. Generering av SQL-kode

Hvis vi grupperer modellen på kanonisk form ved hjelp av en automatisk grupperingsalgoritme ligger alt til rette for å kunne generere SQL-kode. Vi får generert alle fremmednøkler. Vi har tilgang til alle assosiasjoner med sine multiplisiteter og kan sette skranker utifra dette og vi har datatypene til de forskjellige attributtene, disse er Text og Integer.

4.6. Java vs. XSLT gruppering

Prototypen av Classgun benyttet seg av XSLT transformasjon for å gruppere. Det var flere grunner til at XSLT ble valgt som transformasjonsspråk. Stegard ønsket å benytte seg av XSLT for å eksperimentere med det språket og da er XML en nødvendighet. Ved bruk av XSLT ville det også være enkelt å bytte algoritme ved å bytte XSLT skjemaet transformasjonen kjøres med.

Under kjøring synes jeg prototypen brukte litt tid fra jeg trykket på ”Group” knappen til den var ferdig med grupperingen. Årsaken til dette er at verktøyet bruker Java objekter internt for å holde på modelldata, ikke XML. Dette gjør at modellen må gjøres om til XML-format for XSLT transformering og etterpå gjøres om til Java objekter igjen. Det er denne transformasjonen mellom intern lagring og xml som er synderen. Utifra kravene om at responstiden skal være kort (se krav til GUI kapittel 2.7) vil jeg her benytte meg av Java.

Jeg hadde som mål å finne en grupperingsalgoritme som kunne gruppere en modell både delvis og helt. Argumentet om å lett kunne skifte grupperingsalgoritme følte jeg ikke gjaldt dette verktøyet, da selve grupperingen alltid skal føre til samme resultat. Å bytte en slik algoritme vil være en sjeldenhet i dette verktøyet.

Jeg ville derfor se om transformering direkte på Javaobjektene kunne gjøre grupperingen raskere.

I verktøyet er det lagt opp til å kunne lagre både i binært format og XML. Ved å gå vekk fra XSLT blir XML kun brukt for lagring av data. Dette åpnet også for å gå vekk fra XML som lagringsformat og kun benytte seg av binær lagring. Jeg valgte å beholde XML formatet fordi dette er med på å holde filstrukturen oversiktlig. XML er i dag et veletablert format for utveksling av data og egner seg godt hvis man eventuelt. vil importere modellene inn i et annet program, noe som kan bli aktuelt i IFI UML Total. Man kan også tenke seg at arbeid kan bli gjort direkte på modellen (i filformat) uten å måtte gå gjennom Classgun.

5. Dagens verktøy

5.1. Metamodell og hoveddatastruktur

5.1.1. Metamodell

Metamodellen som er foreslått brukt i kapittel 3 er i bruk i dagens verktøy.

