KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIJOS FAKULTETAS

PROGRAMŲ INŽINERIJOS KATEDRA

Viktoras Gurgždys

Vienetų testų generavimas, remiantis testų duomenų baze

Magistro darbas

Darbo vadovas
prof. E. Bareiša

KAUNAS, 2009

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIJOS FAKULTETAS

PROGRAMŲ INŽINERIJOS KATEDRA

Viktoras Gurgždys

Vienetų testų generavimas, remiantis testų duomenų baze

Magistro darbas

Recenzentas
prof. V. Pilkauskas
2009 05

Darbo vadovas
prof. E. Bareiša
2009 05

Atliko
IFM-3/2 gr. stud.
Viktoras Gurgždys
2009 05 22

KAUNAS, 2009

TURINYS

PAVEIKSLĖLIŲ TURINYS

LENTELIŲ TURINYS

UNIT TESTS GENERATION USING TEST HISTORY DATA

SUMMARY

Software testing is critical part in software development. Not only testers are responsible for software quality. Programmers should use quality assurance techniques as code static analysis (code review) and unit testing. Sometimes developers omit testing their code or perform just superficial testing because unit testing consumes development time. Automated unit testing tools provide the way for reducing of testing costs.

There are various techniques for generating test cases by using specialized tools. The goal of generation of tests cases is to achieve higher code coverage and expose unexpected errors. In many cases it is impossible to achieve full code coverage by using only generated test cases. Then manual efforts are needed. It is reasonable to reuse unit tests for method similar to that already tested. This work helps to analyse unit test reusability problem. The approach presented is based on unit test storing and reusing by comparing current testing method and method that has saved tests. This method should be used in combination with particular unit test generation tool.

1 ĮVADAS

1.1 Dokumento paskirtis

Šis dokumentas yra programų sistemų inžinerijos magistro baigiamasis darbas. Dokumente trumpai supažindinama su kodo vienetų testavimu, automatiniu testų generavimu bei pagrindiniais įrankiais, kurie gali tai atlikti. Supažindinta su įrankiais, padedančiais įvertinti vienetų testus, matuojant kodo padengimą ir trumpai aptarti kodo padengimo būdai bei jų skirtumai. Išanalizavus esamus vienetų testų sprendimus pasiūlytas būdas generuoti testus naudojantis jau esamais vienetų testais, tiksliau pakartotinai juos panaudoti atliekant minimalius pakeitimus.

Siūlomo sprendimo įgyvendinimui suprojektuota ir sukurta programinė įranga ReUnit. Pagrindiniai projektavimo etapai aptariami trečiojoje šio darbo dalyje. Detaliai aprašytas pagrindinis sistemos algoritmas skirtas, vienetų testų parinkimui lyginant metodus, kuriuos jie testuoja. Darbo metu ištirta sukurtos programinės įrangos kokybė naudojant automatizuoto testavimo įrankius. Penktojoje darbo dalyje praktiškai patikrintas siūlomo metodo veikimas naudojant sukurtą sistemą. Pateikti pagrindiniai rezultatai bei išvados.

1.2 Santrauka

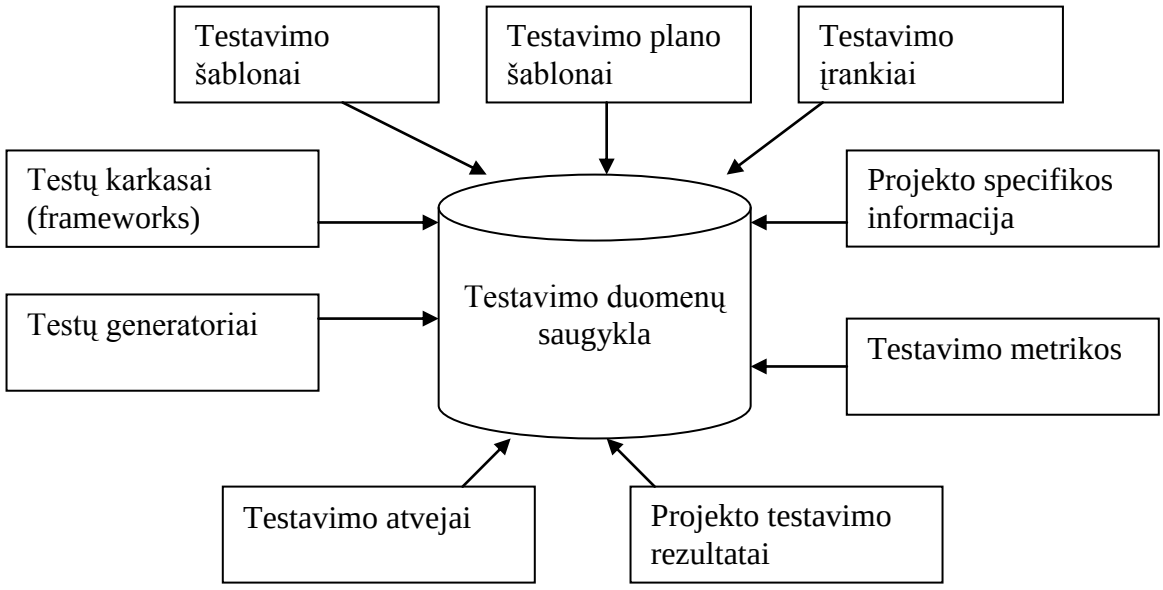
Testavimas – svarbus etapas, norint užtikrinti, kuriamos programinės įrangos kokybę. Testavimas atliekamas visuose programinės įrangos kūrimo etapuose, pradedant analizės dokumento peržiūra bei tikrinimu ir baigiant sukurto produkto testavimu vartotojo aplinkoje. Šiuo metu vis daugiau dėmesio skiriama testavimui programavimo etape, kurio metu testuojami atskiri programos kodo gabalai. Tai yra, taip vadinamas, vienetų testavimas (Unit Testing). Vienetų testai yra automatizuoti programų testai, tikrinantys kodo vienetus atskirai vieną nuo kito. Kiekvienas kodo vienetas (t.y. funkcija, klasė, metodas) turi savo testą ar kelis testus. Parašytas vienetų testas, turėdamas pradinius duomenis, kviečia metodą, gauti rezultatai lyginami su tikėtaisiais ir sužinoma ar kodo gabalas veikia taip kaip tikimasi.

Rašant testus dažnai naudojamasi pagalbiniais įrankiais, kurie palengvina šį darbą. Įrankių pagalba galima patogiai vykdyti vienetų testus, analizuoti rezultatus, vykdyti automatinį testavimą. Pastebėta, jog dažnai rašomi vienetų testai metodams, panašiems į tuos, kurie jau buvo kažkada testuoti. Atsižvelgiant į tai, naudinga turėti bei nuolatos pildyti vienetų testų duomenų bazę, o vėliau naujus testus generuoti pagal jau išsaugotų testų duomenis.

2 ANALITINĖ DALIS

2.1 Pakartotinis panaudojimas testavimo procese

Programos įrangos kūrime testavimas dažniausiai užima 50-60% laiko. Logiška, kaip ir visuose kituose programinės įrangos kūrimo etapuose, testavimo etape ieškoti panašumų su testavimu kituose projektuose ir stengtis pakartotinai panaudoti anksčiau sukauptas žinias. *Gupta* savo pranešime [1] apibendrino pakartotinio panaudojimo idėją testavimo procese ir pasiūlė saugoti visus testavimo duomenis, kartu su projektu, kuriame jie buvo naudojami. Tai taikoma testavimo šablonams, testavimo atvejams, testavimų metrikų duomenims, specifinei projekto informacijai ir pan.



Pav. 1 Testavimo duomenų saugykla

Tai abstrakti idėja, tačiau nesigilinama į patį testavimo atvejų pakartotinį panaudojimą. Mano darbe toliau gilinamasi į vienetų testavimą, testavimų atvejų generavimą ir technines galimybes pakartotinai panaudoti testavimo atvejus.

2.2 Programos kodo testavimas

Programos kodas gali būti testuojamas jį peržiūrint (statinis testavimas [23]) arba vykdant vienetų testus. Šiame darbo toliau gilinsimės tik į testavimą naudojant vienetų testus. Vienetų testą galima suprasti kaip metodą, kuris turėdamas pradinius duomenis (įėjimą) vykdo testuojamą metodą ir gautą rezultatą lygina su pageidaujamu rezultatu. Jei gautas ir pageidaujamas rezultatai

sutampa – testas sėkmingas. Vienetų testų visuma, kuria testuojamas tam tikras modulis (pvz.: klasė, tam tikra verslo logikos metodų eilė ir pan.) vadinama testavimo atveju. Pagrindinė metrika, kurią galime naudoti, kad pasakytume ar testavimo atvejis yra geras testuojant konkretų modulį yra kodo padengimas.

Paprastai vienetų testai kuriami naudojant konkretų vienetų testų karkasą (*framework*), kuris suteikia visas reikalingas priemones testo kūrimui ir vykdymui. Priemonės tai klasių ir metodų visuma leidžianti tikrinti rezultatą, įrankiai skirti vykdyti vienetų testus r apdoroti gautus rezultatus.

2.2.1 Vienetų testų karkasai

Rinkoje yra daug vienetų testų karkasų, dalis jų sukurti visoms pagrindinės programavimo platformoms (*.Net, Java, Php, Phyton ir t.t.*), kiti veikia tik vienoje platformoje. Plačiausiai žinoma kodo testavimo karkasų šeima yra *xUnit.* Atitinkamai kiekvienai platformai sukurta *nUnit (.Net), jUnit (Java), FUnit (Fortan), DUnit (Delphi), PHPUnit (Php)* ir daugelis kitų. Microsoft bendruomenė pradėjo atsiriboti nuo *xUnit* šeimos, čia naudojamas į MS Visual Studio integruotas vienetų testų karkasas *Unit Test Framework*[1]. Karkaso pasirinkimą dažniausiai apsprendžia jo teikiamos funkcijos (rezultatų analizės galimybės, *Assert* sakinių įvairovė) bei integracija su kitais kokybės užtikrinimo bei vienetų testų metrikų matavimo įrankiais.

2.2.2 Vienetų testų kodo padengimo matavimas

Kodo padengimas [9] yra matas naudojamas programinės įrangos kodo testavime. Jis aprašo ištestuoto programos išeities kodo laipsnį.

Padengimo kriterijus

Programos elgsenos padengimui testavimo atveju matuoti naudojama vienas arba daugiau padengimo kriterijų. Yra daug padengimo kriterijų, tačiau svarbiausi yra šie:

- Funkcinis padengimas – Ar kiekviena funkcija buvo vykdoma?
- Sakinių (kodo eilučių) padengimas – Ar kiekviena kodo eilutė buvo vykdyta?
- Sprendimo variantų (atšakų) padengimas – Ar kiekviena sąlyga (tokia kaip *if* sakinys) buvo įvykdyta visais atvejais (*true* ir *false*)?

[1] http://msdn.microsoft.com/en-us/library/ms243147(VS.80).aspx

- Sąlygos sakinių padengimas – Ar kiekviena loginių (*boolean*) sakinių atšaka įvertinta? Pvz.: sakinys „`boolean isAnimal = isCat ? true : false;`“
- Kelio padengimas – Ar kiekvienas galimas programos kelias įvykdytas vykdant testavimo atveją?
- Įėjimo/išėjimo padengimas – Ar kiekvienas įmanomas funkcijos iškvietimas ir galimas rezultato gražinimas (*return*) iškviestas?

Kodo padengimas praktikoje

Programos kodas kompiliuojas naudojant specialius nustatymus, papildomas bibliotekas, papildomus kintamuosius, kurie yra įterpiami į programos kodą. Jie nekeičia elgsenos, tačiau jų pagalba galima skaičiuoti kodo padengimą. Šis procesas leidžia rasti programos kodo dalis, kurios naudojant testavimo atvejį niekada nebuvo vykdytos (pvz.: klaidų ir išimtinių atvejų gaudymas). Turint padengimo matavimo rezultatus, galima įvertinti testavimo atvejį ir jei reikia jį tobulinti ir kartoti jo kokybės matavimo procesą. Tuo atveju kai pasiekiamas nepilnas kodo padengimas, analizuojant kodą galima nustatyti ir perteklinio kodo atvejus, tuomet testavimo atvejis yra geras.

Padengimo matavimo įrankiai

JTest, .Test[2] – matuoja procentinį kodo padengimą. Tai mokamas įrankis, turintis ir daugiau funkcijų, tokių kaip vienetų testų generavimas ar statinė kodo analizė. Yra sukurtos versijos pagrindinėms programų kūrimo aplinkoms ir programavimo kalboms. Praktikoje nedažnai naudojamas dėl savo kainos ir todėl kad pagrindinės aplinkos, tokios kaip MS Visual Studio 2008, turi daugumą šio įrankio funkcijų, o kodo padengimas matuojamas nemokamais alternatyviais įrankiais.

Atlassian Clover[3] – Java programos kodo padengimo matavimo ir testų vizualizavimo įrankis, galintis parodyti kokį padengimą suteikia kiekvienas testas. Jis yra nemokamas ir atviro kodo įrankis.

PartCover[4] **–** Nemokamas, atviro kodo įrankis, skirtas MS .Net 2.0 technologijai. Gali generuoti kodo padengimo ataskaitas HTML formatu. Sukurta ir šio įrankio mokama versija NCover[5].

[2] www.parasoft.com
[3] http://www.atlassian.com/software/clover/

CodeCover[6] – Java / Cobol kodo padengimo matavimo įrankis, naudojamas tiek Windows tiek Linux operacinėse sistemose. Populiarus tarp Java programuotojų, nes turi Eclipse (viena iš populiariausių Java programavimo aplinkų) skirtą įskiepį, taip pat rezultatus galima pateikti XML formatu, kas leidžia ateityje juos patogiai apdoroti ir saugoti.

2.3 Esamų vienetų testų generavimo sprendimų analizė

Vienetų testų generatoriaus [14-15] tikslas – sugeneruoti testavimo atvejį, t.y. testų rinkinį, skirtą ištestuoti, konkretų metodų rinkinį ar vieną metodą. Vienetų testų generatoriai, pagal savo generavimo techniką, skirstomi į generatorius paremtus vienetų testų generavimu analizuojant programos kodą arba analizuojant programos modelį, tai gali būti, klasių diagrama, kurioje yra nustatyti objekto apribojimai OCL pagalba [7]. Savo ruožtu kodu paremti generatoriai skirstomi [7] į atsitiktines įėjimo reikšmes generuojančius įrankius, analizuojančius programos kelius, genetiniais algoritmais paremtus įrankius, bei įrankius, kurie vykdo programos kodą.

