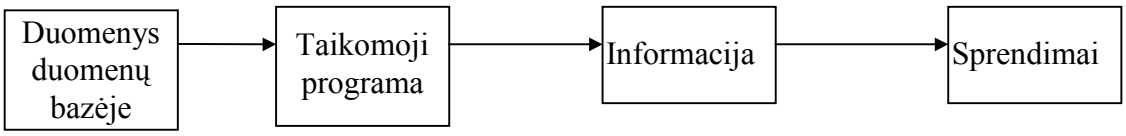
Duomenų bazių projektavimas

Duomenų bazių projektavimas yra didesnio uždavinio - informacinės sistemos projektavimo - dalis. Informacinėje sistemoje yra ne tik renkami, saugomi ir klasifikuojami duomenys, jie taip pat transformuojami į informaciją. Reikalavimų informacinei sistemai nustatymas ir jos ribų apibrėžimas yra sistemų analizės esmė. Informacinės sistemos kūrimo procesas vadinamas taip pat sistemų vystymo (system development) vardu.

Duomenų transformavimas į informaciją

Duomenys yra žaliava, saugoma duomenų bazėse. Transformuojant juos į informaciją, lemia efektyvus duomenų bazių projektavimas. Kadangi tinkamas duomenų rinkimas, saugojimas ir atranka yra bet kokios organizacijos esminė veikla, jos iškreipti negalima. Asmeniui priimant sprendimus reikalinga **informacija**, kuri yra ne kas kita kaip prasmingai išdėstyti duomenys – tai ir yra duomenų transformavimo prasmė.



Informacinės sistemos našumas priklauso nuo faktorių trijulės:

- duomenų bazės projekto ir įdiegimo,
- taikomųjų programų projekto ir įdiegimo
- administracinių procedūrų.

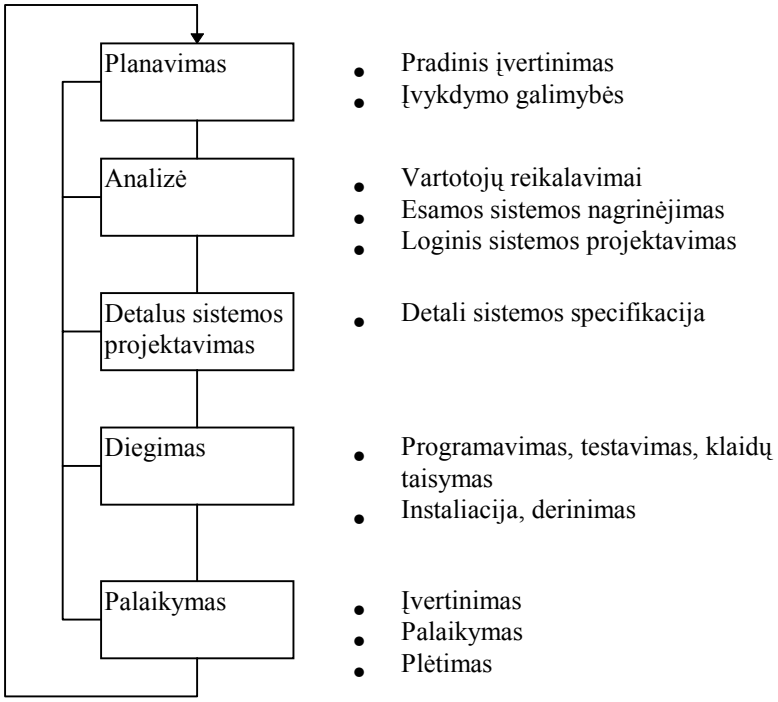
Sistemų analizė ir vystymas reikalauja gana daug planavimo tam, kad užtikrinti sąveiką tarp visų veiklos komponentų, tam kad jie papildytų vienas kitą ir kad užsibaigtų laiku.

Plačiąja prasme terminas **duomenų bazės vystymas** yra naudojamas duomenų bazės projektavimo ir įdiegimo procesų aprašymui. Pagrindinis duomenų bazės projektavimo tikslas yra sukurti pilnus, normalizuotus, (kiek įmanomą) neperteklinius bei koncptualiai, logiškai ir fiziškai pilnai neprieštaringus duomenų bazių modelius. Diegimo fazė apima duomenų bazės saugojimo struktūros sukūrimą, reikamų vartotojų duomenų surąšymą, ir duomenų valdymo priemonių sudarymą.

Sistemų Vystymo Gyvavimo Ciklas (SVGC)
The Systems Development Life Cycle (SDLC)

Sistemų Vystymo Gyvavimo Ciklas (SVGC) dokumentuoja informacinės sistemos istoriją (gyvavimo ciklą). Duomenų bazių projektavimas vyksta informacinės sistemos aplinkoje ir yra sunku atskirti duomenų bazių projektavimą nuo SVGC (ir atvirkščiai).

SVGC yra daugiau iteratyvus nei nuoseklus procesas. Pavyzdžiui, įvykdymo galimybių analizė padeda nustatyti pradinį įvertinimą, o informacija, gauta nagrinėjant SVGC vartotojo reikalavimus ar jų dalį padeda nustatyti įvykdymo galimybes. Panašiai kaip informacinės sistemos, duomenų bazės, kurios yra jų dalys, yra subjektai, kuriuos veikia gyvavimo ciklas.



Sistemų Vystymo Gyvavimo Ciklas (SVGC)

Planavimas

SVGC planavimo fazė sudaro bendrą kompanijos ir jos tikslų apžvalgą. Čia padaromas pradinis informacijos srautų ir tų srautų įtakos ribų įvertinimas, kuris turi duoti atsakymus į klausimus:

- Ar toliau naudotis egzistuojančia informacine sistema? Ar informacijos generatorius gerai atlieka savo darbą? Ar reikia jį modifikuoti arba pakeisti?
- Ar reikalinga nedidelė, ar didelė modifikacija? Kai modifikavimas bus atliekamas, reikia jausti ir vertinti skirtumą tarp norų ir poreikių.
- Ar reikia pakeisti esamą sistemą? Turint omeny pastangas, reikalingas sukurti naują sistemą, reikia labai kruopščiai atskirti norus ir poreikius.

SVGC pradinio įvertinimo dalyviai turi išnagrinėti ir įvertinti alternatyvius sprendimus. Įgyvendinimo galimybių nagrinėjimas turi apimti:

- aparatinės ir programinės įrangos techninius aspektus (aparatūros tipas, reikalavimai programinei įrangai, duomenų bazės tipas ir jos programinė įranga, programavimo kalbos, naudojamos taikomiesiems tikslams ir t.t.;
- sistemos sąnaudas.

Analizė

Problemos, nustatytos planavimo fazėje, detaliau nagrinėjamos analizės fazėje. Atliekama tiek mikroanalizė, remiantis individualiais poreikiais, tiek ir makroanalizė, reminantis visos organizacijos poreikiais:

- Kokie yra tikslūs galutinių vartotojų reikalavimai esamai sistemai ?
- Ar šie reikalavimai tinka prie visuotinių informacinių reikalavimų ?

SVGC analizės fazė faktiškai yra kruopštus vartotojų reikalavimų auditas.

Analizės fazėje taipogi nagrinėjamos esamos aparatinės ir programinės sistemos. Analizės rezultatas turi būti kur kas išsamesnis sistemos funkcinių sričių, potencialių problemų ir galimybių pažinimas.

Kartu su vartotojų reikalavimų ir esamų sistemų nagrinėjimu, analizės fazėje kuriamas loginis sistemos projektas. Kuriant loginį projektą projektuotojas gali naudoti įrankius:

- Duomenų Tėkmės Diagramas (DTD),
- Hierarchines Įvedimo Apdorojimo ir Išvedimo (HĮAI) diagramas,
- Esybių - Ryšių (E-R) diagramas.

Šis duomenų bazių proektavimo procesas dar vadinamas **duomenų modeliavimu** (data-modeling), ir jo tikslas yra rasti ir aprašyti visas esybes, jų atributus ir santykius tarp esybių duomenų bazėje.

Kiekvienam procesui duomenų bazės aplinkoje, loginis sistemos aprašymas sudaro sistemos komponentų (modulių) funkcinius aprašus (FA). Visos duomenų transformacijos (procesai) yra aprašomos ir dokumentuojamos naudojant tokius sistemų analizės įrankius, kaip DTD (Duomenų Tėkmės Diagramos). Jų pagalba yra patikrinamas koncepcinis duomenų modelis.

Detalus sistemos projektavimas

Detalaus sistemos projektavimo fazėje, projektuotojas užbaigia sistemos procesų projektavimą. Tai apjungia visas reikalingas technines specifikacijas formoms, meniu sistemoms, ataskaitoms ir kitiems dalykams, kurių tikslas - padaryti sistemą efektyvesne informacijos generavimo prasme. Išdėstomi perėjimo iš senos sistemos į naują žingsniai. Suplanuojami ir pateikiami valdybos pritarimui vartotojų apmokymo principai ir metodai.

Įdiegimas

Diegimo fazėje yra instaliuojama aparatūra ir pridėtinės programos, be to įdiegiamas duomenų bazės projektas. Pradinėse diegimo fazės dalyse sistema įeina į programavimo, testavimo ir taisymo ciklą, kol netaps paruošta naudojimui. Pačios duomenų bazės sistema yra suderinama kuriant lenteles, duomenų vaizdus, po to autorizuojami vartotojai ir t.t.

Duomenų bazės turinys gali būti suvestas interaktyviai arba paketiniame režime, naudojant įvairius metodus ar įrenginius:

1. tam pritaikytas vartotojo programas
2. duomenų bazės interfeiso programas
3. konvertavimo programas, kurios importuoja duomenis iš vienos failų sistemos į kitą, naudojant automatizavimo programas, duomenų bazės pagalbines programas, irpan.

Sistemą reikia intensyviai testuoti, kol ji netaps tinkama vartojimui. Tradiciškai įdiegimas ir testavimas užima nuo 50 iki 60 proc. viso vystymo laiko, tačiau galingų programų kūrimo bei testavimo įrankių pasirodymas vis mažina programavimo ir testavimo laiką. Baigus testavimą, peržiūrima ir atspausdinama galutinė dokumentacija bei apmokomi vartotojai. Šios fazės pabaigoje sistema yra pilnai veikianti, toliau ji bus pastoviai prižiūrima ir derinama.

Palaikymas

Beveik iš karto, kai tik sistema pradeda veikti, vartotojai pradeda reikalauti pakeitimų. Tie pakeitimai ir sudaro sistemos palaikymo veiklą, kurią galima išskaidyti į tris tipus:

1. Taisymo palaikymas, reaguojant į sistemos klaidas;
2. Derinimo palaikymas dėl pasikeitimų veiklos aplinkoje;

3. Tobulinimo palaikymas sistemos plėtimui.

Kadangi kiekvienas struktūrinis pakeitimas reikalauja pakartotinio SVGC žingsnių praėjimo, ta prasme sistema pastoviai yra SVGC cikle.

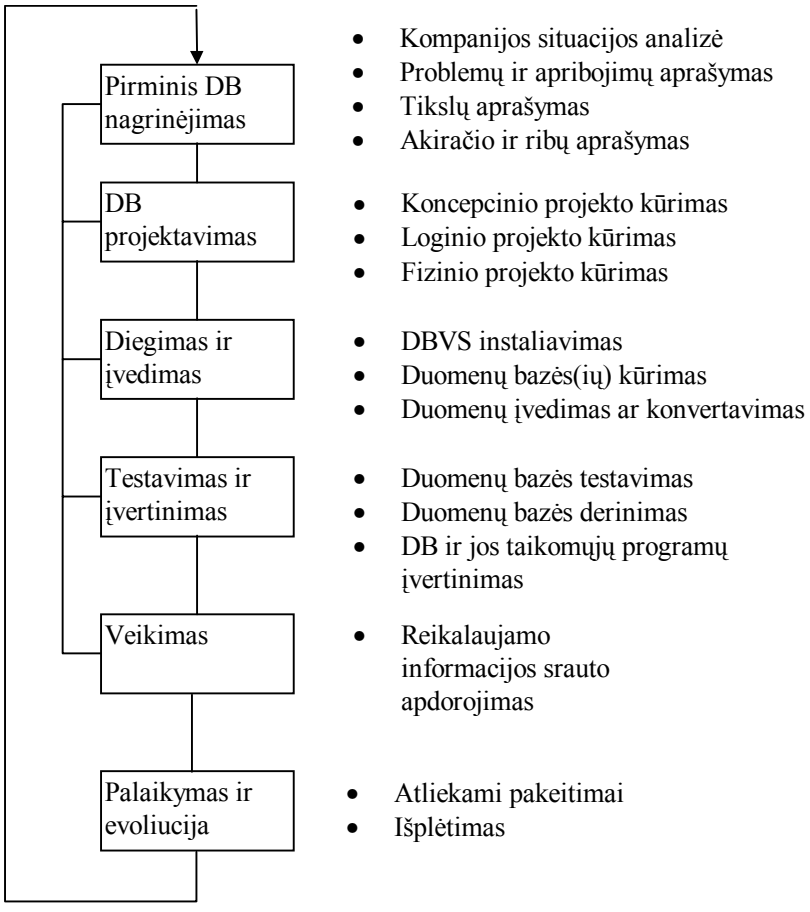
Faktoriai, kurie trumpina sistemos darbo ciklą:

- greita technologijos pažanga;
- sistemos palaikymo kaina.

Jei palaikymo kaina didelė - sistemos naudojimas kelia įtarimą. Kompiuterinės sistemų inžinerijos (CASE) technologija įgalina kurti geresnes sistemas per priimtiną laiko tarpą ir priimtina kaina. Be to, CASE pagalba sukurtų taikomųjų programų rafinuotumas leidžia pratęsti sistemos gyvavimo trukmę.

Duomenų Bazės Gyvavimo Ciklas (DBGC)

Stambesnėje informacinėje sistemoje duomenų bazės irgi turi gyvavimo ciklą. Duomenų Bazės Gyvybės Ciklas yra sudarytas iš šešių fazių:



Duomenų bazės pirminis nagrinėjimas

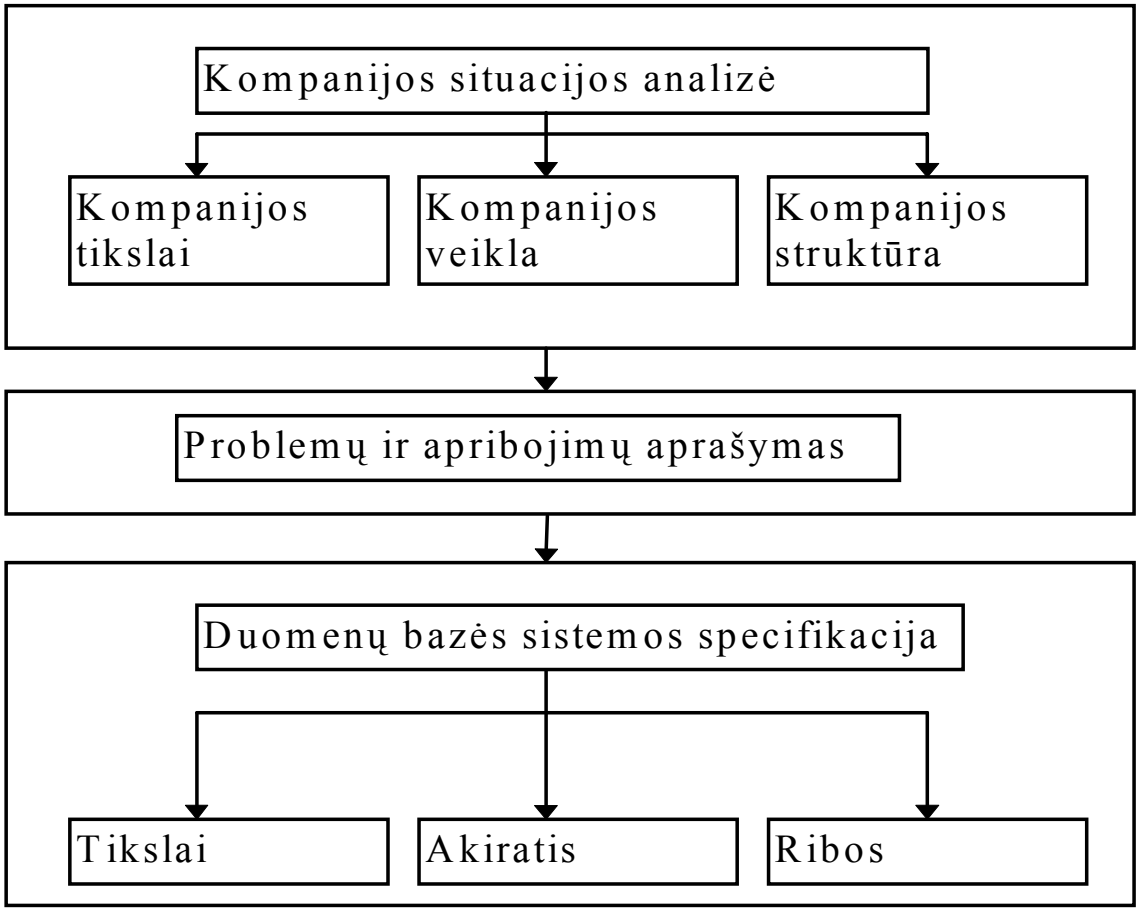
Tyrinėdamas esamos sistemos veikimą kompanijoje, projektuotojas turi nustatyti ko ir kodėl esama sistema nesugeba. Nors duomenų bazės projektavimas yra grynai techninė veikla, bet ji gan socialiai orientuota. Duomenų bazių projektuotojams reikia daug komunikabilumo.

Pastaba: priklausomai nuo duomenų bazės aplinkos prognozuojamo sudėtingumo ir akiračio , projektuotojas gali būti vienas asmuo arba gali būti visa sistemų vystymo komanda, sudaryta iš projekto vadovo ir sistemų analitikų.

Bendras pradinio duomenų bazės nagrinėjimo tikslas yra:

- Atlikti kompanijos analizę
- Aprašyti problemas ir apribojimus
- Aprašyti tikslus
- Aprašyti akiratį ir ribas (scope and boundaries)

Duomenų bazės pradinis nagrinėjimas veda prie duomenų bazės sistemos tikslų vystymo.



Duomenų bazės pradinio nagrinėjimo veiklos santrauka

Kompanijos situacijos analizė:

- Kokia yra organizacijos pagrindinės veiklos aplinka ir kokie yra jos tikslai šioje aplinkoje? Projektas turi patenkinti veiklos poreikius sukurtus pagal organizacijos tikslus;
- Kokia yra organizacinė struktūra? Žinojimas kas ką valdo ir kas kam atsiskaitinėja yra naudingas nustatant reikiamus informacijos srautus, ataskaitas, užklausų formatus ir kt.

Problemų ir apribojimų aprašymas (apimant formalius ir neformalius informacijos šaltinius):

- Kaip veikia esama sistema?
- Kokių įvedamų duomenų reikia sistemai?
- Kokius dokumentus sistema generuoja?
- Kaip naudojami sistemos išvedami duomenys?
- Kas naudoja?

Popierinių kelių nagrinėjimas būna labai informatyvus. Nuošaliai nuo oficialios informacijos egzistuoja neformali reali informacija. Projektuotojas turi būti pakankamai įžvalgus, kad pastebėti kuom jie skiriasi.

Tikslų aprašymas.