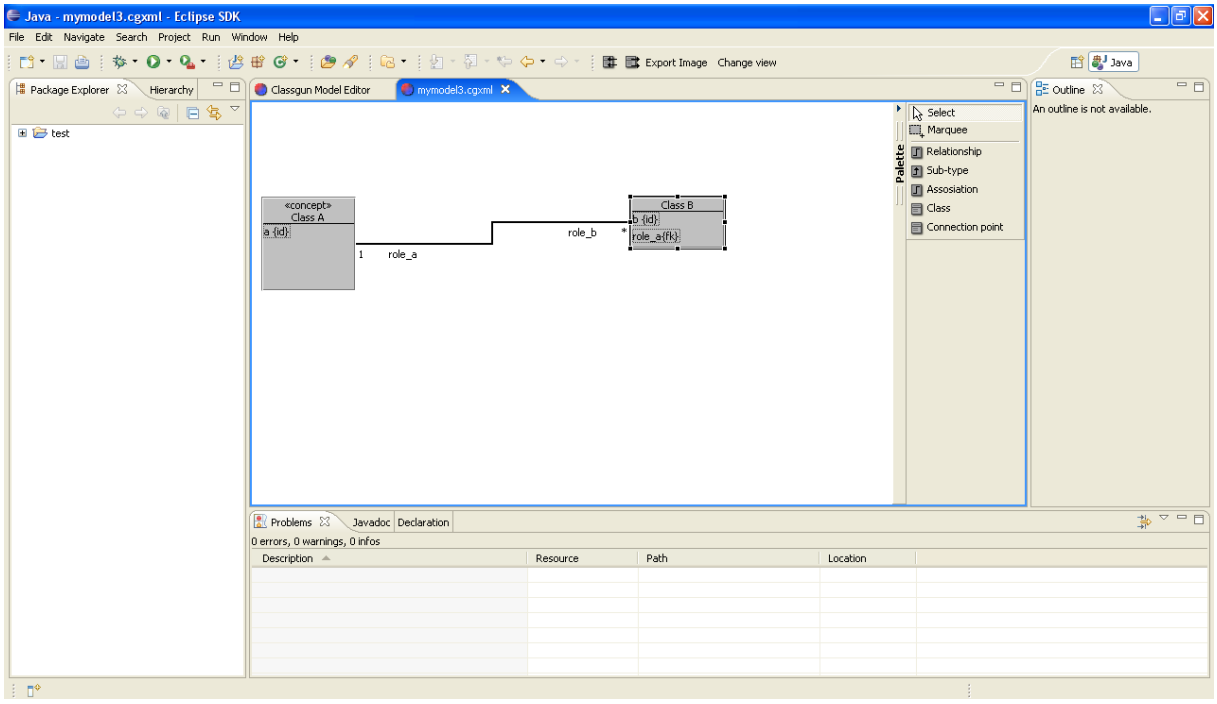
5.1.2. Filformat

Filformatet er nesten likt det som er foreslått brukt i kapittel 3.5. Eneste forskjellen er at den foreløpig ikke støtter flere modeller i en fil.

5.2. Teknologi

Prototypen bruker Eclipse og GEF som teknologi. Denne står nærmere forklart i kapittel 3.3.1

5.3. Brukergrensesnitt



Figur 34: Skjermbilde av prototype

Brukergrensesnittet vil virke kjent for de som tidligere har arbeidet med Eclipse.
Til venstre finner man Package Explorer som inneholder Prosjekter og filer (fortrinnsvis Classgun diagramfiler).
I midten er et stort vindu hvor man har views med klassediagrammene.
På høyre side har man panelet[7] der man har muligheten til å velge to forskjellige selection tools og velge UML-elementer (klasser, assosiasjoner etc., se figur 2) som man ønsker å opprette i modellen.
Over viewet er en liten toolbar med knapper for å gruppere, degruppere, eksportere view til bilde og forandre view på assosiasjonsklasse.

5.4. Kort beskrivelse av hovedfunksjonalitet

Opprette/slette klasser
Velg Class fra panelet. Klikk i viewet eller klikk og dra for å gi ønsket størrelse på klasseelementet. Klassen slettes ved å høyre-klikke på den og velge delete.
Endre attributter på klasser
Høyreklikk på klassen og velg Edit Properties for å endre attributter på klasse som navn, navn på attributt og attributtets datatype.
Opprette/slette forbindelser
Velg forbindelsestype fra panelet. Klikk på klassen du vil skal være source og deretter på klassen du vil skal være target.
Endre attributter for en forbindelse
Høyreklikk på en forbindelse og velg Edit properties for å endre attributter på forbindelsen som mulitplisiteter, roller og velge om forbindelsen skal være identifiserende eller ikke.
Opprette/slette koblingspunkt
Velg Connection point fra panelet. Klikk i viewet der du vil det skal plasseres.
Gruppere/degruppere
Velg klasser som skal grupperes eller degrupperes og klikk på henholdsvis Group- eller Ungroup-knappen fra toolbar.

