KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Povilas Jurna

Aspektinis objektinis duomenų bazių modelis pilno kliento sistemoms

Magistro darbas

Darbo vadovas
doc. B. Paradauskas

Kaunas, 2006

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Povilas Jurna

Aspektinis objektinis duomenų bazių modelis pilno kliento sistemoms

Magistro darbas

Kalbos konsultantė
Lietuvių k. katedros lekt.
J. Mikelionienė

Vadovas
doc. B. Paradauskas

Recenzentas

Atliko
IFM-02 gr. stud.
Povilas Jurna

Kaunas, 2006

ASPECT ORIENTED OBJECT DATABASE MODEL FOR RICH CLIENT APPLICATIONS

SUMMARY

A big variety of new modern programming technologies exist in today's market and each of it provide different approaches for the same problems. It is quite a challenge for a project manager or a system architect to decide which technology is best for their project and a lot of time should be spent for analysis before some decisions could be made. The main purpose of this work is to create a reusable model for JAVA applications that is based on cutting edge technologies such as Aspect-Oriented Programming, Object databases and Model-View-Controller architecture. This work provides research data that could be used for analysing what influence will these new technologies have for the system.

Created model is based on aspect oriented programming. The key component is a TransactionalAspect which does automatic database session and transaction management. It also provides session pooling for better reliability and performance and thread safety by using ThreadLocal for more complex applications. As a result a model was created that helps to manage 4 main stages of system development processes: project analysis precess, architecture process, coding process and system support process. Helps to develop a quality system on time and save project expenses at the same time.

Turinys

1. Įvadas....6
2. Duomenų saugojimo problemos bei sprendimo būdai objektiniame programavime 7
2.1. Nagrinėjamas objektas....7
2.2. Pilnas klientas....7
2.3. ORM ir objektinės duomenų bazės....8
2.3.1. Problema....8
2.3.2. ORM....10
2.3.3. Objektinės duomenų bazės....12
2.3.4. Išvados....13
2.4. AOP....14
2.5. Vartotojo sąsajos platforma....16
2.5.1. MVC....16
2.5.2. SWING....18
2.5.3. SWT/JFace....18
2.6. Išvados....19
3. Aspektinis objektinis duomenų bazių modelis....20
3.1. Kuriamas modelis....20
3.2. DepthCounter....23
3.3. SessionPool....24
3.4. TransactionManager....26
3.5. ThreadLocal....28
3.6. DAO....29
3.7. TransactionalAspekt....30
3.8. Contoller dalis....32
3.9. Rezultatas....33
4. Modelio veikimo charakteristikų ir išplečiamumo galimybių tyrimas....34
4.1. Modelio teisingumo tyrimas....35
4.1.1. Paprastas metodo iškvietimas....35
4.1.2. Rekursinis metodo iškvietimas....36
4.1.3. Dviejų metodų sinchroniškas iškvietimas....36
4.1.4. Dviejų metodų iškvietimas iš skirtingų programos gijų....37
4.1.5. Modelio patikimumo tyrimas....38
4.2. Modelio resursų sunaudojimo tyrimas....40
4.2.1. Vidutinio metodo iškvietimo greičio tyrimas....41
4.2.2. Vykdymų laiko priklausomybė nuo vykdymų skaičiaus bei rekursijos gylio..43
4.2.3. Tyrimo išvados....47
4.3. Modelio išplečiamumo galimybės....48
4.3.1. Vartotojo teisių valdymas....48
4.3.2. Vartotojo funkcijų veikimo valdymas....49
4.3.3. Programos versijų funkcionalumo valdymas....50
4.4. Išvados....50
5. Modelio egzistuojančios realizacijos....51
6. Išvados....52
7. Literatūra....54
8. SAntrauka anglų kalba....55
9. Terminų ir santrumpų žodynas....56
10. Priedai....57
10.1. Eksperimente naudoti įrankiai bei programos....57
10.2. Tyrimo bei eksperimentų pilna informacija....57
10.2.1. Iškvietimų skaičiaus trukmės tyrimo rezultatai....57

10.2.2. Iškvietimų trukmės su rekursija tyrimo rezultatai.. 58
10.2.3. Laiko pasiskirstymo tarp komponentų tyrimo rezultatai............................... 58
10.2.4. Gijų skaičiaus įtaka ThreadLocal veikimui... 60
10.3. Pavyzdinė modelio realizacija.. 60

1. ĮVADAS

Šiuo metu programavimo rinkoje egzistuoja gausi įvairovė komponentų bei technologijų siūlančių vienokius ar kitokius technologinius sprendimus. Kuriant naują projektą iškyla problema kokias technologijas pasirinkti, kaip jas tinkamai ir efektyviai realizuoti. Taip pat yra sunku įvertinti tam tikrų technologijų būsimą įtaką kuriamam projektui neturint praktinės patirties, naudojant šias naujas technologijas. Dėl didelės konkurencijos įmonės, kuriančios sistemas, dažnai taupo projekto išlaidas apribodamos galimų projekto technologinių sprendimų analizę arba jos visai neatlieka ir naudoja senus, laiko patikrintus, bet ne visuomet pačius geriausius sprendimus. Kita vertus, naudoti naujausias technologijas yra rizikinga, nes jos ne visada yra pakankamai gerai išbandytos ir sunku prognozuoti šių technologijų panaudojimo pasekmes. Kita problema yra dažnai besikeičianti kuriamo projekto reikalavimų sritis. Dažnai reikalavimai keičiasi projekto kūrimo, netgi baigto projekto stadijoje. Neretai projektai vėluoja ar žlunga dėl besikeičiančių ar nepakankamai išanalizuotų reikalavimų. Sistemą būtina projektuoti taip, kad ji būtų lanksti ir lengvai modifikuojama priklausomai nuo besikeičiančios aplinkos.

Šio darbo tikslas yra bandyti išspręsti minėtas problemas. Pagrindinis tikslas yra sukurti pakartotiniam panaudojimui skirtą modelį, naudojantį naujausias technologijas, išnaudojant jų teikiamas galimybes ir palengvinant programinės įrangos tiek kūrimo, tiek priežiūros procesą. Taip pat pateikti išsamius tyrimų rezultatus bei išvadas kokią įtaką sukurtas modelis turės kuriamam projektui. Turint tokią informaciją galima tiksliai nuspręsti ar šis modelis yra tinkamas konkrečiu programinės įrangos kūrimo atvejui.

Šis modelis yra skirtas Java programavimo kalbai, tačiau gali būti pritaikomas ir .NET platformai. Šiam modeliui sukurti panaudotos tokios pagrindinės technologijos, kaip objektinis programavimas, aspektinis programavimas, objektinio tipo duomenų bazės, taip pat kiti smulkesni architektūriniai bei technologiniai sprendimai. Tikslas yra sukurti kuo labiau objektinį modelį bei sumažinti programos kodo pasikartojimą kuriamoje sistemoje. Taip pat šis modelis yra orientuotas į pilno kliento sistemas. Pilnas klientas yra ne visiem žinoma sąvoka, tačiau ją apžvelgsime analitinėje dalyje bei išsiaiškinsime kada tokio tipo sistemos turėtų būti kuriamos. Visos panaudotos technologijos privalo būti nemokamos.

2. DUOMENŲ SAUGOJIMO PROBLEMOS BEI SPRENDIMO BŪDAI OBJEKTINIAME PROGRAMAVIME

2.1. Nagrinėjamas objektas

Magistro praktinio darbo metu, kuriant sistemą buvo stengiamasi panaudoti kiek įmano daugiau naujausių technologijų. Buvo surinkta didelis informacijos kiekis bei patirtis, kuri gali būti panaudojama analogiškos programinės įrangos pakartotinio panaudojimo modelio kūrimui. Šio darbo galutinis rezultatas turi būti aspektinis objektinis duomenų bazių modelis pilno kliento programoms. Prieš apibendrinant turimą informaciją bei sukuriant modelį, reikalinga išsiaiškinti 4 pagrindinius kūrimo aspektus: kokiu principu bus saugomi duomenys, kokį programavimo modelį pasirinkti (OOP, AOP), kokią vartotojo sąsajos platformą pasirinkti ir kokį architektūrinį modelį pasirinkti. Išnagrinėsime kiekvieną iš šių aspektų detaliau. Taip pat apibrėšime kas yra pilnas klientas ir kada jį yra pravartu naudoti.

2.2. Pilnas klientas

Šiuo metu rinkoje vartotojo sąsajos sistemos išsiskyrė į dvi šakas. Spartėjant internetui, sistemos vis labiau kuriamos taip, kad išnaudoti šią interneto plėtrą. Prieš interneto atsiradimą bei išpopuliarėjimą buhalterinio, administracinio ir panašaus tipo programos buvo kuriamos taip, kad visa sistemos logika būtų atliekama kliento programinėje įrangoje. Serveris atlikdavo tik duomenų bazės vaidmenį. Atsiradus internetui, bei jam spartėjant, taip pat greitėjant kompiuterinei įrangai, serveriams, pradėtos kurti sistemos, kurių visa logika atliekama serveryje. Kliento programinė įranga atlieka tik duomenų atvaizdavimą. Paprastai kliento programine įranga yra laikoma interneto naršyklė. Toks modelis labai išpopuliarėjo dėl savo paprastumo galutinio vartotojo atžvilgiu. Taip pat tokios sistemos pasižymi lengva priežiūra bei tobulinimu, kadangi užtenka modifikuoti tik serverio dalį [1]. Tokio tipo sistemos vadinamos nepilno kliento sistemomis, kadangi pagrindinė sistemos logika vyksta centriniame sistemos taške. Sistemos, kurių visa logika vykdoma kliento pusėje vadinamos pilno kliento programomis. Nepaisant minėtų nepilno kliento programų privalumų jos turi šiuos pagrindinius trūkumus [2]:

- didelis serverio apkrovimas. Didėjant sistemoms bei jų poreikiams nepaprastai apkraunamas tiek serveris tiek kompiuterinis tinklas. Tokiu atveju daug logiškiau kurti pilno kliento sistemas, tam kad sumažinti informacijos srautą į serverį;
- vartotojo sąsajos funkcionalumo apribojimas. Nepaisant visų pastangų kurios

yra dedamos tam kad padidinti internetinių puslapių vartotojo sąsajos funkcionalumo galimybes, jos vis dar stipriai atsilieka nuo paprastų vartotojo sąsajos programų;

- vartotojo sąsajos greitis. Jis yra vienas iš pagrindinių trūkumų nepilno kliento sistemose. Paprastai kiekviena operacija generuoja po užklausą pagrindiniam serveriui.

Apibendrinant galima pasakyti, kad pilno kliento sistemas yra geriau naudoti, kuomet reikalinga kurti specifinę programinę įrangą. Tai paprastai yra profesinei sričiai skirtos programos. Tokios sistemos yra daug greitesnės ir paprastai patogesnės naudoti. Šiame darbe kuriamas modeli yra skirtas pilno kliento programoms kurti. Pagrindinė problema su kuria susiduriama kuriant pilno kliento duomenų bazių sistemas yra sunkumai tvarkingai valdyti duomenų bazių sesijas bei sandorius. Skirtingai nuo nepilno kliento sistemų, čia paprastai veikia asinchroniniai įvykiai, sandoriai gali būti kuriami ir patvirtinami bet kuriuo programos veikimo momentu, todėl sunku sesijos bei sandorių valdymo dalį išskirti į vieną nepriklausomą komponentą. Tokioje sistemoje sunku išvengti kodo, skirto sesijos valdymui, pasikartojimo. Dėl šios priežasties buvo panaudotas aspektinis programavimas, leidžiantis atskirti šia dalį nuo bendros sistemos logikos ir sutaupyti kodo pasikartojimą.

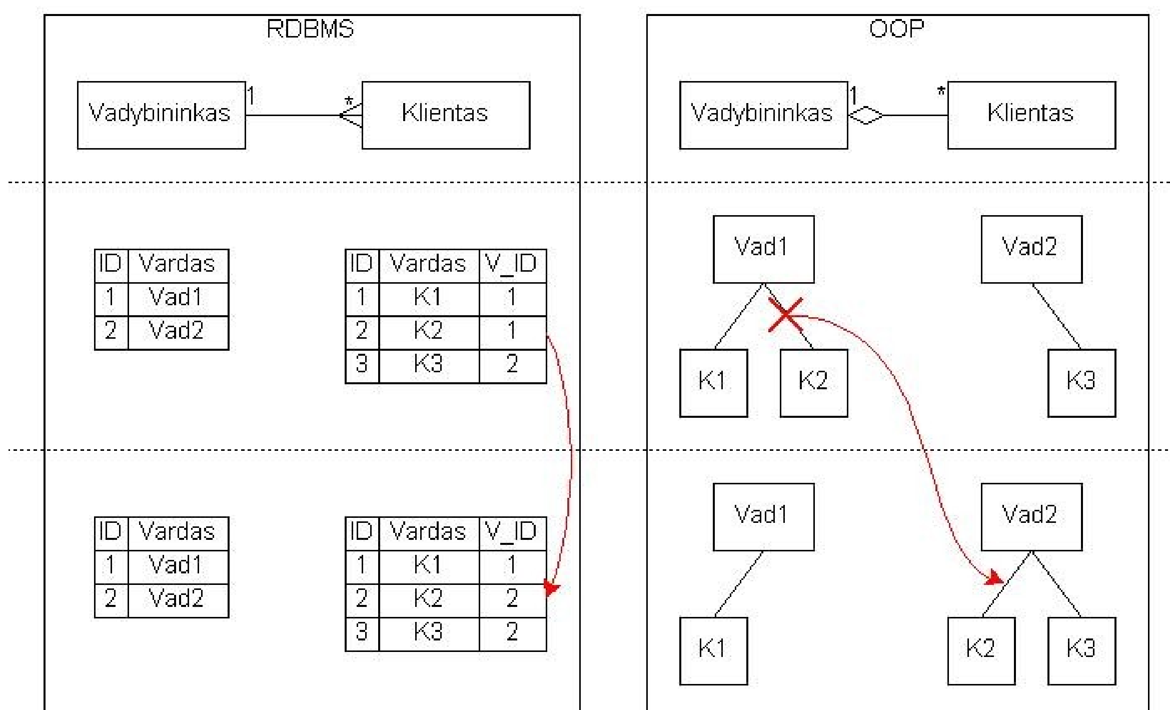
2.3. ORM ir objektinės duomenų bazės

Vienas iš pagrindinių šios skyriaus tikslų išsiaiškinti esminius skirtumus tarp RDBMS, ORM ir objektinių duomenų bazių, taip pat išsiaiškinti pagrindinius jų privalumus bei trūkumus. Žinant šią informaciją galėsime tiksliai apibrėžtį į kokias sistemas mūsų kuriamas modelis bus orientuotas ir kada šio modelio geriau nenaudoti.

2.3.1. Problema

Objektiniame programavime (OOP), pagrindinis principas yra kurti objektus, kurie atitiktų realiame pasaulyje egzistuojančius objektus. Vienas iš iššūkių yra kaip šių objektų informaciją transformuoti į tokį formatą, kad jį būtų patogu saugoti failuose ar duomenų bazėse, taip pat šią informaciją paversti atgal į objektinį modelį. Vienas iš saugojimo būdų – sąryšinių duomenų bazių valdymo sistemos (RDBMS) arba tiesiog sąryšinės duomenų bazės, kurių pirmos versijos atsirado apie 1970 metus [4]. Nors pirma, pilnai OOP orientuota programavimo kalba „Smalltalk“ išleista panašiu metu (1970-1972) [3], tačiau OOP ir RDBMS turėjo esminius skirtumus kurie visuomet apsunkindavo šių dviejų modelių suderinamumą.

Vienas iš skirtumų tarp OOP ir RDBMS yra duomenų priklausomybių kryptis. Tarkim, kad pagal OOP modelyje turime tėvą, kuris turi sąrašą savo vaikų, tai RDBMS modelyje šis ryšys yra iš kitos pusės, vaikai saugo informaciją apie tai, kas yra jų tėvas, o tėvas šios informacijos neturi. Pavyzdžiui, turime vadybininkus, kurie gali turėti sąrašą klientų (1 pav.) . Vadybininkui Vad1 priklauso klientai K1 ir K2, o vadybinikui Vad2 priklauso K3. Objektiniame modelyje, norėdami klientą K2 perkelti iš Vad1 į Vad2, turėtume jį išimti iš Vad1 klientų sąrašo ir įrašyti į Vad2 sąrašą, t.y., modifikuojami vadybininkų objektai. Tuo tarpu RDBMS atlikdami tą patį veiksmą, turėtume kliento lentelėje pakeisti vadybininko ID, kuriam jis priklauso, t.y., modifikuojamas būtų atvirkščiai – klientas.