2.3.1 Kodo analize paremti vienetų testų generavimo įrankiai ir metodai

Tai vienetų testų generatoriai, kurie analizuoja programos kodą tam, kad galėtų sugeneruoti jam vienetų testus. Dauguma rinkoje esančių vienetų testų generatorių besiremiantys kodo analize. Toliau pateikiami, skirtingi populiariausi generatoriai.

Microsoft Visual Studio Team Test[7]

Tai statinis vienetų testų generatorius (metodai nevykdomi). Įrankio tikslas – sugeneruoti projektą kartu su vienetų testų klasėmis. Tai pats primityviausias generatorius, kurio paskirtis palengvinti programuotojo darbą. Sugeneruotas projektas, turi tik šabloninius metodus, jų vykdyti negalima. Programuotojas privalo parašyti vieneto testo logiką. Privalumas tas, jog generuoti, o po to aprašyti pačiam užtrunka mažiau laiko, nei viską rašyti. Generatorius pilnai aprašo testuojamų objektų sukūrimą. Microsoft vienetų testų karkasas, palaiko ir neviešų metodų testavimą (*private, protected, internal*), įrankis tai išnaudoja ir sugeneruoja testus šiems metodams.

JTest/.Test

[4] http://sourceforge.net/projects/partcover/
[5] http://www.ncover.com/
[6] http://www.codecover.org/
[7] http://msdn.microsoft.com/en-us/teamsystem/

Tai komercinis įrankis, sukurtas *Parasoft*. Pagrindinis *JTest/.Test* įrankio tikslas – sugeneruoti vienetų testų seką, naudojant atsitiktinius duomenis, kad būtų iškelto į paviršių nenumatytos vykdymo klaidos (angl.: *runtime exceptions*). Testai generuojami taip, kad būtų padengta kuo daugiau kodo atšakų arba sakinių, tai priklauso nuo vartotojo pateiktos konfigūracijos. Įrankis bandymų būdu parenka tokias pradines reikšmes, kad sukeltų standartines vykdymo klaidas, tokias kaip *NullReferenceException* ir pan.

Nors generavimas paremtas atsitiktinai generuojamais duomenimis, tačiau pastebėta, jog įrankis sugeneruoja visą laiką tuos pačius testinis duomenis. Taip pat nepasiekiamas pilnas kodo padengimas. Kaip ir Microsoft įrankis, palaiko testų generavimą ne viešiems metodams.

Įrankis, be vienetų testų generavimo, turi daug papildomų funkcijų, pavyzdžiui kodo padengimo matavimas, statinė kodo analizė, geriausių praktikų taikymo pasiūlymai ir pan.

Java PathFinder[8]

Java PathFinder (*JPF*) nėra standartinis testų generatorius. Šį įrankį reiktų suvokti, kaip *Java* virtualią mašiną (*JVM*[9]), kuri vykdo programos kodą. Nuo standartinės *JVM* šis įrankis skiriasi tuo, kad analizuojamas programinis kodas vykdomas keletą kartų, nustatant vis naujas kintamųjų reikšmes, pavyzdžiui jei programoje mes naudosime atsitiktinių reikšmių generavimo funkciją, *JPF* vykdys programą sugeneruodama vis naujas reikšmes iš duoto intervalo, jei programą vykdytume rankiniu būdu naudojant standartinę *JVM* – atsitiktinių reikšmių generatorius greičiausiai nesugeneruotų visų įmanomų reikšmių, tad liktų klaidos tikimybė.

Priešingai nei vykdant programą standartiniu būdu, įvykus klaidai galime matyti pilną programos kelią, kuris sukėlė gautą klaidą, taip pat matysime sėkmingai įvykdytus kelius. Tai veikia ir vykdant programas su keliomis gijomis.

Bandymais nustatyta, jog testuojant dideles programas *JPF* veikia lėtai, taip pat nepavyksta pasiekti aukšto kodo padengimo.

CodePro[10]

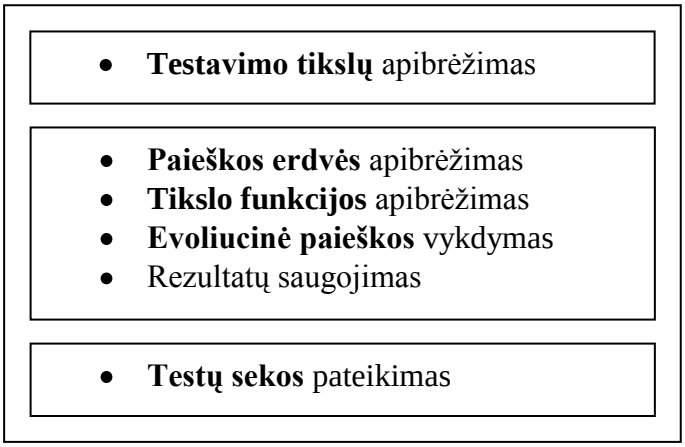
Komercinis įrankis, sukurtas *Instantiations* kompanijos. Kaip ir dauguma kitų mokamų įrankių, turi daug funkcijų iš kurių viena – generuoti vienetų testus. Generavimo būdas labai panašus į *JTest/.Test* įrankio. Įrankis parinkinėja įvairias parametrų kombinacijas, kad išgautų

[8] http://javapathfinder.sourceforge.net/
[9] http://java.sun.com/docs/books/jvms/
[10] http://www.instantiations.com/codepro/

kuo didesnį kodo padengimą, tiesa parinkimo būdas yra euristinis, tai reiškia, kad parenkamos protingos, labiausiai tikėtinos kombinacijos, tai yra pagrindinis šiuo įrankio pranašumas.

Evoliucinis klasės testavimas (*EvoUnit*)

Wappler 2007m. pasiūlė evoliucinio klasės testavimo idėją [2]. Šis metodas orientuotas į visą klasę – ne į atskirą metodą, kai tuo tarpu kiti vienetų testų generavimo sprendimai skirti vienetų testų generavimui konkrečiam metodui. Idėja paremta genetiniu algoritmu. Šio metodo tikslas pasiekti pilną kodo padengimą, nesvarbu ar norėtume padengti kodo eilutes, atšakas ar kelius. Toliau pateikiami šios idėjos įgyvendinimo žingsniai.



Pav. 2 Evoliucinio kasės testavimo idėja

Visi evoliucinio testavimo žingsniai gali būti atlikti automatiškai. Pirmasis idėjos žingsnis – nustatyti testavimo tikslus. Tai daroma analizuojant klasės kodą. Jei testuosime kodo padengimą atšakų atžvilgiu, tuomet kiekviena atšaka taps testavimo tikslu. Esant kitai testavimo technikai, parenkami kiti testavimo tikslai, priklausomai nuo technikos. Antras žingsnis – iteruoti per visus testavimo tikslus ir kiekvienoje iteracijoje išskirti paieškos erdvę. Ši erdvė sudaryta iš visų įmanomų testų sekos. Paieškos erdvės apibrėžimo procedūra baigta, tiksliai nusakius funkcijas ir jų argumentų bei rezultatų tipus. Šiuos duomenis naudoja genetinio programavimo algoritmas, kuris optimizuoja paieškos erdvę ir suranda tik tuos testus, kurie reikalingi kodo padengimui. Šiame darbe nesigilinsime į genetinio programavimo algoritmus, jie plačiai aptariami kituose moksliniuose veikaluose. Taip pat pagrindinės žinios pateikiamos ir Wappler darbe [2].

Tai abstraktus idėjos aprašymas. Akcentuojant skirtumus nuo kitų testų generavimo idėjų reiktų paminėti, jog tai nėra dinaminis testų generatorius (kodas nevykdomas). Jis paremtas paieškos algoritmais, kurie analizuoja programinį kodą ir ieško ryšių tarp metodų, paieškos erdvės sudarymo tikslui pasiekti. Algoritmas sukuria kvietimų priklausomybės grafą, kuris

nusako ryšius tarp metodų. Jis skirtas išsiaiškinti, duomenų tipams bei metodų kvietimų tvarkai, t.y. gauname veiksmų atlikimo seka (grafiškai galime įsivaizduoti UML sekų diagramą). Turint veiksmų seką, atliekama paieška ir nustatomos kviečiamų metodų parametrų reikšmės. Parametrų parinkimui taip pat atliekama paieška kode, kad būtų pasiektas testavimo tikslas – dažniausiai padengta tam tikra atšaka.

Ši idėja realizuota įrankyje *EvoUnit* (kūrėjas – tas pats autorius). Tai *Java* testų generatorius. Programos tikslas sugeneruoti *JUnit* testų klases. Kaip didžiausią privalumą prieš kitus testų generatorius autorius pateikia aukštą kodo padengimą, pateikiami duomenys, jog eksperimentiškai pavyko padengti testuojamą kodą nuo 65,7% iki 100% (matuojant padengimą atšakomis), o vidutinis padengimas – 91,6%.

2.3.2 Modelio analize paremti vienetų testų generavimo įrankiai ir metodai

PĮ kūrimo procese testavimas atliekamas nuo pirmųjų ciklo etapų. Atliekama reikalavimų dokumento peržiūra (specifikacijos testavimas), prototipo testavimas, projekto testavimas ir taip iki galutinio produkto testavimo kliento aplinkoje. Apie testavimą vienetų testais galima pradėti galvoti dar prieš programos logikos įgyvendinimą kodu. Kadangi šiuo metu vis plačiau PĮ kūrime taikomas išankstinio modelio sudarymas, galima generuoti testus remiantis modeliu. Tam gali būti panaudoti formalūs (tokie kaip Z arba Object-Z[11] specifikacijos, LOTOS[12] specifikacijos) ir neformalūs modeliai [7] (UML).

OZTest testų generatorius

Tai testavimo karkasas, leidžiantis generuoti testus iš Object-Z specifikacijų. Object-Z tai objektinė formalių specifikacijų kalba, ja sudarytas modelis yra pilnai formalus. OZTest [24] analizuoja formalios specifikacijos kalbą. Šis įrankis neturi sukurto orakulo įėjimo reikšmių parinkimui tad reikalingas vartotojo įsikišimas. Tai pusiau automatinis įrankis. Jo pakalba generuojamos klasės ir testai skirti joms testuoti remiantis modeliu.

Testavimo atvejų generavimas naudojant netikslius OCL apribojimus [7]

OCL tai kalba, sukurta kaip UML praplėtimas, skirta reikšmių apribojimų kūrimui. OCL pagalba galima nustatyti metodo parametrų apribojimus, tai gali būti galimų reikšmių rėžiai. Šios kalbos pagalba galima nustatyti apribojimus galimam rezultatui, tai yra, taip vadinamoji, „sąlyga-po“ (angl. post condition). Šią sąlygą Packevičius siūlo naudoti kaip orakulą testavimo atvejų

[11] http://www.itee.uq.edu.au/~smith/objectz.html
[12] http://www.cs.stir.ac.uk/~kjt/research/well/

įėjimų reikšmių generavimui [7]. Generavimui keliamų apribojimų nėra, tai gali būti, bet kuri testavimo atvejų generavimo technika.

2.4 Projektavimo metodologijos ir technologijų analizė

Šiame skyriuje išanalizuoti pagrindiniai įrankiai skirti vienetų testų generavimui. Dauguma įrankių veikia kaip programavimo aplinkos įskiepiai (angl. Add-in). Integracija su programavimo aplinka suteikia galimybę naudoti įrankius neatsitraukiant nuo pagrindinio darbo. Taip pat tai leidžia naudoti kelis įrankius vienu metu. Pavyzdžiui, vienetų testų generavimui galima naudoti *Parasoft .Test* įrankį, o sugeneruotus testus leisti kitu *Visual Studio* įskiepiu *ReSharper*[13], svarbiausia, kad naudojami įrankiai būtų suderinami tarpusavyje, pateiktame pavyzdyje testai generuojami naudojant *NUnit* vienetų testų karkasą, o *ReSharper* gali paleidinėti šios rūšies testus.

Pakartotiniam vienetų testų panaudojimui svarbu, sukurtus testus būtų galima naudoti ne vienoje platformoje, o bent keliose pagrindinės, pavyzdžiui *.Net ir Java*. Šios platformos techniškai yra skirtingos, tad vienoje platformoje sukurtos bibliotekos, kitoje naudoti negalima. *.Net* platformos programavimo kalbos (C#, VB.NET) vykdomos CLR[14] (angl. Common Language Runtime) aplinkoje [20]. *Java* programavimo kalba vykdoma virtualioje mašinoje JVM (angl. Java Virtual Machine). Dėl šios priežasties standartinis duomenų apsikeitimas tarp šių (ar kitų dviejų skirtingų platformų) neveikia. Reikalingas bendras duomenų formatas, kuris būtų vienodai interpretuojamas skirtingose aplinkose. Viena iš galimybių tai pasiekti – naudoti į servisus orientuotą SOA architektūros modelį.

Integracija su programavimo aplinka

Dauguma rinkoje esančių produktų: programavimo aplinkos, biuro programų paketai, verslo valdymo sistemos ar operacinės sistemos, turi praplėtimo nauju funkcionalumu galimybes. Dažniausiai tai yra gamintojo paskelbta sąsaja API (angl. Application programming interface), kurią gali pasinaudoti programuotojas norėdamas sąveikauti su sistemos naudojamais duomenimis ir įrankių rinkinys, taip vadinamas SDK (angl. Software development kit), sąveikaujantis su API ir galintis atlikti aibę veiksmų, kurių reikia naujo funkcionalumo programavimui. Kuriant naujus, integruojamus į konkrečią programą įrankius [16] (Add-in), dažniausiai naudojamas tos programos SDK.

[13] http://www.jetbrains.com/resharper/
[14] http://msdn.microsoft.com/en-us/library/8bs2ecf4(VS.71).aspx

Kuriama sistema bus integruojama į *MS Visual Studio* programavimo aplinką, kitaip IDE (angl. Integrated development environment). Sistema, tai programavimo aplinkos įskiepis. Šio įskiepio kūrimui naudojamas *Visual Studio SDK,* kurio pagalba galima lengvai kurti įskiepius bei pralėsti *Visual Studio* turimas galimybes. Integracija ir galimybių praplėtimas dažniausiai paremtas COM (angl. Component object model) modeliu.

