Objektinės Duomenų Bazės

Paskaitos MiF Informatikos specialybės studentams

Arūnas JANELIŪNAS

VU MiF Informatikos katedra

Email: Arunas.Janeliunas@gmail.com

1998 — 2006

TURINYS

Įvadas

Žodžių junginys "Duomenų bazės" šiandien nestebina nieko. Jis nekelia nei nustebimo, pagarbaus virpulio, ar asociacijos su kosmosu. Niekas net neperklausia "Kas?". Ir visiems duomenų bazės yra be galo reikalingos. Jeigu tik reikia sužinoti ką nors, kas neapsiriboja daugybos lentele, kiekvienas, pradedant mediku ir baigiant filologu, sako "Reikia duomenų bazės".

Informatikos specialistų duomenų bazės taip pat jau senokai nestebina. Jeigu prieš 30 metų duomenų bazės įgaudavo visokiausius keitus ir neregėtus pavidalus ir jas programavo tikri guru (pažiūrėkite bet kurį, kad ir šiuolaikinį, Hollywood'o filmą...), tai po dešimtmečio juos visi ėmė vadinti "Foksininkais". Dar vėliau namudinius foksininkus pakeitė išdidūs "Oraklistai" ir "IBMeriai", o dabar mySQL'o "lentas" be didžiausio vargo kuria bet kuris WEB dizaineris.

Jeigu atmesime ironiją, turėsime sutikti, kad duomenų bazės egzistuoja jau labai seniai: pradedant nuo šumerų molinių lentelių saugyklų. Jeigu anksčiau žmonės duomenis saugodavo užrašuose, pergamentuose, knygose, bibliotekose, tai nereikėtų stebėtis, kad duomenų bazių esamam ir kompiuteriuose. O kadangi kiekvienas iš mūsų yra daugiau ar mažiau linkęs užmiršti daugybę dalykų, mes juos užsirašom. Į savo duomenų bazę.

Lentelė iki šiol yra vienas iš labiausiai paplitusių būdų tvarkingai užsirašyti daugybę dalykų. Todėl "lentelinės" arba – reliacinės, duomenų bazės yra visiems savos, mielos ir suprantamos. *Oracle, DB 2, Sybase, MySQL, FoxPro, Access* ir t.t. , visa tai — reliacinės duomenų bazių valdymo sistemos. Po šiais vardais slypi vienas principas, viena idėja — IBM sukurta "*System R*" ir kartu su ja aprašyta Reliacinių Duomenų Bazių teorija. Ar yra kitokių?

Tokių pat gerų — turbūt, ne.

Objektinė paradigma taip pat jau neišgyvendinamai įsiskverbusi į informatikos pasaulį. Ji sutinkama ne vien Objektiškai Orientuotose programavimo kalbose (*Java*, *C++, SmallTalk, Eiffel* ir pan.) bet ir programų sistemų inžinerijoje, sistemų teorijoje ir t.t. Ne išimtis ir DB valdymo sistemos.

Žmonės, kurie viską įsivaizduoja objektais, negalėjo neatkreipti dėmesio į lenteles, kurios, deja, su objektais neturi daug bendro. Reikia sutikti su jais, kad turint beveik tobulą "objektinę" aplinką, kurioje viskas objektiškai sumodeliuota, objektiškai suprogramuota, viską saugoti ne objektais yra mažų mažiausiai keista. Objektinės duomenų bazės čia — neišvengiamos.

Taigi, jos — Objektinės Duomenų Bazės (ODB) — ir yra mūsų studijų dalykas. Noriu, kad suprastumėte teisingai — jos nėra nei geresnės, nei blogesnės, negu kitos DB valdymo sistemos. Jos turi savo nišą uždaviniuose, rinkoje ir mūsų tikslas yra jas išmanyti arba bent jau apie jas žinoti — o jeigu prireiks?

Objektinės duomenų bazės

Skyrelis ruoštas besinaudojant [MRL98], [Mcl97].

Objektinės duomenų bazės imtos nagrinėti vėlyvaisiais 1970—aisiais. 1980—ųjų pradžioje jos tapo gana smarkiai besivystančių tyrinėjimų sritimi, o vėlyvaisiais 1980—aisiais atsirado ir pirmieji komerciniai produktai.

Vėlyvaisiais 1990—aisiais vienoje garbioje konferencijoje buvo paskelbta, kad objektinės duomenų bazės mirė.

Markas Tvenas kažkada sakė "Gandai apie mano mirtį yra gerokai perdėti..." Objektinių DB atveju — panašiai. Nors teiginiui apie jų mirtį yra pagrindas, tačiau visiškos ODB agonijos dar teks palaukti. Šiuo metu jos turi savo naudojimo nišą ir daugybė sistemų veikia būtent ODB pagrindu.

Taigi, Objektinės duomenų bazės yra objektinio programavimo ir duomenų bazių technologijų apjungimas. Kaip tai vyksta?

Programos paprastai yra rašomos kuria nors programavimo kalba, po to jos gali būti kompiliuojamos ir pan. Pavadinkime visą šį procesą ir jo aplinką "Programavimo sistema". Programavimo sistemos turi vieną trūkumą — jos negali išsaugoti duomenų, sukurtų jų darbo metu ir vėliau tais duomenimis pasinaudoti.

Jeigu turėsime omenyje, kad duomenis programa gali išsaugoti kad ir kokiame nors faile — tai ir yra primityviausia duomenų bazė. Atkreipiame dėmesį, kad duomenys visuomet išsaugomi kur nors kitur, bet — NE PROGRAMOJE.

Šiuos du dalykus (išsaugoti duomenis ir vėliau vėl paimti) moka Duomenų bazių valdymo sistemos. Todėl tipinis duomenų bazių programavimas susideda iš dviejų dalių:

- Programos logikos kūrimo. Čia naudojama Programavimo sistema (C++, Java, ...);
- Manipuliacijų su duomenimis programavimo. Čia programuojama Duomenų bazių valdymo sistema (gali būti naudojamas SQL, galbūt kitos duomenų manipuliavimo kalbos).

Tačiau bet kuriuo atveju programuotojas turi priešais save dvi skirtingas sistemas, kuriose yra skirtingi duomenų modeliai, tipai, skirtingi veikimo principai ir t.t. Kyla klausimas — ar negalima šių dviejų sistemų sudėti į vieną? Ar negalima ir logikos, ir manipuliavimo duomenimis programavimo padaryti viena sistema?

Būtent tai ir yra išskirtinis Objektinių duomenų bazių bruožas. Jos tarytum praplečia programavimo kalbų galimybes duomenų bazių valdymo sistemos funkcijomis, šitaip sukuriant naują (ar — naujovišką) sistemą. Tuo pačiu programavimo kalba ir taip praplečiama, kad nenukentėtų jos bendri principai ir susitarimai, įtvirtinti tos kalbos standarte.

Du pagrindiniai dalykai, kuriais yra išplečiamos programavimo kalbos yra šie: joms suteikiama duomenų išsaugojimo (*persistence*) ir užklausų (*querying*) galimybės. Toliau seka tokie dalykai, kaip duomenų apsauga, konkurencijos valdymas ir pan.

Noriu atkreipti dėmesį, jog programavimo kalbai nereikia suteikti duomenų sukūrimo, modifikavimo ar naikinimo galimybių (standartiniai INSERT, UPATE ir DELETE sakiniai SQL). Šias galimybe programavimo kalba jau turi. Užtat reikia ją "išmokyti" užklausų, t.y. SELECT sakinio SQL kalboje.

Žinoma, įvairiose ODB sistemose programavimo kalbų ir duomenų bazių funkcijų apjungimas būna nevienodas. Priklausomai nuo kiekvienos konkrečios ODB sistemos, šis apjungimas gali būti:

I. Pats paprasčiausias apjungimo būdas yra suteikti objektinei programavimo kalbai paprastą duomenų išsaugojimo galimybę. Priėjimą prie duomenų gali susiprogramuoti pats programuotojas. Šiai užduočiai pridedamos minimalios duomenų bazių galimybės: konkurencijos valdymas, transakcijos ir pan.

II. Vidutinio sudėtingumo lygio objektinės duomenų bazių sistemos jau turi praktiškai visas "bendrąsias" duomenų bazių sistemų savybes. Tokias sistemas galima traktuoti kaip "duomenų bazę su objektine kalba duomenų elgesiui aprašyti". T.y., jeigu lyginsime su reliacinėmis DB, pastarosiose yra tiktai aprašomi duomenys ir suteikiamos galimybės juos valdyti (tvarkyt). Visi veiksmai su duomenimis, jų keitimas ir pan. (išskyrus primityviausius veiksmus) yra vykdomi "išorėje", aplikacijose. Gi objektinėse duomenų bazėse duomenų "elgesys" taip pat yra išsaugomas, todėl duomenys gali keistis, evoliucionuoti ir be "išorinės" aplikacijos komandų. Be to, skirtingai nuo pirmojo varianto, suteikiama ne tik tiesioginė objektų susiejimo galimybė (per objektų atributus—nuorodas), bet ir asociatyvus objektų susiejimas (analogiškas į *JOIN* operatorių reliacinėse DB).

III. Aukščiausio sudėtingumo lygio objektinės duomenų bazės palaiko ir aukšto lygio deklaratyvius duomenų valdymo mechanizmus: duomenų optimizavimui, integralumo palaikymui, veiksmų apribojimams, saugumui ir pan.

Vidutinio sudėtingumo sistemų visiškai pakanka "įprastoms" duomenų bazių aplikacijoms kurti. Gal todėl dauguma šiuo metu rinkoje egzistuojančių produktų yra būtent šio sudėtingumo lygio.

Jeigu pabandytume surašyti pagrindinius privalumus, dėl kurių reikėtų rinktis objektines duomenų bazes, gautume štai tokį sąrašą:

I. Programavimo efektyvumas

Programuojant objektine programavimo kalba ir duomenis saugant reliacinėse lentelėse, visuomet reikalinga konversija tarp DBVS ir programavimo kalbos duomenų tipų. Programos objektai negali būti tiesiog išsaugomi duomenų bazėje ir atvirkščiai: duomenų bazėje saugomi duomenys nėra programos objektas. Nėra "permatomumo" (*transparency*) tarp DBVS ir programavimo kalbos duomenų modelių. Duomenų konversijos ten/atgal tarp programos ir duomenų bazės gali užimti iki 30% programos kodo. Antra vertus, "objektinis" programuotojas yra priverstas išmanyti dar vieną programavimo kalbą — SQL. Darbdaviui, investuojančiam į savo specialistus tai neparanku. Tuo tarpu ODB sistemos yra tarsi objektinės programavimo kalbos praplėtimas, leidžiantis objektus išsaugoti ir vėl pasiimti. Jokių kitos programavimo kalbos intarpų, jokių konversijų, daroma tiesiog kažkas panašaus į `objektas.save()`.

II. Kompleksinių duomenų palaikymas

Kadangi duomenų bazėse saugomi vis sudėtingesni duomenys, jie reikalauja vis sudėtingesnio tvarkymo. Standartiniai SQL duomenų tipai ("atominiai" duomenų tipai) yra per paprasti. Šių tipų išplėtimai — BLOB'ai, vartotojo apibrėžtos struktūros ir pan. — leidžia išsaugoti sudėtingus duomenis, tačiau jų negalima tiesiogiai naudoti užklausose, pvz. ieškant paveikslėlio, panašaus į obuolį ar pan. Objektinėse duomenų bazėse saugant objektus kartu su jų metodais, šis klausimas išsprendžiamas gana trivialiai. ODBS leidžia naudoti bet kurias objekto savybes užklausose. Kitaip tariant, jeigu duomenų modelį sudaro:

a) Statiniai duomenys,
b) Integralumo taisyklės, padedančios išlaikyti neprieštaringą DB būseną,
c) Dinaminės taisyklės, leidžiančios keisti DB būsenas,

Tai reliacinės DB palaiko tik pirmuosius du punktus, trečiąjį "atiduodamos" išorinėms aplikacijoms. Gi ODB palaiko visas tris duomenų modelio savybes.

III. Sudėtingų sąsajų palaikymas

Sistemose, kuriose yra begalės esybių, kurios tarpusavy susijusios dar didesne galybe ryšių, nėra taip paprasta "naviguoti" tarp jų. Reliacinės DB palaiko tik asociatyvų navigavimą, kai susijusi esybė randama pagal tam tikrą atitikimą su turima esybe (standartinė pirminio—išorinio raktų sąsaja). Tačiau kai ryšys tarp dviejų esybių yra komplikuotas, per daugelį "tarpininkų" su savom charakteristikom, navigavimas SQL pagalba tampa sudėtingu. Gelbsti *view*'ai. O ODB palaiko ir asociatyvų, ir tiesioginį navigavimą. T.y. be pirminio—išorinio raktų sąsajos palaikoma tiesioginė "kelio" išraiška, pvz.: `mano.tevas.arklys.brolis` . Pirmosios ODB taikymo sritys buvo CASE ir CAD sistemos, kuriose sąsajos tarp objektų ir yra ypač "komplikuotos".

IV. Paskirstytos sistemos

Reliacinių DB sistemų teorijos pagrindas buvo sukurtas serverio—kliento architektūrai su trumpalaikėmis transakcijomis. Pesimistinė transakcijų valdymo paradigma verčia blokuoti bet kokius veiksmus su duomenimis, kuriuos modifikuoja kitas vartotojas. Tačiau taip daryti galima tik trumpą laiką, nes ilgi "laukimo" tarpai gali paversti sistemą visiškai nepanaudojama. ODB sistemos kuriamos daugiausia turint omenyje aplikacijų tinklą, kurios naudojasi bendrais (*shared*) objektais. Transakcijos su šiais objektais gali būti labai ilgalaikės (diena, dvi, savaitė), todėl kuriami papildomi mechanizmai (objektų versijos, duomenų suliejimo technikos ir pan.) siekiant užtikrinti tokių paskirstytų sistemų veiksnumą. Geras pavyzdys — tos pačios CAD ar CASE sistemos, kurių atskirus objektus dizaineriai/architektai kuria ar modifikuoja labai ilgai.

V. Lengvesnis supratimas

Specialistai, ginantys ODB, teigia, kad "Reliacinių duomenų bazių kūrimas iš tikrųjų yra bandymas įsivaizduoti, kaip čia realaus pasaulio objektus įspraudus į lenteles, kad galima būtų turėti didelį našumą ir būtų galima išsaugoti duomenų integralumą. Objektinių DB kūrimas yra visiškai kitoks." (Mary Loomis, *Versant* ODBS architektė). Iš tiesų, žmonėms, kuriantiems klases, kuriomis atvaizduojami modeliuojami objektai, nebereikia vargti, stengiantis sugalvoti dar ir unikalią lentelių sistemą. Be to, teigiama, kad objekto koncepcija yra artimesnė žmogaus mastymui, tam, kaip jis supranta savo aplinką. Taigi, Objektinės DB yra lengvesnis būdas susikalbėti sistemos architektui, programuotojui ir paprastam jos vartotojui. UML išpopuliarėjimas tik patvirtina šį argumentą.

Taigi, šiandien turime situacija, kai ODB ir RDB yra pasidalinusios užduotis taip:

a) Kai reikalinga sistema didelės apimties paprastų duomenų apdorojimui – dažniausiai renkamasi reliacinė DB,
b) Sudėtingiems duomenims, su mažesne apimtimi renkamais objektinė DB.

ODB standartas

Skyrelis ruoštas besinaudojant [ODMGBE97], [Barr97], [Barr98], [MRL98], [DD95], [ABDDMZ].

Jeigu norėtumėme sužinoti tiksliau, kas yra objektinės duomenų bazės ir kokiomis konkrečiomis savybėmis jos pasižymi, tokius apibrėžimus turėtumėme rasti *Objektinių duomenų bazių standarte*. Žinoma, toks standartas, bendrai paėmus, gali ir neegzistuoti, tačiau standarto buvimas turi daug teigiamų savybių.

Reliacinių duomenų bazių sistemų sėkmė priklausė ne vien nuo aukštesnio duomenų nepriklausomumo lygio ar nuo paprastesnio, nei daugelio pirmtakių, duomenų modelio. Didelė dalis šios sėkmės priklausė ir nuo standartizavimo, kuris buvo pasiūlytas kartu su reliacinėmis DB valdymo sistemomis. SQL[1] standarto priėmimas užtikrino aplikacijų pernešamumą, palengvino naujų RDBVS mokymąsi ir taip suteikė didžiulę paramą "reliaciniams" uždavinių sprendimams.

Visi šie faktoriai svarbūs ir objektinėms DB. Be to, kaip jau minėta, ODB ypač yra svarbus integravimas su programavimo kalbomis, apimantis tiek operacijas, tiek duomenis. Todėl standarto buvimas yra kritinis, norint padaryti tokį integravimą veiksmingu.

Taigi, Objektinių DB standartas egzistuoja. Taip susiklostė, kad šis standartas yra "standartas" daugiau *de facto*, negu *de jure*. Tai reiškia, kad yra ODB sistemų, kurios laikosi šio standarto daugiau, kitos sistemos – mažiau. Tačiau šio standarto svoris yra pakankamas, o pats standartas — puikiai sudarytas. Taigi, paskirsime jam visą šį skyrių.

I. Bendrosios ODB savybės

Visų pirma, jeigu turėsime omenyje bendrąjį ODB veikimo principą, galima spėti, kad ODB standartas yra lyg ir tam tikras atskirų objektinių programavimo kalbų išplėtimų standartas. Šie išplėtimai leidžia programavimo sistemoms, paremtoms išplečiamąja kalba, tapti Objektinėmis duomenų bazių sistemomis.

Pagal [ABDDMZ], nurodžiusį gaires tolimesniam ODB vystymuisi, objektinės duomenų bazės turėtų pasižymėti šiomis savybėmis:

1. **ODB turi palaikyti sudėtinius objektus (*complex objects*).** Sudėtiniai objektai yra sudaromi iš paprastesnių, pritaikant tam tikrus *konstruktorius*. Yra įvairių konstruktorių: klasės, masyvai, sąrašai, aibės, struktūros ir pan. Sudėtinių objektų naudojimas leidžia modeliuoti realaus

[1] Kadangi šiame skyriuje bus daug sutrumpinimų, Jūsų patogumui, visi sutrumpinimai bus pateikti bendroje lentelėje šio skyriaus gale.

pasaulio esybes, naudojantis būtent to pasaulio terminais. Tai yra alternatyva objektų dekomponavimui į atominius duomenų tipus ir denormalizavimui, kaip tai daroma reliacinėse DB.

2. **ODB turi palaikyti objektų identifikavimą.** Kiekvienas objektas identifikuojamas savo identifikatoriumi (OID). OID, kaip jau nusistovėjo objektinėse programavimo kalbose, nesiremia objekto atributų reikšmėmis. Tai — vidinės sistemos reikšmės. OID naudojimas palengvina sąryšių tarp objektų modeliavimą. Netgi reliacinėse DB objektams (teisingiau – eilutėms) dažnai yra suteikiami unikalūs, vidiniai sistemos identifikatoriai.

   Panaikinus objektą, jo identifikatorius gali būti naudojamas kitiems objektams arba ne. Tai palikta spręsti atskiroms ODBS. Nenaudojant identifikatorių iš naujo, didėja identifikatorių dydis ir numenčia ODB kompaktiškumas. Naudojant nebereikalingus OID, reikalingi mechanizmai, užtikrinantys, kad ištrinamojo objekto OID būtų sunaikintas ir kituose objektuose, kad vėliau nebūtų sudaromos klaidingos asociacijos tarp objektų.

3. **ODB turi palaikyti klases ir tipus.** *Klasė* yra bene svarbiausia sąvoka visame objektiniame pasaulyje. Nuo klasės neatsiejama sąvoka yra *tipas*. Kuo jie skiriasi, paaiškinsime vėlesniuose skyriuose. Čia gi paminėsime, jog dažnai programavimo kalbose klasė yra sutapatinama su tipu (C++, Simula, Vbase, ...). Kitur gi turima tik klasės sąvoka (SmallTalk, Gemstone, ...). Grubiai tariant, tipas — tai sąvoka, dažniau naudojama kompiliavimo metu, o klasė — dažniau naudojama programos vykdymo metu (*runtime*). Šiaip ar taip, kurią sąvoką palaikyti (o gal ir abi...), paliekama spręsti atskiroms ODB sistemoms.

   Dar paminėsime, kad klasės yra naudojamos aprašant ODB schemas. Klasių apibrėžimai nurodo, kokie objektai gali būti saugomi ODB.

4. **ODB turi palaikyti klasių (tipų) hierarchijas.** Klasių (tipų) hierarchija ir paveldėjimo mechanizmas yra labai galingas objektinių sistemų konceptas, kurio nepalaikymas būtų, turbūt, tolygus bent pusės objektinių technologijų privalumų atsisakymui. Paveldėjimas ODB sistemose gali būti dvejopas — kai kiekviena klasė gali būti "kilusi" tik iš vienos klasės ir kai viena klasė gali turėti kelias "super" klases. Tai daugeliu atveju priklauso tiek nuo pačios ODB sistemos sudėtingumo, tiek nuo programavimo kalbos (ar kalbų) naudojamų manipuliacijai tokios ODB duomenimis.

5. **ODB turi palaikyti inkapsuliaciją.** Inkapsuliacija leidžia "paslėpti" konkrečios klasės objektų elgesio mechanizmus nuo tos klasės "klientų" (t.y. objektų, kurie kreipiasi į tos klasės objektus). Bendrai paėmus, klasės savybės gali būti neinkapsuliuotos (pvz.: *public* atributai C++ klasėse) ir inkapsuliuotos (*private* arba *protected* atributai C++ klasėse). Pasikeitus neinkapsuliuotų savybių logikai, dažnai tenka perprogramuoti ir pasikeitusios klasės klientų savybes (metodus, kurie kreipėsi į neinkapsuliuotą savybę). Gi pasikeitus inkapsuliuotai savybei, klientų keisti paprastai nereikia. Tokiu būdu, inkapsuliacija yra mechanizmas, užtikrinantis gana aukšto lygio duomenų tarpusavio nepriklausomumą.

6. **ODB turi palaikyti metodų užklojimą ir vėlyvąjį išsprendimą (*overloading and late binding*).** Savybė, susijusi su 5 punktu. Metodų užklojimas leidžia sukurti klasei jos unikalų elgesį, o vėlyvasis išsprendimas užtikrina "teisingą" objekto elgesį sistemos veikimo metu. Šie du mechanizmai padidina duomenų nepriklausomumo lygį, kadangi leidžia įvesti į ODB naujas klases su savitu elgesiu ir nemodifikuojant jokių kitų klasių.

7. **ODB turi būti pilna skaičiavimų atžvilgiu (*computationally complete*).** Tai reiškia, jog jos kalba turi būti galima aprašyti bet kokią apskaičiuojamą funkciją. Labai griežtas reikalavimas. Atkreipsime dėmesį, kad SQL nėra pilna skaičiavimų atžvilgiu, ja negalima užrašyti įvairių iteracijų ir pan. Šiems veiksmams reikia pasitelkti išorines aplikacijas ir programavimo kalbas. Objektinės DB privalo neturėti šios silpnybės. Todėl jų integravimas su programavimo kalbomis turi būti pakankamai gilus. Juk programavimo kalbos paprastai būna pilnos skaičiavimų atžvilgiu (priešingu atveju — kam jos būtų kuriamos?).

8. **ODB turi būti praplečiama.** Čia reikalaujama, jog jos vartotojas turėtų galimybę kurti savus klases ar tipus, kuriuos pilnai būtų galima naudoti duomenų bazėse. Nereikalaujama tik, kad vartotojas galėtų išplėsti sudėtinių tipų konstruktorių aibę (masyvai, sąrašai, aibės ir t.t.).

   Svarbus yra reikalavimas turėti galimybę išplėsti ODB naujomis klasėmis. Tai reiškia, jog karts nuo karto objektai yra priversti migruoti iš vienos klasės į kitą. Tai yra specifinis ir sudėtingas ODB uždavinys.

9. **ODB privalo turėti galimybę išsaugoti savo duomenis**. Iš duomenų bazių požiūrio taško tai — akivaizdus reikalavimas. Juk toks yra ODB tikslas — padaryti taip, kad objektai būtų išsaugomi.

   Išsaugojimas (*persistence*) gali būti kelių tipų:

   - Kai kurios klasės paskelbiamos "saugotinos" (*persistent*) ir bet kuris tos klasės objektas yra išsaugomas duomenų bazėje. Tai toks išsaugojimo modelis, kai išsaugojimas priklauso nuo klasės.
   - Sukuriant kiekvieną objektą, nurodoma — reikia jį išsaugoti, ar ne. Tai išsaugojimo modelis, kai išsaugojimas priklauso nuo kiekvieno objekto.
   - Trečias tipas — kai objektai išlieka pagal pasiekiamumą. Jeigu objektas yra pasiekiamas iš išsaugoto (—jamo) objekto, tai jis irgi yra išsaugomas.

   Nuo objektų saugojimo tipo priklauso ir objektų naikinimas (kiekvieną objektą reikia iš duomenų bazės pašalinti išreikštinai, yra tam tikras "šiukšlių surinkėjas" ir t.t.).

10. **ODB turi valdyti priėjimo prie duomenų konkurenciją ir duomenų atstatymą (*concurency and recovery*).** Tipinės DBVS savybės. Kaip jau minėta, ODB naudoja optimistinį konkurencijos valdymo mechanizmą, kurio esmė yra prielaida, jog duomenų prieigos konfliktai būna retai, todėl objektų skaitymas nebūna griežtai ribojamas, o transakcijos užbaigimo momentu taikomi įvairūs konfliktų sprendimų mechanizmai: panaudojant tiek vartotojo įsikišimą, tiek objektų versijas ir pan.

    Duomenų atstatymas yra kelių lygių. Atsitikus aplikacijos lygio sutrikimams (nutrūkęs ryšys, "pakibusi" aplikacija ir pan.) pašalinti užtenka transakcijų mechanizmo (*commit/rollback*). Nuo sistemos sutrikimų (pvz.: dingus elektrai) gelbsti *rollback* ar *roll forward* mechanizmai, leidžiantys užbaigti pradėtas transakcijas. Atsitikus media gedimams, naudojamas *backup recovery* mechanizmas, t.y. atstatymas iš atsarginių kopijų.

11. **ODB turi palaikyti asociacijas**. Iš pažiūros paprastas punktas. Tačiau štai kodėl jis yra svarbus: asociacijos yra tas dalykas, kuris padeda sukonstruoti objektinėje duomenų bazėje tam tikrą realios sistemos modelį ir vėliau padeda palaikyti to modelio integralumą. Visų pirma, asociacijos gali būti skirtingos: (i) jos gali būti vienpusės arba abipusės; (ii) jos gali turėti kardinalumą; (iii) jos gali turėti tam tikrą sutvarkymo semantiką (sąrašai). Antras dalykas — ką reikia daryti, jeigu šalinamas objektas turi

kažkokių nuorodų į kitus objektus? Ar juos irgi reikia šalinti? Ar reikia neleisti šalinti tokio objekto su nuorodomis? Jeigu šaliname objektą, ar reikia surinkti visas nuorodas į jį ir jas "išvalyti"? O gal irgi negalima leisti šalinti tokio objekto? Yra daugybė klausimų ir daugybė atsakymų į juos variantų.

12. **ODB gali palaikyti objektų versijas**. Atsiradus objektinėms duomenų bazėms, atsirado ir "ilgų transakcijų" terminas, reiškiantis transakcijas, kurios gali vykti dieną, dvi ir pan. Pagrindinė jų idėja yra tokia — kiekvienas vartotojas "pasiima" iš duomenų bazės sau objektą ir daro su juo ką nori ir kiek nori laiko. Jeigu kitas vartotojas tuo pat metu irgi kažką daro su tuo pačiu objektu, tuomet atsiranda dar viena to objekto *versija* su savo istorija (sukūrimo laikas ir t.t.). Vėliau ODB, sudėjusi kelias objekto versijas, privalo "sukombinuoti" jas, pagamindama "teisingą" objektą.

    Objektų versijavimas gali būti dvejopas — tiesinis ir šakinis. Tiesinis versijavimas reiškia, kad objektų versijos gali būti kuriamos tik viena po kitos, kuriant vienintelę objekto versiją iš tuo metu turimos. Šakinio versijavimo atveju vienu metu gali būti kuriamos kelios objekto versijos. Vėliau, vienokiu ar kitokiu metodu, įsikišant žmogui, tos versijos gali būti sulietos vėl į vieną.

    Su versijos sąvoka susijusi ir konfigūracijos sąvoka. Konfigūracija yra tam tikra objektų versijų sistema, kurioje objektų versijos yra tarpusavy suderinamos. T.y. konfigūracija nusako tokias objektų versijas, kurios sudaro neprieštaringą DB.

    Vietoj objektų versijų yra siūloma palaikyti duomenų bazės versijas. Vienos versijos duomenų bazėje objektų versijų nebūna. Kiekvienas vartotojas savo "ilgąją" transakciją vykdo "savoje" duomenų bazėje ir jos pabaigoje gauna "savą" duomenų bazės versiją. Vėliau suliejimo mechanizmai taikomi skirtingoms duomenų bazės versijoms suderinti.

    Ar palaikomos objektų versijos ir koks versijavimo mechanizmas — paliekama spręsti atskiroms ODB sistemoms. Tačiau reikia pažymėti, jog dauguma ODB palaiko objektų versijas.

13. **ODB gali palaikyti dinaminius DB schemos pokyčius (*runtime schema definition/modification*).** Labai naudinga savybė. Tai reiškia, kad aplikacija gali pati pakeisti savo naudojamos duomenų bazės schemą: pridėti naujas klases ar pakeisti jau egzistuojančias. Iškart reikia pripažinti, jog ši idėja visiškai svetima C++ programavimo kalbai ir sunkiai būtų derinama su C++ paremtomis ODB. Daugiau dėmesio DB schemos pokyčiams bei objektų migracijai skirsime atskirame skyrelyje.

Tokius fundamentalius samprotavimus ODB iškėlė "Objektinių duomenų bazių manifestas", o formaliai joms keliamus reikalavimus aprašo taip vadinamasis "ODMG standartas".

II. ODMG standartas

Iš pradžių šiek tiek istorijos. ODMG[2] — tai objektinių ir reliacinių DBVS pardavėjų bei kitų suinteresuotų šalių konsorciumas. Jis kūrė ir rūpinosi standartu, kuris užtikrintų aplikacijų pernešamumą tarp įvairių ODB sistemų.

[2] ODMG — *(angl.)* Object Database Management Group.

Konsorciume dalyvavo: GemStone Systems, IBEX Object Systems, Lockheed Martin, Object Design, Objectivity, POET Software, Unidata, Versant Object Technology, Andersen Consulting, Baan, CERN, Computer Associates, Electronic Data Systems (EDS), Hitachi, Microsoft, NEC Corporation ir Sybase. Organizacija buvo suburta 1991 metais, kai Rick Cattell, vėlesnis ODMG vadovas, o tuomet — JavaSoft darbuotojas, paragino tada dar mažas kompanijas — ODB kūrėjas ir pardavėjas — dirbti kartu, kad būtų sukurti standartiniai programavimo kalbų susiejimai objektinėms duomenų bazėms. Suburtoji grupė nutarė sukurti savo organizaciją, kuri vystytų savą standartą. Dirbama buvo greitai ir 1993 metais buvo publikuotas standartas ODMG—93 Release 1.0 . Jo pataisymai (aukštesnė versija) buvo publikuoti 1995 m. O dar vėlesnė standarto versija — ODMG 2.0 — paskelbta 1997—ųjų liepą ("The Object Database Standard: ODMG 2.0" Morgan Kaufman Publishers, 1997). Antroji standarto versija buvo gerokai patobulinta, palyginus su pirmąja. Plačiai paplitus *Java* programavimo kalbai, standarte atsirado ODB sąsaja Java kalbai (iki tol buvo tik C++ bei *SmallTalk*'ui). Standarte atsiranda sąsaja apsikeitimui objektais (ODB sąsaja). Ši ODMG standarto versija buvo pakankamai išsami ir stabili, reikalaujanti mažai pakeitimų. Todėl trečioji standarto versija pasirodė tik 2000—aisiais metais ("The Object Data Standard: ODMG 3.0" Morgan Kaufman Publishers, 2004). Trečiojoje standarto versijoje buvo patobulinta ODB sąsaja su *Java*. Šiek tiek buvo patobulintas ir objektinis duomenų modelis. Standarte atsirado pakeitimai, leidžiantys juos pritaikyti ir objektinėms—reliacinėms duomenų bazių valdymo sistemoms, ne vien "originaliosioms" ODB.

Iš pradžių ODMG specifikacija buvo kuriama kaip standartinė sąsaja (API) objektinių duomenų bazių sistemoms, tačiau jau antrojoje standarto versijoje ji evoliucionavo į standartinę sąsają objektų išsaugojimui. Tokiu būdu, pavyzdžiui, ODMG standartas pateikė pirmąją ir vienintelę standartinę sąsają (interfeisą), leidžiančią išsaugoti Java objektus, naudojantis standartiniu API, kuris yra visiškai nepriklausomas nuo duomenų bazės. Ji gali būti reliacinė, objektinė ar dar kokia nors. Jeigu ji palaiko šį standartą — Java objektai gali būti saugojami tiesiog, API pagalba. Tuo pasinaudojo *Sun Microsystems*, kurdama savo *Java Blend*, skirtą reliacinėms DB.

2001—aisiais metais ODMG grupė nutarė baigusi savo darbą ir išsiskirstė. Iki to laiko ODMG standarte sukurta sąsaja su *Java* programavimo klaba buvo pateikta *Java Community Process* grupei kaip pagrindas JDO (*Java Data Objects*) specifikacijai. Tolesni "objektų išsaugojimo" standartizavimo veiksmai vyksta būtent šia linkme.

ODMG standarte Objektinės duomenų bazės apibrėžiamos taip:

Apibrėžimas Objektinė duomenų bazių valdymo sistema yra DBVS, integruojanti duomenų bazių savybes su objektinės programavimo kalbos savybėmis. ODBVS leidžia duomenų bazės objektams veikti kaip programavimo kalbos objektai, vienoje ar keliose programavimo kalbose. ODBVS išplečia programavimo kalbą duomenų išsaugojimo, konkurencijos valdymo, duomenų atstatymo, asociatyvių užklausų ir kitomis duomenų bazių savybėmis.

Duomenų bazių programavimas visiškai perkeliamas į programavimo kalbos aplinką. Toks duomenų bazės ir aplikacijos programavimo sukombinavimas vienoje aplinkoje reiškia, jog turimas tiktai *vienas* duomenų modelis. Tai palengvina programavimo darbą, nes, pasak statistikos, net iki 30% su duomenų bazėmis dirbančios programos kodo gali sudaryti programoje naudojamų duomenų atvaizdavimui į duomenų bazę.