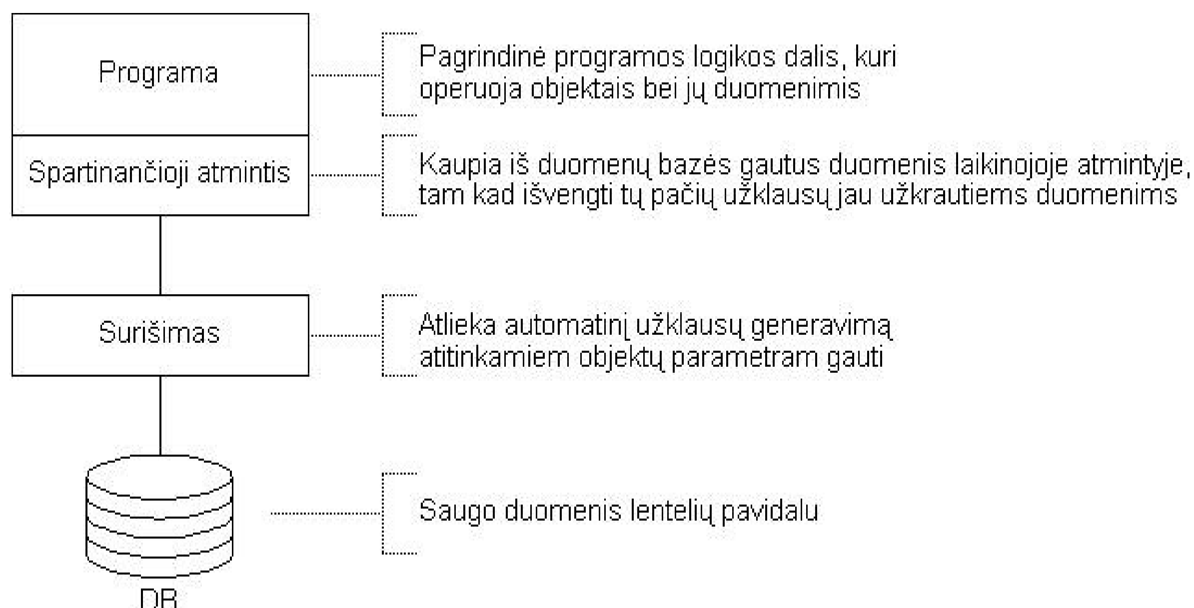


1 pav. RDBMS ir OOP objektų ryšių skirtumai

Kitas skirtumas tarp OOP ir RDBMS yra pati ryšio sąvoka. OOP tiesiogiai saugo ryšį su kiekvienu su juom susijusiu objektu, tuo tarpu RDBMS lentelių įrašai, kuriose saugoma informacija, neturi tiesioginio ryšio su kitom lentelėm. Šie ryšiai sudaromi tik užklausų metu ir priklauso nuo to, kokia užklausa bus parašyta. Dėl šių pagrindinių priežasčių negalima tiesiogiai sujungti OOP ir RDBMS [5]. Yra du sprendimo būdai: naudoti tarpininką, kuris galėtų sujungti šiuos du modelius, arba naudoti alternatyvų duomenų bazių modelį, kuris labiau atitiktų OOP. Tarpininko vaidmenį galėtų atlikti objektinis-sąryšinis sujungimas (ORM), o alternatyvus duomenų bazių modelis būtų objektinės duomenų bazės.

2.3.2. ORM

ORM sistemų tikslas yra išspręsti problemas minėtas 2.2.1 skyriuje. Egzistuojančios bibliotekos automatiškai sujungia RDBMS ir OOP. Šioms bibliotekoms pateikiant duomenų bazės lentelių sąrašą bei programos klasių sąrašą, jos dinamiškai generuoja užklausas informacijai apsikeisti tarp dviejų modelių. Ši informacija gali būti kaupiama spartinančiojoje atmintyje ir vėliau sinchronizuojama su duomenų baze (2 pav.)



2 pav. ORM struktūra

Teoriškai programuotojas duomenų bazę mato kaip objektinį modelį ir gali operuoti su šia objektų aibe, dėl to šių objektų informacija automatiškai keisis duomenų bazėje. Praktikoje ne visada pavyksta sklandžiai atlikti pilnai automatinį duomenų sujungimą. Naudojant ORM ne visada pavyksta visiškai išvengti tiesioginio „kontakto“ su duomenų baze, atsiranda būtinybė rašyti tiesiogines užklausas, tam kad sugeneruoti tam tikras ataskaitas. Taip pat atsiranda tokių situacijų, kada parašius paprastą užklausą sutaupoma daug sistemos resursų. Taigi ne visada naudojant ORM pavyksta visiškai pilnai padengti duomenų bazės operacijas objektiškom išraiškom. Kita problema yra duomenų sinchronizavimas tarp objektų ir duomenų bazės lentelių. Naudojant ORM, programuotojas visada turi galvoti, kaip duomenys bus sinchronizuojami ir ar nebus sinchronizacijos konfliktų. Taip pat vienas iš trūkumų gali būti neoptimalios duomenų užklausos, kurias ORM biblioteka generuoja norint gauti duomenis konkrečiam objektui.

Palyginus greičio skirtumus tarp RDBMS ir ORM, galima pažymėti, kad RDBMS pasižymi didesniu greičiu, tada kada reikia vykdyti dideles užklausas į duomenų bazę ir operuoti su dideliais duomenų kiekiais vienu metu. ORM ir objektinis modelis pasižymi didesniu greičiu tada, kada reikalinga daug mažų užklausų ir užklausos operuoja mažais duomenų kiekiais. Šis greičio pranašumas atsiranda dėl duomenų kaupimo spartinančiojoje atmintyje. Duomenys per užklausas užkraunami į objektus ir kol duomenų bazės sandoris nesibaigia, duomenys skaitomi tiesiogiai iš objektų. Taip sutaupomas realių užklausų į duomenų bazę skaičius, tuo pačiu perkeliant dalį resursų sunaudojimo iš duomenų bazės serverio į kliento kompiuterį [5].

Apibendrinus minėtas problemas galima išskirti pagrindinius ORM privalumus:

- leidžia sąryšinį modelį paversti į objektinį,
- pagerina programos palaikomumą,
- padaro sistemą nepriklausomą nuo konkrečios sąryšinės duomenų bazės,
- dėl spartinančiosios atminties, pagreitina nedidelių duomenų kiekių apsikeitimą su duomenų baze.

Galima nurodyti ir šiuos trūkumus:

- nepraktiškas, kai dirbama su statistiniais duomenimis ir ataskaitomis,
- programuotojas, kurdamas sistema, turi pakankamai dažnai galvoti, kaip duomenys apsikeis tarp objektų spartinančiosios atminties ir duomenų bazės, nes visada yra galimybė sudaryti konfliktines situacijas, ypač pilno kliento programose, kuomet vienu metu gali būti vykdomi keli duomenų bazių sandoriai ar vykdomos kelios programos gijos.

Atsižvelgiant į privalumus bei trūkumas galima nustatyti, kada pravartu naudoti ORM modelį:

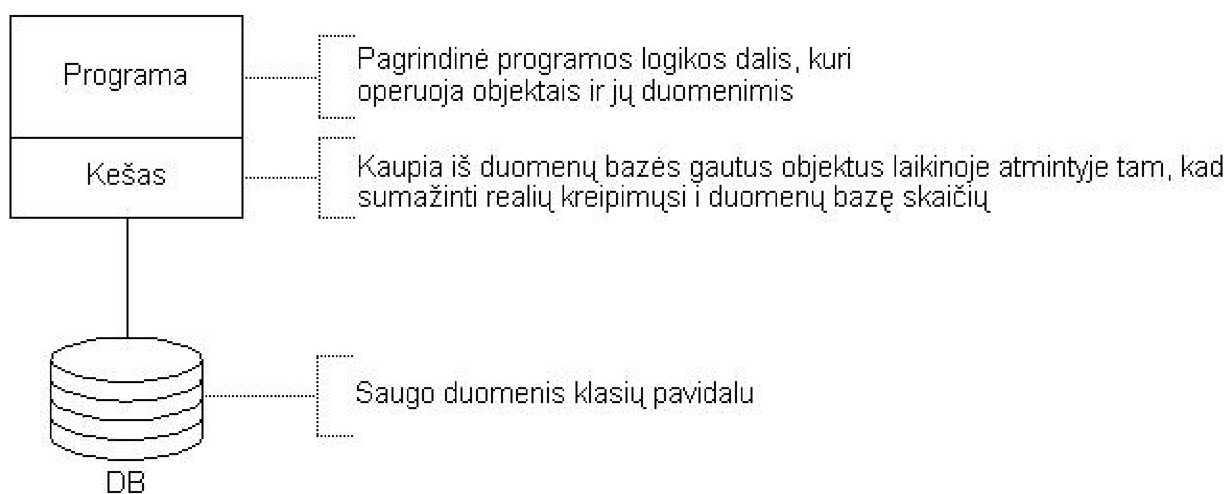
- kai kuriama programa yra skirta operuoti su duomenimis atitinkančiais realaus pasaulio objektus.
- kai yra būtinybė naudoti sąryšinę duomenų bazę, tačiau norima išlaikyti objektinį programavimo modelį,
- dėl techninių ar politinių priežasčių negalima panaudoti objektinės duomenų bazės.

2.3.3. Objektinės duomenų bazės

Viena iš alternatyvų ORM yra Objektinės duomenų bazės (ODBMS). Šios duomenų bazės yra specialiai orientuotos į OOP modelį ir jose visi duomenys traktuojami kaip objektai. Šios duomenų bazės paprastai neturi RDBMS būdingų lentelių ir įprastos SQL kalbos. Vietoj SQL, naudojama OQL, kuri savo principu kiek primena SQL, tačiau nėra taip gerai ištobulinta kaip SQL. Objektinės duomenų bazės pasižymi šiomis savybėm [6]:

- duomenys saugomi objektais,
- duomenys kaupiami laikinoje spartinančiojoje atmintyje, tam kad sumažinti užklausų į duomenų bazę skaičių,
- turi OQL kalbą, kuria galima generuoti objektinio pobūdžio užklausas.

Jei palyginti ORM struktūrą (2 pav.) su ODBMS, ji atrodyti taip (3 pav.):



3 pav. ODBMS struktūra

Kaip matome čia nėra ORM būdingos sujungimo dalies. Tai nulemia šiuos privalumus ORM atžvilgiu:

- didesnis efektyvumas greičio atžvilgiu, kadangi nereikia sujungti objektų su duomenų bazės lentelėmis,
- objektinės duomenų bazės panaudojimas yra daug paprastesnis, kadangi nereikia galvoti kaip duomenys bus surišami,

- nereikia kurti sąryšinės duomenų bazės modelio, paprastai užtenka sukurti norimą klasių diagramą, kuri bus saugoma duomenų bazėje.

2.3.4. Išvados

Atlikus analizę, galima teigti, kad objektinės duomenų bazės daug geriau pritaikomos objektiniam programavimui nei ORM, tačiau ORM paprastai suteikia tiesioginį priėjimą prie duomenų bazės, kas leidžia panaudoti SQL užklausas didelėms ataskaitoms generuoti. ORM gali būti panaudojamas kada yra būtinybė turėti sąryšinę duomenų bazę, tačiau norima išlaikyti objektinį duomenų bazės modelį. Atsižvelgiant į minėtas savybes galima nubrėžti schemą, kurioje matoma, kad kuo programa yra labiau orientuota į objektus ir operacijas su jais, tuo labiau verta naudoti objektines duomenų bazes ir atvirkščiai, kuo programa labiau orientuota į statistinės informacijos rinkimą ir raportų generavimą, tuo labiau verta naudoti RDBMS (4 pav.).

4 pav. Sąryšinio ir objektinio tipo panaudojamumas

Pagal tai galima apsibrėžti, kad šiame darbe kuriamas modelis bus labiau orientuotas į objektinį programavimą ir saugos realiam pasauliui būdingus objektus. Jei sistema yra skirta statistikai rinkti ir ataskaitas generuoti, šis modelis gali būti netinkamas ir tik labiau apsunkins operacijas dėl savo objektinio modelio. Nepaisant to, kiekvienas atvejis yra unikalus ir turi būti individualiai įvertinama, kokias pasekmes turės modelio panaudojimas.

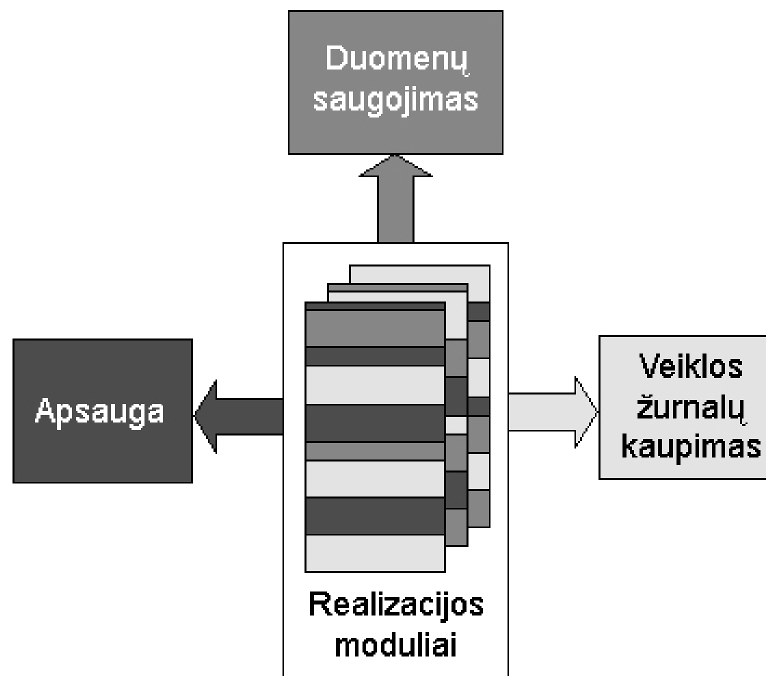
2.4. AOP

Šio skyriaus tikslas – apsibrėžti, kuriose sistemos vietose naudosime AOP ir kaip tai darysime.

Pažvelgus į daugumą sistemų pastebėsime, kad jos dažnai susideda iš tam tikrų loginių dalių, kurios kartojasi skirtinguose programos moduliuose. OOP gali apsunkinti tokių loginių dalių realizavimą, padaro sistemą sunkiau suprantamą ir sunkiau vystomą būtent dėl to, kad ne visada įmanoma šias besikartojančias programos dalis išskirti į atskiras dalis[7]. AOP palengvina tokių besikartojančių dalių išskyrimą į atskirus modulius ir sumažina kodo pasikartojimą sistemoje. Toks programavimo būdas dažnai ne tik pagreitina sistemos kūrimo procesą, bet ir palengvina sistemos architektūros dizainą, kodo realizavimą bei sistemos vystymą, taisymą. Šis metodas taip pat pagerina sistemos kokybę pagal elementarų principą: mažiau kodo – mažiau klaidų. Kadangi pasikartojantys elementai sujungiami į aspektus, programa yra daug lengviau taisoma bei prižiūrima [8].

Kyla klausimas kur ir kada reikėtų naudoti AOP? Iš vienos pusės AOP palengvina sistemos kūrimą, dėl anksčiau minėtų priežasčių, bet iš kitos pusės gali sistemą padaryti painią ir sunkiai suprantamą jei bus naudojamas tam netinkamose vietose. Todėl pagrindinis šio skyriaus tikslas yra nustatyti kada ir kaip geriausia naudoti AOP. Yra 3 pagrindinės AOP panaudojimo sritys [8] (5 pav.):

- apsauga ir vartotojo teisės,
- programos veiklos žurnalų kaipimas,
- duomenų saugojimas.



5 pav. AOP panaudojimo sritys

Atsižvelgdami į šiuos duomenis apsibrėšime, kad AOP bus naudojamas duomenims saugoti. AOP veiklos žurnalams kaipti savo modelyje nenaudosime, kadangi tai gali įpareigoti naudoti tam tikras egzistuojančias veiklos žurnalų kaupimo realizacijas, naudojant šiame darbe kuriamą modelį, kas gali būti nepriimtina daugeliui šio modelio vartotojų. Tai taip tai gali turėti nepageidaujamų pasekmių efektyvumo greičio atžvilgiu, kadangi AOP yra ganėtinai lėtesnis palyginti su grynu OOP. Veiklos žurnalų kaupimo atveju jis būtų naudojamas kone visose programos vietose. Apsauga ir vartotojų teisių valdymas taip pat nebus naudojamas mūsų modelyje, kadangi tai yra kiekvienam projektui specifinė sritis, tačiau apžvelgsime vieną iš variantų kaip AOP ir vartotojo teisių valdymas gali būti panaudojamas kartu su kuriamu modeliu.

Apsauga gali būti skirstoma į 2 pagrindines dalis:

- funkcijų kontrolę skirtingiems vartotojams,
- duomenų kontrolę skirtingiems vartotojams.

Šiam funkcionalumui realizuoti gali būti panaudotas AOP. Funkcijų kontrolės skirtingiems vartotojams valdymą galima realizuoti sukuriant aspektą, kuris tikrina pagrindines vartotojo sąsajos funkcijas. Kiekvienam vartotojo tipui sudaromas leistinų funkcijų sąrašas. Prieš kviečiant tam tikrą funkciją, aspektas patikrina ar kviečiama funkcija

yra nurodyta vartotojo funkcijų sąraše ar ne. Jei nurodyta – funkcija vykdoma. Priešingu atveju funkcija yra ignoruojama ir yra sugeneruojama klaida. Klaidos pranešimas priklauso nuo konkrečios sistemos realizacijos, tai gali būti dialogo langas vartotojui arba klaidos užregistravimas programos veiklos žurnale.

