KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

Tomas Daukantas

Varžybų monitoringo programa ir architektūra, įvertinanti reikalavimų pasikeitimą

Magistro darbas

Darbo vadovas
doc. E. Karčiauskas

Kaunas, 2005

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
PROGRAMŲ INŽINERIJOS KATEDRA

TVIRTINU
Katedros vedėjas
doc. dr. E. Bareiša
2005 01

Varžybų monitoringo programa ir architektūra, įvertinanti reikalavimų pasikeitimą

Magistro darbas

Vadovas

doc. E. Karčiauskas
2005 01

Recenzentas

2005 01

Atliko

T. Daukantas
2005 01

Kaunas, 2005

Turinys

Santrauka

Darbe apibrėžiamos problemos, kylančios kuriant bei programinę įrangą ir susijusios su pakitimu realizavimu kūrimo bei palaikymo fazėse. Aptariami reikalavimai programinės įrangos architektūra, leidžiančiai lengviau realizuoti pakeitimus. Nagrinėjamos pasaulyje paplitusios architektūros, palengvinančios programos palaikymą. Taip pat nagrinėjama trijų lygių architektūros realizacija projekte “paskirstyta krepšinio rungtynių registravimo bei analizės programa”, rungtynių analizės dalyje. Aptariami realizuotos architektūros privalumai bei trūkumai.

Summary

Competition monitoring program and architecture, which evaluates changes of requirements.

Problems, emerging from requirements changes in software development and maintenance phases are analyzed in this thesis. Requirements for an architecture, which allow easier implementation of requirements changes, are discussed. There are analyzed world spread architectures, which ease the maintenance of the software. Also there is analyzed the three tier architecture implementation in the project “distributed basketball competition registration and analysis software”, analysis subsystem. The cons and pros of implemented architecture are discussed also.

1 Įvadas

Kuriant programinę įrangą, reikia atkreipti dėmesį į tai, kad reikalavimai programinei įrangai nėra galutiniai ir nekintami. Jie dažnai ima kisti jau programinės įrangos kūrimo fazėje. Labiausiai tokie pokyčiai įtakoja sistemas, kuriamas evoliuciniu principu. Toks kūrimo modelis taikomas kai vartotojas nori naudotis dalimi programos funkcijų kaip galima anksčiau ir sutinka, kad likusios funkcijas bus realizuotos bei prijungtos jau besinaudojant programine įranga. Tokiu atveju vartotojas gali nuspręsti ar gauna reikiama funkcionalumą, kurio norėjo. Kaip taisyklė, pasirodo, kad vartotojas norėjo kiek kitokio veikimo, todėl reikalavimai programinei įrangai dažnai kinta. Pasitaiko, kad vartotojas panori pridėti papildomą funkcionalumą, kas gali pakeisti naudojamą duomenų struktūrą, ir tai verčia stipriai modifikuoti jau sukurtą programinį kodą.

Reikalavimų pokyčiai neapsiriboja tik evoliuciškai kuriamomis sistemomis. Nuo jų neapsaugotos ir griežtai programinės įrangos gyvavimo ciklu apibrėžta programinė įranga.

Kiekviena tokia modifikacija keičia programos struktūrą, daro programą mažiau suprantamą ir sunkiau modifikuojamą, kol net patys kūrėjai pasiklysta modifikacijų labirinte. Programą taisyti bei plėsti darosi vis sudėtingiau ir kiekvienam išplėtimui bei modifikacijai sugaištama vis daugiau laiko. Tai buvo pastebėta dar 1980 metais Lehmano suformuluotame antrajame programų evoliucijos dėsnyje. Jei nesiimti priemonių, programą tenka dažnai restruktūrizuoti, kas taip pat sueikvoja daug laiko, bet negarantuoja pagerėjimo, arba išvis remiantis buvusia patirtimi kurti naują, atsižvelgiant į buvusios sistemos sėkmingus sprendimus bei sukauptą patirtį ją kuriant. Programuojant evoliuciniu principu restruktūrizavimas duoda labai nedaug naudos, kadangi ši metodika labiau tinka paveldėtosioms sistemoms, kurtoms senu programavimo stilium bei senomis kalbomis. Restruktūrizavimas evoliuciškai kurtai sistemai reikštų buvusios architektūros peržiūrėjimą iš pagrindų ir jos taisymą. Architektūros taisymas programinės įrangos kūrimo metu reiškia didelio kiekio programinio kodo perrašymą ir sunaudoja labai daug brangaus laiko. Kurti produktą iš naujo, naudojantis iki tol sukurta programine įranga kaip prototipu taip pat reiškia milžiniškas laiko sąnaudas ir mažai kuo skiriasi nuo aukščiau minėto metodo.

Paskutinis likęs sprendimas yra kurti architektūrą pritaikytą galimiem reikalavimų bei pačios duomenų struktūros pokyčiams. Tokia architektūra palengvintų programos kūrimą evoliuciniu principu ir sutaupytų galybę laiko, kurį tenka paaukoti perrašant didelę programos kodo dalį.

Būtent su tokia problema teko susidurti kuriant paskirstytą krepšinio rungtynių registravimo ir analizės (monitoringo) programinės įrangos sistemą. Pasitaikydavo tokių

reikalavimų pokyčių, kurie priversdavo keisti duomenų struktūrą tam, kad būtų galima įgyvendinti normą funkcionalumą: reikėjo registruoti iš pradžių nenumatytus parametrus, perdarant sistemą į paskirstytą, duomenų saugumui ir programos nenutrūkstamam darbui užtikrinti teko net keletą kartų radikaliai pakeisti duomenų struktūrą. Kiekvienas toks pokytis net keletui savaičių pristabdydavo programinės įrangos kūrimo darbus, versdamas perrašyti nemažą programinio kodo dalį, idant programinė įranga galėtų vėl funkcionuoti. Žinoma, jei visi reikalavimai būtų surinkti iškart, tokios problemos neiškiltų, bet kaip jau buvo minėta, kuriant programinę įrangą niekada negali būti tikras kad vartotojas nepakeis reikalavimų taip, kad pokyčių įgyvendinimas reikalaus architektūrinių sprendimų peržiūrėjimo. Jau iškilus antram rimtam reikalavimų pokyčiui, kuriam įgyvendinti teko peržiūrėti visą programinės įrangos architektūrą buvo nutarta architektūrą keisti taip, kad duomenų struktūros pokyčiai jos taip stipriai neįtakotų, ir netektų kiekvieną sykį sunaudoti galybę laiko vien tam kad atstatyti programos funkcionalumo “status kvo”.

2 Tikslai ir uždaviniai

Darbo tikslas – išnagrinėti problemas kylančias dėl reikalavimų pakeitimų bei veiksnius, įtakojančius pakeitimų realizavimo sudėtingumą. Išnagrinėti plačiausiai pasaulyje taikomus metodus programinės įrangos pakeitimų įtakai švelninti, jų privalumus bei trūkumus, o taip pat panagrinėti metodus taikytus kuriant paskirstytąją krepšinio rungtynių registravimo bei analizės sistemą.

Siekiant šių tikslų kyla tokie uždaviniai:

- Išnagrinėti problemas kylančias dėl reikalavimų pasikeitimo;
- Išnagrinėti veiksnius kurie labiausiai įtakoja programinės įrangos modifikavimą;
- Išnagrinėti dažniausiai pasaulyje taikomus metodus programinės įrangos modifikavimui palengvinti;
- Išnagrinėti projekte realizuotus metodus programinės įrangos modifikavimui palengvinti.
- Įvertinti metodo pasirinkimą, jo privalumus bei trukumus.

3 Problemų analizė

3.1 Pagrindinės pakeitimų problemos bei juos įtakojantys veiksniai.

Pagrindinė problema, kylanti dėl reikalavimų pasikeitimo yra tai, kad norint jį realizuoti dažnai tenka koreguoti dalį programos architektūros ir perrašinėti jau sukurtą programinį kodą. Tai reiškia, kad tenka grįžti prie jau sukurtų bei ištestuotų komponentų, juos vėl išnagrinėti, suplanuoti pakeitimo realizavimą taip, kad jis kuo mažiau įtakotų kitų komponentų veikimą, realizuoti pakeitimą dažnai perrašant didelę keičiamo elemento dalį iš naujo, o taip pat iš naujo testuoti patį komponentą bei jo integraciją visoje programinėje įrangoje. Tai eikvoja laiką, gadina suplanuotą architektūrą, kadangi neplanuotas pakeitimas labai dažnai verčia daryti sprendimus, iš pradžių nenumatytus kuriant architektūrą. Dėl to darosi vis sunkiau aptikti susidariusias klaidas, pajungti naujus modulius. Jei pakeitimus programoje reikia atlikti dažnai, tai pastovi architektūros modifikacija daro ja griozdišką, kas sunkina kiekvieno sekančio pakeitimo realizavimą. Laikas, kurį tenka užgaišti atliekant pakeitimus proporcingas keičiamų elementų paplitimui programinėje įrangoje bei jų pasiskirstymą programos kode. Tai reiškia kuo daugiau programos modulių (procedūrų, metodų, klasių) apima pakeitimas, tuo ilgiau laiko tenka užgaišti atliekant pakeitimą, kadangi tenka atsižvelgti į didesnę jų svarbą programos architektūroje bei perrašyti didesnį kiekį programinio kodo. Atitinkamai kuo daugiau procedūrų apima pakeitimas, tuo daugiau ryšių susidaro tarp komponentų kuriuos įtakoja pakeitimas, taigi tuo daugiau laiko tenka praleisti ieškant nagrinėjant komponentų tarpusavio ryšius. Taigi, jei modifikacijos įtakojamas programinis kodas sukoncentruotas keliose funkcijose, kurios bendrauja per apibrėžtą sąsają, tada pakeitimų atlikimui sunaudojama gerokai mažiau laiko. Tai tiesiogiai priklauso nuo naudojamos programinės įrangos architektūros.

Taip pat ne pačią mažiausią liko dalį užima testavimas po atliktų pakeitimų. Pakeitimų testavimui galioja tie patys kriterijai kaip ir jų realizavimui – laikas skirtas testavimui priklauso nuo to, kiek programinės įrangos komponentų apima vykdomas pakeitimas, kadangi net jei vykdant modifikavimą teko pakeisti tik vieną metodo (procedūros) eilutę, vis viena teks ištestuoti visą komponentą bei jo integravimą į sistemą. Atatinkamai kuo daugiau modulių apima pakeitimas ir kuo daugiau tarpusavio ryšių susidaro tarp pakeistų modulių, o taip pat tarp pakeistų modulių ir senų programos modulių, kurių pakeitimas tiesiogiai neįtakojo, tuo daugiau tenka vykdyti integracijos į sistemą testavimų.

Taip pat nagrinėjant jau sukurtą programinį kodą didelę reikšmę turi pats programinio kodo sudėtingumas, komentarų kiekis bei jų aiškumas. Šių parametrų reikšmė gerokai išauga, jei pakeitimus realizuoja ne tas pats asmuo, kuris ir sukūrė nagrinėjamą modulį. Programos kodo sudėtingumas rodo kiek laiko darantis pakeitimą asmuo užtruks nagrinėdamas jau sukurtą kodą bei planuodamas pakeitimą. Sudėtingumo poveikis mažesnis jei programuojant kodas buvo taisyklingai bei tinkamai komentuojamas. Net pačiam to kodo programuotojui be komentarų gali būti sunku susigaudyti savo parašytame kode, jei pakeitimą reikia daryti praėjus kuriam laikui po modulio sukūrimo.

3.2 Apibendrinimas

Darosi aišku, kad pagrindinis veiksnys, įtakojantis pakeitimų realizavimą – tinkamos architektūros naudojimas kuris koncentruotų naudojamus komponentus į grupes pagal paskirtį ir tokios grupės bendrautų per nustatytą sąsają. Toks bendravimas per nustatytą sąsają supaprastina bendradarbiavimą tarp komponentų grupių bei sumažina integracijos testavimo kiekį.

Taip pat pakeitimų atlikimą galima palengvinti ir pagreitinti vengiant sudėtingo kodo rašymo bei jį taisyklingai komentuojant.

.

