KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

PROGRAMŲ INŽINERIJOS KATEDRA

Ramūnas Gečiauskas

Programinio kodo statinės analizės taisyklių kūrimas ir tyrimas

Magistro darbas

Darbo vadovas

prof. dr. E. Bareiša

Kaunas, 2010

KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

PROGRAMŲ INŽINERIJOS KATEDRA

Ramūnas Gečiauskas

Programinio kodo statinės analizės taisyklių kūrimas ir tyrimas

Magistro darbas

Recenzentas:	Vadovas:
prof. dr. L. Nemuraitė	prof. dr. E. Bareiša
2010-05-26	2010-05-26
	Atliko:
	IFM-4/2 gr. studentas
	Ramūnas Gečiauskas
	2010-05-26

Kaunas, 2010

Turinys

1. Įvadas ... 6
1.1. Dokumento paskirtis ... 6
1.2. Santrauka ... 6
2. Analitinė dalis ... 7
2.1. Programinio kodo statinės analizės taisyklių kūrimo analizė ... 7
2.2. „SourceHQ“ įrankio analitinė dalis ... 7
2.3. Egzistuojančių sprendimų ir analogų analizė ... 9
2.3.1. CVS (Concurrent Versions System) sprendimas ... 10
2.3.2. Subversion sprendimas ... 10
2.3.3. GNU Arch sprendimas ... 11
2.3.4. Kiti sprendimai ... 11
2.4. FxCop įrankis ir egzistuojančios taisyklės ... 12
2.5. Pasirinktų taisyklių analizė ... 15
2.5.1. Tuščių eilučių taisyklė ... 16
2.5.2. Neteisingo eilučių palyginimo taisyklė ... 16
2.5.3. Eilučių sujungimo taisyklė ... 16
2.6. Pasiūlytų taisyklių panaudojimo analizė ... 17
3. Projektinė dalis ... 20
3.1. „SourceHQ“ architektūros pateikimas ... 20
3.2. „SourceHQ“ statinio sistemos vaizdo apžvalga ... 21
3.3. Kokybės apžvalga ... 23
3.4. Užsibrėžtų taisyklių projektavimas ... 24
4. Tyrimo dalis ... 26
4.1. Kokybės analizė ir įvertinimas ... 26

4.2. Statinė kodo analizė ... 27
4.3. Papildomų taisyklių realizacija ... 28
4.3.1. Tuščių eilučių taisyklė ... 29
4.3.2. Neteisingo palyginimo taisyklė ... 31
4.3.3. Eilučių sujungimo taisyklė ... 33
5. Eksperimentinė dalis ... 36
5.1. Bandymai su specialiu programiniu kodu ... 36
5.2. Bandymai su „SourceHQ“ programos paketu ... 40
5.3. Bandymai su kitais programų paketais ... 42
6. Išvados ... 44
7. Literatūra ... 45
8. Terminų ir santrumpų žodynas ... 47

DESIGN AND ANALYSIS OF CUSTOM STATIC CODE ANALYSIS RULES

SUMMARY

This final master's thesis consists of three major parts. The first section covers engineering aspects of source code management system called "SourceHQ" that we developed, including its analysis and design details. We will provide key details and basis of chosen technologies, business analysis, and design decisions as well as discuss system functionality and its future prospects.

The second part is dedicated to testing and ensuring system quality, which led us to design and develop custom static source code analysis rules. We will formulate and explain their potential use and benefits. We will describe additional tools and methods being used and provide main results of static source code analysis.

In the final part of our work we go deeper into static source code analysis. We perform experiments based on our designed and developed custom rules on various .NET Framework applications and systems. We cover major performance benefits and drawbacks of every rule separately and display how this approach lets developers optimize their source code in an early development stage. We provide research data based on extensive experiments and conclude how using FxCop tool provided with our improved custom code analysis rules can automatically discover and suggest improvements in CIL code.

1. ĮVADAS

1.1. Dokumento paskirtis

Šis dokumentas su visais esančiais priedais yra programų sistemos inžinerijos magistro baigiamasis darbas. Dokumente yra aprašytos programinio kodo statinės analizės taisyklių kūrimo metodikos, jų analizė, realizavimo būdai bei eksperimentiniai testavimo rezultatai. Taip pat detalizavome baigiamojo magistro darbo metu sukurto produkto „SourceHQ“ architektūrinius sprendimus, kurie buvo priimti projektuojant sistemą. Aprašėme programos funkcijas, galimybes ir perspektyvas.

Dokumente pateikiama kaip programinio kodo statinės analizės taisyklės yra taikomos kuriamai programų sistemai, jų pagrindiniai privalumai ir trūkumai. Tam tikslui aprašėme pagalbinius įrankius, tokius kaip „FxCop“ ir jų panaudojimą. Ištyrėme vyraujančią esamą rinkos padėtį ir galimas ateities tendencijas.

Pasiūlėme ir dokumente detaliau aprašėme bei buvo nagrinėjamos tris programinio kodo rašymo taisyklės, jų kokybės tyrimas ir panaudojimas. Eksperimentiškai pagrindėme kaip patobulinti ir sukurti mūsų pasiūlyti sprendimai gali padidinti daugelio „.NET Framework“ technologija parašytų programų našumą.

1.2. Santrauka

Šiame dokumente aprašytas darbas susideda iš trijų dalių. Pirmoje dalyje atlikome inžinerinę programinio kodo valdymo sistemos „SourceHQ“ analizę ir projektavimą. Palyginome rinkoje esančius analogus, jų privalumus ir trūkumus. Glaustai pateikėme architektūrą ir aprašėme pasirinktus realizavimo sprendimus.

Dokumento antroje dalyje ištyrėme jau egzistuojančias statinės kodo analizė taisykles ir pasiūlėme jas išplėsti naujomis. Aprašėme analizei naudojamus įrankius, jų veikimo principus ir pateikiamus rezultatus.

Paskutinėje darbo dalyje ištyrėme ir eksperimentiškai išbandėme naujai realizuotas mūsų pasiūlytas taisykles. Palyginome testų rezultatus programų veikimo našumo charakteristikos aspektais. Įrodėme mūsų programinio kodo statinės analizės taisyklės privalumus.

2. ANALITINĖ DALIS

2.1. Programinio kodo statinės analizės taisyklių kūrimo analizė

Pagrindinis baigiamojo magistrinio darbo uždavinys susideda iš dviejų dalių: a) sukurti failų versijų kontrolės programinės įrangos paketą „SourceHQ“; b) sukurti naujas programinio kodo parašyto „.NET Framework“ platformoje statines taisykles. Šių taisyklių tikslas - padidinti programinės įrangos našumą dviem pagrindiniais aspektais: sumažinti procesoriaus apkrovimą ir/arba sumažinti reikalingos atminties (RAM) kiekį. Taisyklės privalo būti bendros, t. y. pritaikomos bet kokiai programų sistemai besiremiančiai CIL („Common Intermediate Language“) kodu bei eksperimentiškai pagrįstos.

Šių taisyklių realizavimui naudojome vieną populiariausių, nemokamą ir visiems prieinamą CIL kodo analizės įrankį FxCop. Tai Microsoft kompanijos programa, lengvai integruojama į populiarius programų sistemų kūrimo paketus, tokius kaip „Visual Studio“, tikrinanti parašyto programinio kodo suderinamumą su kompanijos siūlomais standartais ir gairėmis (pagal Microsoft‘s .NET „Framework Design Guidelines“ gidą [3]). Skirtingai nei daugelis kitų įrankių, FxCop analizuoja jau sukompiliuotą objektinį kodą, o ne parašytą išeities kodo tekstą. Tai leidžia realizuotas taisykles taikyti ne tik vienai konkrečiai, bet visoms su .NET platforma suderinamoms programavimo kalboms [1].

2.2. „SourceHQ“ įrankio analitinė dalis

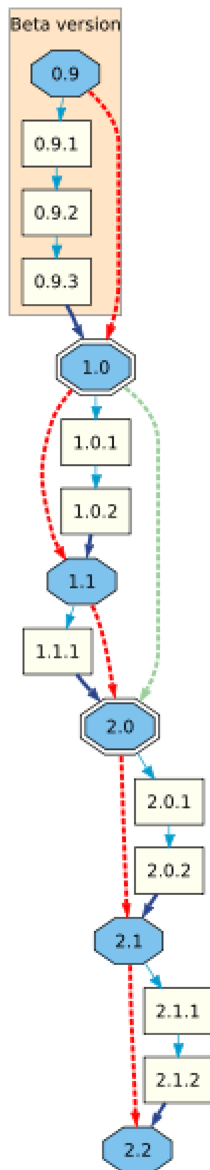
Realizuotų trijų atskirų programinio kodo statinės analizės taisyklių demonstravimui ir detalesniam nagrinėjimui sukūrėme „SourceHQ“ produktą. Ši programinė įranga yra skirta nedidelės-vidutinės apimties pirminio programinio kodo versijų valdymui, statistinei analizei bei atsarginių kopijų saugojimui. Pagrindinis sistemos vartotojas – programinio kodo autorius, programuotojas.

Rašant programinį kodą, tenka atlikti tam tikrus jo pataisymus ar patobulinimus kurie gali būti dvejopi ir tarpusavyje visiškai nesuderinami. Įrankis „SourceHQ“ ir suteikia pagalbinę galimybę programuotojams:

- nesunkiai išskaidyti savo programinį kodą į kelias skirtingas versijas bei tęsti programavimo etapus su kiekvieną iš jų;
- modifikuoti skirtingas programinio kodo versijų atšakas pagal atskirus scenarijus;

- patikrinti jo skirtumus ir pasikeitimus nuo kitų versijų;
- atlikti statistinę kodo analizę;
- sujungti anksčiau išsiskyrusias programinio kodo versijas;

Bet kuri pilnavertiška programinio kodo versijų kontrolės sistema privalo turėti kodo išskaidymo ir išsišakojimo galimybes. Tai leidžia programuotojams, taupant laiką ir kaštus, vieną didelį projektą išskaidyti į atskiras mažesnes dalis bei tęsti kūrimo etapus su kiekviena iš jų atskirai.



1 pav.
Versijų išsišakojimas

Pirminio programinio kodo išskaidymas, taip pat gali būti naudojamas:

- kuriant sistemą skirtingoms kompiuterių platformoms;
- išleidžiant bandomąsias programos versijas;
- taikant skirtingas sistemos realizacijas;
- testuojant sistemą su skirtingomis jos realizacijomis;

Viena svarbiausių programinio kodo versijų kontrolės sistemos funkcijų – galimybė patikrinti ir išsaugoti teksto skirtumus kodo keičiantis.

Didelės apimties produktai dažnai yra išleidžiami tik po kelėtos ar net kelių šimtų pradinių versijų. Perėjus prie aukštesnės programos versijos yra būtina išsaugoti visų buvusių pirminio programinio kodo versijų tekstus, tačiau tiesioginis saugojimo būdas ne visada yra parankus (tačiau kartais neišvengiamas). Todėl, naudojant „SourceHQ“, sukūrus naują programinio kodo versiją yra išsaugomi tik pasikeitusios teksto dalys naudojant Delta kompresijos algoritmą, o nepasikeitusios teksto dalys yra sujungiamos su naujos versijos išleidimu. Visomis šiomis funkcijomis ir pasižymi sukurtas „SourceHQ“ produktas.

„SourceHQ“ buvo kuriamas atsižvelgiant į Microsoft kompanijos siūlomus standartus ir principus. Viena iš šių metodikų yra versijų numeracijos notacija. Microsoft .NET pasiūlė tokį versijų vadinimo ir identifikavimo šabloną:

<Major>.<Minor>.<Revision>.<Build>

- Major – Atitinka pagrindinę leidimo versiją, kuri kinta tik iš esmės pakeitus sistemos branduolį ar architektūrą;
- Minor – Viešam publikavimui ir užbaigtam produktui įvardinti skirtas numeris;
- Revision – Bandomosios versijos (kandidato į viešą išleidimą) numeris;
- Build – Augantysis (angl. Incremental) projekto versijos numeris, kuris didėja su kiekvienu pilnu sistemos paruošimu ar kompiliavimu.

Šį modelį sėkmingai naudoja ir dauguma kitų projektų. Atviro kodo sistemose išlieka populiarus „Kernel“ tipo versijų šablonas, turintis tokią pačią struktūrą kaip ir Microsoft .MET, tačiau jį išplečiant ir <Minor> skaičiuje atspindint versijos tipą: stabili (lyginis skaičius) sistema ar testuojama (nelyginis skaičius) ir pilnai neišbandyta [4].

Daugelis PKVS sistemų yra suderinamos su .NET pasiūlytu šablonu. <Revision>/<Build> skaičius su kiekvienu išleidimu vietoje augančio, pakeičiant į kompiliavimo datą ir laiką. Taip pat prie versijos yra pridedamos žymės nurodančios išleidimo pobūdį: alpha, beta, rc, ctp, rtm [5].

2.3. Egzistuojančių sprendimų ir analogų analizė

Peržvelgiant rinką paviršutiniškai, galima susidaryti įspūdį, kad rinka yra perpildyta panašiomis ir labai galingomis programinio kodo versijų kontrolės ir valdymo sistemomis, tokiomis kaip:

- Revision Control System – Walter F. Tichy ir Paul Eggert sukurta sistema skirta Linux ir UNIX operacinėms sistemoms;
- Project Revision Control System – supaprastintas RCS sistemos analogas, neturintis grafinės vartotojo sąsajos ir skirtas tik Linux OS;
- Aegis – transakcijomis paremta programinio kodo kontrolės sistema;
- CVS – viena seniausių ir labiausiai paplitusių, tačiau tiesiogiai jau nebeatnaujinamų kodo valdymo sistemų;

Nors rinka yra užimta bendro pobūdžio kodo valdymo sistemomis (PKVS), tačiau jaučiamas specializuotų (ir kartu – viešai prieinamų, nemokamų) programinio kodo valdymo

sistemų trūkumas. Apibendrinant išanalizuotas PKVS, galime apibrėžti pagrindinius požymius kuriomis pasižymi daugelis iš jų:

- Skirta Linux/UNIX operacinėms sistemoms;
- Neturi grafinės vartotojo sąsajos;
- Labai didelės ir galingos sistemos, dažnai reikalaujančios didelių sistemos resursų;
- Sudėtingas vartotojo apmokymas;
- Skirtos koordinuoti dideliam programuotojų skaičiui;

2.3.1. CVS (Concurrent Versions System) sprendimas

Viena seniausių PKVS sistemų, kurios kūrimas prasidėjo dar 1980 metais. Programa yra sukurta C programavimo kalba bei yra pritaikyta UNIX ir Windows operacinėmis sistemoms. Dėl GNU GPL licencijos, tai labai paplitusi ir dažnai modifikuojama PKVS.