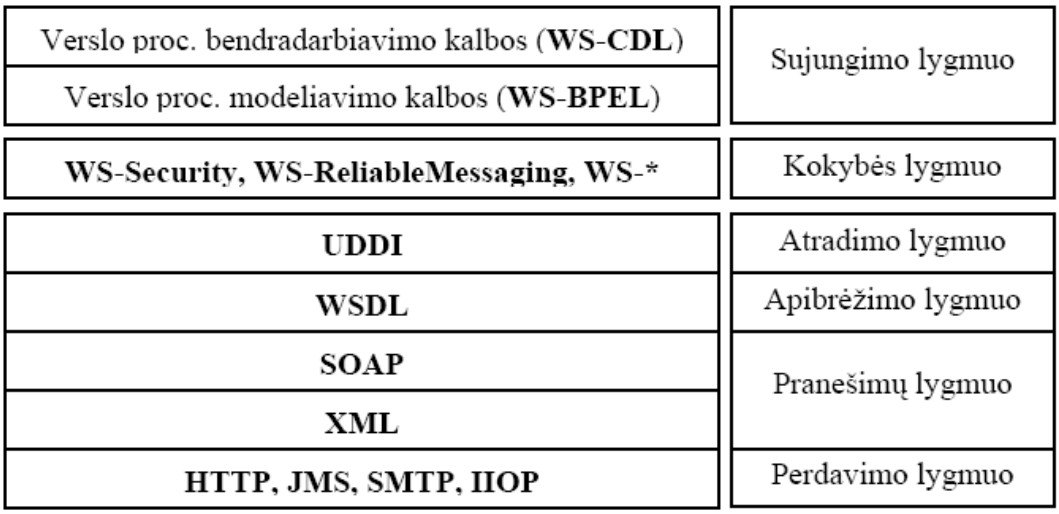
COM

Microsoft COM [11-12] technologija leidžia tarpusavyje bendrauti programinės įrangos komponentams. Ji įgalina programuotojus kurti pakartotinai panaudojamus komponentus ir iš jų sudaryti aplikaciją. COM modelis naudojamas kuriant MS Windows operacinių sistemų šeimos programas. Ne visos programos kuriamos naudojant šį modelį, tačiau siekiant didesnio lankstumo, rekomenduojama laikytis jo arba programuoti .Net, nes .Net, kaip sako Microsoft, yra technologija papildanti COM.

SOA architektūra ir Web-servisai

Į servisus orientuota architektūra SOA [6] – tai architektūros stilius, kurio pagalba sistemos komponentai atskiriami taip, kad jie būtų kuo mažiau suporuoti tarpusavyje. Ši technologija leidžia kurti sistemos komponentus taip, kad jie galėtų būti pakartotinai panaudojami. Pasikeitimai viename komponente neįtakoja arba minimaliai įtakoja kitus komponentus. Duomenų apsikeitimui tarp komponentų naudojami gerai žinomi duomenų protokolai (dažniausiai SOAP). Dažniausiai naudojamos technologijos SOA [17-19] architektūros principams pasiekti yra Web-servisai, Enterprise-servisai, .Net Remoting objektai. Vienintelė iš išvardintų technologijų yra nepriklausoma nuo realizacijos platformos – tai Web-servisai. .Net Remoting objektai pasižymi dideliu našumu ir saugumu, tačiau jie gali būti prieinami tik iš .Net technologija parašytų komponentų. Enterprise-servisai taip pat kaip ir .Net remoting objektai yra surišti su Microsoft .Net platforma.

XML Web service – tai protokolų ir standartų rinkinys, naudojamas duomenų apsikeitimui tarp aplikacijų ir sistemų [21]. Įvairiomis programavimo kalbomis parašytos aplikacijos, veikiančios skirtingose operacinėse sistemose, naudoja Web service duomenų mainams kompiuteriniuose tinkluose. Technologijas, naudojamas įgyvendinant Web-servisą, galima suskirstyti į keletą lygmenų (Pav. 3 [22]). Tai perdavimo (angl. Transportation), pranešimų (angl. Messaging), apibrėžimo (angl. Definition), atradimo (angl. Discovery), kokybės (angl. Quality of Service) ir sujungimo lygmenys (angl. Service Aggregation).



Pav. 3 Web-serviso technologijų medis

Perdavimo lygmenyje, gali būti panaudoti keli protokolai, tačiau dažniausiai naudojamas HTTP. Pranešimų lygmenyje, pagrindinės naudojamos technologijos yra XML ir SOAP. *Web service* sąsajos apibrėžimą galima suprasti kaip susitarimą tarp serviso ir jo naudotojo, užtikrinantį, kad servisas atliks tam tikrus veiksmus, gaudamas tam tikrus parametrus. Tai sąsajos (angl. Inteface) analogas objektinio programavimo kalbose. Tokiam serviso apibrėžimui yra sukurta XML paremta kalba WSDL. Atradimo lygmenyje, naudojama UDDI technologija. UDDI specifikacija aprašo registrą, skirtą kaupti informacijai apie servisus bei juos teikiančias organizacijas. Servisų kokybės užtikrinimui skirtos ištisos standartų grupės. *WS-Security* – tai standartų grupė, nurodanti kaip galima užtikrinti *Web service* saugumą *WS-ReliableMessaging* standartas užtikrina, kad pranešimai pasieks adresatą reikiama tvarka bei nebus dubliuojami. Sujungimo lygmuo tai lygmuo, skirtas serviso sujungimui su kitu servisu.

2.5 Analizės etapo rezultatai

Išsiaiškinta, jog vienetų testų pakartotinio panaudojimo tema nepasiūlyta jokių rimtų sprendimų. Didžiausias dėmesys skiriamas vienetų testų generavimui. Šioje dalyje supažindinta su pagrindiniais ir dažniausiai naudojamais generatoriais ir testavimo įrankiais. Daugumos generatorių trūkumas – nedidelis kodo padengimas sugeneruotais testais. Kiti, tokie kaip *Microsoft Visual Studio Team Test*, generuoja tik šabloninius metodus, kurie yra nevykdomi, kol rankomis nepadengiama logika.

Iš visų įrankių išsiskiria *EvoUnit*, nes jis skirtas visos klasės testavimui, ne atskiro metodo, taip pat pasiekiamas aukštas kodo padengimas, lyginant su kitais generatoriais. Trūkumas – ilgai trunkantis generavimas.

Šiame darbe, išanalizavus įrankių privalumus ir trūkumus, siūlomas įrankis, kurio pagalba galima išsaugoti esamus metodų testus ir juos pakartotinai panaudoti kitiems metodams, atliekant minimalius pakeitimus testų kode. Įrankio projektavimui, atlikta projektavimo technologijų analizė, jos metu, pagal žinomus reikalavimus, parinktos technologijos. Tai nėra pakaitalas, jau egzistuojantiems generavimo metodams pakeisti, tai metodas praplečiantis testavimo galimybes, dėl to pasiekiamas aukštesnis kodo padengimo laipsnis testuojant panašios probleminės srities kodą. Kitame skyriuje pateikiamas šios problemos sprendimui sukurtas algoritmas ir jį realizuojantis įrankis, pabrėžiant esminius projektavimo etapo rezultatus bei algoritmo esmę.

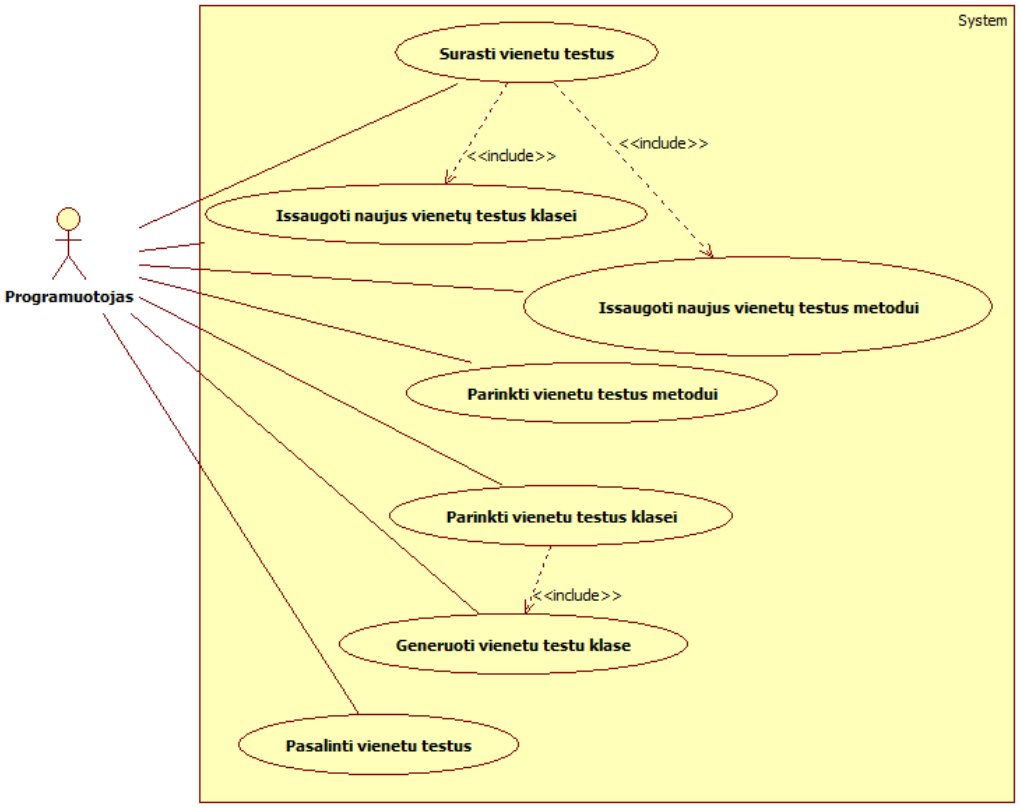
3 PROJEKTINĖ DALIS

3.1 ReUnit įrankis

Antrajame skyriuje išanalizuoti įrankiai, skirti programos kodo testavimui, naudojant vienetų testus. Didžiausias dėmesys skirtas testų generavimui. Plėtojant pakartotinio panaudojimo programinės įrangos kūrime idėją, siūloma saugoti turimus kodo vienetų testus duomenų bazėje ir vėliau juos naudoti kietiems metodams.

ReUnit programinė įrangą galima atskirti į dvi dalis. Pirmoji dalis, tai testų duomenų bazė ir kodo biblioteka skirta atlikti veiksmus su kodo duomenų baze. Ši dalis atsakinga už testų saugojimą ir paiešką pagal metodą. Antroji dalis, tai testų duomenų bazės klientas. Realizuota sukūrus MS Visual Studio 2008 įskiepį (angl. Add-in). Sistema suprojektuota taip, kad klientinė dalis galėtų būti ne tik *.Net* platformai sukurta programa, bet ir *Java* virtualioje mašinoje veikianti aplikacija. Kaip analogas tai galėtų būti *Eclipse* įskiepis.

3.1.1 Sistemos panaudos atvejai

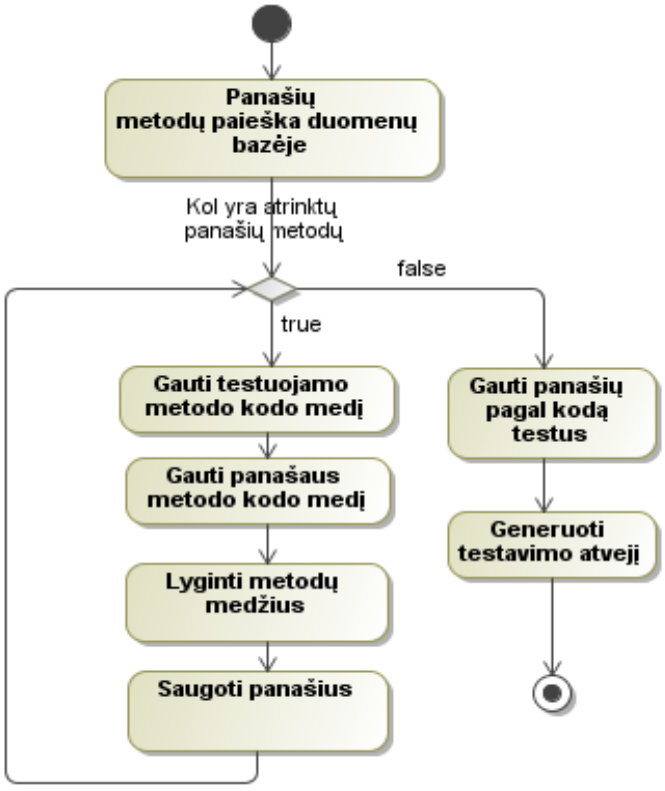


Pav. 4 Sistemos panaudos atvejai programuotojo atžvilgiu

3.1.2 Vienetų testų parinkimo algoritmas

Šiame skyriuje plačiau aprašomas algoritmas, kuris iš esamų testų metodui parenka vienetų testus. Saugant vienetų testus duomenų bazėje, jie saugomi kartu su metodu, kurį testuoja. Kad būtų galima parinkti testą, tinkantį metodui, duomenų bazėje atliekama panašių metodų paieška (Pav. 5). Sudėtingiausia algoritmo dalis yra nuspręsti ar metodai yra panašūs. Paieška atliekama dviem etapais:

- **Pradinės testuojamų metodų erdvės išskyrimas.** Šis etapas atliekamas duomenų bazėje. Į testų duomenų bazę siunčiama suformuota užklausa, skirta atrinkti metodus, kurių parametrų tipai analogiški testuojamo metodo parametrų tipams. Iš rezultatų suformuotas metodų (kartu su testais) sąrašas toliau perduodamas apdoroti programai.
- **Panašių metodų paieška.** Šis etapas atliekamas programos logikoje (ne duomenų bazėje). Toks sprendimas pasirinktas dėl to, kad sudėtingą logika naudojant rekursijas paprasčiau ir optimaliau realizuoti verslo logikos sluoksnyje, bet ne duomenų bazėje naudojant T-SQL. Taip pat pagal nutylėjimą, duomenų bazės serveriai turi apribojimų susijusių su vykdomos rekursijos gyliu. Šį apribojimą galima keisti ne visuose duomenų bazės serveriuose.