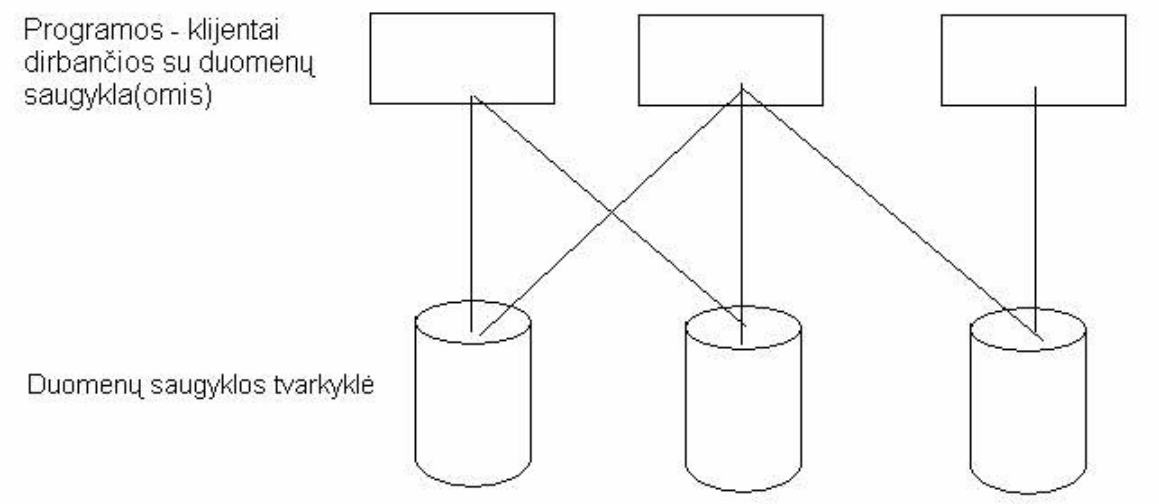
4 Pasaulyje paplitusios architektūros, palengvinančios pakeitimų realizavimą

Nagrinėjant trečią skyrių pasidaro aišku, kad laiką, užtrunkamą pakeitimui padaryti tiesiogiai įtakoja programinio kodo sudėtingumas, kurį iki tam tikros ribos mažina komentarų teisingas komentarų naudojimas, bei pati programinės įrangos architektūra. Iš šių veiksnių geriausiai planuojamas ir reguliuojamas turėtų būti architektūra. Pagaliau teisinga architektūra gali sumažinti ir patį kodo sudėtingumą.

Šiame skyriuje panagrinėsiu pasaulyje taikomas architektūras, kurios palengvina pakeitimų realizavimą. Pirmiausia panagrinėsiu dviejų lygmenų (two tear) architektūra, kadangi ji panašiausia į tai, kokia buvo realizuota projekte pradžioje.

4.1 Dviejų lygmenų principas

Nagrinėjant paskirstytos architektūros kūrimo variantus ir lyginant juos su naudojama architektūra buvo pastebėti esminiai jos panašumai su dviejų lygmenų architektūra. Tokios architektūros principas parodytas pirmame paveiksle.



Pav.1 Dviejų lygmenų architektūra

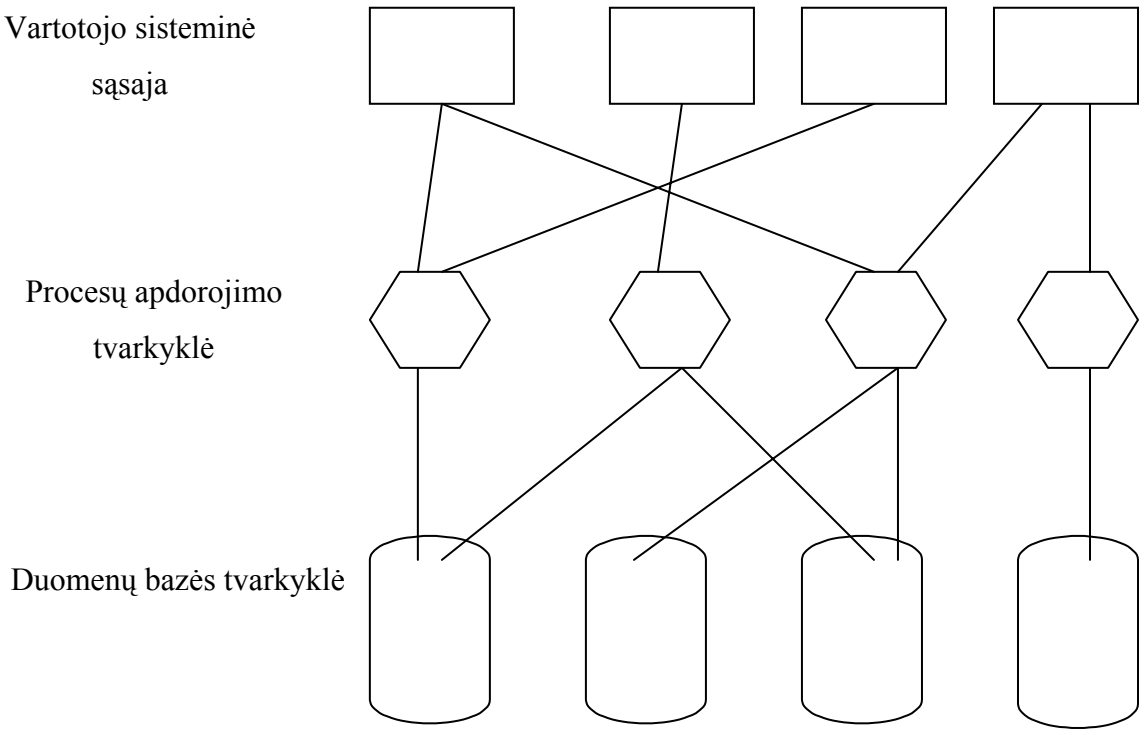
Duomenų saugyklos tvarkykle gali būti tiek programinė įranga, skirta saugoti informaciją failinėse struktūrose, tiek duomenų bazių valdymo sistemos, paremtos SQL.[5] Atliekamo projekto atveju duomenų saugyklą sudarė ir duomenų bazė, ir failinė sistema. Abi sistemos turėjo veikti suderintai, tam kad būtų įmanoma užtikrinti nepertraukiamą programos darbą tuo atveju jei staiga dingtų ryšys su duomenų baze. Duomenų bazių valdiklis turėjo rūpintis duomenų saugojimu bei atnaujinimu dirbant su programa – klientu. Turint omeny kad sistema

paskirstyta, duomenys į suvestais duomenimis dalinasi keletas registruotojų bei trenerių. Klientu šiuo atveju yra pati programa. Programa darbo metu keičiasi duomenimis su duomenų saugykla – registruotojai suveda duomenis apie varžybų eigą, o treneris stebi varžybų metu besikeičiančius analizių pateikiamus rezultatus priklausomai nuo registruotojų suvedamų duomenų. Pastovus nuskaitymas iš duomenų saugyklos būtinas, nes paskirstytoje sistemoje registruoja ne vienas žmogus, o be to analizės pateikiami rezultatai neprivalo būti pateikiami tame pačiame kompiuteryje, prie kurio dirba registruotojas, ir todėl nemaišo registruotojo darbo. Duomenų struktūros priėmimas ir perdavimas tolygiai paskirstytas tarp saugyklos tvarkyklės ir programos, programa visada priima tokios pat struktūros duomenis, kokios jie saugomi saugykloje ir į saugyklą visada perduoda taip pat identiškos struktūros. Reiškia programai priimant protokolo eilutę, ji gauna visada tikslius parametrus kokie buvo suvesti, nes tai būtina tam, kad užtikrinti darbą tarp failinės duomenų saugyklos ir duomenų bazės. Tai reiškia kad programa visada turi žinoti kokios struktūros duomenis reikalingi duomenų bazėje, o bazės tvarkyklė – kokių duomenų reikia programai ir kokiu formatu juos pateikti. Tai reiškia, kad programa savo viduje gautus duomenis laiko tokia pat struktūra, kaip kad ir duomenų bazėje. Be viso kito duomenų bazė paskirstytoje sistemoje paprastai nėra tame pačiame kompiuteryje kaip ir programa. Duomenų bazė turi aptarnauti keletą programų – klientų. Taigi, duomenų saugykla ir kliento programos bendrauja per griežtai nustatytą sąsają. Būtent tokia architektūra iš pradžių buvo planuota ir realizuota paskirstytoje krepšinio rungtynių registravimo ir analizės programoje.

4.2 Trijų lygmenų principas

Nagrinėdami galimas alternatyvias architektūros aptikom taip vadinamą trijų lygmenų architektūrinį sprendimą. Pagrindinė tokios architektūros idėja pavaizduojama 2 paveiksle.

Trijų lygmenų architektūra atsirado apie 1990. Jos specifika – vidurinis lygmuo (angliškai dažnai vadinamas *middleware*) kuris veikia kaip tarpininkas tarp duomenų tvarkymo lygmens(duomenų saugyklos tvarkyklė) ir vartotojo sąsajos, į kuria paduodami jau apdoroti duomenys. Vidurinis lygmuo skirtas apdorojimo logikai realizuoti. Teisingai realizuoti trijų lygmenų architektūra lenkia dviejų lygmenų savo efektyvumu, lankstumu, pakartotinu panaudojamumu, o taip pat palaikymo ir pakeitimų realizavimo lengvumu, o tuo pačiu metu paslepiant realizacijos sudėtingumą nuo vartotojo [9 p. ,5-6; 4]. Pagrindinė problema realizuojant trijų lygmenų architektūrą yra lygmenų išskyrimas. Vartotojo sąsajos logika, pardorojimo valdymo logika ir duomenų saugojimo logika kartais būna sunku tiksliai apibrėžti [4].



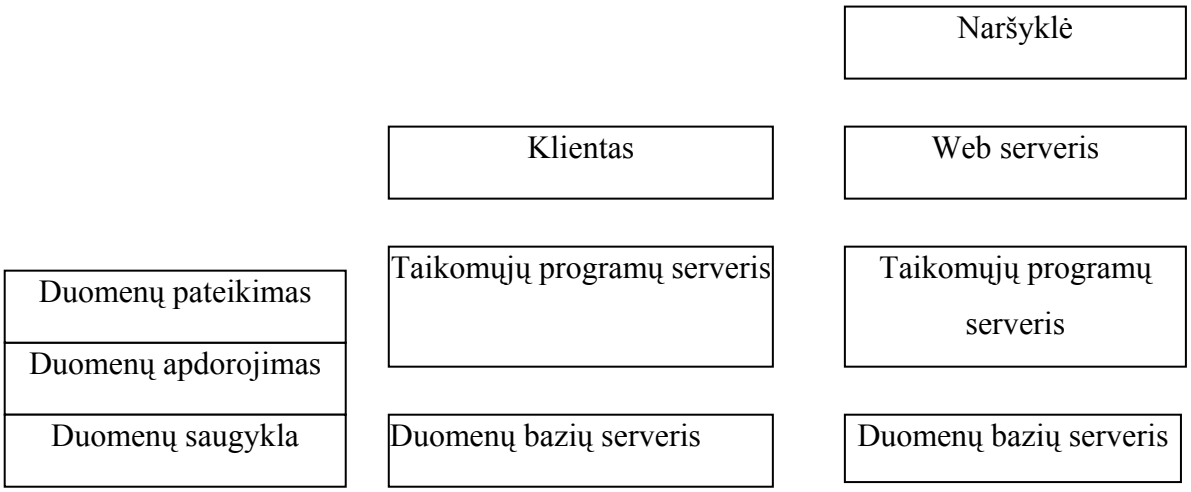
2. pav. Trijų lygmenų architektūros modelis

Pirmiausia tada reikia išsiaiškinti kas būtent yra ta vidurinio lygmens programinė įranga. Programinės įrangos industrijoje nėra tikslaus susitarimo ką vadinti vidurinio lygmens programine įranga. Gaunasi kaip anekdote, kad paklausus penkių žmonių gausi šešias skirtingas nuomones. Reikia atkreipti dėmesį į sudėtingą vidurinio lygmens programinės įrangos panaudojimą, jos apibrėžimo duomenų bazių bei integruotose programose peržiūras popieriuje.