Siūloma duomenų bazių sistema turi padėti spręsti bent didžiausias problemas identifikuotas problemų aprašymo fazėje. Kai atskleidžiamas problemų sąrašas, tai greičiausiai bus surasti bendri šaltiniai. Pavyzdžiui marketingo ir gamybos vadybininkai yra nepatenkinti inventorizacijos neefektyvumu. Jei projektuotojas sukuria duomenų bazę, kuri įgalina efektyvesnį inventoriaus valdymą, tai abu skyriai išlošia. Taigi pradinis tikslas turi būti efektyvesnės inventoriaus apskaitos ir valdymo sistemos sukūrimas.

- Koks yra siūlomos sistemos pradinis tikslas ?
- Ar sistema galės dirbti su kitomis esamomis ar būsimomis kompanijos sistemomis?
- Ar sistema dalinsis duomenimis su kitomis sistemomis ar vartotojais?

Akiratis ir ribos.

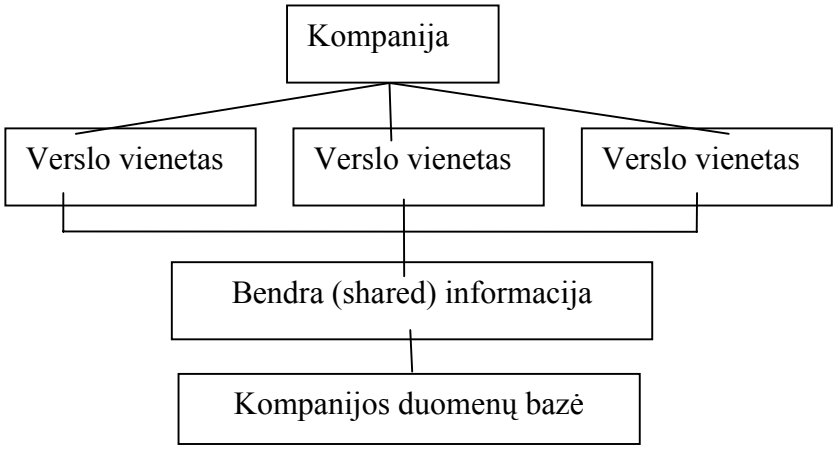
Projektuotojas turi kreipti dėmesį į dviejų tipų apribojimus, kurias turi sistemos: akiratį ir ribas. Sistemos akiratis apima projekto panaudojamumą, kaip jį apibrėžia operaciniai reikalavimai (ar projektas apima visą organizaciją, vieną ar kelis organizacijos skyrius, ar vieną ar kelias vienintelio skyriaus funkcijas).

Sistema taip pat apribota eile išorinių faktorių: biudžeto, egzistuojančios aparatinės ir programinės įrangos

Akiratis ir ribos tampa faktoriais, kurie pakreipia projektą sava linkme. Projektuotojo darbas yra sukurti geriausią sistemą, kuri apimtų reikiamą akiratį ir tilptų į ribas. (Verta pastebėti, kad problemų aprašymai ir tikslų aprašymai kartais turi būti pakeisti tam, kad prisiderinti prie šių dviejų apribojimų).

Duomenų bazės projektavimo procesas

Šis etapas yra susietas su duomenų bazės, kuri palaikytų kompanijų operacijas ir objektus, projektavimu. Duomenų bazės projektavimo procese reikia koncentruotis į duomenų charakteristikas, reikalingas DB modelio sudarymui.

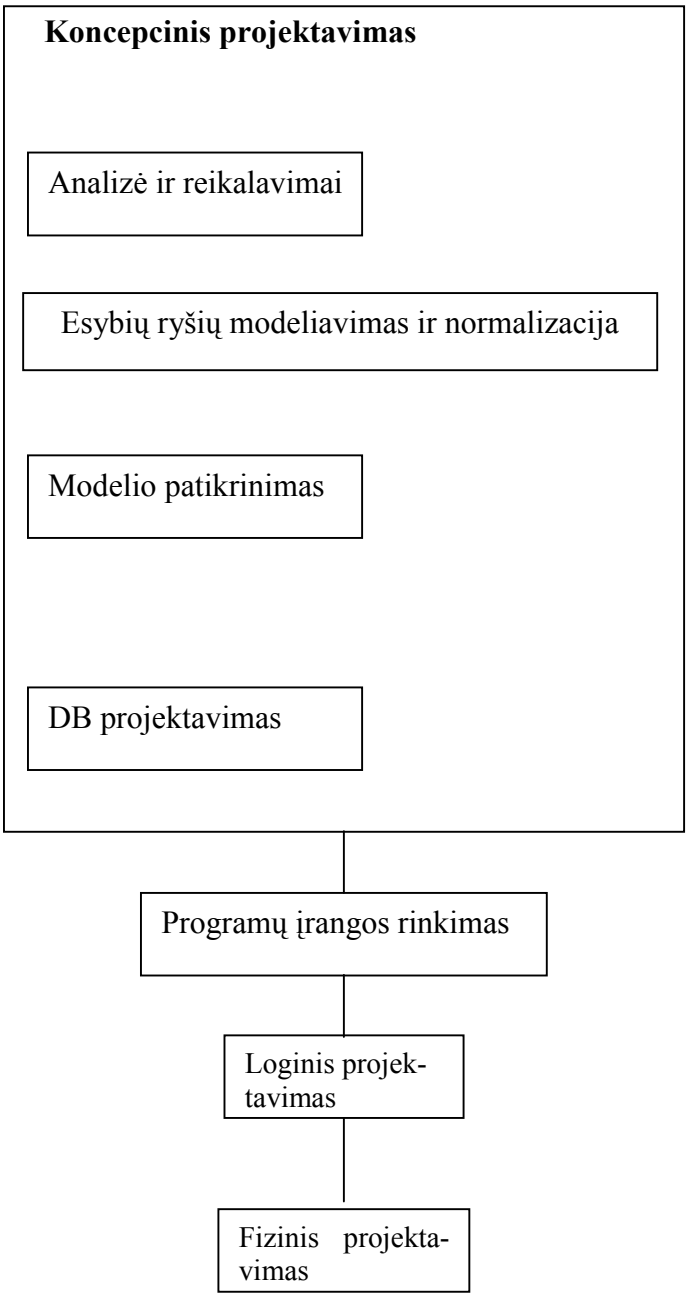


Verslo požiūris:

- Kas yra problemos?
- Kas yra sprendimai?
- Kokios informacijos reikia sprendimų įgyvendinimui?
- Kokių duomenų reikia tam, kad kurti naudingą informaciją?

Projektuotojo požiūris:

- Kaip duomenys turi būti strukturizuoti?
- Kaip duomenys turi būti prieinami?
- Kaip duomenys turi būti transformuojami į informaciją?



Nagrinėjant procedūras, reikalingas DBGC projektavimo etapui, reikia:

- DB projektavimo procesas yra priklausomas nuo didesnės apimančios sistemos nagrinėjimo ir projektavimo. Duomenų komponentė yra tik vienas iš informacinės sistemos elementų.
- Sistemos analitikai arba sistemos programuotojai kartu projektuoja ir kitas sistemos komponentes; jie kuria procedūras, kurios padeda transformuoti DB duomenis į naudingą informaciją.
- DB projektavimas nesudaro nuoseklaus proceso; priešingai, tai yra pasikartojantis procesas, kuris turi nuolatinį grįžtamąjį ryšį, suprojektuota tam, kad sekti ir ankstesnius veiksmus.

Koncepcinis (scheminis) projektavimas

Scheminiame projektavime duomenų modeliavimas yra naudojamas, kad sukurti abstrakčios DB struktūrą, kuri atvaizduoja realaus pasaulio objektus. Scheminis modelis turi turėti savyje verslo ir jo sferų supratimą. Šitame abstrakcijos lygyje kompiuterinės aparatūros ir/arba DB modelis gali buti nenustatytas ir jis turi buti nepriklausomas nuo kompiuterinės aparatūros ir nuo programinės įrangos, kad sistema galėtų dirbti bet kokioje, vėliau parinktoje, platformoje.

Minimalių duomenų taisyklė:

Visa, ko reikia - yra, ir visa, kas yra - reikalinga.

Visi duomenų elementai, reikalingi DB tranzakcijoms, turi būti apibrėžti modelyje, ir visi duomenų elementai, apibrėžti modelyje, turi buti panaudoti bent vienoje iš tranzakcijų. Tačiau reikia apibrėžti ne tik duomenis, reikalingus dabar versle, bet ir duomenis, kurių reikės vėliau (ateityje), paliekant vietos būsimoms modifikacijoms ir papildymams ir užtikrinant tęstinumą.

Duomenų analizė ir reikalavimų parinkimas

Sekantis scheminio projektavimo žingsnis yra apibrėžti duomenų-komponenčių detales. Duomenų-komponenčių dalys yra tai, kas gali buti transformuota į atitinkamą informaciją. Projektuotojo pastangos turi buti susikoncentruotos į:

- *Informacijos reikalingumą*. Kokios rūšies informacijos reikia? Kokį išvedimą (atsakymą) turi sugeneruoti sistema?
- *Informacijos šaltinius*. Kur informacija turi būti randama? Kaip informacija turi buti paimta, kai ji jau surasta?
- *Informacijos vartotojus*. Kas naudosis šią informacija? Kaip informacija turi būti naudojama? Kokie yra įvairių vartotojų požiūriai į duomenis?
- *Informacijos striktūrą*. Kokių duomenų reikia tam,kad sudaryti informaciją? Kokie yra duomenų atributai? Kokie ryšiai egzistuija tarp duomenų? Kokia yra duomenų apimtis? Kaip dažnai duomenys yra vartojami? Kokios duomenų transformacijos turi buti panaudotos tam, kad sukurti reikiamą informaciją?

  Projektuotojui rekia atkreipti dėmesį į tokius aspektus:
- *vartotojų požiūrių į duomenis formavimas ir surinkimas*.
- *esamos sistemos tiesiogniai stebėjimai: egzistuojantis ir reikalaujami išvedimai*.
- *pastovus kontaktas su sistemos projektavimo grupe* (DB projektavimas yra SVGC dalis. Tačiau kartais sistemos analitikas, kuris projektuoja naują sistemą, plečia schematišką DB modelį (tai dažniausiai atsitinka mikrokompiuterių aplinkoje). Kitais atvejais, DB projektavimas laikomas kaip DB administratoriaus darbas, administratorius projektuoja DB pagal specifikacijas, sukurtas sistemos analitikų).

Tam, kad sudaryti tikslų duomenų modelį, projektuotojas turi turėti pilnus ir pilnai suprantamus kompanijos duomenis. Bet vien tik duomenys neduoda pilno verslo supratimo. Duomenų rinkinys yra tik objektas, kurį veikia **verslo taisyklės**. Šitos tasyklės - tai išdėstytas specifikuotos verslo aplinkos aprašymas, jos sudaro ir vykdo veiksmus verslo aplinkoje. Verslo taisyklės aprašo pagrindinias kompanijos duomenų charakteristikas.

Sistemų projektavime verslo taisyklės yra naudingos:

- padeda standartizuoti kompanijos požiūrį į duomenis;
- sudaro bendravimo tarp vartotojų ir projektuotojų priemonę;
- duoda projektuotojui galimybes nustatyti duomenų tipus, vaidmenis, galimybes;
- duoda projektuotojams galimybes suprasti verslo procesą;
- duoda projektuotojui galimybes apibrėžti atitinkamas ryšių taisykles ir išorinius raktus (nustatyti, ar duotas ryšys yra būtinas ar ne, dažniausiai yra verslo taisyklių funkcija).

Esybių ryšių modeliavimas ir normalizacija

Prieš sukuriant esybių-ryšių (E-R) modelį, projektuotojas turi nustatyti atitinkamus projektavimo dokumentacijos standartus. Standartuose aprašomi diagramų ir simbolių naudojimai, dokumentacijos rašymo stiliai, maketai ir kitos savybės. Tačiau nesėkmės dokumentacijos standartuose dažnai reiškia nesėkmes vėlesnėse komunikacijose, o tada ir blogą projekto darbą.

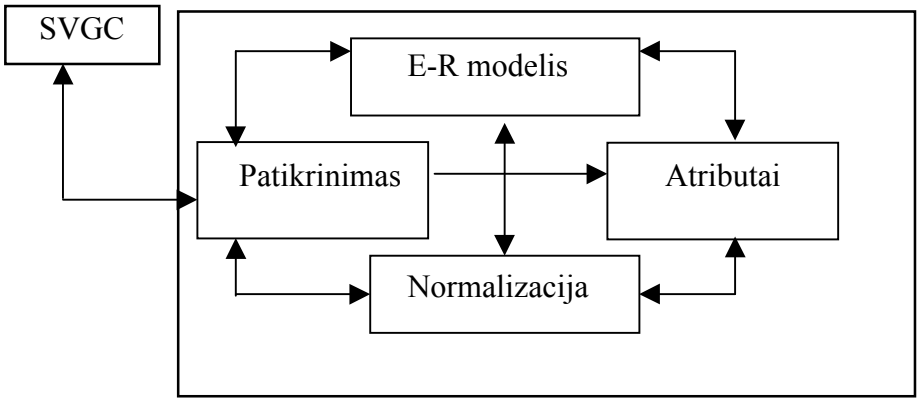
Verslo taisyklių apibrėžimo ir suprantamo modelio sudarymo procesas, naudojantis E-R diagramas, gali būti aprašytas taip:

1. išnagrinėtos ir apibrėžtos galimos verslo taisyklės;
2. turėti pagrindinių vartotojų patikrintas ir apjungtas verslo diagramas;
3. nustatyti pagrindiniai faktai, kurie yra reikalingi kompanijai, jie taps sistemos pagrindinėmis esybėmis;

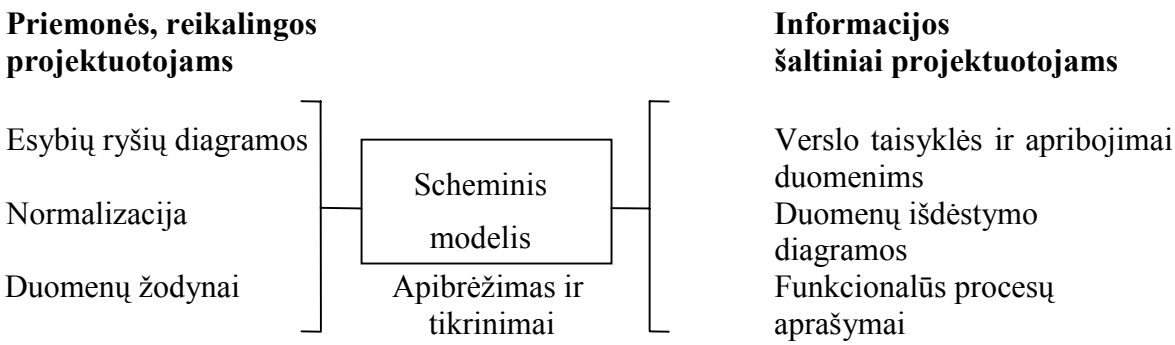
4. apibrėžti ryšiai tarp esybių.
5. kiekvienai esybei apibrėžti raktai ir atributai;
6. normalizuotas modelis, garantuojantis duomenų vientisumą.

Visi objektai (esybės, atributai, ryšiai ir t.t.) turi būti užrašyti ar apibrėžti duomenų žodyne. Pastarasis naudojamas normalizacijos procese, kad padėtų pašalinti duomenų anomalijas ir įvairias problemas. Per šį procesą projektuotojas turi:

- apibrėžti esybes, atributus, ryšius ir išorinių raktų taisykles;
- susitarti dėl elgesio su daugiareikšmiais atributais;
- susitarti dėl išorinių raktų 1:1 santykiuose;
- išvengti nebūtinų trigubų ryšių;
- nubrėžti atitinkamas E-R diagramas;
- normalizuoti duomenų modelį;
- įtraukti visus duomenų elementų apibrėžimus į duomenų žodyną



E-R modeliavimas kaip iteratyvus procesas



Koncepcinio modeliavimo priemonės ir informacijos šaltiniai

Duomenų modelio tikrinimas

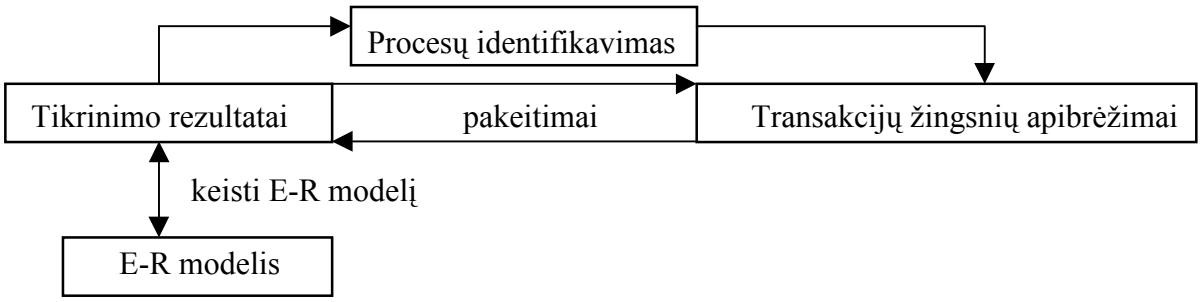
E-R modelis turi buti patikrintas prieš patvirtinant visus sistemos procesus. Tam reikia atlikti visą seriją testų, vertinamų:

- galutinio vartotojo požiūriu į duomenis ir jiems taikomas transakcijas (SELECT, INSERT, UPDATE, DELETE operacijas, kitas užklausas);
- priėjimo keliais, duomenų saugumo ir operacijų išlygiagretinimo kontrolės požiūriu (išlygiagretinimo kontrolė yra savybė, kuri leidžia prieiti prie DB vienu metu keliems vartotojams, išsaugant duomenų vientisumą);
- Verslo duomenų iškeltų reikalavimų ir apribojimų požiūriu.

E-R modelio patikrinimo procesas turi būti atliekamas tokia veiksmų seka (**pastaba**: žemiau minimas **modulis** yra informacinės sistemos komponentė, kuri atlieka specifines funkcijas):

E-R modelio tikrinimo procesas

1. identifikuoti modelio pagrindines esybes;
2. identifikuoti kiekvieną modulį ir jo komponentes;
3. identifikuoti kiekvieno modulio procesus;
4. patikrinti visus modelio procesus;
5. padaryti visus būtinus pakeitimus, nurodytus 4-ame žingsnyje;
6. pakartoti 2-ą - 5-ą žingsnius visiems moduliams.



Tikrinimo procesas pradedamas išrenkant centrinę esybę. Tokia esybė apibrėžiama, atsižvelgiant ar ji dalyvauja daugumoje sisteminių modulių, ir ar ji yra svarbi daugumoje sisteminių modulių. Praktiškai apibrėžiant centrinę esybę, projektuotojas išrenka esybes, turinčias daugiausia ryšių.

Sekantis žingsnis yra modulio arba posistemės, kuriai priklauso centrinė esybė, identifikavimas ir modulių apribojimų ir akiračio nustatymas. Kai kiekvienas toks modulis yra apibrėžtas, centrinė esybė yra nagrinėjama jo struktūros požiūriu, kad koncentruotis į modulio detales.

Centrinės esybės/modulio struktūros požiūriu turi būti padaryta:

- Užtikrinta modulio ryšių **tvirtumas** (cohesivity). Šis terminas apibrėžia ryšių tarp modulio esybių tvirtumą. Modulis turi turėti stiprius ryšius, t.y. esybės turi būti stipriai susietos ir modulis turi būti išbaigtas ir pakankamas;
- Analizuota kiekvieno modulio ryšiai su kitais moduliais (kad patikrinti modulių sujungimus). Terminas '**modulių sujungimas**' rodo modulių tarpusavio nepriklausomybės laipsnį. Moduliai tarpusavyje turi turėti silpnus sujungimo laipsnius, juos galima interpretuoti kaip nepriklausomus vienus nuo kitų. Silpnas sujungimo laipsnis mažina nebūtinas tarpmodulines priklausomybes ir leidžia kurti tikslias modulines sistemas, vengti nebūtinų ryšių tarp esybių.