Dabar pristatysime pagrindinius standarto ODMG 3.0 komponentus.

Objektinis modelis

Tai — duomenų modelis. ODMG standarte aprašomas objektinis meta modelis. Jis yra visą standartą vienijantis konceptas ir visiškai nepriklauso nuo jokios programavimo kalbos.

Kuriant ODMG objektinį modelį, kaip bazė buvo panaudotas OMG (*Object Management Group*) objektinis modelis. Pastarasis buvo sukurtas kaip bendras objektinio modelio branduolys objektinėms programavimo kalboms, objektinėms duomenų bazėms, ORB (*Object Request Brokers*) ir kitoms aplikacijoms. Prisilaikant bendros OMG modelio architektūros, buvo įvestos papildomos jo savybės, atitinkančios objektinių DB poreikius: asociacijas, sritis, kolekcijų klases ir konkurencijos valdymą.

Objektinės specifikacijų kalbos

Viena iš šių kalbų yra ODL (*Object Definition Language*). Jos sintaksės bazei buvo panaudota OMG interfeiso aprašymo kalba IDL (*Interface Definition Language*). ODL yra naudojama duomenų bazės schemos apibrėžimui. Ši schema gali būti naudojama įvairiose duomenų bazėse ir yra nepriklausoma nuo programavimo kalbos.

Kita specifikacijų kalba yra OIF (*Object Interchange Format*), kuri gali būti naudojama objektų apsikeitimui tarp duomenų bazių, kuriant duomenų bazės dokumentaciją ar vykdant duomenų bazės testus.

Objektinė užklausų kalba

ODMG standarte aprašoma deklaratyvi (neprocedūrinė) užklausų kalba OQL (*Object Query Language*). Jos pagrindas, kur įmanoma, yra standartinis SQL ir Objektinės DBS turi savyje visas reliacinių užklausų galimybes. Tačiau OQL turi didesnes galimybes, nes gali manipuliuoti sąrašais, masyvais bei bet kokio tipo rezultatais.

Pastaruoju metu, atsižvelgiant į reliacinių duomenų bazių išplitimą, jų rinkos tvirtumą bei kitus aspektus, ODBS kūrėjai stengiasi užtikrinti kuo pilnesnį SQL palaikymą. Todėl labai dažnai ODB valdymui galima naudoti ir SQL (dažniausiai — pasinaudojant ODBC ar JDBC mechanizmais).

ODMG standarte teigiama, kad manipuliavimo duomenimis komandos turėtų integruotis su naudojama aplikacijos programavimo kalba. Taip pat standarto autoriai teigia nemaną, jog ODL bus vienintelė duomenų apibrėžimo kalba. Tiek duomenų apibrėžimas, tiek vėlesnis manipuliavimas jais turėtų vykti naudojamos programavimo kalbos lygyje. Šia dvasia parašyti kitos trys ODMG standarto dalys.

Susiejimas su C++

Ši standarto dalis nusako, kaip rašyti C++ kodą, kuris leistų sukurti, įvardinti, manipuliuoti ir naikinti objektus objektinėje DB. Be to, susiejimas su C++ leidžia kurti pagal paveldėjimą klases, kurių objektai gali būti išsaugomi duomenų bazėje.

Susiejimas su C++ aprašo ODL versiją, naudojančią C++ sintaksę, nusako OQL iškvietimo mechanizmą, ir apibrėžia, kaip iškviečiamos operacijos su duomenų bazėmis bei transakcijomis. Pateikiami STL (*Standard Template Library*) algoritmai ir išskirtinių įvykių (*exception*) apdorojimas.

Susiejimas su SmallTalk

Šis susiejimas apibrėžia atvaizdavimą tarp ODL ir SmallTalk, besiremiančią OMG siūlomu SmallTalk siejimu su IDL.

Susiejimas nekeičia SmallTalk sintaksės. Taipogi atsižvelgiama į tai, jog SmallTalk yra dinamiškai tipizuojama.

Taip pat numatomas OQL iškvietimo mechanizmas ir operacijų su duomenų bazėmis bei transakcijomis naudojimas. Atsižvelgiama į dinaminę atminties valdymo SmallTalk'e semantiką — objektai išlieka duomenų bazėje pagal pasiekiamumą.

Susiejimas su Java

Čia siejamas ODMG objektinis modelis su Java programavimo kalba. Nustatoma vieninga tipų sistema, naudojama Java ir objektinių DB. Java sintaksė nekeičiama.

Numatomas OQL iškvietimo mechanizmas ir operacijų su duomenų bazėmis bei transakcijomis naudojimas. Kaip ir SmallTalk atveju, objektai išlieka duomenų bazėje pagal pasiekiamumą.

Yra visiškai įmanoma rašyti ir skaityti iš tos pačios duomenų bazės, naudojant C++, SmallTalk ir Java programavimo kalbas, kol manipuliuojama bendrai palaikomais duomenų tipais.

Pradedant ODMG—2.0 standarto versija, pridedami ir du papildomi skyriai, kurie aprašo:

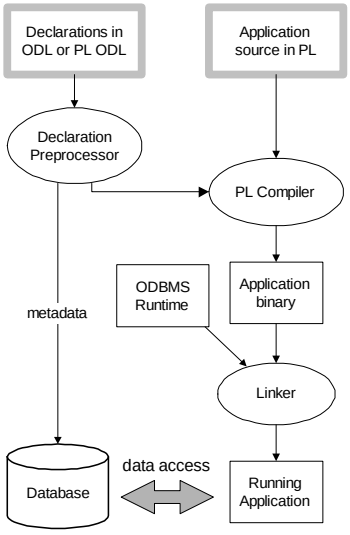
OMG Objektinio modelio skirtumus

Čia aprašomi skirtumai tarp ODMG bei OMG objektinių modelių.

Susiejimą su OMG ORB

Specifikuojamas ODBS objektų naudojimas per ORB. Taip pat nusakoma, kaip ODBS gali pasinaudoti OMG ORB interfeisu.

ODMG standartas nusako ir aplikacijų, veikiančių su ODB, kūrimo procesą, kurį pateikiame Paveikslėlyje 1.



Paveikslėlis 1 ODB naudojimas (paimta iš [ODMGBE97])

Programuotojas parašo "schemos" (duomenų ir interfeisų) deklaraciją bei programos išeities tekstus kuria nors programavimo kalba (PL — *Programing Language*), pasinaudodamas kartu su ODB pateikiamomis bibliotekomis užklausų vykdymui, transakcijų valdymui ir t.t. Schemos deklaracija gali būti rašoma tiek ODL, tiek ODL modifikacija naudojamai programavimo kalbai. ODL galima naudoti kaip aukšto lygio duomenų apibrėžimo kalbą, nepriklausančią nuo pasirinktos programavimo kalbos.

Po to parašyti tekstai yra transliuojami ir susiejami su ODB bibliotekomis. Gautoji vykdoma programa gali naudotis duomenų baze, kurioje saugomų duomenų tipai turi būti suderinti su schemos deklaracijomis. Transakcijų valdymą, duomenų prieinamumą bei konkurencijos valdymą užtikrina ODB.

Tolesniuose skyriuose pristatysime objektinį modelį kartu su trumpu duomenų apibrėžimo kalbos ODL aprašymu, o paskui — ir objektinę užklausų kalbą OQL.

O dabar, kaip ir žadėta, pateiksime vartotų sutrumpinimų lentelę.

Santrumpa	Reikšmė	Komentaras
IDL	Interface Definition Language	OMG naudojama duomenų apibrėžimo kalba.
JDO	Java Data Objects	Specifikacija, skirta *Java* objektams išsaugoti. Integruojama su EJB ir J2EE.
ODBS	Objektinės Domenų Bazių Sistemos	
ODL	Object Definition Language	ODMG standarte pasiūlyta duomenų apibrėžimo kalba.

ODMG	Object Databases Management Group	Grupė, besirūpinanti objektinių duomenų bazių standartizavimu. `http://www.odmg.org`
OIF	Object Interchange Format	ODMG standarte siūloma kalba objektų apsikeitimui su ODB aprašymui.
OMG	Object Management Group	Grupė, besirūpinanti objektinių technologijų standartizavimu. `http://www.omg.org`
OQL	Object Query Language	ODMG standarte siūloma užklausų kalba.
ORB	Object Request Broker	OMG sukurtas objektų apsikeitimo standartas.
SQL	Structured Query Language	Reliacinių DB manipuliavimo duomenimis kalbos standartas.

Objektinis modelis

Skyrelis paremtas medžiaga iš [AVH95], [Banc96], [BC97], [Benz97].

Pristatant objektinį modelį, naudojamą ODBS, bus remiamasi dviem šaltiniais. Visų pirma, bus prisilaikoma ODMG standarte suformuluoto objektinio modelio. Bet, kadangi šis modelis yra pakankamai bendro pobūdžio, pavyzdžiams dažniausiai naudosime O_2 notaciją (nepamiršdami priminti ir ODMG atitikmenų šiai notacijai).

O_2 [BDK92] yra viena iš labiausiai paplitusių pasaulyje ODBS, sukurta ir vystoma kompanijos *O_2 Technology*. Ji palaiko ODMG standartą ir, be to, jos naudojama notacija pakankamai artima C žymėjimams, todėl tikimės, jog bus gan lengvai suprantama be ypatingų komentarų.

Formalus (daugiau ar mažiau matematinis) objektinio modelio pristatymas remiasi [AVH95].

I. Semantiniai modeliai

Visų pirma, reikia pažymėti, kad objektinis duomenų modelis remiasi prieš daugelį metų pasiūlytu semantiniu modeliu. Dabar šiuo modeliu nieko nenustebinsime, tačiau panagrinėkime jį šiek tiek plačiau, kad išsiaiškintume pagrindinius dalykus.

Paplitus reliacinėms duomenų bazių valdymo sistemoms ir pradėjus masiškai jas naudoti, be abejo buvo pastebėti ir reliacinio duomenų modelio trūkumai tam tikrais atvejais. Stengiantis juos kompensuoti (ar kokiu nors būdu apeiti), duomenų modelių (duomenų bazėse) teorijos evoliucija išsiskyrė į dvi kryptis:

- Egzistuojančio reliacinio modelio išplėtimas. Taip atsirado įvairūs reliacinio duomenų modelio išplėtimai abstrakčiaisiais tipais, o tuo pačiu ir — objektinės—reliacinės duomenų bazių valdymo sistemos;
- Naujų duomenų modelių siūlymas. Semantiniai modeliai priklauso pastarajai kategorijai.

Reikia pasakyti, jog semantiniai modeliai nėra viena iš modeliavimo *technologijų* (kaip, pavyzdžiui, UML, E/R ir t.t.). Po "semantinių modelių" sąvoka mes suprasime tam tikrą sąvokų, principų ir mechanizmų rinkinį. Šis rinkinys turėtų padėti suprasti, kas turima omenyje tolesniuose paskaitų skyriuose.

I.A Esybės

Esybės — tai uždavinio srities *objektai*, kurie mus domina sistemos modeliavimo prasme ir kuriems mes norime priskirti kurias nors jų savybes.

Pavyzdžiui: "studentas Sigitas", "dėstytojas Ragaišis" ar "Objektinių duomenų bazių kursas" yra esybės modeliuojamoje Programų sistemos katedros darbo srityje. Dar kartą pabrėžiame, jog kiekviena esybė yra vienintelė ir nepakartojama, unikali modeliuojamoje srityje.

Žinoma, unikalumas unikalumui nelygus. Jeigu mes modeliuojame tik abstrakčias Programų sistemų katedros funkcijas, tai tuomet modeliui gali pakakti esybių "Dėstytojas", "Sekretorė", "Studentas" ir t.t. Tačiau jeigu pereisime prie konkrečios katedros veiklos modeliavimo, tai neapsieisime be esybių "dėstytojas Ragaišis" ar "studentas Sigitas".

I.B Asociacijos

Asociacijos padeda susieti kelias (nebūtinai dvi !) esybes ryšiu "`asociacijos_pavadinimas`". Pavyzdžiui, norint nusakyti, jog "asistentas Janeliūnas" dėsto "Objektinių duomenų bazių kursą", naudojama asociacija "dėsto", kuri sieja esybes "asistentas Janeliūnas" ir "Objektinių duomenų bazių kursas". Analogiškai, jeigu norime pasakyti, jog ODB kursas dėstomas asistento Janeliūno, tai susiejame esybę "Objektinių duomenų bazių kursas" su esybe "asistentas Janeliūnas" asociacija "yra dėstomas". Tokiu būdu, abipusis ryšys tarp ODB kurso ir asistento Janeliūno išreiškiamas dvejomis asociacijomis, kurių "kryptys" yra skirtingos.

*Šioje vietoje noriu pabrėžti, jog duotame pavyzdyje ryšys tarp dviejų esybių buvo sumodeliuotas **dvejomis** asociacijomis. Tai paaiškinama tuo, jog asociacijos yra kryptinės, ir ne šiaip kryptinės, o vienkryptės. Todėl gali būti taip, jog ryšys būna modeliuojamas tik viena asociacija. Tuomet viena esybė "žino" apie savo ryšį su kita esybe, o pastaroji apie šį ryšį — ne.*

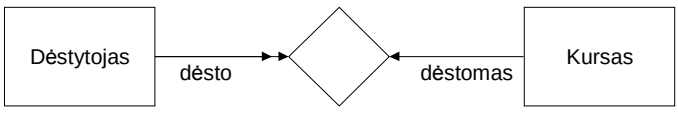
Yra skiriami du pagrindiniai asociacijų tipai:

- vienatinės asociacijos
- daugybinės (*multiple*) asociacijos.

Tokiomis asociacijomis ir modeliuojami visi ryšiai.

Ryšys paprastai nusakomas *kardinalumu*. Kardinalumas yra žymimas 1:N, 1:1 ir pan. Kaip kardinalumas susijęs su asociacijų tipu? Tarkime, kad ryšį modeliuojame dviem asociacijomis. Tuomet ryšys, kurio kardinalumas yra 1:1, bus modeliuojamas dvejomis vienatinėmis asociacijomis. Ryšys, kurio kardinalumas 1:N bus nusakomas viena daugybine ir viena vienatine asociacija. Ir pagaliau kardinalumą N:M ryšiui modeliuoti bus panaudotos dvi daugybin4s asociacijos.

Patogumo dėlei, asociacijas galime pavaizduoti ir grafiškai, pavyzdžiui, kaip parodyta paveikslėlyje 2:



Paveikslėlis 2 Grafinis asociacijų vaizdavimas

Labai svarbus reikalavimas asociacijoms yra šis: *asociacijomis galima susieti tik dvi EGZISTUOJANČIAS esybes*. Nesuklysime pasakę, kad toks reikalavimas yra natūralus, tačiau ar paprastas? Noriu atkreipti dėmesį, kad toks reikalavimas yra labai svarbus naikinant esybes iš modelio. Laikantis reikalavimo, turi būti pakeistos (ar — panaikintos) visos asociacijos, kuriose gali dalyvauti naikinamoji esybė, kad būtų išsaugotas modelio integralumas.

I.C Atributai

Kiekvienos esybės charakteristikos nusakomos tam tikromis jos savybėmis. Tos savybės vadinamos atributais. Pavyzdžiui, studijų knygelės numeris, lygus 91100457, gali būti esybės "studentas Sigitas" atributas. Kiekviena esybė gali turėti savus atributus (studentas Sigitas — studijų knygelės numerį, dėstytojas Ragaišis — darbo sutarties numerį ir pan.). Be to, skirtingos esybės, netgi turėdamos to paties pavadinimo atributus, gali turėti skirtingas jų reikšmes: pavyzdžiui, studento Sigito studijų knygelės numeris gali būti 91100457, o studentės Sigutės — 91100458.

Atributas vadinamas *vienatiniu*, jeigu jis gali vienu metu įgyti tiktai vieną reikšmę, ir jis vadinamas *daugybiniu*, jeigu vienu metu gali turėti aibę reikšmių.

Atributas vadinamas *paprastu*, jeigu jo reikšmė yra atominė (neskaidoma). Priešingu atveju jis vadinamas *sudėtiniu*. Pavyzdžiui, Asmens atributas `adresas` yra sudėtinis, nes adresas susideda iš gyvenvietės, gatvės bei namo numerio.

Atributas (arba atributų aibė) vadinamas *raktiniu*, jeigu jo (jų) reikšmė(—ės) vienareikšmiškai nusako esybę, to atributo ar atributų turėtoją (tradicinis *rakto* apibrėžimas).

Atributas yra *privalomas* (*totalus*), jeigu kiekvienas klasės elementas privalo turėti šio atributo reikšmę. Priešingu atveju atributas vadinamas *nebūtinu* (*fakultatyviu*).

Jeigu atributo reikšmė negali kisti, jis vadinamas *pastoviu*. Priešingu atveju — *kintamu*. Pavyzdžiui: Asmens gimimo data yra pastovus atributas, kai tuo tarpu jo pavardė ir netgi lytis gali kisti.

Atributas vadinamas *asociacija* (prisiminkime ankstesnįjį poskyrį !), jeigu jo reikšmė yra nuoroda į kokią nors esybę. Priešingu atveju atributas vadinamas *aprašančiuoju*.

Reikia pastebėti, jog su viena esybe susieta kita esybė nebūtinai turi būti "kita". Esybę, reikalui esant, galime susieti su ja pačia.

Atributas vadinamas *išvedamu*, jeigu jo reikšmė (reikšmės) gali būti išskaičiuojamos iš kitų atributų (netgi kitų klasių atributų) reikšmių. Pavyzdžiui, atributas `amžius` gali būti išskaičiuotas iš atributo `gimimo_data` reikšmės. Arba — "Dėstytojo" atributas `dėstomi_kursai` gali būti išskaičiuojamas, naudojantis klasės "Kursas" atributu `dėstantis_dėstytojas`.

Čia pateiksime bendrą lentelę atributų skirstyme:

Tipas	Alternatyva	Komentaras
Raktinis	Neraktinis	Atributas ar aibė raktinių atributų vienareikšmiškai nusako esybę

Privalomas	Nebūtinas	Kiekvienas klasės elementas turi šio atributo reikšmę
Paprastas	Sudėtinis	Šios atributo reikšmė yra neskaidoma
Vienatinis	Daugybinis	Atributas gali įgyti vieną ir tik vieną reikšmę vienu laiko momentu
Asociacija	Aprašantysis	Atributo reikšmė yra kurios nors klasės elementas
Išvedamas	Apibrėžtas	Atributas gali būti išskaičiuojamas iš kitų atributų reikšmių
Pastovus	Kintamas	Atributo reikšmė negali kisti

I.D Abstrahavimo mechanizmai

Kad būtų galima kuo tiksliau sumodeliuoti nagrinėjamą sritį, reikalingi tam tikri abstrakcijų įvedimo mechanizmai (*abstrahavimo* mechanizmai), leidžiantys kaip nors sugrupuoti esybes. Tuomet būtų galima akcentuoti tam tikras esybių savybes (atributus), pagal kurias ir buvo atliktas sugrupavimas.

Semantiniuose modeliuose siūlomi trys abstrahavimo mechanizmai:

- klasifikavimas,
- agregavimas,
- generalizavimas/specializavimas.

Tuojau pristatysime juos šiek tiek plačiau.

Klasifikavimas

Klasifikavimas yra tiesiog esybių sugrupavimas į *klases*. Imame visas tiriamas esybes ir suskirstome jas — štai ir viskas.

Šis mechanizmas leidžia tarti aibę skirtingų esybių esant homogeniškomis. T.y. naudojant šį mechanizmą, kiekviena esybė priskiriama kokiai nors klasei, kur klasė yra *aibė to paties tipo elementų*. Pavyzdžiui: galima tarti, jog "studentas Sigitas" ir "dėstytojas Ragaišis" yra dvi esybės, priklausančios tai pačiai "Asmens" klasei. Tai leidžia akcentuoti tokias bendras jų charakteristikas, kaip vardas, pavardė, gimimo data ir t.t.

Klasifikacijos mechanizmas turi du aspektus (į jį galima žiūrėti iš dviejų pozicijų), kuriuos reikėtų gerai suprasti:

- Klasė yra *aibė* (rinkinys, krūva, sambūris, ...) elementų, kurie yra mus dominančios duotu laiko momentu esybės. Būtent taip mes toliau dažniausiai ir suprasime klasę: kaip aibę, matematine to žodžio prasme. Tai yra *ekstensionalus* (išplėtimo) klasės aspektas, kintantis laiko atžvilgiu, nes mus dominančių esybių aibė su laiku kinta: vienos esybės tose aibėse atsiranda, kitos dingsta. Tuo pačiu kinta klasė, kuriai tos esybės priklauso(ė).

- Klasės elementai yra panašūs, nes yra to paties tipo. Todėl galime laikyti, jog klasė neatsiejama nuo tam tikro tipo, kurį atitikti turi visi jos elementai — esybės. Tai — intencionalus aspektas, nekintantis laiko atžvilgiu, nes, nors ir kintant mus dominančių objektų aibei, tos aibės elementų tipas nesikeičia.

Šie du aspektai (ar — koncepcijos) dažnai būna suplakami į vieną krūvą arba vienas iš jų (dažniausiai intensionalusis) užgožia kitą. Tačiau jie abu yra lygiai svarbūs ir tiek pat dažnai naudojami. Klasifikavimas išnaudoja būtent pirmąjį, ekstencionalųjį klasės sąvokos aspektą.

Agregavimas

Agregavimo mechanizmas leidžia apibrėžti klasę, nustatant jos elementų (esybių) struktūrą kaip įvairių sudėtinių dalių (atributų) agregatą. Šis agregatas aprašomas kaip baigtinė aibė atributų, bendrų visoms "kuriamai" klasei priklausančioms esybėms. Atributai aprašomi suteikiant jiems vardą, o taip pat ir tipą reikšmių, kurias atributas gali įgyti.

Taigi, agregavimo mechanizmas išnaudoja intensionalųjį klasės sąvokos aspektą.

Pavyzdžiui, įvairios mūsų tiriamos esybės turi pačius įvairiausius atributus: vardą, pavadinimą, spalvą, gimimo datą, tūrį ir pan. Galime nutarti, jog esybės, turinčios atributus `vardas`, `pavardė`, `gimimo_data`, priklauso vienai klasei ir tą klasę pavadiname "Asmuo". Panašiu būdu sugrupavę kitus atributus, galime "pasigaminti" ir kitas klases.

Generalizavimas ir specializavimas

Šiuos du mechanizmus galime taikyti, kai jau turime (po klasifikavimo ar agregavimo) tam tikras klases. Tuomet mes galime "pasigaminti" kitas klases, apibendrindami (generalizuodami) jau esančias klases, arba jas "pasmulkindami" (specializuodami). Tuomet klasės tampa susietos tarpusavyje tam tikra hierarchija. O klasių hierarchija nusakoma daline tvarka.

Pagal matematinį apibrėžimą, tranzityvus, nekomutatyvus sąryšis vadinamas tvarka. Esant tvarkai Θ*, jeigu* $\forall x, y : x \Theta y$ *arba* $y \Theta x$ *, tai tvarka vadinama totalia, priešingu atveju — daline.*

Klasės, kurias galima palyginti pagal šią tvarką, modeliuoja esybių klases skirtinguose abstrakcijos lygiuose. Klasė, kuri nėra mažesnė už jokią kitą klasę, vadinama *bazine klase* (jos gali būti kelios!). Priešingu atveju ji vadinama *poklasiu*.

Būtent šis mechanizmas "suveikia", kai mes suprantame kiekvieną Studentą ir Dėstytoją kaip Asmenį. Sakome, jog klasė "Asmuo" *generalizuoja* klases "Studentas" ir "Dėstytojas". Atvirkščias mechanizmas yra specializavimas. Šiuo atveju mes sakome, kad klasė "Studentas" *specializuoja* klasę "Asmuo".

Abiem atvejais, tiek Studentai, tiek Dėstytojai (esybės) galėjo būti suklasifikuoti (pavyzdžiui, naudojant klasifikavimo mechanizmą) ir kaip Asmenys. Tai reiškia, jog abiejų klasių esybės tikrai turi tam tikras savybes, bendras visiems asmenims. Asmenų skirstymas į Studentus ir Dėstytojus išreiškia tik šiek tiek tikslesnį abstrakcijos (modeliavimo) lygmenį, kai įvedamos papildomos charakteristikos. Tai ir atspindi šis specializavimas.

Generalizavimo/specializavimo mechanizmas turi dvi taisykles:

- jeigu klasė C_1 yra klasės C_2 poklasis, vadinasi kiekvienas klasės C_1 elementas yra ir klasės C_2 elementas (atsižvelgiama į ekstensionalų klasės aspektą);

- jeigu klasė C_1 yra klasės C_2 poklasis, vadinasi visi klasės C_1 elementai *paveldi* visas klasės C_2 savybes arba, kaip mes sakome — atributus (atsižvelgiama į intensionalų klasės aspektą).

Generalizavimas/specializavimas leidžia išskirti tris poklasių apibrėžimo modalumus:

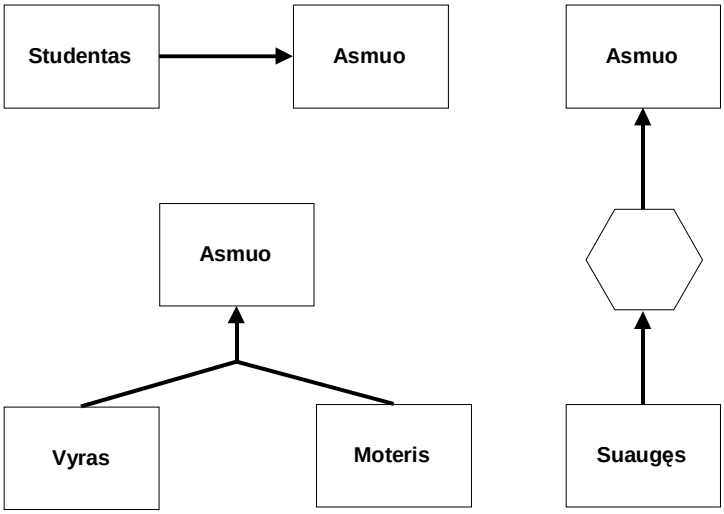
- įjungimą,
- padalijimą,
- apribojimą.

Pirmasis modalumas reiškia ryšį "yra" (*Is A*). Pavyzdžiui: Studentas *yra* Asmuo. Dėstytojas irgi *yra* Asmuo. Bendru atveju tai nereiškia, jog poklasiai nesikerta tarpusavy.

Antrasis modalumas nusako ne vien ryšį "yra", bet taipogi ir tą faktą, kad poklasiai yra nesikertantys. Pavyzdžiui: Asmuo yra Vyras arba Moteris. Reikia pažymėti, jog dviejų "padalintų" poklasių sąjunga yra "superklasės" (viršklasės) *poaibis*.

Poklasiai gali būti gaunami ir pritaikant tam tikrus apribojimus klasės elementų savybėms. Šis modalumas vadinamas apribojimu. Pavyzdžiui, Suaugusieji yra visi Asmenys, kurių `amžius` yra daugiau arba lygus 18 metų.

Generalizavimo/specializavimo sąryšį galima pavaizduoti ir grafiškai:



Paveikslėlis 3 Grafinis generalizavimo/specializavimo modalumų vaizdavimas

II. Neformalus objektinio modelio pristatymas

Pagrindinės objektinio modelio sąvokos yra šios:

- literalas,
- objektas,
- tipas,
- klasė,
- atributas,
- asociacija (*relationship*),
- metodas,
- sritis (*extent*),
- paveldėjimas,
- inkapsuliacija,
- polimorfizmas.

Būtent ties šiais aspektais ir bus koncentruojamas mūsų dėmesys šiame skyriuje.

Literalas yra bet kuri reikšmė, turinti savo tipą (arba, jeigu labiau patinka — bet kurio tipo reikšmė). Tokių reikšmių sukonstruoti ir išvardinti mes galime begales: 1, 2, 3.1414, "Pi" ir t.t. Tačiau literalas yra beasmenis, laikina substancija, abstrakcija. "Penki" šiuo momentu ir "penki" po metų yra vienas ir tas pats penketas. Tai sąvoka, nekintanti jau tūkstančius metų. Tokia sąvoka ir yra literalas. Bet jeigu mes penketą kaip nors identifikuosime ir nuo to momento kiekvieną sykį pagal tą identifikatorių sugebėsime atrasti, koks gi tai yra literalas — tuomet mes jau turėsime ne šiaip sau reikšmę, o *objektą*.

Reliacinė teorija turi vieną ypatybę: esybių (reliacinio pasaulio objektų) identifikavimui turi būti naudojami atominių tipų atributai, sudarantys "raktą". Iš vienos pusės tai yra natūralu — kaip gi kitaip mes galime nusakyti identifikuoti) objektus mūsų pasaulyje? Tačiau toks identifikavimo būdas gali pasirodyti ir kaip trūkumas, nes, visų pirma, tokiais raktais pakankamai sunku manipuliuoti (ypač, jeigu į raktą įeina ir simbolių eilutės; tuomet raktų skaitymo, lyginimo bei kitokių operacijų "kaina" smarkiai pašoka). Dar daugiau problemų atsiranda tuomet, kai keičiasi raktinio atributo reikšmė. Tada tenka keisti ir visus kitus, per tą raktą susietus, duomenis.

Objektinis modelis perkelia objektų identifikaciją iš konceptualaus į fizinį lygį. Jis įveda *objekto identifikatoriaus* sąvoką. Šis identifikatorius (*oid*) automatiškai suteikiamas kiekvienam objektui, saugomam duomenų bazėje ir išlieka nekintamas visą objekto gyvavimo ciklą, netgi jeigu keičiasi objekto atributai. Tokiu būdu, bet kuris objektas *o* gali būti "naudojamas" (susietas) kitų objektų, per jo identifikatorių oid. Bet kokie objekto atributų pakeitimai niekaip neatsispindi jį naudojančių objektų atributams, nes oid išlieka nepakitęs.

Ar tai nėra panašu iš žmogaus identifikavimo pagal vardą/pavardę pakeitimą į identifikavimą pagal DNR?

Žinia, atsietą nuo modeliuojamos srities oid sunkiau naudoti duomenų identifikavimui. Juk žmogų mes paprastai atpažįstame pagal jo vardą ir pavardę, ir todėl būtų labai nepatogu, jeigu pagal šiuos atributus mes negalėtume identifikuoti objekto. Todėl semantiniuose modeliuose buvo išskirti raktiniai esybių atributai.

Tas literalas, kurį identifikavę mes "pasigaminame" objektą, paprastai yra *struktūrinė reikšmė*. Ši struktūrinė reikšmė (dar vadinama "įrašu" — *tuple*) gali būti sudaryta tiek iš atominių reikšmių, tiek iš kitų struktūrinių reikšmių ar

aibių. Struktūroms gali priklausyti ir oid, kurie "rodytų" į tą patį ar kokį nors kitą objektą. Tie oid yra *asociacijos*, jau matytos semantiniuose modeliuose.

Kaip ir semantiniuose modeliuose, to paties tipo objektai yra grupuojami į *klases*.

Pateikime nedidelį pavyzdį O_2 notacija:

```
class Asmuo
  type tuple ( vardas: String,
               lytis: String,
             tevelis: Asmuo);
```

Pavyzdėlyje tariama, jog turime klasę "Asmuo". Šios klasės objektams priskiriamas įrašas (*tuple*), sudarytas iš dviejų atominių reikšmių (`vardas` ir `lytis`) bei vieno oid (`tevelis`), rodančio į kitą klasės "Asmuo" objektą.

Objektiniame modelyje, klasės naudojamos dar ir kaip struktūriniai vienetai, nusakantys objektų elgesį. Elgesys aprašomas priskiriant objektui tam tikrą aibę *metodų*. Metodas turi savo vardą, signatūrą ir implementaciją. Vardas ir signatūra tarnauja išoriniam metodo interfeisui. Metodo implementacija yra rašoma kuria nors programavimo kalba, kurios pasirinkimas nėra duomenų modelio dalis.

Štai dar vienas pavyzdys O_2 notacija:

```
method Kreipinys:String in class Asmuo
 {
   if (self.lytis == "VYR")
      return "Dėdė " + self.vardas;
   else
      return "Teta " + self.vardas;
 }
```

Čia mes turime metodo `Kreipinys` implementaciją, kuri parašyta O_2C kalba. Kadangi programavimo kalba nėra duomenų modelio dalis, ši implementacija galėjo būti parašyta ir C++ ar kuria nors kita programavimo kalba, kuriai realizuotas susiejimas su ODBS O_2 .

Klasės susietos tarpusavyje paveldimumo hierarchijos tvarka. Faktas, jog klasė c yra klasės C poklasis (užrašoma $c \prec C$) reiškia jog:

1. Reikšmių, priskiriamų klasės c objektams, tipas turi būti klasės C objektams priskiriamų reikšmių tipo potipis. Tipo/potipio ryšys nusako apribojimus objektų atributams ir bus aprašomas vėliau, formaliame objektinio modelio pristatyme. Intuityviai galime manyti, jog potipis "praturtina" savo supertipą tam tikromis detalėmis.

2. Jeigu klasė C turi metodą *m*, tai tokio pat vardo metodas turi egzistuoti ir klasėje c. Tam tikrais atvejais metodo *m* implementacija gali sutapti abiem klasėms. Tuomet sakoma, jog metodas *m* klasėje c yra *paveldėtas* iš klasės C ir antrą kartą perrašyti jo implementacijos nebereikia. Tačiau, jeigu metodo implementacija šioms dviem klasėms skiriasi, sakoma, jog klasės c metodas *užkloja* klasės C metodą *m*. Kuri metodo implementacija naudojama objektui *o*, nustatoma programos vykdymo metu pagal objekto *o* klasę. Šis pasirinkimas vadinamas *vėlyvuoju metodo išsprendimu*.

Dar viena svarbi sąvoka objektiniame modelyje yra *inkapsuliacija*. Inkapsuliacija leidžia ryškiai atskirti objektų manipuliavimo manierą nuo to, kaip tie objektai fiziškai saugomi duomenų bazėje. Nepaisant objektams suteikiamų struktūrinių reikšmių, manipuliuoti tomis reikšmėmis galima tik naudojantis klasės interfeisu.

Idealiu atveju klasės interfeisas yra jos metodai, tačiau yra priimta ir "viešų" atributų bei asociacijų sąvoka.

Todėl po vienodu interfeisu gali slėptis skirtinga objektų struktūra ar net (metodų užklojimo dėka) skirtingas elgesys. Ir tuomet "iš pažiūros" vienodi (interfeiso prasme) objektai gali elgtis skirtingai: tai mes vadiname *polimorfizmu*.

III. Formalus objektinio modelio pristatymas

Dabar pereisime prie formalaus objektinio modelio pristatymo.