Jiehan Zhou, Eila Niemelä savo straipsnyje [8 p. 2] išskyrė tokius vidurinio lygio apibrėžimus, naudojamus literatūroje:

1. *Vidurinis lygmuo.* Vidurinis lygmuo vykdomas serverio srityje ir dažnai vadinamas taikomųjų programų serveriu, kuris iš tiesų apdirba duomenis trijų lygmenų kliento/serverio architektūroje.
2. *Tarpininkas tarp duomenų bazės sistemos ir kliento taikomosios programos.* programinė įranga, reikalinga ryšiui palaikyti tarp taikomosios kliento programos ir duomenų bazės serverio, kuri visa paimta kaip grupė vadinama vidurinio lygmens programine įranga.

3. *Tarpininkas duomenų bazės sistemos ir Web serverio.* Vidurinio lygmens programinė įranga yra tokia programinė įranga, kuri jungia duomenų bazių sistemą su Web serveriu



a) Eilinė taikomoji programa

b) Kliento/serverio tipo taikomoji programa

a) Naršyklės/serverio taikomoji programa

3. pav. Su duomenų bazėmis dirbančios programinės įrangos kūrimo modeliai

Kaip pastebėjo autoriai, visi trys apibrėžimai ganėtinai panašūs. Atsižvelgiant į projekto architektūros niuansus, labiausiai vidurinio lygmens programinei įrangai apibrėžti tinka antrasis apibrėžimas. Dėl programos kūrimo specifikos vidurinio lygio programinė įranga turi būti realizuota kliento pusėje. Aukščiau parašyti apibrėžimų modeliai grafiškai atvaizduoti 3 paveiksle.

Taigi iš trečiame paveiksle atvaizduotų modelių darosi aišku, kad bet kurią programą galima išskaidyti į tris lygius, kurie yra: duomenų pateikimas, duomenų apdorojimas ir duomenų saugojimas (3 pav. a dalis). Vidurinio lygmens programinė įranga, kaip jau buvo apibrėžta anksčiau, gali būti charakterizuojama kaip tarpininkė, jungianti vartotojo sąsajas ir duomenų bazių sistemą, taip sukurdama taikomąją duomenų bazių taikomąją programinę įrangą. Tokia programinė įranga, priklausomai nuo tarpininkės, turi tokias savybes:

- Programinė įranga gali būti suskaidyta į tris lygmenis: klientą, taikomosios programinės įrangos serverį ir duomenų bazių serverį
- Kiekvienas lygmuo vykdomas skirtingoje platformoje. Tai reiškia, kad kiekvienas lygmuo gali būti kuriamas ir tvarkomas lygiagrečiai ir nepriklausomai vienas nuo kito.
- Lengviau tampa modifikuoti ar pakeisti vieną lygmenį neįtakojant kitų.
- Atskiriant taikomąją programą nuo duomenų bazių funkcijų palengvina užkrovimo balansavimo įgyvendinimą.

- Tarpininkas susideda iš dviejų dalių. Viena jų tvarkosi su duomenų bazės ryšiu bei SQL sintakse. Kita susijusi su programinės įrangos logika ir tvarkosi su duomenų apdirbimu. [8 p. 3-4]

Adaptuojant trijų lygmenų architektūrą mus domino būtent pakartotino panaudojimo bei pakeitimų realizavimo lengvumas. Iš tiesų pakeitimus lengviau realizuoti kai jie sukoncentruojami vienoje vietoje, nei ieškant po gabaliuką ką pakeisti per visą programinį kodą. Kita idėja išplaukianti iš trijų lygmenų realizavimo yra tai, kad pirmas ir trečias lygmenys neprivalo tiksliai žinoti vienas kito struktūrą, bei kiekvieną kartą pas vieną iš lygmenų jai pasikeitus keisti ją ir pas kitą. Paskyrus vidurinį lygmenų tvarkyti duomenų struktūros koregavimą iš vienam lygmeniui tinkamo į tinkamą kitam galima sumažinti pataisymų poveikį pirmam ir trečiam lygmenims, sukoncentruojant pagrindinį pakeitimų svorį viduriniajame lygmenyje.

“Integruotose taikomosiose programose vidurinio lygmens programinei įrangai galima priskirti dvi reikšmes. Visu pirma vidurinio lygmens programinė įranga yra protokolų ar standartų rinkinys. Antra, vidurinio lygmens programinė įranga yra protokolų ar standartų realizacija.” (Jiehan Zhou, Eila Niemelä, [8 p. 5]) . Tai reiškia, kad vidurinysis lygmuo gali būti standartu kuriant pirmą ir trečią lygmenis. Vidurinis lygmuo savyje saugo informaciją apie išorinių lygmenų sąsajas, jų priimamų ir naudojamų duomenų struktūras. Supaprastintai antrą reikšmę galima interpretuoti taip: vidurinis lygmuo yra išorinių lygmenų standartų realizacija. Trumpai tariant sąsajos tarp visų lygmenų turi būti standartizuotos, kad lygmenys galėtų kartu dirbti.

Objektiškai orientuotos programinės technologijos bei nustatyta objektiškai orientuota analizė (angl. *OOA*) bei programų kūrimas (angl. *OOD*) puikiai tinka trijų lygmenų serverio/kliento architektūroms projektuoti bei kurti [6, 7]. Panaudojimo atvejų reikalavimų modeliavimas labai panašus į užduotimis paremtą organizacinį modelį, aprašytą Inji Wijegunaratne ir George Fernandez [1 p. 41-77]. Realių objektų modeliavimas per klases, programos objektus bei paketus (angl. *package*) leidžia sudėtingą verslo reikalavimų rinkinį natūraliai transformuoti į loginę programinės įrangos architektūrą. Naudojant objektiškai orientuotą programų kūrimą galima išskirstyti modulius į lygmenis tiek ankstyvame kūrimo etape, tiek ir gana vėlyvame, kai gaunamos tikslios žinios apie objektų abstrakcijas. Naudojant objektinį modeliavimą kaip šalutinis produktas gaunama galimybė kiekvieną paketą laikyti potencialiu programinės įrangos komponentu pakartotiniam panaudojimui dabartinėje arba ateityje kuriamoje architektūroje [1, p. 94].

4.3 Agentais paremta transformacinė architektūra

Nagrinėjant architektūras skirtas lankstesniam pakeitimų įgyvendinimui teko susidurti ir su taip vadinama agentais paremta transformacine architektūra (angl. *agent – based transformational architecture*). Programinės įrangos architektūra identifikuoja ir aprašo programinės įrangos komponentus bei ryšius tarp jų. Jei architektūros pagrindas – nelanksti struktūra, programą tenka modifikuoti statiškai pereinant visą programinės įrangos kūrimo ciklą [10 p. 3]. Naudojantis agentais paremta architektūra pasidaro įmanomas dinaminis architektūros restruktūrizavimas.

Dinaminės architektūros idėjos buvo iškeltos 1990 m. Kramer ir Magee. Į dinaminį architektūros modifikavimą patenka:

a) Komponentų prijungimas/panaikinimas;
b) Esamų komponentų pakeitimais naujai patobulintais;
c) Sistemos pertvarkymas bei praplėtimas pridedant bei panaikinant ryšius bei komponentus, kurie tais ryšiais susiję [11 p. 2].

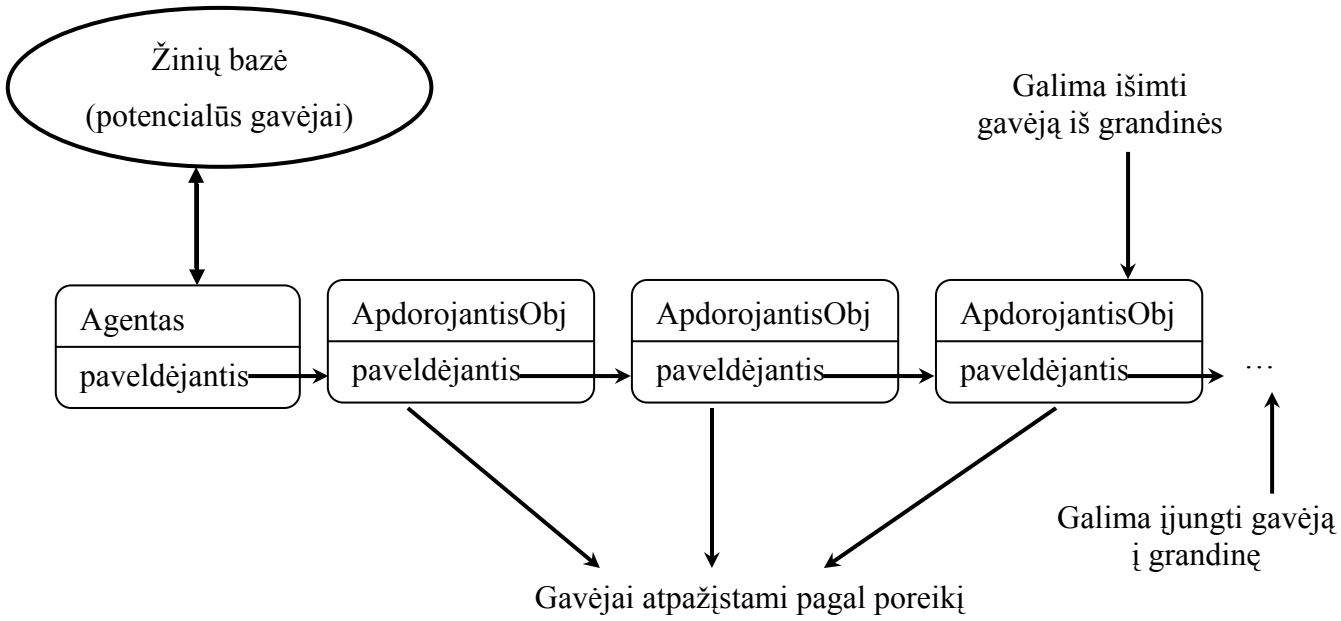
Sukuriant metodus, leidžiančius atlikti tokius veiksmus yra raktas į dinaminį architektūros pertvarkymą. Sudėtingesni pertvarkymai susideda iš apibrėžtų bazinių, bet kartais jų prireikia ir daugiau. Idėjiškai dinaminė adaptacija vyksta realiam laike priklausomai nuo programos vykdymo eigos rezultatų.

Principinė agentais ir multi–agentais paremtų sistemų (sutr. *MAS*) idėja yra tai, kad programą galima išskaidyti į mažus autonominius bei efektyvius programinius robotus. Kurie vartotojui sprendžia sudėtingus uždavinius. [10 p. 1]. Literatūroje agento apibrėžimas gana miglotas ir dažnai priklauso nuo to, kas kuria agentus ir kokiems tikslams. Agentą galima apibrėžti kaip programinės įrangos modulį, kuris patalpintas pačioje programinėje įrangoje ir kuris žino savo pareigas toje aplinkoje ir gali atlikti autoniminius veiksmus tikslui pasiekti [10 p. 1]. Šitaip apibrėžti agentai žino ką, kaip ir kada jie turi atlikti ir tam, kad agentai atliktų savo darbus nereikia tiesioginio žmogaus įsikišimo.

Agentai dažnai lyginami su objektais, objektiškai orientuotame programavime. Objektas turi būseną (arba žinias), ir veiksmus, kuriuos gali atlikti su savo būsena.agentai taip pat turi būseną, bet vietoj veiksmų jei turi pareigas, kurias privalo atlikti tam, kad pasiekti tikslą. Pagrindinis skirtumas tarp agento ir objekto yra tai, kad objektas žino tik apie save, jis turi žinoti kaip pakeisti savo būseną kai iš jo to pareikalaujama. Agentas privalo žinoti ne tik apie save, bet ir apie aplinką, kurioje jis egzistuoja, kas jam leidžia operuoti joje autonomiškai. Pagrindinis agentų privalumas prieš objektus yra tai, kad aplinka gali netiesiogiai veikti agentus. Tuo labiau agentas gali būti suprogramuotas spręsti uždavinius labiau apibendrintai.

Vietoj to kad nurodyti agentui kaip tiksliai jis turi bendrauti, jam galima sureikti žinais apie kitų agentų tikslus ir uždavinius programoje. Tada galima palikti agentui nuspręsti kokia sąveika geriausiai tinka uždaviniui išspręsti.