Procesai gali būti klasifikuojami atsižvelgiant į jų:

- vykdymo dažnumą (kasdien, savaitėmis, mėnesiais, metais ir t.t.)
- operacijos tipą (INSERT ar ADD, UPDATE ar CHANGE, DELETE, užklausos ir ataskaitos, paketinis apdorojimas, priežiūra, rezervinės ir t.t.)

Užbaigus tokius tikrinimus, scheminis (koncepcinis) modelis yra pilnai apibrėžtas, tuo lygiu, kas jis yra nepriklausomas nuo aparatūros (hardware) ir programinės įrangos (software) parinkimo.

DBVS programinės įrangos išrinkimas

Tai yra sudėtinga operacija, todėl programinės sistemos pranašumai ir nuostoliai turi būti kruopščiai išnagrinėti. Kad išvengti nemalonumų, galutinis vartotojas turi žinoti apie DB ir DBVS apribojimus. Nors programinės įrangos išrinkimo faktoriai, kuriais vadovaujamąsi, priklauso nuo kiekvienos kompanijos atskirai, galima išskirti kelis pagrindinius faktorius:

- kainos (pirkimo, palaikymo, instaliavimo , licenzijų ir t.t.)

- DBVS savybių (kai kurių DBVS programinė įranga turi daug priemonių, kurios lengvina darbo užduotį, kaip ekrano navigacija, pranešimai, duomenų žodynai, kitos padeda sukurti daug patogesnę darbo aplinką; administravimo patogumas, užklausų vartojimo lengvumas, saugumas, išlygiagretinimo kontrolė, tranzakcijų valdymas irgi apsisprendžia DBVS programinės įrangos parinkimą);
- naudojamas duomenų bazių modelis: herarchinis, tinklinis, reliacinis, dar koks nors;
- portabilumas(platforma, operacinė sistema, programavimo kalba);
- DBVS keliami reikalavimai kompiuterinei įrangai.

Loginis projektavimas

Loginis projektavimas yra jau po DB modelio (hierarchinio, tinklinio, reliacinio) nustatymo, jis taikomas išrinktam modeliui ir yra priklausomas nuo programinės įrangos. Loginis projektavimas reiškia sheminio projekto pervedimą į vidinį modelį. Į loginį reliacinės DB modelį įeina lentelių, indeksų, virtualių lentelių, tranzakcijų projektavimas.

Loginio projektavimo etape yra nustatomos teisės vartotojams ir kitiems asmenims naudotis DB. Kas galės naudotis lentelėmis ir kokios jų dalys bus prieinamos kokiems vartotojams? Atsakymas į tokius klausymus reikalauja aatitinkamų priėjimo teisių apibrėžimų.

Trumpai sakant, loginis projektavimas perveda nuo programinės įrangos nepriklausomą scheminą modelį į priklausomą modelį, nustatant atitinkamų sričių apibrėžimus, reikiamas lenteles ir sričių apribojimus, kompiuterinės įrangos platformą, fizinius reikalavimus, kurie leis sistemai funkcionuoti su išrinktąja kompiuterine aplinka.

Fizinis projektavimas

Fizinis projektavimas – tai duomenų saugojimo ir duomenų priėjimo charakteristikų apibrėžimo procesas, kuris veikia duomenų saugojimo vietą įrenginyje ir sistemos našumą. Fizinį projektavimą geriau aprašyti kaip techninį darbą. Jis yra labai svarbus hierarchiniame ir tinkliniame modeliuose. Reliacinė DB yra daugiau izoliuota nuo fizinio detalių išdėstymo. Tačiau dėl skirtingų kompiuterių fizinių charakteristikų skirtumo, reliacinių DB našumas gali skirtis. Sistemos vykdymas gali būti paveiktas tokiomis charakteristikomis kaip išrinkimo būdas ir laikas, sektoriaus ir bloko dydžiai, ir t.t. Papildomai, indekso sukūrimas ar panašūs veiksmai gali būti reikšmingi reliacinės DB duomenų priejimo greičiui ir efektyvumui.

Reikalavimai duomenų tipams turi buti kruopščiai išnagrinėti, kad apibrėžti optimalų priėjimo metodą, nustatyti reikalingus duomenų dydžius. Kai kurios DB automatiškai išskiria atminties erdvę, kuri yra skirta DB pastoviems apibrėžimams ir duomenims saugoti.Tai užtikrina, kad duomenys bus užrašyti nuoseklia tvarka, kuri sumažina priėjimo prie duomenų laiką ir padidina sistemos efektyvumą.

Implementavimas ir Pakrovimas

Reliacinėse DBVS, pvz. DB2, duomenų bazių implementavimas reikalauja saugojimo grupės, lentelių erdvės ir lentelių sukūrimo. Lentelių erdvė gali tureti daugiau nei vieną lentelę.

Loginio projektavimo realizavimui reikia:

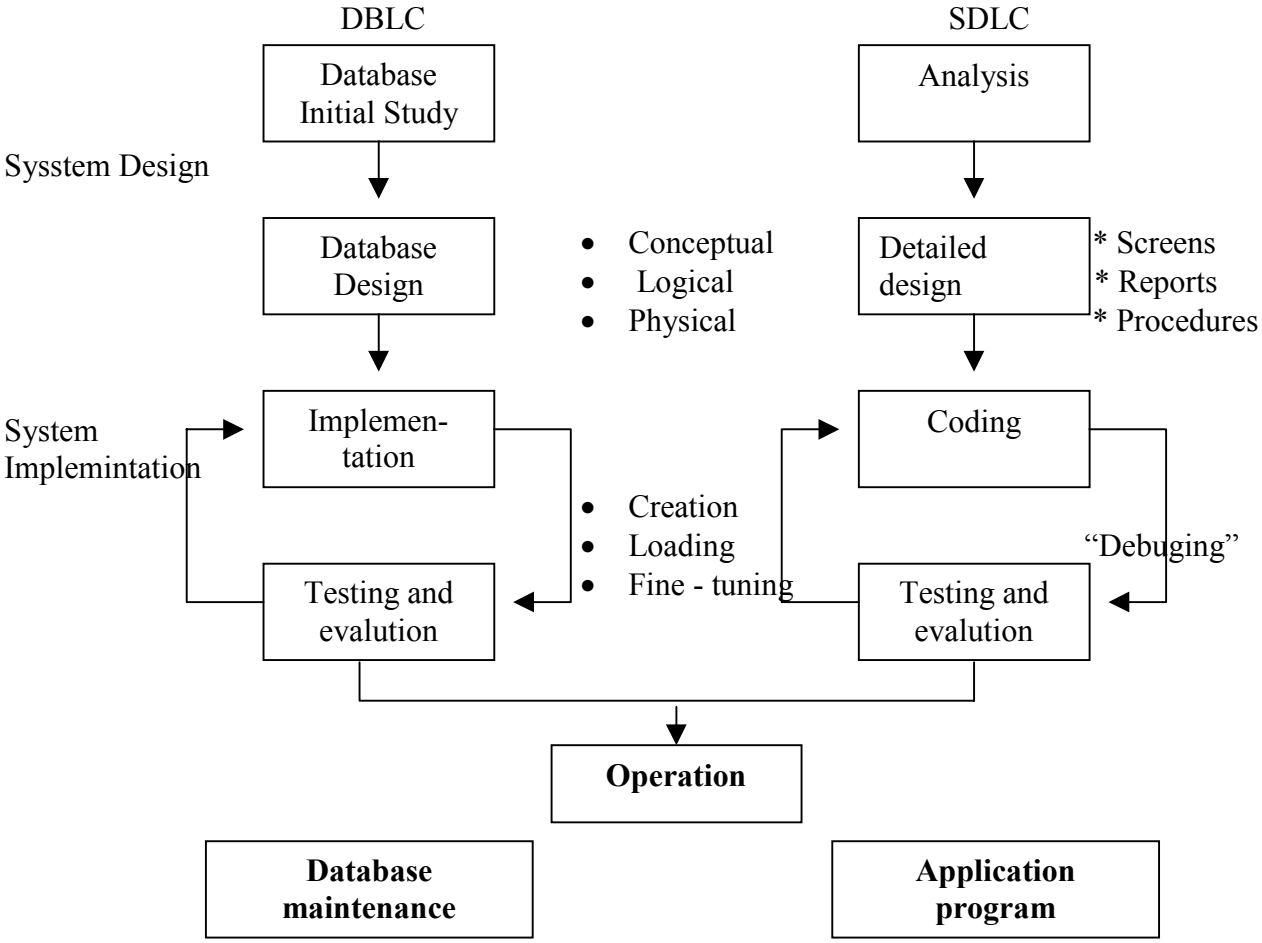
- Sukurti duomenų bazės saugojimo grupę;
- Saugojimo grupėje sukurti DB;
- Suteikti teises duomenų bazės administratoriui;
- Duomenų bazėje sukurti lentelių erdvę arba erdves;
- Lentelių erdvėje (erdvėse) sukurti lentelę (lenteles);
- Priskirti vartotojams priėjimo teises prie lentelių erdvių ir lentelių konkrečiose erdvėse.

Kai DB jau sukurta, į jos lenteles turi būti įvesti duomenys. Jei duomeys saugomi kitame formate, nei to reikalauja nauja DBVS, tai prieš pakrovimą jie turi būti konvertuojami.

Jei naudojama herarchinė DB, tai jos realizavimas turi turėti duomenų apibrėžimo kalbos (DDL) komponentę kiekvenai DB. Be to, hierarchinė DB reikalauja programos specifikacijos blokų (PSB) sukūrimo, kuris leis programoms pasiekti DB.

Tinklinio modelio naudojimas reikalauja duomenų bazės DDL sukūrimo, kuris turi aprašyti visų įrašų tipus, vietos atmintyje nustatymo būdus, įterpimo būdus, aibes, tvarką, ir t.t. Tinklinėms DB reikia techninės įrangos valdymo kalbos (DMCL) ir visų būtinų poschemių (subschemas), kad programos leistų pasiekti DB.

Lygiagretūs veiksmai, kurie vyksta SVGC ir DBGC:



Kiti realizavimo ir pakrovimo etapai

Našumas: DB našumas yra vienas iš svarbiausių faktorių kai kurių DB darbe ir realizavime.Tačiau ne visos DBVS turi vykdymo valdymo ir tikslaus pritaikymo (fine-tuning) priemones savo programinėje įrangoje, todėl kartais sunku vertinti jų našumą.

Apsauga. Duomenys, saugomi kompanijos DB, turi būti apsaugoti nuo neteisėtų vartotojų naudojimo. Tam naudojami:

- **Fizinė apsauga** leidžia fiziškai pasiekti zonas, atviras tiktai personalui su p**riėjimo** teise. Priklausomai nuo duomenų bazės įgyvendinimo tipo, įrangos fizinės apsaugos sistemos ne visada yra praktiškos;
- **Slaptažodžiai;**
- **Priėjimo teisės** sukuriamos naudojant DB programinę įrangą, jų priskyrimas gali apriboti operacijas (CREATE, UPDATE, DELETE ir t.t.) su tokiais objektais kaip DB, lentelės, vaizdai, užklausos ir ataskaitos;
- **Audito seansai** skirti patikrinti DBVS nuo nesankcionuoto įsiveržimo. Nors tokie seansai veikia po veiksmo įvykdymo, jų egzistavimas dažnai yra tinkama priemonė neutralizuoti neteisėtų įsiveržimų rezultatus;
- **Duomenų kodavimas;**

- **Bediskės darbo stotys** leidžia galutiniams vartotojams pasiekti DB, be galimybės pakrauti duomenis iš serverių.

Kiti svarbūs faktoriai:

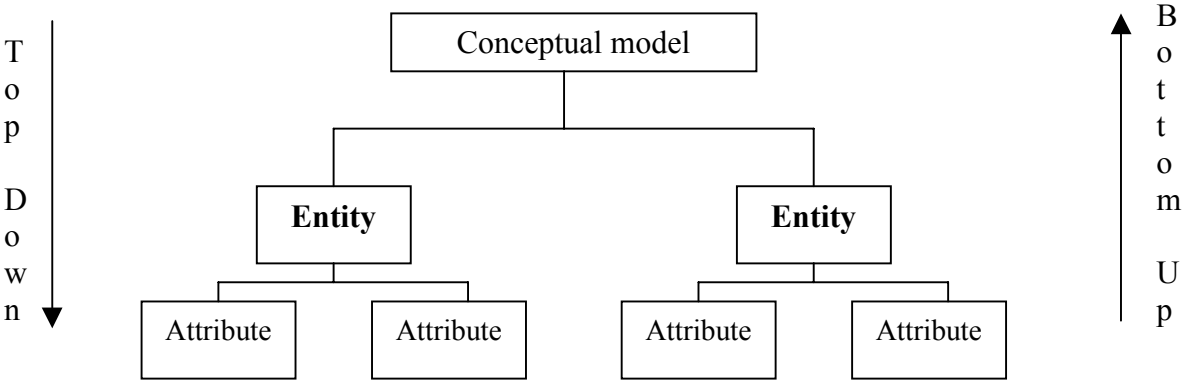
- **Backup and Recovery**
- **Integrity**
- **Company standarts**
- **Concurrency control**
- **Testing and Evaluation**
- **Operation**
- **Maintenance and Evolution**

Speciali pastaba apie DB projektavimo strategijas

Du klasikiniai DB projektavimo būdai.

1. Nuo viršaus į apačią: projektavimas pradedamas identifikuojant duomenų rinkinius, po to apibrėžiant duomenų elementus kiekvenam iš rinkinių. Vėliau šis procesas apima esybių nustatymą, jų skirtingus tipus, ir kiekvienos esybės atributų apibrėžimą.
2. Nuo apačios į viršų: projektavimas pradedamas pirmiausia nustatant duomenų elementus, po to juos apjungiant į duomenų rinkinius. Kitaip tariant, pirmiausia apibrėžiami atributai, o po to apjungiami į grupes, kad suformuoti esybes.

Pirmo ar antro būdo pasirinkimas projektavimo procese dažnai priklauso nuo užduoties apimamų temų, specifikos, arba ir nuo asmeninio pasirinkimo. Nors šie būdai yra daugiau vienas kitą papildantys, o ne prieštaraujantys, antro būdo pasirinkimas yra labiau produktyvus mažoms duomenų bazėms su mažu esybių, atributų, santykių ir tranzakcijų skaičiumi. Situacijoms, kuriose esybių, santykių, ir tranzakcijų skaičius yra didelis, jų įvairumas ir sudėtingumas pasireiškia daug labiau, pirmas būdas yra efektyvesnis. Daugelis kompanijų turi susikūrę vidinius standartus informacinių sistemų projektavimui, vystymui ir duomenų bazių kūrimui.



Object-Oriented Databases

In the late 1980s and early 1990s database experts increasingly faced complex data requirements that were difficult to handle with standard relational technology. The size of the databases and their changing composition-the database now might include graphics, video, and sound, as well as numbers and text-invited a reorganization of existing information systems. This reorganization effort led to emerging database technologies based on object-oriented concepts.

Object-oriented (OO) systems are usually associated with **applications** that draw their strength from **intuitive** graphical user interfaces, powerful modeling techniques, and advanced data-management capabilities. Since OO technology is an important contributor to the evolution of database systems, there are of interest such questions as:

- What basic OO concepts govern OO systems?
- What effect are OO concepts likely to have on data modeling?

- How are OO features related to the more traditional relational and E-R models?
- What are the basic features of an OO database management system (OODBMS)?
- How are database implementation features affected by OO concepts?
- What effect will OO concepts have on database design?
- What are the pros and cons of OO systems?

Object Orientation And Its Benefits

Object orientation means different things to different people. The definition of OO:

Object Orientation

A set of design and development principles based on the idea of conceptually autonomous structures. Each autonomous structure represents a real-world entity with the ability to interact with itself and with other objects.

Although object orientation has become the subject of intensive research efforts, consistent OO concept definitions appear to be elusive. There seem to be three main reasons for the failure to agree on basic definitions:

- OO concepts are still relatively foreign to the majority of computer users;
- The broad applicability of OO concepts has stretched their reach far beyond the initial programming-languages target;
- OO concepts were developed by many people in different places at different times. The relational model's standard-bearer was Dr. Codd; object-oriented concepts lack an equivalent guru.

Modularity is one of the primary goals of structured programming and good design. The conceptually autonomous structure, makes the much sought-after modularity almost inevitable. Therefore, object orientation is likely to be the key to many complex programming and design solutions. The table below illustrates just some of the benefits of object orientation in a variety of computer-related areas and suggests a broad applicability of object-oriented (OO) concepts.

Unfortunately, the lack of OO standards creates uncertainties that make businesses wary of it. Yet, in spite of the absence of precise and generally accepted standards, it's useful to develop acceptable definitions of fundamental OO concepts. And these concepts will pave the way to a reasonably precise terminology to build a useful framework for understanding.

The Evolution Of Object-Oriented Concepts

Object-oriented concepts stem from **object-oriented programming (OOP),** which was developed as an alternative to traditional programming methods. Before OOP, data and procedures were isolated from each other. Programmers were trained to identify data sources, to group data into files or tables, to establish relations and constraints, and to write the procedures required to produce a given output. Such a programming environment causes the data to be the **passive component**, while the procedures that manipulate the data become the **active component**.

The rigid distinction between data and procedure was encouraged by the use **of procedural** languages. The programmer invokes an application, which then uses data to produce information. In stark contrast, in an OOP environment the programmer asks objects to perform operations on themselves.

OO concepts first appeared in programming languages such as Ada, Algol, LISP, and SIMULA. Each of these programming languages set the stage for the introduction of more refined

OO concepts, which were subsequently expanded by successors. As of the time of this writing, Smalltalk and C++ are the two dominant **object-oriented programming languages (OOPL).** Actually, Smalltalk and C++ differ substantially in terms of the level of OO inclusion; Smalltalk represents a purer OOP environment, while C++ is essentially a variant of the C language that supports OO extensions.

Table: Object Orientation's Benefits

COMPUTER-REIATED AREA	OO BENEFITS
Programming language	Reduces the number of lines of code
	Decreases development time
	Enhances code reuseability
	Makes code maintenance easier
	Enhances programmer productivity
Graphical user interfaces (G.U.I.)	Enhances ability **to** create easy-to-use interfaces
	Improves system user-ftiendliness
	Makes it easier to define standards
Databases	Supports abstract data types
	Supports complex objects
	Supports multimedia databases
Design	Captures more of the data model's semantics
	Represents the real world better
Operating systems	Enhances system portability, thus improving
	systems interoperability

OOPLs were developed to provide a more natural environment for software programmers. The main objectives of OOPLs were:

- To provide an easy-to-use software-development environment;
- To provide a powerful software modeling tool used for applications prototyping;
- To decrease development time by reducing the amount of code and by making that code reuseable, thereby improving programmer productivity.

The adoption of OOP changes not only the way in which programs are written but also how those programs behave. Keep in mind that **the object-oriented view of the world endows data with manipulative ability.** Consequently, the OO environment has several important attributes:

- **The data set is no longer passive;**
- **Data and procedures are bound together, creating an** OBJECT;
- **The object has an innate ability to act on itself.**

In effect, an object appears to have a life of its own and can interact with other objects to create a system:

- Because such lifelike objects carry their own data and code, it becomes easier and more natural to produce reuseable modular systems.
- It is precisely this lifelike characteristic that makes OO systems natural to those with little programming experience but confusing to many whose traditional programming expertise causes them to split data and procedures.
- Given the lifelike nature of object orientation, it is not surprising that OO notions became much more viable with the advent of personal computers.
- It is also not surprising that OO concepts' impact on programming has an effect on a host of other computer-based activities, including those based on databases.