[7] ”Fly-out”-panel som skyver seg ut når bruker klikker eller holder mus over

Eksport av modell til bilde
Klikk på Export Image. Et filvelger vindu vil dukke opp med mulighet for å velge navn, plassering på fila og ønsket filformat (jgp, gif, png eller bmp)

Mulighet for å skifte view på assosiasjonsklasser
For å gå fra mange-til-mange assosiasjon velger man koblingspunkt og klikker man på Change View. For å gå andre veien velges klassen i to eller flere identifiserende en-til-mange assosiasjoner og klikker på Change View.

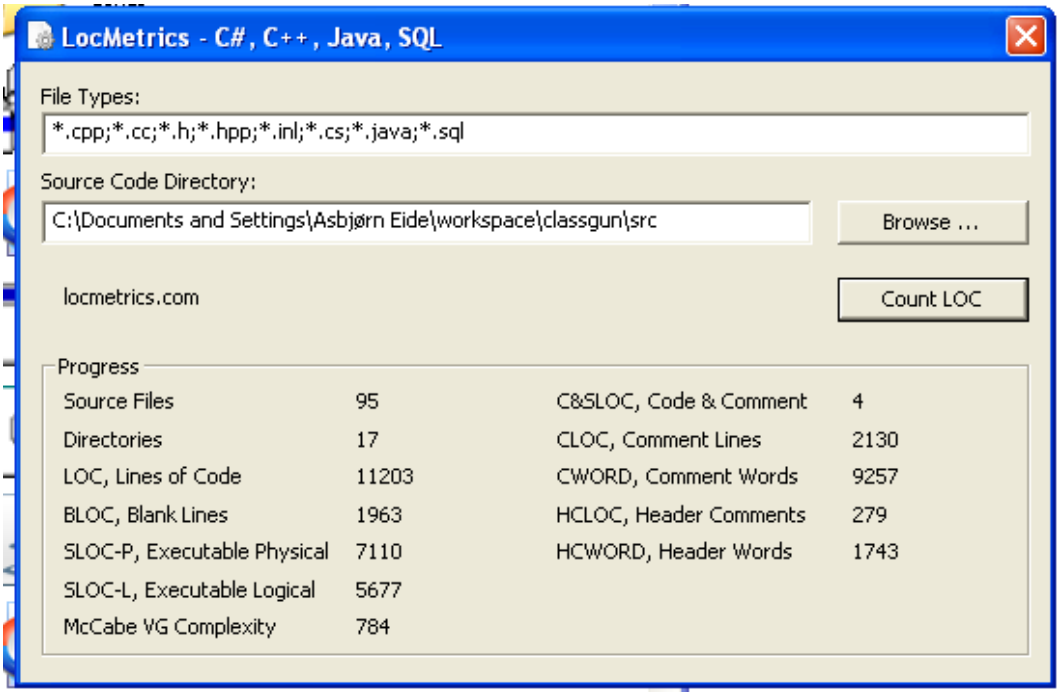
5.5. Feilhåndtering

Bruker hindres i å gjøre ulovlige operasjoner, enten ved at operasjonen ikke utføres, og i noen tilfeller også ved at ”forbudt” tegn vises.

Det gis ingen tilbakemelding til bruker om at operasjonen ikke kan utføres utenom at funksjonen enten ikke kan utføres eller ingenting skjer.

I tillegg har bruker muligheten for å angre (Undo/Redo). Dette gir brukeren bedre oversikt over sin egen bruk av verktøyet og man unngår ekstraarbeid med å rette egne feil.

5.6. Kort informasjon om koden (størrelse, lengde)



Figur 35: Skjermbilde av LocMetrics som viser antall linjer kildekode i verktøyet

I figur 25 ser vi at at kildekoden i dagens verktøy består av 95 filer i 17 kataloger og et totalt antall linjer med kode på rundt 11200 linjer.

