Hovedopgave, Datanomuddannelsen ved

Niels Brock - *Copenhagen Business College*

Forårssemesteret 1998

En undersøgelse af en syntese mellem den relationelle og den objektorienterede databasemodel

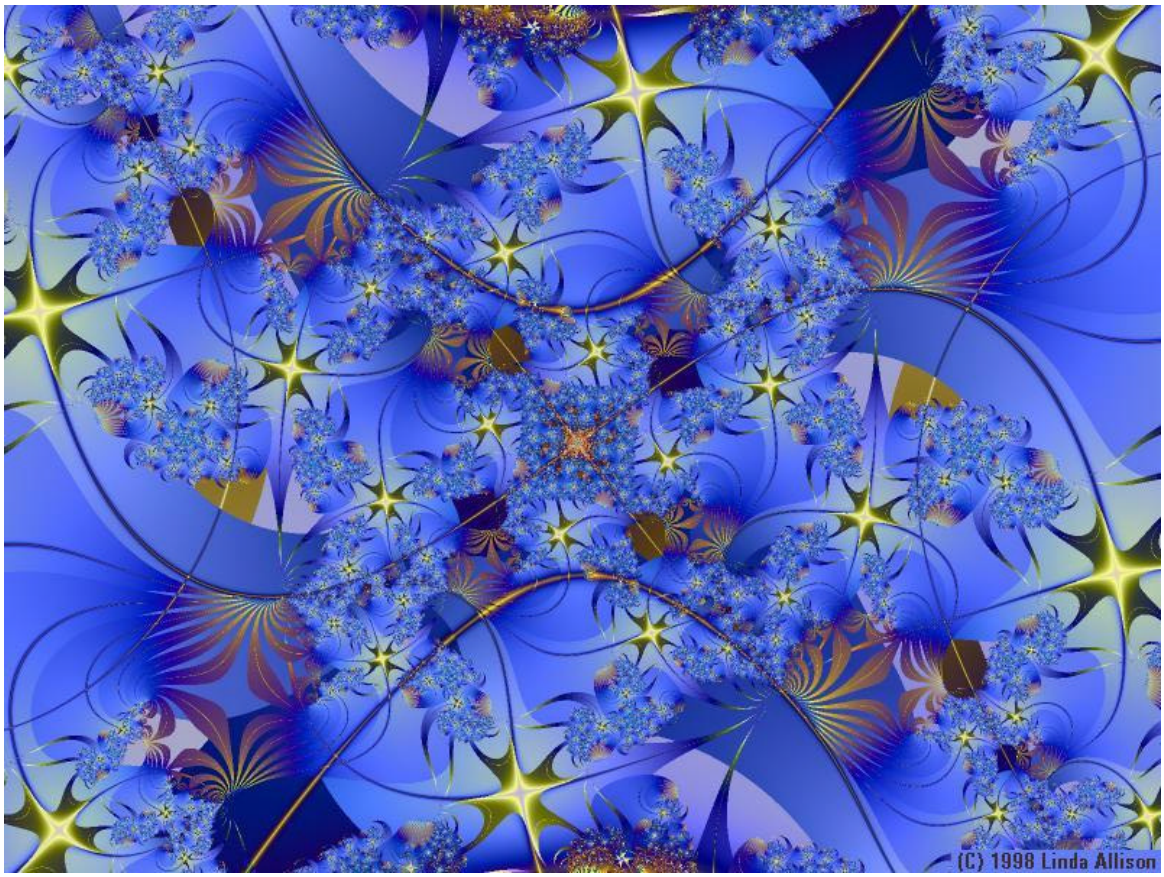


Illustration fra usenet nyhedsgruppen alt.binaries.pictures.fractals

af Damián Arguimbau

Vejleder: Berit Thaysen

Indholdsfortegnelse

1. Indledning

Hvori der gøres rede for opgavens emne, karakter, arbejdsmetoder, ambitioner og problemformulering.

1.1. Personlige bemærkninger

De følgende sider skal give en introduktion til den tilgang, jeg personligt har haft til hovedopgaven.

1.1.1. Begrundelse for hovedopgavens emnevalg

Det, du er begyndt at læse i, er en teoretisk opgave om databaser. I takt med at prisen pr. GigaByte harddisk er formindsket og processorhastigheden er forøget, spiller databaserne en stadig vigtigere rolle i vores samfund. I databaser gemmer vi CPR-numre, bilnummerplader, pasnumre, forbryderprofiler, skatteoplysninger, lovsamlinger, bibliotekskataloger - og som noget nyt inden for de seneste par år - også de oplysninger, internetagenter indsamler om internetsider, samt billeder, filmklip og lyd. Går en database ned, kan det betyde at bankerne ikke tør udbetale penge, at en arbejdsplads må stå stille, eller at politiet ikke kan håndtere hurtige eftersøgninger. Folketinget har anset det for fornuftigt at lave en registerlovgivning, der blandt andet sikrer mod samkøring af visse registre. Databaser er grundlaget for at man kan håndtere mange følsomme og vigtige systemer på en fornuftig måde.

De relationelle databaser er den type databaser, der er mest udbredte i dag. Men disse har problemer med at håndtere billeder, film og lyd på en effektiv, fornuftig og brugervenlig måde. Der er dukket konkurrenter op: Objekt-relationelle databaser og objektorienterede databaser. De første forsøger at bibeholde en tilknytning til det relationelle univers, mens de sidstnævnte først og fremmest læner sig op ad de objektorienterede programmeringssprog C++, Smalltalk og Java. At man skal gemme nye datatyper i databaser vil få nogle konsekvenser for den måde, man håndterer data på. Måske vil det få nogle konsekvenser for, hvordan man i det hele taget designer databaser. *Det* synes jeg er det spændende ved databaseudviklingen.

Det andet, jeg synes var spændende, var det “objektorienterede”. Problemet var, at det eneste jeg vidste om objektorientering i det hele taget, inden jeg begyndte på min hovedopgave, bestod af det, jeg havde lært på datanomuddannelsens C++ kursus plus en begrænset erfaring i Java og Javascript. Det var imidlertid interessant, synes jeg, at man kunne bygge egne klasser, genbruge og udbygge dem. Jeg kunne sagtens se for mig, at arv, polymorfisme m.m. burde kunne bruges på databasedesign. Det ville også være praktisk hvis C++ sproget kunne gemte data med det samme i en database, som jeg kunne kommunikere med via C++ kommandoer og ikke via SQL. Klasser, arv og polymorfisme, samt det at programmere var spændende. Jeg ville gerne koble denne interesse med min interesse for databaser.

Læsningen af det objektorienterede manifest, som Atkison udgav i 89 [Atkinson et al 89], og Date og Darwens manifest fra 95 [Darwen og Date 95] var de to tekster, jeg fandt mest interssante. Og efterhånden som tiden gik, og jeg læste Dates introduktion til databaseverdenen [Date 95], blev jeg mere og mere overbevist om, at Date på en eller anden subtil måde gik galt i byen, når han behandlede de objektorienterede databaser. Der var noget i hans og Darwens manifest, som jeg ikke helt kunne acceptere. Et af problemerne, mente jeg, lå i det faktum, at selve arbejdsmetoden var meget anderledes afhængig af, om man kom fra den relationelle eller den objektorienterede verden. Det synes Date ikke at tage hensyn til. Det ville jeg gerne undersøge nærmere.

1.1.2. Definitionen af hvad opgaven gik ud på

Jeg ville altså to ting: Få hold på hvad forskellen på de relationelle databaser og de objektorienterede databaser var, samt undersøge om Darwen og Date havde ret i, at den relationelle model kunne alt det, som de objektorienterede databaser sagde, at de kunne.

Selve sammenligningen mellem de relationelle og de objektorienterede databaser viste sig at være sværere end jeg havde forestillet mig. Det var forholdsvis nemt at finde frem til en definition af de relationelle databaser, som langt de fleste kunne blive enige om. Jeg ville bruge Date som hovedteoretiker, fordi han er en markant personlighed, der har bidraget konstruktivt til den relationelle database teori i mange år. Han bruger selv sin bog fra 95 som selve definitionen af, hvad en relationel database er. Så det ville jeg også gøre.

Det var sværere at finde ud af, hvordan jeg skulle gribe de objektorienterede databaser an. Problemet bestod i, at der viste sig at være flere opfattelser og implementeringer af, hvad objektorienterede databaser rent faktisk var. Og selv om det, som Atkinson m.fl. havde forfattet i 89 var et godt udgangspunkt, så var det ikke godt nok til at kunne bruges som en definition, jeg kunne sammenligne ud fra. Tendensen var imidlertid, at flere og flere gik over til den såkaldte ODMG standard, som Object Data Management Group (ODMG) havde defineret og for objektorienterede databaser. Det synes umiddelbart som en god vej at gå, ikke mindst fordi mange store spillere på markedet havde deltaget i arbejdet med at definere standarden. Så det blev den, jeg valgte at bruge.

Computer Associates udgav nogenlunde på dette tidspunkt i mine overvejelser en objektorienteret database ved navn *Jasmine*. Via nogle diskussioner i et digitalt forum (Politiken on-line, et privat BBS-system) kom jeg i kontakt med Computer Associates. De var så venlige at udlåne en *enterprise* udgave til mig. *Jasmine* er en ren objektorienteret database - men den implementerer ikke alle de features, som ODMG anbefaler.

På internettet kunne jeg endvidere downloade et produkt kaldet *Poet*, som er ODMG kompatibelt. Via deres C++ interface kunne jeg arbejde med en ODMG database, men desværre viste det sig at være sværere end først antaget. Jeg var ikke så fortrolig med Visual C++ som compiler, som var det, Poet brugte i de downloadede eksempler (der fandtes også eksempler til Borlands compiler, som jeg ikke har). Det viste sig, at de dels manglede at vedlægge en vigtig systemfil, og dels viste koden sig at være yderst vanskelig at kompilere. Det lykkedes til sidst, efter megen brok og hjælp fra både Poets support afdeling samt en C++ programmør fra mit nuværende arbejde, som var så venlig at hjælpe mig, da jeg gik i sort.

1.1.3. Afgrænsing og strukturering af opgaven

Der var disse to yderpunkter: relationelle kontra objektorienterede databaser. Det var godt nok. Jeg havde nu en officiel definition på begge, jeg kunne bruge for at forstå dem hver især og sætte deres verdensforståelse op mod hinanden.
Så havde jeg Darwen og Dates syntese af disse to verdener, som de beskriver det i deres manifest. Den syntese ville jeg belyse. Det var endnu bedre, for nu havde jeg fået foræret strukturen på min opgave:

1. Beskriv relationelle databaser,

2. Beskriv objektorienterede databaser,
3. Belys den praktiske forskel i verdensopfattelse mellem disse to modeller
4. Find ud af om Darwen og Dates syntese rent faktisk også kan rumme den samme funktionalitet, som objektorienterede databaser kan.

Det er denne struktur, opgaven følger.

Mit eneste problem var nu, at der fandtes disse objekt-relationelle databaser, som jeg i et vist omfang også gerne ville tage stilling til. At inddrage dem i selve analysearbejdet ville dog være alt for omfattende. Det ville svare til at definere en hel ny opgave. Hensigten med opgaven er at finde ud af, om den relationelle teori, sådan som Darwen og Date ser den, kan håndtere de samme opgaver som de objektorienterede databaser. Objekt-relationelle databaser har store indbyrdes forskelle, og derfor er de kun medtaget i det omfang, det virker relevant i sammenhængen. De steder, jeg har læst om dem, synes at antyde, at disse objekt-relationelle databaser forsøger at leve op til den SQL-3 standard, som var annonceret til at være klar i 1995, men som i følge rygter er udskudt til engang i 1999. Det er denne SQL-3 standard, Date og Darwen i virkeligheden forsøger at påvirke via deres manifest, da de har samme ønsker som SQL3-komitéen (bortset fra den ikke ubetydelige forskel, at Date og Darwen vil finde en erstatning for SQL). Hele diskussionen om SQL3 og de objekt-relationelle databaser er forholdsvis komplex. Dels er implementeringerne af de objekt-relationelle databaser som tidligere nævnt forskellige og ikke-kompatible, dels er SQL3 versionen stadig en "draft[1]". Ingen af de objekt-relationelle er landet på en løsning, der ligner den, SQL3 har i draften, omend der er overlappende ting. I opgaven har jeg set det formålstjenligt at afgrænse mig væk fra SQL3 og de objekt-relationelle databaser, da hele diskussionen ville blive forplumret i forhold til hovedopgavens egentlige ambition.

1.1.4. Litteratursøgninger

Bortset fra min vejleder, har internettet været min største informationskilde. Jeg har benyttet mig voldsomt af ACM (Association for Computing Machinery, http://www.acm.org), som er en "international scientific and educational organization dedicated to advancing the arts, sciences, and applications of information technology." Deres digitale bibliotek kan søge på ord i mange års artikler fra deres udgivelser og har

[1] Dvs. Et SQL3-arbejdspapir fra ANSI, da standarden endnu ikke foreligger i en godkendt version

været mig en uudtømmelig kilde til viden. Artikler, som jeg ikke ville have regnet med indeholdt noget interessant, dukkede pludselig op i mine søgninger. På den måde fik jeg mange uddybende forklaringer, der kunne bringe mig videre.

På samme måde har jeg brugt nyhedsgrupperne på internettet, især comp.databases.object, hvor jeg har bedt om hjælp de gange, jeg ikke helt kunne forstå nogle af ODMG's særheder. Jeg har fået svar fra så forskellige steder som University of Illinois og INSA Rennes i Frankrig. Især har Dipl.-Inform. Stefan H. Rupp fra Geodaetisches Institut der RWTH i Aachen været til stor hjælp.

Læsning af nyhedsgrupperne har også givet en ret god fornemmelse for, hvad det var, folk havde vanskeligheder med. Blandt andet viste det sig, at jeg ikke var den eneste, der havde problemer med at få C++ databaser til at fungere uproblematisk.

Denne opgaves litteraturliste består i høj grad af artikler fra diverse tidsskrifter, som også er blevet publiceret på internettet (heriblandt Darwen og Dates manifest samt Atkinsons). Desværre lykkedes det ikke at finde artikler, der behandlede Darwen og Dates manifest. Det skyldes sandsynligvis, at diskussionen om denne finder sted i universiteterne (det viser de henvendelser, jeg har fået fra nyhedsgrupperne), og at opgaver og specialer i universitetsregi ikke har nogen nævneværdig offentlig udbredelse.

Men selve søgning af litteratur på internettet er en metode, som kan anbefales til andre studerende. De bedste steder, fx ACMs digitale bibliotek, skal man betale for (ca. 300 kr.), men pengene er godt givet ud!

1.2. Problemformulering

Efter al denne baggrundsinformation følger hermed den officielle problemformulering for opgaven:

For det første vil jeg gerne undersøge, hvilken forskel der er i praksis mellem den relationelle og den objektorienterede databasemodel, især ved at afklare om der allerede i databaseanalytikerens designfase i realiteten optræder en markant divergens mellem disse to opfattelsesmodeller og om dette vil have stor betydning for den måde, en databasedesigner vil strukturere sine data på.

For det andet vil jeg kigge på den kritik, Date og Darwen retter mod de aktuelle implementeringer af relationelle databasesystemer. De hævder, at disse ikke til fulde lever op til den relationelle models teorier. De kritiserer også de objektorienterede databasers mangel på ordentlig teoretisk grundlag og foreslår en implementering af den relationelle model, der fuldt ud hviler på dets teoretiske grundlag, og som de mener samtidig overflødiggør den objektorienterede databasemodel. Denne hovedopgave skal afklare om de har ret i, at deres syntese i virkeligheden kan rumme alle de elementer, som den objektorienterede databasemodel implementerer.

2. Definitionsafsnit

Hensigten med dette afsnit er kort at definere nogle af de hovedbegreber, der bliver benyttet i nærværende hovedopgave.

En **databasemodel** forstås som en række begreber, der kan benyttes til at beskrive en databases struktur samt definitioner på de operationer, man kan udføre for at opdatere og trække oplysninger ud af databasen.

Højniveau databasemodeller beskriver strukturen på en måde, der ligger tæt op ad hvordan en bruger kunne opfatte data. Lavniveau eller fysiske modeller beskriver data, således som de bliver gemt på en computer. Herudover findes implempenterings- eller repræsentationsmodeller, som forsøger at beskrive data således som brugeren forstår dem, men som samtidig søger ikke at fjerne sig alt for meget fra hvordan data organiseres på en computer.

Som et absolut minimum vil alle højniveau modeller indeholde en beskrivelse af, hvordan følgende tre ting skal gribes an: datastrukturen, dataintegriteten samt datahåndteringen. I opgaven er beskrivelsen af de to databasemodeller indrettet efter disse tre hovedingredienser. Det gør det også nemmere at sammenligne dem senere.

I hovedopgaven arbejder jeg med to implementeringsmodeller, nemlig den relationelle og den objektorienterede databasemodel.

Et **databasesystem (DBS)** er en grundlæggende set et "computerized record keeping system" [Date 95] eller en "collection of programs that enables users to create and maintain a database" [Elmasri & Navathe 94]. Grundlæggende består et databasesystem af data, hardware, software og brugere. Til ethvert databasesystem hører et administrationssystem, et såkaldt **database management system (DBMS)**. Dette administrationssystem giver adgang til at *definere* databasens enkelte komponenter, til at *konstruere* selve databasen og til at *manipulere* de data, som databasen inkluderer. Langt de fleste DBMS'er vil også have et *data dictionary*, hvor databasens struktur dokumenteres og en *transaktions administrator*, som udfører "recovery and concurrency control" [Date 95].

Databaseadministrationssystemet (DBMS'en) håndterer i princippet al adgang til data. Det, der sker er følgende:

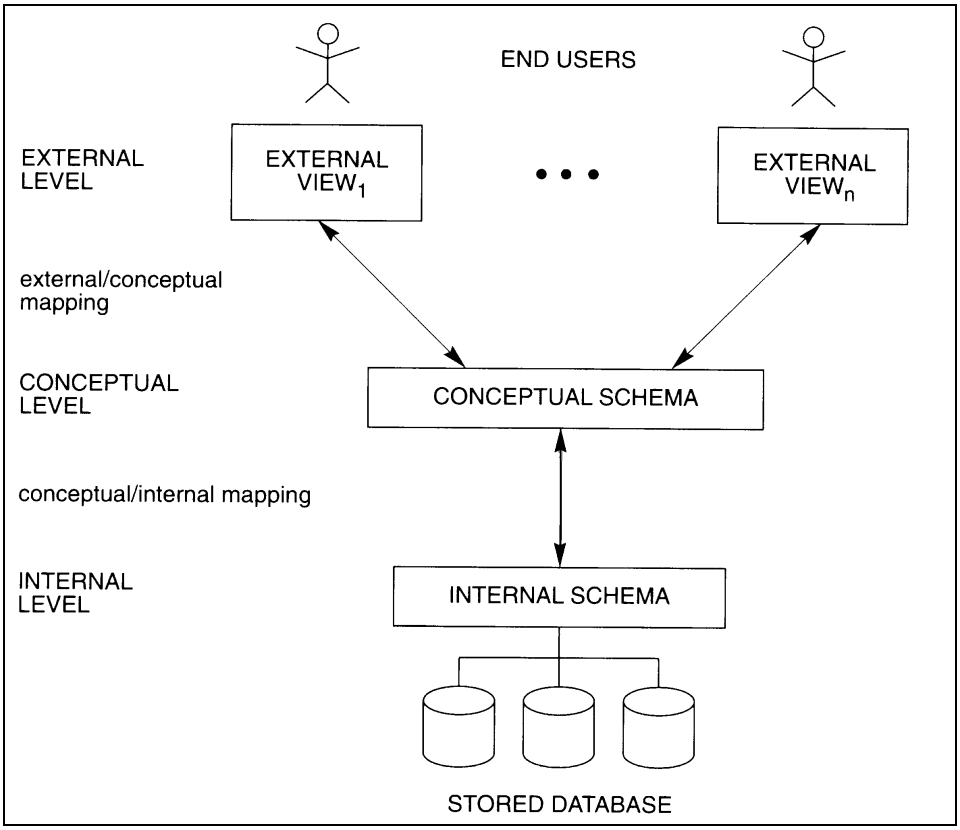
1. En bruger sender en opdaterings- eller udtrækskommando via et databaseadministrationssprog.
2. DBMS'en modtager kommandoen og analyserer det.
3. DBMS'en undersøger det eksterne skema for den bruger, der har sendt kommandoen, det konceptuelle skema samt den fysiske datastruktur
4. DBMS'en udfører de nødvendige operationer på databasen og genererer et output til brugeren.

DBMS'en er i virkeligheden det, der udgør forskellen mellem et filsystem og et databasesystem. I et filsystem har man adgang til data direkte, mens man i et databasesystem med DBMS vil have et system, der sikrer at datahåndteringen respekterer datastrukturen samt dataintegriteten.

2.1. Tre-skema arkitekturen

Hensigten med tre-skema arkitekturen er at adskille den fysiske database fra brugerapplikationen. De tre lag i databasearkitekturen er som følger:

1. Det **interne** niveau beskriver den måde, data fysisk lagres på.
2. Det **konceptuelle** niveau beskriver hvordan hele databasens struktur ser ud for databasens brugere. Højniveau- og implementeringsmodeller kan benyttes til at beskrive det konceptuelle niveau.
3. Det **eksterne** niveau indeholder en række modeller, der hver især definerer, hvordan en bestemt gruppe brugere skal se en bestemt mængde af databasens oplysninger. Dette niveau kan også beskrives af højniveau og implementeringsmodeller.