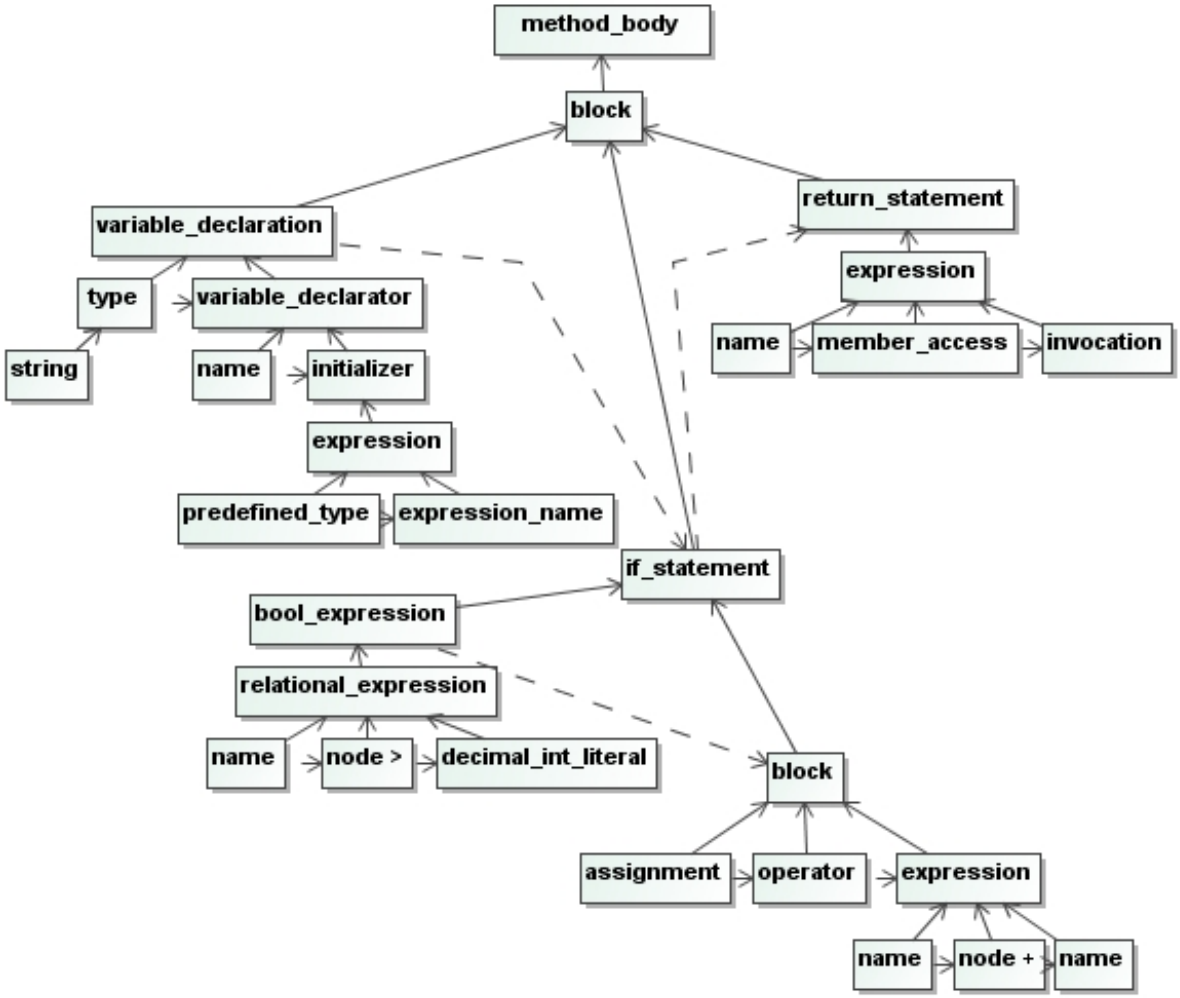


Pav. 5 Testų generavimo veiklos diagrama

Kodo medis

Programavimo kalba parašytas kodas (algoritmas) nėra vien tekstas, tai hierarchinė duomenų struktūra, kitaip tariant medis. Kad būtų lengviau įsivaizduoti, toliau pateikiamas elementarus kodo ir jo medžio pavyzdys.

```
public int SimpleMethodExample(int a)
{
    string doubleLine = string.Empty;
    if (a > 1)
    {
        doubleLine += a + a;
    }
    return doubleLine.Count();
}
```

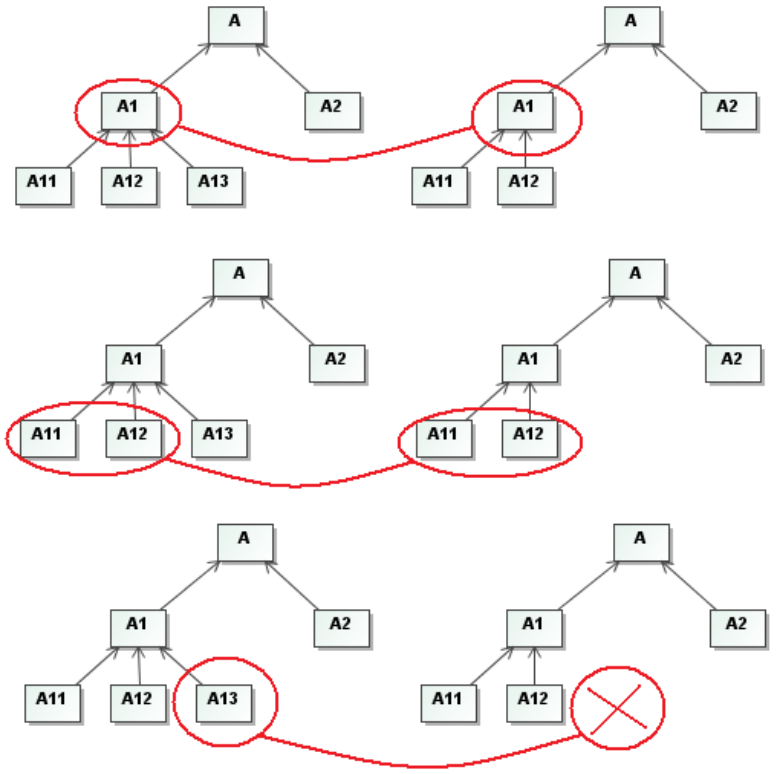


Pav. 6 Kodo medžio pavyzdys

Iš pateikto pavyzdžio matosi, jog suprasti kodą iš jo hierarchinės struktūros yra sudėtingiau nei iš jo tekstinės išraiškos, tačiau analizuojant reikalinga būtent hierarchija.

Kodo medžių palyginimas

Ieškant panašių metodų vienu metu lyginamos dvi hierarchinės kodo struktūros. Lyginant metodų medžius naudojamas gerai žinomas *paieškos gilyn* [5] metodas. Grafų teorijoje šį metodą pirmasis 1972 m. pasiūlė R. Tarjanas. Vadovaujantis šio metodo idėja, medis apeinamas nuo viršūnės gilyn. Šis algoritmas taikomas lygiagrečiai dviejų medžių apėjimui. Aiškumo dėlei pateikiamas grafinis pavyzdys (Pav. 7).



Pav. 7 Medžių palyginimas

Tarpusavyje lyginami Pav. 7 pavaizduoti du medžiai. Tėvinė abiejų medžių viršūnėn *A* turi du vaikus: *A1* ir *A2*. Kadangi abu medžiai turi šias viršūnės, o jos turi vaikinių elementų, tad norėdami sužinoti panašumą turime eiti gilyn. Abiejuose medžiuose viršūnės *A1* vaikiniai elementai savo vaikinių elementų neturi. Tad lyginame jų turimas reikšmes tarpusavyje. Jei viršūnė yra abiejose atšakose, tuomet jų panašumas 100%, jei vienoje atšakoje viršūnės nėra – 0%. Matome, jog abiejuose medžiuose *A1* viršūnė turi *A11* ir *A12* viršūnes, tačiau antras medis neturi *A13* viršūnės, kuri yra pirmame medyje. Tad suskaičiavę viršūnių panašumo vidurkį ((100% + 100% + 0%) / 3 = 66,666%), gauname, jog lyginant abiejų medžių *A1* viršūnes gauname, jog jų panašumas yra 66,666%. Toliau lygindami *A* viršūnių panašumą tuo pačiu būdu

gauname (66,666% + 100%) / 2 = 83.333%. Tai elementarus algoritmo paaiškinimas, detalesnis algoritmas pateikiamas pseudokodu (pavyzdyje pateikta notacija artima C++).

```
1.      decimal GautiPanasuma(Node r1, Node r2)
2.      {
3.          decimal childSum = 0;
4.          decimal childCount = 0;
5.
6.          if (r1 == null || r2 == null) //jei nėra vienos is viršūnių tuomet
7.      //panašumas lygus 0
8.          {
9.              return 0;
10.         }
11.         // jei bent viena viršūnė turi vaikinių elementų - einame gilyn ir
12.     //didiname vaiku skaičių
13.         if (r1.child != null || r2.child != null)
14.         {
15.            // gauname vaikinių elementų panašumą, pradedama paieška gilyn
16.             childSum = GautiPanasuma(r1.child, r2.child);
17.             childCount++; //didiname vaikinių elementų skaičių
18.         }
19.         Node tempR1 = r1;
20.         Node tempR2 = r2;
21.
22.         decimal mIndexSum = 0;
23.         decimal mIndexCount = 1;
24.         //apeiname gretimus viršūnių elementus
25.         while (true)
26.         {
27.             if (tempR1 == null || tempR2 == null)
28.             {
29.                 break;
30.             }
31.           //jei yra gretimų elementų - einame gilyn
32.             if (tempR1.next != null || tempR2.next != null)
33.             {
34.                 //gauname šalia esančių elementų panašumą
35.                 mIndexSum += GautiPanasuma(tempR1.next, tempR2.next);
36.                 mIndexCount++;
37.             }
38.
39.             tempR1 = tempR1.next;
40.             tempR2 = tempR2.next;
41.         }
42.         //pradinis panašumas
43.         decimal match = 0;
44.         if (r1 == r2)
45.         {
46.           //panašumas, kai elementas rastas abiejuose medžiuose
47.             match = (100 + mIndexSum) / mIndexCount;
48.         }
49.         if (childCount > 0)
50.         {
51.              //skaičiuojame dviejų viršunių panašumą
52.             return (match + childSum) / 2;
53.         }
54.
55.         return match;
56.     }
```

Algoritmas 1 Medžių palyginimo algoritmas

Metodų palyginimo pavyzdys

Toliau pateiktas dviejų elementarių metodų palyginimo pavyzdys ir paaiškintas Algoritmas 2 pateiktas algoritmas. Metodai pateikti C# programavimo kalba.

Pirmasis metodas	Antrasis metodas

`public void HelloWorldA(int a, int b)`
`{`
`  int i = 1;`
 	
`  while (i < b)`
`  {`
`  if (a > 1)`
`  {`
`  MessageBox.Show("Hi!");`
`  }`
 	
`  i++;`
`  }`
`}`
	`public void HelloWorldB(int a, int b)`
	`{`
	`  int i = 1;`
	
	`  while (i < b)`
	`  {`
	`  if (a < 1)`
	`  {`
	`  MessageBox.Show("Hi!");`
	`  }`
	
	`  i++;`
	`  }`
	`}`

Lentelė 1 Lyginami metodai (kodas)

Pirmojo metodo medis	Antrojo metodo medis
method_declaration method_header method_body block local_variable_declaration while_statement boolean_expression relational_expression name Node <<> name block if_statement boolean_expression relational_expression name Node <<> decimal_integer_literal block call_or_post_incr_decr name member_access name invocation argument_list argument string_literal call_or_post_incr_decr name post_incr	method_declaration method_header method_body block local_variable_declaration while_statement boolean_expression relational_expression name Node <<> name block if_statement boolean_expression relational_expression name Node <<> decimal_integer_literal block call_or_post_incr_decr name member_access name invocation argument_list argument string_literal call_or_post_incr_decr name post_incr

Lentelė 2 Lyginami metodai (kodo medis)

Medžių apėjimui naudojamas paieškos gilyn metodas. Tai reiškia, kad pirmiausiai aplankomi kiekvieno elemento vaikiniai elementai (Algoritmas 3, 16 eilutė), iki kol bus pasiektas elementas, neturinti vaikinių elementų. Jei aplankius vaikinius elementus, einamasis elementas turi gretimų atšakų, tada algoritmas apeina jas (Algoritmas 4, 25 - 41 eilutės). Pasiekus giliausiai esantį elementą, jis lyginamas su kitame medyje esančiu elementu (Algoritmas 5, 44 eilutė). Jei elementai lygūs, tuomet jų panašumas – 100%. Tėvinių elementų panašumas yra lygus vaikų panašumų aritmetiniam vidurkiui (Algoritmas 6, 47 eilutė).

Lentelėje 3 pateikti metodų, kuriuos lyginsime, kodai. Pažymėtos kodo vietos, kurios skiriasi. Lentelėje 4 pateiktos šių metodų hierarchinės struktūros. Taip pat pažymėti elementai, kurie skiriasi. Metodo antraštė (*method_header* atšaka) yra suskleista, nes šios atšakos identiškos 100%, naudojant šį algoritmą metodų paieškai, antraštės visada sutampa, nes iš duomenų bazės išrenkami metodai tik su vienodais parametrais. Pateiktame pavyzdyje skiriasi metodo kūno elementai (*method_body*). Tad jie ir yra lyginami tarpusavyje. Algoritmui paduodami abiejų medžių *method_body ->block.* Paieška vykdoma gilyn, nes yra vaikinių elementų. Lyginami *method_body->block->local_variable_declaration*. Jie identiški, tad šios atšakos panašios 100%. Algoritmas parenka šalia esančią viršūnę *method_body->block->while_statement,* einama gilyn *method_body->block->while_statement->boolean_expression,* šios atšakos panašios 100%. Algoritmas parenka šalia esančią viršūnę *method_body->block->while_statement->block->if_statement.* Einame gilyn iki šios viršūnės šakose yra nesutapimų. Nesutampa *method_body->block->while_statement->block->if_statement->boolean_expression->relational_expression->Node<<>* ir ... *Node<>>*. Šie elementai sutampa 0%. Kadangi *method_body->block->while_statement->block->if_statement->boolean_expression->relational_expression* turi dar du 100% sutampančius elementus, šios šakos panašumas apskaičiuojamas, kaip aritmetinis vidurkis ir lygus 66,666%. ((100% + 100% + 0%) / 3 = 66,666%). Greta esantis elementas *method_body->block->while_statement->block->if_statement->block* yra lygus 100%. Skaičiuojamas *method_body->block->while_statement->block->if_statement* šakos panašumas (66% + 100%) / 2 = 83.333%. Einame aukštyn ir lyginame *method_body->block->while_statement->block* elementus (83.333% + 100%) / 2 = 91,6665%. Einame aukštyn ir lyginame *method_body->block->while_statement* elementus (91,6665% + 100%) / 2 = 95,8333%. Beliko palyginti *method_body->block* elementus (95,8333% + 100%) / 2 = 97.916625%. Iš pateikto pavyzdžio matome, jog duoti metodų medžiai yra labai panašūs.