Iš pradžių išvardinkime konstantinius dalykus, kurių egzistavimą laikysime savaime suprantamu ir jų struktūros nenagrinėsime.

Tarkime, kad turime:

- begalinę objektų identifikatorių aibę **obj** = { o_1 , o_2 , ... };
- klasių vardų aibę **class** = { c_1 , c_2 , ... };
- atributų vardų aibę **att** = { a_1 , a_2 , ... };
- metodų vardų aibė **meth** = { m_1 , m_2 , . . . }.

Naudodamiesi šiais pažymėjimais, galime pradėti apibrėžti pagrindines objektinio modelio sąvokas.

III.A Literalai ir objektai

Objektiniame modelyje išskirsime atskirus objekto ir literalo (reikšmės) konceptus. Iš pradžių apibrėšime, kas yra literalas, o paskui, naudodamiesi šia sąvoka, apibrėšime objektą.

Šalia jau išvardintų keturių "konstantinių" aibių, tarsime, jog egzistuoja atominiai duomenų tipai, kurių reikšmės imamos iš atskirų domenų. ODMG standartas jų (atominių tipų) išskiria vienuoliką:

1. Long,
2. Short,
3. Unsigned long,
4. Unsigned short,
5. Float,
6. Double,
7. Boolean,
8. Octet,
9. Char,
10. String,
11. Enum.

*Čia pastebėsime, jog nėra rodyklės (*pointer*) tipo. Šis tipas nėra tinkamas Objektinėms duomenų bazėms, nes tos pačios rodyklės naudojimas, naudojant tą patį objektą keliose programose, yra nepriimtinas.*

Tais pačiais vardais vadinsime ir šių duomenų tipų domenus. Visų išvardintų domenų sąjungą žymėsime **dom**. Domeno **dom** elementus vadinsime *konstantomis*. Specialiu simboliu *nil* žymėsime neapibrėžtą reikšmę— konstantą.

Tuomet literalu arba paprasčiausiai — reikšme — vadinsime:

Apibrėžimas Esant duotai oid aibei $O \subset$ **oid**, reikšmių aibę virš O apibrėšime kaip:

1. *nil* yra reikšmė virš aibės O;
2. visi **dom** priklausantys elementai yra reikšmės virš aibės O;
3. visi O elementai yra reikšmės virš aibės O;
4. jeigu $v_1 , \ldots , v_n$ yra reikšmės virš aibės O, o $a_1 , \ldots , a_n$ yra skirtingi atributų vardai iš **att**, tuomet struktūra $[a_1 : v_1 , \ldots , a_n : v_n]$ yra reikšmė virš aibės O ;
5. jeigu $v_1 , \ldots , v_n$ yra reikšmės virš aibės O, tai aibė $\{v_1 , \ldots , v_n \}$ yra reikšmė virš aibės O.

Reikšmių virš O aibė žymima **val**(O).

Kaip matome, apibrėžimas yra rekursyvus. Iš pradžių apibrėžiama, jog reikšmės yra *nil*, konstantos bei objektų identifikatoriai (tai — reikšmių aibės **val**(O) konstravimo bazė). Vėliau, remiantis tomis reikšmėmis, galime konstruoti įvairias aibes bei struktūras.

ODMG standarte aibė turi kelias galimas variacijas:

- aibė *Set*;
- multiaibė *Bag* (aibė, galinti turėti pasikartojančius elementus);
- sąrašas *List* (sutvarkyta aibė);
- masyvas *Array*.

Tačiau formaliame modelio apibrėžime šių skirtingų aibių tipų neskirsime ir naudosime vienintelį žymėjimą { }.

Štai keletas pavyzdžių:

```
Reikšmės:

    1,

    "Reikšmė",

    oid12,

    [ kino_teatras: oid12,
      laikas: "16h30",
      kaina: nil,
      filmas: oid4
    ],
```

```
    { "G.Massina", "S.Loren", "M.Mastroianni" },

    [ pavadinimas: "La Strada",
      rezisierius: "F.Fellini",
      aktoriai: {oid25, oid14, oid51}
    ]
```

O dabar, kai jau žinome, kas tai yra reikšmė, galime apibrėžti ir objektą:

Apibrėžimas Objektu vadinsime porą <*identifikatorius*, *reikšmė* >. Identifikatoriai yra iš aibės **obj**, o reikšmės yra struktūra arba aibė.

Tiek objektas, tiek literalas turi savo "reikšmę". Kaip matome iš apibrėžimo, skirtumas yra tas, jog objektas yra pora $<o, v>$ (kur o yra objekto oid, o v yra reikšmė), o literalas tėra viena reikšmė v. Be to, objektai įgyja tik struktūros arba aibės (kartais dar vadinamos *kolekcija*) tipo reikšmes. Literalas, be šių dviejų tipų, gali įgyti ir atominę (priklausančią **dom**) ar objekto identifikatoriaus reikšmę.

Pateikiame ir keletą objektų pavyzdžių:

```
Objektai:

    < oid147, {"G.Massina", "S.Loren", "M.Mastroianni"} >

    < oid148, [ pavadinimas: "La Strada",
                rezisierius: "F.Fellini",
                aktoriai: {oid25, oid14, oid51}
              ] >
```

III.B Klasės ir tipai

Kiekvienas literalas apibūdinamas jo tipu (arba tiesiog — turi tipą).

Literalus konstruoti mes jau mokame, vadinasi dabar reikia "išmokti" konstruoti tipus. Tik tuomet, kai žinosime, kas yra tipas, galėsime bandyti sieti juos su atitinkamais literalais.

Tipai apibrėžiami kokios nors klasių vardų aibės C atžvilgiu.

Apibrėžimas Esant duotai klasių vardų aibei $C \subset$ **class**, tipai virš aibės $\boldsymbol{C}$ apibrėžiami kaip:

1. klasės vardas **any** yra tipas;
2. visi atominiai tipai (**short, long, unsigned short** ir t.t.) yra tipai;
3. klasių **vardai** iš C yra tipai;
4. jeigu $\tau_1, \ldots, \tau_n$ yra tipai, o $a_1, \ldots, a_n$ yra skirtingi atributų vardai iš **att**, tuomet $[a_1 : \tau_1, \ldots, a_n : \tau_n]$ yra struktūrinis tipas;
5. jeigu τ yra tipas, tuomet $\{ \tau \}$ irgi yra tipas (aibė).

Tipų virš C aibė žymima **types**(C).

Kaip matome, tipų apibrėžime irgi yra penki punktai, kaip ir literalo apibrėžime. Ir kiekvienas iš tipo apibrėžimo punktų savotiškai atitinka reikšmės apibrėžimo punktą. Išimtis yra tik reikšmė *nil* ir tipas **any**. Apie pastarąjį pakalbėsime vėliau.

*ODMG standarte klasę **any** atitinka klasė "Object".*

Greta jau duotų atominių tipų, ODMG standarte iš anksto apibrėžti (*predefined*) ir keletas struktūrinių tipų. Štai jie:

- Date,
- Interval,
- Time,
- Timestamp.

Juos taip pat prijunkime prie atominių duomenų tipų.

Tipams galima suteikti vardus.

Štai keletas tipų pavyzdžių:

```
Tipas:

    KinoTeatras,

    { Time },

    [ kino_teatras: KinoTeatras,
            laikas: String,
             kaina: Short,
            filmas: Filmas
    ]

Keletas įvardintų tipų:

    TypeKinoTeatras = [ pavadinimas: String;
                              adresas: String;
                            telefonas: Long
                        ]

    TypeFilmas = [ pavadinimas: String;
                   rezisierius: String;
                      aktoriai: {Asmuo}
                 ]
```

Kai turime apibrėžę tipą, galima apibrėžti, kas yra klasė:

Apibrėžimas Klasė – tai objektų, kurie saugo to paties tipo reikšmes, aibė.

Norėtume dar sykį atkreipti dėmesį į tai, kad klasė yra objektų aibė (ir nieko daugiau!) ir kad visi toje aibėje esantys objektai privalo turėti to paties tipo reikšmes (pamenate, kad objektas yra pora $<o, v>$...).

Tuomet galimi klasių apibrėžimai (O_2 notacija):

```
    Class KinoTeatras
```

```
    type TypeKinoTeatras
end;

class Filmas
    type TypeFilmas
end;
```

Čia klasė `KinoTeatras` bus aibė objektų, o tie objektai bus tokios poros $<o, v>$, kur reikšmė v yra tipo `TypeKinoTeatras`.

Taigi, tokiu būdu su kiekviena klase c yra asocijuojamas "jos" tipas, žymimas $\sigma(c)$.

Apibrėžimas Jeigu $\boldsymbol{C}$ yra klasių vardų aibė $C \subset$ **class**, tuomet $\sigma(c)$ yra funkcija $\sigma : C \to$ **types** (C).

Pastebime, kad apibrėžime panaudota funkcijos sąvoka. Kyla klausimas, kaip ta funkcija apibrėžiama? Ogi labai paprastai — išvardinimu. Paskutinis mūsų duotas pavyzdys ir yra tokios funkcijos apibrėžimo būdas.

III.C Klasių hierarchija

Klasės yra organizuotos pagal *specializacijos hierarchiją*. Tokia hierarchija apibrėžiama kaip:

Apibrėžimas Klasių hierarchija yra trejetas $< C, \sigma, \prec >$, kur

- C yra baigtinė klasių vardų aibė,
- $\sigma : C \to$ **types** (C),
- $\prec$ yra dalinės tvarkos sąryšis aibėje C.

Natūralu, jog į klasių hierarchijos apibrėžimą įeina jų vardų aibė $\boldsymbol{C}$. Dalinė tvarka $\prec$ tarp klasių organizuoja jas į hierarchinį medį, kuris *privalo turėti* vienintelę šaknį. Ta šaknis (šakninė klasė) ir yra žymima **any**. O funkcija σ į klasių hierarchiją įvedama tam, kad būtų galima įvesti tipo/potipio sąryšį.

Kalbant neformaliai, jeigu mes turime klasę ir jos poklasį, tai poklasio tipas turi būti "praturtintas" superklasės tipas. Poklasio tipas gali būti *specializuotas*, prie superklasės tipo pridedant naujus atributus ar asociacijas. Pavyzdžiui (jeigu grįšime prie senųjų pavyzdžių apie Studentus, Asmenis ir t.t.), visiškai natūralu, jog klasė "Studentas" patikslina (papildo naujais atributais) klasėje "Asmuo" saugomus duomenis.

Šiam tipų "specializavimui" išreikšti naudojamas tipo/potipio sąryšis $\leq$. Užrašas $\tau_1 \leq \tau_2$ reiškia jog "tipas τ_1 yra tipo τ_2 potipis".

Taigi:

Apibrėžimas Tegu $< C, \sigma, \prec >$ yra klasių hierarchija. Tuomet tipo/potipio sąryšis $\leq$ yra dalinė

---

tvarka aibėje **types**(*C*), tenkinanti sąlygas

1. $c \prec k \Rightarrow c \le k$,
2. $(\forall i \in [1, n], n \le m : \tau_i \le \tau'_i) \Rightarrow [a_1 : \tau_1 , \dots , a_m : \tau_m] \le [a_1 : \tau'_1 , \dots , a_n : \tau'_n]$,
3. $\tau \le \tau' \Rightarrow \{ \tau \} \le \{ \tau' \}$,
4. $\forall \tau : \tau \le$ **any**.

Dažnai iškyla nesusipratimų dėl šio apibrėžimo 2 punkto. Kodėl $n \le m$? Atsakymas paprastas: struktūra, kuri turi daugiau elementų, negu kita, tėra tos kitos struktūros papildymas, praturtinimas, vadinasi — potipis.

Kaip matome, dabar mes turime dvi tvarkas: tvarką $\prec$ tarp klasių ir tvarką $\le$ tarp su tomis klasėmis susietų tipų. Koks šių dviejų tvarkų santykis?

Apibrėžimas Klasių hierarchija vadinama *geros struktūros*, jeigu kiekvienai klasių porai c ir *k* yra tenkinama sąlyga

$$c \prec k \Rightarrow \sigma(c) \le \sigma(k).$$

Pavyzdžiui, mūsų paskutiniame pavyzdyje tipas `TypeKinoTeatras` nėra tipo `TypeFilmas` potipis. Todėl, jeigu, formuodami klasių hierarchiją, nurodysime `TypeKinoTeatras` $\prec$ `TypeFilmas` , tai mūsų klasių hierarchija nebus geros struktūros.

Klasių hierarchijos, kurios nėra geros struktūros, yra tam tikra prasme išsigimęs atvejis ir toliau mes į jas nekreipsime dėmesio, o nagrinėsime tik geros struktūros klasių hierarchijas.

III.D Klasių hierarchijos ir tipų semantika

Dabar apibrėšime klasių ir tipų semantiką. Klasės semantika — tai atsakymas į klausimą "Kokie objektai priklauso tai klasei?". Analogiškai tipo semantika atsako į klausimą, "Kokios yra šio tipo reikšmės?".

Pagrindinė šio poskyrio sąvoka yra *oid priskyrimas*, nusakanti, kaip su kiekviena klase susiejama tam tikra oid aibė, t.y. kaip klasė "gauna" savo objektus.

Apibrėžimas Tegu $< C, \sigma, \prec >$ yra (geros struktūros) klasių hierarchija. *Oid priskyrimas* yra funkcija π, kuri kiekvienam aibės *C* elementui priskiria atskirą oid aibę.

Esant duotam oid priskyrimui π, klasės c *sava sritis* yra $\pi(c)$, o klasės c *sritis* (žymimas $\pi^*(c)$) yra

$$\bigcup \{ \pi(k) \mid k \in C, k \prec c \text{ arba } k = c \}.$$

Kaip matome, oid priskyrimas yra paprasčiausia funkcija π, kuri ima ir susieja klasės vardą *c* su kažkokia oid aibe (nesvarbu kaip!). Ir jeigu klasės *c* sava sritis yra jai ir tik jai priklausančių objektų identifikatoriai, tai klasės *c* sritis (apskritai) apima ir visiems klasės *c* poklasiams priklausančių objektų oid.

Pagal tokį apibrėžimą, esant duotam oid priskyrimui π, jeigu $c \prec k$, tai $\pi^*(c) \subseteq \pi^*(k)$. Taip išreiškiamas faktas, jog kiekvienas klasės *c* objektas priklauso ir klasei *k*. Tipizavimo požiūriu, ši išraiška reiškia, kad visos operacijos, kurias galima atlikti su klasės *k* objektais, tinka ir klasės *c* objektams. T.y. klasės *c* objektai gali būti naudojami bet kur, kur tikimasi klasės *k* objekto.

Dabar galima apibrėžti ir tipų semantiką. Šis apibrėžimas daromas duotos klasių hierarchijos ir duoto oid priskyrimo atžvilgiu.

Apibrėžimas Tegu $< C, \sigma, \prec >$ yra klasių hierarchija. Tegu $O = \cup\{ \pi(c) \mid c \in C \}$. Tada prilyginkime $\pi^*(\mathbf{any}) = O$. Tuomet tipo τ *sava interpretacija* , žymima $dom(\tau)$, yra apibrėžiama kaip:

1. $dom(\mathbf{any}) = \mathbf{val}(O)$;
2. kiekvienam atominiam tipui τ, $dom(\tau)$ yra įprastinė to tipo interpretacija;
3. $\forall c \in C : dom(c) = \pi^*(c) \cup \{ nil \}$;
4. $dom(\{\tau\}) = \{ \{v_1, \ldots, v_n\} \mid n \geq 0, v_i \in dom(\tau), i \in [1, n] \}$
5. $dom([a_1 : \tau_1, \ldots, a_n : \tau_n]) = \{[a_1 : v_1, \ldots, a_n : v_n] \mid v_i \in dom(\tau_i), i \in [1, n] \}$.

Kaip matome, kiekvieno tipo τ reikšmės žymimos $dom(\tau)$. Ir kiekvienam tipo apibrėžimo punktui yra atskiras punktas, aprašantis, kaip atrodo to tipo aibė *dom*.

III.E Elgesys ir metodai. Interfeisas

Klasė apsprendžia (aprašo) jai priklausančių objektų *elgesį*. Elgesys nusakomas metodais, arba, jeigu kalbėsime tiksliau — metodų aibe.

Metodas susideda iš trijų sudėtinių dalių:

- vardas,
- signatūra,
- implementacija.

Apibrėžti (užrašyti) metodo vardą bei signatūrą nėra sunku. O štai, norint užrašyti jo implementaciją, reikia programavimo kalbos ir mokėti programuoti. Kadangi programavimo kalbos pasirinkimas neįeina į objektinį modelį, tai apie metodų implementacijų rašymą mes daug ir nekalbėsime. Bendrame, abstrakčiame objektinio modelio aprašyme daugiau koncentruosimės į metodo signatūrą bei jo taikymą objektui.

Metodų vardai ir signatūros yra apibrėžiami kartu su tipais, klasėmis bei jų hierarchijomis.

Objektiškai orientuotų programavimo kalbų terminologijoje dažnai sakoma, jog metodo `m` vardas yra `pranešimas`, o metodo iškvietimas objektui vadinamas `pranešimo pasiuntimu` tam objektui. Pats gi objektas vadinamas `pranešimo gavėju`.

Apibrėžimas Tebūnie $< C, \sigma, \prec >$ klasių hierarchija. Metodo vardui m , jo *signatūra* yra išraiška

$$m : c \times \tau_1 \times \ldots \times \tau_{n-1} \rightarrow \tau_n$$

kur c yra klasės vardas iš C, kiekvienas τ_i yra tipas virš C ($\tau_i \in$ **types**(C)).

Šita signatūra susiejama su klase c ir yra sakoma, jog metodas m yra *taikomas* klasės c objektams. Kadangi, kaip jau sakėme, visi klasės c poklasiams priklausantys objektai priklauso ir klasei c, tuomet gauname situaciją, kad tas pats metodas gali būti taikomas ir skirtingų klasių objektams. Dar daugiau: tam pačiam metodo vardui m galima turėti skirtingas signatūras, susietas su skirtingomis klasėmis. Tačiau visoms šioms galimybėms taikomos tam tikros išlygos, apie kurias ir pakalbėkime.

Pabandysime apibrėžti *užklojimą*, *paveldėjimą* ir *išsprendimą*. Tarkime, turime tris klases (pavyzdžiuose viskas bus pateikiama O_2 notacija):

```
class Asmuo
  type TypeAsmuo
end;

class Tarnautojas inherits Asmuo
  type TypeTarnautojas
end;

class Destytojas inherits Tarnautojas
  type TypeDestytojas
end;
```

Visose trijose klasėse yra metodas `Kreipinys`.

```
Method Kreipinys:String in class Asmuo;
method Kreipinys:String in class Tarnautojas;
method Kreipinys:String in class Destytojas;
```

Šių metodų signatūros yra:

```
Kreipinys : Asmuo       → String
Kreipinys : Tarnautojas → String
Kreipinys : Destytojas  → String
```

O implementacijos yra tokios:

```
method Kreipinys:String in class Asmuo
 {
   if (self.lytis == "VYR")
      return "Dėdė " + self.vardas;
   else
      return "Teta " + self.vardas;
 }

method Kreipinys:String in class Tarnautojas
 {
   return "Tarnautojas " + self.vardas;
 }

method Kreipinys:String in class Destytojas
 {
   return "Dėstytojas " + self.vardas;
```

```
|       }
```

Klasė "Tarnautojas" yra klasės "Asmuo" poklasis. Vadinasi, klasės "Asmuo" metodas `Kreipinys` turi būti pritaikomas ir klasei "Tarnautojas" (kadangi kiekvienas tarnautojas kartu yra ir Asmuo). Tuomet sakoma, jog klasė "Tarnautojas" *paveldi* metodą `Kreipinys` iš klasės "Asmuo". Tariant formaliai:

Apibrėžimas Esant duotoms klasėms *c* ir *k*, tokioms, kad:

1. metodas *m* būtų apibrėžtas klasėje *c*,
2. $k \prec c$,
3. neegzistuoja jokios kitos klasės *p*, tokios, kad $k \prec p \prec c$, ir tokios, kad metodas *m* būtų apibrėžtas klasėje *p*

tuomet sakoma, jog klasė *k paveldi* metodą *m* iš klasės *c*.

Pastebėkime, jog mūsų pavyzdyje klasė "Dėstytojas" savo metodą `Kreipinys` paveldi ne iš klasės "Asmuo", o iš klasės "Tarnautojas".

Dažnai būna taip, jog viena klasė, specializuojanti kokią nors kitą, bendresnę klasę, privalo turėti ir "specializuotą" elgesį. T.y. į tą patį pranešimą tos klasės objektas privalo reaguoti šiek tiek kitaip, negu bendresnės klasės objektai. Tai reiškia, jog metodą paveldinčioje klasėje privalo būti galimybė pakeisti jo implementaciją.

Mūsų pavyzdyje, tas pats metodas `Kreipinys` turi tris skirtingas implementacijas ir čia mes turime reikalą su *užklojimu*. Jeigu kalbant tiksliau — metodo `Kreipinys` implementacija klasėje "Tarnautojas" užkloja to paties metodo implementaciją klasėje "Asmuo". Tas pats ir su klasių "Dėstytojas" bei "Tarnautojas" atveju: metodo implementacija klasėje "Dėstytojas" užkloja to paties metodo implementaciją klasėje "Tarnautojas".

Tarkime, jog klasė *k* yra klasės *c* poklasis ($k \prec c$) ir mes turime metodą

$$m : \mathrm{c} \times \tau_1 \times \ldots \times \tau_{n-1} \to \tau_n \qquad (1)$$

Taip pat tarkime, jog klasė *k* užkloja metodo *m* implementaciją sava implementacija. Tuomet automatiškai turime klasėje *k* ir naują, kitą *to paties metodo* signatūrą

$$m : \mathrm{k} \times \tau'_1 \times \ldots \times \tau'_{j-1} \to \tau'_j \qquad (2)$$

Tačiau signatūrų rašymas turi atitikti du reikalavimus:

1. *Neprieštaringumas*. Jeigu *c* yra klasių *k* ir *p* poklasis (tiesioginis), ir jeigu metodas *m* aprašytas abiejose klasėse *k* ir *p*, tuomet metodo *m* aprašymas privalo būti ir klasėje c.

2. *Kovariacija*. Jeigu $m : \mathrm{c} \times \tau_1 \times \ldots \times \tau_i \to \tau$ ir $m : \mathrm{k} \times \tau'_1 \times \ldots \times \tau'_j \to \tau'$ yra dvi skirtingos metodo *m* signatūros, o $k \prec c$, tuomet i = j ir $\forall \mathrm{n} : \tau'_n \le \tau_n$ bei $\tau' \le \tau$.

Pirmasis reikalavimas padeda išvengti prieštaravimų, kai klasė turi dvi ar daugiau tiesioginių superklasių. O apie antrąjį reikalavimą bus kalbama vėliau, aprašant tipizavimą bei kovariacijos—kontravariacijos konfliktą.

Pagal kovariacijos reikalavimą, signatūrose (1) ir (2) turi būti j = n.

Jeigu klasėje k ($k \prec c$) nėra apibrėžta kita metodo m implementacija, tuomet ji sutampa su tąja iš klasės c. Tačiau, jei šios dvi implementacijos skiriasi, tuomet programos vykdymo metu reikia rinktis, kuri iš jų yra naudojama duotu momentu.

Apibrėžimas Pasirinkimas, kuri iš klasės c metodo m implementacijų turi būti vykdoma programos veikimo metu, vadinamas *metodo m išsprendimu*. O pasirinktoji metodo implementacija vadinama *m sprendiniu* klasei c.

Tarkime, jog turime klasės k ($k \prec c$) objektą o, kuriam yra taikomas metodas m. Kurią metodo m implementaciją reikia taikyti objektui o ? T.y. reikia išspręsti metodą m. Yra du sprendimo būdai:

1. *Statinis susiejimas*. Šiuo būdu, metodas išsprendžiamas pagal prieš programos vykdymą išskaičiuotą objekto o klasę. Sprendžiama yra kompiliavimo metu. Tačiau jau žinome, jog klasės k objektas gali būti naudojamas ir kaip klasės c objektas. Tai reiškia jog, nepaisant tikrosios objekto o klasės (k), jeigu prieš programos vykdymą jo klasė bus išskaičiuota c, tai jam bus taikoma klasės c metodo implementacija.

2. *Dinaminis susiejimas*. Pagal šią strategiją, metodas išsprendžiamas programos veikimo metu, atsižvelgiant į tikrąją objekto o klasę. Taigi, mūsų atveju, jam bus pritaikyta klasės k metodo implementacija.

Kuo remiantis galima išspręsti metodą m klasei c ? Vienos programavimo kalbos, sprendžiant metodą, naudoja tiktai objekto—gavėjo klasę. Kitos kalbos atsižvelgia dar ir į visų metodo parametrų tipus (τ_1 , ... , τ_{n-1}).

Dabar pateiksime formalų objekto interfeiso apibrėžimą:

Apibrėžimas Objektui taikomų metodų aibė vadinama objekto *interfeisu*.

ODMG—2.0 standarte buvo patikslinta, jog interfeiso apibrėžimas sudarytas iš metodų vardų ir signatūrų aibės. Atominiams objektams (t.y. objektams, kurių reikšmė nėra aibė) prie interfeiso aprašymo prisideda dar ir atributų vardai su jų tipais, o taip pat ir asociacijų vardai su jų tipais bei *atgalinių* asociacijų pavadinimais.

Inkapsuliacijos principas sako, jog visos manipuliacijos su objektu gali būti atliekamos tik per jo interfeisą.

III.F Schemos ir egzemplioriai

Duomenų bazės schema aprašo joje saugomų duomenų struktūrą. Čia aprašomi tipai, klasės ir jų hierarchija, interfeisai ir vardai.

Jau minėjome, jog tipams galima suteikti vardus. Tačiau vardus galima suteikti ir literalams arba objektams. Tuomet juos duomenų bazėje galima "pasiekti"

tiesiogiai, naudojantis vardu. Objektų ir literalų vardai yra duomenų bazės "įėjimo taškai", vadinami *saugojimo šaknimis* (*persistence root*). Visi įvardinti objektai išlieka duomenų bazėje (t.y. nėra sunaikinami po programos darbo). Ir tikai naudojantis įvardintų objektų/reikšmių vardais, galima pasiekti duomenų bazėje saugomą informaciją.

Pavyzdžiui, galime "apibrėžti" savo duomenų bazėje keletą saugojimo šaknų:

```
name KinoTeatai     : set(KinoTeatras);
name KinoFilmai     : set(Filmas);
name SavaitesFilmas : Filmas
```

Matome, jog mes saugome aibę kino teatrų, aibę kino filmų ir, galų gale, įvardijame dar ir vieną objektą, kuris atstovauja savaitės filmą. Reikia priminti, jog, kartu su kiekvienu klasės "Filmas" objektu, aibėje "KinoFilmai" turėtų būti išsaugomi ir tie klasės "Asmuo" objektai, kurie priklauso tipo "TypeFilamas" atributui "Aktoriai" (tipas "TypeFilmas" aprašytas 31 puslapyje).

O dabar — formalus apibrėžimas:

Apibrėžimas *Duomenų bazės schema* yra penketas $\mathbf{S} = < C, \sigma, \prec, M, G >$, kur:

1. G yra vardų aibė, nesikertanti su C,
2. σ yra funkcija $\sigma : G \cup C \to \mathbf{types}(C)$,
3. $< C, \sigma, \prec >$ yra geros struktūros hierarchija,
4. M yra metodų signatūrų aibė

Čia aibę G sudaro visi saugojimo šaknų vardai. Funkcija σ suteikia visiems vardams iš G ir klasėms iš C jų tipus. Tvarkos sąryšis $\prec$ nusako klasių (ir tipų!) hierarchiją. Na, ir galų gale schemoje mes turime dar ir metodų signatūrų aibę M.

Su kiekviena DB schema galime susieti vieną arba keletą tos schemos *egzempliorių* (*instance*). Paprastai tariant, jeigu duomenų bazės schema yra jos struktūros aprašymas, tai schemos egzempliorius yra duomenys, tenkinantys tą aprašymą.

Apibrėžimas Schemos $< C, \sigma, \prec, M, G >$ *egzempliorius* yra ketvertas $\mathbf{I} = < \pi, \nu, \gamma, \mu >$, kur:

1. π yra oid priskyrimo funkcija,
2. ν yra funkcija $\nu : O \to \mathbf{val}(O)$ (čia $O = \cup\{ \pi(c) \mid c \in C \}$), išlaikanti tipų suderinamumą, t.y. $\forall c \in C\ \forall o \in \pi(c) : \nu(o) \in dom(\sigma(c))$,
3. γ yra funkcija, priskirianti kiekvienam vardui iš G , kurio tipas τ, reikšmę domene $dom(\tau)$,
4. μ priskiria kiekvienam metodui iš M jo semantiką[3]. Tiksliau tariant, kiekvienai signatūrai $m{:}c \times \vec{\alpha} \to \tau$,

$$\mu\left(m{:}c \times \vec{\alpha} \to \tau\right) \equiv \ f : dom\left(c \times \vec{\alpha}\right) \to dom\left(\tau\right).$$

---

[3] Metodo semantika — ką tas metodas daro.

Kaip matome, kurdami atskirą DB schemos egzempliorių, visas klases užpildome jų objektais (funkcija π), visiems objektams suteikiame jų reikšmes (funkcija ν). Visoms saugojimo šaknims iš *G* taipogi suteikiama jų semantika (funkcija γ), o visoms metodų signatūroms suteikiame jų implementacijas (funkcija μ)

Galima teigti, jog klasės, objektai, literalai ir vardai aprašomi ekstensionaliai (t.y. suteikiant jiems turinį, aibę), o metodai — intensionaliai (nusakant, ką tie metodai turi daryti).

Tuo baigiame skyrių apie objektinį duomenų modelį, taikomą objektinėse DB. Kitas skyrius — apie duomenų aprašymą, t.y. *Object Definitioon Language* ODL.

Duomenų apibrėžimo kalba ODL

Skyrelis paremtas medžiaga iš [Sec97], [Banc96].

Apibrėžus ODB duomenų modelį, reikėtų papasakoti, kaip jų konstruktai aprašomi kuria nors kalba.

Šiame skyriuje trumpai pristatysime ODMG siūlomą duomenų apibrėžimo kalbą ODL. Egzistuoja skirtingos ODL "modifikacijos", pritaikytos skirtingoms programavimo kalboms. Tačiau šiame skyriuje mes pateiksime "standartinio" kalbos varianto šiek tiek supaprastintus pagrindinius konceptus.

I. Tipai

Iš pradžių pažiūrėkime, kaip ODL aprašomi įvairūs tipai.

```
<tipas> ::= <atominis_tipas> |
            <klasės_vardas> |
            <aibės_tipas> |
            <struktūros_tipas> |
            <išvardinamasis_tipas>
```

Atominiai tipai

Atominius tipus jau išvardijome skyriuje "Objektinis modelis". Todėl dabar pateiksime paprasčiausią jų sąrašą:

```
<atominis_tipas> ::= long | short | unsigned long |
                     unsigned short | float | double |
                     boolean | octet | char | string |
                     date | interval | time | timestamp
```

Klasės

Klasės vardas yra paprasčiausias vardas:

```
<klsės_vardas> ::= <vardas>
```

Aibės

Aibės tipas aprašomas naudojantis kelis galimus aibių konstruktorius. Skiriasi tik masyvo apibrėžimas.

```
<aibės_tipas> ::= <aibės_konstruktorius> < <tipas> > |
                  <masyvo_apibrėžimas>

<aibės_konstruktorius> ::= set | bag | list
```

Aibės konstruktorius nurodo, kokia bus aibė, o po to įrašomas tipas reiškia, kokio tipo elementai bus saugomi toje aibėje. Pavyzdžiui, galime apibrėžti tokią asmenų aibę:

```
Set <Asmuo>
```

O masyvai apibrėžiami taip:

```
<masyvo_apibrėžimas> ::= <tipas> <vardas> [<dydis>]
                                          [{, [<dydis>]}]
```

Matome, jog ODL leidžia apibrėžti fiksuoto ilgio ir neribotos dimensijos masyvus. Štai pavyzdys su keliais apibrėžtais masyvais:

```
octet TicTacToe [3][3];

char Eilute [256]
```

Struktūros

Struktūros ODL aprašomos naudojantis raktiniu žodžiu **struct**:

```
<struktūros_tipas> ::= struct <vardas>
                       {
                        <tipas>    <vardas>    [{;    <tipas>
<vardas>}]
                       }
```

Vardas po raktinio žodžio **struct** apibrėžia naujo tipo vardą.

Štai keletas pavyzdžių, kuriuose apibrėžiame naujus struktūrinius tipus:

```
struct Data
{
  octet Diena;
  octet Menuo;
  unsigned short Metai;
};

struct Mokinys
{
  string Vardas;
  string Pavarde;
  short Pazymiai [10];
}
```

Išvardinamasis tipas

Išvardinamajame tipe yra išvardijamos visos reikšmės, kurias gali įgauti šio tipo objektai (kintamieji). Tipas apibrėžiamas taip:

```
<išvardinamasis_tipas> ::= enum <vardas>
                           {
                            <simbolis> [{, <simbolis>}]
                           }
```

Vardas po raktinio žodžio **enum** apibrėžia naujo tipo vardą. Žemiau apibrėžiame kelis naujus išvardijamuosius tipus:

```
enum Spalvos
{
  Raudona, Žalia, Mėlyna, Juoda, Balta, Geltona, Ruda
};

enum DarboDienos
{
  Pirmadienis, Antradienis, Trečiadienis, Ketvirtadienis,
  Penktadienis
}
```

Tipo paskelbimas

Galima apibrėžti naujus tipus, paskelbiant jų vardus, o vėliau tie vardai naudojami taip pat, kaip ir visų kitų (atominių) tipų vardai. Tai daroma raktinio žodžio **typedef** pagalba:

```
<tipo_paskelbimas> ::= typedef <tipas> <vardas>
```

Keletas pavyzdžių:

```
typedef long TypeTimeOut;

typedef char Stekas[256]
```

II. Klasės

Klasėms aprašyti ODL naudojamas raktinis žodis **interface**. Po šio žodžio rašomas klasės pavadinimas, superklasės(—ių) vardas ir tuomet tarp riestinių skliaustų **{}** surašomi klasės atributai, asociacijos bei metodai:

```
<klasė> ::= interface <klasės_vardas>
                      [: <superklasė> [{, <superklasė>}]]
            {
              <savybė> {;<savybė>}
            };

<klasės_vardas> ::= <vardas>

<superklasė> ::= <vardas>

<savybė> ::= <atributas> |
             <asociacija> |
             <metodas>
```

Atributai

Atributai klasės apibrėžime aprašomi pagal sintaksę

```
<atributas> ::= attribute <tipas> <vardas>
```

Čia pateikiamas pavyzdys klasės, turinčios kelis atributus:

```
interface Tarnautojas : Asmuo
{
  attribute string Vardas;
  attribute string Pavarde;
  attribute struct TypeAdresas
                   { string Miestas;
                     string Gatve;
                     short Namas;
```

```
                        short Butas;
                      } Adresas;
    attribute Asmuo Vaikai[5];
  }
```

Asociacijos

Nors asociacijos yra lyg ir tokie patys atributai, kaip ir visi kiti, tačiau jos aprašomos kitaip. Asociacijos pažymimos raktiniu žodžiu **relationship**. Tolesnis asociacijos aprašymas panašus, kaip ir kitų atributų.