Fra Elmasri & Navathe 1994

Tre-skema arkitekturen kan blandt andet bruges til at forklare ideen om **datauafhængighed**. Ved at adskille de tre niveauer kan man ændre i hver enkelt af de tre niveauer uden at de øvrige to nødvendigvis påvirkes af ændringerne. Dvs. at man for eksempel kan definere en anden måde at lagre data på, uden at det nødvendigvis betyder, at man skal lave om på databasestrukturen eller på den måde, brugere ser data.

2.2. Definitioner relateret til objektorienterede databaser

Bredt opfattet er objekter "a real-world or abstract entities that we model in a database" [Catell 94]. Mere specifik definition følger her:

- **Dataindkapsling** "dækker muligheden for at indkapsle forskellige dataelementer (…), der udefra kan betragtes som en enhed." [Glaven 92] Denne enhed kaldes en **objektklasse**. Variable af klassen kaldes **objekter**. Man kan sikre at adgang til det indkapslede kun kan ske gennem en specificeret grænseflade.
- Objekter kan have bestemte måder at opføre sig på, idet man kan knytte **metoder** til objekterne.

- **Arv** "dækker muligheden for at definere nye objektklasser, som arver egenskaber fra allerede eksisterende objektklasser. Ved arv og tilføjelse af nye egenskaber kan der opbygges et hierarki af mere og mere specialiserede klasser." [Glaven 92]
- Via **polymorfisme** kan man opnå, at det samme metodekald udfører forskellige opgaver, afhængig af det kaldte objekts type. Polymorfisme forudsætter arv.

Den objektorienterede programmeringssprogs typebaserede logik gør, at en type beskriver egenskaberne ved variablen, såsom størrelsen, det mulige indhold samt regler for anvendelse og operationer på variablen. Typen er bestemmende for reservering af plads til data og for behandlingen af dem [jf Glaven 92]. Alle operationer på data forudsætter, at man definerer en given returtype. En objektklasse er en brugerdefineret type.

3. Relationsdatabaser

I dette hovedafsnit er det min hensigt at gøre to ting:

1. Kort at ridse relationsdatabasens historie op
2. Nærmere at definere den relationelle databasemodel

Det første punkt er at betragte som en introduktion af relationsdatabaser som sådan. Det er for mig at se vigtigt, at det historiske perspektiv ikke glemmes, da denne rummer nøglen til at forstå hvorfor og hvordan tingene blev som de er, samt rummer værdifulde oplysninger om, hvordan man i det hele taget kan gribe databaser an.

Det andet punkt er at betragte som det grundlag, hvorudfra sammenligningen med de objektorienterede databaser finder sted. Jeg har derfor valgt enkelte punkter af den relationelle databaseteori ud, som jeg specielt ville fokusere på. Dermed er også sagt, at jeg forudsætter en vis forudgående viden om relationsdatabaser. Sammenlignet med dette afsnit er beskrivelsen af de objektorienterede databaser mere omfattende, fordi jeg betragter det som sandsynligt, at en potentiel læser kender til relationelle databaser, men kun ved lidt om de objektorienterede databaser. Hvor det er særlig relevant drager jeg paralleller mellem de to typer databaser allerede i dette afsnit. Disse vil typisk bestå i nogle egenskaber, som man særligt skal lægge mærke til, da de har betydning for den senere sammenligning med de objektorienterede databaser.

3.1. Historien bag

Codd lagde i 1970 [Codd 70] kimen til den moderne relationsdatabase teori, der skulle komme til at afløse den hierarkiske model og netværksmodellen.

Hensigten med Codds forskning var at sørge for en uafhængighed mellem den måde de fysiske data lagres på og den måde, de repræsenteres på over for brugeren. Den relationelle model, skriver Codd [Codd 70]: “provides a means of describing data with its natural structure only - that is, without superimposing any additional structure for machine representation purposes.” I parentes bemærket er det præcis det samme som de objektorienterede modeller hævder at gøre. Men det er værd at bemærke, at da Codd skrev dette, var der tale om et paradigmeskift, idet de daværende databasesystemer i høj grad krævede, at data tilpassede sig maskinen snarere end omvendt.

Codd fortsætter gennem 70'erne med at udvikle den relationelle tankegang i en række artikler og udgiver en formel definition af den relationelle algebra i 1972. Algebraen bliver det teoretiske grundlag for forespørgsels sproget SQUARE (Specifying Queries as Relational Expressions), som Boyce m.fl. beskrev i 1974. SQUARE udviklede sig igen til SEQUEL og endte til sidst med at blive den SQL, som ANSI i 1986 ophøjer til standard.

Det er et diskussionsemne, hvornår de første egentlige kommercielle relationelle produkter kommer - det afhænger af, hvor stringent man vil følge Codds definitioner af databasekravene. Men først hen mod slutningen af firserne kan man tale om, at de relationelle databaser fuldstændig erstatter de hidtidige hierarkiske databaser og netværksdatabaserne.

I 1990 fremfører Codd det, han kalder for den relationelle model, version 2 [Codd 90]. Hovedforskellen mellem v1 og v2 er, at Codd i anden omgang forsøger at beskrive hele databasen og ikke kun de tre hovedelementer[2]. Version 2 indeholder hele 18 elementer i modsætning til version 1's tre. Der er dog ikke konsensus omkring Codds nye tilføjelser til den relationelle model, og jeg har derfor valgt slet ikke at inddrage hans version 2 i min gennemgang af den relationelle teori.

Allerede før Codd udgiver sin relationelle model, Version 2, er Date og Darwen godt på vej til at bliver særligt toneangivende teoretikere inden for den relationelle database teori. Dates bog, *An Introduction to Database Systems,* er på dette tidspunkt standardlærebogen i mange universiteter. Og såvel Date som Darwen udgiver et hav af artikler og bøger i løbet af firserne og halvfemserne, herunder en kritik af Codds Version 2 [Date 92].[3] Vigtigt i denne sammenhæng er at notere sig, at især Date påtager sig rollen som den relationelle databasemodels fortolker, forkæmper og beskytter.

Grunden til, at Date pludselig skal se sig som beskytter er, at de seneste år har den relationelle model været under “angreb” af de nyere objektorienterede modeller, der

---

[2] Nemlig datastrukturen, dataintegriteten og datahåndteringen

[3] Som en kuriøsitet kan nævnes at Codd og Date i 1984 havde etableret firmaet “Codd & Date consulting Group” sammen. Firmaet eksisterer stadig og rådgiver kunder omkring databaseløsninger og design. Date forlod firmaet i 1991.

begyndte inden for programmering (først og fremmest Simula, C++, Smalltalk og senest Java), men hvis praktiske og teoretiske anvendelse snart begyndte at omfatte databaser. En af grundene til, at databasefolk begyndte at se sig om efter en afløser til relationsdatabaserne var, at man begyndte at få brug for at ordne komplekse strukturer i en database, som for eksempel filmklip, fotos m.m., snarere end tekst eller talstrenge. Dette skyldes igen den revolution inden for processorkraft, harddiskkapacitet og båndbredde, der har fundet sted inden for de seneste år. De relationelle databaser håndterer data ud fra deres værdier. Men filmklip, fotos, lyd m.m. har ingen menneskeligt forståelige værdier, hvis man kun ser på bitstrengene. Derfor var relationsdatabaserne slet ikke klar til at håndtere disse typer informationer, da de skulle ordnes i databaser.

Date og Darwen har derfor de seneste år taget fat på at undersøge den relationelle model endnu nøjere og i bøger og artikler (se bl.a. Date 95) forsøgt at få indpasset nogle elementer inspireret af de objektorienterede databaser i den, bl.a. indkapsling, metoder m.v. *uden* at disse tilføjelser går ud over den relationelle dataintegritet.

Den relationelle database teori har været genstand for megen forskning. Den har derfor et solidt teoretisk grundlag, den kan hvile på. Det betyder dog ikke, at der ikke fortsat forskes og videreudvikles inden for emnet. Som Date skriver: “I too have changed my opinions on many matters and will no doubt continue to do so.” [Date 95]. Udviklingen fortsætter - også inden for forskningen.

De øgede krav om funktionalitet til databaser giver sig udslag i nye produkter, produkter, som gør deres bedste for at levere (hver deres) implementerbare løsninger på de udfordringer, som den nyeste udvikling i samfundet præsenterer databaseprodukterne for. Samtidig knokler forskerne på for at finde holdbare teorier, der kan sørge for, at implementeringen af de nye databaser kan basere sig på en god, teoretisk ballast, der sikrer mod fejl og indbyggede besynderligheder i programmerne. Et eksempel på disse produkter er de objekt-relationelle databaser.

I det følgende vil vi kigge på den del af den relationelle model, som danner det teoretiske grundlag for de kommercielle relationelle databaser.

3.2. Den relationelle model

Hensigten med dette afsnit er at give et overblik over den relationelle model. De dele af den relationelle model, som har en særlig betydning for sammenligningen med de objektorienterede modeller, vil blive behandlet. Det gælder især de punkter, hvor Date og Darwen dels kritiserer den relationelle models implementering i de kommercielle produkter, dels udvider den relationelle models funktionalitet ved at give en ny vinkel til definitionen af den relationelle models enkeltdele.

3.2.1. Overordnet gennemgang

Den relationelle model er en måde at repræsentere data på. Set fra en almindelig brugers øjne er en relationel database organiseret i tabeller (relationer). Rækkerne i tabellerne repræsenterer de enkelte databaserecords (tupler) og kolonnerne i tabellen de enkelte felter (attributter) i databasen.

Den relationelle model rummer faciliteter til at håndtere:

1. datastrukturen
2. dataintegriteten
3. datahåndteringen

Hvordan data herudover lagres rent fysisk på et givet datamedie vedkommer ikke den relationelle model, da den er en implementationsmodel, der beskriver databasen på et konceptuelt niveau.

3.2.2. Datastrukturen

Datastrukturen har at gøre med, hvordan man repræsenterer en given entitet i en relation. Hensigten med dette afsnit er at give et overblik over de datastrukturelle dele af den relationelle model for dernæst at positionere de særlige ting ved den datastrukturelle del, der har indflydelse på sammenligningen med de objektorienterede databasers datastruktur.

De overordnede regler for den relationelle model er opsummeret i nedenstående skema.

Overordnede regler for datastrukturen [Codd 85]**:**

1. Der kan ikke forekomme ens tupler i en relation og tuplers rækkefølge er ligegyldig, hvilket også gælder for deres attributter

2. Resultatet af en forespørgsel er en ny relation, som kan gemmes og som man senere kan arbejde videre på.
3. Tabeller bruges til at repræsentere de gemte data
4. Hver enkel kolonne i en tabel svarer til en attribut
5. View tabeller er virtuelle relationer, som repræsenteres internt af en eller flere relationelle komandoer og ikke via gemte data.
6. Snapshot modeller er relationer som er blevet evalueret og gemt i databasen sammen med en dato, der angiver hvornår snapshoten blev taget.
7. Domæner er det sæt værdier, hvorfra en eller flere kolonner validerer deres data

3.2.2.1. Domænet

Domænet er den del af relationsteorien, der danner grundlag for Dates forsøg på at vise, at den objektorienterede models begreber og funktionaliteter allerede ligger latent i den relationelle models teorier.
Domænet er en definition af en komplex mængde af legale værdier. Et par eksempel viser, hvad Date mener [Date 95]:

- mængden af legale værdier for en attribut som “danske postnumre” svarer til mængden af en række intervaller af firecifrede tal.
- mængden af legale værdier for “dag_navn” er ugedagenes navne

Pointen er, at domænet kan indeholde en datatype af forskellig kompleksitet - herunder nestede, komplekse strukturer - men som følger visse lovmæssigheder, der kan defineres nærmere og derfor valideres.

Codd [Codd 85] er helt på linie med Date, når han nævner vigtigheden af, at databasesystemer (og SQL) understøtter domæner.
Fordelene er, at:

- man for eksempel kunne nøjes med at angive en definition for et domæne (fx et interval af fire-cifrede tal), hvorfra en eller flere attributter kan validere sine data. Det er pladsbesparende, fordi vi ikke fysisk behøver at oprette en relation, der indeholder alle de legale tupler, som attributten kan bruge.
- Ved de algebraiske operationer kan DBMS'en først tage stilling til, om to inkompatible domæner bliver sammenlignet og advare brugeren om dette.
- Domæner er med til at sikre dataintegriteten uden at det går ud over databasers distribuerbarhed.

3.2.3. Dataintegriteten

Dataintegriteten sikrer konsistens mellem data, således at man ikke risikerer at miste data. Dette gøres ved at sikre, at der er:

- Dataentydighed
- Referenceintegritet
- Brugerdefineret dataintegritet

Hensigten med dette afsnit er at give et overblik over de dataintegritetsmæssige dele af den relationelle model og fremhæve de særlige ting ved dem, der adskiller dem fra de objektorienterede databaser.

3.2.3.1. Dataentydigheden

Dataentydigheden indebærer, at man altid kan finde en relevant tupel. Det gøres ved, at man har defineret mindst en kandidatnøgle, som kan sikre, at man altid kan finde de relevante data frem[4].

En del af entitetsintegriteten handler også om at sikre, at der ikke er redundans. Eller, hvis der er redundans, at sikre at begge tupler bliver opdaterede på én gang. I modsat fald kan man få problemer med inkonsistente data, hvor den samme entitet, der er repræsenteret to forskellige steder i databasen, optræder med forskellige værdier. Disse problemer relaterer sig også til referenceintegriteten.

3.2.3.2. Referenceintegritet

Referenceintegriteten sikres ved at anvende en kandidatnøgle som fremmednøgle i en relateret tabel. Fremmednøglen repræsenterer således en *reference* til tuplen, som indeholder den matchende kandidatnøgle. DBMS'ens opgave er at sikre, at alle fremmednøgler ulig fra null matcher en kandidatnøgle i den tabel, der refereres til, da der ellers ville optræde en referenceintegritetsfejl.
Det er vigtigt at notere sig, at der ikke tilsvarende kræves en fremmednøgle i den tabel, som fremmednøglen refererer til. Der er netop ikke tale om et “to-vejs link”, men om en

---

[4] En kandidatnøgle er en nøgle, der for R er en delmængde af mængden af R's attributter, K, således at der ikke er to forskellige tupler i R, der på noget tidspunkt har samme værdi i K.
Ingen af attributterne i K kan bortfalde uden at K derved mister sin evne til unikt at identificere en tupel i R. En primærnøgle er den kandidatnøgle, man har valgt som den primære nøgle i sin relation.

angivelse af, at en række oplysninger hører til en bestemt relation. Med andre ord, referenceintegriteten siger simpelthen, at *hvis B refererer til A, så må A eksistere*.

3.2.3.3. Brugerdefineret integritet

Den brugerdefinerede dataintegritet indebærer, at brugeren har sat nogle betingelser op, som altid skal opfyldes: fx at fornavnet i en navnetabel altid skal være udfyldt, eller at den skal have en standard-værdi.

Endvidere introducerer Date en ny integritetsregel: *domæneintegriteten*. Denne regel fastslår det indlysende, at ethvert attribut skal validere sine værdier i det relevante domæne.

3.2.4. Datahåndteringen

Den tredje og sidste del af den relationelle teori omhandler datahåndteringen, som er baseret på den relationelle algebra. I dette afsnit vil jeg hovedsageligt beskæftige mig med datahåndtering i forbindelse med udtræk af data. Det skyldes, at man i den objektorienterede model kun har datahåndtering defineret for udtræk, mens datahåndtering for opdatering af data afhænger af den funktionalitet, man lægger i sin database. Det er derfor ikke formålstjenligt at gennemgå opdateringsfaciliteter i den relationelle model.

Det er vigtigt at forstå algebraens funktion i den relationelle teori, fordi de objektorienterede databaser forholder sig anderledes til verden end de relationelle. En indlysende grund til, at den objektorienterede model er nødt til at forholde sig ambivalent til den algebraiske model er, at input og output i den relationelle algebra netop er *relationer* og ikke *objekter*. Det skyldes, at objekter ikke nødvendigvis overholder 1NF.

En anden pointe, man bør udlede af ovenstående er, at relationsdatabasemodellen fuldt dækker et matematisk område (algebraen). Relationsdatabaserne kan derfor benytte sig af algebraens matematiske operationer og samtidig have fuld teoretisk dækning for operationerne.

Date understreger, at der i algebraen er tale om “set-operations”, dvs. at man arbejder på en given relation, der er afgrænset ud fra nogle kendetegn (fx alle postnumre, der begynder med cifrene 14), ganske som det er tilfældet i mængdelæren. Denne måde at arbejde på er med til at sikre dataintegriteten i en relationel database.

Med den relationelle algebra kan vi skrive kommandoer (“expressions”) , der giver mulighed for at man kan:

- finde data via forespørgsler
- opdatere data
- definere views
- definere snapshots
- definere autorisationsregler
- sikre dataintegritet ved flere samtidige brugere
- definere integritetsregler

Såvel input som output i algebraen er relationer. Det betyder, at outputtet fra en operation kan benyttes som input til den næste operation.

Via algebraen skal det være muligt at definere et abstrakt højniveau sprog, der kan håndtere data i den relationelle database. Et sprog kan derfor siges at være fuld relationelt, såfremt den kan udvirke præcis de samme funktioner som den relationelle algebra definerer og overholder de betingelser, der er nævnt ovenfor.

3.2.4.1. SQL-sproget

Mens den relationelle algebra repræsenterer det teoretiske grundlag for, hvilke forespørgsler, der i det hele taget er muligt at udføre på relationer, så er SQL det sprog, som i praksis bruges i de kommercielle databaser. SQL er altså en implementation af den relationelle algebra. I SQL overlader man til DBMS'en at optimere forespørgslen bedst muligt. SQL er samtidig defineret som en ANSI-standard, der skal gøre det muligt at migrere fra et relationelt databaseprodukt til et andet uden nævneværdige problemer.

Jeg vil i det følgende forholde mig til SQL-sproget som defineret af ANSI i 92 (kendt som SQL-2 eller SQL/92) . Som nævnt tidligere arbejdes der i øjeblikket på en SQL-3 version, der skal indeholde nogle af de muligheder, man har i objektorienterede databaser. Hvordan SQL-3 vil ende med at se ud, er endnu uklart.

Darwen og Date skriver i deres manifest, at SQL er en “perversion” af den relationelle model [Date & Darwen 95] og foreslår i stedet, at man begynder at konstruere et nyt

sprog, som de har givet det midlertidige navn, “D”. Codd er på linie med dem: allerede i 85 skriver han, at “the proposed ANSI standard does not fully comply with the relational model, because it is based heavily on that nucleus of SQL that is supported in common by numerous vendors.” [Codd 85].

SQL kan for eksempel godt returnere andet end relationer (fx hvis der er to ens records i en tabel) - dermed overtræder den faktisk de regler den relationelle algebra sætter op. Herudover anfører Date, at SQL ikke supporterer snapshots [Date 95].

I praksis er problemet ikke så synligt, men det er rigtigt, at det forstyrrer det teoretiske billede. Det er dog ikke helt tilfældigt, hvordan SQL overtræder den relationelle model: Det kan for eksempel være en fordel at tillade ens tupler i en relation, fx. hvis man udfører lønsammentællinger.

Det alvorlige problem ved SQL er imidlertid, at man kan definere ganske komplicerede forespørgsler, som ikke umiddelbart er til at gennemskue, og hvor man derfor kan være i tvivl om resultatets sandhedsværdi.

Kritikken mod SQL er derfor først og fremmest af konceptuel karakter. Der er for lidt logik i SQL og for mange fejlmuligheder. SQL er ikke stringent nok og er præget af en hel del lappeløsninger. Som Date skriver et sted, så er SQL udtryk for en række programmørers forsøg på at implementere en teori, som de ikke fuldt har forstået. Det er derfor, at Date og Darwen foretrækker et helt nyt sprog i stedet for SQL

3.2.5. Det relationelle databasedesign

Efter at have fået et overblik over den relationelle models komponenter og SQL sproget, er ballasten til stede for at undersøge designaspektet nærmere. Designet skal tage hensyn til de datastrukturelle, datahåndteringsmæssige samt dataintegritetsmæssige aspekter i den relationelle model. Til at hjælpe designere med dette, har teoretikere fundet frem til forskellige algoritmer, som man kan benytte når man skal designe sin relationelle database. Disse er dog ikke behandlet grundigt her, idet algoritmerne ikke har nogen særlig betydning for de objektorienterede databasers designmetode.

I det følgende vil jeg se på designaspektet i den relationelle model. Denne vil blive behandlet grundigt på det konceptuelle niveau, da et af de spørgsmål jeg ønsker at afklare med denne hovedopgave er, om der allerede i designfasen optræder en divergens mellem den relationelle og den objektorienterede måde at arbejde på. Der vil være et tilsvarende afsnit under gennemgangen af de objektorienterede databaser, hvor jeg vil trække fronterne op mellem de to arbejdsmetoder, for senere, i sammenligningsafsnittet, at eksemplificere de forskelle i designtilgange, jeg har fundet frem til.

3.2.5.1. Designangrebsvinkler

Databaser er et udtryk for et udsnit af verden. Databasedesign er den disciplin, der forsøger at konvertere de entiteter i verden, man ønsker at organisere i en database, til en række datastrukturer med indbyggede integritetsregler og datahåndteringsmekanismer. I ethvert design ligger der endvidere et anvendelseskriterie, nemlig et hvor man forsøger at operationalisere en mængde data.