Object-Oriented Concepts

As it was noted earlier, OO concepts have their roots in OOPLS, which are often regarded as languages created mainly for programmers by programmers, who tended to program in their own way in their own world. Perhaps this is yet another of the reasons why OOPLs have not yet conformed to universally accepted standards.

Objects: Components and Characteristics

In OO systems everything is dealed with is an object, whether it be a student, an invoice, an airplane, an employee, a service, a menu panel, a report, and so forth. Some objects are tangible and some are not. Formally, we may define an object within the OO environment this way:

Object

An object is an abstract representation of a real-world entity that has a **unique identity**, **embedded properties**, and the **ability** to interact with other objects and itself.

The difference between object and the entity concept is a lack of manipulative ability for entity. Other differences later.

Object Identity

The most revealing part of an object is its identity. The object's identity is represented by an **Object** ID **(OID),** which is **unique** to that object; no two objects can share the same OID. The OID is assigned by the system at the moment of the object's creation and cannot be changed under any circumstance.

Do not confuse the relational model's primary key with an OID. In contrast to the OID, a primary key is based on **user-given values** of selected attributes and can be changed at any time. The OID is assigned by the system, does not depend on the object's attribute values, and cannot be changed. **The OID can be deleted only if the object is deleted, and the same OID can never be reused.** Moreover, the unique OID is not tied to a physical address in permanent memory (disk), as were the records of past hierarchical and network database systems. This characteristic allows an OO system to maintain physical data independence.

Attributes (Instance Variables)

Objects are described by their attributes, referred to as **instance variables** in an OO environment. Each attribute has a unique name and a data type associated with it. Traditional data types, also known as **base data types,** are used in most programming languages and include real, integer, string, and so on.

Object attributes may reference one or more other objects. For example, the attribute MAJOR refers to a Department object, the attribute *COURSES-TAKEN* refers to a list (or collection) of Course objects, and the attribute ADVISOR refers to a Professor object.

At the implementation level, the OID of the referenced object is used to link both objects, thus allowing the implementation of relationships between two or more objects. The **attribute also can contain the OID of an object that contains a list of objects**; such an object is known as a **collection object.**

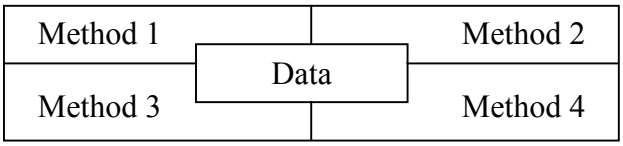
Note: The difference between the relational and OO models is also at this point. In the relational model a table's attribute may contain only a **value** that may be used to **join** rows in different tables. **The OO model does not need such JOINs to relate objects to one another.**

Object State

The **object state** is the set of values that the object's attributes have at a given time. Although the object's state can vary, its OID remains the same. If we want to change the object's state, we must change the values of the object attributes. To change the object's attribute values, we must send a **message** to the object. This message will invoke a **method.**

Messages and Methods

Messages and methods are organized as if the object is to be a nutshell. The nutshell's nucleus (the nut) represents the object's data structure, and the shell represents its methods.



Object X

Every operation to be performed on an object must be implemented by a **method.** Methods are used to change the object's attribute values or to return the value of selected object attributes. **Methods represent real-world actions**. In effect, methods are the equivalent of procedures in traditional programming languages. In OO terms, **methods represent the object's behavior**.

Every method is identified by a **name** and has a **body**. The body is composed of computer instructions written in some programming language, to represent a real-world action.

Methods can access the instance variables (attributes) of the object for which the method is defined.

To invoke a method it is necessary to send a **message** to the object. A message is sent by specifying a **receiver object,** the name of the method, and any required parameters. **The internal structure of the object cannot be accessed directly** by the message **sender**, which is another object. Denial of access to the structure ensures the **integrity** of the object's state and hides the object's internal details. The ability to hide the object's internal details (attributes and methods) is known as **encapsulation.**

An object may also send messages to **change or interrogate** (apklausti) another object's state. Interrogation implies asking for the interrogated object's instance variable value(s). To perform such object-change and interrogation tasks, the method's body can contain references to other object's methods (send messages to other objects).

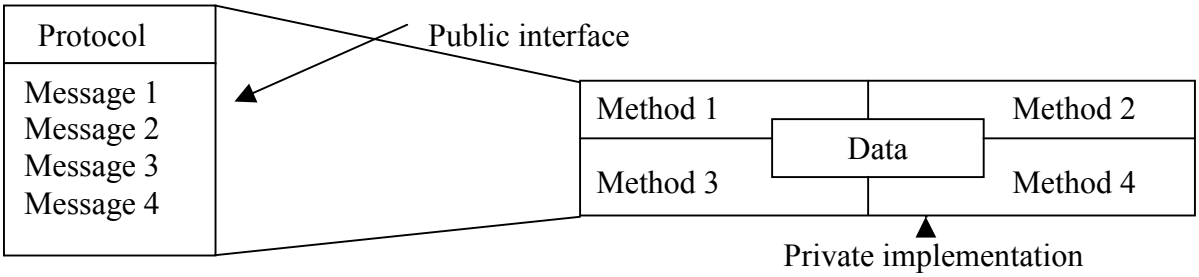
Classes

OO systems classify objects according to their similarities and differences. Objects that share common characteristics are grouped into **classes.** In other words, a class is a collection of similar objects with shared structure (attributes) and behavior (methods).

A class contains the description of the data structure and the method-implementation details for the objects in that class. Therefore, all objects in a class share the same structure and respond to the same messages. In addition, a class acts as a 'storage bin' for similar objects. Each object in a class is known as a **class instance** or **object instance**.

Protocol

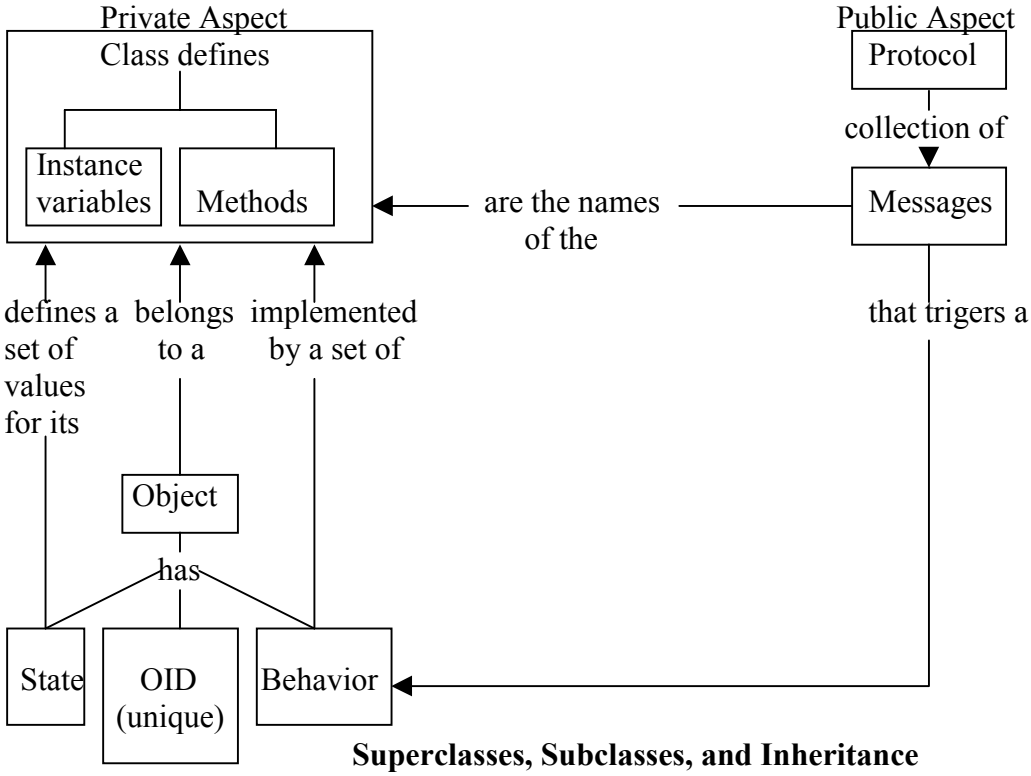
The class's collection of messages, each identified by a message name, constitutes the object or class **protocol.** The protocol represents an object's **public aspect,** that is, it is known by other objects as well as end users. In contrast, the implementation of the object's structure and methods constitutes the object's **private aspect**.



Usually, a message is sent to an object instance. However, it is also possible to send a message to the **class** rather than to the **object.** When the receiver object is a class, the message will invoke a **class method**. One example of a class method is Smalltalk's method ***new***. Using Smalltalk, the ***new*** class method is triggered by the message ***new*** to create a new object instance (with a unique OID) in the receiver class. Because the object does not exist yet, the message ***new*** is sent to the class and not to the object.

What is the difference? Actually, none: in an OO system a class is also an object!

OO Summary: Objects characteristics



Superclasses, Subclasses, and Inheritance

Classes are organized into a **class hierarchy** if each class has only one parent class. The class hierarchy is known as a **class lattice** if its classes can have multiple parent classes. Generalization is used to classify objects into classes of objects that share common characteristics (the generalization **automobile** includes large luxury sedans as well as compact cars, and the generalization **government** includes federal, state, and local governments).

The class hierarchy concept introduces a powerful OO concept known as **inheritance.** Inheritance is the innate ability of an object within the hierarchy to inherit the data structure and behavior (methods) of the classes above it.

In pure OO systems like Smalltalk, all objects are derived from the superclass Object. Therefore, all classes share the characteristics and methods of the superclass Object. The inheritance of data and methods goes from top to bottom in the class hierarchy. Two variants of inheritance exist: **single** and **multiple inheritance.**

Single Inheritance. Single inheritance exists when a class has only one immediate (parent) superclass above it. Most of the current OO systems support single inheritance.

When the system sends a message to an object instance, the entire hierarchy is searched for the matching method, using the following sequence:

1. Scan the class to which the object belongs.
2. If the method is not found, scan the superclass.

The scanning process is repeated until

1. The method is found, or
2. The top of the class hierarchy is reached without finding the message. The system will then generate a message to indicate that the method was not found.

Multiple Inheritance. A class may be derived from several parent superclasses located one level above that class.

The assignment of instance variable or method names must be treated with some caution in a multiple-inheritance class hierarchy. For example, if using the same name for an instance variable or method in each of the superclasses, the OO system must be given some way to decide which method or attribute to use. The OO system, however, cannot make such value judgements and may:

1. Produce an error message in a pop-up window explaining the problem.
2. Ask the end user to supply the correct value or to define the appropriate action.
3. Yield an inconsistent or unpredictable result.

To solve the multiple-inheritance conflicts, some OO systems implement the support for multiple inheritance through the use of user-defined **inheritance rules** for the subclasses in the class lattice. These inheritance rules govern a subclass's inheritance of methods and instance variables.

Method Overriding and Polymorphism

It’s possible to override a superclass's method definition by redefining the method at the subclass level.

Polymorphism allows different objects to respond to the same message in different ways. Polymorphism is a very important feature of OO systems because its existence allows objects to behave according to their specific characteristics.

In OO terms, polymorphism means that

- The **same** name may be used for a method defined in different classes in the class hierarchy.
- The user may send the **same message** to different objects that belong to different classes and yet always generate the correct response.

Polymorphism thus augments method override to enhance the code reuseability so prized in modular programming and design.

Abstract Data Types

Classes provide the means for another important OO property: the ability to use **abstract data types** (ADT). **A data type** describes a set of objects with similar characteristics.

All conventional programming languages make use of a set of predefined data types, known as **conventional data types** or base data types. Base data types include real, integer, and string or character. Base data types are subject to a **predefined set of operations.** For example, the integer base data type allows operations such as addition, subtraction, multiplication, and division.

Conventional programming languages also include type constructors, the most common of which is the **record** type constructor. For example, a programmer can define a CUSTOMER record type by describing its data fields. The CUSTOMER record represents a new data type that will store CUSTOMER data, and the programmer may directly access such data structure by referencing the record's field names. A record data type allows operations such as WRITE, READ or DELETE. However, new operations cannot be defined for base data types.

Like conventional data types, abstract data types also describe a set of similar objects. However, an abstract data type (ADT) differs from a conventional data type because:

- The ADT's operations are **user-defined.**
- The ADT does not allow direct access to its internal data representation or method implementation. In other words, the ADT **encapsulates its** definition, thereby hiding its characteristics.

Pure OO systems such as Smalltalk implement base data types as ADTs. To define an ADT or class one has to define:

1. Its *name.*
2. The data representation or instance variables of the objects belonging to the class or ADT. Each instance variable has a data type that may be a base data type or another ADT.
3. The ADT or class operations and constraints, both of which are implemented through methods.

It is to note that the terms **abstract data type and class** are used as synonyms. Some OO systems differentiate between class and type, using type to refer to the **class data structure and methods and class** to refer to the **collection of object instances**. A type is a more static concept, while a class is a more run-time concept. In other words, when defining a new class, actually a new type is defined. The type definition is used as a pattern or template to create new objects belonging to a class at run-time.

Abstract data types together with inheritance provide support for **complex objects.** A complex object is formed by combining other objects in a set of complex relations. An example of such a complex object might be found in a security system that uses different data types, such as:

- Conventional (tabular) employee data such as name, phone, date of birth, and so on.
- Bitmapped data to store the employee's picture.
- Voice data to store the employee's voice pattern.

The ability to deal relatively easily with such a complex data environment is what gives OO credibility in today's database marketplace.

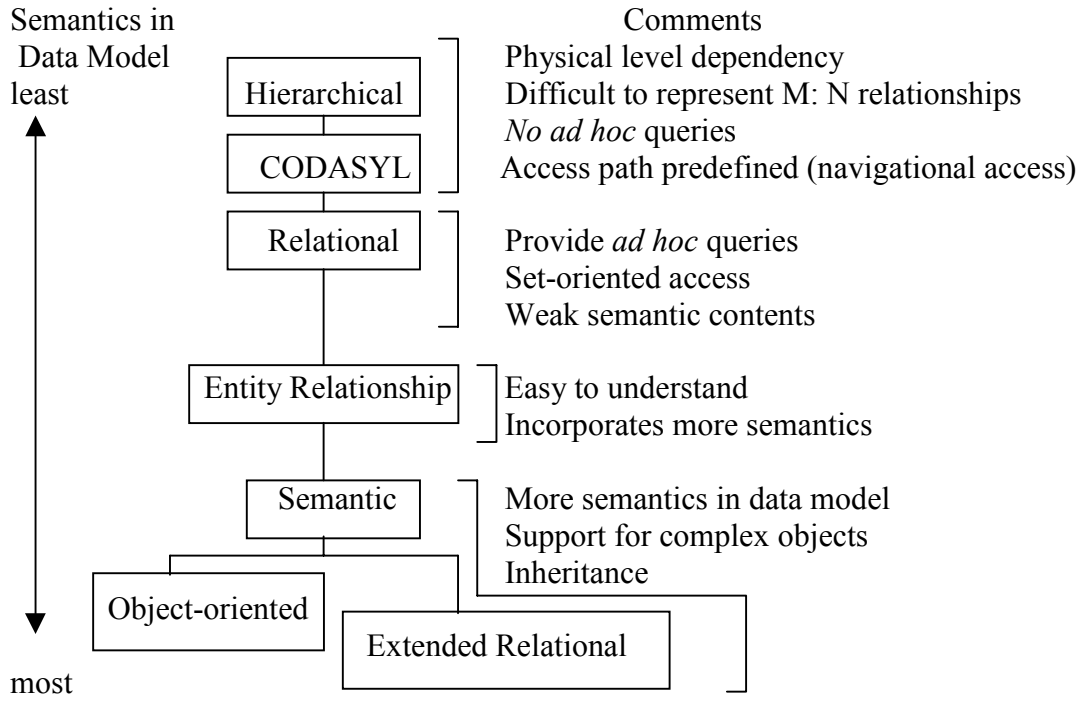
As for the basic OO concepts it’s easy to create a class hierarchy by using Smalltalk/V, from Digitalk Inc., Los Angeles, CA. Smalltalk provides an OOP environment that **does not** provide for persistent storage of objects, so it is not an **OO database** management system. However, Smalltalk is especially suited to OO modeling work because it provides a class library, through which a programmer may build new ADTs or classes easily.

The Evolution Of Data Models

The evolution of database management systems has always been driven by the search for new ways of modeling increasingly complex real-world data. This search has yielded many proposed data models, yet few have been implemented, and even fewer have achieved commercial success.

Each new data model capitalized on the shortcomings of previous models. The network (CODASYL) model replaced the hierarchical model because the former made it much easier to represent complex (many-to-many) relationships. In turn, the relational model offered several advantages over the hierarchical and CODASYL models through its simpler data representation, superior data independence, and relatively easy-to-use query language. Although the debate about the relative merits of the older hierarchical and CODASYL models and the newer relational model lasted several years, the relational model emerged as the dominant database model for business applications.

The Entity Relationship model, appearing after the relational model, introduced an easy-to-use graphical data representation, thus becoming the database design standard. However, no commercial DBMS based on the E-R model has appeared.



The Development of Data Models

As more intricate real-world problems were modeled, the need for a different data model that even more closely represented the real world became evident. The Semantic Data Model (SDM), developed by M. Hammer and D. McLeod, was able to capture more meaning from the real-world objects; that is, it incorporated more **semantics** into the data model and introduced such concepts as class, inheritance, and so forth, which helped to model real-world objects more effectively.

In response to the increasing complexity of **applications**, two new data models have emerged: the **Object-Oriented Data Model (OODM)** and the **Extended Relational Data Model (ERDM).** The OODM is based on concepts derived from OOPLs and has gained strength during the past several years. The ERDM, championed by relational-database researchers, constitutes the relational model's response to the OODM challenge.

The current OODM-ERDM battle for dominance in the **future** database marketplace seems remarkably **similar** to the one waged by the hierarchical and CODASYL models against the relational model more than a decade ago. The OODM and ERDM are similar in the sense that each attempts to address the demand for more semantic information to be incorporated into the model. However, the OODM and the ERDM **differ** substantially both in terms of underlying philosophy and the nature of the problem to be addressed. The ERDM is based on the relational data model's concepts, while the OODM is based on the OO and Semantic Data Model concepts. The ERDM is primarily geared to business applications, while the OODM focuses on very specialized engineering and scientific applications.

It is of interest to examine the evolution of data models and to determine some common characteristics that data models must have in order to be widely accepted:

1. A data model must show some degree of **conceptual simplicity** without compromising the semantic completeness of the database. It does not make sense to have a data model that is more difficult to conceptualize than the real world.
2. A data model must represent the real world as closely as possible. This goal is more easily realized by adding more **semantics** to the model's data representation. (Semantics concern the dynamic data **behavior**, while the data representation constitutes the static aspect of the real world scenario).
3. The representation of the real-world transformations (behavior) must be in compliance with the consistency and integrity characteristics of any data model.

Characteristics Of An Object-Oriented Data Model

The main problem in defining the OO data model is that **there is no standard OO data model**. Therefore, most researchers find it easier to describe some minimum set of characteristics that a data model must have before it can be considered an OO data model. At the very least, an OO data model:

1. Must support the representation of **complex objects**.
2. Must be **extensible.** That is, it must be capable of defining new data types as well as the operations to be performed on them.
3. Must support **encapsulation**; that is, the data representation and the method's implementation must be hidden from external entities.
4. Must exhibit **inheritance**. An object must be able to inherit the properties (data and methods) of other objects.
5. Must support the notion of **object identity** (OID) described earlier in this chapter.