Sistemoje gali būti vienintelis agentas (pavyzdys – elektroninio pašto filtras) arba daugiau (pavyzdžiu gali būti vaidmenų žaidimai kur kiekvienas veikėjas gali būti sukurtas kaip agentas). Multi–agentų sistemos (*MAS*) tai agentų tinklas kur agentai bendrauja tarpusavyje spręsdami uždavinius, kurių atskirai būdami išspręsti nesugebėtų. Naudojant agentus galima sukurti lankstesnę architektūrą, kuri leistųsi modifikuojama vykdymo metu. Tai padidintų programinės įrangos pakartotinį panaudojamumą bei pakeitimų joje darymą. Kuriant tokią architektūrą reiktų naudoti atitinkamus kūrimo šablonus (angl. *design patterns*) bei genetinių algoritmų metodiką (angl. *genetic algorythms*).



4. pav. Grandininės atsakomybės transformacinis šablonas

Transformaciniai šablonai (angl. *Transformational patterns*) tai kūrimo šablonai praplėsti taip, kad padėtų programinei įrangai adaptuotis pagal poreikius. Jie buvo įvesti kartu su agentų technologija. Agentų bei transformacinių šablonų panaudojimo pavyzdys gali būti agento panaudojimas grandininės atsakomybės šablone (angl. Chain responsibility pattern). Šis šablonas neleidžia susidaryti situacijai, kai užklausa paduodama apdoroti iškart keliem objektam. Jis sujungia objektus – gavėjus į grandinę ir paeiliui leidžia užklausimą kol objektas jį apdoroja. Jei eilę reikia kaip nors pakeisti, reikia įdėti arba išimti kokį objektą, tai turi padaryti programinės įrangos kūrėjas, bei pati programinė įranga po to turi būti perkompiliuota. 4 paveiksle pavaizduota transformacinė šablono versija. Joje agentas stebi

eile siunčiamus užklausimus grandine. Kaupiami duomenys apie tai, kaip dažnai gavėjai apdoroja užklausą bei užklausų kiekį, kuriu grandinė nesugeba apdoroti. Vienoje šablono versijoje agentas yra atsakingas už gavėjų grandinės pergrupavimą taip, kad gavėjai, kurie užklausas apdoroja dažniau, būtų atkeliami į grandinės pradžią ir taip sumažinant užklausų persiuntimo skaičių. Kitoje versijoje agentas gali konsultuotis su potencialių gavėjų žinių baze, kurių kiekvienas patenka į kategoriją kaip galintis apdoroti įvairias užklausas. Agentas prijungia naują gavėją kai aptinka, kad grandinėje esantys objektai negali apdoroti užklausos, bei išima iš grandinės objektus, kurie grandinėje nenaudojami.

4.4 Architektūros pasirinkimas bei planuojami jos privalumai

Trečiame skyriuje buvo išsiaiškinta kad pakeitimų darymo sudėtingumą bei jiem užtrunkamą laiką įtakoja du veiksniai: kodo suprantamumas, kurį išmatuoti sudėtinga, galima tik nustatyti empyriškai pagal tai, kaip naudojami komentarai bei kokio sudėtingumo algoritmai naudojami, bei programos architektūra. Pastarąją planuoti paprasčiau ir galima jai kelti tam tikrus reikalavimus, kurių laikantis suprogramuotai programinei įrangai ateityje bus lengviau daryti pakeitimus bei pataisymus.

Iš plačiai pasaulyje pripažintų architektūrų agentais paremta architektūra atrodo sudėtingesnė ir reikalaujanti gerokai daugiau laiko suprojektuoti bei realizuoti, jau nekalbant apie tai kad reiks naudoti papildomus priedus jos realizavimui. Antra vertus pereinant iš dviejų lygmenų architektūros į trijų lygmenų nemažą dalį kodo bus galima panaudoti iš naujo, nerašant visos programos nuo pradžių iš naujo. Teisingai suplanavus trijų lygmenų architektūrą galima bus sugrupuoti panaudojimus į grupes, kurios bendraus per nustatytas sąsajas ir taip sumažins tam tikrai funkcijai atlikti reikalingų kodo eilučių išsibarstymą klasėse. Pakeitimas paveiks mažesnį kodo gabalą ir atatinkamai sumažės darbas bei laikas sunaudojamas jam atlikti.

4.5 Apibendrinimas

Duomenų struktūros laukų panaudojimo atvejų kiekį retai pavyksta sumažinti taip, kad nenukentėtų programos funkcionalumas. Vienintelis kitas faktorius, leidžiantis sutrumpinti duomenų struktūrai pakeisti sunaudojamą laiko kiekį yra koncentruoti duomenų struktūros laukų panaudojimo atvejus bei krepinius į duomenų struktūros laukus.

Naudojantis dviejų lygmenų architektūra koncentruoti panaudojimo atvejus sudėtinga, nes abu lygmenys privalo tiksliai žinoti vienas kito naudojamą duomenų struktūrą, ir paprastumo dėlei ji padaroma identiška. Tai padidina duomenų panaudojimo atvejų pasiskirstymą, nes patogumo dėlei lygmenys apsikeitinėja didesniais duomenų blokais.

Nagrinėjant trijų lygmenų architektūrą randamos tokios savybės, kurios leidžia labiau koncentruoti duomenų struktūros laukų panaudojimus:

- Lygmenų atskyrimas sumažinant jų įtaką vienas kitam ir bendraujant tarpusavyje per nustatytas sąsajas. Tai leidžia pakeisti vieną lygmenį minimaliai įtakojant kitus lygmenis, todėl pakeitimai koncentruojasi viename lygmenyje.
- Išoriniai lygmenys bendrauja tarpusavyje per vidurinį lygmenį, todėl vienas išorinis lygmuo neprivalo tiksliai žinoti kito lygmens naudojamą duomenų struktūrą. Tokiu atveju išoriniai lygmenys gali turėti skirtingą duomenų struktūrą. Vidurinis lygmuo transformuoja vieno lygmens naudojamą duomenų struktūrą į tinkamą kitam lygmeniui. Pasitaiko atvejų kai būtent šita savybė stipriai sumažina laiką sunaudojamą pakeitimui atlikti.
- Išskiriami loginiai lygmenys susiję su duomenų struktūros saugojimu, duomenų apdirbimu bei duomenų pateikimu vartotojui. Toks išskyrimas leidžia labiau sugrupuoti duomenų laukų panaudojimą tam tikruose paketuose.

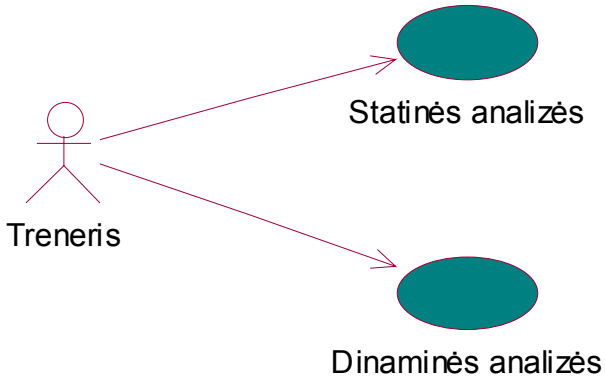
Nagrinėjant literatūra rastas patvirtinimas kad trijų lygmenų architektūra gana lengvai realizuojama naudojanti objektiškai orientuotomis programavimo kalbomis, duomenų lygmenų išskyrimas galimas ne vien tik ankstyvose kūrimo fazėse, todėl pereiti nuo dviejų lygmenų architektūros prie trijų lygmenų nereikalaus viso programos kodo perrašymo. Šalutinis trijų lygmenų architektūros produktas – padidinta programos paketų pakartotinio panaudojimo galimybė.

5 Trijų lygmenų architektūros realizacija ir įvertinimas

Prieš pradėdamas nagrinėti trijų lygmenų architektūros realizaciją paskirstytoje krepšinio rungtynių registravimo bei analizės sistemoje, trumpai nušviesiu krepšinio rungtynių analizės dalies reikalavimus ir darbo specifiką. Būtent analizės dalyje ryškiausiai pasireiškia trijų lygmenų architektūra.

5.1 Sistemos analizės dalies reikalavimai bei darbo specifika

Aptariama sistemos analizės dalis skirta registravimo metu surinktus duomenis apdoroti ir pateikti treneriui. Kadangi sistema paskirstyta, treneris gali rezultatus gauti operatyviai, vykstant rungtynėms, ir tuo netrukdydamas registruotojo(ų) darbo.



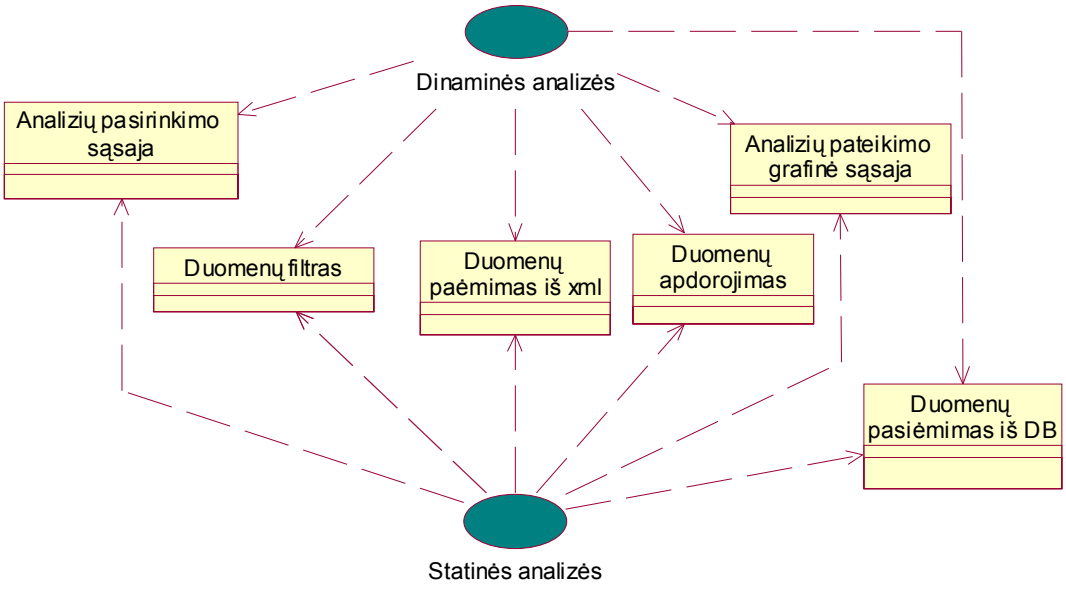
5. pav. Analizių panaudojimo atvejų diagrama

Kaip matome 5 paveikslėlyje, gaunamos analizės grubiai imant gali būti suskirstytos į dvi grupes: statines ir dinamines analizes. Statinės analizės atvaizduoja rungtynes neatsižvelgdamos į parametrų kitimo specifiką rungtynių eigoje. Klasikinės statinės analizės – įmestų taškų kiekis, atkovotų kamuolių kiekis, baudų kiekis ir panašiai. Dinaminės statistikos gi parodo kaip kito trenerį dominantys parametrai visoje rungtynių eigoje. Nors statinės analizės nuo dinaminių skiriasi pradinių duomenų poreikiu, skaičiavimų algoritmais bei atvaizdavimo specifika, principinis jų darbo modelis yra vienodas.