5.7. Kort brukerdokumentasjon

5.7.1. Nødvendig programvare

- Java 1.6 JRE [Java]
- Eclipse 3.2.1 [Eclipse]
- GEF [GEF]

Dette er versjoner som verktøyet er skrevet for. Jeg kan ikke garantere for kompatibilitet med eldre eller nyere versjoner.

5.7.2. Installering

1. Installer Java JRE (installasjonsverktøy)
2. Installer Eclipse (installasjonsverktøy)
3. Installer GEF (plugin .jar filen legges i /plugins/ katalogen til Eclipse)

4. Classgun plugin legges i /plugins/ katalogen til Eclipse

5.7.3. Kjøring

1. Start Eclipse
2. Lag et prosjekt:
3. For å lage en ny modell:
    - File -> New -> Other...
    - Classgun –> New Classgun Model
    - Velg mappen modellen skal ligge i og skriv inn filnavn i ”File Name”
4. For å åpne en eksisterende fil:
    - File -> Open File...
    - Velg filen du vil åpne fra filvelger og klikk Open

5.8. Oppsummering av utvikling gjort på verktøyet bygd på prototypen

Ny grupperingsalgoritme skrevet i Java

Valgt grupperingsalgoritme (se kapittel 4.4) er implementert i dagens verktøy og oppfyller dermed kravene.

Mulighet for å lage assosiasjonsklasse-forbindelser

Det er mulig å lage assosiasjonsklasse-forbindelser i dagens verktøy. Disse følger metamodellen beskrevet i figur 21 og 22.

Mulighet for å lage koblingspunkt

Det er mulig å lage koblingspunkt for å ha høyere ordens assosiasjoner i et diagram. Dette koblingspunktet kan kobles til klasser med assosiasjoner og en assosiasjonsklasse-forbindelse.

Mulighet for å skrive ut diagram

Det er mulig å skrive ut et diagram. Mer spesifikt vil det si at man kan skrive ut det som er i viewet. For å skrive ut kan man man gå på ”File” menyen og velge ”Print”, eller man kan bruke hurtigtastene Ctrl + P.

Eksportere diagram til bilder

Viewet kan eksporteres til bildeformat. Dette gjøres ved å klikke på knappen ”Export Image”- Man får da opp et filvelger vindu der man kan velge plassering, filnavn og ønsket filtype. Det er støtte for fire filtyper: jpg, gif, png og bmp.

Forandre måten å se på assosiasjonsklasser automatisk

Det er lagt til mulighet for å endre måten å se på en assosiasjonsklasse. Dette case’et er beskrevet i kapittel 2.3.1. Dagens verktøy oppfyller kravene som står beskrevet der.

Diverse små forandringer på utseende

Det er gjort diverse små endringer på utseende i dagens verktøy i forhold til prototypen. Dette er små endringer som plassering av knapper, navngivning av felter etc.

Forandring av XML-skjema for lagring av de nye objekttypene

Siden det er gjort endringer på metamodellen i dagens verktøy siden prototypen, måtte også lagringsformatet endres for å håndtere de nye objekttypene (assosiasjonsklasse-forbindelser og koblingspunkt). I tillegg til nytt XML-skjema måtte også innlesing fra fil og skriving til fil endres i henhold til det nye XML-skjemaet.

6. Rapport - Empirisk undersøkelse

UML-editoren er tenkt brukt som et pedagogisk hjelpemiddel for bl.a. de som tar kurs INF1050. Derfor valgte jeg å la studenter som tar dette kurset være testbrukere. I samråd med en av gruppelærerne fikk jeg avsatt 30 minutter av en gruppetime for å se verktøyet i bruk og snakke med brukerne.

6.1. Mål for undersøkelsen:

- Hva synes brukerne om GUI?
- Hva synes brukerne om funksjonaliteten?
- Hvor intuitivt er programmet?
- Er dette noe de kunne tenke seg som et hjelpemiddel i undervisningen?