A quick summary will help in reading the subsequent material more easily:

1. The OODM models real-world entities as **objects**.
2. Each object is composed of **attributes** and a set of **methods**.
3. Each attribute can reference another object or a set of objects.
4. **Encapsulation** means that the attributes and the methods implementation are hidden from other objects.
5. Each object is identified by a unique **object ID (OID)**, which is independent of the value of its attributes.
6. Similar objects are described and grouped in a **class** that contains the description of the data (attributes or instance variables) and the implementation of the methods.
7. Classes are organized in a **class hierarchy**.
8. Each object of a class **inherits** all properties -of its superclasses in the class hierarchy.
9. The class describes a **type** of object.

Armed with this summary OO-component description, note the comparison between the OO and E-R model components:

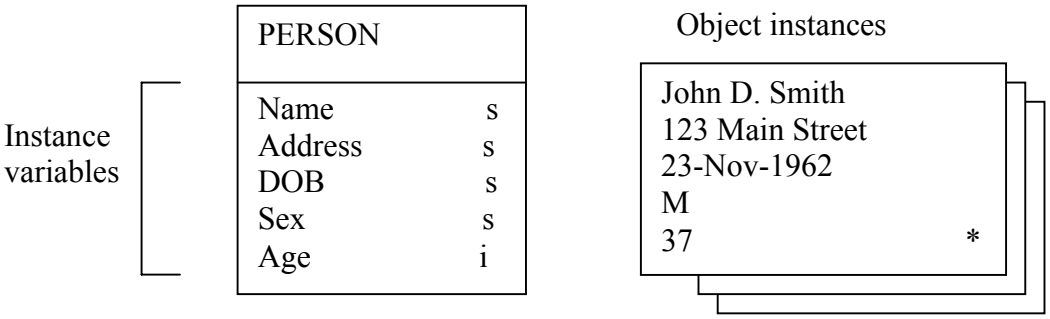
Comparing the OO and E-R Model Components

OO DATA MODEL	E-R MODEL
T'ype	Entity definition
Object	Entity
Class	Entity set
Instance Variable	Attribute
N/A	Primary key
OID	N/A
Method	N/A
Class hierarchy	E-R diagram (database schema)

The Graphical Representation of Objects: Object Diagrams

The currently published research projects contain several proposed graphical object representations. It is elected to base the following presentation on the work published by Dr. Setrag Khoshafian (1986, 1990, 1996) and that of Drs. Hammer and McLeod (1981).

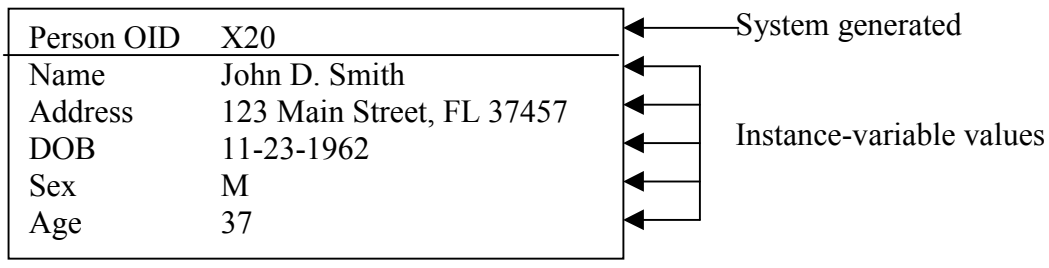
A graphical representation of an object resembles a box, with the instance-variable names inside the box. Generally speaking, the object representation is shared by all objects in the class. Therefore, the terms **object and class** are often used interchangeably in the illustrations.



s – string data type, I – integer data type, * - derived attribute

Shared Representation for All Objects of the Class Person

Next, to examine the **state** of a Person **object instance**, the note is, that the AGE instance variable can also be viewed as a **derived** attribute. Derived attributes may be implemented through **methods**. For instance, a **method** named Age for the Person class could be created. This method will return the difference in years between the current date and the date of birth (DOB) for a given object instance. Quite aside from the fact that methods may generate derived attribute values, **methods have the added advantages of encapsulation and inheritance.**



State of a Person Object Instance

It is necessary tp keep in mind that the OO environment allows to create abstract data types from base data types. For example, the NAME, ADDRESS, and DOB are composite attributes that can be implemented through classes or ADTS.

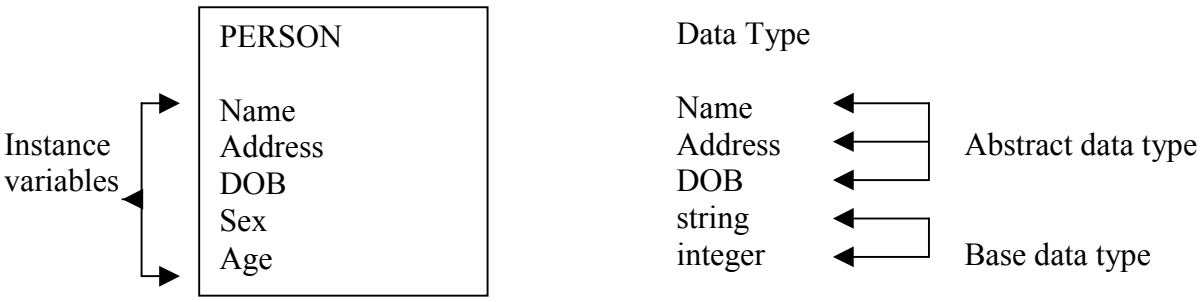
Name	
First_name	s
Middle_Init	s
Last_name	s

Address	
Street_num	s
Street	s
Apartment	s
City	s
State	s
Zip	i

DOB	
Day	I
Month	I
Year	i

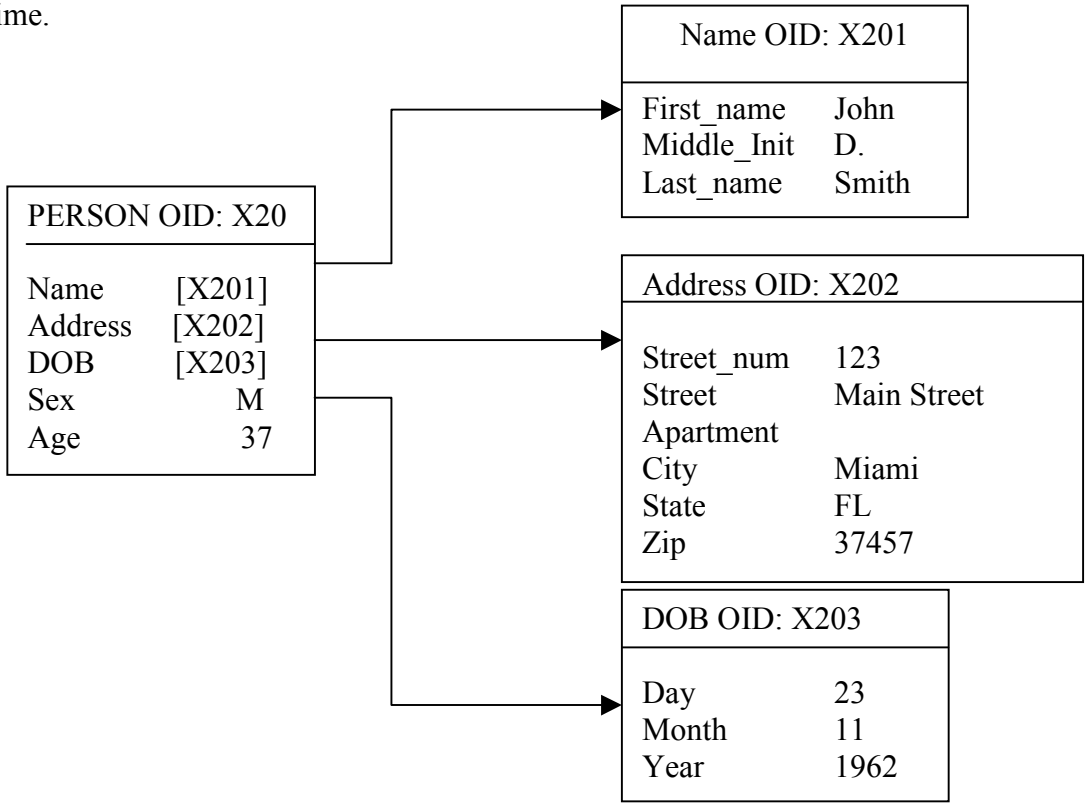
Defining Three Abstract Data Types

Note that **the Person class will now contain attributes that point to objects of other classes or abstract data types.** The new data types for each instance variable of the class Person are shown in next figure.



Object Representation for Instances of the Class Person with ADTs

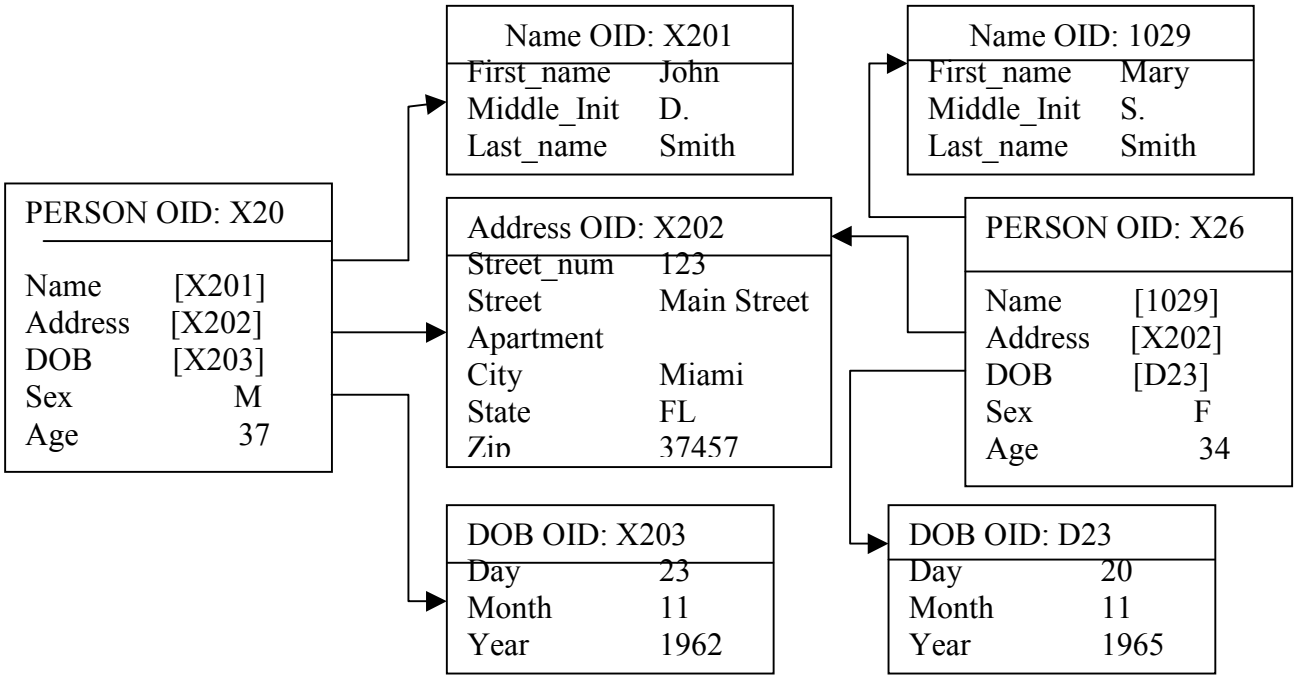
The **object space** or **object schema** is the equivalent of a database schema at a given time. The object space is used to represent the composition of the state of an object at a given time.



The object's state for an instance of class Person is illustrated in figure below. While examining the picture note the use of OlDs to reference other objects. For example, the attributes NAME, ADDRESS, and DOB now contain an OID of an instance of their respective class or ADT instead of the base value. The use of OlDs for object references avoids the data-consistency problem that would appear in a relational system if the primary key value is changed by the end user when changing the object's state. This is because the OID is independent of the object's state.

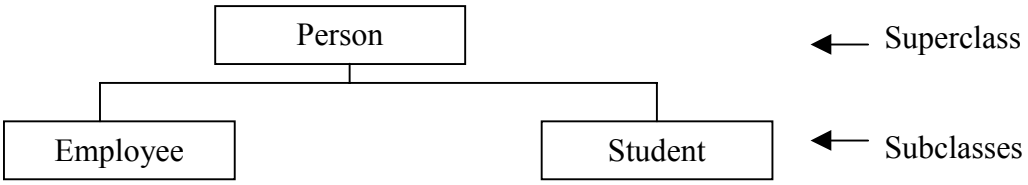
Object's State for an **Instance of the Class Person Using ADTs**

To illustrate this point further, it is to demonstrate that two persons who live at the same address are likely to reference the **same** Address object instance, rather than referencing two different Address object instances with equal object states. This condition is sometimes labeled as **referential object sharing:** a change in the Address object's state will be reflected in both Person instances.



Referential Sharing of Objects

Figure above illustrates the state of two different object instances of the class Person; both object instances reference the same Address object instance. Note that this figure depicts four different classes or ADTs: **Person** (two instances), **Name** (two instances), **Address**, and **DOB** (two instances).

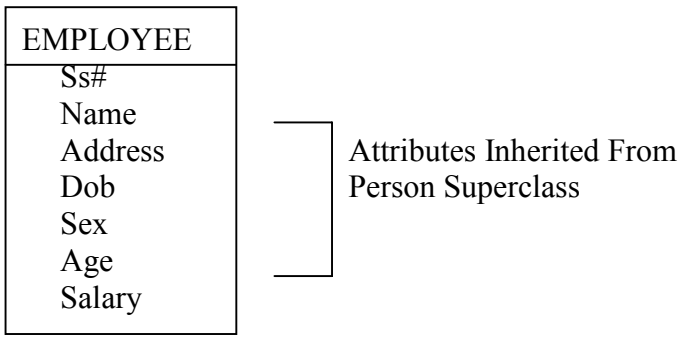


Class Hierarchy

Class-Subclass Relationships

Classes inherit the properties of their superclasses in the class hierarchy. This property leads to the use of the label **is a** to describe the relationship between the classes within the

hierarchy. That is, an employee **is a** person, and a student **is a** person. This basic idea is sufficiently important to warrant a more detailed illustration based on the class hierarchy depicted in the figure below.



Employee Object Representation

Within this hierarchy, the Employee object is described by two attributes: the social security number (SS#), recorded as a string base type, and the SALARY, recorded as an integer base type. The name, address, DOB, and age are all inherited from the Person superclass.

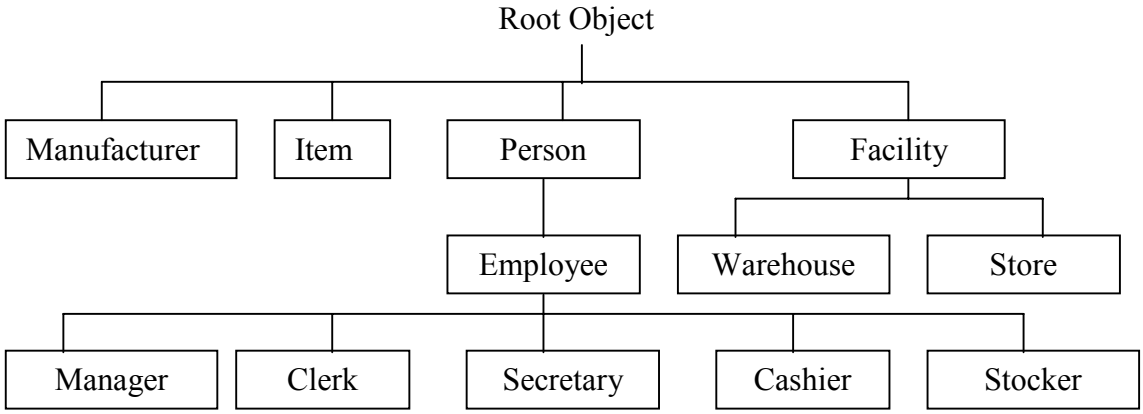
This example is based on the fact that the OODM supports the **class-subclass relationship**, for which it enforces the necessary integrity constraints. Note that the relationship between a subclass and a superclass is 1:1; that is, each subclass instance is related to only one superclass instance, if single inheritance is assumed.

Interclass Relationships: Attribute-Class Links

In addition to supporting the class-subclass relationship, the OODM supports **interclass relationships**. An interclass relationship is created when an attribute's data type is described with reference to another **abstract data type**. By defining the attribute data type the attribute is linked to a class or ADT.

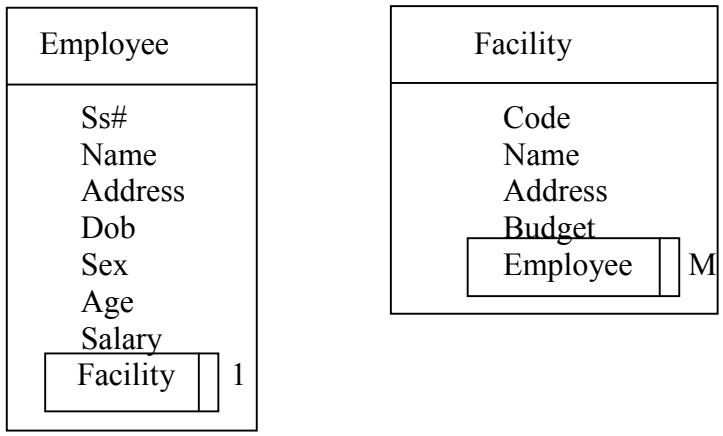
The interclass relationships are different from the class-subclass relationships explored before. To illustrate this difference, it is to examine the class hierarchy for the EDLP (Every Day Low Prices) Retail Corporation, shown in figure below.

Note that all classes are based on the Root Object superclass. The class hierarchy contains the classes Manufacturer, Item, Person, and Facility. The Facility class contains the subclasses Warehouse and Store. The Person class contains the subclass Employee, which in turn contains the subclasses Manager, Clerk, Secretary, Cashier, and Stocker. In the following discussion this simple class hierarchy will be used to illustrate basic 1:M and M:N relationships and to demonstrate the implementation of M:N relationships.



Class Hierarchy for the EDLP Retail Corporation

Representing I:M **Relationships.** Based on the figure's hierarchy above, a one-to-many relationship exists between Employee and Facility: Each Employee works in only one Facility, and each Facility has several Employees. Figure below shows how this relationship may be represented.



| | = mandatory participation 1,M = connectivity

Representing a 1:M **Relationship**

Examining the relationship between Employee and Facility portrayed above, note that the Facility object is included within the Employee object and vice versa; that is, the Employee object is also included within the Facility object. To examine the employee-facility relationship in greater detail, the following techniques may be used:

1. Related classes are enclosed in boxes to make relationships more noticeable.
2. The double line on the box's right side indicates that the relationship is **mandatory**.
3. **Connectivity** is indicated by labeling each box. In this case, a 1 is put next to Facility in the Employee object to indicate that each employee works in only one facility. The M beside Employee in the Facility object indicates that each facility has many employees.

Note that the E-R notation is used to represent a mandatory entity and to indicate the connectivity of a relationship (1:M). The purpose of this notation is to maintain consistency with earlier diagrams, to avoid confusion.

Rather than just including the object box within the class, it is better to use a name that is descriptive of the class **characteristic** that we are trying to model. Such a procedure is especially useful when two classes are involved in more than one relationship; in such cases it is suggested that the attribute's name has to be written above the class box and that the class box be indented to indicate that the attribute will reference that class.