Kartais asociacija turi savo "atvirkštinę" asociaciją kitoje klasėje. Atvirkštinė asociacija nurodoma iškart po asociacijos aprašymo, nurodant raktinį žodį **inverse**, po to "asocijuotos" klasės vardą ir tos klasės atributą—asociaciją, kuris ir yra toji atvirkštinė asociacija.

Užrašykime tai formaliai, o po to — paaiškinkime pavyzdžiu:

```
<asociacija> ::= relationship <tipas> <vardas>
                 [inverse           <klasės_vardas>           ::
<asociacijos_vardas>]

<klasės_vardas> ::= <vardas>

<asociacijos_vardas> ::= <vardas>
```

```
  interface Asmuo {
     relationship Butas gyvena_bute
                              inverse Butas::apgyvendintas;
  };

  interface Butas {
     relationship Asmuo apgyvendintas
                              inverse Asmuo::gyvena_bute;
  };
```

Čia matome, jog atributu — asociacija `gyvena_bute` Asmens objektas susijęs su savo butu, o iš Buto objekto jo šeimininkas pasiekiamas per asociaciją `apgyvendintas`.

Galima modeliuoti ir daugybines asociacijas, naudojantis atributais, kurių tipas yra aibė:

```
  interface Asmuo {
     relationship Set<Asmuo> tevai
                              inverse Asmuo::vaikas;
     relationship List<Asmuo> vaikas
                              inverse Asmuo::tevai;
  };
```

Metodai

Metodų aprašymo sintaksė yra tokia:

```
<metodas> ::= <tipas> <metodo_vardas> ( <argumentas>
                                        {, <argumentas>} )

<metodo_vardas> ::= <vardas>
```

Čia tipas, užrašytas prieš metodo vardą, reiškia metodo rezultato tipą. O metodų argumentai aprašomi pagal tokią sintaksę

```
<argumentas> ::= <argumento_kvalifikatorius> <tipas> <vardas>
```

Čia tipas reiškia argumento tipą, vardas — argumento vardą. O argumento kvalifikatorius yra viena iš trijų reikšmių:

```
<argumento_kvalifikatorius> ::= in | out | inout
```

Kvalifikatorius **in** reiškia, jog argumentas yra perduodamas metodui, **out** reiškia, jog metodas jį grąžina, o **inout** reiškia, jog parametras yra perduodamas metodui ir tikimasi, jog metodas jį turi grąžinti.

Taigi, žinodami tokias taisykles, jau galime aprašyti paprasčiausias klases:

```
interface Asmuo
{
   attribute String Vardas;
   attribute String Pavarde;
   attribute Date GimData;

   void vedybos ( in Asmuo Sutuoktinis );
};

interface Tarnautojas: Asmuo {
   attribute Float Alga;
};
```

Kaip matote, mūsų klasė "Tarnautojas" yra klasės "Asmuo" poklasis ir, be visų paveldėtų klasės "Asmuo" atributų ir metodų (specialiai nurodyti paveldėjimo nereikia), jis turi dar ir algą.

Taigi, tuo baigiame trumpą ODL aprašymą. Čia mes pamatėme, kaip aprašomos klasės (interfeisai), kurių objektai vėliau gali būti saugomi duomenų bazėje. Kaip jau minėjome, tą patį galima padaryti ir modifikuotomis ODL versijomis, kurių sintaksė nelabai skiriasi nuo programavimo kalbos, kuriai yra pritaikyta ta ODL versija. Tačiau standartinio ODL naudojimas padidina juo parašytų tipų ir klasių apibrėžimų pernešamumą.

Objektinė užklausų kalba OQL

Paruošta pagal [Del9495], [Banc97], [Feg97], [Lief97], [OQL99], [BC97], [Cas95]. Tikimasi, jog gerbiamas Skaitytojas bent kažkiek susipažinęs su kita užklausų kalba: SQL.

Šiame skyriuje pristatysime objektinę užklausų kalbą OQL (*Object Query Language*), kuri yra patvirtinta ODMG standarte. Kalba paremta ODMG (tuo pačiu — ir OMG) duomenų modeliu, tačiau, stengiantis perimti tuos pozityvius dalykus, kuriuos atnešė į Duomenų Bazių pasaulį reliacinis duomenų modelis su SQL, daug OQL konceptų buvo perimta iš SQL2 standarto. Žinoma, nemaža dalimi prie tokio panašumo prisidėjo ir faktas, jog SQL yra šiuo metu plačiausiai naudojama ir labiausiai išvystyta užklausų kalba. Tačiau tai jokiu būdu nereiškia, kad OQL yra SQL išplėtimas — ji yra visiškai savarankiška kalba.

Taigi — OQL, visų pirma, yra *deklaratyvi* užklausų kalba. OQL užklausos nenurodo jų vykdymo algoritmo, o nusako tiktai tas savybes, kurias turi tenkinti užklausos rezultatas. Tai leidžia įgyvendinti kalbos nepriklausomumą nuo fizinio duomenų lygio. Tai yra, OQL yra nepriklausoma nuo įvardijimo bei duomenų išliekamumo (*persistence*) strategijos. Taip pat šis faktas leidžia užklausų optimizavimą.

Antra — OQL yra *ortogonali* kalba: kiekvienas jos mechanizmas pritaikomas bet kuriam duomenų modelio konstruktui (žinoma, tas ortogonalumas galioja duomenų tipų sistemos rėmuose; t.y. OQL kalbos mechanizmai gali būti taikomi tik tiems duomenų modelio konstruktams, kuriems šį mechanizmą pritaikyto leidžia jų tipas).

Trečia — OQL yra *apibendrinta* (*general*), kadangi jos mechanizmai taikomi visiems duomenų modelio konstruktams: tiek objektams, tiek struktūrinėms reikšmėms, tiek jų aibėms.

Ketvirta —— OQL kalba yra *funkcionali*. Tai reiškia du dalykus: (i) užklausose galima naudoti metodus ir (ii) užklausas galima įdėti vieną į kitą, pagal pageidavimą, bet kurioje užklausos dalyje.

Penkta — OQL gali būti naudojama tiek interaktyviai (*ad hoc*), tiek programavimo rėžime.

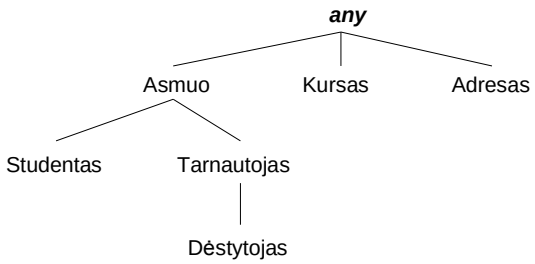
Ir, galų gale, OQL galima praplėsti kitų programavimo kalbų galimybėmis. Tai padaroma per metodų naudojimą ir jau minėtus ODMG standarte susiejimus su kitomis programavimo kalbomis. Todėl egzistuoja netgi tokia teorinė galimybė, kaip siejimo su SQL standartizavimas ir visų SQL galimybių inkorporavimas.

Po tokio bendro OQL pristatymo, pateiksime šią užklausų kalbą smulkiau.

I. Duomenų bazė — pavyzdys

Visų pirma aprašykime duomenų bazę, kuria naudosime kaip pavyzdį OQL pristatymui.

Tarkime, jog mūsų duomenų bazėje yra ši klasių hierarchija:



Paveikslėlis 4 Pavyzdžio DB klasių hierarchija

Kiekviena iš klasių turi šias savybes (atributus bei metodus):

```
      Asmuo: vardas, pavarde, gimimo_data, adresas,
             amzius()
  Studentas: pazymejimo_nr, kursai
Tarnautojas: departamentas, alga()
 Destytojas: laipsnis, kursai
     Kursas: pavadinimas, numeris, destytojai, studentai
    Adresas: namas, butas, gatve, miestas, indeksas
```

Ir, galų gale, tarkime, jog mūsų duomenų bazėje yra šios įvardintos reikšmės ("įėjimo taškai"):

```
    Zmones: Bag <Asmuo>
  Studentai: Set <Studentas>
Tarnautojai: Set <Tarnautojas>
 Destytojai: Set <Destytojas>
     Kursai: List <Kursas>
    Vedejas: Destytojas
```

Pasinaudodami šia DB ir pristatysime užklausų kalbą OQL.

II. Trumpos pastabos apie kalbą

OQL yra funkcionali kalba. Kiekviena užklausa yra funkcija. O tokios funkcijos argumentais gali būti kitos užklausos/funkcijos.

```
Vedejas

Vedejas.kursai
```

```
count (Vedejas.kursai)

count (Vedejas.kursai) > 10
```

Šiame pavyzdyje iš pradžių turime užklausą `Vedejas` , kuri grąžina Vedėjo objektą. Ši užklausa vėliau panaudojama kaip argumentas užklausoje `(...).kursai` ; pastaroji — užklausoje `count(...)` ir t.t.

OQL apibrėžiama aibe užklausų/funkcijų ir taisyklėmis, pagal kurias galima sukonstruoti naujas užklausas, pasinaudojant funkcijų kompozicija bei iteratoriais. Štai iteratoriaus `select` pavyzdys:

```
Select p.pavarde
  from p in Destytojai
 where count(p.kursai) > count(Vedejas.kursai)
```

Užklausų/funkcijų argumentai yra bet kokie objektai ar reikšmės, o rezultatai gali būti:

- bet kuris objektas ar reikšmė, saugoma duomenų bazėje;
- bet kokia naujai sukonstruota reikšmė, tenkinanti duomenų modelio reikalavimus;
- bet koks sukurtas objektas, kurio klasė apibrėžta DB schemoje.

Objektų savybes (atributai ir metodai) pasiekiami naudojantis funkcija "." (arba "—>"). Tačiau realiai atributų naudojimas užklausose yra labiau pageidautinas, nei metodų, nes atributų naudojimas leidžia optimizuoti užklausas (kas yra neįmanoma su metodais; metodai turi savus algoritmus, kurie "nematomi" ODBS; todėl ji negali naudoti nei statistinės, nei kitokios informacijos, leidžiančios surasti gerą užklausos vykdymo planą).

Paprastai užklausų kalbos deklaratyvumas pristatomas kaip priešprieša jos funkcionalumui. OQL atveju turime abi šias savybes kartu. Žinoma — vietoj to, kad rašytume OQL užklausą, galime ją suprogramuoti kokiu nors metodu ir vėliau pasinaudoti ja, panašiau kaip kad rašėme užklausą `Vedejas.kursai`. Tačiau, norint tikrai pasiekti fizinio ir loginio duomenų lygių nepriklausomumą, tokio užklausų programavimo vertėtų vengti.

Dar viena pastaba: OQL neleidžia kitaip keisti duomenų bazėje saugomos informacijos, kaip tiktai naudojantis saugomų objektų metodais. T.y. jame nėra tradicinių SQL sakinių "UPDATE" ar "INSERT". OQL yra *užklausų* kalba.

Faktas, jog OQL tėra tik užklausų kalba ir tai, jog OQL yra visiškai ortogonali, gerokai supaprastina jos sintaksės aprašymą. Palyginimui: SQL3 standartą sudaro apie 1300 puslapių specifikacijos, o OQL specifikacija — tik apie 40 psl. Tačiau SQL aprašo keturis — duomenų, jų saugojimo, įvardijimo bei paieškos — modelius, o OQL — tik paieškos modelį.

III. OQL sintaksė

O dabar — apie OQL sintaksę formaliau.

III.A Duomenų pasiekimas

Visi duomenų bazėje esantys objektai ar reikšmės pasiekiami per *vardus*. Vardai yra tas įėjimo taškas, per kurį paskui galima paimti bet kokią DB saugomą informaciją.

```
Vedejas
```

Tai yra OQL išraiška. Jos rezultatas — klasės `Destytojas` objektas, saugantis Vedėjo duomenis.

```
Tarnautojai
```

Tai — irgi OQL išraiška. Tik šį kartą rezultatas yra klasės `Tarnautojas` objektų aibė.

Prie tokių pat išraiškų galime priskirti ir užklausą `1+1`. SQL šitaip užrašyti užklausos neįmanoma. O kadangi ODMG objektiniame duomenų modelyje visos atominių tipų reikšmės yra (iš anksto apibrėžtos) konstantos, "įvardijamos" savo pačių užrašymu, tai jas galime naudoti lygiai taip pat, kaip ir visus kitus vardus.

Taigi, norint tiesiog pasiimti duomenis iš duomenų bazės, netaikant jokių filtrų ar apribojimų, nereikia rašyti jokių SELECT išraiškų — užtenka tiesiog "išsikviesti" duomenis jų vardu.

III.B Unarinės kelio išraiškos

Kelio išraiška (*path expression*) yra tiesioginio navigavimą duomenyse priemonė, būdinga visoms objektinio programavimo kalboms (primenu, kad alternatyva tiesioginiam navigavimui yra *asociatyvus* navigavimas).

Jau minėjau, kad objektinėje duomenų bazėje išlieka įvardinti objektai (reikšmės) bei visi iš tų objektų pasiekiami objektai ir reikšmės. Taigi — visi objektai, kurie gali būti "pasiekti" yra pasiekiami per kelio išraiškas. Kelio išraiška nurodo tam tikrą "navigacinį" kelią po objekto savybes (atributus bei metodus). Štai kelios kelio išraiškos:

```
Vedejas.vardas;

Vedejas.adresas.namas;
```

Šios užrašytos kelio išraiškos irgi yra *užklausos* OQL kalboje (pirmosios rezultatas yra simbolių eilutė, reiškianti vedėjo vardą, o antrosios — namo numeris, kuriame gyvena vedėjas).

Kelio išraiškos yra "unarinės", nes kiekvienas kelio elementas yra *vienas* objektas ar reikšmė, ir nuo vieno objekto prie kito pereinama iškart ir vienareikšmiškai: kelias yra vienos šakos pavidalo.

Tokias unarines kelio išraiškas galima naudoti ir bet kuriose kitose OQL užklausų išraiškose, pavyzdžiui — SELECT užklausoje (matysime vėliau).

III.C Manipuliavimas reikšmėmis ir objektais

OQL kalboje galima tiek konstruoti reikšmes bei objektus, tiek išgauti juose saugomą informaciją

Informacija išgaunama, kaip jau matėme, per įvardintus objektus (reikšmes), bei per *kelio* išraiškas.

```
Vedejas.alga();

Vedejas.adresas.miestas
```

Jeigu OQL užklausoje kuriame naujus objektus ar reikšmes, tuomet klasių vardais (objektams) ar reikšmių *konstruktoriais*. (jeigu tokie yra). Pavyzdžiui: struktūrinės reikšmės kuriamos naudojantis konstruktoriumi **`struct`**, o štai reikšmei `5` sukurti nereikia jokio konstruktoriaus — užtenka parašyti patį skaičių.

Štai reikšmių ir objektų konstravimo pavyzdžiai (nepamirškite, jog tai — savarankiškos OQL užklausos!):

```
struct ( gatve: "Lokio g–vė",
         miestas: "Meškai" );

Adresas ( namas: 5,
          butas: 14,
          gatvė: "Partizanų",
        miestas: "Joniškėlis",
       indeksas: 223322 )
```

Konstruoti galime ir aibes. Aibių konstruktoriai yra keturi: `set`, `bag`, `array` ir `list`. Pirmieji trys naudojami vienodai:

```
set (1, 3, 5, 7, 9);

array (1, 3, 5, 7, 9)
```

O sąrašus galima konstruoti dvejopai:

```
list (1, 2, 3, 4, 5, 6, 7, 8, 9);

list (1..9)
```

Žinoma, visus tris aukščiau minėtus manipuliavimo reikšmėmis ir objektais mechanizmus galima kombinuoti tarpusavyje:

```
struct ( pavarde: Vedejas.pavarde,
         adresas: Adresas ( namas: 5,
                            butas: 14,
                            gatvė: "Partizanų",
                          miestas: "Joniškėlis",
                         indeksas: 223322 )
       )
```

III.D SELECT užklausos

OQL leidžia užrašyti standartines SELECT užklausas. Jų užrašymo sintaksė mažai kuo skiriasi (o ODMG—2.0 standarte priimama ir lygiai tokia pat sintaksė) nuo SQL. Pavyzdžiui:

```
Select s
  from s in Studentai
 where s.vardas = "Sigitas"

Select struct ( vrd: s.vardas,
                amz: s.amzius() )
  from s in Studentai

Select k.destytojas.adresas.miestas
  from k in Kursai
 where k.pavadinimas = "Objektinės duomenų bazės"

Select s.vardas
  from s in Studentai,
       d in Destytojai
 where s.vardas = d.vardas
```

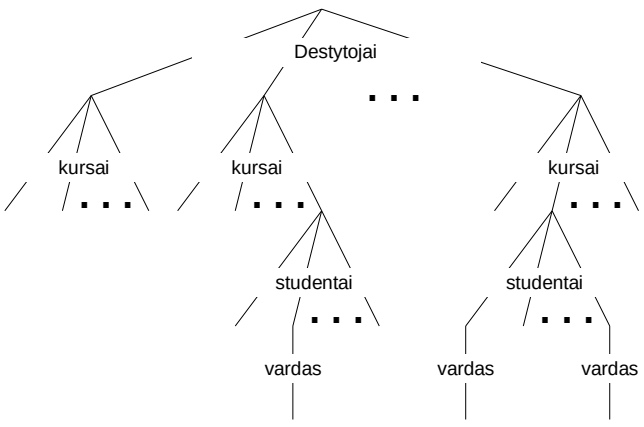
Paskutinėje užklausoje mes užrašėme standartinę (tradicinę) reliacinę *join* operaciją. Kaip pamatysime vėliau, OQL leidžia išreikšti ir šiek tiek kitokio tipo apjungimus (*join*).

III.E N—narinės kelio išraiškos

OQL išraiškose galima naudoti ir n—narines kelio išraiškas. Tai reiškia, kad tam tikrose kelio dalyse, jis gali "išsišakoti" ir tuomet kelias turi medžio pavidalą. Taip būna tuomet, kai vienas iš objekto atributų yra aibė ir tada užklausoje reikia peržiūrėti visus aibės elementus. Tai lemia kelio išsišakojimą. Tokios kelio išraiškos panaudojimo pavyzdys:

```
select d.pavarde
  from d in Destytojai,
       k in d.kursai,
       s in k.studentai
 where s.vardas = "Sigitas"
```

Čia mes gauname tokį užklausos medį, iliustruojantį n—narinę kelio išraišką:



Paveikslėlis 5 N—narinę kelio išraišką naudojančios užklausos medis

III.F Sąjunga pagal pasiekiamumą

Dar viena ypatybė OQL kalboje (kylanti vėlgi iš duomenų modelio), yra sąjunga pagal pasiekiamumą (*pointer join*).

Jau minėjome, jog OQL leidžia išreikšti tradicinę reliacinę sąjungą, kai sąjunga išreiškiama per dviejų atributų santykį. Čia gi sąjunga apibrėžiama pagal pasiekiamumą arba, tiksliau, pagal galimybę sudaryti kelio išraišką tarp dviejų norimų elementų. Pavyzdžiui:

```
select s.pavarde, d.pavarde
  from d in Destytojai,
       k in d.kursai,
       s in k.studentai
```

Šioje užklausoje studentas su dėstytoju susiejami ne per kokį nors jų bendrų atributų sulyginimą, bet per kelio išraišką nuo dėstytojo objekto iki jo studento.

III.G Metodų naudojimas

Dar vienas skirtumas nuo SQL glūdi galimybėje naudoti metodus. Bet kurioje užklausos vietoje galima iškviesti objekto metodą ir pasinaudoti jo rezultatais. Pavyzdžiui, galima užrašyti tokias užklausas:

```
Vedejas.alga();

select a.vardas
  from a in Asmenys
 where a.amzius() > 21
```

III.H Manipuliavimas kolekcijomis (aibėmis)

Dažniausiai užklausos daromos iš kokių nors mus dominančių objektų ar reikšmių aibių. Todėl labai svarbu, kokias galimybes manipuliavimui su aibėmis pateikia užklausų kalba. Imkime, pavyzdžiu, tokią OQL užklausą:

```
select struct ( pavarde: d.pavarde,
                DBkursai: select k
                            from k in d.kursai
                           where k.pavadinimas like "*DB*"
              )
  from d in Destytojai
 where count(select k
               from k in d.kursai
              where k.pavadinimas like "*DB*") > 0
```

Joje renkama (SELECT sakiniai) iš aibių, vienas iš konstruojamos struktūros atributų yra aibė (`DBkursai`), ir, galų gale, aibei pritaikoma jos kardinalumo funkcija **`count`**.

Taigi, OQL leidžia atlikti su aibėmis šiuos veiksmus:

Rūšiavimas

Rūšiavimas OQL užklausoje išreiškiamas raktinių žodžių **order by** pagalba. Pavyzdžiui:

```
select s
  from s in Studentai
order by s.amzius() asc,
         s.vardas
```

Pavyzdžiui, jau minėtoje ODBS O_2 naudojama ir kita sintaksė, netgi kitas raktinis žodis: ***sort****. Pateiksime ją čia be didesnių paaiškinimų, nes intuityviai ši sintaksė irgi yra aiški:*

```
sort stud in Studentai
  by stud.amzius() asc, stud.vardas
```

Grupavimas

Aibės grupavimas daromas pagal tam tikrus joje esančių objektų požymius (tai gali būti tiek objektų atributų reikšmės, tiek kokios nors funkcijos nuo objektų atributų). Po aibės sugrupavimo, rezultatas yra kita aibė, kurios elementai — *struktūrinės reikšmės* . Jas sudaro grupavimo požymiai (t.y. jų reikšmės), bei vienas standartinis atributas **partition**. Šis atributas yra ne kas kita, kaip aibė sugrupuotų objektų (ar reikšmių), kurie atitinka toje pačioje struktūroje esančius grupavimo požymius.

Kad būtų aiškiau, peržiūrėkime tokį grupavimo pavyzdį:

```
select *
  from t in Tarnautojai
group by      daug: t.alga() > 10000,
       vidutiniskai: t.alga() > 2000 and t.alga() < 10000,
              mazai: t.alga() < 2000
```

Šios užklausos rezultatas yra tokia struktūrų aibė:

```
{
  [daug: false, vidutiniskai: false, mazai: true,
                                    partition: { [t:t11],[t:t12],...}],
  [daug: false, vidutiniskai: true, mazai: false,
                                    partition: { [t:t21],[t:t22],...}],
  [daug: true, vidutiniskai: false, mazai: false,
                                    partition: { [t:t31],[t:t32],...}],
}
```

Grupavimo sakiniuose galima naudoti ir apribojimo sakinį **having**:

```
select *
  from t in Tarnautojai
group by      daug: t.alga() > 10000,
       vidutiniskai: t.alga() > 2000 and t.alga() < 10000,
              mazai: t.alga() < 2000
having avg( select t.alga() from t in partition ) > MGL
```

T.y. šioje užklausoje mes norime gauti tik tas grupes (domimės tik tais tarnautojais), kurių darbo užmokesčio vidurkis didesnis už Minimalų praGyvenimo Lygį.

Vėlgi paminėsime O₂ naudojamos sintaksės skirtumą. Taigi, mūsų tarnautojų sugrupavimas būtų užrašomas taip:

```
group t in Tarnautojai
   by (         daug: t.alga() > 10000,
        vidutiniskai: t.alga() > 2000 and t.alga() < 10000,
               mazai: t.alga() < 2000 )
```

Agreguojančios funkcijos

Aibėms galima taikyti agreguojančias funkcijas **sum, max**, **min**, **count** ir **avg.** Pavyzdžiui:

```
max ( select t.alga()
        from t in Tarnautojai )
```

Konversijos

Galima konvertuoti aibes tiek į objektus, tiek į kito tipo aibes (aibę į multiaibę ir pan.).

```
element ( select k
            from k in Kursai
           where k.pavadinimas like "*DB*" and
                 k.numeris = 101 );

first ( element( select k
                   from k in Kursai
                  where k.pavadinimas like "*DB*" and
                        k.numeris = 101
                ).destytojai
      );

listtoset (Vedejas.kursai);
```

Panašiai naudojamos ir funkcijos `last` (grąžina paskutinį sąrašo ar masyvo elementą) bei `distinct` (eliminuoja pasikartojančius multiaibės elementus).

Objektus irgi galima "paversti" aibėmis:

```
set (Vedejas);

list (1, 2, 3)
```

Įdomi yra operacija, leidžianti iš aibių aibės gauti vieną ("plokščią") aibę:

```
flatten ( select d.kursai
            from d in Destytojai )
```

Čia kiekvienas Dėstytojas turi aibę dėstomų Kursų. Užklausa (SELECT sakinys), surenka visų Dėstytojų atributus "kursai". Taigi, rezultatas yra aibių aibė (kadangi `d.kursai` yra aibės tipo). O operatorius **`flatten`** "išgliaudo" visas

SELECT rezultato aibėje esančias aibes ir grąžina vientisą aibę, kurios elementai yra kursai, pasitaikę bent vienoje iš `d.kursai`.

*Operatorius **`flatten`** apdoroja tik vieną aibių "įdėjimo" lygį.*

Aibių operacijos

OQL galimos ir operacijos su aibėmis: **`union`**, **`intersect`**, **`except`** (kurios atitinka aibių sąjungą, sankirtą bei atimtį) bei konkatenacija '||':

```
Asmenys except Studentai;

Studentai union set(Vedejas);

Kursai || list ( Kursas( pavadinimas: "ODB",
                         numeris: 13,
                         destytojai: set(Vedejas),
                         studentai: Studentai )
               )
```

Kvantoriai

OQL kalboje yra du aibių kvantoriai: visuotinis ir egzistencijos:

```
   select s.vardas
     from s in Studentai
    where for all k in s.kursai : k.pavadinimas like
"*DB*";

   select s.vardas
     from s in Studentai
    where exists k in s.kursai :
            (exists d in k.destytojai :
               d.pavarde = "Janeliūnas")
```

III.I Sudėtinių išraiškų sudarymas

Kadangi OQL užklausa yra funkcija su tam tikru rezultatu, tai bet kokios formos užklausą galima laisvai naudoti bet kurioje kitos užklausos vietoje. Štai pavyzdys:

```
Select struct ( studentas: s.pavarde,
                destytojas: select d
                            from k in s.kursai,
                                 d in k.destytojai )
  from s in Studentai
```

Jeigu gerai įsižiūrėsime ir atsiminsime, kas yra užklausa OQL, tai pamatysime, kad yra dvi užklausos: `s.kursai` ir `k.destytojai` , kurios dalyvauja trečioje, vidinėje SELECT užklausoje. Šioji gi dalyvauja užklausoje, konstruojančioje naują struktūrinę reikšmę. Ir pagaliau šis konstravimas dalyvauja išorinėje SELECT užklausoje.

III.J Įvardintosios užklausos

Užklausoms, kaip ir objektams ar reikšmėms, galima suteikti vardus. Tiesa, reikia atminti, jog užklausoms vardai suteikiami tik *ad hoc* (t.y. interaktyviame) darbo rėžime, o programavimo (programų veikimo) metu to daryti negalima.

Taigi, užklausoms vardai suteikiami naudojantis raktiniu žodžiu **`define`**:

```
Define Sigitai as
  select distinct s
    from s in Studentai
   where s.vardas = "Sigitas"
```

Vėliau įvardintą užklausą gaila naudoti kaip ir bet kurią kitą užklausą ten, kur tai leidžia tipizavimo taisyklės. Pavyzdžiui:

```
Select ss.adresas.miestas
  from ss in Sigitai
```

Kai įvardintoji užklausa nebereikalinga, ją galima panaikinti:

```
Undefine Sigitai
```

III.K Trumpa OQL sintaksės santrauka

Šio poskyrio pabaigai pateiksime trumpą OQL išraiškų sintaksę.

OQL išraiška yra:

1) **c** bet kokia konstanta;
2) **v** vardai (duomenų bazės saugojimo šaknys);
3) **x** iteratorių kintamieji;
4) konstruktoriai `struct`, `set`, `list`, `bag`, `array` :

```
struct (name₁ : expr₁ , ... , nameₙ : exprₙ)
   set (expr₁ , ... , expr ₙ)
  list (expr₁ , ... , expr ₙ)
  list (expr₁ .. expr₂)
   bag (expr₁ , ... , expr ₙ)
 array (expr₁ , ... , expr ₙ)
```

5) operacijos:

skaitinės:	`+ , – , * , /, mod, abs(`**`expr`**`)`
loginės:	`not, and, or`
struktūrinės:	`.` (atributo ekstraktorius)
aibių:	`except, union, intersect, flatten, element, distinct, count, sum, avg, min, max, count`
Sąrašų	`||, first, last, listtoset` ir aibių

operacijos

multiaibių: `distinct` ir aibių operacijos

objektų: `.` (pranešimo objektui pasiuntimas)

6) predikatai:

```
expr1 θ expr2    θ ∈ { = , !=, <, >, <=, >= };
for all x in col: boolexpr[4](x);
exists x in col: boolexpr(x)
```

7) išrinkimas

```
select [ distinct ] expr1(x1 , ... , xn),
                          . . .
                    exprm(x1 , ... , xn)
  from x1 in col1 ,
            . . .
       xn in coln
 [ where boolexpr (x1 , ... , xn)]
 [ group by name1 : expr1 , ... , namen : exprm ]
 [ having boolexpr(name1 , ... , namen) ]
 [ order by expr1 [asc, desc], ... , exprq  [asc, desc]]
```

8) įvardinimas

```
define name as expr
undefine name
```

9) komentarai

```
// vienos eilutės komentaras
/* blokinis komentaras,
   galintis tęstis per kelias eilutes */
```

IV. OQL tipizavimas

Kaip ir visos programavimo kalbos, OQL irgi turi savo sintaksę, tipizavimą bei semantiką. Sintaksę mes jau pristatėme ir dabar laikas pereiti tipizavimo.

Šiame skyrelyje pristatysime OQL išraiškų tipizavimą. Tiesa, pristatymas nebus absoliučiai pilnas — nepateiksime tipizavimo taisyklių kiekvienai kalbos išraiškai. Tačiau toks išsamumas ir nėra mūsų kurso tikslas. Svarbiausia — suprasti tipizavimo mechanizmą ir, reikalui esant, pritaikyti šį supratimą tolesniems ieškojimams.

[4] `boolexpr` yra išraiškos, kurių reikšmė *boolean* tipo. Tai yra predikatai bei jų kombinacijos (junginiai), sudarytos naudojantis jau minėtomis loginėmis operacijomis.

IV.A Vardų išsprendimas

Prieš pradedant nagrinėti išraiškų tipizavimą, visų pirma reikia pakalbėti apie vardų išsprendimą. Nes nuo to, kaip bus išspręstas išraiškoje esantis vardas, priklauso ir tai, kaip tas vardas bus tipizuojamas.

OQL išraiškų sintaksinis analizatorius atpažįsta:

- atominių tipų konstantas (skaičius, simbolių eilutes ir pan.),
- tarnybinius, kitaip dar vadinamus raktiniais, žodžius (`select`, `from`, ...),
- "vardus" (visi likę išraiškos elementai).

Deja, tie patys vardai gali būti naudojami skirtingiems kalbos konceptams įvardinti. Pavyzdžiui:

```
class Asmuo
   type tuple ( . . . );

class Automobilis
   type tuple (
                  asmuo: Asmuo;
                    . . .
               );

name Asmuo : . . .

define asmuo as . . .

select . . .
  from asmuo in Asmuo,
       auto in ManoAutomobiliai
 where auto.asmuo = . . .
```

Sintaksinis analizatorius pagal vardo vietą (poziciją) išraiškoje gali nustatyti kai kurių vardų "reikšmę". Pirmame pavyzdžio sakinyje visiškai aišku, jog vardas "Asmuo" yra klasės vardas. O štai antroje išraiškoje "asmuo" yra iš pradžių atributo vardas, paskui — tipo (klasės) pavadinimas.

Tačiau vis vien lieka tam tikras kiekis "neišspręstų" vardų. Pastariesiems taikoma tokia išsprendimo prioritetų tvarka:

1. kintamasis;
2. savybė (atributas arba metodas, į kurį *galima* kreiptis);
3. užklausos vardas;
4. schemos vardas.

Taigi, kai jau aptarėme vardų išsprendimą, galime pereiti ir prie užklausų tipizavimo.

IV.B Dinaminis ar statinis tipizavimas?

Bendru atveju galimi du išraiškų tipizavimo variantai:

- **Statinis**. Jis daromas prieš išraiškos vykdymą, pagal jos sintaksinį medį. Šitoks tipizavimo būdas yra griežtas, nes, turint klasės/poklasio

ryšį, dažnai neįmanoma pasakyti tikrosios objekto klasės, kol tas objektas nepatenka į išraišką jos vykdymo metu. Tačiau toks tipizavimas yra efektyvesnis, nes visos išraiškos būna tipizuotos dar prieš vykdymą ir to nebereikia daryti užklausos vykdymo metu. Be to, taip sumažinamas galimų programos klaidų skaičius.

- **Dinaminis**. Išraiškos yra tipizuojamos jų vykdymo metu. Todėl galime pamiršti viliojančią išankstinio programų teisingumo tikrinimo idėją — neįmanoma tiksliai iš anksto nustatyti programos rezultato, kol nėra jos išankstinio tipizavimo. Be to, išraiškų vykdymas yra mažiau efektyvus. Užtat toks tipizavimas yra daug lankstesnis ir "galingesnis".

Pavyzdžiui:

```
select a.alga()
  from a in Asmenys
 where a.statusas = "dirbantis"
```

Statinis tipizavimas šioje užklausoje išduotų klaidą, nes Asmuo nebūtinai turi Algą. Galima būtų daryti tipų konversiją ir rašyti `select (Tarnautojas)a.alga()`, tačiau tai nėra "gražus" sprendimas. Be to, objektas `a` gali būti ir Dėstytojas. Tokiu atveju tipų konversija nebūtų visiškai teisinga — būtų prarandama informacija.

O štai esant dinaminiam tipizavimui, šią užklausą galima būtų traktuoti kaip teisingą ir kompiliavimo metu neturėtume jokių bėdų. Tačiau, jeigu pereisime prie užklausos vykdymo — bėdų gali ir atsirasti. Kas bus, jeigu Statusą "dirbantis" turės objektas, kuris neturi metodo "alga" ?