6.2. Forberedelser:

Jeg gikk gjennom noen forelesningsfoiler de hadde hatt om gruppering. Utifra foilene fant jeg et diagram som jeg modellerte i verktøyet. Hensikten var at de skulle kunne få se og arbeide med en modell de har sett før. Da kunne de heller konsentrere seg om verktøyet og ikke måtte sette seg inn i ny modell.

6.3. Metode for brukerundersøkelsen:

Hovedmetode ved brukerundersøkelsen var observasjon og samtale med brukere.

6.4. Selve brukerundersøkelsen:

Det var 7 studenter tilstede under gruppetimen. Disse delte seg i 3 grupper som satt ved hver sin pc med verktøyet kjørende.
To av gruppene brukte verktøyet aktivt og ga gode tilbakemeldinger.

6.5. Oppsummering av tilbakemeldinger:

- Det er få knapper noe som gjorde det enklere å sette seg inn i.
- Diagramelementene (klassene osv.) ser like ut som i lærebøkene og foilene deres.
- Alle sa at de kunne tenke seg dette verktøyet som en del av undervisningen i gruppering.

6.6. Erfaringer fra observasjonen:

Det var hovedsaklig to personer, en på hver av de aktive gruppene, som ga meg tilbakemelding. De andre satt mer og så på den som arbeidet med verktøyet, så jeg kan bare basere mine erfaringer ut fra på disse to sine tilbakemeldinger.
Etter å ha gitt dem en rask forklaring på hva de forskjellige knappene gjør, klarte de selv å gruppere, degruppere, lage nye klasser og koblinger, og forandre navn på attributter.
At verktøyets objekter følger standarden for NIAM/ORM-inspirerte profilen av UML-klassediagram viste seg å fungere bra da de lettere kjente seg igjen.

På det nåværende tidspunkt føler jeg at det ikke er nødvendig med flere brukertester siden verktøyet fortsatt er på utviklingsstadiet. Etterhvert som resterende funksjonalitet er lagt til og behovet for feiltesting, kvalitetssikring osv. øker vil dette tas opp til ny vurdering.

7. Videre utbedringer

Her beskriver jeg ønsket funksjonalitet som kan være gode å ha med i verktøyet.

7.1. Utvide skrankebehandling

Verktøyet tar for seg noen skranketyper, men det kan være fint å kunne ha funksjonalitet for å håndtere flere. Eksempelvis kan vi tenke oss å få inn mengdeskranker.

7.2. Import/eksport til Rational Rose

Rational Rose er hovedverktøyet som blir brukt i kurset INF1050. Studentene vil ofte arbeide med de samme UML diagrammene ved både modellering og gruppering. Det vil være tungvint å tegne opp UML-modellen de arbeider med i Classgun og f.eks. gruppere for så å oppdatere modellen manuelt i Rational Rose basert på hvordan den er visualisert i Classgun verktøyet. Det er ønskelig å kunne eksportere modellen til Rational Rose. Som nevnt i kapittel 3.5 vil jeg her benytte filformatet til XMI som er støttet i Rational Rose.

7.3. Integrasjon mot IFI UML Total

Det har vært to forskjellige veier å ta underveis i utviklingen av dette verktøyet. Et viktig krav til design har vært at verktøyet skal se mest mulig likt ut lærematerialet i INF1050. I integrasjonssammenheng mot IFI UML Total skulle verktøyet vært fulgt mer generelt design der. Dessuten vil integrasjon kreve at man arbeider sammen på et veldig tidlig tidspunkt. Som Det er veldig vanskelig å få en slik integrasjon til å fungere skikkelig i ettertid. Men det vil fortsatt være mulig å utføre import/eksport til IFI UML Total selv om dette ikke ikke er en like elegant løsning.

7.4. Legge ut på Open Source nettsted

For å kunne nå flest mulig i Open Source nettverket kan det lønne seg å benytte velkjente Open Source kanaler. En av disse er [SourceForge]. I skrivende øyeblikk er det registrert 146,058 prosjekter og 1,561,580 brukere på nettstedet.