Duomenų kontrolė skirtingiems vartotojams būdingas tada, kada norima sukurti tokią sistemą, kurios tie patys programos metodai sugebėtų skirtingiems vartotojams rodyti skirtingą informaciją. Tai gali būti realizuojama aspektų pagalba. Tarkime, kad sistemoje yra sukurtas komponentas DAO, kuriame realizuota visa duomenų apsikeitimo su duomenų baze logika. Galima sukurti tokį aspektą, kursi galėtų tikrinti koks vartotojas yra prisijungęs, ir priklausomai nuo vartotojo tipo, vykdomiems metodams perduoti konkrečiai tam vartotojui skirtą DAO realizaciją. Tokiu būdu galima galima valdyti kokius duomenis kiekvienas vartotojas matys savo programoje, bei kokius duomenis galės modifikuoti.

2.5. Vartotojo sąsajos platforma

Šio skyriaus tikslas išsiaiškinti kokią platformą tinkamiausia naudoti kartu su kuriamu modeliu. Pagrindiniai kriterijai būtų:

- operacinės sistemos nepriklausomumas,
- MVC architektūros palaikymas.

JAVA programavimo kalba turi 2 pagrindinius įrankių rinkinius vartotojo sąsajai kurti: SWING, kuri yra įtraukta į JFC specifikaciją ir SWT/JFace, kuri yra sukurta kompanijos IBM.

Kitas šio skyriaus tikslas yra išsiaiškinti kas yra MVC architektūra ir kodėl ji turėtų būti naudojama.

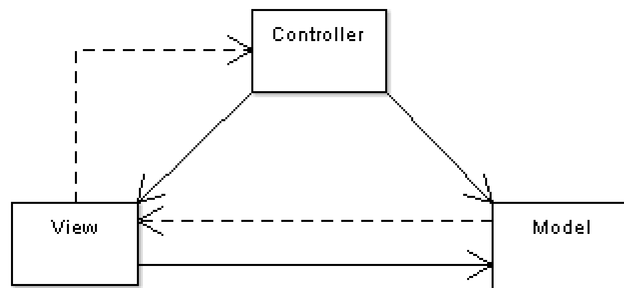
2.5.1. MVC

MVC – tai sistemos architektūra, atskirianti sistemos duomenų modelį, vartotojo sąsają ir duomenis kontroliuojančią logiką į atskirus, aiškiai atskirtus modulius taip, kad vieno modulio pakeitimai turėtų minimalias pasekmes kitam moduliui. MVC architektūra pagrįstą sistemą sudaro šios 3 pagrindinės dalys (6 pav.):

- modelis – jį sudaro duomenų struktūra bei elementari logika šiems duomenims apdoroti, pavyzdžiui, patikrinti ar šiandien vartotojo gimtadienis;
- vaizdas – verčią modelį į tokį formatą, kuriuo jis atvaizduojamas vartotojui,

pavyzdžiui, internetinių puslapių sistemose tai būtų HTML kodo generavimas pagal turimus duomenis.

- valdiklis – apdoroja tokius įvykius, kaip įvairūs vartotojo veiksmai ir atitinkamai modifikuoja modelį. Kitaip tariant sujungia vaizde vykdomus pakeitimus su egzistuojančiu modeliu [9].



6 pav. MVC architektūra

Išanalizuosime kodėl MVC turėtų būti naudojamas. MVC architektūra pasižymi šiomis savybėmis [9]:

- Skirtingų vaizdai tam pačiam modeliui sukūrimo galimybė, galima sukurti skirtingas vartotojo sąsajas naudojant tą patį modelį;
- Lengvesnis palaikomumas naujoms vartotojo sąsajos platformoms, norint pereiti prie kitos platformos ar naujos jos versijos užtenka sukurti naują vaizdo modulį;
- Funkcionalumo kontrolės galimybė. Turint vieną modelį, galima sukurti 2 skirtingus vaizdus bei valdiklius, kurie apimtų tik norimą dalį modelio funkcionalumo.
- Projekto išplečiamumas. Tinkamai sukurtos naujos funkcijos modelio dalyje gali būti naudojamos tiek su sena vaizdo versija (naujos funkcijos nebus palaikomos), tiek su nauja.

Dėl išvardintų priežasčių šis architektūros modelis yra labai parankus kuriamam, kadangi pagrindinis darbo tikslas yra sukurti universalų programų kūrimo modelį, tai reiškia, kad šis modelis turi būti nepriklausomas nuo konkrečios vartotojo sąsajos bei jos realizacijos.

2.5.2. SWING

SWING yra įrankių rinkinys vartotojo sąsajai kurti naudojant Java programavimo kalbą. Skirtingai nei jos pirmtakas AWT, kuris naudoja operacinės sistemos dalis, SWING yra pilnai sukurtas naudojant Java ir visi elementai piešiami naudojant Java. Tai sąlygoja šias teigiamas savybes [11]:

- SWING atrodo vienodai visose operacinėse sistemose,
- elgiasi vienodai visose sistemose, kitaip tariant, sukūrus programą Windows operacinėje sistemoje, gali būti tikras kad vartotojo sąsaja elgsis taip pat ir Linux operacinėje sistemoje.

Pagrindinė bloga savybė yra programų, sukurtų su SWING greitis. Kadangi visi vartotojo sąsajos komponentai tiesiogiai piešiami Java virtualios mašinos, tai stipriai lėtina programą.

SWING yra sukurtas taip, kad būtų suderinamas su MVC architektūra. Taip pat turi RCP platformą, tačiau ji nėra pakankamai gerai ištobulinta ir rinkoje nėra daug sistemų, sukurtų šios RCP platformos pagrindu, todėl būtų rizikinga ją pasirinkti.

2.5.3. SWT/JFace

SWT yra sukurtas „Eclipse Foundation“ specialiai Eclipse IDE. Tai yra alternatyva egzistuojančiam standartiniam SWING įrankių rinkiniui vartotojo sąsajai kurti. JFace yra aukštesnis šio įrankių rinkinio abstrakcijos lygis, kuris palengvina bei pagreitina programavimo procesą, bei suteikia tokį funkcionalumą, kaip MVC architektūros palaikymas. SWT yra sukurtas Java programavimo kalba ir, naudojant Java Native Interface, naudoja skirtingas bibliotekas priklausomai nuo operacinės sistemos kurioje programa dirba. Šios bibliotekos paprastai naudoja operacinės sistemos vartotojo sąsajos komponentus. Tai laikoma pagrindiniu privalumu. Dėl šių priežasčių sistema pasižymi šiomis savybėm [10]:

- programos sukurtos su SWT yra greitesnės nei sukurtos su SWING,
- programa gerai įsilieja į operacinės sistemos vartotojo sąsają, kitaip tariant, skirtingai nuo SWING, sunku pastebėti, kad programa yra sukurta su Java ir atrodo lygiai taip pat kaip ir kitos operacinės sistemos dalys.

SWT/JFace, kaip ir SWING taip pat turi RCP platformą, tačiau ji yra daug geriau ištobulinta nei SWING. Su šia platforma sukurtos tokios sistemos kaip Eclipse IDE, IBM

Rational Application Developer. Įvertinus visas šias savybes bei sulyginus jas su SWING, galima padaryti išvadą, kad SWT/JFace yra pranašesnis nei SWING. Dėl šios priežasties jis ir buvo pasirinktas praktiniame darbe, o taip pat mūsų kuriamas modelis bus labiau orientuotas į šią Eclipse RCP platformą, tačiau galės būti pritaikomas ir SWING.

2.6. Išvados

Apibendrinant apžvelgtus aspektus galima pasakyti, kad buvo pasirinkti šis sprendimai:

- pasirinktos objektinės duomenų bazės duomenims saugoti dėl jų gero suderinamumo su objektiniu programavimu,
- pasirinktas AOP duomenų saugojimui realizuoti,
- pasirinktas MVC architektūrinis modelis vartotojo sąsajai kurti, tam kad būtų galima sukurti duomenų modelį minimaliai priklausomą nuo vartotojo sąsajos implementacijos,
- pasirinktas SWT/JFace bei Eclipse RCP vartotojo sąsajai kurti dėl teigiamų greičio savybių, gero susiliejimo su operacine sistema bei RCP funkcionalumu.

3. ASPEKTINIS OBJEKTINIS DUOMENŲ BAZIŲ MODELIS

Šios dalies tikslas – suprojektuoti aspektinį objektinį duomenų bazių modelį pilno kliento programoms bei detaliai išnagrinėti sukurtos architektūros aspektus. Pradžioje bus apžvelgiamas bendras modelio vaizdas, po to nagrinėjamas kiekvienas komponentas individualiai užduodant ir atsakant į klausimą kam šis komponentas reikalingas, kaip jis turėtų būti realizuojamas ir kodėl. Pabaigoje bus apžvelgiama kokiais metodais šis modelis bus tiriamas, kokie jo aspektai bus tiriami ir apžvelgiami programos kokybės užtikrinimo būdai naudojant šį sukurtą modelį.

3.1. Kuriamas modelis

Pagrindinis mūsų tikslas yra sukurti universalų programų kūrimo modelį, paremtą objektinėmis duomenų bazėmis bei aspektiniu programavimu. Reikalinga sukurti aspektą, skirtą automatiniam duomenų saugojimui duomenų bazėje, jo sujungimą su duomenų baze bei valdikliais MVC architektūriniame modelyje. Analitinėje dalyje puvo nuspręsta naudoti MVC architektūrą, kuri leidžia atskirti konkrečią GUI realizaciją nuo visos duomenų logikos. Šis modelis neturi susieti sistemą su konkrečios GUI realizacijos bei konkrečia duomenų struktūra, todėl mes nagrinėsime tik kontroliuojančiąją MVC dalį – valdiklį.

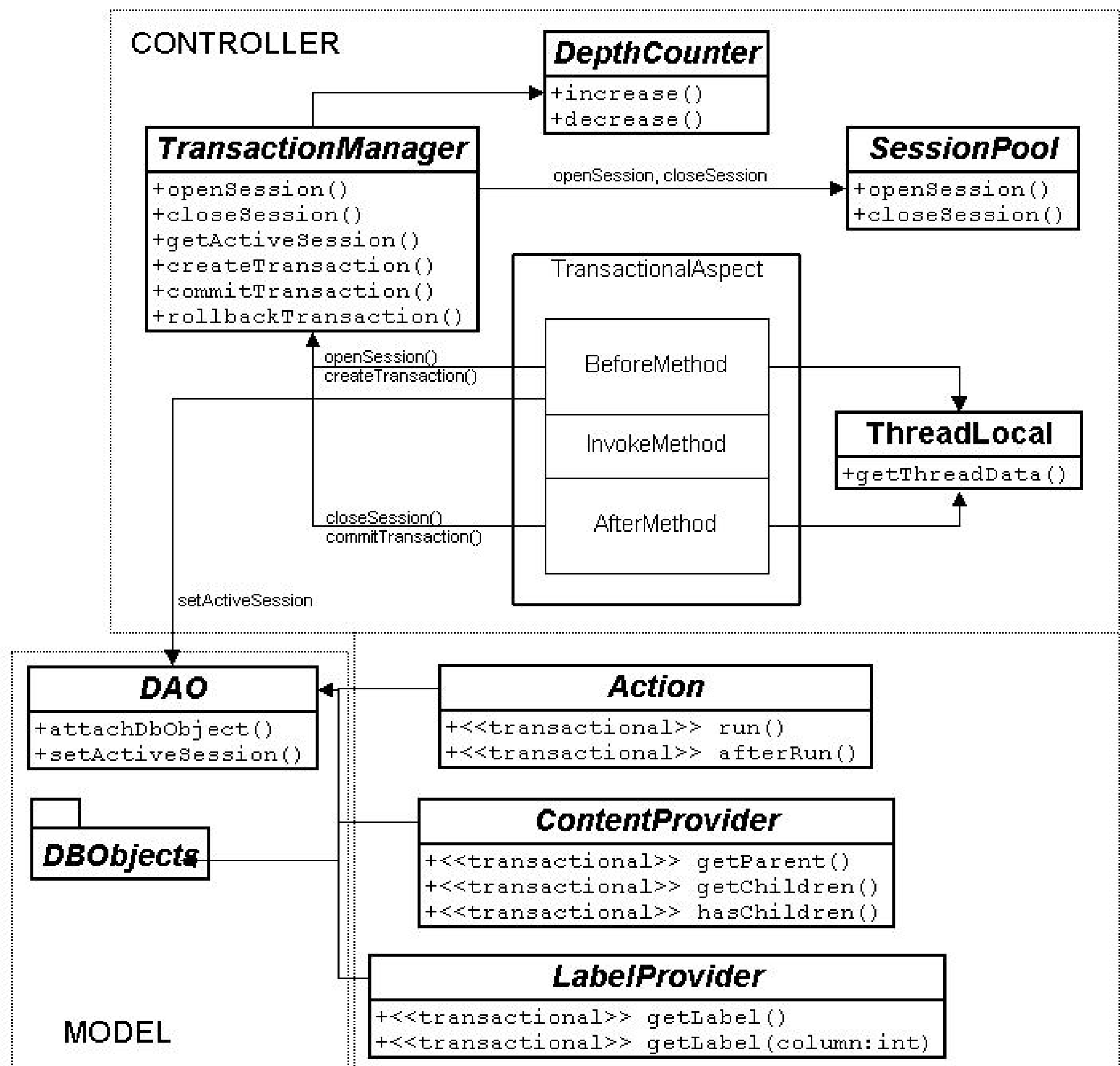
Pirmiausia apžvelgime bendrą modelio vaizdą ir trumpai pažvelgsime kuri dalis už ką atsakinga. Toliau išnagrinėsime kiekvieną dalį individualiai, pateikiant detalias diagramas kaip šios dalys turėtų būti realizuojamos.

Kuriamas modelis suskirstytas į 2 pagrindines dalis: valdiklis ir modelis (7 pav.). Modelio pagrindinis svarbus elementas yra DAO, kuris yra atsakingas už elementarias operacijas su saugomais objektais. Valdiklis yra esminė šio modelio dalis, kuri atlieka automatinį duomenų saugojimą, sesijų bei duomenų bazės sandorių valdymą, bei duomenų sujungimą su vartotojo sąsają. Jį sudaro šie pagrindiniai elementai:

- TransactionalAspect. Jis atsakingas už automatinį sesijų atidarymą, uždarymą, sandorių valdymą, bei klaidų fiksavimą ir pranešimą. Tam jis naudoja papildomus 3 elementus:
    - TransactionManager. Saugo aktyvią sesiją, bei aktyvių sandorį,
    - SessionPool – atsakingas už sesijos gyvavimo laiką,
    - DepthCounter – reikalingas sekti TransactionAspekt rekursijos gyliui,

- Action – atsakingas už vartotojo iškviestų veiksmų apdorojimą,
- ContentProvider – atsakingas už duomenų pateikimą vartotojo sąsajai (tokiems komponentams, kaip List, Combo, Table, Tree)

- LabelProvider – atsakingas už duomenų vertimą į GUI reikalingą formatą,



7 pav. Modelio struktūra

Kiekvieną elementą apžvelgsime detaliau, pateikiant ryšius su kitais objektais, galimas realizacijas bei galimas problemas. Modelį apžvelgsime „iš apačios į viršų" būdu, kitaip tariant, pradžioje apžvelgsime mažus komponentus, pabaigoje – svarbiausius, didžiausius.

3.2. DepthCounter

Tai pats paprasčiausias šio modelio elementas, kuris yra paprasčiausias skaitliukas. Jo pagrindinė užduotis yra sekti ar TransactionalAspekt nėra kviečiamas rekursiškai. Rekursyvumas gali atsirasti tokiu atveju, jei, pavyzdžiui, Action klasė, atliekanti vartotojo sąsajos sužadintus veiksmus, savo viduje kviečia kitą Action klasę. Tokiu atveju, TransactionAspekt, kuris automatiškai, prieš kviečiant pažymėtą metodą atidaro sesiją, bei sukuria duomenų bazės sandorį, sukurtų dvi skirtingus duomenų bazės sandorius bei sesijas šiems metodams. Tai yra nepageidaujamas reiškinys ir gali turėti tokių pasekmių, kaip duomenų praradimas, bei konkurencija tarp dviejų sandorių. DepthCounter seka kiek kartų buvo iškviestas TransactionAspect ir kiek kartų baigė darbą. Jei aptinkama, kad darbą pradeda pirmas aspektas, jis vykdomas, jei antras ar dar vėlesnis, jis yra ignoruojamas. Taip pat su aspekto darbo baigimu. Jei aptinkama kad tai ne paskutinis aspektas, jis ignoruojamas, tačiau jei tai paskutinis aspektas, jis įvykdomas.