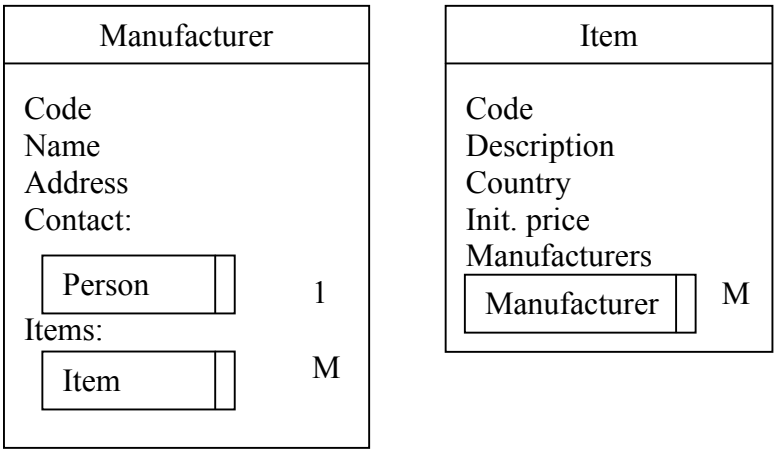
While examining the figure, note that the 1:M relationship is represented in both participating classes. This condition allows us to invert the relationship, if necessary. For example, we could add another Facility object within the Employee object to represent the 'Manager-of' relationship. In such a case, the Facility object would be optional and would have a connectivity of 1.

Another type of I:M relationship may be illustrated by examining the relationship between employees and their dependents. To establish this relationship it is first to create a

Dependent subclass using Person as its superclass. **Note that we cannot create a Dependent subclass by using Employee as its superclass because the class hierarchy represents an 'is a' relationship.** In other words, each Manager is an Employee, each Employee is a Person, each Dependent is a Person, and each Person is an Object in our object space; but each Dependent is **not** an Employee!

Then it is to note that Dependent is optional to Employee and that Dependent has a 1:M relationship with Employee. However, Employee is mandatory to Dependent. **The weak-entity concept disappears in the OODM because each object instance is identified by a unique OID.**

Representing M:N Relationships. Using the same EDLP Retail Corp. class hierarchy, a many-to-many (M:N) relationship may be illustrated by exploring the relationship between Manufacturer and Item represented in figure to follow. It depicts a condition in which each Item may be manufactured by many Manufacturers, and each Manufacturer may manufacture many Items. Figure below thus represents a **conceptual** view of the M:N relationship between Item and Manufacturer.



Representing an M:N Relationship

Also, note that the CONTACT attribute in Manufacturer class references only one instance of the Person class. A slight complication arises at this point: It is likely that each contact (person) has a phone number, yet a phone-number attribute was not included in the Person class. In this case, the designer may structure the attribute so that it will be available to all Person **subclasses.**

Representing M:N Relationships with an Intersection Class. Suppose that a condition to the just-explored M:N relationship is added to allow to keep track of additional data for each pair of object instances. For example, let's expand the relationship between Item and Facility so that each Facility may contain several Items and each Item may be located at several Facilities. In addition, we want to keep track of the quantity and location (aisle and row) of each Item at each Facility. Therefore, each Item instance may contain several occurrences of Facility, each accompanied by related values for appropriate attributes. The inverse case is true for each instance of Facility.

To translate the preceding discussion to a more **relational** view of such an M:N scenario, it is necessary to define an intersection (bridge) class to connect both Facility and Item and store the associated attributes. In this case, a Stocked-Item class might be created to contain the Facility and Item object instances and the values for each of the corresponding attributes. Such a class is equivalent to the Interclass-Connection construct of the Semantic Data Model. It is possible to show how the Item, Facility, and Stocked-Item object instances may be represented.

Late and Early Binding: Use and Importance

A very desirable OODM characteristic is its ability to let any object's attribute contain objects that define different data types (or classes) at different times. With this feature an object can contain a numeric **value** for a given instance variable, and the next object (of the same class) can contain a **character value** for the same instance variable. This characteristic is achieved through **late binding.** Through late binding the data type of an attribute is not known until execution time or run-time. Therefore, two different object instances of the same class can contain values of different data types for the same attribute.

In contrast to the OODM's ability to use late binding, a conventional DBMS requires that a base data type be defined for each attribute at the time of its creation. For example, suppose it is necessary to define an INVENTORY to contain the following attributes: ITEM-TYPE, DESCRIPTION, VENDOR, WEIGHT, and PRICE. In a conventional DBMS a table named INVENTORY is created and a base data type to each attribute is assigned.

When working with conventional database systems, the designer must define the data type for each attribute when **the table structure is defined.** This approach to data-type definition is called **early binding.** Early binding allows the database to check the data type for each of the attribute's values at compilation or definition time. For instance, the ITEM-TYPE attribute is limited to numeric values. Similarly, the **VENDOR** attribute may contain only numeric values to match the primary key of some row in a VENDOR table with the same numeric value restriction.

Now let's take a look to see how an OODM would handle this early-binding problem. As was true in the conventional database environment, the OODM allows the data types to be defined at creation time. However, quite unlike the conventional database, the OODM allows the data types to be user-defined ADTS. In this example of early binding, the **abstract** data types Inv-type, String-of-characters, Vendor, Weight, and Money are associated with the instance variables at definition time. Therefore, the designer may define the required operations for each data type. For example, the Weight data type can have methods to show the weight of the item in pounds, kilograms, and so on. Similarly, the Money data type may have methods to return the price as numbers or letters and denominated in U.S. dollars, German marks, or British pounds. (Remember that **abstract data types are implemented through classes.)**

In a late-binding environment, the object's attribute data type is not known prior to its use. Therefore, an attribute can have any type of value assigned to it. Using the same basic data set described earlier, the attributes (instance variables) INV-TYPE, DESCRIPTION, VENDOR, WEIGHT, and PRICE without a prior data-type definition.

Since no data types are predefined for the class instance variables, two different objects of the Inventory class may have different value types for the same attribute. For example, INV-TYPE can be assigned a character value in one object instance and a numeric value in the next one. Late binding also plays an important role in polymorphism, to allow the object to decide which method's implementation to use at run-time.

Support for Versioning

Support for **versioning** is another characteristic of an OODM. This feature allows to keep track of the history of changes of the state of an object. Versioning is thus a very powerful modeling feature, especially in computer-aided design (CAD) environments. For example, an engineer using CAD may load a machine-component design in his/her workstation, make some changes, and see how those changes affect the component's operation. If the changes do not live

up to expectations, (s)he can undo those changes and restore the component to its original state. Versioning is one of the reasons why the OODBMS is such a strong player in the CAD and computer-aided manufacturing (CAM) arenas.

OODM concepts are derived from intensive research done in several areas and by many different people. It would be difficult to mention all the academicians and commercial researchers who have made contributions in the development of OO concepts. However, it is necessary to give particular credit to the work of M. Hammer and D. McLeod in the **Semantic Data Model** (New York: ACM, 1981), which formally introduced such key concepts as classes, subclasses, and attribute inheritance. Other important work has been done by J. Banerjee, H. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou, and H. Kim in the development of the **ORION** OODBMS (New York: ACM, 1987).

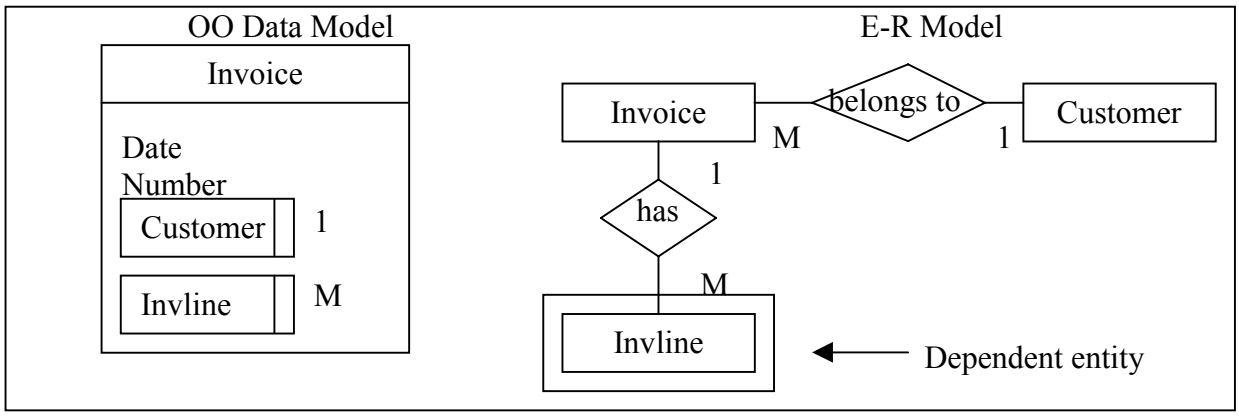
OODM And Previous Data Models: Similarities And Differences

Although the OODM has much in common with relational or E-R data models, OODM introduces some fundamental differences. The following summary is designed **to** make some detailed comparisons in order to clarify the OODM characteristics introduced in this chapter.

Object, Entity, and Tuple

The OODM concept of **object** moves well beyond the concept of **entity or tuple** in other data models. Although an OODM object resembles the entity and the tuple in the E-R and relational models, an OODM object has additional characteristics such as behavior, inheritance, and encapsulation. Such OODM characteristics make OO modeling much more natural than E-R and relational modeling. In fact, the E-R and relational models often force the designer to create new artificial entities in order to represent real-world entities. For example, in the E-R model an invoice is usually represented by two separate entities; the second entity is usually weak because its existence depends on the first entity.

Note that the E-R approach requires the use of two different entities to model a single real-world INVOICE entity. Such an artificial construct is imposed by the relational model's inherent limitations. The E-R model's artificial representation introduces additional overhead in the underlying system. In contrast, the OODM's Invoice object is directly modeled **as an object** into the object space or object schema.



An Invoice Representation

Class, Entity Set, and Table

The concept of class can be associated with the E-R and relational models'concept of **entity set** and **table**, respectively. However, class is a more powerful concept, which allows not

only the description of the data structure but also the description of the behavior of the class objects. A class also allows both the concept and the implementation of abstract data types in the OODM. The ADT is a very powerful modeling tool because it allows the end user to create new data types and use them as any other base data type that accompanies a database. The ADT thus yields an increase in the **semantic** content of the objects being modeled.

Encapsulation and Inheritance

ADT brings two other OO features that are not supported in previous models: encapsulation and inheritance. Classes are organized in class hierarchies. An object belonging to a class inherits all the properties of its superclasses. Encapsulation means that the data representation and the method's implementation are hidden from other objects and the end user. In an OODM only the methods can access the instance variables. In contrast, the conventional system's data components or fields are directly accessible from the external environment.

Conventional models do not incorporate the **methods** found in the OODM. The closest thing to methods is the use of triggers in a few SQL databases, but triggers do not yield the same functionality as methods.

Object ID

Object ID is not supported in either the E-R or the relational model. The hierarchical and the CODASYL models support some form of ID that may be considered similar to the OID, thus leading to the argument presented by some researchers who insist that the OO evolution is a step back on the road to the old pointer systems.

Relationships

The main property of any data model is found in its representation of relationships among the data components. The relationships in an OODM can be of two types:

- interclass references
- or class hierarchy inheritance.

The E-R and the relational models use a **value-based** relationship approach. Using a value-based approach means that a relationship among entities is established through a common value in one or several of the entity attributes. In contrast, the OODM uses the object ID, which is indentity-based, to establish relationships among objects, and **such relationships are independent of the state of the object.**

Access

A data model must provide efficient data access. The E-R and relational data models depend on the use of SQL to retrieve data from the database. SQL is an **ad hoc,** set-oriented query language that uses associative access to retrieve related information from a database, based on the value of some of its attributes.

As a consequence of having more semantics in the data model, the OODM produces a schema in which relations form part of the structure of the database. Accessing the structured object space resembles the record-at-a-time access of the old structured hierarchical and network models, especially if you use a 3GL or even the OOPL supported by the OODBMS. This is also true in relational database applications, in which programmers may revert to the use of cursors to compensate for the lack of 3GL support for set operations or because the data processing is essentially record-at-a-time, which is typical of traditional business transactions. Hopefully, the

mismatch between the programming language and the database language will be reduced in a future DBMS environment.

The OODM is suited to support both navigational and set-oriented access. The navigational access is provided and implemented directly by the OODM through the OlDs. The OODM uses the OlDs to navigate through the object space structure developed by the designer.

Associative set access in the OODM must be provided through explicitly defined methods. Therefore, the designer must implement operations to manipulate the object instances in the object schema. The implementation of such operations will have an effect on the performance and the database's ability to manage data.

Object-Oriented Database Management Systems

During the past few years the data-management and application environment has become far more complex than the one envisioned by the creators of the hierarchical, network, or relational DBMSes. For example, just think about some common current applications with complex data-management requirements such as:

- Computer-aided design (CAD) and computer-aided manufacturing (CAM). Such applications make use of complex data relations as well as multiple data types.
- Computer-assisted software engineering (CASE) applications, which are designed to handle very large amounts of interrelated data.
- Multimedia applications that use video, sound, and high-quality graphics that require specialized data-management features. An example of such an application is Geographic Information Systems (GIS).

Such complex application environments are often best served by an objectoriented database management system (OODBMS). The current generation of OODBMS combines object-oriented concepts with the traditional DBMS

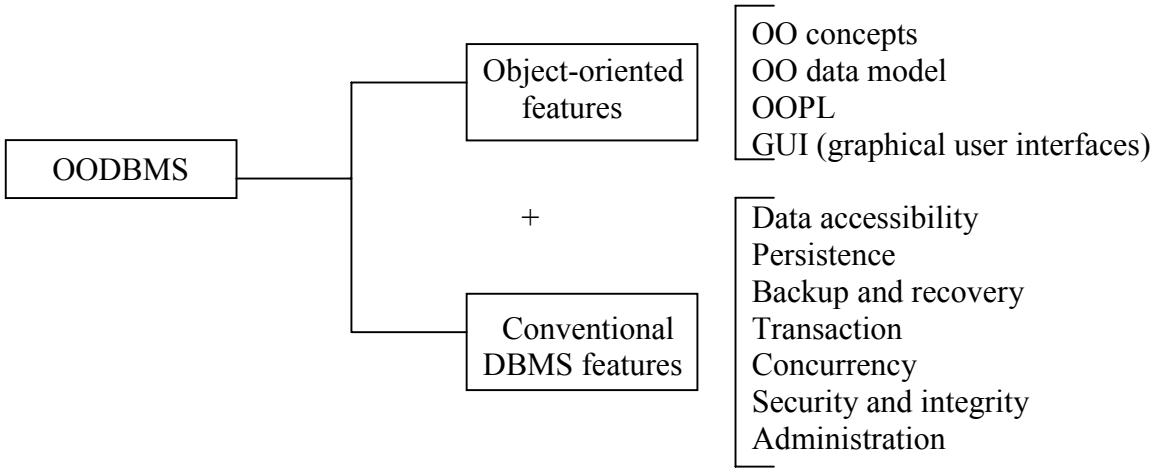
Many OODBMSes use a subset of the OO concepts and the OO data-model features explained earlier. The reason for the implementation of **subsets** is simple:At this writing there is no standard set of OODBMS features to be implemented. Instead, there is a large and growing collection of OO features from which to choose. Therefore, those who create the OODBMS tend to select the OO features that best serve the OODBMS's purpose, such as support for early or late binding of the data types and methods, and support for single or multiple inheritance. Whatever the choices, the critical factor for a successful OODBMS implementation appears to be finding the best mix of OO and conventional DBMS features that will not sacrifice the benefits of either one. Several OODBMS implementations currently exist in the research and commercial arenas, and each one has a different set of features. Examples of such OODBMSes include:

- "The O2 System," O.Deux et al., *Communications of the ACM* 34(10), October, 1991, pp. 35-48.
- "Object Store," C.Lamb et al., *Communications of the ACM* 34(10), October, 1991, pp. 51-63.
- "GemStone," P.Butterworth, A.Otis, and J.Stein, *Communications of the ACM* 34(10), October, 1991, pp. 65-77.

Common OO characteristics found in such OODBMS implementations are:

- The use of graphical user interfaces (GUIS) to manage the DBMS. OODBMSes come equipped with a GUI such as a **class hierarchy browser** to let the end user explore the classes contained in the design;
- The use of some kind of OOPL. The DDL, DCL, and DML commands are embedded in such a language;

- The language supported by the OODBMS is considered to be compurationally complete; that is, entire applications can be written in this language, and the end user does not need to learn two different languages to develop the application.



Object-Oriented Database Management Systems

Features of an Object-Oriented DBMS

The Thirteen OODBMS Commandments

Rules that make it an OO system:

Rule 1. The system must support complex objects.

Rule 2. Object identity must be supported.

Rule 3. Objects must be encapsulated.

Rule 4. The system must support types or classes.

Rule 5. The system must support inheritance.

Rule 6. The system must avoid premature binding.

Rule 7. The system must be computationally complete.

Rule 8. The system must be extensible.

Rules that make it a DBMS:

Rule 9. It must be able to remember data locations.

Rule 10. It must be able to manage very large databases.

Rule I 1. It must accept concurrent users.

Rule 12. It must be able to recover from hardware and software failures.

Rule 13. Data query must be simple.

There is no current OO standard to define a list of features that must be supported by a database before it can be considered an object-oriented database. Nevertheless, many attempts

have been made to procure a standard. The best-known and most thorough one is “The Object Oriented Database System Manifesto,” written in 1989 by Malcolm Atkinson, Francois Bancilhon, David DeWitt, Klaus Kittrick, David Maier, and Stanley Zdonik for the First International Conference in Deductive and Object Oriented Databases, in Kyoto, Japan. This represents the first comprehensive attempt to define a list of required OODBMS features.

The “Manifesto” illustrates thirteen mandatory features and other optional characteristics of OODBMS. The thirteen “commandments” are divided into two sets of rules: The first eight characterize an OO system, and the last five characterize a DBMS. The thirteen rules or commandments are listed in a table.

Rule 1. **The system must support complex objects**. It must be possible to construct complex objects from existing objects. Examples of such object constructors are sets, lists, and tuples, which allow the user to define aggregations of objects as attributes.

Rule 2. **Object identity must be supported**. The OID must be independent of the object's state. This feature allows the system to compare objects at two different levels: comparing the OID (identical objects) and comparing the object's state (equal or shallow equal objects).

Rule 3. **Objects must be encapsulated**. Objects have a public interface, but private implementation of data and methods. The encapsulation feature ensures that only the public aspect of the object is seen, while the implementation details are hidden.

Rule 4. **The system must support types or classes**. This rule leaves up to the designer to choose whether the system will support types or classes. Types are used mainly at compile time to check type errors in attribute value assignments. Classes are used to store and manipulate similar objects at execution time. In other words, class is a more dynamic concept, and type is a more static one.

Rule 5**. The system must support inheritance.** An object must inherit the properties of its superclasses in the class hierarchy. This property ensures code reuseability.

Rule 6**. The system must avoid premature binding**. This feature allows us to use the same method's name in different classes. The OO system decides which implementation to access at run-time, based on the class to which the object belongs. This feature is also known as **late binding or dynamic binding**.

Rule 7. **The system must be computationally complete**. The basic notions of programming languages are augmented by features common to the database data manipulation language (DML), thereby allowing to express any type of operations in the language.

Rule 8. **The system must be extensible**. The final OO feature concerns its ability to define new types. There is no management distinction between user-defined types and system-defined types.