Nagrinėjant šeštame paveikslėlyje pavaizduotoje diagramoje, tiek statinės, tiek ir dinaminės analizės turi tuos pačius principinius veikimo blokus:

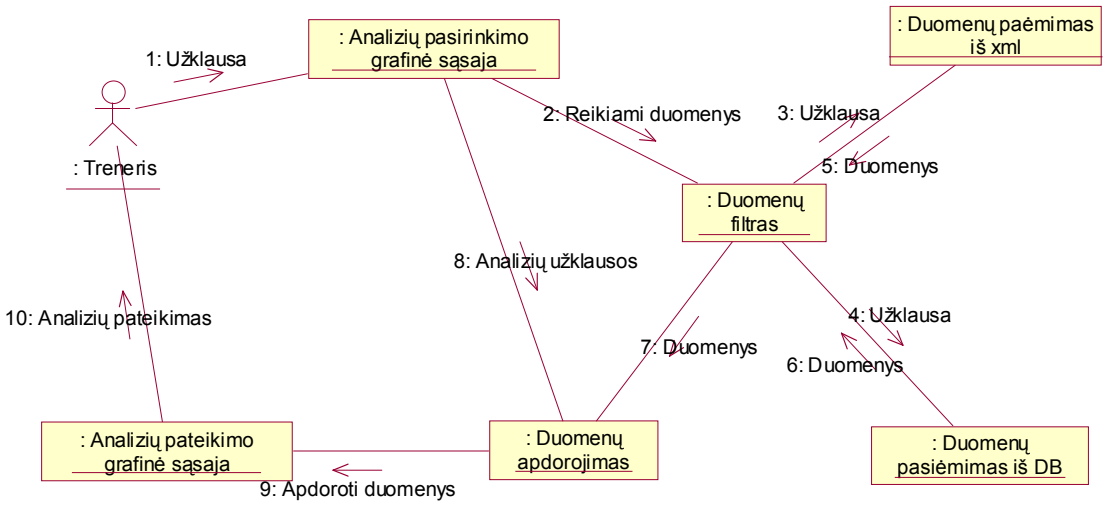
- Grafinę vartotojo sąsają, skirtą pasirinkti norimoms analizėms;
- Duomenų, reikiamų užsakytoms analizėms atrinkimą;
- Atrinktų duomenų paėmimą iš duomenų saugyklos, kuri gali būti tiek rungtynių protokolas įvestas į duomenų bazę, tiek ir išsaugotas XML formatu faile;

- Duomenų apdorojimą siekiant gauti treneriui reikalingas analizes;
- Gautų analizių grafinį pateikimą treneriui.



6. pav. Panaudojimo atvejų klasių diagrama

Plačiau duomenų srautus bei užsakytų analizių apdorojimo kelią demonstruoja septintame paveiksle pavaizduota sekų diagrama.



7. pav. Sekų diagrama

Iš diagramos matosi, kad su duomenų saugykla tiesiogiai tenka bendrauti tik duomenų filtrui, kuris privalo duoti užklausą reikiamiems duomenims. Klientas pageidavo, kad su

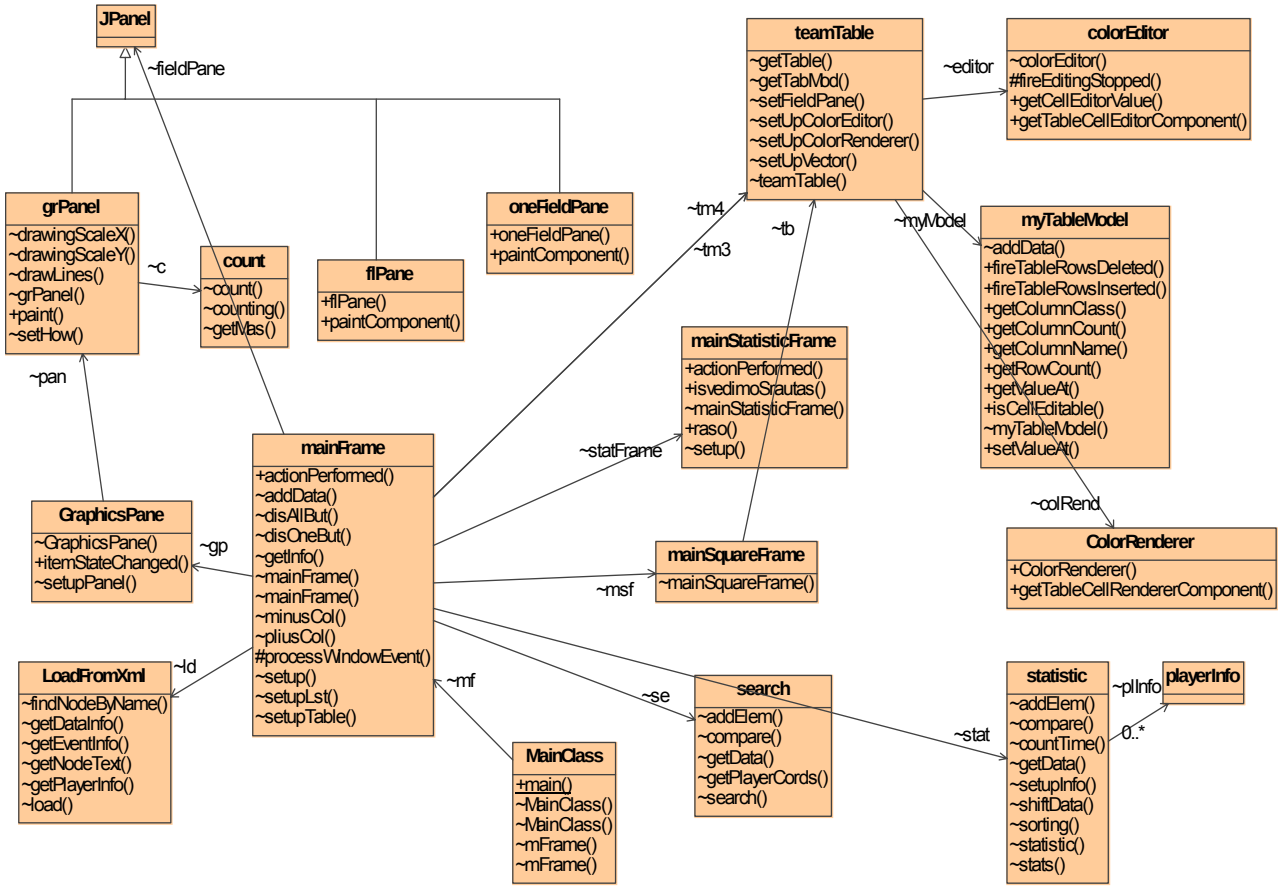
programa būtų galima dirbti net jei nutrūksta ryšys su centrine duomenų baze, ir vos ryšiui atsistačius, būtų automatiškai atnaujinami duomenys duomenų bazėje bei kliento kompiuteryje. Tuo pagrįstas duomenų pasiėmimo klasių išskyrimas į pasiėmimą iš duomenų bazės ir pasiėmimą iš XML failo. Be to duomenų saugykla XML formatu faile – programos kurią reikėjo perdaryti į paskirstytą sistemą palikimas. Turint šią principinę diagramą jau galima numatyti lygmenų pasiskirstymą bei duomenų srautus tarp jų. Į pirmą lygmenį – duomenų saugyklos apdorojimą patenka duomenų paėmimo iš DB ir duomenų paėmimo iš XML klasės. Antrame lygmenyje – *middleware* – būtų klasės duomenų filtras ir duomenų apdorojimas. Galų gale vartotojo sąsajos lygmenyje lieka dvi klasės – analizių pasirinkimo grafinė sąsaja bei analizių pateikimo grafinė sąsaja.

Kyla rimtas klausimas ar duomenų filtro nereiktų perkelti į pirmą lygmenį, nes tai taip pat susiję su duomenų paėmimu, bet tada gautųsi situacija, kurioje pirmas lygmuo tiesiogiai bendrauja su trečiu. Tai yra suardomas pats trijų lygmenų principas ir grįžtama prie pradžioje realizuotos architektūros. Schemoje toks padalinimas neatrodytų labai svarbus ar daug lemiantis, bet realizuojant architektūrą tai gali būti esminiu niuansu. Taip pat galima atskirti preliminarius srautus tarp lygmenų. Tokie srautai būtų 2, 8, 9 tarp trečio ir antro lygmens bei 3, 4, 5, 6 tarp pirmo ir antro. Norint sumažinti jų kiekį tarp lygmenų būtų galima aštuntą srautą perduoti per duomenų filtrą arba antrą srautą siųsti kartu su aštuntu, o antro lygmens viduje sudaryti dvipusį srautą. Tokiu atveju duomenų filtro klasę galima būtų perkelti į pirmą lygmenį nesuardant trijų lygmenų architektūros principo. Taip pat galima pastatyti klasę, kuri tvarkytų 3, 4, 5, 6 srautus ir juos sutvarkiusi bendrautų su antru lygmeniu. Taip informacijos srautų su pirmu lygmeniu sumažėtų perpus.

5.2 Buvusios dviejų lygmenų architektūros apžvalga

Šiame skyriuje bus apžvelgiama realizuota paveldėtosios sistemos dviejų lygmenų architektūra. Aštuntame paveiksle pavaizduotoje diagramoje parodomas visas analizės dalies vaizdas. Analizės dalis buvusioje sistemoje buvo projektuota kaip atskira nuo registravimo dalies, iškvietus analizių langą būdavo sukuriama iki tol suregistruotų duomenų kopija, kurią naudodavo analizės programa. Analizės dalis nereaguodavo į naujai užregistruotus įvykius, norint kad jie atsispindėtų analizėje tekdavo uždaryti analizių langus ir vėl užsakyti analizes iš naujo. Kol programa nebuvo paskirstyta, tai nebuvo didelis trūkumas, nes nebuvo įmanoma vienu metu stebėti analizių ir atlikti rungtynių registravimo. Kuriant paskirstytąją sistemą tai jau tapo dideliu kliuviniu.

Nagrinėjant pavaizduotą architektūrą darosi akivaizdu, kad visi analizėms reikalingi apskaičiavimai atliekami grafinio duomenų pateikimo lygmenyje. Užkrovimo iš duomenų saugyklos vaidmenį atlieka *LoadFromXml* klasė, ir ji galėtų būti pavadinta pirmo lygmens klase. Visa kita tenka pavadinti antru lygmeniu, kadangi neįmanoma išskaidyti ją atskiriant grafinio pateikimo vartotojui dalį nuo skaičiavimų. Klasė *statistic* yra skirta asmeninių žaidėjo statinių parametrų skaičiavimui, bet duomenis jei perduoda klasė, skirta analizių lango sukūrimui bei valdymui – klasę *mainFrame*, o apskaičiuoti rezultatai į klasę, atsakingą už jų grafinį pateikimą – *teamTable* bei *teamTableModel* klases – taip pat perduodami per lango klasę. Pats rungtynių protokolas irgi saugomas lango klasėje, rezultatai – *myTableModel* klasėje, o žaidėjų sąrašams saugoti jau išskiriama atskira klasė. Toks duomenų struktūros panaudojimo išmėtymas apsunkina pakeitimų darymą bei jų testavimą.



8 pav. Krepšinio rungtynių analizių dviejų lygmenų architektūros klasių diagrama

Architektūra visiškai nepritaikyta atidaryti keletą analizių skirtinguose languose, kas svarbu treneriui, nes jis nori matyti keletą analizių ir greit perjungti iš vienos analizės į kitą, o ne užsakinėti norimas analizes iš naujo. Architektūra turi sudėtingus vidinius sąryšius, o pabandžius modifikuoti pavaizduotą architektūrą prijungiant naują analizės tipą bei jei reikia u¸registruoti jai reikalingą naują duomenų lauką, jos sudėtingumas dar labiau išauga.