3.1.3 ReUnit architektūra

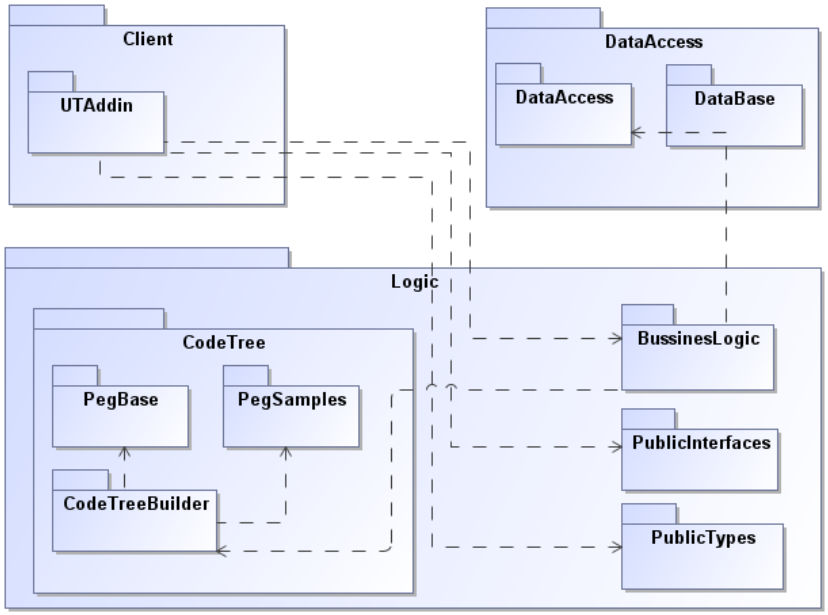
Šiame skyriuje pateiksime sukurtos programinės architektūrą ir pagrindines projektavimo idėjas. Sistemos projektavimui buvo naudojama *MagicDrow UML*[15], *Microsoft Visual Studio Team Suite 2008*. Jau projektuojant sistemą buvo žinoma, kad programinė įranga bus realizuota naudojant *Microsoft .Net framework* technologiją.

3.1.3.1 Tikslai ir apribojimai

Projektavimo etape jau žinoma, kad suprojektuota sistema turi būti prieinama tiek iš *Microsoft .Net*, tiek iš *Java* platformų. Šios problemos sprendimui buvo pasirinkta atskirti klientinę sistemos dalį, o logikos dalį realizuoti panaudojant *Web-servisą*. Web-servisas gali būti apibūdinamas kaip programų sistema, kurios pagrindinis tikslas yra apjungti tinkle esančius kompiuterius. Web-servisai naudoja SOAP (angl. Simple Object Access Protocol) protokolą. SOAP ir HTTP protokolų pagalba Web-servisai perduoda žinutes XML formatu. Web-servisai turi turėti dvi pagrindines savybes: 1. Web-servisas turi aptarnauti visus klientus, nepriklausomai nuo platformos, kurią klientas naudoja; 2. Klientas turi turėti galimybę surasti ir naudoti Web-servisus nepriklausomai nuo jų realizacijos ypatybių ir programinės terpės, kurioje yra Web-servisas. Plačiau apie Web-servisus rašoma šio darbo antrojoje dalyje.

Klientinė sistemos dalis turi būti realizuota kaip programavimo aplinkos įskiepis.

3.1.3.2 Statinis sistemos vaizdas



Pav. 8 Sistemos paketų diagrama

[15] http://www.magicdraw.com/

- **Paketų detalizavimas**
    - **Client paketas**

Šį paketą sudaro klasės skirtos *Visual Studio* įskiepio įgyvendinimui. Grafinės vartotojo sąsajos langai realizuoti kaip COM+ komponentai [11-12]. Jie turi būti registruojami *Windows* sistemos registre.

Pav. 9 Client paketo klasių diagrama

- **Data Access paketas**

Pav. 10 Data Access paketo klasių diagrama

- **CodeTree paketas**

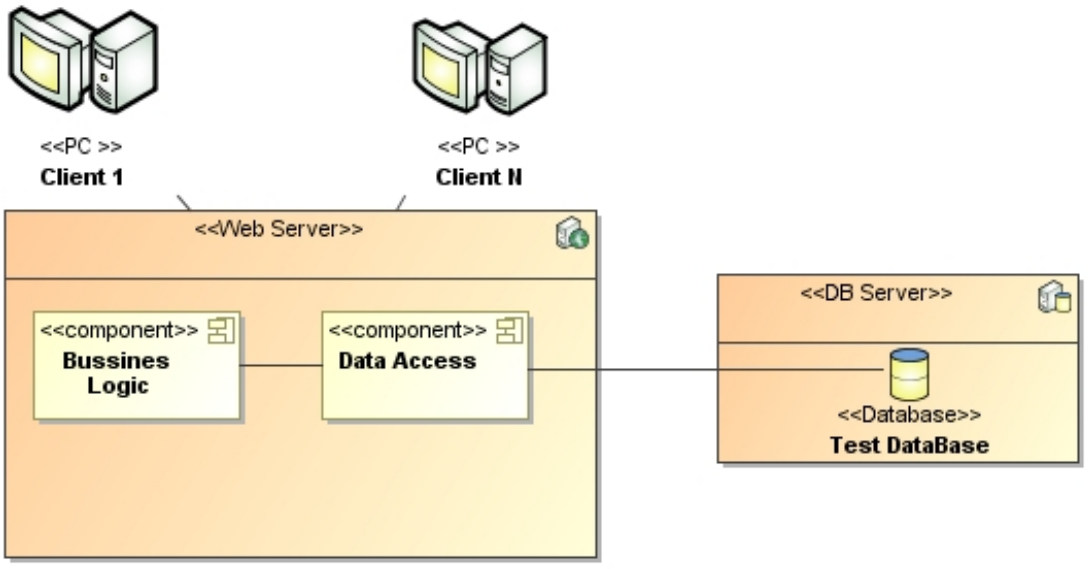
Pav. 11 CodeTree paketo klasių diagrama

- **BussinesLogic**

Pav. 12 BussinesLogic paketo klasių diagrama

3.1.3.3 Išdėstymo vaizdas

Vienas iš sistemos projektavimo reikalavimų yra tas, kad sistema būtų galima praplečiama naujais klientais. Tai reiškia, kad tais pačiais vienetų testais galėtų naudotis skirtingų platformų (pvz.: *.Net* ir *Java*) klientai. Tam verslo logikos sluoksnį turime suprojektuoti taip, kad galėtume diegti atskirai. Sistemos išdėstymo vaizdas pateikiamas žemiau esančioje diagramoje.



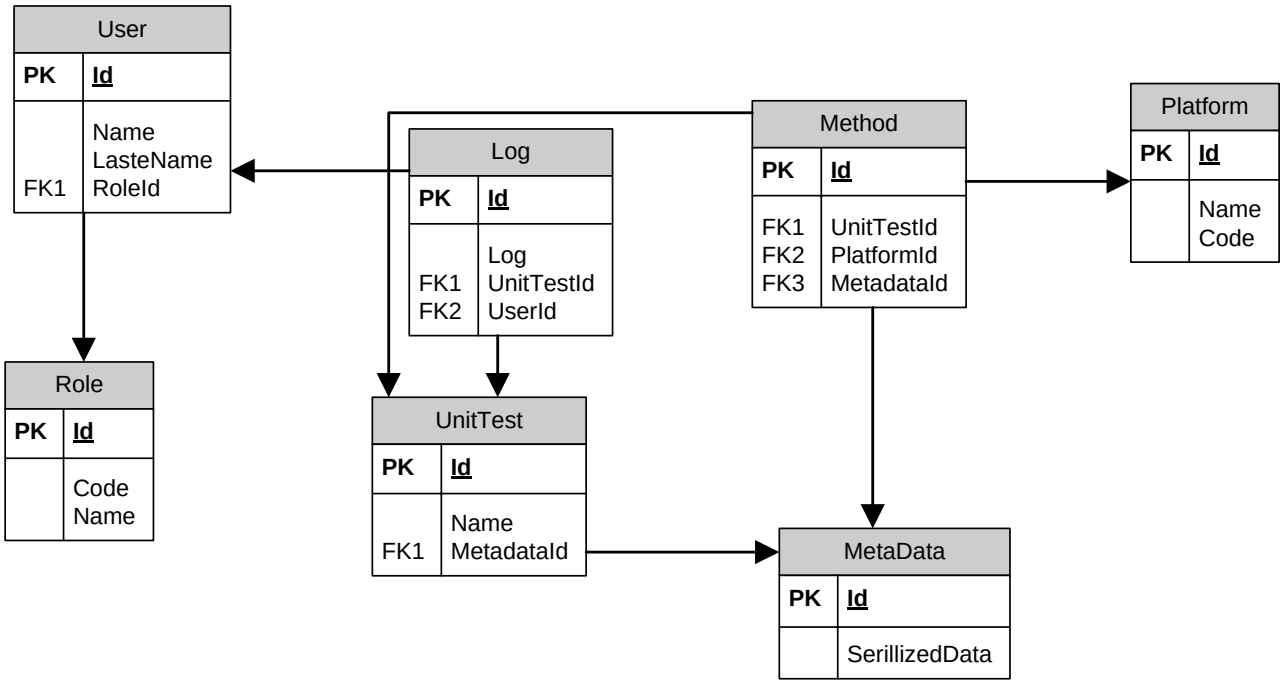
Pav. 13 Sistemos išdėstymo vaizdas

Sistema kuriama ir diegiama remiantis trijų sluoksnių architektūra. Aplinką sudaro šie komponentai:

- DB server – tai duomenų bazės serveris, kuriame yra vienetų testų duomenų bazė.
- Web server – tai web serveris. Naudojamas Microsoft Internet Information Services 6.0. Šiame komponente yra duomenų prieigos komponentas, kurį naudoja verslo logikos komponentas. Verslo logikos komponentas yra realizuotas kaip Web servisas.
- Client 1 .. Client N – tai klientinė sistemos dalis. Čia yra integruota programavimo aplinka bei klientinė vienetų sistemos dalis.

3.1.3.4 Duomenų vaizdas

ReUnit visus duomenis (vienetų testų ir metodų informacija) saugo duomenų bazėje, kurios schema pateikta toliau esančiame paveiksle. Projektuojant duomenų bazę buvo atsižvelgta į galimas greitaveikos problemas, tad duomenų bazėje saugomi metodai ir vienetų testai tekstiniame formate. Toks sprendimas pasirinktas, nes vienetų testų parinkimas atliekamas lyginant kodo medžius. Saugant kodą kaip hierarchinę struktūra duomenų bazėje, tektų atlikti sudėtingus skaičiavimus naudojant *SQL*, o tai šiuo atveju būtų labai sudėtinga ir neoptimalu. Pagrindinė duomenų analizė perkelta į verslo logikos sluoksnį.



Pav. 14 Duomenų vaizdas

3.1.4 Projektavimo etapo rezultatai

Projektavimo etape išanalizuoti sistemai keliami reikalavimai, suprojektuotas algoritmas metodų palyginimui ir vienetų testų parinkimui. Išanalizavus sukurtą algoritmą matome, jog lyginant metodų hierarchines struktūras, didžiausią reikšmę turi aukščiau medyje esančių elementų panašumas.

Aprašyta architektūra įtakoja sistemos išplečiamumą ir pernešamumą. Sistemos architektūra realizuoja trijų sluoksnių architektūros šabloną. Šis projektavimo šablonas skirtas atskirti duomenis, biznio logiką ir atvaizdavimą. Vienetų testų sistemos architektūros atveju šios dalys išskirtos taip:

- *Duomenys* – duomenų bazė su lentelėmis, *stored* procedūromis. Taip pat šiai daliai priklauso klasės darbui su duomenų baze.
- *Biznio logika* – aibė klasių, kuriose yra specifiniai duomenų apdorojimo. Šis sluoksnis prieinamas kaip Web servisas. Tai reikalinga, nes biznio logika prieinama iš įvairių platformų.
- *Atvaizdavimas* – formos, meniu ir kiti vartotojo sąsajos elementai, kuriuose atvaizduojama informacija, tačiau nėra logikos, susijusios su duomenų apdorojimu.

Nors atrodo, kad architektūra labai komplikuota, tačiau jos pranašumas pasireiškia vėliau, kai reikia kurį nors iš sluoksnių redaguoti. Pavyzdžiui, pakeitus duomenų sluoksnį (kad ir lentelės ar lauko pavadinimą ar netgi visą duomenų bazės serverį) užteks pakoreguoti tik biznio logikos sluoksnį, o atvaizdavimo sluoksnio šie pakeitimai neįtakoja.

4 TYRIMO DALIS

Šiame skyriuje pateiksime projekto apimtį (4.1 skyrius) ir ištirsime sukurtos programinės įrangos *ReUnit* kokybę. Kiekybinei analizei atlikti panaudosime *Microsoft Visual Studio 2008* ir *Parasoft .Test* įrankius. Atliksime statinę išeities kodų analizę *Parasoft .Test* įrankiu, kuris tiria kodą pagal *Parasoft* siūlomas geriausias praktikas.