Der er to typiske angrebsvinkler for den relationelle databasedesigner:

1. Manuel med efterfølgende mapning: Man laver et ER eller EER-model, som man bagefter mapper til en relationel model. Med lidt andre ord så går man fra en konceptuel model, der beskriver data i overordnede kasser, til en mere detaljeret (relationsmodellen). Umiddelbart efter mapningsprocessen sørger man for at tjekke de funktionelle afhængigheder og de normalformer, den relationelle model arbejder med.
2. Algoritmisk, hvor man dekomponerer relationer: Det vil sige, at man ud fra nogle givne relationer, man har fundet frem til, tjekker funktionelle afhængigheder og gennemfører de algoritmiske procedurer, der ligger implicit i normalformerne. Via denne procedure dekomponerer man relationerne til man til sidst har et relationelt design.

Selv om de to angrebsvinkler er konceptuelt forskellige, så gennemfører de begge de samme øvelser, den første i forbindelse med sin mapning, den anden i forbindelse med dekomponeringen. Det skyldes, at den relationelle designers værktøjer består af to ting: Den funktionelle afhængighed og normalformerne.

3.2.5.2. Værktøjerne

Den funktionelle afhængighed viser, hvordan de enkelte attributter hænger sammen indbyrdes. Via den funktionelle afhængighed kan vi afdække de interne strukturer i relationer og eventuelt dekomponere dem i mindre strukturer[5].

De generelle normalformer[6] sikrer først og fremmest de relationelle datastrukturelle, datahåndteringsmæssige og dataintegritetsmæssige aspekter. Derigennem vil man i sine relationer have en struktur, der understøtter den relationelle algebras operationer. Et af de hensyn, der ligger underforstået i ovenstående procedurer er ønsket om at fjerne redundans i data samt sikre, at der ingen uægte tupler kan opstå ved joins. Herudover er det også et ønske at minimere null-værdier i sit design.

Således får man til sidst et design, der formodentligt består af en hel række små enheder, der kan kombineres via naturlige joins, så de over for brugeren kan repræsentere den virkelighed, han forventer, på en måde, så der ikke opstår problemer med dataintegriteten og referenceintegriteten.

Der er dog udbredt enighed blandt databasefolk [Date 92 og Elmasri 92] om, at databasedesign stadig er lidt af en kunst. Der kan være tilfælde hvor man eksempelvis af hensyn til performance undlader at tage hensyn til alle funktionelle afhængigheder og laver en database, der ikke lever op til samtlige normalformer som beskrevet ovenfor. I en virkelighed, der langt fra er perfekt, er det svært at lave en perfekt repræsentation af selvsamme virkelighed.

Date anbefaler, at man forsker mere i design-algoritmer. Han går så langt som til at sige [Date 92], at man er nødt til at foretage den netop beskrevne arbejdsmetode *uanset* hvilken databasemodel, man vil anvende: “the right way to do database design in a non-relational

---

[5] Definitionen af den funktionelle afhængighed er, at *x* bestemmer funktionelt *y* hvis der for alle par af tupler t1, t2 i relationen (R) gælder at: t1[x] = t2[x] -> t1[y] = t2[y].

[6] 1NF: Kræver, at alle attributter i R er simple og enkeltværdiede og at der er mindst én kandidatnøgle; 2NF: Kræver, at enhver ikke primær attribut i R er fuldt funktionelt afhængig af enhver kandidatnøgle; 3NF: Kræver, at der for enhver funktionel afhængighed x -> a gælder, at x er en kandidatnøgle eller at a er en primær attribut; BCNF: Kræver, at der for enhver funktionel afhængighed x -> a gælder, at x er en kandidatnøgle; Der findes også en 4NF og en 5NF.

system is to do database design first, and then, as a separate and subsequent step, to map that relational design into whatever nonrelational structures the target DBMS happens to support". Hvorfor objektorienterede databasefolk er ikke spor enige med Date i dette, vil vi se på senere.

3.2.6. Opsummering

De foregående afsnit danner grundlaget for, at jeg senere kan sammenligne de relationelle databaser med de objektorienterede databaser. At jeg har defineret de relationelle begreber jeg senere skal bruge sikrer, at sammenligningen i opgaven kan foretages ud fra nogle kendte præmisser, som kan kritiseres og evalueres.

Undervejs i gennemgangen af de enkelte afsnit har jeg trukket de ting frem, som man især skal lægge mærke til i forbindelse med de relationelle databaser.

Vedrørende **datastrukturen** så fremhævede jeg domænedefinitionen, som fik forøget vægt i forhold til aktuelle implementeringer af de relationelle databaser.

I relation til **dataintegriteten** nævnte jeg, at der i alle relationer var behov for at definere mindst en kandidatnøgle samt at referencer foregik via fremmednøgler.

Datahåndteringsmæssigt bemærkede jeg, at den relationelle model er baseret på den relationelle algebra, og at SQL er det værktøj, man i aktuelle implementeringer benytter til at håndtere data på. Jeg berørte kort, at Date, Darwen og Codd fandt SQL utilstrækkelig som et praktisk værktøj til den relationelle model.

Designmæssigt gennemgik jeg de to hovedangrebsvinkler på den relationelle database samt konkretiserede, at begge metoder er baserede på afklaring af funktionelle afhængigheder samt normalformerne.

4. Objektorienterede databaser

Hensigten med dette afsnit er at give et overblik over den objektorienterede model. Samtidig vil jeg tydeliggøre forskellene mellem den relationelle og den objektorienterede model. Det betyder, at jeg i de følgende afsnit i et vist omfang også vil sætte de objektorienterede databaser i forhold til de relationelle.

Som vi husker, så sagde jeg i min problemformulering, at jeg gerne ville undersøge, hvilken forskel der i praksis er mellem den relationelle og den objektorienterede databasemodel, især ved at afklare om der allerede i databaseanalytikerens designfase optræder en divergens i arbejdsmetoden. Af afgrænsningsgrunde er der derfor ikke i opgaven lavet en egentlig grundig sammenligning mellem den relationelle og den objektorienterede model som sådan. Jeg interesserer mig i hovedopgaven for en besvarelse, der tydeliggør de praktiske og grundlæggende forskelle i databasedesignet - og dermed indirekte afklarer de to databasers konsekvenser.

Jeg benytter derfor gennemgangen af de objektorienterede databaser til også at fortælle om forskelle i de to involverede modeller. Disse forskelle vil dels gøre det nemmere at se de objektorienterede databaser i perspektiv, dels vil disse forskelle senere blive endnu mere tydelige, når jeg i det næste hovedafsnit går i krig med at finde ud af, hvordan designet i de to konkurrerende modeller fungerer.

4.1. Historien bag

Objektorienterede koncepter har udviklet sig inden for tre discipliner [Kim 94]: først i programmeringssprog, hvor Simula 67 sædvanligvis anskues som det første OOP sprog. Dernæst i AI-forskingen (kunstig intelligens), for derefter at fortsætte i analyse og design-metoder af edb-systemer og slutteligt i databasedisciplinen.

De data, brugere benytter sig af i dag, og dermed ønsker at organisere, består i højere grad end nogensinde før af oktetstrømme, der repræsenterer billeder, film og andre typer komplexe data. Disse er svære at gemme og kalde frem på en praktisk måde i dagens relationelle databaser. Behovet for objekt orienterede databaser opstod især på grund af disse komplexe data.

Forskellige typer objektorienterede databaser begyndte så småt at komme frem i løbet af firserne, hver med deres fortolkning af, hvad en objektorienteret database var. De første, der med relativt stort held forsøgte at definere hvilke krav man kunne stille til en objektorienteret database system var Atkinson m.fl. i 1989 [Atkinson 89]: "An object-oriented database system must satisfy two criteria: it should be a DBMS, and it should be an object-oriented system, i.e., to the extent possible, it should be consistent with the current crop of object-oriented programming languages. The first criterion translates into five features: persistence, secondary storage management, concurrency, recovery and an ad hoc query facility. The second one translates into eight features: complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extensibility and computational completeness."

Der er i dag to hovedtyper objektorienterede databaser:

1. De databaser, der ved hjælp af ekstra standardbiblioteker til et givet objektorienteret programmeringssprog udvider sproget med funktionskald, der giver "transparently persistent data, concurrency control, data recovery, associative queries, and other database capabilities." [Catell 97]. Det gælder fx en database som *Poet*. Tanken er, at disse typer databaser er kompatible med alle produkter inden for samme programmeringssprog (dvs. at man kan benytte fx en C++ database via enten Visual C++, Borland C++ eller Metrowerks C++ produkter). Det er altså ikke tanken, at en C++ database skal kunne benyttes med en Java compiler. Det er denne type databaser ODMG standarden især behandler.
2. De databaser, hvor man kan udvikle løsninger med et eget, lukket (proprietært) DBMS sprog. Det gælder fx en database som *Jasmine*. Her kræver det, at man har Jasmine installeret, hvis man vil håndtere databasestrukturen, men man kan så tilgengæld typisk benytte sig af standard Object Query Language (OQL) (som defineret af ODMG) for at trække data ud af databasen. Jeg vender tilbage til OQL senere.

Herudover er der de objekt-relationelle databaser. Kendetegnede for disse er, at de baserer sig på en relationel model, men tillader, at man definerer en relation som et objekt, man kan indlejre som en attribut i en anden relation. Disse hybriddatabaser behandles ikke.

Den nyeste trend, som heller ikke er medtaget i opgaven, da det falder udenfor problemformuleringen, er "*persistent programming languages*" som PJAVA, der er

objektorienterede programmeringssprog med mulighed for at gemme data og udføre databaselignende operationer på disse.

Som det blev antydet ovenfor er der derfor ikke en entydig objektorienteret model for databasesystemer. Der pågår standardiseringsarbejde i ISO og ANSI regi i forbindelse med C++ samt SQL 3 standarderne [Cattell 97], som er relevant for de objektorienterede databaser.

Det største arbejde i form af standardisering af objektorienterede databasesystemer er udført af ODMG (Object Database Management Group), der i juli 97 udgav deres standard version 2.0 [Catell 97]. Parallelt hermed er der standardiseringsarbejde i gang i OMG (Object Management Group), som blev stiftet af HP, SUN og et dusin andre virksomheder. Deres mål er at fremme og standardisere objektorienteret teknologi. Mest relevant for databasearbejdet i forbindelse med OMG er deres CORBA eller "Object Request Broker" standard, der fortæller hvordan tilgangen til objekter i et givet system skal struktureres. Denne OMG standard har berøres kort i afsnit *6.6.1. Måder at håndtere komplexe data på*.

ODMG har arbejdet ud fra devisen om, at de gerne vil have fuld kompatibilitet med OMG-projektet. I mangel af andre standarder på området, og i erkendelse af at mange vigtige spillere på markedet har bundet sig til ODMG-standarden, har jeg valgt at plante fødderne solidt i denne standard. Således vil den objektorienterede model i det følgende blive beskrevet med ODMG standarden som udgangspunkt.

4.2. Den objektorienterede model

Nærværende afsnit skal give en fælles referenceramme, som senere kan benyttes dels til sammenligning med den relationelle model, dels som en nødvendig baggrund for at kunne evaluere Dates og Darwens syntesemodel.

4.2.1. Overordnet gennemgang

Som nævnt tidligere er grundideen i den objektorienterede model, at man kan integrere de objektorienterede programmeringssprogs muligheder med de muligheder, man forventer af en database. Det vil sige, at man i sin database ønsker faciliteter som dataindkapsling, arv og polymorfisme implementeret.

Objektmodellen specificerer hvilke regler, der gælder for de objekter, der kan dannes af en ODBMS.

- Grundelementerne er *objekter* og *literals* (konstanter). Hvert objekt har et entydigt identifikationsnummer, der genereres af ODBMS'en. En literal har intet ID-nummer
- Objekter og konstanter kan kategoriseres via deres typer. Alle elementer af en given type har fælles egenskaber og et ens antal definerede operationer, man kan udføre på dem. Et objekt omtales til tider som en forekomst af dets type.
- Objektets tilstand defineres ved de værdier den har i dets sæt af egenskaber. Disse egenskaber kan være objektets egne attributter eller relationsforhold.
- Objektets opførsel er defineret ved dets metoder. Metoder skal have en liste af in- og output parametre, hver med en specificeret type. Hver metode kan returnere et typebestemt resultat.

4.2.2. Datastrukturen

I dette afsnit vil jeg gennemgå den objektorienterede datastruktur.

4.2.2.1. Objektklasser

En objektklasse indeholder to deldefinitioner:

1. Den eksterne specifikation, kaldet **interfacen**.
   Denne definerer de karakteristika, man kan få øje på *eksternt*. Med andre ord: Metoderne, som man kan kalde i forbindelse med objektet samt de attributter, man har adgang til som bruger via OQL. En klasse kan godt have flere interfaces, der alle definerer forskellige typer adgange til samme objektklasse. Bemærk, at der ingen adgang er til et objekt, med mindre objektets klasse har fået defineret mindst ét interface.
2. Den interne specifikation, dvs. **objektklassens interne struktur**:
   Selve implementationen af objektklassens metoder (fx C++ koden) og objektets interne struktur i øvrigt.

Ved at anvende denne struktur, sikrer man sig, at attributter og metoder defineret i klassen automatisk er indkapslede. Via sit interface definerer man derefter hvilken adgang, man ønsker brugere kan få til klassens attributter og metoder.

Denne struktur betyder i virkeligheden, at tre-skema arkitekturen ikke længere overholdes: Objektklassens interface er nu både udtryk for det eksterne niveau samt for det konceptuelle. Objektklassens interne struktur beskriver nu det konceptuelle niveau, men indeholder programkode, der direkte udnytter kendskab til hvordan data fysisk er lagret. Det betyder, at man ikke længere umiddelbart vil kunne ændre en af de tre dele uden at påvirke de øvrige niveauer. Det er en meget væsentlig forskel.

Objekter er forekomster af en given objektklasse.

4.2.2.2. Literals

"Literals" eller konstanter består af tre hovedtyper:

1. Atomiske literals
2. "Collection" literals
3. Sammensatte (*structured*) literals

Atomiske literals er typisk numre og bogstaver. Det er ikke typer, man som programmør skaber, snarere nogle, der implicit er til stede. Det drejer sig i praksis om de typer, der er til rådighed i c og c++: signed og unsigned long, short, etc.

Collection literals er, som navnet antyder, typer som kan indeholde en samling objekter. Collection typerne er:

- set<t> (uordnet samling unikke objekter)
- bag<t> (uordnet samling objekter)
- list<t> (ordnet samling objekter)
- array<t> (en ordnet samling af objekter med andre finesser end en list-literal)
- dictionary<t,v> (uordnede unikke nøgleværdi-par (nøgle/fremmednøgle))

Strukturerede literals er som ved SQL, dato, tidsinterval, tid og timestamp. Dog er der her også en brugerdefineret type kaldet *structure<name>* . Finten er, at SQL typerne som fx `time` er sammensatte typer, der består af forskellige felter (`time` = (`hour` + `minute` + `second` + `milisecond`). *Structure<name>* giver muligheden for, at man selv opretter egne standardtyper.

ODMG standarden er “strongly typed” [Cattell 97]. Alt i en objektorienteret database har en type og hver operation kræver “typed operands”. Via dette kan man tjekke for type-kompabilitet og advare brugeren, hvis han er ved at udføre en funktion med inkompatible størrelser.

4.2.3. Dataintegriteten

I dette afsnit vil jeg gennemgå dataentydigheden samt referenceintegriteten. Afsnittet om arv er også placeret i dette afsnit, selv om man kan diskutere, om det i stedet bør høre under datastrukturafsnittet.

4.2.3.1. Dataentydighed

Alle objekter får af databasesystemet automatisk tildelt et objekt identifikationsnummer (OID), der dels fortæller hvilken klasse, objektet er en forekomst af, dels tildeler objektet et unikt nummer.
OID betyder dog ikke, at man i objektorienterede databaser kan undvære kandidatnøgler, da OID typisk vil være skjult for brugeren.

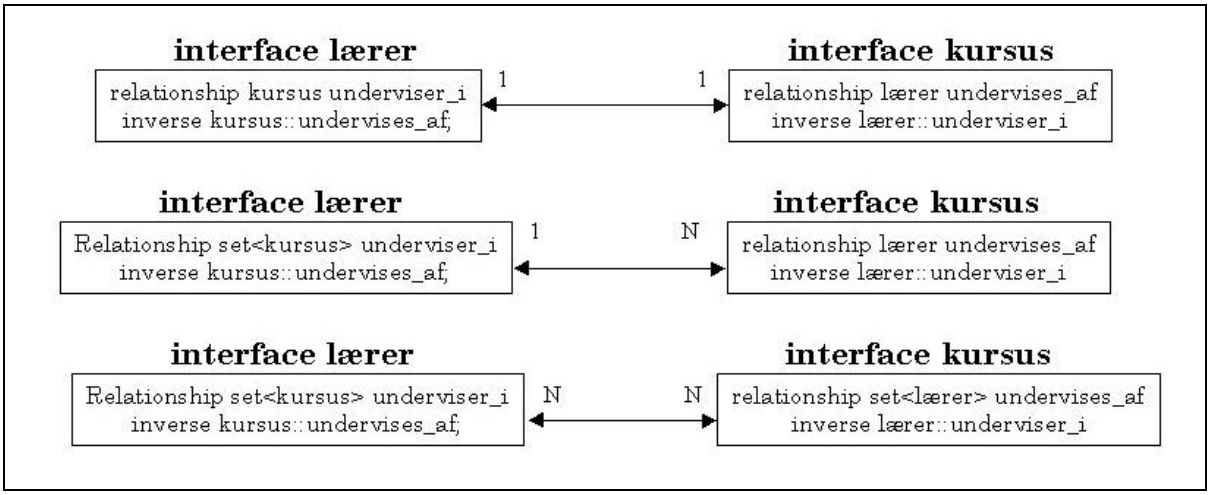
4.2.3.1. Referenceintegriteten

Et ægte relationsforhold etableres udelukkende via OID og består af et to-vejslink (et såkaldt “traversal path”). Det vil i praksis betyde, at man i objektklassen angiver en logisk sti til det objekt, man skal træde i forbindelse med.

Et relationsforhold defineres i objektklasserne på følgende måde [Cattell 94 & 97]:

- **En-til-en** relationsforhold mellem to objekttyper repræsenteres ved at tilføje en reference-attribut i hver type.
- **Mange-til-én** relationsforhold mellem to objekter A og B, ordnes ved at tilføje en reference attribut til A og et reference-sæt attribut til B
- **Mange-til-mange** relationsforhold mellem to objekttyper A og B ordnes ved at tilføje et reference-sæt attribut til hver af typerne. Et eksempel: En lærer underviser i et givet kursus.

De tre ovenstående muligheder er opsummeret i nedenstående illustration[7]:



ODMG understøtter i øjeblikket kun binære relationsforhold og ikke n'ary. Et relationsforhold defineres kun i en liste som “set”, “bag” osv. Man kan oprette n'ary relationsforhold ved at oprette et særskilt objekt til at varetage relationsforholdet ganske som i relationelle databaser. I kommende versioner af ODMG forventer man at ændre standarden, så programmøren kan bruge n'ary relationer direkte.

Ifølge ODMG standarden er det objekt DBMS'sen, der er "responsible for mantaining the referential integrity of relationships" [Cattell 97].

4.2.3.2. Arv

I objektorienterede databaser er arv defineret som følger:

1. “IS-A” relationen, som også kendes under betegnelserne supertype og subtype. Her er multipel arv tilladt. ISA relationer kan kun arves fra interfaces.
2. EXTENDS-relationen, som er arv fra én klasse til en anden klasse. Multipel arv er ikke tilladt.

Det er naturligvis tåbeligt, at man kan lade et klasseinterface arve et andet interface uden at dette automatisk forårsager at selve klasserne også semantisk arver hinanden. De to

[7] Omkring syntaksen: "relationship" betegner et relationsforhold, dernæst kommer objektklassen/typen ("kursus" eller "lærer", alternativt "set<objektklasse>") og sluttelig selve relationsattributtens navn, fx. "underviser_i" eller "undervises_af".

forskellige former for arv indfører en klodset og uhensigtsmæssig forskel, som ikke var til stede i den tidligere version af ODMG[8].

I forbindelse med arv kan man anvende polymorfisme, dvs. at en metode kan returnere forskellige resultater afhængig af hvilken klasse det objekt, metoden udføres på, tilhører[9].

4.2.3.3. Opsummering af dataintegriteten

Nedenstående opsummerer dataintegriteten i de objektorienterede databaser. Jeg sammenligner også her dataintegritetsaspekterne i de objektorienterede databaser med dem, der er nævnt i relationsdatabaseafsnittet:

Entydigheden er ikke problematisk, idet hvert enkelt objekt automatisk af DBMS'en bliver tildelt et ID-nummer. Ingen objekter kan derfor være ens. Det er dog vigtigt at notere sig, at OID ikke erstatter behovet for brugerdefinerede kandidatnøgler. OID feltet er typisk skjult for den almindelige bruger.

Referenceintegriteten er ikke længere en del af en relationel definition, som man kan definere via fremmednøgler, idet den ikke længere er værdibaseret. I stedet er der tale om et "to-vejs link" skabt via referencer til OID. DBMS sikrer referenceintegriteten mellem objekter, der deltager i et relationsforhold.

Brugerdefineret integritet er naturligvis stadig til stede - og er i og med at man har et helt programmeringssprog i ryggen - ganske udvidet i forhold til de relationelle databaser. Ikke at det i praksis har nogen betydning, idet de relationelle databaser kan indføre en kompleks brugerdefineret integritet via den applikation, som brugeren anvender til at indtaste oplysningerne ind i databasen. Og denne applikation har også et helt programmeringssprog i ryggen.

---

[8] Efter al sandsynlighed er denne kunstige skelnen indført for at opnå overensstemmelse med Java-syntaksen.