Før dette kan gjøres må kildekoden struktureres og kommenteres bedre slik at andre brukere lettere kan sette seg inn i koden.

7.5. Debugging

Det finnes en feil som av og til slår inn ved lagring av modell. Utifra det jeg har sett kan feilen tyde på mangler ved sletting av visse objekter. Rekonstruksjon av feilsituasjon og debugging for å finne feilen er nødvendig, samt utbedring. Dette gjelder også oppdaging og retting av andre feil som måtte finnes.

7.6. Ferdigstille verktøyet (produksjon)

Kravene som ble stilt om funksjonalitet slik at verktøyet skal kunne brukes blant studentene er nesten møtt. Et kortsiktig mål er å ha et ferdig verktøy som er klart for bruk høstsemesteret 2007 av studenter. Viktige steg for å gjennomføre dette er testing og kvalitetssikring, gjerne sammen med framtidige brukere av verktøyet.

7.7. Endring av utseende på koblinger

Koblingene er i dag rette linjer med ankerpunkter kun på endene. Måten ankerpunkter er implementert på gjør at det er vanskelig å forandre plasseringen av disse i diagrammet. Dette bidrar til at bruker mister kontrollen og må derfor unngås. Koblingene skal forandres slik at man kan sette flere punkter på selve koblingen, dermed kan man lettere strukturere viewet etter eget ønske. Dette vil kreve forandring på i modellen (Relationship elementet) og i view delen (figur og ankerpunkter).

8. Oppsummering og konklusjon

I denne oppgaven har jeg satt opp krav til en UML editor som kan ha en modell med flere views. I tillegg til kravene om dette skillet har jeg også stilt krav til funksjonalitet både ved GUI og grupperingsalgoritme. Jeg har så gått gjennom mulige løsninger på problemstillingene og presentert den løsningen jeg velger å gå for. Mye av det jeg har kommet fram til i denne oppgaven er basert på videreutviklingen av prototypen som Stegard har laget. Andre viktige deler av problemløsningen har kommet gjennom konstruktive samtaler med min veileder.

Målet med denne oppgaven var å se på hvordan man kan skille modell og view i en UML-editor. Parallellt med dette har jeg videreutviklet en prototype av en UML-editor som skal kunne gruppere og som skal være funksjonell nok til å kunne taes i bruk av studenter til neste semester (høst 2007). Oppgaven og verktøyet har på denne måten egentlig vært litt separert. Jeg hadde et tidspress på å få gjort forandringer på prototypen og gjøre den funksjonell nok til bruk. Samtidig har jeg sett på mulighetene for å videreutvikle dette verktøyet til å følgene kravene om skille modell – view. Spesielt det siste har presentert noen utfordringer med tanke på at jeg arbeider videre på en prototype. Mye av teknologi og rammer er derfor allerede satt og jeg vurderte flere ganger om et verktøy som kan tolere modell-view kravene satt i denne oppgaven måtte bygges opp fra bunnen av igjen. Godt forarbeid lønner seg og det kan godt hende at det finnes andre gode løsninger på de problemstillingene jeg har møtt med denne oppgaven. Men jeg føler også med de undersøkelsene jeg har gjort i forhold til den nåværende implementasjonen og de tekniske rammene, at jeg har funnet løsninger som lar seg integrere i dagens verktøy.

Arbeidet på verktøyet vil ikke stoppe etter jeg har levert denne oppgaven. Dette verktøyet er det største prosjektet jeg har arbeidet med hittil hvis man ser på mengden programmering og jeg har stor egeninteresse av å se verktøyet ferdig utviklet slik at det tilfredstiller alle krav jeg har satt opp. Det vil bli en hobbyvirksomhet å fullføre verktøyet og det er inspirerende å tenke på at studenter vil arbeide med det.