CVS naudoja klientas-serveris architektūrinį modelį. Einamoji bei visos ankstesnės kodo versijos yra saugomas serveryje. Kiekvienas į sistemą įkeliamas failas yra registruojamas jam priskiriant versijos numerį, o visi to failo pasikeitimai yra saugomi atskirame „loginfo“ faile [7].

Pagrindiniai CVS apribojimai ir trūkumai [2]:

- CVS neindeksuoja failų perkėlimo ar pervadinimo;
- Minimalus suderinamumas su Unicode koduotės failais;
- Veiksmai neatominiai, todėl kodo modifikavimo metu įvykus sistemos sutrikimui, kodas taip pat gali būti pažeistas ir neteisingai išsaugotas;
- Sudėtingas projektų perkėlimas į aukštesnes versijas (Minor->Major);
- Pilnai suderinama tik su tekstiniais failais;
- Nėra suderinamumo paskirstytai kodo kontrolei (angl. distributed revision control);

2.3.2. Subversion sprendimas

Iš CVS kilusi, nemokama (pagal „Apache“ licenciją leidžiama) CollabNet kompanijos kuriama ir palaikoma sistema. Subversion programą naudoja daugelis atviro kodo sistemų tokių kaip: Apache Software Foundation, KDE, GNOME, Free Pascal, FreeBSD, GCC, Python, Django, Ruby, Mono ir SourceForge.net [11].

Didžiausi Subversion privalumai:

- Atlieka atomines operacijas. Išvengiama klaidų atsiradimo tikimybė esant sistemos trukdžiams;
- Failų pervadinimas, kopijavimas ir perkėlimas yra pilnai indeksuojamas;
- Suderinama su netekstiniais failais;
- Naudojami WebDAV/DeltaV protokolai duomenų mainams;
- Galimybė užrakinti nuo modifikavimo įvairius failus;
- Programinio kodo skirtumų analizės rezultatai gali būti saugomi tiek DIFF tiek XML formatu;

2.3.3. GNU Arch sprendimas

GNU Arch yra labai įdomus konkurentas dirbantis visiškai kitokiais būdas nei CVS ar Subversion. GNU Arch yra išleista pagal GNU GPL licenciją, tai visiškai decentralizuota PKVS, kuri yra skirta paskirstytų sistemų kūrimui (pavyzdžiui, Linux branduolio vystymosi procesas). Sistema neturi vieno bendro pagrindinio serverio kuriame saugomi visa informacija, todėl bet koks saugus FTP priėjimas ar bendrai pasiekiamas katalogas gali tapti informacijos saugykla.

Pagrindinis ir bene didžiausias GNU Arch trūkumas – prastas suderinamumas ir lėtas darbas Windows platformose. Ši PKVS sistema taip pat išsiskiria failams priskiriamų vardų unikalumu, kurie neretai iššaukia nesuderinamumus su įvairiomis programomis (vi, diff) [7].

2.3.4. Kiti sprendimai

Viena didžiausių problemų su kuriomis susiduria projekto kūrėjai renkantis optimaliausią PKVS savo projektui – suderinamumas ir pritaikymo sunkumai. Didžioji dalis bendrųjų (ir ypač atviro kodo, nekomercinių) sistemų teikia tik paviršutines failų ar pirminio programos teksto revizijų funkcijas. Todėl daugelis didelių projektų remiasi specializuotais įrankiais, kurie yra sukurti tik realizuojamam produktui. Šios programinio kodo valdymo sistemos dažniausiai būna uždaros bei pasižymi tik konkrečiam produktui skirtu funkcionalumu [6].

Kaip teigia Linux įkūrėjas Linus Torvalds – sudėtingiems projektams, tokiems kaip Linux branduolys, neefektyvu taikyti plataus panaudojimo PKVS tokias kaip CVS ar BitKeeper. Vienintelė alternatyva – sukurti unikalią programinio kodo valdymo sistemą konkrečiam

dideliam projektui arba naudoti modifikuotas, lengvai pritaikomas ir svarbiausia – išplečiamas pagal vartotojo ir projekto poreikius sistemas [7].

Centralizuotos ir decentralizuotos sistemos

Centralizuota PKVS dažniausiai yra paremta serveris-klientas architektūriniu sprendimu. Visi projekto resursai yra saugomi viename serveryje. Leidimus turintys vartotojai, prisijungę prie sistemos turi galimybę patalpinti, pasiimti ar peržiūrėti informaciją susijusią su projekto failais.

Decentralizuotos (paskirstytos) sistemos - tai ganėtinai nauja PKVS atmaina, pagrindiniai jų skirtumai lyginant su centralizuotomis sistemomis yra [8]:

- Kiekvienas programuotojas turi savo atskirą saugyklą;
- Bet kuris vartotojas gali skaidyti kodą į atskiras šakas;
- Centrinių (pagrindinių) saugyklų gali būti ne viena;
- Nereikalauja tiesioginio susijungimo su centriniu serveriu, todėl galima dirbti autonominėse, neturinčiose interneto ar intraneto prieigų kompiuteriuose;
- Leidžia individualų darbą, todėl ne visos kodo versijos privalo būti publikuojamos viešai;
- PKVS išskaidymas į daugelį fizinių kompiuterių padidina projekto saugumą, nes vieno fizinio kompiuterio sutrikimas neturi įtakos bendram projekto kūrimui;

Tačiau, nepaisant akivaizdžių decentralizuotų sistemų privalumų, daugelis projektų reikalauja centralizuotų sprendimų ir valdymo [9].

„SourceHQ“ įrankis yra decentralizuotas, nemokamas, optimizuotas ir skirtas vieno konkretaus programuotojo darbo sekimui ir palengvinimui, kas ir padaro jį unikaliu su rimta perspektyvą rinkoje.

2.4. FxCop įrankis ir egzistuojančios taisyklės

FxCop yra nemokamas statinio kodo analizės įrankis nuo 2002 metų leidžiamas Microsoft kompanijos. Įrankis gali būti naudojamas kaip atskira programa arba kaip Visual Studio

Teamsystem paketo dalis. Skirtingai nei daugelis kitų programų (pavyzdžiui „lint“ įrankis C programavimo kalbai), FxCop analizuoja jau sukompiliuotą dvejetainį objektinį kodą CIL (Common Intermediate Language) sugeneruotą .NET kompiliatorių. Jo veikimas yra pagrįstas multigrafo algoritmu, apeinant ir padengiant visas programinio kodo dalis ir jį galima naudoti tiek vykdančiųjų (EXE) failų analizei, tiek ir bibliotekoms (DLL). Nors įrankio paskirtis yra pateikti programinės įrangos .NET platformoje kūrėjams karkasą ir paruoštas bibliotekas statinio kodo analizei, tačiau prie standartinio paketo yra pateikiamos „Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries“ ir „.NET Design Guidelines for Class Library Developers“ gidais paremtos jau paruoštos taisyklės tikrinančios kodo korektiškumą, kintamųjų vardų atitikimą rekomenduojamom notacijom, vykdymo charakteristikų, saugumo, lokalizacijos ir kitais aspektais.

Viena svarbiausių FxCop savybių, pateikiamas „FxCop SDK“ dviejų bibliotekų rinkinys leidžiantis išplėsti ir sukurti papildomas taisykles tokias kaip:

- Užtikrinti objektų vardų atitikimą norimom notacijom;
- Patikrinti ar pageidautini komponentai ir klasės yra naudojami vietoje alternatyvių;
- Tikrinti kode deklaruotų eilučių turinį;
- Nagrinėti įvairias kodo struktūras, tokias kaip sąlyginiai sakiniai ir ciklai, įvertinant programinio kodo metrikas;
- Nustatyti metodų vykdymo eiliškumą;
- Gramatiškai patikrinti įvairius teksto elementus bei resursus;
- Užtikrinti teisingą kodo elementų XML dokumentavimą;
- Pagerinti kodo vykdymo greičio ir/arba resursų suvartojimo metrikas atsižvelgiant į pasirinktų algoritmų įgyvendinimą;

Mūsų keliami uždaviniai apima būtent paskutinio punkto realizavimą sukuriant konkrečias taisykles.

Microsoft pateikiamas „.NET Design Guidelines for Class Library Developers“ gidas aprašo pagrindines gaires programinio kodo rašymui, tačiau tik dalis jų yra realizuota statinei kodo analizei atlikti. FxCop įrankis (1.36 versija) pateikia 174 paruoštas statinio kodo analizės taisykles kurios yra suskirstytos į 9 atskirus tipus. Trumpai apžvelgsime kiekvieną iš šių tipų.

Dizaino/Projektavimo (angl. Design) taisyklės

Didžiausia taisyklių grupė apimanti kodo rašymo stiliaus gaires. Kadangi galutiniam vartotojui taisyklių atitikimas ar neatitikimas dažniausiai įtakos neturi, rastos klaidos šalinamos su mažiausiu prioritetu. Tačiau, laikantis dizaino taisyklėmis apibrėžtais bendriniais nurodymais, programinis kodas tampa universalesnis, griežčiau susietas su objektinio programavimo principais bei lengviau skaitomas. Pateikiamos rekomendacijos nenaudoti tuščių sąsajų, abstraktūs metodai neturi turėti viešai prieinamų konstruktorių, atsisakyti pagal nutylėjimą objektams priskiriamų reikšmių pakartotinio priskyrimo, nedeklaruoti virtualių metodų užbaigtose (angl. sealed) klasėse ir daugelis kitų kodo stiliaus išlaikymo gairių.

Globalizacijos (angl. Globalization) taisyklės

Taisyklių rinkinys padedantis užtikrinti kodo suderinamumą su įvairiomis kalbomis, koduotėmis ir kultūriniais aspektais. Reikalauja nustatyti CultureInfo objektą prieš vykdant bet kokias kultūrai jautrias (skirtingo laiko, datos formatų, pinigų ir t.t.) palyginimo operacijas taip išvengiant nenumatytų klaidų atsiradimo kai kuriamas produktas yra naudojamas skirtingose pasaulio rinkose.

Suderinamumo (angl. Interoperability) taisyklės

Šios taisyklės prižiūri tinkamą COM objektų panaudojimą: statinių metodų nenaudojimą, tam tikrų Win32 funkcijų korektišką iškvietimą, papildomus saugumo lygius vykdant tiesiogines operacijas su atmintimi.

Mobilumo ir portatyvumo (angl. Mobility, Portability) taisyklės

Kadangi .NET platforma yra pritaikyta ne tik personalinių kompiuterių tačiau ir įvairių mobilių įrenginių rinkoms, šios taisyklė ir yra aktualios rašant programinį kodą tokioms sistemoms. Atsižvelgiama į programų energijos suvartojimo optimizavimą (draudžiant naudoti tuščius ciklus ir laukimo būsenas (angl. Idle State), sumažiną kreipinių į procesorių skaičių, užtikrina vienodą suderinamumą su 32 ir 64 bitų sistemomis.

Vardų (angl. Naming) taisyklės

Kaip ir dizaino taisyklės, vardų taisyklių rinkinio užduotis - padidinti programinio kodo skaitomumo lygį. Tikrina, kad nebūtų naudojami tam tikri rezerviniai žodžiai kintamųjų vardams,

C# vėliavėlių tipo atributai privalo būti pavadinti daugiskaitos forma, taikoma vengriška notacija (pirmas kintamojo vardo simbolis nurodo jo tipą), užtikrina, kad paveldėti parametrai vadintųsi tais pačiais vardais kaip ir baziniame objekte.

Saugumo (angl. Security) taisyklės

Šių taisyklės padeda užkirsti kelią galimų saugumo klaidų atsiradimui. Draudžia masyvo deklaravimą tik skaitymo (angl. read only) režimu, tiesioginės rodyklės į atminties blokus klasėse turėtų būti deklaruotos kaip privačios ir nematomos išoriškai, taip pat kaip ir statiniai konstruktoriai bei kitos dažniausiai pasitaikančių saugumo klaidų išvengimo taisyklės.

Panaudojimo (angl. Usage) taisyklės

Taisyklių rinkinys padedantis užtikrinantis kuriamo produkto didesnį panaudojimo lygi. Aprašomos tokios taisyklės kaip senų ir nerekomenduotinų Win32 API funkcijų pakeitimą kitomis, nenaudojamų parametrų ir atributų pašalinimą, teisingą ir konkretų kintamųjų tipo parinkimą, griežtą „serializable“ laukų deklaravimą, laikinieji objektai neįtrauktini į .NET šiukšlių surinkimo mechanizmą (GC) privalo turėti jų pačių sunaikinimo metodą.

Greitaveikos (angl. Performance) taisyklės

Taisyklės skirtos programinio kodo optimizavimui, pagerinant jo veikimo našumo charakteristikas. Rekomenduoja naudoti greičiau veikiančius metodus vietoje jų alternatyvų, įspėja apie nenaudojamą ir nepasiekiamą kodą, siūlo naudoti C# “jagged” tipo masyvus vietoje multi-dimensinių.

2.5. Pasirinktų taisyklių analizė

Išanalizavus baigiamojo magistrinio darbo metu mūsų kuriamą produktą, pagrindinis jam keliamas nefunkcinis reikalavimas yra vykdymo greitis ir naudojami resursai. Pagrindinis uždavinys tapo ne tik produkto funkcionalumas, bet ir jo kūrimo metodai. Tam pasitelkėme ir realizavome tris atskiras programinio kodo statinės analizės taisykles, kurios plačiau ir detaliau yra aprašytos šio dokumento tyrimo dalyje.

2.5.1. Tuščių eilučių taisyklė

Siūlomos programinio kodo rašymo taisyklės principas yra paprastas - kiekvieną kartą deklaruojant „string“ tipo tuščią eilutę, vietoje standartinio konstruktoriaus naudoti „String.Empty“ identifikatorių.

Nors MSDN standartai rekomenduoja būtent tokį būdą, tačiau jis nėra griežtai apibrėžtas ir nėra laikoma kaip standartų neatitikimo klaida [12]. Mūsų siūlomas sprendimas leidžia sutaupyti tiek tiesioginį (kodas yra vykdomas greičiau), tiek netiesioginį (eliminuojamas uždelsto „šiukšlių surinkimo“ (angl. Garbage Collector) kvietimas) procesoriaus laiką bei sumažina sunaudojamos atminties kiekį.