Klasę sudaro du reikalingi metodai:

- increase(). Šis metodas padidina skaitliuką per vieną reikšmę ir grąžina naujai gautą reikšmę.
- decrease(). Šis metodas sumažina skaitliuką per vieną reikšmę ir grąžina naują gautą reikšmę

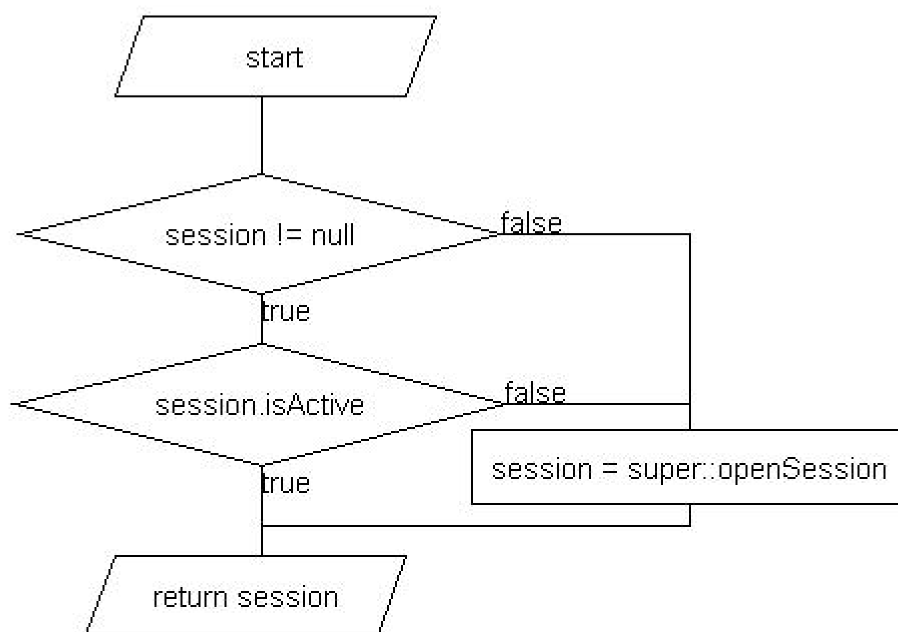
Problema išlieka kai aspektą vienu metu kviečia skirtingos programos gijos (Threads). Čia gali iškilti konkurencijos problemos. Taip pat atskirai gijai turėtų būti sukuriamos dvi skirtingos sesijos. Detaliau apie šią problemą bei jos sprendimo būdą apžvelgsime prie TransctionalAspect.

3.3. SessionPool

SessionPool pagrindinė užduotis yra sesijų su duomenų baze gyvavimo trukmės valdymas. Viena iš problemų pilno kliento programose yra sesijos gyvavimo trukmė.

Paprasčiausias sesijos valdymo būdas yra sesijos sukūrimas programos paleidimo metu, ir jos uždarymas programos sustabdymo metu. Tai yra paprastas, tačiau blogas ir nerekomenduotinas sprendimas, kadangi sesija bet kurio momentu gali nutrūkti ir sistema nustos veikti. Dėl šios priežasties reikėtų praktiškai kiekvienoje programos vietoje tikrinti sesijos būseną, ir jei sesija nebeaktyvi, ją iš naujo atidaryti. Tai labai padidina kodo pasikartojimą ir sulieja duomenų bazės sesijos valdymą su visa duomenų logika.

Kitas valdymo būdas yra sesijų atidarymas prieš vykdant kiekvieną operaciją. Prieš vykdant kokį nors veiksmą ar veiksmų seką su duomenų bazės duomenimis, sesija atidaroma, o veiksmui pasibaigus veiksmams sesija iš karto uždaroma. Naudojant tokį modelį galima praktiškai visuomet garantuoti kad sesija bus aktyvi reikiamu momentu ir šį sesijų valdymą galima išskirti kaip atskirą aspektą. Tai sumažina kodo pasikartojimą. Tačiau šis modelis turi trūkumą – dažnas sesijų atidarymas, uždarymas turi įtakos programos veikimo greičiui, bei duomenų bazės apkrovimui.



8 pav. SessionPool openSession realizacija

Geriausias sprendimas yra panaudoti prieš tai minėtą metodą, tačiau vietoj tikros sesijos naudoti sesijos pakartotinį panaudojimą. Tai gali būti realizuojama „Decorator“ architektūrinio šablono principu. Tarkim turime klasę Session su 2 pagrindiniais metodais:

- openSession
- closeSession.

Tuomet sukuriame savo SessionPool klasę, kuri paveldi tikrąja Session klasę ir pakeičia minėtus metodus. CloseSession metodas nieko neatlieka ir paprasčiausiai ignoruojamas, o openSession veikimas realizuojamas pagal schemą atvaizduotą 8 paveikslėlyje.

Tokiu būdu realizavus duomenų bazės sesijų valdymą užtikrinamas sistemos patikimumas. Nutrūkus ryšiui su sesija, ji bus automatiškai atstatoma. Jei automatiškai sesijos atstatyti nepavyksta, fiksuojama sistemos klaida ir gali būti pranešama vartotojui. Atsiradus ryšiui su duomenų baze, SessionPool objektas automatiškai atnaujins sesiją bet kokio metodo kvietimo metu.

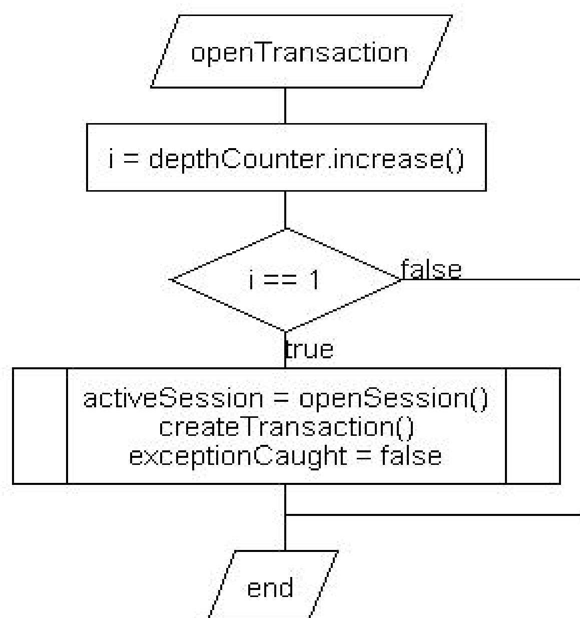
Toks sesijų valdymo realizavimas sumažina programos kodo pasikartojimą. Ryšys su duomenų baze tikrinamas vienoje konkrečioje vietoje. Nereikalinga visuose komponentuose, naudojančiuose sesiją, vykdyti jos aktyvumo tikrinimo. Tokį programos kodą yra daug lengviau prižiūrėti ir modifikuoti.

3.4. TransactionManager

TransactionManager tikslas automatiškai valdyti duomenų bazės sandorių gyvavimo trukmę. Jis privalo nustatyti kada reikalinga atidaryti naują sesiją ir sukurti naują sandorį, kada sandorį baigti ir uždaryti sesiją.

Ši klasė turi 4 pagrindinius metodus:

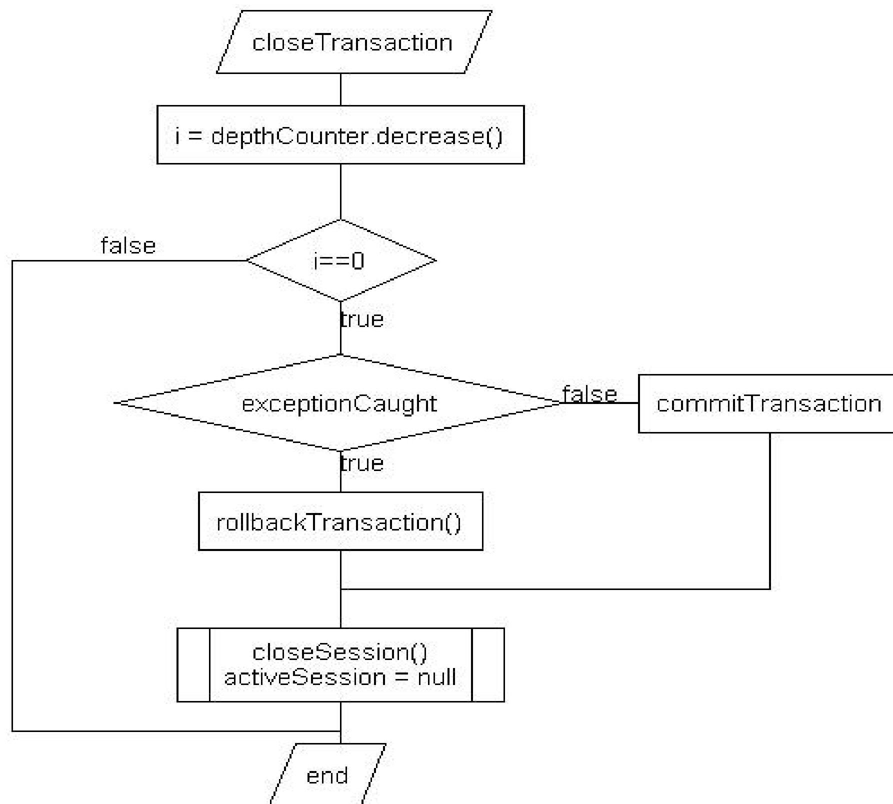
- openTransaction,
- commitTransaction,
- rollbackTransaction,
- getActiveSession.



9 pav. TransactionManager::openTransaction

Ši klasė privalo ne tik valdyti duomenų bazės sandorio gyvavimo trukmę. Ji taip pat turi apsaugoti papildomą sandorių bei sesijų kūrimą rekursijos atveju. Privalo būti sekamas rekursijos gylis ir pagal tai sprendžiama ar reikalinga kurti naują duomenų bazės sandorį ar jį patvirtinti ar ne. Pavyzdžiui, jei sandoris atidaromas pirmą kartą (9 pav.), jis yra realiai sukuriamas. Tačiau jei bandoma jį atidaryti antrą kartą, ši operacija ignoruojama. Sandoris nesukuriamas, o paliekamas galioti senas duomenų bazės sandoris. Sandorio patvirtinimas veikia analogišku principu. Tikrinama kiek kartų buvo iškviestas openTransaction metodas.

Jei buvo iškviestas daugiau nei vieną kartą, commitTransaction metodas realiai nepatvirtina sandorio. Tačiau jei rekursijos gylis siekia tik vieną iškvietimą, duomenų bazės sandoris patvirtinamas ir sesija uždaroma. Kitaip tariant, sandorio sukūrimų skaičius turi atitikti patvirtinimų skaičių (10 pav.).



10 pav. TransactionManager::commitTransaction

3.5. ThreadLocal

Thread local pagrindinis tikslas yra užtikrinti saugų sesijų paskirstymą tarp skirtingų programos gijų.

Dauguma duomenų bazių sesijų yra nesaugios jei naudojamos tiesiogiai iš skirtingų gijų ir gali mesti programos išimtį ar netgi duoti neprognozuojamus rezultatus. Paprastai sistema kuri naudoja duomenų bazių sesijas privalo pasirūpinti kad į šia sesiją nebus kreipiamasi vienu metu iš skirtingų programos gijų. Nepaisant to, visuomet rekomenduojama vengti tokių situacijų, kuomet operacijas su duomenų baze atlieka skirtingos gijos. Kartais tokios situacijos yra neišvengiamos, kuomet programinę įrangą sudaro 2 nepriklausomos dalys, vykdančios tarpusavyje nesusijusias duomenų bazės užklausas. Tokiu atvejų šioms dviems dalims gali būti sukuriamos dvi skirtingos duomenų bazės sesijos ir taip išvengiama konfliktinių situacijų. Būtent tokiai situacijai ir yra skirtas ThreadLocal komponentas.

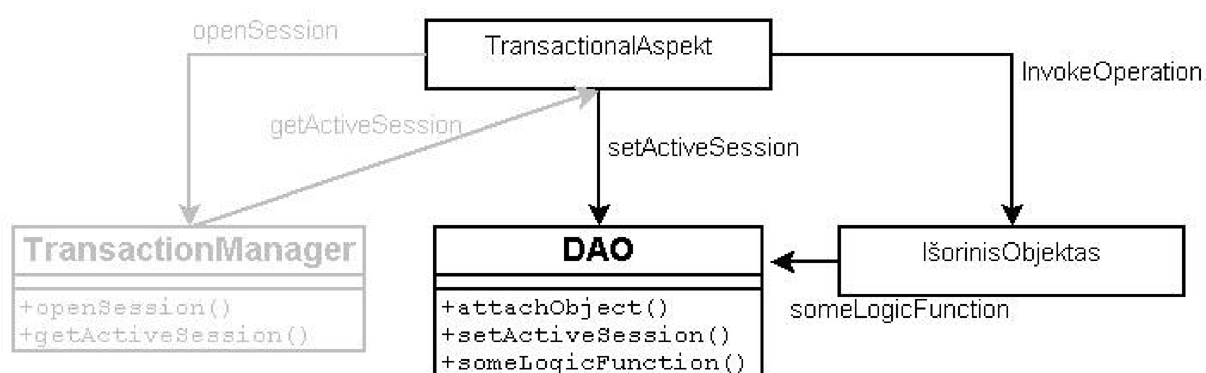
ThreadLocal yra laikina duomenų saugykla, kuri kiekvienai programos gijai saugo būtent jos objektų kopijas ir gijos kreipimosi atveju grąžina būtent jai skirtą objektą, kitaip tariant, gija, pasiimdama objektą iš ThreadLocal klasės visada gaus tik jai skirtą objektą. Tokiu būdu užtikrinama, kad skirtingos gijos niekuomet nesinaudos tais pačiais resursais ir taip išvengiama konfliktinių situacijų.

Konkrečiu atveju ThreadLocal bus naudojamas tik TransactionManager bei DAO realizacijai saugoti. Kiekviena gija TreadLocal dėka turės po sau atskirą TransactionManager objektą bei DAO realizaciją. Tokiu būdų atskirai bus sekama kiekvienos gijos tranzakcijų atidarymų/uždarymų rekursija (3.4 skyrius), bei bus naudojamos skirtingos sesijos į duomenų bazę.

ThreadLocal naudojimo reikia vengti ir reikia stengtis gijų sinchronizavimą išspręsti aukštesniame lygyje. Paprastai pilno kliento programose užtenka vienos pagrindinės gijos duomenų bazės veiksmams atlikti. Jei sukuriama nauja gija ir iš jos tiesiogiai kreipiamasi į duomenų bazę, reikia turėti omenyje, kad šios gijos tranzakcija nebus susinchronizuota su kitos gijos tranzakcija. Šios dvi gijos nematys kitos gijos nepatvirtintų tranzakcijos duomenų.

3.6. DAO

DAO yra paprasčiausias funkcijų rinkinys konkrečioms duomenų bazės operacijoms atlikti. Čia privalo būti realizuota visa duomenų logika. Pagrindiniai šio objekto metodai yra setActiveSession ir attachObject. Išorinis objektas, mūsų atveju TransactionalAspect, kurį apžvelgsime vėliau, naudodamas setActiveSession perduodą egzistuojančią aktyvią sesiją, kurią DAO naudoja duomenims apsikeisti su duomenų baze (11 pav.). AttachObjekt metodas yra skirtas prijungti senos sesijos objektą su nauja sesija. Tai yra dažnai naudojama tiek objektinėse duomenų bazėse tiek ORM bibliotekose. Naudojant senos sesijos objektą su nauja sesija, galima gauti neprognozuojamus rezultatus ar programinės įrangos klaidą.



11 pav. DAO panaudojimas

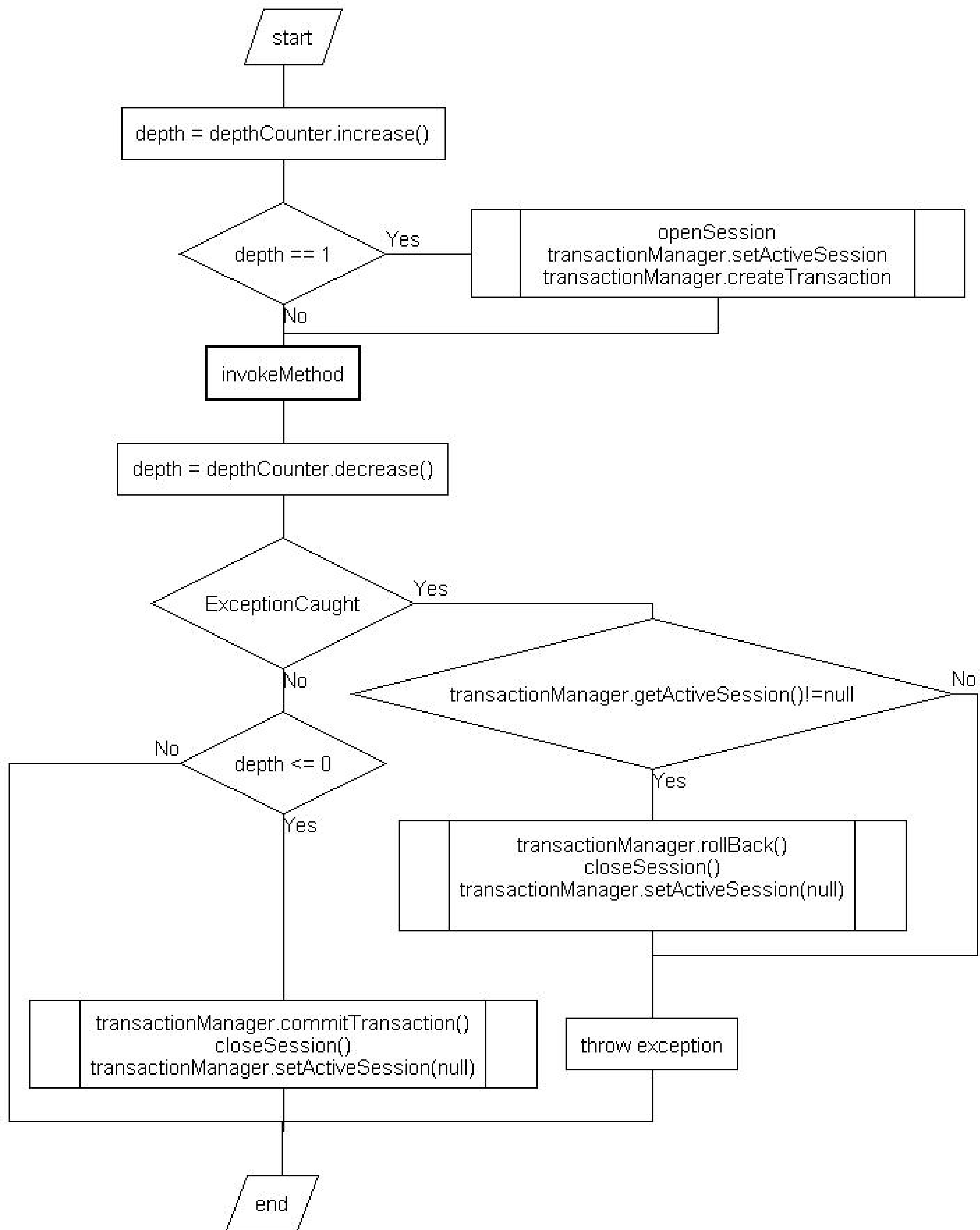
Dao programoje privalo globalus objektas visai programos gijai. Skirtingoms programos gijoms turi būti naudojamos skirtingos DAO kopijos, kadangi čia saugoma aktyvi, konkrečiai gijai naudojama, sesija. Šis objektas yra saugomas ThreadLocal objekte. Visi komponentai, naudojantys DAO objektą privalo jį pasiimti tik iš ThreadLocal. Kitokiu atveju gali atsirasti resursų panaudojimo konfliktai esant kelioms programos gijoms.