[9] Overloading af operatorer er også mulig. Det betyder at man ved at overloade en "+" (plus) funktion for eksempel kan lægge objekter sammen og få det resultat, man nu har defineret i sin overloading (man kan for eksempel definere at elementerne æble + pære = element frugtkurv).

4.2.4. Datahåndteringen

I dette afsnit vil jeg beskrive, hvordan datahåndteringen fungerer i objektorienterede databaser. Den kan deles op i to overordnede kategorier: Object Definition Language og Object Query Language. Mest interessant i forbindelse med opgavens formål er det at undersøge Object Query Language, hvorfor jeg har ofret flere kræfter på dette frem for på definitionssproget ODL. Der er naturligvis mange praktiske forskelle mellem SQL og OQL. Jeg har valgt at afgrænse afsnittet til kun at omhandle nogle enkelte principielle forskelle der skal benyttes senere i opgaven til sammenligningsformål.

4.2.4.1. Object Definition Language (ODL)

ODL er et sprog, der bruges til at definere og specificere de objekter, der skal indgå i en objektorienteret database.

ODL bruges uafhængigt af det programmeringssprog, som skal understøtte den aktuelle OOD. Derfor giver ODL mulighed for at specificere de enkelte objektklasser, deres attributter, relationsforhold samt metoder, herunder hvorvidt objektklassen har en superklasse eller ej.

ODL følges af et Object Interchange Format (OIF), som definerer hvordan ovenstående specifikationer samt de data, en given database indeholder, kan gemmes i en fil og senere benyttes til sammenligningsformål, udveksling af objekter mellem forskellige objektorienterede databaser, dokumentation m.m.

4.2.4.2. Object Query Language (OQL)

OQL er de objektorienterede databasers SQL. Der er gjort lidt fra ODMG's side gjort anstrengelser for, at der var en vis (men også kun en vis) overensstemmelse mellem OQL og SQL 92.

OQL er ikke baseret på den relationelle algebra, som det er tilfældet med SQL, men på den objektorienterede programmeringssprogs typebaserede logik. Den har ikke et teoretisk fundament, der bygger på mængdelæren, og definerer som sådan kun hvilke operationer, der er mulige at udføre på de enkelte typer.

Enkelte har arbejdet med at definere en objekt algebra. Således diskuterer en forsker allerede i 1993 [Alhajj 93] forskellene mellem at udøve forespørgsler på hhv. objektorienterede databaser og relationelle baser: “object algebra is more powerful because the relational algebra handles only atomic domains and only stored values can be retrieved (which is nothing more than a restriction that leads to an embedded query language and hence impedance mismatch) in contrast to object algebra, which handles stored as well as derived values and hence is computationally complete.”

4.2.4.3. Brug af OQL

OQL er et selvstændigt sprog, som man kan bruge til at finde og sortere objekter med, men sproget kan ikke uden videre bruges til at opdatere attributter eller objekter med. Her er et eksempel på, hvordan en simpel forespørgsel kan se ud:

```
select distinct x.age
from Persons x
where x.name = “Pat”
```

Ovenstående udvælger alderen for alle de personer, der hedder Pat og returnerer en collection konstant.

I objektorienterede databaser er det lidt mere kryptisk at håndtere situationer, hvor attributter ikke er udfyldt, end det er i SQL. I OOD er det nemlig slet ikke sikkert, at objektet i det hele taget er oprettet. Lad os antage at vi har en database med personer, deres adresser og telefonnumre. Personerne danner i dette eksempel et objekt og adresser et nyt, der konceptuelt er indeholdt i personobjektet. Vi har kun tre personer i databasen. Den ene bor i Søllerød, den anden i Valby og den tredje kender vi ikke adressen på.

En forespørgsel kan se sådan ud:

```
select p
from Personer p
where p.adresse.by = “Søllerød”
```

Ovenstående returnerer en bag, der indeholder personen, der bor i Søllerød.

En forespørgsel til:

```
select p.addresse.by
from Personer p
```

Ovenstående ville ifølge ODMG standarden [Cattell 97, side 89] resultere i en run-time fejl, idet personens byattribut ikke eksisterer. Adresseobjektet er ikke oprettet, og når vi adresserer attributten direkte via "`p.addresse.by`", så vil databasen melde fejl. ODMG anfører, at forespørgslen i stedet bør formuleres:

```
select p.addresse.by
from Personer p
where is_defined(p.addresse.by)
```

som ville returnere en bag med de to bynavne Søllerød og Valby. Man kan komme ud over dette problem ved at sørge for at altid oprettes et (tomt) adresseobjekt, når en person oprettes.

Men det virker som en klodset implementering af et forespørgselssprog. Her er det ikke på nogen måde lykkedes at komme ud over den addresseringskomplexitet, som C++ lider af.

Det er endvidere vigtigt at notere sig, at OQL kan få fat i alle attributter, som der er interfaces til, men ikke udføre `update`-statements andet end ved at kalde definerede metoder for et objekt. Man skal i så tilfælde kende den kommando, man skal bruge for eksempelvis at få lov til at ændre en værdi. Det er ikke særlig hensigtsmæssigt sammenlignet med SQL's universelle "`UPDATE`" statement.

4.2.4.4. Polymorfisme i OQL

En anden ændring i forhold til SQL er, at man kan benytte polymorfisme, idet der er en "late binding mechanism" - dvs. mulighed for at definere queries, der først udføres under kørslen. Et kald som

```
select p.aktiviteter
from Personer p
```

hvor `aktiviteter` er en metode, der bruges tre steder: Hvis personen er en student vil OQL kalde den aktivitetsmetode, der er defineret for subklassen `student`, hvis personen er

en ansat, så vil OQL kalde den aktivitetsmetode, der er defineret for subklassen `Ansat` og hvis personen blot er en person defineret i klassen `Person`, vil OQL kalde den metode `aktiviteter`, der er defineret i klassen `Person`. Denne logik fungerer takket være Objekt Id'en, der bl.a. hjælper med at holde styr på objektets klassetilhørsforhold. I OQL kan man også kalde objekter via deres OID.

4.2.5. Objektorienteret database design

Der er mange hensyn at tage når en objektorienteret database skal designes. Vi husker, at Codd mente at den relationelle model gav mulighed for at repræsentere data direkte som de optrådte i verden, uden at tilføje specielle strukturer for maskinens skyld. Objektmodellen forbliver i dette paradigme, men ud fra en fuldstændig anderledes synsvinkel. Allerede i Cattells introduktion til objektorienterede databaser, får vi at vide, at normalformerne “may be useful for a database administrator designing a relational schema; for the purposes of this book, you will not need to understand these formalisms - a little common sense will do” [Catell 94]. Så let tror jeg ikke, at nogen relationsdatabasebog ville springe over normalformerne. Men det giver et godt fingerpeg om, at de objektorienterede databaser opfatter verden ganske anderledes. Skal man tro de objektorienterede folk, så holder Dates idé om at alle databaser, relationelle eller ej, bliver nødt til at arbejde med funktionelle afhængigheder og normalformer, ikke.

I det følgende vil jeg redegøre for det objektorienterede databasedesign og løbende sammenligne det med den relationelle. Hermed vil jeg kunne trække nogle klare linier op mellem disse to forskellige designmåder og i afsnittet *5. Design issues*, vise forskellene i praksis.

4.2.5.1. Standardens betydning for designet

Opgaven med at beskrive det objektorienterede databasedesign kompliceres yderligere af at ODMG 2 standarden forsøger at rumme kompatibilitet med C++, Smalltalk og Java. Disse sprogs forståelse af design og objekter er indbyrdes uforenelige. For eksempel hedder det i Smalltalk-afsnittet, at “Since Smalltalk has no distinct notion of literal objects, both ODMG objects and ODMG literals may be implemented by the same Smalltalk classes.” [Catell 97] Endvidere kan man læse, at relationsforhold ikke er noget, som er “directly supported by Smalltalk, and must be implemented by Smalltalk methods that support a standard protocol.”[Catell 97] At det overhovedet kan lade sig gøre at lave en

standard, der kan rumme så mange undtagelser på grundlæggende begreber, kan kun lade sig gøre hvis man klipper en hæl, hugger en tå og i høj grad nedtoner alle formalistiske krav omkring selve databasedesignet i forhold til selve databasens programmerbare funktionalitet. Det ligner lidt et motto, der hedder: "Så længe det *ligner*, så går det nok alt sammen." Det er naturligvis med til at udvande ODMG-standarden. Som ODMG standarden siger: "Thus some of the constructs that appear here as part of the Object Model may be modified slightly by the binding to a particular programming language."[Cattell 97]

På baggrund af ovenstående må vi derfor konstatere, at formålet med ODMG standarden er at sikre portabilitet *inden* for de enkelte programmeringssprog. Dvs. at det ikke er meningen, at man skal kunne flytte en database fra Java og over til Smalltalk. Dette står imidlertid ikke tydeligt at læse i ODMG standarden.

4.2.5.2. Datastruktur kontra funktionalitet

En anden stor forskel er, at designerens opgave i den relationelle model er at strukturere data, således at anvendelsen af databasen ikke senere skal være begrænset af databasedesignet. Databasedesignet skal være let at ajourføre og udvide, hvis nye attributter eller relationer senere skal indføjes i databasen.

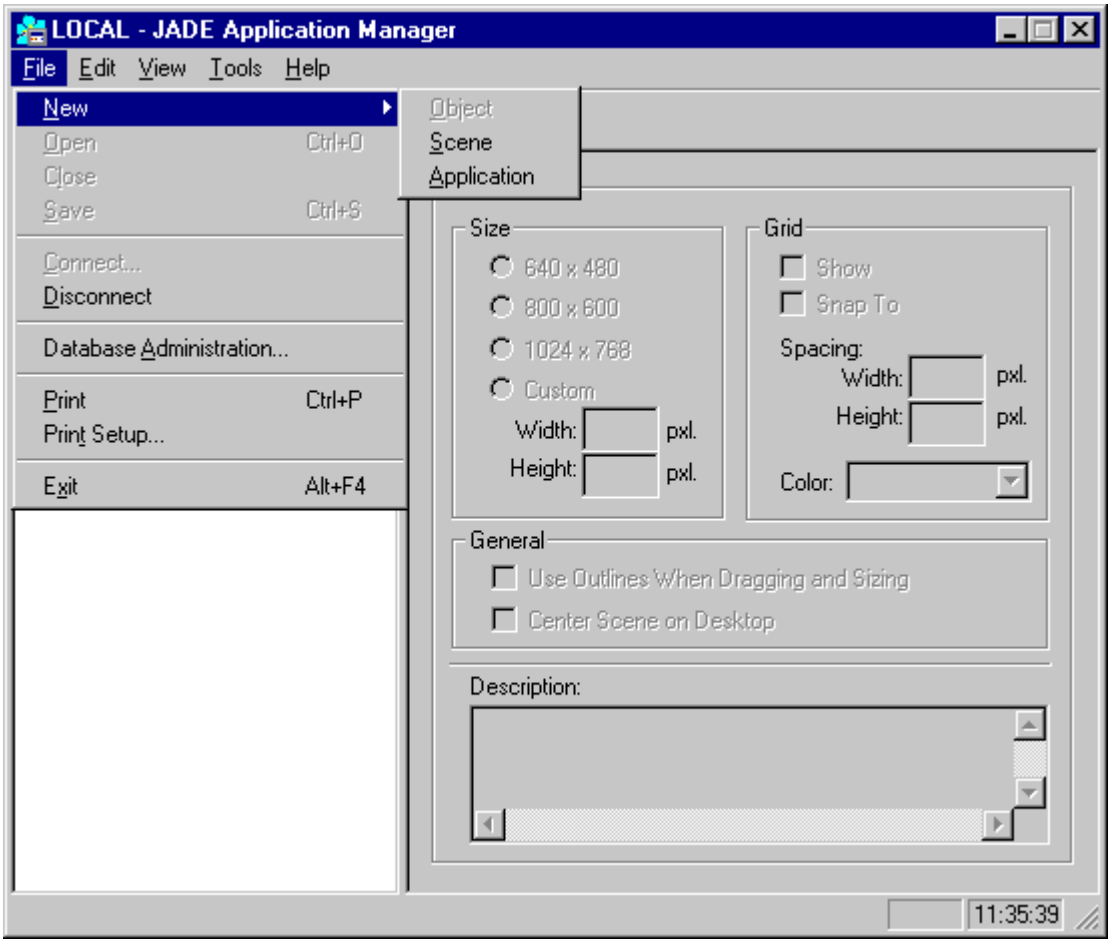
Objektorienterede designere tænker anderledes: dataernes interne struktur er skjult for brugere, så de ikke kan spørge direkte på attributter. Den måde, man er i stand til at udøve queries og opdateringer på, kan styres fuldstændig diktatorisk via interfacen til de enkelte klasser. Designeren skal derfor ikke lægge så mange kræfter i datastrukturen som i applikationernes funktionalitet. Fokus flyttes fra selve datastrukturen til det, programmørens kerneområde handler om: Funktionalitet. Hvad skal vi kunne spørge databasen om? Hvad skal databasen returnere, hvis vi spørger om dette eller hint? Hvilken type resultat skal brugeren have: En `bag`, et `set` eller noget helt tredje? Det ville for en relationsdatabasemand svare til, at en objektorienteret database programmør skulle opfinde den dybe tallerken forfra hver gang en ny database skal designes. Det virker ikke særlig smart, at man skal koncentrere sig om hvad databasen skal spørges om, i stedet for at koncentrere sig om, at selve datastrukturerne er så holdbare som muligt: dvs. at man senere skal kunne tilføje nye attributter, udvide databasen med nye tabeller m.m.

Et helt andet aspekt ved designet er, at objektorienterede databaser håndtere andre typer objekter end dem relationsdatabaser er i stand til. Eksempelvis kan man med en objektorienteret database gemme jpg-filer (billedfiler) som okteter (`octet-stream`) og programmere en forespørgsel ud fra sin viden om jpg-fil formatet: Find i databasen billeder, der indeholder en hvid kugle.
Den hvide kugle ville svare til en bestemt bit-struktur, som forholdsvis nemt ville kunne identificeres i oktetstrømmen i tuplen via en for klassen defineret metode.

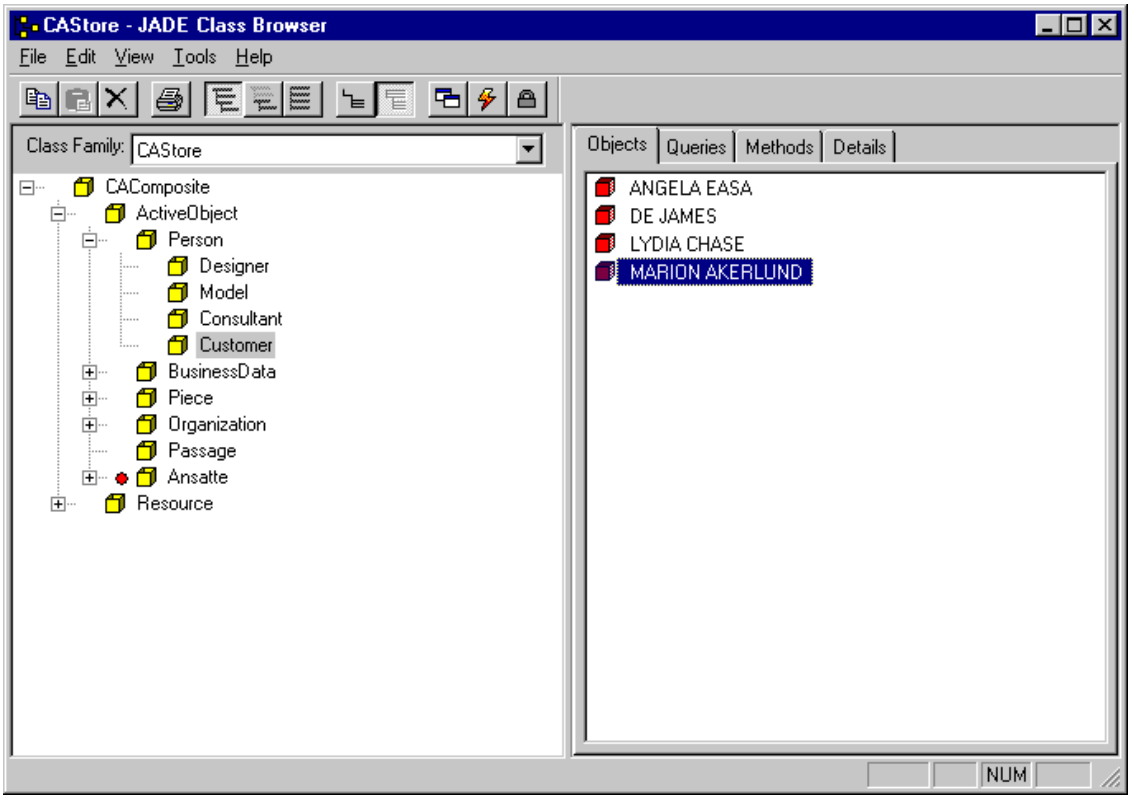
Objektorienterede databaser er derfor gode til veldefinerede, specialiserede formål, der i databasedesignet vil indeholde felter med typer, som ikke umiddelbart kan behandles fornuftigt af og i relationsdatabaser.

4.2.5.3. Design i ikke ODMG-kompatible objektorienterede databaser

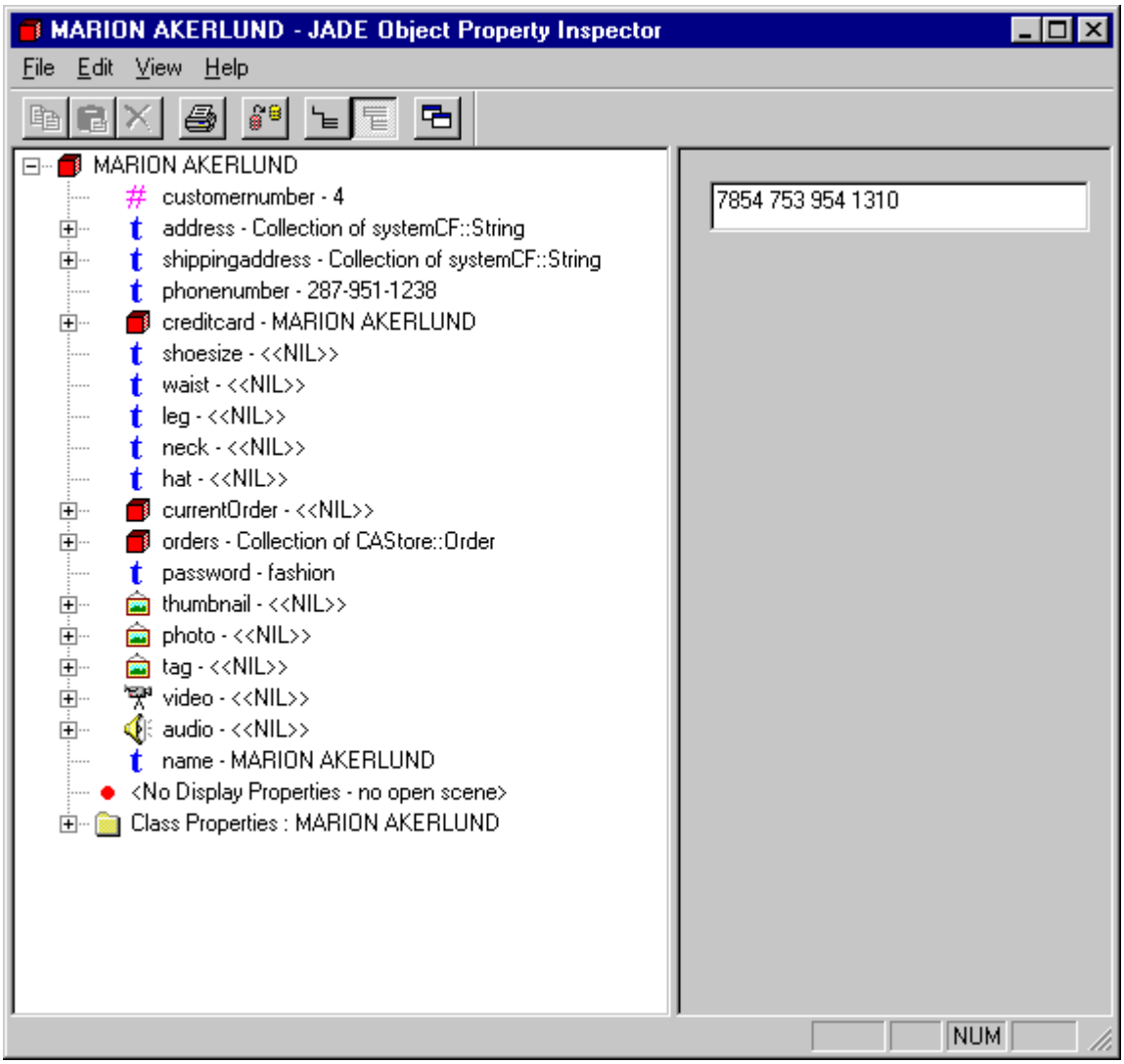
At fokus er flyttet fra databasestruktur til applikation er meget tydeligt at se i en database som *Jasmine*, som efter eget udsagn (men også efter egen vurdering) er den første større og professionelle rene objektorienterede database med et fornuftigt grafisk interface. Her er der tale om ren “træk og slip” design. Her er applikationstanken ført til sin yderste konsekvens. Når man vil oprette en ny database, så spørger *Jasmine* ikke om databasens navn, men om applikationens:



Hver applikation kan så have et antal "scener", som kan trigges når de korrekte forudsætninger er til stede. Selve databasen består af objektklasser:



Ovenstående viser en databasestruktur med en specialisering af Person i fire nye kategorier, hhv. designer, model, consultant og customer. Til venstre på billedet kan man så se de objekter, der p.t. er oprettet i databasen under `customer`. Hvert objekt indeholder forskellige attributter:



"t" står for tekstattributter, "#" for nummerattributter og som man ser er der også markeringer for billeder, lyd og film. Herudover er der nogle æsker, der markerer at der er tale om indlejrede objekter, der igen kan indeholde egne attributter.