Rule 9. **The system must be able to remember data locations.** The conventional DBMS stores its data permanently on disk; that is, the DBMS displays **data persistence.** OO systems usually keep the entire object space in memory; once the system is shut down, the entire object space is lost. Much of the OODBMS research has been focused on finding the way to permanently store and retrieve objects from secondary storage (disk).

Rule 10. **The system must be able to manage very large databases.** Typical OO systems limit the object space to the amount of primary memory available. For example,

Smalltalk cannot handle objects larger than 64K. Therefore, a critical OODBMS feature is to optimize the management of secondary-storage devices by using buffers, indexes, data clustering, and access-path selection techniques.

Rule 11. **The system must support concurrent users**. Conventional DBMSes are especially capable in this area. The OODBMS must support the same level of concurrency as conventional systems.

Rule 12. **The system must be able to recover from hardware and software failures**. The OODBMS must offer the same level of protection from hardware and software failures that traditional DBMSes provide; that is, the OODBMS must provide support for automated backup and recovery tools.

Rule 13. **Data query must be simple**. Efficient querying is one of the most important features of any DBMS. Relational DBMSes have provided a standard database query method through SQL, and the OODBMS must provide some similar capability.

There is neither a current nor a foreseeable standard-query-language norm for OODBMS. In fact, there are many OO pundits who dismiss the viability of such a query language for OODBMS because any query language is likely to violate the **encapsulation rule,** which states that the internal data representation can be accessed only through the object's methods.

The "Manifesto" addresses several additional, but **optional**, OODBMS features, including:

- Support for multiple **inheritance.** Multiple inheritance introduces greater complexity by requiring the system to manage potentially conflicting properties between classes and subclasses. The ***ORION*** OODBMS supports multiple inheritance by allowing the user to design inheritance rules for the object instances of the class hierarchy.
- Support for **distributed** OODBMSes. The trend toward systems-applications integration constitutes a powerful argument in favor of distributed databases. If the OODBMS is to be integrated seamlessly with other systems through networks, the database must support some degree of distribution. ONTOS DB from Ontologic is an example of an OO database that supports both data and process distribution in a client-server environment on different platforms, such as SUN, DEC, HP, and OS/2.
- Support for **versioning**. Versioning is a new characteristic of OODBMS that is especially useful in such applications as CAD or CAM. Versioning allows us to maintain a history that lets us keep track of all object transformations. Therefore, we can browse through all the different object states, in effect letting us walk back and forth in time.

Contrasting Traditional and Object-Oriented DBMSes

The most obvious difference between traditional and OO databases is derived from the object's ability to interact with other objects and with itself The OODBMS objects are an **active** database component. Conventional database systems assigned a passive role to the object's equivalent.

Another major difference concerns the support for object identity. OODBMSes are **identity-based** systems. In contrast, relational databases do not provide OID support; primary keys lack the permanence of the OID.

Support for abstract data types and encapsulation makes OODBMS more appealing for applications with complex data or multimedia extensions, such as CAD, engineering, or desktop publishing systems. ADTs allow support for **complex objects** - that is, objects created from other, lower-level objects. Although most relational databases do not provide support for complex

objects, some now include a few data-type extensions, such as **binary large object (BLOB), LONG VARCHAR, IMAGE, or PICTURE** data types to store and share images within the database.

Another distinguishing OODBMS feature is inheritance. Relational database systems do not provide inheritance of attributes and/or methods, as the OODBMS would provide. Inheritance increases code and data sharing within the system, thereby fostering reuseability.

It is not to leave the impression that the OODBMS has made other DBMSes obsolete. In fact, there are some database features in which traditional DBMSes have excelled in recent years. Such features include transaction and concurrency control, security, persistence, and query capabilities.

Relational database systems are very well established in business-transaction processing and concurrency control. The numerous and short sequential transactions that tend to exist in the business environment are handled well by conventional database systems, especially by relational DBMSes. OODBMSes are more appropriate in a transaction environment in which the number of transactions is low and the duration of transactions is large. In other words, OODBMSes are especially desirable in CAD, CASE, and engineering applications.

The relational DBMS and its immediate predecessors have also held their own in security and database administration. To date, most OODBMSes have been designed to support the type of applications that do not share the tight security and centralized control requirements that are typical of the business areas in which relational databases tend to operate.

Persistence is one of the most critical features of any database system. Database performance is, to an important extent, dependent on the efficiency with which the database handles persistent secondary storage. The OODBMS is not particularly adept at secondary-storage management, and for this reason most of the present OODBMS research efforts are focused on this area.

Finally, the OODBMS's weakest aspect appears to be its relative lack of support for **ad hoc** associative access. Relational databases have capitalized in this area: SQL makes it possible to query any relational database without having to learn an entirely new language for each one. OODBMSes are far from adopting any standard query language, and there exists some resistance to **ad hoc** querying because such activity violates the encapsulation concept. Partly in response to this limitation, some OODBMSes have adopted a different query approach: They modify SQL to support object extensions, thereby allowing **ad hoc** queries on class hierarchies that will yield the selection of object instances.

How Object Orientation Affects Database Design

A conventional relational-database design process involves the application of E-R modeling and normalization techniques to devel6p and implement a design. During such a design process, the emphasis is placed on modeling real-world objects through simple tabular relations, usually presented in 3NF. Unfortunately, we have already seen that the relational and E-R models sometimes cannot adequately represent some objects. Consequently, we were forced to introduce artificial constructs, such as bridge entities, that widen the semantic gap between the real-world objects and their corresponding representations.

You may have noticed that, generally, our database design process focused on the identification of the data elements, rather than including the data **operations** as part of the process. In fact, the definition of data constraints and data transformations is usually considered

late in the database design process. Such definitions are implemented within external-application program code. In short, operations are not a part of the database schema.

The object-oriented database design approach provides an answer to the often lamented data-procedures dichotomy by providing both the data identification and the procedures or data manipulations to be performed. However, if we do not make use of object-oriented languages in the implementation phase, even the inclusion of object-oriented *procedures* does not necessarily merit the label .object-oriented design' within an object-oriented system.

Why does the conventional model tolerate and even require the existence of the data-procedures dichotomy? After all, the idea of object-oriented design had been contemplated even in the classical environment. The reason is simple: Database designers simply had no access to tools that *bonded* data and procedures.

Object-oriented database design is the result of applying object-oriented concepts to the database design process. Object-oriented database design forces us to think of data and procedures as a self-contained entity, thereby producing a database that explicitly describes objects as a unit. Specifically, the OO design requires the database description to include the objects and their data representation, constraints, and operations. Such design features clearly produce a much more complete and meaningful description of the database schema than was possible in the conventional database design.

OO design is iterative and incremental in nature. The database designer identifies each real-world object, defines its internal data representation, semantic constraints, and operations. Next, the designer groups similar objects in classes and implements the constraints and operations through methods. At this point the designer faces two major challenges:

1. Build the class hierarchy or the class lattice (if multiple inheritance is allowed) using base data types and existing classes. This task will define the superclass-subclass relationships.
2. Define the interclass relationships (attribute-class links) using both base data types and ADTS.

The importance of this task can hardly be overestimated because the better the use of the class hierarchy and the treatment of the interclass relationships, the more flexible and closer to the real world the final model will be.

Code reuseability does not come easy. One of the hardest tasks in OODB design is the creation of the class hierarchy, using existing basic classes to construct new ones. Future DBAs will have to develop specialized skills to properly perform this task and to incorporate **code** to represent data behavior. Thus DBAs are likely to become surrogate database programmers who must define data-intrinsic behavior. The DBA’s role is likely to change when (s)he takes over some of the programming burden of defining and implementing operations that affect the data. Such a shift in procedures means that the programmer's work load is likely to decrease while the DBA’s work load increases.

Both DBAs and designers face additional problems: In contrast to the relational or E-R design processes, there are no computerized OODB design tools available to help design and document a database. An OODB design can be implemented in any database model. However, if the design is to be implemented in any of the conventional DBMSes, it must be translated carefully because conventional databases do not support abstract data types, non-normalized data, inheritance, encapsulation, and other OO features.

As is true in any of the object-oriented technologies, the lack of standards also affects OO database design. There is neither a widely accepted standard methodology to guide the design

process, nor a set of rules (like the normalization rules in the relational model) to evaluate the design.

OODBMS: PROS AND CONS

OODBMSes yield several benefits over conventional systems. Most of these benefits are based on the previously explored OO concepts. However, we think it appropriate to provide the following summary.

Pros

- ·OODBMSes allow the inclusion of more semantic information in the database, thus providing a more natural and realistic representation of real-world objects.
- ·OODBMSes provide an edge in the support for complex objects, which makes them especially desirable in specialized application areas. Conventional databases simply lack the ability to provide efficient applications in CAD, CAM, and multimedia environments.
- ·OODBMSes permit the extensibility of base data types, thereby increasing both the database functionality and its modeling capabilities.
- OODBMSes make **full** use of technological improvements in computers, especially cheaper and faster CPUs and memory capabilities. If the platform allows efficient caching, OODBMSes provide dramatic performance improvements compared to relational database systems.
- Versioning is a **useful** feature for specialized applications such as CAD, CAM, engineering, text management, and desktop publishing.
- ·The reuseability of classes allows for faster development and easier maintenance of the database and its applications.
- Faster application development time is obtained through inheritance and reuseability. This benefit is obtained only after mastering the use of the OO development features such as
- The proper use of the class hierarchy-for example, how to use existing classes to create new classes.
- OO design methodology.
- Strong market penetration in specialized engineering areas make OODBMS a technology-driven product that leads the development toward the nextgeneration DBMS.
- The OODBMS provides a possible solution to the problem of integrating existing and future DBMSes into a single environment. This solution is based on its strong data-abstraction capabilities and its promise of portability. We may speculate that future systems will manage objects with embedded data and methods, rather than records, tuples, or files. Although the portability details are not clear yet, they will have a major and lasting impact on how we design and use databases. As is true in any technological breakthrough environment, we still have a long way to go; but the train is moving out of the station.

Cons

- OODBMSes are based on a new technology that is still in its early growing phase and, therefore, lacks the maturity that defines a stable end-user environment. Business hates instability. As is true for any new market entrant, OODBMSes face strong opposition from the already-established players such as relational, hierarchical, and network database systems.
- The technology's novelty produces a lack of standards. In fact, the lack of standards may be the single most damaging factor when business management evaluates database technology. Not surprisingly, much OO effort is directed at standard development, and some standards are in fact starting to appear.
- The relational DBMS users emphasize the lack of a theoretical foundation for the object-oriented model.

- In some sense, OODBMSes are considered a throwback to the old pointer systems used by hierarchical and network models. This criticism is not quite on the mark when it associates the pointer system with the navigational datamanipulation style and fixed access paths that led to the relational system's dominance.
- OODBMSes do not provide a standard *ad hoc* query language as relational systems do. Some OODBMS implementations are beginning to provide extensions to the relational SQL to make the integration of OODBMS and RDBMS possible.
- Lack of standards, *ad hoc* queries, and development tools create hidden complexities in OODBMS development and make the penetration of OODBMS into the business market more difficult. Such limitations are especially cumbersome in the design of OO databases. The relational DBMS provided a comprehensive solution to the business database design-and-management needs, supplying both a data model and a set of fairly straightforward normalization rules to design and evaluate relational databases. OODBMSes do not yet provide a similar set of tools.
- The initial learning curve for OODBMS is steep. Learning to design and manage an OO database takes a considerable amount of time. If you consider the direct training costs and the time it takes to fully master the uses and advantages of object orientation, you will appreciate why OODBMSes seldom are rated as the first option when solutions are sought for noncomplex business-oriented problems. (Programmers are especially likely to resist change!)
- The OODBMS novelty, combined with its steep learning curve, means that there are few people who are qualified to make use of the presumed power of OO technology. Most of the technology is currently focused on engineering application areas of software development. Therefore, only companies with the right mix of resources (money, time, and qualified personnel) can afford to invest in OO technology.

How Oo Concepts Have Influenced The Relational Model

Most relational databases are designed to serve general business applications that require *ad hoc* queries and easy interaction. The data types encountered in such applications are well defined and are easily represented in common tabular formats with equally common short and well-defined transactions. However, RDBMSes are not quite as well suited as OODBMS to the complex application requirements found in CAD, CAM, engineering design, simulation modeling, architectural design, or pure scientific modeling. Also, the RDBMS is beginning to reach its limits in a business data environment that is changing with the advent of mixed-media data storage and retrieval.

The fast-changing data environment has forced relational-model advocates to respond to the OO challenge by extending the relational model's conceptual reach. The result of their efforts is found in the *Extended Relational Model,* examples of which are:

- *'The POSTGRES Next-Generation Database Management System,'* M. Stonebraker and G. Kemnitz, *Communications of the* ACM 34(10), October, 1991, pp. 78-92.
- *STARBURST* by IBM researchers at the IBM Research Almaden Center, San Jose, CA (Lohman et al. 1991).
- Dr. E. F. Codd's work at the IBM Research Laboratory, which gave the Extended Relational Model its theoretical underpinning (Codd 1979).

Our discussion will be based primarily on these three pieces of research, which add several significant new features to the relational model. Most of these features provide support for

- Extensibility of new user-defined (abstract) data types.
- Complex objects.
- Inheritance.
- Procedure calls (rules or triggers).
- System-generated identifiers (OID surrogates).

We do not mean to imply that we have just provided you an exhaustive list of all the extensions added to the relational model. Nor do we imply that all extended relational models incorporate all of the listed additions. However, most researchers appear to agree that the list contains the most crucial and desirable extended relational features.

The philosophy that guides the relational model's enhancements is based on the following concepts:

- Semantic and object-oriented concepts are necessary to support the new generation of applications.
- These concepts can be and must be added to the relational model.
- The benefits of the relational model must be preserved to protect the investment in relational technology and to provide downward compatibility.

Most current extended relational DBMSes conform to the stated philosophy; they also provide additional and very useful features:

- The POSTGRES RDBMS adds support for object and knowledge management. Object management provides support for complex objects such as multimedia, video, and bitmaps. Knowledge management provides support for data semantics, integrity constraints, and inference capabilities. POSTGRES permits the definition of classes and supports both single and multiple inheritance as well as user-defined data types and functions.

- The STARBURST RDBMS also permits the definition of user-defined types and functions, support for complex objects, encapsulation, and inheritance. Both POSTGRES and STARBURST use some form of system-generated and system-managed IDs or OID surrogates to relate objects to one another; at the same time, they maintain their *ad hoc* query capabilities. Examples of new data types supported by these types of systems include sets, lists, and arrays of tuples of one or several relations.

The Next Generation Of Database Management Systems

The adaptation of OO concepts in several computer-related areas has changed both systems design and system behavior. The next generation of DBMS is very likely to incorporate features borrowed from object-oriented database systems, artificial intelligence systems, expert systems, and distributed databases.

OODBMSes represent only one step toward the next generation of database systems. The use of OO concepts will enable future DBMSes to handle more complex problems with both normalized and non-normalized data. The extensibility of database systems is one of the many major object-oriented contributions that enable databases to support new data types such as Sets, lists, arrays, video, bitmap pictures, voice, map, and so on.

We do not believe that OODBMS will soon replace the relational DBMS as the standard for all, or even most, database applications. Instead, the OODBMS will carve its own niche within the database market. This niche will be characterized by applications that require very large amounts of data with several complex relations and with specialized data types. For example, the OODBMS is very likely to become a standard in CAD, CAM, computer-integrated manufacturing, multimedia applications, medical applications, architectural applications, and scientific applications.

We believe that the extended relational databases will become dominant in most complex business applications. This belief is based on the need to maintain compatibility with existing systems, the universal acceptance of the relational model as a standard, and the sheer weight of its considerable market share.

Multimedia Databases

Motivation: the study of multimedia databases is influenced by various applications and their exponential expantion in the real life.

Concepts: What is Multimedia and Hypermedia? A lot of definitions (the definitions below were given by Marmann and by Steinmetz).

Multimedia (by Marmann):

A multimedia system is a computer controlled integration of medial information objects of different types (text, images, audio, video,). The integration refers to:

- Data modeling
- Storage
- Presentation
- 'I'ime synchronization

A promise is that the media must be digitally represented, or at least digitally controllable.

Multimedia (by Steinmetz):

A multimedia system is to be defined as computer controlled, integrated

- generation,
- manipulation
- representation and
- communication

of independent information. This information is coded in at least one continuous (time dependent) and one discrete (time independent) medium.

Hypertext:

Text, that is not linear, but has a net structure with nodes and links. Link sources and link targets are called anchors.

Hypermedia:

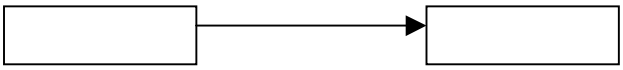
Nodes can contain any arbitrary media, not only text. This means that the linka from and to time dependent media are possible.

Link concepts:

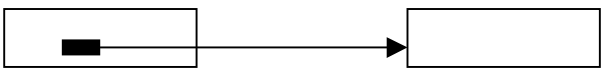
Links may be classified by various attributes. These include granularity, direction, functionality, locality, representation and dynamics of links.

1. **Anchor granularity:**

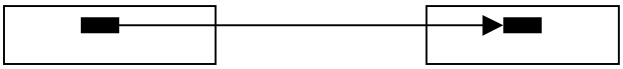
Node-to-Node-Links. They only allow complete hypertext nodcs as source and target nodes.



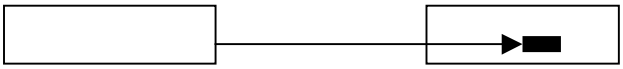
Span-to-Node-Links. They point from a single, free definable area within a node (a span) to a complete node.



Span-toSpan-Links. They admit spans within nodes as both, link source and link target.



Node-to-Span-Links. They point from a whole node to a span (used very rarely).



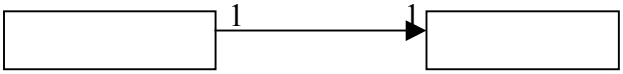
2. Direction of links:

onedirectional links: can be followed only from the source to the target

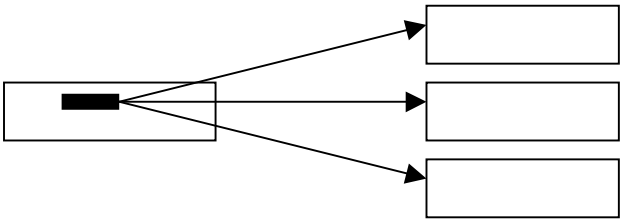
bidirectional links: **can be followed both ways**

3. Functionality of links:

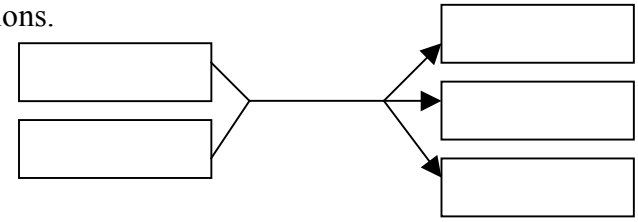
- 1:1 - link – only one source and one destination anchor (a clssical form of links)



- 1:M - link – starting with one source, several destinations can be reached

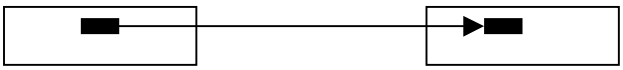


- N:M - link or MSMD-link (Multi-Source-Multi-Destination-Link): it has several sources and several destinations.