4.1 ReUnit kiekybinė kodo analizė

Paketas	Priežiūros indeksas	Atšakų skaičius	Paveldėjimo gylis	Klasių priklausomybė	Kodo eilutės 1	Kodo eilutės 2
Client\TestGenBase.UTAddin	77	521	7	205	1776	5823
CodeTree\CodeTreeBuilder	91	7	1	12	13	118
CodeTree\PEG Base	86	726	2	48	971	2144
CodeTree\PEG Samples	82	12495	3	138	9494	22772
TestGenBase	84	43	7	58	112	640
TestGenBase.BL	89	798	2	111	1367	3267
TestGenBase.BL.UnitTests	72	11	1	14	481	620
TestGenBase.DataAccess	82	127	2	26	232	889
TestGenBase.DataBase	100	0	0	0	0	0
TestGenBase.PublicTypes	92	20	1	6	26	92

Lentelė 5 ReUnit kiekybinė kodo analizė

Priežiūros indeksas (angl. Maintainability Index**)** [3] **–** tai indeksas, nurodantis kodo palaikomumą. Indeksas gali įgyti reikšmes iš intervalo [1; 100]. Šio indekso skaičiavimui naudojama: eilučių skaičius, atšakų skaičius, taip pat operandų ir operatorių skaičius. Reikšmės iš intervalo [20; 100] reiškia, kad kodo palaikomumas yra aukštas.

Atšakų skaičius (angl. Cyclomatic Complexity**) –** tai sąlygos taškų (if, switch) ir ciklo skaičius. Tai geras indikatorius, nusakantis kiek reikia vienetų testų, kad būtų pasiektas pilnas kodo padengimas. Geriau kai šis skaičius yra mažesnis.

Paveldėjimo gylis (angl. Depth of Inheritance**) –** tai klasės paveldėjimų medžio gylis. Didelis paveldėjimo gylis rodo, jog projektuojant persistengta. Tai sukelia problemų testuojant, taip pat tokį kodą sunku palaikyti ateityje.

Klasių priklausomybė (angl. Class Coupling**) –** tai priklausomybių skaičius, reiškiantis kiek viena klasė priklauso nuo kitų. Kuo šis skaičius mažesnis tuo lengviau galima pakartotinai panaudoti klasę.

Kodo eilutės 1 (angl. Lines of Code**) –** tai bendras eilučių skaičius, neįskaitant tuščių eilučių ir komentarų.

Kodo eilutės 2 (angl. Lines of Code**) –** tai bendras eilučių skaičius, įskaitant tuščias eilutes ir komentarus.

4.2 Statinė ReUnit išeities kodų analizė

Šioje dalyje atliksime ReUnit įrankio statinę kodo analizę [4]. Analizuodami remsimės *.Test* sugeneruotomis ataskaitomis. Darbą atliksime su visais 4.1 skyriuje išvardintais projektais išskyrus *TestGenBase.DataBase,* nes šis projektas skirtas duomenų bazės skriptams, kurie parašyti SQL kalba, o ji čia neanalizuojama. Statinei analizei atlikti panaudosime skirtingus taisyklių rinkinius, analizuojančius išeities tekstus skirtingais aspektais.

Taisyklių rinkinys	**Taisyklių skaičius**	**Paskirtis**
BugDetective	30	Tai taisyklių rinkinys, skirtas surasti standartinėms klaidoms, tokioms kaip: galima dalyba iš nulio, sunaikintų resursų naudojimas, duomenų bazės resursų laikymas ir pan.
Critical Rules	35	Tai kritinės klaidos, tokios kaip netinkamas tipų konvertavimas, objektų serilizavimas, operatorių perdengimas ir pan.
Efective C#	85	Tai taisyklės skirtos patikrinti ar projektavimo ir programavimo stilius yra pakankamai efektyvus. Pavyzdžiui visos kolekcijos turi įgyvendinti *ICollection* sąsają (angl. Interface).
Security Assessment	75	Tai klaidos susijusios su saugumu. Tikrinama, kad tinkamai būtų dirbama su objektų rodyklėmis, ieškoma potencialių SQL įterptinių sakinių galimybių ir pan.
COM Guidelines	5	Tai pagrindinės taisyklės, kurių reikia laikytis programuojant COM komponentus. Šis taisyklių rinkinys neįtrauktas į pagrindinių taisyklių rinkinių sąrašą, tačiau šiame projekte naudojami COM komponentai, tad šį rinkinį įtraukiau atlikdamas statinę kodo analizę.

Lentelė 6 Statinės kodo analizės taisyklės

- **TestGenBase.UTAddin išeities kodų analizė**

Taisyklių rinkinys	**Taisyklių skaičius**	**Pažeidimų kiekis**	**Pažeidimų kiekis 1000 eilučių**

BugDetective	30	16	2,74
Critical Rules	35	30	5,15
Efective C#	85	90	15,46
Security Assessment	75	150	25,76
COM Guidelines	5	2	0,34

Lentelė 7 TestGenBase.UTAddin projekto taisyklių pažeidimai

- **CodeTree.CodeTreeBuilder išeities kodų analizė**

Taisyklių rinkinys	**Taisyklių skaičius**	**Pažeidimų kiekis**	**Pažeidimų kiekis 1000 eilučių**
BugDetective	30	0	0
Critical Rules	35	0	0
Efective C#	85	2	16,94
Security Assessment	75	0	0
COM Guidelines	5	0	0

Lentelė 8 CodeTree.CodeTreebuilder projekto taisyklių pažeidimai

- **CodeTree.PEG Base išeities kodų analizė**

Taisyklių rinkinys	**Taisyklių skaičius**	**Pažeidimų kiekis**	**Pažeidimų kiekis 1000 eilučių**
BugDetective	30	11	5,1
Critical Rules	35	17	7,8
Efective C#	85	23	10,6
Security Assessment	75	89	41,1
COM Guidelines	5	0	0

Lentelė 9 CodeTree.Peg Base projekto taisyklių pažeidimai

- **CodeTree.PEG Samples išeities kodų analizė**

Taisyklių rinkinys	**Taisyklių skaičius**	**Pažeidimų kiekis**	**Pažeidimų kiekis 1000 eilučių**
BugDetective	30	44	1,9
Critical Rules	35	30	1,3
Efective C#	85	50	2,1
Security Assessment	75	220	9,6
COM Guidelines	5	0	0

Lentelė 10 CodeTree.Peg Samples projekto taisyklių pažeidimai

- **TestGenBase išeities kodų analizė**

Taisyklių rinkinys	**Taisyklių skaičius**	**Pažeidimų kiekis**	**Pažeidimų kiekis 1000 eilučių**
BugDetective	30	4	6,25
Critical Rules	35	6	9,3
Efective C#	85	23	35,9
Security Assessment	75	4	6,25

COM Guidelines	5	1	1,56

Lentelė 11 TestGenBase projekto taisyklių pažeidimai

- **TestGenBase.BL išeities kodų analizė**

Taisyklių rinkinys	Taisyklių skaičius	Pažeidimų kiekis	Pažeidimų kiekis 1000 eilučių
BugDetective	30	4	1,22
Critical Rules	35	8	2,44
Efective C#	85	38	11,6
Security Assessment	75	49	14,99
COM Guidelines	5	0	0

Lentelė 12 TestGenBase.BL projekto taisyklių pažeidimai

- **TestGenBase.UnitTests išeities kodų analizė**

Taisyklių rinkinys	Taisyklių skaičius	Pažeidimų kiekis	Pažeidimų kiekis 1000 eilučių
BugDetective	30	4	6,4
Critical Rules	35	7	11,29
Efective C#	85	2	3,2
Security Assessment	75	3	4,8
COM Guidelines	5	1	1,6

Lentelė 13 TestGenBase.UnitTests projekto taisyklių pažeidimai

- **TestGenBase.DataAccess išeities kodų analizė**

Taisyklių rinkinys	Taisyklių skaičius	Pažeidimų kiekis	Pažeidimų kiekis 1000 eilučių
BugDetective	30	2	2,2
Critical Rules	35	1	1,1
Efective C#	85	0	0
Security Assessment	75	4	4,4
COM Guidelines	5	0	0

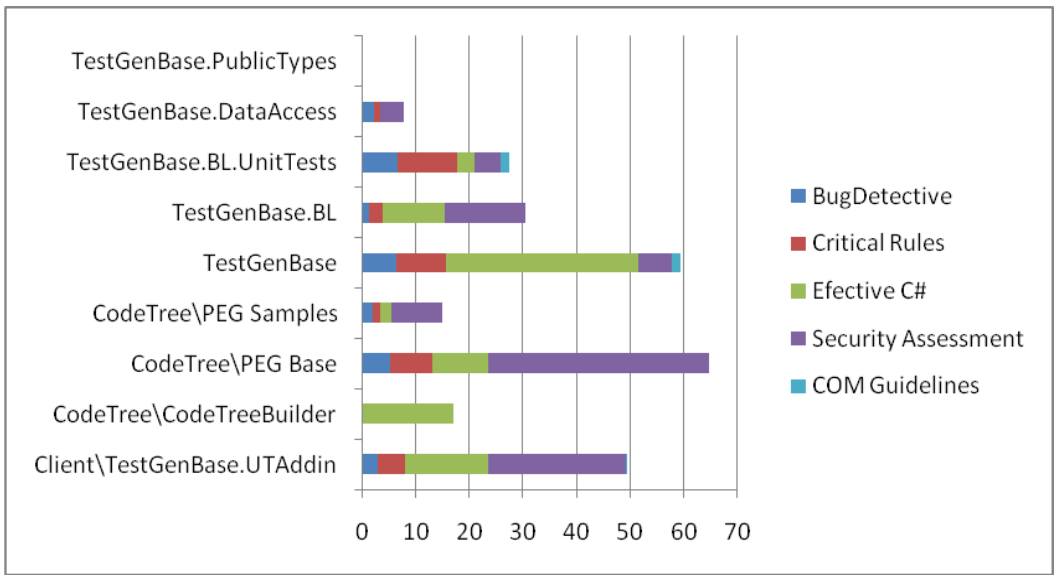
Lentelė 14 TestGenBase.DataAccess projekto taisyklių pažeidimai

- **TestGenBase.PublicTypes išeities kodų analizė**

Taisyklių rinkinys	Taisyklių skaičius	Pažeidimų kiekis	Pažeidimų kiekis 1000 eilučių
BugDetective	30	0	0
Critical Rules	35	0	0
Efective C#	85	0	0
Security Assessment	75	0	0
COM Guidelines	5	0	0

Lentelė 15 TestGenBase.PublicTypes projekto taisyklių pažeidimai

- **Projektų pažeidimai 1000 eilučių**



Pav. 15 Projektų taisyklių pažeidimai 1000 eilučių

4.3 ReUnit vienetų testų analizė

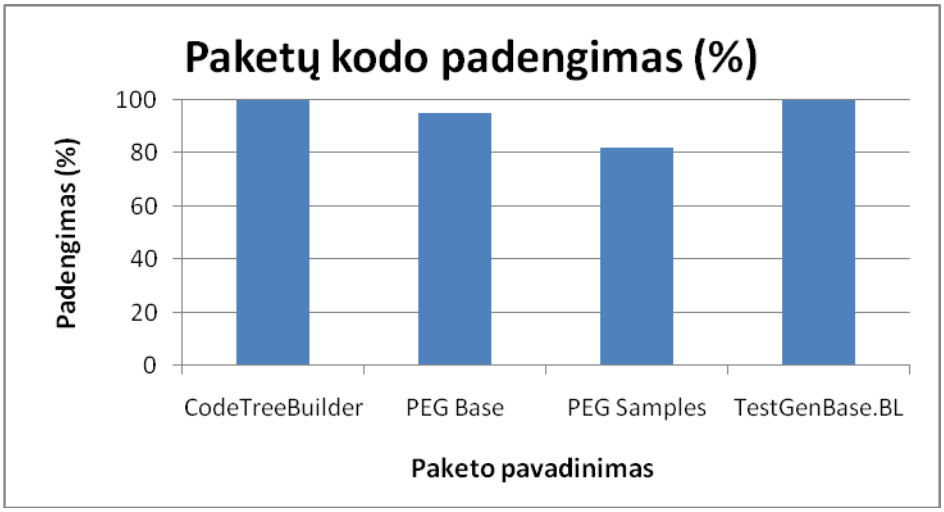
Testavimą vienetų testais suskaidėme į dvi dalis. Pirmojoje testavome programinę įrangą tikėdamiesi rasti nenumatytų išimtinių atvejų. Tam panaudojome vienetų testų generavimą, kad išgautume kuo didesnį kodo padengimą vienetų testais. Sugeneravę testus juos vykdėme tikėdamiesi, jog vienetų testai atskleis silpnąsias kodo vietas ir gausime išimtinius (nenumatytus) atvejus (*Exception*). Buvo matuojamas kodo procentinis padengimas, kadangi vien generuoti vienetų testai nesuteikia aukšto kodo padengimo, tad sugeneruoti testai buvo redaguojami, keičiant metodų įėjimų parametrus atsižvelgiant į metodo šakų vykdymą. Gavus rezultatus, jie buvo analizuojami ir iš rastų klaidų sąrašo pašalinamos tos, kurios sukeltos sugeneravus testavimo atvejus su logiškai neįmanomais įėjimais.

- **Vienetų testų generavimas ir vykdymas**

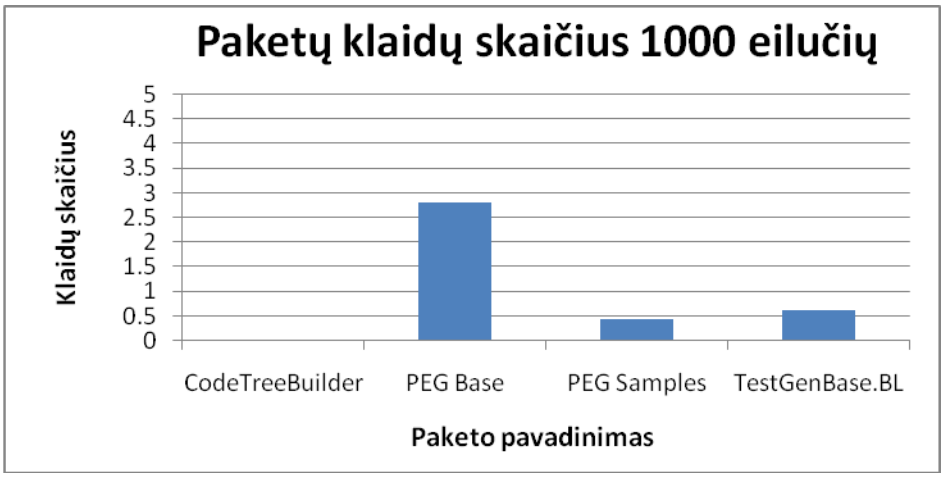
Lentelėje pateikiami testavimo vienetų testais rezultatai. Dviejuose projektuose pasiekiamas kodo padengimas yra mažesnis nei 100%. Šie projekto paketai yra sukurti ne konkrečiai šiam projektui, tad turi galimybių, kurios čia nenaudojamos, todėl taisant sugeneruotus vienetų testus buvo nesistengiama padengti visų šių atvejų.