3.7. TransactionalAspekt

TransactionalAspect yra esminis šio modelio komponentas, kuris integruoja visus prieš tai minėtus komponentus į vieną visumą.

TransactionalAspektas yra „aplink“ (arround) tipo, tai reiškia, kad aspektas apglėbia pažymėtą metodą. Tam tikra dalis kodo įvykdoma prieš įvykdant metodą, o metodui baigus darbą aspektas toliau tęsiamas aspekto darbas. Aspekto tikslas yra atidaryti naują tranzakciją prieš iškviečiant pažymėtą metodą, o metodui baigus darbą, patvirtinti tranzakciją. Jei metodas praneša apie klaidą, pavyzdžiui, metama programos išimtis, aspektas privalo atšaukti tranzakcijos pakeitimus. Panaudojant prieš tai aprašytus objektus, galima sukurti pilnai automatizuotą modelį tranzakcijoms valdyti.

Prieš vykdant metodą, aspektas iš ThreadLocal objekto paima vykdomosios gijos objektus: TransactionManager bei DAO. Naudodamas TransactionManager klasės metodą createTransaction, sukuriama nauja sesija bei tranzakcija. Sukurta sesija priskiriama DAO. Toliau galima realiai vykdyti kviečiamą metodą. Šis metodas taip pat per ThreadLocal pasiima sau skirtą DAO objektą ir naudoja jį duomenų bazė operacijoms vykdyti. Metodui baigus darbą, aspektas DAO panaikina sesija (perduoda null reikšmę), taip patikrina ar metodas nepranešė klaidų. Jei klaidų nebuvo, TransactionManager iškviečiamas tranzakcijos patvirtinimo funkcija commitTransaction, priešingu atveju tranzakcija atšaukiama (12 pav.).



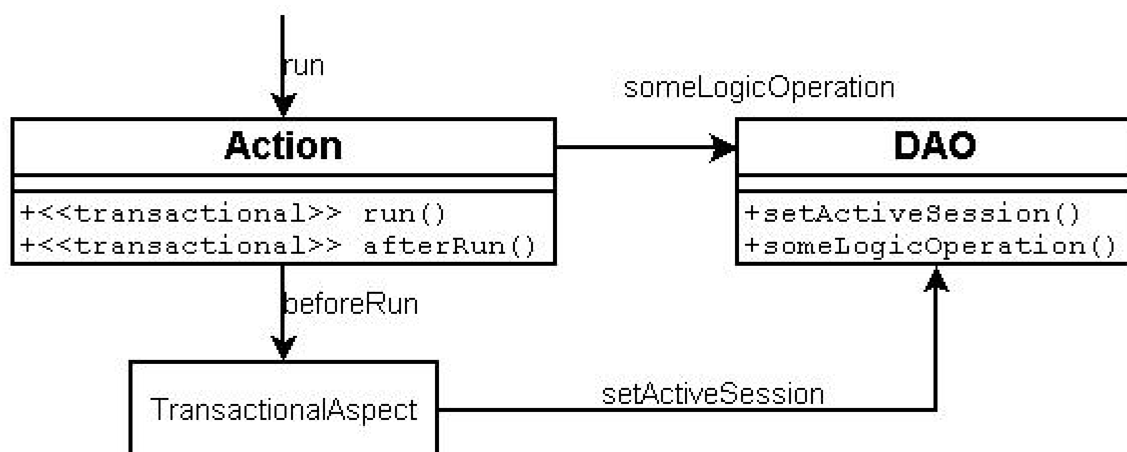
12 pav. TransactionalAspekt algoritmas

3.8. Contoller dalis

Controller dalį sudaro 3 pagrindiniai komponentai:

- Action – atsakingas už vartotojo iškviestų veiksmų apdorojimą,
- ContentProvider – atsakingas už duomenų pateikimą vartotojo sąsajai (tokiems komponentams, kaip List, Combo, Table, Tree)
- LabelProvider – atsakingas už duomenų vertimą į GUI reikalingą formatą.

Pagrindiniai šių klasių metodai privalo būti pažymėti kaip „transactional“, tam kad jiems būtų pritaikytas TransactionalAspect. Pritaikius šį aspektą, automatiškai bus nustatyta aktyvi sesija DAO objektui (13 pav.). Kiekvienas šių klasių metodas operacijų atlikimui privalo naudoti DAO objektą, kurį gali bet kurio momentu pasiimti iš ThreadLocal objekto. Jei naudojamas duomenų bazės objektas, perduodamas per metodo parametrus, šis objektas privalo būti prijungiamas prie egzistuojančios sesijos, naudojant DAO attach metodą.



13 pav. Action klasės iškvietimas

3.9. Rezultatas

TransactionalAspekt dėka griežtai atskiriamas sesijų valdymas (TransactinoManager), duomenų logika (DAO) bei programos logika (likusi programos dalis). DAO nežino iš kur atsiranda sesijos ir kas jas kuria, taip pat nežino kas jį kviečią. TransactionManager taip pat nežino kas naudoja jo sukurtas sesijas, nei kas inicijuoja sesijų sukūrimą. Vartotojo sąsaja nieko nežino apie sesijos egzistavimą, ji naudoja DAO objektą bei objektines duomenų struktūras.

Sukurtos architektūros panaudojimas užtikrina komponentų Tarpusavio nepriklausomumą. Prireikus modifikuoti vieną komponentas, kito komponento modifikacijos yra minimalios arba visai nereikalingos. Tai tiek palengvina sistemos kūrimo procesą, tiek pagerina sistemos kokybę architektūrine prasme. Ši architektūra palengvina migravimo procesą tiek tarp duomenų bazės versijų ar netgi tipo, tiek tarp vartotojo sąsajos bibliotekų versijų ar tipo. Turint tokį architektūrinį modelį, taip pat nesunku organizuoti funkcionalumą. Tinkamai suprojektavus DAO, galima išleisti skirtingas programos versijas su skirtingomis DAO realizacijomis nekeičiant likusių komponentų. Taip pat lengvai realizuoti funkcionalumo kontrolę programos vykdymo metu (runtime), kaip pavyzdžiui vartotojų teisių organizavimas. Priklausomai nuo vartotojo tipo, kuris prisiregistravęs programoje, galima grąžinti būtent jam skirtą DAO realizaciją, kuri leidžia atlikti tik jam leistinas operacijas bei leidžia matyti tik jam skirtus duomenis (duomenų filtravimas skirtingiems vartotojams).

Dėl gero komponentų architektūrinio atskyrimo tokią sistemą yra nesunku testuoti. Kadangi visi pagrindiniai objektai yra individualūs ir konkrečiai nežino su kokia realizacija jie dirba, tokius objektus galima testuoti atskirai panaudojant programos kamščius (mock objects). Tai galima nepriklausomai ištestuoti visus elementus ir tiksliai lokalizuoti egzistuojančias problemas. Tai padidina sistemos kokybę bei palengvina jos priežiūrą bei tobulinimą.

4. MODELIO VEIKIMO CHARAKTERISTIKŲ IR IŠPLEČIAMUMO GALIMYBIŲ TYRIMAS

Pagrindinis kuriamo modelio tikslas yra palengvinti kūrimo procesą bei pagerinti programinės įrangos priežiūrą bei kokybę. Projektinėje dalyje apžvelgėme kaip šis modelis turėtų būti realizuojamas. Nepaisant kūrimo proceso palengvinimo bei priežiūros užtikrinimo, šis modelis privalo užtikrinti efektyvų resursų panaudojimą. Būtent tai užtikrins šio modelio kokybiškumą. Šio skyriaus tikslas – ištirti sukurto modelio veikimo parametrus bei palyginti su analogiškomis situacijomis naudojant kitokius tokios pačios situacijos realizavimo būdus.

Vienas pagrindinių tyrimo aspektų bus modelio veikimo teisingumas ir patikimumas. Sukurtam modeliui patikrinti sukursime testavimo aplinką – eksperimentinę sistemą, tam kad patikrinti ar modelis veikia taip kaip numatyta. Patikimumui tikrinti bus tikrinama, kaip modelis elgiasi klaidų atveju. Užtikrinus modelio veikimo teisingumą bei patikimumą, taip pat būtina ištirti resursų sunaudojimo parametrus. Ištirsime kaip šis modelis įtakoja visos sistemos veikimo greitį, nustatysime kurios modelio dalys gali būti modifikuojamos tam, kad pritaikyti prie sistemos keliamų reikalavimų. Galiausiai ištirsime modelio išplečiamumo galimybes. Nustatysime kaip šio modelio pagalba galima vykdyti apsaugos, bei vartotojo teisių kontrolę. Kaip galima vykdyti funkcionalumo kontrolę tarp skirtingų sistemos versijų.

4.1. Modelio teisingumo tyrimas

Šio tyrimo tikslas yra nustatyti modelio veikimo korektiškumą. Korektiškumą nustatysime sukuriant eksperimentinę sistemą pagal sukurtą modelį. Apsibrėšime galimus sistemos veikimo scenarijus ir ir tikėtiną rezultatą. Atlikus eksperimentinį bandymą palyginsime realius rezultatus su planuotais.

Eksperimentinė sistema bus sukuriama panaudojant aspectj biblioteką skirtą Java aspektams kurti. Duomenims saugoti pasirinkta DB4O nemokama objektinė duomenų bazė. Pirma išbandysime pačio modelio veikimą be duomenų bazės. Patikrinsime ar teisingai valdomas sesijų gyvavimo ciklas.

Tiriant sesijų trukmės valdymą, sukursime TransactionManager testavimo kamštį ir vietoj tikro objekto perduosime objektą, kuris išveda sesijos atidarymų ir uždarymų skaičių į ekraną. Tokiu būdu įvykdžius tam tikrą scenarijų galėsime matyti rezultatą ir jį palyginti su tikėtinais rezultatais. Sudarysime 3 galimus scenarijus:

- paprastas metodo iškvietimas,
- rekursinis metodo iškvietimas,
- dviejų metodų sinchroniškas iškvietimas,
- dviejų metodų iškvietimas iš skirtingų programos gijų.

4.1.1. Paprastas metodo iškvietimas

Paprasto metodo iškvietimo atveju turi būti vieną kartą atidaroma sesija bei sukuriama tranzakcija. Sukurta sesija turi būti perduodama DAO objektui, tada turi būti įvykdomas metodas. Baigus metodui darbą, tranzakcija patvirtinama, o sesija uždaroma. Tai yra pagrindinis ir dažniausiai naudojamas scenarijus. Įvykdžius programą gauti šie rezultatai:

```
TransactionManagerMock.openSession()
TransactionManagerMock.createTransaction()
TransactionManagerMock.getActiveSession()
DaoMock.setActiveSession()
Main.method1()
TransactionManagerMock.closeSession()
TransactionManagerMock.commitTransaction()
DaoMock.setActiveSession()
```

Iš rezultatų matyti kad eksperimentas yra sėkmingas. Vykdant metodą method1 automatiškai buvo vykdomas sesijos valdymas.

4.1.2. Rekursinis metodo iškvietimas

Rekursinio iškvietimo atveju sesijos atidarymas bei tranzakcijos sukūrimas privalo būti vykdomas tik prieš pirmą metodo iškvietimą, o uždarymas vykdomas tik pirmam iškviestam metodui baigus darbą. Tą patį metodą rekursiškai kviečiant antrą kartą sesija neliečiama. Įvykdžius programą gauti šie rezultatai:

```
TransactionManagerMock.openSession()
TransactionManagerMock.createTransaction()
TransactionManagerMock.getActiveSession()
DaoMock.setActiveSession()
Main.method2() depth:0 begin
Main.method2() depth:1 begin
Main.method2() depth:2 begin
Main.method2() depth:2 end
Main.method2() depth:1 end
Main.method2() depth:0 end
TransactionManagerMock.closeSession()
TransactionManagerMock.commitTransaction()
DaoMock.setActiveSession()
```

Kaip matome pagal gautą programos sugeneruotą tekstą, metodas method2 buvo rekursiškai kviečiamas 3 kartus. Sesijos valdymas vykdomas tik prieš pirmo metodo iškvietimą bei jo darbo baigimą. Eksperimentas laikomas sėkmingu.

4.1.3. Dviejų metodų sinchroniškas iškvietimas

Dviejų metodų sinchroniško iškvietimo atveju sesijos valdymas privalo būti vykdomas kiekvienam metodui, t.y. Prieš kiekvieną iš metodų privalo būti atidaroma sesija bei sukuriama tranzakcija, o baigus darbą tranzakcija patvirtinama, o sesija uždaroma.

```
TransactionManagerMock.openSession()
TransactionManagerMock.createTransaction()
TransactionManagerMock.getActiveSession()
DaoMock.setActiveSession()
Main.method1()
TransactionManagerMock.closeSession()
TransactionManagerMock.commitTransaction()
DaoMock.setActiveSession()
TransactionManagerMock.openSession()
TransactionManagerMock.createTransaction()
TransactionManagerMock.getActiveSession()
DaoMock.setActiveSession()
Main.method3()
TransactionManagerMock.closeSession()
TransactionManagerMock.commitTransaction()
DaoMock.setActiveSession()
```

Eksperimentas laikomas sėkmingu, kadangi jis patvirtino tikėtinus rezultatus. Dviem sinchroniškai vykdomiem metodam vykdomas atskiras sesijos valdymas.

4.1.4. Dviejų metodų iškvietimas iš skirtingų programos gijų

Šio tyrimo eksperimento tikslas yra patikrinti ar teisingai veikia ThreadLocal realizuota klasė, paskirstantis atskiras reikiamų objektų kopijas skirtingoms programų gijoms. Šis paskirstymas taip pat yra valdomas automatiškai. Eksperimento uždavinys sukurti dvi programos gijas, kurios iškviestų tranzakcija pažymėtą metodą. Iškviečiant antrą metodą, pirmo metodo vykdymas dar būti nepasibaigęs. Tokiu būdu bus patikrinama ar nenaudojami tie patys objektai skirtingoms gijoms. Taip pat papildomai reikalinga įdėti į TransactionManager metodų iškvietimą objekto adreso numerio išvedimą, tam kad būtų galima sulyginti ar skirtingos gijos naudoja skirtingus objektus. Atlikus eksperimentą gauti šie rezultatai:

```
TransactionManagerMock.openSession()
objectId=lt.jurna.magistras.TransactionManagerMock@743399
TransactionManagerMock.createTransaction()
objectId=lt.jurna.magistras.TransactionManagerMock@743399
TransactionManagerMock.getActiveSession()
objectId=lt.jurna.magistras.TransactionManagerMock@743399
DaoMock.setActiveSession()
Main.method1() entering
TransactionManagerMock.openSession()
objectId=lt.jurna.magistras.TransactionManagerMock@e7b241
TransactionManagerMock.createTransaction()
objectId=lt.jurna.magistras.TransactionManagerMock@e7b241
TransactionManagerMock.getActiveSession()
objectId=lt.jurna.magistras.TransactionManagerMock@e7b241
DaoMock.setActiveSession()
Main.method1() entering
Main.method1() exiting
TransactionManagerMock.closeSession()
objectId=lt.jurna.magistras.TransactionManagerMock@743399
TransactionManagerMock.commitTransaction()
objectId=lt.jurna.magistras.TransactionManagerMock@743399
DaoMock.setActiveSession()
Main.method1() exiting
TransactionManagerMock.closeSession()
objectId=lt.jurna.magistras.TransactionManagerMock@e7b241
TransactionManagerMock.commitTransaction()
objectId=lt.jurna.magistras.TransactionManagerMock@e7b241
DaoMock.setActiveSession()
```

Rezultatuose aiškiai matyti, kad tą patį metodą kviečiant iš skirtingų programos gijų naudojami skirtingi objektai sesijos valdymui vykdyti. Taip gauname skirtingas sesijų kopijas, bei skirtingas sesijų valdymo būsenas atskiroms programos gijoms. Eksperimentas laikomas sėkmingu.