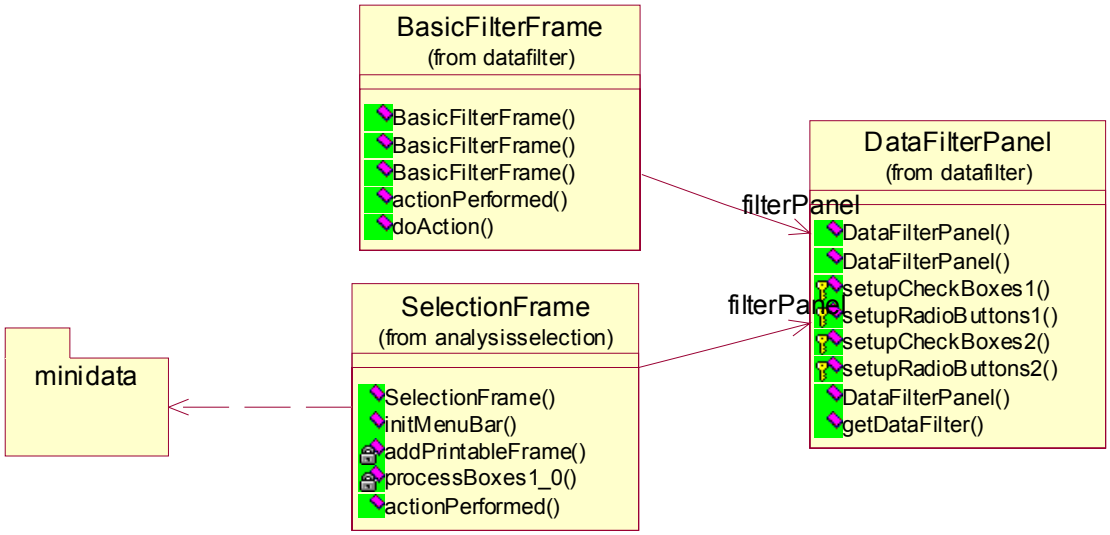
Atliekant pirmus pakeitimus paaiškėjo, kad paveldėta architektūra neatspari ne tik duomenų struktūros pasikeitimams, ji apskritai neatspari jokiems joje vykdomiems pakeitimams. Tiek analizės, naudojančios iki tol nenumatytą duomenų lauką, prijungimas, tiek rungtynių protokolo stebėjimo realizavimas tokioje architektūroje buvo labai sudėtingas. Viena iš tokio sudėtingumo priežasčių – neįmanoma tiksliai žinoti kokioms klasėms pakeitimas turės poveikį, o kokioms – ne. Kita priežastis – sunku nagrinėti duomenų struktūros panaudojimą, kai didelė dalis kreipinių į struktūrą bei analizių skaičiavimams skirto kodo susimaišę su kodu, skirtu grafiniam jų rezultatų pateikimui. Sudėtingiausias ir daugiausiai laiko atimantis darbas atlikus pakeitimą – testavimas, kadangi pakeitimas paveikdavo beveik visas klases, o integracijos testavimas kartais pateikdavo tiesiog stulbinančių siurprizų

Išnagrinėjus architektūrą pasidaro aišku, kad norint užtikrinti programinės įrangos atsparumą pakeitimams, nepaisant to, pakeitimas susijęs su duomenų struktūros pakitimu ar ne, tenka kurti visiškai naują architektūrą, nes paveldėtoji yra nelanksti ir bet koks pakeitimo atlikimas labai ją gadina, tuo mažindamas programinės įrangos patikimumą bei apsunkindamas tolimesnių pakeitimų atlikimą.

5.3 Sistemoje realizuotos architektūros aprašymas

Trumpai išsiaiškinus kokį uždavinį teko realizuoti, galima gilintis į trijų lygmenų architektūros niuansus realizuotoje sistemoje. Nagrinėjimą pradėsiu nuo trečio – vartotojui matomo lygmens ir leisiuosi iki pirmo. Klasių diagramose nerodomi klasių atributai bei metodų parametrai. Tai padaryta norint sumažinti diagramų apimtis bei padidinant jų aiškumą.

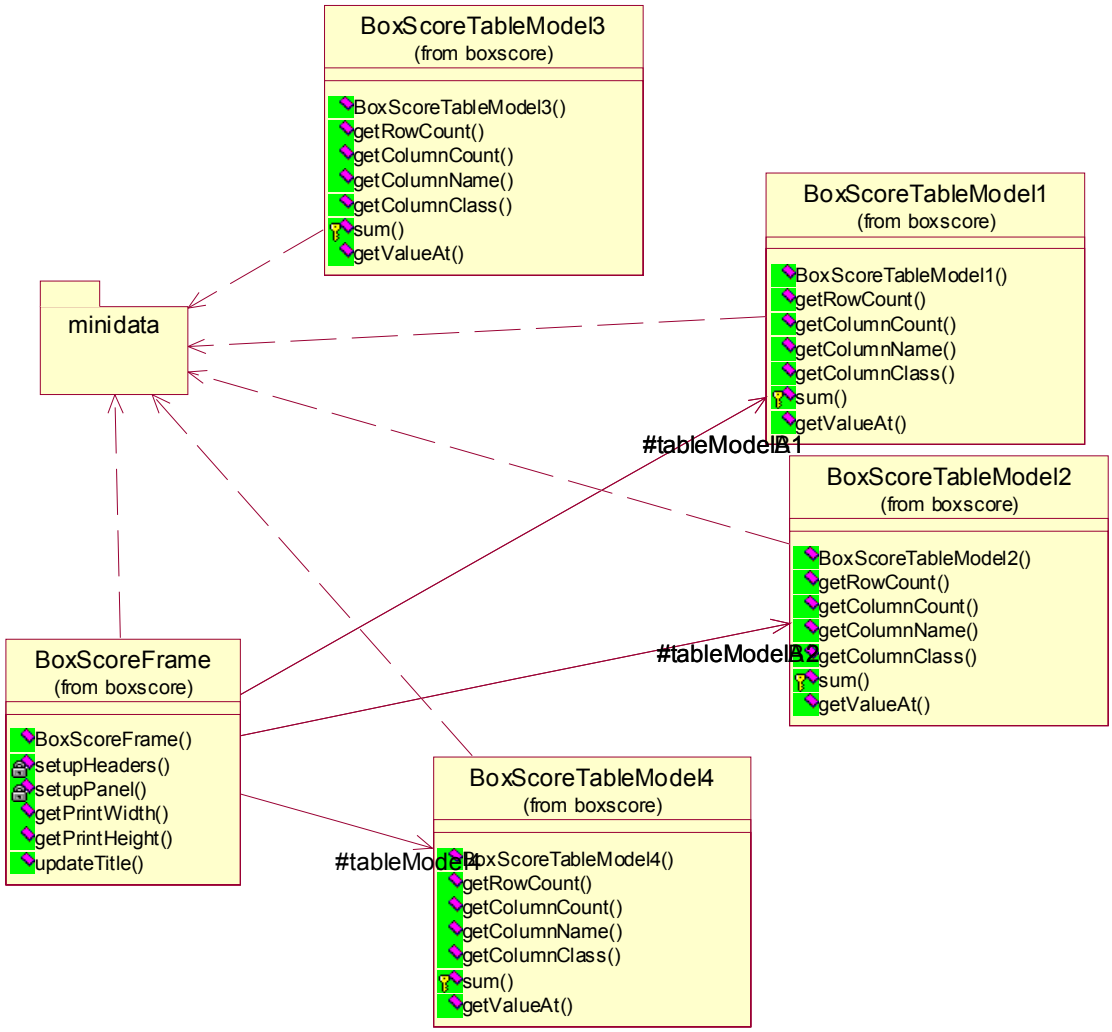
Taigi devintame paveikslėlyje pavaizduotoje klasių diagramoje matoma duomenų filtro grafinės vartotojo sąsajos realizaciją. Ryšys su paketu *minidata* rodo ryšį su antro lygmens duomenų apdorojimo klasių rinkiniu. Tokia architektūra buvo priimta norint duomenų filtrą perkelti į pirmą lygmenį. Dėl to kilo daug ginčų. Visgi galutiniame variante duomenų filtras buvo priskirtas antram lygmeniui, kadangi jis kartu turėjo ir stebėti užsakytų analizėms duomenų laukų pasikeitimus bei siųsti užklausimą perskaičiuoti jas, jei duomenys pasikeistų ar būtų įvesti nauji. Tai reiškia jis turėjo transformuoti pirmo lygmens duomenų struktūrą į antram lygmeniui reikalingą. Ryšys su antru lygmeniu vyksta išskirtinai tik per *SelectionFrame* klasę. *DataFilterPanel* skirta pagrindiniam tvarkymuisi su grafine vartotojo sąsaja, ji taip pat formuoja vartotojo pasirinktų analizių sąrašą. *BasicFilterFrame* klasė daugiausia skirta padoroti įvykius (angl. *event*).



9. pav. Duomenų filtro grafinė sąsaja, trečias lygmuo

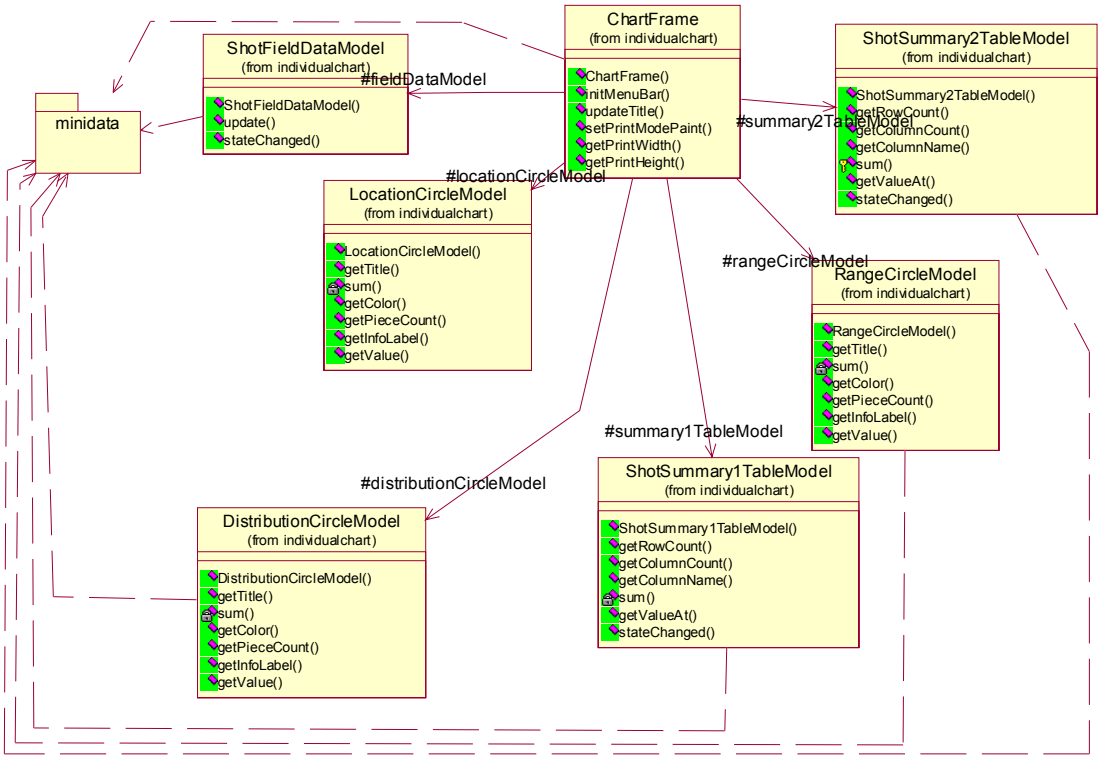
Dešimtame paveiksle pavaizduotoje diagramoje matoma statinės analizės atveju gautų rezultatų lentelių pateikimo vartotojui klasių rinkinį. Kaip ir buvusioje diagramoje, ryšiai su *minidata* paketu rodo ryšį su antru lygmeniu – duomenų apdorojimu. Klasė *BoxScoreFrame* atsakinga už rezultatų grafinį pateikimą į freimą. Be jos neįmanoma pateikti rezultatų. Keturios *BoxScoreTableModel* klasės formuoja duomenų lenteles, gautas iš antrame lygmenyje atliktų skaičiavimų. Priklausomai nuo to, kokias analizes užsako treneris pateikiami rezultatai suskirstyti į keturias atvaizdavimo grupes, todėl gaunasi keturios lentelių modelių klasės. Jos formuoja lentelių vaizdą bei reikšmes *BoxScoreFrame* objekte. Kiekvienas iš lentelių modelių gauna duomenis savo laukams iš *minidata* paketo (tai yra iš antro lygmens). *BoxScoreFrame* iš *minidata* paketo gauna tik nuorodą, kokį duomenų lentelių modelį naudoti. Lentelių modeliai savo metodais atrodo identiški. Iš tiesų jų skirtumai pasireiškia būtent saugomais viduje atributais (angl. *attribute*), o tai reiškia laukais, kurie stebimi ir perduodami analizių perskaičiavimui.

Taigi pakeitus skaičiavimus vieno tipo analizėje, prireikus naujų laukų analizei apdoroti pakanka trečiame lygmenyje pakoreguoti atitinkamai dominantį lentelių modelį pagal tai, kokius duomenis jai turi paduoti *minidata* klasių paketas, visiškai neįtakojant *BoxScoreFrame* klasės. Taip pat paprasta pridėti naują analizių lentelių modelį – tereikia užregistruoti jį *minidata* pakete, kad siųstų duomenis reikiamus duomenis į naują lentelių modelį bei informuotų *BoxScoreFrame* kokį lentelių modelį naudoti. Tai reikalinga kai analizei reikia duomenų rinkinio, kuris dar nebuvo numatytas.