Projektas	**Sugeneruotų vienetų testų skaičius**	**Testų, atskleidusių klaidas, skaičius**	**Klaidų kiekis 1000 eilučių**	**Padengtos eilutės (%)**
CodeTree\CodeTreeBuilder	10	0	0	100
CodeTree\PEG Base	245	6	2,8	95
CodeTree\PEG Samples	560	10	0,44	82
TestGenBase.BL	500	2	0,61	100

Lentelė 16 Sugeneruoti ir įvykdyti vienetų testai



Pav. 16 Testuojamų paketų kodo padengimas



Pav. 17 Paketų klaidų skaičius 1000 eilučių

- **Logikos testavimas vienetų testais**

Pagrindinis testavimo vienetų testais tikslas – padengi kuo daugiau kodo. Logikos padengti neįmanoma. Paketo TestGenBase.BL testavimui buvo rašomi vienetų testai, nes čia yra visa pagrindinė, susijusi su testų saugojimu ir parinkimu, sistemos logika. Čia vienetų testai buvo kuriami vykdomi [13] lygiagrečiai su metodų kūrimu. Pavyzdžiui metodo skirto, kodo medžių lyginimui, vienetų testai buvo skurti pirmiau nei pačių metodų logika. Nes iš anksto buvo žinomi pavyzdiniai kodo medžiai ir jų palyginimo rezultatai. Turimi vienetų testai pasitarnavo kaip aplinka kuriamų metodų vykdymui, nes dar nebuvo sukurta vartotojo sąsaja skirta darbui su šiais metodais. Šie vienetų testui buvo vykdomi pakartotinai, atlikus pakeitimus programos kode.

4.4 ReUnit grafinės vartotojo sąsajos testavimas

Grafinės vartotojo sąsajos (toliau tekste GUI) testavimui [15] rašyti vienetų testus, vykdančius GUI metodus, yra komplikuota, nes grafinės sąsajos elementai sąveikauja su programos logika per vartotojo sukeltus įvykius. Plačiausiai GUI automatizuotam testavimui [8] naudojama metodika yra programos veikimo scenarijų sukūrimas ir pakartinis vykdymas.

ReUnit GUI testavimui buvo naudotas *Test Automation FX*[16]. Primityviausių atvejų testavimu buvo naudojamas automatinis scenarijaus įrašymas. Tai atliekama tiesiogiai sąveikaujant su testuojamos PĮ GUI elementais, o kiekvienas atliktas veiksmas įrašomas ir saugomas. Vėliau tereikia atkartoti įrašytą scenarijų. Sudėtingesnių atvejų testavimui rašomas kodas, kurio pagalba galima sąveikauti su GUI elementais (atliekami mygtukų paspaudimai, meniu elementų pasirinkimai ir t.t.).

Akivaizdi automatinio GUI testavimo nauda pasireiškia, kai atliekami pakeitimai programos kode, o scenarijaus atkartojimas testuojant rankomis užima daug laiko.

ReUnit GUI paketą (TestGenBase.UTAddin) sudaro 68 grafinės vartotojo sąsajos įvykiai.

Paketo pavadinimas	Įvykių skaičius	Testavimo atvejų skaičius	Klaidos
TestGenBase.UTAddin	68	234	15

Lentelė 17 Pirmo automatizuoto GUI testavimo rezultatai

Lentelėje pateikti grafinės vartotojo sąsajos rezultatai sudarius testavimo scenarijus. Tai pradinio testavimo rezultatai. Sukurti scenarijai buvo nuolatos pakartotinai vykdomi atlikus pakeitimus programos kode. Papildant grafinę vartotojo sąsają naujais elementais, reikia papildyti scenarijus įtraukiant naujų elementų veiksmus.

4.5 ReUnit kokybės analizės išvados

Atlikus tiriamąjį darbą padarytos šios išvados:

- Kiekybinės analizės rezultatas yra parodytas ReUnit projekto apimtis ir sudėtingumas. 4.1 skyriuje pateiktoje lentelėje matomi kiekybiniai projekto dalių įverčiai. Išanalizavę šiuos duomenis galime daryti pradines išvadas apie kodo kokybę. Pateiktas „Priežiūros indeksas“ parodo, jog pagal *Microsoft* standartus [4] kodas yra pakankamos kokybės, t.y. šis indeksas ženkliai viršija rekomenduojamą minimalią reikšmę 20 (mažiausia gauta reikšmė 72).

[16] http://www.testautomationfx.com

- Statinė kodo analizė atskleidė potencialias programavimo klaidas ir projektavimo trūkumus. Norint gerinti sukurtos programinės įrangos kokybę būtina detaliai peržiūrėti visus statinės analizės taisyklių pažeidimus. Kartu su taisyklėmis prie *.Test* įrankio pateikiama ir dokumentacija, kurioje taisyklės aprašomos detaliau, tad galima remtis šia informacija, kad būtų galima lengviau atsirinkti, kurios taisyklės yra aktualios, o į kurias galime nekreipti dėmesio.
- Geriausių praktikų atitikimo tikrinimas didžiąja dalimi yra informacinio pobūdžio, tad įrankio pateiktais rezultatais aklai vadovautis nereiktų.
- Testavimas sugeneruotais vienetų testais atskleidė nemažai išimtinių atvejų (Exceptions), į kuriuos nebuvo įsigilinta atliekant programavimo darbus. Išanalizavus gautus duomenis rastos kodo vietos, kurias reikia tobulinti.
- Sugeneruoti vienetų testai nesuteikia 100% kodo padengimo, tad analizuodami kodo padengimo duomenis, turėtume parašyti vienetų testus rankomis, kad padengtume tas kodo vietas, kurios vis dar gali turėti neatskleistų klaidų.
- GUI automatizuotas testavimas naudingas, esant poreikiui dažnai kartoti sudėtingus scenarijus, atlikus pakeitimus programiniame kode.
- Išeities kodo logikos tikrinimui vienetų testai turi būti rašomi rankomis. Šiame darbe rankomis rašyti testai *TestGenBase.BL* paketo klasėms. Tai padėjo išvengti klaidų keičiant duomenų struktūras.

5 EKSPERIMENTINĖ DALIS

Realizuotą idėją saugoti ir pakartotinai panaudoti vienetų testus išbandėme su realia sistema. Eksperimentui buvo pasirinktas užsakymų aptarnavimo projektas (toliau šis projektas nebus aprašomas, tyrimui bus naudojamas tik jo kodas). Ši sistema puikiai tiko eksperimentui, nes didžiąją dalį kodo sudaro verslo logikos kodas. Kode nemažai struktūriškai panašių metodų.

Eksperimento tikslas

Eksperimento tikslas – praktiškai įsitikinti, jog vienetų testų saugojimas ir pakartotinis panaudojimas veikia. Šio eksperimento rezultatas yra testuojamo metodo kodo eilučių padengimo palyginimas su anksčiau testuoto, panašaus metodo kodo eilučių padengimu.

Eksperimento įrankiai

Pagrindinis įrankis eksperimento atlikimui yra sukurta sistema *ReUnit*. Papildomai panaudotas *Parasoft .Test* įrankis, kad galėtume sugeneruoti pradinius vienetų testus ir matuoti kodo padengimą.

5.1 Testuojamo projekto kiekybinė kodo analizė

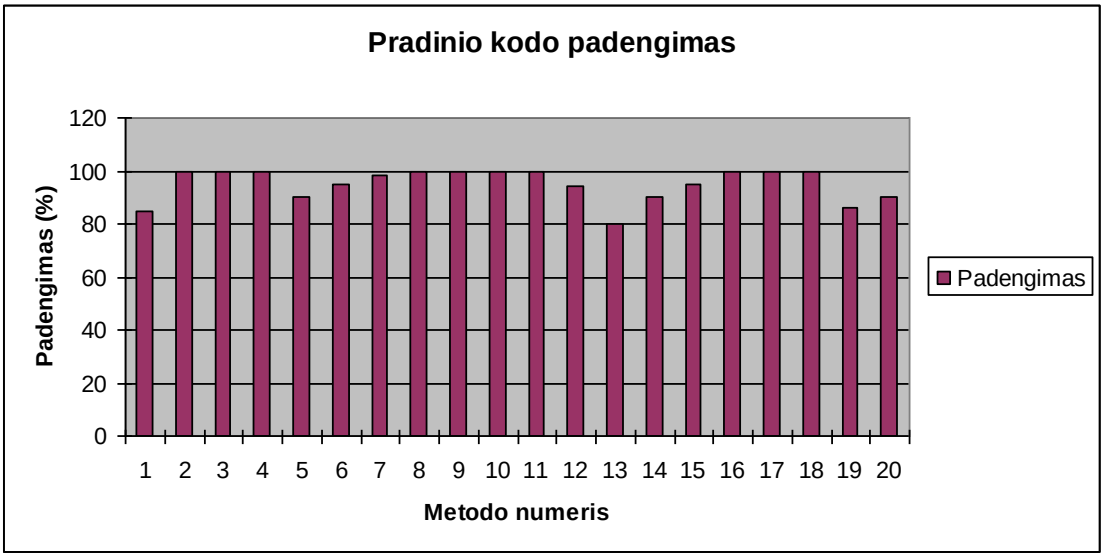
Paketas	Atšakų skaičius	Paveldėjimo gylis	Klasių priklau-somybė	Kodo eilutės 1	Kodo eilutės 2
MNPN.SelfService.Web	4309	7	324	12976	21890
MNPN.SelfService.MNResourses	2	1	2	300	321
MNPN.SelfService.Integration	9	1	2	16	68
MNPN.SelfService.Framework	437	6	141	942	1967
MNPN.SelfService.Exceptions	1	2	2	24	58
MNPN.SelfService.DB	0	0	0	0	0
MNPN.SelfService.DataAccess	4825	3	239	12975	16328
MNPN.ComboBox	51	7	20	84	271
MNPN.SelfService.Controls	135	7	71	244	360
MNPN.EnumDataSource	47	4	28	82	143

Lentelė 18 Testuojamo projekto kiekybinė kodo analizė

5.2 Testavimo metu parinktų vienetų testų įvertinimas

Tiriamam kodui, *.Test* įrankio pagalba, sugeneruoti vienetų testai. Kiekvienam metodui generuota po į vienetų testus. Šie metodai kartu su vienetų testais išsaugoti testų duomenų bazėje. Išsaugojus testus, parinkta 20 metodų, kurių dar nėra duomenų bazėje ir jiems surasti testai iš jau išsaugotų testų. Sistema lygindama testuojamų metodų panašumą parinko 100 vienetų testų naujo kodo testavimui (kiekvienam metodui po 5 kaip ir buvome sugeneravę pradiniam kodui). Pav. 14

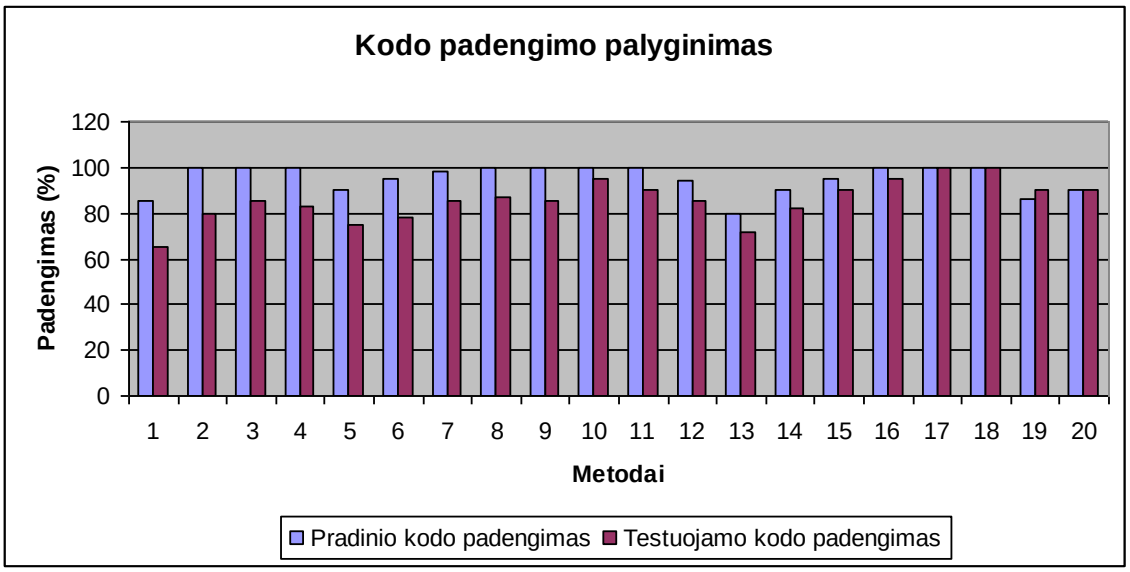
pateikiame jau išsaugoto kodo procentinį padengimo grafiką. Vėliau šiuos duomenis galėsime lyginti su naujo kodo padengimui parinktais vienetų testais.