Analogiškas eksperimentas buvo atliktas panaudojant realią duomenų bazę DB4O. Buvo tikrinamas duomenų apsikeitimas tarp duomenų bazės bei programos. Eksperimentai taip pat davė teigiamą rezultatą.

Atsižvelgiant į atliktus eksperimentus galima teigti kad metodas veikia korektiškai idealiomis veikimo sąlygomis. Tam kad nustatyti metodo elgseną netikėtose situacijose atliksime modelio patikimumo tyrimą.

4.1.5. Modelio patikimumo tyrimas

Patikimumo tyrimo tikslas yra išsiaiškinti kaip modelis elgiasi netikėtose situacijose. Netikėtos situacijos gali būti 2 tipų:

- programos išimčių metimas/pagavimas,
- gijų resursų naudojimo konfliktai.

Pirmu atveju sukursime tokį veikimo scenarijų, kuomet iškviečiami 2 metodai (vienas metodas kito metodo viduje). Paskutinysis iškviestas metodas meta programos išimtį. Tokiu atveju aspektas privalo pagauti šią išimtį, nutraukti tranzakciją ir uždaryti sesiją bei toliau mesti pagautą išimtį. Baigus pirmajam metodui darbą, aspektas privalo neatlikti jokių veiksmų su sesija bei tranzakcija. Eksperimentams vykdyti panaudojama ta pati eksperimentinė programa, kuri buvo naudojama korektiškumo tyrime. Atlikus eksperimentą gauti šie rezultatai:

```
TransactionManagerMock.openSession()
TransactionManagerMock.createTransaction()
TransactionManagerMock.getActiveSession()
DaoMock.setActiveSession()
Main.method5()
Main.method6()
TransactionManagerMock.rollbackTransaction()
TransactionManagerMock.closeSession()
DaoMock.setActiveSession()
Main method caught exception
```

Matome, kad programa teisingai pagavo programos išimtį, ir baigus paskutiniam metodui darbą vietoj to, kad vykdyti tranzakcijos patvirtinimą, kaip kad vykdoma įprastu atveju, vykdomas tranzakcijos atšaukimas. Ši programos išimtis tolia metama ir ji pagaunama pagrindiniame programos metode.

Gijų resursu naudojimo konfliktų testavimą vykdysime sukuriant apkrovimo testus. Bus sukurtos 2 programos gijos, kurios vienu metu vykdys be perstojimo metodų iškvietimus taip maksimaliai padidinant resursų panaudojimo konfliktų tikimybę. Testas bus vykdomas 30 minučių. Jei per šį laiką neįvyksta klaida, laikoma, kad konfliktų nerasta. Atlikus apkrovimo testavimą nebuvo aptikta resursų panaudojimo konfliktų.

Apibendrinant gautus rezultatus galima teigti, kad pasiektas reikiamas modelio patikimumas. Šis modelis gali būti naudojamas esant kelioms gijoms vienu metu. Taip pat kontroliuojamas programos išimčių valdymas bei klaidų apdorojimas.

4.2. Modelio resursų sunaudojimo tyrimas

Šis modelis nėra orientuotas į konkrečią probleminę sritį, taip pat kuriant šį modelį buvo laikomasi pagrindinio tikslo: sukurti kuo universalesnį modelį, kuris galėtų būti pritaikomas skirtingo pobūdžio programose. Dėl šios priežasties labai susiaurėją modelio tyrimo sritis resursų sunaudojimo atžvilgiu. Pagrindiniai parametrai įtakojantys sistemos darbą yra pagrindinio aspekto veikimo greitis. Nustačius jo veikimo charakteristikas tuo pačiu nustatysime ir bendrą modelio įtaka visai jį naudojančiai sistemai.

Tyrimui atlikti sukursime testavimo aplinką panaudojant kuriamą modelį. Šio testavimo tikslas kokią įtaką turi aspektų panaudojimas bei jų realizavimas pasirinktu būdu. Šis tyrimas privalo būti nepriklausomas nuo konkrečios duomenų bazės, todėl testavimo aplinkai sukurti nebus naudojama duomenų bazė. Veiksmai bus vykdomi tik iki įrašymo į duomenų bazę momento. Tokiu būdu bus ištirtas tik kuriamo modelio veikimo greitis.

Testavimo aplinkai sukurti panaudosime aspectj biblioteką kuri suteikia aspektų kūrimo funkcionalumą. Tai yra įprasto JAVA kompiliatoriaus papildymas. Prieš kompiliuojant programą įprastu būdu, programa sukompiliuojama naudojant aspectj biblioteką. Gautas rezultatas toliau sukompiliuojamas su įprastu kompiliatoriumi. Tam kad šis procesas vyktų automatiškai, panaudosime Eclipse IDE programavimo aplinką, bei aspectj įskiepę šiai programavimo aplinkai.

Tyrimą suskirstysime į šiuos etapus:

- Vidutinio metodo iškvietimo greičio tyrimas.
- Vykdymų laiko priklausomybė nuo vykdymų skaičiaus bei rekursijos gylio.
- Resursų sunaudojimo tyrimo išvados.

4.2.1. Vidutinio metodo iškvietimo greičio tyrimas

Šio tyrimo metu nustatysime kiek papildomai laiko užtrunka kviečiant tranzakcija pažymėtą metodą su sukurto modelio aspektu.

Panaudojant sukurtą testavimo aplinką, kviečiame tranzakcija pažymėtą metodą 1000000 kartų. Kviečiant metodą, jam kiekvieną kartą bus pritaikomas TransactionalAspect. Bandymas atliekamas 3 kartus, o iš gautų rezultatų išvedamas vidurkis. Atlikus šiuos skaičiavimus gavome, kad 1000000 kartų iškvietimo trukmė yra lygi 761ms, o vieno iškvietimo trukmė yra lygi 0.7ns. Galima daryti išvadą kad toks vykdymo greitis tikrai neturės nepastebimai mažą įtaką bendram sistemos veikimo greičiui.

Papildomam tyrimui, panaudojant profiliavimo įrankį „NetBeans Profiler", ištirsime kurios modelio dalys ilgiausiai vykdomos.

Atlikus 1 000 000 tranzakcija pažymėto metodo iškvietimų, gautas toks profiliavimo pagal komponentų veikimo trukmę rezultatas (14 pav.):

14 pav. iškvietimų profiliavimas

Akivaizdžiai matome, kad daugiausia veikimo trukmę įtakoja aspekto vykdymas prieš metodo iškvietimą. Tai yra natūralu, kadangi ši dalis atlieka pagrindinę loginę dalį ir sujungia visus modelio komponentus. Šis aspekto dalis privalo tikrinti rekursijos gylį, nuspręsti kada reikia atidaryti sesiją ar naują tranzakciją, bei nustatyti ryšius tarp naudojamų objektų. Bendru

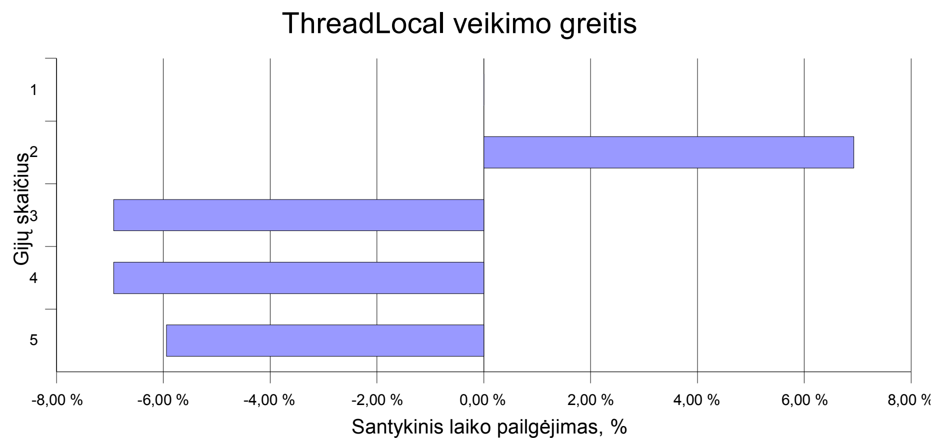
atvejų, susumavus 3 aspekto daliu (aspektas prieš metodą, aspektas po metodo ir AspectOf) laiko trukmę gausime, kad jis užima 50,5% viso modelio veikimo trukmės.

Antras pagal ilgiausią vykdymo trukmę yra ThreadLocal komponento veikimas, kuris atsakingas už elementų paskirstymą tarp kelių skirtingų programos gijų. Jei sistemoje naudojama tik viena gija duomenims apdoroti, tuomet šio komponento galima atsisakyti, taip labiausiai sutaupant šio modelio resursų sunaudojimą. Iš šios išvados išplaukia naujas tyrimas, kurio tikslas yra nustatyti kaip kinta ThreadLocal vykdymo trukmė esant skirtingam gijų skaičiui.

ThreadLocal trukmės tyrimui sukursime analogišką tyrimą, tik prieš vykdant testą, sukursime skirtingą gijų skaičių ir tikrinsime ThreadLocal vykdymo trukmės pailgėjimą procentais, lyginant su testu turinčiu tik vieną giją. Rezultatams apskaičiuoti bus naudojama ši formulė:

$$f(i)=\frac{-(t_1-t_i)}{t_1}*100 \quad (1).$$

Atlikus bandymus gauti šie rezultatai :



15 pav. ThreadLocal veikimo trukmė keičiant gijų skaičių

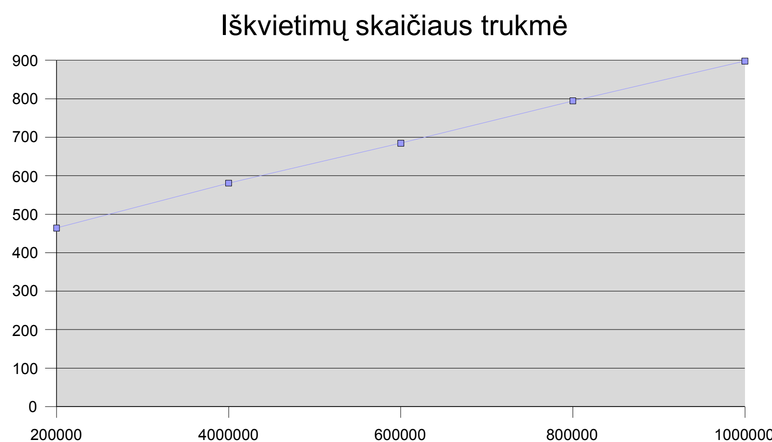
Iš rezultatų matyti, kad negalima prognozuoti aiškaus ThreadLocal vykdymo laiko priklausomai nuo gijų skaičiaus. Tai yra todėl, kad gijų skaičius arba neįtakoja jo veikimo greičio arba ši įtaka yra per maža, kad ją pastebėtume. Todėl šį tyrimą labiau įtakoja ne ThreadLocal, o pašaliniai veiksniai, tokie kaip kitų operacinės sistemos procesų įtaka procesoriaus darbui, Java virtualios mašinos darbo ypatumai. Atsižvelgiant į minėtus faktorius galime teigti, kad gijų kitimas nuo 1 iki 5 neįtakoja ThreadLocal darbo. Modelis nebuvo tiriamas su didesniu gijų skaičiumi nei 5. Realiose sistemose gijų, aptarnaujančių duomenis turėtų būti kuo mažiau. Turint daugiau nei 2 programos gijas sistemos darbą labiau įtakos gijų sinchronizavimas nei ThreadLocal veikimas.

Pažvelgus į laikų trukmės grafika (15 pav.), pastebėsime, kad palyginus mažai įtakos bendram veikimo laikui turi sesijos atidarymo vykdymas. Tai yra dėl SessionPool komponento egzistavimo. Vykdant 1 000 000 operacijų, idealiu atveju sesijos atidarymo funkcija pilnai bus įvykdoma tik vieną kartą. Jei aktyvi sesija jau egzistuoja, šis metodas bus ignoruojamas, tačiau jei vykdymo metu sesijos ryšys nutrūko, sesijos atidarymas vyks pilnai numatytą laiką. Taip sutaupomas resursų panaudojimas, tuo pačiu užtikrinant sistemos patikimą veikimą.

4.2.2. Vykdymų laiko priklausomybė nuo vykdymų skaičiaus bei rekursijos gylio

Šio tyrimo metu nustatysime kaip kinta programos vykdymo trukmė esant skirtingam tranzakcija pažymėto metodo iškvietimų skaičiui. Taip pat ištirsime kokią įtaka veikimo trukmei turi rekursijos atsiradimas. Šiuos rezultatus sulyginsime tarpusavyje.

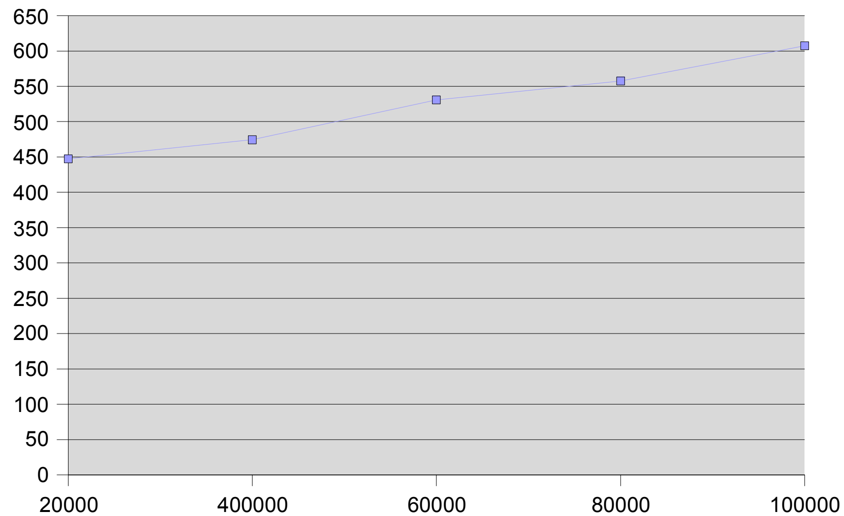
Pirmiausia apskaičiuosime laiko pasikeitimą keičiant metodų iškvietimo skaičių. Tyrimas atliekamas naudojant prieš tai sukurtą testavimo aplinką. Bus gaunami 5 rezultatai vykdant metodų iškvietimą atitinkamai 200 000 kartų, 400 000 kartų kol pasieksime 1 000 000 kartų. Kiekvienam rezultatui gauti bus vykdomi 3 testavimai ir išvedamas jų vidurkis. Tai daroma tam, kad būtų sumažinama tyrimo paklaida. Atlikus testavimą, gauti šie rezultatai (16 pav.):



16 pav. Iškvietimų skaičiaus trukmė

Aiškiai matyti, kad trukmė tiesiškai kinta nuo iškvietimų skaičiaus. Tai yra todėl, kad metode nenaudojama iteravimo procesų. Tiesinė priklausomybė leidžia lengvai prognozuoti sistemos veikimo greitį sistemai augant bei sudėtingėjant. Grafike taip pat matyti, kad vidutinė vieno metodo vykdymo trukmė didėjant metodo iškvietimų skaičiui mažėja. Tai galima pagrįsti tuo, kad pirmam metodo vykdymui reikalinga daug laiko inicializuoti pradines reikšmes. Taip pat po pirmų metodo iškvietimų Java virtuali mašina optimizuoja programos veikimą, taip jį paspartindama. Bendru atveju galima teigti, kad šis modelis minimaliai įtakos sistemą, turinčią daug operacijų bei daug tranzakcijų, bei sistemos augimas nesukels neigiamų pasekmių modelio resursų sunaudojimui.

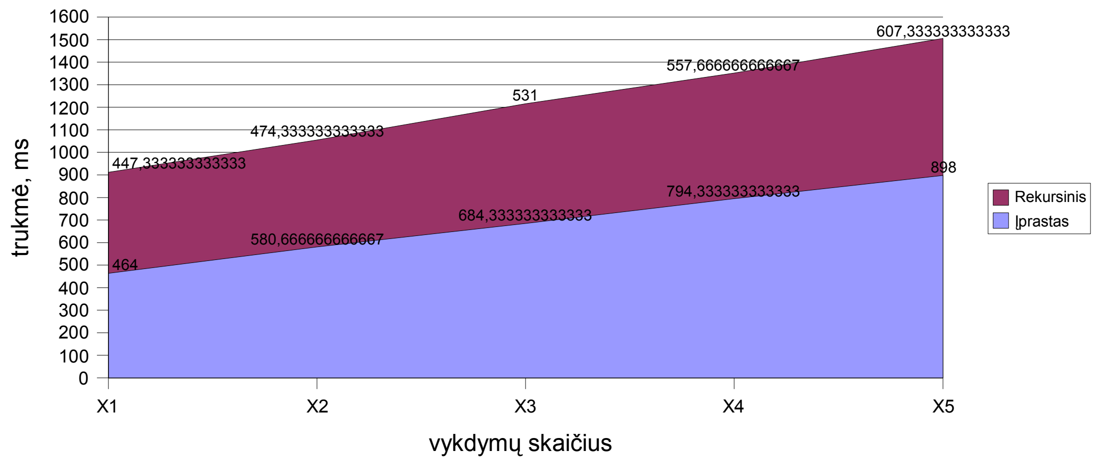
Rekursijos tyrimo tikslas yra patikrinti kaip elgiasi modelis, kuomet tranzakcija pažymėtas metodas kviečiamas rekursiškai. Kaip auga tokio metodo iškvietimų skaičius. Šiam tyrimui panaudosime tą pačią testavimo aplinką, tačiau modifikuosime kviečiamą, tranzakcija pažymėtą metodą. Šis metodas bus rekursiškai kviečiamas 10 kartų. Atitinkamai 10 kartų bus sumažintas metodų kvietimų skaičius, tam kad galutiniame rezultate turėtume duomenis, kuriuos galėtume sulyginti su iškvietimų skaičiaus tyrimo duomenimis. Atlikus tyrimą gauti šie rezultatai (17 pav.):



17 pav. Rekursijos testavimas

Čia taip pat matyti, kad rekursijos iškvietimas tiesiškai priklauso nuo vykdymų skaičiaus. Todėl galime teigti kad sukurtas aspektas veikia efektyviai ir mažai įtakoja su rekursijos vykdymą. Matomas nežymus tiesės svyravimas esant skirtingam iškvietimų skaičiui. Tai gali būti paklaida, atsirandanti dėl tiek operacinės sistemos veikimo bei resursų skirstymo arba Java virtualios mašinos veikimo ypatumų.