9. Litteraturliste

[Rumbaugh, Jacob...] Rumbaugh, Jacobson & Booch, *The Unified Modeling Language Reference Manual*, 2004

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

[Skagestein 2007] Gerhard Skagestein, Foiler fra INF1050 21.02.07,
http://www.uio.no/studier/emner/matnat/ifi/INF1050/v07/undervisningsmateriale/skranker_repr_4.pdf [lest 23.04.07]

[Bostrøm 1993] Edgar Bostrøm, *Datamodelleringsverktøyet Modellator: en del prinsippielle og praktiske aspekter*, 1993

[Bostrøm 1993] Edgar Bostrøm, *Datamodeller og relasjonsdatabaser: kardinalitet, attributtegenskaper og fremmednøkkelstrukturer*, 1993

[Stegard 2005] Øyvind Stegard, *Grafisk editor for automatisk gruppering og degruppering av dataorienterte klassediagrammer*, 2005

[Normann, Ressem...] Ragnar Normann og Jan Erik Ressem, *Kompendium 60 for IN 113 – Del II*, 1994

[Halpin,Bloesch 19…] Dr. Terry Halpin og Dr. Anthony Bloesch, *A comparison of UML and ORM for data modeling*, 1998

[Burke 2004] Eric M. Burke, *GUI programming is hard*, 2004,
http://www.oreillynet.com/onjava/blog/2004/04/gui_programming_is_hard.html
[lest 19.03.07]

[Hobart 1995] James Hobart, *Principles of good GUI design*, 1995,
http://www.classicsys.com/css06/cfm/article_1995_10.cfm [lest 19.03.07]

[Wikipedia MVC] Wikipedia,
ModelViewControllerDiagram,http://en.wikipedia.org/wiki/Image:ModelViewControllerDiagram.svg [lest 05.02.07]

[Draw2d Figur] http://wiki.eclipse.org/index.php/Image:Gef_desc_im8.gif [lest 05.02.07]

[Wiki - Open Source] Wikipedia, *Open Source*, http://en.wikipedia.org/wiki/Open_source [lest 03.02.07]

[SourceForge] SourceForge.net, http://www.sourceforge.net

[Bradley] Angela Bradley, *SQL Data Types*, http://php.about.com/od/learnmysql/ss/mysql_4.htm [lest 24.03.07]

[Haugen 2006] Øystein Haugen, Foiler fra OMS Seminar 27. april 2006

[Java] Java 1.6 JRE, http://java.sun.com/

[Eclipse] Eclipse 3.2.1, www.eclipse.org

[GEF] GEF, www.eclipse.org/gef

[Hudson] Randy Hudson, *Create an Eclipse-based application using the Graphical Editing Framework*, http://www-128.ibm.com/developerworks/opensource/library/os-gef/ [lest 10.03.07]

[Inteks] Inteks UML Editor, http://www.inteks.ru/uml_editor.shtml [lest 25.03.07]

[Mook 2004] Colin Mook, *Essential Action Script 2.0, Chapter 18. The Model-View-Controller Design Pattern*, 2004, http://www.adobe.com/devnet/flash/articles/mv_controller/as2ess_ch18.pdf [lest 10.03.07]

[Graphviz] Graphviz, http://www.graphviz.org/

[eNode 2002] eNode, http://www.enode.com/x/markup/tutorial/mvc.html [lest 05.04.07]

[OMG XMI] OMG, *XMI*, www.omg.org

[Model 2001] Mitchell L Model, *Model View Controller History*, 2001, http://c2.com/cgi/wiki?ModelViewControllerHistory [lest 12.03.07]

[Metropolis] *Metropolis Model Structure Viewer User's Guide* http://embedded.eecs.berkeley.edu/metropolis/metropolis_ui/metro_ui.htm

[GEF Wiki] Eclipsipedia, GEF Description, http://wiki.eclipse.org/index.php/GEF_Description

[W3C XML] Extensible Markup Language (XML) http://www.w3.org/XML/

[Draw2d Figur] http://wiki.eclipse.org/index.php/Image:Gef_desc_im8.gif [lest 24.04.07]