Pasvarstymų pabaigai norėtume pranešti, jog, po tam tikro išankstinio (statinio) tipizavimo, OQL tipizuojama dinamiškai. Jeigu sugrįšime prie mūsų pavyzdžio, tai jame pateikta užklausa OQL tipizavimo atžvilgiu yra klaidinga. Visi objektai `a` aibėje `Asmenys` yra klasės "Asmuo". Todėl, nepriklausomai nuo atributo `statusas` reikšmės, jiems negalima taikyti metodo `alga()` . Jeigu padarytume minėtą tipų konversiją `(Tarnautojas)a` , tai tipizavimo atžvilgiu užklausą ištaisytume. Tačiau dabar visą atsakomybę dėl galimų tipų klaidų prisiima programuotojas.

Taigi, einame toliau ir dabar kalbėsime apie OQL tipizavimo taisykles.

IV.C OQL tipizavimo taisyklės

OQL užklausos tipizavimas atsako į klausimą "ar užklausa gerai (teisingai, korektiškai) tipizuota?"

Jau sakėme, jog OQL yra funkcionali kalba: užklausos yra funkcijos, kurių argumentais gali būti kitos užklausos. Iš pradžių užklausos sudaromos iš *konstantinių* užklausų (atominių reikšmių ar Duomenų Bazėje esančių vardų), kurių tipai yra žinomi. Tuomet šios užklausos yra tipizuojamos naudojantis *tipizavimo taisyklėmis*, kurios leidžia nustatyti užklausos (funkcijos) tipą pagal jos argumentų tipus. Jeigu naujai sudarytoji užklausa bus naudojama kitose užklausose — jos tipas jau žinomas ir ji galės būti panaudojama tų naujųjų užklausų tipizavime.

Pateikime keletą tipizavimo taisyklių.

Kai kurios vieno argumento funkcijos

- Konversija (*casting*)

$$\frac{\mathtt{q} :: c' \,,\; c \le c' \text{ arba } c \ge c'}{\mathtt{(c)q} :: c}$$

Viršutinėje taisyklės dalyje rašomi užklausų/argumentų tipai, o apatinėje — mūsų tipizuojamos užklausos tipas. Todėl taisyklę galime skaityti "jeigu ... (viršutinė taisyklės dalis), tuomet turi būti... (apatinė dalis)". Jei išraiška netenkina tipizavimo taisyklės, vadinasi ji yra blogai tipizuota.

Tuomet, ši mūsų taisyklė sako, kad, jeigu reiškinys `q` yra tipo *c'* , o *c* yra bet koks kitas tipas, kažkaip susietas su *c'* tipo/potipio sąryšiu, tai po konversijos `q(c)` reiškinys bus tipo *c*.

- **`Element`**

$$\frac{\mathtt{q} :: \{t\}}{\mathtt{element(q)} :: t}$$

Čia nurodoma, kad, jeigu reiškinys `q` yra aibė iš tipo *t* elementų, tai reiškinys `element(q)` yra tipo *t*.

Tipizavimo taisyklės tarnauja tik tipų skaičiavimui. Jos jokiu būdu nenusako užklausos vykdymo algoritmo ir dar negarantuoja, jog užklausa bus įvykdyta teisingai. Todėl ši taisyklė visiškai netvirtina, kad aibė `q` turės vienintelį elementą, kad ją būtų galima konvertuoti į tą elementą. Ji nusako tik tipų "santykį" užklausoje.

Kreipimasis į savybes

- Ši taisyklė nusako struktūrinių reikšmių tipizavimą.

$$\frac{\mathtt{q} :: t \,,\; t \le [\mathtt{a} : t']}{\mathtt{q.a} :: t'}$$

Čia sakoma, jog, jeigu `q` yra struktūrinė reikšmė, kuri turi tipo *t'* atributą `a` , tai galima į tą atributą kreiptis, o rezultatas bus tipo *t'* .

- O šioji taisyklė skirta objektų tipizavimui.

$$\frac{\mathtt{q} :: c \,,\; type(c) \le [\mathtt{a} : t]}{\mathtt{q.a} :: t}$$

Ši taisyklė panaši į prieš tai buvusią. Tik čia imamas ne reiškinio `q` tipas (t.y. klasės vardas), o su klase *c* susietas tipas *type(c)* .

- Ši taisyklė — metodu iškvietimui.

$$\frac{\mathtt{q} :: c \,,\quad \mathtt{m} : (c, t_1, \ldots, t_n) \to t \,,\quad \forall i = \overline{1,n} \quad \mathtt{q}_i :: t'_i \,,\; t'_i \le t_i}{\mathtt{q.m(q_1,\ldots,q_n)} :: t}$$

Sąlyga t'i ≤ ti norima pasakyti, jog metodui perduodami argumentai gali būti ir "mažesnio" tipo, negu nurodyti jo signatūroje.

Keletas dviejų argumentų funkcijų

- Viena taisyklė sudėčiai

$$\frac{\mathtt{q}_1 :: t_1 \ , \mathtt{q}_2 :: t_2 \ , \{real\} \subseteq \{t_1\} \cup \{t_2\} \subseteq \{int, real\}}{\mathtt{q}_1 + \mathtt{q}_2 :: real}$$

  Šia taisykle teigiama, jog, jeigu bent vienas iš sudėties argumentų yra realu skaičius, tai suma — irgi realus skaičius.

- Antra taisyklė sudėčiai

$$\frac{\mathtt{q}_1 :: int \ , \mathtt{q}_2 :: int}{\mathtt{q}_1 + \mathtt{q}_2 :: int}$$

  O štai jeigu abu argumentai yra sveiki skaičiai — suma irgi yra sveikas skaičius.

- Viena iš taisyklių sąjungai

$$\frac{\mathtt{q}_1 :: \{t_1\} \ , \mathtt{q}_2 :: \{t_2\} \ , t_1 \le t_2}{\mathtt{q}_1 + \mathtt{q}_2 :: \{t_2\}}$$

  Sujungiant dvi aibes (sąjunga daroma operatoriaus + pagalba), gautosios aibės elementai bus didesniojo tipo.

Kelios kintamo argumentų skaičiaus funkcijų

- Struktūrinės reikšmės konstravimas

$$\frac{\forall i = \overline{1,n} \quad \mathtt{q}_i :: t_i}{\mathtt{struct}(\mathtt{a}_1 : \mathtt{q}_1, \ldots, \mathtt{a}_n : \mathtt{q}_n) :: [\mathtt{a}_1 : t_1, \ldots, \mathtt{a}_n : t_n]}$$

  Taisyklė nurodo, kokio tipo struktūrą gausime, konstruodami ją iš atskirų elementų $\mathtt{q}_i$.

- Aibės konstravimas

$$\frac{\forall i = \overline{1,n} \quad \mathtt{q}_i :: t_i \ , t = \sup(t_1, \ldots, t_n)}{\mathtt{set}(\mathtt{q}_1, \ldots, \mathtt{q}_n) :: \{t\}}$$

  O štai čia, panašiai, kaip aibių sąjungos tipizavimo taisyklėje, konstruojant aibę iš kelių elementų, gautos aibės elementų tipas yra pats didžiausias iš konstravimui panaudotų elementų tipų.

Iteratoriai

- `select–from–where` iteratorius. Aprašyme naudosime tokius pažymėjimus: `col` reiškia kolekciją (t.y. tipą *set, bag, list* arba *array*); `[]` reiškia fakultatyvinį apskliaustojo reiškinio pobūdį ir `{{ }}` reiškia *bag* tipą.
  Taip pat reikia priminti, jog iterartoriaus argumentai yra *funkcijos*, o ne *funkcijų rezultatai.*

$$\frac{\mathtt{q}_1 :: col(t_1),\ \mathtt{q}_2 :: [t_1] \to col(t_2),\ \ldots,\ \mathtt{q}_n :: [t_1,\ldots,t_{n-1}] \to col(t_n),\quad \mathtt{p} :: [t_1,\ldots,t_n] \to bool,\quad \mathtt{q} :: [t_1,\ldots,t_n] \to t}{\mathtt{x}_1 :: t_1,\ \ldots,\ \mathtt{x}_n :: t_n,\ \left(\begin{array}{l}\mathtt{select\ q[x_1,\ldots,x_n]}\\ \mathtt{\ \ from\ x_1\ in\ q_1,\ x_2\ in\ q_2[x_1],\ \ldots,x_n\ in\ q_n[x_1,\ldots,x_{n-1}]}\\ \mathtt{where\ p[x_1,\ldots,x_n]}\end{array}\right) :: \{\{t\}\}}$$

Jeigu $\mathtt{q}_i$ yra kolekcijos iš tipo t_i elementų (arba — funkcijos, kurios grąžina tokias kolekcijas), predikatas p grąžina loginę reikšmę, o funkcija q — grąžina ("gamina") tipo t reikšmes, tuomet visi kintamieji užklausoje $\mathtt{x}_i$ bus tipo t_i, o mūsų užklausa bus multiaibė iš tipo t elementų.

- `select–from–where–groupby` iteratorius.

  Šią ir dar kitą taisyklę pateiksime be didesnių komentarų, nes jų prasmė panaši į prieš tai buvusią taisyklę.

$$\frac{\begin{array}{l}\mathtt{q}_1 :: col(t_1),\ \mathtt{q}_2 :: [t_1] \to col(t_2),\ \ldots,\ \mathtt{q}_n :: [t_1,\ldots,t_{n-1}] \to col(t_n),\\ \mathtt{p} :: [t_1,\ldots,t_n] \to bool\\ \mathtt{q'}_1 :: [t_1,\ldots,t_n] \to t'_1,\ \ldots,\ \mathtt{q'}_m :: [t_1,\ldots,t_n,t'_1,\ldots,t'_{m-1}] \to t'_m\\ \mathtt{p'} :: [t'_1,\ldots,t'_m,\{[\mathtt{x}_1 : t_1,\ldots,\mathtt{x}_n : t_n]\}] \to bool\\ \mathtt{q} :: [t'_1,\ldots,t'_m,\{[\mathtt{x}_1 : t_1,\ldots,\mathtt{x}_n : t_n]\}] \to t\end{array}}{\begin{array}{l}\mathtt{x}_1 :: t_1,\ \ldots,\ \mathtt{x}_n :: t_n,\ \mathtt{a}_1 :: t'_1,\ \ldots,\ \mathtt{a}_m :: t'_m\\ \mathtt{partition} :: \{[\mathtt{x}_1 : t_1,\ldots,\mathtt{x}_n : t_n]\}\\ \left(\begin{array}{l}\mathtt{select\ q[x_1,\ldots,x_n,partition]}\\ \mathtt{\ \ from\ x_1\ in\ q_1,\ x_2\ in\ q_2[x_1],\ \ldots,x_n\ in\ q_n[x_1,\ldots,x_{n-1}]}\\ \mathtt{where\ p[x_1,\ldots,x_n]}\\ \mathtt{group\ by\ a_1 : q'_1[x_1,\ldots,x_n]\ ,\ldots,a_m : q'_m[x_1,\ldots,x_n,a_1,\ldots,a_{m-1}]}\\ \mathtt{having\ p'[a_1,\ldots,a_m,partition]}\end{array}\right) :: \{\{t\}\}\end{array}}$$

- `select–from–where–orderby` iteratorius. Čia pažymėjimas <> reiškia sutvarkytą sąrašą (*list*).

$$\frac{\mathtt{q}_1 :: col(t_1),\ \mathtt{q}_2 :: [t_1] \to col(t_2),\ \ldots,\ \mathtt{q}_n :: [t_1,\ldots,t_{n-1}] \to col(t_n),\quad \mathtt{p} :: [t_1,\ldots,t_n] \to bool,\quad \mathtt{q} :: [t_1,\ldots,t_n] \to t,\quad \forall i = \overline{1,m}\ \ \mathtt{q}_i :: [t_1,\ldots,t_n] \to bool + real + int + string + Object}{\mathtt{x}_1 :: t_1,\ \ldots,\ \mathtt{x}_n :: t_n,\ \left(\begin{array}{l}\mathtt{select\ q[x_1,\ldots,x_n]}\\ \mathtt{\ \ from\ x_1\ in\ q_1,\ x_2\ in\ q_2[x_1],\ \ldots,x_n\ in\ q_n[x_1,\ldots,x_{n-1}]}\\ \mathtt{where\ p[x_1,\ldots,x_n]}\\ \mathtt{order\ by\ q'_1[x_1,\ldots,x_n],\ \ldots,\ q'_m[x_1,\ldots,x_n]}\end{array}\right) :: <t>}$$

- Kvantoriai

$$\frac{\mathtt{q} :: col(t)\ ,\ \mathtt{p} :: [t] \to bool}{\mathtt{x} :: t\ ,(\mathtt{for\ all/exists\ x\ in\ q : p[x]}) :: bool}$$

Čia teigiama, jog kvantoriaus, pritaikančio kolekcijai `q` sąlygą (predikatą) `p`, reikšmė yra tipo *bool*.

Dabar, kai turite šias kelias tipizavimo taisykles, pabandykite nustatyti, ar teisingas žemiau pateiktos užklausos tipizavimas:

```
select struct ( dept: d,
                vidurkis: avg (select p.t.alga()
                                  from p in partition
                               )
              )
  from t in Tarnautojai
group by d: t.departamentas
having avg ( select p.t.alga()
               from p in partition ) < 1000
```

Atsakymą galite pažiūrėti išnašoje[5].

IV.D Tipizavimo klaidos

Dabar trumpai apžvelgsime pasitaikančias tipų klaidas vykdant išraiškas, kurios lyg ir buvo gerai tipizuotos kompiliavimo metu. Klasikinėse programavimo kalbose "tipo klaida" vadinama tokia klaida, kai su duomenimis bandomos atlikti operacijos, neleistinos tų duomenų tipui. Objektiniame pasaulyje prie tų klaidų dar prijungiamas ir bandymas pasiųsti objektui pranešimą, į kurį jis negali "atsakyti".

Reikia pažymėti, kad čia aprašomos klaidos pasitaiko ne vien OQL užklausose, ar ODB saugomų objektų metoduose — tos klaidos vienu ar kitu aspektu galimos ir objektinėse programavimo kalbose[6]. Ir šių problemų sprendimas neapsiriboja objektinių duomenų bazių sritimi. Tai yra globalus objektinių programavimo kalbų tipizavimo uždavinys. Bet, kadangi jis liečia ir mūsų kurso sritį, nepraleisime jo pro šalį.

Iš pradžių — paprastesnės tipų klaidos.

Vykdymo metu atsirandančios klaidos

Šiame skyrelyje šiek tiek pakalbėsime apie vykdymo (*runtime*) klaidas, kurios gali įvykti net esant korektiškam užklausų tipizavimui. Tai:

[5] Taip, užklausa yra teisinga tipizavimo atžvilgiu. Elementai *t* yra klasės *Tarnautojas*. Po sugrupavimo iteratorius grąžina struktūras su dviem atributais: *d* ir *partition* . Todėl vėlesnis *partition* dalyvavimas subužklausose visai "teisėtas". Be to, *partition* tipas yra *{[t: Tarnautojas]}*. Vadinasi, subužklausos į *partition* yra teisingos. Taigi — turime teisingą (tipizavimo atžvilgiu) užklausą.

[6] Šioje vietoje prašosi mažytė pastaba: objektų metodai yra rašomi kuria nors programavimo kalba. Todėl tipų klaidos metoduose atsiranda tiek, kiek jos galimos naudotoje programavimo kalboje...

1. operacijos **`min`**, **`max`**, **`avg`**, **`sum`**, ir pan. su tuščia aibe;
2. operacija **`element`** su aibe, turinčia ne vieną elementą;
3. dalyba, kai daliklis lygus nuliui;
4. kreipimosi į sąrašus ar simbolių eilutes operacija, kai nurodytas neteisingas kreipimosi indeksas;
5. tipų konversija, kai konvertuotas elementas pasirodo nesąs to tipo, į kurį buvo konvertuotas, potipis;
6. kreipimasis į `nil` objekto savybes.

Plačiau verta pakalbėti apie paskutinę klaidą, nes ji yra dažnai pasitaikanti ir šios problemos sprendimas įneša tam tikrus pataisymus į OQL.

Bet kurioje užklausoje, kuri kreipiasi į objektus, galime turėti tokią situaciją, kai objektas yra `nil`. Pavyzdžiui:

```
select a
  from a in Asmenys
 where a.vardas = "Sigitas"
```

Niekas negali garantuoti, jog aibėje "Asmenys" nėra objekto `nil` ir kad kreipimasis `a.vardas` bus korektiškas. Žinoma, galima būtų bandyti rašyti taip:

```
select a
  from a in Asmenys
 where a != nil and a.vardas = "Sigitas"
```

Bet, visų pirma, rašyti šitaip visas užklausas būtų daugiau negu kad nepatogu. O antra — toks rašymas vis vien nepataiso padėties. Neužmirškime, jog OQL yra deklaratyvi kalba, t.y. nenusakanti užklausos vykdymo algoritmo. Todėl užklausos sąlygų užrašymo tvarka negarantuoja jų vykdymo tvarkos. Ir jeigu ODBS optimizatorius "nuspręs" visų pirma patikrinti, koks yra Asmens vardas, o tik po to — ar Asmuo yra `nil`, tai mes vis vien neišvengsime klaidos užklausos vykdymo metu.

Norint išspręsti šitą "`nil` problemą", buvo įvesta nauja, neapibrėžta reikšmė: `null`. Ji neegzistuoja ODMG duomenų modelio standarte ir ji yra vidinė OQL reikšmė. Dabar galime apibrėžti:

1. jeigu operandas yra `nil` arba `null`, tai kreipimasis į jo atributą grąžina `null`;
2. visos loginės operacijos gražina `false`, jeigu bent vienas iš jų operandų yra `null`;
3. apibrėžiama funkcija `IsNull`, kuri grąžina `true`, jeigu argumentas yra `null`.

Taigi, jeigu dabar paimsime užklausą

```
select a
  from a in Asmenys
 where a.vardas = "Sigitas"
```

ir jeigu pasitaikys koks nors objektas `a=nil`, tai `a.vardas` duos reikšmę `null`. Ši reikšmė nelygi "Sigitui" ir automatiškai objektas nebus grąžinamas į rezultatą.

Kas bus, jeigu tą pačią užklausą perrašytume štai taip:

```
select a
  from a in Asmenys
 where not (a.vardas != "Sigitas")
```

Jeigu asmuo yra ne `nil` objektas, tuomet užklausos sąlygos prasmė sutampa su prieš tai buvusiąja. Tačiau, jeigu `a=nil` , tuomet `a.vardas` yra reikšmė `null` . Vadinasi, palyginimo reikšmė yra `false` ir, kadangi dar turime operatorių **`not`** , tai `nil` objektas patenkina užklausos sąlygą ir jį turėsime užklausos rezultate. Kadangi reikšmė `nil` priklauso duomenų modeliui — jokios klaidos nėra.

Tačiau kas atsitiks, jei užrašysime tokią užklausą:

```
select a.adresas.miestas
  from a in Asmenys
 where a.vardas = "Sigitas"
```

Visgi — galima užklausos vykdymo klaida! Kodėl? Atsakymas nėra visiškai akivaizdus. O klaida galima todėl, kad į rezultatą šį kartą gali patekti reikšmė `null`. "Kas čia blogo?" — paklausite. Bet ji nepriklauso duomenų modeliui. Tai — tik vidinė OQL reikšmė. `Null` neturi tipo duomenų modelio prasme. Vadinasi, negalima nustatyti užklausos rezultato tipo, t.y užklausos rezultatas gali būti nepanaudojamas. O tai yra klaida. Užklausos rezultate galime turėti reikšmę `null` tik tuo atveju, jeigu ji yra tiktai subužklausa kitoje užklausoje. Gi grąžinti reikšmės `null` į OQL "išorę" negalima.

Specializuotų atributų problema

Pirmoji tipizavimo — ar, teisingiau, *saugaus tipizavimo* (*type safety*) — problema kyla iš to, jog objektiniame modelyje leidžiama specializuoti atributų tipus.

Primename: jeigu mes turime struktūrinius tipus, tai jų tipo/potipio ryšys nustatomas taip:

$$(\forall\, i \in [1, n], n \le m : \tau_i \le \tau'_i) \Rightarrow [A_1: \tau_1 , \dots , A_m: \tau_m] \le [A_1: \tau'_1 , \dots , A_n: \tau'_n]$$

Kaip matome, vienas struktūrinis tipas yra kito struktūrinio tipo potipis, jeigu jis turi bent jau tuos pačius atributus (gali turėti jų ir daugiau), ir jeigu tų atributų tipai yra struktūros supertipo atributų tipų potipiai. Būtent šiame leidime turėti $\tau_i \le \tau'_i$, i = 1,..., n ir slypi galimų klaidų šaltinis.

Imkime tokį pavyzdį (vėl pasinaudosime O_2 notacija):

```
class A
  type tuple ( x: tuple ( Bool: boolean )
             )
end;

method set_x_to_true:boolean in class A
  {
    if self.x.Bool == False
      { self.x = tuple(Bool:True);
        return True;
      }
    else return False;
  };

class B inherit A
```

```
  type tuple ( x: tuple ( Bool: boolean,
                          Int: integer)
             )
end;

method read_x_int:integer in class B
  {
    return self.x.Int;
  };
```

Čia mes apibrėžėme dvi klases. Pirmoji klasė, `A`, turi vieną atributą `x`, kuris yra struktūra iš vieno atributo. Kita klasė, `B`, yra klasės `A` poklasis. Jis paveldi iš klasės `A` tą vienintelį atributą `x` ir jį užkloja, pridėdamas dar vieną atributą `Int`.

Jeigu patikrinsime, ar gerai tipizuoti šie klasių apibrėžimai, tai pamatysime, jog:

1. Pagal apibrėžimą [Bool : *boolean* , Int : *integer*] ≤ [Bool : *boolean*]. Vadinasi, užklojant atributą `x`, jo tipas "sumažėjo";

2. Iš pirmosios išvados seka, jog [x: [Bool : *boolean* , Int : *integer*]] ≤ [x: [Bool : *boolean*]]. Tuomet mes tikrai turime geros struktūros klasių hierarchiją, kur $B \prec A \Rightarrow \sigma(B) \leq \sigma(A)$.

Taigi, mūsų apibrėžimai yra korektiški ir jokių tipų klaidų lyg ir neturėtų būti. O dabar pabandykime atlikti tokias užklausas:

```
select b.read_x_int()
  from b in Bobjects
 where b.set_x_to_true();
```

Manau, jog sutiksite — tipų klaida, vykdant šią užklausą, neišvengiama. Metodas `read_x_int` bandys nuskaityti reikšmę `Int`, kuri neegzistuoja po metodo `set_x_to_true` vykdymo.

Klaida atsirado todėl, jog leidome pakeisti atributo `x` tipą "mažesniu" arba, tiksliau — labiau specializuotu. Todėl, jeigu su `x` bus "pasielgta" kaip su superklasės atributu, dings jo specializuotumas, t.y. prarasime dalį jam būdingos informacijos.

Problemos sprendimas yra paprastas ir pakankamai gerai tipizavimo teorijos pagrįstas: negalima leisti, kad paveldėti atributai keistų savo tipą: jeigu įvesime šį apribojimą, tuomet tokio tipo klaidų nebelieka.

Kovariacija vs. kontravariacija?

Pabaigai aprašysime šiek tiek didesnę ir gerokai sudėtingesnę tipizavimo problemą. Ji susijusi su metodų užklojimu.

Pradėkime iškart nuo problemą iliustruojančio pavyzdžio:

```
class Point
  type tuple ( x: real,
               y: real )
end;

class ColorPoint inherit Point
  type tuple ( c: string      /* x ir y paveldimi iš Point
*/
             )
end;
```

```
method equal (p:Point):boolean in class Point
  {
    return ((self.x == p.x) && (self.y == p.y));
  };

method equal (p:ColorPoint):boolean in class ColorPoint
  {
    return ( (self.x == p.x) &&
             (self.y == p.y) &&
             (self.c == p.c)
           );
  };
```

Kaip matote, vėl turime dvi klases. Viena iš jų vaizduoja taškus dvimatėje erdvėje, o antroji — tokį pat tašką, tik su spalvos charakteristika. Be to, klasėje `ColorPoint`, norint atsižvelgti dar ir į spalvą, perrašytas metodas lyginimui `equal`. Kaip pamename, metodų užklojimui buvo taikomi du apribojimai: (i) neprieštaringumas bei (ii) kovariacija.

Neprieštaringumo sąlygos tikrinti čia neprireiks, nes mes neturime pavyzdyje daugialypio paveldimumo. O štai kovariacijos apribojimą reikia patikrinti.

Visų pirma, pagal šį apribojimą, turi sutapti metodų argumentų skaičius. Tenkinama. Antra — argumentų tipai užklojančiame metode turi būti užklojamo metodo argumentų tipų potipiai. Kadangi $\sigma(\text{ColorPoint}) \leq \sigma(\text{Point})$ — tenkinama. Trečia — užklojančio metodo rezultato tipas turi būti užklojamojo metodo rezultato tipo potipis. Boolean $\leq$ Boolean — tenkinama. Taigi — metodas `equal` užklotas teisingai.

Tiesą sakant, kovariacijos taisyklė, nors ir atrodytų intuityviai teisinga, negarantuoja saugaus tipizavimo. Šį teiginį galima labai greit iliustruoti tokiu pavyzdžiu:

```
method break_it (p:Point):boolean in class Point
  {
    p.equal (self);
  };
```

Dabar, jei parašysime išraišką `(new Point).break_it(new ColorPoint)`, tai galime lengvai įsitikinti, jog ji gerai tipizuota. Tačiau, jeigu pabandysime ją įvykdyti, turėsime klaidą: klasės `ColorPoint` metodas `equal` veltui stengsis ieškoti atributo `c` klasės `Point` objekte.

Pabandykime formalizuoti šią problemą. Tarkime, turime dvi klases: C_1 ir C_2, kur klasė C_2 yra klasės C_1 poklasis. Tuomet, klasės C_2 objektai gali būti "saugiai" naudojami vietoj klasės C_1 objektų, jeigu visi C_1 metodai gali būti saugiai pakeisti klasės C_2 metodais. Jeigu visi C_2 metodai yra paveldėti iš C_1 , tuomet jokių problemų nėra, nes metodas abiem klasėms yra tas pats. Tačiau problema iškyla, jeigu klasės C_1 metodas yra užklojamas klasės C_2 analogu. Tada reikia garantuoti, kad naujasis metodas būtų saugiai naudojamas vietoj senojo *visuose kontekstuose*. Tai reikštų, jog naujojo metodo tipas privalo būti užklotojo metodo tipo potipis.

Kaip apibrėžti metodų tipus? Metodas yra funkcija. Tuomet jo tipas yra $A \to B$. Tada kyla klausimas: kada $A \to B \leq C \to D$? Arba — kada tipo $A \to B$ metodas gali būti saugiai naudojamas vietoj tipo $C \to D$ metodo ?

Pabandykime suformuluoti atsakymą į šį klausimą. Visų pirma, naujosios funkcijos (ar naujojo metodo) rezultatas turi būti panaudojamas vietoj senojo

metodo rezultato. T.y. $B \le D$. Antra — naujasis metodas būtinai turi priimti senojo metodo argumentus. Tai reiškia: $C \le A$. Kaip matote, šis reikalavimas skiriasi nuo kovariacijos taisyklės ir yra vadinamas *kontravariacija* [7]. Atrodytų — jeigu jau tipizavimo saugumą garantuoja kontravariacijos taisyklė, tai ją ir naudokime metodų užklojimui. Deja, tuomet klasės `ColorPoint` metodo `equal` argumentas turėtų būti bent jau klasės `Point` ir kalbos apie galimybę palyginti taškų spalvas negali net ir būti. Tai reiškia, jog netenkame to specializavimo, kurio siekdami įsivedėme klasę `ColorPoint`. Taigi, atrodytų, jog turime rinktis — arba saugus tipizavimas (kontravariacija), arba — specializavimo galimybė (kovariacija). Kovariacija vs. kontravariacija?

Pasirodo, išeitis yra. Norintiems plačiau su ja susipažinti, siūlome paskaityti [Cas95]. O čia paminėsime tiktai pagrindinę sprendimo idėją.

Paprastai, iškviečiant objektui metodą, kurią jo realizaciją iškviesti, nusprendžiama pagal objekto klasę. Šiuo atveju, tipizuojant, naudojamasi klasės/struktūrinio tipo analogija (*record—based*) ir metodas laikomas vienu iš struktūrinės reikšmės atributų. Tai reiškia, jog kiekvienai struktūrai egzistuoja vienintelis metodo variantas ir, kartą identifikavus "struktūrą" (t.y. klasę), metodo realizacija jau pasirenkama automatiškai.

Bet galimas ir kitas požiūris į metodus (ir klases). Jis paremtas užklojimo analogija (*overloading—based*). Pagal šį požiūrį su kiekvieno metodo vardu susiejama aibė jo realizacijų (teisingiau — įvairių funkcijų su jų signatūromis). Ir, kiekvieną kartą, kai objektui iškviečiamas metodas, su juo susietoje aibėje išsirenkama atitinkama funkcija. O pasirenkama ji pagal kai kuriuos funkcijos argumentus.

Šį paskutinį teiginį reiktų paaiškinti. Kiekviename metode, be visų jam siunčiamų parametrų visada yra dar vienas (sakykime — pirmas), apie kurį dažnai net nesusimąstome, ir kuris paprastai metodo viduje vadinamas ***self****. Tai yra tas objektas, kuriam siunčiamas pranešimas. Jo tipą irgi galima įjungti į metodo parametrus signatūroje. Ir tuomet tikrai metodo "versija" pasirenkama pagal jo parametrus — bent jau pagal pirmąjį.*

Kai turime šitokį požiūrį, reikia išskirti dvi skirtingas sąvokas, kurios yra labai panašios:

- *funkcijos pakeitimas kita*. Čia kalbame apie tai, kaip vieną funkciją mes norime panaudoti vietoj kitos. Tuomet turime pasirūpinti tokio pakeitimo saugumu ir naudoti kontravariacijos taisyklę. Bet šis atvejis pasitaiko tik tuomet, kai mes norime funkcijas perduoti kaip parametrus. Tai nebūna taip dažnai, todėl netgi be kontravariacijos taisyklės reikalavimo objektinės DBVS dirba visai gerai.

- *funkcijos pasirinkimas iš sąrašo*. O čia kalbame apie tai, kaip mes renkamės iš aibės funkcijų vieną, reikalingą. Tarkime, turime klasę C_1 ir jos poklasį C_2. Klasėje C_1 metodą m realizuoja funkcija f , o klasė C_2 užkloja šį metodą kita funkcija g. Taigi, su metodo vardu m susieta funkcijų aibė $\{f, g\}$. Bet tai nereiškia, jog klasės C_2 objektams funkcija g naudojama vietoj funkcijos f. Paprasčiausiai naudojama *arba* funkcija f , *arba* funkcija g.
  Tarkime $g : A \to B$, $f : C \to D$. f atitinka klasę C_1 , o g — klasę C_2 (C_2 yra C_1 poklasis). Tarkime, jog po specializacijos $A \le C$. Be to, turi būti $A \to B$

[7] "Kontravariacija" — ne todėl, kad priešinga kovariacijai. Kovariacijos taisyklėje potipio ryšys išlaikomas abiem rodyklės pusėms. O kontravariacijos taisyklėje kairiosioms rodyklės pusėms potipio ryšys yra priešingas.

≤ C→D. Tarkime dar, jog objektas *o* kompiliavimo metu yra klasės C_1. Tuomet su jam pasirenkama funkcija *f*. Tuomet jo argumentai yra tipo C, o rezultatas — tipo D. Tačiau programos vykdymo metu objekto *o* klasė gali "sumažėti" ir tada jam jau bus taikoma funkcija *g*. Bet tuomet jos rezultatas bus tipo B. O tą rezultatą turi būti galima panaudoti kaip tipo D reikšmę, kas buvo išskaičiuota kompiliavimo metu. Tai reiškia B ≤ D. Kaip matote — pasirenkant funkciją iš aibės, turi būti naudojama kovariacijos taisyklė.

Jeigu sudėsime visus šiuos samprotavimus į vieną vietą, tai gausime tokį rezultatą: tarkime, turime metodų užklojimą. Pasirenkant metodo realizaciją konkrečiam objektui, kai kurie jo parametrai imami domėn, o kiti — ne. Tie, pirmieji, turi būti tipizuojami pagal kovariacijos taisyklę (nes pagal juos pasirenkama reikalinga funkcija ir jų atžvilgiu taikoma tik ta arba kita iš jų). O visi likę argumentai privalo būti tipizuojami pagal kontravariacijos taisyklę (nes funkcijos pasirinkime jie nedalyvauja, ir jų atžvilgiu vyksta funkcijos *pakeitimas*).

V. OQL semantika

O dabar pereiname prie pakankamai sudėtingo, bet labai svarbaus klausimo — OQL išraiškų semantikos. Mes jau mokame nustatyti, ar užklausa užrašyta sintaksiškai teisingai. Jei taip — mokame patikrinti, ar ta užklausa gerai tipizuota, t.y. ar ją vykdant nekils problemų dėl netinkamų duomenų (operandų) tipų. Vienintelis dalykas, į kurį belieka išmokti atsakyti — o koks gi bus tos teisingai sintaksiškai užrašytos bei gerai tipizuotos užklausos rezultatas? Atsakyti į šį klausimą reiškia apibrėžti užklausos (bendru atveju — OQL išraiškų) semantiką.

OQL semantika apibrėžiama naudojantis "OQL algebra". Kaip ir reliacinės algebros atveju, toks sprendimas leidžia išskirti ir formalizuoti esminius (bazinius) veiksmus su duomenimis, tokiu būdu sudarant galimybę formaliam veiksmų nagrinėjimui. Be to — algebrinių operatorių naudojimas leidžia apibrėžti tam tikras algebrinių išraiškų užrašymo/perrašymo taisykles, o tai įgalina įvesti užklausų optimizavimą.

Tačiau algebros įvedimas į OQL nėra toks paprastas.

V.A Semantikos nusakymo sunkumai

Pirmas sunkumas įvedant algebrines išraiškas į objektinį pasaulį slypi metodų naudojime. Ar galima (ar verta) metodų semantiką įtraukti į algebros semantiką? O gal galima įvesti į algebrą kokį nors mechanizmą, leidžiantį atlikti su duomenimis "išorinius" veiksmus (t.y. tuos veiksmus, kurių algoritmai ir semantika neįeina į algebrą)?

Ši problema OQL algebroje sprendžiama pasitelkiant specialų operatorių, kuris įgalina algebrinėse išraiškose naudoti klasių metodus.