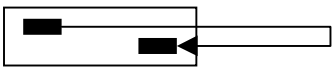


4. Locality of links:

- inter-document link



- intra-document link



5. Representation of links:

- implicit, i.e. the link does not exist as an entity of its own, but is defined implicitely in the document. (Example: in WWW, the link target is specified directly in the HTML document within the link source anchor)

- explicit, i.e. the link exists as an entity of its own, and attributes can be accorded to it. These attributes specify the link, e.g. author or type. In the document there exist only anchor-definitions. The link entity refers to these anchor-IDs.

6. Dynamic of links:

static (stored link), i.e. the link is persistently stored in the system (implicit or explicit).
dynamic (computer link), i.e. the link (resp. the link target) is computed during runtime. The system does not contain any specifIcation on the link. Dynamic links are uscd very often in WWW for database search on cgi-scripts (also compare to links in digital library systems including full text search; sometimes known in the literature as content based search).

Features of Multimedia Database Systems

- The multimedia database systems are to be used when it is required to administrate a huge amounts of multimedia data objects of different types of data media (optical storage, video tapes, audio records, etc.) so that they can be used (that is, efficiently accessed and searched) for as many applications as needed.
- The Objects of Multimedia Data are: text, images. graphics, sound recordings, video recordings, signals, etc., that are digitalized and stored.
- Multimedia Data are to be compared in the following way:

Medium	Elements	Configuration	Typical size	Time dependent	Sense
Text	Printable characters	Sequence	10 KB (5 pages)	no	visuall/ acoustic
Graphic	Vectors, regions	Set	10 KB	no	visuall
Raster image	Pixels	Matrix	1 MB	no	visuall
Audio	Sound/volume	Sequence	600 MB (AudioCD)	yes	acoustic
Video-Clip	Raster image/ graphics	Sequence	2 GB (30 min.)	yes	visuall

The need and efficiency of MM-DBS are to be defined by following requirements:

Basic service:

- to be used for multiple applications
- not applicable as a real end-user system (like program interface)

Storage and retrieval of MM-Data:

For the Storage:

- input of MM objects
- cmnposition (to multimedia objects) (example: authoring systems)
- archive of data (in hardware and format independent way)

For the Retrieval:

- support of complex search
- efficiency (indices etc.)
- evaluation (aggregation, filtering)
- preview
- also conversions (needed to gain or lead to hardware and format independence)

For the Update

- only replace or also edit? (the complexity depends on).

Multimedia Database Systems have to be capable:

1. Support of multimedia data types, i.e. data types as data structures, including type of data and operations

2. Capability to manage very numerous multimedia objects, store them and search for them

3. To include a suitable memory management system, to improve performance, high capacity, cost optimization

4. Database system features:
   - persistency
   - transaction concept
   - multi-user capability
   - recovery
   - ad-hoc queries
   - integrity constrains (which leads to cocsistency)
   - safety
   - performance

5. Information retrieval features:
   - attribute-based search
   - content-based search

Integrity Constrains for MM-DB Applications

The following features are typical for MMDB:

- Unique, Primary-key Constraints
- Referential integrity
  - via foreign keys (RM)
  - via OIDs (OO)
- Existential integrity
- NOT NULL constraints
- Integrity rules (check clauses)
- Trigger

Specifically for OO:

- Pre- and postconditions for methods
- Constraints of the class hierarchy
- Partition conditions (Disjointness constraints)

The Dexter Reference Model

The Dexter Hypertext Reference Model

The goal of reference models is to built a common basis for the architectures of hypertext systems, so that can interact to and to be compared to each other.

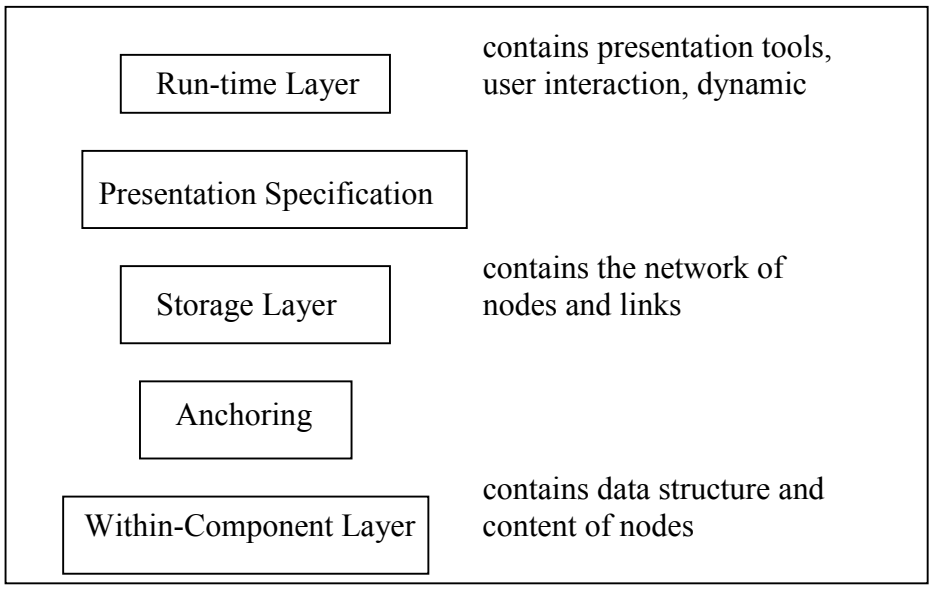
Objectives for the model:

- Standard terminology
- Minimum functionality
- Reference model – a meta-model for the formal description of properties of (existing) hypermedia systems
- Basis for the development of exchange formats.

The results are also applicable to hypermedia systems.

An Overview on the Dexter Model:

It consists of three layers and two interfaces



The three layers:

- The **Storage Layer** describes the network of nodes and links (most important layer). The storable Hypermedia objects are called components. The three basic components arc:
  - Atom
  - Link
  - Composite Component (for (hierarchical) structuring)

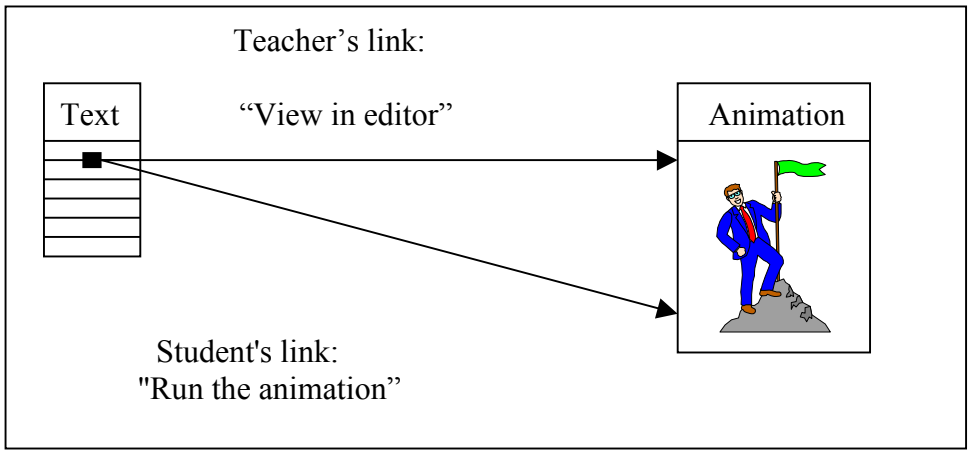
- The **Within-Component Layer** describes the content and structure of the components (nodes, links); e.g. the data structure for text, images, animation etc. This layer is system specific, so it will not be defined in the Dexter Model (e.g. defined in ODA, IGES, etc.)

- **The Run-time Layer** manages the presentation of the components in the user interface at runtime. There will be administered one session per user. This layer also handles the read/write-copies of the components in the cache (because updates are possible).

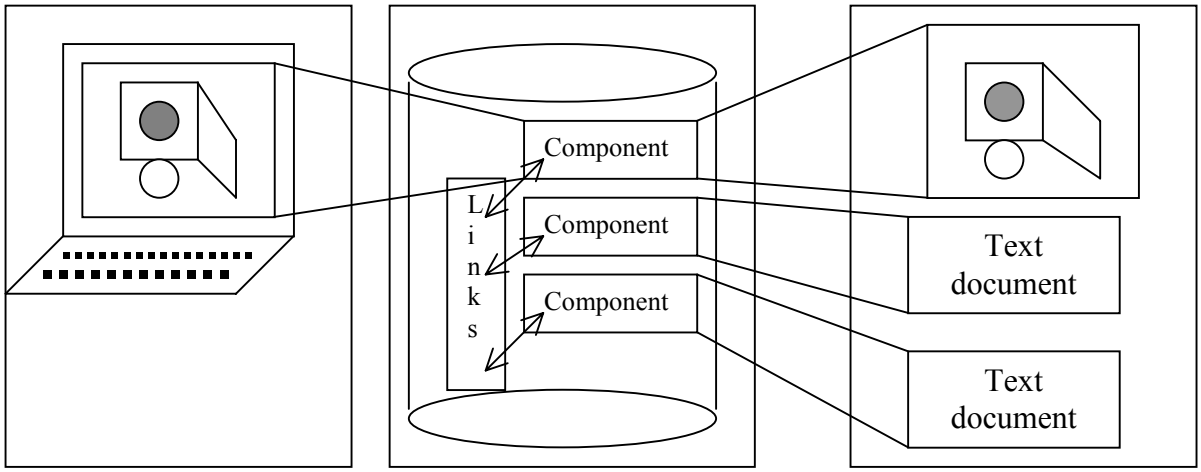
The two interfaces:

- The Anchoring offers the mechanism for addressing the link sources and targets even inside of the components (Span-to-Span-Links).

Thee Presentation Specification offers a possibility to deposit information on the display of components in the Storage Layer by the Run-Time-Layer (e.g. editable/noneditable)



Example for presentation specification even on links.



The Three Layers in Detail

The Storage Layer

This layer describes the hypertext structure as a finite set of components and a set of two access functions: resolver- and accessor- function.

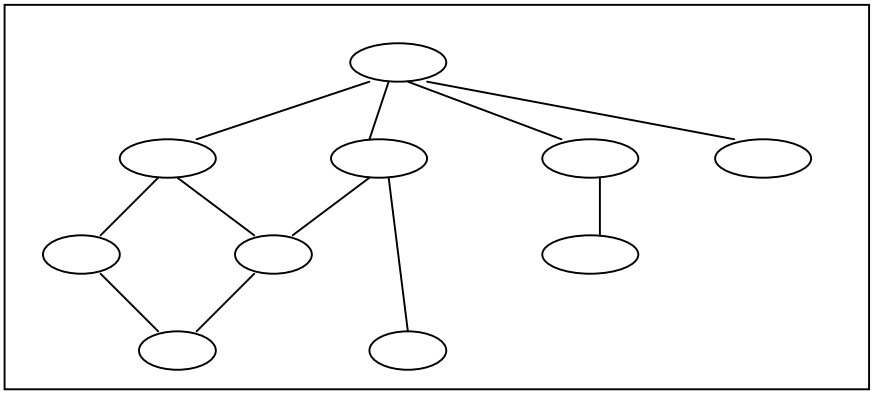
Component = Atom
Link
Composit Component

Atom: is cobsidered as a "node", that means as a primitive, it's substructure and content are specified in the Within-Component Layer.

Link: is specified by two or more anchors (= endpoints). Anchors can be components or parts of components (Span-to-Span).

Composite Component: it is a hierarchical structuring of components, particularly: DAG

Identification: Each component has a "global unique identifier" (UID).



The two access functions on components:

accessor-function: UlD → component

resolver-function: predicative spezification of the link target → UIDs or Ω

Anchors:

Span-to-Span-Links must be able to address substructures of components.

Anchor = (anchor-id, anchor-Value)

locally unique within a component (fixed length)

specifies location inside the component (variable length)

globally unique link anchor: Tuple (UID, anchor-id)

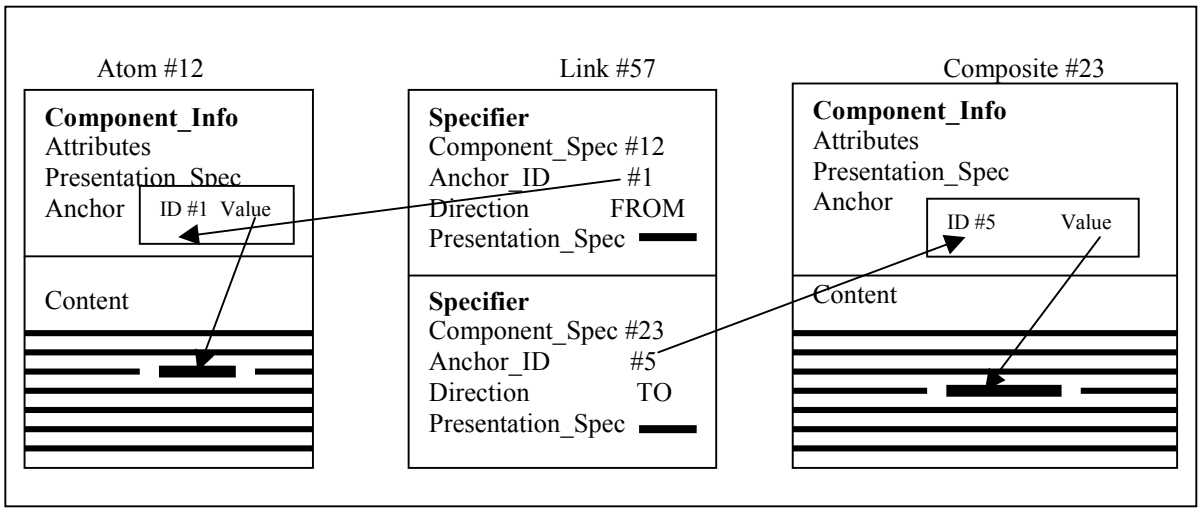
Note:

According to the definition, links are components. Thus the following are possible: Link with endpoints that are links, but also: endpoints of a link inside another link. Example: link on a linkdescription or on the type or direction of another link.

Specifier: of possible link endpoints; a specifier consists of

- component specification (e.g. the UID)
- anchor-id (= AID)
- direction (optional) ∈ [FROM, TO, BIDIRECT, NONE]
- presentation specification (optional)

Link: sequence of two or more specifiers (1:1-, 1:M-, M:N-links). Mostly: 1:1 links directed links

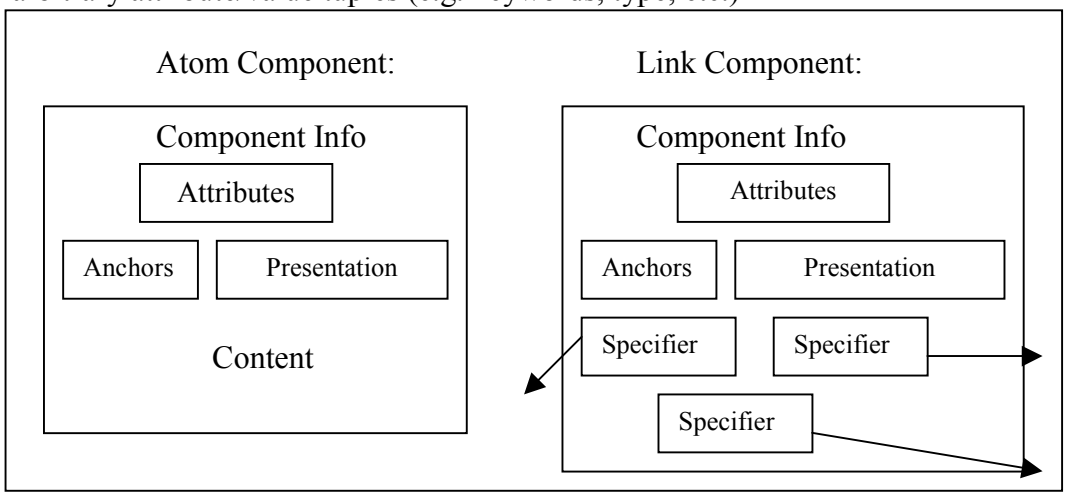


Component-Info:

In addition, components have the socalled "component informations".

These do not describe the content, but the properties of a component:

- all the anchors
- presentation information for the Run-time Layer
- arbitrary attribute/value tuples (e.g. keywords, type, etc.)



Components Structure in the Storage Layer

Link-Consistency:

The component-specification of each link-specifier must lead to an existing component (resolve, access).

Links are entities of their own, thus deleting a component requires deletion or update of all the corresponding links.

The Within-Component Layer:

This layer contains the data structure of the component and their contents. It is system specific, so it is not defined any closer in the Dexter Model.

The Run-time Layer

Runtime system for presentation. It contains plenty of dynamic functions, like "present component", "follow link", "realize", etc.

A new session is started for every user (session management).

Instance of a component = the users representation of the component
More precise: it is the copy of the component in the run time cache (compare to modification and rewrite), i.e. several instances of a component may exist at the same time. Each instance has it's own instance identifier (IID).

The instantiation of a component necessitates the instantiation of the anchors (= link markers).

Session-Entity contains:
- the hypertext
-the mapping IlDs -> UlDs
-History (at the moment this is defined only for read-only;

writes: compare to the TA-concept in DBS)
-runtime resolver function: (specification -> UlDs)
-instantiator function: (UID + presentation-spec. -> IID + Instance)
-realizer function: (invcrs to instantiator, = write back the cache after a modification)

Follow-Link operation: input: IID of an instance and the Link Marker.

Example for a simple exchange format according to the Dexter Model:

```
<hypertext>
    <component>
        <uid> 21 </uid>
        <type> text </type>
        <anchor>
            <id> I </id>
            <location> d13 </location>
        </anchor>
        <data> This is some text ... </data>
    </component>
    <component>
        <uid> 777 </uid>
        <type> text </type>
        <anchor>
            <id> I </id>
            <location> 13-19 </location>
        </anchor>
        <data> This is some other text ... </data>
    </component>
    <component>
        <uid> 881 </uid>
        <type> link </type>
        <specifier>
            <component_uid> 21 </component_uid>
            <anchor_id> 1 </anchor_id>
            <direction> FROM </direction>
        </specifier>
        <specifier>
            <component_uid> 777 </component_uid>
            <anchor_id> 1 </anchor_id>
            <direction> TO </direction>
        </specifier>
    </component>
</hypertext>
```

Evaluation and Review

The Dexter Model is a more powerful model then the models that are usually used in Hypemedia systems:
-1:M links, M:N-Links
-Links to links (hard to implement)
-Complex components
-bidirectional links
-extremely powerfull resolver-function

Weak Points of the Model:

- strict separation of the layers is weakened by the anchors!
- the direction of a link is not specified until in the Link Specifier; thus inconsistency problems may occur;
- model extensions will be necessary in respect of time-critical media (synchronisation, etc.) (see also: Amsterdam Hypermedia Model).
- in editing mode: all relevant components must already exist in the system, before a link can be defined.

Summary:

The Dexter Model is not a standard in it's true sense of the meaning, (not in the sense: "We fulfill the Dexter-Model"), but a reference model (instead of: "Our ... corresponds to the Dexter Model").

Advantages of Multimedia Database Systems

- integrated administration of huge amounts of multimedia data
- optimized storage
- efficient access
- manyfold complex search possibilities
- referential integrity of links
- transaction protected multiuser mode
- recovery
- etc.

Multimedia-DB applications

Fields of application:

• *static/passiv:*
Retrieval / Information / Archive
(Libraries, video on demand, information systems, press, hospitals)
Databases, information retrieval

• *static/aktiv*
Education / Commercials/ Entertainment
(School, university, professional training, games, commercials)
CSE, Teachware, Courseware, CBT,

• *dynamic/passiv*
Writing / Publications/ Design
(Press, engineering, architecture)
Editors, layout generators, CAD-systems

• *dynamic/active*
Controlling/Monitoring (Factories, traffic, weatherforecast, military)
Process control systems