Al design og definition af databasen i *Jasmine* foregår i forbindelse med ovenstående interfaces. Man kan benytte Java som OQL interface til Jasmine-databasen, når først den er designet. Men Jasmine har naturligvis også sit eget query interface.

Der er ingen tvivl om, at ovenstående typer databaser vil blive mere og mere populære. Det er et nogenlunde enkelt interface og man slipper for mange af de problemer, som man møder, når man skal kompilere ODMG databaser. I ODMG-databaser kan man nemlig nemt komme i tvivl om, hvorvidt det er det ekstra bibliotek, som man benytter til at gøre sit programmeringssprog databaseegnet, der er noget galt med, eller om det er ens C++ kode. Fejlfinding kan blive noget så grusomt besværligt, hvad henvendelser til nyhedsgrupper også tyder på.

4.2.6. Opsummering

De foregående afsnit om de objektorienterede databaser har også indeholdt en positionering af den objektorienterede model i forhold til den relationelle. De danner også grundlaget for, at jeg nu kan gå i gang med mere dybtgående at kigge på design af relationelle kontra objektorienterede databaser.

Men herudover giver de foregående afsnit mulighed for, at man kan sætte sig ind i den løsning Darwen og Date foreslår til at parre de objektorienterede databaser med de relationelle. Det vil vi se på i afsnit *6. Darwen og Dates udvidede relationelle model.*

Undervejs i gennemgangen af de enkelte afsnit har jeg trukket de ting frem, som man især skal lægge mærke til, når man taler om objektorienterede databaser.

Datastrukturelt skal man bide mærke i at de objektorienterede databaser giver køb på tre-skema arkitekturen.

Dataintegritetsmæssigt, at man arbejder med objekt ID-numre for hvert objekt, samt at relationsforhold fungerer via et to-vejslink. Endvidere at det er muligt at benytte arv.

Datahåndteringsmæssigt skal man lægge mærke til, at vi har bevæget os væk fra algebraen og at polymorfisme er mulig. Herudover at man skal definere metoder for alle `update`-statements, mens OQL giver adgang til alle de attributter, der er interfaces til.

Ovenstående egenskaber har store konsekvenser for de **designmæssige** aspekter, især, at man i objektorienterede databaser fokuserer på databasens funktionalitet mere end på datastrukturen.

5. Design issues

I de to foregående afsnit har jeg beskrevet de ting, der i forhold til min problemformulering var relevante for henholdsvis de relationelle og de objektorienterede databaser. I afsnittet omkring de objektorienterede databaser har jeg visse steder sammenlignet de to typer databaser og positioneret dem i forhold til hinanden.

I dette afsnit eksemplificeres designangrebsvinklerne, således at vi kan finde ud af, om der er stor forskel på design i den relationelle og den objektorienterede databasemodel og i så fald, hvordan forskellene kommer til udtryk.

Den bedste måde at illustrere designforskellene mellem et relationelt og et objektorienteret databasedesign vil være at tage et konkret og relativt simpelt eksempel.
Lad os benytte følgende:

En undervisningsinstitution udbyder hvert semester et antal kurser. Til institutionen er der knyttet et antal studerende samt et antal lærere. Lærere kan enten være lektorer eller undervisningsassistenter. Undervisningsassistenter kan enten være færdiguddannede eller – i nogle tilfælde – stadig være studerende. Institutionen ønsker at lave en database, der kan:

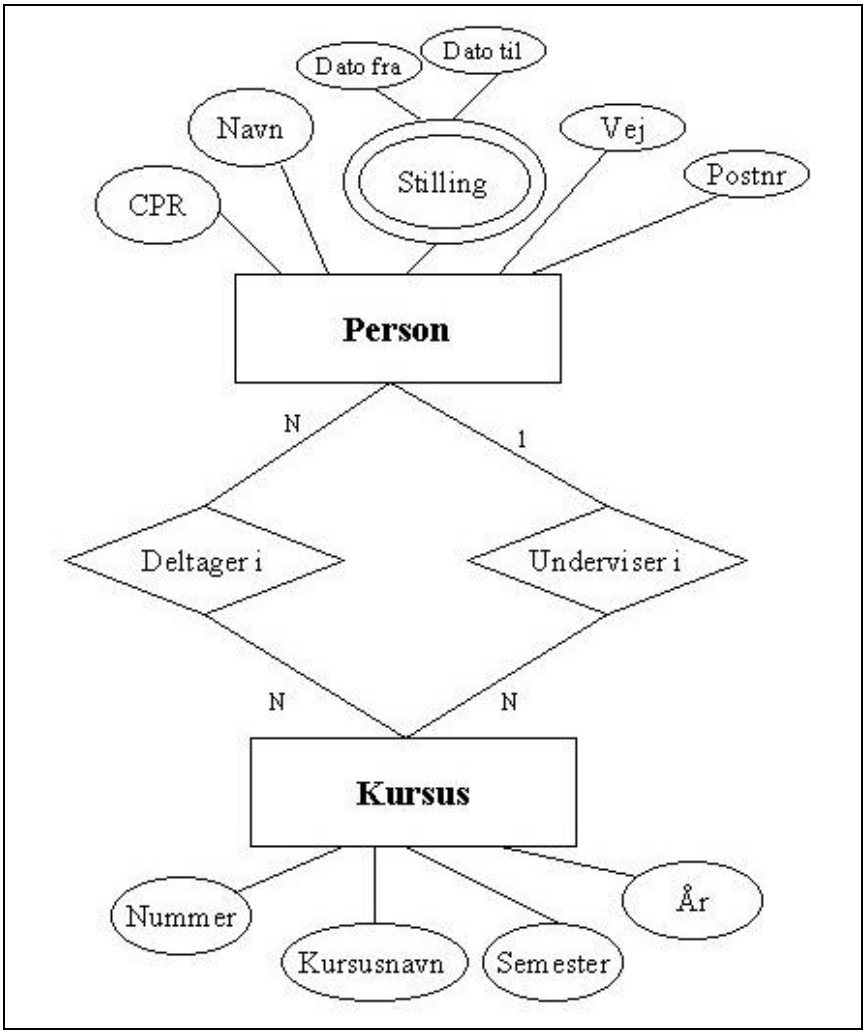
- registrere lærere og studerende i et kursus.
- registrere såvel lærernes som de studerendes adresser

Disse oplysninger skal bruges til at

- udskrive en liste over kursets lærer og deltagere
- lave en statistik over, hvor mange af lærerne, der i et givet semester var a. lektorer, b. undervisningsassistenter og færdiguddannede eller c. undervisningsassistenter og fortsat studerende.

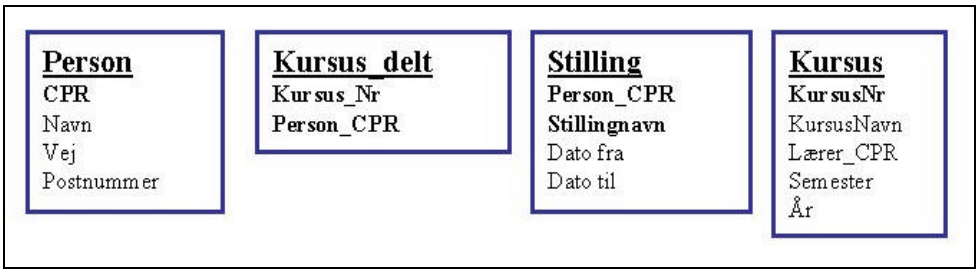
For at forenkle problemstillingen har jeg i det følgende valgt at definere adressen så den kun består af to felter: Vej samt Postnummer. Endvidere ser vi bort fra den mulighed, at flere lærere kan undervise på det samme kursus.

Ovenstående specifikationer giver følgende ER-model:



5.1. Design i en relationel database

Omsat til en relationel database, kunne designet se således ud:



Ovenstående database overholder alle normaliseringsformer.

I stillingsdatabasen oprettes følgende stillingsrecords:

- lektor
- undervisningsassistent
- studerende

Fordelen ved ovenstående design er, at den samme person kan have flere stillinger, for eksempel som studerende eller undervisningsassistent. Via datoangivelse for stillingen, bibeholdes oplysningen om vedkommendes titler på et givet tidspunkt.

Vi ser, at en relationel database først og fremmest har sørget for at dekomponere alle oplysninger via normaliseringer, således at databasen er nem at udvide. Skal der for eksempel senere oprettes et professorat, ville det ikke skabe problemer i forhold til databasen. Man kan blot indtaste den nye stilling i stillingsrelationen.

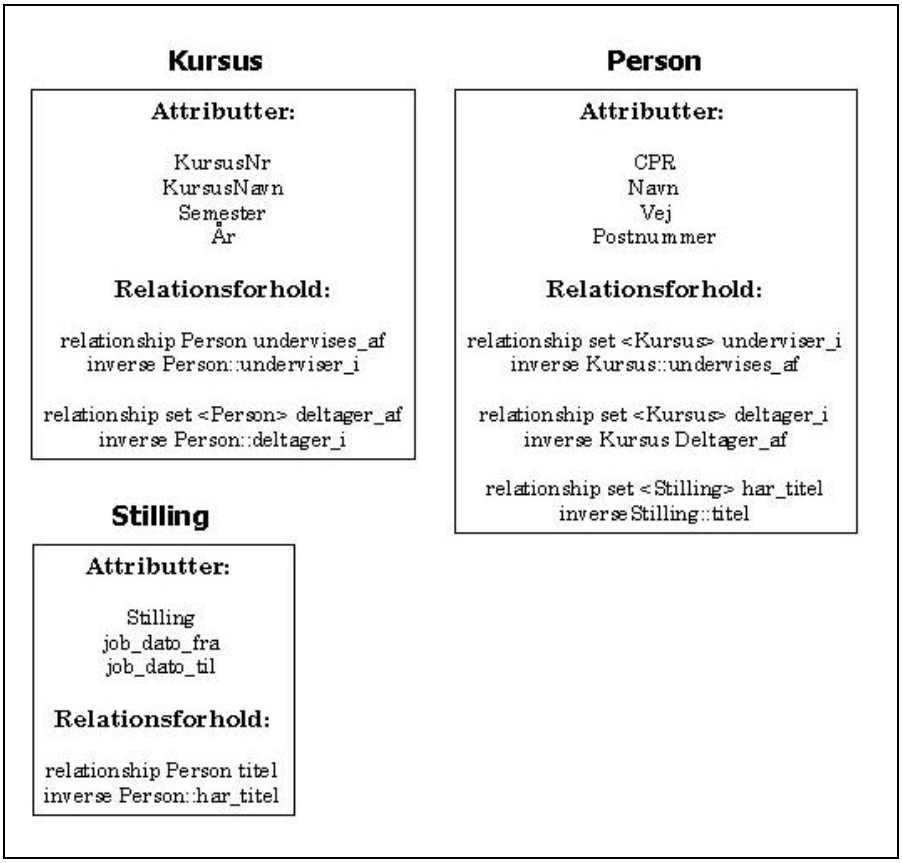
5.2. Design i en objektorienteret database

En objektorienteret database ville designe ud fra nogle andre principper. Vi skal huske, at i en OOD ville designerens baggrund og forståelse af opgaven spille meget mere ind end det er tilfældet i den relationelle database, hvor normaliseringsformerne hjælper designeren med at finde ud af, hvordan databasen skal skrues sammen.

Læser vi de objektorienterede udviklingsværktøjers egen reklame for hvilke fordele der er ved at tænke objektorienteret, får vi følgende at vide [Fra Jade – et australsk objektorienteret databaseprodukt http://www.jade.co.nz./ oovsrel.htm]:
"This advantage is centered on the fact that object orientation offers a natural representation of data - one that mirrors the user's view of their world. Objects generally correspond to things, people, roles and/or relationships that are a normal part of business life. This makes it much simpler to model a very complex business in a way that is understandable to anyone, not just the computer experts."[10]

En hensigtsmæssig objektorienteret løsning kunne se således ud:

[10] Codd ville sikkert kunne nikke genkendende til disse ord …



Hvert objekt indeholder attributter, der kun har noget med selve objekterne at gøre. Herudover er de relevante relationsforhold defineret. Der er ikke brug for at splitte deltagerne ud i et objekt for sig, idet disse er indeholdt i relationsforholdet "`deltager_i`" og "`deltager_af`".

Når man skulle præsentere designet for en bruger, ville designeren naturligvis ikke vise relationsforholdsspecifikationerne, men kun objektklassernes attributter med nogle pile, der anførte hvordan relationsforholdene ville være. I stedet for den relationelle model, med fire relationer, har vi i den objektorienterede database kun tre klasser.

5.3. Fordele og ulemper ved de to designmodeller

I Jades salgsmateriale [jf. http://www.jade.co.nz./oovsrel.htm] hævdes der, at det reelt øger udviklingshastigheden, at der er større overensstemmelse mellem virkelighedens objekter og databasens. Det tror jeg er noget vrøvl. Som diskuteret under OOD afsnittet har det nogle følger, at de objektorienterede databaser ikke har samme datahåndteringsfaciliteter som relationelle databaser. Det betyder, at den objektorienterede udvikler skal programmere mange af de faciliteter, som allerede ligger indbygget i en relationel database, selv banale update-kommandoer.

Heri ligger der mange fejlmuligheder. Dels kan man glemme at implementere nogle af de features, som databasen skal kunne håndtere, dels kan selve metoderne blive fejlprogrammeret.

Objektorienterede databaser kan med en vis ret hævde, at også relationelle databaser skal have programmeret særlige metoder (dem, man lægger i selve databaseintefacen), men i objektorienterede databaser drejer det sig om at programmere en hel række banale funktioner for overhovedet at være i stand til at tildele nye værdier til attributter. Det gælder dog ikke de ikke-ODMG kompatible objektorienterede databaser.

Laver man en fejl i nogle af de basale funktioner, vil det være dyrt for en virksomhed at rette op på fejlen. For større og meget mere komplexe systemer end den ovenfor beskrevne database ville der være meget stor sandsynlighed for, at udvikleren fejlede undervejs og at udviklingen derfor ville strække sig over længere tid end ved en relationel løsning.

En anden ting, som objektorienterede databaser synes at overse er, at selve datastrukturen ikke påvirker den måde, en bruger ser data på. Der bliver altid lavet et interface til den underliggende fysiske database og derfor er den måde, data i praksis er struktureret på af mindre betydning for en bruger. At data således logisk er repræsenteret i færre objekter end der ville være relationer i en relationel database, betyder derfor meget lidt i den store sammenhæng.

Det, man til gengæld kan se med objektorienterede løsninger er et paradigmeskift hen mod at anskue databasen – ikke som en database – men som en applikation. Med relationelle løsninger designer man først en underliggende sql-database, og bagefter programmerer man en front-end til brugeren. Denne front-end kan laves i alle programmeringssprog, der tillader nestede sql-kald (dvs. stort set alle).
I objektorienteret design udvikler man databasen som en applikation, hvis funktionalitet kan udvides, efterhånden som man har brug for det. Hvor man tidligere har adskilt data og applikation, så sammensmeltes disse komponenter i det objektorienterede design.

5.4. Konklusion på design issues

At data og applikation sammensmeltes i objektorienterede databaser er logisk nok, fordi de data, de hidtil har skullet håndtere hovedsageligt har været komplexe datastørrelser, fx. video, audio, CAD/CAM tegninger m.fl. En tilgang til disse data implicerer altid et kald til en applikation (fx. et video-fremviser). Søgning i disse data implicerer også søgninger i komplekse oktet-strukturer. Andre eksempler for brugen af OOD har for eksempel været, når man løbende skulle gemme forskellige versioner af den samme database.

Det objektorienterede design virker dog umoden i forhold til den stringente relationelle approach. Fejlmulighederne er flere, fordi man ikke længere har en matematisk teori, der ligger underneden og beskriver hvordan datahåndteringen skal foregå. I stedet skal man til at stole på en programmørs evne til at få det hele til at fungere. Er der tale om et større system, hvor mange programmører skal samarbejde om at få delkomponenter til at arbejde sammen, kan man hurtigt løbe ind i problemer.

Man kan dog ikke udelukke, at objekt-orienterede databaser, der arbejder i samme retning som produkter som *Jasmine* vil gøre objektorienterende databaser lige så attraktive som relationelle[11]. I disse produkter er det at tilføje ny funktionalitet ikke så vanskeligt, fordi der er en skal indbygget i databasen, der tilbyder de basale tilgange. Resten foregår via en visuel træk og slip funktionalitet. Selv om *Jasmine* ikke implementerer ODMG standarden fuldt ud, er der ikke tale om helt proprietær løsning, idet *Jasmine* understøtter OQL via Java.

Vi kan i alt fald ud fra ovenstående diskussion uden tøven svare positivt på det indledende spørgsmål om hvorvidt der er en markant divergens mellem den relationelle og den objektorienterede opfattelsesmodel allerede i databaseanalytikerens designfase.

---

[11] Tendensen kan også ses inden for programmeringssprog, hvor Borlands Delphi, der giver et grafisk interface til Pascal, har vundet stor udbredelse. Det samme gælder C++ Builderen, der på samme måde giver en grafisk brugerflade til C++ programmeringen. Begge programmers styrke er, at de er i stand til at øge udviklingshastigheden ved at give programmøren mulighed for at lade programmet selv generere en stor del af den nødvendige kode.

6. Darwen og Dates udvidede relationelle model

I dette afsnit vil jeg først give et overblik over Darwen og Dates udvidede relationelle model, således som de præsenterer den i deres "The Third Manifesto". Derefter vil jeg vælge de væsentligste ting i modellen ud, der kan bidrage til at opklare, om deres model kan rumme de samme faciliteter som de objektorienterede databaser.

De faciliteter, jeg har valgt at fokusere på, er: Arv, polymorfisme, understøttelse af komplexe datatyper, dataindkapsling, versionering samt CAD/CAM relaterede opgaver. Dette er i god overensstemmelse med den definition af hvad man kan forvente af en objektorienteret database, jf. den definition, der blev givet i afsnit *4. Objektorienterede databaser*.

6.1. Hvad indeholder "The Third Manifesto"?

Såvel Darwen og Date anerkender manglerne ved de aktuelle implementeringer af relationelle databaser. De foreslår derfor nogle ændringer, hvoraf enkelte, men langt fra alle, ser ud til at blive inkluderet i det forslag, som er godt på vej til at blive den nye ANSI godkendte SQL version 3[12].

Hensigten med disse ændringer er at få det bedste ud af begge verdener: "firmly rooted in the Relational Model of Data" skal den supportere "certain features that have been much discussed in more recent times, including some that are commonly regarded as aspects of Object Orientation" [Darwen & Date 95].

De features, som nævnes, er [Date 95]:

- Objekter og objektklasser, deres tilhørende constructor funktioner samt andre adgangs- og vedligeholdelses funktioner. Det betyder også, at der skal være en klar adskillelse mellem objektklasser, objekter og collections (sets, arrays og bags)
- Klassehierarkier med arv og polymorfisme med den dertil hørende run-time binding.

[12] Det er relevant her at nævne, at Hugh Darwen er medlem af ISOs SQL-komite.

Date [Date 95] understreger dog, at man skal passe på ikke at indkapsle data for meget: "There will always be a need to access data in ways that were not foreseen when the database was originally created."

Måden, Date og Darwen ønsker at implementere disse ændringer på, er ved at benytte sig af domænedefinitionen: "Domains encapsulate, relations don't," som Date [Date 95] skriver. Det ligger i Dates idé at domænet dermed svarer til en klassedefinition i objektorienteret forstand. I et domæne bør man derfor også kunne definere en constructor og deconstructor samt forskellige metoder, som kan udøves på attributter, der tilhører et bestemt domæne. Dog skal metoderne ikke defineres i selve domænet, da han synes det er noget rod at blande den domænedefinition, man validerer sine data ud fra, med den metodedefinition, der skal være med til at ændre på attributters værdier.

Manifestet er delt op på følgende måde:

1. "Relational Model Prescriptions"
   Dem er der 26 af, disse definerer begreber som domæne, relationer m.m.
2. "Relational Model Postscriptions"
   Dem er der 10 af, som definerer alt det, som relationer i hvert fald *ikke* har, som for eksempel, at der *ikke* er en speciel attributorden i en given relation.
3. "Other Orthogonal Prescriptions[13]"
   Dem er der 7 af, som især beskæftiger sig med arv og transaktioner
4. "Other Orthogonal Postscriptions"
   som er 4 regler, som definerer det, som *ikke* må misforstås, fx at relationer ikke er lig med objekter, idet attributværdierne ikke kan indkapsles
5. "Relational Model Very Strong Suggestions"
   som indeholder 9 forskelligartede anbefalinger
6. "Other Orthogonal Very Strong Suggestions"
   som indeholder 4 forskellige anbefalinger.

Alt i alt er der 60 forskellige regler og anbefalinger.