2.5.2. Neteisingo eilučių palyginimo taisyklė

Taisyklė užtikrina, kad dviejų „string“ tipo eilučių palyginimui, vietoje įprastų lygybių ir nelygybių operatorių, būtų griežtai kviečiama „String.Compare“ funkcija. Be tyrimo dalyje detalizuotai aprašytos programos našumo naudos, ši taisyklė iš dalies padidina ir kuriamos programinės įrangos saugumą:

Dėl skirtingų teksto koduočių, saugojimo formatų bei .NET Framework galimybių naudoti visiškai autonomiškai ir atskirai parašytus kodo fragmentus juos sujungiant, net ir paprasta tekstinių eilučių palyginimo operacija gali sukelti rimtų, sunkiai pastebimų, problemų. Bandant programiniu būdu nustatyti ar dvi teksto eilutės kurios vartotojui iš pirmo žvilgsnio gali pasirodyti identiškos, tačiau išsaugotos skirtingomis koduotėmis, galimi sukelti nenumatytą situaciją ir programos klaidą. Ši taisyklė užkerta tam kelią, kadangi „String.Compare“ funkcija griežtai reikalauja nurodyti koduotę pagal kuria bus lyginamos eilutės (arba naudoti iš anksto nustatytą pagal nutylėjimą).

2.5.3. Eilučių sujungimo taisyklė

Šios siūlomos taisyklės esmė - užtikrinti, kad visos eilučių sujungimo (konkatenacijos) operacijos būtų atliekamos griežtai naudojant tik .NET Framework bibliotekų pateikiamą „StringBuilder“ klasę, vietoje įprastų programinių operatorių.

Šios taisyklės galimas paplitimas ir pritaikymas yra pats mažiausias lyginant su kitomis siūlomomis taisyklėmis, tačiau ji teikia didžiausią programos našumo padidėjimą ir naudą. Visi

pagrindiniai .NET Framework programavimo standartai bei rekomenduojamos gairės, įskaitant ir MSDN dokumentaciją, rekomenduoja būtent šį metodą. Nors taisyklės nauda akivaizdi, jos panaudojimas ir paplitimas tarp programuotojų nėra didelis dėl vienos priežasties - programinis kodas tampa ilgesnis bei sudėtingesnis skaitymui. Tačiau, kaip rodo rezultatai ir tyrimo išvados pateikiamos šio dokumento tyrimo dalyje, net ir ilgesnis ir komplikuotas kodas yra vykdomas greičiau kompiuterio resursų atžvilgiu.

2.6. Pasiūlytų taisyklių panaudojimo analizė

Pasiūlytos ir sukurtos taisyklės nėra tiesiog gairės programuotojams. Jos yra griežtai struktūrizuotos, apibrėžtos ir jų laikymasis programuotojams, tose kompanijose kur jos yra naudojamos, yra privalomas. Šios taisyklė yra pilnai dokumentuotos, su pasiūlytais alternatyviais sprendimais bei realizuotos sukompiliuotuose DLL failuose suderinamuose su FxCop įrankiu. Tai leidžia bet kokį .NET platformos pagrindu parašytą programinį kodą statiškai išanalizuoti tiek autonomiškai naudojant savarankišku FxCop įrankiu įkėlus taisyklių DLL failą, tiek integruotais FxCop paremtais metodais Microsoft „Visual Studio“ aplinkose (tik „TeamSystem“ versijose).

Egzistuoja dvi pagrindinės šių taisyklių panaudojimo problemos:

1. FxCop įrankio apribojimai;
2. Orakulo problema;

FxCop - Microsoft uždaro kodo, tačiau visiškai nemokamas ir visiems laisvai prieinamas produktas. Jis programuotojams ir taisyklių autoriams pateikia bibliotekų ir su jomis susietų resursų bei struktūrizuotų funkcijų rinkinį. Naudojantis šiais metodais programuotojas gali įvairiapusiškai išanalizuoti bet kurios programos ar jos fragmento sukompiliuotą CIL kodą. Tai tarpinis, architektūriniu požiūriu panašus į „Sun Microsystems“ sukurtą ir populiarų Java „baitkodą“, pilnas specializuotai paruoštas programos aprašymas kuris yra užkraunamas į .NET Framework naudojamą realaus laiko kompiliatorių, verčiamas į funkcines mašinines operacijas, startuojant programai. Bene didžiausias CIL privalumas - unifikuotas kodas. Bet kokia .NET palaikoma programavimo kalba parašytas pirminis programinis kodas bus identiškai atvaizduotas CIL kodu. Taigi, mūsų sukurtos taisyklės, kurios ir yra taikomos CIL kodui analizuoti, tinka plačiam spektrui įvairių programavimo kalbų. Tačiau, nors Microsoft FxCop kūrėjai ir pateikė

didžiulį metodų rinkinį, jis vis vien yra baigtinis ir visapusiškai išanalizuoti CIL kodą yra neįmanoma [14].

Antras iššūkis - Orakulo problema. Kad ir kaip efektyviai būtų parašytos programinio kodo rašymo taisyklės yra labai sunku, o kartais ir neįmanoma, numatyti ar parašytas kodo fragmentas yra klaida ar sprendimas kurio ir norėjo programuotojas. Ši problema ypač išryškėja analizuojant konkrečias programavimo kalbas.

Kaip pavyzdį galima pateikti C# eilučių palyginimo operacijas. Mūsų siūloma eilučių palyginimo taisyklė naudojant „String.Compare“ metodą gali ne visada pateikti korektišką klaidos aptikimą, kadangi CIL kodas yra analizuojamas tik apibrėžtoje aplinkoje. Taisyklės tikslas paprastas - be našumo padidinimo užtikrinti ir didesnį saugumą. Tačiau naudojamas analizatorius negali žinoti apie galbūt naudojamas išorines saugumo užtikrinimo priemones (todėl papildomas saugumas šioje vietoje nėra būtinas) ar naudojamą techninę aparatūrą (galbūt našumo didinimas nėra svarbiausias programuotojo uždavinys). Galimas ir toks scenarijus, kad programuotojas, puikiai suprasdamas C# programavimo kalbos ypatybes, tyčia naudoja įdingus metodus: pavyzdžiui sukeldamas klaidos iššaukimą (ThrowException) ir juos kontroliuojamai apdorojant naudojant „try... catch... finally...“ blokus [21].

Kitas, taip pat labai svarbus, pavyzdys galėtų būti C# programavimo kalboje leidžiami standartinių operatorių perrašymai (angl. override). Tekstinių eilučių palyginimo taisyklės užtikrinimas remiasi standartinių C# operatorių paieška CIL kode ir jų lauko (angl. scope) analize, tačiau nėra nagrinėjama šių operatorių vykdomi veiksmai. Egzistuoja galimybė, kad programuotojas iš anksto yra apsirašęs skirtingus operatorius ir taip pakeitęs jų standartinį veikimą apie kurį FxCop įrankis nėra ir negali būti informuotas. Todėl su statinės analizės įrankiu tikrindamas net ir teisingai parašytą kodą bus gaunamas klaidingas rezultatas informuojantis apie neatitikimą užsibrėžtomis taisyklėmis.

Dėl šių dviejų problemų, atliekant statinę CIL analizę, bet koks neatitikimas taisyklėmis yra traktuojamas kaip „galima klaida“ arba „įspėjimas“ (angl. warning), o ne „kritinė klaida“ (angl. error). Įspėjimas reiškia, kad yra aptiktas konkretus galimas netikslumas nebūtinai yra klaida, o tiesiog reikalauja papildomos programuotojo peržiūros. Taip pat, svarbu paminėti, kad nepriklausomai ar kodas atitinka siūlomas taisykles ar ne, jo funkcionalumas nesikeičia, todėl

kompiliavimas ir programos paleidimas yra leidžiamas net ir esant surastiems galimų klaidų įspėjimams. Dėl to, griežtai žymėti neatitikimus kaip „kritines klaidas“ nėra tikslinga.

Galiausiai našumas tam tikruose programinės įrangos paketuose, ar net tiesiog daliniuose kodo fragmentuose, metoduose ar algoritmuose nėra svarbiausias faktorius. Egzistuoja daugybė scenarijų kur specifinė techninė aparatūra yra svarbus faktorius, o našumo reikalavimai skiriasi nuo keliamų įprastiems kompiuteriams. Vienuose sistemose, operatyvinė atmintis gali būti beveik neribota, o kiekvienas procesoriaus ciklas ypač brangus. Kitose - galbūt procesoriaus apkrovimas neturi didelės įtakos, bet egzistuoja smarkiai limituotas atminties kiekis. Todėl kurias taisykles naudoti ir taikyti bei kaip nuodugniai tai daryti, turi spręsti pats programuotojas.

FxCop įrankis turi galimybę identifikuoti kiekvieną problemą atskirai ir jai priimti atitinkamą sprendimą. Jei po pakartotinės peržiūros yra rasta klaida arba tiesioginis neatitikimas apibrėžtoms taisyklėm, kodas yra taisomas ir vykdoma jo pakartotinė statinė analizė. Jei kodas yra korektiškas, bet dėl vienos iš aukščiau įvardintų problemų yra randama fiktyvi klaida - naudojant SuppressMessage() metodą, jis yra identifikuojamas kaip korektiškas ir jo analizė su konkrečia nurodyta taisykle nebus atliekama.

3. PROJEKTINĖ DALIS

3.1. „SourceHQ“ architektūros pateikimas

„SourceHQ“ programinė įranga bei jos dokumentacija buvo sudaryta pagal IBM kompanijos rekomenduojama RUP (Rational Unified Process) šabloną ir procesą skirtą architektūros specifikavimui. RUP gyvavimo ciklas yra pagrįstas spiralės modeliu kuris puikiai tinka atviro kodo sistemoms, leidžia šį šabloną lengvai pritaikyti konkrečiam kuriamam projektui bei užtikrina jo tolesnį vystymą. Nors tiek sistemos specifikacijas, tiek realizavimą ir kitus etapus inicijavo ir kūrė vienas asmuo, dėl programinės įrangos pobūdžio, pasirinktos licencijos ir viešo prieinamumo, yra tikėtina, kad ateityje „SourceHQ“ gali būti vystoma kitų trečiųjų asmenų. Tuo ir yra grindžiamas RUP modelio pasirinkimas.

RUP šablonu paremta sistemos gyvavimo ciklo spiralė užduotis organizuoja į fazes ir iteracijas. Egzistuoja keturios pagrindinės fazės [16].

- Inspekcijos - šioje fazėje yra apibrėžiama verslo aplinka, reikalavimų supratimas, prioritetų ir rizikos valdymas bei realizacijos gairių kūrimas;
- Tyrimo - šioje fazėje projektas įgauna tikrąją savo formą. Atliekama probleminės srities analizė. Sukuriamas užduočių modelis, viso projekto planas ir prototipai;
- Kūrimo - pagrindinis dėmesys šioje fazėje yra skiriamas programos realizacijai, o fazės galutinis tikslas sukurti pirmąją veikiančią sistemos versiją;
- Perdavimo - šiame etape produktas yra perduodamas galutiniams vartotojams. Vykdomas produkto tikrinimas atitikimui kokybės reikalavimams. Atliekamas beta testavimas, palaikymas ir vartotojų mokymai. Jei neatitinka kokybės lygio, standartų arba galinių vartotojų poreikių, visas ciklas pakartojamas iš naujo;

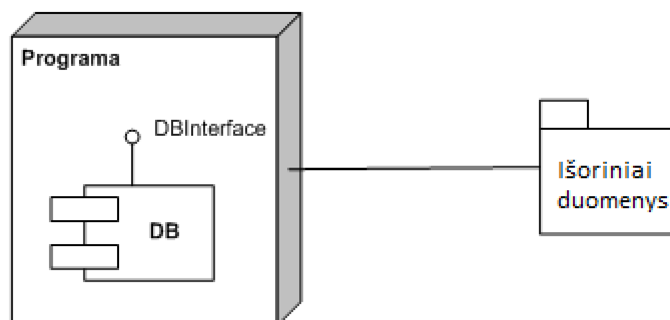
Didžiajai daliai architektūros specifikavimo buvo naudojama UML 2.0 (Unified Modeling Language) notacija ir diagramos. Tai OMG (Object Management Group) bendrijos 1997 metais sukurta modeliavimo kalba, kuri leidžia vieningai ir stabiliai, t.y. vienareikšmiškai suprantamai, apibrėžti mūsų kuriamo produkto architektūrą bei ją aptarti su užsakovais.

Pagrindinėms UML diagramoms, tokioms kaip panaudojimo atvejų, klasių, sekų, būsenos, išdėstymo ir veiklos, buvo naudojami Microsoft „Visio“ ir „Visual Studio TeamSystem“ paketai. Jie leidžia glaudžiai susieti realizavimo ir projektavimo etapus,

palengvinti paskirstytų sistemų architektūros kūrimą ir atnaujinimą, o naudojant paskirstytų sistemų kūrimo dizainerį (paketuose integruotą įrankį), galima išskirstyti kelis aukštos abstrakcijos lygmenis ir darbus: peržiūrėti, specifikuoti, keisti, konfigūruoti ir tarpusavyje jungti programų sistemas; specifikuoti duomenų centrų vaizdą; sistemos išdėstymą; įgyvendinti ir realiai sugeneruoti veikiantį programinį kodą.

3.2. „SourceHQ“ statinio sistemos vaizdo apžvalga

Supaprastintas kuriamos programinės įrangos statinis vaizdas yra pateiktas 2 pav.