Tam kad patikrinti kaip auga metodų iškvietimas įprastu bei rekursijos būdu, sulyginsime gautus rezultatus. Rezultate aiškiai matyti, kad metodo vykdomo rekursiškai vykdymo greitis yra trumpesnis nei įprasto iškvietimo. Taip atsitinka dėl TransactionalAspect logikos. TransactionalAspect seka metodo rekursijos gylį ir priklausomai ar metodas kviečiamas pirmą kartą ar ne sesijos valdymas yra atliekamas arba ne. Šis tyrimas patvirtina TransactionalAspect efektyvumą bei teisingą veikimą.



18 pav. Vykdymo trukmės palyginimas

Šis rezultatas neparodo, kad yra efektyviau naudoti rekursiją negu įprastą metodų kvietimą. Realioje programoje rekursinis tranzakcija pažymėto metodo kvietimas vyks nuolatos. Geriausias pavyzdys yra su vykdymo klasėmis MVC architektūroje. Tarkime turime 2 vykdymo klases, kurių pagrindinis metodas pažymėtas kaip tranzakcijos metodas. Tarkime, kad turime trečią vykdymo klasę, kuri yra aukštesniame abstrakcijos lygyje nei kitos dvi klasės. Pagrindinis šios klasės metodas yra taip pat pažymėtas kaip tranzakcijos metodas ir jis kviečia prieš tai minėtas dvi vykdymo klases. Kviečiant trečiosios klasės metodą TransactionalAspekt bus pritaikomas 3 kartus, tačiau tik pirmas aspekto pritaikymas bus pilnas, o kiti du bus ignoruojami. Taip kviečiant šiuos objektus modelio įtaka vykdymo trukmei bus mažesnė nei vykdant kiekvieną metodą atskirai.

4.2.3. Tyrimo išvados

Atlikus greičio tyrimą buvo nustatytas pagrindinio modelio elemento – TransactionalAspect veikimo greitis. Modelis šiame tyrime nebuvo lyginamas su įprastais sesijų valdymo būdais, kuomet naudojamas įprastinis sesijų valdymas nenaudojant aspektų. Bendru atveju modelio įtaka sistemos veikimo greičiui turi minimalią įtaką.

Nustatytas greitis gali būti naudojamas vykdant įvairias išankstines sistemos prognozes pridedant gautus rezultatus prie suskaičiuotų galimų sistemos veikimo parametrų. Tyrimo metu buvo nustatyta kad modelis tiesiškai priklausomas metodų, gijų skaičiui, todėl galima teigti, kad didelės sistemos atveju modelis prognozuojamai turės įtaką visai sistemai.

4.3. Modelio išplečiamumo galimybės

Sukurtas modelis yra skirtas duomenų bazių sesijų valdymui palengvinti. Šis modelis nagrinėja tik sesijos valdymo problemas bei siūlo jų sprendimo būdus. Jis yra orientuotas į siaurą sritį. Nepaisant to, šis architektūrinis modelis suteikia papildomas galimybes, kurios gali būti naudojamos kartu su šiuo modeliu:

- vartotojo teisių valdymas – nusako kaip gali būti valdoma programoje leistinų veiksmų apsauga skirtingiem programos vartotojams,
- vartotojo funkcijų veikimo valdymas – nusako kaip galima dinamiškai keisti funkcijų grąžinamus rezultatus priklausomai nuo programos vartotojo,
- funkcionalumo valdymas – nusako kaip galima nesunkiai leisti skirtingas programų versijas su apribotu ar modifikuotu funkcionalumu.

Sukurtas modelis leidžia lengvai realizuoti šį papildomą funkcionalumą panaudojant egzistuojančius komponentus. Toks realizavimas lygiai taip pat palengvina tiek sistemos kūrimo procesą tiek sistemos priežiūrą.

4.3.1. Vartotojo teisių valdymas

Pagrindinis vartotojo teisių valdymo tikslas yra leistinų veiksmų apribojimas skirtingiem programos vartotojams. Paprastai vartotojo teisių valdymas sukelia daug problemų realizuojant sistemą, kadangi sunku šį valdymo modulį išskirti į atskirą komponentą. Naudojant sukurtą modelį ši problema gali būti išsprendžiama panaudojant egzistuojančią TransactionalAspect realizaciją.

Vartotojo teisių realizacijai reikalinga sukurti 2 papildomus komponentus:

- vartotojo klasę, sauganči informaciją apie prisijungusį vartotoją,
- TransactionalAspekt išplėtimas, įdedant papildomą tikrinimą kviečiamoms funkcijoms.

Vartotojo klasė privalo turėti veiksmų pavadinimų sąrašą, kurie leidžiamos vykdyti konkrečiam vartotojui. Prisijungus tam tikram vartotojui šis sąrašas užpildomas veiksmai, kurie leistini prisijungusio vartotojo tipui ar konkrečiai šiam vartotojui, bei šis vartotojo objektas perduodamas į TransactionalAspect. Tuo tarpu TransactionalAspect prieš iškviesdamas tam tikra programos metodą patikrina ar jis yra vartotojo funkcijų sąraše. Jei ši sąlyga yra tenkinama, yra vykdoma. Priešingu atveju vietoj to, kad būtų vykdomas metodas,

jis yra ignoruojamas, o vartotojui parodomas klaidos pranešimas, informuojantis apie vartotojo teisių trūkumą. Tokiu būdu galima valdyti vartotojo teises tiek statiškai tiek dinamiškai. Statiškai vartotojo teisės tiesiog gali būti suprogramuojamos programos kode, o dinamiškai gali būti nuskaitomos iš failo ar duomenų bazės.

Sistemą su tokia vartotojo teisių realizacija yra lengva tobulinti bei prižiūrėti. Taip pat kaip ir su sesijų valdymu, čia realūs metodai atliekantys loginius veiksmus su duomenimis nieko nežino apie vykdomą vartotojo teisių apsaugą. Apsauga yra vykdoma vienoje programos vietoje ir ją nesunku modifikuoti. Taip pat yra lengva keisti vartotojui leidžiamų funkcijų sąrašą.

4.3.2. Vartotojo funkcijų veikimo valdymas

Dažnai duomenų valdymo sistemose pasitaiko situacijos kuomet skirtingiems vartotojams ta pati funkcija privalo gražinti skirtingus duomenis priklausomai nuo vartotojo tipo. Vienas iš pavyzdžių yra duomenų filtravimas. Tarkime, kad vartotojų tesės organizuojamos pagal hierarchijos lygius. Aukščiausias hierarchijos lygis mato visus galimus objektus. Žemesnis hierarchijos lygis mato tik jo arba žemesnio lygio objektus. Aukštesniame lygyje egzistuojantys objektai jam yra neprieinami. Tokiu atveju turime tą pačią funkciją, tačiau skirtingiem vartotojams ji turi generuoti skirtingas užklausas į duomenų bazę. Tokį funkcionalumo valdymą realizuojant įprastu būdu reikėtų kiekvienoje funkcijoje atlikti tikrinimą koks vartotojas yra prisijungęs ir kokią užklausą generuoti į duomenų bazę.

Realizuojant vartotojo funkcijų veikimo valdymą panaudojant sukurtą modelį galima šį valdymą palengvinti. Modelyje pagrindinis objektas atliekantis logines funkcijas yra DAO. DAO sukūrus kaip interfeisą, jam galima sukurti skirtingas realizacijas, kurios realiai atlieka skirtingus veiksmus su duomenų baze. Programos starto metu patikrinama koks vartotojas prisijungė. Priklausomai nuo prisijungusio vartotojo sukuriama skirtinga DAO realizacija ir perduodama ThreadLocal. Tokiu būdu prisijungęs vartotojas galės vykdyti tik jam būdingas duomenų bazės užklausas, realizuotas jam skirtame DAO.

Turint tokią vartotojo funkcijų veikimo valdymo realizaciją yra nesunku modifikuoti bei prižiūrėti turimą sistemą. Norint įvesti naują vartotojo tipą ir jam reikiama funkcionalumą, jam sukuriama nauja DAO realizacija ir pakeičiami reikalingi metodai. Likusi sistemos dalis nežino kad yra vykdomas funkcionalumo valdymas, kas palengvina sistemos priežiūrą.

4.3.3. Programos versijų funkcionalumo valdymas

Besivystant kuriamai sistemai ir atsirandant vis daugiau jos vartotojų reikalinga organizuoti sistemos versijų valdymą. Atsiranda poreikis skirtingiems vartotojams teikti skirtingas programų versijas, kurios skiriasi naudojamos duomenų bazės ne tik versijomis bet ir jų gamintojais. Taip pat tobulėjant sistemai gali atsirasti būtinybė sistemos migracijai nuo vienos duomenų bazės prie kitos, ar netgi nuo vienokio saugojimo būdo prie kitokio. Dažnai ši tema yra skaudi bet kuriai sistemai.

Realizuojant sistema su kuriamu modeliu, šia problemą galima panašiai išspręsti kaip išsprendėme vartotojo funkcijų veikimo valdymo problemą. Galima sukurti skirtingas programų versijas su skirtingomis DAO realizacijomis. Tarkim vienam vartotojui reikalinga sistema dirbanti su TeraData duomenų baze, kadangi jis tikisi didelio duomenų kiekio. Kitas vartotojas tikisi visiškai mažo vartotojų skaičiaus ir jam TeraData duomenų bazė yra per brangi ir visai nenaudinga. Jam yra patogiau naudoti nemokamą PostgreSQL duomenų bazę. Nors abiejose duomenų bazėse yra realizuota ta pati SQL kalba, realybėje visuomet migruojant nuo vienos duomenų bazės prie kitos susiduriama su tam tikrais duomenų bazių nesuderinamumo niuansais. Todėl galima sukurti dvi skirtingas DAO realizacijas skirtingoms duomenų bazėms ir taip patenkinti abiejų vartotojų poreikius. Toks realizavimo būdas palengvina programos priežiūrą, kadangi keičiamas tik vienas programos objektas. Likusi programos dalis nežino, kad ji gali dirbti su skirtingo tipo duomenų bazėmis.

4.4. Išvados

Ištyrus sukurtą modelį nustatėme esminius modelio veikimo parametrus – tai veikimo greičio kitimą priklausomai nuo skirtingų parametrų. Nustatėme, kad greičio atžvilgiu modelis minimaliai įtakoja bendrą sistemos darbą ir vienas šio modelio pritaikymas metodui trunka mažiau nei 1 ns. Vartotojo sąsajos programose tai yra nepastebima dalis.

Ištyrus modelio korektiškumą bei patikimumą nustatėme kad modelis veikia taip kaip buvo numatyta. Modelis prognozuojamai elgiasi klaidų atveju bei tinkamai naudoja resursus esant kelioms gijoms.

Taip pat išnagrinėjome papildomas modelio išplečiamumo galimybes, kurios leidžia ne tik palengvinti programos kūrimo procesą, bei pagerinti jos kokybę bei priežiūrą.

5. MODELIO EGZISTUOJANČIOS REALIZACIJOS

Sukurtas modelis atsirado praktinio projekto dėka. Kuriant duomenų apskaitos sistemą buvo surinkta patirtis bei žinios pakartotinio panaudojimo modeliui sukurti. Šis modelis yra magistro praktinio darbo rezultatas.

Magistrinio darbo praktinė užduotis „LSŠF varžybų apskaitos bei klasių valdymo sistema“ yra praktinis šio modelio realizacijos pavyzdys. Naudojant šio modelio architektūrą buvo pasiekta aukšta produkto kokybė, kadangi sistema, modelio dėka, tapo lengvai prižiūrima bei testuojama. Eksperimentiniais sumetimais buvo bandoma projekto realizavimo vidurinėje fazėje keisti duomenų baze iš sąryšinės duomenų bazės, panaudojant ORM į pilnai objektinę duomenų bazę. Pakeitimai turėjo minimalią įtaką sistemos modifikavimui. Duomenų bazės pakeitimas pavyko modifikuoti tik esminius, modelyje aprašytus komponentus.

Sukurta sistema buvo realizuota panaudojant šias bibliotekas bei įrankių rinkinius

- vartotojo sąsaja – Eclispe RCP, SWT/JFace. Tai įrankių rinkinys bei biblioteka vartotojo sąsajai kurti. Ši biblioteka tenkina modelio keliamus reikalavimus, nes yra palaikoma MVC architektūra,
- duomenų bazė – DB4O, tai nemokama pilnai objektinė duomenų bazė,
- aspektai – aspectj, leidžia realizuoti aspektų programavimą.

Sukurta sistema pasižymėjo tokiomis savybėmis kaip geras sistemos tobulinimas, kokybės valdymas, geras, atsižvelgiant į tai, kad tai yra Java programa, programos veikimo greitis. Tačiau turėjo vieną iš žymesnių trūkumų – tai atminties sunaudojimas. Tačiau šis veiksnys nėra įtakotas kuriamo modelio, kadangi pagrindinę programos atminties sunaudojimo dalį sudarė vartotojo sąsajos elementai.

6. IŠVADOS

Darbo metu buvo išanalizuoti dauguma galimų pasirinktos srities realizavimo variantų. Buvo apžvelgtas galimų technologijų variantas ir pasirinktos tos technologijos, kurios užtikrina modelio lankstumą, patikimumą. Buvo pasirinkti šie pagrindiniai aspektai:

- objektinių duomenų bazių panaudojimas,
- aspektinis programavimas,
- objektinis programavimas,
- MVC architektūrinis modelis,
- sesijų pakartotinis panaudojimas,
- automatinis duomenų bazės sandorių modelis.

Sukūrus modelį minėtų technologijų pagrindu, gautas lankstus, patikimas, pakartotiniam panaudojimui skirtas modelis. Šis modelis palengvina 4 pagrindinius programinės įrangos kūrimo bei gyvavimo etapus:

- analizės stadiją,
- architektūros kūrimo stadiją,
- kodavimo stadiją,
- sistemos priežiūros stadiją.

Palengvinamas analizės etapas, kadangi šiame darbe apžvelgiamos pagrindinės galimos technologijos modelio srityje. Peržvelgus analitinę dalį bei modelio tyrimo rezultatus, galima nesunkiai nuspręsti kokias technologijas pasirinkti. Modelis palengvina projekto architektūros stadiją, kadangi suteikia kaip pagrindą visai sistemos architektūrai. Lieka sukurti tik sistemai specifines sritis. Palengvinamas sistemos kodavimo procesas. Šio modelio vienas iš tikslų yra pasikartojančio kodo sumažinimas, bei komponentų išskyrimas į nepriklausomas dalis. Dėl šių priežasčių programos kodas gali būti geriau organizuojamas. Dėl nepriklausomų dalių išskyrimo projekto kodavimo procesą yra lengviau paskirstyti tarp turimų resursų. Galiausiai palengvinamas sistemos priežiūros etapas. Dėl geros kodo struktūros, bei išvengto kodo pasikartojimo, sistemą yra daug lengviau prižiūrėti bei tobulinti. Dėl tokių sprendimų kaip aspektų panaudojimas bei galimybė keisti DAO realizaciją įgalina funkcionalumo kontrolę. Dėl MVC architektūros panaudojimo palengvinama sistemos migracija tarp

skirtingų naudojamų bibliotekų versijų ar netgi skirtingų bibliotekų. Dėl aiškaus komponentų atskyrimo, sistemos dalies perorganizavimas turi nedidelę įtakai sistemos daliai.

7. LITERATŪRA

1. SEGALA J. *Thick vs Thin: Different Mobile Apps Need Different Clients.* 2006. [žiūrėta 2006-04-29]. Prieiga per internetą: http://wbt.sys-con.com/read/41075.htm .

2. LAZUTKIN, E. *Clients: thin vs. thick. 2006, sausis*. [žiūrėta 2006-04-29]. Internetinė prieiga: http://lazutkin.com/blog/2006/jan/15/clients-thin-vs-thick/ .

3. KAY C, ALAN. *The Early History of Smalltalk.* 1993. [žiūrėta 2006-04-29]. Prieiga per internetą: http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html.