Dar vienas sunkumas susijęs su kolekcijų apjungimu (*join*). Mat apjungiamos kolekcijos gali būti skirtingų tipų: aibė, multiaibė ir sąrašas. Tačiau šis pirmasis sunkumas įveikiamas pakankamai lengvai: visos kolekcijos algebrinėse išraiškose konvertuojamos į multiaibes (kaip universaliausią kolekcijų tipą).

Antroji problema, susijusi su apjungimo operacija — apjungiamos kolekcijos gali priklausyti viena nuo kitos (pavyzdžiui: `a in Asmenys, v in a.vaikai`). Šiai problemai spręsti įvedamas specialus apjungimo operatorius *d—join* (*dependent join*), kuris realizuojamas ciklu, "įdėtu" (*nested*) į kitą ciklą.

Ir, galų gale, yra dar viena ir, galbūt, didžiausia problema. Su kokiais duomenimis turi dirbti algebros operatoriai? Kalbant paprastai, algebra — operacijos, apibrėžtos tam tikroje aibėje. Taigi, mūsų klausimas — kokioje aibėje apibrėžti OQL algebrą? ODB duomenų modelis, matėme, toks įvairialypis, kad sunku tikėtis unifikuoti algebros operatorius visiems to modelio konceptams.

Tokia problema ir sprendžiama ne taip paprastai. Visi duomenų tipai algebroje transformuojami į struktūrinį tipą. Pavyzdžiui, skaitmuo `5` bus paverstas į struktūrinę reikšmę `[x:5]`, objektas Obj — į `[y:Obj]`, aibė `{a, b}` — į `[z:{a, b}]` ir t.t. Ir visos algebrinės operacijos atliekamos su būtent tokiomis struktūromis.

Šiuo sprendimu tam tikra prasme grįžtama į "reliacinį pasaulį". Nes struktūrinės reikšmės darosi panašios į lentelių įrašus. Tačiau tai nėra visiškas grįžimas prie reliacinio duomenų modelio: juk operacijos su struktūromis (ar, taip sakant, "įrašais") yra kitokios negu reliacinės...

O dabar pereisime prie algebrinių operatorių aprašymo.

V.B OQL algebros operatoriai

Restruktūrizavimo operatorius MAP

Pirmasis mūsų pristatomas operatorius MAP naudojamas "išorinių" veiksmų pritaikymui algebrinių išraiškų operandams. Taip pat šis operatorius apima ir reliacinę projekciją. Trečioji operatoriaus funkcija yra sukonstruoti/eliminuoti struktūras, naudojamas algebrinėse išraiškose.

Ši pastaba susijusi su šiek tiek anksčiau išreikštu tvirtinimu, kad visos algebrinės operacijos atliekamos tik su tam tikro tipo struktūromis. Iš tikrųjų, kaip matote, tai nėra visiška tiesa. Kadangi operatorius MAP leidžia "įsikišti" išoriniams algoritmams į algebrines išraiškas, tai savaime aišku, jog išorinių išraiškų operandai gali būti ne vien struktūros. Šiuo atžvilgiu operatorius MAP yra lyg ir išimtis OQL algebroje. Tačiau leiskit pastebėti, jog operatoriaus MAP argumentai yra struktūrinės reikšmės ir rezultatai — irgi STRUKTŪRINĖS REIKŠMĖS.

Remiantis šiuo paskutiniu pastebėjimu, galima ir ne taip formaliai paaiškinti, ką atlieka operatorius MAP. Jis yra tiltas tarp dviejų pasaulių: tarp algebrinių struktūrų ir nealgebrinių duomenų (operacijų) tipų. Jis ima struktūras, "atiduoda" jas išoriniams algoritmams ir tų algoritmų darbo rezultatą vėl "apipavidalina" kaip struktūras.

Čia pasidaro ir formaliai aišku, kodėl ODBS optimizatorius negali optimizuoti metodų iškvietimo. Ogi todėl, jog metodas įeina kaip išorinis algoritmas MAP operatoriuje. Jis nepriklauso algebrinei išraiškai (ar teisingiau — algebrinės išraiškos medžiui). Todėl jis ir nepatenka į optimizatoriaus "akiratį".

Pagal šias tris išvardintas operatoriaus MAP funkcijas, skiriamos ir trys operatoriaus užrašymo sintaksės:

1. `exp[a] = { [a:x] | x∈exp }`

Šis operatorius paverčia išraišką exp struktūrinių reikšmių aibe, kurioje exp elementai tampa atributu a. Pavyzdžiui, išraiška Tarnautojai[t] paverčia aibę Tarnautojai į aibę struktūrų {[t: t_1], [t: t_2], ... }, kur visi t_i yra klasės Tarnautojas objektai.

2. $MAP_{exp2}(exp1) = \{ exp2(x) \mid x \in exp1 \}$

Šis operatorius skirtas metodų, o galbūt ir išorinių veiksmų exp2 pritaikymą operandams iš exp1. T.y. jis kiekvienam elementui, esančiam išraiškos exp1 rezultate pritaiko išraišką exp2 , ir taip sukonstruoja naują aibę. Toks operatoriaus MAP formatas gali atlikti ir standartinę projekciją, jeigu išraiškoje exp2 paprasčiausiai išrašysime atributų vardus. Pavyzdžiui, išraiška $MAP_{t.vardas}$(Tarnautojai[t]) paverčia aibę Tarnautojai į jau matytą struktūrų aibę, o operatorius MAP savo ruožtu ją paverčia į vardų (o tariant tiksliau — simbolių eilučių) aibę. Būtent dėl tokio užrašymo formato operatorius MAP vadinamas restruktūrizavimu: jis paima vienokios struktūros rezultatą ir iš jo padaro kitokią struktūrą.

3. $MAP_{a:exp2}(exp1) = \{ x \bullet [a:exp2(x)] \mid x \in exp1 \}$

Šioje sintaksėje operatorius MAP pritaiko išraišką exp2 kiekvienam operandui iš exp1 ir prideda jos rezultatą prie jau buvusios operando struktūros. Operatorius • čia reiškia dviejų struktūrų konkatenaciją. Taigi, jeigu užrašysime išraišką $MAP_{v:t.vardas}$(Tarnautojai[t]), tai rezultate turėsime tokią struktūrinių reikšmių aibę:

{[t:t_1, v:<vardas>], [t:t_2, v:<vardas>], ... }

Filtravimo (išrinkimo) operatorius σ

Šis operatorius visikai panašus į reliacinį išrinkimo operatorių.

$$\sigma_p(exp) = \{ x \mid x \in exp \wedge p(x)=True \}$$

Čia p yra loginė išraiška. Operatorius σ nekeičia reikšmių, kurios yra išraiškos exp rezultatas, struktūras. Jis tik nufiltruoja tas reikšmes, kurios netenkina sąlygos p.

Apjungimo operatorius ⋈

Kaip jau buvo minėta, yra du apjungimo operatoriai: įprastinis "reliacinis" ir *d—join*:

1. $exp1 \bowtie_p exp2 = \{ x \bullet y \mid x \in exp1 \wedge y \in exp2 \wedge p(x,y)=True \}$

   Įprastinis *join* operatorius.

2. $exp1 < exp2 > = \{ x \bullet y \mid x \in exp1 \wedge y \in exp2(x) \}$

   D—join operatorius.

Pergrupavimo operatorius Γ

Pakankamai "painus" operatorius, išreiškiantis regrupavimo veiksmą.

$$\Gamma_{g,A,\Theta,f}(exp) = \{ x.A \bullet [g:G] \mid x \in exp \wedge G=f(\{y \mid y \in exp \wedge x.A \Theta y.A\}) \}$$

Čia g, kaip matote, yra atributo vardas; A yra struktūrų iš exp atributai, pagal kuriuos grupuojama aibė exp; Θ yra santykių aibė (tų santykių skaičius atitinka atributų aibėje A skaičių), o f — funkcija.

Pergrupavimo operatorius ima struktūrines reikšmes iš exp, atskiria atributų A reikšmes ir prie kiekvienos iš jų prideda dar vieną atributą g, kurio reikšmė yra funkcija f nuo tų reikšmių iš exp, kurių atributų aibė A turi nustatytą santykį Θ su paimtos struktūrinės reikšmės atributų aibe A.

Rūšiavimo operatorius Sort

Operatorius surūšiuoja argumento multiaibę pagal atributus A ir tvarką Θ.

$$Sort_{A,\Theta}(exp) = \{ x_1,...,x_n \mid \forall i=1..n\ (x_i \in exp \wedge x_i.A\ \Theta'\ x_{i+1}.A) \wedge \forall x \in exp\ x \in \{x_1,...,x_n\} \}$$

Čia Θ' apibrėžtas taip:

$$A = \{a_1,...,a_n\},\ \Theta = \{\Theta_1,...,\Theta_n\}$$

$$x.A\ \Theta'\ y.A \Leftrightarrow (x.a_1\ \Theta_1\ y.a_1) \vee (x.a_1 = y.a_1 \wedge x.a_2\ \Theta_2\ y.a_2) \vee \ldots \vee (x.a_1 = y.a_1 \wedge \ldots \wedge x.a_{n-1} = y.a_{n-1} \wedge x.a_n\ \Theta_n\ y.a_n)$$

Po tokio gana sauso algebros operatorių pristatymo, pažiūrėkime, kaip jie pasireiškia OQL išraiškose. Tai mums tarnaus lyg ir pavyzdžiu. Tuomet ir pamatysime viską smulkiau.

V.C OQL užklausų vertimas į algebrines išraiškas

Dabar aprašysime taisykles, pagal kurias OQL užklausos yra išverčiamos į algebrines išraiškas.

Tarkime, jog turime tokią (apibendrintą) užklausą:

```
  select   s
    from   x1 in f1, ..., xn in fn
   where   p
group by   a1:c1, ... , am:cm
  having   q
order by   o1, ... , ok
```

Tarsime, jog rūšiuojama kuria nors tvarka Θ (didėjančiai arba mažėjančiai).

Žingsnis po žingsnio paseksime, kaip ši užklausa bus paversta į OQL algebrą.

Dalis FROM

Užklausa pradedama apdoroti nuo FROM dalies. Ji yra išverčiama į tokį pavidalą:

$$F = f_1[x_1] <f_2[x_2]> \ldots <f_n[x_n]>$$ [8]

[8] Išraiškoje laikoma, jog operacija *d—join* asociatyvi į kairę, t.y. suskliausta ji turėtų atrodyti $(..((f_1[x_1] <f_1[x_1]>)<f_2[x_2]>)...)<f_n[x_n]>$

Čia tiktai reikia pastebėti jog priklausomas apjungimas *d—join* reikalingas tik ten, kur x_i priklauso nuo x_j, $j<i$. Priešingu atveju, vietoj *d—join* operatoriaus daromas tas pats $f_i[x_i]$.

Dalis WHERE

Tarkime, jog predikatui p suskaičiuoti reikia įvykdyti funkcijas (subužklausas) $g_1, \dots, g_w$. Tuomet po dalies WHERE apdorojimo, algebrinė išraiška turės jau tokį pavidalą:

$$W = \sigma_{p(v_1,\dots,v_w)}(\mathrm{MAP}_{v_1:g_1,\dots,v_w:g_w}(F))$$

Čia predikate p kintamaisiais $v_1, \dots, v_w$ pakeistos subužklausos $g_1, \dots, g_w$.

Dalis GROUP BY

Po dalies WHERE apdorojamas sugrupavimas:

$$G = \Gamma_{\mathrm{partition},<a_1,\dots,a_m>,<=,\dots,=>,\mathrm{Id}}(\mathrm{MAP}_{a_1:c_1,\dots,a_m:c_m}(W))$$

Kaip matome, tas "papildomas" atributas po sugrupavimo yra partition. Be to, grupavime naudojami tik lygybės sąryšiai ir jokia funkcija nuo vienon grupėn papuolančių elementų neatliekama — imama funkcija Id. Taigi, į aibę partition jie sudedami tokie, kokie yra.

Dalis HAVING

Atlikus sugrupavimą, reikia patikrinti HAVING sąlygas.

Tarkime, jog predikatui q suskaičiuoti reikia įvykdyti funkcijas (subužklausas) $h_1, \dots, h_j$. Tuomet:

$$H = \sigma_{q(u_1,\dots,u_j)}(\mathrm{MAP}_{u_1:h_1,\dots,u_j:h_j}(G))$$

Čia predikate q kintamaisiais $u_1, \dots, u_j$ pakeistos subužklausos $h_1, \dots, h_j$.

Dalis ORDER BY

Rezultato išrūšiavimas užrašomas taip:

$$O = \mathrm{Sort}_{<b_1,\dots,b_k>,\Theta}(\mathrm{MAP}_{b_1:o_1,\dots,b_k:o_k}(H))$$

Čia Θ , kaip jau minėjome yra mūsų pasirinkta tvarka. Jeigu tvarka yra "mažėjanti", tai $\Theta = <>,>,\dots,>>$, o jei "didėjanti", tai $\Theta = <<,<,\dots,<>$.

Dalis SELECT

Ir galų gale, kai jau turime visus "reikiamus" atributus, galime padaryti SELECT dalies projekciją:

$$S = \mathrm{MAP}_s(O)$$

Tik reikia neužmiršti, jog ką tik buvome padarę aibės sutvarkymą. Vadinasi, operacija MAP šiuo atveju turi išlaikyti argumento įrašų tvarką.

Taigi, S ir yra mūsų užklausos algebrinė išraiška.

V.D Užklausos vertimo į algebrinę išraišką pavyzdys

Pavyzdžiui išverskime į algebrinę išraišką štai tokią užklausą:

```
select struct ( amzius: a,
                kiekis: count(partition)
              )
  from d in Destytojai,
       k in d.kursai
 where k.pavadinimas = "ODBS"
group by a: d.amzius()
```

Ši užklausa turėtų peržiūrėti visus dėstytojus, dėstančius objektines duomenų bazių valdymo sistemas ir grąžinti kaip rezultatą aibę struktūrinių reikšmių, kuriose būtų įrašytas dėstytojų amžius bei kiek tokio amžiaus dėstytojų dėsto ODBS. Rašome "turėtų", nes jokios semantikos į šią užklausą dar "neįdėjome". Jei pavyks sudaryti algebrinę išraišką, tuomet jau tvirtai galėsime pasakyti, jog užklausą darys būtent tai, ko iš jos norime (arba, teisingiau, kaip ir daugelio programų atveju — darys tai, ką užrašėme, o ne tai, ko norime...).

Taigi, pradėkime nuo FROM dalies. Ji bus išversta taip:

$$\texttt{d in Destytojai} \Rightarrow \texttt{Destytojai[d]}$$

Ši operacija pavers dėstytojų aibę į aibę struktūrinių reikšmių {[d:d1], [d:d2],...}.

$$\texttt{d in Destytojai, k in d.kursai} \Rightarrow \texttt{Destytojai[d] <d.kursai[k]>}$$

O šitos operacijų sekos rezultatas bus jau aibė

```
{ [d:d1, k:k11], [d:d1, k:k12],...,
  [d:d2, k:k21], [d:d2, k:k22],...  }
```

Tolesnis žingsnis — dalies WHERE vertimas. Šiai daliai išversti, reikia "prisijungti" prie struktūros atributą, atitinkantį k.pavadinimas ir po to palyginti šį atributą su "ODBS":

$$\sigma_{\texttt{p="ODBS"}}(\texttt{MAP}_{\texttt{p:k.pavadinimas}}(\texttt{Destytojai[d] <d.kursai[k]>}))$$

Po šių dviejų operacijų turėsime maždaug tokią struktūrinių reikšmių aibę:

```
{[d:d1, k:k11, p:"ODBS"], [d:d2, k:k23, p:"ODBS"],...}.
```

O dabar atėjo laikas ir sugrupavimui. Visų pirma, reikia "pasidaryti" atributą, pagal kurį bus galima grupuoti. Vadinasi, reikės dar vieno restruktūrizavimo, kuris prie struktūros pridėtų atributą su amžiumi. O tuomet pagal tą atributą ir grupuosime.

$$\Gamma_{\texttt{partition,<a>,<=>,Id}}(\texttt{MAP}_{\texttt{a:d.amzius()}}(\sigma_{\texttt{p="ODBS"}}(\texttt{MAP}_{\texttt{p:k.pavadinimas}}(\ .\ .\ .$$

Po sugrupavimo turėsime tokią rezultato aibę:

```
{ [a:a1, partition:{[a:a1, d:d2, k:k23, p:"ODBS"],
                    [a:a1, d:d4, k:k41, p:"ODBS"],
```

```
                    . . .}
  ],
  [a:a2, partition:{[a:a2, d:d5, k:k52, p:"ODBS"],
                    [a:a2, d:d4, k:k21, p:"ODBS"],
                    . . .}
  ],
 . . .

}
```

Kai jau turime sugrupavę elementus, galime padaryti ir galutinę projekciją/restruktūrizavimą:

$$\mathrm{MAP}_{[amzius:a,\ kiekis:count(partition)]}(\Gamma_{partition,<a>,<=>,Id}(\mathrm{MAP}_{a:d.amzius()}(. \ . \ .$$

Taigi, galutinis rezultatas bus tokia aibė:

```
{[amzius:a1, kiekis:kk1], [amzius:a2, kiekis:kk2], ... }
```

Nors visur, rodant tarpinius užklausos rezultatus, vartojome žodį "aibė", nereikia užmiršti, jog po tuo slepiasi multiaibės sąvoka. Būtent ją ir turėjome omeny, rašydami "aibė".

V.E OQL algebros realizavimas

Tai, ką mes pristatėme kaip OQL algebrą, yra teorinė algebra. Realiose ODBS tos teorinės algebros operatoriai realizuojami *fizine algebra*. Fizinės algebros operatoriai yra funkcijos (programos, algoritmai), kurie realizuoja teorinės algebros operatorius.

Taigi, dabar pristatysime tuos operatorius, kurie realizuoja OQL algebrą. Visi šie operatoriai turi vieną "bendrą" argumentą `monoid`, nurodantį, kokio tipo yra operatoriaus rezultato aibė: `list`, `set` ar `bag`.

1. `get (monoid, extent_name, range_variable, predicate)`

   Šis operatorius atitinka OQL užklausą **`select`** `*` **`from`** `range_variable` **`in`** `extent_name` **`where`** `predicate`. Funkcija sukuria kolekciją tokių struktūrinių reikšmių, kurios turi vienintelį atributą , pavadintą `range_variable` ir kurio reikšmės yra elementas iš `extent_name`. Be to, visi šie elementai turi tenkinti sąlygą `predicate`. Sąlyga yra formos $\mathtt{and}(p_1,\ldots,p_n)$, kur p_i yra predikatai. Jokios įdėtos užklausos predikatuose neleidžiamos — visos jos jau turi būti eliminuotos.

   Kaip matote, šis operatorius atitiktų mūsų jau pažįstamus

   $$\sigma_{predicate}(\mathtt{extent\_name[range\_variable]})$$

2. `reduce (monoid, expr, variable, head, predicate)`

   Kiekvienai struktūrinei reikšmei iš `expr`, kuri atitinka sąlygą `predicate`, suskaičiuojama funkcija `head`. Suskaičiuota reikšmė (ar aibė reikšmių) susiejama su atributo vardu `variable`.

   Operatoriumi `reduce` realizuojam išraiška

   $$\mathrm{MAP}_{variable:head}(\sigma_{predicate}(expr))$$

3. `join (monoid, left, right, predicate, keep)`

   Reliacinis apjungimas. Jis konkatenuoja dvi struktūras, esančias išraiškose `left` ir `right`, jeigu jos tenkina sąlygą `predicate`. Jei `keep=left`, tuomet apjungimas elgiasi kaip kairysis išorinis apjungimas (*left—outer join*), kai kairioji struktūra gali apjungime įgyti `null` reikšmes. `Keep=right` reiškia, jog turime dešinįjį išorinį apjungimą. O `keep=none` reiškia reguliarų apjungimą.

   Pavertus operatorių į teorinę algebrą, turėtume paprasčiausią

   $$\texttt{left} \bowtie_{\texttt{predicate}} \texttt{right}$$

4. `unnest (monoid, expr, variable, path, predicate, keep)`

   Operatorius išskleidžia "įdėtą" kolekciją, pasiekiamą pagal `path` išraikoje `exp` ir atrenka visas struktūrines reikšmes, kurios tenkina `predicate`. Pavyzdžiui: jeigu `expr` tipas yra `set([ x:[vardas:String, vaikai:set(Asmuo)] ])`, ir `path=x.vaikai` tai po išskleidimo turėsime aibę `monoid ([ x: [vardas: String, vaikai: set(Asmuo)], variable: Asmuo ])`. Išskleidžiant imamos tik tos `path` reikšmės, kurios tenkina `predicate`. Jei skleidžiamoje aibėje nėra elementų, arba nei vienas iš jų netenkina `predicate`, o `keep=true`, tai `variable` susiejamas su reikšme `null`.

   Šis operatorius gali išreikšti

   $$\sigma_{\texttt{predicate}}(\texttt{expr[root\_of\_path]< path[variable] >})$$

5. `nest (monoid, expr, var, head, groupby, nestvars, predicate)`

   Operatorius sugrupuoja išraišką `expr` pagal kintamuosius `groupby` (`groupby` yra kintamųjų aibė $\texttt{vars}(v_1,\ldots,v_n)$, kur v_i yra kintamasis, apibrėžtas išraiškoje `expr`). Tuomet kiekvienai atributų `groupby` kombinacijai struktūroje pridedami atributai iš `nestvars` su savo reikšmėmis (`nestvars` yra kintamieji, esantys `expr` "viduje") ir dar vienas atributas `var`, kurio reikšmė yra `reduce (monoid, expr, var, head, predicate)`.

   Operatorius `nest` atitiktų tokią teorinės algebros išraišką:

   $$\texttt{MAP}_{\texttt{nestvars:Id}}(\Gamma_{\texttt{var,groupby, <=,...,=>, head}}(\texttt{expr}))$$

6. `map (monoid, expr, variable, function)`

   Išraiškai `expr` įvykdoma funkcija `function`, o jos rezultatas susiejamas su atributo pavadinimu `variable`. Tai tik paprastesnis operatoriaus `reduce` atvejis.

   Šis operatorius išreiškia

   $$\texttt{MAP}_{\texttt{variable:function}}(\texttt{expr})$$

7. `merge (monoid, left, right)`

   Operatorius sulieja (*merge*) dvi išraiškas: `left` ir `right` (jeigu jas galima sulieti).

O dabar pabandykime perrašyti tą pačią pavyzdžio užklausą fizinės algebros operatoriais. Primename, jog užklausa buvo tokia:

```
select struct ( amzius: a,
                kiekis: count(partition)
              )
  from d in Destytojai,
       k in d.kursai
 where k.pavadinimas = "ODBS"
group by a: d.amzius()
```

Tuomet galima tokia fizinės algebros išraiška:

```
reduce ( set,
         nest ( bag,
                unnest ( bag
                         unnest ( bag,
                                  get ( set,
                                        Destytojai,
                                        d,
                                        and()
                                      ),
                                  k,
                                  d.kursai,
                                  and( k.pavadinimas="ODBS" )
                                ),
                         a,
                         d.amzius,
                         and()
                       )
                partition,
                d,
                vars(a),
                vars(),
                and()
              )
         result,
         struct( bind(amzius,a), bind(kiekis,count(partition)) ),
         and()
       )
```

Pateikime šios išraiškos tarpinių rezultatų tipus:

```
get    → set ([d: Destytjas])
unnest → bag ([d: Destytjas, k: Kursas])
unnest → bag ([d: Destytjas, k: Kursas, a: integer])
nest   → bag ([a: integer, partirion: bag(Destytojas)])
reduce → set ([amzius: integer, kiekis: integer])
```

Integralumo palaikymas

Paruošta pagal [BS97].

Kuriant objektines duomenų bazes ir programavimo kalbas joms, pagrindinis dėmesys buvo skiriamas išvystytų tipų sistemų bei duomenų saugojimo mechanizmų integravimui. Tačiau nuošaly liko tokie specifiniai uždaviniai kaip DB atvaizdžių (*view*) valdymas, rolių apibrėžimas ar duomenų integralumo palaikymas. Ir tik paskutiniaisiais metais pagaliau buvo atkreiptas dėmesys ir į šias problemas. Tačiau vieningų sprendimų dar nėra.

Duomenų integralumo palaikymas leidžia (bent jau tam tikru lygiu) užtikrinti, kad saugomi duomenys tikroviškai atvaizduotų realų pasaulį. Tas atitikimas išreiškiamas tuo, kad duomenys tenkina tam tikrus *semantinius integralumo apribojimus*. Pavyzdžiui, jeigu duomenų bazėje saugome asmenis ir jų sutuoktinius, tai visiškai natūralus būtų integralumo apribojimas

Nei vienas asmuo negali būti vedęs (ištekėjusi už) savęs paties

Tokių integralumo apribojimų palaikymas ir valdymas visada buvo vienas iš didžiausių ODBS kūrėjų pažadų. Deja, kol kas nei viena iš jų nepateikė vartotojus tenkinančio sprendimo.

Viena iš priežasčių, atgrasančių ODBS kūrėjus nuo rimto duomenų integralumo palaikymo yra tai, jog integralumo apribojimai privalo būti tikrinami *sistemiškai*, aplikacijų vykdymo metu, po kiekvieno duomenų pasikeitimo. Nors buvo pasiūlyti įvairūs dinaminio tikrinimo optimizavimo mechanizmai, sistemos efektyvumas vis vien paveikiamas labai stipriai. Todėl darosi aišku, kodėl integralumo palaikymas ODBV sistemose paliktas daugiau ar mažiau nuošalyje.

Padėtis būtų gerokai pagerinta, jeigu kažkokia (galbūt netgi didžioji) integralumo apribojimų tikrinimo darbo dalis būtų atliekama statiškai, t.y. *kompiliavimo metu* . Mūsų aprašomas integralumo palaikymo mechanizmas būtent ir remiasi šiuo principu. Jo esmė yra ta, jog kompiliavimo metu patikrinami duomenų bazėje saugomų objektų metodai ir nusprendžiama, ar jie gali pažeisti duomenų integralumo apribojimus. Sprendimas priimamas *formaliai*, t.y. egzistuoja jo įrodymas (pagrindimas). Šiame skyriuje aprašysime siūlomą techniką tokiam metodų tikrinimo formalizavimui.

Aprašymas parengtas pagal vienos iš siūlomo mechanizmo kūrėjų pateikta medžiaga. Nors tyrimas dar nėra baigtas, tačiau egzistuoja ir veikiantis tokios sistemos prototipas [BD93]. O mums ši medžiaga bus puikiu pavyzdžiu to, kas yra daroma, norint išspręsti duomenų integralumo objektinėse duomenų bazėse problemą.

I. Objektinė programavimo kalba su apribojimais

Čia mes pristatysime, kaip šalia įprastinių tipų, klasių, metodų apibrėžimo (aprašymo) programavimo kalba galėtų leisti aprašyti ir duomenų integralumo apribojimus, kurie būtų išreiškiami kaip loginės formulės.

Reikia pažymėti, jog programavimo kalbos bei duomenų bazių valdymo sistemos duomenų modelių ypatumai čia nevadina jokios reikšmės. Svarbiausia yra (ir į tai mes sukoncentruosime savo dėmesį) formulių užrašymas ir jų panaudojimas. Todėl, nemenkindami bendrumo, grubiai išskirkime tokias mūsų duomenų bazių programavimo kalbos dalis: klasės, saugojimo šaknys, metodai ir integralumo apribojimai.

I.A Klasės ir saugojimo šaknys

Čia pateiksime mūsų duomenų bazės schemą, kurią naudosime tolesniame temos dėstyme. Kad būtų aiškiau ir paprasčiau, panaudosime jau pažįstamą O_2 notaciją. Todėl, turbūt, galima susilaikyti nuo platesnių schemos aiškinimų, išskyrus, galbūt, kai kurias detales. Bet tai — jau po schemos pristatymo...

```
class Asmuo
  type tuple ( vardas: String,
               sutuoktinis: Asmuo,
               draugas: Asmuo,
               pinigai: Integer
             )
  method tuoktis (partneris: Asmuo);
         skirtis ();
         leistiPinigus (kiek: Integer);
         isigytiDrauga (patikimas: Asmuo);
end;

class Tevas inherit Asmuo
  type tuple ( vaikai: set(Asmuo)
             )
  method tuoktis (meiluzis: Asmuo);
         skirtis ();
         leistiPinigus (kiek: Integer);
end;

class Tarnautojas inherit Asmuo
  type tuple ( bosas: Asmuo
             )
  method skirtis ();
end;

name Asmenys    : set(Asmuo);
name Tevai      : set(Tevas);
name Tarnautojai: set(Tarnautojas);
```

Kaip matote — turime tris klases. Viena iš jų, "Asmuo", yra šakninė, o kitos dvi — jos poklasiai.

Poklasiuose kai kurie metodai užkloja klasės "Asmuo" metodus. Tai reiškia, jog jiems egzistuoja savos realizacijos tuose poklasiuose.

Kiekvienai iš aprašytų klasių apibrėžiama ir sava saugojimo šaknis. Nors, žinoma, niekas nedraudžia tą patį Tarnautoją priskirti dviem aibėms,

"Tarnautojai" ir "Asmenys" — nelaikyti tarnautojų žmonėmis būtų mažų mažiausia netaktiška...

I.B Integralumo apribojimai

Savo schemai apibrėžkime tokius integralumo apribojimus:

1. Turi egzistuoti bent vienas nevedęs (netekėjusi) asmuo.
2. Nė vienas asmuo negali būti savo paties sutuoktinis.
3. Nė vienas tėvas negali vesti (ištekėti už) savo paties vaiko.
4. Nevedęs tarnautojas privalo turėti viršininką.
5. Asmuo privalo turėti bent kiek nors pinigų.
6. Nevedęs (netekėjusi) asmuo negali turėti vaikų.
7. Bent vienas asmuo privalo neturėti draugo.

Šie apribojimai gali būti išreikšti uždaromis pirmos eilės logikos formulėmis, užrašytomis tam tikra kalba. Mes naudosime tokią kalbą:

- Simboliai apima konstantas (`0`, `1`, ... , `nil`, ...) ir kintamuosius (`x`, `y`, ...).
- Termai konstruojami taip: jei `t` yra termas, o `a` — atributas, tuomet `t.a` irgi yra termas.
- Predikatai palyginimui ir priklausymui išreikšti yra $<$, $=$, $>$, $\neq$, $\notin$, $\in$, ir t.t.
- Formulės sudaromos naudojantis įprastais loginiais konektoriais ($\wedge$, $\vee$, $\neg$, ...) ir kvantoriais (`forall`, `exists`)

Norėtume priminti pirmos eilės kalbas.

Pirmos eilės kalba L užduodama:

- *simbolių aibe C, kurios elementai vadinami konstantomis,*
- *simbolių aibe F, kurios elementai vadinami funkcijomis,*
- *netuščia simbolių aibe P, kurios elementai vadinami predikatais,*
- *begaline simbolių aibe V, kurios elementai vadinami kintamaisiais,*
- *konektorių aibe CON.*

Kalbos L termų aibė $\mathbf{Term}(L)$ apibrėžiama taip:

- $V \cup C \subset \mathbf{Term}(L)$,
- *jei f yra funkcijos simbolis, o $t_1, \ldots, t_n$ — termai, tuomet $f(t_1, \ldots, t_n)$ irgi yra termas.*

Atominių formulių aibė $\mathbf{AtForm}(L)$ apibrėžiama kaip aibė $R(t_1, \ldots, t_n)$, kur $R \in P$, o $t_1, \ldots, t_n \in \mathbf{Term}(L)$.

Kalbos L formulių aibė $\mathbf{Form}(L)$ apibrėžiama kaip:

- $\mathbf{AtForm}(L) \subset \mathbf{Form}(L)$,
- *sujungus formules loginiais konektoriais iš CON, gaunamos naujos formulės.*

Taigi, šia mūsų kalba visi septyni integralumo apribojimai būtų užrašyti taip:

```
C1: exists x in Asmenys: x.sutuoktinis=nil;
```

```
C2: forall x in Asmenys: x.sutuoktinis≠x;

C3: forall x in Tevai: x.sutuoktinis ∉ x.vaikai;

C4: forall x in Tarnautojai: x.sutuoktinis=nil ⇒ x.bosas≠nil;

C5: forall x in Asmenys: x.pinigai > 0;

C6: forall x in Tevai: x.sutuoktinis=nil ⇒ ¬(exists y∈x.vaikai);

C7: exists x in Asmenys: x.draugas=nil
```

Duomenų bazė gali būti traktuojama kaip tam tikras domenas su savais elementais ir ryšiais tarp tų elementų. Duomenų bazę *B* vadinsime suderinama su integralumo apribojimais *C*, jeigu *B* yra *C* modelis.

Primename, jog aibė A yra formulių aibės F modelis, jeigu kiekviena formulė iš F yra teisinga aibėje A.

Šis faktas logikoje žymimas kaip $B \models C$.

I.C Metodai

Metodai yra programos blokai, kurie gali keisti duomenų bazėje saugomus duomenis. Jeigu programuotojas neįveda jokių apsaugos priemonių, metodai gali atvesti bazę į prieštaringą būseną.

Pateikime klasės "Asmuo" metodų realizacijas:

```
method skirtis() in class Asmuo
  {
    self.sutuoktinis = nil;
  };

method tuoktis(partneris: Asmuo) in class Asmuo
  {
    if (self <> partneris)
     { self.sutuoktinis = partneris;
       partneris.sutuoktinis = self;
     }
  };

method leistiPinigus(kiek: Integer) in class Asmuo
  {
    if (self.pinigai–kiek > 0)
     self.pinigai = self.pinigai–kiek;
  };

method isigytiDrauga(patikimas: Asmuo) in class Asmuo
  {
    if (exists a in Asmenys: a<>self ∧ a.draugas=nil)
     self.draugas = patikimas;
  };
```

Kaip matote, metodų realizacijose mes jau panaudojome apibrėžtąją pirmos eilės kalbą.

Visi šie metodai parašyti *saugiai* , nes, prieš atliekant veiksmus su duomenimis, pasitikrinamos sąlygos, garantuojančios, jog nė vienas iš aukščiau surašytų apribojimų nebus pažeistas. Kai kuriais atvejais pakanka paprasčiausiai patikrinti tik tam tikras atributų reikšmes (metoduose `tuoktis` ir `leistiPinigus`), tačiau kartais tenka pasinaudoti *loginėmis užklausomis* , kaip tai daroma metode `isigytiDrauga`. Nors tokių užklausų panaudojimas sulėtina metodo darbą, bet jų praleisti negalima.