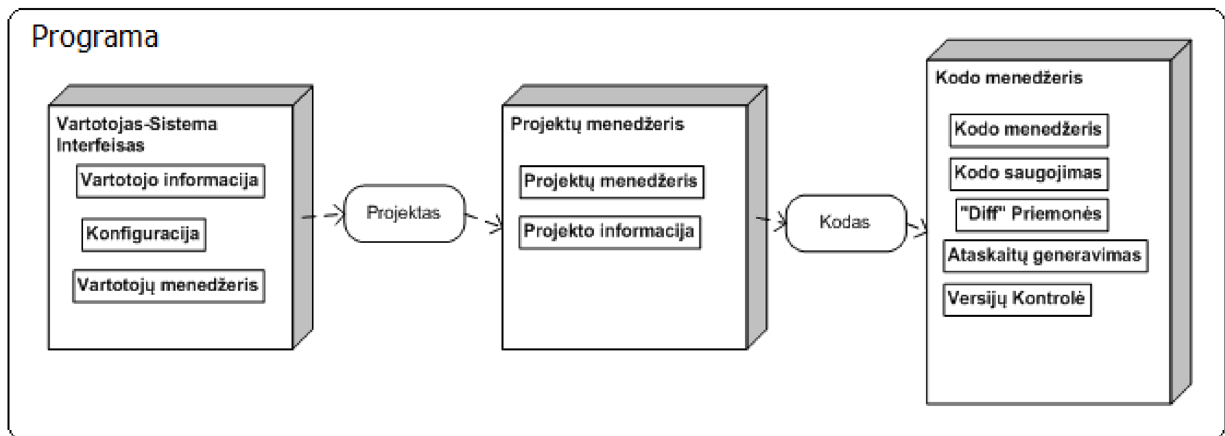


2 pav. Statinis konteksto vaizdas

Aukščiau pateiktame paveikslėlyje pavaizduota programų sistemos apjungimo diagrama aukščiausiame abstrakcijos lygmenyje, išskiriant tik fizines taikomąsias programas, o jų pačių nedetalizuojant. Programa bendradarbiauja su išoriniais duomenimis kurie yra duomenų failai sudaryti iš programinio kodo kurį norima valdyti šia programa. „SourceHQ“, naudojant Windows API, automatiškai seka jų pakeitimus bei juos registruoja. Pakeitimams išsaugoti yra naudojamas dvejetainio palyginimo ir suglaudimo „Delta“ algoritmas, išsaugantis tik pasikeitusias failo dalis ir gebantis iš jų atkurti pradinį originalų failą.

Programos viduje reikia išskirti atskirą duomenų bazės komponentą su „DBInterface“ sąsaja. Šis komponentas yra atsakingas už vidinių duomenų saugojimą duomenų bazėje tačiau nėra griežtai surištas su konkrečia duomenų bazės valdymo sistema ir gali būti pritaikytas MSSQL, MySQL, SQLite ir su daugeliu kitų duomenų bazių servisais.

Sistemos išdėstymo vaizdas yra pateikiamas 3 pav.



3 pav. Sistemos išdėstymo vaizdas

„SourceHQ“ susideda iš 3-jų pagrindinių sisteminių sluoksnių: Vartotojo grafinės sąsajos, projektų ir kodo menedžerio.

Vartotojas-Sistema grafinė sąsaja apima klases kurios valdo vartotojo informaciją, konfigūraciją ir sistemos nustatymus kurie nurodo programos elgseną. Projektų menedžeris valdo sistemoje esančių priregistruotų projektų informaciją ir jų unikalius nustatymus. Kodo menedžeris savyje turi deklaruotas esmines sistemos klases užtikrinančias programos funkcionalumą: Kodo saugojimo struktūra - kurioje yra valdikliai kodo pasikeitimų išsaugojimui išorinėje duomenų bazėje; „Diff“ priemonės - užtikrinančios aktyvų kodo pasikeitimo sekimą; Ataskaitų generavimas - suteikia galimybę generuoti ir išsaugoti įvairią informaciją išorinėse laikmenose; Versijų kontrolė - automatiškai identifikuoja kodo failus, juos kataloguoja bei priskiria unikalius versijos numerius.

Bendri apribojimai.

Kaip ir daugelis kitų [17], kuriama PKVS turi tenkinti tokius reikalavimus:

- Kodo saugojimo resursai

  Rašant didelės apimties programinį kodą, arba jį skaidant į daugelį skirtingų versijų, smarkiai išauga atsarginių kopijų apimtys. Saugant ši programinį kodą tradiciniu būdų – įsimenant visą pilną kodo kopiją, yra švaistomi sistemos resursai, kadangi daugeliu atveju pirminis programinis tekstas tarp skirtingų jo versijų lieka nepakitęs. Kinta tik maži jo fragmentai.

Naudojamas „Delta“ kompresijos algoritmas mažins atminties poreikį, kadangi į atminties saugyklą bus įrašomi tik atlikti kodo pokyčiai ir jo modifikacijos. Toks saugojimo būdas, dideliuose projektuose (vertinant „Linux“ patirtį) gali padėti sutaupyti apie 90% sisteminės atminties poreikių.

- Suderinamumas su IDE aplinkomis

  Pilnai išnaudoti sistemą nebūtina turėti ir naudoti papildomų priemonių ar įrankių, tačiau daugelis programuotojų naudoja savitas programavimo aplinkas. Būtina užtikrinti duomenų suderinamumą tarp šių aplinkų.

- .NET Framework suderinamumas

  Sistema yra kuriama .NET Framework architektūros pagrindu, todėl svarbu užtikrinti, kad ji sklandžiai veiktų visuose operacinėse sistemose suderintose su .NET technologija.

3.3. Kokybės apžvalga

Produkto kokybės ir funkcionalumo įvertinimas bus atliekamas viso sistemos kūrimo proceso bei bandomosios programos versijos eksploatavimo metu. Produkto kokybė bus vertinama pagal šiuos pagrindinius kriterijus:

- Sistemos naudingumas ir tinkamumas naudoti – sukurtoji sistema turės būti naudinga vartotojams bei bus tinkama naudoti kaip alternatyva turimoms PKVS sistemoms.
- Vartotojo sąsaja – sistemos sąsaja su vartotoju turi būti paprasta, nekomplikuota ir intuityvi.
- Klaidų nebuvimas – sistema turi veikti be klaidų arba klaidų rizika turi būti sumažinta iki minimumo.
- Sistemos funkcionalumas – sistemoje turi būti įgyvendintos visos norimos funkcijos.

Pirmiausias ir turbūt vienas svarbiausių architektūros suteikiamų privalumų yra išplėtimo galimybė. Kadangi sistema prisijungimui su Microsoft „Visual Studio“ paketu naudos atvirą sąsają, bus patogu praplėsti sistemą, pritaikant ją įvairių tipų ir platformų vartotojams.

Numatyta galimybė pasinaudoti tokių sistemų kaip „SubVersion“, „Aegis“ ir „RCS“ komponentais bei pateikiamomis specifikacijomis. Taip pat, yra galimybė pasinaudoti įvairiomis

delta kompresijos algoritmo modifikacijomis ir realizuotomis bibliotekomis, kuriant kodo saugojimo posistemę.

Patikimumui ir kodo atstatymui užtikrinti bus nuolat sekamas sistemos darbas, fiksuojami ir protokoluojami įvykiai įvykių registre. Atliekama statinio kodo analizė, kaip parodė įvairūs tyrimai [20] užtikrina didesnį testavimo padengimą.

„SourceHQ“ įrankis turi būti pilnai suderinamas ir integruojamas į Microsoft „Visual Studio“ paketo C# modulį. Integracija vykdoma programos diegimo metu, tačiau turi būti numatyta galimybe šios integracijos atsisakyti ir programą naudoti kaip atskirą nepriklausomą įrankį.

3.4. Užsibrėžtų taisyklių projektavimas

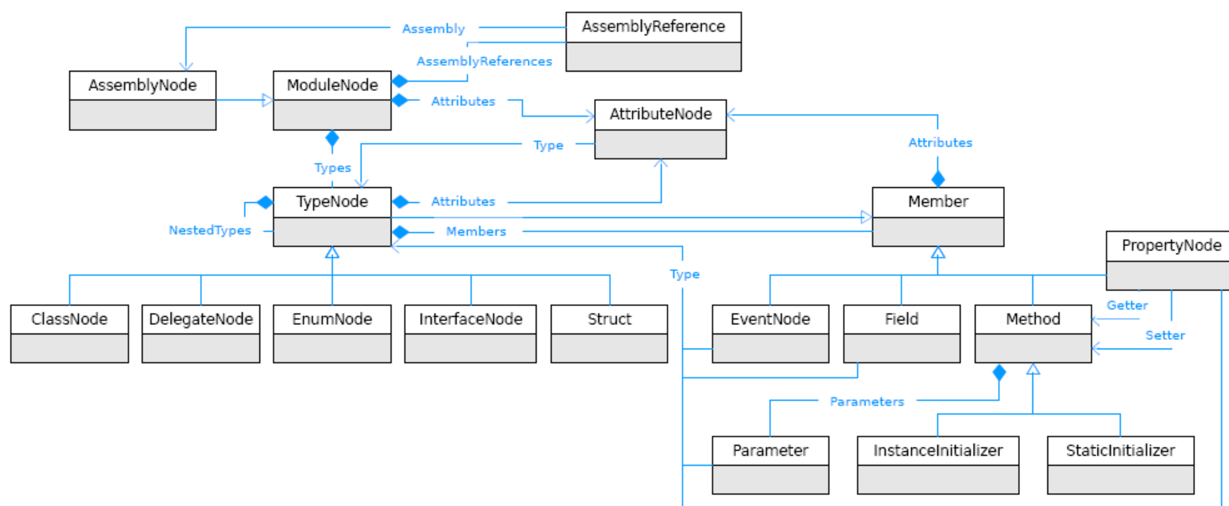
Norint realizuoti naują statinės kodo analizės taisyklę FxCop įrankyje reikia atlikti du pagrindinius veiksmus: aprašyti taisyklės meta duomenis XML formatu (žiūrėti 1 lentelę) bei sukurti taisyklės veikimo logiką DLL bibliotekoje.

1 lentelė. Taisyklės aprašymo XML failo struktūra

```
<?xml version="1.0" encoding="utf-8"?>
<Rules FriendlyName="">
      <Rule TypeName="" Category="" CheckId="">
            <Name>         Taisyklės pavadinimas          </Name>
            <Description>  Taisyklės aprašymas            </Description>
            <Url>          Nuoroda į platesnę informaciją </Url>
            <Resolution>   Sprendimo aprašymas            </Resolution>
            <MessageLevel Certainty="">Klaidos svarbumas  </MessageLevel>
            <Email>        Autoriaus el. paštas           </Email>
            <FixCategories>Klaidos tipas                  </FixCategories>
            <Owner>        Autorius                       </Owner>
      </Rule>
</Rules>
```

Viename XML faile gali būti deklaruotos kelios taisyklės. Šis failas yra sukompiliuojamas kaip resursų dalis DLL bibliotekoje arba gali būti atskirai pateikiamas įvairiems statinės analizės įrankiams. Failas pateikiama glausta informacija apie taisyklę, jos sprendimo būdus, kategoriją, svarbumo lygi (kritinė klaida, klaida, kritinis įspėjimas, įspėjimas), o visa ši informacija yra saugoma ir koduojamas UTF-8 formatu.

Taisyklės loginė ir funkcinė dalis yra kompiliuojama į DLL failą paveldintį ir naudojantį FxCopSdk.dll ir Microsoft.Cci.dll bibliotekas. Skirtingai nei daugelis kitų analizės priemonių kurios naudoja .NET Framework „System.Reflection" įrankius, FxCop naudoja introspektinį (žvelgiantį giliau į kodą, angl. introspection) modelį. Jo pagalba, kodo nebūtina užkrauti į CLR ir startuoti programos (tuo ir pasižymi statinė analizė) [18].



4 pav. Mazgų objektinė priklausomybė

FxCop naudojamas kodo peržiūros mechanizmas išskaido analizuojamą kodą į fundamentalius ir neskaidomus primityvus dar kitaip vadinamus mazgais. Kiekvienas mazgas atstoja .NET platformos klasę introspekcijos API, į kurią kreipiantis mes galime gauti visu norimo objekto atributus ir parametrus. Visi galimi mazgai ir jų sąryšiai yra pavaizduoti 4 pav. Turint prieigą prie šių mazgų, naudodamiesi viena iš .NET palaikomų programavimo kalbų (mūsų atveju - C#), jau galime projektuoti ir galiausiai realizuoti papildomas taisykles. Kadangi mūsų projektuojamos taisyklės turi aptikti tiesioginius operatorius (priskyrimo, palyginimo ir kintamojo deklaravimo), mes analizuosime „Method" mazgą kuris savyje turi „Intructions" parametrą atvaizduojantį visas metode vykdomas operacijas. Mūsų kodo tikrinimo pagal apibrėžtas 3 taisykles galime išskaidyti į atskirus etapus: Kiekvieno CIL kode esančio metodo peržiūra instrukcijų lygmenyje; Raktinių (sužadinančių vieną iš mūsų taisyklių) instrukcijų paieška; Instrukcijos panaudojimo korektiškumas; Rezultato apie metode esančią klaidą grąžinimas. Kiekvienos atskiros taisyklės konkreti analizės logika mūsų darbe buvo atliekama naudojant C# programavimo kalbą ir Microsoft Visual Studio 2008 paketą bei kompiliatorių.

4. TYRIMO DALIS

Tiriamojoje dalyje pateiksime programų sistemos projekto kokybės vertinimą, realizuoto produkto statinės analizės rezultatus bei pasiūlysime, pateiksime bei apibrėšime patobulintas statinės kodo analizės taisykles.

4.1. Kokybės analizė ir įvertinimas

2 lentelė. Bendra kodo saugyklos informacija

Kodo gyvavimo trukmė	28 sav.
Versijų skaičius	41
Failų atnaujinimų skaičius	1107

Projekto paraiška buvo pateikta 2008 m. lapkričio 20d. Projekto planas ir projektavimo etapai pradėti 2009 m. vasarį. Sistemos realizavimas, kodavimo etapas bei aplinkos sukūrimas (SVN ir „Bugzilla" registracija) 2009 m. birželį. Per 28 savaites buvo išleista 41 unikali programos versija (1.0.0.0 - 1.1.2.1), o SVN sistemoje iš viso užregistruoti 1107 failų pakeitimai.