4. MCJONES, PAUL. *A Brief History of Databases*. 2000. [žiūrėta 2006-05-02]. Internetinė prieiga: http://wwwdb.web.cern.ch/wwwdb/aboutdbs/history/industry.html.

5. KELLER, W. *Object/Relational Access Layers*. Lochham, Vokietija, 1998.

6. HEINCKIENS, P. *Building Scalable Database Applications.* Addison-Wesley 1998.

7. LADDAD, R. *I Want My AOP!. 2000*. [žiūrėta 2006-05-02]. Internetinė prieiga: http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html.

8. KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J.; IRWIN, J. *Aspect-Oriented Programming: Europos objektinio programavimo konferencijos medžiaga.* Suomija, 1997, p. 13-20.

9. DASS, K. *Model-View-Controller*. 2003. [žiūrėta 2006-05-03]. Internetinė prieiga: http://www.indiawebdevelopers.com/technology/java/mvcarchitecture.asp.

10. SCARPINO, M.; HOLDER, S.; NG, S.; MIHALKOVIC, L. *SWT/JFace in Action.* Greenwich, 2004.

11. ROBINSON, M.; VOROBIEV, P. *SWING.* Greenwich, 2003.

8. SANTRAUKA ANGLŲ KALBA

A big variety of new modern programming technologies exist in today's market and each of it provide different approaches for the same problems. It is quite a challenge for a project manager or a system architect to decide which technology is best for their project and a lot of time should be spent for analysis before some decisions could be made. The main purpose of this work is to create a reusable model for JAVA applications that is based on cutting edge technologies such as Aspect-Oriented Programming, Object databases and Model-View-Controller architecture. This work provides research data that could be used for analysing what influence will these new technologies have for the system.

Created model is based on aspect oriented programming. The key component is a TransactionalAspect which does automatic database session and transaction management. It also provides session pooling for better reliability and performance and thread safety by using ThreadLocal for more complex applications. As a result a model was created that helps to manage 4 main stages of system development processes: project analysis precess, architecture process, coding process and system support process. Helps to develop a quality system on time and save project expenses at the same time.

9. TERMINŲ IR SANTRUMPŲ ŽODYNAS

Nr.	*Santrumpa*	*Angliškas pavadinimas*	*Lietuviškas pavadinimas*
1.	OOP	Object Oriented Programming	Objektais orientuotas programavimas
2.	AOP	Aspect Orientes Programming	Aspektais orientuotas programavimas
3.	RDBMS	Relational Database Management System	sąrišinė duomenų bazių valdymo sistema
4.	ORM	Object-relational mapping	Objektinis-realiacinis surišimas, skirtas surišti realiacines duomenų bazes su objektiniu modeliu.
5.	ODBMS	Object Database Management System	Objektinė duomenų bazių valdymo sistema
6.	SQL	Structures Query Language	Struktūrizuota užklausų kalba
7.	OQL	Object Query Language	Objektinė užklausų kalba
8.	JFC	Java Foundation Classes	Java pagrindinės klasės
9.	MVC	Model View Controler	Modelis-vaizdas-valdytojas. Architektūros šablonas skirtas vartotojo sąsajoms kurti, griežtai atskiriant loginę dalį nuo atvaizduojamosios dalies.
10.	HTML	Hyper Text Markup Language	Programavimo kalba skirta kurti puslapius skaitomus su interneto naršyklėmis.
11.	AWT	Abstract Windowing Toolkit	Įrankių rinkinys vartotojo sąsajai kurti.
12.	PĮ		Programinė įranga

10. PRIEDAI

10.1. Eksperimente naudoti įrankiai bei programos

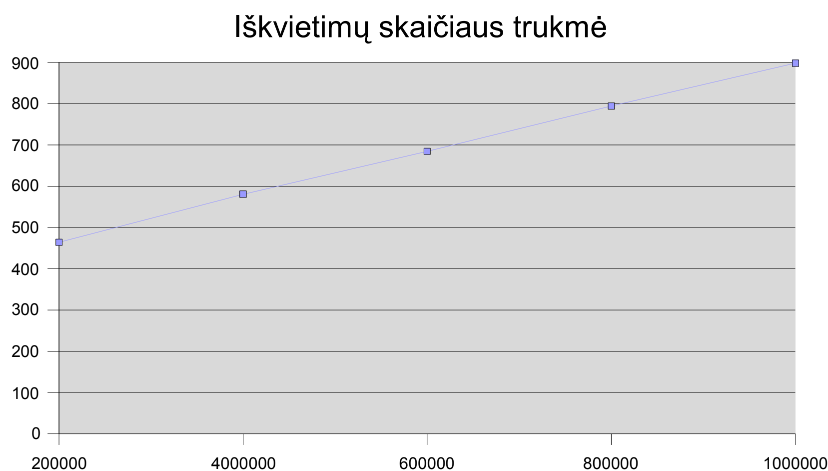
- Eclipse IDE – programų kūrimo aplinka skirta Java programavimo kalbai,
- AspectJ – biblioteka suteikianti aspektinio programavimo galimybę,
- AspectJ Eclipse įskiepis – palengvina programavimo darbus susijusius su aspektiniu programavimu Eclipse IDE aplinkoje,
- DB4O – objektinė duomenų bazė,
- NetBeans profiler – programų profiliavimo įrankis, skirtas analizuoti programos veikimo parametrams.

10.2. Tyrimo bei eksperimentų pilna informacija

10.2.1. Iškvietimų skaičiaus trukmės tyrimo rezultatai

1 lentelė. Iškvietimų skaičius

200000	4000000	600000	800000	1000000
461	581	681	791	901
460	580	681	801	891
471	581	691	791	902
464	580,67	684,33	794,33	898

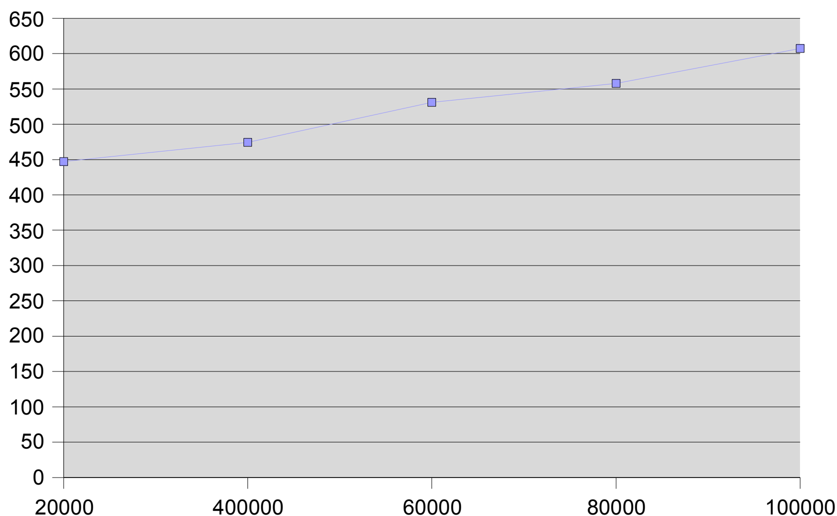


19 pav. Iškvietimų skaičiaus trikmė

10.2.2. Iškvietimų trukmės su rekursija tyrimo rezultatai

2 lentelė. Rekursinio iškvietimo trukmė

20000	400000	60000	80000	100000
441	481	521	551	600
440	471	541	571	601
461	471	531	551	621
447,33	474,33	531	557,67	607,33

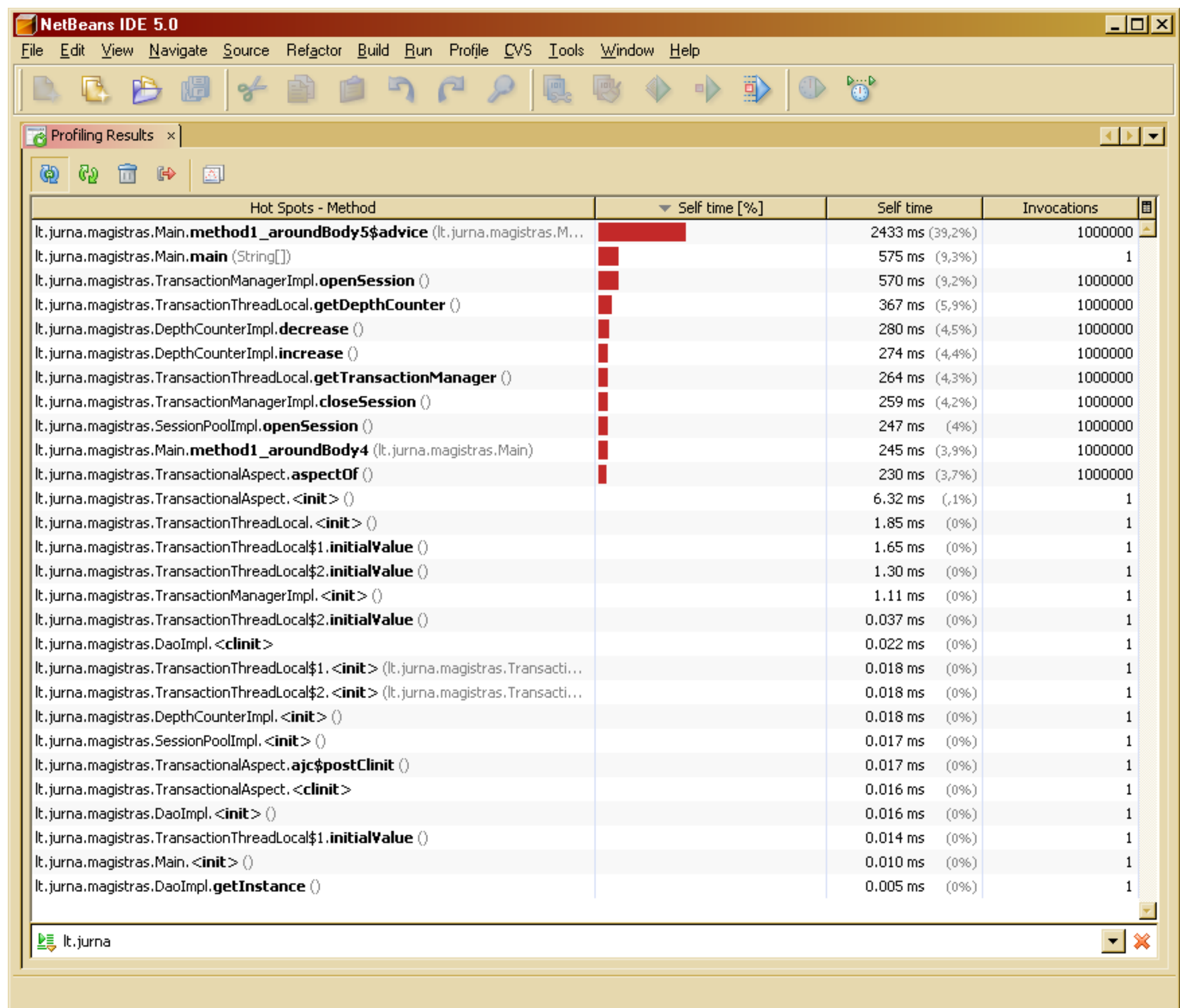


20 pav. Rejursijos iškvietimų skaičiaus grafikas

10.2.3. Laiko pasiskirstymo tarp komponentų tyrimo rezultatai

3 lentelė. Komponentų veikimo trukmė

Objektas	Trukmė, %
Aspektas prieš metodą	40,3
ThreadLocal	10,8
Aspektas po metodo	7,3
OpenSession	6,8
Metodo iškvietimas	6,3
DepthCounter increase/decrease	5,4
AspectOf	2,9
CloseSession	2,6
Kita	17,6



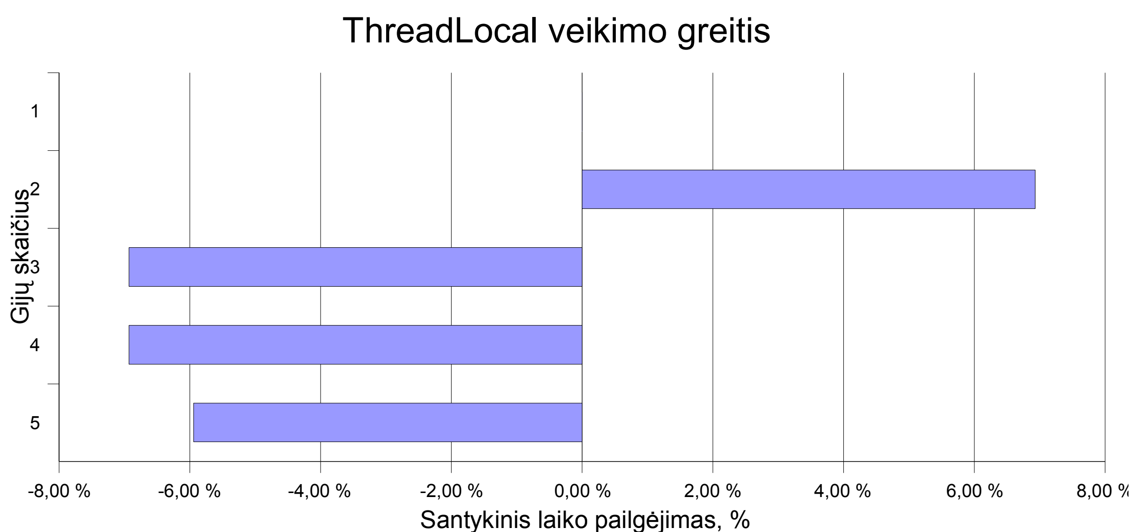
Hot Spots - Method	Self time [%]	Self time	Invocations
lt.jurna.magistras.Main.**method1_aroundBody5$advice** (lt.jurna.magistras.M...		2433 ms (39,2%)	1000000
lt.jurna.magistras.Main.**main** (String[])		575 ms (9,3%)	1
lt.jurna.magistras.TransactionManagerImpl.**openSession** ()		570 ms (9,2%)	1000000
lt.jurna.magistras.TransactionThreadLocal.**getDepthCounter** ()		367 ms (5,9%)	1000000
lt.jurna.magistras.DepthCounterImpl.**decrease** ()		280 ms (4,5%)	1000000
lt.jurna.magistras.DepthCounterImpl.**increase** ()		274 ms (4,4%)	1000000
lt.jurna.magistras.TransactionThreadLocal.**getTransactionManager** ()		264 ms (4,3%)	1000000
lt.jurna.magistras.TransactionManagerImpl.**closeSession** ()		259 ms (4,2%)	1000000
lt.jurna.magistras.SessionPoolImpl.**openSession** ()		247 ms (4%)	1000000
lt.jurna.magistras.Main.**method1_aroundBody4** (lt.jurna.magistras.Main)		245 ms (3,9%)	1000000
lt.jurna.magistras.TransactionalAspect.**aspectOf** ()		230 ms (3,7%)	1000000
lt.jurna.magistras.TransactionalAspect.**<init>** ()		6.32 ms (,1%)	1
lt.jurna.magistras.TransactionThreadLocal.**<init>** ()		1.85 ms (0%)	1
lt.jurna.magistras.TransactionThreadLocal$1.**initialValue** ()		1.65 ms (0%)	1
lt.jurna.magistras.TransactionThreadLocal$2.**initialValue** ()		1.30 ms (0%)	1
lt.jurna.magistras.TransactionManagerImpl.**<init>** ()		1.11 ms (0%)	1
lt.jurna.magistras.TransactionThreadLocal$2.**initialValue** ()		0.037 ms (0%)	1
lt.jurna.magistras.DaoImpl.**<clinit>**		0.022 ms (0%)	1
lt.jurna.magistras.TransactionThreadLocal$1.**<init>** (lt.jurna.magistras.Transacti...		0.018 ms (0%)	1
lt.jurna.magistras.TransactionThreadLocal$2.**<init>** (lt.jurna.magistras.Transacti...		0.018 ms (0%)	1
lt.jurna.magistras.DepthCounterImpl.**<init>** ()		0.018 ms (0%)	1
lt.jurna.magistras.SessionPoolImpl.**<init>** ()		0.017 ms (0%)	1
lt.jurna.magistras.TransactionalAspect.**ajc$postClinit** ()		0.017 ms (0%)	1
lt.jurna.magistras.TransactionalAspect.**<clinit>**		0.016 ms (0%)	1
lt.jurna.magistras.DaoImpl.**<init>** ()		0.016 ms (0%)	1
lt.jurna.magistras.TransactionThreadLocal$1.**initialValue** ()		0.014 ms (0%)	1
lt.jurna.magistras.Main.**<init>** ()		0.010 ms (0%)	1
lt.jurna.magistras.DaoImpl.**getInstance** ()		0.005 ms (0%)	1

21 pav. Komponentų trukmės profiliavimo rezultatai

10.2.4. Gijų skaičiaus įtaka ThreadLocal veikimui

4 lentelė. Gijų skaičiaus įtakos rezultatai

Gijų skaičius	Trukmė	Trukmės pailgėjimas
1	10,1	0,00 %
2	10,8	6,93 %
3	9,4	-6,93 %
4	9,4	-6,93 %
5	9,5	-5,94 %



22 pav. Gijų skaičiaus rezultatų grafikas

10.3. Pavyzdinė modelio realizacija

Pavyzdinė programa, naudojanti sukurtą modelį patalpinta prisektoje laikmenoje.