Omsat i praksis betyder Darwens og Dates forslag følgende [Darwen & Date 95]:

---

[13] Darwen og Date vil ikke så gerne bruge udtrykket "Object Oriented", så i stedet kalder de det for "Other Orthogonal"

Domæner	Værdityperne er indkapslede og har altid en type (char, num etc.). Der er altid defineret en metode, der gør det muligt at udlede en værdi for en forekomst af et domænet. Arv (inkl. multipel arv) er tilladt mellem domæner. Et domæne kan have en "constructor[14]"-metode defineret.
Metoder	Defineres for sig, dvs. ikke i selve domænedefinitionen. Men der er altid defineret hvilket domæne resultatet tilhører.
Attributter	Disse tilhører altid et domæne. Manglende værdier er ikke længere tilladte. Fx er en attribut "beløb", der ikke er udfyldt, ikke mulig. I stedet ville den være lig med fx "?". Det betyder at domænet for attribut "beløb" ikke kun er mængden af positive talværdier, men også indbefatter værdien "?" Attributters værdier er ikke indkapslede.

Med ovenstående overblik *in mente* vil jeg i det følgende forsøge at finde ud af, om det så også betyder, at den objektorienterede databasemodel kan overflødiggøres.

6.2. Arv

Arv er en så grundlæggende egenskab ved de objektorienterede databaser og de objektorienterede programmeringssprog, at det er oplagt at undersøge hvordan Date & Darwen har tænkt sig at få denne egenskab ind i den relationelle model.

Men her bliver vi skuffet: Darwen og Date er ikke helt afklaret med hensyn til hvordan arv i praksis skal implementeres. Selv begrunder Darwen og Date deres tøven med at der ikke findes en universel anerkendt arvmodel [Kalman 94]. Det er en interessant påstand set i lyset af at arv har været brugt i objektorienterede systemer siden Simula indførte begrebet (der dengang dog kaldtes *concatenation*) i 1960'erne. Baggrunden for denne holdning skal nok ses i lyset af, at der ikke findes én veldefineret semantisk model af hvordan arv bør bruges. Der er medgået en hel del forskning inden for feltet[15]. Grundlæggende kan man sige, at der er fire abstraktionsprincipper, man kan bruge arv til [Taivalsaari 96]:

1. *Klassifikation*, som nedtoner forskellene, men understreger fælles egenskaber

[14] underforstået som fx i C++

[15] Se fx artiklen i ACM nævnt i litteraturlisten [Tivalsaari 96]

2. *Generalisering*, som nedtoner de fælles egenskaber og understreger forskellene
3. *Aggregering*, som nedtoner detaljer ved de enkelte komponenter til fordel for at få et overblik over komponenternes relationer
4. *Gruppering*, som nedtoner detaljer ved en gruppe objekter for at understrege deres gruppetilhørsforhold

Ovenstående er illustreret her:

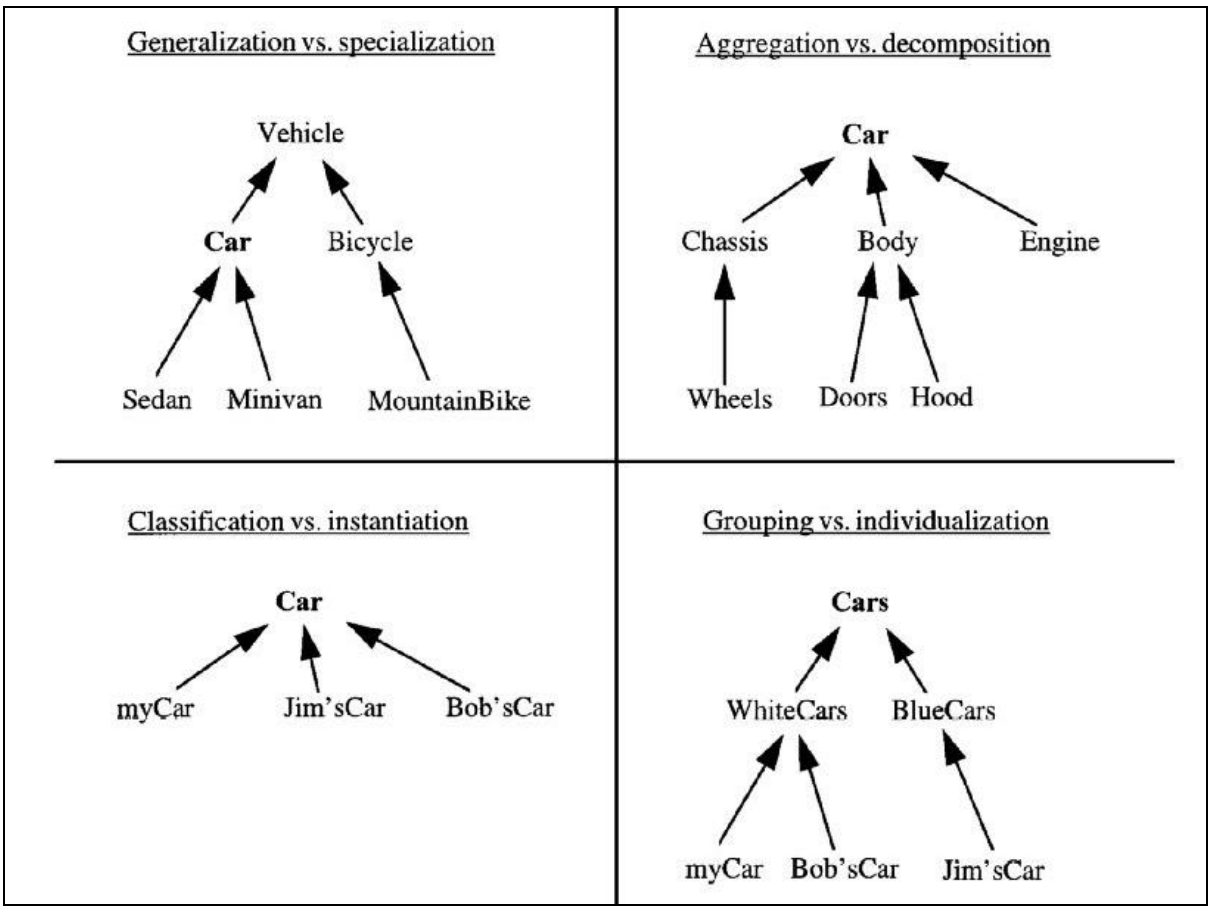


Illustration fra [Taivalsaari 96]

Problemet for nogle er, at arv bliver brugt i anden forbindelse end til konceptuelt at opfylde ovenstående fire opgaver. Enkelte, som Wegner og Zdonik har forsøgt at opstille regler for, hvornår man kan sige, at arv benyttes fornuftigt. Men her taler vi om det konceptuelle, semantiske niveau. I praksis er der således ikke nogen tvivl om, *hvordan* arv virker. Således har fx Stoustrup og Ellis meget fyldestgørende gjort rede for teknikker for arv (herunder multipel arv) i C++ miljøer (se fx kap. 10 i Ellis & Stroustrup 90).

Der er selvfølgelig praktiske forskelle på den måde fx C++ og Smalltalk håndterer arv på. Disse er dog først og fremmest begrundede i de enkelte programmeringssprogs karakter og

typeforståelse. Eller, som Backlawski og Indurkhya skriver [Backlawski og Indurkhya 94]: “This whole discussion can be summed up as follows. What a programming language provides is a set of mechanisms. While this mechanisms certainly restrict what one can do in that language and what views of inheritance can be implemented there, they do not by themselves validate some view of inheritance or other. Classes, specializations, generalizations and inheritance are only concepts, and like other concepts they do not have a universal objective meaning but depend heavily on the objects involved and the kind of operations allowed on these objects. This implies that how inheritance is to be incorporated in a specific system is up to the designers of the system, and it constitutes a policy decision that must be implemented with the available mechanisms.” På mig virker det utroværdigt at hævde, at der ikke findes veldefinerede former for arv. Ikke mindst fordi man netop betragter arv og polymorfisme som værende de væsentligste årsager til de objektorienterede programmeringssprogs store succes.

Det virker under alle omstændigheder som om Darwen og Date har vanskeligheder med at få indpasset en acceptabel arvmodel i relationelle databaser. De hævder i deres manifest dateret i 1995 [Darwen og Date 95] at de har en skitse til inkludering af arv liggende klar, men her, tre år senere, er denne skitse så vidt vides endnu ikke blevet offentliggjort. I afsnittet om polymorfisme kommer jeg tilbage til hvilke problemer man kunne forestille sig at løbe ind i, hvis man definerede en arvstruktur i deres foreslåede syntesemodel.

6.3. Array attributter og 1NF

En anden konsekvens af Date & Darwens forslag er, at man kan have et array som en attribut uden at dette bryder med første normalform. Det kan man fordi man kan definere metoder, der tager arrays som parametre og returnerer en værdi. Disse metoder er den eneste indgang til attributværdien. Date beskriver det således [Kalman 94]:

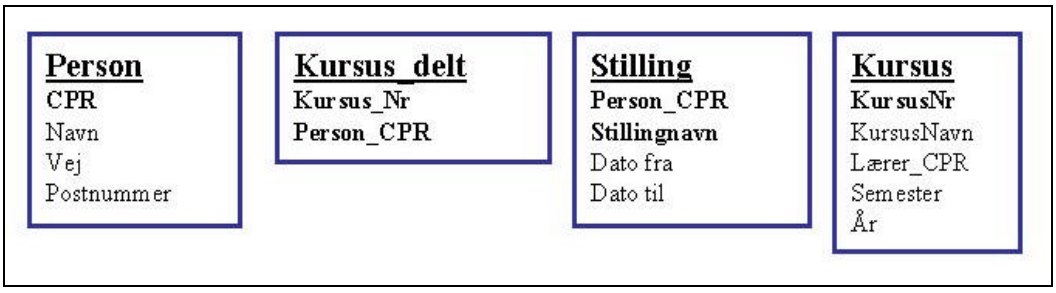
“These operators will probably include subscripting for picking out elements of the array, and so on. With a subscripting operation, I could write a query in SQL that says something like “select* from this table where A[3] = 10.” That's no different from saying “select * from this table where some substring of a string is equal to ‘abc’.” If I can do the string operation, I can do an array operation too. They're exactly analogous. The problem, of course, is that once you recognize that you can have arbitrarily complex values in row and column cells, the database design problem becomes much more difficult. Now you have

multiple ways of representing things logically. You have far more freedom, but as far as I know there's not much in the way of good design methodologies. I'm certainly not saying we've solved all the problems. I'm simply saying that if you want object-oriented constructs, this is how to do it.” [Kalman 94]. Jeg vender tilbage til Darwen og Dates forslag og dets konsekvenser lidt senere. Lad os først se, hvordan denne nye feature og de andre, som Date og Darwen foreslår, kan bruges i det praktiske databasedesign.

6.4. Databasedesignet

Hvordan ville den database, vi før designede, se ud, hvis vi implementerede den i en version som den, Darwen og Date foreslår?

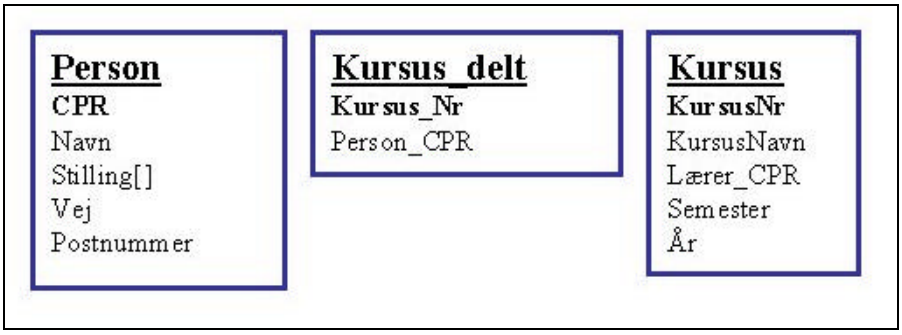
Det oprindelige design så sådan ud:



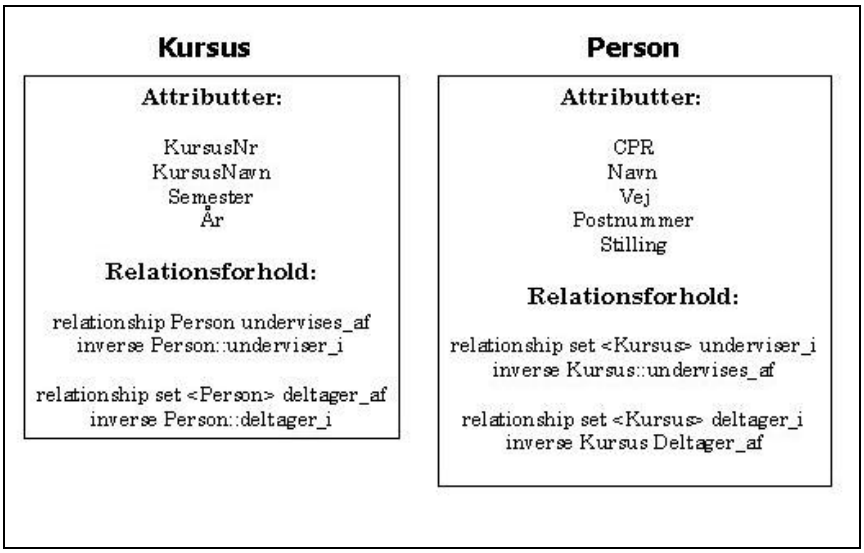
- Relationen med kursusdeltagerne kan vi ikke lave om til et array. At man ikke kan det skyldes at den referentielle integritet ikke ville kunne afklares ud fra det relationelle koncept alene.
- De øvrige relationer kan heller ikke ændres uden at vi sætter nogle af de relationelle normalformer over styr.

Der er derfor ingen ændringer i forhold til den database, vi tidligere designede.

Lad mig for eksemplets skyld derfor omdefinere databasens formål, således at der ikke længere stilles krav om, at man bibeholder en historik vedrørende lærernes stilling. Det betyder, at man kan undvære attributterne "`dato fra`" og "`dato til`" i stillingsrelationen. Og så er der intet til hinder for, at vi indsætter stillingen som et array-attribut i Personrelationen (markeret med "[ ]"). En database defineret ud fra Darwen og Dates principper vil derefter se således ud:



En tilsvarende objektorienteret database ville se sådan ud:



I den objektorienterede database er stillingsattributten også et array. Det kræver, at man definerer en metode, så brugeren kan udvælge et eller flere af de valide stillingsbetegnelser, som dernæst kan blive indført i Personens stillingsarray. Desuden skal programmøren finde ud af, hvordan han skal håndtere tilfælde, hvor nye stillinger skal oprettes og/eller eksisterende skal omdøbes.

I Dates og Darwens udvidede relationelle verden ville vi stadig have en database, der var i 1NF, selv om stillingsattributten var defineret som et array. Problemet er imidlertid, at en bruger vil have problemer med at ændre, slette eller tilføje nye stillingsnavne. Det skyldes følgende:

- de valide stillinger er defineret direkte i domænet – det er egentlig ikke meningen, at brugene skal ændre i domænedefinitioner.
- Man ikke kan opdatere værdier i et array direkte, men skal definere en metode som et interface til det. Men ved at man gør dette, betyder det samtidig, at man overlader til

den programmør, der skal definere metoden, at definere hvordan stillingsarrayet skal håndteres rent datastrukturelt, dataintegritetsmæssigt og datahåndteringsmæssigt.

Det forårsager en væsentlig diskussion, nemlig den om normalformernes aktualitet og værdi, når og såfremt en model som den Darwen og Date foreslår, bliver aktualiseret. Vi vil i det følgende afsnit nøjes med at se på hvad det ville komme til at betyde for 1NF.

6.5. Diskussion af 1NF's validitet

Ligegyldigt om man via en hårfin teoretisk skelnen kan erklære stillingsattributten for at være i overensstemmelse med 1NF, så viser det sig, at de fordele, man i relationsdatabaseverdenen havde regnet med via 1NF, forsvinder, såfremt man udnytter multiværdiede felter i form af arrays i sit design. I vores lille eksempel betyder det følgende:

- det kræver en programmør eller en metode at indføje en ny stilling i vores database - dvs. præcis samme problematik som de objektorienterede databaser har.
- hvis man senere omdøber en stilling (fx således at stillingsbetegnelsen "lektor" skal hedde noget andet), så vil referenceintegriteten ikke længere sørge for at hele databasen er opdateret. Man skal manuelt udføre en "`select`" på alle stillinger og omdøbe alle lektorer til den nye titel[16]. Havde man haft stillingsnavnet i en tabel for sig ville det ikke have været et problem at opdatere databasen.

I relation til eksemplet ville det i praksis derfor være bedst at holde sig til det oprindelige relationelle design, selv om databasen via Date & Darwens manifest har fået øgede muligheder for at definere sine attributter.

Selv om der ikke umiddelbart synes at være en brist i logikken hos Darwen og Date, så har deres ændrede definition af den relationelle model nogle uheldige konsekvenser på noget så grundlæggende som 1NF. Disse konsekvenser er ikke blot noget man kan affeje med bemærkningen om, at det vil blive sværere at designe databaser, hvis man udvider mulighederne for logisk at repræsentere data. Det, der er problemet med ovenstående er, at Darwen og Dates forslag slår fast, at databasen stadig vil være i 1NF, selv om man

[16] Dette gælder såfremt man forestiller sig, at titlen var blevet overført som en strengværdi i en attribut.

definerer stillingsattributten som et array. I praksis vil 1NF reglen dog skulle omdefineres hvis man skal sikre sig mod problemer som den, vi netop har beskrevet.

6.6. Understøttelse af bit-strømme

Selv om man med Darwen og Dates tilføjelser teoretisk set uproblematisk nu kan inkludere en attribut i personrelationen, der indeholder et billede af personen, giver det nogle alvorlige problemer m.h.t. hvordan man skal adressere selve billedet. Date og Darwen forbyder explicit at man gemmer andet end attributværdien *per se* i attributten. Dvs. at man ikke fx må gemme en henvisning til et billede i et felt (fx "c:\bin\pics\billede1.jpg") som repræsenterer selve billedet. Det gør at databasen i virkeligheden på implementationsniveau skal kunne håndtere at gemme, komprimere og vise billeder. I princippet skal databaserne derfor have muligheden for at kunne tage højde for alle de typer data, man kunne forestille sig at gemme i dem. Man skal i den forbindelse huske, at et billede kan gemmes i adskillige formatter - de fleste af dem ganske inkompatible med hinanden. Det samme gælder for lyd, video og andre typer data. Herudover findes der en række proprietære data-formater, som man sagtens kunne forestille sig, at man gerne ville gemme og organisere i en database.

6.6.1. Måder at håndtere komplexe data på

Især web-teknologien har forårsaget en revolution med hensyn til brugerkravet omkring organisering af data. Man ønsker dokumenthåndteringsprogrammer, der kan gemme og organisere Word, Excel og andre typer dokumenter, som skal være tilgængelige for hele organisationen. Man vil gerne kunne søge i de gemte dokumenttekster og regnearkstabeller. At få ovenstående funktionalitet programmeret i et domæne (fx domænet af Worddokumenter) virker lidt tosset siden dette er en funktion, som Word har implementeret i forvejen. Hvorfor ikke kalde Words funktion i stedet? Og hvis nogen nu har været så letsindige at gemme tekster i Lotus WritePro eller WordPerfect, så skal de enkelte programmers søgefunktioner arbejde sammen om en global tekstsøgning.

Problematikken omkring indlejringen og håndteringen af komplexe data i databasen bliver i dag i praksis grebet an på tre forskellige måder [Davis 97]:

- "universal server" tilgangen:
  variant a: Man udvider de relationelle databaser således at de kan forstå, gemme og håndtere komplexe data i databasen selv (Informix, DB2 og Oracle bruger denne approach).

variant b: en extended universal server approach, som giver muligheden for at data gemmes et andet sted end i selve databasen. Kun en pointer gemmes i selve databasen.

- "middleware" tilgangen
  variant a: et stykke middelware, der koordinerer en brugerinput til at finde ud af, hvilken specialiseret server, der skal behandle requesten. Middlewareprogrammel sørger for, at der er en ens datatilgang og at queries kan foregå transparent. (DataJoiner og Omniconnect bruger denne tilgang)
  variant b: applikationsmiddelware, såsom Microsoft's proprietære OLE DB og DCOM modeller. Den deler databasesystemets funktionalitet op i små komponenter, således at de enkelte komponenter kan fungere enten på operativsystem niveau eller på et middelwaredefineret område uafhængigt af hinanden (fx tidligere eksempel med søgninger i Word: man kan kalde lige præcis den funktion i Word, man har brug for, men behøver ikke nødvendigvis at bruge det samme Word-interface til den). OMG's ORB (Object Request Brokers) standard går blandt andet ud på at danne konsensus, så denne model kan implementeres så uproblematisk som muligt.
- "object layer" tilgangen
  Hvor man kan mappe databaseobjekter direkte til deres oprindelige applikationer. Man kan også direkte fra applikationen gemme dokumentet (hvad enten det er et billede, en videofilm eller noget helt tredje) i databasen. (Microsoft giver delvist denne funktionalitet)

Problemet er naturligvis ikke anderledes i objektorienterede databaser. Typisk vil objektorienterede databaser give mulighed for at se og gemme billeder og andre typer data, men ikke for at redigere dem. Undtaget herfra er CAD/CAM applikationer, der foruden databaseegenskaber også indeholder muligheder for at redigere billeder. Der er - mest på grund af ORB-modellen - tendenser i retning af, at objektorienterede databaser benytter sig af middelware indgangen (variant b).

Selv om der er meget langt igen før det lykkes, er der for mig ingen tvivl om, at vejen frem er "object layer" tilgangen i kombination med middelware (variant b). Hvis den er fuldstændig gennemført vil det betyde, at man ville kunne gemme et hvilket som helst dataformat i en database og - når man kaldte det frem - kunne redigere det i brugerens foretrukne miljø (og sprogversion).

6.6.2. Opsummering

Hvilke konsekvenser har ovenstående tre tilgange i forhold til Darwen og Dates teorier?