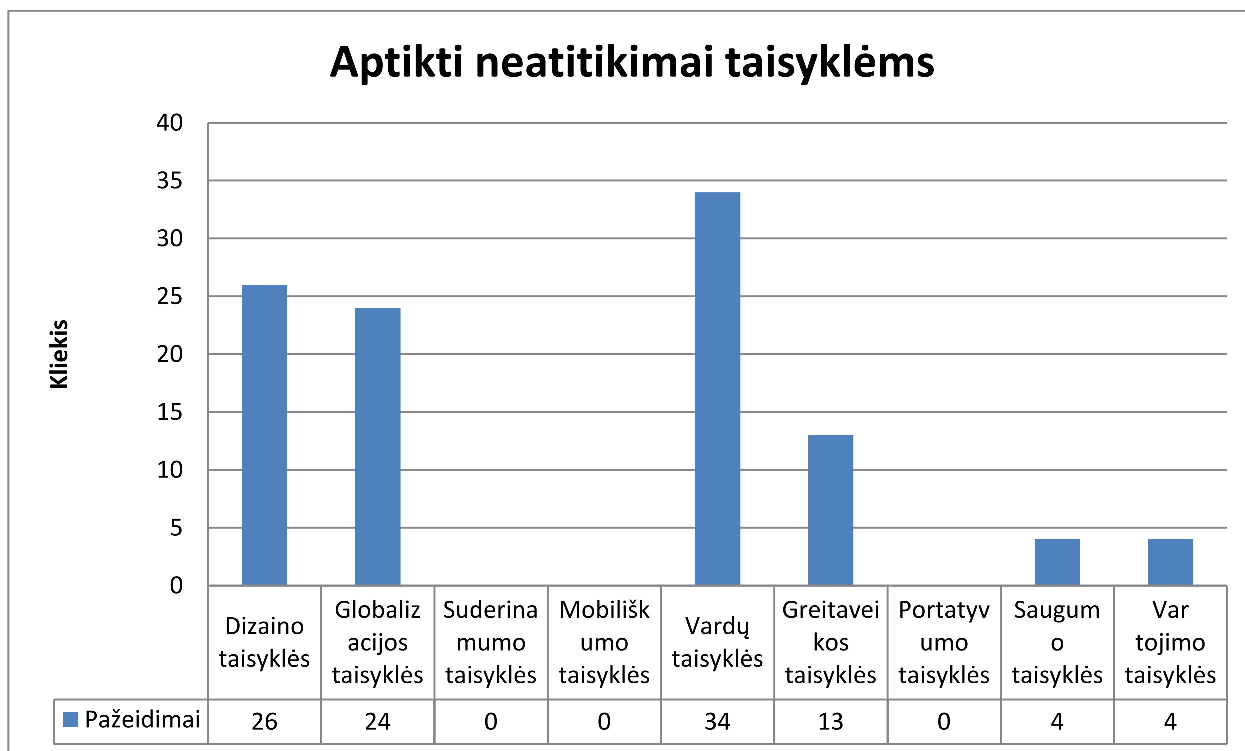
3 lentelė. SLOC statistika

Bendras kodo eilučių skaičius	10,856
- kodas	9,205 (84,8%)
- komentarai	1,162 (10,7%)
- kita	489 (4,5%)
Įskaičiuotinų kodo failų skaičius	27

Skirtingai MSDN pateikiamos vidutinės C# programinio kodo statistikos, pagal SLOC matavimo metodiką, kuri yra 70% - 20% - 10% (kodas - komentarai - kita (tuščios eilutės)) [10], „SourceHQ" rezultatai 85% - 10% - 5% skiriasi į kodo proporcijos naudą išimtinai dėl dviejų priežasčių. Kodo eilutės su juose esančiais komentarais buvo traktuojamos kaip gryno kodo eilutės, o tokia praktika aprašant kintamuosius ir vienos eilutės struktūras ar operacijas buvo plačiai taikoma. Kodo pertvarkymui ir struktūrizavimui aiškesniam ir kompaktiškam skaitymui buvo naudojamas NArrange įrankis, kuris užtikrindavo kodo stiliaus standartų išlaikymą, automatiškai modifikuodavo ir grupuodavo kodo fragmentus, pašalindamas daugelį tuščių kodo eilučių ir padidindavo funkcinių taškų programoje tankį.

4.2. Statinė kodo analizė

Statinei kodo analizei tikrinant galutinį produktą buvo naudojamas FxCop įrankis. Tyrimui naudojome visas Microsoft statinės kodo analizės taisykles, rekomendacijas ir naudojimo gidus pateikiamus „.NET Design Guidelines for Class Library Developers“. Platesnis šių taisyklių aprašymas pateikiamas 2.6. skyriuje. Galutinės versijos, neatlikus optimizacijos ir pažeidimų šalinimo veiksmų, buvo aptikta 105 taisyklių pažeidimai. Jų pasiskirstymas pagal tipus pateikiamas 5 pav.



5 pav. Aptiktų taisyklių nesilaikymų kiekis pagal jų tipus

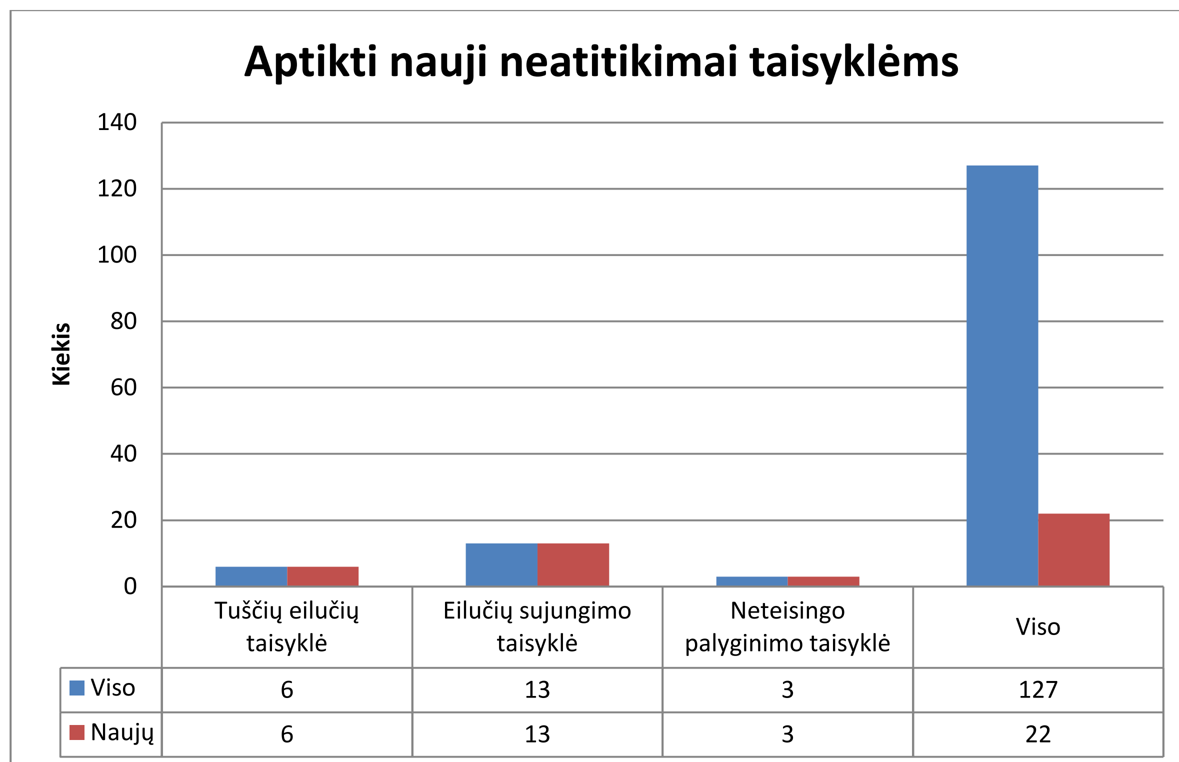
Pagrindinės klaidos dizaino grupėje buvo kintamųjų matomumo ribų klasėje neteisingas deklaravimas ir bendro nenumatytų klaidų valdymo mechanizmo neatitikimas rekomendacijomis. Iš globalizacijos pogrupio dažniausiai pasitaikanti klaida buvo neteisingas arba išvis nepanaudotas IFormatProvider parametras nurodantis spausdinančiosios funkcijos kultūros lenteles. Daugiausia klaidų „SourceHQ“ pakete kaip ir daugelyje kitų programų kurios nesilaiko Microsoft rekomenduojamos vengriškos notacijos (pirmas kintamojo vardo simbolis nurodo jo tipą) buvo susijusios su vardų taisyklių (angl. Naming Rules) tipu. Tarp jų dažnai pasitaikanti klaida, kai baziniai kintamieji, jų deklaravimo ir sukūrimo fazėje yra papildomai ir nereikalingai priskiriami (pavyzdžiui „Boolean“ tipo kintamojo papildomas priskirimo „false“ reikšmei jo

sukūrimo metu). Greitaveikos, į kurią kūrimo metu buvo kreipiamas didesnis dėmesys, klaidų kiekis siekė 13. Iš jų daugiau nei pusė buvo susiję su statinių klasių nepanaudojimų. „.NET Design Guidelines for Class Library Developers“ gidas rekomenduoja, kad visos klasės kurios paveldi arba kitu būdu turi priėjimą į savo paties objektą (C# kalboje - „this“, VB kalboje - „Me“) tačiau jo nenaudoja ir yra autonominės, turi būti deklaruojamos kaip statinės. Šis neatitikimas siūlomiems standartas nebuvo taisomas, kadangi papildomos priemonės kurios buvo naudojamos dinaminei programos metrikų analizei, reikalavo aktyvias (t.y. naudojamas analizei) klases sukurti dinamiškai naudojantis specialias užklausas. Iš saugumo ir panaudojimo tipų buvo užfiksuota po 4 klaidas, kurios visos susijusios su nepanaudotų kintamųjų deklaravimu arba papildomų saugumo priemonių nenaudojimų „LinkDemands“ srityje.

Kadangi produktas buvo kuriamas personalinių kompiuterių rinkai, kaip ir tikėtasi bei iš anksto numatyta, suderinamumo, mobilumo ir portatyvumo klaidų aptikta nebuvo.

4.3. Papildomų taisyklių realizacija

Realizavus tris papildomas programinio kodo statinės analizės taisykles buvo aptiktos 22 papildomos klaidos, kurių detalus pasiskirstymas pateiktas 6 pav.



	Tuščių eilučių taisyklė	Eilučių sujungimo taisyklė	Neteisingo palyginimo taisyklė	Viso
Viso	6	13	3	127
Naujų	6	13	3	22

6 pav. Naujų pasiūlytų taisyklių nesilaikymų paplitimas

Didžioji dalis aptiktų klaidų buvo susijusios su neteisingu eilučių sujungimu. Tai ypač aktualu metoduose kurie tiesiogiai dirba su eilutės tipo kintamaisiais juos dinamiškai formuodami, kuo ir pagrįstas „SourceHQ“ veikimas. Tuščios eilutės priskyrimo taisyklės pažeidimas buvo užfiksuotas 6 kartus, o neteisingo palyginimo - 3.

Toliau aptarkime kiekvienos taisyklės aptikimą individualiai.

4.3.1. Tuščių eilučių taisyklė

Ši taisyklė uždraudžia tiesioginį tuščių eilučių panaudojimą ir rekomenduoja jas pakeisti „String.Empty“ atributu. Dažniausiai pasitaikantys taisyklės nesilaikymo ir pažeidimo atvejai yra kai naujai sukurtam „string“ tipo kintamajam yra iš karto priskiriama tuščios eilutės reikšmė ir neteisingas tuščios eilutės perdavimas metodų parametruose. Aptarsime abu šiuos atvejus.

Žemiau pateikiame tą pačią funkciją, parašytą su spragomis (t.y. nesilaikant mūsų apibrėžtų programinio kodo rašymo taisyklių) ir korektišką jos variantą.

```
public static string GetFileUniqueKey(CodeInfo code)
{
    if (code == null)
        return "";

    return GetFileUniqueKey(code.absPath, "");
}
```

7 pav. Neatitinkantis tuščios eilutės taisyklės kodo fragmentas

```
public static string GetFileUniqueKey(CodeInfo code)
{
    if (code == null)
        return String.Empty;

    return GetFileUniqueKey(code.absPath, String.Empty);
}
```

8 pav. Kodo fragmentas atitinkantis tuščios eilutės taisyklę

GetFileUniqueKey metodas paskirtis yra patikrinti gaunamo parametro korektiškumą ir jei jis egzistuoja, perduoti jo reikšmę kitam metodui, priešingu atveju - grąžinti tuščią eilutę. Nors galbūt klaidingo pavyzdžio atveju kodas ir atrodo trumpesnis bei lengviau skaitomas ir suprantamas, taip pat sumažėja kreipinių į globalius kintamuosius („String.Empty“), tačiau dėl .NET architektūrinių ypatybių, toks kodas nėra optimalus [13].

Išnagrinėkime ir palyginkime šių metodų CIL kodą naudodamiesi .NET IL Disassembler įrankiu:

Kodas neatitinkantis taisyklių	Korektiškas kodas

`.method public hidebysig static string GetFileUniqueKey(class SourceHQ.CodeInfo code) cil managed`
`{`
`  // Code size       26 (0x1a)`
`  .maxstack  8`
`  IL_0000:  ldarg.0`
`  IL_0001:  brtrue.s   IL_0009`
`  IL_0003:  ldstr      ""`
`  IL_0008:  ret`
`  IL_0009:  ldarg.0`
`  IL_000a:  ldfld      string SourceHQ.CodeInfo::absPath`
`  IL_000f:  ldstr      ""`
`  IL_0014:  call       string SourceHQ.FileTools::GetFileUniqueKey(string, string)`
`  IL_0019:  ret`
`}`
	`.method public hidebysig static string GetFileUniqueKey(class SourceHQ.CodeInfo code) cil managed`
	`{`
	`  // Code size       26 (0x1a)`
	`  .maxstack  8`
	`  IL_0000:  ldarg.0`
	`  IL_0001:  brtrue.s   IL_0009`
	`  IL_0003:  ldsfld     string [mscorlib]System.String::Empty`
	`  IL_0008:  ret`
	`  IL_0009:  ldarg.0`
	`  IL_000a:  ldfld      string SourceHQ.CodeInfo::absPath`
	`  IL_000f:  ldsfld     string [mscorlib]System.String::Empty`
	`  IL_0014:  call       string SourceHQ.FileTools::GetFileUniqueKey(string, string)`
	`  IL_0019:  ret`
	`}`

9 pav. GetFileUniqueKey metodo CIL kodo fragmentai

CIL kodo fragmentai skiriasi tik dviem eilutėmis (paryškintos). Klaidingas kodas naudoja LDSTR („Load String“) operaciją, tuo tarpu korektiškas kodas kviečia LDSFLD („Load String Field“). Įsigilinus į šių operacijų turinį ir paskirtį, korektiškai parašyto kodo privalumai akivaizdūs. LDSTR operacija nurodo sukurti naują „string“ tipo atmintyje, kuris paveldi visas String klasės savybes, išskiriant jam privačią atmintį bei priregistruojant jį GC (šiukšlių surinkimo, angl. .NET Garbage Collector) lentelėje. Tai sudėtingas ir papildomų resursų reikalaujantis procesas. Be to, GC lentelėje esantis objektas, papildomai sunaudos procesoriaus ciklų jį sunaikinant ir atlaisvinant užimtą atmintį, baigiantis programos darbui.