Pažymėkime dar, jog panaudota loginė užklausa labai primena integralumo apribojimą `C7`, kuriam išlaikyti, tiesą sakant, ji ir yra skirta. Tačiau šios dvi išraiškos nėra visiškai vienodos. Ir esminė problema yra įrodyti, kad sąlyga (loginė užklausa) užtikrina integralumo apribojimo išlaikymą. Netgi jeigu vietoj sąlygos panaudotumėme tikslią integralumo apribojimo išraišką — tai nebūtų pakankama. Mat integralumo apribojimas išreiškia duomenų bazės būseną *po* pakeitimo, o metodo vykdymo sąlyga tikrinama *prieš* pakeitimą.

Pateikime kitų klasių metodų realizacijas:

```
method skirtis() in class Tarnautojas
  {
    if (self.bosas <> nil)
      self.sutuoktinis = nil;
  };

method skirtis() in class Tevas
  {
    if (¬(exists v ∈ self.vaikai))
      self.sutuoktinis = nil;
  };

method tuoktis(meiluzis: Asmuo) in class Tevas
  {
    if ( self <> meiluzis ∧
         ¬(meiluzis ∈ self.vaikai) ∧
         ¬(self ∈ meiluzis.vaikai)
       )
     { self.sutuoktinis = meiluzis;
       meiluzis.sutuoktinis = self;
     }
  };

method leistiPinigus(kiek: Integer) in class Tevas
  {
    if (self.pinigai–kiek > 0)
     self.pinigai = self.pinigai–kiek;

    forall v where (v in self.vaikai) do
      if (v.pinigai–kiek > 0)
         v.pinigai = v.pinigai–kiek;
  };
```

Čia mes apibrėžėme metodus, užklojančius klasėje "Asmuo" esančius jų analogus. Tolesniame temos nagrinėjime tokį faktą žymėsime taip: tarkime, turime metodą m , kuris yra užklotas keliomis kitomis versijomis, esančiomis klasės "Asmuo" poklasiuose. Tuomet rašysime $m = \{m_1 , \ldots , m_n\}$, kur m_i yra skirtingos metodo m realizacijos.

O dabar pereisime prie formalaus metodų analizavimo. Iš pradžių patikslinkime kalbą, kuria bus (yra) parašyti analizuojami metodai.

Šitoks apibrėžimas nesumenkina bendrumo, nes tą kalbą galime apibrėžti bet kokią. Apibrėžiame tik norėdami palengvinti tolesnį atliekamų veiksmų

formalizavimą.

Apibrėžimas 1 Termai ir programos instrukcijos rekursyviai apibrėžiami kaip:

termas ::= *kintamasis* | *termas.atributas* | *termas*←*metodas* | *termas* Θ *termas*

čia Θ reiškia aritmetinę operaciją.

instrukcija ::= *termas* `:=` *termas*
| instrukcija `;` instrukcija
| `{` instrukcija `}`
| `if` sąlyga `then` instrukcija
| `forall` termas `where` sąlyga `do` instrukcija

II. Metodų vertimas į tarpinę formą

Objektinių programavimo kalbų analizė yra pakankamai sudėtinga, ypač kai objektai gaunami *naviguojant* kitų objektų atributais, arba kai jie yra metodų rezultatas. Mūsų apibrėžtoji kalba leidžia parašyti išraišką:

```
(o←m).a := v
```

Kad išvengtume nepatogumų, susiaurinkime mūsų programavimo kalbą, pakeisdami termo formą:

Apibrėžimas 2 *termas* ::= *kintamasis* | *termas.atributas* | *termas* Θ *termas*

Net ir įvedus tokį supaprastinimą, savo programavimo kalba vis dar galime užrašyti pakankamai sudėtingas išraiškas, kurių analizė nebūtų tokia jau paprasta. Todėl suformuluokime tokią metodų analizės strategiją:

- visų pirma, metodai išverčiami į tarpinę, paprastesnę, formą (žinoma, neprarandant metodo veiksmų logikos);
- po to analizuojama būtent tarpinė forma. Dėl jos paprastumo ir analizė darosi paprastesnė.

Taigi, nepaisant jau įvesto programavimo kalbos supaprastinimo, lieka dar vienas sunkumas — nagrinėti navigavimą objektų atributais. Pavyzdžiui, tarkime, jog turime išraišką

```
a.sutuoktinis.sutuoktinis = b.sutuoktinis
```

Čia keičiami ne patys objektai `a` ar `b`, o objektai, *pasiekiami* iš objektų `a` ir `b`. Kad galėtume supaprastinti tokio tipo išraiškas, įveskime instrukciją

`forone` *objektas* `where` *sąlyga* `do` *instrukcija*

Tuomet mūsų išraiška su navigavimu objektų atributais gali būti perrašyta kaip

```
forone o1 where a.sutuoktinis = o1 do
  forone o2 where b.sutuoktinis = o2 do
    o1.sutuoktinis = o2
```

Toks vertimas gali būti atliktas su bet kokio ilgio kelio išraiškomis. Be to, yra dar vienas privalumas: termų apibrėžime belieka tik "*kintamasis.atributas*" vietoj bendresnio "*termas.atributas* ".

Dabar galime apibrėžti, į ką verčiama mūsų programavimo kalba.

Apibrėžimas Metodu laikysime programą $m\ (p_1\ ,\ \ldots\ ,\ p_n) = instrukcija$, kur m yra metodo vardas, $p_1\ ,\ \ldots\ ,\ p_n$ — jo parametrai, o *instrukcija* vadinama metodo kūnu ir jos sintaksė yra tokia:

instrukcija ::= *kintamasis.atributas* `:=` *kintamasis*
| *instrukcija* `;` *instrukcija*
| `{` *instrukcija* `}`
| `if` *sąlyga* `then` *instrukcija*
| `forall` *kintamasis* `where` *sąlyga* `do` *instrukcija*
| `forone` *kintamasis* `where` *sąlyga* `do` *instrukcija*

Kai kurių programavimo kalbos išraiškų (instrukcijų) vertimas į šią formą yra tiesioginis, o kitoms, sudėtingesnėms, vertimo algoritmus jau aprašėme.

Taip aprašytiems metodams taikysime formalią analizę, padedančią nustatyti (įrodyti), ar metodas gali pažeisti integralumo apribojimus.

III. Predikatų transformacijos

Šiame skyriuje aprašysime automatinio įrodymo, kad metodas išlaiko duomenų bazės integralumą, techniką.

Programų analizės tyrimai buvo stipriai inspiruoti Deikstros (*Dijkstra*) ir Horės (*Hoare*) logikos darbų. Taip pat galime sakyti, jog šie tyrimai priklauso abstraktaus programų interpretavimo sričiai.

Abstraktus programų interpretavimo tikslas yra pateikti iš anksto, kompiliavimo metu, tam tikrą informaciją apie programos elgesį jos vykdymo metu. Tai informacijai pateikti visai nereikia realaus programos vykdymo realioje aplinkoje. Informacija gaunama po *abstraktaus* vykdymo *abstrakčioje* programos aplinkoje. Būtent šiuo principu ir remiasi mūsų aprašoma technika.

Pagrindinė sąvoka tolesniame aprašyme yra *predikatų transformacija* (*predicate transformer*). Aprašysime du transformacijų tipus ir parodysime, kaip jos gali būti taikomos metodų tikrinimui. Tiesioginė predikatų transformacija ima pirminę sąlygą φ, metodą m ir generuoja galinę sąlygą $\vec{m}(\varphi)$. Atvirkštinė predikatų transformacija daro atvirkščiai: pagal galinę sąlygą φ ir metodą m generuoja pradinę sąlygą $\overleftarrow{m}(\varphi)$. Jeigu tiksliau, abiejų predikatų transformacijų veikimas apibrėžiamas šitaip:

- jeigu φ teisinga prieš m vykdymą, tai $\vec{m}(\varphi)$ teisinga po m įvykdymo;
- jeigu $\overleftarrow{m}(\varphi)$ teisinga prieš m vykdymą, tai φ teisinga po m įvykdymo.

Predikatų transformacijų pritaikymas mūsų uždaviniui yra visiškai natūralus: metodas m negali pažeisti integralumo apribojimo C, jeigu $\vec{m}(C) \Rightarrow C$ arba jeigu $C \Rightarrow \vec{m}(C)$.

Dabar aprašysime šią idėją plačiau.

III.A Tiesioginė predikatų transformacija

Šioje dalyje aprašysime, kaip metodams sudaroma tiesioginė predikatų transformacija. Jos apibrėžimui panaudosime indukciją tiek pagal metodo, tiek pagal integralumo apribojimo formulės sandarą. Kad būtų aiškiau, pradėsime nuo paprastų metodų, kuriuose nėra ciklų ir metodų užklojimo. Pastaruosius du atvejus panagrinėsime vėliau. Taigi, tiesioginė predikatų transformacija:

Apibrėžimas Tegu m yra metodas. Jam mes apibrėžiame *tiesioginę predikatų transformaciją* $\vec{m}$ (predikatų transformacija yra logikos formulė) šitokiu būdu:

Tarkime, kad φ ir ψ yra formulės, o u, v, x ir y — kintamieji. Tebūnie a — atributo vardas. Tuomet tiesioginė predikatų transformacija apibrėžiama pagal indukciją:

- Indukcija pagal formulės sandarą:
  1. $\vec{m}((\varphi)) \equiv \vec{m}(\varphi)$
  2. $\vec{m}(\varphi \wedge \psi) \equiv \vec{m}(\varphi) \wedge \vec{m}(\psi)$
  3. $\vec{m}(\varphi \vee \psi) \equiv \vec{m}(\varphi) \vee \vec{m}(\psi)$
  4. $\vec{m}(\texttt{forall x: } \varphi(x)) \equiv \texttt{forall x: } \vec{m}(\varphi(x))$
  5. $\vec{m}(\texttt{exists x: } \varphi(x)) \equiv \texttt{exists x: } \vec{m}(\varphi(x))$
- Indukcija pagal metodo sandarą:
  6. Jeigu $m \equiv \texttt{u.a:=v}$ ir φ yra literalas[9], tuomet:
     — Jei $\varphi \equiv (x.a = y)$, tai $\vec{m}(\varphi) \equiv (u = x \wedge u.a = v) \vee (u \neq x \wedge u.a = v \wedge x.a = y)$
     — Jei $\varphi \equiv (x.a \neq y)$, tai $\vec{m}(\varphi) \equiv (u = x \wedge u.a = v) \vee (u \neq x \wedge u.a = v \wedge x.a \neq y)$
     — Priešingu atveju $\vec{m}(\varphi) \equiv \varphi \wedge u.a = v$
  7. Jeigu $m \equiv i_1 ; i_2$, tuomet $\vec{m}(\varphi) \equiv \vec{i}_2(\vec{i}_1(\varphi))$
  8. Jeigu $m \equiv \{i\}$, tuomet $\vec{m}(\varphi) \equiv \vec{i}(\varphi)$
  9. Jeigu $m \equiv \texttt{if } \psi \texttt{ then } i$, tuomet $\vec{m}(\varphi) \equiv \vec{i}(\psi \wedge \varphi) \vee (\neg\psi \wedge \varphi)$
  10. Jeigu $m \equiv \texttt{forone } v \texttt{ where } \psi(v) \texttt{ do } i$, tuomet $\vec{m}(\varphi) \equiv \texttt{exists } v : \vec{i}(\psi(v) \wedge \varphi)$

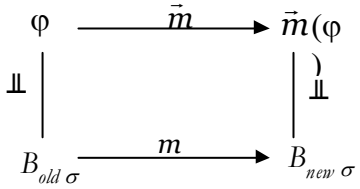
---

[9] *literalas* yra atominė formulė, arba jos neiginys.

Apibrėžime tariame, jog nėra jokių sutapimų tarp vardų, žyminčių skirtingus objektus (reikšmes, atributus, ...). Priešingu atveju, visuomet galime padaryti pervardinimą.

Jeigu tarsime, kad formulėse, aprašančiose integralumo apribojimus yra laisvų kintamųjų, kurie įgauna reikšmes skirtingiems ribojamos duomenų bazės egzemplioriams, tai tų kintamųjų inicializavimo (*assignement*) funkciją vadinsime σ, o duomenų bazės egzempliorius tuomet bus B_σ.

Taigi, tuomet ryšį tarp duomenų bazės, metodo ir tiesioginės predikatų transformacijos galime pavaizduoti štai taip:



Paveikslėlis 6 Tiesioginės predikatų transformacijos veikimo diagrama

Dabar būtų gerai įrodyti, jog mūsų predikatų transformacija suteikia patikimą informaciją apie metodo elgesį jo vykdymo metu, arba, kitais žodžiais tariant, reikia įrodyti, jog tiesioginė transformacija yra *korektiška*.

Teorema 1 Tegu B_{old} yra duomenų bazės egzempliorius, m yra metodas, o B_{new} — duomenų bazė, gauta iš B_{old} , įvykdžius metodą m. Taip pat tegu φ yra formulė, o σ — formulės φ laisvų kintamųjų inicializacija. Tuomet

$$B_{old\ \sigma} \models \varphi \Rightarrow B_{new\ \sigma} \models \vec{m}(\varphi)$$

arba — predikatų transformacija $\vec{m}$ yra korektiška.

▶

Korektiškumą įrodinėsime pagal tiesioginės predikatų transformacijos apibrėžimą — naudodami indukciją pagal formulės ir pagal metodo sudėtį.

- sąlygai (1) įrodymas trivialus.
- sąlygoje (2) tarkime, jog φ ir ψ yra formulės ir, pagal indukcijos hipotezę, tarkime jog predikatų transformacija šioms formulėms yra korektiška .

  Tuomet tariame $B_{old\ \sigma} \models \varphi \wedge \psi$.

  Vadinasi, $B_{old\ \sigma} \models \varphi$ ir $B_{old\ \sigma} \models \psi$.

  Pagal indukcijos hipotezę turime, jog

  $$B_{old\ \sigma} \models \varphi \Rightarrow B_{new\ \sigma} \models \vec{m}(\varphi) \text{ ir}$$

  $$B_{old\ \sigma} \models \psi \Rightarrow B_{new\ \sigma} \models \vec{m}(\psi)$$

  Taigi, iš čia išplaukia, jog $B_{old\ \sigma} \models \varphi \wedge \psi \Rightarrow (B_{new\ \sigma} \models \vec{m}(\varphi) \wedge \vec{m}(\psi))$. O tai reiškia, jog predikatų transformacija yra korektiška.
- įrodymas (3) sąlygai visiškai panašus.

- sąlygoje (4) tarkime, jog φ yra formulė, kuriai predikatų transformacija yra korektiška (indukcijos hipotezė).

  Tada imame $B_{old\,\sigma} \models$ `forall x:` φ(x), kuris reiškia, jog kiekvienam x $B_{old\,\sigma} \models \varphi(x)$.

  Pagal indukcijos hipotezę, kiekvienam x $B_{new\,\sigma} \models \vec{m}(\varphi(x))$.

  Bet tai reiškia, jog $B_{new\,\sigma} \models$ `x:` $\vec{m}(\varphi(x))$ — predikatų transformacija yra korektiška.

- sąlygai (5) įrodymas analogiškas.

- sąlyga (6) yra indukcijos bazė. Tegu $m \equiv u.a := v$, o $\varphi \equiv x.a = y$. Jeigu $B_{old\,\sigma} \models \varphi$, tai:

  turime $B_{new\,\sigma} \models u = x$, ir tada $B_{new\,\sigma} \models u.a = v$

  arba turime $B_{new\,\sigma} \models u \neq x$, ir tuomet $B_{new\,\sigma} \models u.a = v \wedge x.a = y$.

  Taigi, turime, jog $B_{new\,\sigma} \models (u = x \wedge u.a = v) \vee (u \neq x \wedge u.a = v \wedge x.a = y)$.

  Kiti du atvejai (kur keičiasi φ sandara) yra analogiški.

- sąlyga (7) — tarkime, jog $m \equiv i_1 ; i_2$. pagal indukcijos prielaidą imame, kad predikatų transformacija korektiška abiem instrukcijoms i_1 ir i_2 .pažymėkime B_1 bazę B_{old} po instrukcijos i_1 įvykdymo. Tuomet, remdamiesi indukcijos hipoteze, turime, kad

  $B_{old\,\sigma} \models \varphi \Rightarrow B_{1\,\sigma} \models \vec{i}_1(\varphi)$

  $B_{1\,\sigma} \models \vec{i}_1(\varphi) \Rightarrow B_{new\,\sigma} \models \vec{i}_2(\vec{i}_1(\varphi))$

Taigi, predikatų transformacija korektiška ir šiuo atveju.

- sąlyga (8) įrodoma trivialiai, o sąlygos (9) įrodymas analogiškas (6) atvejui.

- paskutinė sąlyga (10). Indukcijos hipotezė yra, jog instrukcijai i teorema teisinga. Taigi, $B_{old\,\sigma} \models \varphi$. Tuomet:

  — galbūt egzistuoja toks objektas v, kad $B_{old\,\sigma} \models \psi(v)$. Tada turime jog egzistuoja v toks, kad toji $B_{old\,\sigma} \models \psi(v) \wedge \varphi$. Iš indukcijos hipotezės išplaukia, jog egzistuoja v toks, kad $B_{new\,\sigma} \models \vec{i}\,(\psi(v) \wedge \varphi)$. vadinasi, galų gale turime $B_{new\,\sigma} \models$ `exists` $v : \vec{i}\,(\psi(v) \wedge \varphi)$;

  — galimas atvejis, jog objektui v netenkinama sąlyga ψ. Tuomet metodo vykdymo metu įvyksta klaida[10], ir, formaliai sakant, $B_{new\,\sigma} \models$ `false` .

Todėl galų gale mes turime $B_{new\,\sigma} \models$ (`exists` $v : \vec{i}\,(\psi(v) \wedge \varphi)) \vee$ `false` , kas ekvivalentu sąlygos (10) rezultatui.

◀

Šia teorema mes įrodėme, kad apibrėžtoji predikatų transformacija pateikia teisingą informaciją apie metodų elgesį vykdymo metu. Bet ar tos informacijos

[10] Neužmirškime, kam skirta instrukcija `forone`. Ji *neišrenka* visų objektų, kurie tenkina sąlygą ψ. Objektas v jau egzistuoja — jam tikrinama sąlyga ψ. Jeigu kalbant paprasčiau — instrukcija `forone` skirta kelio išraiškai nurodyti, kur imamas v ir kreipiamasi į jo atributą (tas kreipimasis ir atitinka sąlygą ψ). Todėl natūralu, kad, jeigu objektas nemoka atsakyti į tokį kreipimąsį (sąlyga ψ netenkinama), įvyksta klaida.

pakanka? Tarkime, apibrėžiame predikatų transformaciją $\vec{m}(\varphi)$ = `true` kiekvienam metodui m ir kiekvienai formulei φ. Tokia transformacija yra korektiška (pagalvokite, kodėl), bet nesuteikia jokios informacijos... Tačiau dabar mes įrodysime teoremą, kuri sako, kad tokia predikatų transformacija, kokią apibrėžėme, padeda įrodyti metodų "saugumą" integralumo apribojimų atžvilgiu.

Teorema 2 Tegu C yra apribojimas, o m — metodas. Tuomet, jeigu $\vec{m}(C) \Rightarrow C$, tai m nepažeidžia apribojimo C.

▶

m bus saugus apribojimo C atžvilgiu, jeigu, bet kuriuo metu įvykdžius jį duomenų bazėje B_{old}, tenkinančioje C, rezultate gausime duomenų bazę B_{new}, irgi tenkinančią C. Taigi, pagal prielaidą mes turime $B_{old} \models C$. Pagal anksčiau įrodytą teoremą, gauname, jog $B_{new} \models \vec{m}(C)$. Pagal teoremos sąlygą turime $\vec{m}(C) \Rightarrow C$. Tada $B_{new} \models C$ ir metodas m išlaiko apribojimą C.

◀

Taigi, norint įrodyti metodo saugumą, belieka mokėti įrodyti $\vec{m}(C) \Rightarrow C$. Deja, tai nėra taip paprasta. Šiuo metu pateikta nemažai automatinių teoremų įrodinėjimo metodikų. Tačiau jie turi vieną savybę — jie yra *pilni*. Jeigu formulė teisinga, tai jie visuomet suranda jos įrodymą per *baigtinį* laiką. Bet ką daryti, jeigu formulė *neteisinga*? Tuo atveju vis bandoma rasti įrodymą, kuris neegzistuoja. Kadangi realiai bandymo įrodyti laikas tokiu atveju būna apribojamas iš anksto kažkokia konstanta, tai mūsų įrodinėjimo technika:

- aptiks esančius saugiais kai kuriuos saugius metodus (priklausomai nuo įrodymo laiko apribojimo),
- nurodys esant nesaugiais visus nesaugius ir kai kuriuos saugius metodus.

Šitų dviejų faktų verta neužmiršti, kalbant apie tokį integralumo apribojimų palaikymo būdą.

O dabar, kaip jau buvo žadėta, panagrinėsime tiesioginę transformaciją metodams su ciklais, arba, tiksliau sakant — predikatą ciklo sakiniui, nes visi kiti įrodyti rezultatai išlieka nepakitę.

Mūsų apibrėžtoje programavimo kalboje ciklo sakinys yra toks:

`forall` *kintamasis* `where` *sąlyga* `do` *instrukcija*

Norint įrodyti, kad ciklas yra saugus integralumo apribojimo atžvilgiu, visų pirma reikia įrodyti, kad *instrukcija* yra saugi. O jeigu instrukcija bet kada išlaiko integralumo apribojimą, tai ir visas ciklas instrukcijų turi išlaikyti tą apribojimą. Tokia pagrindinė idėja. O dabar — pati predikatų transformacija ciklo sakiniui:

$$\vec{m}(\varphi) = \begin{cases} C & \text{jeigu } \varphi \Rightarrow C \text{ ir } \vec{i}(C) \Rightarrow C \\ \texttt{true} & \text{priešingu atveju} \end{cases}$$

Po tokio apibrėžimo, transformacijos korektiškumas išlieka.

Teorema 3 Tegu B_{old} yra duomenų bazės egzempliorius, m yra metodas, o B_{new} — duomenų bazė, gauta iš B_{old} , įvykdžius metodą m. Taip pat tegu φ yra formulė, o σ — formulės φ laisvų kintamųjų inicializacija. Tuomet

$$B_{old\,\sigma} \models \varphi \Rightarrow B_{new\,\sigma} \models \vec{m}(\varphi)$$

arba — predikatų transformacija $\vec{m}$ yra korektiška.

▶

Visi ankstesni atvejai jau įrodyti. Belieka įrodyti, kad, jeigu $m \equiv$ `foralll` x `where` $P(x)$ `do` $i(x)$ ir $\vec{i}$ yra korektiška (indukcijos prielaida), tai $\vec{m}$ — irgi korektiška.

Tegu B_{old} yra duomenų bazė prieš m vykdymą, o $\{ x \mid B_{old} \models P(x) \} = \{ a_1 , \dots , a_n \}$. Čia $a_1 , \dots , a_n$ yra būtent tie duomenų bazės objektai, kuriems ir vykdoma ciklo instrukcija. Tuomet galimi du atvejai:

- $\varphi \Rightarrow C$ ir $\vec{i}(C) \Rightarrow C$. Tada $\vec{m}(\varphi) = C$. Jeigu $B_{old} \models \varphi$, o $\varphi \Rightarrow C$, tai $B_{old} \models C$. Dabar apibrėžiame n duomenų bazių taip:

  B_1 yra instrukcijos $i(a_1)$ vykdymo bazei B_{old} rezultatas.

  B_2 yra instrukcijų sekos $i(a_1)$; $i(a_2)$ vykdymo bazei B_{old} rezultatas.

  . . .

  B_n yra instrukcijų sekos $i(a_1)$; ... ; $i(a_n)$ vykdymo bazei B_{old} rezultatas.

  Kadangi seka $i(a_1)$; ... ; $i(a_n)$ ekvivalenti ciklo vykdymui, tai turime $B_n = B_{new}$. O kadangi $\vec{i}(C) \Rightarrow C$, tai $B_1 \models C , \dots , B_n \models C$ ir, vadinasi, $B_{new} \models C$.

- priešingu atveju $\vec{m}(\varphi)$ = `true` ir trivialiai $B_{new} \models C$.

◀

Šioje vietoje įterpkime mažytę pastabėlę, kad ciklo rezultatas gali priklausyti nuo instrukcijų $i(a_j)$ *vykdymo tvarkos ir tuo atveju mūsų panaudotas duomenų bazių "ekvivalentumas" nėra visiškai teisingas. Tačiau predikatų transformacijos atžvilgiu, tvarka neturi jokios reikšmės.*

Esant ciklams metode, išlieka teisinga ir **Teorema 2**. Tik reikia pridurti dar du apribojimus įrodymo technikai: įrodymai turi būti rasti per baigtinį laiką ne vien faktui $\vec{m}(C) \Rightarrow C$, bet ir $\varphi \Rightarrow C$ bei $\vec{i}(C) \Rightarrow C$.

Antra mūsų "pažadėtoji" išnagrinėti tema yra rekursijos ir užklotų metodų naudojimas. Pradėkime nuo rekursijos.

Rekursija yra gan sudėtingas procesas ir todėl gan sudėtingai analizuojamas. Mes nesunkinsime situacijos ir aprėpsime tik gan ribotą rekursijos atvejį: metodą su rekursija apibendrintai užrašysime taip:

```
method m (parametrai) in class K
 { i ;
   if not rekursijos_bazė
```

rekursyvus_kreipinys
}

Tokiam parametrui tiesioginę predikatų transformaciją vėl gi apibrėžiame kaip

$$\vec{m}(\varphi) = \begin{cases} C & \text{jeigu } \varphi \Rightarrow C \text{ ir } \vec{i}(C) \Rightarrow C \\ \texttt{true} & \text{priešingu atveju} \end{cases}$$

Transformacija lieka korektiška:

Teorema 4 Tegu B_{old} yra duomenų bazės egzempliorius, m yra metodas, o B_{new} — duomenų bazė, gauta iš B_{old} , įvykdžius metodą m. Taip pat tegu φ yra formulė, o σ — formulės φ laisvų kintamųjų inicializacija. Tuomet

$$B_{old\,\sigma} \models \varphi \Rightarrow B_{new\,\sigma} \models \vec{m}(\varphi)$$

arba — predikatų transformacija $\vec{m}$ yra korektiška.

▶

Bet kuris m vykdymas atitiks keletą to paties metodo vykdymų vis su kitais parametrais. Todėl jį galima pakeisti tokia instrukcijų seka:

i (parametrai$_1$) ; ... ; i (parametrai$_n$)

Ir vėl galimi du atvejai:

- $\varphi \Rightarrow C$ ir $\vec{i}\,(C) \Rightarrow C$. Tada $\vec{m}(\varphi) = C$. Jeigu $B_{old} \models \varphi$, o $\varphi \Rightarrow C$, tai $B_{old} \models C$. Dabar apibrėžiame n duomenų bazių taip:

  B_1 yra instrukcijos i (parametrai$_1$) vykdymo bazei B_{old} rezultatas.

  B_2 yra instrukcijų sekos i (parametrai$_1$) ; i (parametrai $_2$) vykdymo bazei B_{old} rezultatas.

  . . .

  B_n yra instrukcijų sekos i (parametrai $_1$) ; ... ; i (parametrai $_n$) vykdymo bazei B_{old} rezultatas.

  Dėl metodo m ir instrukcijų sekos ekvivalntumo turime $B_n = B_{new}$. O kadangi $\vec{i}\,(C) \Rightarrow C$, tai $B_1 \models C$, ... , $B_n \models C$ ir, vadinasi, $B_{new} \models C$.

- priešingu atveju $\vec{m}(\varphi) = \texttt{true}$ ir trivialiai $B_{new} \models C$.

◀

Teorema 2 vėl lieka teisinga. Tik, jeigu turime reikalą su užklotais metodais, reikia pastebėti, kad tuomet metodas m yra saugus, jeigu visos jo realizacijos saugios:

Teorema 5 Tegu $m = \{ m_1 , \ldots , m_n \}$ yra užklotas metodas, o C — integralumo apribojimas. Tada m išlaiko apribojimą C , jeigu $\forall i \in \{ 1, \ldots , n \}$, $\vec{m}_i\,(C) \Rightarrow C$.

Statinis metodų saugumo tikrinimas leidžia moduliarizuoti šį procesą. Kartą patikrinus visas metodo realizacijas ir, atsiradus naujai, pakanka patikrinti tik atsiradusiąją — kitoms metodo "versijoms" tikrinimo jau nebereikia.

III.B Atvirkštinė predikatų transformacija

Dabar aprašysime atvirkštinę predikatų transformaciją. Ši transformacija yra funkcija, kuri, esant duotam metodui m ir formulei φ, pateikia pakankamas sąlygas m veikiamiems duomenims, kad po to poveikio jie tenkintų sąlygą φ.

Parodysime, kad atvirkštinė transformacija irgi gali būti naudojama metodų saugumo tikrinimui. Dar daugiau — atvirkštinė transformacija pateikia netgi papildomą tokio tikrinimo funkcionalumą (kokį — aprašysime vėliau). Tačiau, atsižvelgdami į tai, jog didelė dalis teorijos bus labai panaši į tiesioginės predikatų transformacijos atvejį, pateiksime tik pagrindinius faktus, palikdami skaitytojui pačiam padaryti smulkesnes analogijas.

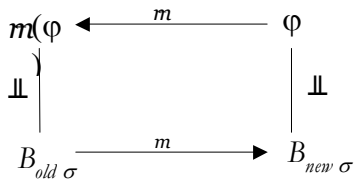
Taigi, atvirkštinė predikatų transformacija:

Apibrėžimas Tegu m yra metodas. Jam mes apibrėžiame *atvirkštinę predikatų transformaciją* $\bar{m}$ (predikatų transformacija yra logikos formulė) šitokiu būdu:

Tarkime, kad φ ir ψ yra formulės, o u, v, x ir y — kintamieji. Tebūnie a — atributo vardas. Tuomet atvirkštinė predikatų transformacija apibrėžiama pagal indukciją:

- Indukcija pagal formulės sandarą:
  1. $\bar{m}(\varphi) \equiv \bar{m}(\varphi)$
  2. $\bar{m}(\varphi \wedge \psi) \equiv \bar{m}(\varphi) \wedge \bar{m}(\psi)$
  3. $\bar{m}(\varphi \vee \psi) \equiv \bar{m}(\varphi) \vee \bar{m}(\psi)$
  4. $\bar{m}(\texttt{forall x: } \varphi(x)) \equiv \texttt{forall x: } \bar{m}(\varphi(x))$
  5. $\bar{m}(\texttt{exists x: } \varphi(x)) \equiv \texttt{exists x: } \bar{m}(\varphi(x))$
- Indukcija pagal metodo sandarą:
  6. Jeigu $m \equiv \texttt{u.a:=v}$ ir φ yra literalas, tuomet:
     — Jei $\varphi \equiv (x.a = y)$, tai $\bar{m}(\varphi) \equiv (u = x \wedge v = y) \vee (u \neq x \wedge x.a = y)$
     — Jei $\varphi \equiv (x.a \neq y)$, tai $\bar{m}(\varphi) \equiv (u = x \wedge v \neq y) \vee (u \neq x \wedge x.a \neq y)$
     — Priešingu atveju $\bar{m}(\varphi) \equiv \varphi$
  7. Jeigu $m \equiv i_1 ; i_2$, tuomet $\bar{m}(\varphi) \equiv \bar{i_1}(\bar{i_2}(\varphi))$
  8. Jeigu $m \equiv \{i\}$, tuomet $\bar{m}(\varphi) \equiv \bar{i}(\varphi)$
  9. Jeigu $m \equiv \texttt{if } \psi \texttt{ then } i$, tuomet $\bar{m}(\varphi) \equiv (\psi \wedge \bar{i}(\varphi)) \vee (\neg\psi \wedge \varphi)$
  10. Jeigu $m \equiv \texttt{forone } v \texttt{ where } \psi(v) \texttt{ do } i$, tuomet $\bar{m}(\varphi) \equiv (\texttt{exists } v : \psi(v)) \wedge \bar{i}(\varphi)$

Dabar ryšį tarp duomenų bazės, metodo ir atvirkštinės predikatų transformacijos galime pavaizduoti taip:



Paveikslėlis 7 Atvirkštinės predikatų transformacijos veikimo diagrama

Atvirkštinė predikatų transformacija irgi yra korektiška.

Teorema 6 Tegu B_{old} yra duomenų bazės egzempliorius, m yra metodas, o B_{new} — duomenų bazė, gauta iš B_{old} , įvykdžius metodą m. Taip pat tegu φ yra formulė, o σ — formulės φ laisvų kintamųjų inicializacija. Tuomet

$$B_{old\ \sigma} \models \bar{m}(\varphi) \Rightarrow B_{new\ \sigma} \models \varphi$$

arba — predikatų transformacija $\bar{m}$ yra korektiška.

▶

Korektiškumą įrodinėsime pagal atvirkštinės predikatų transformacijos apibrėžimą — naudodami indukciją pagal formulės ir pagal metodo sudėtį.

- sąlygai (1) įrodymas trivialus.
- sąlygoje (2) tarkime, jog φ ir ψ yra formulės ir, pagal indukcijos hipotezę, tarkime jog predikatų transformacija korektiška šioms formulėms.

  $B_{old\ \sigma} \models \bar{m}(\varphi) \Rightarrow B_{new\ \sigma} \models \varphi$ ir

  $B_{old\ \sigma} \models \bar{m}(\psi) \Rightarrow B_{new\ \sigma} \models \psi$

  Tuomet $(B_{old\ \sigma} \models \bar{m}(\varphi) \wedge \bar{m}(\psi)) \Rightarrow (B_{new\ \sigma} \models \varphi \wedge \psi)$. O tai reiškia, jog predikatų transformacija yra korektiška.
- įrodymas (3) sąlygai visiškai panašus.
- sąlygoje (4) tarkime, jog φ yra formulė, kuriai predikatų transformacija yra korektiška (indukcijos hipotezė).

  Tada imame $B_{old\ \sigma} \models$ `forall x:` $\bar{m}(\varphi(x))$, kuris reiškia, jog kiekvienam x $B_{old\ \sigma} \models \bar{m}(\varphi(x))$.

  Pagal indukcijos hipotezę, kiekvienam x $B_{new\ \sigma} \models \varphi(x)$.

  Bet tai reiškia, jog $B_{new\ \sigma} \models$ `forall x:` $\varphi(x)$ — predikatų transformacija yra korektiška.
- sąlygai (5) įrodymas analogiškas.
- sąlyga (6) yra indukcijos bazė. Tegu $m \equiv u.a := v$, o $\varphi \equiv x.a = y$. Jeigu $B_{old\ \sigma} \models (u = x \wedge v = y) \vee (u \neq x \wedge x.a = y)$, tai:

turime $B_{old\,\sigma} \models (u = x \wedge v = y)$, ir tada, po m vykdymo, $B_{new\,\sigma} \models u.a = v$, o tai reiškia ir $B_{new\,\sigma} \models x.a = y$

arba turime $B_{old\,\sigma} \models (u \neq x \wedge x.a = y)$, ir tuomet gauname $B_{new\,\sigma} \models x.a = y$, nes m nekeičia x atributų.