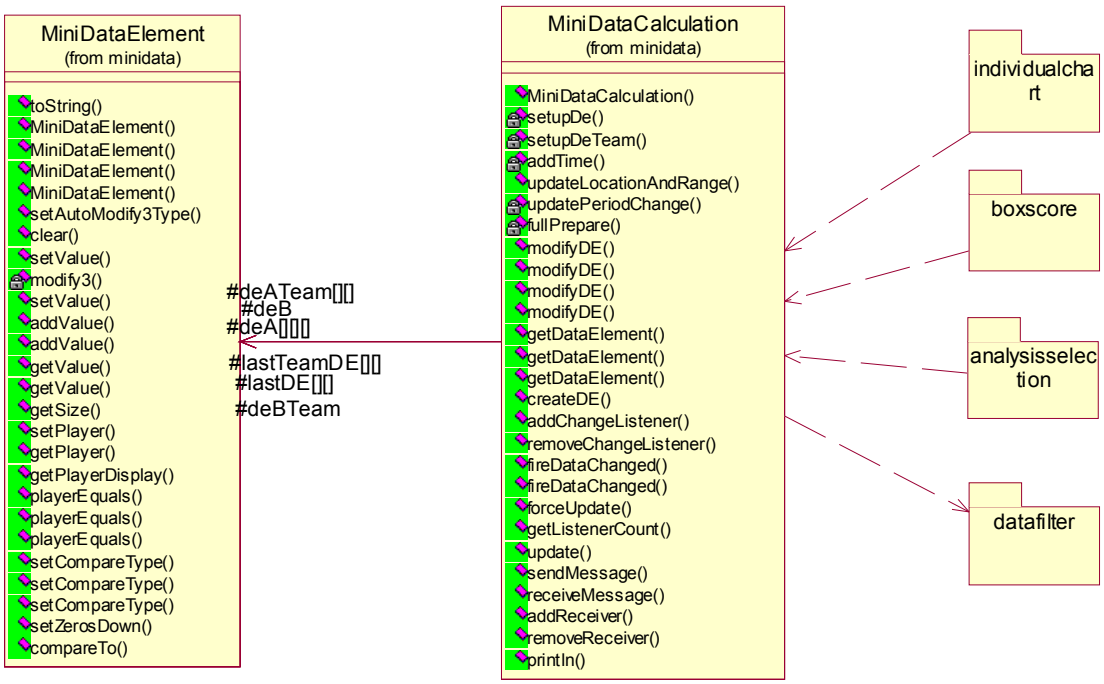
10. pav. Analizėse gautų rezultatų lentelių atvaizdavimas, trečias lygmuo

Vienuoliktame paveiksle pateikta diagrama savo struktūra labai panaši į aukščiau aprašytą. Joje *ChartFrame* klasė atlieka panašų darbą, kaip aukščiau pateiktoje diagramoje atliko *BoxScoreFrame* – atsakinga už atvaizdavimo lango sukūrimą, jo perpiešimą ir panašias funkcijas. *ShotSummaryTableModel* klasių funkcijos atitinka *BoxScoreTableModel* klasių funkcijas aštunto paveikslėlio diagramose. Visi kiti modeliai atsakingi analizių rezultatų pateikimą grafiškai, o ne lentelių pavidalu. Pavyzdžiui klasė *LocationCircleModel*, *DistributionCircleModel* bei *RangeCircleModel* atsakingi už rezultatų pateikimą per skritulines diagramas, o *ShotFieldDataModel* grafiškai rodo metimų dislokacijos vietas. Architektūros privalumai duomenų struktūros pasikeitimo atveju identiški aprašytiems nagrinėjant aštuntame paveiksle aprašytą diagramą.



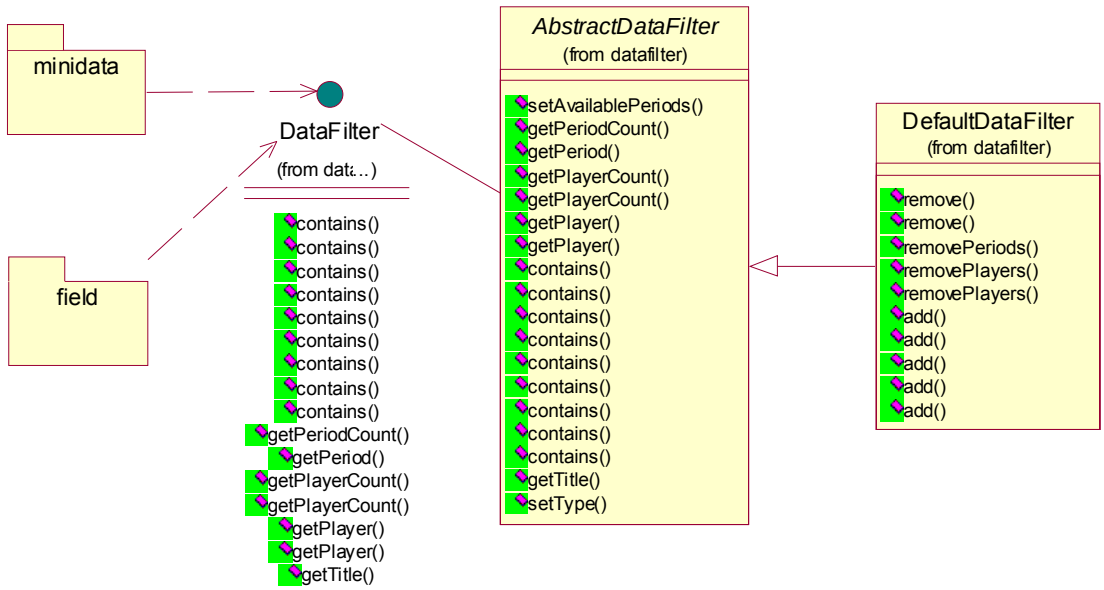
11. pav. Analizėse gautų rezultatų lentelių bei grafikų atvaizdavimas, trečias lygmuo

Dvylikto paveikslo diagramoje jau vaizduojama antro lygmens architektūra. Ryšiai su trečiu lygmeniu pavaizduoti ryšiais su paketais *individualcharts* bei *boxscore* – tai ryšiai su grafiniu apdorotų duomenų atvaizdavimu – bei ryšiu su paketu *analysisselection* – tai ryšys su analizių pasirinkimo langu. Ryšys su paketu *datafilter* yra ryšys vidinis antro lygmens ryšys su duomenų filtru. *MiniData* paketas klasės apdirba visas statines analizes. Glausta dinaminių analizių apdorojimo architektūra bus pateikta kiek vėliau. Jos architektūra idėjiniu požiūriu nesiskiria nuo statinių analizių architektūros. Užsakytos analizės keliauja per *MiniDataCalculation* klasę, kuri užsako reikiamus duomenis iš duomenų filtro, o paskui apdorojusi pateikia atitinkamiems analizės paketams. Praktiškai visi analizių skaičiavimai susikoncentravę šioje klasėje. Atitinkamai *MiniDataCalculation* klasė dar koreguoja ir sinchronizaciją su atvaizdavimo lygmeniu, formuoja įvykį kada rezultatai pasikeitė ir reikia perpiešti analizės atvaizduojamus duomenis. Pridedant naują analizę reikia pridėti atatinkamai naują metodą, o taisant – perrašyti seną metodą negriaunant kitų.



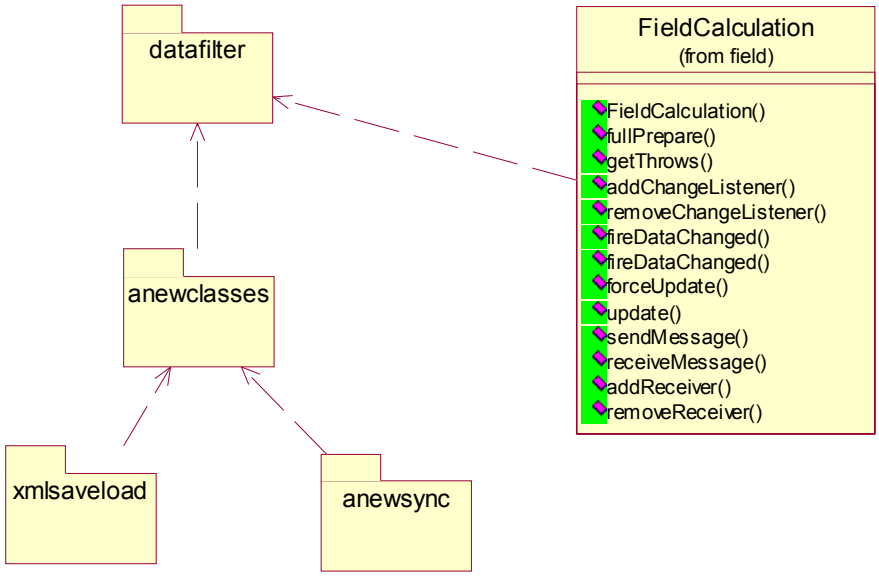
12. pav. Analizių skaičiavimai, antras lygmuo

Tolimesnė antro lygmens architektūra vaizduojama trylikto paveikslo pateikiamoje diagramoje. Kaip jau buvo minėta anksčiau, duomenų filtras pradžioje buvo projektuojamas kaip pirmo lygmens narys, taigi tuo pasireiškia *minidata* bei *field* paketų ryšys su duomenų filtru per sąsają (angl. *interface*).



13. pav. Duomenų filtras, antras lygmuo

Keturioliktame paveiksle tęsiamas duomenų filtro ryšių pavaizdavimas. Jame daugiausia vaizduojamas ryšys su pirmu lygmeniu. Pirmą lygmenų programavo mano partneris, todėl aš detaliai jo nenagrinėsiu, tiesiog aprašysiu bendrai. Tryliktame paveiksle demonstruojamas filtro ryšys su *minidata* buvo aptartas anksčiau. Ryšys su paketu *field* detalizuojamas dvyliktame paveiksle, *FieldCalculation* klase. Šita klasė atsakinga už stebėjimą kada pasikeičia stebimi duomenys ir duoda signalą perskaičiuoti analizių rezultatus. Pakete *anewclasses* saugomi visi varžybų duomenys, tame pačiame pakete yra klasės skirtos duomenų paėmimui iš DB bei padėjimui į ją. Paketas *anewsync* skirtas susinchronizuoti programos darbą su duomenų baze, paprastai reikalingas tik pradžioje pasirenkant duomenų bazę, ir nenutrūkstamo darbo užtikrinimui. Paketas *xmlsaveload* skirtas duomenų saugojimui bei užkrovimui į/iš XML failo. Ryšys tarp *datafilter* ir *anewclasses* yra ryšys tarp pirmo ir antro lygmens.