Pav. 18 Pradinio kodo padengimas (%)

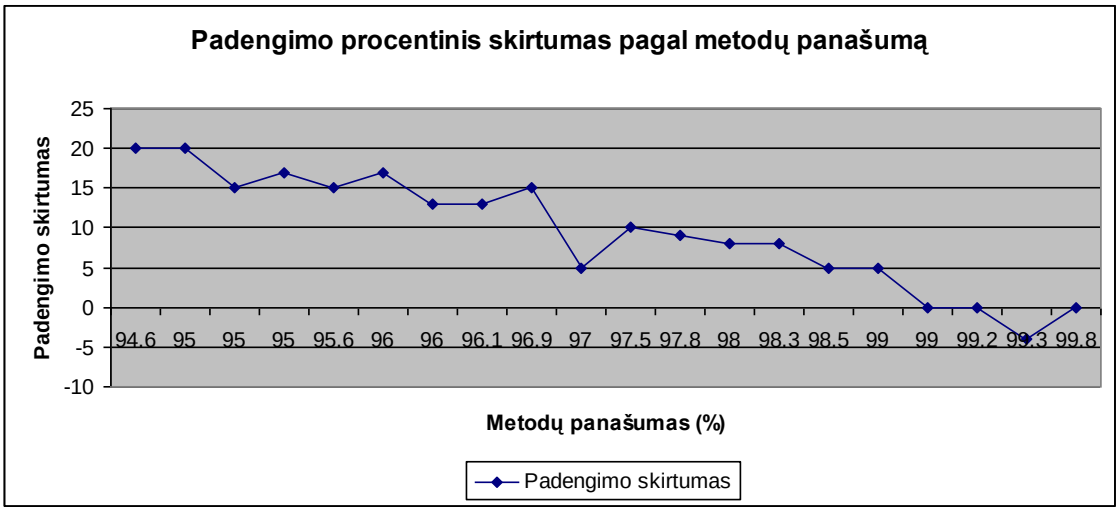
Parinkdami panašius metodus iš duomenų bazės nustatėme paieškos kriterijų, kad būtų parinkti tik panašiausi metodai, tai yra, tik tokie, kurių panašumas su testuojamu metodu, yra ne mažesnis kaip 94% (plačiau apie kodo lyginimą ir panašumą 3.1.2 skyriuje).

Pagal testuojamų metodų panašumą parinktais testais buvo testuojami nauji metodai ir matuojamas kodo padengimas. Lyginant su pradinio (išsaugoto) kodo padengimo duomenimis, testuojamo kodo padengimas yra mažesnis dėl to, kad parinkti testai skirti panašiems savo struktūra metodams (ne identiškiems). Palyginimas pateiktas žemiau esančiame grafike.



Pav. 19 Kodo padengimo palyginimas

Gautus rezultatus įvertinome pagal parinktais testais testuojamų metodų panašumą su išsaugotais testais.



Pav. 20 Metodų padengimo procentinis skirtumas

5.2.1 Testų parinkimo pavyzdys

Pateiksime detalų pavyzdį su duomenimis paimtais iš šiame skyrių atlikto tyrimo. Pavyzdyje paimti 2 panašūs metodai, kurių vienas turi jam sukurtus vienetų testus.

Pirmasis metodas

```
1.   public bool AllRequestRequestLinesRequisitesFullyFilled(int requestId, int q)
2.   {
3.       using (MNPN_SelftServiceDBDataContext context = GetDataContextForDataSource())
4.       {
5.           var requestLines = from rl in context.RequestLines
6.                              where rl.RequestId == requestId select rl;
7.
8.           foreach (RequestLine line in requestLines)
9.           {
10.              foreach (StationRequestLine stationRequestLine in line.StationRequestLines)
11.              {
12.                  decimal quantity = stationRequestLine.Quantity;
13.                  decimal requisiteQuantity = 0;
14.
15.                  foreach (Requisite requisite in stationRequestLine.Requisites)
16.                  {
17.                      foreach (RequisiteLine requisiteLine in requisite.RequisiteLines)
18.                      {
19.                          requisiteQuantity += requisiteLine.Quantity;
20.                      }
21.                  }
22.
23.                  if (q < quantity)
24.                  {
25.                      return false;
26.                  }
27.              }
28.          }
29.      }
30.
31.      return true;
32.  }
```

Lentelė 19 Metodas, turintis testus

Antras metodas

```
1.   public bool IsStationRequestLinesRequisitesFullyFilled(int stationRequestLineId, int q)
2.   {
3.       using (MNPN_SelftServiceDBDataContext context = GetDataContextForDataSource())
4.       {
5.           var stationRequestLines = from rl in context.StationRequestLines
6.                                     where rl.Id == stationRequestLineId
7.                                     select rl;
8.
9.           foreach (StationRequestLine line in stationRequestLines)
10.          {
11.              StationRequestLine stationRequestLine = stationRequestLines.First();
12.              decimal quantity = stationRequestLine.Quantity;
13.              decimal requisiteQuantity = 0;
14.
15.              foreach (Requisite requisite in stationRequestLine.Requisites)
16.              {
17.                  foreach (RequisiteLine requisiteLine in requisite.RequisiteLines)
18.                  {
19.                      if (requisiteLine.Quantity < 0)
20.                      {
21.                          requisiteLine.Quantity = 0;
22.                      }
23.                      requisiteQuantity += requisiteLine.Quantity;
24.                  }
25.              }
26.
27.              if (q < quantity)
28.              {
29.                  return false;
30.              }
31.          }
32.      }
33.
34.      return true;
35.  }
```

Lentelė 20 Testuojamas metodas

Antro metodo kode paryškintos eilutės, kuriomis šis metodas skiriasi nuo pirmojo. Šių metodų panašumas yra 94,6%. Iš kairės pusės pabrauktos eilutės, kurių vienetų testais padengti nepavyko. Antrajame metode kodas esantis nuo 27 iki 30 eilutės yra analogiškas kodui, esančiam pirmame metode nuo 23 iki 26 eilutės, tačiau skiriasi duomenų struktūros, kuriose atliekami skaičiavimai, tad antrajame metode šio kodo padengti nepavyko. Pirmojo metodo kodas vienetų testais buvo padengtas 100%. Naudojant tuos pačius testus kitam metodui pavyko pasiekti 80% kodo padengimą. Esant tokiai situacijai logiška papildyti parinktą testavimo atvejį dar vienu ar keliais testais, kad būtų padengta likusi kodo dalis.

5.3 Eksperimento rezultatai

Atlikus eksperimentą įsitikinta, jog galimą atlikti testavimą su vienetų testais, kurie buvo rašyti kitiems metodams ir gautos išvados:

- Testuojamam metodui turime parinkti kuo panašesnius metodus, taip pasieksime kodo padengimą, kuris yra artimas, išsaugoto metodo padengimui;

- Metodų, kurių panašumas su testuojamais metodais yra mažesnis nei 94%, testai dažniausiai netinka, nes testuojamo metodo padengimas gali skirtis nuo išsaugoto daugiau nei 20%;
- Testuojant panašius metodus įmanoma gauti didesnį kodo padengimą tuo atvejų kai metodai yra panašūs, o skirtumas tik toks, jog testuojamas metodas turi mažiau kodo, tiksliau neturi to kodo, kuris išsaugotame metode yra nepadengtas, tai reta situacija, tačiau eksperimento metu gautuose rezultatuose matosi prie 19 metodo padengimo palyginimo (testuojamas metodas padengtas 90%, o išsaugotas 86%);
- Esant 97% ir didesniam metodų panašumui, kodo padengimas skiriasi ne daugiau kaip 10%, jei pradinio kodo padengimas siekia 90% tuomet tikėtina, jog gaunamas rezultatas bus aukštas, t.y. pasieksime 80% ir didesnį kodo padengimą.

6 IŠVADOS

Darbe išanalizuotos galimybės generuoti ir pakartotinai panaudoti vienetų testus. Pasiūlytas metodas parinkti vienetų testus pagal testuojamo metodo hierarchinę struktūrą.

Analizuojant generavimo galimybes apžvelgti skirtingi vienetų testų generavimo būdai ir įrankiai, apimant būdus nuo testo šablono sugeneravimo ir evoliucinio visos klasės testavimo atvejo generavimo.

Siūlomas metodas, saugoti vienetų testus duomenų bazėje ir pakartotinai panaudoti, nėra pakaitalas kitiems metodams, jį tikslinga naudoti kartu su kitais populiariais įrankiais, pvz. *Parasoft .Test*. Taip pasiekiamas didesnis kodo padengimas, testuojant panašų kodą, kuriam jau turime išsaugotus testavimo atvejus. Pagrindinė metodo idėja – lyginti testuojamą metodą su jau ištestuotu ir radus panašią metodo kodo struktūrą jam pritaikyti ištestuoto metodo vienetų testus.

Idėja praktiškai realizuota įrankiu *ReUnit,* tai į programavimo aplinką integruotas įrankis, kurio pagalba galima lengvai išsaugoti naujai parašytus testus kartu su jų testuojamais metodais. Kelių klavišų paspaudimu galima lengvai paimti, panašių metodų testus iš duomenų bazės. Esant nepakankamam testų skaičiui duomenų bazėje, galima likusius testus sugeneruoti įrankiu *.Test* arba parašyti.

Sistemos projektavimo metu numatyta įrankio praplėtimo galimybė – visa logika iškelta į atskirą mašiną, su kuria bendrauja klientinė dalis per SOAP protokolą, tai įgalina priimti duomenis įvairiose platformose. Detaliai, su paaiškinimais pateiktas metodų lyginimo algoritmas.

Atlikus praktinį tyrimą, paaiškėjo, jog siūlomas metodas labiausiai pasiteisina esant didesniam struktūriniam kodo panašumui (panašumas ne mažesnis nei 94%). Tai naudinga, testuojant panašios probleminės srities programinį kodą.

7 LITERATŪRA

[1] **Gupta M.**, Possibility of Reuse in Software Testing, // 6th annual International Software Testing Conference in India, 2006.

[2] **Wappler S.**, Automatic Generation Of Object-Oriented Unit Tests Using Genetic Programming, 2007, p. 53-98.

[3] **Morrison C.,** Maintainability Index Range and Meaning, http://blogs.msdn.com/fxcop/archive/2007/11/20/maintainability-index-range-and-meaning.aspx, 2007.

[4] **Kean D. M.,** New for Visual Studio 2008 - Code Metrics, http://blogs.msdn.com/fxcop/archive/2007/10/03/new-for-visual-studio-2008-code-metrics.aspx, 2007**.**

[5] **Plukas K., Mačikėnas E., Jarašiūnienė B.**, Taikomoji diskrečioji matematika, 2006, p. 66.

[6] **E.Pulier, H.Taylor.** Understanding Enterprise SOA. Manning, 2006

[7] **Packevičius Š., Ušaniov A., Bareiša E.**, Universal unit tests generator based on software models. // Information Technologies' 2009 : proceedings of the 15th International Conference on Information and Software Technologies, IT 2009, Kaunas, Lithuania, April 23-24, 2009 / Kaunas University of Technology. Kaunas : Technologija. ISSN 2029-0020. 2009, p. 201-206.

[8] **Memon A. M**. 2002. GUI Testing: Pitfalls and Process. *Computer* 35, 8 (Aug. 2002), 87-88

[9] **Hong Z, Patrick A. V., Hall J., May H. R.**, Software Unit Test Coverage and Adequacy, ACM Computing Surveys, 1997, 366-427

[10] **Bernal M., Estrada H.**, **Figueroa-Nazuno J.**: Code Similarity on High Level Programs CoRR abs/0710.5547: 2007

[11] **Raj G. S.**, Introduction to COM+, http://gsraj.tripod.com/com/basic_com/introduction.html, 2007.

[12] **Gunderloy M.**, Calling COM Components from .NET Clients, http://msdn.microsoft.com/en-us/library/ms973800.aspx 2001.

[13] **Hunt A. D. Thomas, Hargett M.**, Pragmatic Unit Testing in C# with NUnit, 2nd Edition, 2007.

[14] **Xie T. D., Notkin D**, Tool-Assisted Unit-Test Generation and Selection Based on Operational Abstractions, in Proceedings of the 16th IEEE Conference on Automated Software Engineering. 2003.

[15] **Rushby J.**, Automated Test Generation And Verified Software. 2008.

[16] **Gold M.**, Creating Visual Studio Add-Ins. http://www.c-sharpcorner.com/UploadFile/mgold/ AddIns11292005015631AM/AddIns.aspx. 2007.

[17] **Wilkes S. L.** Understanding Service-Oriented Architecture. Microsoft Architect Journal**,** 2004

[18] **Hashimi S**. “Service-Oriented Architecture Explained”, O'Reilly Media http://dev2dev.bea.com/technologies/soa/articles/soa_hashimi,jsp, 2004.

[19] **Colan M.** Service-Oriented Architecture expands the vision of Web services, http://www.ibm.com/developerworks/library/ws-soaintro.html 2004.

[20] **Meijer E, Gough J.** Technical Overview of the Common Language Runtime.

[21] **Booth D., Haas H., Mccabe F**. Web Services Architecture. http://www. w3.org/TR/ws-arch/

[22] **Weerawarana S., Curbera F., Leymann F.** Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WSBPEL, WS-Reliable Messaging, and More. Prentice Hall PTR, New Jersey, 2005, p. 27–113

[23] **Randolph N., Gardner D.** Professional Visual Studio 2008, 2008, ISBN: 978-0-470-22988-0, p. 890-900

[24] **Fletcher R., Sajeev A. S. M. A.** Framework for Testing Object Oriented Software Using Formal Specifications: proceedings of the International Conference on Reliable Software Technologies: Ada-Europe, Lecture Notes in Computer Science, 1996

8 TERMINŲ IR SANTRUMPŲ ŽODYNAS

SDK – Programinės įrangos kūrimo reikmenys (Software development kit)

SOA – Į servisus orientuota architektūra (Service-oriented architecture)

COM – Komponento objektinis modelis (Component object model)

SQL – Struktūrinė užklausų kalba (Structured Query Language)

SOAP – Paprastas duomenų prieigos protokolas (Simple Object Access Protocol)

UML – Vieninga modeliavimo kalba (Unified modeling language)

OCL – Objekto apribojimų kalba (Object Constraint language)

XML – Išplėsta pažymėjimo kalba (Extended Markup Language)

MSIL – Tarpinė .Net išeities tekstų kalba (Microsoft Intermediate Language)

CLR – Bendra Microsoft .Net kalbų vykdymo aplinka (Common Language Runtime)

JVM – Virtuali java mašina (Java Virtual Machine)

WSDL – Web-serviso aprašo kalba (Web Service Definition Language)

UDDI – Registro, informacijai apie servisus kaupti, aprašas (Universal Description, Discovery and Integration)

API – Programų kūrimo sąsaja (Application programming interface)

GUI – Grafinė vartotojo sąsaja (Graphical user interface)