Taigi, turime, jog $B_{new\,\sigma} \models x.a = y$ bet kuriuo atveju.

Kiti du atvejai (kur keičiasi φ sandara) yra analogiški.

- sąlyga (7) — tarkime, jog $m \equiv i_1 ; i_2$. pagal indukcijos prielaidą imame, kad predikatų transformacija korektiška abiem instrukcijoms i_1 ir i_2 .pažymėkime B_1 bazę B_{old} po instrukcijos i_1 įvykdymo. Tuomet, remdamiesi indukcijos hipoteze, turime, kad

  $B_{old\,\sigma} \models \overleftarrow{i_1}(\overleftarrow{i_2}(\varphi)) \Rightarrow B_{1\,\sigma} \models \overleftarrow{i_2}(\varphi)$

  $B_{1\,\sigma} \models \overleftarrow{i_2}(\varphi) \Rightarrow B_{new\,\sigma} \models \varphi$

Taigi, predikatų transformacija korektiška ir šiuo atveju.

- sąlyga (8) įrodoma trivialiai, o sąlygos (9) įrodymas analogiškas (6) atvejui.

- paskutinė sąlyga (10). Indukcijos hipotezė yra, jog instrukcijai i teorema teisinga. T.y. $B_{old\,\sigma} \models \overleftarrow{i}(\varphi) \Rightarrow B_{new\,\sigma} \models \varphi$. Tuomet:

  $B_{old\,\sigma} \models (\texttt{exists}\ v : \psi(v)) \wedge \overleftarrow{i}(\varphi)$. Vadinasi, instrukcija i yra vykdoma. Tada B_{new} yra instrukcijos i vykdymo bazėje B_{old} rezultatas. O tuomet iš indukcinės hipotezės gauname $B_{new\,\sigma} \models \varphi$.

◀

Kartu su šia nauja predikatų transformacija, mes gauname naują būdą įrodyti metodų saugumą:

Teorema 7 Tegu C yra apribojimas, o m — metodas. Tuomet, jeigu $C \Rightarrow \bar{m}(C)$, tai m nepažeidžia apribojimo C.

▶

m yra saugus apribojimo C atžvilgiu, jeigu, bet kuriuo metu įvykdžius jį duomenų bazėje B_{old}, tenkinančioje C, rezultate gausime duomenų bazę B_{new}, irgi tenkinančią C .Taigi, pagal prielaidą mes turime $B_{old} \models C$. Pagal teoremos prielaidą $C \Rightarrow \bar{m}(C)$. Vadinasi, $B_{old} \models \bar{m}(C)$. Iš **teoremos 6** gauname, jog $B_{new} \models C$ ir metodas m išlaiko apribojimą C.

◀

Prieš teoremą paminėjome, jog atvirkštinė predikatų transformacija suteikia naują galimybę įrodyti metodų saugumą. Tai tiesa. Bet dar daugiau — ši metodų transformacija suteikia ir galimybę automatiškai *pataisyti* metodus taip, kad jie pasidarytų saugūs integralumo apribojimų atžvilgiu.

Idėja paprasta — jeigu pagal metodo rezultatui keliamas sąlygas galime surasti pradines sąlygas, kurias turi tenkinti metodo argumentai, tai tas pradines sąlygas galime įdėti į metodą. Truputį formaliau, viskas atrodytų taip:

tarkime, turime metodą m

```
method m(parametrai) in class K
    kūnas
```

O dabar tarkime, jog turime integralumo apribojimą C, kurį metodas m privalo išlaikyti. Tuomet automatiškai galime jį pataisyti į naują metodą:

```
method mC(parametrai) in class K
  { if m̄(C)
      kūnas
  }
```

Teorema 8 Tegu C yra apribojimas, o m — metodas. Tuomet metodas m_C nepažeidžia apribojimo C.

▶

Įrodymas visiškai paprastas. Turime tik dvi galimybes:

- $B_{old} \models \bar{m}(C)$. Tuomet metodo m_C kūnas yra vykdomas ir, pagal **teoremą 6** $B_{new} \models C$.
- $B_{old} \models \bar{m}(C)$. Tada metodo kūnas ir nevykdomas: $B_{old} = B_{new}$. O kadangi $B_{old} \models C$, tai ir $B_{new} \models C$.

◀

Tačiau šitoks automatinis metodų taisymas susijęs su kai kuriomis problemomis. Visų pirma: galime aptikti kai kuriuos saugius metodus esant "nesaugiais" (dėl įrodymo ilgio apribojimo). Tada metodo originalus kūnas bus sudarkytas bereikalingais testais. Ir ši problema yra neišvengiama.

Antra problema susijusi su tuo, kad predikatų transformacijos visuomet "pagamina" formules, kurios yra gerokai didesnės (sudėtingesnės) už transformuotąsias formules. Todėl sistemiškas metodų koregavimas gali užimti nepagrįstai daug laiko. Priešingai nuo pirmos problemos, antrosios sprendimui dar galime pasitelkti įvairias optimizavimo technikas ir pan.

ODBS architektūra

Aprašyta pagal [Del97].

Ir pagaliau šiame, jau paskutiniame skyriuje aprašysime realių ODB valdymo sistemų naudojamas architektūras. Kadangi tai pakankamai techninis mūsų nagrinėjamos srities aspektas. Tai labai giliai į jį ir nesileisim. Tačiau be bendrų "realios" architektūros sprendimų pristatymo, mūsų nuomone, kursas būtų nevisiškai pilnas...

Visų pirma, paminėkime pagrindinius ODBS rinkos dalyvius. Tai:

Gemstone , Servio Corp., JAV

Ontos , Ontos Inc., JAV

ObjectStore , Object Design Inc., JAV

Objectivity , Objectivity Inc., JAV

Versant , Versant Object Technology Corp., JAV

O_2 , O2 Technology, Prancūzija

Orion , Itasca (uniSQL), JAV

Kaip matote, absoliuti kompanijų dauguma yra amerikietiškos kilmės, nors europiečių indėlis vystant ODB koncepciją irgi yra tikrai nemažas.

O dabar pereikime prie reikalo:

I. Serveris

Išskirkime šias pagrindines DB serverio funkcijas:

- realizuoti duomenų, jų saugojimo, įvardijimo ir paieškos modelius;
- pateikti efektyvius priėjimo prie objektų diske ar atmintyje metodus: indeksavimas, asociatyvus priėjimas[11], objektų klasterizavimas[12] ir t.t.
- valdyti priėjimo konkurenciją;
- pateikti duomenų apsaugą: autorizacija, integralumas, DB atstatymas;

[11] Asociatyvus priėjimas realizuojamas užduodant tam tikrą bitų šabloną. Esant paieškai, serveris taiko šį šabloną binariniam duomenų pavidalui. Ir tik kai randamas kažkoks atitikimas su šablonu — imamasi ieškoti būtent to, ko reikia.

[12] Dažnai serveris siunčia klientui duomenų kiekius, didesnius negu objektas (pvz. disko „puslapius"). Galima tarti, jog dažnai objektai turi nuorodas/asociacijas į kitus objektus, į kuriuos klientas taip pat tikėtina kad kreipsis. Todėl tarpusavy susiję objektai klasterizuojami taip, kad kuo daugiau jų patektų į vieną disko puslapį.

- atminties valdymas;
- sistemos adaptyvumas vystymuisi: keičiantis DB schemai, versijai ir pan.

Primename keturis modelius, kuriuos paminėjome aprašydami OQL ir kurių realizavimą įtraukėme prie serverio funkcijų:

- *duomenų modelis: ką aprašyti ir kaip manipuliuoti ?*
- *saugojimo modelis: ką išsaugoti ?*
- *įvardijimo modelis: kaip visa tai vadinti ?*
- *paieškos modelis: kaip ieškoti ir išgauti ?*

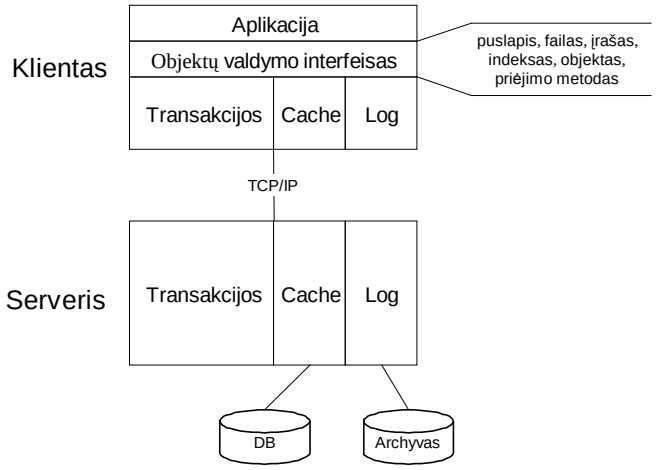
SQL'e viename yra visi keturi modeliai. O OQL tėra užklausų kalba.

Apibendrintai galime pavaizduoti tokią architektūrą:

Aplikacija	• programavimo kalbos lygis • atminties valdymas • katalogavimas (schemos palaikymas) • tipizavimas
Objektų valdymas	• indeksavimas • failų ir atminties puslapių valdymas • duomenų atnaujinimo (*update*) vykdymas • konkurencijos valdymas • duomenų atstatymas • resursų rezervavimas

Paveikslėlis 8 Apibendrinta ODBS architektūra

Žvelgiant į viską pro klient—serverinę prizmę, ši schema išsilieja į tokią:



Paveikslėlis 9 Klient—serverinė ODBS architektūra (pagal Claude Delobel)

Būtent ties šia architektūra mes ir sustokime truputį ilgiau. Esant šiuolaikinėms technikos ir technologijos vystymosi tendencijoms, vis daugiau skaičiavimo galios perkeliama į darbo vietą arba, kitaip tariant — klientą. Be to, ryšio su serveriu pralaidumas irgi auga. Ryšys darosi vis mažesnė problema. Tačiau kreipimasis į diską vis dar lieka kritiniu sistemos veikimo faktoriumi. Nors diskų talpa auga labai greitai, jų darbo greitis mažai evoliucionuoja. Todėl bet kokio serverio darbo pagreitinimas dažnai išsilieja į bandymą minimizuoti kreipimųsi į diską skaičių.

ODB serverių atveju išskiriami trys variantai:

- puslapių serveris,
- objektų serveris,
- failų serveris.

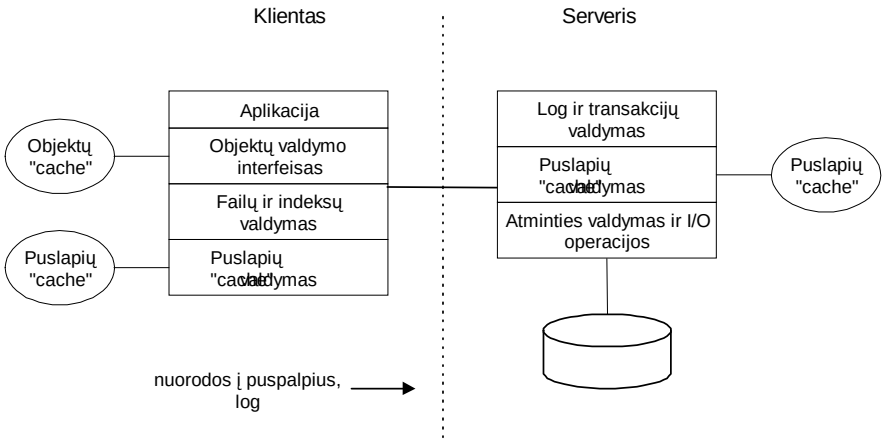
I.A Puslapių serveris

Puslapių serveris moka manipuliuoti tik diskinės atminties puslapiais. Jeigu reikia klientui pasiųsti kokį nors objektą — jis siunčia puslapį, kuriame yra tas prašomasis objektas. Teisingiau — pats klientas prašo iš serverio ne objekto, o puslapio su objektu. Patį objektą iš puslapio susiranda ir nusiskaito pats klientas.

Schematiškai puslapių serveris atrodo taip, kaip pavaizduota Paveikslėlyje 10.

Serveris sudaro savo nuskaitytų puslapių buferį arba "*cache*". Jeigu klientas iš jo prašo puslapio, kuris jau yra tame buferyje — antrą kartą skaityti iš disko nebereikia.

Savo ruožtu, klientas irgi turi savo puslapių buferį — į jį dedami puslapiai, gauti iš serverio. Todėl, prieš prašant puslapio, visuomet pasitikrinama, ar to puslapio nėra "čia pat" esančiame buferyje.



Paveikslėlis 10 Puslapių serveris (pagal Claude Delobel)

Na ir, galų gale, trečias buferis skirtas objektams, išgautiems iš puslapio, saugoti. Todėl bet kuris objekto pareikalavimas gali iššaukti trijų buferių peržiūrėjimą: iš pradžių žiūrima, ar to objekto nėra objektų buferyje, vėliau — ar objekto nėra iš serverio gautuose puslapiuose. Jeigu ne — serveris prašomas atsiųsti reikiamą puslapį. Šis, savo ruožtu, irgi pasitikrina savame buferyje, ar reikalaujamo puslapio nėra jo puslapių buferyje ir tik tuomet (!) skaito reikiamą puslapį iš disko, siunčia jį klientui, kuris iš to puslapio pasiima norimą objektą.

Tokio serverio privalumai yra:

- sumažinamas serverio apkrovimas ir supaprastinama jo architektūra,
- geriau išnaudojami objektų pergrupavimo mechanizmo teikiami privalumai.

Tačiau egzistuoja ir trūkumai:

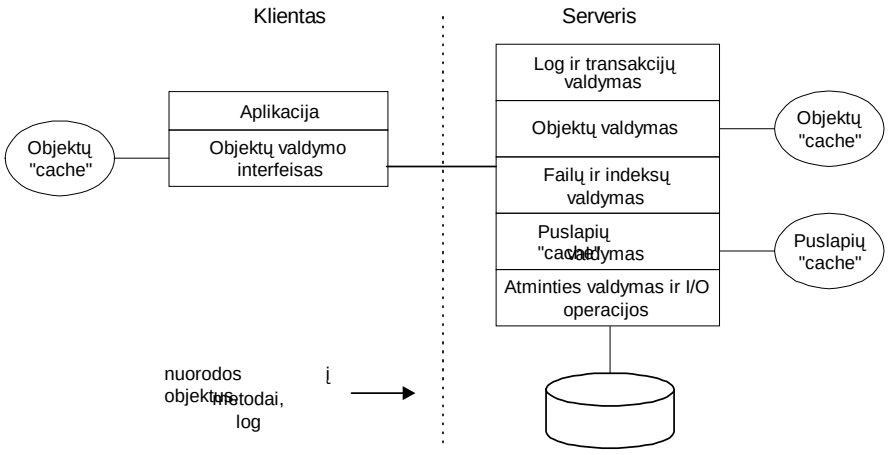
- negalima pakankamai nufiltruoti duomenų, kai serveris žino objektų struktūrą (vis vien juk siunčiamas visas puslapis),
- problemos bandant pakeisti skirtingus objektus, esančius tame pačiame puslapyje (kaip išspręsti duomenų prieinamumo konkurenciją ?).

Kitas serverio variantas yra

I.B Objektų serveris

Šis serveris klientui teikia atskirus objektus, nepriklausomai nuo to, kokiuose puslapiuose jie yra.

Turime tokią objektų serverio schemą:



Paveikslėlis 11 Objektų serveris (pagal Claude Delobel)

Čia klientas turi tik vieną buferį — objektams. O štai serveryje vėl yra du buferiai. Puslapių buferis reikalingas todėl, kad iš disko informacija vis vien skaitoma puslapiais — tokia diskų technologija. Vadinasi, reikia išnaudoti jos ypatumus.

Toks serveris turi kelis privalumus:

- metodus galima vykdyti ir serveryje,
- domenų prieinamumo konkurenciją galima valdyti objektų lygyje.

Tačiau iškart iškyla ir tam tikros problemos:

- objektų serveris yra daug sudėtingesnis,
- jeigu turime didelį objektą, užimantį kelis puslapius, kontroliuoti jo viso atsiuntimą yra sąlyginai sunkiau, negu užtikrinti, kad buvo atsiųstas visas puslapis,
- pagrindinis duomenų persiuntimo laikas yra sugaištamas siuntimo inicializavimui. Todėl praktiškai nėra skirtumo: siųsti objektą, ar siųsti visą puslapį. Vadinasi mažų objektų persiuntinėjimas yra sąlyginai brangesnis už didelių puslapių siuntimą.

I.C Objektų serveris

Šio tipo serverio architektūra yra tokia: klientas bendrauja su failų serveriu (pvz. Sun NFS), kuris užtikrina puslaių skaitymą/rašymą. O duomenų priėjimo konkurencijai bei duomenų atstatymui valdyti naudojamas atskiras ODBS servisas.

Tokia architektūra supaprastina serverio relaizavimą, tračiau dvejų servisų tarpusavio bendravimo sudėtingumas ir yra didžiausias tokios architektūros trūkumas.

Nėra tikslių kriterijų, leidžiančių pasakyti, kuris serverio tipas yra geresnis. Kaip matėte, abu jie yra vienodai geri ir vienodai blogi. Todėl kokio nors serverio kokybės negalime vertinti vien žinodami jo tipą. Viskas priklauso nuo to tipo realizavimo bei uždavinio, kuriam serveris taikomas, specifikos.

O dabar persikelkime į kiek labiau specializuotą, bet ganėtinai įdomią sritį — objektų valdymą objektinėse duomenų bazių valdymo sistemose.

II. Objektų valdymas

Visų pirma nustatykime, ką turėtų "žinoti" ar "mokėti" objektų valdymo modulis (dalis) serveryje. Šį žinojimą galima grubiai suskirstyti į dvi dalis:

- žinojimas apie modelį (—ius): duomenų, saugojimo, įvardijimo ir paieškos modeliai;
- žinojimas apie programavimo kalbą: kaip susieti programavimo kalbos instrukcijas su DB operacijomis, kaip realizuoti pranešimo pasiuntimą objektui, ir pan.

Žinojimas pagal lygį gali būti skiriamas į silpno, vidutinio ir aukšto lygio žinojimus. Trumpai apie kiekvieną:

II.A Silpno lygio žinojimas

Objektų valdyme pažįstamos tik kintamo dydžio baitų eilutės, turinčios savo identifikatorius. Šitoks "požiūris" turi kelis pliusus:

- tokį objektų valdymą galima taikyti įvairiems duomenų modeliams (esant nedideliems modifikavimams — netgi reliaciniams !);
- silpno žinojimo lygio objektų valdymas labai paprastai realizuojamas.

Tačiau esama ir nemažų minusų:

- duomenų interpretavimas gali būti daromas tik aplikacijos lygyje;
- asociatyvaus priėjimo ir navigacijos operacijos negali būti atliekamos serveryje.

II.B Vidutinio lygio žinojimas

Šiuo atveju objektų valdyme žinoma objektų struktūra ir saugojimo modelis. Tokio lygio žinojimą palaiko *O₂* ir *Orion* ODB valdymo sistemos.

Kaip ir galima laukti, didesnis žinojimo lygis suteikia daugiau privalumų:

- serveryje galima naviguoti objektų atributais;
- galima serveryje paskaičiuoti predikatų, naudojančių objektų atributus, reikšmes;
- galimas užklausų optimizavimas.

Tačiau vis dar lieka problema: negalima vykdyti metodų serveryje; į kliento mašiną atsiunčiami duomenys (atributų reikšmės) ir metodai vykdomi būtent kliento dalyje.

II.C Aukšto lygio žinojimas

Aukšto lygio žinojimas užtikrina dar didesnį serverio lankstumą. Šis žinojimo lygis reiškia, jog serveris žino apie objektus viską: ir jų struktūrą, ir taikomus metodus, ir saugojimą. Todėl serveryje galima vykdyti metodus. Be to — serveris gali pagerinti priėjimo prie duomenų konkurencijos valdymą, aptikdamas komutatyvias operacijas.

Komutatyvūs veiksmai yra tokie veiksmai su objektais, kurie gali būti atliekami vienu metu. Paprastai, jeigu vienas metodas keičia objekto atributų reikšmes, tai kitiems metodams tas objektas būna neprieinamas. Tačiau juk, pavyzdžiui, dukart pridėti prie atributo skaičių visiškai nekenkia niekam. Vadinasi, šiuos du metodus galima vykdyti vienu metu.

Nepaisant tokių akivaizdžių privalumų, nedaugelis ODBS turi tokį objektų valdymą. Vienu iš pavyzdžių gali būti *GemStone* . Priežastis gan paprasta: realizuoti aukšto žinojimo lygio objektų valdymą yra labai sudėtinga ir toks serveris gali būti pakankamai griozdiškas.

III. Objektų adresavimas

Duomenų bazę galime laikyti dideliu grafu, kurio viršūnės yra objektai, o briaunos — nuorodos tarp objektų duomenų bazėje. Čia kyla klausimas — kaip tos nuorodos realizuojamos ? Jeigu visi objektai yra diske, tai atsakymas paprastas — objektas adresuojamas kokiu nors adresu diske. Tačiau padėtis gerokai pakinta, jeigu operuojame objektu, kuris yra atmintyje. Kaip susieti objektą atmintyje su objektu diske ? O jeigu objektas iš atminties perkeliamas diskan — ar keičiasi tas nustatytas ryšys ?

Šiam klausimui išspręsti siūlomi trys būdai:

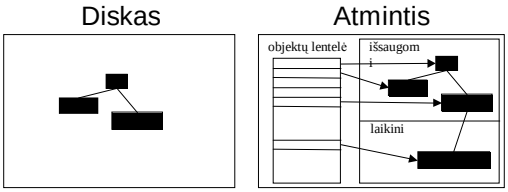
- į diską orientuotas sprendimas: netgi atmintyje objektai adresuojami adresu diske;
- du adresavimo lygiai: atmintyje esančių objektų adresai (ir visi ryšiai į juos) iš adreso diske verčiami adresu atmintyje;
- vienas adresavimo lygis: naudojama virtuali atmintis, kuri "taikoma" tiek diske esantiems objektams, tiek perkeliant juos į atmintį — atmintin perkeliama virtualios atminties dalis, stengiantis objektą atmintyje patalpinti tuo pačiu virtualiu adresu, kuriuo jis buvo diske.

Dabar — šiek tiek plačiau apie kiekvieną iš tų sprendimų.

III.A Adresavimas, orientuotas į diską

Šiuo adresavimo metodu naudojasi O_2 , *Objectivity*, *GemStone*, *Versant* .

"Bendrą vaizdą" galima įsivaizduoti taip:



Paveikslėlis 12 Objektų adresavimas, orientuotas į diską (pagal Claude Delobel)

Tiek atmintyje, tiek diske esantys objektai vienas kitą nurodo pagal adresą diske.

Kiekvieną kartą, kai objektas talpinamas iš disko į atmintį, daromas atitinkamas įrašas objektų lentelėje.

Objektų lentelė turi du laukus: **Oid** ir **Adresas diske**. Jeigu objekto oid nėra lentelėje — reiškia jo nėra ir atmintyje. Kiekvienas objektas iš atminties grąžinamas į diską pagal adresą, esantį objektų lentelėje.

Jeigu atmintyje sukuriamas objektas, kuris turi būti išsaugotas, tai atliekami trys veiksmai:

- išskiriama vieta diske (ir gaunamas objekto adresas diske),
- suteikiamas objektui oid,
- padaromas atitinkamas įrašas objektų lentelėje.

Toks objektų realizavimas turi savus privalumus:

- kontroliuojamas objektų talpinimas diske,
- atmintis ribojama tik virtualiai,
- nereikia daryti jokių adresavimo konversijų, keliant objektus iš disko į atmintį ir atvirkščiai.

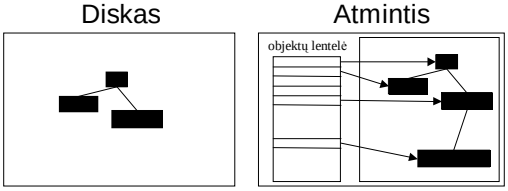
Tačiau yra ir kai kurie trūkumai:

- jeigu atminties formatas skiriasi nuo disko formato, perkeliant objektą reikia visuomet daryti jo kopiją,
- objektų susiejimas visuomet netiesioginis; net jeigu du objektai yra atmintyje, juos susiejant visuomet reikia perbėgti objektų lentelę, norint gauti "tikrą" objekto adresą diske.

III.B Dviejų lygių adresavimas

Tokį adresavimą realizuoja *Orion* .

Ši adresavimo galimybė gali būti pavaizduota taip:



Paveikslėlis 13 Dviejų lygių objektų adresavimas (pagal Claude Delobel)

Čia objektai, esantys diske, adresuojami adresu diske, o, perkelti į atmintį, jau adresuojami adresu atmintyje. Vadinasi, reikia susieti objekto adresą atmintyje su jo adresu diske. Todėl objektų lentelė dabar turi tokius du laukus: **Adresas diske** ir **Adresas atmintyje**. Pagal šią lentelę visuomet galima nustatyti, į kurį objektą, esantį atmintyje, "rodo" objektas, esantis diske. Tačiau į kitą pusę ryšys komplikuotesnis. Mat objektai diske susisieja vienas su kitu pagal adresą diske. Atmintyje esantys objektai siejami pagal adresą atmintyje. Būtent taip yra traktuojama nuoroda į kitą objektą iš atmintyje esančio objekto. Todėl, perkeliant objektą į atmintį, reikia kartu perkelti ir visus objektus, į kuriuos rodo "keliamasis" objektas. Priešingu atveju nebus galima užpildyti keliamajame objekte esančių nuorodų, kadangi nebus žinoma rodomųjų objektų vieta atmintyje. Iš pirmo žvilgsnio toks "papildomas" perkėlimas gali atrodyti tik kaip nedidelis sunkumas. Tačiau reikia įvertinti galimą "grandininę reakciją", nes kartu su vienu objektu reikia kelti jo rodomus objektus, su jais — jų rodomus ir t.t. Iš to galima išsisukti darant tam tikrą "optimizavimą": reikalingus rodomus objektus atkelti į atmintį tik tuomet, kai į juos kreipiamasi. Tada "pirminis" perkėlimas yra greitas ir, kas svarbiausia, gana griežtai apibrėžtas. Bet už tai pralaimimas sistemos veikimo greitis, mat kiekvienas kreipimasis į objekto atributą/nuorodą gali būti susijęs su naujo objekto paieška diske ir jo perkėlimu į atmintį.

Vėl gi susumuokime dviejų lygių adresavimo privalumus ir trūkumus. Privalumai:

- programavimo kalbai visi objektai yra identiški, nes visi jie adresuojami adresu atmintyje,
- atmintis ribojama tik virtualiai,
- objektams, jau esantiems atmintyje, nuorodų pakeitimas yra greitas, nes yra tiesioginis,
- išlieka tam tikra objektų esančių diske, kontrolė.

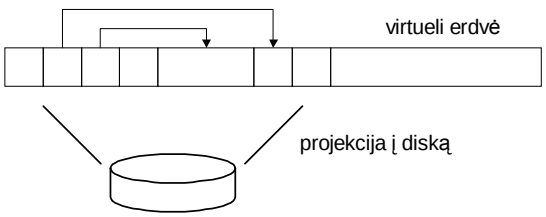
O trūkumai yra tokie:

- brangiai atsieinantis objektų kėlimas iš disko į atmintį (apie ką jau buvo kalbėta plačiau),
- jeigu atminties formatas skiriasi nuo disko formato, perkeliant objektą vis dar reikia daryti jo kopiją,

III.C Vieno lygio adresavimas

Tokiu adresavimu naudojasi *ObjectStore* .

Vieno lygio adresavimo idėja yra sukurti virtualią "atminties erdvę", kuri susiprojektuotų tiek į diską, tiek į atmintį. Keliant objektą (ar kelis objektus) iš disko į atmintį, perkeliama visa virtualios atminties erdvės dalis, stengiantis patalpinti objektus atmintyje santykinai ten pat (virtualios adresų erdvės prasme), kur jie buvo talpinti diske. Taip bus išlaikomas vieningas objektų adresavimas, nepriklausomai nuo jų buvimo vietos.



Paveikslėlis 14 Vieno lygio objektų adresavimas (pagal Claude Delobel)

Visų pirma, esant tokiai adresavimo sistemai, iškyla problema — kaip atskirti laikinus objektus nuo tų, kurie turi išlikti (išlieka). Vienas iš sprendimų būtų padalinti virtualią erdvę į dvi dalis, kurių viena skirta laikiniems, o kita — "pastoviems" objektams.

Antra problema glūdi sukurtos virtualios erdvės tvarkyme. Jeigu išmetame kai kuriuos objektus — kaip sutvarkyti atmintį taip, kad neliktų joje "skylių"? Kalbant labiau techniškais terminais — turime klasikinę fragmentacijos problemą.

Esant vieno lygio adresavimo sistemai, kai kas yra laimima:

- visi objektai yra vienodi, nepriklausomai nuo jų buvimo vietos; užrašomi jie vienodu formatu.
- greitas nuorodų į kitus objektus pakeitimas,

Tačiau esama nemažai ir trūkumų:

- pasunkėja duomenų bazės atstatymo uždavinys, nes, užrašius objektą į virtualią atmintį, nebeužfiksuojamas momentas, kada jis buvo (ir — ar buvo) tikrai perkeltas į diską,
- sunkiai galima kontroliuoti objektų vietą diske (tai labai svarbu darant objektų pergrupavimu paremtas optimizacijas),
- objektų formatas, arba teisingiau — virtualios erdvės formatas, daugiau atsižvelgia į tam tikros kalbos kompiliatoriaus reikalavimus; todėl atsiranda sunkumai, norint palaikyti keletą manipuliavimo duomenimis kalbų.

Štai tiek trumpai apie ODBS architektūrą, nesistengiant gilintis į smulkmenas, bet, vis dėlto, bandant bent kiek perteikti tą atmosferą, skonį, problemas, susijusius su ODB serverių realizavimu. Juk geram specialistui visada būna pravartu žinoti, kokie principai ir galimos problemos slypi po tuo magišku ir dažnai mistiškai abstrakčiu (abstrakčiai mistišku) duomenų bazių serverio terminu.

Pabaiga

Tuom ir baigsime trumpą Objektinių Duomenų Bazių kursą. Nenorėčiau dėti galutinio taško ar konkretaus patarimo — rinkitės Objektines duomenų bazes. Visos duomenų bazės geros tiek, kiek jos atitinka jos naudotojų poreikius. Jūs, gerbiami skaitytojai, patys geriau žinote, ko reikia Jums. O jeigu žinote tai — belieka išsirinkti geriausią alternatyvą. Jeigu šis kursas pagilins žinias ir leis geriau pasirinkti, tai ir ačiū Aukščiausiajam...

Esu labai dėkingas žmonėms, išmokiusiems mane to, ką moku ir žinau dabar. Ypač Veronique Benzaken — už jos simpatiškumą, Giuseppe Castagna — už jo guvumą ir, žinoma, Claude Delobel'iui — už kantrybę.

Visada žavėjausi savo kolegomis, iš kurių mokiausi kantrybės ir gerumo — Romualdas Kašuba, Viktoras Golubevas, Balys Šulmanas, Evaldas Drąsutis ir daug kitų. Ačiū ir jiems.

Taip pat noriu padėkoti ir tiems žmonėms Lietuvoje, kurie parėmė tokio kurso idėją ir savo patarimais bei pastabomis padėjo man jį paruošti.

Viskas...

Paveikslėlių rodyklė

Literatūra

[ABDDMZ] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik "The Object—Oriented Database System Manifesto".

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. "Foundations of Database Systems", chapter 21, pages 542—578. Adison Wesley, 1995

[Banc96] F. Bancilhon "The ODMG Standard: The Object Model", in "Object Curent", June 1996. Sigs Publications Inc.

[Banc97] F. Bancilhon "OQL, the ODMG Query language" , Ardent Software, 1997

[Benz97] V. Benzaken "Modeles orientés—objet", 1997

[Barr97] D. K. Barry "ODMG 2.0: An Overview. Release 2.0 adds Java binding and more", 1997.
Galima rasti WWW adresu:
`http://www.ddj.com/articles/1997/9717/9717a/9717a.htm`.

[Barr98] D. K. Barry "There's an ODMG database in Your Future", 1998
Galima rasti WWW adresu `http://www.odmg.org`, skyrelyje `Articles`.

[BC97] V. Benzaken, G. Castagna "Langages de Programation pour bases de Données: Notes de cours du DEA Théorie et Ingénierie des Bases de Données.", 1997

[BD93] V. Benzaken, A. Doucet "Thémis: a database programming language with integrity constraints". In Shasha Beeri, Ohori, editor, *Proceedings of the 4th International Workshop on Database Programming Languages*, Workshop in Computing, pages 243—262, New York City, USA, September 1993. Springer—Verlag.

[BDK92] F. Bancilhon, C. Delobel, P. Kanellakis, editors. "Building an Object—Oriented Database System: the Story of O_2" . Morgan Kaufman, San Mateo, California, 1992.

[BS97] V. Benzaken, X. Schaefer "Forward and Backward analysis of object—oriented database programming languages: an application to static integrity management", 1997

[Cas95] G. Castagna "Covariance and Contravariance: Conflict without a Cause", *ACM Transactions on Programing languages and Systems* . 17(3): 431—447, 1995

[DD95] H. Darwen, C.J. Date "The Third manifesto", 1995.

[Del9495] C. Delobel "DEA 94/95"

[Del97] C. Delobel "Architecture des Systémes a Objets" 1997

[Feg97] L. Fegaras "OQL Algebra", 1997
Galima rasti WWW adresu:
http://ranger.uta.edu/~fegaras/optimizer/oql—algebra.html.

[Lief97] H. Liefke "O_2 OQL", 1997
Galima rasti WWW adresu:
http://www.cis.upenn.edu/~cse330/o2tut/oql.html

[Mcl97] Steve McClure "Object database vs. Object—Relational Databases", IDC Bulletin #14821E, 1997

[MRL98] G. McFarland, A. Rudmik, D. Lange "Object—Oriented Database Management Systems Revisted", 1998. Prepared for Air Force Research Laboratory — Information Directorate (AFRL/IF)

[ODMGBE97] "ODMG Book Extract", 1997
Galima rasti WWW adresu:
http://www.odmg.org/standard/odmgbookextract.html

[OQL99] "OQL Sample Grammar for Object Data Management Group (ODMG)", Micro Data Base Systems, 1999

[Sec97] "Object Definition Language", 1997
Galima rasti WWW adresu:
http://utopia.secant.com/2tierdocs/ug/ch3.htm