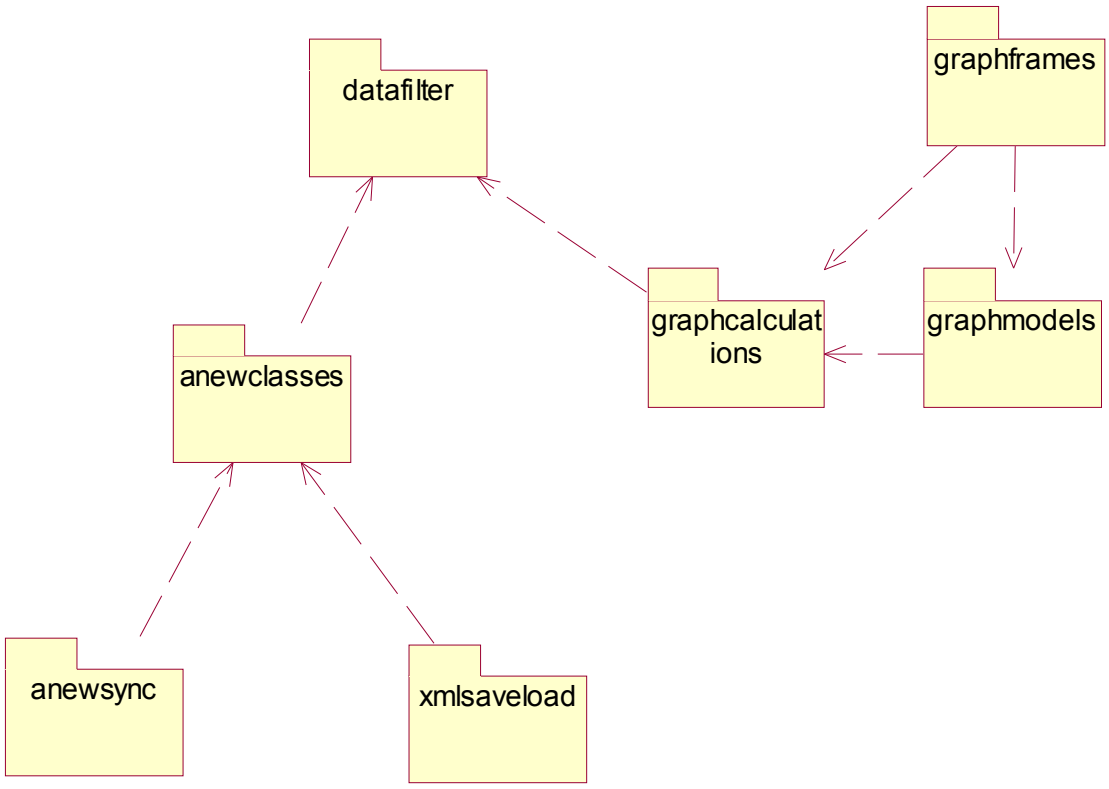


12. pav. Duomenų paėmimas iš duomenų saugyklos, pirmas lygmuo

Iki šiol buvo nagrinėta tik statinių analizių architektūra. Dėl perskaičiavimo specifikos dinaminių analizių skaičiavimai skiriasi nuo skaičiavimų, atliekamų vykdant statines analizes. Visgi principinė dinaminių analizių architektūra panaši į statinių. Lygiai taip pat išskiriami trys lygmenys. Principinė dinaminių analizių architektūra parodyta tryliktame paveiksle. Kaip ir statinėse analizėse galima išskirti lygmenis:

- Į pirmą lygmenį patenka paketai *anewsync*, *anewclasses* ir *xmlsaveload*. Tai visiškai sutampa su statinių analizių pirmu lygmeniu. Ryšį tarp pirmo ir antro lygmens apdirba *datafilter* paketas, taip pat kaip ir statinių analizių atveju;

- Į antrą lygmenį patenka paketai *datafilter* ir *graphcalculations*. *Datafilter* paketas identiškas aprašytam nagrinėjant statines analizes. Paketas *graphcalculations* savo struktūra panašus į *minidata*. Vienintelis esminis skirtumas – jis neapdoros filtrų analizių pasirinkimo. Dinaminės analizės taip pat pasitenkamos per aukščiau aprašytą mechanizmą ir kokią analizę reikia atlikti perduodama jau per duomenų filtrą.
- Į trečią lygmenų patenka paketai *graphframes* ir *graphmodels*. Jų veikimo struktūra labai panaši į statinių analizių trečią lygmenį.



15. pav. Dinaminių analizių architektūra, bendras principinis vaizdas

Apibendrinus, penkioliktame paveiksle pademonstruota diagrama su nedideliais pakeitimais išreiškia ir statinių analizių principinę architektūrą, antra vertus statinių analizių detali klasių architektūra labai panaši į dinaminių, todėl nėra prasmės pateikti detalios dinaminių analizių architektūros bei principinės statinių analizių diagramos. Tai būtų bereikalingas kartojimasis.

5.4 Trijų lygmenų architektūros realizacijos įvertinimas

Realizavus 5.2 skyriuje aprašytą architektūrą, pagrindinis duomenų struktūros panaudojimo atvejų kiekis buvo koncentruotas antrame lygmenyje. Atliekant analizes, gauti

rezultatai nesaugomi duomenų saugyklose, todėl pirmą lygmenį mažai įtakoja naujos analizės įdėjimas arba senosios pataisymas. Pirmame lygmenyje daugiausia taisymo u˛ima pats rungtynių protokolo registravimas. Duomenų registravimo man projektuoti neteko, todėl neprašinėsiu jo specifikos. Paminėsiu, kad jį realizuojant taip pat galima taikyti trijų lygmenų architektūros principą. Pirmame lygmenyje duomenų struktūros pasikeitimai pagrindinai įtakoja tik *anewclasses* paketo struktūrą. Pagrindiniai pakeitimai koncentruoti antrajame lygmenyje. trečią lygmenį tai taip pat silpnai įtakoja.

Kadangi daugiausia teko atlikti naujų analizių įvedimą, pakeitimų principinis atlikimas būdavo toks: jei analizė reikalaudavo naujo duomenų lauko, atitinkamoje klasėje *anewclasses* pakete įvedamas naujas atributas(jei tai laukas, kurio niekada, jokiose analizėse nenaudojo, pavyzdžiui registruojamas visiškai naujas parametras), bei paėmimo metode nurodomas duomenų bazės lauko pavadinimas iš kur imti. Tada sutvarkomas antras lygmuo. Tada reikia užfiksuoti stebėjimą duomenų filtre bei sukurti atitinkamus metodus, apdorojančius naują duomenų lauką. Projekto darymo metu didesnė dalis pakeitimų buvo susiję su naujo funkcionalumo įvedimu. Tai reiškia naujos analizės įvedimu. Po to kai skaičiavimai jau sukurti tenka sukurti atvaizdavimui sukurtą klasę, architektūroje paprastai tai klasė – modelis. Statinių analizių atveju, jei reikšmė patenka į kurią nors statinę lentelę, užtenka tiesiog modifikuoti lentelės klasę – modelį.

Praktikoje tokios architektūros naudojimas stipriai palengvino bei pagreitino ne tik pakeitimų, susijusių su duomenų struktūros pakitimais realizavimą, bet ir paprastų programinių pakitimų atlikimą. Programos architektūra tapo lengviau modifikuojama.

Kaip trūkumą tenka paminėti padidėjusį pradinį architektūros sudėtingumą lyginant su dviejų lygmenų architektūra. Atsiranda daugiau klasių bei sąryšių tarp jų. Visgi, programoje suprojektuotoje naudojant trijų lygmenų architektūrą, pakeitimai gerokai mažiau gadina iki pakeitimų sukurtą architektūrą, taigi po tam tikro pakeitimų skaičiaus dviejų lygmenų architektūros sudėtingumas pralenkia trijų lygmenų.

6 Išvados

1. Reikalavimų pakeitimus įvertinanti architektūra yra aktuali problema šiuolaikiniame programų kūrime.
2. Ištirtos tokios architektūros: dviejų lygmenų, trijų lygmenų, agentais patemtos ir jų taikymo sritys.
3. Projektui realizuoti pasirinkta trijų lygmenų architektūra, kadangi ji įvertina reikalavimų pakeitimus, ir projektas nereikalauja savybių, kurių realizavimui reikalinga agentais paremta architektūra.
4. Eksperimentiškai ištirtos dviejų ir trijų lygmenų architektūros. Trijų lygmenų architektūra pasirodė esanti efektyvesnė pakeitimų realizavimo požiūriu. Pavyzdžiui nustatyta, kad, pridedant naują statinę analizę dviejų lygmenų architektūroje įtakojamos 8 iš 16 klasių (50%), trijų lygmenų architektūroje įtakojamos 6 klasės (7-8, jei reikia užsakyti iki tol nenaudotą duomenų rinkinį) iš 20 klasių (30-40%) (jei neskaičiuojamos klasės dirbančios tiesiogiai su duomenų baze, bei klasių dirbančių su dinaminėmis analizėmis, kadangi dviejų lygmenų architektūroje jos nebuvo realizuotos).

7 Literatūra

1. Inji Wijegunaratne and George Fernandez Distributed applications engineering. – Springer, 1998 – 267 p.
2. Sun Microsystems Designing Enterprise Applications with the J2EE Platform, Second Edition. Iš Java.sun.com [interaktyvus]. 2004 gegužė [žiūrėta 2004 gegužę]. Prieiga per internetą: http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/index.html
3. Client/Server software architectures. Iš www.sei.cmu.edu [interaktyvus]. Prieiga per internetą: http://www.sei.cmu.edu/str/descriptions/clientserver.html [žiūrėta 2004 gegužę]
4. Three tier software architectures. Iš www.sei.cmu.edu [interaktyvus]. Prieiga per internetą: http://www.sei.cmu.edu/str/descriptions/threetier.html [žiūrėta 2004 gegužę]
5. Two tier software architectures. Iš www.sei.cmu.edu [interaktyvus]. Prieiga per internetą: http://www.sei.cmu.edu/str/descriptions/twotier.html [žiūrėta 2004 gegužę]
6. Object oriented design. Iš www.sei.cmu.edu [interaktyvus]. Prieiga per internetą: http://www.sei.cmu.edu/str/descriptions/oodesign.html [žiūrėta 2004 gegužę]
7. Object oriented analysis. Iš www.sei.cmu.edu [interaktyvus]. Prieiga per internetą: http://www.sei.cmu.edu/str/descriptions/ooanalysis.html [žiūrėta 2004 gegužę]
8. Jiehan Zhou, Eila Niemelä. "Middard, Middleware, Middardware and Middleware technology." Internete: http://bscw.enst-bretagne.fr/pub/bscw.cgi/d2299802/CADS2003_ZHOU.pdf [žiūrėta 2004 gegužę]
9. Petr Tuma. "Modern Software Architectures: Novel Solutions or Old Hats?" Internete: http://nenya.ms.mff.cuni.cz/publications/Tuma-DATAKON03-SoftArchitectures.pdf [žiūrėta 2004 gegužę]
10. Stephen Rochefort "Agent – based transformational architectures"
11. A. Ramdante – Cherif and N. Levy "An approach for dynamic reconfigurable software architectures" – Integrated Design and Process Technology IDPT-2002

8 Terminų žodynas

Middleware	Vidurinio lygmens programinė įranga, programinės įrangos dalis veikianti kaip tarpininkas tarp DB ir vartotojo sąsajos.
DB(database)	Duomenų bazė. Gali būti failų sistema arba DBVS
DBVS	Duomenų bazių valdymo sistema. Speciali programa, skirta palengvinti ir standartizuoti darbą su duomenų saugykla.
JAVA	Objektiškai orientuota programavimo kalba.
OOA (object oriented analysis)	objektiškai orientuota analizė. Tai standartizuotas procesas, palengvinantis programų kūrimą objektiškai orientuotomis programavimo kalbomis.
OOD (object oriented design)	Objektiškai orientuotas programų kūrimas. Tai bendros metodikos skirtos programoms kurti objektiškai orientuotomis kalbomis.
SQL (querry language)	Kalba skirta darbui su DBVS.
Package	Paketas. Klasių rinkinys skirtas tam tikrai funkcijai ar jų grupei atlikti, paprastai prijungiamas programoje kaip vientisas elementas
Server	Serveris. Tai programinė įranga, kuri aptarnauja kitą programinę įrangą pateikdama, saugodama ar apdorodama informaciją reikalingą programai – klientui.
Event	Įvykis. Programavimo klabos terminas nusakantis programos vidinį įvykį, kuris įvykdamas kartu turi sukelti tam tikrą programos reakciją. Paprasčiausias įvykio pavyzdys – mygtuko paspaudimas dialogo lange.
Frame	Freimas. Programavimo struktūra JAVA kalboje pagrindinai atsakinga už grafinį lango pateikimą. Programuotojas pats nustato papildomus parametrus bei klasės funkcionalumą.
Atribute	Atributas. Tai duomenų laukas arba laukų grupė (pvz masyvas) klasės viduje. Jis gali būti prieinamas

	išorinėms klasėms, bet programuojant objektiškai stengiamasi vengti kreiptis į klasės atributus tiesiogiai ne iš klasės vidaus
Metodas	tai funkcija skirta atlikti operacijas su klasės atributais bei išoriniais į klasę patenkančiais duomenimis.
Interface	Sąsaja. Specifinė JAVA struktūra, per kurią galima pajungti sąsają apdorojančią klasę.