Korektiškame metodo CIL kode yra naudojama LDSFLD operacija. Ji yra daug paprastesnė ir nereikalaujanti naujų objektų sukūrimo. Perduodamam parametrui yra priskiriamas atminties adresas į iš anksto .NET sukurtą „String.Empty“ objektą (esantį MSCorLib.dll bibliotekoje) išvengiant bet kokių GC kreipinių. Svarbu paminėti, kad programuojant, tiek tiesioginė tuščia eilutė, tiek „String.Empty“ parametras yra identiški programiniu atžvilgiu. Todėl mūsų užsibrėžtos taisyklės pritaikymas ir kodo ištaisymas pagal pateiktas rekomendacijas, niekaip neįtakoja pačios programos logikos ir veikimo.

4.3.2. Neteisingo palyginimo taisyklė

Ši taisyklė apibrėžia rekomenduotinas dviejų „string“ tipo eilučių palyginimo gaires naudojant „String.Compare“ metodą vietoje tiesioginio „==” operatoriaus. Palyginkime žemiau pateiktą metodą parašyta nesilaikant nurodytos taisyklės ir korektišką jo variantą.

```
public bool FileDiffer(string path1, string path2)
{
    if (GetFileChecksum(path1) == GetFileChecksum(path2))
        return false;
    return true;
}
```

10 pav. Neatitinkantis neteisingo palyginimo taisyklės kodo fragmentas

```
public bool FileDiffer(string path1, string path2)
{
    if (String.Compare(GetFileChecksum(path1), GetFileChecksum(path2)) == 0)
        return false;
    return true;
}
```

11 pav. Kodo fragmentas atitinkantis neteisingo palyginimo taisyklę

„FileDiffer“ metodas palygina ir grąžina loginę reikšmę nurodančią ar du parametrais nurodyti failai yra skirtingi (rezultatas „true“) ar identiški (rezultatas „false“). Metodo viduje esantis metodas „GetFileChecksum“ grąžina „string“ tipo eilutę su specialiu algoritmu paruošta tekstine informaciją identifikuojančią nurodytą failą. Jei šie identifikatoriai yra lygūs (t.y. dvi gautos tekstinės eilutės yra vienodos), vadinasi ir nurodyti failai nesiskiria, kas ir yra šio metodo rezultatas. Tačiau 10 pav. nurodytas metodas naudoja tiesioginius lygybės operatorius, o 11 pav. esantis metodas laikosi mūsų taisyklės užsibrėžtų programavimo gairių ir kviečia „String.Compare“ metodą, kuris esant identiškom eilutėm gražina „0“.

Šių dviejų metodų atitinkamas sukompiliuotas CIL kodas atrodo taip:

Kodas neatitinkantis taisyklių	Korektiškas kodas

.method public hidebysig instance bool FileDiffer(string path1, string path2) cil managed
{
  // Code size   25 (0x19)
  .maxstack 8
  IL_0000: ldarg.0
  IL_0001: ldarg.1
  IL_0002: call   instance string SourceHQ.DiffTools::GetFileChecksum(string)
  IL_0007: ldarg.0
  IL_0008: ldarg.2
	.method public hidebysig instance bool FileDiffer(string path1, string path2) cil managed
	{
	  // Code size 25 (0x19)
	  .maxstack 8
	  IL_0000: ldarg.0
	  IL_0001: ldarg.1
	  IL_0002: call instance string SourceHQ.DiffTools::GetFileChecksum(string)
	  IL_0007: ldarg.0
	  IL_0008: ldarg.2

`IL_0009:  call       instance string SourceHQ.DiffTools::GetFileChecksum(string)`
`IL_000e:  call       bool [mscorlib]System.String::op_Equality(string, string)`
`IL_0013:  brfalse.s  IL_0017`
`IL_0015:  ldc.i4.0`
`IL_0016:  ret`
`IL_0017:  ldc.i4.1`
`IL_0018:  ret`
`}`
	`IL_0009:  call       instance string SourceHQ.DiffTools::GetFileChecksum(string)`
	`IL_000e:  call       int32 [mscorlib]System.String::Compare(string, string)`
	`IL_0013:  brtrue.s   IL_0017`
	`IL_0015:  ldc.i4.0`
	`IL_0016:  ret`
	`IL_0017:  ldc.i4.1`
	`IL_0018:  ret`
	`}`

12 pav. FileDiffer metodo CIL kodo fragmentai

Kaip matome iš CIL kodo fragmentų, „==” tiesioginis operatorius yra ekvivalentus „op_Equality()“ metodo kvietimui su užduodamomis eilutėmis palyginimui kaip parametrai. Tuo tarpu korektiškas kodas kviečia „Compare()“ metodą. Tik plačiau išnagrinėjus šiuos du metodus galime spręsti, kad taisykles atitinkantis programinis kodas yra pranašesnis ir efektyvesnis dėl šių 3-jų punktų:

- op_Equality metodas yra abstraktus ir universalus tinkantis bet kokioms palyginimo operacijoms atlikti, todėl dviejų eilučių palyginimo operacijai atlikti, jis kviečia kitą metodą ir grąžina apdorotą rezultatą. Tuo tarpu kviečiant Compare tiesiogiai yra išvengiama metodų peradresavimų ir papildomų funkcijų kvietimų. Šis metodas yra specializuotas eilučių palyginimo operacijai ir rezultatą formuoja tiesiogiai jo viduje.
- op_Equality grąžina „boolean“ tipo reikšmę nurodančią ar eilutės yra vienodos ar ne. Compare metodas tikrina ne tik eilučių identiškumą bet ir reliatyvią (pagal leksiką) jų poziciją viena kitos atžvilgiu.
- Bene svarbiausias privalumas nesusijęs su veikimo spartos charakteristikomis yra tas, kad Compare metodas, skirtingai nei jo op_Equality alternatyva, eilutes tikrina pagal programoje nustatytą kultūros informaciją. Tai yra labai svarbus saugumo veiksnys, kurį neretai yra sunku aptikti net ir specializuoto testavimo metu [19].

Nors tiek veikimo greičio charakteristikomis, MSDN rekomendacijomis ir saugumo aspektais pagal mūsų pasiūlytą taisyklę realizuotas programos kodas lenkia aukščiau aprašytą nekorektišką kodą, tačiau šios taisyklės griežtai taikyti negalime dėl .NET programavimo kalbos ypatybių. Tiesioginis „==” operatorius (kaip ir visi kiti) gali būti perkrautas (angl. overloaded) kitu, iš anksto deklaruotu privačiu metodu atliekančiu jam paskirtą konkrečią funkciją. Todėl

FxCop programai rodant šios taisyklės pažeidimą, programuotojas turėtu pilnai įsitikinti, kad tai yra klaida, o ne norimas logiškas programos veikimas.

4.3.3. Eilučių sujungimo taisyklė

Ši taisyklė rekomenduoja, vietoje tiesioginio „+“ operatoriaus, naudoti „StringBuilder“ klasę įvairioms eilučių apjungimo operacijoms atlikti. Nors ir pagrindinis dokumentas nurodantis .NET platformos programavimo gaires „Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries“ turi atskirą skyrių dedikuotą būtent šiai problemai spręsti, tačiau nekorektiškas programavimas yra vis dar labai paplitęs [15]. Ši klaida buvo dažniausiai pasitaikanti ir mūsų kuriamoje „SourceHQ“ programoje. Tai galima paaiškinti tuo, kad vietoje vieno eilučių sudėties operatoriaus tenka rašyti žymiai ilgesnį kodą, su naujų klasių deklaravimu ir panaudojimu, kas apsunkina ne tik kodo rašymą, bet ir jo skaitomumą. Žemiau pateikiame ir palyginame du pagal veikimo principą analogiškus, tačiau skirtingai parašytus programinio kodo fragmentus.

```
static string GetVersion(string p)
{
    string curVer = p + VER;
    return curVer;
}
```

13 pav. Kodo fragmentas neatitinkantis nurodytos taisyklės

```
static string GetVersion(string p)
{
    StringBuilder curVer = new StringBuilder(p);
    curVer.Append(VER);
    return curVer.ToString();
}
```

14 pav. Korektiškas kodo fragmentas

Iškviestas „GetVersion“ metodas grąžina iš dviejų dalių (nekintančios statinės eilutės „VER“ kintamojo ir perduotos konkrečios reikšmės) sudarytą naują eilutę, su suformuota versijos informacija. Abiejų metodų veikimo logika yra tokia pati. Akivaizdu, kad 13 pav. pateiktas kodas yra gerokai trumpesnis, lengviau skaitomos ir suprantamas. 14 pav. esančiame kode yra deklaruojamas ir sukuriamas naujas objektas, o eilučių apjungimui naudojamas „StringBuilder“ klasės „Append“ metodas. Galiausiai grąžinama eilutė verčiama į „string“ tipą naudojant „ToString“ kreipinį. Tačiau kuris metodas yra optimalesnis, galime spręsti tik išanalizavę CIL kodą, kuris yra pateiktas 15 pav.

Kodas neatitinkantis taisyklių

```
.method private hidebysig static
string  GetVersion(string p) cil
managed
{
  // Code size       19 (0x13)
  .maxstack  2
  .locals init ([0] string curVer, [1]
string CS$1$0000)
  IL_0000:  nop
  IL_0001:  ldarg.0
  IL_0002:  ldstr      "1.0.0.0"
  IL_0007:  call       string
[mscorlib]System.String::Concat(string
, string)
  IL_000c:  stloc.0
  IL_000d:  ldloc.0
  IL_000e:  stloc.1
  IL_000f:  br.s       IL_0011
  IL_0011:  ldloc.1
  IL_0012:  ret
}
```

Korektiškas kodas

```
.method private hidebysig static
string  GetVersion(string p) cil
managed
{
  // Code size       31 (0x1f)
  .maxstack  2
  .locals init ([0] class
[mscorlib]System.Text.StringBuilder
curVer, [1] string CS$1$0000)
  IL_0000:  nop
  IL_0001:  ldarg.0
  IL_0002:  newobj     instance void
[mscorlib]System.Text.StringBuilder::.
ctor(string)
  IL_0007:  stloc.0
  IL_0008:  ldloc.0
  IL_0009:  ldstr      "1.0.0.0"
  IL_000e:  callvirt   instance class
[mscorlib]System.Text.StringBuilder
[mscorlib]System.Text.StringBuilder::A
ppend(string)
  IL_0013:  pop
  IL_0014:  ldloc.0
  IL_0015:  callvirt   instance string
[mscorlib]System.Object::ToString()
  IL_001a:  stloc.1
  IL_001b:  br.s       IL_001d
  IL_001d:  ldloc.1
  IL_001e:  ret
}
```

15 pav. GetVersion metodo CIL kodo fragmentai

Kadangi metodo algoritmas eilutėms apjungti skiriasi kardinaliai, šie skirtumai atsispindi ir aukščiau pateiktame CIL kode. Tačiau esminis skirtumas yra tas, kad „+“ operatorius „string“ tipo eilučių apjungimui kviečia „Concat“ metodą, tuo tarpu korektiškas metodas atlieka 3 atskirus veiksmus: 1. Sukuria nauja „StringBuilder“ objektą; 2. Kviečia „Append“ metodą nurodytoms eilutėms apjungti; 3. Galutinį rezultatą paverčia „string“ tipo eilute naudojant „ToString“ metodą. Atlikus šių dviejų metodų veikimo greičių charakteristikų palyginimą (platesnė analizė pateikiama eksperimentinėje šio dokumento dalyje), matyti, kad korektiškas taisyklių atžvilgiu parašytas metodas yra tiek lėtesnis, tiek reikalauja daugiau kompiuterio atminties papildomiems objektams saugoti. Be to, programa tampa sunkiau skaitoma, kodas komplikuotas. Todėl kyla natūralus klausimas - kodėl taikyti šią taisyklę? Atsakymas slypi architektūriniame šių dviejų metodų lygmenyje. „Concat“ operacija dėl savo abstrakcijos lygio (ją galima taikyti ne tik dviejų eilučių sujungimui) reikalauja daugiau resursų ir veikia gerokai lėčiau nei „Append“ metodas kuris yra specializuotas dviejų eilučių sujungimo operacijoms atlikti. Tačiau papildomi 2 žingsniai: „StringBuilder“ objekto sukūrimas ir rezultato vertimas į „string“ tipą reikalauja

papildomų procesoriaus ciklų ir atminties, kas lemia viso siūlomo algoritmo greičio charakteristikos sumažėjimą. Tačiau šios dvi operacijos, nors ir lėtos, yra kviečiamos tik vieną kartą, kol „Append“ operacija gali būti naudojama pakartotinai. Galime prieiti išvados, kad norint pagerinti algoritmo veikimo charakteristikas, reiktų atsižvelgti į konkretų atvejį: jei eilučių sujungimo operacijų daug - rekomenduojama taikyti nurodytą taisyklę; jei eilučių sujungimas naudojamas tik vieną (arba nedaug) kartą, galima šia taisyklę ignoruoti. Efektyvumo ir eilučių sujungimo skaičiaus sąryšis pateikiamas eksperimentinėje šio dokumento dalyje. Svarbu paminėti, kad naudojant korektišką kodą, padidėja kodo portatyvumas ir galimybė jį plėsti. Tačiau, kaip ir neteisingo eilučių palyginimo taisyklės atveju, kiekvienas FxCop įrankiu rastas nekorektiškas kodo fragmentas, nebūtinai yra klaida, o gali būti taisyklinga loginė išraiška. Todėl papildoma programuotojo peržiūra kiekvienu konkrečiu atveju yra būtina.

5. EKSPERIMENTINĖ DALIS

Savo realizuotas statinės kodo analizės taisykles išbandėme su magistro darbo projekto metu realizuota realia veikiančia „SourceHQ“ programa, taip pat su testine specialiai tam parašyta „Hello World“ programa bei su šiomis atviro kodo sistemomis:

NQuery - reliacinės duomenų bazės užklausų biblioteka parašyta C# kalba. Ji leidžia vykdyti „SELECTquery“ visiems .NET objektams įskaitant masyvus, duomenų setus (angl. data sets) bei įvairias lenteles.

Dejavu - lengvo pakartotinio panaudojimo biblioteka skirta atstatymo (angl. undo/redo) funkcijai taikomosiose programose realizuoti. Ji automatiškai seka vartotojo veiksmus, juos registruoja su galimybe atkurti ankščiau buvusią būseną.

NFileStorage - projektas skirtas palengvinti programuotojams valdyti didelius failų kiekius be duomenų bazių poreikio.

USPS OneCode - biblioteka skirta USPS pašto siuntinių kodo generavimui ir šifravimui.

DotNetZip - populiariausia nemokama ZIP archyvų valdymo ir kūrimo biblioteka.