- "Universal Server" tilgangen repræsenterer ikke et problem, såfremt man gemmer data i selve databasen og ikke nøjes med en pointer, da dette ville kompromittere et af de krav, som Darwen og Date stiller til dataintegriteten.
- Hverken middelwaretilgangen eller "object layer"-tilgangen har ved første øjekast meget at gøre med den måde, man gemmer og strukturerer sine data på. Men alligevel kan vi konstatere, at de begge indirekte understøtter Darwen og Dates ønske om at sætte lighedstegn mellem domæne og objektklasse. Det ville gøre databasedesignet enklere og meget mere fleksibelt, at man kunne nøjes med at definere, at hvis man kalder en attributværdi, så er det dét program, der er relateret til et specifikt domæne, der sørger for interfacen. Ellers skal man til manuelt at kode dette ind i de enkelte, komplexe attributter, alternativt definere et objekt med nogle egenskaber, som kan agere som attribut i et objekt. Det vil for eksempel sige, at jeg ville blive nødt til at definere klassen "PhotoshopBillede", blandt andet med metoderne "rediger()", og "view()" som starter Photoshop op og linker denne klasse til min klasse "Person", hvis personbillederne skal inkluderes i Personobjektet.

De praktiske implementationer af databaser, der inkluderer komplexe data, peger således i retning af at det ville være fornuftigt at sætte lighedstegn mellem domæner og objektklasser, som Darwen og Date gør i deres manifest. Det er der imidlertid ikke noget nyt i. De objektorienterede databaser har altid kunnet definere et komplext objekt som et domæne. Men herudover indkapsler de også attributværdier, så objektklasser og relationer også kan svare til hinanden. Dette bruger Date [Date 95] meget krudt på at argumentere imod, men hans kritik går først og fremmest ud på at argumentere mod de objekt-relationelle databaser og kan ikke uden videre overføres til de rent objektorienterede databaser.

6.7. Dataindkapsling

I objektorienterede programmeringssprog sikrer dataindkapslingen, at det er programmøren, der *altid* bestemmer, hvordan en bruger hhv. kan se og håndtere en given attribut. Data i objektorienterede databaser er indkapslede, så man kan ikke lave forespørgsler uden først at have defineret et interface, der tillader adgangen til attributten.

"The notion of having to write new procedural code every time a new data access requirement arises is simply not acceptable," skriver Date [Date 95].

Det er imidlertid ikke korrekt. I objektorienterede databaser har man i ODMG standarden løst problemet på en ret så kreativ måde: OQL kan nemlig få fat i alle attributværdier, som der er interfaces til (men ikke dem, hvor der ikke er defineret interfaces), men ikke lave datahåndtering andet end ved at kalde definerede metoder for et givet objekt. Problemet er `UPDATE`-statements, som skal programmeres for hver enkelt attribut. Det er ret klodset, men mon ikke mange kommercielle DBMS'er vil sørge for, at der er indbyggede funktioner i databasen, ganske som *Jasmine* fx har gjort det.

Darwen og Dates forslag er nogenlunde den samme: Værdier er ikke indkapslede. Det er domæner derimod. Grunden til, at de definerer det således er at de gerne vil have, at man altid skal kunne uddrage en værdi af en attribut i en relation, uagtet om der er defineret et interface til samme eller ej. Er der ikke en given forståelig værdi, skal der defineres en indgang til værdien, således at den på en eller anden måde kan "selectes". Her skal man huske, at det *ikke* er tilladt at give en værdi andre kendetegn end dem, værdien *per se* har. Det gælder naturligvis at man i forbindelse med de komplexe data ikke vil kunne skrive:

"`Select * from Relation R where Complex_Attribute = "picture of my daugther`".

Men man ville nok kunne skrive:

"`Select * from Relation R where content(Complex_Attribute) = "picture of my daugther`"

hvor `content()` er en metode, der er defineret for Complex_Attribute's domæne[17].

På samme måde vil man kunne have defineret forskellige metoder, så det for eksempel er muligt at udføre en forespørgsel på billedet af en bestemt person, der har et bestemt

[17] Det forekommer mig imidlertid inkonsekvent, at tilgangen til komplexe data skal være anderledes end ved almindelige data. Men da der er tale om et tænkt syntax eksempel og ikke umiddelbart kan bevises ud fra manifestets egne præmisser, lader jeg sagen ligge.

mønster som baggrund. Det kræver, at man definerer en metode, der udregner baggrunden. Og det er da blandt andet også ovenstående krav, der gør, at det databasesprog, man skal benytte (SQL eller andet), skal være "*computationally complete*" [Darwen og Date 95].

6.8. Versionering & historik

En af de vigtige ting, som visse avancerede objektorienterede databaser har haft mulighed for, er versionering af dataværdierne. Med versionering menes den mulighed, at man er i stand til at vælge tidligere versioner af sin database - og data i databasen, som det blandt andet beskrives af Stonebraker [Kalman 94], som p.t. er direktør for Informix, en af de store spillere inden for objektorienterede databaser. Citatet er lidt langt, men det bringes i sin helhed for at bibringe en forståelse af versioneringsteknikken og hvordan den kan bruges på noget så prosaisk som en standard "employee"-database:

"In all current relational systems, when Dave Kalman gets a ten-percent raise, the salary bits get overwritten. To ensure you can recover from a crash, you write in the log an arcane, coded procedure that walks around in the log and fixes up transactions that weren't committed when the system failed. This is standard write-ahead log technology.
There are several problems with this. First, you cannot instantaneously recover. If the system goes down, the recovery program runs as a two-pass algorithm over the log. Recovering quickly is something no relational system can do. Secondly, if the recovery code fails, you can't recover your database. If that happens on an important enough customer, it makes the front page of The New York Times. The recovery code has to work perfectly. And, by the way, the only way to test it is to fail. This is a software engineering nightmare.
Also, you lose history. If you want to ask about Dave Kalman's salary in June of 1991, that data's gone. The system used to have it, but it went into the log, which was truncated. Postgres, and now Montage, use a different storage system. When you get a raise, new information is added to the database and your old information stays in the database. There is no log. You can think of the log as integrated into the database by just not overwriting the data. For a while, both the old salary and the new salary exist in the database. The runtime system knows which one is which. At commit time, your old salary becomes available for garbage collection. If Montage crashes, recovery time is a fraction of a millisecond.

Also, you get an extra service. On a table-by-table basis, instead of having the garbage collector just throw old values away, you can add old values to an archive table. Then, you can ask for the state of the table as of any time past. Montage gives you what we call “snapshot” semantics, where you can ask for Dave Kalman's salary as of a particular date. That's very useful in any system that requires a longterm audit trail. In environments where you want to keep the history, Montage keeps it for you automatically. If you don't want the history, then you throw it away.
User-defined functions can also time travel. If you add a new Sunset() function and ask for sunsets for last year, you get the old Sunset() function. Rules also time travel. Also, let's say the 1991 employee schema is different than the 1992 employee schema, which is different than the 1993 schema. Time travel means that when you run a query as of 1991, you get the schema as of 1991.”

Selv om Stonebraker i ovenstående fortæller, at forskellen hovedsageligt ligger i lagringsmekanismen, så er der noget, der tyder på, at der i processen gemmes visse objekt ID'er med tidsoplysninger, sådan at man kan skelne tidligere versioner fra nuværende. At håndtere dette på denne måde er udtrykkelig forbudt i de krav, som Darwen og Date stiller i deres manifest. Der nævnes heller ikke noget som helst om versionering i manifestet. På den anden side tror jeg ikke, at Darwen og Date ville have noget imod, at relationelle databaser havde en sådan funktion. At de ikke i deres manifest har nævnt muligheden skyldes sandsynligvis, at der ikke for nærværende er nogen optimal løsning for at inkludere versioneringer og historik i den relationelle løsning, som de har skitseret i manifestet. De kan selvfølgelig vælge at kalde det for en implementationsløsning, som ikke influerer på den abstrakte datamodel, men som vi kan læse af ovenstående skal man i sin forespørgselssprog kunne skelne mellem om det er data fra “før nu” eller om det er de aktuelle data, man forespørger på.

Man kan selvfølgelig implementere denne funktionalitet som en art “*dbvar*”, som hos Darwen og Date defineres som “database state” eller “database” [Darwen og Date 95], men på den anden side forbyder de explicit at domæner kan tilhøre en bestemt dbvar. Det betyder, at man ikke historisk set kan bladre i de forskellige domænedefinitioner, man måtte have foretaget i løbet af databasens levetid. Det er heller ikke afklaret, hvordan man skal håndtere omdefinitioner af metoder, relationer og attributter og stadig kunne håndtere en søgning via aktuelle og tidligere værdier i databasen (fx en `løn()` funktion for de seneste 10 år).

6.8.1. Opsummering af versioneringsproblematik

Sådan som det ser ud i øjeblikket vil versioneringsfunktioner ikke kunne lade sig gøre, hvis man benytter sig af Darwen og Dates udvidede relationelle model.

Deres projekt har da også først og fremmest været at integrere helt bestemte muligheder, som objektorienterede databaser har tilbudt brugerne. Først og fremmest håndtering af komplexe data. Da hverken de relationelle databaser eller langt de fleste af de objektorienterede databaser har håndteret historik, er det ikke underligt, at Darwen og Date ikke har lagt vægt på dette punkt. I sin bog om databaser [Date 95] skriver Date om versionering: “Such operations basically involve a lot of pointer juggling - but there are major implications (beyond the scope of this book) for language syntax and semantics in general.”

Måske er det derfor, at Date kalder versionering for en “nice-to-have” ting, snarere end en nødvendighed i forbindelse med bestemte operationer. Der kan dog ikke herske tvivl om, at versionering vil være et væsentligt konkurrenceparameter for databaserne, og at det er en funktionalitet, som mange virksomheder kan få glæde af - også i almindelige databaser, regnskabssystemer (i regnskabssystemer ville det jo næsten gøre det umuligt at ændre tal med tilbagevirkende kraft uden at det blev opdaget!) m.v.

6.9. CAD/CAM relaterede opgaver

CAD/CAM opgaver kræver typisk at man kan definere og fremvise de samme objekter på forskellige måder:

1. Fysisk (realiseret) niveau, hvor man kræver ægte tredimensional modellering
2. Højniveau blok diagram abstraktioner
3. Logiske diagrammer, fx med dele af blokkomponenterne
4. Specialiserede diagrammer (fx strømkreds, mekaniske strukturer etc.)

Herudover kræver det, at man kan kombinere de definerede objekter og at databasesystemet kan definere deres indbyrdes formodede samspil. Dette skal blandt andet bruges når hele designet skal testes igennem inden det konkretiseres i en fysisk model.

I forbindelse med dokumentation vil CAD/CAM applikationer i industrien typisk have en versioneringsfunktionalitet. Denne er diskuteret i det foregående afsnit.

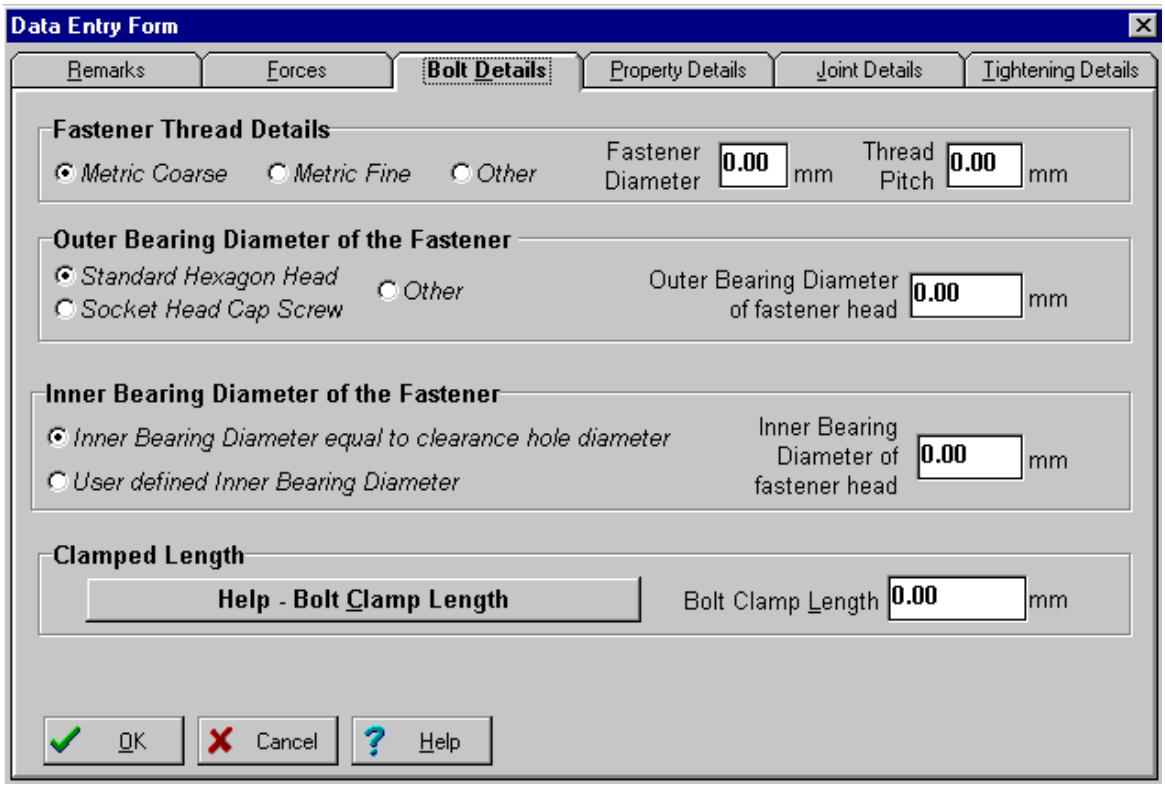
Typisk i CAD/CAM applikationer vil man kæde objekter sammen via museklik. Det er netop i CAM/CAD verdenen (især i ECAD[18] sammenhænge), at objektorienterede databaser oprindeligt blev brugt. Først og fremmest fordi man i disse miljøer var vant til C++ [Kalman dec. 94]

I CAD/CAM verdenen bevæger vi os over i noget, der mere og mere ligner applikationer, der har en database som underliggende "repository". Ikke desto mindre er Date overbevist om, at en korrekt implementering af hans og Darwens model ville betyde, at man i fremtiden "would have relational systems that do CAD/CAM" [Kalman okt. 94].
Når man taler om CAD/CAM bør man også inkludere GIS (Geographical Information Systems), da der her er tale om at strukturere data i en spatial kontext, og der er mange sammenlignelige problemstillinger mellem disse to.

Til understøttelse af disse applikationer har man i Oracle (fra version 7) og i visse andre databaser, som mestendels er objektorienterede, inkluderet en objektklasse, som i virkeligheden er en typeunderstøttelse eller - hvis man vil - domænespecifikation [Spitzer 1996].

Med fødderne solidt plantet i Darwen og Dates teorier ville der ikke være problemer i at definere en database, der kan rumme elementer omkring et bestemt fysisk objekt, her exemplificeret via Westcoms program *Boltcalc* (http://www.westcom-uk.demon.co.uk/pages/bc1.htm):

[18] Electrical Computer Aided Design (design af elektroniske kredse, såsom microprocessorer ol.)



I relationen kan man tilføje en attribut, der indholder billeder - også i 3 D - af ovenstående komponent. Med et komponentID-nummer ville man sætte denne i et relationsforhold til alskens andre relationer, der ville fortælle, om de passede sammen eller ej. Det, der imidlertid er kernen i CAD/CAM applikationen er, at man kan finde ud af hvordan et tilfældigt komponent ville reagere under helt bestemte påvirkninger, fx. metaltrætheden og holdbarhed ud fra den antagelse, at komponentet skal skrues i en storebæltsbro under vandoverfladen?

Ovenstående forespørgsel kunne måske løses ved at man lavede et nyt relationsforhold, nemlig mellem skrue og produktionsmateriale: Plastic, metal etc. Så kunne man for eksempel vælge at benytte samme bolt, men lavet af et andet materiale. Så på dette punkt ville Darwen og Date kunne honorere CAD/CAM kravene. Spørgsmålet er imidlertid om man ikke ville ende med et meget komplext system af n'ary sammenhænge, der kræver mange tabeller før det lykkes. Operationer ville derfor blive forholdsvis langsomme at gennemføre, og man kunne også nemt frygte at selve databasedesignet ville være ganske komplekst i forhold til når man har med almindelige data at gøre. Skal man for eksempel definere et domæne for materialer som plastic og metal for sig? Eller skal disse være almindelige tupler i relationer, med attributter, der yderligere beskriver deres karakter?

Og hvordan ville ændringer i boltens basiskomponener påvirke dens udseende (og omvendt)? Hvis man for eksempel gerne vil navigere i vektorbaseret grafik, vil en sådan navigering kræver en run-time generering af det grafiske udseende, brugeren skal se. Dette ville kræve, at databasen besad polymorfiske evner (se næste afsnit). Igen ville en sådan run-time generering muligvis involvere så mange joins og metodekald, at det ville være svært for en relationel baseret applikation at honorere kravene.

6.9.1. Opsummering af duelighed i forhold til CAD/CAM

Der er så mange aspekter inden for CAD/CAM programmering at det ville kræve et studie for sig at finde ud af, hvorvidt Date har ret i sin antagelse af, at relationelle databaser ville kunne finde indpas i CAD/CAM markedet. Set med særdeles venlige øjne tyder det på, at det godt ville kunne lade sig gøre - forudsat polymorfisme er til stede. Vi må dog konstatere, at der skal tænkes en del mere inden for emnet inden dette endegyldigt kan fastlås. Det ville kræve en ret omfattende undersøgelse at finde ud af dette. Dette skønnes at ligge for langt væk i forhold til det, nærværende hovedopgave og - ikke mindst - Darwen og Dates manifest stræber efter. Jeg vil derfor her nøjes med at vurdere, at det ud fra ovenstående ufuldstændige diskussion skønnes sandsynligt, at Darwen og Dates teorier i fremtiden kan implementeres under en eller anden form i CAD/CAM og GIS sammenhænge, men ikke uden at teorien videreudvikles til specifikt at behandle CAD/CAM problematikker. Dette forudsætter som tidligere nævnt, at polymorfisme er mulig. Om det så er tilfældet, vil jeg undersøge i det næste afsnit.

6.10. Polymorfisme

Polymorfisme giver databasen evnen til at udføre forskellige operationer ud fra den samme kommando, takket fra en viden i run-time kørsel om, hvilken klasse et konkret objekt hører til.

I og med at Darwen og Date i deres OO-prescription 1 nævner “compile-time checking” [Darwen og Date 95], så skal der ikke herske nogen tvivl om, at polymorfisme under en eller anden form skal være mulig. Der er heller ingen grund til, at polymorfisme i en situation, hvor metoder kan specificeres, ikke skulle kunne findes i relationelle databaser. Det lidt problematiske er, at metoder hos Date og Darwen - i modsætning til

objektorienterede databaser - ikke er defineret i selve domænet, men for sig, kun med angivelse af hvilket domæne, et resultat skal tilhøre. Det gør en faktisk implementation af polymorfe egenskaber ganske besværlig, hvad jeg skal prøve at godtgøre i det følgende.

6.10.1 Gennemgang af et eksempel i polymorfisme

Polymorfisme giver kun mening i det øjeblik, man kan benytte sig af arv. Arvmodellen er, som nævnt tidligere, ikke specificeret i manifestet. Men lad os alligevel kigge på et af de eksempler, der blev nævnt i afsnittet om arv:

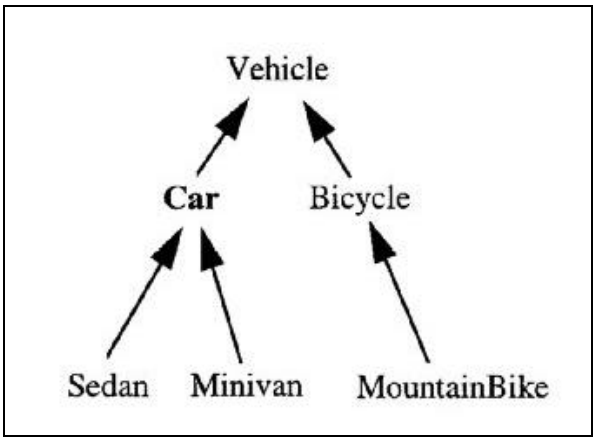


Illustration fra Antero Taivalsaari [Taivalsaari 96]

Lad os sige, at vi gerne ville have en metode, vi kalder `Pris()`. `Pris()` skal, afhængig af den bilmodel, man har valgt, skrive en pris ud. I en objektorienteret database ville vi fx i `Car` klassen[19] skrive:

```
Class Car{
  Attributliste, herunder:
           Long Pris_radio;
           Long Pris_soltag;
  Metodeliste, herunder:
           Pris() { return (this.Pris_radio + this.Pris_soltag)}
}
```

I `Bicycle` klassen ville vi skrive:

[19] En abstrakt klasse *Vehicle* ville også have defineret en Pris() metode. Denne ville sikre, at hvis man senere fik andre Vehicles defineret (båd, lastbil, tog, bus etc.), så ville man også automatisk have en Pris() dér også. For overskuelighedens skyld glemmer vi dette i eksemplet.

```
Class Bicycle{
  Attributliste, herunder:
           Long Pris_saddel;
           Long Pris_ringeklokke;
  Metodeliste, herunder:
           Pris() {return (this.Pris_saddel + this.Pris_ringeklokke)}
}
```

I klasser under Bicycle og Car ville vi ikke skulle huske at definere en Pris metode da denne allerede ville være nedarvet fra hhv. Car og Bicycle -klassen. Vi kunne dog altid tilføje nye ting, der skulle indeholdes i en eventuel `pris()`, i nedarvede klasser.