ScintillaNET - didžiulis projektas kurio pagrindinė užduotis sukurti universalų daugia-platforminį ir daugiakalbį teksto redaktorių. Šį komponentą kaip savo veikimo pagrindą naudoja daugybė žinomų programinės įrangos paketų tokių kaip CSharp Developer, CSh ir kiti.

IronPython - iš daugiau nei pusė milijono kodo eilučių parašytas Python programavimo kalbos kompiliatorius skirtas .NET platformai.

5.1. Bandymai su specialiu programiniu kodu

„Hello World“ programa buvo sukurta konkrečioms mūsų tiriamoms taisyklėms analizuoti. Programa turi atskiras klases, kur kiekvienoje iš jų yra parašyti specialūs metodai su programuotojams įprastu ir pagal pasiūlytas taisykles korektišku kodu. Eksperimento metu kiekvienas metodas buvo vykdomas atskirai, naudojant „CLR Profiler“ įrankį ir vidines .NET konstrukcijas bandant nustatyti šias veikimo charakteristikas: 1) vykdymo greitį; 2) operatyvinės atminties (RAM) suvartojimą;

Eilučių sujungimo taisyklės testavimui parašėme ir išbandėme šiuos metodus:

Test1A metodas	Test1B metodas

`private string Test1A()`
`{`
`    String res = String.Empty;`
`    for (int i = 0; i < 1000; i++)`
`    {`
`        res = res + "ABC";`
`    }`
`    return res;`
`}`
	`private string Test1B()`
	`{`
	`    String res = String.Empty;`
	`    StringBuilder b = new`
	`StringBuilder();`
	`    for (int i = 0; i < 1000; i++)`
	`    {`
	`        b.Append("ABC");`
	`    }`
	
	`    return b.ToString();`
	`}`

16 pav. Test1A ir Test1B metodų programiniai kodai

Testavimo metu tyrėme metodo vykdymo greitį, bendros išskirtos atminties kiekį (taip pat išskyrėme pakartotinai perskirstytos ir „krūvos“ (heap) atminties kiekius), nustatėme sukurtų kreipinių į objektų adresus (handles) bei „Gen 0-3“ objektų kuriuos apdoroja .NET GC priemonė kiekius. Rezultatai pateikiami 4 lentelėje.

4 lentelė. Test1A ir Test1B vykdymo charakteristikų palyginimo lentelė

Charakteristika	Test1A metodas	Test1B metodas
Įvykdymo greitis [s.]	1.637011967	0.04296831
Išskirta atmintis (Allocated) [baitai]	3,038,054	33,532
Perskirstyta atmintis (Relocated) [baitai]	48,268	0
Išskirtos "krūvos" dydis (Final Heap) [baitai]	164,922	33,532
Kreipiniai į adresus (Handles) [baitai]	28	28
"Gen 0-3" objektų	11	0

Kaip matome iš rezultatų, pagal mūsų taisykles parašytas metodas yra vykdomas beveik 40 kartų greičiau ir reikalauja 90 kartų mažiau atminties. Vykdymo greičio charakteristika labiausiai įtakoja metodo viduje esančio ciklo vykdymo kiekis. Nors Test1B metodas ir reikalauja dviejų papildomų resursams intensyvių metodų panaudojimą (StringBuilder sukūrimą ir rezultato vertimą į „string“ tipo eilutę), tačiau kai eilutės sujungimo veiksmų yra daug (šiuo atveju 1000), greitai įvykdomas „Append“ metodas visa tai kompensuoja. Test1A metodas pareikalavo išskirti net 3 MB atminties. Taip pat buvo sukurti 11 „Gen0“ objektų, kuriuos vėliau .NET platformos GC įrankis išlaisvinantis nebenaudojamus resursus, turės apdoroti. Tuo tarpu

Test1B metodo visa reikalaujama atmintis buvo išskirta „krovos“ (angl. Heap) struktūroje, kuri yra greitai prieinama ir lengvai išvaloma (t.y. atlaisvinama).

Eilučių palyginimo taisyklės tyrimui naudojome tokius metodus:

Test2A metodas	Test2B metodas

`private bool Test2A(string x)`
`{`
`    if (x.ToLower() == "testas")`
`        return true;`
`    return false;`
`}`
	`private bool Test2B(string x)`
	`{`
	`    if (String.Compare(x, "testas",`
	`true) == 0)`
	`        return true;`
	`    return false;`
	`}`

17 pav. Test2A ir Test2B metodų programiniai kodai

Vykdymo charakteristikos pateikiamos 5 lentelėje.

5 lentelė. Test2A ir Test2B vykdymo charakteristikų palyginimo lentelė

Charakteristika	Test2A metodas	Test2B metodas
Įvykdymo greitis [s.]	0.198623009	0.13635504
Išskirta atmintis (Allocated) [baitai]	19,020	19,596
Perskirstyta atmintis (Relocated) [baitai]	0	0
Išskirtos "krūvos" dydis (Final Heap) [baitai]	19,020	19,596
Kreipiniai į adresus (Handles) [baitai]	28	28
"Gen 0-3" objektų	0	0

Šio metodo esmė, palyginti perduodamą parametrą su konstantos reikšme neatsižvelgiant į didžiąsias ir mažąsias raides. Jei šios dvi lyginamos eilutės yra lygios - grąžinti „true“ reikšmę, jei ne - „false“. Test2A metodas naudoja tiesioginį „==“ lyginimo operatorių, kas iššaukia „neteisingo eilučių palyginimo“ klaidą FxCop įrankyje, tuo tarpu metodas Test2B naudoja pagal mūsų apibrėžtas taisykles parašytą korektišką kodą.

Eksperimento rezultatai įdomūs tuo, kad nors ir Test2B metodas buvo įvykdytas beveik 30% greičiau, tačiau jis pareikalavo daugiau atminties. Nors ir pareikalautas atminties kiekis skyrėsi vos 3% tačiau sistemose kur atmintis yra itin brangi, o procesoriaus ciklai - pigūs, tai gali būti esminis veiksnys. Dažniausiai pasitaikančiose taikomosiose programose 30% vykdymo pagreitėjimas, net ir 3% padidinus atminties suvartojimą yra daug svarbesnis, todėl šios taisyklės taikymas yra rekomenduojamas. Galime prieiti išvados, kad absoliučiai daugumai atveju, mūsų

taisyklės pritaikymas pagerins bendras algoritmo vykdymo charakteristikas, padidins jo galimybę plėsti, portatyvumą ir kaip aprašėme šio dokumento analizės dalyje - saugumą.

Norėdami nustatyti tuščių eilučių taisyklės privalumus ir trūkumus naudojome šiuos metodus, kurie geriausiai atspindi taisyklės savybes:

Test3A metodas	Test3B metodas

`private string Test3A()`
`{`
`    string str = "";`
`    return "";`
`}`
	`private string Test3B()`
	`{`
	`    string str = String.Empty;`
	`    return String.Empty;`
	`}`

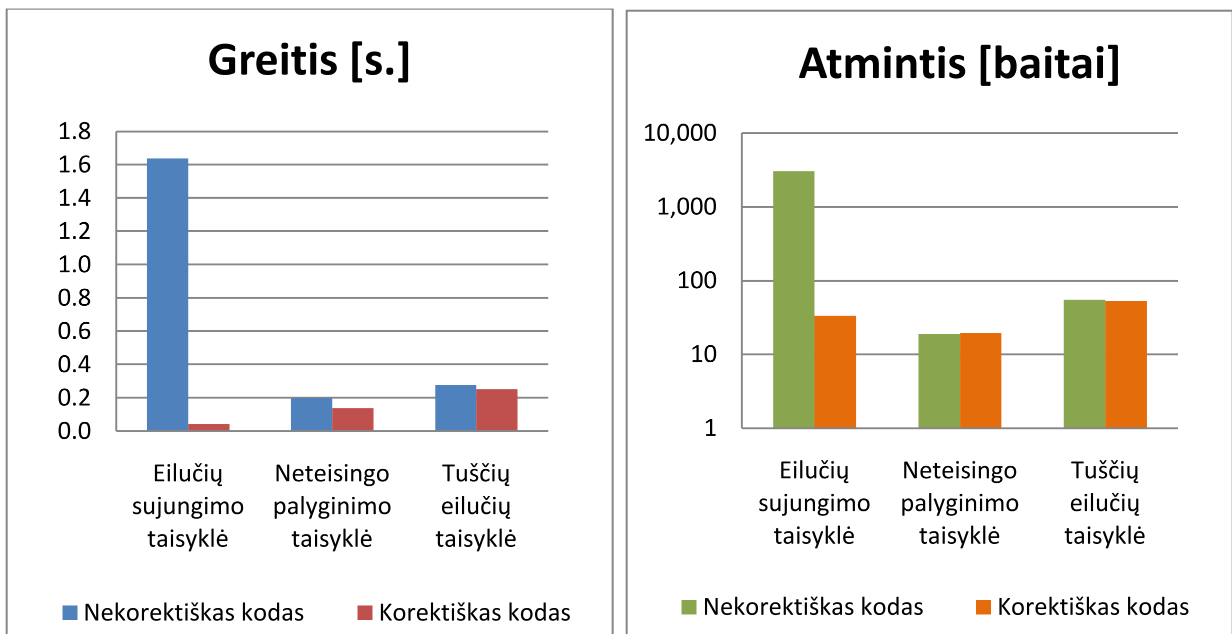
18 pav. Test3A ir Test3B metodų programiniai kodai

Įvykdžius ir ištyrus šiuos metodus gavome tokius rezultatus:

6 lentelė. Test3A ir Test3B vykdymo charakteristikų palyginimo lentelė

Charakteristika	Test3A metodas	Test3B metodas
Įvykdymo greitis [s.]	0.27681	0.24987
Išskirta atmintis (Allocated) [baitai]	55,290	53,314
Perskirstyta atmintis (Relocated) [baitai]	0	0
Išskirtos "krūvos" dydis (Final Heap) [baitai]	55,290	53,314
Kreipiniai į adresus (Handles) [baitai]	35	35
"Gen 0-3" objektų	0	0

Naudodami .NET Stopwatch klasę nustatėme, kad naudojant „String.Empty“ vietoje tuščios eilutės deklaravimo, kodas yra įvykdomas 10% greičiau, o „CLR Profiler“ įrankio rezultatai parodė 4% mažesnį atminties suvartojimą. Taisyklės naudojimas ir metodo perrašymas į korektišką niekaip neįtakoja kodo vykdymo logikos, kodo skaitomumo ir aiškumo laipsnis taip pat nekinta. Korektišką kodą rekomenduoja naudoti tiek oficialūs .NET dokumentai, tiek neoficialūs MSDN gidai, todėl atsižvelgę į visą turimą informaciją galime griežtai teigti, kad šios taisyklės pritaikymas programiniame kode turėtų būti privalomas be išimčių.



19 pav. Veikimo charakteristikų palyginimo grafikas

Iš palyginamųjų diagramų ir apžvelgus bendrus rezultatus galime teigti, kad visų 3-jų taisyklių panaudojimas padidina kodo vykdymo greitį ir sumažina arba beveik nepakeičia („neteisingo palyginimo taisyklės“ atveju - padidėjimas 3 procentais).

5.2. Bandymai su „SourceHQ“ programos paketu

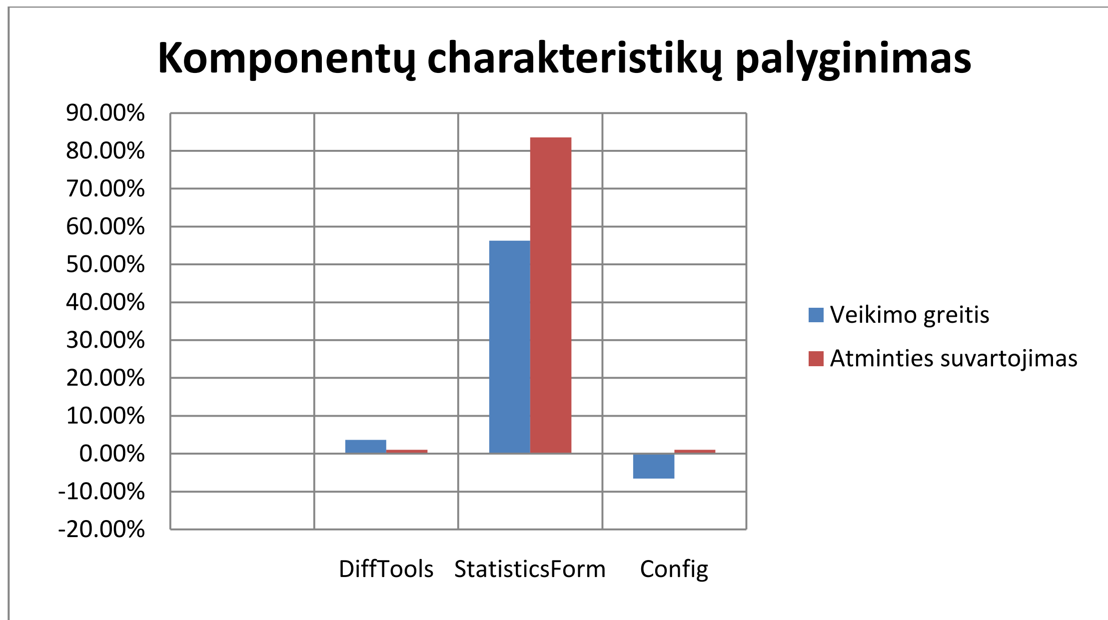
Kaip jau aprašėme 4.3. dalyje atlikus statinę analizę su trimis naujomis mūsų apibrėžtomis taisyklėmis, buvo aptikta 22 nauji nekorektiško kodo fragmentai. Kadangi visą projektą sudaro 9205 programinio kodo eilutės, galime teigti, kad vidutiniškai šios 3 eilutės paveikia 1 iš 418 eilučių arba 0,0024%. Kadangi „SourceHQ“ programinė įranga veikia vartotojo dialogo principu ir visapusiškas jos testavimas veikimo charakteristikų atžvilgiu yra sudėtingas ir nevienareikšmiškas, todėl sistemoje išskyrėme 3 autonomines klases (DiffTools, StatisticsForm ir Config) ir jas ištyrėme išbandydami korektišką ir nekorektišką kodą.