Når nu vi havde fyldt data i vores database, kunne vi fx i OQL skrive: `Pris(objektforekomst)` og få udskrevet prisen på en bestemt `Vehicle`, hvad enten det var en mountainbike eller en bil.

Hos Darwen og Dates ville vi ikke kunne have brugt polymorfisme med mindre vi havde defineret hver enkel relations-specialisering som et domæne, hvilket ikke var hensigten med Darwen og Dates teorier.

Det eneste alternativ hvis man skal følge Darwen og Dates logik, sådan som jeg kan se det, er at definere forskellige metoder i stil med:

```
Pris(Car.UniqueID) { return (this.Pris_radio + this.Pris_soltag)}

Pris(Bicylcle.uniqueID) {return (this.Pris_saddel +
     this.Pris_ringeklokke)}
```

6.10.2 Opsummering af polymorfismeproblematik

Darwen og Dates løsning er ikke så hensigtsmæssig. Problemerne ved deres løsning er blandt andet

- at man skal huske at definere Pris() for alle de relationer, man opretter
- at en kommende programmør ikke har en “basis”klasse (eller abstrakt klasse) at kigge på som i objektorienterede databaser, som kan fortælle ham, hvad en evt. ny klasse bør indeholde af attributter og metoder.
- Der er mere manuel kodning involveret og der er derfor flere fejlmuligheder

- Metoden er mindre fleksibel. Hvis man vil omdefinere Pris(), er man nødt til at omdefinere den i alle de metoder, man har oprettet (jeg forestiller mig fx en database, der indeholder 23 forskellige bilmærker inden for gruppen Sedan. Disse er alle nødt til at blive defineret som relationer idet bilmærkerne samtidig hænger sammen med forskellige leverandører etc. Resultatet er at der bliver defineret 23 forskellige Pris(), en for hver af de 23 bilmærker. én ændring i Pris() og alle 23 skal manuelt rettes - i modsætning til i objektorienterede databaser, hvor kun den Pris() i Sedan skal rettes).

Kort og godt, så dækker en sådan løsning slet ikke de faciliteter, som man finder i de objektorienterede databaser. En udvidet løsning, for eksempel hvor metoderne kunne arve fra metoder som fx `Pris(vehicle)::Pris(car)` kan jeg ikke umiddelbart forestille mig hvordan kunne realiseres i praksis. Da det at definere en relation som et domæne heller ikke er vejen at gå, virker det umiddelbart som om det endnu kræver en del forskning, før man kan få arv og polymorfisme til at gå op i Darwen og Dates model - hvis det da kan lade sig gøre i det hele taget.

Indtil arv mellem relationer er tilladt vil polymorfisme i relationelle databaser være ganske uanvendelig i sammenligning med den funktionalitet, arv har i objektorienterede databaser. Og dermed er også spørgsmålet om CAD/CAM i virkeligheden er afklaret, idet de forudsætter en polymorfisk datastruktur. Vi kan derfor konkludere, at CAD/CAM kun vanskeligt lader sig realisere med Darwen og Dates nuværende model.

6.11. Konklusion

Nu er Darwen og Dates model for, hvordan man kan indkorporere de objektorienterede databasers “gode” egenskaber i den relationelle model gennemgået og jeg er derfor i en situation, hvor jeg kan svare på det spørgsmål, jeg stillede i min problemformulering: Kan den relationelle model i virkeligheden rumme de elementer, som den objektorienterede databasemodel implementerer?

Svaret må være et rungende: “Det er i hvert fald ikke bevist endnu!”, idet den model, Darwen og Date foreslår, ikke formår at finde en tilfredsstillende løsning på arv-problematikken (jf. afsnit *6.2. Arv* og *6.10. Polymorfisme*), og dermed heller ikke på polymorfisme (jf. afsnit *6.10. Polymorfisme*). Disse to egenskaber må siges at være

grundlæggende for objektorienterede databaser og det er også nogle af dem, Darwen og Date nævner, at de gerne vil have med i deres model.

Et andet - og større problem - er, at Darwen og Date ved at indføre arrays reelt kompromiterer den gyldne relationelle regel omkring 1NF (jf. afsnit *6.5. Diskussion af 1NF's validitet*). Dermed er der en reel risiko for at referenceintegriteten ikke helt fungererer efter hensigten.

Darwen og Date sætter lighedstegn mellem objekter og domæner for at sørge for, at man altid vil have adgang til attributværdierne, men samtidig kan beskytte ændringer i domænedefinitionerne. Det giver, som vi har set, nogle kolossale problemer, når man skal implementere arv og polymorfisme.

I virkeligheden handler det måske om, at disse problemer skyldes at Date og Darwen forudsætter, at al datamanipulation nødvendigvis skal foregå *direkte* via datahåndteringssproget. I SQL kan vi således blot skrive `UPDATE`, hvis vi ønsker at ændre attributters værdier. I OQL er vi nødt til at kende navnet på den metode, der tillader os at ændre en værdi. Det er ikke længere nok at skrive:

```
UPDATE PERSON
SET NAME = “Inger Johansen”
WHERE CPR_NUMMER = 3003461010
```

I OQL ville man først skulle definere metoden `updatename()` for klassen `Person` - og i OQL kalde metoden nogenlunde således:

```
if (Person.cpr_nummer == 3003461010) person.updatename(Inger
Johansen);
```

Det er på baggrund af ovenstående at Date skriver, at endnu en stor forskel mellem relationelle og objektorienterede databaser er, at relationelle kommer “ready to use” [Date 95], mens objektorienterede databaser først skal programmeres. Og det er i virkeligheden denne forskel, der ligger bag Darwen og Dates forslag: De vil gerne have, at de relationelle databaser skal kunne håndtere de samme ting som de objektorienterede databaser, men de vil gerne have, at disse nye relationelle databaser samtidig skal komme “ready to use”. Det må ikke være sådan, at man skal til at definere opdateringsmetoder for relationer. De må

være givet på forhånd, sådan at det ikke kræver en større indsats at sætte databasen i drift. At det er simpelt sikrer også, at andre senere umiddelbart vil kunne forstå databasens opbygning. De ser nemlig fuldstændig korrekt, at det kan være svært at gennemskue hvordan en objektorienteret database er skruet sammen, hvis man både skal gennemskue objekter, metoder, attributter og definerede relationsforhold - alt sammen designet ud fra hvordan en programmørs forståelse af opgaven har været og uden nogle veldefinerede metoder til at hjælpe sig med det.

Desværre er virker det på mig som om det både er at ville blæse og have mel i munden, at gøre det så enkelt, at det er “ready to use”. Man kan ikke både kræve meget avancerede, ikke-intuitive funktioner og samtidig sige, at det skal være meget nemt at gå til.

7. Konklusion

Hvori jeg først trinvist besvarer problemformuleringen og dernæst udfører den halvumulige kunst at tegne nogle store, brede og især langtrækkende perspektiver på fremtidens databasemodeller og de udfordringer, de vil møde.

7.1. Forskelle mellem de to databasemodeller

En af de hensigter Codd havde med den relationelle model var, at data skulle repræsenteres naturligt, uden at maskinen påtvang mennesket unødvendige strukturer. Det samme ønske har den objektorienterede databasemodel. Men ud fra beskrivelsen af henholdsvis den relationelle og den objektorienterede databasemodel kan vi konstatere, at der er enddog meget store forskelle på den måde, de to modeller strukturerer en database på.

Datastrukturelt adskiller de to modeller sig derved at man i den relationelle har et domæne, Date definerer som værende indkapslet. Attributter kan trække værdier fra disse domæner, men kan ikke selv være indkapslede. I den objektorienterede model er en tupel et objekt, som er en klasseforekomst. Til klassen hører en interfacedefinition, der fortæller, hvad man som bruger har tilgang til. Alt er i princippet indkapslet, med mindre man specifikt fortæller, at man har adgang til en attribut via et interface til klassen. Arv og polymorfisme er tilladte i den objektorienterede model, hvilket ikke er tilfældet i den relationelle, ligesom den objektorienterede model via klassen tillader definitionen af metoder.

Dataintegritetsmæssigt adskiller de to modeller sig ved at man i den objektorienterede model har identificeret hver tupel med et objekt ID, hvor man i den relationelle har en kandidatnøgle som sikkerhed for dataentydigheden. Referenceintegriteten i den relationelle foregår ved en fremmednøgle i en relateret tabel, hvor den objektorienterede bruger et to-vejs link med såkaldte "traversal paths" via objekternes respektive objekt ID'er. Her gælder det igen, at arv og polymorfisme er tilladte i den objektorienterede model, hvilket ikke er tilfældet i den relationelle.

Datahåndteringsmæssigt ser vi igen større forskelle end ligheder. Den relationelle datahåndtering hviler fuldt ud på en matematisk model, nemlig den relationelle algebra. Den objektorienterede datahåndtering foregår via de metoder, man har defineret i

databasernes klasser. Deres databehandling hviler ikke fuldt ud på en matematisk model, men på de teknikker, de objektorienterede programmeringssprog har til rådighed. Hertil gælder endvidere, at metoderne i objektorienterede databaser vil være applikationsafhængige, og ikke som ved SQL-kompatible baser være metoder, som er defineret som en standard, fx `UPDATE`.

Måden, man griber **databasedesignet** an på hviler naturligvis på de respektive modellers styrker og svagheder. Den relationelle model har en analytisk tilgang til data, som på et overordnet niveau afspejler modellens baggrund i en stringent matematisk model. En databasedesigner analyserer de funktionelle afhængigheder, der findes mellem de enkelte entiteter, der skal registreres. Dernæst normaliserer datadesigneren de data, der skal gemmes i databasen, således at redundans så vidt muligt fjernes og databasens struktur er så logisk, at det dels bliver nemt at tilføje nye data, dels nemt at indarbejde ny funktionalitet i en eventuel overliggende applikation. I det relationelle design er applikation og databasestruktur i princippet adskilte størrelser. I en objektorienteret database vil man først finde ud af, hvad man har tænkt sig at bruge databasen til. Ud fra de givne oplysninger vil man danne sig et overblik over de data, man skal organisere for at få de ønskede svar ud fra den færdige databaseapplikation, samt de metoder, databaseapplikationen som helhed er nødt til at indeholde. Fokus er i den objektorienterede model flyttet fra selve databasen til det, databasen bliver brugt til - hvilket er meget naturligt, når man tænker på, at designeren skal tage stilling til, hvilke metoder, der fx via polymorfisme skal understøtter de queries, en database skal kunne håndtere.

Ud fra ovenstående forskelle, er det ikke overraskende at kunne konstatere, at der er store praktiske forskelle i den måde, man er nødt til at arbejde med henholdsvis en relationel og en objektorienteret database. Denne forskel træder med al ønsket tydelighed frem allerede i designfasen. Allerede når databasedesigneren får stillet en opgave, tænker han i forskellige baner afhængig af hans valg af databasemodel. De spørgsmål han vil stille opdragsgiveren vil ofte være en kende anderledes. De værktøjer og den metodik, han benytter sig af, når han skal designe databasen vil være vidt forskellige. Vi kan derfor svare bekræftende på det spørgsmål i problemformuleringen, der skulle afklare, hvorvidt forskellene i de to modeller *har stor betydning for den måde, en databasedesigner vil strukturere sine data på*.

7.2. Date og Darwens syntesemodel

De relationelle databaser trænger til modernisering, således at de kan håndtere komplexe data som billeder, lyd, video etc. på en måde, der er i overensstemmelse med den relationelle models tankegang og matematiske fundering. De nyere hybride baser, som løbende har været omtalt har givet hver deres bud på hvordan dette skulle ske. SQL 3, som formodentlig vil være klar engang i 1999 vil også give et bud på, hvordan dette skal håndteres.

Men når nu man alligevel er i gang med at modernisere og omfortolke den relationelle model, så kan man lige så godt få nogle af den objektorienterede models gode egenskaber med i en ny relationel model. Date og Darwen mener, at hvis den relationelle model blev implementeret korrekt, så ville den kunne løse præcis de samme opgaver som de objektorienteret databaser. Og det endda baseret på et stringent matematisk fundament. Det er udgangspunktet for Date og Darwens "The Third Manifesto", som er et forsøg på at give retningslinierne for en sådan syntesemodel. En forudsætning for Date og Darwens syntesemodel er, at den objektorienterede models egenskaber skal indpasses fuldt ud i det relationelle koncept. En implicit følge af denne forudsætning er, at ingen attributværdier må være indkapslede.

Date og Darwen udfører så et kunstgreb, som ved første øjekast synes at løse flere problemer: De siger, at domæner er at sammenligne med klasser. Heraf følger nemlig, at relationer og deres attributter ikke kan indkapsles, mens selve attributternes domæner godt kan indkapsles. Så har de løst indkapslingsproblemet, og da domænet i princippet er lig med klassen, kan domæner godt arve fra hinanden. På den måde får de også indkorporeret arv-ideen i deres model, omend de er uenige om, hvordan det helt præcist skal implementeres. Domænet er jo som sagt lig med objektklassen, og Date og Darwen tillader derfor en constructor funktion i forbindelse med skabelsen af en ny domæneforekomst. De sikrer, at for eksempel billeder, der måtte have særlige tilgange defineret, også får dem konstrueret. Dernæst definerer de metoder som noget, der defineres uden for domænedefinitionen, men med en returværdi, der specificerer domænetilhørsforholdet. Så har de et fortsat skel mellem datastruktur og datahåndtering.

Date og Darwens syntesemodel virker umiddelbart som en logisk model, der overholder de forudsætninger, som de har sat sig selv. Men når man forsøger at tænke konsekvenserne

igennem og laver en teoretisk implementering af deres model, så er der flere uløste problemer i det. Ved at tillade komplexe, brugerdefinerede størrelser (via domænedefinitioner) som attributværdier, kunne man risikere at 1NF reelt blev sat ud af kraft uden at man designmæssigt kunne påvise, at dette var tilfældet. Dette kan man selvfølgelig løse ved at definere en ny normalform, der tager højde for dette.

Versionering kan Date og Darwens syntesemodel ikke håndtere, men det har de til fælles med mange implementeringer af de objektorienterede databaser. Man kan heller ikke hævde, at det er en indbygget egenskab ved den objektorienterede model. Dog tror jeg at netop versionering vil være en ret stærk egenskab, som vil være en selvfølge om nogle år.

Det er i polymorfisme, at Date og Darwens model tydeligst begynder at vise sine svagheder. Man har brug for, at metoder kan arve og omdefinere hinanden undervejs. Ellers fungerer de ikke polymorfisk. For at fungere polymorfisk er det også nødvendigt, at metoder fungerer på tupelniveau og ikke kun på domæneniveau. Derfor kan polymorfisme simpelthen ikke lade sig gøre i deres syntesemodel.

Polymorfisme er en af de store grunde til, at arv benyttes. En arvmodel, der ikke tillader en gennemførlig polymorfisk opførsel i de definerede metoder, er ikke en god arvmodel. Dette ligger indbygget som en modsætning i Date og Darwens syntesemodel, og det er sandsynligvis årsagen til, at der endnu ikke er defineret en tilfredsstillende arvmodel, der kan overholde de specifikationer, som de to teoretikere tegner i deres manifest.

Sådan som jeg ser det lader det sig derfor ikke gøre at definere en syntesemodel, der fuldt ud hviler på det relationelle koncept, hvis man samtidig ønsker, at den relationelle model skal kunne håndtere præcis samme opgaver som de objektorienterede databaser kan. Dermed kan jeg svare benægtende på Date og Darwens påstand om, at *den relationelle model i virkeligheden kan rumme de elementer, som den objektorienterede databasemodel implementerer*.

Jeg vil gerne understrege, at det ikke dermed betyder, at jeg synes at Dates og Darwens databasemodel er tåbelig, for det synes jeg ikke, at den er. Den kan håndtere og løse mange problemer. Men den kan ikke overflødiggøre de objektorienterede databaser.

7.3. En åben afrunding

Jeg indledte denne hovedopgave ved at understrege den betydning databaser har for vores samfund. I takt med at nye muligheder afdækkes, øges også de krav, vi har til databaserne. GPS-systemer muliggør en præcis placering af objekter. Det er ikke utænkeligt, at man ønsker, at databaser skal kunne følge en given entitets bevægelser, registrere disse interaktivt, og vise dem over for brugeren via et kort – eller endda spacialt ved at skabe et lille 3D rum. Det er slet ikke sikkert, at en lang række af de data, vi ønsker registreret fremover, vil blive ved med at være egnet til tabelrepræsentationer. Overfør eksempelvis ovenstående billede til noget så ordinært som en biblioteksdatabase: Tænk hvis databasen også viste os, hvor bogen rent faktisk stod på biblioteket.

Verden forandres og dermed forandres også den måde, vi ønsker at registrere og derved overskue en lille del af verden på. Den relationelle model er god til at strukturere en bestemt type data. Denne type data er den, de fleste brugere p.t. ønsker registreret. Den objektorienterede model er bedre til at strukturere andre typer data. For mig at se er de gode til den type data der kræver en bestemt fortolkning lagt på sig, før man som bruger kan forstå værdien i en given attribut. For eksempel ved at skabe den geografiske eller spaciale repræsentation, som jeg beskrev ovenfor.

Det er fejlagtigt at tro at den relationelle model eller den objektorienterede model er de endegyldige datarepræsentations- og strukturerings modeller. Efterhånden som vi bliver klogere, vil der komme andre databasemodeller, og hver især vil igen have forskellige styrker og svagheder.

Nu har Date og Darwen forsøgt at få den objektorienterede models gode egenskaber indarbejdede i den relationelle model. Et spændende projekt ville nu være at undersøge, om man omvendt kunne få de mange gode egenskaber ved den relationelle model indpasset i den objektorienterede.

8. Litteraturliste

Alagic, Suad: The ODMG Object Model: Does it Make Sense? OOPSLA '97 10. ACM 1997.

Alhajj, Reda and Arkun, M. Erol. Bilkend University, Faculkty of Engineering, Ankara, Turkey: An Object Algebra for Object-Oriented Database Systems. Article from Database, August 1993 pp 13-22.

Atkison et al: The Object Oriented Database System Manifesto, Proc. DOOD 89, Kyoto/Japan, 1989

Baclawski, Kenneth and Indurkhya, Bipin: The Notion of Inheritance in Object Oriented Programming. In Communications of the ACM, September 1994

Codd, E.F. : A relational Model of Data for Large Shared Data Banks CACM 13, No. 6, juni 1970

Codd, E.F. : Data Models in Database Management. Proc. Workshop on Data Abstraction, Databases, and Conceptual Modelling, Pingree Park, Colo. 1980

Codd, E.F. : Is Your DBMS really relational? artikel i to dele fra ComputerWorld i oktober 1985.

Codd, E.F.: The Relational Model for Database Management, Version 2. Addison-Wesley (1990).

Cattell, R.G.G. et al.: Object Database Standard: ODMG 2.0. Morgan Kaufmann 1997.

Cattell, R.G.G.: Object Database Management. Object-Oriented and Extended Relational Database Systems. Addison-Wesley 1994.

Date, C. J.: A critical Review of the Relational Model Version 2 (RM/V) in C.J. Date og Hugh Darwen, Relational Database Writings 1981-1991, Addison-Wesley, 1992.

Date, C. J.: “Notes toward a Reconstituted Definition of the Relational Model Version 1 (RM/V1). in C.J. Date og Hugh Darwen, Relational Database Writings 1981-1991, Addison-Wesley, 1992.

Date, C. J. and Hugh Darwen: The Third Manifesto. ACM SIGMOD, vol 24, no 1, March 1995.

Date, C. J.: An Introduction to Database Systems, Sixth Edition, Addison-Wesley, 1995.

Davis, Judy: Extended Relational DBMSs: The Technology, Part 1. In DBMS Online, June, 1996. (http://www.dbmsmag.com/9706d13.html)

Davis, Judy: Universal Servers: The Players. In DBMS Online, July, 1996. (http://www.dbmsmag.com/9707d14.html)

Ellis, Margaret A. & Stroustrup, Bjarne: The Annotated C++ reference manual. Addison Wesley, 1990

Elmasri & Navathe: Fundamentals of Database Systems, 2nd edition. Addison-Wesley, 1994.

Glaven, Torkild: C++ programmering. Teknisk forlag 1992.

Jade udviklingsværktøj & website: http://www.jade.co.nz./

Kalman, David: Moving Forward with Relational (C. J. Date, Independent Author, Lecturer, and Consultant) - Looking for objects in the relational model, Chris Date finds they were there all the time. (Interview) ; DBMS Online October 1994. Kan findes på http://www.dbmsmag.com/

Kalman, David: A new direction in DBMS: Montage Software's Dr. Michael R. Stonebraker takes the wraps off his new object-relational DBMS. DBMS Online February 1994. Kan findes på http://www.dbmsmag.com/

Kalman, David: Object Database essentials: HP's Dr. Mary Loomis explains the fundamentals of object database technology. DBMS Online December 1994. Kan findes på http://www.dbmsmag.com/

Kim, Won: Introduction to Object-Oriented Databases. MIT Press, 1992.

Naughton, Patrick & Schildt, Herbert: Java 1.1. The Complete Reference. Osborne 1998.

Object Agency, The: A Comparison of Object-Oriented Development Methodologies. Kan findes i sin helhed på internetadressen: http://www.toa.com/pub/html/mcr.html, 1995.

Spitzer, Tom: A Database Perspective on GIS, Part 1 DBMS Online November 1996. Kan findes på http://www.dbmsmag.com/

Taivalsaari, Antero: On the Notion of Inheritance. ACM Computing Surveys, Vol. 28, No. 3, September 1996, pages 438-479.