7 lentelė. Pažeistų taisyklių skaičiaus klasėse palyginimo lentelė

Klasė	Kodo eilučių skaičius	Klaidų skaičius taisyklėse			
		Tuščios eilutės	**Neteisingo palyginimo**	**Eilučių sujungimo**	**Viso**
DiffTools	384	1	1	0	**2**
StatisticsForm	474	0	1	5	**6**
Config	562	3	0	1	**4**

8 lentelė. Veikimo charakteristikų pasikeitimo palyginimo lentelė

Klasė	Veikimo greitis				Atminties sunaudojimas		
	Pradinio kodo	**Ištaisyto kodo**	**Pokytis**		**Pradinio kodo**	**Ištaisyto kodo**	**Pokytis**
DiffTools	0.74754	0.72124	**3.64%**		712,870	712,811	**0%**
StatisticsForm	1.27088	0.55591	**56.25%**		1,752,502	954,955	**83.51%**
Config	0.11212	0.12004	**-6.59%**		32,756	32,756	**0%**



20 pav. Klasių veikimo charakteristikų palyginimo grafikas

Kaip matome iš testų rezultatų DiffTools klasės kodo perrašymas į korektišką davė minimalios naudos, tačiau tai galima paaiškinti nedideliu pradinių klaidų kiekiu. StatisticsForm klasės optimizavimas pagal taisykles davė ženklius teigiamus veikimo greičio ir atminties suvartojimo pasikeitimus, kadangi ši klasė intensyviai vykdo daug resursų reikalaujančias operacijas su eilutėmis generuodama ataskaitas. Config klasės optimizavimas, dėl realizuotų algoritmų subtilybių davė priešingus rezultatus, nes norint sėkmingai kodą pritaikyti pagal taisyklių gaires teko jį papildyti. Šis algoritmo vykdymo charakteristikų sumažėjimas nėra didelis (6,52% greičio atžvilgiu), o kodas pakeitus jį korektišku analogu, tapo portatyvesnis ir atitinkantis visus MSDN keliamus reikalavimus. Apibendrinant „SourceHQ“ rezultatus atliekant statinę kodo analizę pagal užsibrėžtas papildomas 3 taisykles galime teigti, jog kodą pakeisti į korektišką yra verta tiek veikimo spartos, tiek atminties suvartojimo atžvilgiu.

5.3. Bandymai su kitais programų paketais

Norėdami išbandyti pasiūlytų statinės kodo analizės taisyklių efektyvumą ir paplitimą realiomis sąlygomis, eksperimentus tęsėme su 7-iomis populiariomis taikomosiomis programomis ir bibliotekomis: NQuery, Dejavu, NFileStorage, USPS OneCode, DotNetZip, ScintillaNET ir IronPython. Jas pasirinkome neatsitiktinai ir norėdami atlikti bandymus su paketų įvairovę. Dalis šių sistemų yra kuriamos kaip maži, vieno asmens valdomi projektai turintys mažiau nei 10 tūkst. programinio kodo eilučių. Kiti - didžiuliai projektai, susidedantys iš 50 tūkst. kodo eilučių, didelių programuotojų komandų ir turi šimtus tūkstančių vartotojų visame pasaulyje. Žymus veiksnys pasirenkant šiuos paketus buvo tai, kad visi jie yra atviro kodo sistemos. Nors FxCop įrankio pagalba, tirti ir analizuoti statinį kodą galime bet kokio .NET platformoje sukompiliuoto vykdomojo (EXE) ar bibliotekos (DLL) failo, tačiau turint visus pilnus išeities kodus, mums leidžia paskaičiuoti klaidų ir nekorektiško kodo paplitimą kodo eilučių skaičiaus atžvilgiu.

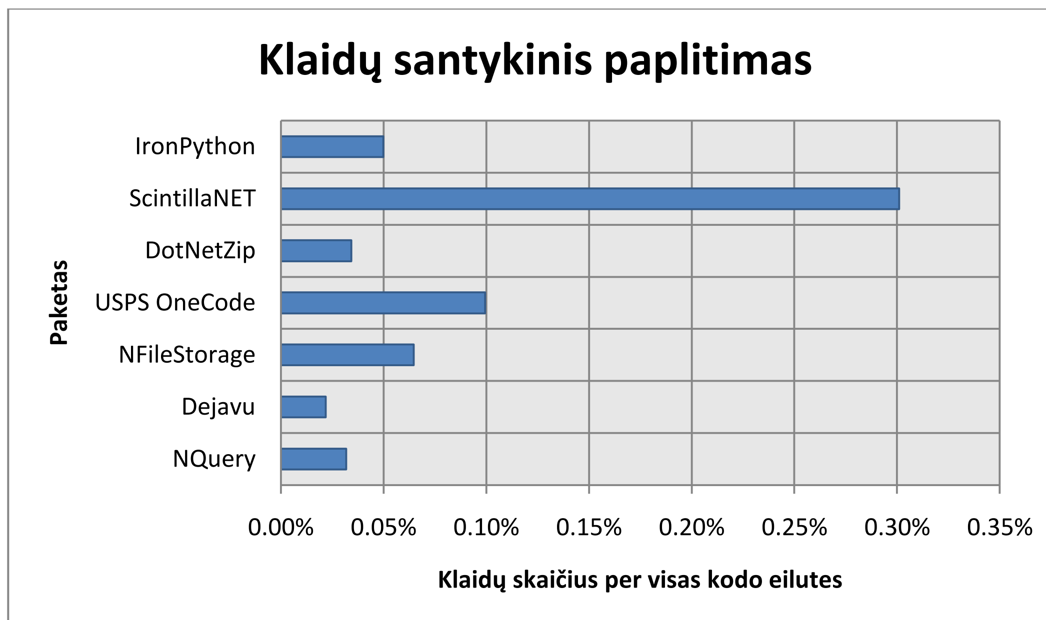
Analizės rezultatai pateikiami 9 lentelėje.

9 lentelė. Klaidų paplitimas skirtinguose paketuose

Paketas	Aptiktas klaidų skaičius				Kodo eilučių skaičius	Klaidų paplitimas	
	Tuščios eilutės	**Eilučių sujungimo**	**Neteisingo palyginimo**	**Viso**		**Kodo eilučių vienai klaidai**	**Klaidingo kodo santykis**
NQuery	0	8	12	**20**	62,963	3148	0.0318%
Dejavu	0	0	1	**1**	4,576	4576	0.0219%
NFileStorage	0	3	0	**3**	4,635	1545	0.0647%
USPS OneCode	1	2	0	**3**	3,017	1006	0.0994%
DotNetZip	8	12	11	**31**	90,174	2909	0.0344%
ScintillaNET	5	56	41	**102**	33,886	332	0.3010%
IronPython	40	149	99	**288**	577,871	2007	0.0498%

Atlikę eksperimentus galime apibendrinti rezultatus. Visuose testuojamose paketuose išskyrus ScintillaNET klaidų paplitimo dažnis buvo tarp 1000-5000 programinio kodo eilučių vienai klaidai. Didžiausius nuokrypius nuo vidurkio turėjo Dejavu ir ScintillaNET paketai. Tai galima paaiškinti tuo, kad Dejavu paketas yra labai mažas (4,5 tūkst. kodo eilučių) ir dėl savo programinio kodo ypatybių nebūtinai atitinka bendros statistikos. Tuo tarpu ScintillaNET paketas turėjo viso net 102 taisyklių pažeidimus, kas atitinka vieną klaidą per 332 kodo eilutes. Šia

anomalija galime paaiškinti tuo, kad šis projektas ypač glaudžiai susijęs su operacijomis su eilutėmis, jų modifikavimu ir keitimu. Tai atspindi ir statistika rodanti, kad šiame pakete „tuščiuos eilutės“ taisyklės pažeidimas buvo užfiksuotas vos 5 kartus, o „eilučių sujungimo“ ir „neteisingo palyginimo“ kartu sudėjus - 97.



21 pav. Klaidų santykinis paplitimas tiriamose paketuose

Suskaičiavus rezultatų vidurkius atsižvelgus į standartinius nuokrypius, galime teigti, kad mūsų taisykles apimantis kodas, realiose sistemose, vidutiniškai pasitaiko vienoje programinio kodo eilutė iš 2000 arba 0,05% viso kodo. Tai patvirtino ir eksperimentai su IronPython paketu, kurio bendras kodo eilučių skaičius yra didesnis, nei visų likusių testuotų sistemų kartu sudėjus.

6. IŠVADOS

Pagrindiniai rezultatai ir atlikti darbai:

1. Suprojektuota ir realizuota komercinė programinio kodo valdymo sistema:
    a. Naudojama .NET platforma;
    b. Pilnas kodo atitikimas FxCop pateikiamom bei papildomai realizuotoms programinio kodo statinės analizės taisyklėms;
2. Ištirtos jau esamos statinio kodo analizės taisyklės programos veikimo greičio ir atminties suvartojimo charakteristikų aspektu. Pasiūlytos naujos patobulintos taisyklės. Sudarytos rekomendacijos tolesniam kodo tobulinimui;
3. Sukurtos 3 papildomos statinio kodo analizės taisyklės. Eksperimentiškai nustatėme, kad jų vidutiniškas paplitimas taikomosiose programose siekia 0,05% viso programinio kodo eilučių skaičiaus:
    a. Tuščių eilučių taisyklė;
    b. Eilučių sujungimo taisyklė;
    c. Neteisingo palyginimo taisyklė;

7. LITERATŪRA

[1] **V. A. Shekhovtsov, Y. Tomilko, M. D. Godlevskiy.** Facilitating Reuse of Code Checking Rules in Static Code Analysis. *Business Information Processing konferencijos medžiaga.* Sidnėjus. 2009. p. 91-102.

[2] **D. Thomas, A. Hunt.** Pragmatic Version Control using CVS. The Pragmatic Programmers, ISBN: 978-0-9745140-0-0. 2003.

[3] **K. Cwalina, B. Abrams.** Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries. Addison-Wesley Professional, ISBN: 978-0321246752. 2005.

[4] **J. Atwood.** What's In a Version Number, Anyway?. [Žiūrėta 2010 04 05]. <http://www.codinghorror.com/blog/archives/000793.html>. 2007.

[5] **N. Willis.** Decline and fall of the version number. [Žiūrėta 2010 04 05]. <http://www.linux.com/articles/45507>. 2005.

[6] **J. Andrews.** Feature: No More Free BitKeeper. [Žiūrėta 2010 04 05]. <http://kerneltrap.org/node/4966>. 2003.

[7] **D. A. Wheeler.** Comments on Open Source Software / Free Software (OSS/FS) Software Configuration Management (SCM) Systems, [Žiūrėta 2010 04 05]. <http://www.dwheeler.com/essays/scm.html>. 2005

[8] **I. Clatworthy.** Distributed Version Control Systems – Why and How. [Žiūrėta 2010 04 05]. <http://people.ubuntu.com/~ianc/papers/dvcs-why-and-how.xhtml>. 2007.

[9] **S. Candrlic, M. Pavlic, P. Poscic.** A Comparison and the Desireable Features of Version Control Tools. 29th International Conference on Volume konferencijos. 2007. 121-126 psl.

[10] **E. Gunnerson, N. Wienholt.** A Programmer's Introduction to C# 2.0, Third Edition. Apress, ISBN: 978-1590595015. 2006.

[11] **A. Orlowski.** Linus Torvalds defers closed source crunch. [Žiūrėta 2010 04 05] <http://www.theregister.co.uk/2005/04/06/torvalds_bitkeeper/>. 2005.

[12] **B. Wagner, A. Wesley.** Effective C#: 50 Specific Ways to Improve Your C#. Addison-Wesley Professional, ISBN: 978-0321245663. 2004.

[13] **C. Morgan, K. De Volder, E. Wohlstadter.** A Static Aspect Language for Checking Design Rules. *6th international conference on Aspect-oriented software development* konferencijos medžiaga. 2008. p. 63-72.

[14] **M. Martin, B. Livshits, M. S. Lam.** Finding application errors and security flaws using PQL: a program query language, *20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications konferencijos medžiaga*. 2005. p. 365-383.

[15] **A. Christensen, A. Moller, M. Schwartzbach.** Precise analysis of string expressions. *International Static Analysis Symposium (SAS'03) konferencijos medžiaga*. 2003.

[16] **A. MacKenzie, S. Monk.** From Cards to Code: How Extreme Programming Re-Embodies Programming as a Collective Practice. *Computer Supported Cooperative Work*. 2006. Nr. 13, p. 91-117.

[17] **S. Thummalapenta, L. Cerulo, L. Aversano, M. Di Penta.** An empirical study on the maintenance of source code. *Empirical Software Engineering*. 2009. Nr. 15, p. 1-34.

[18] **J. Kresowaty.** FxCop and Code Analysis: Writing Your Own Custom Rules. 2008.

[19] **D. Baca, K. Petersen, B. Carlsson, L. Lundberg.** Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter? *Availability, Reliability and Security, 2009. ARES '09 konferencijos medžiaga.* 2009. p. 804-810.

[20] **R. Plosch, H. Gruber, A. Hentschel, G. Pomberger, S. Schiffer.** On the Relation between External Software Quality and Static Code Analysis. *Software Engineering Workshop, SEW '08 konferencijos medžiaga.* 2008. p. 169-174.

[21] **A. Troelsen.** Pro C# 2005 and the .NET 2.0 Platform. Apress, ISBN: 978-1590594193. 2005.

8. TERMINŲ IR SANTRUMPŲ ŽODYNAS

.NET	- Programų kūrimo platforma
API	- Programos sąsaja
C#	- .NET platformos objektiškai orientuota programavimo kalba
CIL	- Common Intermediate Language
CLR	- Common Language Runtime
COM	- Component Object Model
DLL	- Dynamic-Link Library
DSL	- Domain Specific Language. Sričiai orientuota kalba ar taisyklių rinkinys
GC	- .NET Garbage Collector
IDE	- Integruota programinės įrangos kūrimo aplinka (angl. Integrated Developement Environment)
Meta duomenys	- Duomenys, aprašantys kitus duomenis
MSDN	- Microsoft Developer Network
Korektiškas kodas	- Programinis kodas atitinkantis užsibrėžtas programavimo taisykles
OMG	- Object Management Group
OO	- Objektiškai orientuotas (angl. Object Oriented)
PKVS	- Programinio kodo valdymo sistema
RUP	- Rational Unified Process
SLOC	- Programinio kodo eilučių skaičius. (angl. Source Lines Of Code)
Statinė kodo analizė	- Programinio kodo analizės būdas nereikalaujantis aktyvaus programos paleidimo ir veikimo.
StringBuilder	- .NET platformos eilučių apdorojimo klasė
SVN	- Apache Subversion programinė įranga
UML	- Unified Modeling Language
UTF-8	- Koduotė vienam simboliui saugoti skirianti 8 baitus
VCS	- Version Control System
XML	- Extended Markup Language